Section 1.1 – Algorithms

1.1-1 Give a real-world example that requires sorting or a real-world example that requires computing a convex hull.

> Sorting. In a dictionary, it is essential to use sorting so that one can easily find the desired word.
> Convex hull. After conducting a voting intention survey, it may be interesting to know its coverage area. One can calculate the aproximate area by projecting the covered cities to a two dimensional plane, obtain the convex hull of the projected cities, and then compute the approximate area of the convex hull.

1.1-2 Other than speed, what other measures of efficiency might one use in a real-world setting?

> For algorithms in general, we can also optimize for low memory usage or low power consumption. In machine learning algorithms, accuracy (hit rate) is also considered a measure of efficiency.

1.1-3 Select a data structure that you have seen previously, and discuss its strengths and limitations.

> Linked List is a basic data structure. Some of its strengths are:
>
> - Given a pointer to an element in the list, we can insert an element after or before it in constant time.
> - Given a pointer to an element in the list, we can delete it in constant time.
>
> Some of its limitations are:
>
> - The pointers requires extra memory.
> - Since it only has pointers to the next element, it takes linear time to retrieve the i-th element.

1.1-4 How are the shortest-path and traveling-salesman problems given above similar? How are they different?

> They are similar because both of them aims at minimizing the distance between A and B, given a set of possible valid paths. However, the traveling-salesman problem has an additional constraint: for a path to be valid, besides starting in A and ending in B, it also needs to pass through a set of other points C, D, . . . , E before reaching B.

1.1-5 Come up with a real-world problem in which only the best solution will do. Then come up with one in which a solution that is "approximately" the best is good enough.

> In a competition, each candidate received a score for her/his performance. To obtain the ranking list of the candidates, only the best sorting solution is accepted. Approximated sorting algorithms are not feasible in this situation.
> Recently, Facebook computed the approximate degree of separation between every two people in the world. Since Facebook has billion of users, it would take too long to compute the solution that takes into account all the connections between all the users. Also, an approximate result is very feasible in this case. They then used an approximate to get the result of 3.57.

Section 1.2 – Algorithms as a technology

1.2-1 Give an example of an application that requires algorithmic content at the application level, and discuss the function of the algorithms involved.

> The search engines we have today involves a lot of complex algorithms to work. It needs a ranking algorithm to sort the search results appropriately. It is also important to use a crawler that systematically browses and indexes the web content. During searching, this indexed content is gathered and filtered from the database using sophisticated algorithms.

1.2-2 Suppose we are comparing implementations of insertion sort and merge sort on the same machine. For inputs of size n, insertion sort runs in $8n^2$ steps, while merge sort runs in $64n \lg n$ steps. For which values of n does insertion sort beat merge sort?

> For input values less than or equal to 43, insertion sort beats merge sort. We can ignore the case where $n = 1$, since a single element is already sorted by definition.

1.2-3 What is the smallest value of n such that an algorithm whose running time is $100n^2$ runs faster than an algorithm whose running time is 2^n on the same machine?

> The smallest value of n is 15.

Problems

1-1 Comparison of running times. For each function $f(n)$ and time t in the following table, determine the largest size n of a problem that can be solved in time t, assuming that the algorithm to solve the problem takes $f(n)$ microseconds.

$f(n)$	**1 second**	**1 minute**	**1 hour**	**1 day**	**1 month**	**1 year**	**1 century**
$\lg n$	2^{10^6}	$2^{10^7 \times 6}$	$2^{10^8 \times 36}$	$2^{10^8 \times 864}$	$2^{10^9 \times 2592}$	$2^{10^9 \times 31536}$	$2^{10^{11} \times 31536}$
$\sqrt{n}$	10^{12}	$10^{14} \times 36$	$10^{16} \times 1296$	$10^{16} \times 746496$	$10^{18} \times 6718264$	$10^{18} \times 994519296$	$10^{22} \times 994519296$
n	10^6	$10^7 \times 6$	$10^8 \times 36$	$10^8 \times 864$	$10^9 \times 2592$	$10^9 \times 31536$	$10^{11} \times 31536$
$n \lg n$	62746	2801418	133378059	2755147513	71870856404	797633893349	68610956750570
n^2	10^3	7745	$10^4 \times 6$	293938	1609968	5615692	561569229
n^3	10^2	391	1532	4420	13736	31593	146645
2^n	9	25	31	36	41	44	51
$n!$	9	11	12	13	15	16	17

Section 2.1 – Insertion sort

2.1-1 Using Figure 2.2 as a model, illustrate the operation of INSERTION-SORT on the array $A = \langle 31, 41, 59, 26, 41, 58 \rangle$.

(a) 31 41 59 26 41 58

(b) 31 41 59 26 41 58

(c) 31 41 59 26 41 58

(d) 26 31 41 59 41 58

(e) 26 31 41 41 59 58

(f) 26 31 41 41 58 59

2.1-2 Rewrite the INSERTION-SORT procedure to sort into non-increasing instead of non-decreasing order.

The pseudocode is stated below.

```
   InsertionSortNonIncreasing(A)
1  | for j = 2 to A.length do
2  |   | key = A[j]
3  |   | i = j − 1
4  |   | while i > 0 and A[i] > key do
5  |   |   | A[i + 1] = A[i]
6  |   |   | i = i − 1
7  |   | A[i + 1] = key
```

2.1-3 Consider the ***searching problem***:

Input: A sequence of n numbers $A = \langle a_1, a_2, \ldots, a_n \rangle$ and a value ν.
Output: An index i such that $\nu = A[i]$ or the special value NIL if ν does not appear in A.

Write pseudocode for linear search, which scans through the sequence, looking for ν. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

The pseudocode is stated below.

```
   LinearSearch(A, ν)
1  | for i = 1 to A.length do
2  |   | if A[i] == ν then
3  |   |   | return i
4  | return NIL
```

Here is the *loop invariant*. At the start of each iteration of the **for** loop of lines 1–3, the algorithm assures that the subarray $A[1, \ldots, i - i]$ does not contain the element ν. Within each iteration, if $A[i]$ corresponds to the ν element, its index is returned.

- **Initialization.** Before the **for** loop, $i = 1$ and $A[1, \ldots, i - 1]$ constains no element (therefore does not contain ν).
- **Maintenance.** The body of the **for** loop verifies if $A[i]$ corresponds to the ν element. If the element correspond to ν, its index is returned. Otherwise, incrementing i for the next iteration of the **for** loop then preserves the loop invariant.
- **Termination.** The **for** loop can terminate in one of the following conditions: (1) $A[i] = \nu$, which means that ν was found and its index is returned; (2) $i > A.length$ and, since each loop iteration increases i by 1, at that time we have $i = A.length + 1$ which assures (from the previous property) that $A[1, \ldots, A.length]$ does not contain the element ν.

2.1-4 Consider the problem of adding two n-bit binary integers, stored in two n-element arrays A and B. The sum of the two integers should be stored in binary form in an $(n + 1)$-element array C. State the problem formally and write pseudocode for adding the two integers.

The pseudocode is stated below. Integers are stored in little endian format.

```
   AddIntegers(A, B)
1  | let C[1, ..., n + 1] be a new array
2  | C[1] = 0
3  | for i = 1 to A.length do
4  | |  s = A[i] + B[i] + C[i]
5  | |  C[i] = s  mod 2
6  | |  C[i + 1] = s / 2
7  | return C
```

Section 2.2 – Analyzing algorithms

2.2-1 Express the function $n^3/1000 - 100n^2 - 100n + 3$ in terms of Θ-notation.

$\Theta(n^3)$.

2.2-2 Consider sorting n numbers stored in array A by first finding the smallest element of A and exchanging it with the element in $A[1]$. Then find the second smallest element of A, and exchange it with $A[2]$. Continue in this manner for the first $n-1$ elements of A. Write pseudocode for this algorithm, which is known as ***selection sort***. What loop invariant does this algorithm maintain? Why does it need to run for only the first $n-1$ elements, rather than for all n elements? Give the best-case and worst-case running times of selection sort in Θ-notation.

The pseudocode is stated below.

```
   SelectionSort(A)
1  | for i = 1 to A.length − 1 do
2  | |  smallest = i
3  | |  for j = i + 1 to A.length do
4  | |  |  if A[j] < A[smallest] then
5  | |  |  |  smallest = j
6  | |  tmp = A[i]
7  | |  A[i] = A[smallest]
8  | |  A[smallest] = tmp
```

Here is the *loop invariant*. At the start of each iteration of the **for** loop of lines 1–8, the subarray $A[1, \ldots, i-i]$ consists of the $(i-1)$ smallest elements of the array A in sorted order.

- **Initialization.** Before the **for** loop, $i = 1$ and $A[1, \ldots, i-1]$ constains no element.
- **Maintenance.** The body of the **for** loop looks on the subarray $A[i+1, \ldots, A.length]$ for a element that is smaller than $A[i]$. If a smaller element is found, their positions in A are exchanged. Since the subarray $A[1, \ldots, i-1]$ already contains the i smallest elements of A, the smaller element between $A[i]$ and $A[i+1, \ldots, A.length]$ is the i-th smallest element of A, which maintains our *loop invariant* for the subarray $[1, \ldots, i]$.
- **Termination.** The condition causing the **for** loop to terminate is that $i == A.length - 1$. At that time, $i = A.length = n$. Since (from the previous property) the subarray $A[1, \ldots, n-1]$ consists of the $(n-1)$ smaller elements A, the lasting element $A[n]$ can only be the n-th smaller element.

It needs to run only for the first $(n-1)$ element because, after that, the subarray $A[1, \ldots, n-1]$ consists of the $(n-1)$ smaller elements of A and the n-th element is already in the correct position.

Regardless of the content of the input array A, for $i = 1, 2, \ldots, (A.length - 1)$ the algorithm will always look for the i-th element in the whole subarray $A = [i+1, A.length]$. Thus, the algorithm takes $\Theta(n^2)$ for every input.

2.2-3 Consider linear search again (see Exercise 2.1-3). How many elements of the input sequence need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array? How about in the worst case? What are the average-case and worst-case running times of linear search in Θ-notation? Justify your answers.

Lets consider an array of size n, where each element is taken from the set $1, \ldots, k$. If k is not a function of n, its a constant. In the average case, each comparison has probability $1/k$ to find the element that is being searched, resulting in an average of k comparisons. Thus, in the average case, as a function of the input size, the algorithm takes $\Theta(k) = \Theta(1)$. The worst case occurs when $k >= n$, which takes $\Theta(n)$.

2.2-4 How can we modify almost any algorithm to have a good best-case running time?

Verify if the input is already solved. If it is solved, do nothing. Otherwise, solve it with some algorithm.

Section 2.3 – Analyzing algorithms

2.3-1 Using Figure 2.4 as a model, illustrate the operation of merge sort on the array $A = \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$

3	9	26	38	41	49	52	57
3	26	41	52	9	38	49	57
3	41	26	52	38	57	9	49
3	41	52	26	38	57	9	49

2.3-2 Rewrite the MERGE procedure so that it does not use sentinels, instead stopping once either array L or R has had all its elements copied back to A and then copying the remainder of the other array back into A.

The pseudocode is stated below.

```
   Merge(A, p, q, r)
1  | n_1 = q − p + 1
2  | n_2 = r − q
3  | let L[1, ..., n_1] and R[1, ..., n_2] be new arrays
4  | for i = 1 to n_1 do
5  | |  L[i] = A[p + i − 1]
6  | for j = 1 to n_2 do
7  | |  R[j] = A[q + j]
8  | i = 1
9  | j = 1
10 | for k = p to r do
11 | |  if q + j > r or L[i] ≤ R[j] then
12 | |  |  A[k] = L[i]
13 | |  |  i = i + 1
14 | |  else
15 | |  |  A[k] = R[j]
16 | |  |  j = j + 1
```

2.3-3 Use mathematical induction to show that when n is an exact power of 2, the solution of the recurrence

$$T(n) = \begin{cases} 2 & \text{if } n = 2, \\ 2T(n/2) + n & \text{if } n = 2^k, \text{ for } k > 1 \end{cases}$$

is $T(n) = n \lg n$.

The base case is trivial, since $T(2) = 2 \lg 2 = 2$. To prove that it holds for $n > 2$ using mathematical induction, we need to show that if it holds for $n - 1$, it also holds for n. From the recurrence, $T(n) = 2T(n/2) + n$. But by inductive hypothesis, $T(n/2) = (n/2) \lg(n/2)$, so we get that:

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2(n/2) \lg(n/2) + n \\ &= n \lg(n/2) + n \\ &= n(\lg(n) - \lg(2)) + n \\ &= n \lg(n) - n + n \\ &= n \lg(n). \end{aligned}$$

2.3-4 We can express insertion sort as a recursive procedure as follows. In order to sort $A[1, \ldots, n]$, we recursively sort $A[1, \ldots, n-1]$ and then insert $A[n]$ into the sorted array $A[1, \ldots, n-1]$. Write a recurrence for the worst-case running time of this recursive version of insertion sort.

The recurrence is stated below.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ T(n-1) + \Theta(n) & \text{if } n > 1. \end{cases}$$

It takes $\Theta(n^2)$.

2.3-5 Referring back to the searching problem (see Exercise 2.1-3), observe that if the sequence A is sorted, we can check the midpoint of the sequence against ν and eliminate half of the sequence from further consideration. The ***binary search*** algorithm repeats this procedure, halving the size of the remaining portion of the sequence each time. Write pseudocode, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is $\Theta(\lg n)$.

The pseudocode is stated below.

```
   BinarySearch(A, s, e, ν)
1  |  if s > e then
2  |  |  return NIL
3  |  m = ⌊(s + e)/2⌋
4  |  if ν > A[m] then
5  |  |  BinarySearch(A, m + 1, e, ν)
6  |  else if ν > A[m] then
7  |  |  BinarySearch(A, s, m − 1, ν)
8  |  else
9  |  |  return m
```

In each recursion level, the algorithm compares ν with the central element $A[m]$. If $\nu = A[m]$, the element was found and it just returns the position. If $A[m]$ is bigger (or smaller) than ν, the algorithm discards the left half (or the right half) of the array and continues recursively in the remaining $\lfloor (n-1)/2 \rfloor$ elements. Each recursion element compares ν with a single element of A, thus each level takes $\Theta(1)$. Since the number of elements in the array is halved in each level, there will be at most $\lg n$ recursion levels. The algorithm then takes at most $\lg n \times \Theta(1) = \Theta(\lg n)$.

2.3-6 Observe that the while loop of lines 5–7 of the INSERTION-SORT procedure in Section 2.1 uses a linear search to scan (backward) through the sorted subarray $A[1, \ldots, j-1]$. Can we use a binary search (see Exercise 2.3-5) instead to improve the overall worst-case running time of insertion sort to $\Theta(n \lg n)$?

No, because even finding the correct position in $\lg n$, after each search the algorithm will still need to shift up to n the elements to keep the subarray $A[1, \ldots, j]$ sorted. The worst-case running time will remain $\Theta(n^2)$.

2.3-7 ($\star$) Describe a $\Theta(n \lg n)$-time algorithm that, given a set S of n integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x.

Start by sorting S using MERGESORT, which takes $\Theta(n \lg n)$. For each element i of S, $i = 1, \ldots, n$, search the subarray $A[i+1, \ldots, n]$ for the element $\nu = x - S[i]$ using BINARYSEARCH. If ν is found, return its position. Otherwise, continue for the next value of i. It will perform at most n searchs and each search takes $\Theta(\lg n)$. The algorithm then takes $\Theta(n \lg n) + n \times \Theta(\lg n) = \Theta(n \lg n)$.

Problems

2-1 ***Insertion sort on small arrays in merge sort***

Although merge sort runs in $\Theta(n \lg n)$ worst-case time and insertion sort runs in $\Theta(n^2)$ worst-case time, the constant factors in insertion sort can make it faster in practice for small problem sizes on many machines. Thus, it makes sense to ***coarsen*** the leaves of the recursion by using insertion sort within merge sort when subproblems become sufficiently small. Consider a modification to merge sort in which n/k sublists of length k are sorted using insertion sort and then merged using the standard merging mechanism, where k is a value to be determined.

a. Show that insertion sort can sort the n/k sublists, each of length k, in $\Theta(nk)$ worst-case time.

b. Show how to merge the sublists in $\Theta(n \lg(n/k))$ worst-case time.

c. Given that the modified algorithm runs in $\Theta(nk + n \lg(n/k))$ worst-case time, what is the largest value of k as a function of n for which the modified algorithm has the same running time as standard merge sort, in terms of Θ-notation?

d. How should we choose k in practice?

(a) Sort n/k sublists of length k with insertion sort takes $n/k \cdot \Theta(k^2) = \Theta(n/k \cdot k^2) = \Theta(nk)$.

(b) The naive solution is to extend the standard merging procedure to merge n/k sublists at the same time, instead of two. Since there is n/k sublists, in each iteration the algorithm takes $\Theta(n/k)$ to select the lowest element among all the sublists. Since there are n elements (thus n iterations), the total complexity is $n \cdot \Theta(n/k) = \Theta(n^2/k)$.

We can accomplish the requested $\Theta(n \lg(n/k))$ complexity by merging the sublists pairwise, rather than merging them all at the same time. Lets first consider the case in which the number of sublists is even. In the first level there will be $n/(2k)$ pairs of sublists to merge and, since each sublist has length k, each merge will take $\Theta(2k)$. Thus, the first level will take $n/(2k) \cdot \Theta(2k) = \Theta(n)$. The next level will have half the number of sublists and will take $n/(4k) \cdot \Theta(4k) = \Theta(n)$. Since the number of sublists is reduced by two on each level, the total number of levels will be $\lg(n/k)$. Thus, the total cost is $\Theta(n) \cdot \lg(n/k) = \Theta(n \lg(n/k))$. When the number of sublists is odd, it will need one additional level to merge the remaining sublist. Thus, the total cost is $\Theta(n \lg(\lceil n/k \rceil))$.

(c) When $k = 1$ (smallest possible value for k), the modified algorithm takes $\Theta(n \cdot 1 + n \lg(n/1)) = \Theta(n + n \lg n)$. When k grows, the first term grows and the second term decreases. Thus, since the second term can not be greater than $n \lg n$, we just need to pay attention to the first term. The algorithm then takes more than $\Theta(n \lg n)$ when $nk > n \lg n \rightarrow k > \lg n$. Thus, the largest value of k is $\lg n$.

(d) It depends of the constant factors of insertion sort and merge sort. Since the cost of these constants may vary between different machines, in practice one should choose the largest value of k in which insertion sort is faster then merge sort in a given machine.

2-2 ***Correctness of bubblesort***

Bubblesort is a popular, but inefficient, sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order.

```
   BubbleSort(A)
1  | for i = 1 to A.length − 1 do
2  | |  for j = A.length downto i + 1 do
3  | |  |  if A[j] < A[j − 1] then
4  | |  |  |  exchange A[j] with A[j − 1]
```

a. Let A' denote the output of BubbleSort(A). To prove that BubbleSort is correct, we need to prove that it terminates and that

$$A'[1] \leq A'[2] \leq \cdots \leq A'[n].$$

where $n = A.length$. In order to show that BubbleSort actually sorts, what else do we need to prove?

b. State precisely a loop invariant for the **for** loop in lines 2–4, and prove that this loop invariant holds. Your proof should use the structure of the loop invariant proof presented in this chapter.

c. Using the termination condition of the loop invariant proved in part (b), state a loop invariant **for** the for loop in lines 1–4 that will allow you to prove in- equality (2.3). Your proof should use the structure of the loop invariant proof presented in this chapter.

d. What is the worst-case running time of bubblesort? How does it compare to the running time of insertion sort?

(a) A' must be a permutation of A.

(b) Here is the *loop invariant*. At the start of each iteration j of the for loop of lines 2–4, $A[j]$ is the smallest element of the subarray $A[j, \ldots, A.length]$.

- **Initialization.** Prior to the first iteration of the loop, $j = n = A.length$, so the subarray $A[j, \ldots, A.length]$ has only one element, and $A[j]$ is therefore the smallest element of the subarray $A[j, \ldots, A.length]$.
- **Maintenance.** To see that each iteration maintains the loop invariant, let's suppose that $A[j-1] > A[j]$. Because $A[j]$ is the smallest element of the subarray $A[j, \ldots, A.length]$, after line 4 exchanges the position of the elements $A[j]$ and $A[j-1]$, $A[j-1]$ will be the smallest element of the subarray $A[j-1, \ldots, A.length]$. Incrementing j (in the for loop update) reestablishes the loop invariant for the next iteration. If instead $A[j-1] < A[j]$, nothing needs to be done and $A[j-1]$ is already the smallest element of the subarray $A[j-1, \ldots, A.length]$.
- **Termination.** At termination, $j = i$. By the loop invariant $A[i]$ is the smallest element of the subarray $A[i, \ldots, A.length]$.

(c) Here is the *loop invariant*. At the start of each iteration i of the for loop of lines 1–4, the subarray $A[1, \ldots, i-1]$ consists of the i smallest elements of A in sorted order.

- **Initialization.** Prior to the first iteration of the loop, we have $i = 1$, so that the subarray $A[1, \ldots, i-1]$ is empty. This empty subarray contains the $i - 1 = 0$ smallest elements of A in sorted order.
- **Maintenance.** In each iteration i, the subarray $A[1, \ldots, i-1]$ constains the $i-1$ smallest elements of A in sorted order. After the for loop of lines 2–4, $A[i]$ will be the smallest element of the subarray $A[i, \ldots, A.length]$ and thus the i-th smallest element of A. This implies that $A[1, \ldots, i]$ will contain the i smallest elements of A in sorted order. Incrementing i (in the for loop update) reestablishes the loop invariant for the next iteration.
- **Termination.** At termination, $i = A.length$. By the loop invariant the subarray $A[1, \ldots, A.length - 1]$ consists of the smallest elements of A in sorted order. Since $A[A.length]$ can only be the largest element of A, it is already in its correct position and the subarray $A[1, \ldots, A.length]$ consists of the elements of A in sorted order.

(d) The worst running time of Bubble-Sort is $\Theta(n^2)$, which is the same of Insertion-Sort. However, the best running time of Insertion-Sort is $\Theta(n)$ (when the array is already sorted) and Bubble-Sort runs always in $\Theta(n^2)$.

2-3 ***Correctness of Horner's rule***

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

$$\begin{aligned} P(x) &= \sum_{k=0}^{n} a_k x^k \\ &= a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + x a_n) \cdots)), \end{aligned}$$

given the coefficients $a_0, a_1, \ldots, a_n$ and a value for x:

```
1 y = 0
2 for i = n downto 0 do
3 |   y = a_i + x · y
```

a. In terms of Θ-notation, what is the running time of this code fragment for Horner's rule?

b. Write pseudocode to implement the naive polynomial-evaluation algorithm that computes each term of the polynomial from scratch. What is the running time of this algorithm? How does it compare to Horner's rule?

c. Consider the following loop invariant:

At the start of each iteration of the **for** loop of lines 2–3,

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

Interpret a summation with no terms as equaling 0. Following the structure of the loop invariant proof presented in this chapter, use this loop invariant to show that, at termination, $y = \sum_{k=0}^{n} a_k x^k$.

d. Conclude by arguing that the given code fragment correctly evaluates a polynomial characterized by the coefficients $a_0, a_1, \ldots, a_n$.

(a) Since the body of the for loop of lines 2–3 consists of constant operations, the running time depends on the number of iterations of the loop. The running time is then $\sum_{i=0}^{n} 1 = n + 1 = \Theta(n)$.

(b) Here is the pseudocode of the NaivePolynomialEvaluation algorithm:

```
1 y = 0
2 for i = 0 to n do
3 |   y_i = a_i
4 |   for k = 1 to i do
5 |   |   y_i = y_i x
6 |   y = y + y_i
```

The running time of the above algorithm is $\sum_{i=0}^{n} i = (n(n+1))/2 = n^2/2 - n/2 = \Theta(n^2)$, which is slower than the $\Theta(n)$ running time of Horner's rule.

(c) Here is the loop invariant proof:

- **Initialization.** Prior to the first iteration of the for loop of lines 2–3, we have $y = 0$ and $i = n$. Replacing $i = n$ on the above loop invariant equation we have:

$$y = \sum_{k=0}^{n-n-1} a_{k+n+1} x^k = \sum_{k=0}^{-1} a_{k+n+1} x^k = 0,$$

which correctly corresponds to the initial value of y on line 1.

- **Maintenance.** In each iteration i of the loop, the previous value of y is multiplied by x and incremented by a_i (line 3). Performing these two operations on the above loop invariant equation for an iteration i, we have:

$$a_i + x \cdot \left(\sum_{k=0}^{n-i-1} a_{k+i+1} x^k \right) = a_i + \left(\sum_{k=0}^{n-i-1} a_{k+i+1} x^{k+1} \right) = a_i + \left(\sum_{k=1}^{n-i} a_{k+i} x^k \right) = \left(\sum_{k=0}^{n-i} a_{k+i} x^k \right),$$

which correctly corresponds to the loop invariant equation in the iteration $i - 1$ (next iteration, after iteration i).

- **Termination.** At termination, we have $i = -1$, so that:

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

(d) Since the loop invariant holds for all iterations and, at termination, the loop invariant corresponds exactly to the polynomial definition, we can assure that the code fragment correctly evaluates the polynomial characterized by the coefficients $a_0, a_1, \ldots, a_n$.

2-4 ***Inversions***

Let $A[1, \ldots, n]$ 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.

a. List the five inversions of the array $\langle 2, 3, 8, 6, 1 \rangle$.

b. What array with elements from the set {1, 2, . . . , n} has the most inversions? How many does it have?

c. What is the relationship between the running time of insertion sort and the number of inversions in the input array? Justify your answer.

d. Give an algorithm that determines the number of inversions in any permutation on n elements in $\Theta(n \lg n)$ worst-case time. (*Hint: modify merge sort.*)

(a) (1, 4), (1, 5), (2, 5), (3, 5), (4, 5).

(b) $\{n, n-1, n-2, \ldots, 2, 1\}$. It has $\binom{n}{2} = n(n-1)/2$ inversions.

(c) The number of operations of Insertion-Sort in an array A is the same as the number of inversions in A.

(d) The following pseudocode modifies Merge-Sort to count the number of inversions in $\Theta(n \lg n)$.

```
Inversions(A, p, r)
1 | inv = 0
2 | if p < r then
3 | |   q = ⌊(p + r)/2⌋
4 | |   inv = inv + Inversions(A, p, q) + Inversions(A, q + 1, r) + MergeInversions(A, p, q, r)
5 | return inv
```

```
MergeInversions(A, p, q, r)
1  | inv = 0
2  | n_1 = q − p + 1
3  | n_2 = r − q
4  | let L[1, ..., n_1 + 1] and R[1, ..., n_2 + 1] be new arrays
5  | for i = 1 to n_1 do
6  | |  L[i] = A[p + i − 1]
7  | for j = 1 to n_2 do
8  | |  R[j] = A[q + j]
9  | L[n_1 + 1] = ∞
10 | L[n_2 + 1] = ∞
11 | i = 1
12 | j = 1
13 | for k = p to r do
14 | |  if L[i] ≤ R[j] then
15 | |  |  i = i + 1
16 | |  else
17 | |  |  inv = inv + (n_1 − i + 1)
18 | |  |  j = j + 1
19 | return inv
```

Section 3.1 – Asymptotic notation

3.1-1 Let $f(n)$ and $g(n)$ be asymptotically nonnegative functions. Using the basic definition of Θ-notation, prove that $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.

Since $f(n)$ and $g(n)$ are both asymptotically nonnegative,

$$\exists\, n_0 \mid f(n) \geq 0\ g(n) \geq 0\ \forall\, n \geq n_0.$$

From the definition of $\Theta(\cdot)$, we have

$$\exists\, c_1\ c_2\ n_0 \in \mathbb{R}^+ \mid c_1 f(n) + c_1 g(n) \leq \max(f(n), g(n)) \leq c_2 f(n) + c_2 g(n)\ \forall\, n \geq n_0.$$

If $f(n) \geq g(n)$, we have

$$c_1 f(n) + c_1 g(n) \leq f(n) \leq c_2 f(n) + c_2 g(n).$$

The right-hand-side inequality is trivially satisfied with $c_2 = 1$. To find c_1, we notice that,

$$f(n) + g(n) \leq 2f(n),$$

and say,

$$c_1 = \frac{1}{2}.$$

The demonstration is similar for $g(n) > f(n)$, with $c_1 = 1/2$ and $c_2 = 1$.

3.1-2 Show that for any real constants a and b, where $b > 0$, $(n + a)^b = \Theta(n^b)$.

From the definition of $\Theta(\cdot)$, we have

$$\exists\, c_1\ c_2\ n_0 \in \mathbb{R}^+ \mid c_1 n^b \leq (n + a)^b \leq c_2 n^b\ \forall\, n \geq n_0,$$

and from the binomial theorem, we have

$$(n + a)^b = \binom{b}{0} n^b a^0 + \binom{b}{1} n^{b-1} a^1 + \cdots + \binom{b}{b-1} n^1 a^{b-1} + \binom{b}{b} n^0 a^b.$$

To find c_1, we notice that for n big enough,

$$\binom{b}{i} n^{b-i} a^i + \binom{b}{i+1} n^{b-(i+1)} a^{i+1} \geq 0 \quad \forall\, i \in 0, 2, \ldots, b,$$

which implies

$$\binom{b}{0} n^b a^0 + \binom{b}{1} n^{b-1} a^1 \leq (n + a)^b,$$

and also for n big enough,

$$\frac{n^b}{2} \leq n^b + \binom{b}{1} n^{b-1} a^1,$$

which implies

$$\frac{n^b}{2} \leq (n + a)^b,$$

and say

$$c_2 = \frac{1}{2}.$$

To find c_2, we notice that for n big enough,

$$n^b = \binom{b}{0} n^b a^0 \geq \binom{b}{i} n^{b-i} a^i \quad \forall\, i \in 1, \ldots, b,$$

which implies

$$(n + a)^b \leq (b + 1) n^b,$$

and say

$$c_2 = b + 1.$$

3.1-3 Explain why the statement, "The running time of algorithm A is at least $O(n^2)$," is meaningless.

Because the O-notation only bounds from the top, not from the bottom.

3.1-4 Is $2^{n+1} = O(2^n)$? Is $2^{2n} = O(2^n)$?

From the definition of $O(\cdot)$, we have

$$\exists\, c\ n_0 \in \mathbb{R}^+ \mid 0 \leq 2^{n+1} \leq c \cdot 2^n \ \ \forall\, n \geq n_0.$$

To find c, we notice that,

$$2^{n+1} = 2 \cdot 2^n,$$

and say $c = 2$ and $n_0 = 0$.
From the definition of $O(\cdot)$, we have

$$\exists\, c\ n_0 \in \mathbb{R}^+ \mid 0 \leq 2^{2n} \leq c \cdot 2^n \ \ \forall\, n \geq n_0.$$

To show that $2^{2n} \neq O(2^n)$, we notice that,

$$2^{2n} = 2^n \cdot 2^n,$$

which implies

$$c \geq 2^n,$$

which is not possible, since c is a constant and n is not.

3.1-5 Prove Theorem 3.1.

To prove

$$f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \wedge f(n) = \Omega(g(n)).$$

we need to show

$$f(n) = O(g(n)) \wedge f(n) = \Omega(g(n)) \to f(n) = \Theta(g(n)),$$

and

$$f(n) = \Theta(g(n)) \to f(n) = O(g(n)) \wedge f(n) = \Omega(g(n)).$$

From the definition of $O(\cdot)$, we have

$$\exists\, c_1\ n_1 \in \mathbb{R}^+ \mid 0 \leq f(n) \leq c_1 g(n) \ \ \forall\, n \geq n_1,$$

and from the definition of $\Omega(\cdot)$, we have

$$\exists\, c_2\ n_2 \in \mathbb{R}^+ \mid 0 \leq c_2 g(n) \leq f(n) \ \ \forall\, n \geq n_2,$$

which implies

$$\exists\, c_1\ c_2 \in \mathbb{R}^+\ n_0 = \max(n_1, n_2) \mid c_2 g(n) \leq f(n) \leq c_1 g(n) \ \ \forall\, n \geq n_0 \iff f(n) = \Theta(g(n)).$$

From the definition of $\Theta(\cdot)$, we have

$$\exists\, c_1\ c_2\ n_0 \in \mathbb{R}^+ \mid c_2 g(n) \leq f(n) \leq c_1 g(n) \ \ \forall\, n \geq n_0,$$

which implies

$$\exists\, c_1\ n_0 \in \mathbb{R}^+ \mid 0 \leq f(n) \leq c_1 g(n) \ \ \forall\, n \geq n_0 \iff f(n) = O(g(n)),$$

$$\exists\, c_2\ n_0 \in \mathbb{R}^+ \mid c_2 g(n) \leq f(n) \leq 0 \ \ \forall\, n \geq n_0 \iff f(n) = \Omega(g(n)).$$

3.1-6 Prove that the running time of an algorithm is $\Theta(g(n))$ if and only if its worst-case running time is $O(g(n))$ and its best-case running time is $\Omega(g(n))$.

Let $f_b(n)$ and $f_w(n)$ be the best and worst-case running times of algorithm A, respectivelly.
If the running time of A is $\Theta(g(n))$, we have

$$f_b(n) = \Theta(g(n)),$$

and

$$f_w(n) = \Theta(g(n)).$$

From Theorem 3.1,

$$f_b(n) = \Theta(g(n)) \iff f_b(n) = O(g(n)) \wedge f_b(n) = \Omega(g(n)),$$

and

$$f_w(n) = \Theta(g(n)) \iff f_w(n) = O(g(n)) \wedge f_w(n) = \Omega(g(n)).$$

3.1-7 Prove that $o(g(n)) \cap \omega(g(n))$ is the empty set.

From the definition of $o(\cdot)$, we have

$$o(g(n)) = \{f(n) : \forall\, c_1 > 0 \ \exists\, n_1 \in \mathbb{R}^+ \mid 0 \leq f(n) < c_1 g(n) \ \forall\, n \geq n_1\},$$

and from the definition of $\omega(\cdot)$, we have

$$\omega(g(n)) = \{f(n) : \forall\, c_2 > 0 \ \exists\, n_2 \in \mathbb{R}^+ \mid 0 \leq c_2 g(n) < f(n) \ \forall\, n \geq n_2\}.$$

Thus,

$$o(g(n)) \cap \omega(g(n)) = \{f(n) : \forall\, c_1 > 0 \ \forall\, c_2 > 0 \ \exists\, n_0 \in \mathbb{R}^+ \mid 0 \leq c_2 g(n) < f(n) < c_1 g(n) \ \forall\, n \geq n_2\},$$

which is the empty set since, for very large n, $f(n)$ cannot be less than $c_1 g(n)$ and greater than $c_2 g(n)$ for all $c_1, c_2 > 0$.

3.1-8 We can extend our notation to the case of two parameters n and m that can go to infinity independently at different rates. For a given $g(n, m)$, we denote by $O(g(n, m))$ the set of functions

$O(g(n, m)) = \{f(n, m) :$ there exist positive constants $c, n_0,$ and m_0 such that $0 \leq f(n, m) \leq cg(n, m)$ for all $n \geq n_0$ and $m \geq m_0\}$.

Give corresponding definitions for $\Omega(g(n, m))$ and $\Theta(g(n, m))$.

We denote by $\Omega(g(n, m))$ the set of functions

$$\Omega(g(n, m)) = \{f(n, m) : \exists\, c\ n_0\ m_0 \in \mathbb{R}^+ \mid 0 \leq cg(n, m)) \leq f(n, m) \ \forall\, n \geq n_0 \ \forall\, m \geq m_0\}.$$

We denote by $\Theta(g(n, m))$ the set of functions

$$\Theta(g(n, m)) = \{f(n, m) : \exists\, c_1\ c_2\ n_0\ m_0 \in \mathbb{R}^+ \mid 0 \leq c_1 g(n, m) \leq f(n, m) \leq c_2 g(n, m) \ \forall\, n \geq n_0 \ \forall\, m \geq m_0\}.$$

Section 3.2 – Standard notations and common functions

3.2-1 Show that if $f(n)$ and $g(n)$ are monotonically increasing functions, then so are the functions $f(n)+g(n)$ and $f(g(n))$, and if $f(n)$ and $g(n)$ are in addition nonnegative, then $f(n)\cdot g(n)$ is monotonically increasing.

If $f(n)$ and $g(n)$ are both monitonically increasing and $n \leq m$, we have

$$f(n) \leq f(m) \quad \text{and} \quad g(n) \leq g(m),$$

which implies that

$$f(n) - f(m) \leq 0 \quad \text{and} \quad g(n) - g(m) \leq 0.$$

Adding the above inequalities together, we have

$$f(n) - f(m) + g(n) - g(m) \leq 0 \rightarrow f(n) + g(n) \leq f(m) + g(m),$$

which shows that $f(n)+g(n)$ is monitonically increasing.
Also, let $g(n) = p$ and $g(m) = q$. Since $f(n) \leq f(m)$ and $g(n) \leq g(m)$, we have

$$f(p) \leq f(q) \rightarrow f(g(n)) \leq f(g(m)),$$

which shows that $f(g(n))$ is monitonically increasing.
If in addition, $f(\cdot) \geq 0$ and $g(\cdot) \geq 0$, we have

$$f(n) \leq f(m) \rightarrow f(n)g(n) \leq f(m)g(n) \rightarrow f(n)g(n) \leq f(m)g(m),$$

which shows that $f(n)\cdot g(n)$ is monitonically increasing.

3.2-2 Prove equation (3.16).

For all real $a > 0, b > 0, c > 0$,

$$a^{\log_b c} = a^{\frac{\log_a c}{\log_a b}} = \left(a^{\log_a c}\right)^{\frac{1}{\log_a b}} = c^{\frac{1}{\log_a b}} = c^{\log_b a}.$$

3.2-3 Prove equation (3.19). Also prove that $n! = \omega(2^n)$ and $n! = o(n^n)$.

Using the Stirling's approximation, we have

$$\begin{aligned}
\lg(n!) &\approx \lg\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\Theta\left(\frac{1}{n}\right)\right)\right) \\
&= \lg\left(\sqrt{2\pi n}\right) + \lg(\sqrt{n}) + \lg(n^n) - \lg(e^n) + \Theta(\lg(1/n)) \\
&= \Theta(1) + 1/2\lg(n) + n\lg n - n\lg e + \Theta(\lg(1/n)) \\
&= \Theta(1) + \Theta(\lg n) + \Theta(n\lg n) - \Theta(n) + \Theta(\lg(1/n)) \\
&= \Theta(n\lg n),
\end{aligned}$$

which proves Equation (3.19).
We have

$$n! = n\cdot(n-1)\cdot(n-2)\cdots 2\cdot 1 < \underbrace{n\cdot n\cdot n\cdots}_{\text{n times}} = n^n \ \ \forall\, n \geq 2,$$

which implies

$$n! = o(n^n).$$

We have

$$n! = n\cdot(n-1)\cdot(n-2)\cdots 2\cdot 1 > \underbrace{2\cdot 2\cdot 2\cdots}_{\text{n times}} = 2^n \ \ \forall\, n \geq 4,$$

which implies

$$n! = w(2^n).$$

3.2-4 ($\star$) Is the function $\lceil \lg n \rceil !$ polynomially bounded? Is the function $\lceil \lg \lg n \rceil !$ polynomially bounded?

A function $f(n)$ is polynomially bounded if there are constants c, k, n_0 such that for all $n \geq n_0$, $f(n) \leq cn^k$. Thus, $\lg(f(n)) \leq ck \lg n$.
We have

$$\lg(\lceil \lg n \rceil !) = \Theta(\lceil \lg n \rceil \lg(\lceil \lg n \rceil)) = \Theta(\lg n \lg \lg n) = w(\lg n),$$

which implies that $\lg(\lceil \lg n \rceil !) > ck \lg n$, i.e., $\lceil \lg n \rceil !$ is not polynomially bounded.
We have

$$\lg(\lceil \lg \lg n \rceil !) = \Theta(\lceil \lg \lg n \rceil \lg \lceil \lg \lg n \rceil) = \Theta(\lg \lg n \lg \lg \lg n) = o(\lg^2 \lg n) = o(\lg^2 n) = o(\lg n),$$

which implies that $\lg(\lceil \lg \lg n \rceil !) \leq ck \lg n, i.e., \lceil \lg \lg n \rceil !$ is polynomially bounded.

3.2-5 ($\star$) Which is asymptotically larger: $\lg(\lg^\star n)$ or $\lg^\star(\lg n)$?

Let's assume that $\lg^*(x) = k$.
We have

$$\lg(\lg^* x) = \lg k,$$

and

$$\lg^*(\lg x) = k - 1,$$

since the inner logarithm that is applied to x will reduce the number of iterations of the iterative logarithm by 1.
Thus, since $(k-1)$ is asymptotically larger than $\lg(k)$, $\lg^*(\lg x)$ is also asymptotically larger than $\lg(\lg^* x)$.

3.2-6 Show that the golden ration ϕ and its conjugate $\hat{\phi}$ both satisfy the equation $x^2 = x + 1$.

The demonstration follows directly from the formulas of ϕ and $\hat{\phi}$.

$$\phi^2 = \left(\frac{1+\sqrt{5}}{2}\right)^2 = \frac{1+2\sqrt{5}+5}{4} = \frac{2\sqrt{5}+6}{4} = \frac{\sqrt{5}+3}{2} = \frac{1+\sqrt{5}}{2} + 1 = \phi + 1.$$

$$\hat{\phi}^2 = \left(\frac{1-\sqrt{5}}{2}\right)^2 = \frac{1-2\sqrt{5}+5}{4} = \frac{6-2\sqrt{5}}{4} = \frac{3-\sqrt{5}}{2} = \frac{1-\sqrt{5}}{2} + 1 = \hat{\phi} + 1.$$

3.2-7 Prove by induction that the ith Fibonacci number satisfies the equality

$$F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}},$$

where ϕ is the golden ratio and $\hat{\phi}$ is its conjugate.

We have that

$$F_0 = \frac{\phi^0 - \hat{\phi}^0}{\sqrt{5}} = \frac{1-1}{\sqrt{5}} = 0,$$

and

$$F_1 = \frac{\phi^1 - \hat{\phi}^1}{\sqrt{5}} = \frac{1+\sqrt{5}-1+\sqrt{5}}{2\sqrt{5}} = \frac{2\sqrt{5}}{2\sqrt{5}} = 1.$$

which are the correct Fibonacci values for $i = 0$ and $i = 1$. Then we have the inductive step:

$$\begin{aligned}
F_i + F_{i+1} &= \frac{\phi^i + \hat{\phi}^i}{\sqrt{5}} + \frac{\phi^{i+1} + \hat{\phi}^{i+1}}{\sqrt{5}} \\
&= \frac{\phi^i + \phi^{i+1} - (\hat{\phi}^i + \hat{\phi}^{i+1})}{\sqrt{5}} \\
&= \frac{\phi^i(1+\phi) - \hat{\phi}^i(1+\phi)}{\sqrt{5}} \\
&= \frac{\phi^i\phi^2 - \hat{\phi}^i\hat{\phi}^2}{\sqrt{5}} \\
&= \frac{\phi^{i+2} - \hat{\phi}^{i+2}}{\sqrt{5}} = F_{i+2}.
\end{aligned}$$

3.2-8 Show that $k \ln k = \Theta(n)$ implies $k = \Theta(n/\ln n)$.

From the symmetry of Θ, we have

$$k \ln k = \Theta(n) \to n = \Theta(k \ln k),$$

and

$$\ln n = \Theta(\ln(k \ln k)) = \Theta(\ln k \ln \ln k) = \Theta(\ln k).$$

Thus,

$$\frac{n}{\ln n} = \frac{\Theta(k \ln k)}{\Theta(\ln k \ln \ln k)} = \Theta\left(\frac{k \ln k}{\ln k \ln \ln k}\right) = \Theta(k),$$

which implies

$$k = \Theta\left(\frac{n}{\ln n}\right).$$

Problems

Skipped for later.

Section 4.1 – The maximum-subarray problem

4.1-1 What does FIND-MAXIMUM-SUBARRAY return when all elements of A are negative?

A subarray with only the largest negative element of A.

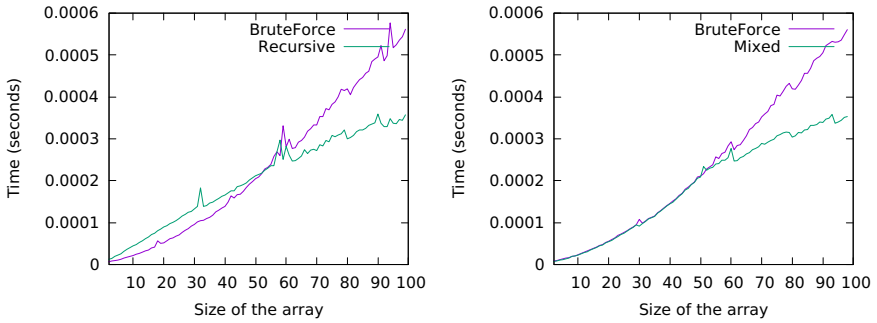
4.1-2 Write pseudocode for the brute-force method of solving the maximum-subarray problem. Your procedure should run in $\Theta(n^2)$ time.

The pseudocode is stated below.

```
   FindMaximumSubarray-BruteForce(A)
1  | low = 0
2  | high = 0
3  | sum = −∞
4  | for i = 1 to A.length do
5  |   | cursum = 0
6  |   | for j = i to A.length do
7  |   |   | cursum = cursum + A[j]
8  |   |   | if cursum > sum then
9  |   |   |   | sum = cursum
10 |   |   |   | low = i
11 |   |   |   | high = j
12 | return low, high, sum
```

4.1-3 Implement both the brute-force and recursive algorithms for the maximum-subarray problem on your own computer. What problem size n_0 gives the crossover point at which the recursive algorithm beats the brute-force algorithm? Then, change the base case of the recursive algorithm to use the brute-force algorithm whenever the problem size is less than n_0. Does that change the crossover point?

Figure below in the lhs ilustrates the crossover point between the BruteForce and Recursive solutions in my machine. In that comparison, $n_0 \approx 52$. Figure below in the rhs ilustrates the crossover point between the BruteForce and Mixed solutions in my machine. The crossover point does not change but the Mixed solution becomes as fast as the BruteForce solution when the problem size is lower than 52.



4.1-4 Suppose we change the definition of the maximum-subarray problem to allow the result to be an empty subarray, where the sum of the values of an empty subarray is 0. How would you change any of the algorithms that do not allow empty subarrays to permit an empty subarray to be the result?

The BruteForce algorithm (stated above in Question 4.1-2) can be updated just by modifying line 3 to $sum = 0$, instead of $sum = -\infty$. In that case, if there is no subarray whose sum is greater than zero, the algorithm will return a invalid subarray ($low = 0, high = 0, sum = 0$), which will denote the empty subarray.

The Recursive algorithm (stated in Section 4.1) can be updated as follows. In the FIND-MAX-CROSSING-SUBARRAY routine, update lines 1 and 8 to initialize *left-sum* and *right-sum* to 0, instead of $-\infty$. Also initialize *max-left* (after line 1) and *max-right* (after line 8) to 0. In the FIND-MAXIMUM-SUBARRAY routine, surround the return statement of line 2 with a conditional that verifies if $A[low]$ is greater than zero. If it is, return the values as it was before. If it is not, return a invalid subarray (denoted by $low = 0$ and $high = 0$) and the sum as zero.

4.1-5 Use the following ideas to develop a nonrecursive, linear-time algorithm for the maximum-subarray problem. Start at the left end of the array, and progress toward the right, keeping track of the maximum subarray seen so far. Knowing a maximum subarray of $A[1,\ldots,j]$, extend the answer to find a maximum subarray ending at index $j+1$ by using the following observation: a maximum subarray of $A[1,\ldots,j+1]$ is either a maximum subarray of $A[1,\ldots,j]$ or a subarray $A[i,\ldots,j+1]$, for some $1 \leq i \leq j+1$. Determine a maximum subarray of the form $A[i,\ldots,j+1]$ in constant time based on knowing a maximum subarray ending at index j.

The pseudocode is stated below.

```
   FindMaximumSubarray-Linear(A)
1  | low = 0
2  | high = 0
3  | sum = 0
4  | current-low = 0
5  | current-sum = 0
6  | for i = 1 to A.length do
7  | | current-sum = max(A[i], current-sum + A[i])
8  | | if current-sum == A[i] then
9  | | | current-low = i
10 | | if current-sum > sum then
11 | | | low = current-low
12 | | | high = i
13 | | | sum = current-sum
14 | return low, high, sum
```

We can make it a little faster (twice as fast on my machine) by avoiding executing lines 7, 8, and 10 when not necessary.

```
   FindMaximumSubarray-Linear-Optimized(A)
1  | low = 0
2  | high = 0
3  | sum = 0
4  | current-low = 0
5  | current-sum = 0
6  | for i = 1 to A.length do
7  | | if current-sum + A[i] ≤ 0 then
8  | | | current-sum = 0
9  | | else
10 | | | current-sum = current-sum + A[i]
11 | | | if current-sum == A[i] then
12 | | | | current-low = i
13 | | | if current-sum > sum then
14 | | | | low = current-low
15 | | | | high = i
16 | | | | sum = current-sum
17 | return low, high, sum
```

Section 4.2 – Strassen's algorithm for matrix multiplication

4.2-1 Use Strassen's algorithm to compute the matrix product

$$\begin{bmatrix} 1 & 3 \\ 7 & 5 \end{bmatrix} \begin{bmatrix} 6 & 8 \\ 4 & 2 \end{bmatrix}.$$

Show your work.

Let

$$A = \begin{bmatrix} 1 & 3 \\ 7 & 5 \end{bmatrix}, B = \begin{bmatrix} 6 & 8 \\ 4 & 2 \end{bmatrix},$$

and $C = A \cdot B$. To compute C using Strassen's algorithm, we start by computing the S_i matrices:

$$\begin{aligned}
S_1 &= B_{12} - B_{22} = 8 - 2 = 6, \\
S_2 &= A_{11} + A_{12} = 1 + 3 = 4, \\
S_3 &= A_{21} + A_{22} = 7 + 5 = 12, \\
S_4 &= B_{21} - B_{11} = 4 - 6 = -2, \\
S_5 &= A_{11} + A_{22} = 1 + 5 = 6, \\
S_6 &= B_{11} + B_{22} = 6 + 2 = 8, \\
S_7 &= A_{12} + A_{22} = 3 - 5 = -2, \\
S_8 &= B_{21} + B_{22} = 4 + 2 = 6, \\
S_9 &= A_{11} - A_{21} = 1 - 7 = -6, \\
S_{10} &= B_{11} + B_{12} = 6 + 8 = 14.
\end{aligned}$$

Then we compute the P_i matrices:

$$\begin{aligned}
P_1 &= A_11 \cdot S_1 = 1 \cdot 6 = 6, \\
P_2 &= S_2 \cdot B_22 = 4 \cdot 2 = 8, \\
P_3 &= S_3 \cdot B_11 = 12 \cdot 6 = 72, \\
P_4 &= A_22 \cdot S_4 = 5 \cdot (-2) = -10, \\
P_5 &= S_5 \cdot S_6 = 6 \cdot 8 = 48, \\
P_6 &= S_7 \cdot S_8 = (-2) \cdot 6 = -12, \\
P_7 &= S_9 \cdot S_10 = (-6) \cdot 14 = -84.
\end{aligned}$$

Using matrices S_i and P_i, we compute C:

$$C = \begin{bmatrix} (P_5 + P_4 - P_2 + P_6) & (P_2 + P_2) \\ (P_3 + P_4) & (P_5 + P_1 - P_3 - P_7) \end{bmatrix} = \begin{bmatrix} 18 & 14 \\ 62 & 66 \end{bmatrix}.$$

4.2-2 Write pseudocode for Strassen's algorithm.

The pseudocode is stated below.

```
Square-Matrix-Multiply-Strassen(A, B)
 1  n = A.rows
 2  let C be a new n × n matrix
 3  if n == 1 then
 4      c_11 = a_11 · b_11
 5  else
 6      partition A, B, and C as into n/2 × n/2 submatrices
 7      let S_1, S_2, ..., S_10 be new n/2 × n/2 matrices
 8      let P_1, P_2, ..., P_7 be new n/2 × n/2 matrices
 9      S_1 = B_12 − B_22
10      S_2 = A_11 + A_12
11      S_3 = A_21 + A_22
12      S_4 = B_21 − B_11
13      S_5 = A_11 + A_22
14      S_6 = B_11 + B_22
15      S_7 = A_12 − A_22
16      S_8 = B_21 + B_22
17      S_9 = A_11 − A_21
18      S_10 = B_11 − B_12
19      P_1 = Square-Matrix-Multiply-Strassen(A_11, S_1)
20      P_2 = Square-Matrix-Multiply-Strassen(S_2, B_22)
21      P_3 = Square-Matrix-Multiply-Strassen(S_3, B_11)
22      P_4 = Square-Matrix-Multiply-Strassen(A_22, S_4)
23      P_5 = Square-Matrix-Multiply-Strassen(S_5, S_6)
24      P_6 = Square-Matrix-Multiply-Strassen(S_7, S_8)
25      P_7 = Square-Matrix-Multiply-Strassen(S_9, S_10)
26      C_11 = P_5 + P_4 − P_2 + P_6
27      C_12 = P_1 + P_2
28      C_21 = P_3 + P_4
29      C_22 = P_5 + P_1 − P_3 − P_7
30  return C
```

4.2-3 How would you modify Strassen's algorithm to multiply $n \times n$ matrices in which n is not an exact power of 2? Show that the resulting algorithm runs in time $\Theta(n^{\lg 7})$.

Pad each input $n \times n$ matrix (rows and columns) with $m - n$ zeros, resulting in an $m \times m$ matrix, where $m = 2^{\lceil \lg n \rceil}$. After computing the final matrix, cut down the last $m - n$ rows and $m - n$ columns (which will be zeros).

Padding the matrix with zeros is done once, in the root of the recursion tree, and takes $O(m^2)$. Since we now have an $m \times m$ matrix, the algorithm runs in $\Theta(m^{\lg 7}) + O(m^2) = \Theta(m^{\lg 7})$. We have that $n \leq m < 2^{(\lg n)+1} = 2^{\lg n} \cdot 2 = 2n$. Thus, the algorithm runs in $\Theta((2n)^{\lg 7}) = \Theta(n^{\lg 7})$.

4.2-4 What is the largest k such that if you can multiply 3×3 matrices using k multiplications (not assuming commutativity of multiplication), then you can multiply $n \times n$ matrices in time $o(n^{\lg 7})$? What would the running time of this algorithm be?

If we modify the Square-Matrix-Multiply-Recursive algorithm to partition the matrices into $n/3 \times n/3$ submatrices, we would have the following recurrence:

$$T(n) = \Theta(1) + 27T(n/3) + \Theta(n^2) = 27T(n/3) + \Theta(n^2).$$

Let's proceed to understand a little more about the above recurrence. Let A and B be the two input matrices in each node of the above recursion tree. Like in the original Square-Matrix-Multiply-Recursive algorithm, our modified version will take $\Theta(1)$ to partition A and B into $n/3 \times n/3$ submatrices. In each node of the tree, the product of A and B is recursively computed by the products of their submatrices. Since the number of recursive (submatrices) products to compute $A \cdot B$ in each node of the recurstion tree is 27 and each of these submatrices is 3 times smaller than A and B, the 27 recursive products takes $27T(n/3)$. Finally, the number of summations to compute the final matrix is $\Theta(3 \cdot 9 \cdot n^2/3) = \Theta(n^2)$.

If after partitioning A and B into $n/3 \times n/3$ submatrices we can compute their product with k multiplications (instead of 27), we would have the following recurrence:

$$T(n) = \Theta(1) + kT(n/3) + \Theta(n^2) = kT(n/3) + \Theta(n^2),$$

We can solve the above recurrence using the master method. We have $f(n) = n^2$ and $n^{log_b a} = n^{\log_3 k}$. Using the first case of the master method, we have

$$\forall\, k \mid \log_3 k > 2,\ n^2 = O(n^{(\log_3 k)-\epsilon}),\ 0 \leq \epsilon \leq \log_3 k - 2,$$

which implies

$$T(n) = \Theta(n^{\log_3 k}).$$

Since $\log_3 21 < \lg 7 < \log_3 22$, the largest value for k is 21. Its running time would be $n^{\log_3 21} \approx n^{2.7712}$.

4.2-5 V. Pan has discovered a way of multiplying 68×68 matrices using 132,464 multiplications, a way of multiplying 70×70 matrices using 143,640 multiplications, and a way of multiplying 72×72 matrices using 155,424 multiplications. Which method yields the best asymptotic running time when used in a divide-and-conquer matrix-multiplication algorithm? How does it compare to Strassen's algorithm?

The algorithms would take:

- $n^{\log_{68} 132{,}464} \approx n^{2.795128}$,
- $n^{\log_{70} 143{,}640} \approx n^{2.795122}$,
- $n^{\log_{72} 155{,}424} \approx n^{2.795147}$.

The fastest is the one that multiplies 70×70 matrices, but all of them are faster then the Strassen's algorithm.

4.2-6 How quickly can you multiply a $kn \times n$ matrix by an $n \times kn$ matrix, using Strassen's algorithm as a subroutine? Answer the same question with the order of the input matrices reversed.

Let A and B be $kn \times n$ and $n \times kn$ matrices, respectivelly. We can compute $A \cdot B$ as follows:

(a) Partition A and B into k submatrices $A_1, \ldots, A_k$ and $B_1, \ldots, B_k$, each one of size $n \times n$.

(b) Compute the desired submatrices C_{ij} of the result matrix C by the product of $A_i \cdot B_j$. Use the Strassen's algorithm to compute each of those products.

Since each of the k^2 products takes $\Theta(n^{\lg 7})$, this algorithm runs in $\Theta(k^2 n^{\lg 7})$.

We can compute $B \cdot A$ as follows:

(a) Partition A and B into k submatrices $A_1, \ldots, A_k$ and $B_1, \ldots, B_k$, each one of size $n \times n$.

(b) Compute the the result matrix $C = \sum_{i=1}^{k} A_i \cdot B_i$. Use the Strassen's algorithm to compute each of those products.

Since each of the k products takes $\Theta(n^{\lg 7})$ and the $k-1$ summations takes $\Theta((k-1)n^2/k) = O(n^2)$, this algorithm runs in $\Theta(kn^{\lg 7}) + O(n^2) = \Theta(kn^{\lg 7})$.

4.2-7 Show how to multiply the complex numbers $a + bi$ and $c + di$ using only three multiplications of real numbers. The algorithm should take a, b, c, and d as input and produce the real component $ac - bd$ and the imaginary component $ad + bc$ separately.

The pseudocode is stated below.

```
Complex-Product(a, b, c, d)
1 | x = a · c
2 | y = b · d
3 | real-part = x − y
4 | imaginary-part = (a + b) · (c + d) − x − y
5 | return real-part, imaginary-part
```

Section 4.3 – The substitution method for solving recurrences

4.3-1 Show that the solution of $T(n) = T(n-1) + n$ is $O(n^2)$.

Our guess is

$$T(n) \leq cn^2 \ \forall\, n \geq n_0,$$

where c and n_0 are positive constants. Substituting into the recurrence yields

$$\begin{aligned} T(n) &\leq c(n-1)^2 + n \\ &= cn^2 - 2cn + c + n \quad (c = 1) \\ &= n^2 - 2n + n + 1 \\ &= n^2 - n + 1 \\ &\leq n^2, \end{aligned}$$

where the last step holds as long as $n_0 \geq 1$.

4.3-2 Show that the solution of $T(n) = T(\lceil n/2 \rceil) + 1$ is $O(\lg n)$.

Our guess is

$$T(n) \leq c \lg n - d \ \forall\, n \geq n_0,$$

where c, d, and n_0 are positive constants. Substituting into the recurrence yields

$$\begin{aligned} T(n) &\leq c \lg(\lceil n/2 \rceil) - d + 1 \\ &\leq c \lg n - d + 1 \\ &\leq c \lg n, \end{aligned}$$

where the last step holds as long as $d \geq 1$.

4.3-3 We saw that the solution of $T(n) = 2T(\lfloor n/2 \rfloor) + n$ is $O(n \lg n)$. Show that the solution of this recurrence is also $\Omega(n \lg n)$. Conclude that the solution is $\Theta(n \lg n)$.

Our guess is

$$T(n) \geq cn \lg n \ \forall\, n \geq n_0,$$

where c and n_0 are positive constants. Substituting into the recurrence yields

$$\begin{aligned} T(n) &\geq 2(c\lfloor n/2 \rfloor \lg \lfloor n/2 \rfloor) + n \\ &\geq 2c(n/4) \lg(n/4) + n \\ &= c(n/2) \lg n - c(n/2) \lg 4 + n \\ &= c(n/2) \lg n - cn + n \\ &\geq cn \lg n, \end{aligned}$$

where the last step holds as long as $c \leq 1$.
Thus, we have

$$c_1 n \lg n \leq T(n) \leq c_2 n \lg n,$$

with $c_1 \leq 1$ and $c_2 \geq 1$, which implies

$$T(n) = \Theta(n \lg n).$$

4.3-4 Show that by making a different inductive hypothesis, we can overcome the difficulty with the boundary condition $T(1) = 1$ for recurrence (4.19) without adjusting the boundary conditions for the inductive proof.

Our new guess is

$$T(n) \leq cn \lg n + n \ \forall n \geq n_0,$$

where c, d, and n_0 are positive constants. Substituting into the recurrence yields

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor \lg \lfloor n/2 \rfloor + \lfloor n/2 \rfloor) + n \\ &\leq cn \lg(n/2) + 2(n/2) + n \\ &= cn \lg n - cn \lg 2n + 2n \\ &= cn \lg n - cn + 2n \\ &\leq cn \lg n + n, \end{aligned}$$

where the last step holds as long as $c \geq 1$.
Now on the boundary condition, we have

$$T(1) \leq c(n \lg n) + n = c1 \lg 1 + 1 = 0 + 1 = 1.$$

4.3-5 Show that $\Theta(n \lg n)$ is the solution to the "exact" recurrence (4.3) for merge sort.

First, we verify if (4.3) is $O(n \lg n)$. Our guess is

$$T(n) \leq c(n-d) \lg(n-d) \ \forall n \geq n_0,$$

where c, d, and n_0 are positive constants. Substituting into the recurrence yields

$$\begin{aligned} T(n) &\leq c(\lceil n/2 \rceil - d) \lg(\lceil n/2 \rceil - d) + c(\lfloor n/2 \rfloor - d) \lg(\lfloor n/2 \rfloor - d) + en \\ &\leq c(n/2 + 1 - d) \lg(n/2 + 1 - d) + c(n/2 - d) \lg(n/2 - d) + en \quad (d \geq 2) \\ &\leq c\left(\frac{n-d}{2}\right) \lg\left(\frac{n-d}{2}\right) + c\left(\frac{n-d}{2}\right) \lg\left(\frac{n-d}{2}\right) + en \\ &= c(n-d) \lg\left(\frac{n-d}{2}\right) + en \\ &= c(n-d) \lg(n-d) - c(n-d) + en \\ &= c(n-d) \lg(n-d) - cn + en + cd \\ &\leq c(n-d) \lg(n-d), \end{aligned}$$

where the last step holds as long as $c > e$ and $n_0 \geq cd$.
Then we verify if (4.3) is $\Omega(n \lg n)$. Our guess is

$$T(n) \geq c(n+d) \lg(n+d) \ \forall n \geq n_0,$$

where c, d, and n_0 are positive constants. Substituting into the recurrence yields

$$\begin{aligned} T(n) &\geq c(\lceil n/2 \rceil + d) \lg(\lceil n/2 \rceil + d) + c(\lfloor n/2 \rfloor + d) \lg(\lfloor n/2 \rfloor + d) + en \\ &\geq c(n/2 + d) \lg(n/2 + d) + c(n/2 - 1 + d) \lg(n/2 - 1 + d) + en \quad (d \geq 2) \\ &\geq c\left(\frac{n+d}{2}\right) \lg\left(\frac{n+d}{2}\right) + c\left(\frac{n+d}{2}\right) \lg\left(\frac{n+d}{2}\right) + en \\ &= c(n+d) \lg\left(\frac{n+d}{2}\right) + en \\ &= c(n+d) \lg(n+d) - c(n+d) + en \\ &= c(n+d) \lg(n+d) - cn + en - cd \\ &\geq c(n+d) \lg(n+d), \end{aligned}$$

where the last step holds as long as $e > c$ and $n_0 \geq cd$.

4.3-6 Show that the solution to $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$ is $O(n \lg n)$.

Our guess is

$$T(n) \le c(n-d)\lg(n-d) \ \forall n \ge n_0,$$

where c, d, and n_0 are positive constants. Substituting into the recurrence yields

$$\begin{aligned}
T(n) &\le 2c(\lfloor n/2 \rfloor - d + 17)\lg(\lfloor n/2 \rfloor - d + 17) + n \\
&\le 2c(n/2 - d + 17)\lg(n/2 - d + 17) + n \qquad (d \ge 34) \\
&\le 2c\left(\frac{n-d}{2}\right)\lg\left(\frac{n-d}{2}\right) + n \\
&= c(n-d)\lg\left(\frac{n-d}{2}\right) + n \\
&= c(n-d)\lg(n-d) - c(n-d) + n \\
&= c(n-d)\lg(n-d) - cn + n + cd \\
&\le c(n-d)\lg(n-d),
\end{aligned}$$

where the last step holds as long as $c \ge 2$ and $n_0 \ge cd$.

4.3-7 Using the master method in Section 4.5 you can show that the solution to the recurrence $T(n) = 4T(n/3) + n$ is $T(n) = \Theta(n^{\log_3 4})$. Show that a substitution proof with the assumption $T(n) \le cn^{\log_3 4}$ fails. Then show how to subtract off a lower-order term to make a substitution proof work.

The initial guess is

$$T(n) \le cn^{\log_3 4} \ \forall n \ge n_0,$$

where c, and n_0 are positive constants. Substituting into the recurrence yields

$$\begin{aligned}
T(n) &\le 4c\left(\frac{n}{3}\right)^{\log_3 4} + n \\
&= 4c\frac{n^{\log_3 4}}{4} + n \\
&= cn^{\log_3 4} + n
\end{aligned}$$

which does not imply $T(n) \le cn^{\log_3 4}$ for any choice of c.

Our new guess is

$$T(n) \le cn^{\log_3 4} - dn \ \forall n \ge n_0,$$

where c, d, and n_0 are positive constants. Substituting into the recurrence yields

$$\begin{aligned}
T(n) &\le 4\left(c\left(\frac{n}{3}\right)^{\log_3 4} - d\frac{n}{3}\right) + n \\
&= 4c\frac{n^{\log_3 4}}{4} - 4d\frac{n}{3} + n \\
&\le cn^{\log_3 4},
\end{aligned}$$

where the last step holds as long as $d \ge 3/4$.

4.3-8 Using the master method in Section 4.5, you can show that the solution to the recurrence $T(n) = 4T(n/2) + n$ is $T(n) = \Theta(n^2)$. Show that a substitution proof with the assumption $T(n) \le cn^2$ fails. Then show how to subtract off a lower-order term to make a substitution proof work.

The initial guess is

$$T(n) \le cn^2 \ \forall n \ge n_0,$$

where c, and n_0 are positive constants. Substituting into the recurrence yields

$$\begin{aligned}
T(n) &\le 4c\left(\frac{n}{2}\right)^2 + n \\
&= cn^2 + n
\end{aligned}$$

which does not imply $T(n) \le cn^2$ for any choice of c.

Our new guess is

$$T(n) \leq cn^2 - dn \ \forall\, n \geq n_0,$$

where c, d, and n_0 are positive constants. Substituting into the recurrence yields

$$\begin{aligned} T(n) &\leq 4\left(c\left(\frac{n}{2}\right)^2 - d\frac{n}{2}\right) + n \\ &= cn^2 - 2dn + n \\ &\leq cn^2, \end{aligned}$$

where the last step holds as long as $d \geq 1/2$.

4.3-9 Solve the recurrence $T(n) = 3T(\sqrt{n}) + \log n$ by making a change of variables. Your solution should be asymptotically tight. Do not worry about whether values are integral.

Renaming $m = \log n$ yields

$$T(10^m) = 3T(10^{m/2}) + m.$$

Now renaming $S(m) = T(2^m)$ yields

$$S(m) = 3S(m/2) + m.$$

With the master method, we have $f(n) = m = \log n$ and $n^{\log_b a} = n^{\lg 3} \approx n^{1.585}$. Using the first case, we have

$$f(n) = \log n = O(n^{\lg 3 - \epsilon}),\ (\epsilon = 0.5)$$

which implies

$$S(m) = \Theta(m^{\lg 3}).$$

We can double-check if $S(m) = O(m^{\lg 3})$ using the substition method. Our guess is

$$S(m) \leq cm^{\lg 3} - dm \ \forall\, m \geq m_0,$$

where c, d, and n_0 are positive constants. Substituting into the recurrence yields

$$\begin{aligned} T(n) &\leq 3\left(c\left(\frac{m}{2}\right)^{\lg 3} - d\frac{m}{2}\right) + m \\ &= 3c\frac{m^{\lg 3}}{3} - 3d\frac{m}{2} + m \\ &\leq cm^{\lg 3} + dm \end{aligned}$$

where the last step holds as long as $d \geq 2/3$.
Now verifying if $S(m) = \Omega(m^{\lg 3})$ with the substitution method. Our guess is

$$S(m) \geq cm^{\lg 3} \ \forall\, m \geq m_0,$$

where c, and n_0 are positive constants. Substituting into the recurrence yields

$$\begin{aligned} T(n) &\geq 3c\left(\frac{m}{2}\right)^{\lg 3} + m \\ &= 3c\frac{m^{\lg 3}}{3} + m \\ &\geq cm^{\lg 3}. \end{aligned}$$

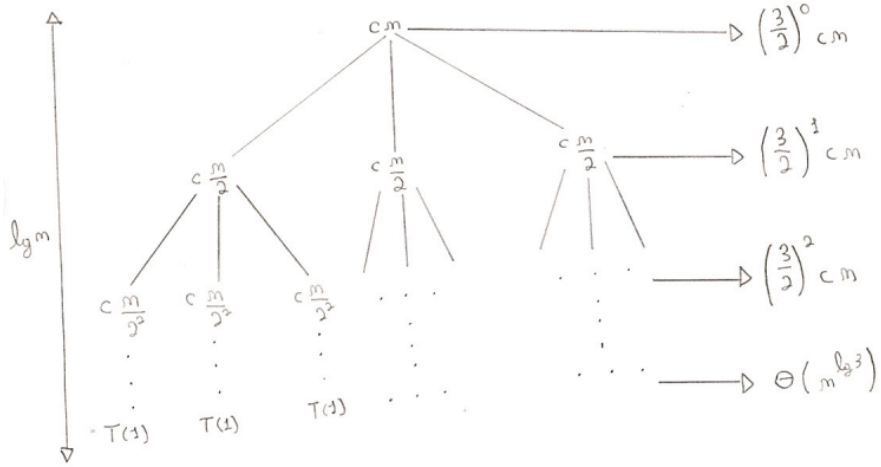
Finally, we have

$$T(n) = T(10^m) = S(m) = \Theta(m^{\lg 3}) = \Theta(\log^{\lg 3} n).$$

Section 4.4 – The recursion-tree method for solving recurrences

4.4-1 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.

Since floors/ceiling usually do not matter, we will draw a recursion tree for the recurrence $T(n) = 3T(n/2) + n$.



The number of nodes at depth i is 3^i. Since subproblem size reduce by a factor of 2, each node at depth i, for $i = 0, 1, 2, \ldots, \lg n - 1$, has a cost of $c(n/2^i)$. Thus, the total cost over all nodes at depth i, for $i = 0, 1, 2, \ldots, \lg n - 1$, is $(3/2)^i cn$. The bottom level, at deph $\lg n$, has $3^{\lg n} = n^{\lg 3}$ nodes, each contributing cost $T(1)$, for a total cost of $n^{\lg 3} T(1) = \Theta(n^{\lg 3})$. The cost of the entire tree is

$$\begin{aligned}
T(n) &= cn + \frac{3}{2}cn + \left(\frac{3}{2}\right)^2 cn + \cdots + \left(\frac{3}{2}\right)^{\lg n - 1} cn + \Theta\left(n^{\lg 3}\right) \\
&= \sum_{i=0}^{\lg n - 1} \left(\frac{3}{2}\right)^i cn + \Theta(n^{\lg 3}) \\
&= cn \frac{\left(\frac{3}{2}\right)^{\lg n} - 1}{\frac{3}{2} - 1} + \Theta(n^{\lg 3}) \\
&= 2cn \left(\left(\frac{3}{2}\right)^{\lg n} - 1\right) + \Theta(n^{\lg 3}) \\
&= 2cn \frac{3^{\lg n}}{2^{\lg n}} - 2cn + \Theta(n^{\lg 3}) \\
&= 2cn \frac{n^{\lg 3}}{n} - 2cn + \Theta(n^{\lg 3}) \\
&= 2cn^{\lg 3} - 2cn + \Theta(n^{\lg 3}) \\
&= O(n^{\lg 3}).
\end{aligned}$$

Our guess is

$$T(n) \leq cn^{\lg 3} - dn \ \ \forall n \geq n_0,$$

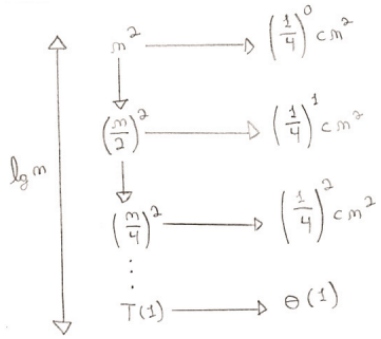
where c, d, and n_0 are positive constants. Substituting into the recurrence yields

$$\begin{aligned}
T(n) &\leq 3\left(c\left\lfloor \frac{n}{2} \right\rfloor^{\lg 3} - d\left\lfloor \frac{n}{2} \right\rfloor\right) + n \\
&\leq \frac{3c}{3} n^{\lg 3} - \frac{3d}{2} n + n \\
&= cn^{\lg 3} - dn - \frac{d}{2} n + n \\
&\leq cn^{\lg 3} - dn,
\end{aligned}$$

where the last step holds as long as $d \geq 2$.

4.4-2 Use a recursion tree to determine a good asymptotic upper bound on the recurrence $T(n) = T(n/2) + n^2$. Use the substitution method to verify your answer.

Figure below ilustrates the recursion tree $T(n) = T(n/2) + n^2$.



The tree has $\lg n$ levels and the cost at depth i is $c(n/2^i)^2 = (1/4)^i cn^2$.
The cost of the entire tree is

$$\begin{aligned} T(n) &= \sum_{i=0}^{\lg n} \left(\frac{1}{4}\right)^i cn^2 \\ &< \sum_{i=0}^{\infty} \left(\frac{1}{4}\right)^i cn^2 \\ &= \frac{1}{1-(1/4)} cn^2 \\ &= \frac{4}{3} cn^2 \\ &= O(n^2). \end{aligned}$$

Our guess is

$$T(n) \leq dn^2 \ \forall n \geq n_0,$$

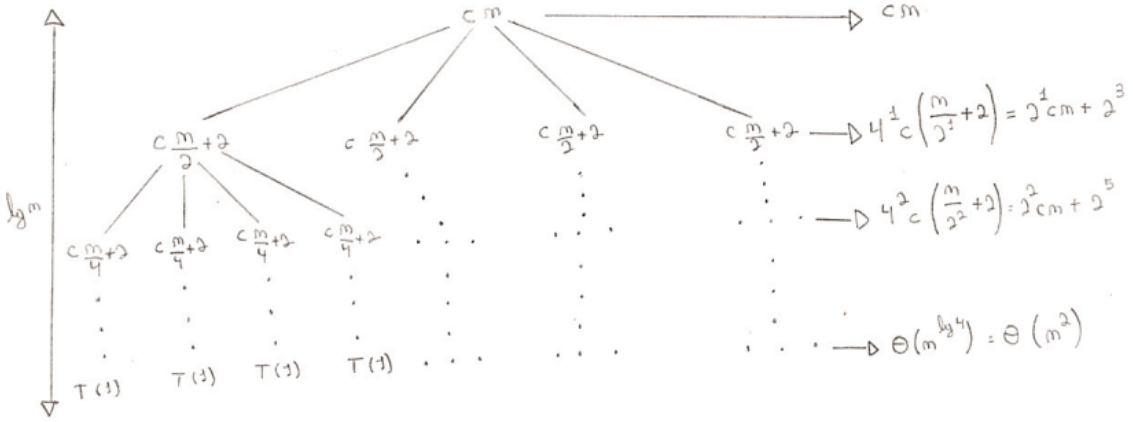
where d, and n_0 are positive constants. Substituting into the recurrence and using the same constant $c > 0$ as before yields

$$\begin{aligned} T(n) &\leq d\left(\frac{n}{2}\right)^2 + cn^2 \\ &= \frac{1}{4} dn^2 + cn^2 \\ &\leq dn^2, \end{aligned}$$

where the last step holds as long as $d \geq (4/3)c$.

4.4-3 Use a recursion tree to determine a good asymptotic upper bound on the recurrence $T(n) = 4T(n/2+2)+n$. Use the substitution method to verify your answer.

Figure below ilustrates the recursion tree $T(n) = 4T(n/2+2)+n$.



The number of nodes at depth i is 4^i. Since subproblem size reduce by a factor of 2 and increment 2, each node at depth i, for $i = 0, 1, 2, \ldots, \lg n - 1$, has a cost of $c(n/2^i + 2)$. Thus, the total cost over all nodes at depth i, for $i = 0, 1, 2, \ldots, \lg n - 1$, is $4^i c(n/2^i + 2) = 2^i cn + 2^{2i+1}$. The bottom level, at deph $\lg n$, has $4^{\lg n} = n^{\lg 4}$ nodes, each contributing cost $T(1)$, for a total cost of $n^{\lg 4}T(1) = \Theta(n^{\lg 4})$.

The cost of the entire tree is

$$\begin{aligned}
T(n) &= \sum_{i=0}^{\lg n-1} \left(4^i c\left(\frac{n}{2^i} + 2\right)\right) + \Theta(n^2) \\
&= \sum_{i=0}^{\lg n-1} \left(4^i c \cdot \frac{n}{2^i}\right) + \sum_{i=0}^{\lg n-1} (4^i c \cdot 2) + \Theta(n^2) \\
&= cn \sum_{i=0}^{\lg n-1} (2^i) + 2c \sum_{i=0}^{\lg n-1} (4^i) + \Theta(n^2) \\
&= cn\frac{2^{\lg n} - 1}{2 - 1} + 2c\frac{4^{\lg n} - 1}{4 - 1} + \Theta(n^2) \\
&= cn(n-1) + \frac{2c}{3}(n^2 - 1) + \Theta(n^2) \\
&= cn^2 - cn + \frac{2cn^2}{3} - \frac{2c}{3} + \Theta(n^2) \\
&= O(n^2).
\end{aligned}$$

Our guess is

$$T(n) \le cn^2 - dn \ \forall\, n \ge n_0,$$

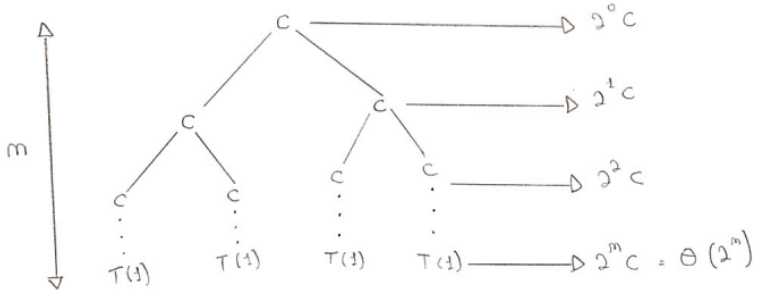
where c, d, and n_0 are positive constants. Substituting into the recurrence yields

$$\begin{aligned}
T(n) &\le 4\left(c\left(\frac{n}{2} + 2\right)^2 - d\left(\frac{n}{2} + 2\right)\right) + n \\
&\le 4\left(c\frac{n^2}{4} + 2cn + 4c - \frac{dn}{2} - 2d\right) + n \\
&= cn^2 + 8cn + 16c - 2dn - 8d + n \\
&= cn^2 - dn - (d - 8c - 1)n - (d - 2c)8 \\
&\le cn^2 - dn,
\end{aligned}$$

where the last step holds as long as $d - 8c - 1 \ge 0$.

4.4-4 Use a recursion tree to determine a good asymptotic upper bound on the recurrence $T(n) = 2T(n-1) + 1$. Use the substitution method to verify your answer.

Figure below ilustrates the recursion tree $T(n) = 2T(n-1) + 1$.



The tree has n levels and 2^i nodes at each level. Since each node costs 1, the cost at depth i is 2^i. The bottom level, at deph n, has 2^n nodes, each contributing cost 1, for a total cost of $2^n = \Theta(2^n)$.

The cost of the entire tree is

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-1}(2^i) + \Theta(2^n) \\ &= \frac{2^n - 1}{2 - 1} + \Theta(2^n) \\ &= 2^n - 1 + \Theta(2^n) \\ &= O(2^n). \end{aligned}$$

Our guess is

$$T(n) \leq c2^n - d \ \forall n \geq n_0,$$

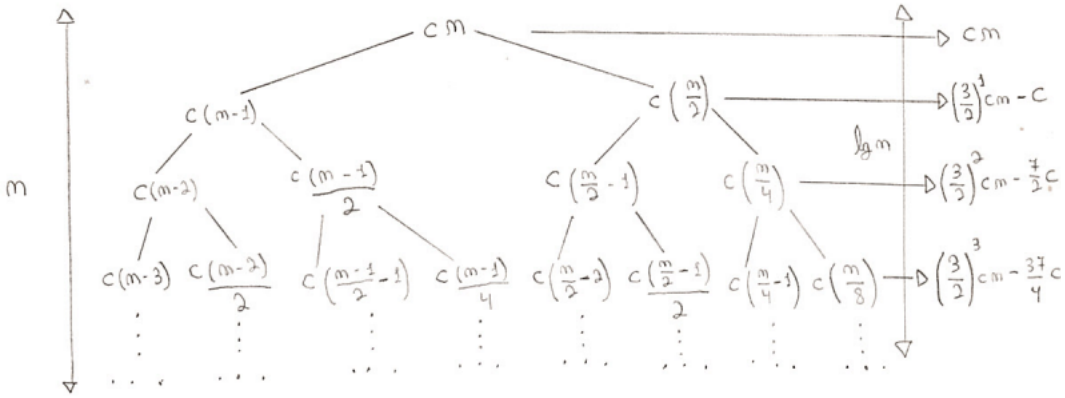
where c, d, and n_0 are positive constants. Substituting into the recurrence yields

$$\begin{aligned} T(n) &\leq 2(c2^{n-1} - d) + 1 \\ &= c2^n - 2d + 1 \\ &= c2^n - d - d + 1 \\ &\leq c2^n - d, \end{aligned}$$

where the last step holds as long as $d \geq 1$.

4.4-5 Use a recursion tree to determine a good asymptotic upper bound on the recurrence $T(n) = T(n-1) + T(n/2) + n$. Use the substitution method to verify your answer.

Figure below ilustrates the recursion tree $T(n) = T(n-1) + T(n/2) + n$.



We start obtaining a lower bound. The cost of the initial levels (before level $\lg n$) of the tree are

$$cn \to (3/2)^1 cn - c \to (3/2)^2 cn - (7/2)c \to (3/2)^3 cn - (37/4)c.$$

Thus, the cost of the tree from the root to level $\lg n$ is at most

$$\sum_{i=0}^{\lg n} \left(\frac{3}{2}\right)^i cn = cn \frac{\left(\frac{3}{2}\right)^{\lg n+1} - 1}{\frac{3}{2} - 1} = 2cn\frac{3}{2}\left(\frac{3}{2}\right)^{\lg n} - 2cn = 3cn\frac{n^{\lg 3}}{n} - 2cn = 3cn^{\lg 3} - 2cn = O(n^{\lg 3}).$$

The cost of the longest simple path from the root to a leaf is

$$\sum_{i=0}^{n} c(n-i) = c\sum_{i=0}^{n} i = c\frac{n(n+1)}{2} = c\frac{n^2}{2} + \frac{c}{2} = O(n^2).$$

Thus, our guess for a lower bound for $T(n)$ is

$$T(n) \geq cn^2 \ \forall\, n \geq n_0,$$

where c, and n_0 are positive constants. Substituting into the recurrence yields

$$\begin{aligned} T(n) &\geq c(n-1)^2 + c\left(\frac{n}{2}\right)^2 + n \\ &= cn^2 - 2cn + 1 + \frac{cn^2}{4} + n \\ &= \frac{5}{4}cn^2 - 2cn + n + 1 \\ &\geq cn^2 - 2cn + n + 1 \\ &\geq cn^2, \end{aligned}$$

where the last step holds as long as $c \geq 1$ and $n_0 \geq 1$. Thus, we have $T(n) = \Omega(n^2)$.
Consider now the recurrence

$$S(n) = 2T(n-1) + n,$$

which is more costly than $T(n)$. We can easily prove that $S(n) = O(2^n)$. Our guess for an upper bound of $S(n)$ is

$$S(n) \leq c2^n - dn \ \forall\, n \geq n_0,$$

where c, d, and n_0 are positive constants. Substituting into the recurrence yields

$$\begin{aligned} S(n) &\leq 2(c2^{n-1} - d(n-1)) + n \\ &= c2^n - 2dn + 2d + n \\ &= c2^n - dn - n(d-1) + 2d \\ &\leq c2^n - dn, \end{aligned}$$

where the last step holds as long as $d \geq 2$ and $n_0 \geq 3$. Thus, we have $T(n) = O(S(n)) = O(2^n)$.

We can obtain a more tight upper bound without using the recursion tree. Let $R(n) = T(n/2) + n$. We have

$$\begin{aligned} T(n) &= T(n-1) + R(n) \\ &= T(n-2) + R(n-1) + R(n) \\ &= R(1) + R(2) + \cdots + R(n-1) + R(n) \\ &\leq n \cdot R(n) \\ &= n \cdot T(n/2) + n^2, \end{aligned}$$

which can be solved using the master method. We have $f(n) = n^2$ and $n^{\log_b a} = n^{\lg n}$. Using the first case, we have

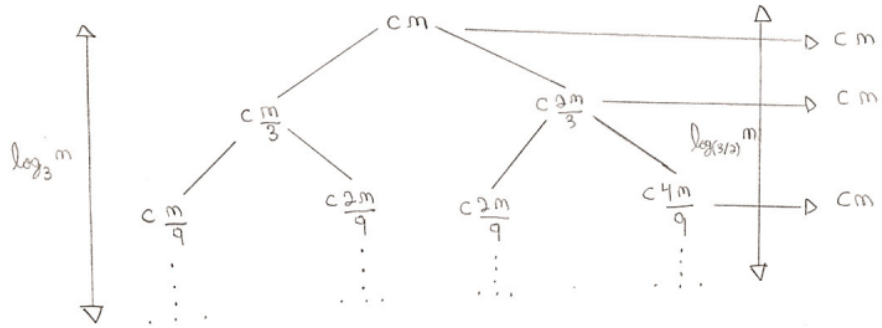
$$f(n) = n^2 = O(n^{\lg n - \epsilon}),\ (\epsilon = 1)$$

which implies

$$T(n) = O(n^{\lg n}).$$

4.4-6 Argue that the solution to the recurrence $T(n) = T(n/3) + T(2n/3) + cn$, where c is a constant, is $\Omega(n \lg n)$ by appealing to a recursion tree.

Figure below ilustrates the recursion tree $T(n) = T(n/3) + T(2n/3) + cn$.



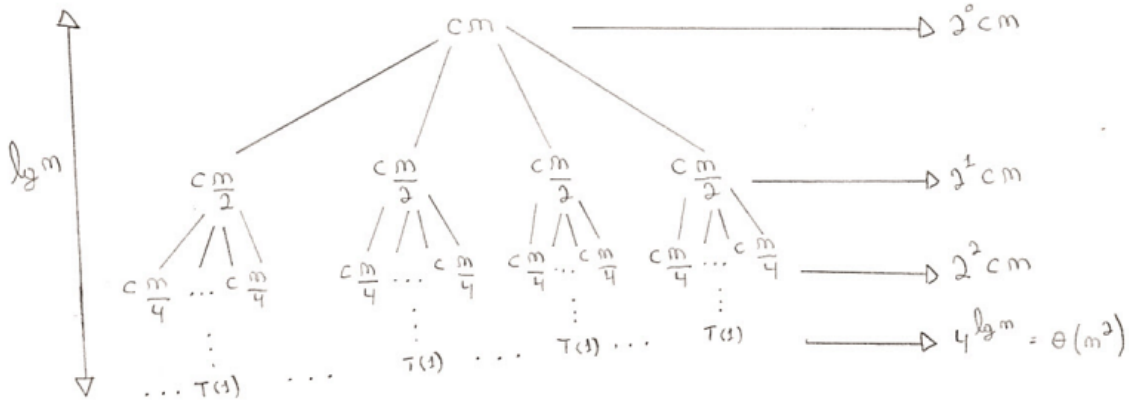
The tree is complete until level $\log_3 n$. The cost of the tree from the root to level $\log_3 n$ is

$$\sum_{i=0}^{\log_3 n} cn = cn \log_3 n,$$

which is $\Omega(n \lg n)$.

4.4-7 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.

Since floors/ceiling usually do not matter, we will draw a recursion tree for the recurrence $T(n) = 4T(n/2) + cn$.



The number of nodes at depth i is 4^i. Since subproblem size reduce by a factor of 2, each node at depth i, for $i = 0, 1, 2, \ldots, \lg n - 1$, has a cost of $c(n/2^i)$. Thus, the total cost over all nodes at depth i, for $i = 0, 1, 2, \ldots, \lg n - 1$, is $(4/2)^i cn = 2^i cn$. The bottom level has $4^{\lg n} = n^2$ nodes, each contributing cost $T(1)$, for a total cost of $n^2 T(1) = \Theta(n^2)$.
The cost of the entire tree is

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

Lets verify with the substitution method. Our guess for an upper bound is

$$T(n) \leq dn^2 - en \ \forall n \geq n_0,$$

where d, e, and n_0 are positive constants. Substituting into the recurrence yields

$$\begin{aligned}
T(n) &\leq 4\left(d\left\lfloor\frac{n}{2}\right\rfloor^2 - e\frac{n}{2}\right) + cn \\
&\leq 4\left(d\left(\frac{n}{2}\right)^2 - e\frac{n}{2}\right) + cn \\
&= 4\left(d\frac{n^2}{4} - e\frac{n}{2}\right) + cn \\
&= dn^2 - 2en + cn \\
&= dn^2 - en - en + cn \\
&\leq dn^2 - en,
\end{aligned}$$

where the last step holds as long as $e \geq c$.
Our guess for a lower bound is

$$T(n) \geq dn^2 \ \forall n \geq n_0,$$

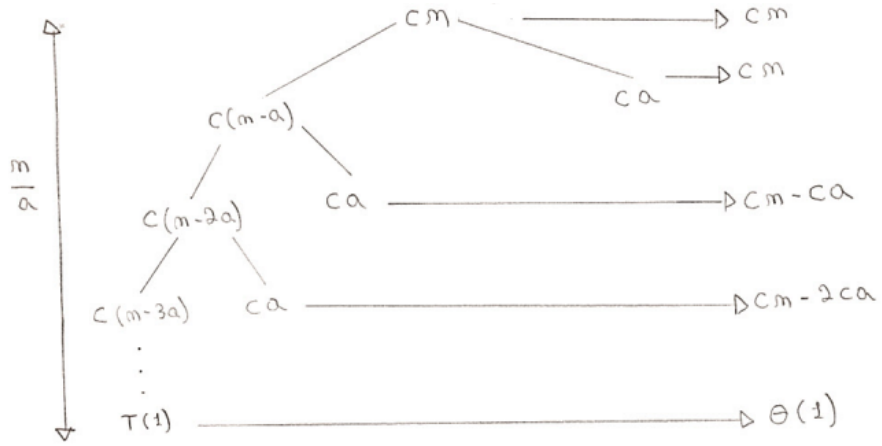
where d, and n_0 are positive constants. Substituting into the recurrence yields

$$\begin{aligned}
T(n) &\geq 4d\left\lfloor\frac{n}{2}\right\rfloor^2 + cn \\
&\geq 4d\left(\frac{n}{2} - 1\right)^2 + cn \\
&= 4d\left(\frac{n^2}{4} - n + 1\right) + cn \\
&= dn^2 - 4dn + 4d + cn \\
&= dn^2 - (4d - c)n + 4d
\end{aligned}$$

where the last step holds as long as $4d - c \geq 4$ and $n_0 \geq d$.

4.4-8 Use a recursion tree to give an asymptotically tight solution to the recurrence $T(n) = T(n-a) + T(a) + cn$, where $a \geq 1$ and $c > 0$ are constants.

Figure below ilustrates the recursion tree $T(n) = T(n-a) + T(a) + cn$.



The height of the tree is n/a. Each level i, for $i = 1, 2, \ldots, (n/a)$, has two nodes, one that costs $c(n-ia)$ and another that costs $T(a) = ca$. Thus, the cost over the nodes at depth i, for $i = 1, 2, \ldots, (n/a)$, is $c(n-a) + ca$. The root level, at deph 0, has a single node that costs cn.
The cost of the entire tree is

$$\begin{aligned}
T(n) &= cn + \sum_{i=1}^{n/a} (c(n-ia) + ca) \\
&= cn + \sum_{i=1}^{n/a} cn - \sum_{i=1}^{n/a} cia + \sum_{i=1}^{n/a} ca \\
&= cn + c\frac{n^2}{a} - \frac{cn(a+n)}{2a} + cn \\
&= c\frac{n^2}{a} - c\frac{n^2}{2a} - c\frac{n}{2} + 2cn \\
&= c\frac{n^2}{2a} + \frac{3}{2}cn \\
&= \Theta(n^2).
\end{aligned}$$

Lets verify with the substitution method. Our guess for an upper bound is

$$T(n) \leq cn^2 \ \forall\, n \geq n_0,$$

where c and n_0 are positive constants. Substituting into the recurrence yields

$$\begin{aligned}
T(n) &\leq c(n^2 - 2an + a^2) + ca + cn \\
&= cn^2 - c(2an - a - n - a^2) \\
&\leq cn^2,
\end{aligned}$$

where the last step holds as long as $n_0 \geq a$.
Our guess for a lower bound is

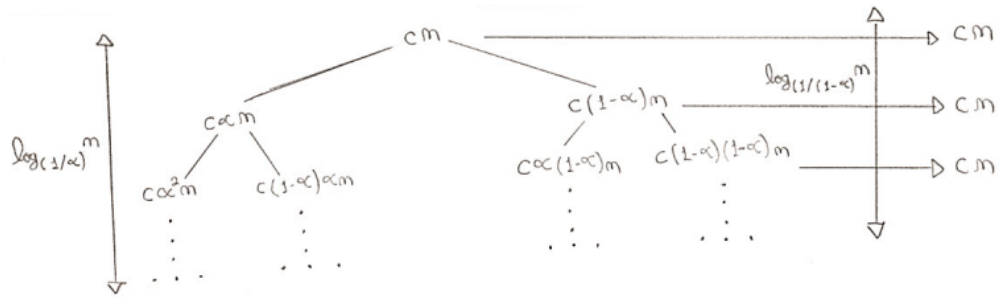
$$T(n) \geq \frac{c}{2a} n^2 \ \forall\, n \geq n_0,$$

where c, and n_0 are positive constants. Substituting into the recurrence yields

$$\begin{aligned}
T(n) &\geq \frac{c}{2a}(n-a)^2 + ca + cn \\
&= \frac{c}{2a}(n^2 - 2an + a^2) + ca + cn \\
&= \frac{c}{2a} n^2 - cn + \frac{1}{2}ca + ca + cn \\
&= \frac{c}{2a} n^2 + \frac{3}{2}ca \\
&\geq \frac{c}{2a} n^2.
\end{aligned}$$

4.4-9 Use a recursion tree to give an asymptotically tight solution to the recurrence $T(n) = T(\alpha n) + T((1-\alpha)n) + cn$, where α is a constant in the range $0 < \alpha < 1$ and $c > 0$ is also a constant.

Let $\alpha \geq 1 - \alpha$. Figure below ilustrates the recursion tree $T(n) = T(\alpha n) + T((1 - \alpha n)n) + cn$.



If it were a complete tree, all the $\log_{1-\alpha} n$ levels would cost cn and the entire tree $cn \log_{1-\alpha} n$. Thus, $T(n) = O(n \log_{1-\alpha} n) = O(n \lg n)$. The tree is complete until level $\log_{1/(1-\alpha)} n$. The cost of the tree from the root to level $\log_{1/(1-\alpha)} n$ is

$$\sum_{i=0}^{\log_{1/(1-\alpha)} n} cn = \left(\sum_{i=1}^{\log_{1/(1-\alpha)} n} cn \right) + cn = cn(\log_{1/(1-\alpha)} n) + cn,$$

which is $\Omega(n \log_{1/(1-\alpha)} n) = \Omega(n \lg n)$.
Lets verify with the substitution method. Our guess for an upper bound is

$$T(n) \leq dn \lg n \ \ \forall\, n \geq n_0,$$

where d and n_0 are positive constants. Substituting into the recurrence yields

$$\begin{aligned}
T(n) &\leq d\alpha n \lg(\alpha n) + d(1-\alpha)n \lg((1-\alpha)n) + dn \\
&= d\alpha n \lg \alpha + d\alpha n \lg n + d(1-\alpha)n \lg(1-\alpha) + d(1-\alpha)n \lg n + cn \\
&= d\alpha n \lg \alpha + d\alpha n \lg n + d(1-\alpha)n \lg(1-\alpha) + dn \lg n - d\alpha n \lg n + cn \\
&= dn \lg n + dn(\alpha \lg \alpha + (1-\alpha) \lg(1-\alpha)) + cn \\
&\leq dn \lg n,
\end{aligned}$$

where the last step holds as long as $d(\alpha \lg \alpha + (1-\alpha) \lg(1-\alpha)) + c \leq 0$.
Our guess for a lower bound is

$$T(n) \geq dn \lg n \ \ \forall\, n \geq n_0,$$

where d, and n_0 are positive constants. Substituting into the recurrence yields

$$\begin{aligned}
T(n) &\geq d\alpha n \lg(\alpha n) + d(1-\alpha)n \lg((1-\alpha)n) + dn \\
&= d\alpha n \lg \alpha + d\alpha n \lg n + d(1-\alpha)n \lg(1-\alpha) + d(1-\alpha)n \lg n + cn \\
&= d\alpha n \lg \alpha + d\alpha n \lg n + d(1-\alpha)n \lg(1-\alpha) + dn \lg n - d\alpha n \lg n + cn \\
&= dn \lg n + dn(\alpha \lg \alpha + (1-\alpha) \lg(1-\alpha)) + cn \\
&\geq dn \lg n,
\end{aligned}$$

where the last step holds as long as $d(\alpha \lg \alpha + (1-\alpha) \lg(1-\alpha)) + c \geq 0$.

Section 4.5 – The master method for solving recurrences

4.5-1 Use the master method to give tight asymptotic bounds for the following recurrences.

a. $T(n) = 2T(n/4) + 1$.
b. $T(n) = 2T(n/4) + \sqrt{n}$.
c. $T(n) = 2T(n/4) + n$.
d. $T(n) = 2T(n/4) + n^2$.

(a) Case 1 applies. $T(n) = \Theta(n^{\log_4 2}) = \Theta(\sqrt{n})$.
(b) Case 2 applies. $T(n) = \Theta(n^{\log_4 2} \lg n) = \Theta(\sqrt{n} \lg n)$.
(c) Case 3 applies. $T(n) = \Theta(n)$.
(d) Case 3 applies. $T(n) = \Theta(n^2)$.

4.5-2 Professor Caesar wishes to develop a matrix-multiplication algorithm that is asymptotically faster than Strassen's algorithm. His algorithm will use the divide-and-conquer method, dividing each matrix into pieces of size $n/4 \times n/4$, and the divide and combine steps together will take $\Theta(n^2)$ time. He needs to determine how many subproblems his algorithm has to create in order to beat Strassen's algorithm. If his algorithm creates a subproblems, then the recurrence for the running time $T(n)$ becomes $T(n) = aT(n/4) + \Theta(n^2)$. What is the largest integer value of a for which Professor Caesar's algorithm would be asymptotically faster than Strassen's algorithm?

Strassen's algorithm costs $\Theta(n^{\lg 7})$. The cost of $T(n)$ is stated below.

- If $a < 16$, Case 3 applies. $T(n) = \Theta(n^2) = o(n^{\lg 7})$.
- If $a = 16$, Case 2 applies. $T(n) = \Theta(n^2 \lg n) = o(n^{\lg 7})$.
- If $a > 16$, Case 1 applies. $T(n) = \Theta(n^{\log_4 a}) = o(n^{\lg 7})$ when $a < 49$.

Thus, the largest integer value of a is 48.

4.5-3 Use the master method to show that the solution to the binary-search recurrence $T(n) = T(n/2) + \Theta(1)$ is $T(n) = \Theta(\lg n)$. (See Exercise 2.3-5 for a description of binary search.)

We have

$$n^{\log_b a} = n^{\lg 1} = \Theta(1) = f(n).$$

Thus, Case 2 applies. $T(n) = \Theta(\lg n)$.

4.5-4 Can the master method be applied to the recurrence $T(n) = 4T(n/2) + n^2 \lg n$? Why or why not? Give an asymptotic upper bound for this recurrence.

We have

$$f(n) = n^2 \lg n,$$

and

$$n^{\log_b a} = n^{\log_2 4} = \Theta(n^2),$$

which is larger than $f(n)$, but not polynomially larger. Thus, we cannot use the master method to solve this recurrence.
We can use a recursion tree to guess the cost of $T(n)$ and verify with the substitution method. Figure below ilustrates the recursion tree of $T(n) = 4T(n/2) + n^2 \lg n$.

Figure here.

The tree has $\lg n$ levels and the number of nodes at depth i is 4^i. Each node at depth i has a cost $c((n/2^i)^2) \lg(n) = 1/4^i cn^2 \lg n$. Thus, the total cost at depth i is $4^i \times 1/4^i cn^2 \lg n = cn^2 \lg n$.
The cost of the entire tree is

$$\sum_{i=0}^{\lg n} cn^2 \lg n = O(n^2 \lg^2 n).$$

Lets verify with the substitution method. Our guess is

$$T(n) \leq cn^2 \lg^2 n \ \forall\, n \geq n_0,$$

where c and n_0 are positive constants. Substituting into the recurrence yields

$$\begin{aligned}
T(n) &\leq 4c\left(\left(\frac{n}{2}\right)^2 \lg^2\left(\frac{n}{2}\right)\right) + n^2 \lg n \\
&= 4c\left(\frac{n^2}{4} \lg\left(\frac{n}{2}\right) \lg\left(\frac{n}{2}\right)\right) + n^2 \lg n \\
&= cn^2 \lg\left(\frac{n}{2}\right) \lg\left(\frac{n}{2}\right) + n^2 \lg n \\
&= cn^2 \lg\left(\frac{n}{2}\right) \lg n - cn^2 \lg\left(\frac{n}{2}\right) + n^2 \lg n \\
&= cn^2 \lg^2 n - cn^2 \lg n - cn^2 \lg n + cn^2 + n^2 \lg n \\
&\leq cn^2 \lg^2 n,
\end{aligned}$$

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

4.5-5 Consider the regularity condition $af(n/b) \geq cf(n)$ for some constant $c < 1$, which is part of case 3 of the master theorem. Give an example of constants $a \geq 1$ and $b > 1$ and a function $f(n)$ that satisfies all the conditions in case 3 of the master theorem except the regularity condition.

Let $a = 1$, $b = 2$, and $f(n) = n \cos n$. We have

$$n^{\log_b a} = n^{\log_2 1} = \Theta(1),$$

which is polynomially smaller than $f(n)$ and satisfies the primary condition of Case 3. However, we have

$$af\left(\frac{n}{b}\right) \leq cf(n) \rightarrow \frac{n}{2} \cos\left(\frac{n}{2}\right) \leq c(n \cos n),$$

which is not valid for some constant $c < 1$ and all sufficiently large n since $\cos(\cdot)$ is not monotonic. Thus, it satisfies the primary condition of Case 3, but not the regularity condition

Problems

4-1 ***Recurrence examples***

Give asymptotic upper and lower bounds for $T(n)$ in each of the following recurrences. Assume that $T(n)$ is constant for $n \geq 2$. Make your bounds as tight as possible, and justify your answers.

a. $2T(n/2) + n^4$.

b. $T(7n/10) + n$.

c. $16T(n/4) + n^2$.

d. $7T(n/3) + n^2$.

e. $7T(n/2) + n^2$.

f. $2T(n/4) + \sqrt{n}$.

g. $T(n-2) + n^2$.

(a) We use the master method. Case 3 applies, since $n^{\lg 2} = n$ is polynomially smaller than $f(n)$. Thus, $T(n) = \Theta(n^4)$.

(b) We use the master method. Case 3 applies, since $n^{\log_{10/7} 1} = 1$ is polynomially smaller than $f(n)$. Thus, $T(n) = \Theta(n)$.

(c) We use the master method. Case 2 applies, since $n^{\log_4 14} = n^2 = \Theta(f(n))$. Thus, $T(n) = \Theta(n^2 \lg n)$.

(d) We use the master method. Case 3 applies, since $n^{\log_3 7}$ is polynomially smaller than $f(n)$. Thus, $T(n) = \Theta(n^2)$.

(e) We use the master method. Case 1 applies, since $n^{\lg 7}$ is polynomially larger than $f(n)$. Thus, $T(n) = \Theta(n^{\lg 7})$.

(f) We use the master method. Case 2 applies, since $n^{\log_4 2} = \sqrt{n} = \Theta(f(n))$. Thus, $T(n) = \Theta(\sqrt{n} \lg n)$.

(g) The recurrence has $n/2$ levels and depth i costs $c(n-2i)^2$. Thus, we have

$$T(n) = \sum_{i=0}^{n/2} c(n-2i)^2 = \sum_{i=0}^{n/2} c(n^2 - 4ni + 4i^2) = c\left(\sum_{i=0}^{n/2} n^2 - \sum_{i=0}^{n/2} 4ni + \sum_{i=0}^{n/2} 4i^2\right) = \Theta(n^3) - \Theta(n^2) + \Theta(n^3) = \Theta(n^3).$$

4-2 ***Parameter-passing costs***

Throughout this book, we assume that parameter passing during procedure calls takes constant time, even if an N-element array is being passed. This assumption is valid in most systems because a pointer to the array is passed, not the array itself. This problem examines the implications of three parameter-passing strategies:

1. An array is passed by pointer. Time $= \Theta(1)$.
2. An array is passed by copying. Time $= \Theta(N)$, where N is the size of the array.
3. An array is passed by copying only the subrange that might be accessed by the called procedure. Time $= \Theta(q - p + 1)$ if the subarray $A[p \ldots q]$ is passed.

a. Consider the recursive binary search algorithm for finding a number in a sorted array (see Exercise 2.3-5). Give recurrences for the worst-case running times of binary search when arrays are passed using each of the three methods above, and give good upper bounds on the solutions of the recurrences. Let N be the size of the original problem and n be the size of a subproblem.

b. Redo part (a) for the MERGE-SORT algorithm from Section 2.3.1.

a. Binary search.

1. *Array passed by pointer.*
   $T(n) = T(n/2) + \Theta(1)$. Case 2 of master method applies, since $n^{\lg 1} = 1 = f(n)$. Thus, $T(n) = \Theta(\lg n)$.
2. *Array passed by copying.*
   $T(n) = T(n/2)+\Theta(N) = T(n/4)+\Theta(N)+\Theta(N) = T(n/8)+\Theta(N)+\Theta(N)+\Theta(N) = \cdots = \sum_{i=0}^{\lg n} \Theta(N) = \Theta(n \lg n)$.
3. *Subarray passed by copying.*
   $T(n) = T(n/2) + \Theta(n)$. Case 3 of master method applies, since $n^{\lg 1} = 1$ is polynomially smaller than $f(n)$.
   Thus, $T(n) = \Theta(n)$.

b. Merge sort.

1. *Array passed by pointer.*
   $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n) \approx 2T(n/2) + \Theta(n)$. Case 2 of master method applies, since $n^{\lg 2} = n = f(n)$.
   Thus, $T(n) = \Theta(n \lg n)$.
2. *Array passed by copying.*
   $T(n) = 2T(n/2)+\Theta(N) = 4T(n/4)+2\Theta(N)+\Theta(N) = 16T(n/8)+4\Theta(N)+2\Theta(N)+\Theta(N) = \cdots = \sum_{i=0}^{\lg n} 2^i \Theta(N) = \Theta(n^2)$.
3. *Subarray passed by copying.*
   $T(n) = 2T(n/2) + \Theta(n)$. Case 2 of master method applies, since $n^{\lg 2} = n = f(n)$. Thus, $T(n) = \Theta(n \lg n)$.

4-3 ***More recurrence examples***

Give asymptotic upper and lower bounds for $T(n)$ in each of the following recurrences. Assume that $T(n)$ is constant for sufficiently small n. Make your bounds as tight as possible, and justify your answers.

a. $T(n) = 4T(n/3) + n \lg n$.
b. $T(n) = 3T(n/3) + n/\lg n$.
c. $T(n) = 4T(n/2) + n^2\sqrt{n}$.
d. $T(n) = 3T(n/3 - 2) + n/2$.
e. $T(n) = 2T(n/2) + n/\lg n$.
f. $T(n) = T(n/2) + T(n/4) + T(n/8) + n$.
g. $T(n) = T(n-1) + 1/n$.
h. $T(n) = T(n-1) + \lg n$.
i. $T(n) = T(n-2) + 1/\lg n$.
j. $T(n) = \sqrt{n}T(\sqrt{n}) + n$.

a. We have $f(n) = n \lg n$ and $n^{\lg_b a} = n^{\log_3 4}$. Since $n \lg n = O(n^{\log_3(4)-0.2})$, case 1 applies and we have $T(n) = \Theta(n^{\log_3 4})$.

b. The tree has $\log_3 n$ levels and depth i, for $i = 0, 1, \ldots, \log_3 n - 1$, costs $n/(\log_3 n - i)$. The cost of the entire tree is

$$T(n) = \sum_{i=0}^{\log_3 n - 1} \frac{n}{\log_3 n - i} = \sum_{i=1}^{\log_3 n} \frac{n}{i} = n \sum_{i=1}^{\log_3 n} \frac{1}{i} = n \cdot H_{\log_3 n} = n \cdot \Theta(\lg \log_3 n) = \Theta(n \lg \lg n).$$

Skipped the proof.

c. We have $f(n) = n^2\sqrt{n} = n^{5/2}$ and $n^{\log_b a} = n^{\log_2 4} = n^2$. Since $n^{5/2} = \Omega(n^{2+1/2})$, we look at the regularity condition in case 3 of the master method. We have $af(n/b) = 4(n/2)^2\sqrt{n/2} = (n^{5/2})/\sqrt{2} \le cn^{5/2}$ for $1/\sqrt{2} \le c < 1$. Case 3 applies and we have $T(n) = \Theta(n^2\sqrt{n})$.

d. The tree has $\log_3 n$ levels and depth i, for $i = 0, 1, \ldots, \log_3 n - 1$ costs $c(n/2) - 2 \cdot 3^i$. The cost of the entire tree is

$$T(n) = \sum_{i=0}^{\log_3 n - 1} \left(c\frac{n}{2} - 2 \cdot 3^i \right) = c \sum_{i=0}^{\log_3 n - 1} \frac{n}{2} - 2 \sum_{i=0}^{\log_3 n - 1} 3^i = \Theta(n \lg n).$$

Our guess for the upper bound is

$$T(n) \le cn \lg n \ \ \forall\, n \ge n_0,$$

where c and n_0 are positive constants. Substituting into the recurrence yields

$$\begin{aligned}
T(n) &\le 3c\left(\frac{n}{3} - 2\right) \lg \left(\frac{n}{3} - 2\right) + \frac{n}{2} \\
&= cn \lg \left(\frac{n}{3} - 2\right) - 6c \lg \left(\frac{n}{3} - 2\right) + \frac{n}{2} \\
&\le cn \lg \left(\frac{n}{3} - 2\right) - 6c \lg \left(\frac{n}{4}\right) + \frac{n}{2} \qquad (n \ge 24) \\
&= cn \lg \left(\frac{n}{3} - 2\right) - 6c \lg n - 12c + \frac{n}{2} \\
&< cn \lg n - 6c \lg n - 12c + \frac{n}{2} \\
&\le cn \lg n,
\end{aligned}$$

where the last step holds as long as $-6c \lg n - 12c + n/2 \le 0$ (skipped simplification).

Our guess for the lower bound is

$$T(n) \ge cn \lg n \ \ \forall\, n \ge n_0,$$

where c, and n_0 are positive constants. Substituting into the recurrence yields

$$\begin{aligned}
T(n) &\ge 3c\left(\frac{n}{3} - 2\right) \lg \left(\frac{n}{3} - 2\right) + \frac{n}{2} \\
&= cn \lg \left(\frac{n}{3} - 2\right) - 6c \lg \left(\frac{n}{3} - 2\right) + \frac{n}{2} \\
&\ge cn \lg \left(\frac{n}{4}\right) - 6c \lg \left(\frac{n}{3} - 2\right) + \frac{n}{2} \qquad (n \ge 24) \\
&= cn \lg n - 2cn - 6c \lg \left(\frac{n}{3} - 2\right) + \frac{n}{2} \\
&\ge cn \lg n,
\end{aligned}$$

where the last step holds as long as $-2cn - 6c \lg(n/3 - 2) + n/2 \ge 0$ (skipped simplification).

e. The tree has $\lg n$ levels and depth i, for $i = 0, 1, \ldots, \lg n - 1$, costs $n/(\lg n - i)$. The cost of the entire tree is

$$T(n) = \sum_{i=0}^{\lg n - 1} \frac{n}{\lg n - i} = \sum_{i=1}^{\lg n} \frac{n}{i} = n \sum_{i=1}^{\lg n} \frac{1}{i} = n \cdot H_{\lg n} = n \cdot \Theta(\lg \lg n) = \Theta(n \lg \lg n).$$

Skipped the proof.

f. The tree has $\lg n$ levels, but is not complete. Considering only the levels in which the tree is complete, depth i, for $i = 1, 2, \ldots, \log_8 n$, costs $(7/8)^i cn$. Thus, the cost of the entire tree is at most

$$T(n) \leq \sum_{i=0}^{\lg n - 1} \left(\left(\frac{7}{8} \right)^i cn \right) = cn \sum_{i=0}^{\lg n - 1} \left(\left(\frac{7}{8} \right)^i \right) = cn \frac{1 - \left(\frac{7}{8} \right)^{\lg n}}{1 - \frac{7}{8}} = cn \frac{1 - n^{\lg 7 - 3}}{\frac{1}{8}} = 8cn - 8cn^{\lg 7 - 2} = O(n).$$

Our guess for the upper bound is

$$T(n) \leq cn \ \forall n \geq n_0,$$

where c and n_0 are positive constants. Substituting into the recurrence yields

$$\begin{aligned} T(n) &\leq c\frac{n}{2} + c\frac{n}{4} + c\frac{n}{8} \\ &= \frac{7}{8}cn + n \\ &\leq cn, \end{aligned}$$

where the last step holds as long as $c \geq 8$.

Our guess for the lower bound is

$$T(n) \geq cn \ \forall n \geq n_0,$$

where c and n_0 are positive constants. Substituting into the recurrence yields

$$\begin{aligned} T(n) &\geq c\frac{n}{2} + c\frac{n}{4} + c\frac{n}{8} \\ &= \frac{7}{8}cn + n \\ &\geq cn, \end{aligned}$$

where the last step holds as long as $c \leq 8$.

g. The tree has n levels and depth i, for $i = 1, 2, \ldots, n - 1$, costs $1/(n - i)$. The cost of the entire tree is

$$\sum_{i=0}^{n-1} \frac{1}{n - i} = \sum_{i=1}^{n} \frac{1}{i} = H_n = \Theta(\lg n).$$

Skipped the proof.

h. The tree has n levels and depth i, for $i = 1, 2, \ldots, n - 1$, costs $\lg(n - i)$. The cost of the entire tree is

$$\sum_{i=0}^{n-1} \lg(n - i) = \sum_{i=1}^{n} \lg i = \lg(n!) = \Theta(n \lg n).$$

Skipped the proof.

i. Skipped.

j. Skipped.

4-4 ***Fibonacci numbers***

This problem develops properties of the Fibonacci numbers, which are defined by recurrence (3.22). We shall use the technique of generating functions to solve the Fibonacci recurrence. Define the generating function (or formal power series) $\mathcal{F}$ as

$$\mathcal{F}(z) = \sum_{i=0}^{\infty} F_i z^i \quad = 0 + z + z^2 + 2z^3 + 3z^4 + 5z^5 + 8z^6 + 13z^7 + 21z^8 + \ldots,$$

where F_i is the ith Fibonacci number.

a. Show that $\mathcal{F}(z) = z + z\mathcal{F}(z) + z^2\mathcal{F}(z)$.

b. Show that

$$\begin{aligned}\mathcal{F}(z) &= \frac{z}{1 - z - z^2} \\ &= \frac{z}{(1 - \phi z)(1 - \hat{\phi} z)} \\ &= \frac{1}{\sqrt{5}}\left(\frac{1}{1 - \phi z} - \frac{1}{1 - \hat{\phi} z}\right),\end{aligned}$$

where

$$\phi = \frac{1 + \sqrt{5}}{2} = 1.61803\ldots$$

and

$$\hat{\phi} = \frac{1 - \sqrt{5}}{2} = 0.61803\ldots.$$

c. Show that

$$\mathcal{F}(z) = \sum_{i=0}^{\infty} \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i) z^i.$$

d. Use part (c) to prove that $F_i = \phi^i/\sqrt{5}$ for $i > 0$, rounded to the nearest integer. (Hint: Observe that $|\hat{\phi}| < 1$.)

a.

$$\begin{aligned}\mathcal{F}(z) &= \sum_{i=0}^{\infty} F_i z^i \\ &= 0 + z + \sum_{i=2}^{\infty} (F_{(i-1)} + F_{(i-2)}) z^i \\ &= z + \sum_{i=2}^{\infty} F_{(i-1)} z^i + \sum_{i=2}^{\infty} F_{(i-2)} z^i \\ &= z + \sum_{i=1}^{\infty} F_i z^{i+1} + \sum_{i=0}^{\infty} F_i z^{i+2} \\ &= z + \sum_{i=0}^{\infty} F_i z^{i+1} + \sum_{i=0}^{\infty} F_i z^{i+2} \qquad (\text{since } F_0 = 0) \\ &= z + z\sum_{i=0}^{\infty} F_i z^i + z^2 \sum_{i=0}^{\infty} F_i z^i \\ &= z + z\mathcal{F}(z) + z^2\mathcal{F}(z).\end{aligned}$$

b.

$$\begin{aligned}
\mathcal{F}(z) &= \mathcal{F}(z) \cdot \frac{1-z-z^2}{1-z-z^2} \\
&= \frac{\mathcal{F}(z) - z\mathcal{F}(z) - z^2\mathcal{F}(z)}{1-z-z^2} \\
&= \frac{\mathcal{F}(z) - (z + z\mathcal{F}(z) + z^2\mathcal{F}(z)) + z}{1-z-z^2} \\
&= \frac{\mathcal{F}(z) - \mathcal{F}(z) + z}{1-z-z^2} && \text{(from previous proof)} \\
&= \frac{z}{1-z-z^2} \\
&= \frac{z}{1-(\phi+\hat{\phi})z + \phi\hat{\phi}z^2} && \text{(since } \phi+\hat{\phi}=1 \text{ and } \phi\hat{\phi}=-1\text{)} \\
&= \frac{z}{(1-\phi z)(1-\hat{\phi}z)} \\
&= \frac{1}{\sqrt{5}}\left(\frac{1}{1-\phi z} - \frac{1}{1-\hat{\phi}z}\right). && \text{(skipped this proof)}
\end{aligned}$$

c.

$$\begin{aligned}
\mathcal{F}(n) &= \frac{1}{\sqrt{5}}\left(\frac{1}{1-\phi z} - \frac{1}{1-\hat{\phi}z}\right) \\
&= \frac{1}{\sqrt{5}}\left(\sum_{i=0}^{\infty}(\phi z)^i - \sum_{i=0}^{\infty}(\hat{\phi}z)^i\right) && \text{(by equation A.6, geometric series)} \\
&= \frac{1}{\sqrt{5}}\sum_{i=0}^{\infty}\left((\phi z)^i - (\hat{\phi}z)^i\right) \\
&= \sum_{i=0}^{\infty}\frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i)z^i.
\end{aligned}$$

d. Skipped.

4-5 ***Chip testing***

Professor Diogenes has n supposedly identical integrated-circuit chips that in principle are capable of testing each other. The professor's test jig accommodates two chips at a time. When the jig is loaded, each chip tests the other and reports whether it is good or bad. A good chip always reports accurately whether the other chip is good or bad, but the professor cannot trust the answer of a bad chip. Thus, the four possible outcomes of a test are as follows:

Chip A says	Chip B says	Conclusion
B is good	A is good	both are good, or both are bad
B is good	A is bad	at least one is bad
B is bad	A is good	at least one is bad
B is bad	A is bad	at least one is bad

a. Show that if at least $n/2$ chips are bad, the professor cannot necessarily determine which chips are good using any strategy based on this kind of pairwise test. Assume that the bad chips can conspire to fool the professor.

b. Consider the problem of finding a single good chip from among n chips, assuming that more than $n/2$ of the chips are good. Show that $\lfloor n/2 \rfloor$ pairwise tests are sufficient to reduce the problem to one of nearly half the size.

c. Show that the good chips can be identified with $\Theta(n)$ pairwise tests, assuming that more than $n/2$ of the chips are good. Give and solve the recurrence that describes the number of tests.

a. Let n_g be the number of good chips and n_b the number of bad chips, such that $n_b \geq n_g$ and $n_g + n_b = n$. If the bad chips decide evaluate the others incorrectly (good as bad and bad as good), the professor will have the following result:

Chip state	Tested as good	Tested as bad
Good	$n_g - 1$ times	n_b times
Bad	$n_b - 1$ times	n_g times

In this jig test, the number of good tests of the bad chips will be equal or greater the number of good tests of the good chips, which will confuse the professor.

b. Group the chips in groups of two (if n is odd, put the remaining chip in the next subproblem), making a total of $\lfloor n/2 \rfloor$ groups, and evaluate each group in the test jig. For each test, do the following:

Group type	Chip A says	Chip B says	Conclusion
1	B is good	A is good	keep one of them
2	B is good	A is bad	discard both
3	B is bad	A is good	discard both
4	B is bad	A is bad	discard both

For each test where at least one of the chips is evaluated as bad (group types 2, 3, and 4), we known that at least one of them is truly bad. Thus, we can safely discard both and assure that the majority of the remaining chips are good. As for the groups where both of the chips are evaluated as good (group type 1), we can assure that at least half of these groups are composed by truly good chips, thus keeping one of them is enough to assure that the subproblem will have at least half of good chips. The case where exactly half of the groups of type 1 is composed by good chips only can happen when n is odd and the remaining chip that we previously added to the subproblem must be good, thus assuring that the majority of the chips from the subproblem is good. Also, since the number of groups is $\lfloor n/2 \rfloor$, the algorithm will perform $\lfloor n/2 \rfloor$ tests and the subproblem will have at most $\lceil n/2 \rceil$ chips.

c. The recurrence of the above algorithm is

$$T(n) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \frac{n}{2}.$$

We have that $f(n) = n/2$ and $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$. Since $n/2 = \Omega(n^{0+0.5})$, we look at the regularity condition in case 3 of masther method. We have $af(n/b) = n/4 \leq cn/2$ for $1/2 \leq c < 1$. Case 3 applies and we have $T(n) = \Theta(n/2) = \Theta(n)$.

4-6 ***Monge arrays***

An $m \times n$ array A of real numbers is a ***Monge array*** if for all i, j, k, and l such that $1 \leq i < k \leq m$ and $1 \leq j < l \leq n$, we have

$$A[i,j] + A[k,l] \leq A[i,l] + A[k,j].$$

In other words, whenever we pick two rows and two columns of a Monge array and consider the four elements at the intersections of the rows and the columns, the sum of the upper-left and lower-right elements is less than or equal to the sum of the lower-left and upper-right elements. For example, the following array is Monge:

10	17	13	28	23
17	22	16	29	23
24	28	22	34	24
11	13	6	17	7
45	44	32	37	23
36	33	19	21	6
75	66	51	53	34

a. Prove that an array is Monge if and only if for all $i = 1, 2, \ldots, m-1$ and $j = 1, 2, \ldots, n-1$, we have:

$$A[i,j] + A[i+1,j+1] \leq A[i,j+1] + A[i+1,j].$$

(Hint: For the "if" part, use induction separately on rows and columns.)

b. The following array is not Monge. Change one element in order to make it Monge. (Hint: Use part(a).)

37	23	22	32
21	6	7	10
53	34	30	31
32	13	9	6
43	21	15	8

c. Let $f(i)$ be the index of the column containing the leftmost minimum element of row i. Prove that $f(1) \leq f(2) \leq \cdots \leq f(m)$ for any $m \times n$ Monge array.

d. Here is a description of a divide-and-conquer algorithm that computes the leftmost minimum element in each row of an $m \times n$ Monge array A:

> Construct a submatrix A' of A consisting of the even-numbered rows of A. Recursively determine the leftmost minimum for each row of A'. Then compute the leftmost minimum in the odd-numbered rows of A.

Explain how to compute the leftmost minimum in the odd-numbered rows of A (given that the leftmost minimum of the even-numbered rows is known) in $O(m+n)$ time.

e. Write the recurrence describing the running time of the algorithm described in part (d). Show that it is $O(m + n \log m)$.

a. The "only if" part is trivial. Since $k = i+1, \ldots, m$ and $l = j+1, \ldots, n$, we have

$$A[i,j] + A[k,l] \leq A[i,l] + A[k,j] \rightarrow A[i,j] + A[i+1,j+1] \leq A[i,j+1] + A[i+1,j],$$

For the "if" part, we first need to show

$$A[i,j] + A[i+1,j+1] \leq A[i,j+1] + A[i+1,j] \rightarrow A[i,j] + A[k,j+1] \leq A[i,j+1] + A[k,j] \tag{1}$$

is valid for all $k > i$. The base case, which occurs when $k = i+1$, is given. Thus, we have

$$A[k,j] + A[k+1,j+1] \leq A[k,j+1] + A[k+1,j].$$

in which $k = i+1, \ldots, m-1$. Now assume that the rhs of (1) holds for a given k

$$A[i,j] + A[k,j+1] \leq A[i,j+1] + A[k,j],$$

then we have

$$\underbrace{A[i,j] + A[k,j+1]}_{\text{assumption}} + \underbrace{A[k,j] + A[k+1,j+1]}_{\text{base case}} \leq \underbrace{A[i,j+1] + A[k,j]}_{\text{assumption}} + \underbrace{A[k,j+1] + A[k+1,j]}_{\text{base case}},$$

cancelling equal terms on both sides, we have

$$A[i,j] + A[k+1,j+1] \leq A[i,j+1] + A[k+1,j],$$

which shows that it also holds for $k+1$ and proves the inductive step.

Then we need to show

$$A[i,j]+A[i+1,j+1]\leq A[i,j+1]+A[i+1,j]\rightarrow A[i,j]+A[i+1,l]\leq A[i,l]+A[i+1,j] \tag{2}$$

is valid for all $l>j$. The base case, which occurs when $l=j+1$, is given. Thus, we have

$$A[i,l]+A[i+1,l+1]\leq A[i,l+1]+A[i+1,l].$$

in which $l=j+1,\ldots,n-1$. Now assume that the rhs of (2) holds for a given l

$$A[i,j]+A[i+1,l]\leq A[i,l]+A[i+1,j],$$

then we have

$$\underbrace{A[i,j]+A[i+1,l]}_{\text{assumption}}+\underbrace{A[i,l]+A[i+1,l+1]}_{\text{base case}}\leq\underbrace{A[i,l]+A[i+1,j]}_{\text{assumption}}+\underbrace{A[i,l+1]+A[i+1,l]}_{\text{base case}},$$

cancelling equal terms on both sides, we have

$$A[i,j]+A[i+1,l+1]\leq +A[i,l+1]+A[i+1,j],$$

which shows that it also holds for $l+1$ and proves the inductive step.
From the "if" and "only if" proofs, we have

$$A[i,j]+A[k,l]\leq A[i,l]+A[k,j]\iff A[i,j]+A[i+1,j+1]\leq A[i,j+1]+A[i+1,j].$$

b. Let M be the $m\times n$ matrix we want to make Monge. In this case, $m=5$ and $n=4$. From item (a), we know that, to be Monge, the following needs to hold:

$$M[i,j]+M[i+1,j+1]\leq M[i,j+1]+M[i+1,j]\ \forall\, i=1,2,\ldots,m-1\ \forall\, j=1,2,\ldots,n-1,$$

which implies

$$M[i,j]+M[i+1,j+1]-M[i,j+1]+M[i+1,j]\leq 0.$$

Let K be an $(m-1)\times(n-1)$ matrix where

$$K[i,j]=M[i,j]+M[i+1,j+1]-M[i,j+1]+M[i+1,j].$$

Thus, we have

$$K=\begin{bmatrix}-1 & 2 & -7\\ -4 & -5 & -2\\ 0 & 0 & -4\\ -3 & -2 & -4\end{bmatrix},$$

which shows that the problem is that $M[1,2]+M[2,3]-M[1,3]+M[2,2]=2>0$.
We can make M monge by changing the element $M[1,3]$ from 22 to 24, now becoming:

$$M=\begin{bmatrix}37 & 23 & 24 & 32\\ 21 & 6 & 7 & 10\\ 53 & 34 & 30 & 31\\ 32 & 13 & 9 & 6\\ 43 & 21 & 15 & 8\end{bmatrix}.$$

c. Lets assume that $f(i+1)<f(i)$. From the definition of a Monge array, we have

$$A[i,f(i+1)]+A[i+1,f(i)]\leq A[i,f(i)]+A[i+1,f(i+1)],$$

which is not possible since from the definition of $f(\cdot)$

$$A[i,f(i+1)]>A[i,f(i)],$$

and

$$A[i+1,f(i)]\geq A[i+1,f(i+1)].$$

d. We know from item (c) that $f(i-1) \leq f(i) \leq f(i+1)$. Thus, for each odd-numbered row i of the matrix, we just need to find the leftmost minimum of row i between the columns $f(i-1)$ and $f(i+1)$, which includes $f(i+1) - f(i-1) + 1$ elements. If i corresponds to the first ($i = 1$) or the last ($i = m$) row of the matrix, consider $f(i-1) = f(0) = 1$ or $f(i+1) = f(m+1) = m$. Since the matrix has $\lceil m/2 \rceil$ odd-numbered rows, finding the leftmost minimum of all of them takes

$$\begin{aligned}\sum_{i=1}^{\lceil m/2 \rceil} (f(i+1) - f(i-1) + 1) &= \left\lceil \frac{m}{2} \right\rceil + \sum_{i=1}^{\lceil m/2 \rceil} (f(i+1) - f(i-1)) \\ &= O(m) + f(\lceil m/2 \rceil) - f(1) \\ &= O(m+n).\end{aligned}$$

e. Since we can partition the array in $O(1)$ (working with pointers), the recurrence can be written as

$$\begin{aligned}T(m) &= T(m/2) + O(m+n) \\ &= \sum_{i=0}^{\lg m - 1} \left(cn + d\frac{m}{2^i} \right) \\ &= cn \lg m + dm \sum_{i=0}^{\lg m - 1} \frac{1}{2^i} \\ &\leq cn \lg m + dm \sum_{i=0}^{\infty} (1/2)^i \qquad \text{(infinity decreasing geometric series)} \\ &= cn \lg m + dm \left(\frac{1}{1 - (1/2)} \right) \\ &= cn \lg m + 2dm \\ &= O(m + n \lg m).\end{aligned}$$

Section 5.1 – The hiring problem

5.1-1 Show that the assumption that we are always able to determine which candidate is best, in line 4 of procedure HIRE-ASSISTANT, implies that we know a total order on the ranks of the candidates.

Let A be the set of candidates in random order and R the binary relation "is better than or equal" on the set A. R is a total order if

(a) R is **reflexive**. That is, $a \; R \; a \;\; \forall \, a \in A$;

(b) R is **antisymmetric**. That is, $a \; R \; b$ and $b \; R \; a$ imply $a = b$;

(c) R is **transitive**. That is, $a \; R \; b$ and $b \; R \; c$ imply $a \; R \; c$;

(d) R is a **total relation**. That is, $a \; R \; b$ or $b \; R \; a \;\; \forall \, a, b \in A$.

The above properties are necessary because

(a) if two different candidates have the same qualification, it is necessary so that they can be compared;

(b) if both a is "better than or equal" than b and b is "better than or equal" than a and they qualifications are not equal, we would not be able to choose one of them and still be hiring "the best candidate we have seen so far";

(c) if we hire b because he is "better than or equal" than a and then we hire c because he is "better than or equal" than b and c is not "better than or equal" than a, we are not hiring "the best candidate we have seen so far";

(d) if R is not a total relation, we would not be able to compare any two candidates.

5.1-2 ($\star$) Describe an implementation of the procedure RANDOM(a, b) that only makes calls to RANDOM(0, 1). What is the expected running time of your procedure, as a function of a and b?

The pseudocode is stated below.

```
   RandomInterval(a, b)
1  | flips = ⌈lg(b − a)⌉
2  | count = ∞
3  | while count > b do
4  | |  count = 0
5  | |  for i = 1 to flips do
6  | |  |  count = count + (2^{i−1} · Random(0, 1))
7  | return count + a
```

The expected running time is

$$\underbrace{2^{\lceil \lg(b-a) \rceil}/(b-a)}_{\text{while loop}} \cdot \underbrace{\lceil \lg(b-a) \rceil}_{\text{for loop}} < 2 \cdot \lceil \lg(b-a) \rceil,$$

where the last inequality is valid since $1 \leq 2^{\lceil \lg(b-a) \rceil}/(b-a) < 2$.

5.1-3 ($\star$) Suppose that you want to output 0 with probability 1/2 and 1 with probability 1/2. At your disposal is a procedure BIASED-RANDOM, that outputs either 0 or 1. It outputs 1 with some probability p and 0 with probability $1 - p$, where $0 < p < 1$, but you do not know what p is. Give an algorithm that uses BIASED-RANDOM as a subroutine, and returns an unbiased answer, returning 0 with probability 1/2 and 1 with probability 1/2. What is the expected running time of your algorithm as a function of p?

The pseudocode is stated below.

```
   Random()
1  | while 1 do
2  | |  r_1 = Random(0, 1)
3  | |  r_2 = Random(0, 1)
4  | |  if r_1 ≠ r_2 then
5  | |  |  return r_1
```

The expected running time is

$$\frac{1}{\underbrace{(1-p)p}_{(r_1, r_2) = (0,1)} + \underbrace{p(1-p)}_{(r_1, r_2) = (1,0)}} \cdot 1 = \frac{1}{2p(1-p)}.$$

Section 5.2 – Indicator random variables

5.2-1 In HIRE-ASSISTANT, assuming that the candidates are presented in a random order, what is the probability that you hire exactly one time? What is the probability that you hire exactly n times?

Since the initial dummy candidate is the least qualified, HIRE-ASSISTANT will always hire the first candidate. It hires exacly one time when the best candidate is the first to be interviewed. Thus, the probability is $1/n$. To hire exactly n times, the candidates has to be in increasing order of quality. Since there are $n!$ possible orderings (each one with equal probability of happening), the probability is $1/n!$.

5.2-2 In HIRE-ASSISTANT, assuming that the candidates are presented in a random order, what is the probability that you hire exactly twice?

The first candidate is always hired, thus the best qualified candidate cannot be the first to be interviewed. Also, among all the candidates that are better qualified than the first candidate, the best candidate must be interviewed first. Otherwise, a third candidate will be hired between them. Now assume that the first candidate to be interviewed is the ith best qualified, for $i = 2, \ldots, n$. This occurs with a probability of $1/n$. To hire exactly twice, the best candidate must be the first to be interviewed among the $i-1$ candidates that are better qualified than candidate i. This occurs with a probability of $1/(i-1)$. Thus, the probability of hiring exactly twice is

$$\sum_{i=2}^{n} \frac{1}{n}\frac{1}{i-1} = \frac{1}{n}\sum_{i=1}^{n-1}\frac{1}{i} = \frac{1}{n}(\lg(n-1) + O(1)).$$

5.2-3 Use indicator random variables to compute the expected value of the sum of n dice.

Let X_i be an indicator random variable of a dice coming up the number i. We have $\Pr\{X_i\} = 1/6$. Let X be a random variable denoting the result of throwing a dice. Then

$$\mathrm{E}[X] = \sum_{i=1}^{6} i \cdot \Pr\{X_i\} = \sum_{i=1}^{6} i \cdot \frac{1}{6} = \frac{1}{6}\sum_{i=1}^{6} i = \frac{1}{6}\frac{6\cdot 7}{2} = 3.5.$$

By linearity of expectations, the expected value of the sum of n dice is the sum of the expected value of each dice. Thus,

$$\sum_{i=1}^{n} \mathrm{E}[X] = \sum_{i=1}^{n} 3.5 = 3.5 \cdot n.$$

5.2-4 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 who get back their own hat?

Let X_i be an indicator random variable of customer i getting back his own hat. We have

$$\Pr\{X_i\} = \mathrm{E}[X_i] = 1/n.$$

Let X be a random variable denoting the number of customers who get back their own hat. Then

$$X = X_1 + X_2 + \cdots + X_n,$$

which implies

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] \\
&= \sum_{i=1}^{n} \mathrm{E}[X_i] \\
&= \sum_{i=1}^{n} \frac{1}{n} \\
&= 1.
\end{aligned}$$

5.2-5 Let $A[1, \ldots, n]$ 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. (See Problem 2-4 for more on inversions.) Suppose that the elements of A form a uniform random permutation of $\langle 1, 2, \ldots, n \rangle$. Use indicator random variables to compute the expected number of inversions.

Let X_{ij} be an indicator random variable for the event that the pair (i, j) is inverted. Since A forms a uniform random permutation, we have

$$\Pr\{X_{ij}\} = \Pr\{\overline{X_{ij}}\} = 1/2,$$

which implies

$$\mathrm{E}[X_{ij}] = 1/2.$$

Let X be a random variable denoting the number of inversions of A. Since there are $\binom{n}{2}$ possible pairs on A, each with probability $1/2$ of being inverted, we have

$$\mathrm{E}[X] = \binom{n}{2}\frac{1}{2} = \frac{n!}{2! \cdot (n-2)!}\frac{1}{2} = \frac{n(n-1)}{4}.$$

Section 5.3 – Randomized algorithms

5.3-1 Professor Marceau objects to the loop invariant used in the proof of Lemma 5.5. He questions whether it is true prior to the first iteration. He reasons that we could just as easily declare that an empty subarray contains no 0-permutations. Therefore, the probability that an empty subarray contains a 0-permutation should be 0, thus invalidating the loop invariant prior to the first iteration. Rewrite the procedure RANDOMIZE-IN-PLACE so that its associated loop invariant applies to a nonempty subarray prior to the first iteration, and modify the proof of Lemma 5.5 for your procedure.

Just select a random element in the array and swap it with the first element.

```
Randomize-In-Place(A)
1 | n = A.length
2 | swap A[1] with A[Random(1, n)]
3 | for i = 2 to n − 1 do
4 | | swap A[i] with A[Random(i, n)]
```

The only difference in the proof of Lemma 5.5 is the initialization of the loop invariant:

- **Initialization.** Consider the situation just before the first loop iteration, so that $i = 2$. The loop invariant says that for each possible 1-permutation, the subarray $A[1, \ldots, 1]$ contains this 1-permutation with probability $(n - i + 1)/n! = (n - 1)!/n! = 1/n$. The subarray $A[1, \ldots, 1]$ has a single element and this element was randomly choosed among the n elements of the array. Thus, $A[1, \ldots, 1]$ contains this 1-permutation with probability $1/n$, and the loop invariant holds prior to the first iteration.

5.3-2 Professor Kelp decides to write a procedure that produces at random any permutation besides the identity permutation. He proposes the following procedure:

```
Permute-Without-Identity(A)
1 | n = A.length
2 | for i = 1 to n − 1 do
3 | | swap A[i] with A[Random(i + 1, n)]
```

Does this code do what Professor Kelp intends?

No. This code enforces that every position i of the resulting array receives an element that is different from the ith element of the original array. However, this requirement discards much more permutations than just the identity permutation. For instance, consider the array $A = [1, 2, 3]$ and a permutation of it $A' = [1, 3, 2]$. In this case, the permutation A' is not identical to the original array A. However, Professor Kelp's code is not able to produce this permutation.

5.3-3 Suppose that instead of swapping element $A[i]$ with a random element from the subarray $A[i, \ldots, n]$, we swapped it with a random element from anywhere in the array:

```
Permute-With-All(A)
1 | n = A.length
2 | for i = 1 to n do
3 | | swap A[i] with A[Random(1, n)]
```

Does this code produce a uniform random permutation? Why or why not?

No. As a counterexample, consider the input array $A = [1, 2, 3]$. Since each call to RANDOM can produce one of three values, the number of possible outcomes after all the RANDOM calls can be seen as the number of strings over the set $\{1, 2, 3\}$, which is $3^3 = 27$. However, since an array of size 3 has $3! = 6$ distinct permutations, and 27 is not divisible by 6, it is not possible that each of the 6 permutations of A has the same probability of happening among the 27 possible outcomes of PERMUTE-WITH-ALL.

5.3-4 Professor Armstrong suggests the following procedure for generating a uniform random permutation:

```
Permute-By-Cyclic(A)
1  n = A.length
2  let B[1...n] be a new array
3  offset = Random(1, n)
4  for i = 1 to n do
5      dest = i + offset
6      if dest > n then
7          dest = dest − n
8      B[dest] = A[i]
```

Show that each element $A[i]$ has a $1/n$ probability of winding up in any particular position in B. Then show that Professor Armstrong is mistaken by showing that the resulting permutation is not uniformly random.

What Professor Armstrong's code does is a circular shift of all the elements to the right by i positions. Since each of the n possible shifts has the same probability of happening, each element has indeed a probability of $1/n$ of winding up in any particular position of the final array B. However, since this code has only n possible outcomes and A has $n!$ permutations, it can not produce a uniform random distribution over A. More precisely, the Professor Armstrong's code is not able to produce any permutation of A that is not a circular shift of A.

5.3-5 ($\star$) Prove that in the array P in procedure PERMUTE-BY-SORTING, the probability that all elements are unique is at least $1 - 1/n$.

Let X_i be an indicador random variable for the event that the ith priority is not unique. Since the subarray $P[1, \ldots, i-1]$ has at most $i-1$ distinct numbers, we have $\Pr\{X_i\} = \mathrm{E}[X_i] \leq (i-1)/n^3$. Let X be a random variable for the event that at least one priority is not unique. Then

$$X = (X_1 \cup X_2 \cup \cdots X_n) = X_1 + X_2 + \cdots + X_n,$$

which implies

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] \\
&= \sum_{i=1}^{n} \mathrm{E}[X_i] \\
&\leq \sum_{i=1}^{n} \frac{i-1}{n^3} \\
&= \frac{1}{n^3} \sum_{i=0}^{n-1} i \\
&= \frac{1}{n^3} \frac{(n-1)\cdot n}{2} \\
&= \frac{n-1}{2n^2} \\
&\leq \frac{1}{n}.
\end{aligned}$$

Thus, the probability that all elements are unique is

$$\mathrm{E}[\overline{X}] = 1 - \mathrm{E}[X] \geq 1 - \frac{1}{n}.$$

5.3-6 Explain how to implement the algorithm PERMUTE-BY-SORTING to handle the case in which two or more priorities are identical. That is, your algorithm should produce a uniform random permutation, even if two or more priorities are identical.

The pseudocode is stated below.

```
   Permute-By-Sorting-Unique(A)
 1 | n = A.length
 2 | let P[1...n] be a new array
 3 | repeat
 4 | |   for i = 1 to n do
 5 | |   |   P[i] = Random(1, n^3)
 6 | |   let Q be a copy of P
 7 | |   sort Q
 8 | |   unique = True
 9 | |   for i = 2 to n do
10 | |   |   if Q[i] == Q[i - 1] then
11 | |   |   |   unique = False
12 | |   |   |   break
   | | until unique
13 | | sort A, using P as sort keys
```

Before sorting A using P as sort keys, the above algorithm verifies if P has unique priorities. If the priorities are not unique, P is generated again until it has unique priorities. Since the probability that a random P is unique is at least $1 - 1/n$, the expected number of iterations of the repeat loop of lines 3-12 is less than 2.

5.3-7 Suppose we want to create a ***random sample*** of the set $\{1, 2, 3, \ldots, n\}$, that is, an m-element subset S, where $0 \leq m \leq n$, such that each m-subset is equally likely to be created. One way would be to set $A[i] = i$ for $i = 1, 2, 3, \ldots, n$, call RANDOMIZED-IN-PLACE(A), and then take just the first m array elements. This method would make n calls to the RANDOM procedure. If n is mucn larger than m, we can create a random sample with fewer calls to RANDOM. Show that the following recursive procedure returns a random m-subset S of $\{1, 2, 3, \ldots, n\}$, in which each m-subset is equally likely, while making only m calls to RANDOM:

```
   Random-Sample(m, n)
1  | if m == 0 then
2  |   | return ∅
3  | else
4  |   | S = Random-Sample(m − 1, n − 1)
5  |   | i = Random(1, n)
6  |   | if i ∈ S then
7  |   |   | S = S ∪ {n}
8  |   | else
9  |   |   | S = S ∪ {i}
10 |   | return S
```

The recursion has $m + 1$ levels. Let R_k, for $k = 0, 1, \ldots, m$, denote the recursion at depth k, in which an k-subset is returned (R_0 returns the empty set; R_m returns the final m-subset). After R_k, S will consist of k elements from the set $\{1, 2, \ldots, n - (m - k)\}$. There are $\binom{n-(m-k)}{k}$ ways to choose k elements from an $(n - (m - k))$-set. Thus, to S be a random sample, we wish to show that, in each recursion level k, this particular k-subset is selected with probability $1/\binom{n-(m-k)}{k}$.

For the base case of the recursion, which occurs when $k = 0$, there are $\binom{n-m}{0} = 1$ distincts 0-subsets and the algorithm returns the empty set with probability $1 = 1/\binom{n-m}{0}$. Now assume R_{k-1} returns an random $(k - 1)$-sample. There are two ways to add the kth element to S on R_k:

- The element $n - (m - k)$ is added. This occurs when line 5 either selects the element $n - (m - k)$ or an element e such that $e \in R_{k-1}$. This probability is

$$\underbrace{\frac{1}{n - (m - k)}}_{(n-(m-k)) \text{ is selected}} + \underbrace{\frac{k - 1}{n - (m - k)}}_{e \in R_{k-1} \text{ is selected}} = \frac{k}{n - (m - k)}.$$

  Thus, R_k produces a particular k-sample with the element $n - (m - k)$ with probability

$$\begin{aligned}
\frac{k}{n - (m - k)} \cdot \frac{1}{\binom{n-(m-k)-1}{k-1}} &= \frac{k}{n - (m - k)} \cdot \left(\frac{(n - (m - k) - 1)!}{(k - 1)! \cdot (n - (m - k) - 1 - (k - 1))} \right)^{-1} \\
&= \left(\frac{(n - (m - k))!}{k! \cdot (n - (m - k) - k)} \right)^{-1} \\
&= \frac{1}{\binom{n-(m-k)}{k}}.
\end{aligned}$$

- An element $j < n - (m - k)$ is added. The probability of line 5 selecting such element is

$$\frac{n - (m - k) - k}{n - (m - k)} = \frac{n - m}{n - (m - k)}.$$

  Thus, R_k produces a particular k-sample with the element j with probability

$$\begin{aligned}
\frac{n - m}{n - (m - k)} \cdot \frac{1}{\binom{n-(m-k)-1}{k}} &= \frac{n - m}{n - (m - k)} \cdot \left(\frac{(n - (m - k) - 1)!}{k! \cdot (n - (m - k) - 1 - k)} \right)^{-1} \\
&= \left(\frac{(n - (m - k))!}{k! \cdot (n - (m - k) - k)} \right)^{-1} \\
&= \frac{1}{\binom{n-(m-k)}{k}}.
\end{aligned}$$

Since each recursion level R_k such that $k > 0$ makes exactly one call to RANDOM, there are m such calls. Also, among the $\binom{n}{m}$ ways of choosing m elements from an n-set, RANDOM-SAMPLE returns each of them with probability

$$\frac{1}{\binom{n-(m-m)}{m}} = \frac{1}{\binom{n}{m}}.$$

Problems

5-1 ***Probabilistic counting***

With a b-bit counter, we can ordinarily only count up to $2^b - 1$. With R. Morri's ***probabilistic counting***, we can count up to a much larger value at the expense of some loss of precision.

We let a counter value of i represent a count of n_i for $i = 0, 1, \ldots, 2^b - 1$, where the n_i form an increasing sequence of nonnegative values. We assume that the initial value of the counter is 0, representing a count of $n_0 = 0$. The INCREMENT operation works on a counter containing the value i in a probabilistic manner. If $i = 2^b - 1$, then the operation reports an overflow error. Otherwise, the INCREMENT operation increases the counter by 1 with probability $1/(n_{i+1} - n_i)$.

If we select $n_i = i$ for all $i \geq 0$, then the counter is an ordinary one. More interesting situations arise if we select, say, $n_i = 2^{i-1}$ for $i > 0$ or $n_i = F_i$ (the ith Fibonacci number – see Section 3.2).

For this problem, assume n_{2^b-1} is large enough that the probability of an overflow error is negligible.

a. Show that the expected value represented by the counter after n INCREMENT operations have been performed is exactly n.

b. The analysis of the variance of the count represented by the counter depends on the sequence of the n_i. Let us consider a simple case: $n_i = 100i$ for all $i \geq 0$. Estimate the variance in the value represented by the register after n INCREMENT operations have been performed.

(a) Let X_i denote a random variable for the expected *increment* of the count represented by a counter of value i after *one* INCREMENT operation. We have

$$\mathrm{E}[X_i] = 0 \cdot \left(1 - \frac{1}{n_{i+1} - n_i}\right) + (n_{i+1} - n_i) \cdot \frac{1}{n_{i+1} - n_i} = 1,$$

which shows that, independently from the current state of the *counter*, the expected *increment* of the *count* after each INCREMENT operation is always 1. Thus, after n INCREMENT operations, the expected *count* is:

$$\sum_{i=1}^{n} \mathrm{E}[X_0] = \sum_{i=1}^{n} 1 = n,$$

(b) We have

$$\begin{aligned}\mathrm{Var}[X_i] &= \mathrm{E}[{X_i}^2] - \mathrm{E}^2[X_i] \\ &= \left(0^2 \cdot \left(1 - \frac{1}{100}\right) + 100^2 \cdot \frac{1}{100}\right) - 1 \\ &= 99,\end{aligned}$$

which shows that the estimated variance after each INCREMENT operation does not depend on the current state of the *counter*. Thus, after n INCREMENT operations, the estimated variance is

$$\sum_{i=1}^{n} \mathrm{Var}[X_0] = \sum_{i=1}^{n} 99 = 99n.$$

5-2 ***Searching an unsorted array***

This problem examines three algorithms for searching for a value x in an unsorted array A consisting of n elements.

Consider the following randomized strategy: pick a random index i into A. If $A[i] = x$, then we terminate; otherwise, we continue the search by picking a new random index into A. We continue picking random indices into A until we find an index j such that $A[j] = x$ or until we have checked every element of A. Note that we pick from the whole set of indices each time, so that we may examine a given element more than once.

a. Write pseudocode for a procedure RANDOM-SEARCH to implement the strategy above. Be sure that your algorithm terminates when all indices into A have been picked.

b. Suppose that there is exactly one index i such that $A[i] = x$. What is the expected number of indices into A that we must pick before we find x and RANDOM-SEARCH terminates?

c. Generalizing your solution to part (b), suppose that there are $k \geq 1$ indices i such that $A[i] = x$. What is the expected number of indices into A that we must pick before we find x and RANDOM-SEARCH terminates? Your answer should be a function of n and k.

d. Suppose that there are no indices i such that $A[i] = x$. What is the expected number of indices into A that we must pick before we have checked all elements of A and RANDOM-SEARCH terminates?

Now consider a deterministic linear serach algorithm, which we refer to as DETERMINISTIC-SEARCH. Specifically, the algorithm searches A for x in order, considering $A[1], A[2], A[3], \ldots, A[n]$ until either it finds $A[i] = x$ or it reaches the end of the array. Assume that all possible permutations of the input array are equally likely.

e. Suppose that there is exactly one index i such that $A[i] = x$. What is the average-case running time of DETERMINISTIC-SEARCH? What is the worst-case running time of DETERMINISTIC-SEARCH?

f. Generalizing your solution to part (e), suppose that there are $k \geq 1$ indices i such that $A[i] = x$. What is the average-case running time of DETERMINISTIC-SEARCH? What is the worst-case running time of DETERMINISTIC-SEARCH? Your answer should be a function of n and k.

g. Suppose that there are no indices i such that $A[i] = x$. What is the average-case running time of DETERMINISTIC-SEARCH? What is the worst-case running time of DETERMINISTIC-SEARCH?

Finally, consider a randomized algorithm SCRAMBLE-SEARCH that works by first randomly permuting the input array and then running the deterministic linear search given above on the resulting permuted array.

h. Letting k be the number of indices i such that $A[i] = x$, give the worst-case and expected running times of SCRAMBLE-SEARCH for the cases in which $k = 0$ and $k = 1$. Generalize your solution to handle the case in which $k \geq 1$.

i. Which of the three searching algorithms would you use? Explain your answer.

(a) The pseudocode is stated below.

```
Random-Search(A, x)
1  | I = ∅
2  | n = A.length
3  | index = −1
4  | while |I| < n do
5  |   | i = Random(1, n)
6  |   | I = I ∪ {i}
7  |   | if A[i] == x then
8  |   |   | index = i
9  |   |   | break
10 | return index
```

(b) This can be viewed as a sequence of Bernoulli trials, each with a probability $p = 1/n$ of success. Let X be a random variable for the number of trials needed to pick i such that $A[i] = x$. From Equation (C.32), we have

$$\mathrm{E}[X] = \frac{1}{p} = n.$$

(c) This can also be viewed as a sequence of Bernoulli trials, but with a probability $p = k/n$ of success. Thus, we have

$$\mathrm{E}[X] = \frac{1}{p} = \frac{n}{k}.$$

(d) Let I be the set of indexes that was already checked. Let X_i be a random variable for the number of trials needed to pick an index i, for $i = 1, 2, \ldots, n$, such that $i \notin I$ and $|I| = i - 1$. This can be viewed as a sequence of Bernoulli trials. Thus, we have

$$p = \frac{n - |I|}{n} = \frac{n - i + 1}{n},$$

and

$$\mathrm{E}[X_i] = \frac{1}{p} = \frac{n}{n - i + 1}.$$

Now let X be a random variable for the number of trials to pick all elements of A. We have

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathrm{E}[X_i] \\
&= \sum_{i=1}^{n} \frac{n}{n - i + 1} \\
&= n \sum_{i=1}^{n} \frac{1}{n - i + 1} \\
&= n \sum_{i=0}^{n-1} \frac{1}{n - i} \\
&= n \sum_{i=1}^{n} \frac{1}{i} && (n\text{th harmonic number}) \\
&= n(\ln n + O(1)).
\end{aligned}$$

(e) Lets first consider the average case. Among the $n - 1$ elements that is not x, $(n - 1)/2$ of them are expected to be before the element x on the array. Thus, the expected running time of the algorithm is

$$\frac{n - 1}{2} + 1 = \frac{n + 1}{2}.$$

The worst-case occur when the number of elements before x is $n - 1$. In this case, the algorithm will make n checks.

(f) Let I be the set of indexes such that $i \in I \rightarrow A[i] = x$. For each element e such that $e \neq x$, there are $k + 1$ possibilities to position e with respect to I (before all elements of I, after one element of I, but before the remaining $k - 1$ elements of I, and so on). Each of these positions is equally likely. Therefore, among the $n - k$ elements that is not x, $(n - k) \cdot 1/(k + 1) = (n - k)/(k + 1)$ are expected to be before all the elements of I. Thus, the expected running time of the algorithm is

$$\frac{n - k}{k + 1} + 1 = \frac{(n - k) + (k + 1)}{k + 1} = \frac{n + 1}{k + 1}.$$

The worst-case occurs when the number of elements before the first x is $n - k$. In this case, the algorithm will make $n - k + 1$ checks.

(g) In every case, the algorithm will check all elements of A. Thus, there will be n checks.

(h) Suppose the algorithm uses Randomize-In-Place to randomize the input array. Independently from the value of k, the algorithm will take n on this operation. Thus, lets focus on the number of checks for each case. When $k = 0$, the algorithm will make exactly n checks in every case. Thus, it the expected running time is $n + n = 2n$. When $k = 1$, the behaviour of the algorithm is similar to the one of item (e). Thus, the expected running time is $n + (n+1)/2 = (3n+1)/2$. As for the worst-case, note that this notation refers to the distribution of inputs. Since for every input the expected running time is the same, the worst-case (over the inputs) is $n + (n + 1)/2 = (3n + 1)/2$. Similarly, for a given k and from item (f), both the expected running time and the worst-case is $n + (n + 1)/(k + 1)$.

(i) Deterministic-Search is better in all cases.

Section 6.1 – Heaps

6.1-1 What are the minimum and maximum numbers of elements in a heap of height h?

> Minimum is 2^h. Maximum is $2^{h+1} - 1$.

6.1-2 Show that an n-element heap has height $\lfloor \lg n \rfloor$.

> A heap of height $h + 1$ is a complete tree of height h plus one additional level with $1 \leq k \leq 2^h$ nodes. This additional level does not count to the height of the heap, which then explain the height of $\lfloor \lg n \rfloor$.

6.1-3 Show that in any subtree of a max-heap, the root of the subtree contains the largest value occurring anywhere in that subtree.

> Every node of the subtree has a path upwards to the root of the subtree. Therefore, the max-heap property assures that each of these nodes are no larger than the root of the subtree.

6.1-4 Where in a max-heap might the smallest element reside, assuming that all elements are distinct?

> In the leaves. Note that, since the bottom level may be incomplete, in addition to the nodes on level zero, some of the nodes on level one may also be leaves.

6.1-5 Is an array that is in sorted order a min-heap?

> Yes, since for each node i, we have $A[\textsc{Parent}(i)] \leq A[i]$.

6.1-6 Is the array with values $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 \rangle$ a max-heap?

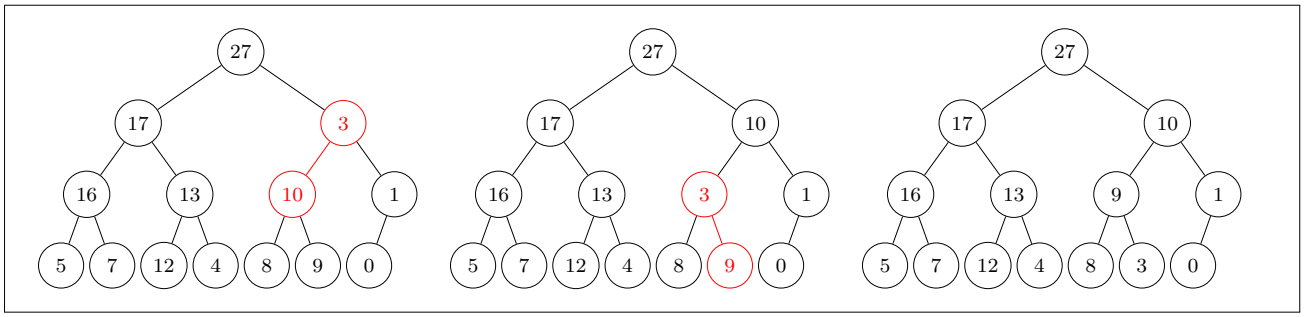
> No. The element 6 is the parent of the element 7 and $6 < 7$, which violates the min-heap property.

6.1-7 Show that, with the array representation for storing an n-element heap, the leaves are the nodes indexed by $\lfloor n/2 \rfloor + 1, \lceil n/2 \rceil + 2, \ldots, n$.

> The parent of the last element of the array is the element at position $\lfloor n/2 \rfloor$, which implies that all elements after $\lfloor n/2 \rfloor$ has no children and are therefore leaves. Also, since the element at position $\lfloor n/2 \rfloor$ has at least one child (the element at position n), the elements before $\lfloor n/2 \rfloor$ also have and therefore can not be leaves.

Section 6.2 – Maintaining the heap property

6.2-1 Using Figure 6.2 as a model, illustrate the operation of MAX-HEAPIFY$(A, 3)$ on the array $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0\rangle$.



6.2-2 Starting with the procedure MAX-HEAPIFY, write pseudocode for the procedure MIN-HEAPIFY(A, i), which performs the corresponding manipulation on a min-heap. How does the running time of MIN-HEAPIFY compare to that of MAX-HEAPIFY?

The pseudocode is stated below.

```
   Min-Heapify(A, i)
1  |  l = Left(i)
2  |  r = Right(i)
3  |  if l ≤ A.heap-size and A[l] < A[i] then
4  |  |  smallest = l
5  |  else
6  |  |  smallest = i
7  |  if r ≤ A.heap-size and A[r] < A[smallest] then
8  |  |  smallest = r
9  |  if smallest ≠ i then
10 |  |  exchange A[i] with A[smallest]
11 |  |  Min-Heapify(A, smallest)
```

The running time is the same.

6.2-3 What is the effect of calling MAX-HEAPIFY(A, i) when the element $A[i]$ is larger than its children?

Node i and its children already satisfies the max-heap property. No recursion will be called and the array will keep the same.

6.2-4 What is the effect of calling MAX-HEAPIFY(A, i) for $i > A.\textit{heap-size}/2$?

Every node $i > A.\textit{heap-size}/2$ is a leaf. No recursion will be called and the array will keep the same.

6.2-5 The code for MAX-HEAPIFY is quite efficient in terms of constant factors, except possibly for the recursive call in line 10, which might cause some compilers to produce inefficient code. Write an efficient MAX-HEAPIFY that uses an iterative control construct (a loop) instead of recursion.

The pseudocode is stated below.

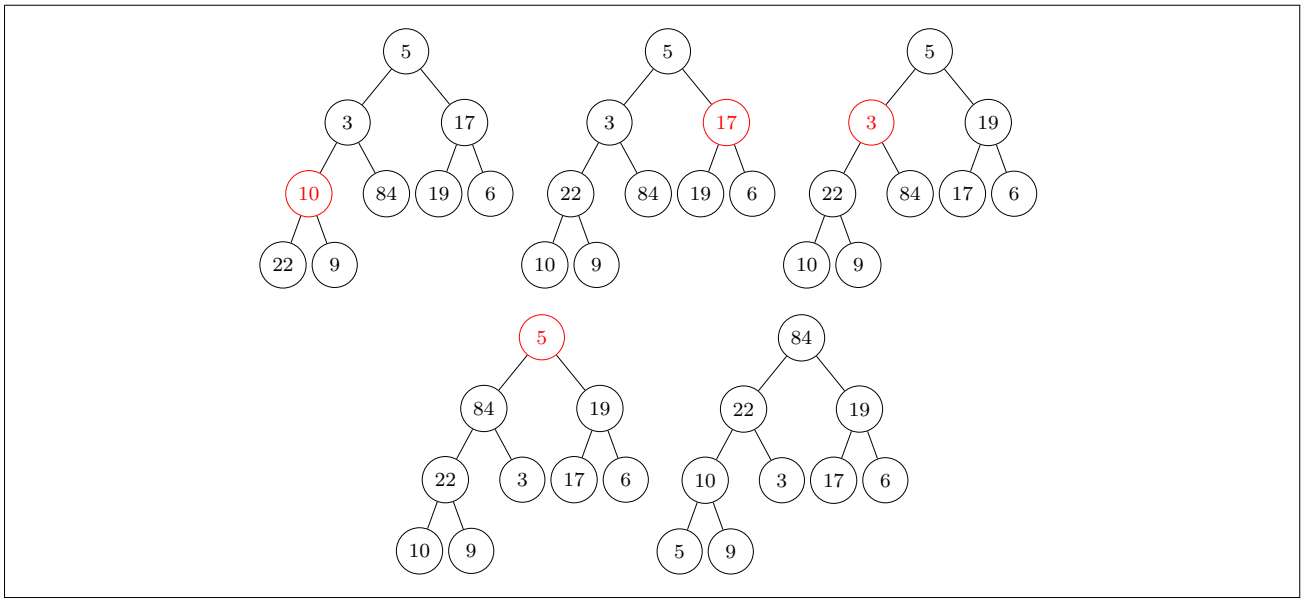
```
   Max-Heapify-Iterative(A, i)
 1 | solved = False
 2 | current-node = i
 3 | while not solved do
 4 | | l = Left(current-node)
 5 | | r = Right(current-node)
 6 | | if l ≤ A.heap-size and A[l] > A[current-node] then
 7 | | | largest = l
 8 | | else
 9 | | | largest = current-node
10 | | if r ≤ A.heap-size and A[r] > A[largest] then
11 | | | largest = r
12 | | if largest ≠ current-node then
13 | | | exchange A[current-node] with A[largest]
14 | | | current-node = largest
15 | | else
16 | | | solved = True
```

6.2-6 Show that the worst-case running time of MAX-HEAPIFY on a heap of size n is $\Omega(\lg n)$. (Hint: For a heap with n nodes, give node values that cause MAX-HEAPIFY to be called recursively at every node on a simple path from the root down to a leaf.)

The worst-case occurs when $A[\textsc{Left}(i)] \geq A[\textsc{Right}(i)] > A[i]$ in each level of the recursion, which will cause the node to be pushed to the leftmost position on the bottom level of the heap. There will be exactly $\lfloor \lg n \rfloor$ recursive calls (in addition to the first call). Since each call is $\Theta(1)$, the total running time is $\lfloor \lg n \rfloor \cdot \Theta(1) = \Theta(\lg n) = \Omega(\lg n)$.

Section 6.3 – Building a heap

6.3-1 Using Figure 6.3 as a model, illustrate the operation of BUILD-MAX-HEAP on the array $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9\rangle$.



6.3-2 Why do we want the loop index i in line 2 of BUILD-MAX-HEAP to decrease from $\lfloor A.length/2\rfloor$ to 1 rather than increase from 1 to $\lfloor A.length/2\rfloor$?

When we use MAX-HEAPIFY in a bottom-up manner, before each call to MAX-HEAPIFY(A, i), we can be sure that the subtrees rooted on its children are max-heaps and thus after exchanging $A[i]$ with $\max(A[\text{LEFT}(i)], A[\text{RIGHT}(i)])$, $A[i]$ will be the largest node among the nodes of the subtree rooted at i. In contrast, when we use MAX-HEAPIFY in a top-down manner, we can not be sure of that. For instance, if in a call to MAX-HEAPIFY(i), LEFT$(i) >$ RIGHT(i) and the largest node of the subtree rooted on i is on the subtree rooted on RIGHT(i), this largest element will never reach the position i, which will then violate the max-heap property.

6.3-3 Show that there are at most $\lceil n/2^{h+1}\rceil$ nodes of heigh h in any n-element heap.

From 6.1-7, we know that the leaves of a heap are the nodes indexed by

$$\lfloor n/2\rfloor + 1, \lfloor n/2\rfloor + 2, \ldots, n.$$

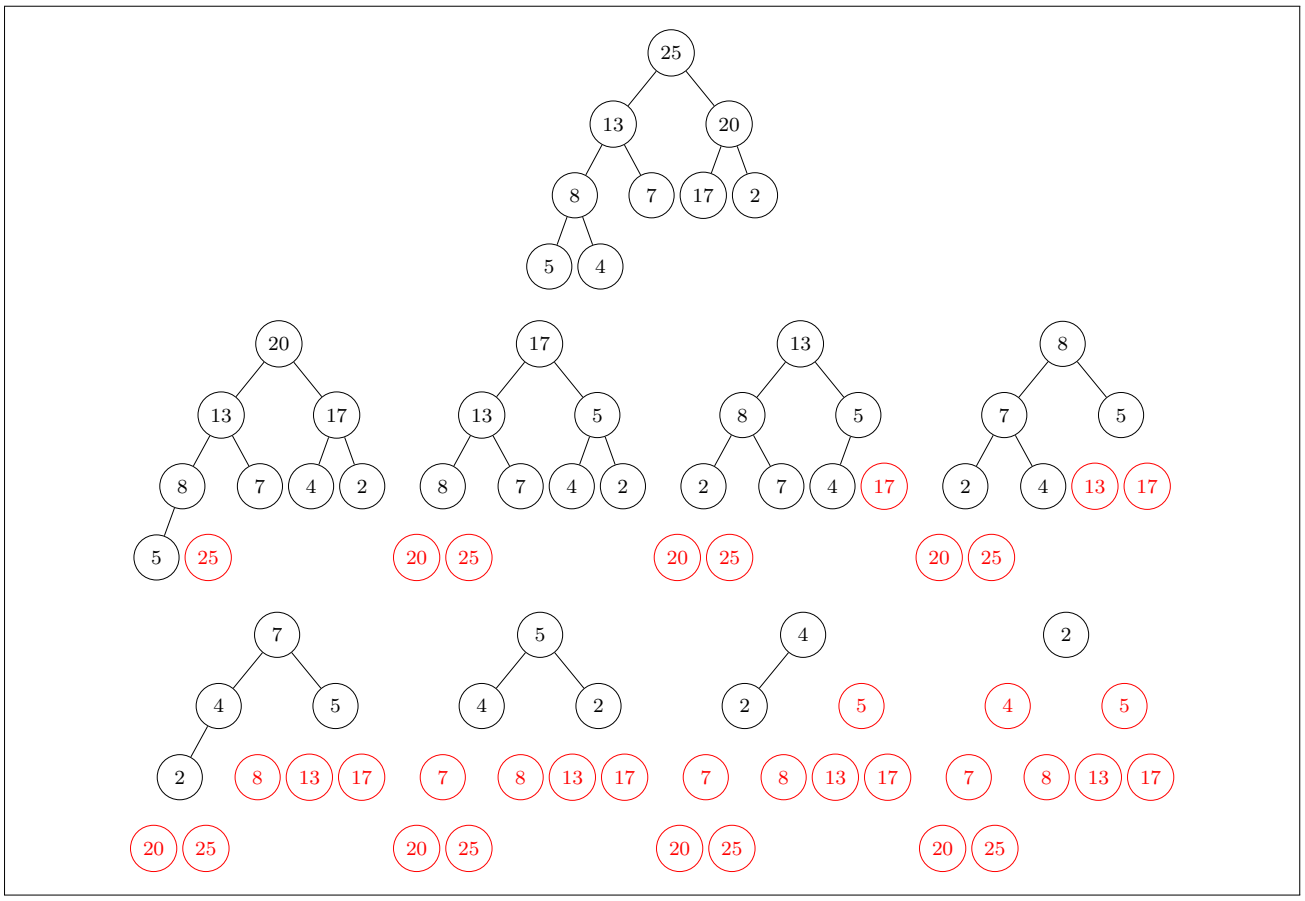
Note that those elements corresponds to the second half of the heap array (plus the middle element if n is odd). Thus, the number of leaves in any heap of size n is $\lceil n/2\rceil$. Lets prove by induction. Let n_h denote the number of nodes at height h. The upper bound holds for the base since $n_0 = \lceil n/2^{0+1}\rceil = \lceil n/2\rceil$ is exactly the number of leaves in a heap of size n. Now assume is holds for $h-1$. We shall prove that it also holds for h. Note that if n_{h-1} is even each node at height h has exactly two children, which implies $n_h = n_{h-1}/2 = \lceil n_{h-1}/2\rceil$. If n_{h-1} is odd, one node at height h has one child and the remaining has two children, which also implies $n_h = \lfloor n_{h-1}/2\rfloor + 1 = \lceil n_{h-1}/2\rceil$. Thus, we have

$$n_h = \left\lceil \frac{n_{h-1}}{2} \right\rceil \leq \left\lceil \frac{1}{2} \cdot \left\lceil \frac{n}{2^{(h-1)+1}} \right\rceil \right\rceil = \left\lceil \frac{1}{2} \cdot \left\lceil \frac{n}{2^h} \right\rceil \right\rceil = \left\lceil \frac{n}{2^{h+1}} \right\rceil,$$

which shows that it also holds for h.

Section 6.4 – The heapsort algorithm

6.4-1 Using Figure 6.4 as a model, illustrate the operation of HeapSort on the array $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$.



6.4-2 Argue the correctness of HeapSort using the following loop invariant:

> At the start of each iteration of the **for** loop of line 2–5, the subarray $A[1 \ldots i]$ is a max-heap containing the i smallest elements of $A[1 \ldots n]$, and the subarray $A[i+1 \ldots n]$ contains the $n - i$ largest elements of $A[1 \ldots n]$, sorted.

We need to show that this invariant is true prior to the first loop iteration, that each iteration of the loop maintains the invariant, and that the invariant provides a useful property to show correctness when the loop terminates.

- **Initialization.** Before the **for** loop, $i = n$ and line 1 assures A is a max-heap. Thus, $A[1, \ldots, i] = A$ is a max-heap containing the i smallest elements of A and $A[i+1, \ldots, n] = \emptyset$ contains the $n - i = 0$ largest elements of A, sorted.
- **Maintenance.** By the loop invariant, $A[1, \ldots, i]$ is a max-heap containing the i smallest elements of A, which implies that $A[1]$ is the ith smallest element of A. Since $A[i+1, \ldots, n]$ already contains the $n - i$ largest elements of A in sorted order, after exchanging $A[1]$ with $A[i]$, the subarray $A[i, \ldots, n]$ now contains the $n - i + 1$ largest elements of A in sorted order. Lines 4-5 maintains the max-heap property on the subarray $A[1, \ldots, i-1]$ and decrement i for the next iteration preserves the loop invariant.
- **Termination.** At termination $i = 1$ and the subarray $A[2 \ldots n]$ contains the $n - 1$ smallest elements of A in sorted order, which also implies that $A[1, \ldots, n]$ is fully sorted.

6.4-3 What is the running time of HeapSort on an array A of length n that is already sorted in increasing order? What about decreasing order?

Since an array that is sorted in increasing order is not a max-heap, Build-Max-Heap will break that ordering. The Build-Max-Heap procedure will take $\Theta(n)$ to build the max-heap and the **for** loop of lines 2−5 will take $O(n \lg n)$, which gives a total running time of $\Theta(n) + O(n \lg n) = O(n \lg n)$.

An array sorted in drecreasing order is already a max-heap, but even in that case Build-Max-Heap will take $\Theta(n)$. Note that, although the input is sorted in decreasing order, the intent of the algorithm is to sort in increasing order. In each iteration of the **for** loop of lines 2−5, it will exchange $A[1]$ with $A[i]$ and will call Max-Heapify on $A[1]$. Since $A[1]$ is not anymore the largest element of A, each call to Max-Heapify may cover the entire height of the heap and thus will take $O(\lg n)$. Thus, the **for** loop of lines 2−5 will run in $O(n \lg n)$ and the algorithm will take $\Theta(n) + O(n \lg n) = O(n \lg n)$.

6.4-4 Show that the worst-case running time of HeapSort is $\Omega(n \lg n)$.

The worst-case is when every call to Max-Heapify covers the entire height of the heap. In that case, HeapSort will take

$$\sum_{i=1}^{n-1} \lfloor \lg i \rfloor \leq \sum_{i=1}^{n-1} \lg i = \lg((n-1)!) = \Theta((n-1)\lg(n-1)) = \Omega(n \lg n).$$

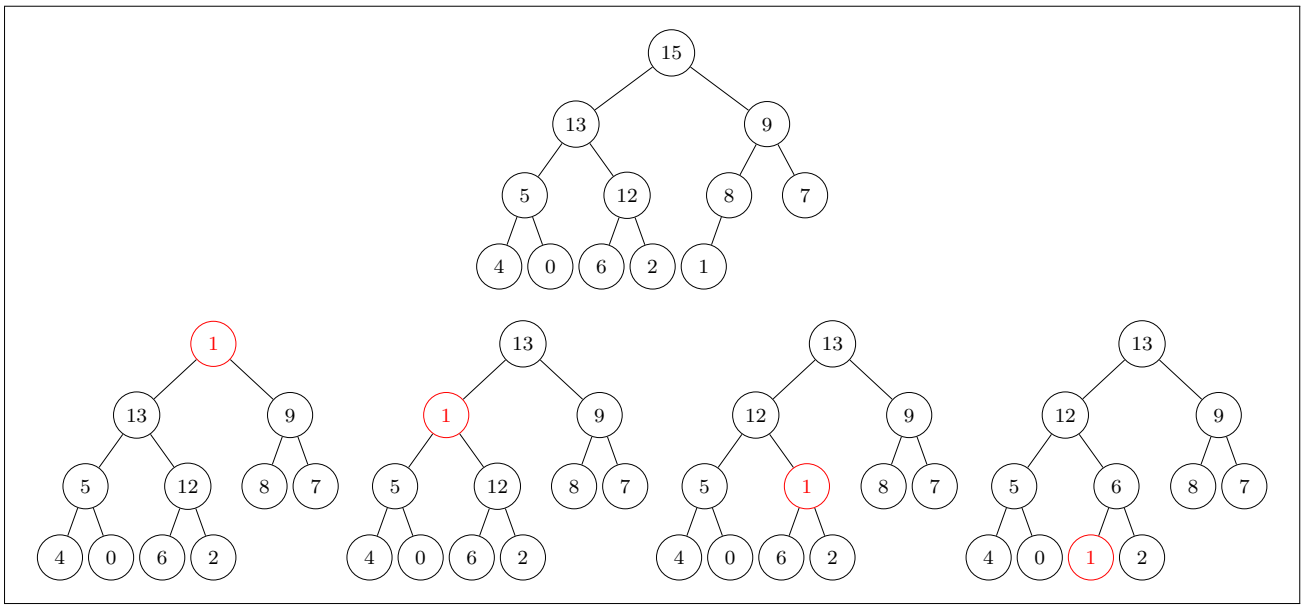
6.4-5 ($\star$) Show that when all elements are distinct, the best-case running time of HeapSort is $\Omega(n \lg n)$.

Proof on (Theorem 1, Page 86):

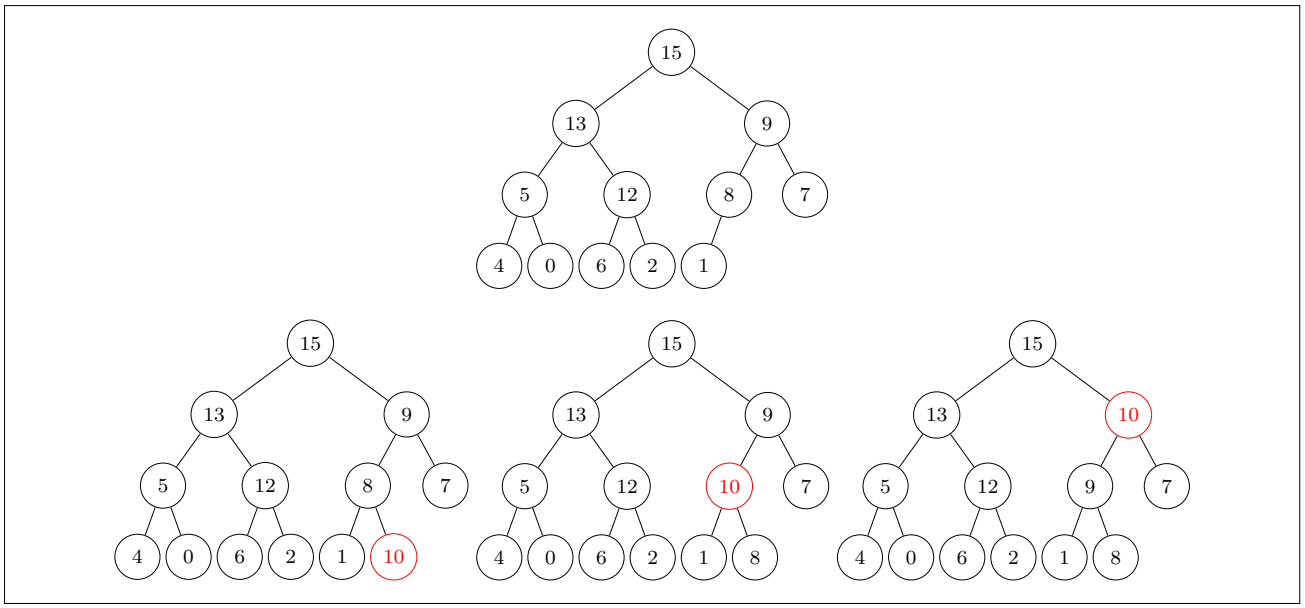
Schaffer, Russel, and Robert Sedgewick. "The analysis of heapsort." *Journal of Algorithms* 15.1 (1993): 76-100.

Section 6.5 – Priority queues

6.5-1 Illustrate the operation of HEAP-EXTRACT-MAX on the heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.



6.5-2 Illustrate the operation of MAX-HEAP-INSERT$(A, 10)$ on the heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.



6.5-3 Write pseudocode for the procedures HEAP-MINIMUM, HEAP-EXTRACT-MIN, HEAP-DECREASE-KEY, and MIN-HEAP-INSERT that implement a min-priority queue with a min-heap.

The pseudocodes are stated below.

```
   Heap-Minimum(A)
1  |  return A[1]

   Heap-Extract-Min(A)
1  |  if A.heap-size < 1 then
2  |  |  error "heap underflow"
3  |  min = A[1]
4  |  A[1] = A[A.heap-size]
5  |  A.heap-size = A.heap-size − 1
6  |  Min-Heapify(A, 1)
7  |  return min

   Heap-Decrease-Key(A, i, key)
1  |  if key > A[i] then
2  |  |  error "new key is larger than current key"
3  |  A[i] = key
4  |  while i > 1 and A[Parent(i)] > A[i] do
5  |  |  exchange A[i] with A[Parent(i)]
6  |  |  i = Parent(i)

   Min-Heap-Insert(A, key)
1  |  A.heap-size = A.heap-size + 1
2  |  A[A.heap-size] = +∞
3  |  Heap-Decrease-Key(A, A.heap-size, key)
```

6.5-4 Why do we bother setting the key of the inserted node to $-\infty$ in line 2 of MAX-HEAP-INSERT when the next thing we do is increase its key to the desired value?

Beacause the HEAP-INCREASE-KEY procedure requires that the new key is greater than or equal to the current key.

6.5-5 Argue the correctness of HEAP-INCREASE-KEY using the folowing loop invariant:

> At the start of each iteration of the **while** loop of lines 4-6, $A[\text{PARENT}(i)] \geq A[\text{LEFT}(i)]$ and $A[\text{PARENT}(i)] \geq A[\text{RIGHT}(i)]$, if these nodes exist, and the subarray $A[1, \ldots, A.\textit{heap-size}]$ satisfies the max-heap property, except that there may be one violation: $A[i]$ may be larger than $A[\text{PARENT}(i)]$.

You may assume that subarray $A[1, \ldots, A.\textit{heap-size}]$ satisfies the max-heap property at the time HEAP-INCREASE-KEY is called.

We need to show that this invariant is true prior to the first loop iteration, that each iteration of the loop maintains the invariant, and that the invariant provides a useful property to show correctness when the loop terminates.

- **Initialization.** Before the **while** loop, A is a valid max-heap with a possible change on the value of the element $A[i]$. Thus, the invariants $A[\text{PARENT}(i)] \geq A[\text{LEFT}(i)]$ and $A[\text{PARENT}(i)] \geq A[\text{RIGHT}(i)]$ holds (these values were not changed before the loop). Since the new value of $A[i]$ is equal or larger its previous value and its previous value was equal or larger than the value of its children (A was a max-heap at the time HEAP-INCRESE-KEY was called), the only possible violation on the heap is that $A[i]$ may be larger than $A[\text{PARENT}(i)]$, thus the second invariant also holds.
- **Maintenance.** By the loop invariant, the only possible violation is that $A[i]$ may be larger than $A[\text{PARENT}(i)]$. If there is no violation ($A[i]$ does not have a parent of $A[i] \leq A[\text{PARENT}(i)]$), the loop terminates and A is a valid max-heap. If there is a violation on $A[i]$, the positions of $A[i]$ and $A[\text{PARENT}(i)]$ are exchanged. From the loop invariant, before the exchange, $A[\text{PARENT}(i)] \geq A[\text{LEFT}(i)]$ and $A[\text{PARENT}(i)] \geq A[\text{RIGHT}(i)]$, which implies that, after the exchange, the new $A[i]$ will not violate the max-heap property anymore and the invariants $A[\text{PARENT}(i)] \geq A[\text{LEFT}(i)]$ and $A[\text{PARENT}(i)] \geq A[\text{RIGHT}(i)]$ will remain valid. The only possible violation after the exchange is that $A[\text{PARENT}(i)]$ may be larger than $A[\text{PARENT}(\text{PARENT}(i))]$, but setting $i = \text{PARENT}(i)$ preserves the loop invariant for the next iteration.
- **Termination.** At termination, either $i = 1$ or $A[i] \leq A[\text{PARENT}(i)]$. In both cases, $A[i]$ is not larger than $A[\text{PARENT}(i)]$, which implies that A is a valid max-heap.

6.5-6 Each exchange operation on line 5 of HEAP-INCREASE-KEY typically requires three assignments. Show how to use the idea of the inner loop of INSERTION-SORT to reduce the three assignments down to just one assignment.

The updated pseudocode is stated below.

```
   Heap-Increase-Key(A, i, key)
1  | if key < A[i] then
2  |   | error "new key is smaller than current key"
3  | while i > 1 and A[Parent(i)] < key do
4  |   | A[i] = A[Parent(i)]
5  |   | i = Parent(i)
6  | A[i] = key
```

6.5-7 Show how to implement a first-in, first-out queue with a priority queue. Show how to implement a stack with a priority queue. (Queues and stacks are defined in Section 10.1.)

A first-in, first-out queue can be implemented using a min-priority-queue, in such a way that each heap element is a tuple (key, handle) and the key of a new element is greater than the key of the current elements. The EXTRACT-MIN operation will always return the oldest element (minimum key value) and the INSERT operation will keep the min-heap property. A stack can be implemented similarly, but with a max-priority-heap instead of a min-priority-heap.

6.5-8 The operation HEAP-DELETE(A, i) deletes the item in node i from heap A. Give an implementation of HEAP-DELETE that runs in $O(\lg n)$ time for an n-element max-heap.

The pseudocode is stated below.

```
   Heap-Delete(A, i)
1  | A[i] = A[heap-size]
2  | A.heap-size = A.heap-size − 1
3  | Max-Heapify(A, i)
```

6.5-9 Give an $O(n \lg k)$-time algorithm to merge k sorted lists into one sorted list, where n is the total number of elements in all the input lists. (Hint: Use a min-heap for k-way merging.)

Let T denote the final sorted list and S_i the ith input list. The pseudocode is stated below.

```
   Merge-Lists-Min-Heap(S_1, S_2, ..., S_k)
1  | Let T be a list
2  | Let H be a min-heap of tuples in the form (key, j)
3  | for i = 1 to k do
4  |   | Insert(H, (S_i[1], i))
5  |   | p_i = 2
6  | while H.heap-size > 0 do
7  |   | (key, j) = Extract-Min(H)
8  |   | add key to T
9  |   | if p_j ≤ S_j.length then
10 |   |   | Insert(H, (S_j[p_j], j))
11 |   |   | p_j = p_j + 1
12 | return T
```

The **for** loop of lines 3-5 runs in $O(k \lg k)$. The **while** loop of lines 6-11 will iterate n times and each iteration takes $O(\lg k)$. Since $n \geq k$, the algorithm runs in $O(n \lg k)$.

Problems

6-1 ***Building a heap using insertion***

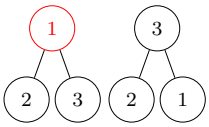
We can build a heap by repeatedly calling MAX-HEAP-INSERT to insert the elements into the heap. Consider the following variation on the BUILD-MAX-HEAP procedure:

```
Build-Max-Heap'(A)
1  A.heap-size = 1
2  for i = 2 to A.length do
3      MAX-HEAP-INSERT(A, A[i])
```

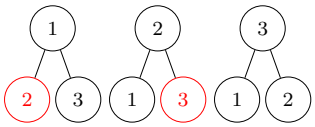
a. Do the procedures BUILD-MAX-HEAP and BUILD-MAX-HEAP′ always create the same heap when run on the same input array? Prove that they do, or provide a counterexample.

b. Show that in the worst-case, BUILD-MAX-HEAP′ requires $\Theta(n \lg n)$ time to build an n-element heap.

(a) No. Consider the array $A = [1, 2, 3]$. BUILD-MAX-HEAP(A) will create the heap



while BUILD-MAX-HEAP′(A) will create the heap



(b) Let A be the input array. The worst-case occurs when there is an integer k such that $n = A.length = 2^k - 1$ (that is, the elements of A forms a complete binary tree) and each call to MAX-HEAP-INSERT covers the entire height of the heap (occurs when A is sorted). Since any heap has $\lceil n/2 \rceil$ leaves, the last $\lceil n/2 \rceil$ elements of A will be the leaves of the final heap. Note that, at the time MAX-HEAP-INSERT is called on these last $\lceil n/2 \rceil$ nodes, the height of the heap will be $\lfloor \lg n \rfloor$. Thus, considering only the last $\lceil n/2 \rceil$ calls to MAX-HEAP-INSERT, the algorithm will take $\lceil n/2 \rceil \cdot \Theta(\lfloor \lg n \rfloor) = \Theta(n \lg n)$, which implies that the worst-case of BUILD-MAX-HEAP′ runs in $\Omega(n \lg n)$. Also, note that there will be exactly $n - 1$ calls to MAX-HEAP-INSERT and each call takes $O(\lfloor \lg n \rfloor)$. Thus, the worst-case of BUILD-MAX-HEAP′ runs in $\Theta(n \lg n)$.

6-2 ***Analysis of d-ary heaps***

A ***d-ary*** heap is like a binary heap, but (with one possible exception) non-leaf nodes have d children instead of 2 children.

a. How would you represent a d-ary heap in an array?

b. What is the height of a d-ary heap of n elements in terms of n and d?

c. Give an efficient implementation of EXTRACT-MAX in a d-ary max-heap. Analyze its running time in terms of d and n.

d. Give an efficient implementation of INSERT in a d-ary max-heap. Analyze its running time in terms of d and n.

e. Give an efficient implementation of INCREASE-KEY(A, i, k), which flags an error if $k < A[i]$, but otherwise sets $A[i] = k$ and then updates the d-ary max-heap structure appropriately. Analyze its running time in terms of d and n.

(a) The root occupies the first $d^0 = 1$ positions of the array, its children occupies the next $d^1 = d$ positions of the array, and so on until the bottom level. That way, the parent of a node with index i will be at position

$$\left\lfloor \frac{i-(d-2)}{d} \right\rfloor = \left\lfloor \frac{i-d+2}{d} \right\rfloor,$$

and its jth children will be at position

$$di - (d-j) + 1 = di - d + j + 1 = d(i-1) + j + 1.$$

(b) $h = \lfloor \log_d n \rfloor$.

(c) The only major modification is the MAX-HEAPIFY procedure. The pseudocode is stated below.

```
Child-D-Ary(d, i, j)
1 | return d(i − 1) + j + 1

Extract-Max-D-Ary(A, d)
1 | if A.heap-size < 1 then
2 | | error "heap underflow"
3 | max = A[1]
4 | A.heap-size = A.heap-size − 1
5 | Max-Heapify-D-Ary(A, d, 1)
6 | return max

Max-Heapify-D-Ary(A, d, i)
1 | largest = i
2 | for j = 1 to d do
3 | | child-j = Child(d, i, j)
4 | | if child-j > A.heap-size then
5 | | | break
6 | | if A[child-j] > A[largest] then
7 | | | largest = child-j
8 | if largest ≠ i then
9 | | exchange A[i] with A[largest]
10 | | Max-Heapify-D-Ary(A, d, largest)
```

In the worst-case, MAX-HEAPIFY-D-ARY covers the height of the heap (which is $\log_d n$). Since each recursive call takes $O(d)$, MAX-HEAPIFY-D-ARY runs in $O(d \log_d n)$. The running time of EXTRACT-MAX-D-ARY is also $O(d \log_d n)$, since it performs a constant amount of work on top of MAX-HEAPIFY-D-ARY.

(d) The MAX-HEAP-INSERT procedure given in the text book also works for a d-ary heap. The only modification is to use PARENT-D-ARY instead of PARENT in the HEAP-INCREASE-KEY subroutine. In the worst case, it will cover the height of the tree. Thus, the running time is $O(\log_d n)$.

(e) The HEAP-INCREASE-KEY procedure given in the text book also works for a d-ary heap. The only modification is to use PARENT-D-ARY instead of PARENT. In the worst case, it will cover the height of the tree. Thus, the running time is $O(\log_d n)$.

6-3 ***Young tableaus***

An $m \times n$ ***Young tableau*** is an $m \times n$ matrix such that the entries of each row are in sorted order from left to right and the entries of each column are in sorted order from top to bottom. Some of the entries of a Young tableau may be ∞, which we treat as nonexistent elements. Thus, a Young tableau can be used to hold $r \leq mn$ finite numbers.

a. Draw a 4×4 Young tableau containing the elements $\{9, 16, 3, 2, 4, 8, 5, 14, 12\}$.

b. Argue that an $m \times n$ Young tableau Y is empty if $Y[1,1] = \infty$. Argue that Y is full (contains mn elements) if $Y[m,n] < \infty$.

c. Give an algorithm to implement Extract-Min on a non-empty $m \times n$ Young tableau that runs in $O(m+n)$ time. Your algorithm should use a recursive subroutine that solves an $m \times n$ problem by recursively solving either an $(m-1) \times n$ or an $m \times (n-1)$ subproblem. (Hint: Think about Max-Heapify.) Define $T(p)$, where $p = m + n$, to be the maximum running time of Extract-Min on any $m \times n$ Young tableau. Give and solve a recurrence for $T(p)$ that yields the $O(m+n)$ time bound.

d. Show how to insert a new element into a nonfull $m \times n$ Young tableau in $O(m+n)$ time.

e. Using no other sorting method as a subroutine, show how to use an $n \times n$ Young tableau to sort n^2 numbers in $O(n^3)$ time.

f. Give an $O(m+n)$-time algorithm to determine whether a given number is stored in a given $m \times n$ Young tableau.

(a) Here it is:

$$\begin{array}{cccc} 2 & 3 & 4 & 5 \\ 8 & 9 & 12 & 14 \\ 16 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \end{array}$$

(b) From the definition of a Young tableau, we have

$$Y[1,1] \leq Y[i,j] \ \forall i, j,$$

which implies

$$Y[1,1] = \infty \rightarrow Y[i,j] = \infty \ \forall i, j.$$

Thus, Y is full of ∞ and is therefore empty.

From the definition of a Young tableau, we have

$$Y[m,n] \geq Y[i,j] \ \forall i, j,$$

which implies

$$Y[m,n] < \infty \rightarrow Y[i,j] < \infty \ \forall i, j.$$

Thus, Y does not have any ∞ and is therefore full.

(c) The pseudocode is stated below.

```
Extract-Min(Y)
1  | min = Y[1,1]
2  | if min == ∞ then
3  |   | error "tableau underflow"
4  | Y[1,1] = ∞
5  | Min-Tableauify(Y, 1, 1)
6  | return min
```

```
Min-Tableauify(Y, i, j)
1  | smallest-i = i
2  | smallest-j = j
3  | if i + 1 ≤ Y.rows and Y[i+1, j] < Y[smallest-i, smallest-j] then
4  |   | smallest-i = i + 1
5  | if j + 1 ≤ Y.cols and Y[i, j+1] < Y[smallest-i, smallest-j] then
6  |   | smallest-i = i
7  |   | smallest-j = j + 1
8  | if i ≠ smallest-i or j ≠ smallest-j then
9  |   | Y[i,j] = Y[smallest-i, smallest-j]
10 |   | Y[smallest-i, smallest-j] = ∞
11 |   | Min-Tableauify(Y, smallest-i, smallest-j)
```

The algorithm has the recurrence $T(p) \leq T(p-1) + \Theta(1) = O(p) + O(m+n)$.

(d) The pseudocode is stated below.

```
Tableau-Insert(Y, key)
1  | i = Y.rows
2  | j = Y.cols
3  | if Y[i, j] == ∞ then
4  |   | error "tableau overflow"
5  | Y[i, j] = key
6  | while i > 1 or j > 1 do
7  |   | if i > 1 and Y[i, j] < Y[i − 1, j] then
8  |   |   | exchange Y[i, j] with Y[i − 1, j]
9  |   |   | i = i − 1
10 |   | else if j > 1 and Y[i, j] < Y[i, j − 1] then
11 |   |   | exchange Y[i, j] with Y[i, j − 1]
12 |   |   | j = j − 1
13 |   | else
14 |   |   | break
```

We first set the element to the position $A[m, n]$ and move it upwards and/or leftwards until we find a valid position. The running time is $O(m + n)$

(e) The pseudocode is stated below.

```
Build-Young-Tableau(A)
1  | n = √A.length
2  | Let Y be an n × n array
3  | for i = 1 to A.length do
4  |   | Tableau-Insert(Y, A[i])
5  | return Y
```

```
Tableausort(A)
1  | Y = Build-Young-Tableau(A)
2  | for i = 1 to A.length do
3  |   | A[i] = Extract-Min(Y)
```

The Build-Young-Tableau procedure runs in $n^2 \cdot O(n+n) = O(n^3)$. Each call to Min-Tableauify runs in $O(n+n) = O(n)$. Thus, the algorithm runs in $O(n^3) + n^2 \cdot O(n) = O(n^3)$.

(f) The pseudocode is stated below.

```
Tableau-Find(Y, key)
1  | i = Y.rows
2  | j = 1
3  | while i ≥ 1 and j ≤ Y.cols do
4  |   | if Y[i, j] == key then
5  |   |   | return True
6  |   | else if Y[i, j] ≤ Y[i − 1, j] then
7  |   |   | i = i − 1
8  |   | else if Y[i, j] ≥ Y[i, j + 1] then
9  |   |   | j = j + 1
10 |   | else
11 |   |   | break
12 |   | return False
```

Section 7.1 – Description of quicksort

7.1-1 Using Figure 7.1 as a model, illustrate the operation of PARTITION on the array $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11 \rangle$.

13	19	9	5	12	8	7	4	21	2	6	11
13	19	9	5	12	8	7	4	21	2	6	11
13	19	9	5	12	8	7	4	21	2	6	11
9	19	13	5	12	8	7	4	21	2	6	11
9	5	13	19	12	8	7	4	21	2	6	11
9	5	13	19	12	8	7	4	21	2	6	11
9	5	8	19	12	13	7	4	21	2	6	11
9	5	8	7	12	13	19	4	21	2	6	11
9	5	8	7	4	13	19	12	21	2	6	11
9	5	8	7	4	13	19	12	21	2	6	11
9	5	8	7	4	2	19	12	21	13	6	11
9	5	8	7	4	2	6	12	21	13	19	11
9	5	8	7	4	2	6	11	21	13	19	12

7.1-2 What value of q does PARTITION return when all elements in the array $A = [p, \ldots, r]$ have the same value? Modify PARTITION so that $q = \lfloor (p + r)/2 \rfloor$ when all elements in the array $A[p, \ldots, r]$ have the same value.

It will return $q = r$. We can update PARTITION to split elements that are equal to the pivot on both sides as follows:

(a) Count the number of elements y such that $y = x$ and set this value to c;

(b) Subtract the final pivot index by $\lfloor c/2 \rfloor$.

The updated pseucode is stated below.

```
   Partition-Improved(A, p, r)
1  | x = A[r]
2  | i = j = p − 1
3  | for k = p to r − 1 do
4  |   | if A[k] ≤ x then
5  |   |   | j = j + 1
6  |   |   | exchange A[j] with A[k]
7  |   |   | if A[j] < x then
8  |   |   |   | i = i + 1
9  |   |   |   | exchange A[i] with A[j]
10 | exchange A[j + 1] with A[r]
11 | q = ⌊((i+1)+(j+1))/2⌋
12 | return q
```

7.1-3 Give a brief argument that the running time of PARTITION on a subarray of size n if $\Theta(n)$.

The **for** loop of lines 3–6 iterates $n - 1$ times and each iteration does a constant amount of work. Thus, it is $O(n)$.

7.1-4 How would you modify QUICKSORT to sort into nonincreasing order?

We just need to update the condition

$$A[j] \leq x,$$

to

$$A[j] \geq x.$$

Section 7.2 – Performance of quicksort

7.2-1 Use the substitution method to prove the recurrence $T(n) = T(n-1) + \Theta(n)$ has the solution $T(n) = \Theta(n^2)$, as claimed at the beginning of Section 7.2.

Our guess is

$$T(n) \leq cn^2 - dn \ \forall\, n \geq n_0,$$

where c, d, and n_0 are positive constants. Substituting into the recurrence, yields

$$\begin{aligned} T(n) &\leq c(n-1)^2 - d(n-1) + en \\ &= cn^2 - 2cn + c - d(n-1) + en \quad (c = 1, d = 2e) \\ &\leq cn^2, \end{aligned}$$

where the last step holds as long as $n_0 \geq 2$.

7.2-2 What is the running time of QUICKSORT when all elements of array A have the same value?

As discussed in (7.1-2), when all elements are the same, q will always be equal to r, which gives the worst-case split. Thus, QUICKSORT as implemented in Section 7.1, will run in $\Theta(n^2)$ in this case.

7.2-3 Show that the running time of QUICKSORT is $\Theta(n^2)$ when the array A contains distinct elements and is sorted in decreasing order.

The pivot index q will always be 1, which gives a 0 to $n-1$ split. The recurrence will be $T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$.

7.2-4 Banks often record transactions on an account in order of the times of the transactions, but many people like to receive their bank statements with checks listed in order by check number. People usually write checks in order by check number, and merchants usually cash them with reasonable dispatch. The problem of converting time-of-transaction ordering to check-number ordering is therefore the problem of sorting almost-sorted input. Argue that the procedure INSERTION-SORT would tend to beat the procedure QUICKSORT on this problem.

Lets assume that each item is out of order by no more than k positions. Note that in the above scenario, k usually can be bounded by a constant. In this case, INSERTION-SORT runs in $O(kn)$ (it will make at most k swaps for each item of the array), which is close to linear for small k. On the other hand, *most* splits given by the PARTITION procedure will be no better than a $k-1$ to $n-k$ split. Assuming that it always give an $k-1$ to $n-k$ split, the recurrence of QUICKSORT will be $T(n) = T(k) + T(n-k) + \Theta(n)$, which is close to quadratic for small k.

7.2-5 Suppose that the splits at every level of quicksort are in the proportion $1-\alpha$ to α, where $0 < \alpha \leq 1/2$ is a constant. Show that the minimum depth of a leaf in the recursion tree is approximately $-\lg n / \lg \alpha$ and the maximum depth is approximately $-\lg n / \lg(1-\alpha)$. (Don't worry about integer round-off.)

Note that

$$\alpha \leq \frac{1}{2} \leq 1 - \alpha,$$

which implies $\alpha n \leq (1-\alpha)n$. Thus, the minimum depth occurs on the path from which the problem size is always divided by $1/\alpha$. This depth is the number of divisions of n by $(1/\alpha)$ until reaching a value less than of equal to one, which is

$$\log_{1/\alpha} n = \frac{\lg n}{\lg(1/\alpha)} = \frac{\lg n}{-\lg \alpha} = -\frac{\lg n}{\lg \alpha}.$$

The maximum depth occurs on the path from which the problem size is always divided by $1/(1-\alpha)$. This depth is the number of divisions of n by $1/(1-\alpha)$ until reaching a value less than or equal to one, which is

$$\log_{1/(1-\alpha)} n = \frac{\lg n}{\lg(1/(1-\alpha))} = \frac{\lg n}{-\lg(1-\alpha)} = -\frac{\lg n}{\lg(1-\alpha)}.$$

7.2-6 ($\star$) Argue that for any constant $0 < \alpha \leq 1/2$, the probability is approximately $1 - 2\alpha$ that on a random input array, PARTITION produces a split more balanced than $1 - \alpha$ to α.

Note that α denotes the proportion of the smallest split. Since the input array is random, the possible proportions for the smallest split forms a uniform probability distribution, such that

$$\Pr\left\{\left[0, \frac{1}{2}\right]\right\} = 1.$$

Thus, the probability of getting a more balanced split is

$$\begin{aligned}\Pr\left\{\left(\alpha, \frac{1}{2}\right]\right\} &= \Pr\left\{\left[\alpha, \frac{1}{2}\right]\right\} \\ &= \frac{1/2 - \alpha}{1/2 - 0} \\ &= \frac{1/2}{1/2} - \frac{\alpha}{1/2} \\ &= 1 - 2\alpha.\end{aligned}$$

Section 7.3 – A randomized version of quicksort

7.3-1 Why do we analyze the expected running time of a randomized algorithm and not its worst-case running time?

> We can analyze the worst-case. However, due to the randomization, it is not very useful since we can not associate a specific input to a specific running time. On the other hand, we can calculate the expected running time, which takes into account all the possible inputs.

7.3-2 When Randomized-Quicksort runs, how many calls are made to the random-number generator Random in the worst case? How about in the best case? Give your answer in terms of Θ-notation.

> First note that counting the number of calls to Random is the same as counting number of calls to Partition.
>
> The worst-case occurs when Partition always gives an $(n-1)$-to-0 split. Note that after the first $n-1$ pivots are selected, the remaining subarray will contain a single element. Since Partition is only called on subarrays of size greater than one, in the worst-case the number of calls to Partition is $n-1=\Theta(n)$.
>
> As for the best case, consider the array $A=[1,2,3]$. If the element 2 is the first to be selected as a pivot, the subarrays [1] and [3] will not be passed to Partition (both of them has size one) and the number of calls to Partition in this case is 1. In general, in the best-case the number of calls to Partition is $\lfloor n/2 \rfloor = \Theta(n)$.

Section 7.4 – Analysis of quicksort

7.4-1 Show that in the recurrence

$$T(n) = \max_{0 \le q \le n-1}(T(q) + T(n-q-1)) + \Theta(n),$$

$$T(n) = \Omega(n^2).$$

We guess that $T(n) \ge cn^2$ for some constant c. Substituting into the recurrence, yields

$$\begin{aligned} T(n) &\ge \max_{0 \le q \le n-1}(cq^2 + c(n-q-1)^2) + \Theta(n) \\ &= c \cdot \max_{0 \le q \le n-1}(q^2 + (n-q-1)^2) + \Theta(n). \end{aligned}$$

The expression $q^2 + (n-q-1)^2$ achieves a maximum at $q = 0$ (proof on (7.4-3)). Thus, we have

$$\max_{0 \le q \le n-1}(q^2 + (n-q-1)^2) = (n-1)^2,$$

which give us the bound

$$\begin{aligned} T(n) &\ge c(n-1)^2 + \Theta(n) \\ &= cn^2 - 2cn + c + \Theta(n) \\ &= cn^2 - c(2n-1) + \Theta(n) \\ &\ge cn^2, \end{aligned}$$

since we pick the constant c small enough so that the $\Theta(n)$ term dominates the $c(2n-1)$ term, which implies

$$T(n) = \Omega(n^2).$$

7.4-2 Show that quicksort's best-case running time is $\Omega(n \lg n)$.

Let $T(n)$ be the best-case time of QUICKSORT on an input of size n. We have the recurrence

$$T(n) = \min_{0 \le q \le n-1}(T(q) + T(n-q-1)) + \Theta(n).$$

We guess $T(n) \ge cn \lg n$ for some constant c. Substituting into the recurrence yields

$$\begin{aligned} T(n) &\ge \min_{0 \le q \le n-1}(cq \lg q + c(n-q-1)\lg(n-q-1)) + \Theta(n) \\ &= c \cdot \min_{0 \le q \le n-1}(q \lg q + (n-q-1)\lg(n-q-1)) + \Theta(n). \end{aligned}$$

For simplicify, assume that n is odd. The expression $cq \lg q + (n-q-1)\lg(n-q-1)$ achieves a minimum when

$$q = n - q - 1,$$

which implies

$$q = \frac{n-1}{2}.$$

Thus, we have

$$\begin{aligned} T(n) &\ge c\left(\frac{n-1}{2}\right)\lg\left(\frac{n-1}{2}\right) + c\left(n - \frac{n-1}{2} - 1\right)\lg\left(n - \frac{n-1}{2} - 1\right) + \Theta(n) \\ &= c(n-1)\lg\left(\frac{n-1}{2}\right) + \Theta(n) \\ &= c(n-1)\lg(n-1) - c(n-1) + \Theta(n) \\ &= cn\lg(n-1) - c\lg(n-1) - c(n-1) + \Theta(n) \\ &\ge cn\lg\left(\frac{n}{2}\right) - c\lg(n-1) - c(n-1) + \Theta(n) \qquad (n \ge 2) \\ &= cn\lg n - cn - c\lg(n-1) - c(n-1) + \Theta(n) \\ &= cn\lg n - c(2n + \lg(n-1) - 1) + \Theta(n) \\ &\ge cn\lg n, \end{aligned}$$

since we pick the constant c small enough so that the $\Theta(n)$ term dominates the $c(2n + \lg(n-1) - 1)$ term, which implies

$$T(n) = \Omega(n \lg n).$$

7.4-3 Show that the expression $q^2 + (n-q-1)^2$ achieves a maximum over $q = 0, 1, \ldots, n-1$ when $q = 0$ or $q = n-1$.

Let $f(q) = q^2 + (n-q-1)^2$. We have

$$f'(q) = 2q + 2(n-q-1)\cdot(-1) = 4q - 2n + 2,$$

and

$$f''(q) = 4.$$

Since the second derivative is positive, $f(q)$ achieves a maximum over $0, 1, \ldots, n-1$ at either endpoint. But we have

$$f(0) = 0^2 + (n-1)^2 = (n-1)^2 + (n-(n-1)-1)^2 = f(n-1),$$

which implies that both endpoints are maximum.

7.4-4 Show that Randomized-Quicksort's expected running time is $\Omega(n \lg n)$.

Combining equations (7.2) and (7.3), we get

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{\lfloor n/2 \rfloor} \sum_{k=1}^{n-i} \frac{2}{k+1} + \sum_{i=\lfloor n/2 \rfloor + 1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&\geq \sum_{i=1}^{\lfloor n/2 \rfloor} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&\geq \sum_{i=1}^{\lfloor n/2 \rfloor} \sum_{k=1}^{n/2} \frac{2}{k+1} \\
&\geq \sum_{i=1}^{\lfloor n/2 \rfloor} \sum_{k=1}^{n/2} \frac{1}{k} && \text{(since } k \geq 1\text{)} \\
&= \left\lfloor \frac{n}{2} \right\rfloor \cdot \left(\lg\left(\frac{n}{2}\right) + O(1) \right) && \text{(approx. of harmonic number)} \\
&= \Omega(n \lg n).
\end{aligned}$$

7.4-5 We can improve the running time of quicksort in practice by taking advantage of the fast running time of insertion sort when its input is "nearly" sorted. Upon calling quicksort on a subarray with fewer than k elements, let it simply return without sorting the subarray. After the top-level call to quicksort returns, run insertion sort on the entire array to finish the sorting process. Argue that this sorting algorithm runs in $O(nk + n\lg(n/k))$ expected time. How should we pick k, both in theory and in practice?

Lets first analyze the modified Quicksort. As in the standard Quicksort, it is easy to see that the worst-case of this modified version is still $O(n^2)$. As for the expected time, we can use a similar argument to the one used on Section 7.2, in which we saw that any split of constant proportionality on Quicksort yields a recursion tree of depth $\Theta(\lg n)$. Assume that Partition on this modified Quicksort always give a 99-to-1 split. The height h of the recursion tree would be

$$\frac{n}{(100/99)^h} = k \to h = \log_{100/99} \frac{n}{k} \to h = \Theta\left(\lg \frac{n}{k}\right).$$

Since each recursion level has cost at most cn, the expected total cost of this modified Quicksort is $O(n \lg \frac{n}{k})$. As for the cost of the Insertion-Sort, note after running the modified Quicksort, every element will be out of order by at most k positions. Thus, each iteration of the outer loop of Insertion-Sort will make at most k swaps, which gives a running time of $O(nk)$. Finally, the cost of the whole algorithm is

$$O(nk) + O\left(n \lg\left(\frac{n}{k}\right)\right) = O\left(nk + n \lg\left(\frac{n}{k}\right)\right).$$

7.4-6 ($\star$) Consider modifying the PARTITION procedure by randomly picking three elements from array A and partitioning about their median (the middle value of the three elements). Approximate the probability of getting at worst an α-to-$(1-\alpha)$ split, as a function of α in the range $0 < \alpha < 1$.

First assume $0 < \alpha \leq 1/2$. There four ways to get a split worse than α-to-$(1-\alpha)$:

(a) The index of exactly two elements are smaller than αn

(b) The index of exactly two elements are greater than $n - \alpha n$.

(c) The index of all three elements are smaller than αn.

(d) The index of all three elements are greater than $n - \alpha n$.

Since we want an approximation, assume that we can repeat the same element. The probability of cases (a) and (b) is

$$\Pr\{(a)\} = \Pr\{(b)\} = 3 \cdot \left(\frac{\alpha n}{n} \cdot \frac{\alpha n}{n} \cdot \frac{(1-\alpha)n}{n} \right) = 3\alpha^2 - 3\alpha^3,$$

in which the multiplication on the left is needed since there are $\binom{3}{1} = 3$ ways to pick one of three elements outside the desired range. The probability of cases (c) and (d) is

$$\Pr\{(c)\} = \Pr\{(d)\} = \frac{\alpha n}{n} \cdot \frac{\alpha n}{n} \cdot \frac{\alpha n}{n} = \alpha^3.$$

Thus, the probability of getting a split worse than α-to-$(1-\alpha)$ is

$$\begin{aligned} 1 - \Pr\{(a) + (b) + (c) + (d)\} &= 1 - (\Pr\{(a)\} + \Pr\{(b)\} + \Pr\{(c)\} + \Pr\{(d)\}) \\ &= 1 - \left((3\alpha^2 - 3\alpha^3) + (3\alpha^2 - 3\alpha^3) + \alpha^3 + \alpha^3\right) \\ &= 1 - (6\alpha^2 - 4\alpha^3) \\ &= 1 - 6\alpha^2 + 4\alpha^3. \end{aligned}$$

The proof is similar for $1/2 \leq \alpha < 1$ and the result is the same.

Problems

7-1 ***Hoare partition correctness***

The version of PARTITION given in this chapter is not the original partitioning algorithm. Here is the original partition algorithm, which is due to C.A.R. Hoare:

```
   Hoare-Partition(A, p, r)
1  | x = A[p]
2  | i = p − 1
3  | j = r + 1
4  | while True do
5  |   | repeat
6  |   |   | j = j − 1
7  |   | until A[j] ≤ x
8  |   | repeat
9  |   |   | i = i + 1
10 |   | until A[i] ≥ x
11 |   | if i < j then
12 |   |   | exchange A[i] with A[j]
13 |   | else
14 |   |   | return j
```

a. Demonstrate the operation of HOARE-PARTITION on the array $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21\rangle$, showing the values of the array and auxiliary values after each iteration of the **while** loop in lines 4–13.

The next three questions ask you to give a careful argument that the procedure HOARE-PARTITION is correct. Assuming that the subarray $A[p, \ldots, r]$ contains at least two elements, prove the following:

b. The indices i and j are such that we never access an element of A outside the subarray $A[p, \ldots, r]$.

c. When HOARE-PARTITION terminates, it returns a value j such that $p \le j < r$.

d. Every element of $A[p, \ldots, j]$ is less than or equal to every element of $A[j + 1, \ldots, r]$ when HOARE-PARTITION terminates.

The PARTITION procedure in Section 7.1 separates the pivot value (originally in $A[r]$) from the two partitions it forms. The HOARE-PARTITION procedure, on the other hand, always places the pivot value (originally in $A[p]$) into one of the two partitions $A[p, \ldots, j]$ and $A[j + 1, \ldots, r]$. Since $p \le j < r$, this split is always nontrivial.

e. Rewrite the QUICKSORT procedure to use HOARE-PARTITION.

(a) The operation is illustrated below:

i	x										j	
	13	19	9	5	12	8	7	4	11	2	6	21
	x, i										j	
	13	19	9	5	12	8	7	4	11	2	6	21
		i								j	x	
	6	19	9	5	12	8	7	4	11	2	13	21
									j	i	x	
	6	2	9	5	12	8	7	4	11	19	13	21

(b) Consider the following ***loop invariant***:

> At the beginning of each iteration of the **while** loop of lines 4-13, $i < j$ and the subarray $A[p, \ldots, j - 1]$ contains at least one element that is lower than or equal to $A[x]$. Similarly, the subarray $A[i + 1, \ldots, r]$ contains at least one element that is greater than or equal to $A[x]$.

We need to show that this loop invariant is true prior to the first iteration, that each iteration of the loop maintains the invariant, and that the invariant provides a useful property to show correctness when the loop terminates.

- **Initialization.** Prior to the **while** loop of lines 4-13, $i = p-1$ and $j = r-1$. Since $A[i+1, \ldots, r] = A[p, \ldots, j-1] = A[p, \ldots, r]$, the element $A[x]$ is present in both subarrays. Thus, the loop invariant is valid before the loop.
- **Maintenance.** From the loop invariant, the **for** loops of lines 5-7 and 8-10 will both stop on valid indices i and j. The loop only goes to the next iteration if $i < j$. In that case, the elements $A[i]$ and $A[j]$ are exchanged, which ensures the loop invariant for the next iteration.
- **Termination.** At termination, the **for** loops of lines 5-7 and 8-10 will stop on valid indices i and j such that $i \ge j$ and the loop terminates before going to the next iteration.

The above loop invariant ensures that the for loops of lines 5-7 and 8-10 will never make $j < p$ or $i > r$, which implies that the HOARE-PARTITION procedure always access elements within the subarray $A[p, \ldots, r]$.

(c) From item (b), we have the lower bound $j \geq p$. As for the upper bound of j, note that:

i. If $A[r] > x$, the condition on line 7 will be false at least one time, which implies that line 6 will be executed at least twice. Since the initial value of j is $r + 1$, in this case we have $j < r$.

ii. If $A[r] \leq x$, $A[r]$ will be exchanged with $A[p]$ in the first iteration of the **while** loop and line 6 will be executed for the second time in the next iteration. Thus, in this case we also have $j < r$.

These observations give us the bound $p \leq j < r$.

(d) Consider the following ***loop invariant***:

> At the beginning of each iteration of the **while** loop of lines 4-13, every element of the subarray $A[p, \ldots, \min(i, j)]$ is less than or equal to every element of the subarray $A[j + 1, \ldots, r]$.

We need to show that this loop invariant is true prior to the first iteration, that each iteration of the loop maintains the invariant, and that the invariant provides a useful property to show correctness when the loop terminates.

- **Initialization.** Prior to the **while** loop, $i = p - 1$, $j = r + 1$. Since the subarrays $A[p, \ldots, i]$ and $A[j, \ldots, r]$ are empty, the loop invariant is trivially satisfied.
- **Maintenance.** To see that each iteration maintains the loop invariant, note that the **for** loops of lines 5-7 and 8-10 will decrease j and increase i until find an $A[j] \leq x$ and an $A[i] \geq x$, respectivelly. At this point, the loop invariant (of previous iteration) along with the conditions of lines 7 and 10 ensures that every element of $A[p, \ldots, i - 1]$ is less than or equal to every element of $A[j + 1, \ldots, r]$. The only possible exceptions to the loop invariant in this iteration are the elements $A[i]$ and $A[j]$. Since $A[i] \geq x$ and $A[j] \leq x$, we have $A[i] \geq A[j]$. To go to the next iteration line 11 must be valid and the exchange of $A[i]$ with $A[j]$ at line 12 maintains the loop invariant.
- **Termination.** At termination, the **for** loop of lines 5-7 and 8-10 will stop on indices i and j such that $i \geq j$. Since $\min(i, j) = j$, the loop invariant (of previous iteration) along with the conditions of lines 7 and 10 ensures that every element of $A[p, \ldots, j]$ will be less than or equal to every element of $A[j + 1, \ldots, r]$.

(e) The pseudocode is stated below.

```
Hoare-Quicksort(A, p, r)
1   if p < r then
2       q = Hoare-Partition(A, p, r)
3       Hoare-Partition(A, p, q)
4       Hoare-Partition(A, q + 1, r)
```

7-2 ***Quicksort with equal element values***

The analysis of the expected running time of randomized quicksort in Section 7.4.2 assumes that all element values are distinct. In this problem, we examine what happens when they are not.

a. Suppose that all element values are equal. What would be randomized quicksort's running time in this case?

b. The PARTITION procedure returns an index q such that each element of $A[p, \ldots, q-1]$ is less than or equal to $A[q]$ and each element of $A[q+1, \ldots, r]$ is greater than $A[q]$. Modify the PARTITION procedure to produce a procedure PARTITION'(A, p, r), which permutes the elements of $A[p, \ldots, r]$ and returns two indices q and t, where $p \leq q \leq t \leq r$, such that

- all elements if $A[q, \ldots, t]$ are equal,
- each element of $A[p, \ldots, q-1]$ is less than $A[q]$, and
- each element of $A[r+1, \ldots, r]$ is greater than $A[q]$.

Like PARTITION, your PARTITION' procedure should take $\Theta(r-p)$ time.

c. Modify the RANDOMIZED-PARTITION procedure to call PARTITION', and name the new procedure RANDOMIZED-PARTITION'. Then modify the QUICKSORT procedure to produce a procedure QUICKSORT'(A, p, r) that calls RANDOMIZED-PARTITION' and recurses only on partitions of elements not known to be equal to each other.

d. Using QUICKSORT', how would you adjust the analysis in Section 7.4.2 to avoid the assumption that all elements are distinct?

(a) In this case, the condition on line 4 of the PARTITION procedure will always be valid and it will always give "bad" splits $((n-1)$-to-0). Thus, the running time will be $\Theta(n^2)$.

(b) This item is similar to the Question 7.1-2. The pseudocode of the modified PARTITION procedure is stated below.

```
Partition'(A, p, r)
1  | x = A[r]
2  | i = j = p − 1
3  | for k = p to r − 1 do
4  | |  if A[k] ≤ x then
5  | |  |  j = j + 1
6  | |  |  exchange A[j] with A[k]
7  | |  |  if A[j] < x then
8  | |  |  |  i = i + 1
9  | |  |  |  exchange A[i] with A[j]
10 | exchange A[j + 1] with A[r]
11 | q = i + 1
12 | t = j + 1
13 | return q, t
```

(c) The pseudocode of the modified RANDOMIZED-PARTITION QUICKSORT procedures are stated below.

```
Randomized-Partition'(A, p, r)
1  | i = Random(p, r)
2  | exchange A[r] with A[i]
3  | return Partition'(A, p, r)

Quicksort'(A, p, r)
1  | if p < r then
2  | |  q, t = Randomized-Partition'(A, p, r)
3  | |  Quicksort'(A, p, q − 1)
4  | |  Quicksort'(A, t + 1, r)
```

(d) We just need to rewrite the sentence

> In general, because we assume that element values are distinct, once a pivot x is chosen with $z_i < x < z_j$, we know that z_i and z_j cannot be compared at any subsequent time.

as

> Once a pivot z_q is chosen with $z_i \leq z_q \leq z_j$, such that $i \neq q$ and $j \neq q$, we know that z_i and z_j cannot be compared at any subsequent time.

7-3 ***Alternative quicksort analysis***

An alternative analysis of the running time of randomized quicksort focuses on the expected running time of each individual recursive call to RANDOMIZED-QUICKSORT, rather than on the number of comparisons performed.

a. Argue that, given an array of size n, the probability that any particular element is chosen as the pivot is $1/n$. Use this to define indicator random variables $X_i = \mathrm{I}\{i\text{th smallest element is chosen as the pivot}\}$. What is $\mathrm{E}[X_i]$?

b. Let $T(n)$ be a random variable denoting the running time of quicksort on an array of size n. Argue that

$$\mathrm{E}[T(n)] = \mathrm{E}\left[\sum_{q=1}^{n} X_q(T(q-1) + T(n-q) + \Theta(n))\right].$$

c. Show that we can rewrite equation (7.5) as

$$\mathrm{E}[T(n)] = \frac{2}{n}\sum_{q=2}^{n-1}\mathrm{E}[T(q)] + \Theta(n).$$

d. Show that

$$\sum_{k=2}^{n-1} k\lg k \le \frac{1}{2}n^2\lg n - \frac{1}{8}n^2.$$

(*Hint:* Split the summation into two parts, one for $k = 2, 3, \ldots, \lceil n/2\rceil - 1$ and one for $k = \lceil n/2\rceil, \ldots, n-1$.)

e. Using the bound from equation (7.7), show that the recurrence in equation (7.6) has the solution $\mathrm{E}[T(n)] = \Theta(n\lg n)$. (*Hint:* Show, by substitution, that $\mathrm{E}[T(n)] \le an\lg n$ for sufficiently large n and for some positive constant a.)

(a) In each recursive call to RANDOMIZED-QUICKSORT, the pivot is randomly chosen among the n elements of the input subarray. Thus, we have

$$\mathrm{E}[X_i] = \Pr\{X_i = 1\} = \frac{1}{n}.$$

(b) Each call to PARTITION tales $\Theta(n)$ and once the qth

(c) Each call to QUICKSORT selects a pivot q, such that $1 \le q \le n$, that partitions the array into two subarrays of sizes $q-1$ and $n-q$. The indicator random variable X_q indicates whether the element with index q is selected as the pivot. Since only one element can be chosen as the pivot at a given call to QUICKSORT and the running time of each call is $\Theta(n)$, the running time of QUICKSORT can be written as

$$T(n) = \sum_{q=1}^{n} X_q(T(q-1) + T(n-q) + \Theta(n)),$$

which implies

$$\mathrm{E}[T(n)] = \mathrm{E}\left[\sum_{q=1}^{n} X_q(T(q-q) + T(n-q) + \Theta(n))\right].$$

(d)

$$\begin{aligned}
\mathrm{E}[T(n)] &= \mathrm{E}\left[\sum_{q=1}^{n} X_q(T(q-q) + T(n-q) + \Theta(n))\right] \\
&= \sum_{q=1}^{n}\mathrm{E}[X_q(T(q-1) + T(n-q) + \Theta(n))] \\
&= \sum_{q=1}^{n}\left(\frac{1}{n}\cdot\mathrm{E}[(T(q-1) + T(n-q) + \Theta(n))]\right) \\
&= \frac{1}{n}\cdot\sum_{q=1}^{n}\mathrm{E}[T(q-1)] + \frac{1}{n}\cdot\sum_{q=1}^{n}\mathrm{E}[T(n-q)] + \frac{1}{n}\cdot\sum_{q=1}^{n}\Theta(n) \\
&= \frac{1}{n}\cdot\sum_{q=2}^{n-1}\mathrm{E}[T(q)] + \frac{1}{n}\cdot\Theta(1) + \frac{1}{n}\cdot\sum_{q=2}^{n-1}\mathrm{E}[T(q)] + \frac{1}{n}\cdot\Theta(1) + \Theta(n) \\
&= \frac{2}{n}\cdot\sum_{q=2}^{n-1}\mathrm{E}[T(q)] + \Theta(n).
\end{aligned}$$

(e) We guess that

$$\mathrm{E}[T(n)] \leq an \lg n,$$

for some constant a. Substituting into the recurrence yields

$$\begin{aligned}
\mathrm{E}[T(n)] &\leq \frac{2}{n} \sum_{q=2}^{n-1} (aq \lg q) + \Theta(n) \\
&= \frac{2a}{n} \sum_{q=2}^{n-1} (q \lg q) + \Theta(n) \\
&\leq \frac{2a}{n} \left(\frac{1}{2} n^2 \lg n - \frac{1}{8} n^2 \right) + \Theta(n) \\
&= an \lg n - a \frac{1}{4} n + \Theta(n) \\
&\leq an \lg n,
\end{aligned}$$

since we pick the constant a large enough so that the $a(1/4)n$ term dominates the $\Theta(n)$ term.

7-4 ***Stack depth for quicksort***

The QUICKSORT algorithm of Section 7.1 contains two recursive calls to itself. After QUICKSORT calls PARTITION, it recursively sorts the left subarray and then it recursively sorts the right subarray. The second recursive call in QUICKSORT is not really necessary; we can avoid it by using an iterative control structure. This technique, called ***tail recursion***, is provided automatically by good compilers. Consider the following version of quicksort, which simulates tail recursion:

```
  Tail-Recursive-Quicksort(A, p, r)
1 | while p < r do
2 | |   // Partition and sort left subarray
3 | |   q = Partition(A, p, r)
4 | |   Tail-Recursive-Quicksort(A, p, q - 1)
5 | |   p = q + 1
```

a. Argue that TAIL-RECURSIVE-QUICKSORT$(A, 1, A.length)$ correctly sorts the array A.

Compilers usually execute recursive procedures by using a ***stack*** that contains pertinent information, including the parameter values, for each recursive call. The information for the most recent call is at the top of the stack, and the information for the initial call is at the bottom. Upon calling a procedure, its information is ***pushed*** onto the stack; when it terminates, its information is ***popped***. Since we assume that array parameters are represented by pointers, the information for each procedure call on the stack requires $O(1)$ stack space. The stack depth is the maximum amount of ***stack space*** used at any time during a computation.

b. Describe a scenario in which TAIL-RECURSIVE-QUICKSORT's stack depth is $\Theta(n)$ on an n-element array.

c. Modify the code for TAIL-RECURSIVE-QUICKSORT so that the worst-case stack depth is $\Theta(\lg n)$. Maintain the $O(n \lg n)$ expected running time of the algorithm.

(a) After the PARTITION call, the algorithm calls itself with arguments $A, p, q-1$, sets $p = q+1$, and repeat the same operations. Since the only difference to the next iteration is the new value of p, the loop is similar as calling itself with arguments $A, q+1, r$. Thus, TAIL-RECURSIVE-QUICKSORT produces the same result of QUICKSORT.

(b) If the PARTITION procedure always select the largest element of the array as the pivot, the left subarray will always have size $n-1$, and the stack depth will be $\Theta(n)$.

(c) To reduce the maximum stack depth, we should always give to the tail recursion the larger of the two subproblems. The updated pseudocode is stated below.

```
  Tail-Recursive-Quicksort-Improved(A, p, r)
1 | while p < r do
2 | |   // Partition and sort left subarray
3 | |   q = Partition(A, p, r)
4 | |   if q < (p + r)/2 then
5 | |   |   Tail-Recursive-Quicksort(A, p, q - 1)
6 | |   |   p = q + 1
7 | |   else
8 | |   |   Tail-Recursive-Quicksort(A, q + 1, r)
9 | |   |   r = q - 1
```

Each recursive call reduces the problem size by at least half. Thus, the stack depth is $O(\lg n)$.

7-5 ***Median-of-3 partition***

One way to improve the RANDOMIZED-QUICKSORT procedure is to partition around a pivot that is chosen more carefully than by picking a random element from the subarray. One common approach is the ***median-of-3*** method: choose the pivot as the median (middle element) of a set of 3 elements randomly selected from the subarray. (See Exercise 7.4-6.) For this problem, let us assume that the elements in the input array $A[1,\ldots,n]$ are distinct and that $n \geq 3$. We denote the sorted output array by $A'[1,\ldots,n]$. Using the median-of-3 method to choose the pivot element x, define $p_i = \Pr\{x = A'[i]\}$.

a. Give an exact formula for p_i as a function of n and i for $i = 2, 3, \ldots, n-1$. (Note that $p_1 = p_n = 0$.)

b. By what amount have we increased the likelihood of choosing the pivot as $x = A'[\lfloor (n+1)/2 \rfloor]$, the median of $A[1,\ldots,n]$, compared with the ordinary implementation? Assume that $n \to \infty$, and give the limiting ratio of these probabilities.

c. If we define a "good" split to mean choosing the pivot as $x = A'[i]$, where $n/3 \leq i \leq 2n/3$, by what amount have we increased the likelihood of getting a good split compared with the ordinary implementation? (*Hint:* Approximate the sum by an integral.)

d. Argue that in the $\Omega(n \lg n)$ running time of quicksort, the median-of-3 method affects only the constant factor.

(a) Note that the number of 3-permutations on a set of n elements is

$$\frac{n!}{(n-3)!} = n(n-1)(n-2).$$

To choose the element i as the pivot, one element needs to be within the first $i-1$ positions of the array, one elements needs to be within the last $n-i$ positions of the array, and one element needs to be the ith element. Also note that each combination of elements that chooses the element i as the pivot has $3! = 6$ permutations; hence 6 ways to be selected. Thus, we have

$$p_i = \frac{3! \cdot 1 \cdot (i-1) \cdot (n-i)}{n!/(n-3)!} = \frac{6(i-1)(n-i)}{n(n-1)(n-2)}.$$

(b) We have

$$\begin{aligned} p_{\lfloor (n+1)/2 \rfloor} &= \frac{6 \cdot \left(\left\lfloor \frac{n+1}{2} \right\rfloor - 1\right)\left(n - \left\lfloor \frac{n+1}{2} \right\rfloor\right)}{n(n-1)(n-2)} \\ &\leq \frac{6 \cdot \left(\frac{n+1}{2} - 1\right)\left(n - \frac{n+1}{2}\right)}{n(n-1)(n-2)} \\ &= \frac{6 \cdot \left(\frac{n-1}{2}\right)\left(\frac{n-1}{2}\right)}{n(n-1)(n-2)} \\ &= \frac{3}{2} \cdot \frac{(n-1)(n-1)}{n(n-1)(n-2)} \\ &= \frac{3}{2} \cdot \frac{(n-1)}{n(n-2)}. \end{aligned}$$

Then, we have the ratio

$$\lim_{n\to\infty} \frac{\frac{3}{2} \cdot \frac{n-1}{n(n-2)}}{\frac{1}{n}} = \lim_{n\to\infty} \frac{3}{2} \frac{n(n-1)}{n(n-2)} = \frac{3}{2} = 1.5.$$

(c) To get a "good" split with the median-of-3 method, the pivot can not be within the first $\lfloor n/3 \rfloor$ elements or within the last $\lceil n/3 \rceil$ elements. Thus, we have

$$\Pr\{\text{good split with median-of-3}\} = \sum_{i=\lceil n/3 \rceil}^{\lfloor 2n/3 \rfloor} p_i \approx \sum_{i=n/3}^{2n/3} p_i = \sum_{i=n/3}^{2n/3} \frac{6(i-1)(n-i)}{n(n-1)(n-2)} = \frac{6}{n(n-1)(n-2)} \cdot \sum_{i=n/3}^{2n/3} (i-1)(n-i).$$

Note that

$$\begin{aligned} \sum_{x=n/3}^{2n/3} (x-1)(n-x) &\approx \int_{n/3}^{2n/3} (x-1)(n-x)dx \\ &= \int_{n/3}^{2n/3} (nx - x^2 - n + x)dx \\ &= -\frac{1}{3}x^3 + \frac{1}{2}x^2(n+1) - xn \Big|_{n/3}^{2n/3} \\ &= \frac{13}{162}n^3 - \frac{1}{6}n^2, \end{aligned}$$

which implies

$$\Pr\{\text{good split with median-of-3}\} \approx \frac{6}{n(n-1)(n-2)} \left(\frac{13}{162}n^3 - \frac{1}{6}n^2\right) = \frac{\frac{13}{27}n^3 - n^2}{n(n-1)(n-2)}.$$

Then, we have the ratio

$$\frac{\Pr\{\text{good split with median-of-3}\}}{\Pr\{\text{good split with one pivot}\}} = \lim_{n\to\infty} \frac{\frac{\frac{13}{27}n^3-n^2}{n(n-1)(n-2)}}{\frac{1}{3}} = \lim_{n\to\infty} \frac{\frac{\frac{13}{27}n^3-n^2}{n^3-n^2-2n}}{\frac{1}{3}} = \lim_{n\to\infty} \frac{\frac{13}{27}}{\frac{1}{3}} \approx 1.44.$$

(d) The only difference is on the choice of the pivot. However, even if the middle element is always chosen as the pivot (which is the best case), the height of the recursion tree will be $\Theta(\lg n)$. Since each recursion level takes $\Theta(n)$, the running time is still $\Omega(n \lg n)$.

7-6 ***Fuzzy sorting of intervals***

Consider a sorting problem in which we do not know the numbers exactly. Instead, for each number, we know an interval on the real line to which it belongs. That is, we are given n closed intervals of the form $[a_i, b_i]$, where $a_i \leq b_i$. We wish to ***fuzzy-sort*** these intervals, i.e., to produce a permutation $\langle i_1, i_2, \ldots, i_n \rangle$ of the intervals such that for $j = 1, 2, \ldots, n$, there exist $c_j \in [a_{i_j}, b_{i_j}]$ satisfying $c_1 \leq c_2 \leq \cdots \leq c_n$.

a. Design a randomized algorithm for fuzzy-sorting n intervals. Your algorithm should have the general structure of an algorithm that quicksorts the left endpoints (the a_i values), but it should take advantage of overlapping intervals to improve the running time. (As the intervals overlap more and more, the problem of fuzzy-sorting the intervals becomes progressively easier. Your algorithm should take advantage of such overlapping, to the extent that it exists.)

b. Argue that your algorithm runs in expected time $\Theta(n \lg n)$ in general, but runs in expected time $\Theta(n)$ when all of the intervals overlap (i.e., when there exists a value x such that $x \in [a_i, b_i]$ for all i). Your algorithm shoukd not be checking for this case explicitly; rather, its performance should naturally improve as the amount of overlap increases.

(a) Note that any subset of intervals that share a common point are already sorted. Using this notion, we can sort an array of fuzzy intervals with an algorithm similar to quicksort, but with a customized partition procedure that returns two indices q and t, where $p \leq q \leq t \leq r$, such that

- There exist x such that $x \in [a_i, b_i]$ for all $q \leq i \leq t$. That is, any permutation of the subarray $A[q, \ldots, t]$ is sorted;
- For all $j < q$, there exist $c_j \in [a_j, b_j]$ such that $c_j < b_i$ for all $q \leq i \leq t$. That is, every interval of $A[p, \ldots, q-1]$ can stay before every interval of $A[q, \ldots, t]$ in the sorted array;
- For all $j > t$, there exist $c_j \in [a_j, b_j]$ such that $c_j > a_i$ for all $q \leq i \leq t$. That is, every interval of $A[t+1, \ldots, r]$ can stay after every interval of $A[q, \ldots, t]$ in the sorted array.

The pseudocode is stated below.

```
Fuzzy-Partition(A, p, r)
 1  a = A[r].a
 2  b = A[r].b
 3  i = j = p − 1
 4  for k = p to r − 1 do
 5      // A[k] can be placed before the pivot
 6      if A[k].a ≤ b then
 7          j = j + 1
 8          exchange A[j] with A[k]
 9          // intervals (A[k] and pivot) do not overlap
10          if A[j].b < a then
11              i = i + 1
12              exchange A[i] with A[j]
13          // intervals (A[k] and pivot) overlap
14          else
15              if A[k].a > a then
16                  a = A[k].a
17              if A[k].b < b then
18                  b = A[k].b
19  exchange A[j + 1] with A[r]
20  q = i + 1
21  t = j + 1
22  return q, t
```

```
Fuzzy-Randomized-Partition(A, p, r)
1  i = Random(p, r)
2  exchange A[r] with A[i]
3  return Fuzzy-Partition(A, p, r)
```

```
Fuzzy-Sort(A, p, r)
1  if p < r then
2      q, t = Fuzzy-Randomized-Partition(A, p, r)
3      Fuzzy-Sort(A, p, q − 1)
4      Fuzzy-Sort(A, t + 1, r)
```

(b) When all intervals share a common point, the array is already sorted. In this case, there will be only one call to FUZZY-PARTITION, which will return $q = p$ and $t = r$. Thus, the algorithm will run in $\Theta(n)$.
The general case is a little tricky to proof.

Section 8.1 – Lower bounds for sorting

8.1-1 What is the smallest possible depth of a leaf in a decision tree for a comparison sort?

The smallest possible depth of a leaf in a decision tree can be obtained by calculating the shortest simple path from the root to any of its reachable leaves. This smallest path occurs when the comparisons is made in the sorted order. For instance, if the input array is sorted, the following comparisons suffices

$$\begin{aligned} a_1 &\leq a_2, \\ a_2 &\leq a_3, \\ &\vdots \\ a_{n-1} &\leq an. \end{aligned}$$

Thus, the smallest depth of a leaf is any decision tree is $n - 1 = \Theta(n)$.

8.1-2 Obtain asymptotically tight bounds on $\lg(n!)$ without using Stirling's approximation. Instead, evaluate the summation $\sum_{k=1}^{n} \lg k$ using techniques from Section A.2.

Assume for convenience that n is even. For a lower bound, we have

$$\begin{aligned} \lg n! &= \lg(n \cdot (n-1) \cdot (n-2) \cdots 1) \\ &= \sum_{k=1}^{n} \lg k \\ &= \sum_{k=1}^{n/2} \lg k + \sum_{k=n/2+1}^{n} \lg k \\ &\geq \sum_{k=1}^{n/2} 0 + \sum_{k=n/2+1}^{n} \lg(n/2) \\ &= \frac{n}{2} \lg \frac{n}{2} \\ &= \frac{n}{2} \lg n - \frac{n}{2} \\ &= \Omega(n \lg n). \end{aligned}$$

And for an upper bound, we have

$$\begin{aligned} \lg n! &= \lg(n \cdot (n-1) \cdot (n-2) \cdots 1) \\ &= \sum_{k=1}^{n} \lg k \\ &\leq \sum_{k=1}^{n} \lg n \\ &= O(n \lg n). \end{aligned}$$

Thus, $\lg n! = \Theta(n \lg n)$.

8.1-3 Show that there is no comparison sort whose running time is linear for at least half of the $n!$ inputs of length n. What about a fraction of $1/n$ of the inputs of length n? What about a fraction $1/2^n$?

Such algorithm only exists if we can build a decision tree such that at least $n!/2$ of its $n!$ leaves has a depth of $\Theta(n)$. Suppose this decision tree exists. Let m be the depth of the leaf with the $(n!/2)$th smallest depth. Remove all nodes with depth greater than m. The result is a decision tree with height m and at least $n!/2$ leaves. Using the same reasoning as in the proof of Theorem 8.1, for every decision tree with at least $n!/2$ leaves, we have

$$\frac{n!}{2} \leq l \leq 2^m,$$

which implies

$$m \geq \lg \frac{n!}{2} = \lg n! - 1 = \Omega(n \lg n),$$

which proves that such a decision tree does not exists. The same reasoning can be applied to obtain the maximum depth of any fraction of the inputs. For a fraction of $1/n$, we have

$$m \geq \lg \frac{n!}{n} = \lg n! - \lg n = \Omega(n \lg n),$$

and for a fraction of $1/2^n$, we have

$$m \geq \lg \frac{n!}{2^n} = \lg n! - \lg 2^n = \lg n! - n = \Omega(n \lg n).$$

8.1-4 Suppose that you are given a sequence of n elements to sort. The input sequence consists of n/k subsequences, each containing k elements. The elements in a given subsequence are all smaller than the elements in the succeeding subsequence and larger than the elements in the preceding subsequence. Thus, all that is needed to sort the whole sequence of length n is to sort the k elements in each of the n/k subsequences. Show an $\Omega(n \lg k)$ lower bound on the number of comparisons needed to solve this variant of the sorting problem. (*Hint*: It is not rigorous to simply combine the lower bounds for the individual subsequences.)

All we know is the ordering of the elements of a given subsequence with respect to the elements of the previous/next subsequence. Thus, for each subsequence, we have $k!$ possible permutations. Since there are n/k input subsequences, the number of possible outcomes for this sorting problem is

$$\prod_{i=1}^{n/k} k! = k!^{(n/k)}.$$

We can use here the same argument used in the text book to prove a lower bound for any comparison sort algorithm. However, in this case, the number of possible permutations is $k!^{(n/k)}$, instead of $n!$. Thus, we need to show that the height of any decision tree with at least $k!^{(n/k)}$ leaves is $\Omega(n \lg k)$. We have

$$k!^{n/k} \leq l \leq 2^h,$$

which implies

$$\begin{aligned}
h &\geq \lg \left(k!^{(n/k)} \right) \\
&= \frac{n}{k} \cdot \lg k! \\
&= \frac{n}{k} \cdot \sum_{i=1}^{k} \lg i \\
&= \frac{n}{k} \cdot \sum_{i=1}^{\lfloor k/2 \rfloor} \lg i + \frac{n}{k} \cdot \sum_{i=\lfloor k/2 \rfloor + 1}^{k} \lg i \\
&\geq \frac{n}{k} \cdot \sum_{i=\lfloor k/2 \rfloor}^{k} \lg i \\
&\geq \frac{n}{k} \cdot \left(\frac{k}{2} \lg \frac{k}{2} \right) \\
&= \frac{n}{2} \lg \frac{k}{2} \\
&= \Omega(n \lg k).
\end{aligned}$$

Section 8.2 – Counting sort

8.2-1 Using Figure 8.2 as a model, illustrate the operation of COUNTING-SORT on the array $A = \langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 \rangle$.

	1	2	3	4	5	6	7	8	9	10	11
A	6	0	2	0	1	3	4	6	1	3	2

	0	1	2	3	4	5	6
C	2	2	2	2	1	0	2
C	2	4	6	8	9	9	11

B	–	–	–	–	–	2	–	–	–	–	–
B	–	–	–	–	–	2	–	3	–	–	–
B	–	–	–	1	–	2	–	3	–	–	–
B	–	–	–	1	–	2	–	3	4	–	6
B	–	–	–	1	–	2	3	3	4	–	6
B	–	–	1	1	–	2	3	3	4	–	6
B	–	0	1	1	–	2	3	3	4	–	6
B	–	0	1	1	2	2	3	3	4	–	6
B	0	0	1	1	2	2	3	3	4	–	6
B	0	0	1	1	2	2	3	3	4	6	6

C	2	4	5	8	9	9	11
C	2	4	5	7	9	9	11
C	2	3	5	7	9	9	11
C	2	3	5	7	8	9	10
C	2	3	5	6	8	9	10
C	2	2	5	6	8	9	10
C	1	2	5	6	8	9	10
C	1	2	4	6	8	9	10
C	0	2	4	6	8	9	10
C	0	2	4	6	8	9	9

8.2-2 Prove that COUNTING-SORT is stable.

Suppose that the integer x appears k times in the output array. Since the **for** loop of lines 10-12 iterates over the input array backwards, the first integer x to be added to the output array on line 11 is the rightmost one. The decrement of the couting of x on line 12 ensures that the next integer x is added to the output array right before the previous one. This process repeats k times, until the leftmost integer x is added to the output array ($k - 1$ positions before the rightmost one). This property ensures that elements with equal value in the input array appears in the same order in the output array. Thus, the algorithm is stable.

8.2-3 Suppose that we were to rewrite the **for** loop header in line 10 of the COUNTING-SORT as

10 **for** $j = 1$ **to** $A.length$

Show that the algorithm still works properly. Is the modified algorithm stable?

The only difference will be in the **for** loop of lines 10-12, in which elements with equal value in the input array will now be added to the output array in the same order as they appear in the input array. As observed on Question 8.2-2, each time an element with value x is added to the output array, the next element with value x is added right before the previous one. This implies that elements with equal value in the input array will appear in reverse order in the output array. Thus, this modified algorithm is not stable.

8.2-4 Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a \ldots b]$ in $O(1)$ time. Your algorithm should use $\Theta(n + k)$ preprocessing time.

For the preprocessing phase, build the array C in the same way it is built in the COUTING-SORT procedure (lines 1-8). This preprocessing will run in $\Theta(k) + \Theta(n) + \Theta(k) = \Theta(n + k)$. If $a > 0$, answer $C[b] - C[a - 1]$. Otherwise, answer $C[b]$.

Section 8.3 – Radix sort

8.3-1 Using Figure 8.3 as a model, illustrate the operation of RADIX-SORT on the following list of English words: COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.

COW		SE**A**		T**A**B		**B**AR
DOG		TE**A**		B**A**R		**B**IG
SEA		MO**B**		E**A**R		**B**OX
RUG		TA**B**		T**A**R		**C**OW
ROW		DO**G**		S**E**A		**D**IG
MOB		RU**G**		T**E**A		**D**OG
BOX		DI**G**		D**I**G		**E**AR
TAB	$\rightarrow$	BI**G**	$\rightarrow$	B**I**G	$\rightarrow$	**F**OX
BAR		BA**R**		M**O**B		**M**OB
EAR		EA**R**		D**O**G		**N**OW
TAR		TA**R**		C**O**W		**R**OW
DIG		CO**W**		R**O**W		**R**UG
BIG		RO**W**		N**O**W		**T**AB
TEA		NO**W**		B**O**X		**T**AR
NOW		BO**X**		F**O**X		**T**EA
FOX		FO**X**		R**U**G		**S**EA

8.3-2 Which of the following sorting algorithms are stable: insertion sort, merge sort, heapsort, and quicksort? Give a simple scheme that makes any comparison sort stable. How much additional time and space does your scheme entail?

INSERTION-SORT and MERGE-SORT are stable. HEAPSORT and QUICKSORT are not. To make any sorting algorithm stable, we can store the original index of each element in the array and use this index to break ties. The additional space required is $\Theta(n)$. The asymptotic running time is the same, since the number of comparisons will be at most twice.

8.3-3 Use induction to prove that radix sort works. Where does your proof need the assumption that the intermediate sort is stable?

Let d be the number of columns in the input array, where the dth column contains the highest-order digit. RADIX-SORT sorts one column at a time, from the column with the lowest-order digits to the column with the highest-order digits. The base case occurs when $d = 1$. Since in this case the elements on the array has a single digit, calling RADIX-SORT in this case is the same as calling its sorting subroutine directly with the input array as an argument. Thus, RADIX-SORT is correct when $d = 1$. Now assume RADIX-SORT works for $d-1$ columns. Note that sorting d columns is equivalent to sorting $d-1$ columns followed by calling the sorting subroutine on the dth column. The induction hypothesis ensures that RADIX-SORT sorts $d-1$ columns correctly. Since the sorting subroutine is stable, when sorting the dth column, digits that have the same value in the dth column will be kept in the same order as it was after sorting the $(d-1)$th column. This implies that RADIX-SORT breaks ties on higher-order digits by the lower-order digits, which is correct.

The sorting subroutine must be stable since a tie that occur while sorting the ith digit is determined by the previous sorts of the lower-order digits. Since lower-order digits are sorted before higher-order digits, this property is ensured with an stable sorting algorithm.

8.3-4 Show how to sort n integers in the range 0 to $n^3 - 1$ in $O(n)$ time.

An integer in the range 0 to $n^3 - 1$ is represented with at most $\lg n^3 = 3 \lg n$ bits. We can view a $(3 \lg n)$-bit integer as three $(\lg n)$-bit integers, so that $b = 3 \lg n$ and $r = \lg n$. With these settings, RADIX-SORT correctly sorts these numbers in

$$\Theta\left(\frac{b}{r}(n + 2^r)\right) = \Theta\left(\frac{3 \lg n}{\lg n}\left(n + 2^{\lg n}\right)\right) = \Theta(3(n + n)) = \Theta(n).$$

8.3-5 ($\star$) In the first card-sorting algorithm in this section, exactly how many sorting passes are needed to sort d-digit decimal numbers in the worst case? How many piles of cards would an operator need to keep track of in the worst case?

Suppose that the card-sorting machine represents numbers in base p, such that $2 \leq p \leq 10$. For a given value of p, we have

- Each card uses $c = \lceil \log_p 10^d \rceil$ columns;
- Each column uses up to p places.

Let cth-digit denote the most significant digit, the $(c-1)$th-digit denote the 2nd most significant digit, and so on.

The algorithm is recursive. It starts sorting on the cth-digit, which requires p piles to distribute the cards in the worst-case. In the next level, the algorithm sorts each of the p piles on the $(c-1)$th-digit, which requires p^2 piles to distribute the cards in the worst-case (each of the p piles is splitted into p piles). This process goes for c levels. Since the piles of the previous level can be reutilized in the current level, the number of piles required to distribute the cards in all levels is the number of piles required in the last level, which is

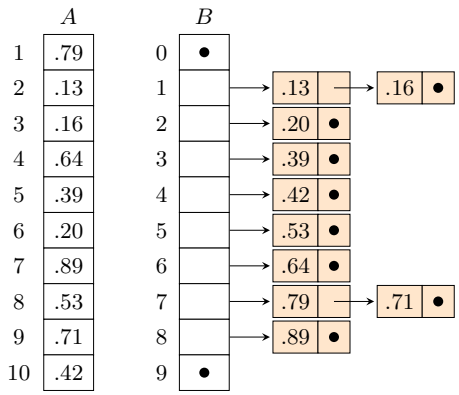
$$p^c.$$

Note that to sort the cards on the cth-digit, only one sorting pass is needed to distribute the cards into p piles. To sort on the next digit, these p piles that were sorted on the cth-digit must now be sorted on the $(c-1)$th-digit, which will require another p^1 sorting passes to distribute them into p^2 piles. In general, to sort on the ith digit, p^{c-i} sorting passes are required in the worst-case. Thus, the number of sorting passes in the worst-case is

$$\sum_{i=1}^{c} p^{c-i} = \sum_{i=0}^{c-1} p^i = \frac{p^c - 1}{p - 1}.$$

Section 8.4 – Bucket sort

8.4-1 Using Figure 8.4 as a model, illustrate the operation of BUCKET-SORT on the array $A = \langle .79, .13, .16, .64, .39, .20, .89, .53, .71, .42 \rangle$.



8.4-2 Explain why the worst-case running time for bucket sort is $\Theta(n^2)$. What simple change to the algorithm preserves its linear average-case running time and makes its worst-case running time $O(n \lg n)$?

The worst-case occurs when the input array is in decreasing order and every element falls into the same bucket. Since INSERTION-SORT takes $\Theta(n^2)$ to sort an array of size n that is in decreasing order, bucket sort will run in $\Theta(n^2)$.
The worst-case running time can be improved replacing INSERTION-SORT with HEAPSORT, which will make it run in $O(n \lg n)$. As for the average-case, the expected running time of this variation of bucket sort is

$$\begin{aligned} \mathrm{E}[T(n)] &= \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i \lg n_i)\right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}[O(n_i \lg n_i)] \\ &= \Theta(n) + \sum_{i=0}^{n-1} O(\mathrm{E}[n_i \lg n_i]). \end{aligned}$$

Using the same logic adopted in the text book to compute $\mathrm{E}[n_i^2]$, we have

$$n_i = \sum_{j=1}^{n} X_{ij},$$

which implies

$$\begin{aligned} \mathrm{E}[n_i \lg n_i] &= \mathrm{E}\left[\sum_{j=1}^{n} X_{ij} \lg \sum_{j=1}^{n} X_{ij}\right] \\ &\leq \mathrm{E}\left[\sum_{j=1}^{n} X_{ij} \sum_{j=1}^{n} X_{ij}\right] \\ &= 2 - \frac{1}{n}. \qquad \text{(from equation (8.2))} \end{aligned}$$

Thus, the average-case running time of this variation of bucket sort is

$$\mathrm{E}[T(n)] = \Theta(n) + \sum_{i=0}^{n-1} O\left(2 - \frac{1}{n}\right) = \Theta(n) + n \cdot O(1) = \Theta(n).$$

8.4-3 Let X be a random variable that is equal to the number of heads in two flips of a fair coin. What is $\mathrm{E}[X^2]$? What is $\mathrm{E}^2[X]$?

Lets define the indicator random variable

$$X_i = \mathrm{I}\{\text{flip } i \text{ comes up heap}\}.$$

Thus, we have

$$X = X_1 + X_2,$$

and

$$X^2 = (X_1 + X_2)^2 = X_1^2 + 2X_1X_2 + X_2^2.$$

Note that

$$\mathrm{E}[X_i] = \Pr\{X_i = 1\} = \frac{1}{2},$$

and

$$\mathrm{E}[X_i^2] = 1^2 \cdot \frac{1}{2} + 0^2 \cdot \frac{1}{2} = \frac{1}{2}.$$

Using the above definitions and linearity of expectation, we have

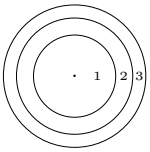
$$\begin{aligned}
\mathrm{E}[X^2] &= \mathrm{E}[X_1^2 + 2X_1X_2 + X_2^2] \\
&= \mathrm{E}[X_1^2] + 2\mathrm{E}[X_1X_2] + \mathrm{E}[X_2^2] \\
&= \frac{1}{2} + 2\left(\mathrm{E}[X_1]\mathrm{E}[X_2]\right) + \frac{1}{2} \qquad \text{(since } X_1 \text{ and } X_2 \text{ are independent)} \\
&= \frac{1}{2} + 2\left(\frac{1}{2} \cdot \frac{1}{2}\right) + \frac{1}{2} \\
&= \frac{3}{2},
\end{aligned}$$

and

$$\begin{aligned}
\mathrm{E}^2[X] &= \mathrm{E}^2[X_1 + X_2] \\
&= \mathrm{E}[X_1 + X_2]\mathrm{E}[X_1 + X_2] \\
&= (\mathrm{E}[X_1] + \mathrm{E}[X_2])(\mathrm{E}[X_1] + \mathrm{E}[X_2]) \\
&= \left(\frac{1}{2} + \frac{1}{2}\right)\left(\frac{1}{2} + \frac{1}{2}\right) \\
&= 1.
\end{aligned}$$

8.4-4 ($\star$) We are given n points in the unit circle, $p_i = (x_i, y_i)$, such that $0 < x_i^2 + y_i^2 \leq 1$ for $i = 1, 2, \ldots, n$. Suppose that the points are uniformly distributed; that is, the probability of finding a point in any region of the circle is proportional to the area of that region. Design an algorithm with an average-case running time of $\Theta(n)$ to sort the n points by their distances $d_i = \sqrt{x_i^2 + y_i^2}$ from the origin. (*Hint:* Design the bucket sizes in BUCKET-SORT to reflect the uniform distribution of the points in the unit circle.)

Considering that the points are uniformly distributed over the area of the unit circle, we can divide the circle into n rings with equal area, such that the expected number of points in each ring is 1. The figure below illustrates a circle that is divided into three rings with equal area.



We them assign to the ith bucket the points that falls within the ith ring, sort each bucket individually with INSERTION-SORT, and combine the elements of each bucket sequentially. This is basically the BUCKET-SORT algorithm with a modification on the way we assign the elements to the buckets. Since the distribution of points over the buckets is uniform, the average-case running time of this algorithm is still $O(n)$.

We now need a function that maps a point to its bucket. Let r_i denote the (larger) radius of the ith ring. We claim that

$$r_i = \frac{\sqrt{i}}{\sqrt{n}},$$

which implies that a point p_j belongs to the ith ring if, and only if,

$$\sqrt{\frac{i-1}{n}} < d_j \leq \sqrt{\frac{i}{n}},$$

squaring both sides and multiplying by n, we have

$$i - 1 < d_j^2 \cdot n \leq i,$$

which implies

$$i = \lceil d_j^2 \cdot n \rceil.$$

Proof for the radius of the ith ring.

Since the area of the unit circle is π and each ring has equal area, for $i > 0$, we have

$$\pi r_i^2 - \pi r_{i-1}^2 = \frac{\pi}{n},$$

which implies

$$r_i = \sqrt{\frac{1}{n} + r_{i-1}^2}.$$

Note that

$$r_1 = \sqrt{\frac{1}{n} + 0^2} = \frac{\sqrt{1}}{\sqrt{n}},$$

$$r_2 = \sqrt{\frac{1}{n} + \left(\sqrt{\frac{1}{n}}\right)^2} = \frac{\sqrt{2}}{\sqrt{n}},$$

$$r_3 = \sqrt{\frac{1}{n} + \left(\sqrt{\frac{2}{n}}\right)^2} = \frac{\sqrt{3}}{\sqrt{n}},$$

which lead us to assume that

$$r_i = \frac{\sqrt{i}}{\sqrt{n}},$$

for $i = 0, 1, \ldots, n$. We shall prove it by induction. Note that it holds for $i = 0$, since

$$r_0 = \frac{\sqrt{0}}{\sqrt{n}} = 0.$$

To show that it holds for $i > 0$, we need to show that if it holds for i, it also holds for $i + 1$. We have

$$r_{i+1} = \sqrt{\frac{1}{n} + \left(\sqrt{\frac{i}{n}}\right)^2} = \sqrt{\frac{i+1}{n}} = \frac{\sqrt{i+1}}{\sqrt{n}},$$

which proves the inductive step.

8.4-5 ($\star$) A ***probability distribution function*** $P(x)$ for a random variable X is defined by $P(x) = \Pr\{X \leq x\}$. Suppose that we draw a list of n random variables $X_1, X_2, \ldots, X_n$ from a continuous probability distribution function P that is computable in $O(1)$ time. Give an algorithm that sorts these numbers in linear average-case time.

Skipped.

Problems

8-1 ***Probabilistic lower bounds on comparison sorting***

In this problem, we prove a probabilistic $\Omega(n \lg n)$ lower bound on the running time of any deterministic or randomized comparison sort on n distinct input elements. We begin by examining a deterministic comparison sort A with decision tree T_A. We assume that every permutation of A's inputs is equally likely.

a. Suppose that each leaf of T_A is labeled with the probability that it is reached given a random input. Prove that exactly $n!$ leaves are labeled $1/n!$ and that the rest are labeled 0.

b. Let $D(T)$ denote the external path length of a decision tree T; that is, $D(T)$ is the sum of the depths of all the leaves of T. Let T be a decision tree with $k > 1$ leaves, and let LT and RT be the left and right subtrees of T. Show that $D(T) = D(LT) + D(RT) + k$.

c. Let $d(k)$ be the minimum value of $D(T)$ over all decision trees T with $k > 1$ leaves. Show that $d(k) = \min_{1 \le i \le k-1}\{d(i) + d(k-i) + k\}$. (Hint: Consider a decision tree T with k leaves that achieves the minimum. Let i_0 be the number of leaves in LT and $k - i_0$ the number of leaves in RT.)

d. Prove that for a given value of $k > 1$ and i in the range $1 \le i \le k-1$, the function $i \lg i + (k-i)\lg(k-i)$ is minimized at $i = k/2$. Conclude that $d(k) = \Omega(k \lg k)$.

e. Prove that $D(T_A) = \Omega(n! \lg(n!))$, and conclude that the average-case time to sort n elements is $\Omega(n \lg n)$.

Now, consider a *randomized* comparison sort B. We can extend the decision-tree model to handle randomization by incorporating two kinds of nodes: ordinary comparison nodes and "randomization" nodes. A randomization node models a random choice of the form $\textsc{Random}(l, r)$ made by algorithm B; the node has r children, each of which is equally likely to be chosen during an execution of the algorithm.

f. Show that for any randomized comparison sort B, there exists a deterministic comparison sort A whose expected number of comparisons is no more than those made by B.

(a) First note that there are $n!$ distinct permutations of an input of size n with distinct elements. To be a valid decision tree, T_A must have each of these permutations as a reachable leaf. Since every permutation is equally likely, each of these leaves must be reachable with probability $1/n!$. If T_A has any additional leaf, it is not reachable for any input and has probability 0.

(b) Note that, since $k > 1$, the root of T if not a leaf. Also, the only node that is present in T and is not present in the subtrees LT and RT together is the root of T. This implies that each leaf at depth d in T is either in LT or in RT, but at depth $d-1$. Since there are k leaves in T, we have $D(T) = D(LT) + D(RT) + k$.

(c) First, we will show that

$$d(k) \le \min_{1 \le i \le k-1}\{d(i) + d(k-i) + k\},$$

by showing that

$$d(k) \le d(i) + d(k-i) + k,$$

for $i = 1, \ldots, k-1$. For every i such that $1 \le i \le k-1$, there exist decision trees LT and RT with i leaves and $k-i$ leaves, respectivelly, such that $D(LT) = d(i)$ and $D(RT) = d(k-i)$. Let T be a decision tree composed by a root node and LT and RT as its left and right subtrees, respectivelly. Note that T has k leaves, since it has all the leaves from LT and RT, and its root is not a leaf. Then, we have

$$\begin{aligned} d(k) &\le D(T) && \text{(from the definition of } d(\cdot)\text{)} \\ &= D(LT) + D(RT) + k && \text{(from item (b))} \\ &= d(i) + d(k-i) + k. \end{aligned}$$

Now, we will show that

$$d(k) \ge \min_{1 \le i \le k-1}\{d(i) + d(k-i) + k\},$$

by showing that

$$\exists\, i \in 1, \ldots, k-1 \mid d(k) \ge d(i) + d(k-i) + k.$$

Let T be a decision tree with k leaves such that $D(T) = d(k)$. Let LT and RT be the left and right subtrees of T and let i and $k-i$ be the number of leaves of LT and RT, respectivelly. We have

$$\begin{aligned} d(k) &= D(T) && \text{(from the definition of } T\text{)} \\ &= D(LT) + D(RT) + k && \text{(from item (b))} \\ &\ge d(i) + d(k-i) + k. \end{aligned}$$

Then, we can conclude that

$$d(k) = \min_{1 \le i \le k-1}\{d(i) + d(k-i) + k\}.$$

(d) Let $f(i) = i \lg i + (k-i)\lg(k-i)$. We have

$$\begin{aligned} f'(i) &= \frac{d}{di}\left(i \lg i + (k-i)\lg(k-i)\right) \\ &= \frac{d}{di}\left(\frac{i \ln i + (k-i)\ln(k-i)}{\ln 2}\right) \\ &= \frac{d}{di}\left(\frac{i \ln i}{\ln 2} + \frac{(k-i)\ln(k-i)}{\ln 2}\right) \\ &= \frac{1+\ln i}{\ln 2} + \frac{-1-\ln(k-i)}{\ln 2} \\ &= \frac{\ln i - \ln(k-i)}{\ln 2} \\ &= \lg i - \lg(k-i), \end{aligned}$$

which is 0 when $i = k/2$. Also note that

$$f'(1) = f'(k-1) = \ln(k-1) \geq \ln 1 = 0 = f(k/2),$$

which implies that the point $i = k/2$ is a minimum. We shall now prove that $d(k) = \Omega(k \lg k)$. Our guess is

$$d(i) \geq ci \lg i \ \forall 1 \leq i \leq k-1,$$

where c is a positive constant. Substituting into the recurrence, yields

$$\begin{aligned} d(k) &= \min_{1 \leq i \leq k-1}\{d(i) + d(k-i) + k\} \\ &\geq \min_{1 \leq 1 \leq k-1}\{c(i \lg i + (k-i)\lg(k-i)) + k\} \\ &= \min_{1 \leq 1 \leq k-1}\{cf(i) + k\} \\ &= cf\left(\frac{k}{2}\right) + k \\ &= c\left(\frac{k}{2}\lg\frac{k}{2} + \frac{k}{2}\lg\frac{k}{2}\right) + k \\ &= ck\lg\frac{k}{2} + k \\ &= ck\lg k - ck + k \\ &\geq ck \lg k, \end{aligned}$$

where the last step holds as long as $c \leq 1$.

(e) Since T_A has $n!$ leaves, we have

$$D(T_A) \geq d(n!) = \Omega(n! \lg n!).$$

Note that the external path length of a decision tree denotes the number of comparisons needed to sort every possible permutation of an input of size n. Since each of the $n!$ permutations have the same probability of happening (from item (a)), the expected number of comparisons for each input is

$$\begin{aligned} \frac{\Omega(n!\lg(n!))}{n!} &= \Omega(\lg(n!)) \\ &= \Omega(n \lg n). \quad \text{(from (3.19))} \end{aligned}$$

(f) Just replace each randomization node with one of its subtrees, particularly the one with the smaller external path. The result is a valid deterministic tree with no more comparisons than the radomized tree. Thus, we can conclude that the number of comparisons for any randomized comparison sort is also $\Omega(n \lg n)$.

8-2 ***Sorting in place in linear time***

Suppose that we have an array of n data records to sort and that the key of each record has the value 0 or 1. An algorithm for sorting such a set of records might possess some subset of the following three desirable characteristics:

1. The algorithm runs in $O(n)$ time.
2. The algorithm is stable.
3. The algorithm sorts in place, using no more than a constant amount of storage space in addition to the original array.

a. Give an algorithm that satisfies criteria 1 and 2 above.

b. Give an algorithm that satisfies criteria 1 and 3 above.

c. Give an algorithm that satisfies criteria 2 and 3 above.

d. Can you use any of your sorting algorithms form parts (a)−(c) as the sorting method used in line 2 of RADIX-SORT, so that RADIX-SORT sorts n records with b-bit keys in $O(bn)$ time? Explain how or why not.

e. Suppose that the n records have keys in the range from 1 to k. Show how to modify counting sort so that it sorts the records in place in $O(n+k)$ time. You may use $O(k)$ storage outside the input array. Is your algorithm stable? (*Hint:* How would you do it for $k = 3$?)

(a) The following is a modified COUNTING-SORT for data records. If runs in $\Theta(n)$, is stable, but do not sort in place.

```
   Counting-Sort-Binary-Records(A, B)
 1 | one-pos = 1
 2 | zero-pos = 1
 3 | for i = 1 to A.length do
 4 | |  key = A[i].key
 5 | |  if key == 0 then
 6 | |  |  one-pos = one-pos + 1
 7 | for i = 1 to A.length do
 8 | |  key = A[i].key
 9 | |  if key == 0 then
10 | |  |  B[zero-pos] = A[i]
11 | |  |  zero-pos = zero-pos + 1
12 | |  else
13 | |  |  B[one-pos] = A[i]
14 | |  |  one-pos = one-pos + 1
```

(b) The following uses a technique similar to the one used for the HOARE-PARTITION, introduced on Question 7-1. It runs in $\Theta(n)$, sorts in place, but is not stable.

```
   InPlace-Sort-Binary-Records(A)
 1 | i = 1
 2 | j = A.length
 3 | while True do
 4 | |  while j > i and A[j].key == 1 do
 5 | |  |  j = j - 1
 6 | |  while i < j and A[i].key == 0 do
 7 | |  |  i = i + 1
 8 | |  if i < j then
 9 | |  |  swap A[i] with A[j]
10 | |  else
11 | |  |  break
```

(c) The following is a modified INSERTION-SORT for data records. It is stable, sorts in place, but is not linear.

```
   Insertion-Sort-Binary-Records(A)
 1 | for j = 2 to A.length do
 2 | |  key = A[j].key
 3 | |  i = j - 1
 4 | |  while i > 0 and A[i].key > key do
 5 | |  |  A[i + 1] = A[i]
 6 | |  |  i = i - 1
 7 | |  A[i + 1] = key
```

(d) To be used as a subroutine of RADIX-SORT, the sorting algorithm must be stable. Also, in order to RADIX-SORT run in $O(bn)$ time, the sorting subroutine must run in $O(n)$. The only algorithm that satisfies both of these constraints is the one on part (a).

(e) Roughly speaking, instead of copying each element to its correct position in a new array, we swap each element with the element that is in its correct position. The algorithm runs in $\Theta(n+k)$, sorts in place, but is not stable. The pseudocode is stated below.

```
   Counting-Sort-InPlace(A, k)
 1 | let C[0,...,k] be a new array
 2 | let T[0,...,k] be a new array
 3 | for i = 0 to k do
 4 | |  C[i] = 0
 5 | for i = 1 to A.length do
 6 | |  key = A[i].key
 7 | |  C[key] = C[key] + 1
 8 | for i = 1 to k do
 9 | |  C[i] = C[i] + C[i - 1]
10 | for i = 0 to k do
11 | |  T[i] = C[i]
12 | pos = 1
13 | while pos < A.length do
14 | |  key = A[pos].key
15 | |  if C[key] > pos then
16 | |  |  swap A[pos] with A[C[key]]
17 | |  |  C[key] = C[key] - 1
18 | |  else
19 | |  |  pos = pos + T[key]
```

8-3 ***Sorting variable-length items***

a. You are given an array of integers, where different integers may have different numbers of digits, but the total number of digits over all the integers in the array is n. Show how to sort the array in $O(n)$ time.

b. You are given an array of strings, where different strings may have different numbers of characters, but the total number of characters over all the strings is n. Show how to sort the strings in $O(n)$ time. (Note that the desired order here is the standard alphabetical order; for example, `a` < `ab` < `b`.)

(a) Let m denote the number of integers in the input array. Start sorting the numbers by their sign, using an algorithm similar to COUNTING-SORT-BINARY-RECORDS, in which negative numbers are placed before positive numbers. This step runs in $O(m)$. Let m_{pos} and m_{neg} denote the number of positive and negative integers, respectivelly. For each group of numbers with the same sign, sort the elements in the group by their number of digits with COUNTING-SORT. This step will take $O(m_{pos} + n) + O(m_{neg} + n) = O(n)$. Let m_i denote the number of integers with i digits. For each group of numbers with i digits, sort the elements in the group with RADIX-SORT. Since each call to RADIX-SORT runs in $O(i \cdot (m_i + 10)) = O(i \cdot m_i)$, this step will take

$$\sum_{i=1}^{n} O(i \cdot m_i) = O\left(\sum_{i=1}^{n} (i \cdot m_i)\right) = O(n).$$

Thus, the running time of this algorithm is

$$O(m + n + n) = O(n).$$

(b) Note that, different from the problem of sorting integers, we can not use the length of the string to sort the elements – strings with longer length does not imply any order with respect to strings with smaller length. However, the leftmost character of each string does imply an order. When the first character of two strings are the same, the strings are untied by their following character, and so on until the last character. We can use this notion to derive a recursive algorithm that sorts and groups the strings by their first character, and recursively sorts each group by their following character. Note that strings that do not have the current character should be placed before the ones that does have the character. Since the range of characters is constant, we can sort each group using counting sort in linear time on the number of elements in the group. To avoid creating multiple subarrays in each recursion call, COUNTING-SORT-INPLACE is more suitable.

Let m denote the number of strings in the input array and let n_i denote the number of characters in the ith string. Since counting sort is linear on the number of elements being sorted and the ith string are considered in at most $n_i + 1$ calls of counting sort (one additional time for the special case when the string does not have the following character), the total cost of this algorithm is

$$O\left(\sum_{i=1}^{m} (n_i + 1)\right) = O\left(\sum_{i=1}^{m} n_i + m\right) = O(n + m) = O(n).$$

8-4 ***Water jugs***

Suppose that you are given n red and n blue water jugs, all of different shapes and sizes. All red jugs hold different amounts of water, as do the blue ones. Moreover, for every red jug, there is a blue jug that holds the same amount of water, and vice versa.

Your task is to find a grouping of the jugs into pairs of red and blue jugs that hold the same amount of water. To do so, you may perform the following operation: pick a pair of jugs in which one is red and one is blue, fill the red jug with water, and then pour the water into the blue jug. This operation will tell you whether the red or the blue jug can hold more water, or that they have the same volume. Assume that such a comparison takes one time unit. Your goal is to find an algorithm that makes a minimum number of comparisons to determine the grouping. Remember that you may not directly compare two red jugs or two blue jugs.

a. Describe a deterministic algorithm that uses $\Theta(n^2)$ comparisons to group the jugs into pairs.

b. Prove a lower bound of $\Omega(n \lg n)$ for the number of comparisons that an algorithm solving this problem must take.

c. Give a randomized algorithm whose expected number of comparisons is $O(n \lg n)$, and prove that this bound is correct. What is the worst-case number of comparisons for your algorithm?

(a) A brute-force algorithm runs in $\Theta(n^2)$. Compare the volume of each red jub to each blue jug until a match is found. The pseudocode is stated below.

```
Water-Jugs-Brute-Force(R, B)
1  n = R.length
2  let P be a new array of size n
3  for i = 1 to n do
4      for j = 1 to n do
5          if Volume(R[i]) == Volume(B[j]) then
6              P[i] = (R[i], B[j])
7              break
```

(b) We shall use a decision tree to compute a lower bound on the worst-case number of comparisons needed to solve this problem. Let each node on the tree be a comparison between one red jug and one blue jug. Each comparison results in one of three results, which implies that each non-leaf node in the decision tree has three children. To compute the number of leaves, note that this problem is the same as sorting the array of blue jugs based on the current order of the array of red jugs. Since the elements of each array are distinct, there are $n!$ different permutations of the array of blue jugs array and only one of them correspond to the order of the array of red jugs, which then implies that the decision tree must have at least $n!$ leaves. Note that the height h of the decision tree is equal to the worst-case number of comparisons needed to solve the problem. Thus, we have

$$3^h \geq n!,$$

which implies

$$\begin{aligned} h &\geq \log_3(n!) \\ &= \frac{\lg n!}{\lg 3} \\ &= \frac{\Omega(n \lg n)}{\lg 3} \quad \text{(from (3.19))} \\ &= \Omega(n \lg n). \end{aligned}$$

(c) We can use an algorithm very similar to Randomized-Quicksort. The partition procedure also need to be updated in order to receive the pivot as an argument. The pseudocode is stated below.

```
Partition-Jugs(A, s, e, x)
1  i = s − 1
2  for j = s to e − 1 do
3      v_j = Volume(A[j])
4      v_x = Volume(x)
5      if v_j == v_x then
6          exchange A[j] with A[e]
7      if v_j < v_x then
8          i = i + 1
9          exchange A[i] with A[j]
10 exchange A[i + 1] with A[e]
11 return i + 1
```

```
Group-Jugs(R, B, s, e, G)
1  | if e < s then
2  | |  return
3  | else if e == s then
4  | |  Insert(G, (R[s], B[s]))
5  | else
6  | |  i = Random(s, e)
7  | |  b = Partition-Jugs(B, s, e, R[i])
8  | |  r = Partition-Jugs(R, s, e, B[b])
9  | |  Insert(G, (R[r], B[b]))
10 | |  Group-Jugs(R, B, s, i − 1)
11 | |  Group-Jugs(R, B, i + 1, e)
```

The analysis this algorithm is almost the same as the analysis of Randomized-Quicksort, which gives an expected running time of $O(n \lg n)$. Also just like in Randomized-Quicksort, the worst-case occurs when partition always make bad partitions, which gives a running time of $\Theta(n^2)$.

8-5 Suppose that, instead of sorting an array, we just require that the elements increase on average. More precisely, we call an n-element array A ***k-sorted*** if, for all $i = 1, 2, \ldots, n-k$, the following holds:

$$\frac{\sum_{j=i}^{i+k-1} A[j]}{k} \leq \frac{\sum_{j=i+1}^{i+k} A[j]}{k}.$$

a. What does it mean for an array to be 1-sorted?

b. Give a permutation of the numbers $1, 2, \ldots, 10$ that is 2-sorted, but not sorted.

c. Prove that an n-element array is k-sorted if and only if $A[i] \leq A[i+k]$ for all $i = 1, 2, \ldots, n-k$.

d. Give an algorithm that k-sorts an n-element array in $O(n \lg(n/k))$ time.

We can also show a lower bound on the time to produce a k-sorted array, when k is a constant.

e. Show that we can sort a k-sorted array of length n in $O(n \lg k)$ time. (*Hint:* Use the solution to Exercise 6.5-9.)

f. Show that when k is a constant, k-sorting an n-element array requires $\Omega(n \lg n)$ time. (*Hint:* Use the solution to the previous part along with the lower bound on comparison sorts.)

(a) It means that it is sorted.

(b) $\langle 2, 1, 4, 3, 6, 5, 8, 7, 10, 9 \rangle$.

(c) Note that

$$\frac{\sum_{j=i}^{i+k-1} A[j]}{k} = \frac{\sum_{j=i+1}^{i+k-1} A[j]}{k} + \frac{A[i]}{k},$$

and

$$\frac{\sum_{j=i+1}^{i+k} A[j]}{k} = \frac{\sum_{j=i+1}^{i+k-1} A[j]}{k} + \frac{A[i+k]}{k}.$$

Thus, we have

$$\frac{\sum_{j=i+1}^{i+k-1} A[j]}{k} + \frac{A[i]}{k} \leq \frac{\sum_{j=i+1}^{i+k-1} A[j]}{k} + \frac{A[i+k]}{k} \iff \frac{A[i]}{k} \leq \frac{A[i+k]}{k}. \iff A[i] \leq A[i+k].$$

(d) We can use a modification of Randomized-Quicksort that only sorts subarrays with size greater than k. The modified pseudocode is stated below.

```
Randomized-Quicksort-K(A, p, r, k)
1 | if r − p ≥ k then
2 | | q = Randomized-Partition(A, p, r)
3 | | Randomized-Quicksort(A, p, q − 1, k)
4 | | Randomized-Quicksort(A, q + 1, r, k)
```

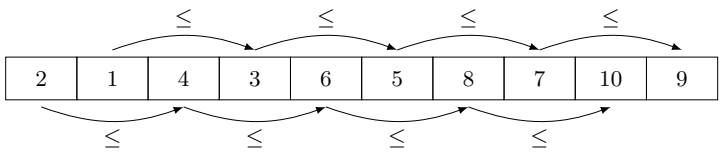
Correctness can be seen as follows. Note that at the start of each recursive call of Randomized-Quicksort-K, every element of the subarray $A[p, \ldots, r]$ is greater than or equal to the elements before the subarray and is smaller than or equal to the elements after the subarray. That is, at the start of each recursive call, the elements of the subarray $A[p, \ldots, r]$ can only be inverted with themselves. Since the base case of the recursion occurs when $r - p < k$, after running Randomized-Quicksort-K we can ensure that every inversion (i, j) has $j - i < k \to j < i + k$, which implies

$$A[i] \leq A[i+k],$$

for all $i = 1, 2, \ldots, n-k$. From item (c), this property implies that the array is correctly k-sorted.

As for the running time, we can use a similar argument as the one used on question 7.4-5. Thus, the expected running time of Randomized-Quicksort-K is $O\left(n \lg \frac{n}{k}\right)$.

(e) A k-sorted array is composed of k 1-sorted subarrays where the ith subarray, $i = 0, 1, \ldots, k-1$, is formed by the elements with index j such that $j \bmod k = i$. For instance, the following 2-sorted array has two 1-sorted subarrays:



The Merge-Lists-Min-Heap algorithm (Question 6-5.9) can be used to merge these k subarrays in $O(n \lg k)$-time.

(f) Item (e) shows that a k-sorted array has k 1-sorted subarrays. Note that each of these subarrays must have at least $\lfloor n/k \rfloor$ elements. Thus, since k is a constant, the lower bound to sort each of these subarrays is

$$\Omega\left(\frac{n}{k} \lg \frac{n}{k}\right) = \Omega\left(\frac{n}{k} \lg n - \frac{n}{k} \lg k\right) = \Omega(n \lg n),$$

which implies that $\Omega(n \lg n)$ is also the lower bound to k-sort the whole array.

8-6 ***Lower bound on merging sorted lists***

The problem of merging two sorted lists arises frequently. We have seen a procedure for it as the subroutine MERGE in Section 2.3.1. In this problem, we will prove a lower bound of $2n - 1$ on the worst-case number of comparisons required to merge two sorted lists, each containing n items.

First we will show a lower bound of $2n - o(n)$ comparisons by using a decision tree,

a. Given $2n$ numbers, compute the number of possible ways to divide them into two sorted lists, each with n numbers.

b. Using a decision tree and your answer to part (a), show that any algorithm that correctly merges two sorted lists must perform at least $2n - o(n)$ comparisons.

Now we will show a slightly tigher $2n - 1$ bound.

c. Show that if two elements are consecutive in the sorted order and from different lists, then they must be compared.

d. Use your answer to the previous part to show a lower bound of $2n - 1$ comparisons for merging two sorted lists.

(a) Note that every list of elements only have one permutation that is in sorted order. Thus, this problem can be seen as counting the number of ways to pick n numbers out of $2n$ numbers to form the first list, and using the remaining numbers to form the second list. We can count that with the binomial notation

$$\binom{2n}{n} = \frac{2^{2n}}{\sqrt{\pi n}}\left(1 + O\left(\frac{1}{n}\right)\right),$$

in which the approximation is proved in (C.1-13).

(b) The input of the merge algorithm are the two sorted lists and we know from item (a) that the number of possible ways to form these lists elements is $\binom{2n}{n}$. Note that these pairs of sorted lists are unique and each of them can be the final merged list if placed side by side. Thus, the number of permutations of the input to form the final merged list is also $\binom{2n}{n}$. We can compute a lower bound on the number of comparisons to solve this problem by determining the height of the decision tree with at least $\binom{2n}{n}$ leaves. Let h be the height of such a decision tree. Thus, we have

$$\begin{aligned} 2^h &\geq \frac{2n}{n} \\ &= \frac{2^{2n}}{\sqrt{\pi n}}\left(1 + O\left(\frac{1}{n}\right)\right), \end{aligned}$$

which implies

$$\begin{aligned} h &\geq \lg\left(\frac{2^{2n}}{\sqrt{\pi n}}\left(1 + O\left(\frac{1}{n}\right)\right)\right) \\ &= \lg(2^{2n}) - \lg(\sqrt{\pi n}) + \lg\left(1 + O\left(\frac{1}{n}\right)\right) \\ &= 2n - o(n). \end{aligned}$$

(c) Let A and B denote the two input sorted lists and assume without loss of generality that their elements are distinct. Let $a \in A$ and $b \in B$ denote two elements that are consecutive in the sorted order. Note that since they are consecutive, the number of elements that are before/after a in the sorted list is equal to the number of elements before/after b in the sorted list. This implies that if we compare a to every element of B except b and compare b to every element of A except a, we will find the two possible positions for them in the sorted list, but we will not be able to determine which one comes first.

(d) In the worst case, every pair of elements in the final list comes from different input lists. There are $2n$ elements in the final list and $2n - 1$ pairs of consecutive elements. Thus, $2n - 1$ is a lower bound on the worst-case number of comparisons to merge two sorted lists.

8-7 ***The 0-1 sorting lemma and columnsort***
Very long statement. Read it on the text book.

Skipped.

Section A.1 – Summation formulas and properties

A.1-1 Find a simple formula for $\sum_{k=1}^{n}(2k-1)$.

$$\begin{aligned}\sum_{k=1}^{n}(2k-1) &= \sum_{k=1}^{n}2k - \sum_{k=1}^{n}1 \\ &= 2\sum_{k=1}^{n}k - n \\ &= 2\cdot\frac{1}{2}n(n+2) - n \\ &= n^2 + n - n \\ &= n^2.\end{aligned}$$

A.1-2 ($\star$) Show that $\sum_{k=1}^{n}1/(2k-1) = \ln(\sqrt{n}) + O(1)$ by manipulating the harmonic series.

$$\begin{aligned}\sum_{k=1}^{n}1/(2k-1) &= \frac{1}{1} + \frac{1}{3} + \frac{1}{5} + \cdots + \frac{1}{2n-3} + \frac{1}{2n-1} \\ &= \left(\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{2n}\right) - \frac{1}{2}\left(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}\right) \\ &= \sum_{k=1}^{2n}\frac{1}{k} - \frac{1}{2}\sum_{k-1}^{n}\frac{1}{k} \\ &= \ln 2n + O(1) - \frac{1}{2}(\ln n + O(1)) \\ &= \ln n + \ln 2 + O(1) - \frac{1}{2}\ln n - \frac{1}{2}O(1) \\ &= \frac{1}{2}\ln n + O(1) \\ &= \ln(\sqrt{n}) + O(1).\end{aligned}$$

A.1-3 Show that $\sum_{k=0}^{\infty}k^2x^k = x(1+x)/(1-x)^3$ for $0 < |x| < 1$.

From Equation A.8, we have

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

differentiating both sides and multiplying by x, we have

$$\begin{aligned}\sum_{k=0}^{\infty}k^2x^k &= x\cdot\frac{1\cdot(1-x)^2 - (2\cdot(1-x)\cdot(-1)\cdot x)}{(1-x)^4} \\ &= x\cdot\frac{(1-x)(1-x) + (1-x)\cdot 2x}{(1-x)^4} \\ &= x\cdot\frac{(1-x)+2x}{(1-x)^3} \\ &= \frac{x(1+x)}{(1-x)^3}.\end{aligned}$$

A.1-4 ($\star$) Show that $\sum_{k=0}^{\infty}(k-1)/2^k = 0$.

$$\begin{aligned}
\sum_{k=0}^{\infty}(k-1)/2^k &= \sum_{k=0}^{\infty}\left(\frac{k}{2^k} - \frac{1}{2^k}\right) \\
&= \sum_{k=0}^{\infty} k\frac{1}{2^k} - \sum_{k=0}^{\infty}\frac{1}{2^k} \\
&= \sum_{k=0}^{\infty} k\left(\frac{1}{2}\right)^k - \sum_{k=0}^{\infty}\left(\frac{1}{2}\right)^k \\
&= \frac{(1/2)}{(1-(1/2))^2} - \frac{1}{1-(1/2)} \\
&= \frac{(1/2)}{1-1-(1/4)} - 2 \\
&= 4/2 - 2 \\
&= 0.
\end{aligned}$$

A.1-5 ($\star$) Evaluate the sum $\sum_{k=1}^{\infty}(2k+1)x^{2k}$ for $|x| < 1$.

$$\begin{aligned}
\sum_{k=1}^{\infty}(2k+1)x^{2k} &= \frac{d}{dx}\cdot\sum_{k=1}^{\infty}x^{2k+1} \\
&= \frac{d}{dx}\cdot x\cdot\sum_{k=1}^{\infty}x^{2k} \\
&= \frac{d}{dx}\cdot x\cdot\sum_{k=0}^{\infty}(x^2)^k - x \\
&= \frac{d}{dx}\cdot x\cdot\frac{1}{1-x^2} - x \\
&= \frac{d}{dx}\cdot\frac{x-x(1-x^2)}{1-x^2} \\
&= \frac{d}{dx}\cdot\frac{x^3}{1-x^2} \\
&= \frac{3x^2(1-x^2)-(-2x)x^3}{(1-x^2)^2} \\
&= \frac{3x^2-3x^4+2x^4}{(1-x^2)^2} \\
&= \frac{(3-x^2)\cdot x^2}{(1-x^2)^2}.
\end{aligned}$$

A.1-6 Prove that $\sum_{k=1}^{n} O(f_k(i)) = O(\sum_{k=1}^{n} f_k(i))$ by using the linearity property of summations.

Skipped.

A.1-7 Evaluate the product $\prod_{k=1}^{n} 2 \cdot 4^k$.

We have

$$\prod_{k=1}^{n}(2 \cdot 4^k) = 2^{\lg\left(\prod_{k=1}^{n}(2 \cdot 4^k)\right)},$$

and

$$\begin{aligned}
\lg\left(\prod_{k=1}^{n}(2 \cdot 4^k)\right) &= \sum_{k=1}^{n} \lg(2 \cdot 2^{2k}) \\
&= \sum_{k=1}^{n} \lg 2^{2k+1} \\
&= \sum_{k=1}^{n}(2k+1) \\
&= 2\sum_{k=1}^{n} k + \sum_{k=1}^{n} 1 \\
&= n(n+1) + n \\
&= n(n+2).
\end{aligned}$$

Thus,

$$\prod_{k=1}^{n}(2 \cdot 4^k) = 2^{n(n+2)}.$$

A.1-8 ($\star$) Evalute the product $\prod_{k=2}^{n}(1 - 1/k^2)$.

We have

$$\prod_{k=2}^{n}\left(1 - \frac{1}{k^2}\right) = 2^{\lg\left(\sum_{k=2}^{n} \lg(1-1/k^2)\right)},$$

and

$$\begin{aligned}
\sum_{k=2}^{n} \lg\left(1 - \frac{1}{k^2}\right) &= \sum_{k=2}^{n} \lg\left(\frac{k^2-1}{k^2}\right) \\
&= \sum_{k=2}^{n} \lg\left(\frac{(k-1)}{k} \cdot \frac{(k+1)}{k}\right) \\
&= \sum_{k=2}^{n} \left(\lg\left(\frac{k-1}{k}\right) + \lg\left(\frac{k+1}{k}\right)\right) \\
&= \lg\frac{1}{2} + \lg\frac{3}{2} + \lg\frac{2}{3} + \lg\frac{4}{3} + \lg\frac{3}{4} + \lg\frac{5}{4} + \cdots + \lg\frac{n-2}{n-1} + \lg\frac{n}{n-1} + \lg\frac{n-1}{n} + \lg\frac{n+1}{n} \\
&= \lg 1 - \lg 2 + \cancel{\lg 3} - \cancel{\lg 2} + \cancel{\lg 2} - \cancel{\lg 3} + \cancel{\lg 4} - \cancel{\lg 3} + \cancel{\lg 3} - \cancel{\lg 4} + \cancel{\lg 5} - \cancel{\lg 4} + \ldots \\
&\quad + \cancel{\lg(n-2)} - \cancel{\lg(n-1)} + \cancel{\lg n} - \cancel{\lg(n-1)} + \cancel{\lg(n-1)} - \cancel{\lg n} + \lg(n+1) - \lg n \\
&= 0 - 1 + \lg(n+1) - \lg n \\
&= \lg(n+1) - \lg(n) - 1.
\end{aligned}$$

Thus,

$$\prod_{k=2}^{n}\left(1 - \frac{1}{k^2}\right) = 2^{(\lg(n+1)-(\lg(n)+1))} = \frac{2^{\lg(n+1)}}{2^{\lg(n)+1}} = \frac{n+1}{2^{\lg n} \cdot 2} = \frac{n+1}{2n}.$$

Section A.2 – Bounding summations

A.2-1 Show that $\sum_{k=1}^{n} 1/k^2$ is bounded above by a constant.

$$\begin{aligned}
\sum_{k=1}^{n} &= 1 + \sum_{k=2}^{n} \frac{1}{k^2} \\
&\leq 1 + \int_1^n \frac{dx}{x^2} \\
&= 1 + \left(-\frac{1}{x}\Big|_1^n \right) \\
&= 1 + \left(-\frac{1}{n} - \left(-\frac{1}{1} \right) \right) \\
&= 2 - \frac{1}{n} \\
&\leq 2.
\end{aligned}$$

A.2-2 Find an asymptotic upper bound on the summation

$$\sum_{k=0}^{\lfloor \lg n \rfloor} \lceil n/2^k \rceil.$$

$$\begin{aligned}
\sum_{k=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^k} \right\rceil &= n \cdot \sum_{k=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{1}{2^k} \right\rceil \\
&\leq n \cdot \sum_{k=0}^{\lg n} \left(\frac{1}{2^k} + 1 \right) \\
&= n \cdot \sum_{k=0}^{\lg n} \left(\frac{1}{2^k} \right) + \sum_{k=0}^{\lg n} 1 \\
&= n \cdot \frac{1}{1 - (1/2)} + \lg n + 1 \\
&= 2n + \lg n + 1 \\
&= O(n).
\end{aligned}$$

A.2-3 Show that the nth harmonic number is $\Omega(\lg n)$ by splitting the summation.

$$\begin{aligned}
\sum_{k=1}^{n} \frac{1}{k} &\geq \sum_{i=0}^{\lfloor \lg n \rfloor - 1} \sum_{j=0}^{2^i - 1} \frac{1}{2^i + j} \\
&\geq \sum_{i=0}^{\lfloor \lg n \rfloor - 1} \sum_{j=0}^{2^i - 1} \frac{1}{2^{i+1}} \\
&= \sum_{i=0}^{\lfloor \lg n \rfloor - 1} \frac{1}{2} \cdot \sum_{j=0}^{2^i - 1} \frac{1}{2^i} \\
&= \sum_{i=0}^{\lfloor \lg n \rfloor - 1} \frac{1}{2} \\
&\geq \sum_{i=0}^{\lg n - 2} \frac{1}{2} \\
&= \frac{1}{2}(\lg(n) - 1) \\
&= \Omega(\lg n).
\end{aligned}$$

A.2-4 Approximate $\sum_{k=1}^{n} k^3$ with an integral.

We have

$$\int_0^n x^3 dx \leq \sum_{k=1}^{n} k^3 \leq \int_1^{n+1} x^3 dx.$$

For a lower bound, we obtain

$$\sum_{k=1}^{n} k^3 \geq \int_0^n x^3 dx = \left.\frac{x^4}{4}\right|_0^n = \frac{n^4}{4} = \Omega(n^4).$$

For the upper bound, we obtain

$$\sum_{k=1}^{n} k^3 \leq \int_1^{n+1} x^3 dx = \left.\frac{x^4}{4}\right|_1^{n+1} = \frac{(n+1)^4 - 1}{4} = O(n^4).$$

Thus,

$$\sum_{k=1}^{n} k^3 = \Theta(n^4).$$

A.2-5 Why didn't we use the integral approximation (A.12) directly on $\sum_{k=1}^{n} 1/k$ to obtain an upper bound on the nth harmonic number?

Applying (A.12) directly, we obtain

$$\sum_{k=1}^{n} \frac{1}{k} \leq \int_0^n \frac{1}{x} dx,$$

but the function $1/x$ is undefined for $x = 0$ (because of the division by zero).

Problems

A-1 ***Bounding summations***

Give asymptotically tight bounds on the following summations. Assume that $r \geq 0$ and $s \geq 0$ are constants.

a. $\sum_{k=1}^{n} k^r$.

b. $\sum_{k=1}^{n} \lg^s k$.

c. $\sum_{k=1}^{n} k^r \lg^s k$.

(a) For a lower bound, we have

$$\begin{aligned}\sum_{k=1}^{n} k^r &\geq \int_0^n x^r dx \\ &= \left.\frac{x^{(r+1)}}{r+1}\right|_0^n \\ &= \frac{n^{(r+1)}}{r+1} - \frac{0^{(r+1)}}{r+1} \\ &\geq n^{(r+1)} \\ &= \Omega(n^{(r+1)}),\end{aligned}$$

and for the upper bound, we have

$$\sum_{k=1}^{n} k^r \leq \sum_{k=1}^{n} n^r = n^{(r+1)} = O(n^{(r+1)}).$$

Thus,

$$\sum_{k=1}^{n} = \Theta(n^{(r+1)}).$$

(b) For a lower bound, we have

$$\begin{aligned}\sum_{k=1}^{n} \lg^s k &= \sum_{k=1}^{n/2} \lg^s k + \sum_{k=n/2+1}^{n} \lg^s k \\ &\geq \sum_{k=1}^{n/2} 0 + \sum_{k=n/2+1}^{n} \lg^s \left(\frac{n}{2}\right) \\ &= \frac{n}{2} \lg^s \left(\frac{n}{2}\right) \\ &= \frac{n}{2} \lg^s n - \frac{n}{2} \lg^s 2 \\ &\geq \frac{1}{2} n \lg^s n - \frac{1}{2} n \\ &= \Omega(n \lg^s n),\end{aligned}$$

and for the upper bound, we have

$$\sum_{k=1}^{n} \lg^s k \leq \sum_{k=1}^{n} \lg^s n = n \lg^s n = O(n \lg^s n).$$

Thus,

$$\sum_{k=1}^{n} \lg^s k = \Theta(n \lg^s n).$$

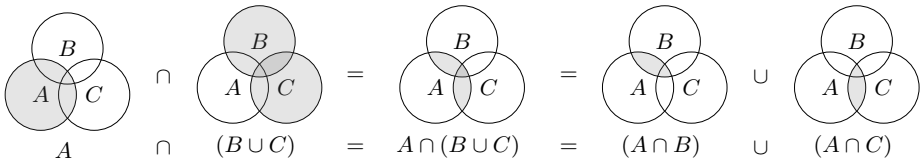
(c) It is easy to see that this summation is greater than the one from item (a). Thus, it is $\Omega(n^{(r+1)})$. Also, we have

$$\sum_{k=1}^{n} k^r \lg^s k \leq \sum_{k=1}^{n} n^r \lg^s n = O(n^{(r+1)} \lg^s n).$$

Thus, I guess it is $\Theta(n^{(r+1)} \lg^s n)$.

Section B.1 – Sets

B.1-1 Draw Venn diagrams that illustrate the first of the distributive laws (B.1).



B.1-2 Prove the generalization of DeMorgan's laws to any finite collection of sets:

$$\overline{A_1 \cap A_2 \cap \cdots \cap A_n} = \overline{A_1} \cup \overline{A_2} \cup \cdots \cup \overline{A_n},$$
$$\overline{A_1 \cup A_2 \cup \cdots \cup A_n} = \overline{A_1} \cap \overline{A_2} \cap \cdots \cap \overline{A_n}.$$

The base case, which occurs when $n = 2$, is given (from the text book). Now, lets assume it holds for n and show that it also holds for $n + 1$.
For the first DeMongan's law, we have

$$\begin{aligned}\overline{A_1 \cap A_2 \cap \cdots \cap A_n \cap A_{n+1}} &= \overline{(A_1 \cap A_2 \cap \cdots \cap A_n) \cap A_{n+1}} \\ &= \overline{(A_1 \cap A_2 \cap \cdots \cap A_n)} \cup \overline{A_{n+1}} \\ &= (\overline{A_1} \cup \overline{A_2} \cup \cdots \cup \overline{A_n}) \cup \overline{A_{n+1}} \\ &= \overline{A_1} \cup \overline{A_2} \cup \cdots \cup \overline{A_n} \cup \overline{A_{n+1}}.\end{aligned}$$

For the second DeMongan's law, we have

$$\begin{aligned}\overline{A_1 \cup A_2 \cup \cdots \cup A_n \cup A_{n+1}} &= \overline{(A_1 \cup A_2 \cup \cdots \cup A_n) \cup A_{n+1}} \\ &= \overline{(A_1 \cup A_2 \cup \cdots \cup A_n)} \cap \overline{A_{n+1}} \\ &= (\overline{A_1} \cap \overline{A_2} \cap \cdots \cap \overline{A_n}) \cap \overline{A_{n+1}} \\ &= \overline{A_1} \cap \overline{A_2} \cap \cdots \cap \overline{A_n} \cap \overline{A_{n+1}}.\end{aligned}$$

B.1-3 ($\star$) Prove the generalization of equation (B.3), which is called the ***principle of inclusion and exclusion***:

$$\begin{aligned}&|A_1 \cup A_2 \cup \cdots \cup A_n| = \\ &\quad |A_1| + |A_2| + \cdots + |A_n| \\ &\quad - |A_1 \cap A_2| - |A_1 \cap A_3| - \cdots \qquad \text{(all pairs)} \\ &\quad + |A_1 \cap A_2 \cap A_3| + \cdots \qquad \text{(all triples)} \\ &\qquad\qquad \vdots \\ &\quad + (-1)^{n-1} |A_1 \cap A_2 \cap \cdots \cap A_n|.\end{aligned}$$

Skipped.

B.1-4 Show that the set of odd natural numbers is countable.

Let $\mathbb{O}$ denote the set of odd natural numbers.
The function $f(n) = 2n + 1$ is a 1-1 correspondence from $\mathbb{N}$ to $\mathbb{O}$. Thus, $\mathbb{O}$ is countable.

B.1-5 Show that for any finite set S, the power set 2^S has $2^{|S|}$ elements (that is, there are $2^{|S|}$ distinct subsets of S).

For the base case, consider a set with a single element x. We have

$$2^{\{x\}} = \{\emptyset, \{x\}\},$$

which shows that the power set of a set with a single element has cardinality $2^1 = 2$.
Let $C(\cdot)$ denote the cardinality of a power set. Let S be a set of size n. Lets assume that the power set of S has cardinality $C(S) = 2^{|S|} = 2^n$. Now, let S' be the set S with one additional element x, such that $|S'| = n + 1$. The power set of S' will consist of all sets in the power set of S plus all those same sets again, with the element x added. Thus, we have

$$C(S') = 2 \cdot C(S) = 2 \cdot 2^n = 2^{n+1}.$$

B.1-6 Give an inductive definition for an n-tuple by extending the set-theoretic definition for an ordered pair.

$$\begin{aligned}
(a) &= \{a\} \\
(a, b) &= \{a, \{a, b\}\} \\
(a, b, c) &= \{a, \{a, b\}, \{a, b, c\}\} \\
(a_1, a_2, \ldots, a_n) &= (a_1, a_2, \ldots, a_{n-1}) \cup \{a_1, a_2, \ldots, a_n\}
\end{aligned}$$

Section B.2 – Relations

B.2-1 Prove that the subset relation "$\subseteq$" on all subsets of $\mathbb{Z}$ is a partial order but not a total order.

Let $\mathbb{S}$ denote all the subsets of $\mathbb{Z}$. Let $A = \{1\}$, $B = \{2\}$ be two subsets of $\mathbb{Z}$. We have $A \not\subseteq B$ and $B \not\subseteq A$. Thus, the subset relation "$\subseteq$" on $\mathbb{S} \times \mathbb{S}$ is not a total relation and therefore is not a total order.

For the relation $\subseteq$ on $\mathbb{S}$ be a partial order, the following properties needs to hold: (1) reflexivity, (2) antisymmetry, (3) transitivity. Since $A \subseteq A$, for all $A \in \mathbb{S}$, the relation "$\subseteq$" on $\mathbb{S} \times \mathbb{S}$ is reflexive. To be antisymmetric, we need to show that if $A \subseteq B$ and $B \subseteq A$, then $A = B$, for all $A, B \in \mathbb{S}$. Since $A \subseteq B$, for all $a \in A$ we have $a \in B$ and since $B \subseteq A$, for all $b \in B$ we have $b \in A$. Thus, $A = B$ and the relation "$\subseteq$" on $\mathbb{S} \times \mathbb{S}$ is antisymmetric. To be transitive, we need to show that if $A \subseteq B$ and $B \subseteq C$, then $A \subseteq C$, for all $A, B, C \in \mathbb{S}$. So let $a \in A$. Since $A \subseteq B$, we have $a \in B$. Since $a \in B$ and $B \subseteq C$, we have $a \in C$. Thus, $A \subseteq C$ and the relation "$\subseteq$" on $\mathbb{S} \times \mathbb{S}$ is transitive.

B.2-2 Show that for any positive integer n, the relation "equivalent modulo n" is an equivalence relation on the integers. (We say that $a \equiv b \pmod{n}$ if there exists an integer q such that $a - b = qn$.) Into what equivalence classes does this relation partition the integers?

To the relation "equivalent modulo n" be an equivalent relation on $\mathbb{Z} \times \mathbb{Z}$, the following needs to hold:

(a) $a \equiv a \pmod{n}$, for all $a, n \in \mathbb{Z}$ (reflexivity)

(b) $a \equiv b \pmod{n}$ implies $b \equiv a \pmod{n}$, for all $a, b, n \in \mathbb{Z}$ (symmetry)

(c) $a \equiv b \pmod{n}$ and $b \equiv c \pmod{n}$ implies $a \equiv c \pmod{n}$, for all $a, b, c, n \in \mathbb{Z}$ (transitivity)

For the reflexivity property, we have that $a - a = qn$ holds directly for $q = 0$.

For the symmetry property, we have that $a - b = pn$ implies $b - a = qn$ holds directly for $q = -p$.

For the transitivity property, we have that $a - b = pn$ and $b - c = qn$ implies $a - c = rn$ holds for $r = p + q$, since

$$(a - b) + (b - c) = pn + qn \to a - c = (p + q)n.$$

B.2-3 Give examples of relations that are

a. reflexive and symmetric but not transitive,

b. reflexive and transitive but not symmetric,

c. symmetric and transitive but not reflexive.

(a) The relation "is neighbor of" is reflexive (one is neighbor of himself) and symmetric (a "is neighbor of" b imply b "is neighbor of" a), but not transitive (a "is neighbor of" b and b "is neighbor of" c does not imply a "is neighbor of" c.

(b) The relation "$\leq$" is reflexive ($a \leq a$) and transitive ($a \leq b$ and $b \leq c$ imply $a \leq c$), but not symmetric ($a \leq b$ does not imply $b \leq a$).

(c) Consider the relation "$a + b > \infty$" on $\mathbb{Z} \times \mathbb{Z}$. This relation is empty. However, it is symmetric ($a \; R \; b$ imply $b \; R \; a$) and transitive ($a \; R \; b$ and $b \; R \; c$ imply $a \; R \; c$), but not reflexive since for no $a \in \mathbb{Z}$ is it the case that $a \; R \; a$.

B.2-4 Let S be a finite set, and let R be an equivalence relation on $S \times S$. Show that if in addition R is antisymmetric, then the equivalence classes of S with respect to R are singletons.

For every $a, b \in S$ such that $a \; R \; b$, by symmetry $b \; R \; a$, and by antisymmetry $a = a$. Thus, $[a] = \{b \in S : a \; R \; b\} = \{a\}$ for all $a \in S$, which implies that the equivalence classes are singletons.

B.2-5 Professor Narcissus claims that if a relation R is symmetric and transitive, then it is also reflexive. He offers the following proof. By symmetry, $a \; R \; b$ implies $b \; R \; a$. Transitivity, therefore, implies $a \; R \; a$. Is the professor correct?

No. This is only true for relations that for every a there is b such that $a \; R \; b$, by symmetry $b \; R \; a$, and by transitivity $a \; R \; a$. For instance, an empty relation (like the one from Question B.2-3, item (c)) are symmetric and transitive, but not reflexive.

Section B.3 – Functions

B.3-1 Let A and B be finite sets, and let $f : A \to B$ be a function. Show that

a. if f is injective, then $|A| \leq |B|$;

b. if f is surjective, then $|A| \geq |B|$.

(a) Since f is injective, we have that $A = f(A)$. Also, we have

$$\begin{cases} |B| = |f(A)|, & f \text{ is surjective,} \\ |B| > |f(A)|, & f \text{ is not surjective.} \end{cases}$$

Thus, $|B| \geq |f(A)| = |A| \to |A| \leq |B|$.

(b) Since f is surjective, we have $|f(A)| = |B|$. Also, we have

$$\begin{cases} |A| = |f(A)|, & f \text{ is injective,} \\ |A| > |f(A)|, & f \text{ is not injective.} \end{cases}$$

Thus, $|A| \geq |f(A)| = |B| \to |A| \geq |B|$.

B.3-2 Is the function $f(x) = x + 1$ bijective when the domain and the codomain are $\mathbb{N}$? Is it bijective when the domain and the codomain are $\mathbb{Z}$?

On the set of naturals, f is injective but not surjective, since there is no $a \in \mathbb{N}$ such that $0 = f(a)$, which makes $f(\mathbb{N}) \neq \mathbb{N}$. On the set of integers, f is both injective and surjective, and therefore bijective.

B.3-3 Give a natural definition for the inverse of a binary relation such that if a relation is in fact a bijective function, its relational inverse is its functional inverse.

Let R be a binary relation on the sets A and B, such that $R \subseteq A \times B$. The general definition of the inverse of R is given by

$$R^{-1} = \{(b, a) \in B \times A : (a, b) \in R\}.$$

When R is a bijective function, we have: (1) for all $b \in B$, there is at most one $a \in A$ such that $a \; R \; b$ (injective) and (2) for all $b \in B$ there is at least one $a \in A$ such that $a \; R \; b$ (surjective). Therefore, when R is bijective, each element of A is related to exactly one element of B and vice-versa, which implies

$$f(a) = b \iff f'(b) = a,$$

for all $a \in A$ and for all $b \in B$.

B.3-4 ($\star$) Give a bijection from $\mathbb{Z}$ to $\mathbb{Z} \times \mathbb{Z}$.

Skipped.

Section C.1 – Counting

C.1-1 How many k-substrings does an n-string have? (Consider identical k-substrings at different positions to be different.) How many substrings does an n-string have in total?

For every position i of the n-string, $i = 1, \ldots, n-k+1$, there is one k-substring the starts at i and ends at $i+k-1$. Thus, the number of k-substrings in a n-string is

$$\sum_{i=1}^{n-k+1} 1 = n-k+1.$$

Thus, the number of substrings (of all sizes) in an n-string is

$$\begin{aligned}\sum_{k=1}^{n} n-k+1 &= n^2 + n - \sum_{k=1}^{n} k \\ &= n^2 + n - \frac{n(n+1)}{2} \\ &= n(n+1) - \frac{n(n+1)}{2} \\ &= \frac{n(n+1)}{2}.\end{aligned}$$

C.1-2 An n-input, m-output ***boolean function*** is a function from $\{\texttt{TRUE}, \texttt{FALSE}\}^n$ to $\{\texttt{TRUE}, \texttt{FALSE}\}^m$. How many n-input, 1-output boolean functions are there? How many n-input, m-output boolean functions are there?

We can view the number of possible inputs of size n as the number of binary n-strings, which is 2^n.
Now, consider a single-valued function from $\{\texttt{TRUE}, \texttt{FALSE}\}^n$ to $\{\texttt{TRUE}\}$. In this case, the number of possible functions is the number of possible inputs, which is 2^n. Since an 1-output boolean function has two possible output values, each of the 2^n functions we referred in the case of a single-valued function now has two ways to pick the output value. We can view this number as the number of binary 2^n-strings, which is 2^{2^n}. As for an n-output function, each of the 2^n functions we referred in the case of a single-valued function now has 2^m ways to pick the output value. Thus, there are $(2^m)^{2^n}$ of those.

C.1-3 In how many ways can n professors sit around a circular conference table? Consider two seatings to be the same if one can be rotated to form the other.

For two seatings to be different from each other, the ordering of professors in each seating needs to be different. This number can be viewed as the number of permutations of a set n elements, which is $n!$. However, note that for each permutation that starts with professor k, $1 \leq k \leq n$, there are $n-1$ other permutations that are just a rotation of it. For instance, the seatings $\{2,3,1\}$ and $\{3,1,2\}$ are a rotation of $\{1,2,3\}$. Thus, the number of different seatings can be viewed as fixing the seat of the first professor and computing the number of permutations of the remaining $n-1$ professors, which is $(n-1)!$.

C.1-4 In how many ways can we choose three distinct numbers from the set $\{1, 2, \ldots, 99\}$ so that their sum is even?

The set has 50 odd numbers and 49 even numbers. For the sum be even, we have to choose three even numbers or one even and two odds. For the case with three even numbers, there are $49!/(3! \cdot (49-3)!) = 18424$ ways of choosing 3 distincts numbers among the 49 even numbers. As for the case with one even and two odds, there are 49 ways to choose one even number and $50!/(2! \cdot (50-2)!) = 1225$ ways of choosing 2 distincts numbers among the 50 odd numbers. Thus, there are $18424 + 49 \cdot 1225 = 78449$ ways to get an even sum.

C.1-5 Prove the identity

$$\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1}$$

for $0 < k \leq n$.

$$\begin{aligned}\binom{n}{k} &= \frac{n!}{k! \cdot (n-k)!} \\ &= \frac{n \cdot (n-1)!}{k \cdot (k-1)! \cdot (n-k)!} \\ &= \frac{n}{k}\frac{(n-1)!}{(k-1)! \cdot ((n-1)-(k-1))!} \\ &= \frac{n}{k}\binom{n-1}{k-1}.\end{aligned}$$

C.1-6 Prove the identity

$$\binom{n}{k} = \frac{n}{n-k}\binom{n-1}{k}$$

for $0 \leq k < n$.

$$\begin{aligned}\binom{n}{k} &= \frac{n!}{k! \cdot (n-k)!} \\ &= \frac{n \cdot (n-1)!}{k! \cdot (n-k) \cdot (n-k-1)!} \\ &= \frac{n}{n-k}\frac{(n-1)!}{k! \cdot ((n-1)-k)!} \\ &= \frac{n}{n-k}\binom{n-1}{k}.\end{aligned}$$

C.1-7 To choose k objects from n, you can make one of the objects distinguished and consider whether the distinguished object is chosen. Use this approach to prove that

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}.$$

Let $S = \{s_1, s_2, \ldots, s_{n-1}\}$ and s_0 the distinguished element. To choose k from the n elements, we have to consider two cases:

(a) If s_0 is selected, it will be necessary to choose the $k-1$ remaining elements from S. There are $\binom{n-1}{k-1}$ combinations.

(b) If s_0 is not selected, it will be necessary to choose the k remaining elements from S. There are $\binom{n-1}{k}$ combinations.

Adding the above together, we have

$$\begin{aligned}\binom{n-1}{k-1} + \binom{n-1}{k} &= \frac{(n-1)!}{(k-1)! \cdot (n-k)!} + \frac{(n-1)!}{k! \cdot (n-k-1)!} \\ &= \frac{k \cdot (n-1)!}{k! \cdot (n-k)!} + \frac{(n-k) \cdot (n-1)!}{k! \cdot (n-k)!} \\ &= \frac{(k+n-k) \cdot (n-1)!}{k! \cdot (n-k)!} \\ &= \frac{n!}{k! \cdot (n-k)!} \\ &= \binom{n}{k}.\end{aligned}$$

C.1-8 Using the result of Exercise C.1-7, make a table for $n = 0, 1, \ldots, 6$ and $0 \leq k \leq n$ of the binomial coefficients $\binom{n}{k}$ with $\binom{0}{0}$ at the top, $\binom{1}{0}$ and $\binom{1}{1}$ on the next line, and so forth. Such a table of binomial coefficients is called ***Pascal's triangle***.

The table with binomials

$$\begin{array}{ccccccccccccc}
 & & & & & & \binom{0}{0} & & & & & & \\
 & & & & & \binom{1}{0} & & \binom{1}{1} & & & & & \\
 & & & & \binom{2}{0} & & \binom{2}{1} & & \binom{2}{2} & & & & \\
 & & & \binom{3}{0} & & \binom{3}{1} & & \binom{3}{2} & & \binom{3}{3} & & & \\
 & & \binom{4}{0} & & \binom{4}{1} & & \binom{4}{2} & & \binom{4}{3} & & \binom{4}{4} & & \\
 & \binom{5}{0} & & \binom{5}{1} & & \binom{5}{2} & & \binom{5}{3} & & \binom{5}{4} & & \binom{5}{5} & \\
\binom{6}{0} & & \binom{6}{1} & & \binom{6}{2} & & \binom{6}{3} & & \binom{6}{4} & & \binom{6}{5} & & \binom{6}{6}
\end{array}$$

Using the above table and the result of C.1-7, we have the Pascal's triangle

$$\begin{array}{ccccccccccccc}
 & & & & & & 1 & & & & & & \\
 & & & & & 1 & & 1 & & & & & \\
 & & & & 1 & & 2 & & 1 & & & & \\
 & & & 1 & & 3 & & 3 & & 1 & & & \\
 & & 1 & & 4 & & 6 & & 4 & & 1 & & \\
 & 1 & & 5 & & 10 & & 10 & & 5 & & 1 & \\
1 & & 6 & & 15 & & 20 & & 15 & & 6 & & 1
\end{array}$$

C.1-9 Prove that

$$\sum_{i=1}^{n} i = \binom{n+1}{2}.$$

We have

$$\begin{aligned}
\binom{n+1}{2} &= \frac{(n+1)!}{2! \cdot ((n+1)-2)!} \\
&= \frac{(n+1) \cdot n \cdot (n-1)!}{2 \cdot (n-1)!} \\
&= \frac{n(n+1)}{2} \\
&= \sum_{i=1}^{n} i,
\end{aligned}$$

which also shows that the third Pascal's diagonal has the triangular numbers.

C.1-10 Show that for any integers $n \geq 0$ and $0 \leq k \leq n$, the expression $\binom{n}{k}$ achieves its maximum value when $k = \lfloor n/2 \rfloor$ or $k = \lceil n/2 \rceil$.

It follows from the Pascal's triangle

$$
\begin{array}{ccccccccccc}
 & & & & & 1 & & & & & \\
 & & & & 1 & & 1 & & & & \\
 & & & 1 & & 2 & & 1 & & & \\
 & & 1 & & 3 & & 3 & & 1 & & \\
 & 1 & & 4 & & 6 & & 4 & & 1 & \\
1 & & 5 & & 10 & & 10 & & 5 & & 1 \\
1 & & 6 & & 15 & & 20 & & 15 & & 6 \quad 1 \\
 & & & & & \vdots & & & & &
\end{array}
$$

We can prove by induction. The base case, which occurs when $n = 0$, holds since

$$\binom{n}{\lfloor n/2 \rfloor} = \binom{n}{\lceil n/2 \rceil} = \binom{0}{0} = 1$$

is maximum on row 0. Now, assume it holds for n. Then, if $n+1$ is even, from Equation (C.3) we have

$$
\begin{aligned}
\binom{n+1}{\lfloor \frac{n+1}{2} \rfloor} = \binom{n+1}{\lceil \frac{n+1}{2} \rceil} &= \binom{n}{(\frac{n+1}{2} - 1)} + \binom{n}{(\frac{n+1}{2})} \\
&= \binom{n}{(\frac{n}{2} - \frac{1}{2})} + \binom{n}{(\frac{n}{2} + \frac{1}{2})} \quad \text{(since } n \text{ is odd)} \\
&= \binom{n}{\lfloor \frac{n}{2} \rfloor} + \binom{n}{\lceil \frac{n}{2} \rceil},
\end{aligned}
$$

which shows that is also holds for $n+1$ since

$$\binom{n}{\lfloor \frac{n}{2} \rfloor} \text{ and } \binom{n}{\lceil \frac{n}{2} \rceil}$$

are both maximum on row n. The proof is similar when $n+1$ is odd.

C.1-11 ($\star$) Argue that for any integers $n \geq 0$, $j \geq 0$, $k \geq 0$, and $j + k \leq n$,

$$\binom{n}{j+k} \leq \binom{n}{j}\binom{n-j}{k}.$$

Provide both an algebraic proof and an argument based on a method for choosing $j+k$ items out of n. Give an example in which equality does not hold.

For any integers $a \geq 0, b \geq 0$, and $a \geq b$, we have

$$\begin{aligned}(a+b)! &= \underbrace{(a+b)\cdot(a+b-1)\cdot(a+b-2)\cdots}_{b \text{ times}} a! \\ &\geq \underbrace{b\cdot(b-1)\cdot(b-2)\cdots}_{b \text{ times}} a! \\ &= a!\cdot b!.\end{aligned}$$

Using the above result, we have

$$\begin{aligned}\binom{n}{j}\binom{n-j}{k} &= \frac{n!}{j!\cdot(n-j)!}\frac{(n-j)!}{k!\cdot((n-j)-k)!} \\ &= \frac{n!}{j!\cdot k!\cdot((n-j)-k)!} \\ &\geq \frac{n!}{(j+k)!\cdot(n-(j+k))!} \\ &= \binom{n}{j+k}.\end{aligned}$$

The expression on the left is the number of ways to choose an $(j+k)$-subset of an n-set (which leaves the reamining $n-(j+k)$ elements). Thus, it is a partition of the original n-set into subsets of cardinalities $(j+k)$ and $n-(j+k)$. The right hand side has two factors: the first binomial coefficient is the number of ways to choose a j-subset of an n-set (which leaves the reamining $n-j$ elements); the second is the number of ways to choose a k-subset from the remaining $n-j$ elements. Thus, it is a partition of the original n-set into subsets of cardinalities j, k, and $n-(j+k)$. Consider now that we choose the $n-(j+k)$ first, leaving behind the remaining $j+k$ elements. There is precisely one way to choose an $(j+k)$-subset out of the remaining $j+k$ elements. On the other hand, when we first choose j and then we choose k, if $j < j+k$, there are *at least* two ways to choose a j-subset from the $(j+k)$-subset and precisely one way to choose a k-subset from the remaining k elements. This notion also applies to the algebraic proof, since $(j+k)! = j!\cdot k! \iff j = 0$ or $k = 0$. Also note that while the left expression does not count any permutation of the $(j+k)$-subsets (since it normalizes by $(j+k)!$), the right expression, despite not counting permutations of each of the subsets indepentently (since it normalizes by $j!\cdot k!$), it counts permutations of two subsets together. For instance, let $A = \{a, b\}$. There is only one way to choose 2 elements from A, which is ab. However, there are two ways to choose one element and then another element from A, which are ab and ba.

C.1-12 ($\star$) Use induction on all integers k such that $0 \leq k \leq n/2$ to prove inequality (C.6), and use equation (C.3) to extend it to all integers k such that $0 \leq k \leq n$.

Skipped.

C.1-13 ($\star$) Use Stirling's approximation to prove that

$$\binom{2n}{n} = \frac{2^{2n}}{\sqrt{\pi n}}(1 + O(1/n)).$$

Skipped.

C.1-14 ($\star$) By differentiating the entropy function $H(\lambda)$, show that it achieves its maximum value at $\lambda = 1/2$. What is $H(1/2)$?

Skipped.

C.1-15 ($\star$) Show that for any integer $n \geq 0$,

$$\sum_{k=0}^{n} \binom{n}{k} k = n2^{n-1}.$$

Skipped.

Section C.2 – Probability

C.2-1 Professor Rosencrantz flips a fair coin once. Professor Guildenstern flips a fair coin twice. What is the probability that Professor Rosencrantz obtains more heads than Professor Guildenstern?

The sample space $\{\mathrm{H},\mathrm{T}\}^3$ has size $2^3 = 8$. Since the only event that satisfies the condition is $\{\mathrm{HTT}\}$, the probability is $1/8$.

C.2-2 Prove the ***Boole's inequality***: For any finite or countably infinite sequence of events $A_1, A_2, \ldots$,

$$\Pr\{A_1 \cup A_2 \cup \cdots\} \le \Pr\{A_1\} + \Pr\{A_2\} + \cdots .$$

From (C.13) we have

$$\Pr\{A_1 \cup A_2\} \le \Pr\{A_1\} + \Pr\{A_2\},$$

which implies

$$\begin{aligned}\Pr\{A_1 \cup A_2 \cup \cdots\} &= \Pr\{A_1 \cup (A_2 \cup \cdots)\} \\ &\le \Pr\{A_1\} + \Pr\{A_2 \cup (A_3 \cup \cdots)\} \\ &\le \Pr\{A_1\} + \Pr\{A_2\} + \Pr\{A_3 \cup (A_4 \cup \cdots)\} \\ &\le \Pr\{A_1\} + \Pr\{A_2\} + \Pr\{A_3\} \cdots .\end{aligned}$$

C.2-3 Suppose we shuffle a deck of 10 cards, each bearing a distinct number from 1 to 10, to mix the cards thoroughly. We then remove three cards, one at a time, from the deck. What is the probability that we select the three cards in sorted (increasing) order?

Let $a < b < c$ denote the number of the three selected cards. There are 3! permutations of $\{a, b, c\}$ and abc is the only one which is in sorted order. Thus, the probability is $1/3! = 1/6$.

C.2-4 Prove that

$$\Pr\{A \mid B\} + \Pr\{\overline{A} \mid B\} = 1.$$

We have

$$\begin{aligned}\Pr\{B\} &= \Pr\{(B \cap A) \cup (B \cap \overline{A})\} \\ &= \Pr\{B \cap A\} + \Pr\{B \cap \overline{A}\} \\ &= \Pr\{A\}\Pr\{B \mid A\} + \Pr\{\overline{A}\}\Pr\{B \mid \overline{A}\}.\end{aligned}$$

Substituting into (C.17) yields

$$\begin{aligned}\Pr\{A \mid B\} + \Pr\{\overline{A} \mid B\} &= \frac{\Pr\{A\}\Pr\{B \mid A\}}{\Pr\{B\}} + \frac{\Pr\{\overline{A}\}\Pr\{B \mid \overline{A}\}}{\Pr\{B\}} \\ &= \frac{\Pr\{A\}\Pr\{B \mid A\} + \Pr\{\overline{A}\}\Pr\{B \mid \overline{A}\}}{\Pr\{B\}} \\ &= \frac{\Pr\{A\}\Pr\{B \mid A\} + \Pr\{\overline{A}\}\Pr\{B \mid \overline{A}\}}{\Pr\{A\}\Pr\{B \mid A\} + \Pr\{\overline{A}\}\Pr\{B \mid \overline{A}\}} \\ &= 1.\end{aligned}$$

C.2-5 Prove that for any collection of events $A_1, A_2, \ldots, A_n$,

$$\Pr\{A_1 \cap A_2 \cap \cdots \cap A_n\} = \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \cdots \Pr\{A_n \mid A_1 \cap A_2 \cap \cdots \cap A_{n-1}\}.$$

It is trivially valid for $n = 1$. As our base case, consider $n = 2$. From (C.16) we have

$$\Pr\{A_1 \cap A_2\} = \Pr\{A_1\}\Pr\{A_2 \mid A_1\}.$$

Now assume it holds for n. For $n+1$, we have

$$\begin{aligned}\Pr\{A_1 \cap A_2 \cap \cdots \cap A_{n+1}\} &= \Pr\{(A_1 \cap A_2 \cap \cdots \cap A_n) \cap A_{n+1}\} \\ &= \Pr\{A_1 \cap A_2 \cap \cdots \cap A_n\}\Pr\{A_{n+1} \mid A_1 \cap A_2 \cap \cdots \cap A_n\} \\ &= \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \cdots \Pr\{A_{n+1} \mid A_1 \cap A_2 \cap \cdots \cap A_n\}.\end{aligned}$$

C.2-6 ($\star$) Describe a procedure that takes as input two integers a and b such that $0 < a < b$ and, using fair coin flips, produces as output heads with probability a/b and tails with probability $(b-a)/b$. Give a bound on the expected number of coin flips, which should be $O(1)$. (Hint: Represent a/b in binary.)

Consider a continuous uniform probability distribution on $[0, 1)$, such that $\Pr\{[0, 1)\} = 1$. We have

$$\Pr\left\{\left[0, \frac{a}{b}\right)\right\} = \frac{a}{b},$$

and

$$\Pr\left\{\left[\frac{a}{b}, 1\right)\right\} = 1 - \frac{a}{b} = \frac{b-a}{b}.$$

With this notion, we can write a procedure that sorts a real number from $[0, 1)$ and return heads if it is lower than a/b or return tails, otherwise. Using fair coin flips and representing numbers in binary, for each flip we have a new decimal place from a random number on $[0, 1)$ (consider an "0" if the coin flip is head and "1", otherwise). Then,

- if the i-th flip is 1 and the i-th decimal place of a/b is 0, the sorted number is larger than a/b and we return tails;
- if the i-th flip is 0 and the i-th decimal place of a/b is 1, the sorted number is smaller than a/b and we return head;
- if the i-th flip and the i-th decimal place are equal, we sort a new decimal place.

Since we do not know how many decimal places a/b has (if periodic, this number is infinite), the above procedure does not have a maximum number of iterations. However, since for each flip we have a probability of $1/2$ of returning head or tails, the probability of terminating at flip i, for $i \geq 1$, is

$$\underbrace{1/2 \cdot 1/2 \cdots}_{i \text{ times}} = \frac{1}{2^i}.$$

Thus, by using the notion of expected value and the result (A.8), the expected number of flips is

$$\sum_{i=1}^{\infty} i \cdot \frac{1}{2^i} = \sum_{i=0}^{\infty} i \cdot \left(\frac{1}{2}\right)^i = \frac{1/2}{(1-1/2)^2} = 2.$$

C.2-7 ($\star$) Show how to construct a set of n events that are paiwise independent but such that no subset of $k > 2$ of them is mutually independent.

Skipped.

C.2-8 ($\star$) Two events A and B are ***conditionally independent***, given C, if

$$\Pr\{A \cap B \mid C\} = \Pr\{A \mid C\} \cdot \Pr\{B \mid C\}.$$

Give a simple but nontrivial example of two events that are not independent but are conditionally independent given a third event.

Skipped.

C.2-9 ($\star$) You are a contestant in a game show in which a prize is hidden behind one of three curtains. You will win the prize if you select the correct curtain. After you have picked one curtain but before the curtain is lifted, the emcee lifts one of the other curtains, knowing that it will reveal an empty stage, and asks if you would like to switch from your current selection to the remaining curtain. How would your chances change if you switch? (This question is the celebrated ***Monty Hall problem***, named after a game-show host who often presented contestants with just this dilemma.)

If you never switch, the only way to win is to choose the right curtain at the beginning (before the emcee lifts one of the others). In this case, your chance to win are $1/3$. If you always switch, the only way to loose is to choose the right curtain at the beginning. In this case, when you choose a curtain without the prize, the emcee will reveal the other empty curtain and you will therefore change to the correct one. Thus, your chance to win are $(1 - 1/3) = 2/3$.

C.2-10 ($\star$) A prison warden has randomly picked one prisoner among three to go free. The other two will be executed. The guard knows which one will go free but is forbidden to give any prisoner information regarding his status. Let us call the prisoners X, Y, and Z. Prisoner X asks the guard privately which of Y or Z will be executed, arguing that since he already knows that at least one of them must die, the guard won't be revealing any information about his own status. The guard tells X that Y is to be executed. Prisoner X feels happier now, since he figures that either he or prisoner Z will go free, which means that his probability of going free is now $1/2$. Is he right, or are his chances still $1/3$? Explain.

His chances are still $1/3$. Let A be the event of prisoner X going free and B the event that the guard tells X that Y is to be executed. We have

$$\Pr(A \mid B) = \frac{\Pr(A)\Pr(B \mid A)}{\Pr(B)} = \frac{1/3 \cdot 1/2}{1/2} = \frac{1}{3}.$$

Section C.3 – Discrete random variables

C.3-1 Suppose we roll two ordinary, 6-sided dice. What is the expectation of the sum of the two values showing? What is the expectation of the maximum of the two values showing?

There are 36 elementary events in the sample space. Since they are ordinary dices, the probability distribution is uniform. Let X be the random variable of the sum of the two values. The possible outcomes of X are

	1	**2**	**3**	**4**	**5**	**6**
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

Thus, we have

$$\begin{aligned}\mathrm{E}[X] &= \sum_{x=2}^{12} x \cdot \Pr(X = x) \\ &= 2 \cdot \frac{1}{36} + 3 \cdot \frac{2}{36} + 4 \cdot \frac{3}{36} + 5 \cdot \frac{4}{36} + 6 \cdot \frac{5}{36} + 7 \cdot \frac{6}{36} + 8 \cdot \frac{5}{36} + 9 \cdot \frac{4}{36} + 10 \cdot \frac{3}{36} + 11 \cdot \frac{2}{36} + 12 \cdot \frac{1}{36} \\ &= 7.\end{aligned}$$

Let Y be the random variable of the maximum of the two values. The possible outcomes of Y are

	1	**2**	**3**	**4**	**5**	**6**
1	1	2	3	4	5	6
2	2	2	3	4	5	6
3	3	3	3	4	5	6
4	4	4	4	4	5	6
5	5	5	5	5	5	6
6	6	6	6	6	6	6

Thus, we have

$$\begin{aligned}\mathrm{E}[Y] &= \sum_{x=1}^{6} x \cdot \Pr(X = x) \\ &= 1 \cdot \frac{1}{36} + 2 \cdot \frac{3}{36} + 3 \cdot \frac{5}{36} + 4 \cdot \frac{7}{36} + 5 \cdot \frac{9}{36} + 6 \cdot \frac{11}{36} \\ &\approx 4.47.\end{aligned}$$

C.3-2 An array $A[1 \ldots n]$ contains n distinct numbers that are randomly ordered, with each permutation of the n numbers being equally likely. what is the expectation of the index of the maximum element in the array? What is the expectation of the index of the minimum element in the array?

Let X and Y be random variables of the index of the maximum and minimum elements, respectivelly. Since each permutation is equaly likely,

$$\mathrm{E}[X] = \mathrm{E}[Y] = \sum_{i=1}^{n} i \cdot \frac{1}{n} = \frac{1}{n} \frac{n(n+1)}{2} = \frac{n+1}{2}.$$

C.3-3 A carnival game consists of three dice in a cage. A player can bet a dollar on any of the numbers 1 through 6. The cage is shaken, and the payoff is as follows. If the player's number doesn't appear on any of the dice, he loses his dollar. Otherwise, if his number appears on exactly k of the three dice, for $k = 1, 2, 3$, he keeps his dollar and wins k more dollars. What is his expected gain from playing the carnival game once?

Let X be a random variable of the total gain. The possible outcomes are $-1, 1, 2, 3$. We have

$$\begin{aligned}
\Pr\{X = -1\} &= (5/6 \cdot 5/6 \cdot 5/6) &&= 125/216,\\
\Pr\{X = 1\} &= (1/6 \cdot 5/6 \cdot 5/6) \cdot 3 &&= 75/216,\\
\Pr\{X = 2\} &= (1/6 \cdot 1/6 \cdot 5/6) \cdot 3 &&= 15/216,\\
\Pr\{X = 3\} &= (1/6 \cdot 1/6 \cdot 1/6) &&= 1/216.
\end{aligned}$$

Thus, we have

$$\mathrm{E}[X] = -1 \cdot \frac{125}{216} + 1 \cdot \frac{75}{216} + 2 \cdot \frac{15}{216} + 3 \cdot \frac{1}{216} \approx -0.0787.$$

C.3-4 Argue that if X and Y are nonnegative random variables, then

$$\mathrm{E}[\max(X, Y)] \leq \mathrm{E}[X] + \mathrm{E}[Y].$$

The expectation of nonnegative random variables is a summation of nonnegative numbers. Thus, since $\mathrm{E}[\max(X, Y)]$ is either $\mathrm{E}[X]$ or $\mathrm{E}[Y]$, it must be equal or lower than $\mathrm{E}[X] + \mathrm{E}[Y]$.

C.3-5 ($\star$) Let X and Y be independent random variables. Prove that $f(X)$ and $g(Y)$ are independent for any functions f and g.

Skipped.

C.3-6 ($\star$) Let X be a nonnegative random variable, and suppose that $\mathrm{E}[X]$ is well defined. Prove ***Markov's inequality***:

$$\Pr\{X \geq t\} \leq \mathrm{E}[X]/t$$

for all $t > 0$.

We have

$$\begin{aligned}
\mathrm{E}[X] &= \sum_x x \cdot \Pr\{X = x\}\\
&\geq \sum_{x \geq t} x \cdot \Pr\{X = x\}\\
&\geq \sum_{x \geq t} t \cdot \Pr\{X = x\}\\
&= t \cdot \sum_{x \geq t} \Pr\{X = x\}\\
&= t \cdot \Pr\{X \geq t\},
\end{aligned}$$

which implies

$$\Pr\{X \geq t\} \leq \mathrm{E}[X]/t.$$

C.3-7 ($\star$) Let S be a sample space, and let X and X' be random variables such that $X(s) \geq X'(s)$ for all $s \in S$. Prove that for any real constant t,

$$\Pr\{X \geq t\} \geq \Pr\{X' \geq t\}.$$

Assuming that the domain of X and X' are the sample space S, we have

$$\begin{aligned}
\Pr\{X \geq t\} &= \sum_{s \in S : X(s) \geq t} \Pr\{X = s\} \\
&= \sum_{s \in S : X'(s) \geq t} \Pr\{X' = s\} + \sum_{s \in S : X(s) \geq t > X'(s)} \Pr\{X' = s\} \\
&\geq \sum_{s \in S : X'(s) \geq t} \Pr\{X' = s\} \\
&= \Pr\{X' \geq t\}.
\end{aligned}$$

C.3-8 Which is larger: the expectation of the square of a random variable, or the square of its expectation?

We have from (C.28)

$$\mathrm{E}[X^2] = \mathrm{Var}[X] + \mathrm{E}^2[X],$$

which implies

$$\mathrm{E}[X^2] \geq \mathrm{E}^2[X],$$

since both $\mathrm{Var}[X]$ and $\mathrm{E}^2[X]$ are nonnegative numbers.

C.3-9 Show that for any random variable X that takes on only the values 0 and 1, we have

$$\mathrm{Var}[X] = \mathrm{E}[X]\mathrm{E}[1 - X].$$

We have

$$\mathrm{E}[X] = 0 \cdot \Pr\{X = 0\} + 1 \cdot \Pr\{X = 1\} = \Pr\{X = 1\},$$

and

$$\mathrm{E}[1 - X] = 1 \cdot \Pr\{X = 0\} + 0 \cdot \Pr\{X = 1\} = \Pr\{X = 0\},$$

which implies

$$\begin{aligned}
\mathrm{Var}[X] &= \mathrm{E}[X^2] - \mathrm{E}^2[X] && \text{(since } X^2 = X\text{)} \\
&= \mathrm{E}[X] - \mathrm{E}[X]\mathrm{E}[X] \\
&= \mathrm{E}[X](1 - \mathrm{E}[X]) \\
&= \mathrm{E}[X](1 - \Pr\{X = 1\}) \\
&= \mathrm{E}[X]\Pr\{X = 0\} \\
&= \mathrm{E}[X]\mathrm{E}[1 - X].
\end{aligned}$$

C.3-10 Prove that $\mathrm{Var}[aX] = a^2\mathrm{Var}[X]$ from the definition (C.27) of variance.

Assuming that X is a random variable and a is a constant, from (C.27) and (C.22) we have

$$\begin{aligned}
\mathrm{Var}[aX] &= \mathrm{E}[(aX - \mathrm{E}[aX])^2] \\
&= \mathrm{E}[(aX - a\mathrm{E}[X])^2] \\
&= \mathrm{E}[a^2(X - \mathrm{E}[X])^2] \\
&= a^2\mathrm{E}[(X - \mathrm{E}[X])^2] \\
&= a^2\mathrm{Var}[X].
\end{aligned}$$