Design and Analysis of Algorithms I

David Ng

Fall 2017

Contents

§1 September 12, 2017

§1.1 Introduction

This course concerns what is means for an algorithm to *correctly* solve a computational problem. We want to do this efficiently, and prove that we have done so. We are also concerned with designing algorithms that solve these problems, and prove that some problems cannot be solved efficiently. By the end of this course, we will be able to understand *preconditions* and *postconditions*, and use these to define computational problems. We will also be able to give proofs of correctness of algorithms, and analyze the running time of algorithms through summations and recurrences using asymptotic notation. (Class wasted).

§2 September 14, 2017

§2.1 Proofs of Correctness

A *specification of requirements for a computational problem* is something that we want to solve using a computer program, and includes the following:

- *Precondition*: A condition or property that is either `true` or `false`, and that is satisfied by any well formed instance of inputs for this problem.
- *Postcondition*: A condition that is satisfied if this problem is solved. This might include relationships between inputs as well as outputs, as well as initial and final values of global data that is accessed and modified.

Example 2.1 (Fibonacci Numbers Again)
Suppose that n is a non-negative integer. Then the nth Fibonacci number F_n is defined using the following rule,

$$F_n = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ F_{n-2} + F_{n-1} & \text{if } n \geq 2. \end{cases}$$

The problem of computing the nth Fibonacci number, when given a non-negative integer n as input can be defined using preconditions and postconditions. The precondition is that a non-negative integer n is given as input. The postcondition is that the nth Fibonacci number F_n is returned as output.

An *algorithm* can be defined as an effective method expressed as a finite list of well-defined instructions for solving a computational problem. Algorithms can be presented using English, using pseudocode, or using executable code. Below is pseudocode for an algorithm that computes the Fibonacci numbers.

```
integer fib (integer n)
{
  if (n == 0)
  {
    return 0
  }
```

```
    else if (n == 1)
    {
        return 1
    }
    else
    {
        return fib(n-2) + fib(n-1)
    }
}
```

An algorithm for a given computational problem is *correct* if whenever the problem's precondition is satisfied and the algorithm is executed, then the execution of the algorithm eventually ends, and the problem's postcondition is satisfied when this happens. There are no changes to the system state that are not otherwise documented in the preconditions and postconditions. Changes should not be caused by the execution of the algorithm if the precondition holds when execution begins. Thus, only undocumented variables whose values might change, should be local variables (variables that do not exist before and after execution). This does not promise anything regarding what should occur when the precondition is not satisfied, and the algorithm is executed.

The following is a proof that the `fib` algorithm is correct. It is assumed that integer arithmetic is correct and exact when included in an algorithm. It is also assumed that equality tests for integers are performed correctly during execution.

Theorem 2.2 (Fibonacci Algorithm)

Suppose that n is a non-negative integer and that the algorithm `fib` is executed with n as input. The execution of this algorithm ends after it has output the nth Fibonacci number.

Proof. This will be proved by induction on n. The strong form of induction is used, with the base cases $n = 0$ and $n = 1$. First, suppose that $n = 0$. Then, on input n, line 1 succeeds, so execution continues with line 2. This causes execution to terminate with $F_0 = 0$ returned. When $n = 1$, line 1 fails, so the execution continues on line 3. This succeeds, so continuing on line 4, this causes $F_1 = 1$ to be returned.

Now, let k be an integer $k \geq 1$. Suppose that n is a non-negative integer such that $0 \leq n \leq k$. When executed with n as input, `fib` eventually terminates with F_n as output. Now, we need to show that when given $n = k + 1$, then `fib` terminates with $F_{k+1} = F_n$ as output. Since $k \geq 1$, $n \geq 2$. Thus, after failing both conditions at lines 1 and 3, it reaches line 5.

Here, we reach a recursive execution with input $n - 2$. Since $n = k + 1 \geq 2$, we have $0 \leq n - 2 = k - 1 \leq k$. It follows by the inductive hypothesis that this recursive execution of the algorithm eventually ends with $F_{n-2} = F_{k-1}$. We also have a recursive execution on input $n - 1$, which produces $F_{n-1} = F_k$ as output through similar reasoning. Line 5 then ends with $F_{k+1} = F_k + F_{k-1} = F_{n-1} + F_{n-2}$, as to be shown. $\square$

Note that we have made use of mathematical induction on n. This traces the execution of the algorithm on various kinds of input in order to establish the basis and complete the inductive step. We can check by inspection that the only input accessed is specified in the precondition, and no global data is accessed. The algorithm also does not modify any input or global data, and the only output returned is specified by the postcondition. Thus, it has no undocumented side effects, and can be considered *correct*.

§2.2 Importance of Algorithm Correctness

We care about the correctness of programs, since some problems are *safety-critical*. For instance, *library software* should be treated in this way, as many people make use of it in ways that cannot be anticipated or controlled. *Testing* and *debugging* are complementary to formal proofs of correctness. *Dynamic testing* allows working code to be run and checked, making it more effective in the identification of coding errors and errors from other assumptions. However, testing can almost never be exhaustive or complete, as there will likely be missed situations or cases. Understanding a proof of the correctness of an algorithm can be helpful for more effective and extensive test design, and more effective debugging.

It is important that there are no undocumented side effects, especially in the case of *library software*. When this is used in larger systems, undocumented side effects can cause systems to fail. Additionally, since adding code to a software library entails adding to *bedrock* that others will depend on, it is important to ensure that the code is built correctly. *Inline documentation* should include the *preconditions* and *postconditions*, a proof of correctness of the algorithm, and time and space bounds.

§2.3 Bound Functions and Assertions

A *bound function* for a recursive algorithm is a mathematical function that is defined on the inputs and global data that are accessed and modified when the recursive algorithm is executed - specifically on those inputs and global data satisfying the precondition for the problem being solved and that also satisfies the following properties:

1. This is an integer-valued function.

2. Whenever the algorithm is applied recursively, the value of the function has been decreased by at least one.

3. If the function's value is less than or equal to zero when the algorithm is applied, then the algorithm does not call itself recursively during this execution.

Example 2.3 (Bound Function)

Consider the algorithm `fib` and the function

$$f(n) = n.$$

Since n is an integer input, this is certainly an integer-valued function. As noted in the proof of the correctness of this algorithm, the value of this function is decreased by at least one every time the algorithm is replied recursively. It was also noted that $n \geq 0$ whenever this algorithm is recursively applied. It follows that it is only possible that $f(n) = n \leq 0$ under these conditions if $n = 0$. In this case, the test at line 1 passes, execution continues with line 2, and execution ends immediately after that without the algorithm having called itself recursively. This function is therefore a bound function of the recursive algorithm `fib`.

The bound function was identified just by checking that all of the properties listed in the definition were satisfied. This only requires an examination of the pseudocode of the algorithm. Establishing a bound function for a recursive algorithm will often (but not always) be as easy as this. It is often possible to prove properties of recursive algorithms using mathematical induction on the initial value of the bound function. This is how the correctness of recursive algorithms will generally be proved.

An *assertion* is a boolean condition (or *predicate*) involving an algorithm's inputs, local variables, outputs, and possibly global data. It is assumed to be satisfied at a specific point in a computation before execution, or immediately before or after a specific instruction in the algorithm has been executed. Assertions can be used to document significant parts of the proof of correctness of the algorithm. If they are reasonably complete, a reader may see the proof of correctness just by reading the bound function and assertions. Many programming languages include statements with names such as `assert` that allow assertions to be actively checked when programs are tested. This makes the code self-checking. Ideally, the designer of an algorithm should supply a proof of its correctness. Other people wishing to use this algorithm should also have access to this proof.

Example 2.4 (Assertion)

Consider the `fib` algorithm. Assertions only need to be satisfied during execution when the precondition is initially satisfied. Thus the assertion that "n is an integer input such that $n \geq 0$" can be listed immediately before line 1 of the `fib` algorithm. Since the test at line 1 has been passed if we have arrived at step 2, the assertion shown between lines 1 and 2 in the code should also reflect this, as "n is an integer input whose value is 0". The assertion immediately following line 2 should also reflect the fact that the output has been returned in this case. Thus, "n is an integer input whose value is 0", and "the value $F_n = F_0 = 0$ has been returned as output". Assertions for later points in the pseudocode should also reflect what is known when each point is reached.

§2.4 Trace of Execution and Recursion Trees

A *trace* of execution of an algorithm on an input is a listing of the statements that are performed when the algorithm is executed on this input, along with a description of the effects of these statements. If the algorithm is recursive, then one execution of the algorithm will often include one or more other executions of the same algorithm on different inputs. One way to deal with this when providing a trace of execution is to number the traces of all the executions that are included, and then give a separate trace of execution for each providing cross-references between traces as needed.

A *recursion tree* is a tree that corresponds to an execution of a recursive algorithm on a fixed input and shows the relationship between all the different executions that get made along the way. The *root* of this tree corresponds to the original execution of the algorithm. If the algorithm calls itself recursively on an execution, then the node for this execution has a *child* corresponding to each of the recursive calls that it makes. If the algorithm does not call itself recursively then the node for this execution does not have any children at all and is a *leaf* in the tree.

Recursion trees help make sense of all the different traces of execution that are needed when one wants to consider the execution of a recursive algorithm on a particular input. They are also very helpful when proving interesting things about the running times and storage requirements for recursive algorithms.

§3 September 19, 2017

§3.1 Proof of Correctness - While Loop

Consider an algorithm that includes a while loop for a given computational problem. An assertion is a *loop invariant* for this while loop if it is satisfied at the following times:

1. At the beginning of every execution of the while loop.
2. At the beginning of every execution of the body of the while loop.
3. At the end of every execution of the body of the while loop.
4. At the end of every execution of the while loop.

To understand proofs of correctness involving while loops, we will once again consider the problem of finding the nth Fibonacci number F_n given an input of n. Now, we make use of a while loop for this calculation. Recall that the precondition is that a non-negative integer n is given as input, and the postcondition is that F_n returned as the output. Below is pseudocode for an algorithm that computes the Fibonacci numbers using a while loop.

```
integer fibLoop (integer n)
{
   if (n == 0)
   {
      return 0
   }
   else
   {
      integer [] F = new integer[n+1]
      F[0] = 0
      F[1] = 1
      integer i = 1
      while (i < n)
      {
         F[i+1] = F[i-1] + F[i]
         i = i + 1
      }
      return F[n]
   }
}
```

Consider the algorithm `fibLoop` for the Fibonacci number computation problem, and the while loop included in this algorithm. The following is a loop invariant for this while loop:

1. n is an input integer such that $n \geq 1$.
2. F is a variable integer array with length $n + 1$.
3. i is an integer variable such that $1 \leq i \leq n$.
4. $F[j] = F_j$ for every integer j such that $0 \leq j \leq i$.

To show that this is indeed a loop invariant, we will need to show that all components of the loop invariant are satisfied at all times required by the definition of the loop invariant.

Proof. First, we show that the proposed loop invariant is satisfied at the beginning of every execution of the while loop. This serves as the base case. We call this the *first claim*.

1. For n, when it is equal to 0 it is trivially satisfied since the while loop is not reached. If instead it was greater than or equal to 1, then since there is no change in n, it is always the case that $n \geq 1$. This while loop is never reached again, so we do not need to consider what happens when we reach this while loop again.

2. F is declared to be an integer array with a length of $n + 1$ in the third step. This holds by the time we first reach the while loop.

3. The variable i is declared as an integer variable with a value of 1 in the sixth step, thus it is certainly the case that $1 \leq i \leq n$ when we reach the loop.

4. When we reach the while loop, we have set $F[0] = 0$ and $F[1] = 1$. Thus, when we reach the while loop at $i = 1$, $F[j] = F_j$ indeed holds for all j in the range of $0 \leq j \leq i$.

Now, we show that if the loop invariant is satisfied at the beginning of an execution of the body, then it will also be satisfied at the end of the execution of the body. This forms the inductive case, since we can use the base case and repeatedly apply this inductive case. We call this the *second claim*.

1. At the beginning of the loop, we have supposed that $n \geq 1$. Thus, since n does not change, it remains the case that $n \geq 1$ at the end of the execution of the body of the loop.

2. It was supposed that F is an integer array with length $n + 1$ at the beginning of the execution of the body of the loop. However, neither the loop test nor the loop body change the type or length of F.

3. At the beginning of the execution of the body of the loop, $1 \leq i \leq n$. Additionally, the loop test passed, so $i < n$. The ninth step involves adding 1 to i, so i is now bounded by $i \leq n$. Therefore, at the end of this execution of the loop body, it remains that $1 \leq i \leq n$.

4. $F[j] = F_j$ holds as specified for $1 \leq j \leq i$ at the beginning of the loop body execution. Moreover, we know that $1 \leq i \leq n - 1$ from the situation above. Thus, when we modify the array in the eight step, it remains that we have not exceeded the array size since $2 \leq i+1 \leq n$, and $F[i+1] = F[i-1]+F[i] = F_{i-1}+F_i = Fi + 1$ as required. At the end of step 8, we have $F[j] = F_j$ for all $0 \leq j \leq i + 1$. However, we increment i in step 9, so it remains that $F[j] = F_j$ for all $0 \leq j \leq i$.

We shall now prove by induction by using the previous two claims that the loop invariant is indeed correct. For the base case of one loop execution, then the loop is clearly reached, so the loop invariant holds as shown by the first claim. After this, we only have the loop test that does not change any value, so the loop invariant holds at the beginning of the first execution of the loop body. By the second claim, the loop invariant is also true at the end of execution of the loop body.

Now, for the inductive hypothesis, suppose that the loop invariant holds for $k \geq 1$ loop executions. We will now show that for $k + 1$ loop executions, the loop invariant still holds. We know from the inductive hypothesis at k loop executions that the loop invariant still holds at the end of the body. The loop test does not change any variable, so the loop invariant holds at the beginning of the $k+1$ loop execution. By the second claim, that the loop invariant holds at the end of the loop body execution.

By using the first and second claims, we have shown that the loop invariant holds whenever we reach the loop, at the beginning of each loop execution, and at the end of each loop execution. We still need to show that the loop invariant holds at the end of the execution of the while loop. The loop is reached whenever $n \geq 1$. For $n = 1$, we have reached the loop with the loop invariant true, and then after the loop test where nothing is changed, the loop invariant still holds when loop execution ends. For $n \geq 2$, the loop body is executed at least once, with the loop invariant holding for the last execution. Since the loop test does not change any variables, when we leave the loop, the loop invariant still holds. $\square$

§3.2 Establishing Loop Invariants

To simplify the process of establishing the loop invariant, we make use of the following theorem.

Theorem 3.1 (First Loop Theorem)
Consider that assertion A pertaining to an algorithm with a while loop of the form

```
while (t)
{
   S
}
```

If an execution of the loop test t has no side effects (not changing the value of any inputs, variables, or global data), A is satisfied whenever the loop is reached during an execution of the algorithm (with the problem's precondition being satisfied), and A being satisfied at the beginning of execution of loop body S implies that A is satisfied when execution of the loop body ends, then A is a loop invariant for the while loop.

The proof of this is a generalization of the previous proof given. The proof is left to the reader as an exercise ;-).

This theorem makes it easier to prove the loop invariant of `fibLoop`, as we have reduced the number of necessary steps. Now, we first need to note that the loop test does not change any inputs, variables, or global data. Then, we need to state and prove the first claim, and finally state and prove the second claim. We then apply the loop theorem to conclude that the loop invariant is indeed correct.

§3.3 Partial Correctness

An algorithm is *partially correct* if at least one of the following properties is satisfied whenever this algorithm is executed when the problem's precondition is satisfied:

1. This execution of the algorithm eventually ends with the problem's postcondition satisfied and with no undocumented inputs, no global data accessed, and no undocumented data being modified.
2. This execution of the algorithm never ends at all.

Well chosen and complete loop invariants are useful because they can be used to establish the partial correctness of algorithms.

Proposition 3.2 (Partial Correctness of `fibLoop`)

The `fibLoop` algorithm is partially correct as an algorithm for computing the nth Fibonacci number.

Proof. Consider an execution of the algorithm such that the precondition, that the input n is an integer, is satisfied. By inspection, it is clear that there is no undocumented access of input, global data, or variables. When $n = 0$, this ends after executing the first and second step, where $F_n = F_0 = 0$ is returned and the postcondition is satisfied, thus proving partial correctness for this case.

When $n \geq 1$, we reach the while loop. This loop either terminates, or it does not. In the case that it does not, then the algorithm does not terminate and thus satisfies the definition of partial correctness. In the case that it terminates, then it follows that the loop invariant holds when the execution of the loop ends. The loop test was checked and failed, so $i = n$ and $F[j] = F_j$ for $0 \leq j \leq n$. But then since $F[n] = F_n$ is returned, this satisfies the postcondition and thus establishes partial correctness. □

Remark 3.3. Correctness implies partial correctness, but the converse is not true

Consider the following algorithm. It can be shown that the loop invariant for `fibLoop` also holds here. Additionally, this algorithm is partially correct. However, it can be shown that this is not correct, as it does not terminate.

```
integer badFibLoop (integer n)
{
   if (n == 0)
   {
      return 0
   }
   else
   {
      integer [] F = new integer[n+1]
      F[0] = 0
      F[1] = 1
      integer i = 1
      while (i < n)
      {
         i  = i
      }
      return F[n]
   }
}
```

§3.4 Bound Function for While Loops

A *loop variant* is a bound function for a while loop. It is a an integer-valued, total function of some of the variables, usually those that are defined when the loop is reached. If the precondition was satisfied when we execute the algorithm, then the loop variant must satisfy the following two properties:

1. When the loop body is executed, the value of this function is decreased by at least one before the loop's test is checked again.

2. If the value of this function is less than or equal to zero and the loop test is checked, then the test fails (ending this execution of the loop).

Proposition 3.4

The function $f(n, i) = n - i$ is a loop variant for the while loop of the `fibLoop` algorithm.

Proof. Since n and i are integers that are defined and initialized, this is an integer valued function of variables defined when we reach the loop. When the body of the loop is executed, i increases by one while n remains constant. Thus, f decreases by one for each loop. When the value of $f \leq 0$, then $n - i \leq 0$. But this is exactly when $i \geq n$ and the loop test fails. Thus, f is the loop variant. □

§3.5 Termination

To prove termination, we make use of the following theorem:

Theorem 3.5 (Second Loop Theorem)

Consider a while loop of the form

```
while (t)
{
   S
}
```

Suppose additionally that the loop test t has no side effects, a loop variant exists for the while loop, and that every execution of the loop eventually ends given that the precondition was satisfied when the loop began. Then, every execution of this while loop ends. Additionally, the loop variant at the beginning of execution is an upper bound for the number of loops.

Corollary 3.6

The Second Loop Theorem implies that the execution of a while loop always includes exactly $n - 1$ executions of the loop body.

Proposition 3.7

When `fibLoop` is executed with preconditions satisfied, then this algorithm terminates

Proof. In the case that $n = 0$, then execution ends after the second step is executed. In the case that $n \geq 1$, the while loop is reached. Since there is a loop variant for this while loop, and it is clear that there are only two steps in the loop, it follows by the Second Loop Theorem that this algorithm eventually terminates. □

It is clear that if an algorithm is both partially correct and terminates, then this algorithm is correct. This is a consequence of the definitions of partial correctness, termination, and correctness. The `fibLoop` algorithm is therefore correct.

§4 September 21, 2017

§4.1 Running Time - While Loop

To measure the use of resources, we will use analytical techniques to discover bounds for running times and storage requirements. Additionally, proofs will be given that these bounds are indeed correct. However, it is not always clear how much the running time of a given statement in the algorithm is. This may not even be well defined. Thus, to simplify analysis, we consider a *uniform cost criterion* where we define the running time of each step in an algorithm to be unity. The amount of *storage* needed to store an element of an elementary data type (such as `integer` or `string`) will also be defined to equal one. Therefore, an `array` will require storage equivalent to its length.

Example 4.1 (Running Time of `fibLoop`)
Once again, we will consider the `fibLoop` algorithm to illustrate the principles of running time. We will make use of the Second Loop Theorem along with a loop variant in order to determine an upper bound on the execution of a while loop body. A *formula* can then be given for the upper bound of steps executed in the while loop as a function of input n.

Solution. Recall that the loop variant for the while loop was given as $f(n, i) = n - i$. Since i is initialized immediately to 1, the initial value of the function is $n - 1$. It follows that the body executes at most $n - 1$ times. In the particular example with `fibLoop`, the value of f decreases by exactly one for each body execution, the loop test fails if and only if the value of $f \leq 0$ when tested, and the loop test must be checked and failed in order for execution of the loop body to end (there are no return statements, or breaks).

Now, since we execute the loop body $k = n - 1$ times, this means that the loop test is executed at most $k + 1 = n$ times since it must be checked after the final execution to know that the condition is no longer satisfied. We consider the cost of the loop test as a function of the jth test where $1 \leq j \leq n$, and denote this as $T_{test}(j)$. In our example, we considered the cost of the test to be 1. Then, clearly the total cost of all executions of the loop test is

$$\sum_{j=1}^{n} T_{test}(j) = \sum_{j=1}^{n} 1 = n.$$

It follows that we can similarly use j to express the total cost of the loop body as a function $T_{body}(j)$. In this specific case, the loop body consists of two steps on lines 8 and 9, so $T_{body}(j) = 2$. Therefore, the total cost of all loop body executions is

$$\sum_{j=1}^{n-1} T_{body}(j) = \sum_{j=1}^{n-1} 2 = 2(n-1) = 2n - 2.$$

It follows that the total number of steps for execution of this while loop is simply the sum, giving us $3n - 2$ steps.

After we have considered the running time of the while loop, we can now consider the running time of the entire algorithm. For $n = 0$, there are two steps since it executes lines 1 and 2. When $n \geq 1$, the test at line 1 fails, so we now follow the else statement. Here, there are four statements along with the while loop. This else statement therefore has $(3n - 2) + 4 = 3n + 2$ statements. Lastly, we have a return statement, so we have a total of $3n + 4$ steps. Therefore, we can conclude that the number of steps executed by `fibLoop` on input n is

$$T(n) = \begin{cases} 2 & \text{if } n = 0, \\ 3n + 4 & \text{if } n \geq 1. \end{cases}$$

■

Remark 4.2. A common mistake is to confuse the loop variant with the initial value it takes when bounding the cost of the loop. The final expression of running time does not depend on the variable i, since this is not an input.

§4.2 Storage Space - While Loop

Note that this algorithm only uses three inputs and variables:

1. n is an integer input, so it has unit cost.
2. i is an integer variable, so it also has unit cost.
3. F is an integer array of length $n + 1$, so it costs $n + 1$ to store.

In this case, i and F are only declared when $n \geq 1$. It suffices to add the storage requirements for each input and variable to define the storage requirements. In the `fibLoop` algorithm, the storage space is given as

$$S(n) = \begin{cases} 1 & \text{if } n = 0, \\ n + 3 & \text{if } n \geq 1. \end{cases}$$

§4.3 Summation Identities (high school math)

The following identities can be proved by induction. These will be useful for proving bounds for certain algorithms.

$$\sum_{j=a}^{b} 1 = b - a + 1,$$

$$\sum_{j=1}^{n} j = \frac{n(n+1)}{2},$$

$$\sum_{j=1}^{n} j^2 = \frac{n(n+1)(2n+1)}{6},$$

$$\sum_{j=1}^{n} j^3 = \frac{n^2(n+1)^2}{4},$$

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1}.$$

§4.4 Bounding Summation Terms

Often, a reasonably good upper bound for a summation can be found by approximating each term by the largest term. The upper bound is then the number of terms multiplied by this largest value.

Example 4.3 (Upper Bound)

Consider the summation of

$$\sum_{k=1}^{n} k^2.$$

Since this is an increasing function of k, and the largest value is n^2, we approximate the summation as

$$\sum_{k=1}^{n} k^2 \leq n \cdot n^2 = n^3.$$

However, the bounds obtained by using the previous method are not always close approximations. The following example illustrates the disparity that results.

Example 4.4 (Lower Bound)

Consider again the previous example. The product of the number of terms with the smallest term is a lower bound for the summation. The smallest value is $k^2 = 1$, so the approximation becomes

$$\sum_{k=1}^{n} k^2 \geq n \cdot 1^2 = n.$$

But this is not close to the upper bound approximation at all.

In these cases, we need to split the sum into two or more pieces. The following example illustrates this concept.

Example 4.5 (Splitting the Sum)

We can split the summation into two parts to obtain

$$\sum_{k=1}^{n} k^2 = \sum_{k=1}^{\lfloor n/2 \rfloor} k^2 + \sum_{k=\lfloor n/2 \rfloor+1}^{n} k^2.$$

On the right hand side, we see that the original summation is split into ones for lower values, and one for upper values. For the lower value summation, there are $\lfloor n/2 \rfloor \geq (n-1)/2$ terms, where each term is greater than or equal to 1. Therefore, the lower bound on this is

$$\sum_{k=1}^{\lfloor n/2 \rfloor} k^2 \geq \left\lfloor \frac{n}{2} \right\rfloor \cdot 1 = \left\lfloor \frac{n}{2} \right\rfloor \geq \frac{n-1}{2}.$$

For the upper value summation, there are $n - \lfloor n/2 \rfloor = \lceil n/2 \rceil \geq n/2$ terms, where each term is greater than or equal to $(\lfloor n/2 \rfloor + 1)^2 \geq ((n+1)/2)^2$. Therefore, the lower bound on this is

$$\sum_{k=\lfloor n/2 \rfloor+1}^{n} k^2 \geq \frac{n}{2} \cdot \left(\frac{n+1}{2}\right)^2 = \frac{n^3}{4} + \frac{n^2}{2} + \frac{n}{4}.$$

It now follows that we add both of these values on the right that act as the lower bound to find that the summation is cubic with respect to n

$$\begin{aligned} n^3 \geq \sum_{k=1}^{n} k^2 &= \sum_{k=1}^{\lfloor n/2 \rfloor} k^2 + \sum_{k=\lfloor n/2 \rfloor+1}^{n} k^2 \\ &\geq \frac{n-1}{2} + \frac{n^3}{4} + \frac{n^2}{2} + \frac{n}{4} \\ &= \frac{n^3}{4} + \frac{n^2}{2} + \frac{3n}{4} - \frac{1}{2} \\ &\geq \frac{n^3}{4}. \end{aligned}$$

Another method that can be used to bound the summation is through the use of integrals. Upper and lower bounds can be established through the following theorem.

Theorem 4.6 (Integral Approximation)

Given integers a and b, and an increasing integrable function $f(x)$ defined for all real numbers $a-1 \leq x \leq b+1$, then the following results when $a \leq k \leq b$

$$\int_{x=k-1}^{b} f(x) = \sum_{k=a}^{b} \int_{x=k-1}^{k} f(x) \leq \sum_{k=a}^{b} f(k) \leq \sum_{k=a}^{b} \int_{x=}^{k+1} f(x) = \int_{x=a}^{b+1} f(x).$$

Example 4.7 (Integral Approximation)

Considering the same summation as in the previous examples, we note that $f(k) = k^2$ is an increasing integrable function, with an antiderivative of $F(n) = k^3/3$. Using $a = 1$ and $b = n$, we find that

$$\sum_{k=1}^{n} k^2 \geq \int_{x=0}^{n} f(x) = F(n) - F(0) = \frac{n^3}{3},$$

$$\sum_{k=1}^{n} k^2 \leq \int_{x=1}^{n+1} f(x) = F(n+1) - F(1) = \frac{n^3}{3} + n^2 + n.$$

This says that the summation is $n^3/3$ plus another value that is at most quadratic with respect to n.

§4.5 Running Time Considerations

Through our analyses, we have made use of the uniform cost criterion assumption. One should note however, that this holds only when all of the inputs, outputs, local variables and global data used by an algorithm are small (small enough to be represented using only a constant number of words of computer memory). An alternative is the *logarithmic cost criterion.*

Aside from analyzing running time, one may also choose to run the actual code and time the execution. The advantage of this approach is that simplifying assumptions, such as uniform cost criterion, are not required. However, execution time is influenced by many factors. For instance, hardware, compiler and system software, simultaneous user activity, choice of input data, and the skill of the programmer ultimately affect the execution time. Sometimes, we may also be concerned with the *space* required to store the program, or the *time* required to code and maintain the program.

§5 September 26, 2017

§5.1 Running Time - Recursion

To demonstrate analysis of the running time of recursive algorithms, consider the `fibPair` algorithm, recalling that $n \geq 0$ when the precondition is satisfied.

```
integer[] fibPair (integer n)
{
   integer[] F = new integer[2]
   if (n == 0)
   {
      F[0] = 0
      F[1] = 1
   }
   else
   {
      integer[] oldF = fibPair(n-1)
      F[0] = oldF[1]
      F[1] = oldF[0] + oldF[1]
   }
   return F
}
```

Example 5.1 (Running Time of `fibPair`)

Determine the running time of the `fibPair` algorithm.

Solution. We will once again make use of the uniform cost criterion assumption to analyze the running time of this algorithm. In the case that $n = 0$, there are exactly five steps, since the algorithm executes steps 1-4, and then step 8. When $n \geq 1$, the algorithm executes steps 1, 2, 5, 6, 7, and 8 for a total of six steps. However, it also calls itself recursively in step 5 on input $n - 1$. Thus,

$$T(n) = \begin{cases} 5 & \text{if } n = 0, \\ T(n-1) + 6 & \text{if } n \geq 1. \end{cases}$$

■

In mathematics, a *recurrence* (or *recurrence relation*) is generally defined to be a relation that recursively defines the elements of a sequence of values. For instance, $T(0), T(1), T(2), \ldots$ Thus, the running time expression can be viewed as a recurrence that defines the sequence. When the recurrence relation is simple enough, it may be possible to guess a solution after a few initial values have been computed.

Example 5.2 (Finding the Closed Form Expression)

The first few values of the recurrence for the running time T of the `fibPair` algorithm are

$$\begin{aligned} T(0) &= 5, \\ T(1) &= T(0) + 6 = 5 + 6 = 11, \\ T(2) &= T(1) + 6 = 11 + 6 = 17, \\ T(3) &= T(2) + 6 = 17 + 6 = 23. \end{aligned}$$

Because $T(n) = T(n-1) + 6$, it would be reasonable to assume that the closed form of the recurrence is of the form $6n + C$ for some constant C. Because $T(0) = 5$, we find that the closed form expression is therefore $T(n) = 6n + 5$ for all integers $n \geq 5$, assuming that the guess is correct.

When a guessed solution is correct, this can often be proven by induction.

Example 5.3 (Equivalence of Closed Form Expression)

Given that T is the running time function of `fibPair` as described in the solution of Example 5.1, then $T(n) = 6n + 5$ for every non-negative integer n.

Proof. We will prove the claim using the standard form of mathematical induction on n. In the base case, consider when $n = 0$. Then $T(0) = 5$ since $n = 0$. But this is $6(0) + 5 = 5$, as required. Now, let k be an integer such that $k \geq 0$ and suppose that $T(k) = 6k + 5$. We will now show that the claim holds for the case $k + 1$. That is, $T(k + 1) = 6(k + 1) + 5 = 6k + 11$. But by definition, $T(k + 1) = T(k) + 6$. However, by the inductive hypothesis, $T(k) = 6k + 5$. Thus, $T(k + 1) = (6k + 5) + 6 = 6(k + 1) + 5$ as to be shown. QED. □

§5.2 Storage Space - Recursion

When computing the amount of storage that is used by a recursive algorithm, one must include storage needed for all of the recursive calls that have been made in this algorithm, including those that have started but have not yet finished. This includes space needed to remember the value of the input and all local variables, as well as the location in the source code for the program where the execution of a statement is in progress (because it included another recursive application of the algorithm). The number of these calls that are in progress is extremely important too. To illustrate these points, we will consider the `fib` algorithm on input $n = 3$.

Example 5.4 (Storage Space for `fib`)

Recall that we had produced a recursion tree for this execution of `fib` for $n = 3$. Consider the execution of $n = 0$, which has been called by $n = 2$, which in turn was called by $n = 3$. These are exactly the calls that are currently in progress. The executions for $n = 3$ and $n = 2$ are currently waiting for $n = 0$ to finish. Therefore, we would need to store all information required at all of these recursive calls.

Here, the "execution of a recursion tree" refers to the nodes of a recursion tree. The *recursion depth of an execution* of a recursive algorithm is the number of other recursive applications that have been started and are still in progress. This is visually the same as the number of edges on the path from the node to the top of the tree. In the previous example, the execution for $n = 0$ has a recursion depth of 2. The *recursion depth of an algorithm* on a given input (that satisfies the precondition) is the maximum of the recursion depths of all executions. This is equivalent to the maximum number of edges on a path from any node to the top in that recursion tree. In the previous example, the recursion depth of the algorithm is equal to 2.

Now, let us consider the following two functions.

1. $R(n)$ is the recursion depth of the algorithm `fib` on input n for any non-negative input n.

2. $S(n)$ is the maximal amount of storage space used by `fib` during its execution on input n.

The recursion trees for executions of `fib` on various inputs will be used to discover recurrences for these functions.

Example 5.5 (Storage Space for `fib`)

In the case that $n = 0$, the recursion tree has no recursive call. Thus, $R(0) = 0$. During execution of this, the only variable stored is n, so $S(0) = 1$. Note that we do not count space for program counters or constants.

For $n = 1$, the recursion tree also has no recursive call, so $R(1) = 0$. The only variable stored is also n, so $S(1) = 1$.

In the case that $n \geq 2$, the algorithm calls itself twice with inputs $n-2$ and $n-1$. The recursion depth in this case is just one more than the maximum of the recursive depth of its left and right subtrees, so it is therefore $R(n) = 1 + \max(R(n-2), R(n-1))$. At this point, we do not yet know that the function is nondecreasing, so we leave this expression as it is for now

$$R(n) = \begin{cases} 0 & \text{if } n = 0 \\ 0 & \text{if } n = 1 \\ 1 + \max(R(n-2), R(n-1)) & \text{if } n \geq 2 \end{cases}$$

Now, recall that there are four stages of execution for $n \geq 2$. Namely, computation before `fib(n-2)`, recursive call of `fib(n-2)`, recursive call of `fib(n-1)`,and the computation after `fib(n-1)`. In the first stage, only n is stored, so this is a storage cost of 1. In the second stage, n is still stored along with the maximum storage requirements of the recursive call $S(n-2)$. Thus, the maximum storage required for the second stage is $S(n-2)+1$. In the third stage, input n and the value returned by `fib(n-2)` is stored, along with the maximum requirements of the recursive call $S(n-1)$. Thus, the maximum storage required for the third stage is $S(n-1)+2$. In the fourth stage, we need a storage space of 3 for n, `fib(n-2)`, and `fib(n-1)`. The maximum storage required is simply the maximum of all four cases. Considering previous results, we find that

$$S(n) = \begin{cases} 1 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \max(S(n-2)+1, S(n-1)+2, 3) & \text{if } n \geq 2 \end{cases}$$

Remark 5.6. It can be shown by induction that the closed form of the recursion depth is $R(n) = n - 1$ for all $n \geq 1$, and the closed form for the storage space is $S(n) = 2n - 1$ for all $n \geq 1$.

§6 September 28, 2017

§6.1 Asymptotic Notation

Asymptotic notation provides information about the relative rates of growth of a pair of functions of a single integer or real variable. It ignores other details such as behaviour on small inputs, multiplicative constants, and lower order terms (which can be implementation or platform dependent). This permits classification of algorithms into classes such as linear, quadratic, polynomial, exponential, etc. Asymptotic notation is also useful for obtaining bounds on running times of algorithms.

Consider a total or partial function $f : \mathbb{N} \to \mathbb{N}$ or $f : \mathbb{R} \to \mathbb{R}$. For $f : \mathbb{N} \to \mathbb{N}$, we say that f is *asymptotically positive* (eventually positive) if there exists a constant c such that when $f(n)$ is defined and $f(n) > 0$ for all $n \in \mathbb{N}$, then $n \geq c$. The real number case is identical in definition. In other words, this simply means that the function is positive from some point onwards.

§6.2 Big-Oh Notation

Suppose that $f, g : \mathbb{N} \to \mathbb{N}$ or $f, g : \mathbb{R} \to \mathbb{R}$, and that f and g are both asymptotically positive. If there exists constants $c > 0$ and $N_0 \geq 0$ such that

$$f(n) \leq c \cdot g(n)$$

for all n in the domain of f such that $n \geq N_0$, then we say that $f \in O(g)$. This means that the rate of growth of f is **at most** g to a multiplicative factor. For example, $4n+3 \in O(n)$ since the definition is satisfied for $c = 5$ and $N_0 = 3$. This is sometimes written as $f = O(g)$ (even though the definition effectively defines an entire set of functions f that are all in the set $O(g)$).

Proposition 6.1
$4x^2 + 2 \in O\left(x^2\right)$.

Proof. By the definition of $O\left(x^2\right)$, it suffices to show that there exists constants $c > 0$ and $N_0 \geq 0$ such that $4x^2 + 2 \leq cx^2$ for all $x \in \mathbb{R}$ such that $x \geq N_0$.

Let $c = 5$ and $N_0 = 2$. Now let $x \in \mathbb{R}$ such that $x \geq N_0$. Then $4x^2+2 \leq 4x^2+x^2 = 5x^2$ because $2 \leq 4 \leq x^2$ whenever $x \geq 2$. Since x was arbitrarily chosen, it follows that $4x^2 + 2 \leq 5x^2 = cx^2$ for all $x \in \mathbb{R}$ where $x \geq N_0$. Because our choices of constants were such that $c > 0$ and $N_0 \geq 0$, this establishes the claim. $\square$

Alternatively, we can invoke the following theorem to prove the previous proposition.

Theorem 6.2 (Big-Oh Limit Test)
Suppose that $f, g : \mathbb{N} \to \mathbb{N}$ or $f, g : \mathbb{R} \to \mathbb{R}$, and that f and g are both asymptotically positive. If the limit

$$\lim_{x\to\infty} \frac{f(x)}{g(x)}$$

exists and is equal to a real constant C (that is not ∞), then $f \in O(g)$.

Remark 6.3. The previous theorem provides a sufficient condition to show that $f \in O(g)$, but it is not a necessary condition. It may be possible that $f \in O(g)$ even though the limit above does not exist and the theorem cannot be applied. We can also apply *l'Hôpital's Rule* when applying the limit in cases when the terms do not cancel easily.

§6.3 Big-Omega Notation

Suppose that $f, g : \mathbb{N} \to \mathbb{N}$ or $f, g : \mathbb{R} \to \mathbb{R}$, and that f and g are both asymptotically positive. If there exists constants $c > 0$ and $N_0 \geq 0$ such that

$$f(n) \geq c \cdot g(n)$$

for all n in the domain of f such that $n \geq N_0$, then we say that $f \in \Omega(g)$. This means that the rate of growth of f is **at least** g to a multiplicative factor. For example, $4n+3 \in \Omega(n)$ since the definition is satisfied for $c = N_0 = 1$.

Theorem 6.4 (Big-Omega Limit Test)

Suppose that $f, g : \mathbb{N} \to \mathbb{N}$ or $f, g : \mathbb{R} \to \mathbb{R}$, and that f and g are both asymptotically positive. If the limit

$$\lim_{x\to\infty} \frac{f(x)}{g(x)}$$

exists and is greater than zero (either resulting in a constant or ∞), then $f \in \Omega(g)$.

Once again, this only provides a sufficient condition, as there are some cases when the above limit does not exist and so the theorem cannot be invoked. However, we can apply another theorem to prove that $f \in \Omega(g)$.

Theorem 6.5 (Transpose Symmetry of O and Ω)

Suppose that $f, g : \mathbb{N} \to \mathbb{N}$ or $f, g : \mathbb{R} \to \mathbb{R}$, and that f and g are both asymptotically positive. It follows that $f \in O(g) \iff g \in \Omega(f)$. The proof follows directly from the definitions, and is left to the reader as an exercise.

§6.4 Big-Theta Notation

Suppose that $f, g : \mathbb{N} \to \mathbb{N}$ or $f, g : \mathbb{R} \to \mathbb{R}$, and that f and g are both asymptotically positive. If there exists constants $c_L, c_U > 0$ and $N_0 \geq 0$ such that

$$c_L \cdot g(n) \leq f(n) \leq c_U \cdot g(n)$$

for all n in the domain of f such that $n \geq N_0$, then we say that $f \in \Theta(g)$. This means that the rate of growth of f is **the same** as g to within a multiplicative factor. For example, $4n + 3 \in \Theta(n)$ since the definition is satisfied for $c_L = 1$, $c_U = 5$, and $N_0 = 3$.

Theorem 6.6 (Equivalence of Big-Theta)

Suppose that $f, g : \mathbb{N} \to \mathbb{N}$ or $f, g : \mathbb{R} \to \mathbb{R}$, and that f and g are both asymptotically positive. Therefore, it follows that $f \in \Theta(g) \iff f \in O(g)$ and $f \in \Omega(g)$. The proof follows directly from the definition.

§6.5 Little-Oh Notation

Suppose that $f, g : \mathbb{N} \to \mathbb{N}$ or $f, g : \mathbb{R} \to \mathbb{R}$, and that f and g are both asymptotically positive. If for all constants $c > 0$ there is a constant $N_0 \geq 0$ such that

$$f(n) \leq c \cdot g(n)$$

for all n in the domain of f such that $n \geq N_0$, then we say that $f \in o(g)$. This means that the rate of growth of f is **less than** g to any multiplicative factor.

Proposition 6.7

$x \in o\left(x^2\right)$

Proof. It follows from the definition of $o\left(x^2\right)$ that it is necessary and sufficient to prove that for every constant $c > 0$, there is a constant $N_0 \geq 0$, so that $x \leq c \cdot x^2$ for all $x \in R$ where $x \geq N_0$.

Now, suppose c is a real constant greater than 0. Now, let $N_0 = 1/c$. Clearly, this is a real value greater than or equal to 0, as required. For all $x \geq N_0 = 1/c$, then it follows that $cx \geq cN_0 = 1$. Thus, the following can be shown.

$$\begin{aligned} x &= 1 \cdot x \\ &= cN_0 \cdot x \\ &\leq cx \cdot x = cx^2. \end{aligned}$$

Thus, we have shown by definition that $x \in o\left(x^2\right)$. $\square$

Alternatively, we can invoke the following theorem.

Theorem 6.8 (Little-Oh Limit Test)

Suppose that $f, g : \mathbb{N} \to \mathbb{N}$ or $f, g : \mathbb{R} \to \mathbb{R}$, and that f and g are both asymptotically positive. If the limit

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} = 0,$$

then $f \in o(g)$.

Remark 6.9. The converse of the previous theorem is also true, so this is a necessary and sufficient condition. Additionally, unlike the synonymous theorems for O and Ω, this limit test can always be used to prove $f \in o(g)$. Note that a proof for $f \in O(g)$ cannot be converted to a proof of $f \in o(g)$ simply by converting the argument from $\leq$ to $<$.

§6.6 Little-Omega Notation

Suppose that $f, g : \mathbb{N} \to \mathbb{N}$ or $f, g : \mathbb{R} \to \mathbb{R}$, and that f and g are both asymptotically positive. If for all constants $c > 0$ there is a constant $N_0 \geq 0$ such that

$$f(n) \geq c \cdot g(n)$$

for all n in the domain of f such that $n \geq N_0$, then we say that $f \in \omega(g)$. This means that the rate of growth of f is **greater than** g to any multiplicative factor. To prove that $f \in \omega(g)$, we follow the same structure as for Little-Oh, but reverse the inequality to $\geq$ instead of $\leq$. Alternatively, we can invoke the following theorems.

Theorem 6.10 (Little-Omega Limit Test)

Suppose that $f, g : \mathbb{N} \to \mathbb{N}$ or $f, g : \mathbb{R} \to \mathbb{R}$, and that f and g are both asymptotically positive. If the limit

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} = \infty,$$

then $f \in \omega(g)$. This limit test can always be used to show that $f \in \omega(g)$

Theorem 6.11 (Transpose Symmetry of o and ω)

Suppose that $f, g : \mathbb{N} \to \mathbb{N}$ or $f, g : \mathbb{R} \to \mathbb{R}$, and that f and g are both asymptotically positive. It follows that $f \in o(g) \iff g \in \omega(f)$. The proof follows directly from the definitions.

§6.7 Standard Functions

A *polynomial function* with degree $d \in \mathbb{N}$ is a function

$$p(n) = a_d n^d + a_{d-1} n^{d-1} + ... + a_1 n + a_0,$$

where $a_d, a_{d-1}, ..., a_1, a_0 \in \mathbb{R}$ and $a_d \neq 0$. If $a_d > 0$, then $p(n) \in \Theta\left(n^d\right)$. If $p(n) \in o\left(n^e\right)$ for all $e \in \mathbb{R}$ where $e > d$, and $p(n) \in \omega\left(n^f\right)$ for all $f \in \mathbb{R}$ where $f < d$.

An *exponential function* of n is a function

$$e(n) = a^n,$$

where $a \in \mathbb{R}$ such that $a > 0$. If $a > 1$, then $e(n) \in \omega(p(n))$ for every polynomial function p, irrespective of the degree of p. If $a = 1$, then $e(n) = 1$, so $e(n) \in \Theta(1)$. If $a < 1$, then $e(n) \in o(1)$.

Furthermore, suppose that $a, b \in \mathbb{R}$, $a, b > 0$, $e_a(n) = a^n$, and $e_b(n) = b^n$. Then, we can draw the following conclusions. If $a > b$, then $e_a(n) \in \omega(e_b(n))$. If $a = b$, then $e_a(n) \in \Theta(e_b(n))$. If $a < b$, then $e_a(n) \in o(e_b(n))$.

A *logarithmic function* is a function

$$l(n) = log_a(n),$$

where $a \in \mathbb{R}$ such that $a > 1$. If $a, b \in \mathbb{R}$ such that $a > 1$, $b > 1$, $l_a(n) = log_a(n)$, and $l_b(n) = log_b(n)$, then $l_a(n) \in \Theta(l_b(n))$. If $a \in \mathbb{R}$ such that $a > 1$, $l_a(n) = log_a(n)$, and p is a polynomial function with degree $d \geq 1$ whose leading coefficient is positive, then $l_a(n) \in o(p(n))$.

§7 October 3, 2017

§7.1 Divide and Conquer - Sorting

The next stage of this course will introduce various kinds of algorithms and design processes that allow one to design such algorithms of their own. We will henceforth be analyzing the *worst case running time* of algorithms as a function of the size of input.

Divide and Conquer algorithms are those that solve a nontrivial instance of a given problem by

1. Using the given instance to form one or more smaller instances of the same problem.
2. Solving these smaller instances using a recursive application of the same algorithm.
3. Using the solutions for the smaller instances to produce a solution for the original.

The trivial instances of the problem are just solved directly without any recursive applications of the algorithm. Most of the time, the algorithm has three stages in which the above steps are carried out. In this case, all of the recursive applications can be performed in parallel. For instance, if steps 1 and 3 can be parallelized, then the algorithm is well suited for *parallel computation*.

Example 7.1 (Sorting Problem)

Consider the sorting problem with the precondition that an integer array A with some positive length n is given as input. The postcondition is that an integer array $\hat{A}$ with the same positive length n is returned as output. $\hat{A}$ stores the values $A[0], A[1], ..., A[n-1]$, but now listed in nondecreasing order. That is, $\hat{A}[i] \leq \hat{A}[i+1]$ for every integer $0 \leq i \leq n-2$. Find a divide and conquer algorithm that accomplishes this.

Solution. Consider the trivial case where $n = 1$. It suffices to return A itself, since there is only one element that is therefore in nondecreasing order. Now, consider the case that $n \geq 2$. To understand how we can sort an array of this size, we shall split the array into two smaller arrays with length $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$. Thus, we consider the subproblem of **splitting an array**.

The precondition for splitting an array is that an array A with a length $n \geq 2$ is given as input. The postcondition is that an array D, of length two of integer arrays $B = D[0]$ and $C = D[1]$, is returned. B has length $\lceil n/2 \rceil$ such that $B[i] = A[i]$ for all $0 \leq i \leq \lceil n/2 \rceil - 1$. C has length $\lfloor n/2 \rfloor$ such that $C[i] = A[\lceil n/2 \rceil + i]$ for all $0 \leq i \leq \lfloor n/2 \rfloor - 1$. It is clear that this problem is easily solved by initializing B and C to the appropriate sizes, then running a loop to copy values from A into B and C as necessary. The result is then stored in D, which is then returned. This can be carried out in linear time with respect to length n. Suppose that this can be done in the worst case in $3n + 10$ steps, and call such a method `split`.

The algorithm will need to recursively sort the arrays $D[0]$ and $D[1]$. Let E be the sorted result of $D[0]$ and F be the sorted result of $D[1]$. Then the entries in E and F combined are exactly the ones found in the original sequence (albeit possibly in a different order). We now need to merge these arrays to produce an array $\hat{A}$ of length $n = \lceil n/2 \rceil + \lfloor n/2 \rfloor$ such that each preceding term is less than or equal to the next. Thus, we consider the subproblem of **merging sorted arrays**.

The precondition for this is that two integer arrays E and F with positive lengths n_E and n_F are provided as input. Note that these arrays must be sorted in nondecreasing order. The postcondition then, is that a single integer array $\hat{A}$ of length $n_E + n_F$ is returned. $\hat{A}$ must consist of those values found in E and F in nondecreasing order. To solve this, we first initialize array $\hat{A}$ and include pointers to the beginning of E and F. So long as we have not passed the end of E and F, we add the smaller of the values pointed to by each pointer to $\hat{A}$, and advance that same pointer. Remaining values are copied into the output array $\hat{A}$. This is clearly linear with respect to $n_E + n_F$. Let us assume that this can be done in at most $5(n_E + n_F) + 7$ steps, and call such a method `merge`.

We now have an algorithm that can efficiently sort. In the case that $n = 1$, then we simply return the original array. When $n \geq 2$, we first call `split` on the array, then call the function itself on inputs $D[0]$ and $D[1]$. Finally, we call `merge` on the results returned from calling the function on $D[0]$ and $D[1]$.

Note that the length n of the input array is a bound function for this `mergeSort` algorithm. If one assumes (or proves) the correctness of the split and merge methods, then it can be shown that the `mergeSort` algorithm correctly solves the Sorting Problem by strong induction on n.

```
integer[] mergeSort (integer[] A)
{
   if (A.length == 1)
```

```
    {
        integer[] Ahat = new Integer[1]
        Ahat[0] = A[0]
        return Ahat
    }
    else
    {
        (integer[])[] D = split(A)
        integer[] E = mergeSort(D[0])
        integer[] F = mergeSort(D[1])
        return merge(E, F)
    }
}
```

■

Example 7.2 (Running Time of `mergeSort`)
Determine the running time of the `mergeSort` algorithm.

Solution. Let $T_{ms}(n)$, $T_m(n)$, and $T_s(n)$ be the number of steps of the `mergeSort`, `merge`, and `split` algorithms in the worst case respectively given that their preconditions are satisfied. It is clear from inspection that

$$T_{ms}(n) \leq \begin{cases} 4 & \text{if } n = 1, \\ T_{ms}(\lceil n/2 \rceil) + T_{ms}(\lfloor n/2 \rfloor) + T_m(n) + T_s(n) + 5 & \text{if } n \geq 2. \end{cases}$$

However, we noted before that $T_s(n) \leq 3n + 10$ and $T_m(n) \leq 5n + 7$. Thus,

$$T_{ms}(n) \leq \begin{cases} 4 & \text{if } n = 1, \\ T_{ms}(\lceil n/2 \rceil) + T_{ms}(\lfloor n/2 \rfloor) + 8n + 22 & \text{if } n \geq 2. \end{cases}$$

It can be shown by induction on n that this is equal to

$$T_{ms}(n) \leq 21n \lceil \log_2 n \rceil + 4,$$

where n is any integer ≥ 1. Thus, the worst case running time of `mergeSort` is $O(n \log_2 n)$. ■

The `mergeSort` algorithm was invented by John von Neumann in 1945. It was the first known sorting algorithm that could be used to sort an array with length n using $\Theta(n \log n)$ steps. It has been proved that there is no comparison-based sorting algorithm that is asymptotically faster.

§8 October 5, 2017

§8.1 Divide and Conquer - Integer Multiplication

For many computational problems involving integers, it is sufficient to consider integers that are reasonably small. These values can all be accurately represented using `int` or `long` data types. On the other hand, applications such as public key cryptography require much larger integers. Thus, libraries that support large integers are available for multiple programming languages.

A *fixed-base representation* is the most commonly used representation of large integers:

- The *base* is an integer constant that is greater than or equal to two.
- The *digit* is a data type that can be used to represent any integer i such that $0 \leq i \leq base$. Indeed, it will be said that i is a *digit* if i is an integer in this range.

The result is that every non-negative integer α has a unique representation of the form

$$\alpha = \sum_{i=0}^{n} a_i \cdot base^i,$$

where $0 \leq a_0, a_1, ..., a_n$. In other words, these are the string of digits that comprise the number when multiplied by their respective power of the base.

Remark 8.1. When $\alpha = 0$, then it follows that $n = 0$ and $a_0 = 0$. We say that the representation has length 0 in this case. On the other hand, if $\alpha \geq 1$, then $n \geq 0$ and $a_n \geq 1$. We now say that this representation has length $n + 1$.

Example 8.2

If $base = 10$, then we have the common decimal representation, and if $base = 2$, then we have binary representation. We also have the $base = 8$ octal representation, and the $base = 16$ hexadecimal representation. The case that is likely most important for practical computation is the case that a *digit* is large enough to be stored in a single word of computer memory without leaving unused storage space. For 64-bit computer memory, $base = 26^4 = 18,446,744,073,709,551,616$.

Large numbers α of the form described will be represented using a `BigInteger` object. In particular, if A is a `BigInteger` representing the integer α, then $A.length$ is the length of the integer α, and $A.digits$ is an array of digits with length at least one if $\alpha = 0$ or at least $n + 1$ otherwise. In particular, $A.digits[i] = a_i$ for every integer $0 \leq i \leq n$.

Negative integers can also be represented. Now, $A.length$ and $A.digits$ is used to represent the absolute value of α. By setting $A.sign$ to be 1 if positive, 0 if zero, or -1 if negative, we can represent negative integers as well.

We take the unit cost operations on *digits* to be as follows.

- Initialization: Setting a new *digit* to a given integer value a such that $0 \leq a \leq base$.
- Comparison: Given *digits* a and b, returning -1 when $a < b$, 0 when $a = b$, and 1 when $a > b$.
- Addition: Given *digits* a and b, returning `BigInteger` $a + b$.
- Subtraction : Given *digits* a and b, returning `BigInteger` $a - b$.
- Multiplication: Given *digits* a and b, returning `BigInteger` $a \cdot b$.

Likewise, we define the unit cost operations on `BigInteger` A to be as follows.

- Sign: Deciding on the sign of α from inspection of $A.sign$.
- Length: Deciding on the length of α by inspection of $A.length$.
- Digits: Determining the digits $a_0, a_1, ..., a_n$ by inspection of $A.digits$.

Now, the linear cost operations on `BigInteger` A will be considered. Suppose that c_0 and c_1 are sufficiently large positive constants.

- Initialization: Let n be a positive integer. The operation to create a new `BigInteger` A of length n representing 0 costs at most $c_1 n + c_0$. $A.digits$ is the array of length n.
- Addition: Let A and B be `BigInteger` representing the integer values α and β. Adding the two and storing them in a `BigInteger` C costs at most $c_1 n + c_0$, where n is the maximum length of A and B. Subtraction has the same cost.
- Multiplication by Power of *base*: Let `BigInteger` A represent the integer α, and let k be a positive integer. The operation of setting `BigInteger` B equal to A times $base^k$ is at most $c_1(n+k) + c_0$, where n is the length of A.
- Division with Remainder by a Power of *base*: Let `BigInteger` A represent the integer α, and let k be a positive integer. The operations of setting `BigInteger` B to A divided by $base^k$ (rounded down), and setting `BigInteger` C to A modulo $base^k$ are at most $c_1 \max(n,k) + c_0$, where n is the length of A.

Example 8.3 (Integer Multiplication Problem)

Consider the integer multiplication problem with the precondition that a pair of `BigInteger` A and B are given as input. The postcondition is that a `BigInteger` C representing the product is returned. Find a divide and conquer algorithm that accomplishes this.

Solution. We will consider the case for positive integers. Note however that if the signs of `BigInteger` can be inspected, then negative integers can be included using only a constant number of additional steps. The standard multiplication algorithm uses quadratic time $\Theta(A.length \cdot B.length)$. However, we shall consider *Karatsuba's algorithm*, which is asymptotically faster. This algorithm was named after Anatoly Karatsuba, who discovered the algorithm in 1960.

We shall now denote $n = \max(A.length, B.length)$. Any inputs such that $n \leq 3$ will be considered a trivial instance of the problem. The standard multiplication algorithm will be used to solve trivial instances. These instances are small enough to assume that that exists a constant c_t such that these instances can be solved in at most c_t steps.

Now, suppose that $n \geq 4$, and let $k = \lceil n/2 \rceil$. It is clear that $k \leq n-2$. Let $A_{low} = A \bmod \left(base^k\right)$ and $A_{high} = A \operatorname{div} \left(base^k\right)$. We now have

$$A = A_{high} \cdot base^k + A_{low},$$

where the length of A_{low} is at most k, since $0 \leq A_{low} \leq base^k - 1$, and the length of A_{high} is at most $n - k = \lfloor n/2 \rfloor \leq k$. Let B_{low} and B_{high} be defined in a similar manner. It follows then that by distributing the terms, the multiplication can be expressed as

$$C = A \cdot B = (A_{high} \cdot B_{high})\, base^{2k} + (A_{high} \cdot B_{low} + A_{low} \cdot B_{high})\, base^k + (A_{low} \cdot B_{low})$$

One might try to develop a divide and conquer algorithm by calculating the four products composed of A_{high}, A_{low}, B_{high}, and B_{low}, but this is actually slower than standard multiplication. Instead, we let $A_{sum} = A_{high} + A_{low}$, where the lengths of A_{high} and A_{low} were $\leq n-2$, so the length of A_{sum} is $\leq n-1$. B_{sum} is similarly defined. The fast integer multiplication will be performed on the smaller recursive cases of $(A_{low} \cdot B_{low})$, $(A_{high} \cdot B_{high})$, and $(A_{sum} \cdot B_{sum})$. Realizing that

$$(A_{sum} \cdot B_{sum}) - (A_{low} \cdot B_{low}) - (A_{high} \cdot B_{high}) = (A_{high} \cdot B_{low} + A_{low} \cdot B_{high}),$$

we have now obtained an efficient solution for integer multiplication.

```
BigInteger kMult (BigInteger A, BigInteger B)
{
   integer n = max(A.length, B.length)
   if (n <= 3)
   {
      // Trivial instance solution using standard integer multiplication to
         return the product of A and B.
   }
   else
   {
      k = ceiling(n/2)

      BigInteger Alow = A * mod(base ^ k)
      BigInteger Ahigh = A * div(base ^ k)
      BigInteger Asum = Ahigh + Alow

      BigInteger Blow = B * mod(base ^ k)
      BigInteger Bhigh = B * div(base ^ k)
      BigInteger Bsum = Bhigh + Blow

      BigInteger Clow = kMult(Alow, Blow)
      BigInteger Chigh = kMult(Ahigh, Bhigh)
      BigInteger Csum = kMult(Asum, Bsum)

      BigInteger D1 = Chigh * base ^ (2 * k)
      BigInteger D2 = Csum - Clow
      BigInteger D3 = D2 - Chigh
      BigInteger D4 = D3 * base ^ k
      BigInteger D5 = D1 + D4
      BigInteger C = D5 + Clow

      return C
   }
}
```

■

It can be shown that the above algorithm is indeed correct using induction on n, where n is also the bound function for this recursive algorithm in the case where $n \geq 4$. We now turn our attention to the running time of this algorithm.

Example 8.4 (Running Time of `kMult`)

Determine the running time of the `kMult` algorithm.

Solution. It is not trivial to determine the running time from this recurrence algorithm, since the size of recursive cases are not fixed. Thus, we will make the simplification that for $n \in \mathbb{N}$, we set $T(n)$ to be the number of steps used by the algorithm when $n \geq \max(A.length, B.length)$ (instead of an equal sign). $T(n)$ is therefore an upper bound on the number of steps when $\max(A.length, B.length) = n$, so it suffices to find an upper bound for $T(n)$. It can be seen by its definition that it is a nondecreasing function of n, so we can use the maximum input size for each recursive instance when forming a bound on $T(n)$.

Suppose that $\max(A.length, B.length) \leq 3$. The standard multiplication algorithm is used. Since A and B are at most a constant length, there is a positive integer d_0 such that the algorithm uses at most d_0 steps on A and B.

Now, suppose instead that $\max(A.length, B.length) \geq 4$. Three smaller instances of the problem are formed and recursively solved.

1. `Alow` and `Blow` are integers in the range of 0 and $base^k - 1$, so their length is at most $k = \lceil n/2 \rceil$. The number of steps required to solve `kMult` on this instance is $T(\lceil n/2 \rceil)$.

2. `Ahigh` and `Bhigh` are integers in the range of 0 and $base^{n-k} - 1$, with length at most $n - k = \lfloor n/2 \rfloor$. This is because `A` and `B` are integers in the range of 0 to $base^n - 1$. The number of steps required to solve `kMult` on this instance is $T(\lfloor n/2 \rfloor)$.

3. `Asum` and `Bsum` are integers in the range of 0 to $base^{k+1} - 1$. Each have a length of at most $k + 1 = \lceil n/2 \rceil + 1$. The number of steps required to solve `kMult` on this instance is $T(\lceil n/2 \rceil + 1)$.

Thus, since T is a nondecreasing function of n, the total number of recursive steps required to solve all three smaller instances of the problem is therefore

$$3T(\lceil n/2 \rceil + 1) = \begin{cases} 3T\left(\frac{n+2}{2}\right) & \text{if } n \text{ is even,} \\ 3T\left(\frac{n+3}{2}\right) & \text{if } n \text{ is odd.} \end{cases}$$

All of the other steps require at most a linear number of steps with respect to n. It follows that there are constants d_1 and d_2 (related to the constants c_1 and c_0 when listing the cost of operations) that result when we sum all of the relevant non-recursive operations in the algorithm for the case where $n \geq 4$. Thus,

$$T(n) \leq \begin{cases} d_0 & \text{if } n \leq 3, \\ 3T\left(\frac{n+2}{2}\right) + d_1 n + d_2 & \text{if } n \geq 4 \text{ and } n \text{ is even,} \\ 3T\left(\frac{n+3}{2}\right) + d_1 n + d_2 & \text{if } n \geq 4 \text{ and } n \text{ is odd.} \end{cases}$$

We would now like to solve the recurrence, but the input values to T on the right of the recurrence relations above are slightly greater than $n/2$. This complicates the use of induction to arrive at a solution for the recurrence. To remedy this, we use a change of variables to overcome this. Let $U(n) = T(n+3)$. Now, we find that if $n = 0$, then

$$U(0) = T(3) \leq d_0.$$

If $n \geq 1$ and is odd, then it follows that in T, $n + 3 \geq 3$ and $n + 3$ is even. In this case,

$$\begin{aligned} U(n) = T(n+3) &\leq 3T\left(\frac{(n+3)+2}{2}\right) + d_1(n+3) + d_2 \\ &\leq 3U\left(\frac{n-1}{2}\right) + d_1 n + 3d_1 + d_2 \end{aligned}$$

If $n \geq 1$ and is even, then in T, $n + 3 \geq 4$ and $n + 3$ is odd. In this case,

$$\begin{aligned} U(n) = T(n+3) &\leq 3T\left(\frac{(n+3)+3}{2}\right) + d_1(n+3) + d_2 \\ &\leq 3U\left(\frac{n}{2}\right) + d_1 n + 3d_1 + d_2 \end{aligned}$$

Thus, we obtain the recurrence derived with respect to n with positive constants d_0, d_1, and d_2,

$$U(n) = \begin{cases} d_0 & \text{if } n = 0, \\ 3d_0 + 4d_1 + d_2 & \text{if } n = 1, \\ 3U\left(\frac{n}{2}\right) + d_1 n + 3d_1 + d_2 & \text{if } n \geq 2 \text{ and } n \text{ is even}, \\ 3U\left(\frac{n-1}{2}\right) + d_1 n + 3d_1 + d_2 & \text{if } n \geq 2 \text{ and } n \text{ is odd}. \end{cases}$$

The additive terms in the recurrence complicate an inductive proof of the solution. However, once the dominant term of the solution has been discovered, one can subtract away lower order terms by cancelling things out in a proof by induction. Leaving constants as unknown values and documenting the conditions on these constants that are required for a proof can allow one to discover values for these constants that permit a proof to be completed. It can be shown by induction using the above recurrence relation for $U(n)$ that for positive constants $\alpha = 3d_0 + 9d_1 + 3d_2/2$, $\beta = 2d_1$, and $\gamma = 3d_1 + d_2/2$, it is the case that $U(n) \leq \alpha n^{\log_2 3} - \beta n - \gamma$. Because T and U are nondecreasing, it follows that

$$T(n) = U(n-3) \leq U(n) \leq \alpha n^{\log_2 3} - \beta n - \gamma \leq \alpha n^{\log_2 3} \in O\left(n^{\log_2 3}\right).$$

Because $\log_2 3 < 1.59$, we have shown that this algorithm is indeed faster than the standard algorithm. ■

Remark 8.5. Some asymptotically fast algorithms are currently only of theoretical interest. When an algorithm is discovered, hidden multiplicative constants when Big Oh notation is used may be so large that the algorithm is not practical. However, as processor speeds increase and larger instances of problems are considered, this can result in emerging uses for an algorithm.

§8.2 Master Theorem

Theorem 8.6 (Master Theorem)
Let a, b, and c be constants such that $a, c \geq 1$ and $b \geq 1$. Let $f : \mathbb{N} \to \mathbb{N}$ be a total function, and suppose that $T : \mathbb{N} \to \mathbb{N}$ such that for all $n \in \mathbb{N}$,

$$T(n) = \begin{cases} c & \text{if } n < b, \\ aT\left(\lfloor n/b \rfloor\right) + f(n) & \text{if } n \geq b. \end{cases}$$

1. If $f(n) \in O\left(n^{\log_b(a)-\epsilon}\right)$ for a positive constant ϵ, then $T(n) \in \Theta\left(n^{\log_b(a)}\right)$.
2. If $f(n) \in \Theta\left(n^{\log_b(a)}\right)$, then $T(n) \in \Theta\left(n^{\log_b(a)} \log_2(n)\right)$.
3. If $f(n) \in \Omega\left(n^{\log_b(a)+\epsilon}\right)$ for a positive constant ϵ, and for every integer $0 \leq m \leq n/b$ it holds that $af(m) \leq df(n)$ for a positive constant $d < 1$, then $T(n) \in \Theta(f(n))$.

The Master Theorem can be applied to certain recurrences For instance, the second result of the theorem can be used to show that `mergeSort` uses $O(n \log_2 n)$ steps in the worst case. Application of the first result can show that the Karatsuba algorithm is bounded by $O\left(n^{\log_2 3}\right)$.

§9 October 10, 2017

§9.1 Divide and Conquer - Closest Points in a Plane

Consider a pair of points $\alpha = (x_1, y_1)$ and $\beta = (x_2, y_2)$, where $x_1, x_2, y_1, y_2 \in \mathbb{R}$. The distance between the two points is given by $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. However, it is simpler to work with the square of the distance, so that $\delta(\alpha, \beta) = (x_1 - x_2)^2 + (y_1 - y_2)^2$. Notice that if α, β, μ and ν are points in the plane, then α and β are closer together than μ and ν if and only if $\delta(\alpha, \beta) < \delta(\mu, \nu)$. The distance between α and β, and between μ and ν, is the same if and only if $\delta(\alpha, \beta) = \delta(\mu, \nu)$. Similarly, α and β are farther apart than μ and ν if and only if $\delta(\alpha, \beta) > \delta(\mu, \nu)$. Thus, if we want to identify a pair of points in the plane that are closer together than any other pair, then the following computational problem is of interest.

Example 9.1 (Closest Pair of Points Problem)
Consider the problem of finding the closest pair of points. The precondition is that $\alpha_1 = (x_1, x_2), \alpha_2 = (x_2, y_2), ..., \alpha_n = (x_n, y_n)$ is given as input, where $x_1, x_2, ..., x_n, y_1, y_2, ..., y_n \in \mathbb{R}$ and $n \geq 2$. The postcondition is that an ordered pair (i, j) of integers is returned, where these are the indices of the pair of points α_i and α_j such that they are the closest pair. Find a divide and conquer algorithm that accomplishes this. Similarly, we assume that real arithmetic is exact. Because we assume the uniform cost criterion, $\delta(\alpha_i, \alpha_j)$ can be computed exactly at unit cost, and one can also decide whether this is less than, equal to, or greater than another computed value at unit cost.

Solution. Generally, not all real numbers can be represented exactly. Fixed precision real arithmetic is generally not exact, as the results of computations are only approximations of the values that should be computed. For our purposes however, we will assume that the values are given exactly, since errors arising from this assumption are generally difficult to detect.

A naive solution would compute $\delta(\alpha_i, \alpha_j)$ for all $i, j \in \mathbb{N}$ where $1 \leq i < j, \leq n$. By keeping track of the smallest distance and the pair of integers i and j that accomplish this, the problem can be solved. Because $\delta(\alpha_i, \alpha_j)$ must be computed and used to initialize the variable that stores the minimum distance so far, we have $\binom{n}{2}$ choices of i and j. It follows from $\binom{n}{2} = \frac{n(n-1)}{2}$ that this algorithm uses $\Theta\left(n^2\right)$ steps.

While the previous solution is simple, it is also unnecessarily expensive. We now consider a preprocessing step that allows us to simplify the problem. We first calculate and store intermediate data that will be useful later in the algorithm.

- An array P_x of ordered pairs of points and integers of length n, storing $(\alpha_1, 1), (\alpha_2, 2)$, ..., (α_n, n). This is simply reordered so that they are listed by non-decreasing order of x coordinates. This is useful, as the problem becomes simplified if one can assume that $x_1, \leq x_2 \leq ... \leq x_n$. This is because it is much easier to form smaller instances of the problem from a nontrivial instance in a useful way using this simplification.

  This constraint is satisfied if we apply an algorithm to solve this problem when the original inputs are reordered. However, the output that is returned must now be adjusted as well, in order to reflect the fact that the input points were reordered. Therefore, it will be assumed that inputs are $\alpha_1 = (x_1, x_2), \alpha_2 = (x_2, y_2), ..., \alpha_n = (x_n, y_n)$, with the additional knowledge that $x_1 \leq x_2 \leq ... \leq x_n$.

- An array P_y of ordered pairs of points and integers of length n storing $(\alpha_1, 1), (\alpha_2, 2)$, ..., (α_n, n). This is simply reordered so that they are listed by non-decreasing order of y coordinates. This allows us to recover a solution for an originally given nontrivial instance of the problem from the solutions of smaller instances.

It is clear that the `mergeSort` algorithm can be used to create arrays P_x and P_y using $O(n \log n)$ steps for a problem with n points.

We now solve a new version of the problem, such that the precondition now becomes $\alpha_1 = (x_1, x_2), \alpha_2 = (x_2, y_2), ..., \alpha_n = (x_n, y_n)$ given as input, where $x_1, x_2, ..., x_n, y_1, y_2, ..., y_n \in \mathbb{R}$ such that $x_1 \leq x_2 \leq ... \leq x_n$, and $n \geq 2$. Additionally, we also require as input an array P_y of ordered pairs of points and integer storing $(\alpha_1, 1), (\alpha_2, 2), ..., (\alpha_n, n)$ reordered so that these are listed by nondecreasing y coordinates of the points. The postcondition is the same as before.

Suppose that any instance of the problem such that $n \leq 3$ is considered to be a trivial instance of the problem. When $n = 2$, then there is only one pair of points to consider, so that the values $i = 1$ and $j = 2$ are returned. When $n = 3$, then the only pairs we need to consider are (α_1, α_2), (α_1, α_3), and (α_2, α_3). Thus, it is possible to solve the trivial instances of the problem using at most a small positive number of steps. Suppose that `solveTrivial` is a method that accesses the inputs for the problem as global data and reports a solution for this instance of the problem using at most a constant number of steps whenever $n \leq 3$.

In the nontrivial case when $n \geq 4$, then we note that $2 \leq \lfloor n/2 \rfloor \leq \lceil n/2 \rceil \leq n - 2$. Thus, it is possible to form two smaller instances with sizes $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ such that each input point is in exactly one of these two instances.

- The first instance includes the points $\beta_i = a_i$ for all $1 \leq i \leq \lceil n/2 \rceil$.
- The second instance includes the points $\gamma_i = \alpha_{\lceil n/2 \rceil + i}$ for all $1 \leq i \leq \lfloor n/2 \rfloor$. Note also that the entires β and γ are sorted by nondecreasing order of x, since the entries of α are.
- The first instance includes the array Q_y with length $\lceil n/2 \rceil$ that stores that values $(\beta_1, 1), (\beta_2, 2), ..., (\beta_{\lceil n/2 \rceil}, \lceil n/2 \rceil)$ reordered in nondecreasing order of y coordinates.
- The second instance includes the array R_y with length $\lfloor n/2 \rfloor$ that stores that values $(\gamma_1, 1), (\gamma_2, 2), ..., (\gamma_{\lfloor n/2 \rfloor}, \lfloor n/2 \rfloor)$ reordered in nondecreasing order of y coordinates.

It is clear that the points β and γ can be formed using a pair of simple while loops, and a small number of extra steps, using $\Theta(n)$ steps in total. The arrays Q_y and R_y can be formed from P_y using $\Theta(n)$ steps as well, by examining the entry $P_y[i]$ for $0 \leq i < n$ and deciding which array Q_y or R_y it should go into. Suppose there is a method `split` that receives P_y as input and produces the ordered pair (Q_y, R_y) as output using $\Theta(n)$ steps.

Now, suppose that (i_1, j_1) is the recursively derived solution for β, and that (i_2, j_2) is the recursively derived solution for γ. These indicate the closest pair of points in the lower and upper half of the points organized by x respectively. One can then compute $\delta(\alpha_{i_1}, \alpha_{j_1})$ and $\delta\left(\alpha_{\lceil n/2 \rceil + i_2}, \alpha_{\lceil n/2 \rceil + j_2}\right)$ to obtain a pair of integers $1 \leq i, j \leq n$ such that $\delta(\alpha_i, \alpha_j) \leq \delta(\alpha_k, \alpha_l)$ for all integers k and l either both in the lower instance, or both in the upper instance. This requires only a constant number of steps. However, there are too many pairs of points with one in the lower instance and one in the higher instance for all of these to be checked efficiently. Thus, the solution cannot be recovered in this way.

Lemma 9.2

Recall that $\alpha_{\lceil n/2\rceil} = \left(x_{\lceil n/2\rceil}, y_{\lceil n/2\rceil}\right)$ for $x_{\lceil n/2\rceil}, y_{\lceil n/2\rceil} \in \mathbb{R}$. Let $mid = x_{\lceil n/2\rceil}$. If $1 \le i \le n$ and $a_i = (x_i, y_i)$ such that

■

Remark 9.3. This problem was considered by Michael Ian Shamos and Dan Hoey in the 1970s as part of early work in the then emerging field of *computational geometry*. The problem has applications in multiple areas, including computer graphics, computer vision, geographic information systems, and molecular modeling.

§10 October 12, 2017

§10.1 Divide and Conquer - Median Finding and Selection

Recall that the `quickSort` algorithm works best when the pivot element chosen at the beginning is close to the middle of the values in the array that is to be sorted.

The *lower median* of a sequence $a_1, a_2, ..., a_n$ of n distinct positive integers is the integer b such that b is among the entries of the sequence, and exactly $\lfloor n/2\rfloor$ elements of the sequence are less than or equal to b. This also means that exactly $\lceil n/2\rceil$ elements of the sequence are greater than b. The precondition of the median finding problem is that an array of length n storing n distinct integers is given as input. The postcondition is that the median of the sequence of values stored is returned. To solve this, we will consider a generalization of the median finding problem.

Example 10.1 (Selection Problem)

The precondition is that an array of length n storing a sequence $a_1, a_2, ..., a_n$ of distinct integers, and an integer k such that $1 \le k \le n$ are given as input. The postcondition is that the integer $b = a_i$ for some integer $1 \le i \le n$ is returned. It must be the case that exactly k elements of the sequence $a_1, a_2, ..., a_n$ are less than or equal to b. Solve the selection problem using a divide and conquer algorithm.

Solution. The selection problem is a generalization of the median finding problem. One could solve the selection problem by using `mergeSort` to sort the input array, and then return the element at position $k-1$ of the resulting array. This would clearly require $\Theta(n \log_2 n)$ operations in the worst case. However, we can actually accomplish this in linear time.

We will consider any instance of the problem with $n < 120$ to be trivial. Each of these can be solved using at most a constant number of steps using the `mergeSort` algorithm. Now, suppose that $n \ge 120$. Let $h = \lceil n/5\rceil$ and split the input sequence into h subsequences so that each have a length of at most five. This means that for sequence i where $1 \le i \le h-1$, there are five input values. Consequently, the last sequence h contains between one and five values.

Let b_i for $1 \le i \le h$ be the median of subsequence i. Each of these can be computed using the `mergeSort` algorithm since they are only of length 5. Thus, all of these values can be computed in $O(n)$ steps. We can now produce an algorithm `form1` that takes the original input array, and returns as output at array B of length h that stores the values $b_1, b_2, ..., b_h$ using $\Theta(n)$ steps in the worst case.

The first smaller instance of the selection problem that should be recursively solved includes the above array B along with the integer input $\lfloor h/2 \rfloor$, so that the median of the sequence $b_1, b_2, ..., b_h$ is returned. That is, we find the median of the medians obtained from each of the h subsequences, and call this value c. By definition of lower median, there are $\lfloor h/2 \rfloor$ elements of $b_1, b_2, ..., b_n$ that are less than or equal to c.

For $1 \leq i \leq h$, exactly two elements in that subsequence i are greater than b_i, and exactly two elements are less than b_i. Recall that $h = \lceil n/5 \rceil < n/5 + 1 < n$. Thus, we note that

$$c > \left\lfloor \frac{h}{2} \right\rfloor - 1 \geq \left\lfloor \frac{(n/5)}{2} \right\rfloor - 1 \geq \frac{n}{10} - 2$$

elements of the $b_1, b_2, ..., b_h$. For each value of b_i that c is greater than, this corresponds to three values of a that c is greater than. Thus, c is greater than at least $3(n/10 - 2) = 3n/10 - 6$ elements of $a_1, a_2, ..., a_n$. Through similar reasoning, c is also less than $3n/10 - 6$ elements of $a_1, a_2, ..., a_n$.

Let l denote the number of values in the sequence $a_1, a_2, ..., a_n$ that are less than c. It follows that $3n/10 - 6 \leq l \leq 7n/10 + 5$. Using c, the following can be computed using $\Theta(n)$ steps.

- The value l defined above.
- An array C_L, with length $l \leq 7n/10 + 5$ including all the entries in the sequence $a_1, a_2, ..., a_n$ that are less than c.
- An array C_U, with length $n - l - 1 \leq 7n/10 + 5$ including all the entries in the sequence $a_1, a_2, ..., a_n$ that are greater than c.

Note that if $n \geq 120$, then $7n/10 + 5 < n$. We shall assume that there is an algorithm `form2` that returns an ordered pair of the above arrays C_L and C_U when given the input array and c as input in $\Theta(n)$ operations in the worst case.

Now, when $l \geq k$, then the second instance of the problem that should be recursively solved should include the array C_L of length l along with the integer k. When $l \leq k - 2$, then the second instance of the problem that should be recursively solved should include the array C_U with length $n - l - 1$ along with the integer $k - l - 1$. Lastly, in the case that $l = k?1$, then the value c should be returned as output.

```
integer select (integer[] A, integer k)
{
   if (A.length < 120)
   {
      integer[] B = mergeSort(A)
      return B[k - 1]
   }
   else
   {
      integer h = ceiling(n / 5)
      integer[] B = form1(A)

      integer c = select(B, floor(h / 2))
      (integer[])[] (CL, CU) = form2(A, c)
      integer l = CL.length

      if (l >= k)
      {
         return select(CL, k)
      }
      else if (l <= k - 2)
      {
         return select(CU, k - l - 1)
      }
      else
      {
         return c
      }
   }
}
```

■

Remark 10.2. Note that the recursive calls of `select` are well formed instances of the selection problem that call on inputs smaller than those of the original. It can be shown by strong induction on n that this algorithm is correct, using a base case of $1 \le n \le 39$. The proof is left to the reader.

Example 10.3 (Running Time of `select`)
Determine the running time of the `select` algorithm.

Solution. It can be established that there exists a positive constant c_1 such that this algorithm uses at most $20c_1n$ steps when it is executed with $1 \le n \le 119$. Additionally, when the algorithm is executed for $n \ge 120$, the number of steps taken is at most the sum of c_1n and the number of steps used by any recursive applications.

For $n \ge 1$, let $T(n)$ be maximum number of steps used by this algorithm when the problem's precondition is satisfied and the input includes an array with length at most n. $T(n)$ is then a nondecreasing function, and it follows from the choice of c_1 and the sizes of subproblems that must be formed that

$$T(n) \le \begin{cases} 20c_1n & \text{if } 1 \le n \le 119, \\ T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\left\lfloor \frac{7n}{10} + 5 \right\rfloor\right) + c_1n & \text{if } n \ge 120. \end{cases}$$

Note additionally that if $n \ge 120$, then the sum of the sizes of inputs for smaller instances that are recursively solved is at most

$$\begin{aligned} \left\lceil \frac{n}{5} \right\rceil + \left\lfloor \frac{7n}{10} + 5 \right\rfloor &\le \frac{n}{5} + 1 + \frac{7n}{10} + 5 \\ &\le \frac{9n}{10} + 6 \\ &\le \frac{9n}{10} + \frac{n}{20} \\ &\le \frac{19n}{20} \end{aligned}$$

This can be used to prove by strong induction on n, that $T(n) \le 20c_1n$ for every positive integer n, with the base case being $1 \le n \le 119$.

■

Remark 10.4. If this algorithm is used to select the pivot element as the median when implementing the `quickSort` algorithm, then the resulting algorithm uses $\Theta(nlog_2n)$

steps to sort an array with length n in the worst case. However, this version is not significantly better than mergeSort, because of the additional costs arising from the inclusion of the select algorithm. Because the selection problem is so fundamental, a worst case linear time algorithm to solve it is still of interest.

§11 October 17, 2017

§11.1 Dynamic Programming

A *Dynamic Programming* algorithm is an algorithm that is derived from a recursive algorithm that solves the same computational program. However, the dynamic programming algorithm solves instances from the bottom up, starting with trivial instances. It stores solutions as it generates them, and looks them up later instead of computing them all over again.

Recall the `fib` algorithm. The issue with this algorithm is that it compute the same values repeatedly. Note that we only need the values F_i where $0 \leq i \leq n$ to be computed by the recursive algorithm when we want to find F_{n+1}. These values can be stored in an array F with length $n + 1$ to store F_i for $0 \leq i \leq n$. If these values are computed and stored by increasing order of i, then for $2 \leq i \leq n$, the values of F_{i-2} and F_{i-1} are already stored in location $i - 2$ and $i - 1$ of the array. So to compute F_i, we simply add the results of those previous array elements. Thus, after storing F_0 and F_1 in locations 0 and 1 of the array, a while loop can be used to store F_i in location i of the array for $2 \leq i \leq n$. The value of $F[n]$ can then be returned as output. The code for this is presented below.

```
integer fibDP (integer n)
{
   if (n == 0)
   {
      return 0
   }
   else if (n == 1)
   {
      return 1
   }
   else
   {
      integer[] F = new integer[n+1]
      F[0] = 0
      F[1] = 1

      integer i = 1
      while (i < n)
      {
         F[i + 1] = F[i - 1] + F[i]
         i = i + 1
      }

      return F[n]
   }
}
```

The proof that `fibDP` correctly solves the Fibonacci number computation problem is similar to the proof for `fibLoop`. That is, the loop invariant now includes information on $n \geq 2$ instead of $n \geq 1$, and the case that $n = 1$ now needs to be treated as a special case in proofs of partial correctness and termination. Modification of the proof for `fibLoop` into a proof for `fibDP` is left as an exercise. It is also easy to determine the number of steps used by the algorithm when executed on non-negative integers n,

$$T(n) = \begin{cases} 2 & \text{if } n = 0, \\ 3 & \text{if } n = 1, \\ n + 2(n-1) + 7 = 3n + 5 & \text{if } n \geq 2. \end{cases}$$

One can arrive at a dynamic programming algorithm by following these steps.

1. Design a Divide and Conquer algorithm that solves this computational problem and prove that the algorithm is correct.
2. Determine which other instances of the same problem must be formed and recursively solved when the Divide and Conquer algorithm is used to solve the original instance of the problem. If the number of smaller instances is reasonably small, then this algorithm may be a good candidate for optimization using Dynamic Programming.
3. Choose a data structure that can store these solutions for smaller problems.
4. Identify an order in which these solutions can be solved so that solutions for smaller problems have already been solved and stored before they are needed.
5. Trivial cases can be recognized and handled in the same way as divide and conquer algorithms. For nontrivial cases, the data structure is initialized, and smaller instances are solved and stored. These solutions are then accessed from the data structure at a later point, when they are required for computation.

While the time requirements do not change drastically, storage requirements may be significantly improved by only storing solutions for smaller problems as needed. When values are not used, they can be erased. Thus, we consider an improved version of the previous dynamic programming algorithm `fibDP`.

```
integer betterFibDP (integer n)
{
   if (n == 0)
   {
      return 0
   }
   else if (n == 1)
   {
      return 1
   }
   else
   {
      integer oldest = 0
      integer middle = 1
      integer i = 1

      while (i < n)
      {
         integer youngest = oldest + middle
         oldest = middle
         middle = youngest
         i = i + 1
      }
      return middle.
   }
}
```

§11.2 Memoization

Memoization is an algorithm design technique that is similar to divide and conquer. However, it produces recursive algorithms instead of algorithms with while loops. They work from the top down instead of from the bottom up, and include code that closely resembles the inefficient divide and conquer algorithm that one starts with. We are essentially storing each result and every smaller result in an array. If we later invoke the function on any value less than or equal to the initial value, then we can immediately return the result stored in the array. If we invoke the function on a larger value, then we have minimized the number of recursive calls. Memoization allows an algorithm to be more time efficient the more often it is called, resulting in an eventual overall speed up.

Suppose again that we want to compute the nth Fibonacci number. For `fibDP`, we used an array F with length $n+1$ to store the values $F_0, F_1, F_2, ..., F_n$. Suppose that `initializeF` is an algorithm that receives n as input and returns an array with length $n+1$ as output so that nothing is initially stored, completing in $3n+7$ steps. Now, consider the use of this array F as global data such that when it is executed on input $0 \leq i \leq n$, $F[i]$ is checked. If it is not empty, then the value is returned and execution halts. Otherwise, code almost identical to `fib` is used to compute F_i. That is, the algorithm calls itself recursively, with $F[i]$ being set to F_i immediately before F_i is returned. The algorithm that does this is presented below.

```
integer recFib (integer i)
{
   if (F[i] != null)
   {
      return F[i]
   }
   else if (i == 0)
   {
      F[i] = 0
      return 0
   }
   else if (i == 1)
   {
      F[i] = 1
      return 1
   }
   else
   {
      F[i] = recFib(i - 2) + recFib(i - 1)
      return F[i]
   }
}
```

We can now deduce a memoized algorithm that solves the problem. This involves initializing an array to store computed values.

```
integer[] memFib (integer n)
```

```
{
   if (n == 0)
   {
      return 0
   }
   else if (n == 1)
   {
      return 1
   }
   else
   {
      integer[] F = initializeF(n)
      return recFib(n)
   }
}
```

Example 11.1

Prove that `memFib` is correct. We make the following claims.

1. **First Claim**: If i is an integer such that $0 \leq i \leq n$ and the `recFib` algorithm is executed for the first time as part of an execution of the `memFib` algorithm with input n, then `F[i] = null` when this execution of the algorithm begins.

2. **Second Claim**: Suppose that i is an integer such that $0 \leq i \leq n$ and the `recFib` algorithm is executed with input i when `F[i]` is equal to F_i at the beginning of the execution of the algorithm. This execution of the algorithm will terminate with F_i returned as output, with the entry in the array unchanged.

3. **Third Claim**: For every integer i such that $0 \leq i \leq n$, every execution of `recFib` on i ends with F_i as output. Additionally, `F[i]` is equal to F_i when the algorithm ends.

4. **Fourth Claim**: The `memFib` algorithm is correct.

Solution. To prove that the `memFib` algorithm is correct, we will proceed by proving the separate claims made in the example.

1. Proof of the first claim follows directly from inspection of the code. Recall that `initializeF` returns an array F with length $n+1$ such that `F[i ] = null` for every integer i such that $0 \leq i \leq n$. During an execution of the `recFib` algorithm, the array F is only modified when `F[i ]` is modified, where i was received as input. If this is the first time that the algorithm is being executed with input i, then the value cannot have been changed. It is therefore equal to `null` as claimed.

2. By inspection of the code, since the array entry is already equal to F_i, then it is clearly not null. Thus, the test checking that it is not null is true, so the value is returned. This value is returned without modifying anything in the array.

3. We use strong induction on i and consider the base cases of $i = 0$ and $i = 1$. For the first execution on any input i, we apply the **First Claim** to conclude that the test that the element at index i is not null fails. Tracing the code and applying the inductive hypothesis, we confirm that `F[i]` is set to be F_i and this is returned for the cases $i = 0$, $i = 1$, and $2 \leq i \leq n$.

For subsequent executions, we prove the subclaim that if j is an integer $j \geq 2$, and the algorithm is executed with input i at least j times, then the j execution also halts with F_i returned as output and `F[i]` equal to F_i when the algorithm ends. This is proved by induction on j using the **Second Claim**.

4. By inspection of the code and the **Third Claim** when the input n is greater than or equal to n, we have shown that the algorithm is correct.

■

§11.3 Memoization Efficiency

The `recursion tree method` is useful for bounding the number of steps performed by `recFib` when it is executed on $n \geq 2$. The recursion tree for $n = 5$ starts by splitting to 3 and 4. The 3 splits into 1 and 2, and this 2 splits into 0 and 1. The 4 from before splits into 2 and 3. Note that this 3 no longer splits, since the previous value was memoized into the array, and can be called subsequently without additional recursive calculations.

One way to count the number of steps executed by `recFib` is to charge to each node the number of steps used by the corresponding execution (excluding the cost of recursive calls). The sum of all these charges will equal the total number of steps used by the original application of the algorithm.

Example 11.2 (Efficiency of `memFib`)
Determine the efficiency of `memFib` on input $n = 5$. We make the following claims.

1. **Fifth Claim**: If $n \geq 2$, the the recursion tree for an execution of the `recFib` algorithm with input n has exactly $n-1$ internal nodes. Specifically, we have one for every execution of $n = i$ where $2 \leq i \leq n$.

2. **Sixth Claim**: If $n \geq 2$, the recursion tree for an execution of the `recFib` algorithm on input n includes at least one node corresponding to each of inputs 0 and 1.

3. **Seventh Claim**: If $n \geq 2$, the recursion tree has $n-2$ leaves corresponding to executions that are not the first on those inputs.

Solution. We will prove the claims above in order to arrive at the efficiency of the algorithm.

1. First note that there are four internal nodes in the recursion tree. Specifically, we have one for every i, where $2 \leq i \leq 5$. It can be shown by induction on n that every execution on input n eventually includes an execution with input i for $2 \leq i \leq n$. For all such i, the first such execution corresponds to an internal node in the tree because it required computation. All later executions with these input values correspond to leaves in the tree, because the test that `F[i]` is not null is passed. Since inputs 0 and 1 are trivial cases, one can see (by inspection of the code) that all executions of the algorithm with these inputs correspond to leaves instead of internal nodes. Since there are no other recursive executions, this establishes the claim. We can see that every execution of an internal node requires five steps. Thus, for $n-1$ internal nodes, this requires $5(n-1) = 5n-5$ steps.

2. We can also see that for the recursion tree, there are leaves corresponding to the first executions of the algorithm on inputs 0 and 1. The **Fifth Claim** implies that

execution includes a first execution of the algorithm with input 2. Thus, one can see by inspection that this includes recursive applications with inputs 0 and 1, as required to establish the claim. The first execution with input 0 and 1 require 5 and 4 steps respectively. Thus, first executions of trivial instances require 9 steps in total.

3. The recursion tree also contains three other leaves corresponding to executions that are not the first execution (specifically for inputs 1, 2, and 3). It follows by the `Fifth Claim` that this recursion tree always has exactly $n-1$ internal nodes. It can also be seen by inspection that the recursion tree is always a binary tree such that every internal node has exactly two children. Thus, it can be shown by induction on the number of internal nodes, that every binary tree with this property has exactly one more leaf than it has internal nodes. So for this recursion tree, we have exactly n leaves. Since two of those correspond to the first executions on 0 and 1, this leaves $n-2$ leaves for executions that are not the first on those inputs, thus proving the claim. Each of these contribute 2 steps, so we have $2(n-2) = 2n-4$ steps.

Therefore, the total number of steps required for `recFib` is $(5n-5)+9+(2n-4) = 7n$. By inspection of `memFib`, we find that the number of steps used by this algorithm becomes

$$T(n) = \begin{cases} 2 & \text{if } n = 0, \\ 3 & \text{if } n = 1, \\ (3n+7)+4+(7n) = 10n+11 & \text{if } n \geq 2. \end{cases}$$

■

One can arrive at a memoization algorithm by following these steps.

1. Design a Divide and Conquer algorithm that solves this computational problem and prove that the algorithm is correct.

2. Determine which other instances of the same problem must be formed and recursively solved when the Divide and Conquer algorithm is used to solve the original instance of the problem. If the number of smaller instances is reasonably small, then this algorithm may be a good candidate for optimization using Memoization.

3. Choose a data structure that can store these solutions for smaller problems.

4. Write an algorithm such as `initializeF` to initialize the data structure using `null` or some other value indicating that the corresponding stance of the problem has not yet been solved. Do this for every entry of the data structure.

5. Write a recursive algorithm such as `recFib` that receives an instance of the problem as input and uses the data structure as global data. This algorithm first accesses the data structure to check whether the input instance of the problem has been solved already. If it has been solved, then the solution is read from the data structure and returned as output with no changes made to the data structure. Otherwise, the process followed by the original Divide and Conquer algorithm is used to solve the input instance by calling itself recursively. This solution is written to the data structure immediately before being returned as output.

6. Write a main method such as `memFib` that solves the trivial instances of the problem, and solves the nontrivial instances by calling the algorithm of step 4 to initialize the data structure, and then calling the algorithm of step 5 in order to return the result.

Proving the correctness and analyzing the running time of the algorithm described at step 4 will generally be straightforward since this algorithm will often be very simple. Claims resembling the first, second, and third claims above can generally be stated and proved to show correctness of the algorithm described in step 5. The first two claims are likely established through inspection of the code, while the third is likely shown using the first two claims and modifying a proof of correctness of the original ?Divide and Conquer? algorithm.

The *recursion tree method* will often be useful when bounding the running time of the algorithm described at step 5. If this has been implemented as described above, then:

- There will be only one internal node corresponding to each nontrivial instance of the problem that must be formed and solved. These corresponding to the first application of the algorithm on each of these instances.

- There will be only one leaf corresponding to a first attempt to solve each trivial instance of the problem that must be formed and solved.

- The only other nodes will be leaves corresponding to subsequent attempts to solve instances of the problem. The number of these can generally be related to the number of internal nodes. Generally, they can also be bounded by an inspection of the code (which determines the shape of the recursion tree).

§11.4 Dynamic Programing vs Memoization

The process of designing a dynamic programming algorithm arguably requires more work than designing a memoized algorithm, since it is necessary to identify an order in which instances of a problem can be solved. However, it can allow more control, as storage requirements can be reduced by discarding information once it is no longer needed. However, the analysis of a memoized algorithm might be more complicated than that of a dynamic programming algorithm.

§12 October 19, 2017

§12.1 Divide and Conquer - Longest Common Subsequence

The application of dynamic programming and memoization to design efficient algorithms for a problem of interest in computational biology will now be considered. Consider the following quote that introduces the problem.

> In biological applications, we often want to compare the DNA of two (or more) different organisms. A strand of DNA consists of a string of molecules called **bases**, where the possible bases are adenine, guanine, cytosine, and thymine.

Representing each of the bases y their initial letters, a strand of DNA can be expressed as a string over a finite set $\{A, C, G, T\}$. For example, the DNA of one organism may be

$$S_1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA,$$

while the DNA or another organism may be

$$S_2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA.$$

Our goal goal of comparing two stands of DNA is to determine how similar the two strands are, as some measure of how closely related the two organisms are. Similarity can be defined in many different ways. For example, we can say that two DNA strands are similar if one is a substring of the other. In our example, neither S_1 nor S_2 is a substring of the other. Alternatively, we could say that two strands are similar if the number of changes needed to turn one into the other is small. Yet another way to measure the similarity of strands S_1 and S_2 is by finding a third strand S_3 in which the bases in S_3 appear in both S_1 and S_2. These bases must appear in the same order, but not necessarily consecutively. The longer the strand S_3, the more similar S_1 and S_2 are. In our example, the longest strand $S_3 = GTCGTCGGAAGCCGGCCGAA$. This last notion of similarity is formalized as the Longest Common Subsequence Problem.

Here, a *subsequence* is defined as the given sequence with zero or more elements left out. Formally, a sequence $Z = z_1, z_2, ..., z_k$ is a subsequence of a sequence $X = x_1, x_2, ..., x_m$ if there are integers $i_1, i_2, ..., i_k$ such that $1 \leq i_1 < i_2 < ... < i_k \leq m$ and $z_j = x_{i_j}$ for every integer j such that $1 \leq j \leq k$. The problem of finding the longest common subsequence of two sequences can then be stated. The precondition is that a pair of finite sequences X and Y over a finite alphabet Σ are given as input. The postcondition is that a common subsequence Z of X and Y that is at least as long as any other common subsequence is returned.

It is useful to compute the length of a longest common subsequence of a given pair of sequences. Additionally, for a sequence $X = x_1, x_2, ..., x_m$, we denote X_i to be the prefix of X with length $0 \leq i \leq m$ such that $X_i = x_1, x_2, ..., x_i$. It will also be useful to consider a version of a problem in which sequences X and Y are accessed (but not modified) as global data, and the lengths of the prefixes of X and Y currently being considered are given as input.

Example 12.1 (Length of Longest Common Subsequence)

The precondition is that a pair of sequences X and Y over a finite alphabet Σ, each with lengths m and n respectively, are accessed (but not modified) as global data. A pair of integers i and j such that $0 \leq i \leq m$ and $0 \leq j \leq n$ are given as input. The postcondition is that the length $c[i, j]$ of a longest common subsequence of X_i and Y_j is returned as output. That is, the longest common subsequence in the specified prefixes of X and Y are returned.

Solution. [Divide and Conquer] Instances of the problem where $i = 0$ or $j = 0$ are considered trivial since either X_i or Y_j is an empty sequence. In this case, the only common subsequence is the empty sequence, so $c[i, j] = 0$. Suppose now that $X = x_1, x_2, ..., x_m$ and $Y = y_1, y_2, ..., y_n$. Suppose also that $1 \leq i \leq m$ and $1 \leq j \leq n$.

In the case that $x_i = y_j$, then every longest common subsequence of X_i and Y_j must end with x_i, because every other common subsequence of X_i and Y_j can be extended by appending x_i onto the end of it. Furthermore, it is clear that the sequence obtained by appending x_i onto the end of any longest common subsequence of $X_i - 1$ and $Y_j - 1$ is a longest common subsequence of X_i and Y_j. Thus, if $i \geq 1$, $j \geq 1$, and $x_i = y_j$, then

$$c\,[i, j] = c[i - 1, j - 1] + 1.$$

In the case that $x_i \neq y_j$, then any longest common subsequence of X_i and Y_j that ends with x_i cannot end with y_j. So it is also the longest common subsequence of X_i with

Y_{j-1}. Alternatively, any longest common subsequence that does not end with x_i must also be the longest common subsequence of X_{i-1} with Y_j. Therefore, if $i \geq 1$, $j \geq 1$, and $x_i \neq y_j$, then

$$c[i,j] = \max\left(c[i,j-1], c[i-1,j]\right).$$

This allows us to deduce a recurrence relation to obtain $c[i,j]$,

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1,j-1]+1 & \text{if } i \geq 1,\ j \geq 1, \text{ and } x_i = y_j, \\ \max(c[i,j-1], c[i-1,j]) & \text{if } i \geq 1,\ j \geq 1, \text{ and } x_i \neq y_j. \end{cases}$$

The following is the corresponding divide and conquer algorithm based on this recurrence relationship.

```
integer maxSeqLength (integer i, integer j)
{
   if ((i == 0) or (j == 0))
   {
      return 0
   }
   else if (x_i == y_j)j
   {
      return maxSeqLength(i - 1, j - 1) + 1
   }
   else
   {
      return max(maxSeqLength(i, j - 1), maxSeqLength(i - 1, j))
   }
}
```

■

Correctness can be proven by strong induction on $i+j$, where the base cases are when the sum equals 0 and 1. While it is not necessary to bound the running time of this algorithm in order to use it to design a dynamic programming or memoization algorithm, it can be shown that this algorithm requires time that is exponential in $i+j$ in at least the case where $i = j$ and the sequences X and Y have no symbols in common.

§12.2 Dynamic Programming - Longest Common Subsequence

We will now solve the problem by designing a dynamic programming solution. This begins by considering the divide and conquer solution provided previously.

Solution. [Dynamic Programming] When called with inputs i and j, some of the instances including pairs of inputs s and t such that $0 \leq s \leq i$ and $0 \leq t \leq j$ are formed and solved. It is difficult to see which of these are needed and which are not. However, this is not of a huge concern, as there are $(i+1)(j+1)$ such instances. Thus, there are $(m+1)(n+1) \in \Theta(mn)$ of these smaller instances when $i = m$ and $j = n$. Therefore, a two dimensional array of integers C with $i+1$ rows and $j+1$ columns is used to store the solutions for the smaller instances. That is, for $0 \leq s \leq i$ and $0 \leq t \leq j$, the value of $c[s,t]$ will be stored in $C[s,t]$.

We now note that several different solution orders are possible.

- The entries can be filled by non-decreasing order of row. Entries in the same row should be filled by increasing column.

- The entries can be filled by non-decreasing order of column. Entries in the same column should be filled by increasing row.
- The entries can be filled by non-decreasing value of $s + t$, marching down each diagonal.

These options are almost equally efficient. Filling by the first option is likely the simplest, so this is the one for which we design the algorithm for.

```
integer lcsLengthDP(integer i, integer j)
{
   if ((i == 0) or (j == 0)
   {
      return 0
   }
   else
   {
      (integer[])[] C = new (integer[i + 1])[j + 1]

      integer t = 0
      while (t <= j)
      {
         C[0, t] = 0
         t = t + 1
      }

      integer s = 1
      while (s <= i)
      {
         C[s, 0] = 0
         t = 1
         while (t <= j)
         {
            if (xs == yt)
            {
               C[s, t] = C[s - 1, t - 1] + 1
            }
            else
            {
               C[s, t] = max(C[s, t - 1], C[s - 1, t])
            }
            t = t + 1
         }
         s = s + 1
      }
      return C[i, j]
   }
}
```

■

Correctness of this algorithm can only be established after considering the *nested loop* in the algorithm. We will need to make use of the *Third Loop Theorem*. There are two steps when $i = 0$ and $j = 0$. When $i \geq 1$ and $jgeq1$, it can be shown that there are at most $4ij + 9i + 3j + 10 \in \Theta(ij)$ steps. Thus, if this algorithm is being used to compute the length of the longest common subsequence of X and Y, then $i = m$ and $j = n$, so there are at most $O(mn)$ steps used in the worst case.

§12.3 Memoization - Longest Common Subsequence

We will now solve the problem by designing a memoization solution. This begins by considering the divide and conquer solution provided previously. Recall that the first few steps are the same as for designing a dynamic programming solution. Thus, we start by considering the other steps.

Solution. [Memoization] We need a subroutine that initializes the array C when $i \geq 1$ and $j \geq 1$. Suppose that a subroutine `initializeC` does exactly this, setting each entry of the array to `null` using at most $3ij + 4i + 4 \in \Theta(ij)$ steps on inputs i and j. We can now arrive at a recursive algorithm that uses the array C as global data and fills it in, while solving the desired problem. It is assumed that the inputs are integers s and t such that $0 \leq s \leq i$ and $0 \leq t \leq j$.

```
integer recLCSLength(integer s, integer t)
{
   if (C[s, t] != null)
   {
      return C[s, t]
   }
   else if ((s == 0) or (t == 0))
   {
      C[s, t] = 0
      return C[s, t]
   }
   else if (xs = yt)
   {
      C[s, t] = recLCSLength(s - 1, t - 1) + 1
      return C[s, t]
   }
   else
   {
      C[s, t] = max(recLCSLength(s, t - 1), recLCSLength(s - 1, t))
      return C[s, t]
   }
}
```

The main method that uses this subroutine to solve the problem is given below. If called on inputs $i = m$ and $j = n$, then this returns the length of the longest common subsequence of the sequences X and Y.

```
integer memLCSLength(integer i, integer j)
{
   if ((i == 0) or (j == 0))
   {
      return 0
   }
   else
   {
      (integer[])[] C = initializeC[i, j]
      return recLCSLength[i, j]
   }
}
```

■

Example 12.2

Prove the correctness of the `memLCSLength` algorithm. We make the following claims.

1. If s and t are integers such that $0 \leq s \leq i$ and $0 \leq t \leq j$, and the `recLCSLength` algorithm is being executed for the first time as part of an execution of the `memLCSLength` algorithm with inputs i and j such that $1 \leq i \leq m$ and $1 \leq j \leq n$, then $C[s,t]$ is null when this execution of the `recLCSLength` algorithm begins.

2. Suppose that s and t are integers such that $0 \leq s \leq i$ and $0 \leq t \leq j$, where $0 \leq i \leq m$ and $0 \leq j \leq n$, and that the `recLCSLength` algorithm is executed with inputs s and t, when $C[s,t]$ is equal to $c[s,t]$ at the beginning of this execution of the algorithm. Then this execution terminates with $c[s,t]$ returned. Furthermore, $C[s,t]$ is equal to $c[s,t]$ when execution ends.

3. Suppose that s, t, i, and j are integers such that $0 \leq s \leq i \leq m$ and that $0 \leq t \leq j \leq n$. Consider any execution of the `recLCSLength` algorithm with inputs s and t that is part of the execution of `memLCSLength` with i and j. The former execution ends with the value of $c[s,t]$ returned as required, with $C[s,t] = c[s,t]$ stored.

4. the `memLCSLength` algorithm correctly solves the length of longest common subsequence problem.

Solution. We shall prove each of the claims to arrive at a proof of correctness.

1. By inspection of the code, $C[s,t]$ is null when `recLCSLength` is called during an execution of `memLCSLength`. Moreover, the value of $C[s,t]$ is only changed when `recLCSLength` is called with inputs s and t.

2. This follows from inspection of the code.

3. This third part can be proven by strong induction on $s + t$, with the base cases of the sum equal to 0 and 1. Recall that we need to state and prove the subclaim concerning what happens after the first time the `recLCSLength` algorithm is called on s and t. This is left as an exercise to the reader.

4. This follows from inspection of the code, with use of the third part of this proof when $i \geq 1$ and $j \geq 1$.

■

The recursion tree method can once again be used to bound the number of steps used by the recursive subroutine. By inspection of the code that, if a nontrivial instance is being solved for the first time, then either one smaller instance is formed and recursively solved, or two smaller instances are formed and recursively solved. It follows that the recursion tree for an execution of the `recLCSLength` algorithm on inputs i and j such that $1 \leq i \leq m$ and $1 \leq j \leq n$ can be considered a binary tree.

It can be proven by induction on either the size or depth of a binary tree that the number of leaves in the tree is at most one more than the number of internal nodes. The internal nodes correspond to the first executions of the algorithm on various nontrivial instances of this problem, and there are ij of these. Leaves of the recursion tree correspond either to initial solutions of instances of this problem on trivial inputs, or to solutions of the problem for instances that have already been solved. Thus, the number of steps used by the `recLCSLength` algorithm when executed on inputs $1 \leq i \leq m$ and $1 \leq j \leq n$ can be shown to be at most $7ij + i + j + 3$. Now, the `memLCSLength` algorithm uses 2 steps if $i = 0$ or $j = 0$, and uses three steps along with `initializeC` and `recLCSLength`. Thus, this requires $3 + (3ij + 4i + 4) + (7ij + i + j + 3) = 10ij + 5i + j + 10 \in \Theta(ij)$ steps.

§13 October 24, 2017

§13.1

Scheduling and *Selection* problems regularly arise. Consider the following quote that illustrates one instance of this problem.

> Imagine you are a highly-in-demand actor, who has been presented with offers to star in n different movie projects under development. Each offer comes specified with the first and last day of filming, as well as the fee that you will be paid to star in it. To take the job, you must commit to being available throughout this entire period. Thus you cannot simultaneously accept two jobs with intervals that overlap.

This suggests the following computational problem of weighted activity scheduling. We may assume that start times, finish times, and values are all integers. The solution is essentially the same if these can be arbitrary real numbers instead.

Example 13.1 (Weighted Activity Scheduling)
The precondition is that a set I of n intervals on the line, including the integer start time s_i, the finish time f_i such that $s_i < f_i$, and the value $v_i > 0$ for every interval i where $0 \leq i \leq n - 1$ is given as input. The precondition is that a subset of mutually non-overlapping intervals in I whose total is as large as possible is returned.

However, we will solve a simpler problem than this. The intervals that we consider will be represented using an array M with length n. For $0 \leq i \leq n - 1$, M

Solution. ■

§14 October 31, 2017

§14.1 Greedy Algorithms - Minimizing Sum of Completion Times

We now shift our focus to optimization problems. Consider a sequence of tasks $a_1, a_2, ..., a_n$ to perform for some positive integer n. These tasks can be completed in any order. For $1 \leq i \leq n$, the task a_i has a non-negative processing time p_i, where $p_1, p_2, ..., p_n$ are distinct. We define the completion time for task a_i as the sum of the processing times of this task and all other tasks that are ahead of this one.

Example 14.1

Suppose that we have $n = 3$. There are $3! = 6$ orderings that are possible. With the ordering a_2, a_3, a_1, the completion time c_1 of task a_1 would be $p_2 + p_3 + p1$. The completion time c_2 of a_2 would be p_2. Similarly, the completion time c_3 of a_3 is $p_2 + p_3$. Suppose instead that we chose the ordering a_1, a_3, a_2. Then the completion time, c_1 of the task a_1 would be p_1. The completion time c_2 of the task a_2 would be $p_1 + p_3 + p_2$, and the completion time c_3 of task a_3 would be $p_1 + p_3$.

Now, suppose that we are paid more for each task if it is completed sooner rather than later. Thus, we want to keep the sum of completion times $c_1 + c_2 + ... + c_n$ as small as possible. This suggests the following computational problem.

Example 14.2 (Minimizing the Sum of Completion Times)

The precondition is that a sequence $a_1, a_2, ..., a_n$ of tasks and corresponding non-negative distinct integer processing times $p_1, p_2, ..., p_n$ for a positive integer n are given as input. The postcondition is that a re-order $b_1, b_2, ..., b_n$ of the input tasks is returned, such that the sum of the corresponding completion times $c_1 + c_2 + ... + c_n$ is minimized.

There is a simple brute force solution for this problem. One could simply list all possible re-orders for the input tasks and return the one that minimized the corresponding sum of completion times. However, there are $n!$ re-orders that must be considered, so this would be an exponential time algorithm that would be too slow to be useful in practice.

This is an example of an *optimization problem*. An optimization problem is a computational problem with a specific structure. These arise in a variety of applications.

1. The problem always defines a set of **feasible solutions**. These are not all correct solutions (or outputs to be returned) for this instance of the problem, but they are "candidates" that must be considered. For instance, consider a sequence $a_1, a_2, ..., a_n$ (with associated processing times) of this optimization problem. Every one of the $n!$ re-orders of this sequence of tasks is a potentially feasible solution for this instance of the problem.

2. The problem definition includes a **measure function** that maps each feasible solution to a real number. For this problem, the measure function maps every re-order $b_1, b_2, ..., b_n$ to the corresponding sum of completion times $c_1 + c_2 + ... + c_n$.

3. The problem is either a **maximization problem** or a **minimization problem**. In the former, a feasible solution that maximizes the value of the measure function must be returned as output as this measures some form of *reward*. In the latter, a feasible solution that minimizes the value of the measure function must be returned as output as this measures a *cost* or *penalty*. As an example, the problem that we are concerned with is a minimization problem, since we want the smallest total completion time.

A *greedy algorithm* is a kind of algorithm that can sometimes (not always) be used to solve an optimization problem:

- Trivial instances of the problem are solved using some simple brute force method.

- A solution for a nontrivial instance is assembled in stages by repeatedly choosing the option that optimizes (either my maximizing or minimizing) the value of some easily computable local objective function. That is, by making a greedy choice.

Unfortunately, things that look like greedy algorithms are often incorrect because they do not maximize or minimize the value of the objective function for maximization and minimization problems respectively. Some greedy strategies that seem plausible do not come close to this at all. Establishing the correctness of a greedy algorithm is possibly the most challenge part of the design and analysis of this kind of algorithm.

Solution. [Minimizing the Sum of Completion Times] Let n be a positive integer and let $a_1, a_2, ..., a_n$ be a sequence of tasks with processing times $p_1, p_2, ..., p_n$. Then there exists a re-ordering $b_1, b_2, ..., b_n$ of $a_1, a_2, ..., a_n$ so that the sum of completion times $c_1 + c_2 + + c_n$ is less than or equal to to the sum of completion times for any other re-ordering of this sequence of tasks.

The set of all re-orderings of the tasks is both *finite* and *nonempty*. In particular, there are $n!$ such re-orderings. The sum of completion times is a well-defined, total, integer-valued function of these re-orderings. It follows immediately from the above that there must be at least one re-ordering of the tasks that minimizes the value of this function, as claimed.

A trivial instance of the problem occurs when $n = 1$. In this case, we have a single task a_1 with its processing time p_1. Since there is one ordering of a single task, input a_1 is returned. To solve the nontrivial instances, we adopt the following greedy strategy. Notice that if we place a task with a processing time that is larger near the end, then its processing time is included in a fewer number of completion times for this reordering. This suggests that our greedy strategy should be to choose task a_i such that $p_i > p_j$ for all integers j where $1 \leq j \leq n$ and $i \neq j$. In other words, the unique task with maximal processing time a_i should be placed at the end of the output reordering so that $b_n = a_i$.

Suppose that we have the sequence $a_1, a_2, ..., a_n$ with $n \geq 2$. If $b_1, b_2, ..., b_n$ is a re-ordering of the tasks such that the processing time of the final task b_n is **not** larger than all of the other processing times, then there is a different reordering $\hat{b}_1, \hat{b}_2, ..., \hat{b}_n$ with a strictly smaller sum of completion times.

To see this, let $b_1, b_2, ..., b_n$ be a re-ordering with $b_n = a_i$ where $1 \leq i \leq n$ and p_i is not the largest processing time. Then there is some integer j with $1 \leq j \leq n$ and $j \neq i$ such that $p_j > p_i$. In this case, a_j must be b_m for some integer $1 \leq m < n$. Then, consider the new re-ordering where we swap b_m with b_n. Let C be the sum of completion times originally, and $\hat{C}$ be the new sum of completion times. It suffices to show that $\hat{C} - C < 0$ to show that there is a different reordering with a strictly smaller sum of completion times. For $1 \leq h \leq m - 1$, $b_h = \hat{b}_h$, so they have the same completion times. For $m \leq h \leq n$, let the corresponding completion times be d_h and $\hat{d}_h$. Clearly, $d_n = \hat{d}_n$ since they contain the same processing times. On the other hand, since $b_m = \hat{b}_n = a_j$, $\hat{b}_m = b_n = a_i$, and $b_h = \hat{b}_h$ for $m + 1 \leq h \leq n - 1$, it can be checked that

$$\hat{d}_h = d_h + p_i - p_j$$

for $m \leq h \leq n$. That is, since we are swapping a longer processing time to the back, each processing time after and including the swap now add the smaller value instead of the larger one. It follows that

$$\hat{C} - C = (n - m)(p_i - p_j) < 0,$$

since $n > m$ and $p_i < p_j$.

As a consequence of the above proof, there is a re-ordering $b_1, b_2, ..., b_n$ of $a_1, a_2, ..., a_n$ that minimizes its sum of completion times so that the processing time for b_n is the largest processing time of any input task. This follows since there must be a correct solution for this instance of the problem as previously shown, and the solution must be one such that the processing time for b_n is the largest of all processing times as previously shown.

But now, consider when $b_1, b_2, ..., b_{n-1}$ is **not** a correct solution of the smaller problem including tasks $a_1, a_2, ..., a_{i-1}, a_{i+1}, a_{i+2}, ..., a_n$, even though $b_n = a_i$ is the task with the largest processing time. But we know that this can be reduced from rearranging in some way so that the largest processing time is once again brought to the back. We show this by concluding that there is another re-ordering of $a_1, a_2, ..., a_n$ with a strictly smaller completion time.

To show this, we consider a re-ordering $\hat{b}_1, \hat{b}_2, ..., \hat{b}_{n-1}$ with completion time $\hat{C}$ that is the correct solution to the smaller problem, compared to the original subset of the re-ordering $b_1, b_2, ..., b_{n-1}$ with completion time C. Clearly, $\hat{C} < C$. Now, considering $\hat{b}_n = b_n = a_i$, we note that $\hat{b}_1, \hat{b}_2, ..., \hat{b}_n$ is a re-ordering of the original input tasks with a completion time of $\hat{C} + (p_1 + p_2 + ... + p_n)$, while $b_1, b_2, ..., b_n$ is a re-ordering with a completion time of $C + (p_1 + p_2 + ... + p_n)$. The difference of the completion times if $\hat{C} - C$, and since $\hat{C} < C$, this establishes the claim.

Therefore, we can determine a unique re-ordering $b_1, b_2, ..., b_n$ of tasks that minimizes the sum of completion times. More specifically, it has the following structure. If $n \geq 2$, $b_n = a_i$ for the integer i such that $1 \leq i \leq n$ and $p_i > p_j$ for all j where $1 \leq j \leq n$ and $j \neq i$. Additionally, $b_1, b_2, ..., b_{n-1}$ is a unique re-ordering of $a_1, a_2, ..., a_{i-1}, a_{i+1}, a_{i+2}, ..., a_n$ that minimizes the sum of completion times.

It follows from induction on n that this provides a correct solution. In the trivial cases, there is a unique correct solution since there is only one feasible solution. For $n = 2$, we can also treat this as a trivial case, since the removal of the task with the largest processing time produces a trivial instance of the problem. This is important, because it produces a trivial problem, instead of a nontrivial one. Thus, we arrive at the following algorithm.

```
minSCT([a1, a2, ..., an], [p1, p2, ..., pn])
{
   if (n == 1)
   {
      return [a1]
   }
   else
   {
      // Set i so that pi is greater than any other processing time. Then
          proceed with the next line.
      [b1, b2, ..., b(n-1)] = minSCT([a1, a2, ..., a(i-1), a(i+1), a(i+2), ...,
          an], [p1, p2, ..., p(i-1), p(i+1), p(i+2), ..., pn])
      return [b1, b2, ..., b(n-1), ai]
   }
}
```

To prove that the algorithm is correct because it eventually halts, and returns a re-ordering of the tasks that minimizes the sum of completion times, we again prove by induction. A consideration of the trivial instances of the problem by inspection is sufficient for the base case. Then, together with the above reasoning that we can determine a unique re-ordering $b_1, b_2, ..., b_n$ of tasks that minimizes the sum of completion times with the recursive form described previously, we can prove the inductive hypothesis (to be formed) along with inspection of the code. ■

Example 14.3

Determine the efficiency of the `minSCT`algorithm

Solution. For $n \geq 1$, let $T(n)$ be the number of steps executed in the worst case by the algorithm when the preconditions are satisfied. We can show from inspection that the following is a correct recurrence for the algorithm.

$$T(n) \leq \begin{cases} c_0 & \text{if } n = 1, \\ T(n-1) + c_1 n + c_2 & \text{if } n \geq 2. \end{cases}$$

We can then prove by induction that on input $n \geq 1$, we find the following.

$$\begin{aligned} T(n) &\leq c_0 + \sum_{i=2}^{n} (c_1 i + c_2) \\ &\leq c_0 + c_1 \left(\frac{n(n+1)}{2} \right) + c_2(n-1) \\ &\leq \frac{c_1}{2} n^2 + \left(\frac{c_1}{2} + c_2 \right) + (c_0 - c_1 - c_2) \\ &\in \Theta\left(n^2\right). \end{aligned}$$

Therefore, this algorithm uses $O\left(n^2\right)$ steps in the worst case. ■

Note that there is an asymptotically faster algorithm that solves the problem. We can use a sorting algorithm such as `mergeSort` to sort the input tasks by increasing order of processing time using $O(n \log(n))$ steps. Then, we return the sorted sequence of tasks that has been obtained. The cost of computation is dominated by the sorting, so this algorithm correctly solves the problem using asymptotically fewer steps in the worst case.

We now summarize the process the solve optimization problems with unique solutions using a *greedy algorithm.*

1. It was proved that every instance of the optimization problem has at least one solution. Generally, we confirm that the set of feasible solutions is nonempty and finite. Additionally, the measure function is a well defined total function from the set of feasible solutions. It follows immediately that there is at least one feasible solution that either maximizes or minimizes the measure function, so there is a correct solution.

2. Trivial instances are identified and the solutions are described.

3. A greedy strategy that is used to construct the output is described.

4. It is proved that the only correct outputs for a given nontrivial instance of the problem are those that are formed using the above strategy. Because it was shown that there is at least one correct output that exists, it follows that the correct output is formed using this strategy. To do this, we generally consider a feasible solution such that the condition in the greedy strategy is not held. We then argue that we could produce a feasible solution that is better. We essentially do this for one element, then successively apply this reasoning.

5. A process to complete a solution for a nontrivial instance is described and proved correct. This is usually done by forming smaller instances of the problem and recursively solving it. As part of this process, it is shown that there is a unique correct solution for any instance of the problem being solved.

6. A simple algorithm that solved the problem is then given.

7. The correctness of this algorithm is a straightforward consequence of the analysis that had been completed.

8. A simple recurrence is formed and solved in order to bound the number of steps executed by the algorithm. This is usually a worst case analysis as a function of the input size.

9. It is noted that an asymptotically faster algorithm may result from making use of sorting. It may be easy enough to argue that this new algorithm is also correct, since it always returns the same output as the original algorithm.

Remark 14.4. It is important to note that proposed greedy strategies are often incorrect, because they do not optimize the value of the measure function in all cases. One can generally prove the incorrectness of a proposed greedy strategy by giving a specific counterexample. That is, we find a nontrivial instance of the problem such that no correct solution for this instance of the problem can be obtained by following the greedy strategy.

§15 November 2, 2017

§15.1 Greedy Algorithms - Unweighted Activity Selection

We now introduce and apply a process that can be used to design and prove the correctness of a greedy algorithm for a given optimization problem, even when the solution for any instance of the problem is not necessarily unique. To accomplish this, we consider a slightly simpler version of a problem that had previously been considered.

Example 15.1

The precondition is that a sequence $I_0, I_1, ..., I_{n-1}$ of n intervals is given as inputs for a non-negative integer n. The ith interval I_i includes a non-negative start time s_i and a non-negative finish time t_i such that $s_i < t_i$ for $0 \leq i \leq n-1$. The postcondition is that a subset S of the input intervals is returned. Specifically, none of the intervals in S can overlap, so if $0 \leq i \leq j \leq n-1$ and $I_i, I_j \in S$, then either $f_i < s_j$ or $f_j < s_i$. Additionally, S is as large as possible, subject to the previous constraint.

Solution. This can be modeled as a maximization problem. For an instance of the problem, a feasible solution is any subset S of input intervals that do not overlap. Let F be the set of all feasible solutions for this instance of the problem. The measure function (with a value that should be maximized) is the total function $f : F \to N$, such that for every feasible solution $S \in F$, we have $f(S) = \|S\|$.

Claim 15.2 (Existence of Correct Solution)**.** Let n be a non-negative integer. Consider an instance of the Unweighted Activity Selection problem including n intervals $I_0, I_1, ..., I_{n-1}$ as described above. Then at least one correct solution for this instance of this problem exists.

Proof. Consider an instance of this problem, as defined above. Then the empty set of the input intervals is always a feasible solution for this instance of this problem, so the set F of all feasible solutions is nonempty. On the other hand, the set F must be a subset of the set of all subsets A of the input instances. In this case, $\|F\| \leq \|A\| = 2^n$, so F is finite because A is. Since the measure function is a well-defined total function from F to N, there must certainly be some feasible solution $S \in F$ that maximizes the value of this measure function. This feasible solution S is by definition, a correct solution for this instance of the problem, as needed to establish the claim. ■

Let us consider a trivial instance of this problem when $n = 0$ or $n = 1$. If $n = 0$, then there is only one feasible solution, the empty set $\varnothing$. This must certainly be the unique correct solution as well. In the case that $n = 1$, then there are exactly two feasible solutions. These are $S_0 = \varnothing$ and $S_1 = \{I_0\}$. Since $\|S_1\| = 1 > 0 = \|S_0\|$, the unique correct solution for this instance of the problem is S_1.

The set S o f intervals to be returned should be initialized to be the empty set $\varnothing$. There are several strategies that might seem like plausible greedy strategies.

1. Include in S an interval I_i such that the start time s_i is as small as possible.

2. Include in S an interval I_i such that the length $f_i - s_i$ is as small as possible.

3. Include in S an interval I_i such that the finish time f_i is as small as possible.

It can be shown that the first two strategies are incorrect, while the third can be proved to be correct. We will use the third option in our design of a greedy algorithm to solve this problem. The process used in the previous lecture depended heavily on the fact that there was only one correct solution for any given instance of the optimization problem being solved. We already know that this is not the case for the problem now being considered. A different process must now be used. Clearly, the third option may identify more than one interval that might be first picked, since finish times are not necessarily unique. There may even be correct solutions that do not include any of these choices. However, this does not matter, so long as we can prove that there is at least one correct solution that is obtained by using this greedy strategy.

Claim 15.3 (Correctness of Greedy Strategy)**.** Let n be an integer with $n \geq 2$. Consider an instance of the Unweighted Activity Selection problem with n intervals $I_0, I_1, ..., I_{n-1}$. Let i be an integer such that $0 \leq i \leq n-1$ and $f_i \leq f_j$ for every integer j such that $0 \leq j \leq n-1$. Then there exists a correct solution $S \subseteq \{I_0, I_1, ..., I_{n-1}\}$ for this instance of the problem where $I_i \in S$.

Proof. It follows from the proof of the existence of a correct solution that some correct solution $\hat{S} \subseteq \{I_0, I_1, ..., I_{n-1}\}$ of this instance of the problem exists. Either $I_i \in \hat{S}$, or it is not. Consider these two cases separately. In the first case, it suffices to set S to be $\hat{S}$ to establish the claim.

In the case that $I_i \notin \hat{S}$, if a feasible solution S such that $I_i \in S$ and $\|S\| = \|\hat{S}\|$ can be described and proved to have these properties, then it follows that S is a correct solution with $I_i \in S$, establishing the claim. Note that $\hat{S}$ must be nonempty, since the set $S' = \{I_i\}$ would otherwise be a feasible solution such that $\|S'\| = 1 > 0 = \|\hat{S}\|$, contradicting the choice of $\hat{S}$ as a correct solution for this instance of the problem. There must now exist some integer j such that $0 \leq j \leq n-1$ with $I_j \in \hat{S}$ and $f_j \leq f_k$ for every integer k with $0 \leq k \leq n-1$ and $I_k \in \hat{S}$. That is, interval j has the smallest finish time.

Now, consider the set S of $I_0, I_1, \hat{...}, I_{n-1}$ obtained from $\hat{S}$ by removing I_j and include I_i. That is,

$$S = \left(\hat{S} \backslash \{I_j\}\right) \cup \{I_i\}.$$

S clearly includes the interval I_i and $\|S\| = \|\hat{S}\|$. All that is left to prove is that S is a feasible solution for this instance of the problem.

Suppose next that $\|\hat{S}\| = 1$. Then $\hat{S} = \{I_j\}$ and $S = \{I_i\}$. Because $0 \le i \le n-1$, S is a feasible solution for this instance of the problem as required, because there are no other properties to be established. Suppose instead that $\|\hat{S}\| \ge 2$ so that $\|S\| = \|\hat{S}\| \ge 2$ as well. Consider any pair of integers s and t such that $0 \le s, t \le n-1$, $s \ne t$, and $I_s, I_t \in S$. If it can be shown that the intervals I_s and I_t do not overlap, then since s and t were arbitrarily chosen, it follows that no pair of distinct intervals in S overlap. Thus, S is a feasible solution for this instance of the problem as required. To show this, either $i \in \{s, t\}$, or it is not. Consider the cases separately.

In the case that $i \notin \{s, t\}$, then $I_s, I_t \in \hat{S}$, so that these intervals cannot overlap as required. This follows since $\hat{S}$ is a feasible solution for the problem. Alternatively, in the case that $i \in \{s, t\}$, switching s and t as necessary so that $i = s$ and $i \ne t$, we find that $I_t \in \hat{S}$. Now, $t \ne j$, since $I_t \in S$ and $I_j \notin S$. Thus, I_j and I_t are distinct intervals in $hatS$ and since $\hat{S}$ is a feasible solution for this instance, it follows that I_j and I_t cannot overlap. By the choice of j, $f_j \le f_t$ since $I_t \in \hat{S}$. Since $s_j \le t_j$, it certainly cannot be that $f_t < s_j$, so it must be $f_j < s_t$. But since $f_i \le f_j$, it follows that $f_i \le f_j < s_t$ as well, as required to establish that the intervals $I_s = I_i$ and I_t do not overlap. It now follows that S is a feasible solution for this instance of the problem. ■

Consider another instance of the problem with a sequence of intervals

$$\hat{I}_0, \hat{I}_1, ..., \hat{I}_{m-1}$$

for some non-negative integer m that includes all (and only) the intervals in the original sequence

$$I_0, I_1, ..., I_{n-1}$$

that are distinct from I_i (the interval with the smallest finish time) and do not overlap with it. That is, we consider the intervals I_j where $0 \le j \le n-1$ and $j \ne i$ such that $f_i < s_j$. This is a smaller instance of the problem, since the interval I_i is not included. Consider any correct solution for this smaller instance of the problem and let

$$\hat{S} \subseteq \{I_0, I_1, ..., I_{i-1}, I_{i+1}, I_{i+2}, ..., I_{n-1}\}$$

be the set of intervals included in this solution. Let $S = \{I_i\} \cup \hat{S}$.

Claim 15.4 (Feasibility of Solution with Smaller Instances)**.** The set S as described above is a feasible solution for the original instance of the problem.

Proof. S is certainly a subset of $\{I_0, I_1, ..., I_{n-1}\}$, so it remains to confirm that there is no pair of distinct intervals $I_s, I_t \in S$ that overlap. Either $i \in \{s, t\}$, or it is not. Consider these cases separately. In the case that $i \notin \{s, t\}$, the intervals I_s and I_t are distinct intervals that were included in the correct solution for the smaller instance of the problem (including the intervals $\hat{I}_0, \hat{I}_1, ..., \hat{I}_{m-1}$). It follows that I_s and I_t cannot overlap.

In the case that $i \in \{s, t\}$, we consider $i = s$, and $i \ne t$ by switching our choice of name arbitrarily. In this case, I_t was included in the correct solution for the smaller instance of the problem, so $I_t \in \{\hat{I}_0, \hat{I}_1, ..., \hat{I}_{m-1}\}$. It follows by the choice of these intervals that $f_i < s_t$ so that the intervals $I_s = I_i$ and I_t do not overlap. Since s and t were arbitrarily chosen, it now follows that S is a feasible solution for the initial problem as desired. ■

Claim 15.5 (Correctness of Solution with Smaller Instances)**.** The above set S is a correct solution for the original instance of the problem.

Proof. It follows from the proof of the correctness of the greedy strategy that there exists at least one correct solution $S' \subseteq \{I-0, I-1, ..., I_{n-1}\}$ for the original instance of the problem such that $I_i \in S'$. Let $S'' = S' \backslash \{I_i\}$. Consider any integer j with $0 \leq j \leq n-1$ and $I_j \in S''$. Since S is a feasible solution, by the proof of feasibility of the solution with smaller instances, the intervals I_i and I_j do not overlap. It follows by the choice of i that $f_i \leq f_j$ so it is not possible for $f_j < s_i$. It must be true that $f_i < s_j$ so that $I_j \in \{\hat{I}_0, \hat{I}_1, ..., \hat{I}_{m-1}\}$. Because this is true for all $I_j \in S''$, $S'' \subseteq \{\hat{I}_0, \hat{I}_1, ..., \hat{I}_{n-1}\}$. Furthermore, no pair of distinct intervals $I_s, I_t \in S''$ can overlap since they are also distinct intervals in the correct solution S'. It follows that S'' forms a feasible solution for smaller instances of the problem with the intervals $\hat{I}_0, \hat{I}_1, ..., \hat{I}_{m-1}$. Since S' forms a correct solution for this instance of the problem, $\|\hat{S}\| \geq \|S''\|$. Because $S = \hat{S} \cup \{I_i\}$, $S' = S'' \cup \{I_i\}$, $I_i \notin \hat{S}$ and $I_i \notin S''$, the magnitudes of S is greater than or equal to the magnitude of S'. Since S is a feasible solution, S' is a correct solution with a magnitude greater than or equal to that of S. Thus, $\|S\| = \|S'\|$ and since S is a feasible solution, it follows that S is now a correct solution because S' is. ■

To obtain the algorithm, we will assume that we can define a class `Interval`. For each object I of type `Interval`, we can examine the name (the unique string that identifies the interval), the start time, and the finish time. Let us assume that another class `IntervalSeq` is used to represent sequences of intervals. For each object of this type, we will assume that we can examine the length, iterate over the intervals in the sequence, initialize it to the empty sequence, or append another interval to the end of the sequence (increasing the length as well). Additionally, we define a class `IntervalSet` that represents sets of intervals. We can initialize the set to be empty, initialize it to be a fixed set of intervals with size one, or include another interval into the set.

Suppose that a subroutine `greedyChoice` receives the input sequence of intervals as input when $n \geq 1$ and returns an interval in this sequence whose finish time is less than or equal to all others as output. This interval can be discovered by making a single iteration over the intervals in the input sequence, so the cost is at most linear. We also have a subroutine `smallerInstance` that receives an interval and a sequence of intervals as input. It returns a sequence of intervals as output, including only those whose start times are greater than the finish time of I. This can also be discovered using at most a linear number of steps. Thus, we obtain the following algorithm.

```
IntervalSet greedyActivitySelection(IntervalSeq intervals)
{
  // We assume that intervals contains I0, I1, ..., I(n-1)
  integer n = intervals.length
  if (n == 0)
  {
    return the empty set
  }
  else if (n == 1)
  {
    return the set containing I0
  }
  else
  {
    Interval I = greedyChoice(intervals)
    IntervalSeq smaller = smallerInstance(I, intervals)
```

```
        IntervalSet S = greedyActivitySelection(smaller)
        S = S union I
        return S
    }
}
```

The claims that have already been proved make it easy to prove correctness of the above algorithm. This can be proven by induction on n. ■

Example 15.6

Determine the number of steps executed by the `greedyActivitySelection` algorithm when an object of type `IntervalSeq` with a length of at most n is given as input.

Solution. It is clear that $T(n)$ is a non-decreasing function of n. One can see by inspection that there are positive constants c_0, c_1, and c_2 such that

$$T(n) \leq \begin{cases} c_0 & \text{if } n \geq 1, \\ T(n-1) + c_1 n + c_2 & \text{if } n \geq 2. \end{cases}$$

This is the same as the greedy algorithm for minimizing the sum of completion times. Clearly, $T(n) \in O\left(n^2\right)$.

With some additional effort, one can prove that we can find another algorithm that is also correct. The following algorithm can be shown correct since it always returns the same set of output as `greedyActivitySelection`. It follows that there is an algorithm that solves the problem using $O(n \log(n))$ steps in the worst case.

1. Start by sorting the input intervals by non-decreasing finish time so that

   $$f_0 \leq f_1 \leq \ldots \leq f_{n-1}.$$

   Leave the relative order of a pair of intervals I_i and I_j unchanged when they have the same finish times.

2. Initialize the set of intervals to be returned S to $\{I_0\}$, and initialize the integer variable `finish` to be f_0.

3. For each integer i such that $1 \leq i \leq n$, consider the interval I_i. If $s_i >$ `finish`, then we add interval I_i to set S and set finish to f_i.

4. Return the set S.

■

§15.2 Greedy Algorithm Design Process

We generalize the design process from the greedy algorithm design process described in the previous lecture that accounted only those with unique solutions. We following design process can be used to solve any optimization problems using a greedy algorithm.

1. It was proved that every instance of the optimization problem has at least one solution. Generally, we confirm that the set of feasible solutions is nonempty and finite. Additionally, the measure function is a well defined total function from the

set of feasible solutions. It follows immediately that there is at least one feasible solution that either maximizes or minimizes the measure function, so there is a correct solution.

2. Trivial instances are identified and the solutions are described.
3. A greedy strategy that is used to construct the output is described.
4. It is proved that there always exists a correct solution for this instance of the problem that includes whatever has been introduced by the application of the greedy strategy. This was not done by establishing that every correct solution must be like this. Instead, an *exchange argument* was used. It was noted that some correct solution must exist. If this solution includes the greedy choice, then there is nothing more to do. Otherwise, a modification of this correct solution was described and it was proved that the result was still a (different) correct solution for this instance of the problem.
5. A process to complete a solution for a nontrivial instance is described and proved correct. This is usually done by forming smaller instances of the problem and recursively solving it. First, we prove that solution S constructed using the above process is feasible (this is generally a consequence of the construction). We then consider that it had already been proved that there is some correct solution S' that is consistent with the application of the greedy strategy. After establishing a small number of other properties of S', it was possible to compare S with S' and show that they have the same value for the measure function. It immediately follows that S is a correct solution because S' is.
6. A simple algorithm that solved the problem is then given.
7. The correctness of this algorithm is a straightforward consequence of the analysis that had been completed.
8. A simple recurrence is formed and solved in order to bound the number of steps executed by the algorithm. This is usually a worst case analysis as a function of the input size.
9. It is noted that an asymptotically faster algorithm may result from making use of sorting. It may be easy enough to argue that this new algorithm is also correct, since it always returns the same output as the original algorithm.

In the case of a unique solution, it was possible to prove by contradiction, since one could argue that if one returned anything else than what the algorithm gave, the output would be incorrect. This strategy fails when correct solutions are not guaranteed unique. Instead, we must use an *exchange argument* to establish correctness. This involves the comparison of the results returned by the algorithm with those of a hypothetically correct solution.

§16 November 7, 2017

§16.1 Greedy Algorithms - Data Compression and Huffman Trees

We will now proceed to analyze a greedy algorithm that solves a problem concerning data compression. The structure of the greedy algorithm used includes an additional initialization phase. The process used to complete the greedy strategy once one has been found is also slightly different from previous examples.

Suppose we wist to transmit an encoded version of a text file, encoded as a sequence of 0's and 1's over a reliable channel. For instance,

MADAM I'M ADAM

We know the frequency, which is the number of occurrences of each symbol. In the above string, "M" and "A" each have a frequency of four, "D" and the blank (from now on referred to as "_") each have a frequency of two, and both "I" and "'" have a frequency of one. No other symbols appear in this string.

Let Σ be the set of symbols included in the message being sent. This problem is trivial and uninteresting if $\|\Sigma\| = 1$, so it will be assumed that $\|\Sigma\| \geq 2$. Note that in our example, we have $\Sigma = \{M, A, D, \_, I, '\}$. Let $\varphi : \Sigma \to \{0,1\}^*$ be a total function such that $\varphi(\sigma) \neq \lambda$, where λ denotes the empty string, for every symbol $\sigma \in \Sigma$. This can be extended to define a mapping $\varphi : \Sigma^* \to \{0,1\}^*$ such that for every non-negative integer m, and every string

$$\omega = \alpha_1\alpha_2 ... \alpha_m$$

with length m, one would set

$$\varphi(\omega) = \varphi(\alpha_1)\varphi(\alpha_2)...\varphi(\alpha_m),$$

so that $\varphi(\omega)$ is the concatenation of the encodings of the symbols in ω. this kind of mapping is only useful if it is injective. That is, for all strings $\omega_1, \omega_2 \in \Sigma^*$, if $\omega_1 \neq \omega_2$, then $\varphi(\omega_1) \neq \varphi(\omega_2)$.

One way to achieve this is to use a **fixed length code**. For instance, set $l = \lceil \log_2 (\|\Sigma\|) \rceil$ and map each symbol $\sigma \in \Sigma$ to a string $\varphi(\sigma) \in \Sigma^*$ with length l in such a way so that the mapping function is injective. In our example, we find that our alphabet Σ has a size of 6. Thus, $l = \lceil \log_2 (\|\Sigma\|) \rceil = 3$. We could therefore define the function as follows.

$$\varphi(M) = 000 \quad \varphi(A) = 001 \quad \varphi(D) = 010$$

$$\varphi(\_) = 011 \quad \varphi(I) = 100 \quad \varphi(\text{'}) = 101$$

Then, we find that we can just concatenate the individual results to find the following.

$$\varphi(\text{MADAM I'M ADAM}) = 000001010001000011100101000011001010001000.$$

This is a string comprised of 0 and 1 (meaning the string is in $\{0,1\}^*$) with a length of 42. If the string $\omega \in \Sigma^*$ has a length m, then this always yields an encoding $\varphi(\omega)$ with a length of $ml = m \lceil \log_2 (\|\Sigma\|) \rceil$.

Another way to encode the symbols in an injective mapping is to use a **prefix code**. This is a mapping $\varphi : \Sigma \to \{0,1\}^*$ such that if $\alpha_1, \alpha_2 \in \Sigma$ and $\alpha_1 \neq \alpha_2$, then $\varphi(\alpha_1)$ is not a prefix of $\varphi(\alpha_2)$. We could therefore define the function as follows.

$$\varphi(M) = 10 \quad \varphi(A) = 01 \quad \varphi(D) = 11$$

$$\varphi(\_) = 001 \quad \varphi(I) = 0000 \quad \varphi(\text{'}) = 0001$$

Indeed it can be confirmed that no string is a prefix of another, so this is a prefix code. Then, we find that we can just concatenate the individual results to find the following.

$$\varphi(\text{MADAM I'M ADAM}) = 1001110110001000000011000101110110.$$

Note that this is actually a shorter string than any encoding that can be obtained by using a fixed length code. Clearly, a length of 34 is shorter than a length of 42.

Now, since $\varphi(\alpha_1)$ is not a prefix of $\varphi(\alpha_2)$ when φ is a prefix code and $\alpha_1, \alpha_2 \in \Sigma$ such that $\alpha_1 \neq \alpha_2$, then every prefix code can be represented using a binary tree. That is, we could obtain a binary code corresponding to the above prefix code.

- The elements of Σ label the leaves, with each symbol in Σ being used as the label for exactly one leaf.
- The edges from a node to its left child are labelled with 0, while edges from a node to its right child are labelled with 1.
- For all $\alpha \in \Sigma, \varphi(\alpha)$ is a string of 0's and 1's that label the edges from the root down to the leaf labelled with α.

If our goal is to minimize the length $\|\varphi(\omega)\|$ of an encoding of a string ω, then it suffices to consider prefix codes whose corresponding binary trees are full binary trees such that every internal node has two children.

Definition 16.1. A prefix code φ with this property that $\|\varphi(\omega)\|$ is as small as possible is called a *Huffman code* for ω. Its corresponding binary tree is known as the *Huffman tree* for ω.

Example 16.2 (Huffman Tree)

The precondition is that a finite alphabet Σ and the function $f : \Sigma \to \mathbb{N}$ such that $f(\alpha)$ is the number of occurrences of α in a string ω for all $\alpha \in \Sigma$ are given as input. The output is a Huffman tree for ω.

Solution. To model this as a minimization problem, we first define the set F of feasible solutions to be the set of all full binary trees, with edges labelled by 0's and 1's, and with leaves labelled by symbols in Σ as described previously. The measure function is the function $g : F \to \mathbb{N}$ so that for any tree $t \in F$,

$$g(t) = \sum_{\alpha \in \Sigma} f(\alpha) \cdot \|\varphi(\alpha)\|.$$

The measure function is the length of the encoding of ω using the prefix code φ corresponding to tree t.

Now, we will show that a correct solution always exists. To do so, we will make use of the following claim.

Claim 16.3. Let $X(1) = 1$ and let $X(n)$ be the number of full binary trees with n leaves, labelled with the distinct values of $\alpha_1, \alpha_2, ..., \alpha_n$ for any integer $n \geq 2$. Then $X(n) < 2^{(n^2)}$ for every positive integer n

Proof. This can be proved by strong induction on n. The base case of $n = 1$, then by definition $X(1) = 1 < 2^{1^2}$. In the base case of $n = 2$, we note that there are exactly two full binary trees where one has α_1 on the left, while the other has α_2 on the left instead. Thus, $X(2) = 2 < 2^{2^2}$.

In the inductive case, let k be an integer such that $k \geq 2$. Additionally suppose that $X(i) < 2^{i^2}$ for every integer i such that $1 \leq i \leq k$. We will now show that $X(k+1) < 2^{(k+1)^2}$. Thus, consider the full trees whose labels are $\alpha_1, \alpha_2, ..., \alpha_{k+1}$. The number of leaves in the left subtree of any such tree must total some integer j such that $1 \leq j \leq k$. For any choice of j, there are $\binom{k+1}{j}$ choices for labels of leaves in the left subtree. For any such choice, there are $X(j)$ possible left subtrees and $X(k+1-j)$ possible right subtrees. Thus, we conclude the following after nothing the inductive hypothesis holds for $1 \leq j, k+1-j \leq k$.

$$\begin{aligned} X(k+1) &= \sum_{j=1}^{k} \binom{k+1}{j} X(j)X(k+1-j) \\ &< \sum_{j=1}^{k} \binom{k+1}{j} 2^{i^2} 2^{(k+1-j)^2} \\ &< \sum_{j=1}^{k} \binom{k+1}{j} 2^{(k+1)^2-2(k+1)j+2j^2} \end{aligned}$$

In the above, let $h(j) = (k+1)^2 - 2(k+1)j + 2j^2$. It can be checked that $h(j) = h(k) = k^2+1$, $h'(j) = -2(k+1)+4j$, and $h''(j) = 4$. Thus, h has a local minimum when $i = (k+1)/2$ and has the same maximal value of k^2+1 at both endpoints of the range for this summation. Thus, $h(j) \leq k^2+1$ for every integer j such that $1 \leq j \leq k$. The proof continues as follows.

$$\begin{aligned} X(k+1) &< \sum_{j=1}^{k} \binom{k+1}{j} 2^{(k+1)^2-2(k+1)j+2j^2} \\ &< \sum_{j=1}^{k} \binom{k+1}{j} 2^{k^2+1} \\ &< 2^{k^2+1} \sum_{j=0}^{k+1} \binom{k+1}{j} = 2^{k^2+1}2^{k+1} < 2^{(k+1)^2} \end{aligned}$$

Because $k \geq 2$, this is as required to complete the inductive step and establish the claim. ■

The set F of all feasible solutions for any instance of the problem is therefore a finite nonempty set. As noted, it is $\|F\| \leq 2^{(n^2)}$. Since the measure function for this minimization problem is a well-defined total function from F to $\mathbb{N}$, it follows that there is a solution of this problem for every instance of it.

Next, we will identify and solve trivial instances of the problem. We will consider an instance to be trivial when $\|\Sigma\| = 2$ so that $\Sigma = \{\alpha_1, \alpha_2\}$ for a distinct pair of symbols α_1 and α_2. There are only two feasible solutions. The first is when α_1 is on the left and α_2 is on the right, and the second is when these positions are swapped. It is easily checked that if φ_1 is the prefix code corresponding to the first of these trees and φ_2 is the prefix code corresponding to the second tree, then these are both correct solutions for this instance of the problem because the following holds.

$$\sum_{k=1}^{n} \|\varphi_1(\alpha_i)\| f(k) = \sum_{k=1}^{n} \|\varphi_2(\alpha_i)\| f(k) = \sum_{k=1}^{n} f(k) = \|\omega\|.$$

Instead, we now suppose that $n = \|\Sigma\| \geq 3$. Now, we will consider the greedy strategy used to solve nontrivial instances of the problem. Let i and j be integers such that $1 \leq i, j \leq n$, where $i \neq j$. We require that α_i and α_j have frequencies that are at least as low as those of all other symbols in Σ. That is, if $1 \leq k \leq n$ and $k \notin \{i, j\}$, then $f(\alpha_k) \geq f(\alpha_i)$ and $f(\alpha_k) \geq f(\alpha_j)$. Our greedy strategy will be to decide that the leaves labelled α_i and α_j have the same parent. Te following is out proof of correctness of our greedy strategy.

Claim 16.4. There exists a solution for this instance of the problem (that is, a Huffman tree) such that the leaves labelled by α_i and α_j are siblings. That is, they have the same parent.

Proof. Let t be a solution for this instance of the problem. It follows by the first claim that at least one solution exists. If the leaves labelled by α_i and α_j are siblings in t, then the claim follows. Suppose instead that the leaves labelled by α_i and α_j are not siblings in t. Switching α_i and α_j if necessary, we may assume that the distance l_i from the root down the the leaf with label α_i is greater than or equal to the distance l_j from the root down the leaf with label α_j in t.

Let $\hat{t}$ be the subtree of t whose root is the sibling of α_i. Now, consider switching this subtree $\hat{t}$ with α_j to obtain the new tree t'. It can be argued that t' is indeed a feasible solution for the problem. Now, let $S \subseteq \Sigma$ be the set of labels of leaves in $\hat{t}$. Since t and t' are both full trees, S is nonempty. Because α_i and α_j both are not in S, then S must include at least one element $\alpha_k \in \Sigma$ as defined previously. For this choice of k, we find the following.

$$\sum_{\alpha \in S} f(\alpha) \geq f(\alpha_k) \geq f(\alpha_j).$$

Let φ be the prefix code of tree t, and φ' be the prefix code of t'.

$$\begin{aligned}\sum_{h=1}^{n} \|\varphi'(\alpha_n)\| f(\alpha_n) - \sum_{h=1}^{n} \|\varphi(\alpha_n)\| f(\alpha_n) &= \sum_{\alpha \in S} (l_j - l_i) f(\alpha) + (l_i - l_j) f(\alpha_j) \\ &= (l_j - l_i) \left(\sum_{\alpha \in S} f(\alpha) - f(\alpha_j) \right)\end{aligned}$$

But the value of the last line above is less than zero, since $l_j \leq l_i$ and $\sum_{\alpha \in S} f(\alpha) \geq f(\alpha_j)$. It follows that $\sum_{h=1}^{n} \|\varphi'(\alpha_n)\| f(\alpha_n) \leq \sum_{h=1}^{n} \|\varphi(\alpha_n)\| f(\alpha_n)$. Since t' is a feasible solution and t is a correct solution, then $\sum_{h=1}^{n} \|\varphi(\alpha_n)\| f(\alpha_n) \leq \sum_{h=1}^{n} \|\varphi'(\alpha_n)\| f(\alpha_n)$. This means that they were in fact equal, so t' is a correct solution because t is. Since the leaves α_i and α_j are siblings in t', this establishes the claim as required. ■

Now that we have shown the correctness of the greedy strategy, we shall now describe the process that permits us to continue. Suppose that we remove α_j from Σ to obtain a smaller alphabet $\hat{\Sigma} = \Sigma \backslash \{\alpha_j\}$ and produce a string $\hat{\omega} \in \hat{\Sigma}^*$ from ω by replacing every copy of $\alpha_j \in \omega$ with a copy of α_i. Using our example, if we chose α_i to be I and α_j to be ', then we would obtain the following.

$$\hat{\Sigma} = \{M, A, D, \_, I\}$$

$$\hat{\omega} = \text{MADAM IIM ADAM}$$

The frequencies of letters have changed. Letting $f(\alpha)$ denote the number of occurrences of α in ω and $\hat{f}(\alpha)$ denote the number of occurrences of α in $\hat{\omega}$, we find the following.

$$\hat{f}(\alpha) = \begin{cases} f(\alpha) & \text{if } \alpha \in \hat{\Sigma} \text{ and } \alpha \neq \alpha_i, \\ f(\alpha_i) + f(\alpha_j) & \text{if } \alpha = \alpha_i. \end{cases}$$

Because the size of the new alphabet is one less than the original alphabet, this is a smaller instance of the problem. Let $\hat{t}$ be a correct solution for this smaller instance of the problem. That is, $\hat{t}$ is a Huffman tree for $\hat{\omega}$. We then continue by replacing the leaf labeled α_i with a subtree. This subtree has 0 on the left edge to the leaf α_i and has 1 on the right edge to the leaf α_j. This gives us our desired tree t. It can be argued that this produces a feasible solution for the original problem. We will now show correction of this method of continuation.

Claim 16.5. Let $\hat{\omega}$ be the string obtained from ω by replacing copies of α_j with copies of α_i as previously described. Let $\hat{t}$ be any Huffman tree for $\hat{\omega}$ and suppose that t is obtained from $\hat{t}$ as previously described. Then t is a Huffman tree for ω.

Proof. Recall from the second claim above that there is a Huffman tree t' for ω such that the leaves labelled α_i and α_j are siblings in t'. Switching the roles as necessary, we may assume that a subtree of t' is the subtree with the left edge labelled 0 leading to leaf α_i and the right edge labelled 1 leading to leaf α_j. Let $\hat{t}'$ be the tree obtained from t' by replacing the above subtree with a leaf labelled α_i. The following can be easily verified.

- $\hat{t}'$ is a feasible solution for the problem instance with string $\hat{\omega}$. Thus, if $\hat{\varphi}$ is the prefix code for $\hat{\omega}$ corresponding to tree $\hat{t}$, and $\hat{\varphi}'$ is the prefix code for $\hat{\omega}$ corresponding to $\hat{t}'$, then we find the following since $\hat{t}$ is a Huffman tree for $\hat{\omega}$.

$$\sum_{\alpha\in\hat{\Sigma}} \|\hat{\varphi}(\alpha)\|\hat{f}(\alpha) \leq \sum_{\alpha\in\hat{\Sigma}} \|\hat{\varphi}'(\alpha)\|\hat{f}(\alpha).$$

- If φ is the prefix code for ω corresponding to t, then since $\|\varphi(\alpha_i)\| = \|\varphi(\alpha_j)\| = \|\varphi'(\alpha_i)\| + 1$, we find the following.

$$\sum_{\alpha\in\Sigma} \|\varphi(\alpha)\|f(\alpha) = \left(\sum_{\alpha\in\hat{\Sigma}} \|\hat{\varphi}(\alpha)\|\hat{f}(\alpha)\right) + f(\alpha_i) + f(\alpha_j).$$

- If φ' is the prefix code for ω corresponding to t', then through similar reasoning, we find the following.

$$\sum_{\alpha\in\Sigma} \|\varphi'(\alpha)\|f(\alpha) = \left(\sum_{\alpha\in\hat{\Sigma}} \|\hat{\varphi}'(\alpha)\|\hat{f}(\alpha)\right) + f(\alpha_i) + f(\alpha_j).$$

The following now follows from application of the second, first, and third properties above.

$$\begin{aligned}
\sum_{\alpha\in\Sigma} \|\varphi(\alpha)\|f(\alpha) &= \left(\sum_{\alpha\in\hat{\Sigma}} \|\hat{\varphi}(\alpha)\|\hat{f}(\alpha)\right) + f(\alpha_i) + f(\alpha_j), \\
&\leq \left(\sum_{\alpha\in\hat{\Sigma}} \|\hat{\varphi}'(\alpha)\|\hat{f}(\alpha)\right) + f(\alpha_i) + f(\alpha_j), \\
&\leq \sum_{\alpha\in\Sigma} \|\varphi'(\alpha)\|f(\alpha).
\end{aligned}$$

On the other hand, since t' is a Huffman tree of ω and t is a feasible solution for the instance of the problem including ω, then

$$\sum_{\alpha\in\Sigma} \|\varphi'(\alpha)\| f(\alpha) \le \sum_{\alpha\in\Sigma} \|\varphi(\alpha)\| f(\alpha).$$

Thus, these two expressions are in fact equal and it follows that t is a Huffman tree for ω because t' is. This is as required to establish the claim. ■

To complete a greedy algorithm for this problem, we first initialize data structures in order to make this process more efficient. Each can be used as global data by subroutines used by the main method for the Huffman tree problem.

- An array T with length n will be used to represent information about the Huffman tree being constructed. For $0 \le i \le n-1$, $T[i] = \mathit{null}$ if the symbol α_{i+1} has not yet been included in the Huffman tree being constructed. Otherwise, $T[i]$ is a reference to the leaf labelled by α_{i+1} in this Huffman tree. This can certainly be initialized by setting $T[i] = \mathit{null}$ for $0 \le i \le n-1$ using $O(n)$ steps.
- A binary heap H (a minheap) will be used to store elements of the alphabet Σ currently being considered, using their frequencies as weights. This can be initialized using $O(n \log_2(n))$ steps.

A recursive algorithm can then be used to implement the greedy strategy continuation method previously described above.

- The elements α_i and α_j described in the proposed greedy strategy can be found using a logarithmic number of steps using the `EXTRACTMIN` operation of the minheap H. The `INSERT` operation can then be to reinsert α_i using its updated frequency.
- Following the solution of the smaller instance of the problem that has been described, the array T can also be updated using at most a constant number of steps.
- Using data structures as described above, it can be argued that there exists a positive integer c such that the number of steps $T(n)$ used by this algorithm in the worst case satisfies the recurrence

$$T(n) \le \begin{cases} c & \text{if } n = 2, \\ T(n-1) + c\lceil \log_2(n) \rceil & \text{if } n \ge 3. \end{cases}$$

- This can be used to show that this algorithm computes a Huffman tree for a string $\omega \in \Sigma^*$ using $O(n \log n)$ steps in the worst case when $\|\Sigma\| = n \ge 2$.

■

§17 November 9, 2017

§17.1 Greedy Algorithms - Offline Caching

The greedy algorithm to be considered solves a simplified problem motivated by a more complex problem related to memory management. This algorithm is commonly known as Belady's Algorithm, or the Clairvoyant Replacement Algorithm. It uses a greedy strategy that is sometimes known as the *longest forward distance.*

Modern computers use a cache to store a small amount of data in fast memory. Even though main memory may be much larger (and much slower), the use of a cache may significantly decrease processing time. Unfortunately, processing is slowed down again if information is not in the cache when needed, as it has to be copied from main memory into the cache, and something else must be removed from the cache to make room for it. In practice it is necessary to decide what must be removed from the cache as soon as something else must be brought in. Of course, we do not have any information about future memory requests. Algorithms that deal with this kind of problem are called online algorithms. The problem hence described is *online hashing.*

Another less realistic problem is one in which the entire sequence of memory requests is made available before it is necessary to figure out how to serve the first (or any later) request. This version of the problem is called *offline caching.* To model this more formally, consider a problem whose instances include the following information.

1. A positive integer M denoting the size of main memory.
2. A positive integer k such that $k < M$ that indicates the size of the cache.
3. A set $C \subseteq \{0, 1, ..., M-1\}$ such that $\|C\| = k$. These are the indices in main memory of the initial contents of the cache.
4. A sequence $(r_1, r_2, ..., r_m)$ of elements of $\{0, 1, ..., M-1\}$ indicating the addresses of pages in main memory that will be requested.

Given the above information, one can define a valid sequence of changes $(\alpha_1, \alpha_2, ..., \alpha_n)$ with corresponding caches $(C_0, C_1, ..., C_n)$ to be any pair of sequences that satisfy the following:

1. $C_0 = C$.
2. For $1 \leq i \leq n$, if $r_i \in C_{i-1}$, then $\alpha_i = null$ and $C_i = C_{i-1}$. We say that there has been a **cache hit** in this case.
3. For $1 \leq i \leq n$, if $r_i \notin C_{i-1}$, then $\alpha_i \in C_{i-1}$ and $C_i = (C_{i-1} \backslash \{\alpha_i\}) \cup \{r_i\}$. Thus α_i is the index of the page being removed from the cache in order to include r_i . We say that there has been a **cache miss** in this case.

Example 17.1 (Offline Caching)

The precondition is that positive integers M and k, a set C, and a sequence $r_1, r_2, ..., r_n)$ as described above are given as input. The postcondition is that a valid sequence $(\alpha_1, \alpha_2, ..., \alpha_n)$ such that the number of cache misses is as small as possible is returned as output.

Solution.

Claim 17.2. Every instance of the Offline Caching problem has a solution.

Proof. Consider an instance of the problem as described above. Then the set F of all feasible solutions is the set of all valid sequences of changes, and it is easily proved by induction on n that $1 \leq \|F\| \leq k^n$, so that this set is both nonempty and finite. The number of caches misses is a well-defined and total function from F to $\mathbb{N}$. The claim now follows. ∎

We now proceed to identify and solve trivial instances. We consider an instance of this problem to be trivial when $n = 1$. If $r_1 \in C$, then $(null)$ is returned since it is the only feasible solution and there are no cache misses at all. If $r_1 \notin C$, then α_1 can be set to be any element of the cache C. Thus, (α_1) is a feasible solution and since r_1 is the final request, the number of cache misses will be one, regardless of the choice that is made.

Next, we consider a greedy strategy for nontrivial instances. Consider an instance of the problem when $n \geq 2$. Now if $r_1 \in C$, then once again *null* must be chosen as the first entry of the sequence of changes to be returned. This must be the greedy choice in this case. Otherwise, if $r_1 \notin C$ and there exists at least one element α of C such that $\alpha \notin \{r_2, r_3, ...r_n\}$ then one can set α_1 to be α because this will never be needed again. If there is more than one such value then it does not matter which one is picked. ■

§18 November 14, 2017

§18.1 Computational Problems and Languages

So far, we have considered computational problems specified by pairs of preconditions and postconditions with fairly elaborate structure. For instance, recall that an integer $n \geq 2$ is prime if the only positive integers k such that $1 \leq k \leq n$ such that n is divisible by k are 1 and n. An integer $n \geq 2$ is composite if n is not prime. An interesting computational problem that involves primes can be specified by the precondition that an integer n such that $n \geq 2$ is given as input, and the postcondition that the smallest positive prime p such that n is divisible by p is returned as output. We call this the Smallest Prime Factor problem.

A *decision problem* is a computational problem with a `true/false` answer. It can be specified more simply by describing an *instance* of the problem and the *question* being answered.

Example 18.1 (Prime Factor in Range)

The instance of this problem is a pair of integers (n, k) such that $2 \leq k \leq n$. The question is whether there exists a positive prime p such that $2 \leq p \leq k$ and n is divisible by p.

Decision problems will be considered for the rest of the course. These are simpler in structure and arguably a bit easier to study. On the other hand, the difficulty of more complex problems can often be related to the difficulty of corresponding decision problems. For example, suppose we have an algorithm `primeInRange` that solves the Prime Factor in Range problem. Then this could be used as a subroutine to produce an algorithm like the following one, which solves the Smallest Prime Factor problem using a variation of Binary Search.

```
integer smallestFactor(integer n)
{
   integer low = 2
   integer high = n
   while (low < high)
   {
      integer mid = floor((low + high)/2)
      if (primeInRange(n, mid))
      {
         high = mid
```

```
        }
        else
        {
            low = mid + 1
        }
    }
    return low
}
```

So far in this course, we have been studying computational problems whose instances and outputs include elements of a variety of (sometimes complex) classes or types. Even after moving from a more complex search problem, to a simpler decision problem, we are still dealing with an ordered pair of positive integers with restrictions on their values as an instance. Problems whose instances can include one or more elements of virtually any type or class are generally called *abstract decision problems*. On the other hand, one might also consider decision problems whose instances are strings in Σ for some finite but nonempty alphabet Σ. These are called *concrete decision problems*.

An *encoding* of a set of instances of an abstract decision problem is what is needed to relate an abstract decision problem to a concrete decision problem. Let I be the set of instances of an abstract decision problem. For the Prime Factor in Range problem, I is the subset of $\mathbb{N} \times \mathbb{N}$ including all ordered pairs (n, k) of integers such that $2 \leq k \leq n$. Let Σ be an alphabet. For this problem, it is advantageous to choose $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ;\}$. Recall that if S is a set, then $P(S)$ denotes the power set of S, which is the set of all subsets of S.

Definition 18.2. An **encoding** of a set I of instances of an abstract decision problem using an alphabet Σ is the total function

$$e : I \to P\left(\Sigma^*\right),$$

satisfying the following additional properties:

1. For all $\alpha \in I$, the set $e(\alpha)$ is nonempty so that every instance α is encoded by at least one string in Σ^*.

2. For all $\alpha, \beta \in I$ such that $\alpha \neq \beta$, then $e(\alpha) \cap e(\beta) = \{\}$. That is, no string in Σ^* encodes more than one instance in I.

Continuing our example, for every integer $n \in \mathbb{N}$ such that $n \geq 2$, let $e(n) \in \Sigma^*$ be the string of symbols giving the unpadded decimal representation. For instance, $e(4376) = 4376$, which is the string in Σ^* of length four. Note that for $n \geq 2$, the length of $e(n)$ is linear in $\log_2(n)$, not in n. That is, notice how the value 4376 is encoded using only 4 characters. For an instance $\alpha = (n, k) \in I$ of the Prime Factor in Range problem, we shall let the following be the encoding function.

$$e(\alpha) = \{e(n); e(k)\}$$

We have slightly abused notation, but this is simply the set that includes the single string of $e(n); e(k) \in \Sigma^*$.

An abstract decision problem A with a set I of instances and an encoding $e : I \to \Sigma^*$ for some alphabet Σ can be used to define the following languages.

1. The **Language of Instances** is the set of all strings in Σ^* that are encodings of instances in I,

$$L_{A,I} = \bigcup_{\alpha \in I} e(\alpha).$$

2. The **Language of Yes-Instances** is the set $L_{A,Yes}$ of all strings in Σ^* that are encodings of Yes-instances of the decision problem A.

3. The **Language of No-Instances** is the set $L_{A,No}$ of all strings in Σ^* that are encodings of No-instances of the decision problem A.

Remark 18.3. It follows from these definitions that $L_{A,Yes} \cup L_{A,No} = L_{A,I}$ and $L_{A,Yes} \cap L_{A,No} = \{\}$. Also, empty set characters look ugly, so I am just typing $\{\}$ for empty set.

Example 18.4
We can now easily understand the the corresponding concrete decision problem of the Prime Factor in Range problem. The instance is the string $\omega \in \Sigma^*$, and the question is whether $\omega \in L_{A,Yes}$. That is, does ω encode a Yes-instance of A.

§18.2 Turing Machines

Definition 18.5. A **deterministic Turing machine** is a 7-tuple,

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject}),$$

where

1. Q is a finite and nonempty set of states.

2. Σ is the finite and nonempty input alphabet. Σ does not include the blank symbol $\_$.

3. Γ is the finite and nonempty tape alphabet such that $\Sigma \subseteq \Gamma$, $\_ \in \Gamma$, and $Q \cap \Gamma = \{\}$. Γ may also include a finite number of additional symbols that are not in $\Sigma \cup \{\_\}$.

4. δ denotes the transition function that is a partial function
$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}.$$
The transition $\delta(q, \sigma)$ should be defined for every state $q \in Q \backslash \{q_{accept}, q_{reject}\}$ and for every symbol $\sigma \in \Gamma$. Neither $\delta(q_{accept}, \sigma)$ nor $\delta(q_{reject}, \sigma)$ should be defined for any symbol $\sigma \in \Gamma$.

5. $q_0 \in Q$ is the start state.

6. $q_{accept} \in Q$ is the accept state.

7. $q_{reject} \in Q \backslash \{q_{accept}\}$ is the reject state. This simply means that the reject state cannot be the same as the accept state.

If this is still here when you are reading it, it means I have not yet added the review section on Turing machines.

§18.3 Complexity Class P

Consider a Turing machine M with input alphabet Σ that decides a language. that is, it halts when executed on all strings in Σ^*. The time used by M on input $\omega \in \Sigma^*$ is the number of steps that M takes on input ω before it halts. The worst case running time of M is a function $T_M : \mathbb{N} \to \mathbb{N}$. For $n \geq 0$, $T_M(n)$ is the maximum time used by M on any input $\omega \in \Sigma^*$ with length $\|\omega\| = n$.

Now, let $f : \mathbb{N} \to \mathbb{N}$ be an arbitrary function. Then $TIME(f)$ or $TIME(f(n))$ is the set of languages L such that L is decided by a Turing machine, with a worst case running time of $O(f)$. The complexity class P can now be defined as

$$P = \bigcup_{k \in \mathbb{N}} TIME\left(n^k\right).$$

for the remainder of this course, we will consider abstract decision problems A and encodings, whose language $L_{A,I}$ of instances are in P. We say that the corresponding concrete decision problem is decidable deterministically in polynomial time if the language $L_{A,Yes}$ of Yes-instances is in P as well.

§18.4 Cobham-Edmonds Thesis

Recall the *Church-Turing thesis*, which states that a problem is solvable using an algorithm if and only if it can be solved using a deterministic Turing machine. On the other hand, the *Cobham-Edmonds thesis* claims that an abstract decision problem is efficiently solvable using an algorithm if and only if there is a reasonable or natural encoding for the instances such that the corresponding decision problem can can solved deterministically in polynomial time. In essence, it asserts that if abstract decision problems have reasonable or natural encodings such that, when one of these encodings is used, then the corresponding language of Yes-instances is in the complexity class P.

In general, it can be said that both the Church-Turing thesis and the Cobham-Edmonds thesis are ill-posed. Various models of computation that were being compared during the 1930?s that led to the formulation of the Church-Turing thesis did not all perform computations on strings of symbols in an alphabet. It is necessary to introduce a translation or encoding from one type of object to another, in order to compare these models of computation. No attempt has been made so far to say which encodings of the set of instances of a given abstract decision problem are reasonable or natural, but these terms are included in the version of the Cobham-Edmonds thesis that is being presented here. Evidence in support of this thesis includes proofs that other proposed formal definitions of efficient computation, involving different models of computation, are equivalent to the definition involving polynomial-time computation by deterministic Turing machines. The next result described in this lecture can be viewed as evidence of this thesis.

For any fixed integer k such that $k \geq 1$, a **k-tape Turing machine** is a generalization of a Turing machine that has k tapes, all with tape heads that can move independently. For example, if $q \in Q$ and $\Gamma = \{a, b, c, d, ?, \_\}$, then a configuration of a 3-tape Turing machine might look like the state q with three tape heads on three different tapes each with different strings comprised of characters in Γ. If $\omega = \sigma_1\sigma_2...\sigma_n \in \Sigma^*$, then the start configuration for ω is as follows.

- The machine is in its start state q_0.
- The first tape is just like the tape of a standard one-tape deterministic Turing

machine for this input. The leftmost n cells store the symbols $\sigma 1, \sigma_2, ..., \sigma_n$ in order, with all other cells storing $\_$. The tape head points to the leftmost cell on the tape.

- All other tapes are completely filled with $\_$, and their tape heads point to the leftmost cell on the tape as well.

The transition function is now a partial function

$$\delta : Q \times \Gamma^k \to Q \times (\Gamma \times \{L, R, S\})^k.$$

For instance, let us consider the transition as follows.

$$\delta(q, \tau_1, \tau_2, ..., \tau_k) = (r, \hat{\tau}_1, d_1, \hat{\tau}_2, d_2, ..., \hat{\tau}_k, d_k),$$

where $q, r \in Q$, $\tau_1, \tau_2, ..., \tau_k, \hat{\tau}_1, \hat{\tau}_2, ..., \hat{\tau}_k \in \Gamma$, and $d_1, d_2, ..., d_k \in \{L, R, S\}$. Then, if the machine is in state q and the tape head for tape i points to a copy of τ_i, then for $1 \leq i \leq k$, the following changes would result after the next move.

- The state would change to state r.
- The copy of τ_i on the ith tape would be overwritten with a copy of $\hat{\tau}_i$ for $1 \leq i \leq k$.
- For $1 \leq i \leq k$, if $d_i = L$, then the ith head goes left one position unless it is already at the leftmost tape. If $d_i = R$, then the tape head goes right one position, and if $d_i = S$, the tape head does not move.
- The transition $\delta(q, \tau_1, \tau_2, ..., \tau_k)$ should be defined for all states $q \in Q \backslash \{q_{accept}, q_{reject}\}$ and for all symbols $\tau_1, \tau_2, ..., \tau_k \in \Gamma$.
- If $q \in \{q_{accept}, q_{reject}\}$, then the transition $\delta(q, \tau_1, \tau_2, ..., \tau_k)$ should not be defined for any symbols $\tau_1, \tau_2, ..., \tau_k$.

Much like regular Turing machines, the k-tape Turing machine K accepts ω if it is possible to go from the start configuration to an accepting configuration which includes q_{accept} using a finite number of moves. Alternatively, K rejects ω if it is possible to go from the start configuration to a rejecting configuration which includes q_{reject} using a finite number of moves. Otherwise, we say that K loops on ω.

Claim 18.6. Let $L \subseteq \Sigma^*$.

1. If L is Turing-recognizable, then there is a k-tape Turing machine for some integer $k \geq 1$ that recognizes L.
2. Additionally, if L is Turing-decidable, then there is a k-tape Turing machine for some integer $k \geq 1$ that decides L. Furthermore, if $f : \mathbb{N} \to \mathbb{N}$ and L is Turing-decidable using time $f(n)$ for an input of length n in the worst case using a one-tape Turing machine, then L is also Turing-decidable using a k-tape with $f(n)$ time in the worst case as well

Proof. It is easy to note that any regular one-tape Turing machine is also a k-tape Turing machine when $k = 1$. The rest of the proof follows from the definition of Turing-recognizable and Turing-decidable. □

Claim 18.7. Let $L \subseteq \Sigma^*$, and let k be an integer such that $k \geq 1$.

1. If there is a k-tape Turing machine M that recognizes L, then there is also a one-tape Turing machine $\hat{M}$ that recognizes L, so that L is Turing-recognizable.

2. If there is a k-tape Turing machine M that decides L, then there is also a one-tape Turing machine $\hat{M}$ that decides L, so that L is Turing-decidable. Furthermore, if $f : \mathbb{N} \to \mathbb{N}$ and M decides L using time at most $f(n)$ in the worst case for an input string with length n, then M decides L using time in $O\left(max(f(n), n)^2\right)$ in the worst case.

Proof. To prove the above claim, a one-tape Turing machine $\hat{M}$ that simulates the execution of the k-tape Turing machine M can be described. □

There can be more than one argument or simulation that can be used to establish the equivalence of multi-tape Turing machines with one-tape Turing machines. In a similar vein, the complexity class P is also provably unchanged if the model of computation is replaced in its definition with either of the following, provided that the requirement that only a polynomial number of steps can be used in the worst case.

- Deterministic Turing machines that use two-dimensional grids of cells to store data instead of one-dimensional tapes.
- Random access machines that store symbols in an infinite sequence of registers with indices in $\mathbb{N}$, provided that we charge $\Theta(\log m)$ steps to access or modify the contents of register m, for $m \in \mathbb{N}$. Note that this model is reasonably similar to a real computer.

However, there is also evidence against the Cobham-Edmonds thesis, based on the observation that P might be too restrictive. It is widely believed that there is no deterministic polynomial-time algorithm for integer factorization. And, the language of Yes-instances of the Prime Factor in Range problem is in P if and only if such an algorithm exists. Indeed, proofs of security of various cryptographic protocols depend on the assumption that these problems cannot be solved deterministically in polynomial time. However, Shor?s algorithm is a quantum polynomial-time algorithm for integer factorization. Thus, if (or when) quantum computation becomes feasible, then this might lead to the conclusion that there are languages that are efficiently decidable that are not in P.

The thesis may also be incorrect since P may contain too much. For instance, it can be proved that there exists a language $L \subseteq \Sigma^*$ for some alphabet Σ such that there is a deterministic Turing machine M that decides L using time in $O\left(n^{1000}\right)$ for an input string with length n in the worst case, so that this language is in P, but there does not exist any deterministic Turing machine that decides L using time in $O\left(n^{999}\right)$ in the worst case! A reasonably strong argument can be made that a language like L is not efficiently decidable in any practical sense.

§18.5 Historical Figures in Theory of Computation

Alan Mathison Turing was a British pioneering computer scientist, mathematician, logician, cryptanalyst, philosopher, mathematical biologist, and marathon and ultra distance runner. Turing is widely considered to be the father of theoretical computer science and artificial intelligence. The ACM A. M. Turing Award is named after him. The receipt of this award is generally considered to be the highest distinction in computer science, as this is generally considered the Nobel Prize of Computing. During the Second World War, Turing worked for the Government Code and Cypher School (GC&CS) at Bletchley Part, Britain's codebreaking center. Turing's pivotal role in cracking intercepted coded messages enabled the Allies to defeat the Nazis in many crucial engagements, including the Battle of the Atlantic. It is estimated that the work at Bletchley Park shortened the war in Europe by between two to four years.

Alonzo Church was an American mathematician and logician who made major contributions to mathematical logic and the foundations of theoretical computer science. Church was the developer of lambda calculus. Early evidence in support of the Church-Turing thesis included a proof of the equivalence of Turing machines and lambda calculus as computational models.

In 1964, Alan Cobham gave a talk that is often recognized as the birth of the complexity class P. This talk, "The Intrinsic Computational Difficulty of Functions", took place during the Congress for Logic, Methodology and Philosophy of Science at the Hebrew University of Jerusalem, which took place from August 26 to September 2, 1964. Jack Edmonds is American computer scientist who was considering similar ideas at roughly the same time. In a 1965 paper, "Paths, Trees, and Flowers", he drew a distinction between algorithms that increase in difficulty exponentially with the size of the input, and those whose difficulty increases only algebraically. In particular he mentioned the graph isomorphism problem, the complexity of which, along with that of integer factorization, is known to belong to NP but not known to belong to either of the subsets P or NP-complete.

§19 November 16, 2017

§19.1 Nondeterministic Turing Machines

Definition 19.1. A k-tape **nondeterministic Turing machine** is a machine,

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject}),$$

where

1. $Q, \Sigma, \Gamma, q_0, q_{accept}$ and q_{reject} are defined in the same way as for deterministic Turing machines.

2. The transition function δ is now a total function
$$\delta : Q \times \Gamma^k \to P\left(Q \times (\Gamma \times \{L, R, C\})^k\right).$$
Because q_{accept} and q_{reject} are the halting states, this means $\delta(q_{accept}, \sigma_1, \sigma_2, ..., \sigma_k) = \delta(q_{reject}, \sigma_1, \sigma_2, ..., \sigma_k) = \{\}$ for all symbols $\sigma_1, \sigma_2, ..., \sigma_k \in \Gamma$.

Computation of a nondeterministic Turing machine on a string $\omega \in \Sigma^*$ can be modeled as a tree with configurations of the machine at its nodes. The initial configuration of M on input ω is the configuration at the root of this tree. If a configuration C of M has the machine in state q, with symbols $\sigma_1, \sigma_2, ..., \sigma_k$ visible on its tapes, then the number of children of the node with this configuration is equal to the size of the set $\delta(q, \sigma_1, \sigma_2, ..., \sigma_k)$, and there is node for the configuration obtained by applying each of the transitions in this set.

Thus, each path down the tree starting from the root corresponds to one possible computation of M on its input string (corresponding to a series of guesses about which possible transition to apply). M accepts a string ω if and only if there exists at least one path leading to a configuration with M in its accepting state q_{accept}. M recognizes a language $L \subseteq \Sigma^*$ if L is the set of strings in Σ^* that M accepts. M decides a language $L \subseteq \Sigma^*$ if M recognizes L, and the tree of configurations for M on input ω is finite for every string $\omega \in \Sigma^*$.

The time used by M on input $\omega \in \Sigma^*$ is defined to be the depth of the tree of configurations for M on input ω. That is, it is the maximum of the length of any path from the root down to any leaf in this tree. If we let $f : \mathbb{N} \to \mathbb{N}$, then M decides L in time $f(\|\omega\|)$ if M decides L, and the time used by M on input ω is at most $f(\|\omega\|)$ for every string $\omega \in \Sigma^*$. We can define $NTIME(f)$ as the set of languages $L \in \Sigma^*$ such that there exists a nondeterministic Turing machine M that decides L using time in $O(f)$.

Consider again the Prime Factor in Range problem. Additionally, we introduced two additional problems.

Example 19.2 (Verification of Inequality)

The instance is a pair (a, b) of non-negative integers a and b. The question is if $a \leq b$. It can be argued that there is a 2-tape deterministic Turing machine that solves the associated concrete decision problem using a linear number of steps.

Example 19.3 (Multiplication Verification)

The instance is the 3-tuple (a, b, c) of non-negative integers. The question is whether $a = bc$. It can be shown that there is a 6-tape Turing machine that solves the associated concrete decision problem using a quadratic number of steps.

We shall add the dotted digits $\dot{0}, \dot{1}, \dot{2}, \dot{3}, \dot{4}, \dot{5}, \dot{6}, \dot{7}, \dot{8}, \dot{9}$ to our machine's tape alphabet. These can be used to temporarily mark the leftmost cell of each tape by writing the dotted copy of the digit onto the cell instead of the normal digit. This will be used by the nondeterministic Turing machine to be described.

Consider a 9-tape nondeterministic Turing machine M_F whose input alphabet Σ is the same one used to encode instances of the Prime Factor in Range problem. On input $\omega \in \Sigma^*$, M_F does the following.

1. If ω does not encode a pair (n, k) of integers such that $2 \leq k \leq n$, then reject.
2. Write a copy of the encoding $e(n)$ and the separator ; onto the second tape.
3. Nondeterministically guess an encoding of an integer $m \geq 2$. Use the copy of $e(k)$ on the first tape to ensure that the encoding $e(m)$ of m is no longer than the encoding of k. Write the encoding of m onto the second tape following the copy of ; and onto the third tape.
4. Write a copy of ; and the encoding $e(k)$ of k onto the third tape so that this stores an instance of the Verification of Inequality problem. Move the tape head for the third tape back to the beginning and unmark the leftmost symbol. Using the third and fourth tapes, check whether $m \leq k$. Reject if this is not true.
5. Write another ; onto the second tape. Nondeterministically guess a number $l \geq 1$, using the copy of $e(n)$ on the first tape to make sure that the encoding of l is no longer than the encoding of n. Write the encoding of l onto the end of the second tape. Move the tape head for this tape back to the left and unmark the tape.
6. The second tape now stores an encoding of an instance of the Verification of Multiplication problem. Use this tape and tapes five to nine to check whether $n = ml$. Accept if this is true and reject otherwise.

Claim 19.4. M_F decides the language of encodings of Yes-instances of the Prime Factor in Range problem using the encoding scheme previously introduced.

Proof. Suppose that $\omega \in \Sigma^*$ encodes a Yes-instance of the Prime Factor in Range problem (with n and k such that $2 \leq k \leq n$). Then there must exist a prime p such that $2 \leq p \leq k$ and n is divisible by p. If M_F guesses p as the integer m at step 3 and guesses the integer n/p as the integer l at the fifth step, then all tests made by this algorithm are passed and ω is accepted at the sixth step as required.

Suppose instead that $\omega \in \Sigma^*$ does not encode a Yes-instance of the Prime Factor in Range problem. If ω does not encode an instance of the Prime Factor in Range problem at all, then ω is rejected at the first step. Otherwise, ω must encode a pair of integers n and k such that $2 \leq k \leq n$, but there is no positive integer prime p such $2 \leq p \leq k$ and n is divisible by p. It follows that there is no integer m such that $2 \leq m \leq k$ and n is divisible by m (otherwise, any prime divisor p of m would be a suitable prime that we were looking for). The only way to avoid rejecting the input string at the fourth line is to have guessed an integer m at line 3 such that $2 \leq m \leq k$. However, n cannot be divisible by m, so it is impossible to choose an integer l at line 5 such that $n = ml$. If the sixth line is reached at all (which is the only way that ω could be accepted), then it must be rejected during this step as required □

Claim 19.5. The time used by M_F on input $\omega \in \Sigma^*$ is in $O\left(\|\omega\|^2\right)$

Proof. Consider the time required for each of the steps implemented by this Turing machine when it is executed on an input string $\omega \in \Sigma^*$. The number of steps used to carry out each of the first two steps is in $O(\|\omega\|)$. Since the copy of $e(k)$ is being used to make sure that the encoding $e(m)$ of m is no longer than the encoding of k, the third step uses $O(\|\omega\|)$ steps as well. Since the Verification of Inequality problem can be solved deterministically in linear time, the fourth step can be carried out using $O(\|\omega\|)$ steps. Because the copy of the encoding of $e(n)$ on the first tape is being used to make sure that the encoding of the guessed number l is no longer than this encoding, the fifth step uses $O(\|\omega\|)$ steps. Finally, since the Verification of Multiplication problem has a quadratic-time deterministic solution, the sixth step can be carried out using $O\left(\|\omega\|^2\right)$ steps. It follows that the computation tree for the execution of M_F has depth in $O\left(\|\omega\|^2\right)$ as required. □

We say that a concrete decision problem (whose language of instances is in P) is decidable nondeterministically in polynomial time if there is a nondeterministic Turing machine that decides its language of Yes-instances, using time at most polynomial in the length of the input string in the worst case. It follows from the above that the concrete decision problem corresponding to the Prime Factor in Range problem, with the encoding scheme previously described, is decidable nondeterministically in polynomial time.

§19.2 Verification of a Language

Consider a language $L \subseteq \Sigma^*$. Let Σ_C be a possibly different alphabet, and suppose that the symbol $\# \notin \Sigma \cup \Sigma_C$, and let

$$\hat{\Sigma} = \Sigma \cup \Sigma_C \cup \{\#\}.$$

Then the ordered pair consisting of a string $\omega \in \Sigma^*$ and a string $\mu \in \Sigma_C^*$ can be represented using the string

$$\omega\#\mu \in \hat{\Sigma}^*,$$

which includes exactly one $\#$.

Definition 19.6. A **verifier** for a language L is an algorithm (or deterministic Turing machine M) with the following properties.

- The input alphabet for M is an alphabet $\hat{\Sigma}$ defined previously.
- M decides a language $\hat{L}$ that is a subset of the set
$$\{\omega\#\mu | \omega \in \Sigma^* \text{ and } \mu \in \Sigma_C^*\}.$$
- For every string $\omega \in \Sigma^*$, $\omega \in L$ if and only if there exists at least one string $\mu \in \Sigma_C^*$ such that $\omega\#\mu \in \hat{L}$.

If $\mu \in \Sigma_C^*$ such that $\omega\#\mu \in \hat{L}$, then μ is referred to as the **certificate** for ω.

Thus, we can arrive at another definition of $NTIME(f)$ for a function $f : \mathbb{N} \to \mathbb{N}$. A language $L \subseteq \Sigma^*$ is in $NTIME(f)$ if there exists a verifier M such that the number of steps used by M on any string $\omega\#\mu$ (where $\omega \in \Sigma^*$ and $\mu \in \Sigma_C^*$) is in $O(f(\|\omega\|))$.

Remark 19.7. The bound on the number of steps used by M given above depends on the length of the string $\omega \in \Sigma^*$, and not on the rest of the input supplied to the verifier M. In particular, it does not depend on the length of μ. We do not worry about (or constrain) the number of steps used by M when its input does not have the form $\omega\#\mu$.

Once again, consider a Yes-instance of the Prime Factor in Range problem. We could arrive at a certificate for this Yes-instance as being the encoding of a pair of positive integers m and l such that $2 \leq m \leq k$ and $n = ml$. As noted in the proof of the first claim, such integers exist if and only if n and k are part of a Yes-instance of the Prime Factor in Range problem. Suppose we set $\Sigma_C = \Sigma_D$ and define the encoding from possible certificates to be the function mapping the ordered pair (m, l) to the set of strings (containing one string) of $\{e(m); e(l)\}$. Note that the length of the string $e(m); e(l)$ is less than or equal to the length of the string $\omega = e(n); e(k)$ encoding the integers n and k. Now, consider an 11-tape deterministic Turing machine M_V that does the following on input $\omega \in (\Sigma_D \cup \{\#\})^*$.

1. If ω does not begin with a string $\mu \in \Sigma_D^*$ followed by a $\#$, then reject. Otherwise write another copy of μ onto the second tape (with leftmost cells marked using dotted copies for all steps to follow).
2. If μ does not encode an instance of the Prime Factor in Range problem, including a pair of integers n and k such that $2 \leq k \leq n$, then reject.
3. ω does not continue (then end) with a string $\nu \in \Sigma_D^*$ such that $\|\nu\| \leq \|\mu\|$ then reject. Otherwise write a copy of ν onto the third tape.
4. If ν does not encode a pair of integers m and l such that $m \geq 2$ and $l \geq 1$, then reject.
5. Use the strings on the second and third tapes to write $e(m); e(k)$ onto the fourth tape. Move the tape head back to its leftmost cell, unmarking the leftmost symbol on the tape.
6. Use the fourth and fifth tapes to check whether $m \leq k$. Reject if this is not the case.

7. Use the strings on the second and third tapes to write $e(n);e(m);e(l)$ onto the sixth tape. Move the tape head back to the leftmost cell, unmarking the leftmost symbol on the tape.

8. Use tapes six to eleven to decide whether $n = ml$. Accept if this is true and reject if this is false.

Claim 19.8. M_V is a verifier for the language of Yes-instances of the Prime Factor in Range problem using the certificates that have been described above.

Proof. Consider an execution of M_V on a string $\omega \in (\Sigma_D \cup \{\#\})^*$. If ω does not have the form $\mu\#\nu$, where $\mu, \nu \in \Sigma_D^*$ and the length of ν is less than or equal to the length of μ, then this is detected and rejected as required at lines 1 or 3. If μ does not encode an instance of the Prime Factor in Range problem including a pair of integers n and k such that $2 \leq k \leq n$, then this is detected and ω is rejected, as required at the second step. If ν does not encode a pair of integers m and l such that $m \geq 2$ and $l \geq 1$, then ν cannot possibly be a certificate for μ, so this is detected and ω is rejected as required at the fourth step. If $m > k$ so that ν cannot possibly be a certificate for μ, then ω is rejected at the sixth step. If ω has not been rejected for any of the above reasons, then ν is a certificate for μ and should be accepted if and only if $n = ml$. thus, the eight step completes and the claim follows. It can also be shown that M_V executed on $\omega = \mu\#\nu$ where $\mu, \nu \in \Sigma_D^*$ takes at most a quadratic number of steps with respect to μ. □

§19.3 Verification Process

The following process has now been used to show that a language $L \subseteq \Sigma^*$ of Yes-instances of a concrete decision problem can be verified deterministically using time that is at most polynomial in the length of the input string in the worst case.

1. Describe a certificate for a Yes-instance of the decision problem being considered. This is often easy to discover because it is often mentioned in the specification of requirements for the decision problem being considered. Check or prove that every Yes-instance of this problem has at least one certificate, and that there are no No-instance for this problem that have any certificates.

2. Describe an alphabet Σ_C and an encoding $e(C) \to P(\Sigma_C^*)$ from the set of (possible) certificates for Yes-instances of this problem, which we have now defined, to sets of strings in Σ_C^*. Check or prove the following:
   a) Every (possible) certificate has an encoding so that $e(\gamma) \neq \{\}$ for all $\gamma \in C$.
   b) No string encodes more than one instance. Thus, if $\gamma_1, \gamma_2 \in C$ and $\gamma_1 \neq \gamma_2$, $e(\gamma_1) \cup e(\gamma_2) = \{\}$.
   c) Every Yes-instance has at least one short certificate. That is, there is a polynomial function $p : \mathbb{N} \to \mathbb{N}$ such that for every string $\mu \in \Sigma^*$ encoding a Yes-instance of the decision problem being considered, there is at least one string $\nu \in \Sigma_C^*$ encoding a certificate such that $\|\nu\| \leq p(\|\mu\|)$.

3. Describe a verification algorithm that uses the certificates identified that satisfy the bound on certificate length described above. The following structure of an algorithm is recommended.
   a) Check for and reject any input string $\omega \in (\Sigma \cup \Sigma_C \cup \{\#\})^*$ that does not even begin with a string $\mu \in \Sigma^*$, followed by a $\#$. This should be easy for the algorithm to do using time that is linear in the length of its input string.

b) Check for and eliminate all input strings ω that do begin in this way, but such that the prefix $\mu \in \Sigma^*$ does not encode an instance of the decision problem being considered. This step will not be difficult as we are only considering languages of instances that are decidable deterministically in polynomial time.

c) Compute the binary (or decimal) representation of $p(\|\mu\|)$ where $\mu \in \Sigma^*$ as described above. In this course, we will be allowed to use the fact that it is possible to perform such a computation deterministically in polynomial time. Check the input string ω and reject it unless it continues, and ends, with a string $\nu \in \Sigma_C^*$ such that the $\|\nu\| \leq p(\|\mu\|)$.

d) Finally, check the string ν that has now been obtained. Confirm that it a valid encoding of a possible certificate for this problem and that it satisfies all the properties needed to establish that μ encodes a Yes-instance of the decision problem being considered. Accept if this is the case and reject otherwise.

4. Sketch a proof that the Turing machine (or somewhat higher level algorithm) that we have now described really is a verifier for the language of Yes-instances of the decision problem considered.

5. Sketch a proof that if the Turing machine (or algorithm) is executed with an input string $\omega \in (\Sigma \cup \Sigma_C \cup \{\#\})^*$ with the form $\mu\#\nu$ where $\mu \in \Sigma^*$ and $\nu \in \Sigma_C^*$, then the number of steps taken before the Turing machine (or algorithm) halts is a bound function that is at most polynomial in $\|\mu\|$ (not $\|\omega\|$) in the worst case.

§19.4 Equivalence of Models

Definition 19.9. A function $f : \mathbb{N} \to \mathbb{N}$ is **time-constructible** if there is a deterministic Turing machine that maps the string 1^n to the binary representation of $f(n)$ (by writing the second string onto another tape) using time in $O(f(n))$. That is, $f(n)$ can be constructed from n by a Turing machine in the time of order $f(n)$. The set of time-constructible functions include those that generally occur as bounds for running time, including n^k for any integer $k \geq 1$, $\lceil n \log_2 n \rceil$, $\lceil n\sqrt{n} \rceil$, and c^n for any integer $c \geq 2$.

Lemma 19.10

It is possible to set a binary counter to zero, and then set it to a given value m by adding one to it m times. Or, we can decrement the value of a binary counter from m down to zero by subtracting one m times. This can all be done using $O(m)$ steps (even with a Turing machine).

Proof. This is a application of the algorithm analysis technique of *amortized analysis*. Notice that one-half of the additions (or subtractions) require only a single bit of the counter to be modified, one-quarter of the additions (or subtractions) require only two bits of the counter to be modified, one-eighth of the additions (or subtractions) require only three bits of the counter to be modified, etc. The most expensive addition (or subtraction) requires a number of steps linear in $l = \lfloor \log_2 m \rfloor$.

This implies that the total number of steps used for either of the operations described in the lemma is at most linear in

$$\sum_{i=0}^{l} \frac{mi}{2^i} < m \sum_{i \geq 1} \frac{i}{2^i}.$$

If $S = \sum_{i \geq 1} \frac{i}{2^i}$, then $2S = \sum_{i \geq 1} \frac{i}{2^{i-1}} = \sum_{i \geq 0} \frac{i+1}{2^i}$, so that

$$S = 2S - S = 1 + \sum_{i \geq 1} \frac{1}{2^i} = 2,$$

implying that the above step bound is linear in m as required. □

It follows that the function $f(n) = n$ is time constructible. This is useful for proving the following result.

Claim 19.11. If $f : \mathbb{N} \to \mathbb{N}$ is a time-constructible function such that $f \in \Omega(n)$, then the definitions of $NTIME(f)$ that have been given (one involving nondeterministic Turing machines, another involving verifiers) are equivalent. That is, they define the same set of languages $L \subseteq \Sigma^*$.

Proof. This can be shown. □

§19.5 Complexity Class NP

Definition 19.12. The complexity class NP is defined as

$$NP = \bigcup_{k \geq 1} NTIME\left(n^k\right).$$

Using the definition of $NTIME(f)$ involving verifiers, the following propositions can be made.

Proposition 19.13

$TIME(f) \subseteq NTIME(f)$ for every time-constructible function $f : \mathbb{N} \to \mathbb{N}$ such that $f(n) \in \Omega(n)$.

Proof. Suppose there exists a deterministic Turing machine M that decides a language $L \subseteq \Sigma^*$ using $O(f(n))$ steps for input strings with length n in the worst case. Then a verifier can simply use a given string $\omega \# \mu$ to make a copy of ω on another tape (which will now be treated as the input tape) and simulate the execution of M using time in $O(f(n))$ as well. □

Proposition 19.14

If the function $f : \mathbb{N} \to \mathbb{N}$ is time-constructible, $f(n) \in \Theta(n)$, and $L \in NTIME(f)$, then there is a deterministic Turing machine that decides L using $O(cf(n))$ steps (given an input string $\omega \in \Sigma^*$ with length $n = \|\omega\|$) for some constant $c \geq 1$.

Proof. Suppose $\hat{M}$ is a verifier for L using at most $\hat{c}f(n)$ steps on input ω of length n. Suppose that the alphabet Σ_C for this verifier (for the production of certificates) has a size at most d.

Every string in L with length n must have a valid certificate with length at most $\hat{c}f(n) - 1$. That is, the verifier will not have time to read more than the first $\hat{c}f(n) - 1$ symbols in longer certificates because it accepts before this happens. A decision is made as soon as the prefix, with length $\hat{c}f(n) - 1$ has been read, so any prefix of a longer valid certificate with this length must also be a valid certificate itself.

Consider a deterministic Turing machine that starts out by computing the binary representation of $f(n)$ (which can be done efficiently, since f is time-constructible), and then iterates over all possible certificates of the verifier with length at most $cf(n) - 1$. It simulates the execution of the verifier using the input string and every such certificate until a valid certificate is found (so that the deterministic Turing machine should accept the input), or all such certificates have been confirmed to be invalid (so the input should be rejected). Since Σ_C has size d, the number of strings in Σ^* with length at most $\hat{c}f(n) - 1$ is

$$\sum_{i=0}^{\hat{c}f(n)-1} d^i = \frac{d^{\hat{c}f(n)-1} - 1}{d - 1} \leq d^{\hat{c}f(n)}.$$

Since $f(n) \in \Omega(n)$, the number of steps used by this entire simulation can be shown to be in $O\left(d^{\hat{c}f(n)}f(n)\right)$ in the worst case. We now set c to be a constant that is strictly greater than $d^{\hat{c}}$ to complete the proof. $\square$

Corollary 19.15

$$P \subseteq NP \subseteq EXPTIME = \bigcup_{k\geq 1} TIME\left(2^{n^k}\right).$$

This is essentially all that has been proven about the relationship between P and NP.

Conjecture 19.16 (Cook's Conjecture)**.**

$$P \neq NP.$$

§20 November 21, 2017

§20.1 Computing Functions

Definition 20.1. Let k be an integer such that $k \geq 2$. A k-tape Turing machine that computes a function can be modeled as a 7-tuple

$$M = (Q, \Sigma_1, \Sigma_2, \Gamma, \delta, q_0, q_{halt}),$$

where

- Q is the set of states.
- Σ_1 is the input alphabet that does not include $\sqcup$.
- Σ_2 is the output alphabet that does not include $\sqcup$.
- Γ is the tape alphabet such that $Q \cap \Gamma = \{\}$, and $\Sigma_1 \cup \Sigma_2 \cup \{\sqcup\} \subseteq \Gamma$.
- $q_0, q_{halt} \in Q$, where q_0 is the start state and q_{halt} is the halt state.
- $\delta : Q \times \Gamma^k \to Q \times (\Gamma \times \{L, R, S\})^k$ is the transition function. As with deterministic Turing machines that recognize languages, let us require that this is a partial function such that $\delta(q, \sigma_1, \sigma_2, ..., \sigma_k)$ is defined whenever $q \in Q$, $\sigma_1, \sigma_2, ..., \sigma_k \in \Gamma$, and $q \neq q_{halt}$. ON the other hand, $\delta(q_{halt}, \sigma_1, \sigma_2, ..., \sigma_k)$ is undefined for all $\sigma_1, \sigma_2, ..., \sigma_k \in \Gamma$.

- For a string $\omega \in \Sigma^*$, the initial configuration of M on input ω is the same as it would be for a deterministic Turing machine that was deciding whether ω belongs to a certain language. The first tape, which store the copy of ω when execution begins is called the input tape.
- Tape k is the output tape. If the execution of M on input ω halts, then a string $\mu \in \Sigma_2*$ should be written onto the leftmost cells of the output tape, with the output tape head located at the leftmost cell of the tape.
- M is computes a function $f : \Sigma_1^* \to \Sigma_2^*$. For all $\omega \in \Sigma_1^*$, if the execution of M on input ω ever halts, then $f(\omega)$ is defined and equal to the string $\mu \in \Sigma_2^*$ that has been written onto the output tape. Otherwise, $f(\omega)$ is undefined.

§20.2 Reducibilities

Definition 20.2. A **reducibility** is any relation $\preceq_Q$ between languages (possibly over different alphabets) such that $L \preceq_Q L$ for every language $L \subseteq \Sigma^*$ (and over every alphabet Σ), and for all languages $L_1 \subseteq \Sigma_1^*$, $L_2 \subseteq \Sigma_2^*$, and $L_3 \subseteq \Sigma_3^*$, if $L_1 \preceq_Q L_2$ and $L_2 \preceq_Q L_3$, then $L_1 \preceq_Q L_3$. That is, the relation is reflexive and transitive.

Definition 20.3. For languages $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$, a **many-one reduction** from L_1 to L_2 is a well-defined total function $f : \Sigma_1^* \to \Sigma_2^*$ such that

$$\omega \in L_1 \iff f(\omega) \in L_2,$$

for every string $\omega \in \Sigma_1^*$, that is computed by some Turing machine M. L_1 is **many-one reducible** to L_2,

$$L_1 \preceq_M L_2,$$

if there exists a many-one reduction from L_1 to L_2.

Proposition 20.4

$\preceq_M$ is a reducibility. To prove this, we need to show that $L \preceq_M L$ for every language $L \subseteq \Sigma^*$ (and over every alphabet Σ), and for all languages $L_1 \subseteq \Sigma_1^*$, $L_2 \subseteq \Sigma_2^*$, and $L_3 \subseteq \Sigma_3^*$, if $L_1 \preceq_M L_2$ and $L_2 \preceq_M L_3$, then $L_1 \preceq_M L_3$.

Proof. Let $L \subseteq \Sigma^*$ and consider the function $f : \Sigma^* \to \Sigma^*$ such that $f(\omega) = \omega$ for every string $\omega \in \Sigma^*$. A deterministic Turing machine M (computing a function) that copies its input onto its output tape, moves its output tape head back to the leftmost cell and halts after that is a deterministic Turing machine that computes f. Furthermore, f is a well-defined total function from $\Sigma^* \to \Sigma^*$ such that $\omega \in L \iff f(\omega) \in L$ for every string $\omega \in \Sigma^*$. Thus, M is a many-one reduction from L to L as required.

To prove the second part, suppose that $L_1 \subseteq \Sigma_1^*$, $L_2 \subseteq \Sigma_2^*$, and $L_3 \subseteq \Sigma_3^*$. Additionally, suppose that $L_1 \preceq_M L_2$ and $L_2 \preceq_M L_3$. Then there exists a Turing machine M_1 computing a well-defined total function $f_1 : \Sigma_1^* \to \Sigma_2^*$ such that $\omega_1 \in L_1 \iff f_1(\omega_1) \in L_2$ for every string $\omega_1 \in \Sigma_1^*$. Then there also exists a Turing machine M_2 computing $f_2 : \Sigma_2^* \to \Sigma_3^*$ such that $\omega_2 \in L_2 \iff f_2(\omega_2) \in L_3$ for every $\omega_2 \in \Sigma_2^*$. Now, let $f = f_1 \circ f_2$ so that f is a well-defined total function from $\Sigma_1^* \to \Sigma_3^*$ where $f(\omega) = f_2(f_1(\omega))$.

$$\omega \in L_1 \iff f_1(\omega) \in L_2 \iff f(\omega) = f_2(f_1(\omega)) \in L_3,$$

for every string $\omega \in \Sigma_1^*$. Now, consider a Turing machine M that does the following given a string $\omega \in \Sigma_1^*$ as input. First, it uses a copy of M_1 to compute $f_1(\omega)$. Note that the output tape of M_1 is exactly the input to M_2 executed on $f_1(\omega)$. Now, using a copy of M_2, it computes $f(\omega) = f_2(f_1(\omega))$ using the output tape of M_1 as output and additional tapes as required. The output tape of M_2 is used as the output tape of M, as required. It follows that $L_1 \preceq_M L_3$ as required. $\square$

Definition 20.5. For languages $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$, a **polynomial-time many-one reduction** from L_1 to L_2 is a well-defined total function $f : \Sigma_1^* \to \Sigma_2^*$ for every $\omega \in \Sigma_1^*$ where f is computed by some Turing machine M. Additionally, there exists a polynomial function $p : \mathbb{N} \to \mathbb{N}$ such that the number of steps used by M on input ω is at most $p(\|\omega\|)$ for every string $\omega \in \Sigma_1^*$. L_1 is **polynomial-time many-one reducible** to L_2,

$$L_1 \preceq P, ML_2,$$

if there exists a polynomial-time many-one reduction from L_1 to L_2.

Example 20.6

Recall that an integer $n \geq 2$ is composite if there is at least one other integer k such that $2 \leq k \leq n$ where n is divisible by k. Consider the abstract decision problem Composite with instances being integers $n \geq 2$, and the question being whether n is composite. Now, let I_1 be the set of instances of the Composite problem (integers greater than or equal to 2, and I_2 be instances of the Prime Factor in Range problem (ordered pairs (n, k)). Now, consider the function $\hat{f} : I_1 \to I_2$ such that for every integer $n \geq 2$,

$$\hat{f}(n) = \begin{cases} (3,2) & \text{if } n = 2, \\ (n, n-1) & \text{if } n \geq 3. \end{cases}$$

Then, $\hat{f}(\alpha) \in I_2$ for every element α of I_1, so $\hat{f}$ is a total function form I_1 to I_2. After considering the special case of $n = 2$, we can confirm that if α is a Yes-instance of the Composite problem, then $\hat{f}(\alpha)$ is a Yes-instance of the Prime Factor in Range problem. Similarly, if α was a No-instance of Composite, then $\hat{f}(\alpha)$ is a No-instance of the Prime Factor in Range problem.

Recall that instances of the Prime Factor in Range problem can be encoded as strings over an alphabet Σ_D. The empty string λ does not encode an instance of the problem at all, so it certainly does not belong to the language of encodings of Yes-instances of this problem, which we call L_F. The same alphabet Σ_D can be used to encode instances of the Composite problem for integers $n \geq 2$ using their unpadded decimal representations. Let us call the language of encodings of all instances of this problem $L_{C,I} \subseteq \Sigma_D^*$ and let us call the language of Yes-instances $L_{C,Yes}$. Clearly, $L_{C,I} \in P$, as it is possible to decide whether a string $\omega \in \Sigma_D^*$ is an unpadded decimal representation of some integer $n \geq 1$ using linear time with respect to the length of the input string. thus, we set $\Sigma_1 = \Sigma_2 = \Sigma_D$ and define the total function $f : \Sigma_1^* \to \Sigma_2^*$ as follows.

- If $\omega \in \Sigma_1^*$ and $\omega \in L_{C,I}$ so that ω encodes some integer $n \geq 2$, then $f(\omega)$ is the encoding of $\hat{f}(n)$ as described previously.
- On the other hand, if $\omega \notin L_{C,I}$, then $f(\omega) = \lambda$.

It follows that for all $\omega \in \Sigma_1^*$,

$$\omega \in L_{C,Yes} \iff f(\omega) \in L_F.$$

As noted above, $L_{C,I} \in P$, so one can decide which case to apply in order to decide how to compute $f(\omega)$ for a given string $\omega \in \Sigma_1^*$. Given an unpadded decimal representation $\omega \in \Sigma_1^*$ of n, one can easily compute the unpadded decimal representation of an integer $n-1$ for $n \geq 3$, or check whether $n = 2$ in the special case. This can be accomplished using a deterministic multi-tape Turing machine in linear time with respect to $\|\omega\|$. The function f can therefore be computed deterministically in polynomial time. It follows that $L_{C,Yes} \preceq_{P,M} L_F$.

§20.3 Polynomial-Time Many-One Reduction Process

The following process is used to show that the language of Yes-instances of one decision problem P_1 is polynomial-time many-one reducible to the language of Yes-instances of another decision problem P_2.

1. Let I_1 be the set of instances of the decision problem P_1 and let I_2 be the set of instances of the decision problem P_2. Describe a total function $\hat{f} : I_1 \to I_2$ such that for all $\alpha \in I_1$, α is a Yes-instance of P_1 if and only if $\hat{f}(\alpha)$ is a Yes-instance of P_2. Show that this is true if it is not obvious.

2. Consider encodings of the sets of instances I_1 and I_2 using alphabets Σ_1 and Σ_2 respectively. These will generally be defined in problems assigned in this course. Confirm that the language $L_{1,I}$ of encodings of instances of the first decision problem P_1 is in the complexity class P. For problems in this course, this will either be easy to prove or we will be allowed to assume it. Choose some string $\mu \in \Sigma_2^*$ that does not encode a Yes-instance of the second decision problem. It is often possible to choose the empty string λ.

3. Let $L_{1,Yes} \subseteq \Sigma_1^*$ be the language of encodings of Yes-instances of decision problem P_1, and let $L_{2,Yes} \subseteq \Sigma_2^*$ be the language of encodings of Yes-instances of decision problem P_2. Consider a total function $f : \Sigma_1^* \to \Sigma_2^*$ such that for all $\omega \in \Sigma_1^*$,

   - If $\omega \in L_{1,I}$ so that ω encodes some instance $\alpha \in I_1$ of decision problem P_1, then $f(\alpha)$ is an encoding of the instance $f(\alpha)$ of P_2 for the the function $f : I_1 \to I_2$ described previously.
   - On the other hand, if $\omega \notin L_{1,I}$, then $f(\omega) = \mu$, the string described previously that does not encode a Yes-instance of the second decision problem.

   Now, it follows that for all $\omega \in \Sigma_1^*$,

   $$\omega \in L_{1,Yes} \iff f(\omega) \in L_{2,Yes}.$$

4. Describe an algorithm that can be used to compute f. Show that it is correct and that this function uses a number of steps that is at most polynomial in the length of the input string in the worst case. In recent examples, multi-tape Turing machines have been used to describe these algorithms. This can take a long time but is reasonably safe. It is possible to describe algorithms in other ways, like give pseudocode, but we need to be careful to avoid using instructions that cannot actually be carried out deterministically in polynomial time. For problems in this course, a list of instructions that one may safely assume to be implementable in polynomial time will sometimes be given.

Definition 20.7. A language $L \subseteq \Sigma^*$ is NP-hard if it is hard for the complexity class NP with respect to polynomial-time many-one reductions. L is NP-complete if it is complete for the complexity class NP with respect to polynomial-time many-one reductions.

Remark 20.8. It would follow that if a language $L \subseteq \Sigma^*$ is NP-complete, and it was the case that $P \neq NP$, then $L \notin P$. However, **we do not know whether $P = NP$.** Our objective is to find out whether this is the case.

§20.4 Historical Figures in Theory of Computation Cont'd

Oracle reductions were used by Turing to prove a significant result about the undecidability of a problem, when his Turing machine model was proposed. These are often called *Turing reductions* in his honour. Emil Post was a Polish-born American mathematician and logician. He was the first person to make use of many-one reductions, in a paper published in 1944.

Stephen Cook is an American-Canadian computer scientist. In his 1971 paper, The Complexity of Theorem-Proving Procedures, a proof that a problem concerning satisfiability of Boolean formulas was complete for NP with respect to polynomial-time oracle-reducibility was given. This was the first problem that was not completely artificial, with NP-completeness that could be established. Consequently, this was a major result at the time and is still considered to be one today. As a result, polynomial-time oracle-reductions are also called *Cook reductions* in recognition of this. Professor Cook won the ACM Turing Award in 1982, and was named as an Officer of the Order of Canada in 2015.

Richard Manning Karp is an American computer scientist. In his 1972 paper, Reducibility Among Combinatorial Problems, Professor Karp noted that the problem proved by Stephen Cook to be complete for NP with respect to polynomial-time oracle reductions, was complete for NP with respect to polynomial-time many-one reductions as well. Professor Karp also proved numerous other problems to be NP-complete in this sense, in that paper. Consequently, polynomial-time many-one reductions are often called *Karp reductions* in his honour. Professor Karp won the Turing award in 1985 in recognition of this and various other contributions.

§21 November 23, 2017

§21.1 NP-Completeness

Recall that a language is NP-hard if it is hard for NP with respect to polynomial-time many-one reductions, and $NP-$complete if it is complete for NP with respect to polynomial-time many-one reductions. It is relatively simply to prove the existence of an NP-complete problem.

Example 21.1

Consider the set of strings over an alphabet Σ_{NTM} that encodes

- A nondeterministic Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$.
- A string $\omega \in \Sigma^*$ where Σ is the input alphabet for M.
- A non-negative integer k encoded in unary.

such that there exists an accepting computation of M on input ω including at most k steps. After reviewing the encoding for a nondeterministic Turing machine, and the basics of universal Turing machines, it can be shown that this language is NP-complete.

While the above example is an NP-complete problem, it is not clear how this rather artificial language can be used to identify other more interesting languages as NP-complete. The contribution of the *Cook-Levin Theorem* is that it establishes the existence of a less artificial NP-complete language that can be used to identify others.

First, we consider the Boolean Satisfiability problem that is the problem of determining if there exists an interpretation that satisfies a given Boolean formula. In other words, it asks whether the variables of a given Boolean formula can be consistently replaced by the values `true` or `false` in such a way that the formula evaluates to `true`. If this is the case, the formula is called satisfiable. On the other hand, if no such assignment exists, the function expressed by the formula is `false` for all possible variable assignments, and the formula is unsatisfiable. Boolean Satisfiability is the first problem that was proven to be NP-complete

Example 21.2
The formula $a \wedge \neg b$ is satisfiable because one can find the values $a = \texttt{true}$ and $b = \texttt{false}$, which make $a \wedge \neg b = \texttt{true}$. In contrast, $a \wedge \neg a$ is unsatisfiable.

Theorem 21.3 (Cook-Levin Theorem)
The Boolean Satisfiability problem is NP-complete. That is, any problem in NP can be reduced in polynomial time by a deterministic Turing machine to the problem of determining whether a Boolean formula is satisfiable.

Proof. Proof is too long, and not going to help me score higher on the final, so I am leaving it out for now. I will likely include it later. □

Remark 21.4. An important consequence of this theorem is that if there exists a deterministic polynomial time algorithm for solving Boolean satisfiability, then every NP problem can be solved by a deterministic polynomial time algorithm. The question of whether such an algorithm for Boolean satisfiability exists is thus equivalent to the P versus NP problem, which is widely considered the most important unsolved problem in theoretical computer science.

§21.2 Historical Figures in Theory of Computation Cont'd

Leonid Levin is a Soviet-American computer scientist who also made an extremely important contribution to complexity theory. In the 1970's, relations between the east and the west were so strained that information about the mathematical sciences did not get communicated very often or effectively, between one side and the other. Consequently it was not known in the west, until years later, that Professor Levin had discovered the existence of reasonably natural problems that are NP-complete, independently of Professor Cook. The timing is so close that it is not clear who knew what first. Professor Levin is also noted for significant contributions concerning average-case complexity. He was awarded the Knuth Prize in 2012 for these contributions. The result that was called Cook's Theorem concerning NP-completeness in older publications is now commonly called the Cook-Levin Theorem in recognition of his independent discovery of an extremely important fact about computational complexity.

§22 November 28, 2017

§22.1 Establishing NP-Completeness

Suppose that we are given a language $L \subseteq \Sigma^*$ and wish to prove that this language is NP-complete. This will usually be the Yes-instance of some decision problem. The following procedure will be used to show that a language is NP-complete.

1. Prove that $L \in NP$. Refer to process described in previous lectures to show this.

2. Prove that L is NP-hard.

a) Choose some language $\hat{L} \subseteq \hat{\Sigma}^*$ that is already known to be NP-complete. If possible, choose some language $\hat{L}$ that is similar to L.

b) Prove that $\hat{L} \preceq_{P,M} L$. That is, show that $\hat{L}$ is polynomial-time many-one reducible to L. Refer to the process described in previous lectures to show that $\hat{L} \preceq_{P,M} L$.

3. Note that it now follows from the definition of NP-completeness that L is NP-complete.

§23 November 30, 2017

§24 December 5, 2017

§24.1 Additional NP-Complete Problems

It is now known that a large number of languages of encodings of Yes-instances of decision problems are NP-complete. We will now discuss some of the many NP-complete problems.

Example 24.1 (k-Vertex Cover)

iven an undirected graph $G = (V, E)$, a *vertex cover* for G is a subset C of E such that for all $v \in V$, $(u, v) \in C$ for some vertex $u \in V$. Recall that since G is an undirected graph, the edge (u, v) is the same as the edge (v, u). The instance is an undirected graph $G = (V, E)$ and a positive integer k. The question is whether there exists a vertex cover $C \subseteq E$ for G with size at most k.

Solution. The set of instances of the k-Vertex Cover problem are the same as for the k-Clique problem, so we use the same encoding scheme. The k-Vertex Cover problem is NP-complete. We first show NP-hardness by showing that

$$k\text{-Independent Set} \preceq_{P,M} k\text{-Vertex Cover}.$$

■

Example 24.2 (k-Set Cover)

Given a finite set U with size m and a set $S = \{S_0, S - 1, ..., S_{n-1}\}$ of subsets of U, a *set cover* for U is a subset C of S such that

$$U = \bigcup_{S \in C} \bigcup_{x \in S} x.$$

That is, a subset C of S such that every element of U is included in at least one of the sets in C. The instance of this problem is a finite set U with some positive size m, a set $S = \{S_0, S_1, ..., S_{n-1}\}$ of subsets of U, and a positive integer k. the question is whether there exists a set cover $C \subseteq S$ for U with size at most k.

Solution. Renaming the elements of U as required, we can assume that

$$U = \{x_0, x_1, ..., x_{m-1}\}.$$

Consider the alphabet

$$\Sigma_S = \{x, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \{, \}, , \}.$$

- The set U can be encoded as the string $e(U) \in \Sigma_S^*$ consisting of a string of m 1's (the unary representation of its size). Using a unary encoding ensures that the encoding of an instance of the Set Cover problem has a length that is at least linear in U. This is reasonable and simplifies the proof.
- For $0 \leq i \leq m-1$, the element x_i of U can be encoded using the string $e(x_i) \in \Sigma_S^*$ that begins with x and ends with the unpadded decimal representation of i.
- A set S_i of U can be encoded by the string $e(S_i) \in \Sigma_S^*$ that begins with $\{$, continues with the encodings of the elements of S_i, separated by commas and sorted by increasing order of index, and ends with $\}$.
- The collection S can now be encoded by the string $e(S) \in \Sigma_S^*$ that begins with $\{$, continues with the encodings of each of the sets $S_0, S_1, ..., S_{n-1} \subseteq U$ included in S, each separated by commas, and ends with $\}$.
- The positive integer k can be encoded by the string $e(k) \in \Sigma_S^*$ using its unpadded decimal representation.

Thus, an instance of the Set Cover problem can now be encoded by the string $e(U)$, $e(S)$, $e(k)$.

If this encoding scheme is used, then with a bit of tedious work, one can show that the language of encodings of instances of the Set Cover problem is in P. It can also be shown that the language of encodings of Yes-instances of this problem is in NP. In order to show that the language of encodings of Yes-instances is NP-hard, we will show that

$$k\text{-Vertex Cover } \preceq_{P,M} k\text{-Set Cover}.$$

Suppose now that we have an encoding of an instance of the Vertex Cover problem that includes an undirected graph $G = (V, E)$ and a positive integer k. One can begin to form an instance of the Set Cover problem by setting $U = V$. Each edge $(u, v) \in E$ is easily turned into a subset $\{u, v\}$. If one does this for every edge in E, then this produces a collection S of subsets of U such that every set S_i included in S is a set of size two. Clearly, a subset C of E is a vertex cover for G if and only if the corresponding subset of S is a set cover for U, and both of these subsets have the same size.

In essence, the Vertex Cover problem is a special case of the Set Cover problem, so it is relatively simple to argue that k-Vertex Cover $\preceq_{P,M}$ k-Set Cover, as required to argue that the k-Set Cover problem is NP-hard. ■

Example 24.3 (Hamiltonian Cycle)

Once again, we consider an undirected graph $G = (V, E)$. Recall that a *cycle* in G is a sequence

$$w_0, w_1, w_2, ..., w_k$$

of vertices in V such that $k \geq 2$, $(w_i, w_{i+1}) \in E$ for $0 \leq i \leq k-1$, and $w_0 = w_k$. This is a *simple cycle* if $w_0, w_1, ..., w_{k-1}$ are distinct. This is a *Hamiltonian cycle* if this is a simple cycle and $\{w_0, w_1, w_2, ..., w_{k-1}\} = V$ so that every vertex in V is included. The instance is an undirected graph $G = (V, E)$, and the question is whether there exists a Hamiltonian cycle for G.

Solution. Encodings of undirected graphs were described when defining an encoding scheme for the k-Vertex Cover problem. These can be used to define an encoding scheme for this problem such that the language of encodings of instances of the problem is in P. It can be shown that the language of encodings of Yes-instances for this problem is in NP. NP-hardness can also be shown by making use of *widgets* or *gadgets* to construct a graph from an instance of k-Vertex Cover, or from 3-CNF Satisfiability (yes, this is not a proof). ■

Example 24.4 (Traveling Salesman Problem)

The Traveling Salesman problem concerns the situation where a traveling salesperson must visit a finite set S of cities. Distances are defined between cities, so there is a well-defined total function $d : S \times S \to \mathbb{N}$. The salesperson wants to visit all the cities, and get back to the home city while traveling as short a distance as possible. The distance from city x to city y is the same as the distance from city y to city x, so that one might choose to model the set of cities as an undirected graph (rather than a directed graph) with weights associated with edges. The instance is a finite set S with positive size n, a total function $d : S \times S \to \mathbb{N}$ such that $d(x, y) = d(y, x)$ for all $x, y, \in \mathbb{N}$, and a positive integer k. The question is whether there exists a *tour* so that

$$\sum_{i=0}^{n-1} d(x_i, x_{i+1}) \leq k.$$

Recall that a tour is a sequence

$$x_0, x_1, ..., x_n$$

where $\{x_0, x_1, ..., x_{n-1}\} = S$ (meaning that they are distinct), and $x_0 = x_n$ (meaning that the salesperson arrives back home).

Solution. Renaming cities as required, it is reasonable to assume that

$$S = \{x_0, x_1, ..., x_{n-1}\}.$$

Let

$$\Sigma_T = \{x, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, (,), \{, \}, , \}.$$

- The set S can be encoded by the string $e(S) \in \Sigma_T^*$ consisting of a string of n 1's, which would be the unary representation of its size.
- For $0 \leq i \leq n-1$, an element x_i of S can be encoded by the string $e(x_i) \in \Sigma_T^*$ that begins with x and continues until it ends with the unpadded decimal representation of i.
- As usual, a non-negative integer i can be encoded using its unpadded decimal representation $e(i) \in \Sigma_T^*$.
- For $0 \leq i, j \leq n-1 = \|S\| - 1$ and $l \in \mathbb{N}$, the information that $d(x_i, x_j) = l$ can be encoded using a string $\{e(x_i), e(x_j), e(l)\} \in \Sigma_T^*$.
- Recall that $d(x_i, x_j) = d(x_j, x_i)$. The value of $d(x_i, x_i)$ is of no use for $0 \leq i \leq n-1$, since no tour could include a loop from x_i back to itself for any $x_i \in S$. Thus, the encoding of the function d can be the string $e(d) \in \Sigma_T^*$ beginning with $\{$, followed

by the encodings as given previously for i and j such that $0 \leq i < j \leq n-1$ sorted by nondecreasing value of i and increasing j for the same value of i, each separated by commas, and ending with }.

- An instance of the Traveling Salesman problem can now be encoded as
$$e(s), e(d) \in \Sigma_T^*.$$

Using this encoding scheme, it can be shown that the language of encodings of the Traveling Salesman problem is in P, and that the language of encodings of Yes-instances of this decision problem is in NP. One can show that the Traveling Salesman problem is NP-hard by showing that Hamiltonian Cycle $\preceq_{P,M}$ Traveling Salesman. ■

Example 24.5 (Subset Sum)

The instance is a finite set $S = \{i_0, i_1, ..., i_{n-1}\}$ of positive integers and a positive integer *target* T. The question is whether there exists a subset C of S such that
$$\sum_{x \in C} x = T.$$

Solution. Let
$$\Sigma_{SS} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \{, \}, ,\}.$$

- A positive integer i can be encoded by its unpadded decimal representation $e(i) \in \Sigma_{SS}^*$.
- A finite subset S of the set of positive integers can be encoded by the string $e(S) \in \Sigma_{SS}^*$ that begins with {, continues with encodings of the elements of S sorted by increasing order and separated by commas, and ends with }.
- An instance of the Subset Sum problem including a set S and positive integer target T can now be encoded using the string
$$e(S), e(T) \in \Sigma_{SS}^*.$$

One can show that the language of instances of the Subset Sum problem is in NP, and that the language of encodings of Yes-instances in in NP. Subset Sum can be shown to be NP-hard by showing that
$$\text{3-CNF Satisfiability} \preceq_{P,M} \text{Subset Sum}.$$

■

§24.2 Complications

Seemingly small changes to the computational problem often leads to large differences in their proofs for membership in NP and NP-hard. Recall that the 3-CNF Satisfiability problem is NP-complete. A boolean formula F is in *2-conjunctive normal form* if it is in conjunctive normal form and every clause in F includes exactly two literals. This can be used to define a 2-CNF Satisfiability problem. However, the 2-CNF Satisfiability problem is in P. As noted previously, the Subset Sum problem is NP-complete. Let c be a positive integer constant. One can define a Constrained Subset Sum problem by requiring that $T \leq n^c$ where $n = \|S\|$. It would be possible to use dynamic programming or memoization to design a deterministic algorithm that solves the Constrained Subset Sum problem using a number of steps that is at most polynomial with respect to the length of the input string in the worst case.

> There is a genre of problem that is often wrongly claimed to be NP-complete (even in published papers) via reduction from this special case of Subset Sum.

§24.3 Encodings

A variety of objects have now been discussed, including integers, Boolean formulas, truth assignments, undirected graphs, and sets of non-negative integers. These have been used to define languages corresponding to a variety of decision problem, and certificates when describing verification algorithms for languages in NP. Some choices made when defining encoding schemes were quite arbitrary. However, these choices can change the languages that are defined, even when one starts with the same decision problem to be modeled.

Consider a set I of objects, possibly the set of valid inputs for a decision problem, or the set of objects whose encodings will be used as certificates for a verification algorithm. Recall that an encoding scheme for I is a well-defined mapping

$$e : I \to P\left(\Sigma^*\right),$$

from elements of I to sets of strings in Σ^* for some alphabet Σ, satisfying the following properties.

- The set $e(\alpha)$ is nonempty for all $\alpha \in I$.
- These sets are disjoint. That is, if $\alpha, \beta \in I$ and $\alpha \neq \beta$, then $e(\alpha \cap e(\beta) = \{\}$.

Definition 24.6. Now, suppose that $e_1 : I \to P\left(\Sigma_1^*\right)$ and $e_2 : I \to P\left(\Sigma_2^*\right)$ are two different encoding schemes for the same set I. The encoding schemes e_1 and e_2 are **polynomially equivalent** if there exists total functions $f_{1,2} : \Sigma_1^* \to \Sigma_2^*$ and $f_{2,1} : \Sigma_2^* \to \Sigma_1^*$ such that the following properties hold.

- For all $\alpha \in I$ and every string $\omega \in \Sigma_1^*$,

$$\omega \in e_1(\alpha) \iff f_{1,2}(\omega) \in e_2(\alpha).$$

- For all $\alpha \in I$ and every string $\mu \in \Sigma_2^*$,

$$\mu \in e_2(\alpha) \iff f_{2,1}(\mu) \in e_1(\alpha).$$

- Both functions $f_{1,2}$ and $f_{2,1}$ can be computed using a deterministic Turing machine using time that is at most polynomial with respect to the length of the input strings.

Example 24.7

Suppose that $I = \mathbb{N}$. Consider encoding schemes for $\mathbb{N}$ as follows.

- $\Sigma_1 = \{0, 1, ..., 9\}$. For every natural number n, $e_1(n)$ is the set with size one including the unpadded decimal representation of n.
- $\Sigma_2 = \Sigma_1$. For every natural number n, $e_2(n)$ is the infinite set including all decimal representations of n, including representations padded with leading zeroes.
- $\Sigma_3 = \{0, 1\}$. For every natural number n, $e_3(n)$ is the set with size one, including the unpadded binary representation of n.
- $\Sigma_4 = \{1\}$. For every natural number n, $e_4(n)$ is the set with size one, including the unary representation (the string in Σ_4^* with length n).

Claim 24.8. If $1 \leq i, j \leq 3$, then the encoding schemes e_i and e_j are polynomially related.

Proof. This follows by an easy proof, though I do not know what that proof is. □

Claim 24.9. None of e_1, e_2, or e_3 is polynomially related to e_4.

Proof. Note that the length of the unary encoding of n has length exponential in the lengths of either the unpadded decimal or binary representation of n. Thus, there is no function mapping either decimal or binary representations to unary representations that can be computed using a deterministic Turing machine in polynomial time. □

Suppose now that I is a set of instances of a decision problem. Let $e_1 : I \to P\left(\Sigma_1^*\right)$ and $e_2 : I \to P\left(\Sigma_2^*\right)$ be encoding schemes for I. Let $Y \subseteq I$ be the set of Yes-instances of the decision problem and, for $i \in \{1, 2\}$, let

$$L_i = \bigcup_{\alpha \in Y} e_i(\alpha) \subseteq \Sigma_1^*,$$

the language of encodings of Yes-instances of the decision problem using encoding e_i. Suppose that e_1 and e_2 are polynomially equivalent and consider the functions $f_{1,2} = \Sigma_1^* \to \Sigma_2^*$ and $f_{2,1} : \Sigma_2^* \to \Sigma_1^*$ included in the definition of polynomially equivalent encoding schemes. The definitions now given imply that $f_{1,2}$ is a polynomial-time many-one reduction from L_1 to L_2 and that $f_{2,1}$ is a polynomial-time many-one reduction from L_2 to L_1. This can be used to proved the following proposition.

Proposition 24.10

Suppose that e_1, e_2, L_1, and L_2 are as given previously and that the encoding schemes e_1 and e_2 are polynomially equivalent.

1. $L_1 \in P \iff L_2 \in P$.
2. $L_1 \in NP \iff L_2 \in NP$.
3. $L_1 \in NP$-hard $\iff$ $L_2 \in NP$-hard.
4. $L_1 \in NP$-complete $\iff$ $L_2 \in NP$-complete.

Thus, decisions made to define encodings (and encoding schemes) do not matter, so long as the different decisions result in encoding schemes that can be proved polynomially equivalent.