The GMU CSGSA has some resources for studying for this exam on their website. In particular, have a look at Matt Revelle’s problem solving guide.

According to the CSGSA wiki, Professor Richards has recommended the following exercises from Cormen et al. [2001, 2nd. edition] as handy for quals preparation:
1 Common Summations

Arithmetic series
\[ \sum_{k=1}^{n} k = \frac{1}{2} n(n + 1) \]

Sum of squares
\[ \sum_{k=0}^{n} k^2 = \frac{n(n+1)(2n+1)}{6} \]

Sum of cubes
\[ \sum_{k=0}^{n} k^3 = \frac{n^2(n+1)^2}{4} \]

Geometric series, \( x \neq 1 \)
\[ \sum_{k=0}^{\infty} x^k = \frac{x^{n+1} - 1}{x - 1} \]

Geometric series, \( |x| < 1 \)
\[ \sum_{k=0}^{\infty} x^k = \frac{1}{1 - x} \]

Derivative of geometric series, \( |x| < 1 \)
\[ \sum_{k=0}^{\infty} kx^k = \frac{x}{(1 - x)^2} \]

Harmonic series
\[ \sum_{k=1}^{\infty} \frac{1}{k} = \ln n + O(1). \]

2 Loop Invariants

Loop invariants are used for proving the correctness of algorithms with loops. A loop invariant has three parts:

- **Initialization**: It is true prior to the first iteration of the loop.
- **Maintenance**: If it is true before an iteration of the loop, it remains true before the next iteration.
- **Termination**: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.
**Example 1.** The following is a loop invariant that holds for the *Merge* operation in the book:

At the start of each iteration of the *for* loop of lines 14-22, the subarray $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1+1]$ and $R[1..n_2+1]$, in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into $A$.

*Proof.* We must show that the loop invariant holds at initialization, maintenance, and termination.

**Initialization:** Prior to the first iteration of the loop, we have $k = p$, so that the subarray $A[p..k-1]$ is empty and contains the $k-p = 0$ smallest elements of $L$ and $R$. Since $i = j = 1$, both $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into $A$.

**Maintenance:** Suppose $L[i] \leq R[j]$. Then $L[i]$ is the smallest element not yet copied back into $A$. Because $A[p..k-1]$ contains the $k-p$ smallest elements, after line 16 copies $L[i]$ into $A[k]$, the subarray $A[p..k]$ will contain the $k-p+1$ smallest elements. Incrementing $k$ (in the *for* loop update) and $i$ (in line 17) reestablishes the loop invariant for the next iteration. If instead $L[i] > R[j]$, lines 19-20 maintain the loop invariant similarly.

**Termination:** At termination, $k = r+1$. By the loop invariant, the subarray $A[p..k-1]$, which is $A[p..r]$, contains the $k-p = r-p+1$ smallest elements of $L[1..n_1+1]$ and $R[1..n_2+1]$, in sorted order. The arrays $L$ and $R$ together contain $n_1 + n_2 + 2 = r-p+3$ elements. All but the two largest have been copied back into $A$, and these two largest elements are the sentinels. □
2.1 Problems


The following code fragment implements Horner’s rule for evaluating a polynomial

\[ P(x) = \sum_{k=0}^{n} a_k x^k \]

\[ = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + x a_n \cdots))) \]
given the coefficients \(a_0, a_1, \ldots, a_n\) and a value for \(x\):

1: \(y \leftarrow 0\)
2: \(i \leftarrow n\)
3: while \(i \geq 0\) do
4: \(y \leftarrow a_i + x \cdot y\)
5: \(i \leftarrow i - 1\)

Prove that the following is a loop invariant for the while loop in lines 3-6: At the start of each iteration of the while loop of lines 3-6,

\[ y = \sum_{k=0}^{n-(i+1)} a_k+i+1 x^k. \]

Proof.

Initialization: At the first execution of the loop, \(y = 0\) and \(i = n\).

\[ y = \sum_{k=0}^{n} n - (n + 1)a_{k+n+1}x^k = \sum_{k=0}^{n-1} a_{k+n+1}x^k = 0. \]

So the invariant holds at initialization.

Maintenance: When a loop begins, the invariant holds, so \(y = \sum_{k=0}^{n} n - (i + 1)a_{k+i+1}x^k\). At the end of the iteration we have \(i = i' + 1\) from line 5, and from line 4:

\[ y' = a_i + x \sum_{k=0}^{n-i-(i+1)} a_{k+i+1}x^k \]

\[ = a_i + \sum_{k=0}^{n-(i+1)} a_{k+i+1}x^{k+1} \]

\[ = a_i + a_{i+1}x + a_{i+2}x^2 + \cdots + a_n x^{n-i} \]

\[ = a_{i'+1} + a_{i'+2}x + a_{i'+3}x^2 + \cdots + a_n x^{n-(i'+1)} \]

\[ = \sum_{k=0}^{n-(i'+1)} a_{k+i'+1}x^k. \]

Therefore the invariant is maintained.

Termination. At the end of the algorithm, \(i = -1\). Since the invariant holds, we have

\[ y = \sum_{k=0}^{n} a_k x^k, \]

which is the solution to the problem the algorithm is meant to solve. \(\Box\)
Problem 2. Exercise 6.5-5 in [Cormen et al. 2001].

1: function HEAP-INCREASE-KEY(A, i, key)
2:   if key < A[i] then
3:     error “new key is smaller than current key”
4:   A[i] ← key
5:   while i > 1 ∧ A[PARENT(i)] < A[i] do
7:     i ← PARENT(i)

Argue the correctness of HEAP-INCREASE-KEY using the following loop invariant:

At the start of each iteration of the while loop of lines 4-6, the array A[1..heap-size[A]] satisfies the max-heap property, except that there may be one violation: A[i] may be larger than A[PARENT(i)].

Proof. Initialization: At the start of the algorithm, A is assumed to be a heap. After line 4, the value of A[i] may be larger than its parent, but lines 2-3 ensure that A[i] is not smaller than its children, so no other element will violate the max-heap property.

Maintenance:

3 Asymptotic Notation and Recurrence Relations

3.1 Asymptotic Notation

Definition 1.

\[ \Theta(g(n)) = \{ f(n) : \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \}. \]

\[ O(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \}. \]

\[ \Omega(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \}. \]

Theorem 1. For any two functions \( f(n) \) and \( g(n) \), we have \( f(n) \in \Theta(g(n)) \) if and only if \( f(n) \in O(g(n)) \) and \( f(n) \in \Omega(g(n)) \).

Definition 2.

\[ o(g(n)) = \{ f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0 \}. \]

\[ \omega(g(n)) = \{ f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0 \}. \]
Theorem 2. (Transitivity) For $\phi \in \{\Theta, O, \Omega, o, \omega\}$,

$$f(n) \in \phi(g(n)) \text{ and } g(n) \in \phi(h(n)) \text{ imply } f(n) \in \phi(h(n))$$

3.2 Recurrence Relations

Recurrence relations emerge when we analyze the complexity of recursive functions.

The following are the most commonly encountered examples of recurrences:

<table>
<thead>
<tr>
<th>Recurrence</th>
<th>Algorithm</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(n) = T(n - 1) + O(1)$</td>
<td>Sequential Search</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$T(n) = T(n - 1) + O(n)$</td>
<td>Selection/Bubble/Insertion Sort</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>$T(n) = T\left(\frac{n}{2}\right) + O(1)$</td>
<td>Binary Search</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>$T(n) = 2T\left(\frac{n}{2}\right) + O(1)$</td>
<td>Tree Traversal</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$</td>
<td>Merge sort / avg. case Quicksort</td>
<td>$O(n \log n)$</td>
</tr>
</tbody>
</table>

See these notes on the substitution and Master methods.

The substitution method for solving recurrences has two steps:

1. Guess the form of the solution.
2. Use mathematical induction to find the constants and show that the solution works.

Trick for finding the maximum path-depth on an asymmetric recursion tree (such as $T(n) = T(n/3) + T(2n/3) + cn$): If the term in the recurrence that results in the largest tree is $T\left(\frac{an}{b}\right)$, then that path terminates at \((\frac{a}{b})^k n = 1\). The maximum depth is therefore

\[
\left(\frac{a}{b}\right)^k n = 1 \\
\left(\frac{a}{b}\right)^k = \frac{1}{n} \\
\left(\frac{b}{a}\right)^{-k} = \frac{1}{n} \\
-k = \log_{\frac{a}{b}} \frac{1}{n} \\
-k = \log_{\frac{a}{b}} 1 - \log_{\frac{a}{b}} n \\
k = \log_{\frac{a}{b}} n.
\]

See example in § 4.2 of Cormen et al. [2001].

Theorem 3. Master Theorem Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + f(n),$$

where we interpret $n/b$ to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows.

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large $n$, then $T(n) = \Theta(f(n))$. 

7
Example 2. [Cormen et al., 2001, ch. 4]. Determine an upper bound on the recurrence $T(n) = 2T(\lfloor n/2 \rfloor) + n$, given $T(1) = 1$.

Proof. We conjecture that the solution is of the form $T(n) = O(n \lg n)$, i.e. that $\forall n \geq n_0, T(n) \leq cn \lg n$ for an appropriate choice of constants $c > 0$ and $n_0$. We proceed to prove this by induction. Show the inductive step as follows: Assume the conjectured bound holds for $\lfloor n/2 \rfloor$, i.e. that $T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$. Substituting this into the recurrence, we have:

$$T(n) \leq 2(c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n$$
$$\leq cn \lg(n/2) + n$$
$$= cn \lg n - cn \lg 2 + n$$
$$= cn \lg n - cn + n$$
$$\leq cn \lg n,$$

where the last step holds as long as $c \geq 1$. Therefore the inductive step holds for $c \geq 1$.

For the base case, consider $T(2)$:

$$T(2) = 2T(1) + 2$$
$$= 4.$$

Now $T(2) = 4 \leq c2 \lg 2$ holds for all $c > 2/\lg 2$. \qed
3.3 Problems

Problem 3. Asymptotic behavior of polynomials [Cormen et al., 2001, ch. 3]

Let
\[ p(n) = \sum_{i=0}^{d} a_i n^i, \]
where \( a_d > 0 \), be a degree-\( d \) polynomial in \( n \), and let \( k \) be a constant. Use the definitions of the asymptotic notations to prove the following properties.

a. If \( k \geq d \), then \( p(n) \in O(n^k) \).

Proof. We proceed by induction on \( d \).

In the base case, let \( d = 0 \). Then \( p(n) = a_0 \in O(1) \). Since \( O(1) \subset O(n^k) \), \( p(n) \in O(n^k) \).

For the inductive step, we must show that if \( p(n) \in O(n^k) \) when \( k \geq d \), then \( p'(n) \in O(n^k) \) when the degree of the polynomial is \( d + 1 \) and \( k \geq d + 1 \). Consider the case where the degree of the polynomial is \( d + 1 \). Then

\[ p'(n) = \sum_{i=0}^{d+1} a_i n^i \]
\[ = a_{d+1} n^{d+1} + \sum_{i=0}^{d} a_i n^i \]
\[ \leq a_{d+1} n^{d+1} + cn^k, \]
where the last step holds by the inductive hypothesis.

b. If \( k \leq d \), then \( p(n) \in \Omega(n^k) \).

c. If \( k = d \), then \( p(n) \in \Theta(n^k) \).

d. If \( k > d \), then \( p(n) \in o(n^k) \).

e. If \( k < d \), then \( p(n) \in \omega(n^k) \).

Problem 4. [Cormen et al., 2001, ch. 4]. Show that the solution of \( T(n) = T(\lfloor n/2 \rfloor) + 1 \) is \( O(\lg n) \).

Proof. Base case: Assume that \( T(1) = 1 \). Then

\[ T(2) = T(1) + 1 = 2 \leq c \lg 2 \]

for any \( c \geq 2 \).

Inductive step: Assume that \( T(\lfloor n/2 \rfloor) \leq c \lg \frac{n}{2} \) for some constant \( c \). Then

\[ T(n) = T(\lfloor n/2 \rfloor) + 1 \]
\[ \leq c \lg \frac{n}{2} + 1 \]
\[ = c \lg n - c \lg 2 + 1 \]
\[ \leq c \lg n, \]
where the last step holds as long as \( c \geq 1 \).

By induction, \( T(n) \leq c \lg n \) for all \( n \geq 2 \) and any \( c \geq 2 \). Since changing the base of the logarithm only changes the inequality by a constant factor, the base is unimportant, and we have \( T(n) \in O(\lg n) \)
Problem 5. [Cormen et al., 2001, ch. 4]. We saw that the solution of \( T(n) = 2T(\lceil n/2 \rceil) + n \) is \( O(n \lg n) \). Show that the solution of this recurrence is also \( \Omega(n \lg n) \). Conclude that the solution is \( \Theta(n \lg n) \).

Proof. Base case: Assume that \( T(1) = 1 \). Then

\[
T(2) = 2T(1) + 2 = 4 \geq c2 \lg 2
\]

for any \( c \leq 2/\lg 2 \).

Inductive step: Assume that \( T(\lceil n/2 \rceil) \geq c \frac{n}{2} \lg \frac{n}{2} \) for some constant \( c \). Then

\[
T(n) = 2T(\lceil n/2 \rceil) + n
\geq 2c \frac{n}{2} \lg \frac{n}{2} + n
= cn \lg n - cn \lg 2 + n
= cn \lg n + (1 - c \lg 2)n
\geq cn \lg n,
\]

where the last step holds for all \( n \) when \( c \leq 1/\lg 2 \).

By induction, \( T(n) \geq cn \lg n \) for all \( n \geq 2 \) as long as \( c \leq 1/\lg 2 \). Therefore \( T(n) \in \Omega(n \lg n) \). From the example in the book, we already know that \( T(n) \in O(n \lg n) \). Therefore \( T(n) \in \Theta(n \lg n) \). □

Problem 6. [Cormen et al., 2001, ch. 4]. Show that \( \Theta(n \lg n) \) is the solution to the “exact” recurrence for merge sort:

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1, \\
T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{if } n > 1. 
\end{cases}
\]

Proof. Base case: Since \( T(1) \in \Theta(1) \) for some \( c \) we have

\[
T(1) \leq c
\]

[INCOMPLETE] □

\[\text{“}T(1) \text{ is not a function, it is a value. So does this notation in fact mean that “some constant function } f(n) = T(1) \text{ is in } \Theta(1)\text{”?}\]
Problem 7. Exercise 4.2-1 in Cormen et al. [2001]. Use a recursion tree to determine a good asymptotic upper bound on the recurrence

\[ T(n) = 3T(\lfloor n/2 \rfloor) + n. \]

Use the substitution method to verify your answer.

Proof. Ignore the floor for simplicity and assume that \( n \) is a multiple of 2 and \( T(1) \in \Theta(1) \). Since each level of the tree branches into three pieces of size \( n/2 \), the cost contributed by the nodes at depth \( i \) is \( \left( \frac{3}{2} \right)^i n \). There are also \( 3^\lg n = n^{\lg 3} \) leaves, each of cost \( \Theta(1) \). The total cost of the recursion tree is thus

\[
T(n) = n \sum_{i=0}^{\lg n - 1} \left( \frac{3}{2} \right)^i + \Theta(n^{\lg 3})
\]

\[
= n \frac{(3/2)^{\lg n} - 1}{(3/2) - 1} + \Theta(n^{\lg 3})
\]

\[
= 2n^{\lg \frac{3}{2}} - 1 + \Theta(n^{\lg 3})
\]

\[
\in O(n^{\lg \frac{3}{2}}),
\]

where in the second step we used the solution to the finite increasing geometric series. Now we verify this bound via substitution.

Base: Let \( n = 2 \). Then

\[
T(2) = 3T(1) + n = 3\Theta(1) + 2 = 5 \leq c2^{1+\lg \frac{3}{2}}
\]

As long as we choose \( c \geq 5/3 \).

Inductive Step: Assume that \( T(\lfloor n/2 \rfloor) \in O(n^{1+\lg \frac{3}{2}}) \). Then

\[
T(n) = 3T(\lfloor n/2 \rfloor) + n \leq 3c \left( \frac{n}{2} \right)^{1+\lg \frac{3}{2}} + n
\]

\[
= \frac{3}{2} cn^{1+\lg \frac{3}{2}} + n
\]

\[
\in O(n^{1+\lg \frac{3}{2}}).
\]

By induction, then, \( T(n) \in O(n^{1+\lg \frac{3}{2}}) \).
Problem 8. Exercise 4.2-3 in Cormen et al. [2001]. Draw the recursion tree for $T(n) = 4T(\lfloor n/2 \rfloor) + cn$, where $c$ is a constant, and provide a tight asymptotic bound on its solution. Verify your bound by the substitution method.

Proof. The tree is of depth $\lg n - 1$, and it has $2^{\lg n} = n$ leaves. The $i$th level of each tree contributes a cost of $2^i cn$, and the $n$ leaves each cost $T(1) = \Theta(1)$, contributing a cost of $\Theta(n)$. The total cost is therefore

$$T(n) = \sum_{i=0}^{\lg n - 1} 2^i cn + \Theta(n)$$

$$= cn \left[ \frac{2^{\lg n} - 1}{2 - 1} \right] + \Theta(n)$$

$$= cn^2 - cn + \Theta(n)$$

$$\in \Theta(n^2).$$

We now verify the upper and lower bound with substitution.

**Base** (both bounds): Let $n = 2$, and let $T(1) = d$. Then

$$T(n) = 4d + 2c$$

$$= 4(d + \frac{c}{2})$$

$$= (d + \frac{c}{2})n^2.$$

**Inductive Step** (upper bound): Let $T(\lfloor n/2 \rfloor) \in O(n^2)$. Then

$$T(n) = 4T(\lfloor n/2 \rfloor) + cn$$

$$\leq 4d \left( \frac{n}{2} \right)^2 + cn$$

$$= dn^2 + cn$$

$$= (d + \frac{c}{n})n^2$$

$$\leq (d + c)n^2.$$

Therefore, by induction, $T(n) \in O(n^2)$.

**Inductive Step** (lower bound): Let $T(\lfloor n/2 \rfloor) \in \Omega(n^2)$. Then

$$T(n) = 4T(\lfloor n/2 \rfloor) + cn$$

$$\geq 4d \left( \frac{n}{2} \right)^2 + cn$$

$$= dn^2 + cn$$

$$\geq dn^2.$$

Therefore, by induction, $T(n) \in \Omega(n^2)$.

\[\square\]
Problem 9. Problem 4-1, a-f from [Cormen et al. 2001].

\[ T(n) = 2T(n/2) + n^3. \]

Proof. We show that the recurrence is described by case 3 of the Master theorem. Let \( a = b = 2 \) and \( f(n) = n^3 \). Now \( n^3 \in \Omega(n^{\log_a b + \epsilon}) = \Omega(n^{\log_2 2 + \epsilon}) \). Choose \( c = 1/2 \). Then the inequality \( af(n/b) \leq cf(n) \) reduces to

\[ \frac{2n^3}{2^3} \leq \frac{n^3}{2} \]
\[ \frac{1}{4} \leq \frac{1}{2}, \]

which holds for all \( n \).

By the Master theorem, \( T(n) \in \Theta(n^3) \).

The remaining problems can be solved as follows:
- b: Case 3.
- c: Case 2.
- d: Case 3.
- e: Case 1.
- f: Case 2.

Problem 10. Problem 4-1, g from [Cormen et al. 2001].

\[ T(n) = \begin{cases} T(n-1) + n & \text{if } n > 2, \\ k & \text{if } n \leq 2. \end{cases} \]

Note: This models the worst-case running time of insertion sort.

Proof. For the upper bound, we conjecture that \( T(n) = O(n^2) \), i.e. that \( \forall n \geq n_0, T(n) \leq cn^2 \) for some choice of \( c > 0 \) and \( n_0 > 0 \). We prove this by induction.

Base case: For the case \( n = 2 \), we have \( T(2) = k \). Now \( k \leq c4 \) as long as \( c \geq k/4 \), and so \( T(2) \leq c2^2 \).

Inductive step: Let \( T(n-1) = O(n^2) \). Then

\[ T(n) = T(n-1) + n \]
\[ \leq c(n-1)^2 + n \]
\[ = cn^2 - 2cn + 1 + n \]
\[ = cn^2 + (1 - 2c)n + c \]
\[ \leq cn^2 + c \text{ (when } c \geq \frac{1}{2} \text{)} \]
\[ = (c + \frac{c}{n^2})n^2 \]
\[ \leq (c + 1)n^2 \text{ (when } n \geq 1 \text{)}. \]

Thus when \( T(n-1) = O(n^2) \), \( T(n) = O(n^2) \).

By induction, \( T(n) = O(n^2) \) for all \( n \geq 2 \).

By a similar argument, it is easy to show that \( T(n) = \Omega(n^2) \). Therefore \( T(n) = \Theta(n^2) \).
Problem 11. (From the January 2014 qualifying exam:) Give tight upper and lower bounds for the recurrence
\[ T(n) = \begin{cases} 
T(n-1) + \frac{1}{n} & \text{if } n > 2 \\
k & \text{if } n = 2.
\end{cases} \]

Proof. Conjecture: We can see that the recurrence reduces to the expression \( \sum_{i=1}^{n} \frac{1}{i} \), which is a harmonic number. The harmonic numbers follow an approximately logarithmic pattern, so we conjecture that \( T(n) \in \Theta(\lg n) \).

We first show the upper bound via induction.

**Base case:**
\[ T(2) = k \leq c \lg 2 = c, \]
so \( T(2) \leq c \lg 2 \) if \( c \geq k \).

**Inductive Step:** Let \( T(n-1) \leq c \lg(n-1) \). Then
\[
T(n) = T(n-1) + \frac{1}{n} \\
\leq c \lg(n-1) + \frac{1}{n} \\
\leq c \lg n + \lg n \quad \text{(with } n \geq 2) \\
= (c+1) \lg n.
\]
Therefore, by induction, \( T(n) \in O(\lg n) \).

Now for the lower bound.

**Base case:**
\[ T(2) = k \geq c \lg 2 = c, \]
so \( T(2) \leq c \lg 2 \) if \( c \leq k \).

**Inductive Step:** Let \( T(n-1) \leq c \lg(n-1) \). Then
\[
T(n) = T(n-1) + \frac{1}{n} \\
\geq c \lg(n-1) + \frac{1}{n} \\
\geq c \lg n.
\]
Therefore, \( T(n) \geq c \lg n \), and by induction \( T(n) \in \Omega(\lg n) \). Therefore, \( T(n) \in \Theta(\lg n) \). □
Problem 12. Problem 4-2 from Cormen et al. [2001].
An array $A[1..n]$ contains all the integers from 0 to $n$ except one. It would be easy to determine the missing integer in $O(n)$ time by using an auxiliary array $B[0..n]$ to record which numbers appear in $A$. In this problem, however, we cannot access an entire integer in $A$ with a single operation. The elements of $A$ are represented in binary, and the only operation we can use to access them is “fetch the $j$th bit of $A[i]$, which takes constant time.
Show that if we use only this operation, we can still determine the missing integer in $O(n)$ time.

Proof. Consider the following algorithm.

function $\text{FindMissingInteger}(A, n, \text{numDigits})$
  for $j$ from 0 to $\text{numDigits}$ do
    $\text{bitCounts}[j] \leftarrow 0$
  for $i$ from 1 to $n$ do
    for $j$ from 1 to $\text{numDigits}$ do
      if fetchBit[$A[i], j] = 1 then $\text{bitCounts}[j] \leftarrow \text{bitCounts}[j] + 1$
  for $j$ from 0 to $\text{numDigits}$ do
    if $\text{bitCounts}[j] \neq n$ then $\text{result}[j] \leftarrow 1$
    else $\text{result}[j] \leftarrow 0$
  return $\text{result}$

The middle loop of $\text{FindMissingInteger}$ clearly dominates the runtime, which is $O(n \cdot \text{numDigits})$. Since $\text{numDigits}$ is typically held fixed (e.g., 32), we can consider this $O(n)$.

Problem 13. Problem 4-4 in Cormen et al. [2001].
a) $T(n) = 3T(n/2) + n \log n$: Master, case 1.
Problem 14. Problem 5.2-4 in Cormen et al. [2001]. Use indicator random variables to solve the following problem, which is known as the hat-check problem. Each of n customers gives a hat to a hat-check person at a restaurant. The hat-check person gives the hats back to the customers in a random order. What is the expected number of customers that get back their own hat?

Proof. Let $I_i$ denote the indicator

$$I_i = \begin{cases} 
1 & \text{if } i \text{ gets their own hat back} \\
0 & \text{if } i \text{ does not get their own hat back}
\end{cases}$$

Then the expected number $N$ of people who get their own hat back is:

$$N = E\left[ \sum_{i=1}^{n} I_i \right]$$

$$= \sum_{i=1}^{n} E[I_i]$$

$$= \sum_{i=1}^{n} \frac{1}{n}$$

$$= 1.$$

Oddly enough, in this context we can treat each probability independently, even though the variables are not independent.

Problem 15. Exercise 5.2-5 in Cormen et al. [2001]. Let $A[1..n]$ be an array of $n$ distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair $(i, j)$ is called an inversion of $A$. Suppose that the elements of $A$ form a uniform random permutation of $\langle 1, 2, ..., n \rangle$. Use indicator random variables to compute the expected number of inversions.

Proof.

$$N = E\left[ \sum_{i=1}^{n} \sum_{j=i+1}^{n} I[A[i] > A[j]] \right]$$

$$= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{1}{2}$$

$$= \sum_{i=1}^{n} (n - i) \frac{1}{2}$$

$$= \frac{1}{2} \sum_{i=1}^{n} (n - i)$$

$$= \frac{1}{4} (n - 1)n.$$
5 Dynamic Programming

Example 3. Matrix-chain multiplication.

1. **Optimal substructure:** Given a chain of matrix multiplication $A_iA_{i+1} \ldots A_j$, an optimal parenthesization splits the product at some point $k$ into the products $A_i \ldots A_k$ and $A_{k+1} \ldots A_j$. Each of these products is in turn optimally parenthesized.

2. **Recursive solution:**

   $$ m[i,j] = \begin{cases} 
   0 & \text{if } i = j, \\
   \min_{i \leq k < j} \{ m[i,k] + m[k+1,j] + p_{i-1}p_kp_j \} & \text{if } i < j, 
   \end{cases} $$

where $p_{i-1}p_kp_j$ denotes the cost of multiply matrices of the given dimensions.

3. **Computing optimal costs.** The solution in the previous step leads to overlapping subproblems. By filling the table $m$ in order of subproblems of increasing matrix chain length, we can fill the table $m$ without redundant computation.

6 Greedy Algorithms

**Problem 16.** Exercise 16.2-1 from Cormen et al. [2001]: Prove that the fractional knapsack problem has the greedy-choice property.

**Proof.** Let $W$ denote what we can carry and $w_i$ denote the amount of weight we take from item $i$. The optimal substructure of the fractional knapsack problem is described by the recursive solution

$$ C[\mathcal{F}, W] = \begin{cases} 
0 & \text{if } \mathcal{F} = \emptyset \lor W = 0 \\
\max & 
\end{cases} $$


7 Amortized Analysis

As mentioned in this lecture, amortized analysis typically starts with an empty data structure and looks at the average cost of an operation from that point on.

For the potential method, we define an amortized cost $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$, where $c_i$ is the true cost, and we define $\Phi(D_i)$ on the algorithm’s data structure $D_i$ at time $i$. Then we have

$$ \sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) $$

$$ = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0), $$

or

$$ \sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \hat{c}_i - \Phi(D_n) + \Phi(D_0). $$

Thus as long as $\Phi(D_n) \geq \Phi(D_0)$, the total amortized cost provides an upper bound on the true cost.
7.1 Stack Example

Operations: PUSH, POP, MULTIPOP

Aggregate: Observe that each object can be popped at most once for each time it is pushed. Therefore, in aggregate, the entire sequence takes $O(n)$ time.

Accounting: Assign amortized costs of 2, 0 and 0 to PUSH, POP and MULTIPOP, respectively to encode the intuition that PUSH comes with a “credit” to pay for up to 1 POP later. Any sequence of operations with these costs will take $O(n)$ time.

Potential: Define $\Phi(D_i)$ as the number of objects on the stack at time $i$. The amortized cost of PUSH is $c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$. The amortized cost of MULTIPOP is $c_i + \Phi(D_i) - \Phi(D_{i-1}) = k - k = 0$, and the amortized cost of POP is $c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 - 1 = 0$. Any sequence of operations with these costs will take $O(n)$ time.

7.2 Binary Counter Example

Operations: INCREMENT

Aggregate: Observe that during $n$ INCREMENTs, the least significant column is flipped every step, the next column is flipped every other step, and in general the $i$th column undergoes $\left\lfloor \frac{n}{2^i} \right\rfloor$ bit flips. The aggregate cost of all INCREMENTs is thus

$$\sum_{i=0}^{\lfloor \log_2 n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i}.$$  

The sum is a geometric series and converges to 2. Thus the aggregate cost is $O(n)$.

Accounting: Observe that each INCREMENT sets at most one bit, and once a bit is set it is unset at most once. INCREMENT may thus be given an amortized cost of at most 2.

Potential: Define $\Phi(D_i)$ to be the number of 1’s contained in the counter at time $i$. Now a given operation resets at most $t_i$ bits and sets at most one bit to zero, giving it an amortized cost of $\tilde{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \leq (t_i + 1) - (t_i - 1) = 2$. 

18
Problem 17. [Cormen et al. 2001, p. 409] Use aggregate, accounting, and the potential method to show that if a \texttt{DECREMENT} operation were included in the \( k \)-bit counter example, \( n \) operations could cost as much as \( \Theta(nk) \) time.

Proof. Consider a sequence that performs \( 2^{k-1} \) \texttt{INCREMENT} operations and then alternates between \texttt{DECREMENT} and \texttt{INCREMENT}.

Aggregate:

By the book’s observation that the \( i \)th column’s bit is flipped \( \lfloor n/2^i \rfloor \) times when \( n < 2^k \), the total cost of the first \( 2^{k-1} \) \texttt{INCREMENTs is} \( \sum_{i=0}^{k-1} \lfloor \frac{2^{k-1}}{2^i} \rfloor = \sum_{i=0}^{k-1} \lfloor 2^{k-1-i} \rfloor \) flips. At that point, the counter contains a 1 in the leftmost column and zeros elsewhere. For instance, if \( k = 5 \) it contains 10000. The next operation is a \texttt{DECREMENT}, which flips \( k \) bits to yield 01111. Then we \texttt{INCREMENT} again, flipping the \( k \) bits back. Every operation in the sequence from this point on thus flips \( k \) bits. The aggregate cost is thus

\[
\sum_{i=0}^{k-1} \lfloor 2^{k-1-i} \rfloor + \sum_{i=2^{k-1}+1}^{n} k.
\]

When \( n >> 2^{k-1} \), the right term dominates, asymptotically approaching \( nk \). Each operation’s asymptotic amortized cost is thus \( nk/n = k \), and the running time of the algorithm is \( \Theta(nk) \).

Accounting:

Potential:
Problem 18. [Cormen et al., 2001, p. 410] A sequence of \( n \) operations is performed on a data structure. The \( i \)th operation costs \( i \) if \( i \) is an exact power of 2, and 1 otherwise. Use aggregate analysis and the accounting method to determine the amortized cost per operation.

Proof.

Aggregate:

Of the set of numbers less or equal to \( n \), \( \log_2 n \) numbers are an exact power of 2, and \( n - \log_2 n \) are not. Thus the total cost of the sequence of operations is

\[
\sum_{i=0}^{\log_2 n} 2^j + (n - \log_2 n),
\]

so the amortized cost of each single operation is \( \hat{c}_i = 1/\sum \hat{c}_j \in O(1) \).  

Accounting:

Each operation costs 1 unless it is a power of two. Consider a subsequence of operations that begins at one power of two and ends at the next power of two, \( 2^k, 2^k + 1, 2^k + 2, \ldots, 2^{k+1} \). The total length of the sequence is then \( 2^{k+1} - 2^k = 2^k(2 - 1) = 2^k \) operations. All but the last of these operations is of cost 1. The last operation in the sequence is the next power of two, so it costs \( 2^{k+1} \). Therefore there are \( 2^k - 1 \) operations of cost 1, and one operation of cost \( 2^{k+1} \). The total cost of the subsequence is thus

\[
\sum_{i=2^k}^{2^{k+1}} c_i = 2^k + 2^{k+1} - 1,
\]

\[
= \frac{1}{2} 2^{k+1} + 2^{k+1} - 1
\]

\[
= \frac{3}{2} 2^{k+1} - 1.
\]

Assign an amortized cost of 3 to each operation. Then the total amortized cost of the \( 2^k \) operations is

\[
\sum_{i=2^k}^{2^{k+1}} \hat{c}_i = 3 \cdot 2^k
\]

\[
= 3 \frac{1}{2} 2^{k+1}
\]

\[
\geq 3 \frac{3}{2} 2^{k+1} - 1
\]

\[
= \sum_{i=2^k}^{2^{k+1}} c_i.
\]

Therefore the sums of the amortized costs is an upper bound on the true cost for any sequence of operations between two consecutive powers of two. But since any operation occurs between two powers of two, the amortized cost of \( 3 \in O(1) \) applies to all operations in any sequence.

\[\square\]

---

\( ^a \)This argument is invalid. We must first show that the sums is \( O(n) \) before asserting that the amortized cost of an operation is \( O(1) \).
Problem 19. [Cormen et al., 2001, p. 410] A sequence of \( n \) operations is performed on a data structure. The \( i \)th operation costs \( i \) if \( i \) is an exact power of 2, and 1 otherwise. Use aggregate the potential method to determine the amortized cost per operation.

Proof. Potential:

Following the intuition behind the accounting method, each operation that is not a power of 2 pays for itself and pays 2 units to the bank. The amount in the bank is thus 2 times the number of operations that have occurred since the last power of 2. Our potential function will describe the amount of credit in the bank after the \( i \)th operation.

Accordingly, denote the potential after the \( i \)th operation in the sequence has been executed by \( \Phi(i) \), and let \( \Phi(i) = 2(i - 2^{\lfloor \log_2 i \rfloor}) \) when \( i > 0 \) and \( \Phi(0) = 0 \). Since a large amount is withdrawn from the bank when we reach a power of 2, \( \Phi(i) \geq \Phi(i - 1) \). However, it is clear that \( \Phi(i) \geq 0 \) for all \( i > 0 \), and the amortized costs found with this potential function will provide an upper bound on the true cost.

Either \( \exists k \in (\mathbb{Z}^+ \cup 0) \) such that \( i = 2^k \) (i.e. \( i \) is a power of 2), or there is no such \( k \). If there is, then \( \lfloor \log_2 i \rfloor = \lfloor \log_2 (i - 1) \rfloor \). Then the amortized cost of the \( i \)th operation is

\[
\hat{c}_i = c_i + \Phi(i) - \Phi(i - 1) \\
= 1 + 2(i - 2^{\lfloor \log_2 i \rfloor}) - 2(i - 1 - 2^{\lfloor \log_2 (i - 1) \rfloor}) \\
= 1 + 2(i - 2^{\lfloor \log_2 i \rfloor}) - 2(i - 1 - 2^{\lfloor \log_2 (i - 1) \rfloor}) \\
= 2^{\lfloor \log_2 (i - 1) \rfloor} + 2^{\lfloor \log_2 i \rfloor} + 3 \\
= 2^{\lfloor \log_2 (i - 1) \rfloor} + 1 + 2^{\lfloor \log_2 i \rfloor} + 3 \\
= 3 \in O(1).
\]

In the case that \( i = 2^k \) for some integer \( k \geq 0 \), \( \log_2 i = k \) and \( \lfloor \log_2 (i - 1) \rfloor = k - 1 \). Then the amortized cost of the \( i \)th operation is

\[
\hat{c}_i = c_i + \Phi(i) - \Phi(i - 1) \\
= i + 2(i - 2^{\lfloor \log_2 i \rfloor}) - 2(i - 1 - 2^{\lfloor \log_2 (i - 1) \rfloor}) \\
= 2^k + 2(2^k - 2^{k-1}) - 2(2^k - 1 - 2^{k-1}) \\
= 2^k - 2^{k-1} + 2 + 2^k \\
= 2 \in O(1).
\]

Since all operations have an amortized cost in \( O(1) \), and since \( \Phi(n) \geq \Phi(0) \), we have proved that the sequence of \( n \) operations has cost in \( O(n) \).
Problem 20. [Cormen et al., 2001, p. 412] A sequence of stack operations is performed on a stack whose size never exceeds $k$. After every $k$ operations, a copy of the entire stack is made for backup purposes. Show that the cost of $n$ stack operations, including copying the stack, is $O(n)$ by assigning suitable amortized costs to the various stack operations.

Proof. Since the stack size is limited to $k$, then if there are $m$ PUSH’s in the sequence of operations, then there must be at least $\max(0, m - k)$ and at most $m$ POP’s.

Using the accounting method, we give each PUSH an extra credit to pay for its POP in case it has one:

\[
\begin{align*}
\text{PUSH} &= 2 \\
\text{POP} &= 0 \\
\text{COPY} &= k
\end{align*}
\]

Now, COPY is performed at most \(\lfloor n/k \rfloor\) times, contributing a total cost of at most $n$. Since these are the only three operations in the sequences, we have $m + (m - k) \leq n \frac{k-1}{k}$. Since $\frac{k-1}{k}$ is a constant between zero and one, the total number of PUSH’s and POP’s is $O(n)$. Thus the execution of the entire sequence is $O(n)$. \[
\]

Problem 21. [Cormen et al., 2001, p. 412] Suppose we wish not only to increment a counter but also to reset it to zero (i.e., make all bits in it 0). Show how to implement a counter as an array of bits so that any sequence of $n$ INCREMENT and RESET operations takes time $O(n)$ on an initially zero counter. (Hint: Keep a pointer to the high-order 1.)

Problem 22. [Cormen et al., 2001, p. 415] Suppose we have a potential function $\Phi$ such that $\Phi(D_i) \geq \Phi(D_0)$ for all $i$, but $\Phi(D_0) \neq 0$. Show that there exists a potential function $\Phi'$ such that $\Phi'(D_0) = 0$, $\Phi'(D_i) \geq 0$ for all $i \geq 1$, and the amortized costs using $\Phi'$ are the same as the amortized costs using $\Phi$.

Proof. Let $\Phi'(D_i) = \Phi(D_i) - \Phi(D_0)$ for all $i$. Then $\Phi'(D_0) = 0$ and we have

\[
\hat{c}_i' = c_i + \Phi'(D_i) - \Phi'(D_{i-1})
\]

\[
= c_i + \Phi(D_i) - \Phi(D_0) - [\Phi(D_{i-1}) - \Phi(D_0)]
\]

\[
= c_i + \Phi(D_i) - \Phi(D_{i-1})
\]

\[
= \hat{c}_i.
\]
Problem 23. [Cormen et al., 2001, p. 416] Consider an ordinary binary min-heap data structure with \( n \) elements that supports the instructions \textsc{INSERT} and \textsc{EXTRACT-MIN} in \( O(\log n) \) worst-case running time. Give a potential function \( \Phi \) such that the amortized cost of \textsc{INSERT} is \( O(\log n) \) and the amortized cost of \textsc{EXTRACT-MIN} is \( O(1) \), and show that it works.

Proof. After first working through the problem informally with the accounting method, I found that it is useful to think of an insert that occurs when the heap is of size \( m \) as being double-charged in order to pay for the next extraction that occurs when the heap is once again of size \( m \).

Accordingly, let \( \Phi(D_i) = \sum_{k=1}^{n_i} \log k \), where \( n_i \) is the number of elements in the heap after the \( i \)th operation. Clearly, \( \Phi \) is always greater than or equal to zero, so \( \Phi(D_l) \geq \Phi(D_0) \) for any sequence length \( l \). The total amortized cost found by the potential method is therefore guaranteed to be an upper bound on the true cost.

Since insertion increases \( n \) by one, the amortized cost of insertion is

\[
\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = \log n + \sum_{k=1}^{n_i} \log k - \sum_{j=1}^{n_{i-1}} \log j = 2 \log n \in O(\log n).
\]

Since extraction decreases \( n \) by one, its amortized cost is

\[
\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = \log n + \sum_{k=1}^{n_{i-1}} \log k - \sum_{j=1}^{n_i} \log j = \log n - \log n = 0 \in O(1).
\]

Problem 24. [Cormen et al., 2001, p. 416] What is the total cost of executing \( n \) of the stack operations \textsc{PUSH}, \textsc{POP} and \textsc{MULTIPOP}, assuming that the stack begins with \( s_0 \) objects and finishes with \( s_n \) objects?

Proof. Let \( \Phi(D_i) \) denote the number of objects on the stack at time \( i \). We know from the example that the amortized cost of the \textsc{PUSH}, \textsc{POP} and \textsc{MULTIPOP} operations are 2, 0 and 0, respectively – thus each \( \hat{c}_i \) is at most 2. Now the total amortized cost is given as

\[
\sum_{i=0}^{n} c_i = \sum_{i=0}^{n} \hat{c}_i - \Phi(D_0) + \Phi(D_n) = \sum_{i=0}^{n} \hat{c}_i - s_n + s_0 \leq 2n - s_n + s_0.
\]

As long as \( |s_0 - s_n| \leq O(n) \), the total amortized cost is in \( O(n) \).

Problem 25. [Cormen et al., 2001, p. 416] Suppose that a counter begins at a number with \( b \) 1’s in its binary representation, rather than at 0. Show that the cost of performing \( n \) \textsc{INCREMENT} operations is \( O(n) \) if \( n = \Omega(b) \). (Do not assume that \( b \) is constant.)
8 Sorting

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Worst</th>
<th>Best</th>
<th>Stable?</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort</td>
<td>$\Theta(n^2)$</td>
<td>$O(n)$</td>
<td>Yes</td>
<td>Fast for small $n$. In place. On-line.</td>
</tr>
<tr>
<td>Mergesort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Heapsort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Quicksort</td>
<td>$\Theta(n^2)$</td>
<td>$O(n \log n)$</td>
<td>Yes</td>
<td>$\Theta(n \log n)$ average case.</td>
</tr>
<tr>
<td>Counting Sort</td>
<td>$O(n)$</td>
<td>Yes</td>
<td></td>
<td>Requires integers on $[0, k]$, where $k \in O(n)$.</td>
</tr>
<tr>
<td>Radix Sort</td>
<td>$O(d(n + k))$</td>
<td>Yes</td>
<td></td>
<td>$d$ digits, $k$ vals/digit, counting sort subroutine.</td>
</tr>
<tr>
<td>Bucket Sort</td>
<td>$O(n)$</td>
<td></td>
<td></td>
<td>Requires numbers on $[0, 1)$ drawn from a uniform distribution.</td>
</tr>
<tr>
<td>Selection</td>
<td></td>
<td>$O(n^2)$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A lower bound on the worst-case running time of all comparison sorts can be found by considering an abstract decision tree of comparisons needed to sort $n$ numbers. The worst-case running time is the height $h$ of the longest path from the root to a leaf. The leaves of the tree must contain the $n!$ possible output permutations of the sort, but since the tree is binary it has a maximum of $2^h$ nodes, we have

$$2^h \geq n!$$

$$h \geq \lg(n!)$$

$$= \Omega(n \log n)$$

Counting sort fills the value of an array $C[i]$ with the number of elements that take the value $i$ (a linear pass over the data). Then it filters across $C$, executing $C[i] \leftarrow C[i] + C[i - 1]$. $C[i]$ thus stores the number of elements that are less than or equal to $i$, i.e. the index of where they should be placed in the output list.

Radix sort: For each digit from least to most significant, run a stable sort.

Bucket sort:

9 Data structures

Each node $n$ of a binary search tree satisfies the property $leftChild(n) \leq n \leq rightChild(n)$. This facilitates fast search, min and max operations.

9.1 Hash Tables

**Definition 3.** The load factor $\alpha$ for a hash table is defined as the average number of elements stored in a chain, i.e. $n/m$ where $n$ is the number of elements and $m$ is the number of slots.

By assuming simple uniform hashing, in which an object is equally likely to get hashed into any slot, a search (both unsuccessful and successful) costs on average $\Theta(1 + \alpha)$ when collisions are resolved by chaining. It’s easy to prove this for unsuccessful searches. For successful searches, the expected cost is the expected location of the element in the list.

The division method for creating hash functions maps an integer key $k$ to one of $m$ slots via $h(k) = k \mod m$. A prime not too close to a power of 2 is often a good choice of $m$.

9.2 Heaps

**MAX-HEAPIFY**  $O(\lg n)$

**BUILD-MAX-HEAP**
**MAX-HEAPIFY** compares a node \( i \) to its children and swaps it with the largest child. That is, it starts at the root of the (sub)heap and bubbles a node down the heap. Thus it is \( O(lg n) \), i.e. \( O(h) \), where \( h \) is the height of the (sub)tree.

**BUILD-MAX-HEAP** builds a heap on an array \( A \) in \( O(n) \) time by calling **MAX-HEAPIFY** on the nodes \( \lfloor length(A) \rfloor / 2 \) down to 1.

**MAX-HEAP-INSERT-KEY** adds a leaf and bubbles the node up via a call to **HEAP-INCREASE-KEY**. Thus it is \( O(lg n) \).

**EXTRACT-MAX** removes the root, replaces it with the last leaf \( A[length(A)] \) and calls **MAX-HEAPIFY** on the new root.

**HEAP-SORT** is based on the observation that the root of a max-heap is its maximum element. It is essentially a sequence of calls to **EXTRACT-MAX**, but instead it swaps the root and the leaf’s locations in memory when extracting so that it sorts in place.
Problem 26. Show that the running time of BUILD-MAX-HEAP is $O(n)$.

Proof. The loop calls MAX-HEAPIFY once on each non-leaf node. Since MAX-HEAPIFY runs in $O(\lg n) = O(h)$ time, the cost of BUILD-MAX-HEAP is at most

$$\sum_{h=1}^{\lfloor \lg n \rfloor} \left( \text{max # of nodes in a subtree of height } h \right) \cdot \left( \text{cost of } \text{MAX-HEAPIFY} \text{ on a subtree of height } h \right)$$

$$= \sum_{h=1}^{\lfloor \lg n \rfloor} 2^{\lg n - h} \cdot ch$$

$$= \sum_{h=1}^{\lfloor \lg n \rfloor} \frac{n}{2^h} \cdot ch$$

$$= cn \sum_{h=1}^{\lfloor \lg n \rfloor} h \left( \frac{1}{2} \right)^h$$

$$\leq cn \sum_{h=1}^{\infty} h \left( \frac{1}{2} \right)^h$$

To recall the solution of this infinite sum, differentiate the solution of the infinite decreasing geometric series on both sides:

$$\frac{d}{dx} \sum_{k=0}^{\infty} x^k = \frac{d}{dx} \frac{1}{1-x}$$

$$\sum_{k=0}^{\infty} kx^{k-1} = \frac{1}{(1-x)^2}$$

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}.$$}

Using this result in the previous equation, we find that the total cost of BUILD-MAX-HEAP is bounded from above by

$$cn \sum_{h=1}^{\infty} h \left( \frac{1}{2} \right)^h = cn \frac{1/2}{(1-1/2)^2}$$

$$= 2cn$$

$$\in O(n).$$
9.3 Binary Search Trees

**Problem 27.** Exercise 12.2-1 in Cormen et al. [2001]. Suppose that we have numbers between 1 and 1000 in a binary search tree and want to search for the number 363. Which of the following sequences could not be the sequence of nodes examined?

- b. 924, 2200, 911, 244, 898, 258, 362, 363.
- d. 2, 399, 387, 219, 266, 382, 381, 278, 363.
- e. 935, 278, 347, 621, 299, 392, 358, 363.

**Proof.** Sequences (c) and (e) cannot be sequences of nodes examined in a binary search tree. The difference between consecutive numbers in the sequence indicates the direction taken in the tree. When the pair \( x \rightarrow y \) appears in the sequence, we are moving left (\( x \xrightarrow{L} y \)) if \( x > y \), or right (\( x \xrightarrow{R} y \)) if \( x < y \).

In sequence (c), the path 911 \( \xrightarrow{L} 240 \xrightarrow{R} 912 \) appears, indicating that the key 912 appears in the left subtree of the node with key 911. But 912 > 911, so this violates the binary-search-tree property.

In sequence (e), the path 347 \( \xrightarrow{R} 621 \xrightarrow{L} 299 \) appears, indicating that the key 299 appears in the right subtree of the node with key 347. But 299 < 347, so this violates the binary-search-tree property. \( \square \)

10 Graph Algorithms

The **adjacency-list representation** of a graph is preferred for **sparse** graphs, i.e. graphs for which \( |E| \) is much less than \( |V|^2 \). Checking if there is an edge from \( u \) to \( v \) in an adjacency list is \( O(deg(u)) \).

The **adjacency-matrix representation** of a graph may be preferred if

1. The graph is **dense**, i.e. when \( |E| \) is close to \( |V|^2 \), or

2. We need to check if two vertices are connected in constant time.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Task</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>Traversal</td>
<td>( O(V + E) )</td>
</tr>
<tr>
<td>DFS</td>
<td>Traversal, Topological Sort</td>
<td>( \Theta(V + E) )</td>
</tr>
<tr>
<td>Two DFS's</td>
<td>Strongly Connected Components</td>
<td>( \Theta(V + E) )</td>
</tr>
<tr>
<td>Kruskal</td>
<td>Minimum spanning tree</td>
<td>( O(E \lg V) )</td>
</tr>
<tr>
<td>Prim</td>
<td>Minimum spanning tree</td>
<td>( O(E \lg V) ) (binary min-heap), ( O(E + V \lg V) ) (Fibonacci heap)</td>
</tr>
<tr>
<td>Bellman-Ford</td>
<td>Single-source shortest paths</td>
<td>( O(V \lg V) ) (binary min-heap), ( O(V \lg V + E) ) (Fibonacci heap)</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>Single-source shortest paths</td>
<td>( \Theta(V^3) )</td>
</tr>
<tr>
<td>Floyd-Warshall</td>
<td>All-pairs shortest paths</td>
<td>( O((V + E) \lg V) ) (binary min-heap), ( O(V \lg V + E) ) (Fibonacci heap)</td>
</tr>
<tr>
<td>Johnson</td>
<td>Sparse all-paris shortest paths</td>
<td>( O(V^2 \lg V + V E) )</td>
</tr>
<tr>
<td>Ford-Fulkerson</td>
<td>Maximum Flow</td>
<td>( O(E</td>
</tr>
</tbody>
</table>

**Breadth-first search** starts from a root (distance 0) and uses a queue to store explored but unvisited nodes along the **frontier** as it expands. When a node at distance \( d \) is visited, all its non-explored, non-frontier child nodes are added to the queue and assigned distance \( d + 1 \), and an edge to them from the node is added.

BFS yields a **breadth-first tree**, which gives the unweighted shortest path from the root to any reachable vertex.
Depth-first search is run on every non-visited node until all nodes are visited (i.e. pick roots arbitrarily). Nodes remember their discovery time (when they are first visited) and their finishing time (when all their descendants have been visited).

Definition 4. 
DFS can be used to classify edges as they are encountered as follows:
- Tree edges are edges in the depth-first forest $G_\pi$. Edge $(u, v)$ is a tree edge if $v$ was first discovered by exploring the edge $(u, v)$.
- Back edges are those edges $(u, v)$ connecting a vertex $u$ to an ancestors $v$ in a depth-first tree. Self-loops, which may occur in directed graphs, are considered to be back edges.
- Forward edges are those non tree edges $(u, v)$ connecting a vertex $u$ to a descendant $v$ in a depth-first tree.
- Cross edges are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees.

Any graph can be drawn with tree and forward edges head down, and back edges pointing up.

Definition 5. A topological sort of a dag $G = (V, E)$ is a linear ordering of all its vertices such that if $G$ contains an edge $(u, v)$, then $u$ appears before $v$ in the ordering. If the graph is not acyclic, then no linear ordering is possible.

Taking the nodes of a DAG in reverse order of their DFS finishing times yields a topological sort.

Definition 6. A strongly connected component of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices $u$ and $v$ in $C$, we have both $u \rightarrow v$ and $v \rightarrow u$; that is, vertices $u$ and $v$ are reachable from each other.

Definition 7. The transpose of a directed graph $G$ is defined as $G^T = (V, E^T)$, where $E^T = \{(v, u) : (u, v) \in E\}$. That is, $E^T$ consists of the edges of $G$ with their directions reversed.

In an adjacency-list representation, $G^T$ can be computed in $O(V + E)$.

Strongly connected components in a directed graph can be found by this algorithm:
1. Run a DFS on $G$ to get finishing times $f[u]$ for each vertex $u$.
2. Compute $G^T$.
3. Run a version of DFS on $G^T$ whose outer loop chooses root notes to expand in order of decreasing $f[u]$.
4. Each tree discovered in the previous step is a strongly connected component.

Definition 8. A cut $(S, V - S)$ of an undirected graph $G = (V, E)$ is a partition of $V$. An edge is a light edge crossing a cut if its weight is the minimum weight of any edge crossing the cut.

Kruskal’s algorithm for finding the minimal spanning tree of a graph $G = (V, E)$:
1. For all $n \in V$, Make-Set$(n)$
2. For each $(u, v) \in E$, taken in nondecreasing order by weight,
   - If Find-Set$(u) \neq$ Find-Set$(v)$, then add $(u, v)$ to the MST and union the two sets.

Unexplored nodes are stored in a priority queue, ordered by their key value.
Definition 9. Prim's algorithm for finding the minimal spanning tree of a graph $G = (V,E)$ starts with an arbitrary node and adds edges greedily – i.e. it adds a light edge to an isolated node. To quickly find a light edge, Prim's algorithm maintains a value $\text{key}[u]$ for each node $u$, where $\text{key}[u]$ is the minimum-weight edge connecting $u$ to an explored node (or $\infty$ if there is no such edge).

```
function PrimMST(G, r)
    for $u \in G$ do
        $\text{key}[u] \leftarrow \infty$  // The minimum-weight edge connecting $u$ to a node in $V - Q$
        $\pi[u] \leftarrow \text{nil}$  // The parent of $u$ in the MST
        $\text{key}[r] \leftarrow 0$  // The root is the first node we visit
    $Q \leftarrow V(G)$  // Min-priority queue containing unvisited nodes
    while $Q \neq \emptyset$ do
        $u \leftarrow \text{EXTRACT-MIN}(Q)$  // Update the nodes adjacent to $u$ if $(u,v)$ provides a cheaper route
        for $v \in \text{Adj}[u]$ do
            if $(v \in Q) \land w(u,v) < \text{key}[v]$ then
                $\pi[v] \leftarrow u$
                $\text{key}[v] \leftarrow w(u,v)$  // This step implicitly calls DECREASE-KEY on $Q$
        end for
    end while
```

Definition 10. The relaxation procedure uses the triangle inequality ($\delta(s,v) \leq \delta(s,u) + w(u,v)$) to update the upper bound distance $d[v]$ on a node $v$ with an incoming edge from $u$ as follows:

```
function Relax(u, v, w)
    if $d[v] > d[u] + w(u,v)$ then
        $d[v] \leftarrow d[u] + w(u,v)$
        $\pi[v] \leftarrow u$
    end if
end function
```

where $\pi[v]$ denotes the parent of $v$ in the under-construction predecessor subgraph.

The single-source shortest paths problem for DAGs can be solved in $O(V + E)$ like so: sort the vertices topologically, then for each node (in topological order starting from the root), relax each outgoing edge once.

The Bellman-Ford algorithm finds the single-source shortest paths problem in the case where $G$ is a directed graph and weights may be negative. If a negative-weight cycle exists, the algorithm will detect it. It simply makes $|V| - 1$ passes, relaxing every edge in each pass ($O(VE)$). At termination, if any of the distances found violate the triangle inequality ($d[v] > d[u] + w(u,v)$), then a negative-weight cycle must exist.

Dijkstra's algorithm is faster than the Bellman-Ford, but only works on directed graphs with nonnegative weights. It uses a min-priority queue to visit nodes in order of their $d[v]$ values. Whenever it visits a node $v$, it relaxes all the out-going edges of $v$.

10.1 Problems

Problem 28. [Cormen et al., 2001, ch. 23]. Kruskal’s algorithm can return different spanning trees for the same input graph $G$, depending on how ties are broken when the edges are sorted into order. Show that for each minimum spanning tree $T$ of $G$, there is a way to sort the edges of $G$ in Kruskal’s algorithm so that the algorithm returns $T$. 
11 NP-Completeness

Definition 11. A language $L_1$ is polynomial-time reducible to a language $L_2$, written $L_1 \leq_P L_2$, if there exists a polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that for all $x \in \{0,1\}^*$,

$$x \in L_1 \text{ if and only if } f(x) \in L_2.$$ 

Definition 12. A language $L \subseteq \{0,1\}^*$ is NP-complete if

1. $L \in \text{NP}$, and
2. $L' \leq_P L$ for every $L' \in \text{NP}$ (i.e. $L$ is NP-hard).

Theorem 4. If $L$ is a language such that $L' \leq_P L$ for some $L' \in \text{NPC}$, then $L$ is NP-hard. Moreover, if $L \in \text{NP}$, then $L \in \text{NPC}$.

That is to say, we can show that a language is NP-complete by proving that A) it is in NP, and B) it is reducible to an NP-complete problem.

NP-complete problems:

- The 3-CNF satisfiability problem: Given a 3-CNF sentence such as

$$(\neg x_1 \lor x_2 \lor x_3) \land (x_1 \lor \neg x_2 \lor \neg x_3) \land (x_1 \lor x_2 \lor x_3),$$

does there exist some assignment to the variables that makes the sentence true?

- The clique problem (CLIQUE): A clique in an undirected graph $G$ is a completely connected subgraph of $G$. The clique problem asks whether $G$ contains a clique of size $k$.

- The vertex cover problem: A vertex cover is a set of nodes such that for every edge, there is a node in the cover that participates in the edge. The VERTEX-COVER problem asks us to find a vertex cover of size $k$ in an arbitrary graph.

Theorem 5. CLIQUE $\in \text{NPC}$.

Proof. Given a graph $G = (V,E)$, once we have obtained a clique $V' \subseteq V$, we can use $V'$ this as a certificate to check in polynomial time that $V'$ is indeed a clique of size $k$. Therefore CLIQUE $\in \text{NP}$.

Furthermore, we sketch the proof that 3-CNFSatis $\leq_P$ CLIQUE. Given a 3-CNF sentence $\phi$ with $k$ clauses, we can construct a graph $G$ such that $\phi$ is satisfiable if and only if $G$ has a click of size $k$. Create three disconnected nodes for the literals in each clause. Then draw an edge between all pairs of literals in different clauses unless one is the negation of the other.

Now we can show that $\phi$ is satisfiable if and only if $G$ contains a clique of size $k$. Left as an exercise, yadda yadda. Then CLIQUE $\in \text{NPC}$.

Theorem 6. VERTEX-COVER $\in \text{NPC}$.

Proof. The vertex cover itself can serve as a certificate and can be checked in polynomial time. Therefore VERTEX-COVER $\in \text{NP}$.

Now we sketch the proof that CLIQUE $\leq_P$ VERTEX-COVER. Consider a graph $G$ with a clique $V'$ s.t. $|V'| = k$. Now take the compliment of $G$, that is, the graph $\bar{G} = (V, \bar{E})$. Then $V - V'$ is a vertex cover in $\bar{G}$. Conversely, if $\bar{G}$ has a vertex cover $V' \subseteq V$, where $|V'| = |V| - k$, then $V - V'$ is a clique of size $k$ in $G$.

12 Problems From the Spring 2014 Qual

The following are problems (from memory) that appeared on the qual when I took it:
• Solve A) \( T(n) = 2T(n/3) + 1 \), B) \( T(n) = T(n-1) + \frac{1}{n} \).

• Consider a collection of nuts and bolts of different sizes, where each nut is the correct size for one bolt and vice versa. You are able to compare sizes, i.e. by asking whether a bold is the right size, too big, or too small for a nut and vice versa. Give an efficient algorithm to pair the nuts and bolts and analyze its expected running time. You can use your knowledge of how to analyze quicksort.

• Given a set of real numbers, give an algorithm to find the smallest set of unit intervals (intervals containing all \( x \) with \( a \leq x \leq b \) for \( a - b = 1 \)) that contain all the numbers and analyze its running time.

• Give an algorithm to solve the problem of aligning words on a page to minimize the slack on each line \( (c_j + \sum_{k=1}^{j-1} c_k + 1) - L \). One way to solve this problem is to minimize the sum of squared slacks. A solution to this problem is discussed here and I hear rumors a version of it can be found in the latest edition of Cormen et al. [2001].

• Give the worst case \( \Theta \) running time for the following algorithms without justification: Quicksort, DFS, Bellman-Ford (and one other I can’t remember... merge sort?)

• Consider the task of merging a set of files \( \{f_1, f_2, \ldots, f_3\} \), where merging two files \( f_1 \) and \( f_2 \) to produce \( f_1 \parallel f_2 \) costs \( |f_1| + |f_2| \), where \( |f| \) is the length of \( f \). Write a greedy algorithm to merge all the files, and argue that it is optimal.

• You are given two stacks \( A \) and \( B \). Use these stacks to implement a FIFO queue with an INSERT that adds things to the back and a DELETE which removes things from the from. Show that both operations can be given an \( O(1) \) amortized cost.

• You have a truck carrying eggs along a road network to a destination city. Each road segment has a weight associated with it which represents the percent of your eggs that will be broken during transit along that segment. Give an algorithm to find the optimal route in terms of minimizing the number of eggs that are broken (i.e. a dynamic programming algorithm).

**Problem 29.** Given a set of real numbers, give an algorithm to find the smallest set of unit intervals (intervals containing all \( x \) with \( a \leq x \leq b \) for \( a - b = 1 \)) that contain all the numbers and analyze its running time.

**Proof.**

1: function FINDUNITINTERVALS(S)
2: \( S \leftarrow \text{sort}(S) \)
3: \( U \leftarrow \{(S[1], S[1] + 1.0)\} \)
4: \( l \leftarrow S[1] \)
5: for \( i \in 2..\text{length}(S) \) do
6: \( \text{if } S[i] > l + 1.0 \text{ then} \)
7: \( U \leftarrow U \cup \{(S[i], S[i] + 1)\} \)
8: return \( U \)

Assume that \( U \) is a implemented such that adding an element takes constant time (this can be done with a list). Then lines 3-7 run in time linear in the size of \( S \). Assume also line 2 is implemented with a comparison sort. Then line 2 dominates the running time, and the algorithm runs in worst case \( O(n \lg n) \).

In practice, since real numbers are represented in silica as floating point numbers, a version of radix sort could be used to improve this bound to \( O(n) \), provided that the number of bits \( d \in O(n) \).
Problem 30. Give an algorithm to solve the problem of aligning words in a paragraph to minimize the sum of the squared slacks on each line.

Proof. Let $w_1, w_2, \ldots, w_n$ denote the sequence of words, $l(w_i)$ be the length of the $i$th word, and $L$ be the maximum number of characters that can fit on a line. Now, if the words $w_i, \ldots, w_j$ are on the same line, then the slack $s(i, j)$ is defined as the number of character slots that are not filled by the words or the spaces between them:

$$s(i, j) = L - (j - i) - \sum_{k=i}^{j} l(w_k).$$

Our goal is to minimize the sum of the square of this quantity for all lines by making the optimal choice of line breaks.

Thinking top down, we can first choose the breakpoint that optimally splits the text in half. Then the total cost (sum of squared slacks) will be the sum of whatever the cost is of each half taken separately. This intuition reveals the optimal substructure of the problem, and yields the following recursive solution:

$$C[i, j] = \begin{cases} s(i, j)^2 & \text{if } s(i, j) \geq 0 \\ \min_{i \leq k < j} \left( C[i, k] + C[k + 1, j] \right) & \text{otherwise} \end{cases}$$

A tabular dynamic programming algorithm follows directly from this expression. Since each choice breaks the size $n$ of the sub-problems approximately in half, and since $n$ steps are examined at each step, the algorithm’s average time is roughly described by the recurrence $T(n) = 2T(n/2) + n$, the solution to which is in $O(n \log n)$.

To show this formally, we would have to be much more precise about the claim that each choice splits the problem “approximately in half” – we could probably do it by drawing on the way Quicksort is analyzed.

Alternatively, we could think of ourselves as choosing the point to break the first line, then the second, and so on (instead of choosing break points in the middle).

$$C[i, n] = \begin{cases} s(i, j)^2 & \text{if } s(i, j) \geq 0 \\ \min_{i \leq k < j} \left( s(i, k)^2 + C[k + 1, j] \right) & \text{otherwise} \end{cases}$$

This makes it easier to see that if we were minimizing the sums of slacks instead of the sum of squared slacks, a greedy solution would be possible. I believe both solutions are equally correct.
Problem 31. Consider the task of merging a set of files \( \{f_1, f_2, \ldots, f_3\} \), where merging two files \( f_i \) and \( f_j \) to produce \( f_i \| f_j \) costs \( |f_i| + |f_j| \), where \( |f| \) is the length of \( f \). Write a greedy algorithm to merge all the files, and argue that it is optimal.

Proof. This is the optimal merge pattern problem. Naively, I thought we could just use tail recursion to fold the merge function over the files in order from least to greatest size (a.k.a. a two-way merge):

\[
\begin{align*}
\text{function } & \text{TailCat}(F) \\
& F \leftarrow \text{sortAscending}(F) \quad \triangleright \text{Sort the files from smallest to greatest in size.} \\
& \text{return GreedyCatSub}(F)
\end{align*}
\]

This algorithm is not optimal, however. Instead of folding, we need to think in terms of a merge tree. The following algorithm (also described here) is in fact optimal:

\[
\begin{align*}
\text{function } & \text{GreedyCat}(F) \\
& Q \leftarrow F \quad \triangleright \text{A priority queue initialized to contain all files keyed on their length.} \\
& \text{while } |Q| > 1 \text{ do} \\
& \quad u \leftarrow \text{extract-min}(Q) \\
& \quad v \leftarrow \text{extract-min}(Q) \\
& \quad Q \leftarrow Q \cup \{u\|v\} \\
& \quad \text{return extract-min}(Q)
\end{align*}
\]

Consider the files \( \{f_1, f_2, \ldots, f_n\} \) taken in order such that \( |f_i| \leq |f_{i+1}| \), and let \( S_{ij} \) denote the cost of optimally merging the files \( \{f_i, f_{i+1}, \ldots, f_j\} \). Then the optimal substructure of the merging problem is described by the recursive solution

\[
S_{ij} = \begin{cases} 
|f_i| & \text{if } i = j \\
\min_{i \leq k \leq j} 2(S_{ik} + S_{kj}) & \text{if } i \neq j,
\end{cases}
\]

where the cost is \( 2(S_{ik} + S_{kj}) \) because we must first compute the solutions to the sub-problems (for a cost of \( S_{ik} + S_{kj} \)) and then merge them (which also costs \( S_{ik} + S_{kj} \)).

This gets us to a tabular dynamic programming solution that runs in \( O(n^3) \), but stops short of proving the greedy-choice property...

\[\text{INCOMPLETE}\]
**Problem 32.** You are given two stacks \(A\) and \(B\). Use these stacks to implement a FIFO queue with an \texttt{INSERT} that adds things to the back and a \texttt{DELETE} which removes things from the front. Show that both operations can be given an \(O(1)\) amortized cost.

**Proof.** Let the queue be represented as the 3-tuple \(D_i = (A, B, onA)\), where \(A, B\) are (initially empty) stacks, and \(onA\) is a Boolean flag.

1: \textbf{function} \texttt{INSERT}(e, A, B, onA) \hspace{0.5cm} \triangleright \text{Expose the back and push.}
2: \hspace{1cm} \textbf{if} onA = \texttt{false}
3: \hspace{1.5cm} \textbf{then}
4: \hspace{2cm} \textbf{MoveAll}(B, A)
5: \hspace{1.5cm} onA \leftarrow \texttt{true}
6: \hspace{1cm} \textbf{push}(e, A)
7: \textbf{function} \texttt{DELETE}(A, B, onA) \hspace{0.5cm} \triangleright \text{Expose the front and pop.}
8: \hspace{1cm} \textbf{if} onA = \texttt{true}
9: \hspace{1.5cm} \textbf{then}
10: \hspace{2cm} \textbf{MoveAll}(A, B)
11: \hspace{1.5cm} onA \leftarrow \texttt{false}
12: \hspace{1cm} \textbf{pop}(B)

11: \textbf{function} \texttt{MoveAll}(X, Y) \hspace{0.5cm} \triangleright \text{Move all elements from } X \text{ to } Y
12: \hspace{1cm} \textbf{while} X \neq \emptyset \hspace{0.5cm} \textbf{do}
13: \hspace{1.5cm} \textbf{push}(\textbf{pop}(X), Y)

Let

\[
\Phi(D_i) = \begin{cases} 
|A| & \text{if } onA_{i-1} = \texttt{true}, \\
|B| & \text{if } onA_{i-1} = \texttt{false}, 
\end{cases}
\]

where \(|X|\) denotes the number of elements on stack \(X\) after the \(i\)th operation. Then \(\Phi(D_i) \geq \Phi(D_0) = 0\), and the total amortized cost of a sequence of \texttt{INSERT} and \texttt{DELETE} operations obtained from \(\Phi(D_i)\) is guaranteed to be an upper bound on the true cost.

When inserting an element, either \(onA = \texttt{true}\) or \(onA = \texttt{false}\). In the case that \(onA = \texttt{true}\),

\[
\check{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 0 \in O(1).
\]

If \(onA = \texttt{false}\) and there are \(t_i\) elements on stack \(B\), then

\[
\check{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = t_i + 1 - t_i = 1 \in O(1).
\]

When deleting an element, either \(onA = \texttt{true}\) or \(onA = \texttt{false}\). In the case that \(onA = \texttt{true}\) and there are \(t_i\) elements on stack \(A\),

\[
\check{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = t_i + 1 - t_i = 0 \in O(1).
\]

If \(onA = \texttt{false}\), then

\[
\check{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 0 \in O(1).
\]

Thus all operations have \(O(1)\) amortized cost, and any sequence of \texttt{INSERT} and \texttt{DELETE} operations on the queue will run in \(O(n)\) time. \(\square\)

*Problem! \(onA\) has a different value when evaluating \(\Phi(D_{i-1})\) than it does when evaluating \(\Phi(D_i)\). I think this solution is totally invalid.*
Problem 33. You have a truck carrying eggs along a road network to a destination city. Each road segment has a weight associated with it which represents the percent of your eggs that will be broken during transit along that segment. Give an algorithm to find the optimal route in terms of minimizing the number of eggs that are broken (i.e. a dynamic programming algorithm).

Proof. Assume that the network among a set of nodes \( V \) is represented by a weight matrix \( W \) with 0’s along the diagonal, and let nonexistent road segments be represented by a weight of \( \infty \). It’s then tempting to express the percentage of eggs remaining after traveling the best route between nodes \( X \) and \( Y \) as

\[
D_{XY} = \begin{cases} 
1 & \text{if } X = Y \\
\max_{v \in V} D_{Xv}(1 - W_{vY}) & \text{otherwise.}
\end{cases}
\]

Turning this directly into an algorithm, however, yields an infinite loop when there are cycles in the network. One almost wishes there were an iterative solution... which is exactly what Dijkstra’s algorithm provides. Since all the weights are on \([0, 1]\), a multiplicative version of the triangle inequality applies. Let \( \delta(x, y) \) denote the fraction of eggs preserved after traveling from \( x \) to \( y \). Then

\[
\delta(s, v) \geq \delta(s, u) \delta(u, v).
\]

We can now modify Dijkstra’s algorithm to use multiplication in the \texttt{relax} step and to maximize the fraction of preserved eggs:

\[
\text{function MaxMultDijkstra}(V, W, s)
\]

\[
S \leftarrow \emptyset \quad \triangleright \text{The set of nodes whose cost has been accurately computed.}
\]

\[
\text{for } v \in V \text{ do}
\]

\[
d[v] \leftarrow -\infty \quad \quad \pi[v] \leftarrow \text{NIL}
\]

\[
d[s] \leftarrow 1
\]

\[
Q \leftarrow V \quad \quad \triangleright \text{A priority queue containing all nodes, keyed on } d.
\]

\[
\text{while } Q \neq \emptyset \text{ do}
\]

\[
u \leftarrow \text{extract-max}(Q)
\]

\[
S \leftarrow S \cup \{u\}
\]

\[
\text{for } v \in \text{children}[u] \text{ do}
\]

\[
\text{mult-relax}(u, v, d, W, \pi)
\]

\[
\text{return } \pi
\]

\[
\text{function mult-relax}(u, v, d, W, \pi)
\]

\[
\text{if } d[v] < d[u] \cdot (1 - W_{u,v}) \text{ then}
\]

\[
d[v] \leftarrow d[u] \cdot (1 - W_{u,v})
\]

\[
\pi[v] \leftarrow u \quad \quad \triangleright \text{This line involves an implicit call to } \texttt{increase-key} \text{ on } Q.
\]

The equivalent of a negative weight in this problem is a weight of greater than 1. Since this doesn’t occur, the Dijkstra approach holds. Technically this is a greedy algorithm, however, not so much a dynamic programming algorithm.

\[
\text{References}
\]


35