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)
for j = 2 to A.length do
    key = A[j]
    i = j − 1
    while i > 0 and A[i] > key do
        A[i + 1] = A[i]
        i = i − 1
    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)
    let C[1, ..., n + 1] be a new array
    C[1] = 0
    for i = 1 to A.length do
        s = A[i] + B[i] + C[i]
        C[i] = s mod 2
        C[i + 1] = s / 2
    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)
    n1 = q - p + 1
    n2 = r - q
    let L[1, ..., n1] and R[1, ..., n2] be new arrays
    for i = 1 to n1 do
        L[i] = A[p + i - 1]
    for j = 1 to n2 do
        R[j] = A[q + j]
    i = 1
    j = 1
    for k = p to r do
        if q + j > r or L[i] ≤ R[j] then
            A[k] = L[i]
            i = i + 1
        else
            A[k] = R[j]
            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)
- **if** $s > e$ **then**
  - **return** NIL
- $m = \lfloor (s+e)/2 \rfloor$
- **if** $\nu > A[m]$ **then**
  - `BinarySearch`$(A, m+1, e, \nu)$
- **else if** $\nu > A[m]$ **then**
  - `BinarySearch`$(A, s, m-1, \nu)$
- **else**
  - **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

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

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)
    inv = 0
    n_1 = q − p + 1
    n_2 = r − q
    let L[1, . . . , n_1 + 1] and R[1, . . . , n_2 + 1] be new arrays
    for i = 1 to n_1 do
        L[i] = A[p + i − 1]
    for j = 1 to n_2 do
        R[j] = A[q + j]
    L[n_1 + 1] = ∞
    L[n_2 + 1] = ∞
    i = 1
    j = 1
    for k = p to r do
        if L[i] ≤ R[j] then
            i = i + 1
        else
            inv = inv + (n_1 − i + 1)
            j = j + 1
    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

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

and

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

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

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

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

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

Thus,

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

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 $0 \le f(n, m) \le cg(n, m)$ for all $n \ge n_0$ and $m \ge m_0\}$.

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

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

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

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

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

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 $n \le m$, we have

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

which implies that

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

Adding the above inequalities together, we have

$$f(n) - f(m) + g(n) - g(m) \le 0 \to f(n) + g(n) \le f(m) + g(m),$$

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

$$f(p) \le f(q) \to f(g(n)) \le f(g(m)),$$

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

$$f(n) \le f(m) \to f(n)g(n) \le f(m)g(n) \to f(n)g(n) \le f(m)g(m),$$

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

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

which implies

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

We have

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

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.

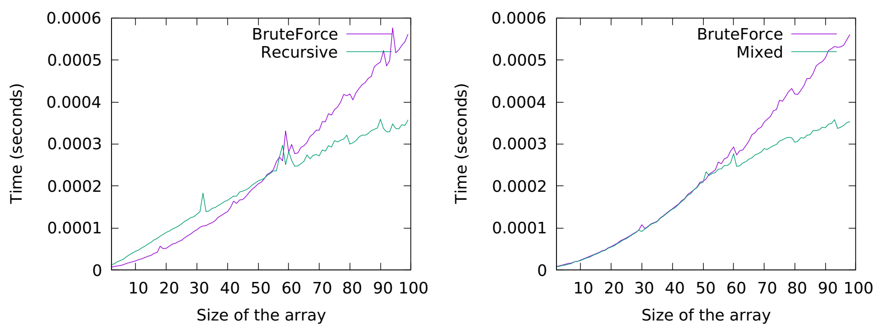
4.1-2 Write pseudocode for the brute-force method of solving the maximum-subarray problem. Your procedure should run in $\Theta(n^2)$ time.

> 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 $left\text{-}sum$ and $right\text{-}sum$ to 0, instead of $-\infty$. Also initialize $max\text{-}left$ (after line 1) and $max\text{-}right$ (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 $A[1, \dots, j]$, extend the answer to find a maximum subarray ending at index $j+1$ by using the following observation: a maximum subarray of $A[1, \dots, j+1]$ is either a maximum subarray of $A[1, \dots, j]$ or a subarray $A[i, \dots, j+1]$, for some $1 \leq i \leq j+1$. Determine a maximum subarray of the form $A[i, \dots, j+1]$ 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)
low = 0
high = 0
sum = 0
current-low = 0
current-sum = 0
for i = 1 to A.length do
    if current-sum + A[i] ≤ 0 then
        current-sum = 0
    else
        current-sum = current-sum + A[i]
        if current-sum == A[i] then
            current-low = i
        if current-sum > sum then
            low = current-low
            high = i
            sum = current-sum
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:

$$\begin{aligned}
P_1 &= A_1 1 \cdot S_1 &&= 1 \cdot 6 &&= 6, \\
P_2 &= S_2 \cdot B_2 2 &&= 4 \cdot 2 &&= 8, \\
P_3 &= S_3 \cdot B_1 1 &&= 12 \cdot 6 &&= 72, \\
P_4 &= A_2 2 \cdot S_4 &&= 5 \cdot (-2) &&= -10, \\
P_5 &= S_5 \cdot S_6 &&= 6 \cdot 8 &&= 48, \\
P_6 &= S_7 \cdot S_8 &&= (-2) \cdot 6 &&= -12, \\
P_7 &= S_9 \cdot S_1 0 &&= (-6) \cdot 14 &&= -84.
\end{aligned}$$

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)
    n = A.rows
    let C be a new n × n matrix
    if n == 1 then
        c11 = a11 · b11
    else
        partition A, B, and C as into n/2 × n/2 submatrices
        let S1, S2, ..., S10 be new n/2 × n/2 matrices
        let P1, P2, ..., P7 be new n/2 × n/2 matrices
        S1 = B12 − B22
        S2 = A11 + A12
        S3 = A21 + A22
        S4 = B21 − B11
        S5 = A11 + A22
        S6 = B11 + B22
        S7 = A12 − A22
        S8 = B21 + B22
        S9 = A11 − A21
        S10 = B11 − B12
        P1 = Square-Matrix-Multiply-Strassen(A11, S1)
        P2 = Square-Matrix-Multiply-Strassen(S2, B22)
        P3 = Square-Matrix-Multiply-Strassen(S3, B11)
        P4 = Square-Matrix-Multiply-Strassen(A22, S4)
        P5 = Square-Matrix-Multiply-Strassen(S5, S6)
        P6 = Square-Matrix-Multiply-Strassen(S7, S8)
        P7 = Square-Matrix-Multiply-Strassen(S9, S10)
        C11 = P5 + P4 − P2 + P6
        C12 = P1 + P2
        C21 = P3 + P4
        C22 = P5 + P1 − P3 − P7
    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 $n \le m < 2^{(\lg n)+1} = 2^{\lg n} \cdot 2 = 2n$. 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)
    x = a · c
    y = b · d
    real-part = x − y
    imaginary-part = (a + b) · (c + d) − x − y
    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

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

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

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

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

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 $T(n) \leq cn^{\log_3 4}$ fails. Then show how to subtract off a lower-order term to make a substitution proof work.

The initial guess is

$$T(n) \leq cn^{\log_3 4} \;\; \forall\, n \geq n_0,$$

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

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

which does not imply $T(n) \leq cn^{\log_3 4}$ for any choice of c.
Our new guess is

$$T(n) \leq cn^{\log_3 4} - dn \;\; \forall\, n \geq n_0,$$

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

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

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

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 $T(n) \leq cn^2$ fails. Then show how to subtract off a lower-order term to make a substitution proof work.

The initial guess is

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

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

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

which does not imply $T(n) \leq cn^2$ 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

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

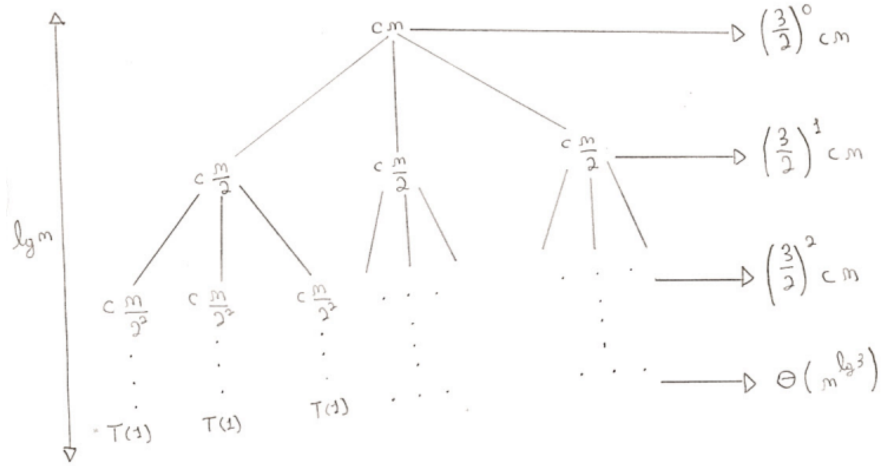
Finally, we have

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

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

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

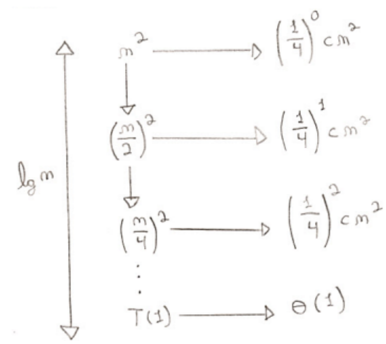
where c, d, and n_0 are positive constants. Substituting into the recurrence yields

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

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

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

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

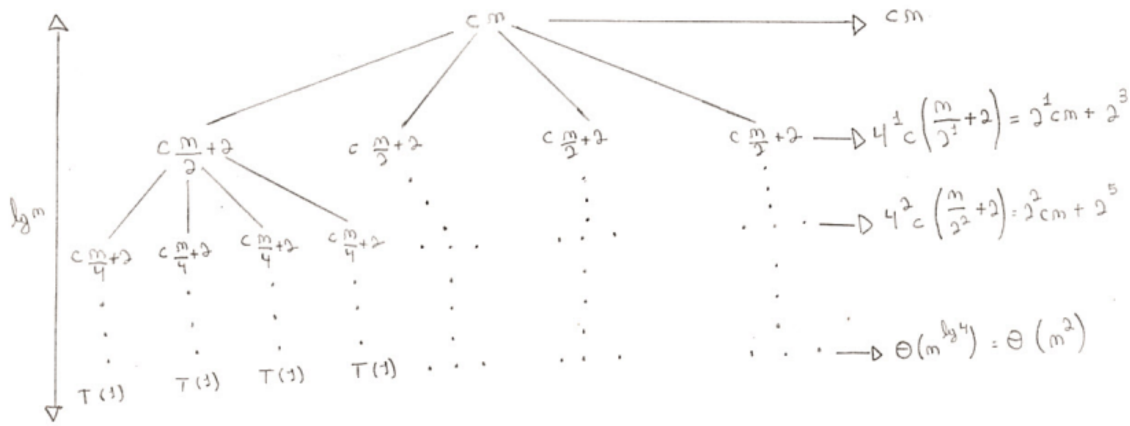
where d, and n_0 are positive constants. Substituting into the recurrence and using the same constant $c > 0$ as before yields

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

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

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

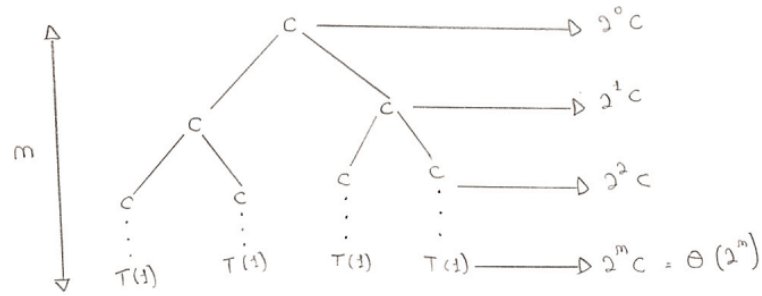
where c, d, and n_0 are positive constants. Substituting into the recurrence yields

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

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

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

$$T(n) \le c2^n - d \ \forall\, n \ge n_0,$$

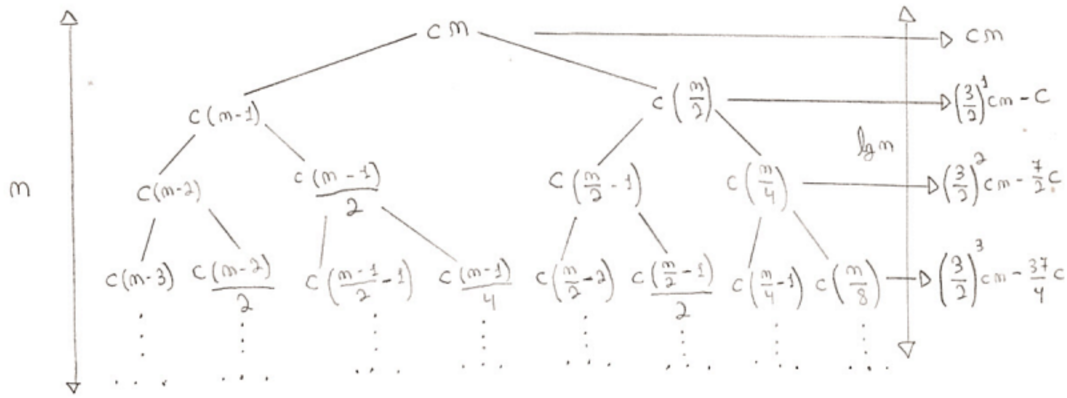
where c, d, and n_0 are positive constants. Substituting into the recurrence yields

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

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

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

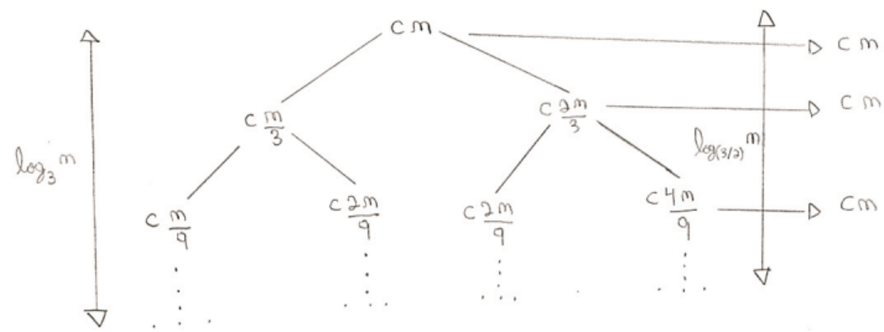
$$f(n) = n^2 = O(n^{\lg n - \epsilon}),\ (\epsilon = 1)$$

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 $T(n) = T(n/3) + T(2n/3) + cn$.



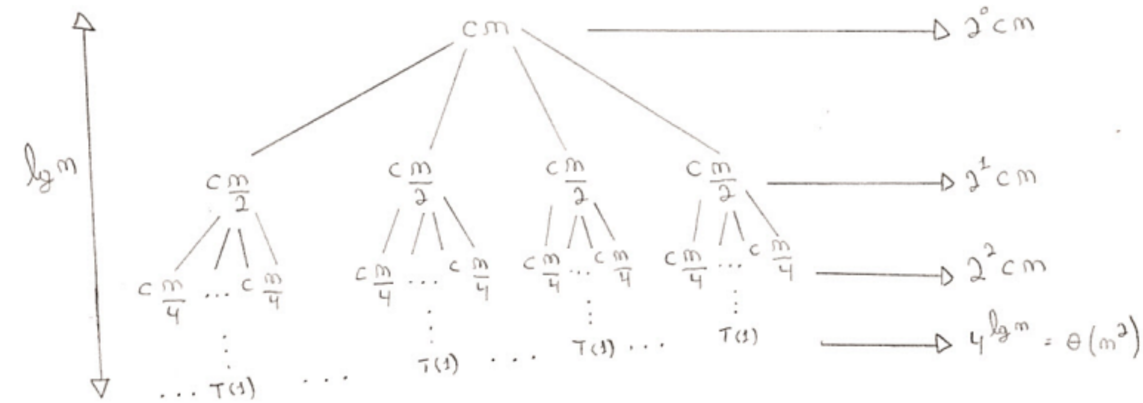
The tree is complete until level $\log_3 n$. The cost of the tree from the root to level $\log_3 n$ is

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

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

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

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

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

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

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

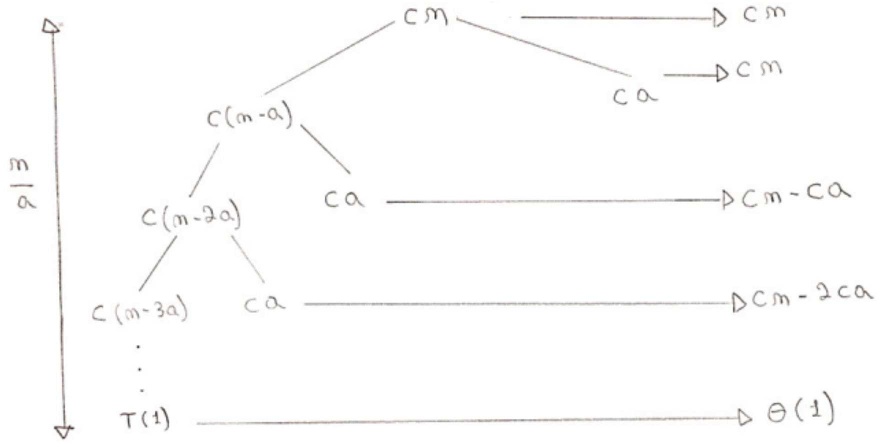
where d, and n_0 are positive constants. Substituting into the recurrence yields

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

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

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

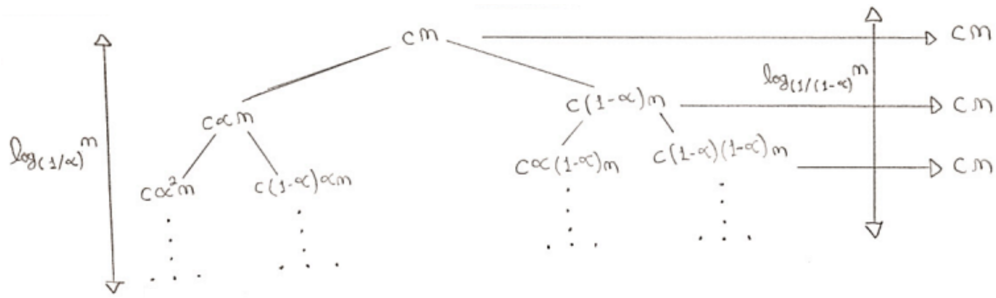
$$T(n) \geq \frac{c}{2a}n^2 \ \forall n \geq n_0,$$

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
$$T(n) \le cn^2 \lg^2 n \ \forall n \ge n_0,$$
where c and n_0 are positive constants. Substituting into the recurrence yields
$$\begin{aligned}
T(n) &\le 4c\left(\left(\frac{n}{2}\right)^2 \lg^2\left(\frac{n}{2}\right)\right) + n^2 \lg n \\
&= 4c\left(\frac{n^2}{4} \lg\left(\frac{n}{2}\right) \lg\left(\frac{n}{2}\right)\right) + n^2 \lg n \\
&= cn^2 \lg\left(\frac{n}{2}\right) \lg\left(\frac{n}{2}\right) + n^2 \lg n \\
&= cn^2 \lg\left(\frac{n}{2}\right) \lg n - cn^2 \lg\left(\frac{n}{2}\right) + n^2 \lg n \\
&= cn^2 \lg^2 n - cn^2 \lg n - cn^2 \lg n + cn^2 + n^2 \lg n \\
&\le cn^2 \lg^2 n,
\end{aligned}$$
where the last step holds as long as $c \ge 1$.

4.5-5 Consider the regularity condition $af(n/b) \ge cf(n)$ for some constant $c < 1$, which is part of case 3 of the master theorem. Give an example of constants $a \ge 1$ 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
$$af\left(\frac{n}{b}\right) \le cf(n) \to \frac{n}{2} \cos\left(\frac{n}{2}\right) \le c(n \cos n),$$
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 $A[p \dots q]$ 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 $af(n/b) = 4(n/2)^2\sqrt{n/2} = (n^{5/2})/\sqrt{2} \leq cn^{5/2}$ for $1/\sqrt{2} \leq c < 1$. 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

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

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

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

where the last step holds as long as $-6c \lg n - 12c + n/2 \leq 0$ (skipped simplification).
Our guess for the lower bound 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

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

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

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

Our guess for the upper bound is

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

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

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

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

Our guess for the lower bound is

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

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

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

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

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

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

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

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

and

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

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.

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

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 $1 \le i < k \le m$ and $1 \le j < l \le n$, we have

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

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:

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

(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 $f(1) \le f(2) \le \cdots \le f(m)$ 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

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

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

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

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

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

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

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

then we have

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

cancelling equal terms on both sides, we have

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

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)
>   flips = ⌈lg(b − a)⌉
>   count = ∞
>   while count > b do
>       count = 0
>       for i = 1 to flips do
>           count = count + (2^(i−1) · Random(0, 1))
>   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()
>   while 1 do
>       r1 = Random(0, 1)
>       r2 = Random(0, 1)
>       if r1 ≠ r2 then
>           return r1
> ```
>
> 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)
    n = A.length
    swap A[1] with A[Random(1, n)]
    for i = 2 to n - 1 do
        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 $A[1, \dots, 1]$ contains this 1-permutation with probability $(n - i + 1)/n! = (n - 1)!/n! = 1/n$. The subarray $A[1, \dots, 1]$ has a single element and this element was randomly choosed among the n elements of the array. Thus, $A[1, \dots, 1]$ 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)
    n = A.length
    for i = 1 to n - 1 do
        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 $A[i, \dots, n]$, we swapped it with a random element from anywhere in the array:

```
Permute-With-All(A)
    n = A.length
    for i = 1 to n do
        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)
    n = A.length
    let B[1 . . . n] be a new array
    offset = Random(1, n)
    for i = 1 to n do
        dest = i + offset
        if dest > n then
            dest = dest − n
        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 $\Pr\{X_i\} = \mathrm{E}[X_i] \le (i-1)/n^3$. 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

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

Thus, the probability that all elements are unique is

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

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 $0 \le m \le n$, 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)
    I = ∅
    n = A.length
    index = −1
    while |I| < n do
        i = Random(1, n)
        I = I ∪ {i}
        if A[i] == x then
            index = i
            break
    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 $i \in I \to A[i] = x$. 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?

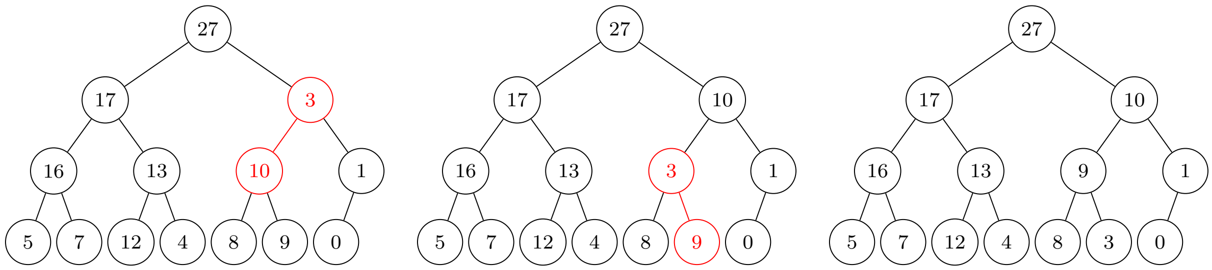
> No. The element 6 is the parent of the element 7 and $6 < 7$, which violates the min-heap property.

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)
    l = Left(i)
    r = Right(i)
    if l ≤ A.heap-size and A[l] < A[i] then
        smallest = l
    else
        smallest = i
    if r ≤ A.heap-size and A[r] < A[smallest] then
        smallest = r
    if smallest ≠ i then
        exchange A[i] with A[smallest]
        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.

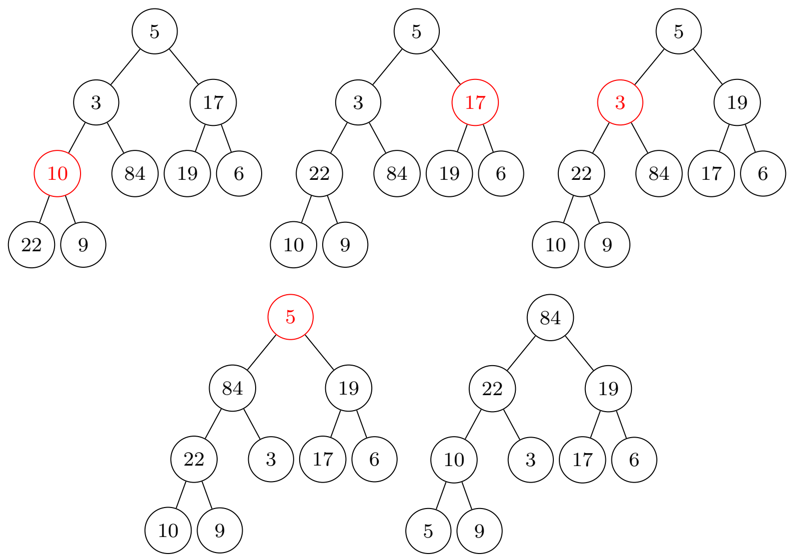
```
Max-Heapify-Iterative(A, i)
    solved = False
    current-node = i
    while not solved do
        l = Left(current-node)
        r = Right(current-node)
        if l ≤ A.heap-size and A[l] > A[current-node] then
            largest = l
        else
            largest = current-node
        if r ≤ A.heap-size and A[r] > A[largest] then
            largest = r
        if largest ≠ current-node then
            exchange A[current-node] with A[largest]
            current-node = largest
        else
            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 $A[\text{LEFT}(i)] \geq A[\text{RIGHT}(i)] > A[i]$ 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,

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

which shows that it also holds for h.

Section 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

$$\begin{aligned}
\sum_{k=2}^{n} \lg\left(1 - \frac{1}{k^2}\right) &= \sum_{k=2}^{n} \lg\left(\frac{k^2-1}{k^2}\right) \\
&= \sum_{k=2}^{n} \lg\left(\frac{(k-1)}{k} \cdot \frac{(k+1)}{k}\right) \\
&= \sum_{k=2}^{n}\left(\lg\left(\frac{k-1}{k}\right) + \lg\left(\frac{k+1}{k}\right)\right) \\
&= \lg\frac{1}{2} + \lg\frac{3}{2} + \lg\frac{2}{3} + \lg\frac{4}{3} + \lg\frac{3}{4} + \lg\frac{5}{4} + \cdots + \lg\frac{n-2}{n-1} + \lg\frac{n}{n-1} + \lg\frac{n-1}{n} + \lg\frac{n+1}{n} \\
&= \lg 1 - \lg 2 + \cancel{\lg 3} - \cancel{\lg 2} + \cancel{\lg 2} - \cancel{\lg 3} + \cancel{\lg 4} - \cancel{\lg 3} + \cancel{\lg 3} - \cancel{\lg 4} + \cancel{\lg 5} - \cancel{\lg 4} + \dots \\
&\quad + \cancel{\lg(n-2)} - \cancel{\lg(n-1)} + \cancel{\lg n} - \cancel{\lg(n-1)} + \cancel{\lg(n-1)} - \cancel{\lg n} + \lg(n+1) - \lg n \\
&= 0 - 1 + \lg(n+1) - \lg n \\
&= \lg(n+1) - \lg(n) - 1.
\end{aligned}$$

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.

$$
\begin{aligned}
\sum_{k=1}^{n} &= 1 + \sum_{k=2}^{n} \frac{1}{k^2} \\
&\leq 1 + \int_1^n \frac{dx}{x^2} \\
&= 1 + \left(-\frac{1}{x}\bigg|_1^n \right) \\
&= 1 + \left(-\frac{1}{n} - \left(-\frac{1}{1} \right) \right) \\
&= 2 - \frac{1}{n} \\
&\leq 2.
\end{aligned}
$$

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

$$\int_0^n x^3 dx \le \sum_{k=1}^{n} k^3 \le \int_1^{n+1} x^3 dx.$$

For a lower bound, we obtain

$$\sum_{k=1}^{n} k^3 \ge \int_0^n x^3 dx = \left.\frac{x^4}{4}\right|_0^n = \frac{n^4}{4} = \Omega(n^4).$$

For the upper bound, we obtain

$$\sum_{k=1}^{n} k^3 \le \int_1^{n+1} x^3 dx = \left.\frac{x^4}{4}\right|_1^{n+1} = \frac{(n+1)^4 - 1}{4} = O(n^4).$$

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

$$\sum_{k=1}^{n} \frac{1}{k} \le \int_0^n \frac{1}{x} dx,$$

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,

$$\sum_{k=1}^{n} \lg^s k = \Theta(n \lg^s n).$$

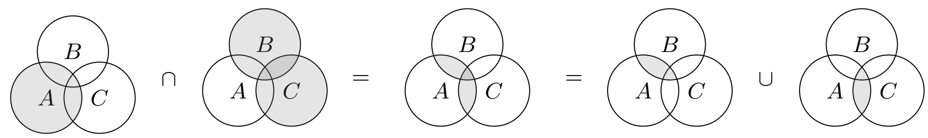
(c) It is easy to see that this summation is greater than the one from item (a). Thus, it is $\Omega(n^{(r+1)})$. Also, we have

$$\sum_{k=1}^{n} k^r \lg^s k \leq \sum_{k=1}^{n} n^r \lg^s n = O(n^{(r+1)} \lg^s n).$$

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).



$$A \quad \cap \quad (B \cup C) \quad = \quad A \cap (B \cup C) \quad = \quad (A \cap B) \quad \cup \quad (A \cap C)$$

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***:

$$\begin{aligned}
&|A_1 \cup A_2 \cup \cdots \cup A_n| = \\
&\quad |A_1| + |A_2| + \cdots + |A_n| \\
&\quad - |A_1 \cap A_2| - |A_1 \cap A_3| - \cdots \qquad \text{(all pairs)} \\
&\quad + |A_1 \cap A_2 \cap A_3| + \cdots \qquad \text{(all triples)} \\
&\quad \vdots \\
&\quad + (-1)^{n-1} |A_1 \cap A_2 \cap \cdots \cap A_n|.
\end{aligned}$$

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 $|A| \le |B|$;

b. if f is surjective, then $|A| \ge |B|$.

(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, $|B| \ge |f(A)| = |A| \to |A| \le |B|$.

(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, $|A| \ge |f(A)| = |B| \to |A| \ge |B|$.

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 $0 < k \le n$.

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

C.1-6 Prove the identity

$$\binom{n}{k} = \frac{n}{n-k}\binom{n-1}{k}$$

for $0 \le k < n$.

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

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 $0 \le k \le n$ 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

$$\begin{array}{c}
\binom{0}{0} \\
\binom{1}{0} \quad \binom{1}{1} \\
\binom{2}{0} \quad \binom{2}{1} \quad \binom{2}{2} \\
\binom{3}{0} \quad \binom{3}{1} \quad \binom{3}{2} \quad \binom{3}{3} \\
\binom{4}{0} \quad \binom{4}{1} \quad \binom{4}{2} \quad \binom{4}{3} \quad \binom{4}{4} \\
\binom{5}{0} \quad \binom{5}{1} \quad \binom{5}{2} \quad \binom{5}{3} \quad \binom{5}{4} \quad \binom{5}{5} \\
\binom{6}{0} \quad \binom{6}{1} \quad \binom{6}{2} \quad \binom{6}{3} \quad \binom{6}{4} \quad \binom{6}{5} \quad \binom{6}{6}
\end{array}$$

Using the above table and the result of C.1-7, we have the Pascal's triangle

$$\begin{array}{c}
1 \\
1 \quad 1 \\
1 \quad 2 \quad 1 \\
1 \quad 3 \quad 3 \quad 1 \\
1 \quad 4 \quad 6 \quad 4 \quad 1 \\
1 \quad 5 \quad 10 \quad 10 \quad 5 \quad 1 \\
1 \quad 6 \quad 15 \quad 20 \quad 15 \quad 6 \quad 1
\end{array}$$

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

$$\begin{array}{ccccccccccccc}
 & & & & & & 1 & & & & & & \\
 & & & & & 1 & & 1 & & & & & \\
 & & & & 1 & & 2 & & 1 & & & & \\
 & & & 1 & & 3 & & 3 & & 1 & & & \\
 & & 1 & & 4 & & 6 & & 4 & & 1 & & \\
 & 1 & & 5 & & 10 & & 10 & & 5 & & 1 & \\
1 & & 6 & & 15 & & 20 & & 15 & & 6 & & 1 \\
 & & & & & & \vdots & & & & & &
\end{array}$$

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 $A_1, A_2, \ldots,$

$$\Pr\{A_1 \cup A_2 \cup \cdots\} \leq \Pr\{A_1\} + \Pr\{A_2\} + \cdots .$$

From (C.13) we have

$$\Pr\{A_1 \cup A_2\} \leq \Pr\{A_1\} + \Pr\{A_2\},$$

which implies

$$\begin{aligned}
\Pr\{A_1 \cup A_2 \cup \cdots\} &= \Pr\{A_1 \cup (A_2 \cup \cdots)\} \\
&\leq \Pr\{A_1\} + \Pr\{A_2 \cup (A_3 \cup \cdots)\} \\
&\leq \Pr\{A_1\} + \Pr\{A_2\} + \Pr\{A_3 \cup (A_4 \cup \cdots)\} \\
&\leq \Pr\{A_1\} + \Pr\{A_2\} + \Pr\{A_3\} \cdots .
\end{aligned}$$

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

$$\begin{aligned}
\mathrm{Var}[X] &= \mathrm{E}[X^2] - \mathrm{E}^2[X] && (\text{since } X^2 = X) \\
&= \mathrm{E}[X] - \mathrm{E}[X]\mathrm{E}[X] \\
&= \mathrm{E}[X](1 - \mathrm{E}[X]) \\
&= \mathrm{E}[X](1 - \Pr\{X = 1\}) \\
&= \mathrm{E}[X]\Pr\{X = 0\} \\
&= \mathrm{E}[X]\mathrm{E}[1 - X].
\end{aligned}$$

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

$$\begin{aligned}
\mathrm{Var}[aX] &= \mathrm{E}[(aX - \mathrm{E}[aX])^2] \\
&= \mathrm{E}[(aX - a\mathrm{E}[X])^2] \\
&= \mathrm{E}[a^2(X - \mathrm{E}[X])^2] \\
&= a^2\mathrm{E}[(X - \mathrm{E}[X])^2] \\
&= a^2\mathrm{Var}[X].
\end{aligned}$$