1 The Role of Algorithms in Computing

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: browse the price of the restaurants with ascending prices on NTU street.
- Convex hull: computing the diameter of set of points.

1.1-2

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

Memory efficiency and coding efficiency.

1.1-3

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

Linked-list:

- Strengths: insertion and deletion.
- Limitations: random access.

1.1-4

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

- Similar: finding path with shortest distance.
- Different: traveling-salesman has more constraints.

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.

- Best: find the GCD of two positive integer numbers.
- Approximately: find the solution of differential equations.

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.

Drive navigation.

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?

$$
\begin{aligned}
8n^2 &< 64n \lg n \\
2^n &< n^8 \\
2 \le n &\le 43.
\end{aligned}
$$

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?

$$
\begin{aligned}
100n^2 &< 2^n \\
n &\ge 15.
\end{aligned}
$$

Problem 1-1

> 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.

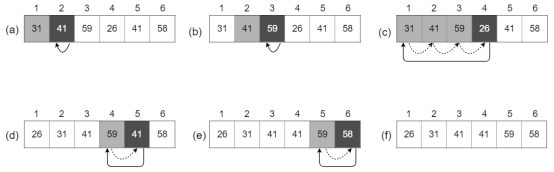
	1 second	1 minute	1 hour	1 day	1 month	1 year	1 century
$\lg n$	2^{10^6}	$2^{6\times10^7}$	$2^{3.6\times10^9}$	$2^{8.64\times10^{10}}$	$2^{2.59\times10^{12}}$	$2^{3.15\times10^{13}}$	$2^{3.15\times10^{15}}$
$\sqrt{n}$	10^{12}	3.6×10^{15}	1.3×10^{19}	7.46×10^{21}	6.72×10^{24}	9.95×10^{26}	9.95×10^{30}
n	10^6	6×10^7	3.6×10^9	8.64×10^{10}	2.59×10^{12}	3.15×10^{13}	3.15×10^{15}
$n \lg n$	6.24×10^4	2.8×10^6	1.33×10^8	2.76×10^9	7.19×10^{10}	7.98×10^{11}	6.86×10^{13}
n^2	1000	7745	60000	293938	1609968	5615692	56156922
n^3	100	391	1532	4420	13736	31593	146645
2^n	19	25	31	36	41	44	51
$n!$	9	11	12	13	15	16	17

2 Getting Started

2.1 Insertion sort

2.1-1

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



The operation of INSERTION-SORT on the array $A = \langle 31, 41, 59, 26, 41, 58 \rangle$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles.

(a)-(e) are iterations of the for loop of lines 1-8.

In each iteration, the black rectangle holds the key taken from $A[i]$, which is compared with the values in shaded rectangles to its left in the test of line 5. Dotted arrows show array values moved one position to the right in line 6. and solid arrows indicate where the key moves to in line 8.

(f) is the final sorted array.

The changes of array A during traversal:

$$
\begin{aligned}
A &= \langle 31, 41, 59, 26, 41, 58 \rangle \\
A &= \langle 31, 41, 59, 26, 41, 58 \rangle \\
A &= \langle 31, 41, 59, 26, 41, 58 \rangle \\
A &= \langle 26, 31, 41, 59, 41, 58 \rangle \\
A &= \langle 26, 31, 41, 41, 59, 58 \rangle \\
A &= \langle 26, 31, 41, 41, 58, 59 \rangle
\end{aligned}
$$

2.1-2

> Rewrite the INSERTION-SORT procedure to sort into nonincreasing instead of nondecreasing order.

```
INSERTION-SORT(A)
    for j = 2 to A.length
        key = A[j]
        i = j - 1
        while i > 0 and A[i] < key
            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 v.
>
> **Output:** An index i such that $v = A[i]$ or the special value NIL if v does not appear in A.
>
> Write pseudocode for ***linear search***, which scans through the sequence, looking for v. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

```
LINEAR-SEARCH(A, v)
    for i = 1 to A.length
       if A[i] == v
            return i
    return NIL
```

Loop invariant: At the start of each iteration of the **for** loop, the subarray $A[1..i-1]$ consists of elements that are different than v.

Initialization: Before the first loop iteration ($i = 1$), the subarray is the empty array, so the proof is trivial.

Maintenance: During each loop iteration, we compare v with $A[i]$. If they are the same, we return i, which is the correct result. Otherwise, we continue to the next iteration of the loop. At the end of each loop iteration, we know the subarray $A[1..i]$ does not contain v, so the loop invariant holds true. Incrementing i for the next iteration of the **for** loop then preserves the loop invariant.

Termination: The loop terminates when $i > A.\text{length} = n$. Since i increases by 1, we must have $i = n + 1$ at that time. Substituting $n + 1$, for i in the wording of the loop invariant, we have that the subarray $A[1..n]$ consists of elements that are different than v. Thus, we return NIL. Observing that $A[1..n]$, we conclude that the entire array does not have any element equal to v. Hence the algorithm is correct.

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.

Input: An array of booleans $A = \langle a_1, a_2, \ldots, a_n \rangle$ and an array of booleans $B = \langle b_1, b_2, \ldots, b_n \rangle$, each representing an integer stored in binary format (each digit is a number, either 0 or 1, **least-significant digit first**) and each of length n.

Output: An array $C = \langle c_1, c_2, \ldots, c_{n+1} \rangle$ such that $C' = A' + B'$ where A', B' and C' are the integers, represented by A, B and C.

```
ADD-BINARY(A, B)
    carry = 0
    for i = 1 to A.length
        sum = A[i] + B[i] + carry
        C[i] = sum % 2  // remainder
        carry = sum / 2 // quotient
    C[A.length + 1] = carry
    return C
```

2.2 Analyzing algorithms

2.2-1

> Express the function $n^3/1000 - 100n^2 - 100n + 3n$ 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.

- Pseudocode:

```
n = A.length
for i = 1 to n - 1
    minIndex = i

    for j = i + 1 to n
        if A[j] < A[minIndex]
            minIndex = j
    swap(A[i], A[minIndex])
```

- Loop invariant:

  At the start of the loop in line 1, the subarray $A[1..i-1]$ consists of the smallest $i - 1$ elements in array A with sorted order.

- Why does it need to run for only the first $n - 1$ elements, rather than for all n elements?

  After $n - 1$ iterations, the subarray $A[1..n-1]$ consists of the smallest $i - 1$ elements in array A with sorted order. Therefore, $A[n]$ is already the largest element.

- Running time: $\Theta(n^2)$.

2.2-3

> Consider linear search again (see Exercise 2.1-3). How many elements of the in- put 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.

If the element is present in the sequence, half of the elements are likely to be checked before it is found in the average case. In the worst case, all of them will be checked. That is, $n/2$ checks for the average case and n for the worst case. Both of them are $\Theta(n)$.

2.2-4

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

You can modify any algorithm to have a best case time complexity by adding a special case. If the input matches this special case, return the pre-computed answer.

2.3 Designing 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] [41] [52] [26] [38] [57] [9] [49]

↓

[3|41] [26|52] [38|57] [9|49]

↓

[3|26|41|52] [9|38|49|57]

↓

[3|9|26|38|41|49|52|57]

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.

```
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
        L[i] = A[p + i − 1]
    for j = 1 to n2
        R[j] = A[q + j]
    i = 1
    j = 1
    for k = p to r
        if i > n1
            A[k] = R[j]
            j = j + 1
        else if j > n2
            A[k] = L[i]
            i = i + 1
        else if L[i] ≤ R[j]
            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 $T(n) = n \lg n$.

- Base case

  For $n = 2^1$, $T(n) = 2 \lg 2 = 2$.

- Suppose $n = 2^k$, $T(n) = n \lg n = 2^k \lg 2^k = 2^k k$.

  For $n = 2^{k+1}$,

$$\begin{aligned} T(n) &= 2T(2^{k+1}/2) + 2^{k+1} \\ &= 2T(2^k) + 2^{k+1} \\ &= 2 \cdot 2^k k + 2^{k+1} \\ &= 2^{k+1}(k+1) \\ &= 2^{k+1} \lg 2^{k+1} \\ &= n \lg n. \end{aligned}$$

By P.M.I., $T(n) = n \lg n$, when n is an exact power of 2.

2.3-4

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

It takes $\Theta(n)$ time in the worst case to insert $A[n]$ into the sorted array $A[1..n-1]$. Therefore, the recurrence

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

The solution of the recurrence is $\Theta(n^2)$.

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 v 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 $\Theta(\lg n)$.

- Iterative:

```
ITERATIVE-BINARY-SEARCH(A, v, low, high)
    while low ≤ high
        mid = floor((low + high) / 2)
        if v == A[mid]
            return mid
        else if v > A[mid]
            low = mid + 1
        else high = mid − 1
    return NIL
```

- Recursive:

```
RECURSIVE-BINARY-SEARCH(A, v, low, high)
    if low > high
        return NIL
    mid = floor((low + high) / 2)
    if v == A[mid]
        return mid
    else if v > A[mid]
        return RECURSIVE-BINARY-SEARCH(A, v, mid + 1, high)
    else return RECURSIVE-BINARY-SEARCH(A, v, low, mid − 1)
```

Each time we do the comparison of v with the middle element, the search range continues with range halved.

The recurrence

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

The solution of the recurrence is $T(n) = \Theta(\lg n)$.

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 $A[i..j-1]$. Can we use a binary search (see Exercise 2.3-5) instead to improve the overall worst-case running time of insertion sort to $\Theta(n \lg n)$?

Each time the **while** loop of lines 5-7 of INSERTION-SORT scans backward through the sorted array $A[1..j-1]$. The loop not only searches for the proper place for $A[j]$, but it also moves each of the array elements that are bigger than $A[j]$ one position to the right (line 6). These movements takes $\Theta(j)$ time, which occurs when all the $j-1$ elements preceding $A[j]$ are larger than $A[j]$. The running time of using binary search to search is $\Theta(\lg j)$, which is still dominated by the running time of moving element $\Theta(j)$.

Therefore, we can't improve the overall worst-case running time of insertion sort to $\Theta(n \lg n)$.

2.3-7 ⋆

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.

First, sort S, which takes $\Theta(n \lg n)$. Then, for each element s_i in S, $i = 1, \ldots, n$, search $A[i+1..n]$ for $s_i' = x - s_i$ by binary search, which takes $\Theta(\lg n)$.

- If s_i' is found, return its position;

- otherwise, continue for next iteration.

The time complexity of the algorithm is $\Theta(n \lg n) + n \cdot \Theta(\lg n) = \Theta(n \lg n)$.

Problem 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. The worst-case time to sort a list of length k by insertion sort is $\Theta(k^2)$. Therefore, sorting n/k sublists, each of length k takes $\Theta(k^2 \cdot n/k) = \Theta(nk)$ worst-case time.

b. We have n/k sorted sublists each of length k. To merge these n/k sorted sublists to a single sorted list of length n, we have to take 2 sublists at a time and continue to merge them. This will result in $\lg(n/k)$ steps and we compare n elements in each step. Therefore, the worst-case time to merge the sublists is $\Theta(n \lg(n/k))$.

c. The modified algorithm has time complexity as standard merge sort when $\Theta(nk + n \lg(n/k)) = \Theta(n \lg n)$. Assume $k = \Theta(\lg n)$,

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

d. Choose k be the largest length of sublist on which insertion sort is faster than merge sort.

Problem 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)
    for i = 1 to A.length − 1
        for j = A.length downto i + 1
            if A[j] < A[j − 1]
                exchange A[j] with A[j − 1]
```

a. Let A' denote the output of $\text{BUBBLESORT}(A)$ To prove that BUBBLESORT is correct, we need to prove that it terminates and that

$$A'[1] \le A'[2] \le \cdots \le A'[n], \tag{2.3}$$

where $n = A.\text{length}$. In order to show that BUBBLESORT actually sorts, what else do we need to prove?

The next two parts will prove inequality (2.3).

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 inequality (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' consists of the elements in A but in sorted order.

b. Loop invariant: At the start of each iteration of the **for** loop of lines 2-4, the subarray $A[j..n]$ consists of the elements originally in $A[j..n]$ before entering the loop but possibly in a different order and the first element $A[j]$ is the smallest among them.

Initialization: Initially the subarray contains only the last element $A[n]$, which is trivially the smallest element of the subarray.

Maintenance: In every step we compare $A[j]$ with $A[j-1]$ and make $A[j-1]$ the smallest among them. After the iteration, the length of the subarray increases by one and the first element is the smallest of the subarray.

Termination: The loop terminates when $j = i$. According to the statement of loop invariant, $A[i]$ is the smallest among $A[i..n]$ and $A[i..n]$ consists of the elements originally in $A[i..n]$ before entering the loop.

c. Loop invariant: At the start of each iteration of the **for** loop of lines 1-4, the subarray $A[1..i-1]$ consists of the $i-1$ smallest elements in $A[1..n]$ in sorted order. $A[i..n]$ consists of the $n-i+1$ remaining elements in $A[1..n]$.

Initialization: Initially the subarray $A[1..i-1]$ is empty and trivially this is the smallest element of the subarray.

Maintenance: From part (b), after the execution of the inner loop, $A[i]$ will be the smallest element of the subarray $A[i..n]$. And in the beginning of the outer loop, $A[1..i-1]$ consists of elements that are smaller than the elements of $A[i..n]$, in sorted order. So, after the execution of the outer loop, subarray $A[1..i]$ will consists of elements that are smaller than the elements of $A[i+1..n]$, in sorted order.

Termination: The loop terminates when $i = A.\text{length}$. At that point the array $A[1..n]$ will consists of all elements in sorted order.

d. The ith iteration of the **for** loop of lines 1-4 will cause $n-i$ iterations of the **for** loop of lines 2-4, each with constant time execution, so the worst-case running time of bubble sort is $\Theta(n^2)$ which is same as the worst-case running time of insertion sort.

Problem 2-3 Correctness of Horner's rule

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

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

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

```
y = 0
for i = n downto 0
    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. $\Theta(n)$.

b.

```
NAIVE-HORNER()
    y = 0
    for k = 0 to n
        temp = 1
        for i = 1 to k
            temp = temp * x
        y = y + a[k] * temp
```

The running time is $\Theta(n^2)$, because of the nested loop. It is obviously slower.

c. Initialization: It is pretty trivial, since the summation has no terms which implies $y = 0$.

Maintenance: By using the loop invariant, in the end of the i-th iteration, we have

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

Termination: The loop terminates at $i = -1$. If we substitute,

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

d. The invariant of the loop is a sum that equals a polynomial with the given coefficients.

Problem 2-4 Inversions

Let $A[1..n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an ***inversion*** of A.

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

b. What array with elements from the set $\{1, 2, \ldots, n\}$ 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 of n elements in $\Theta(n \lg n)$ worst-case time. (Hint: Modify merge sort).

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

b. The array $\langle n, n-1, \ldots, 1 \rangle$ has the most inversions $(n-1) + (n-2) + \cdots + 1 = n(n-1)/2$.

c. The running time of insertion sort is a constant times the number of inversions. Let $I(i)$ denote the number of $j < i$ such that $A[j] > A[i]$. Then $\sum_{i=1}^{n} I(i)$ equals the number of inversions in A.

Now consider the **while** loop on lines 5-7 of the insertion sort algorithm. The loop will execute once for each element of A which has index less than j is larger than $A[j]$. Thus, it will execute $I(j)$ times. We reach this **while** loop once for each iteration of the **for** loop, so the number of constant time steps of insertion sort is $\sum_{j=1}^{n} I(j)$ which is exactly the inversion number of A.

d. We'll call our algorithm COUNT-INVERSIONS for modified merge sort. In addition to sorting A, it will also keep track of the number of inversions.

COUNT-INVERSIONS(A, p, r) sorts $A[p..r]$ and returns the number of inversions in the elements of $A[p..r]$, so left and right track the number of inversions of the form (i, j) where i and j are both in the same half of A.

MERGE-INVERSIONS(A, p, q, r) returns the number of inversions of the form (i, j) where i is in the first half of the array and j is in the second half. Summing these up gives the total number of inversions in A. The runtime of the modified algorithm is $\Theta(n \lg n)$, which is same as merge sort since we only add an additional constant-time operation to some of the iterations in some of the loops.

```
COUNT-INVERSIONS(A, p, r)
    if p < r
        q = floor((p + r) / 2)
        left = COUNT-INVERSIONS(A, p, q)
        right = COUNT-INVERSIONS(A, q + 1, r)
        inversions = MERGE-INVERSIONS(A, p, q, r) + left + right
        return inversions
```

```
MERGE-INVERSIONS(A, p, q, r)
    n1 = q - p + 1
    n2 = r - q
    let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
    for i = 1 to n1
        L[i] = A[p + i - 1]
    for j = 1 to n2
        R[j] = A[q + j]
    L[n1 + 1] = ∞
    R[n2 + 1] = ∞
    i = 1
    j = 1
    inversions = 0
    for k = p to r
        if L[i] <= R[j]
            A[k] = L[i]
            i = i + 1
        else
            inversions = inversions + n1 - i + 1
            A[k] = R[j]
            j = j + 1
    return inversions
```

3 Growth of Functions

3.1 Asymptotic notation

3.1-1

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

For asymptotically nonnegative functions $f(n)$ and $g(n)$, we know that

$$\begin{aligned} \exists n_1, n_2 : f(n) &\geq 0 \qquad \text{for, } n > n_1 \\ g(n) &\geq 0 \qquad \text{for, } n > n_2. \end{aligned}$$

Let $n_0 = \max(n_1, n_2)$ and we know the equations below would be true for $n > n_0$:

$$\begin{aligned} f(n) &\leq \max(f(n), g(n)) \\ g(n) &\leq \max(f(n), g(n)) \\ (f(n) + g(n))/2 &\leq \max(f(n), g(n)) \\ \max(f(n), g(n)) &\leq (f(n) + g(n)). \end{aligned}$$

Then we can combine last two inequalities:

$$0 \leq \frac{f(n) + g(n)}{2} \leq \max(f(n), g(n)) \leq f(n) + g(n).$$

Which is the definition of $\Theta(f(n) + g(n))$ with $c_1 = \frac{1}{2}$ and $c_2 = 1$

3.1-2

Show that for any real constants a and b, where $b > 0$,

$$(n + a)^b = \Theta(n^b). \tag{3.2}$$

Expand $(n + a)^b$ by the Binomial Expansion, we have

$$(n + a)^b = C_0^b n^b a^0 + C_1^b n^{b-1} a^1 + \cdots + C_b^b n^0 a^b.$$

Besides, we know below is true for any polynomial when $x \geq 1$.

$$a_0 x^0 + a_1 x^1 + \cdots + a_n x^n \leq (a_0 + a_1 + \cdots + a_n) x^n.$$

Thus,

$$C_0^b n^b \leq C_0^b n^b a^0 + C_1^b n^{b-1} a^1 + \cdots + C_b^b n^0 a^b \leq (C_0^b + C_1^b + \cdots + C_b^b) n^b = 2^b n^b.$$

$$\implies (n + a)^b = \Theta(n^b).$$

3.1-3

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

$T(n)$: running time of algorithm A. We just care about the upper bound and the lower bound of $T(n)$.

The statement: $T(n)$ is at least $O(n^2)$.

- Upper bound: Because "$T(n)$ is at least $O(n^2)$", there's no information about the upper bound of $T(n)$.
- Lower bound: Assume $f(n) = O(n^2)$, then the statement: $T(n) \geq f(n)$, but $f(n)$ could be any fuction that is "smaller" than n^2. For example, constant, n, etc, so there's no conclusion about the lower bound of $T(n)$, too.

Therefore, the statement, "The running time of algorithm A is at least $O(n^2)$," is meaningless.

3.1-4

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

- True. Note that $2^{n+1} = 2 \times 2^n$. We can choose $c \geq 2$ and $n_0 = 0$, such that $0 \leq 2^{n+1} \leq c \times 2^n$ for all $n \geq n_0$. By definition, $2^{n+1} = O(2^n)$.
- False. Note that $2^{2n} = 2^n \times 2^n = 4^n$. We can't find any c and n_0, such that $0 \leq 2^{2n} = 4^n \leq c \times 2^n$ for all $n \geq n_0$.

3.1-5

Prove Theorem 3.1.

The theorem states:

> For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

From $f = \Theta(g(n))$, we have that

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for } n > n_0.$$

We can pick the constants from here and use them in the definitions of O and Ω to show that both hold.

From $f(n) = \Omega(g(n))$ and $f(n) = O(g(n))$, we have that

$$\begin{aligned} 0 \leq c_3 g(n) \leq f(n) &\qquad \text{for all } n \geq n_1 \\ \text{and } 0 \leq f(n) \leq c_4 g(n) &\qquad \text{for all } n \geq n_2. \end{aligned}$$

If we let $n_3 = \max(n_1, n_2)$ and merge the inequalities, we get

$$0 \leq c_3 g(n) \leq f(n) \leq c_4 g(n) \text{ for all } n > n_3.$$

Which is the definition of Θ.

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 $\Omega(g(n))$.

If T_w is the worst-case running time and T_b is the best-case running time, we know that

$$\begin{aligned} 0 \leq c_1 g(n) \leq T_b(n) &\qquad \text{for } n > n_b \\ \text{and } 0 \leq T_w(n) \leq c_2 g(n) &\qquad \text{for } n > n_w. \end{aligned}$$

Combining them we get

$$0 \leq c_1 g(n) \leq T_b(n) \leq T_w(n) \leq c_2 g(n) \text{ for } n > \max(n_b, n_w).$$

Since the running time is bound between T_b and T_w and the above is the definition of the Θ-notation, proved.

3.1-7

Prove $o(g(n)) \cap w(g(n))$ is the empty set.

Let $f(n) = o(g(n)) \cap w(g(n))$. We know that for any $c_1 > 0, c_2 > 0$,

$$\begin{aligned} &\exists n_1 > 0 : 0 \leq f(n) < c_1 g(n) \\ &\text{and } \exists n_2 > 0 : 0 \leq c_2 g(n) < f(n). \end{aligned}$$

If we pick $n_0 = \max(n_1, n_2)$, and let $c_1 = c_2$, from the problem definition we get

$$c_1 g(n) < f(n) < c_1 g(n).$$

There is no solutions, which means that the intersection is the empty set.

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 function $g(n, m)$ we denote $O(g(n, m))$ the set of functions:

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

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

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

$$\begin{aligned} \Theta(g(n, m)) = \{ f(n, m) : &\text{ there exist positive constants } c_1, c_2, n_0, \text{ and } m_0 \text{ such that} \\ &0 \leq c_1 g(n, m) \leq f(n, m) \leq c_2 g(n, m) \text{ for all } n \geq n_0 \text{ and } m \geq m_0. \} \end{aligned}$$

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.

$$\begin{aligned} &f(m) \le f(n) \quad \text{for } m \le n \\ &g(m) \le g(n) \quad \text{for } m \le n, \\ \rightarrow\ &f(m) + g(m) \le f(n) + g(n), \end{aligned}$$

which proves the first function.

Then

$$f(g(m)) \le f(g(n)) \text{ for } m \le n.$$

This is true, since $g(m) \le g(n)$ and $f(n)$ is monotonically increasing.

If both functions are nonnegative, then we can multiply the two equalities and we get

$$f(m) \cdot g(m) \le f(n) \cdot g(n).$$

3.2-2

> Prove equation (3.16).

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

3.2-3

> Prove equation (3.19). Also prove that $n! \neq \omega(2^n)$ and $n! \neq o(n^n)$.
>
> $$\lg(n!) = \Theta(n \lg n) \tag{3.19}$$

We can use **Stirling's approximation** to prove these three equations.

For equation (3.19),

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

For $n! \neq \omega(2^n)$,

$$\begin{aligned} \lim_{n\to\infty} \frac{2^n}{n!} &= \lim_{n\to\infty} \frac{2^n}{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1 + \Theta\left(\frac{1}{n}\right)\right)} \\ &= \lim_{n\to\infty} \frac{1}{\sqrt{2\pi n}\left(1 + \Theta\left(\frac{1}{n}\right)\right)}\left(\frac{2e}{n}\right)^n \\ &\le \lim_{n\to\infty} \left(\frac{2e}{n}\right)^n \\ &\le \lim_{n\to\infty} \frac{1}{2^n} = 0, \end{aligned}$$

where the last step holds for $n > 4e$.

For $n! \neq o(n^n)$,

$$\begin{aligned} \lim_{n\to\infty} \frac{n^n}{n!} &= \lim_{n\to\infty} \frac{n^n}{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1 + \Theta\left(\frac{1}{n}\right)\right)} \\ &= \lim_{n\to\infty} \frac{e^n}{\sqrt{2\pi n}\left(1 + \Theta\left(\frac{1}{n}\right)\right)} \\ &= \lim_{n\to\infty} O(\frac{1}{\sqrt{n}})e^n \\ &\ge \lim_{n\to\infty} \frac{e^n}{c\sqrt{n}} \qquad \text{(for some constant } c > 0) \\ &\ge \lim_{n\to\infty} \frac{e^n}{cn} \\ &= \lim_{n\to\infty} \frac{e^n}{c} = \infty. \end{aligned}$$

3.2-4 ⋆

> Is the function $\lceil \lg n \rceil !$ polynomially bounded? Is the function $\lceil \lg \lg n \rceil !$ polynomially bounded?

Proving that a function $f(n)$ is polynomially bounded is equivalent to proving that $\lg(f(n)) = O(\lg n)$ for the following reasons.

- If f is polynomially bounded, then there exist constants c, k, n_0 such that for all $n \ge n_0$, $f(n) \le cn^k$. Hence, $\lg(f(n)) \le kc \lg n$, which means that $\lg(f(n)) = O(\lg n)$.
- If $\lg(f(n)) = O(\lg n)$, then f is polynomially bounded.

In the following proofs, we will make use of the following two facts:

1. $\lg(n!) = \Theta(n \lg n)$
2. $\lceil \lg n \rceil = \Theta(\lg n)$

$\lceil \lg n \rceil !$ is not polynomially bounded because

$$\begin{aligned} \lg(\lceil \lg n \rceil !) &= \Theta(\lceil \lg n \rceil \lg \lceil \lg n \rceil) \\ &= \Theta(\lg n \lg \lg n) \\ &= \omega(\lg n) \\ &\neq O(\lg n). \end{aligned}$$

$\lceil \lg \lg n \rceil !$ is polynomially bounded because

$$\begin{aligned} \lg(\lceil \lg \lg n \rceil !) &= \Theta(\lceil \lg \lg n \rceil \lg \lceil \lg \lg n \rceil) \\ &= \Theta(\lg \lg n \lg \lg \lg n) \\ &= o((\lg \lg n)^2) \\ &= o(\lg^2(\lg n)) \\ &= o(\lg n) \\ &= O(\lg n). \end{aligned}$$

The last step above follows from the property that any polylogarithmic function grows more slowly than any positive polynomial function, i.e., that for constants $a, b > 0$, we have $\lg^b n = o(n^a)$. Substitute $\lg n$ for n, 2 for b, and 1 for a, giving $\lg^2(\lg n) = o(\lg n)$.

Therefore, $\lg(\lceil \lg \lg n \rceil !) = O(\lg n)$, and so $\lceil \lg \lg n \rceil !$ is polynomially bounded.

3.2-5 ⋆

> Which is asymptotically larger: $\lg(\lg^* n)$ or $\lg^*(\lg n)$?

We have $\lg^* 2^n = 1 + \lg^* n$,

$$\begin{aligned} \lim_{n\to\infty} \frac{\lg(\lg^* n)}{\lg^*(\lg n)} &= \lim_{n\to\infty} \frac{\lg(\lg^* 2^n)}{\lg^*(\lg 2^n)} \\ &= \lim_{n\to\infty} \frac{\lg(1 + \lg^* n)}{\lg^* n} \\ &= \lim_{n\to\infty} \frac{\lg(1 + n)}{n} \\ &= \lim_{n\to\infty} \frac{1}{1 + n} \\ &= 0. \end{aligned}$$

Therefore, we have that $\lg^*(\lg n)$ is asymptotically larger.

3.2-6

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

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

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.

- Base case

  For $i = 0$,

  $$\begin{aligned} \frac{\phi^0 - \hat{\phi}^0}{\sqrt{5}} &= \frac{1 - 1}{\sqrt{5}} \\ &= 0 \\ &= F_0. \end{aligned}$$

  For $i = 1$,

  $$\begin{aligned} \frac{\phi^1 - \hat{\phi}^1}{\sqrt{5}} &= \frac{(1 + \sqrt{5}) - (1 - \sqrt{5})}{2\sqrt{5}} \\ &= 1 \\ &= F_1. \end{aligned}$$

- Assume
  - $F_{i-1} = (\phi^{i-1} - \hat{\phi}^{i-1})/\sqrt{5}$ and
  - $F_{i-2} = (\phi^{i-2} - \hat{\phi}^{i-2})/\sqrt{5}$,

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

3.2-8

> Show that $k \ln k = \Theta(n)$ implies $k = \Theta(n / \lg n)$.

From the symmetry of Θ,

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

Let's find $\ln n$,

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

Let's divide the two,

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

Problem 3-1 Asymptotic behavior of polynomials

> Let
>
> $$p(n) = \sum_{i=0}^{d} a_i n^i,$$
>
> where $a_d > 0$, be a degree-d polynomial in n, and let k be a constant. Use the definitions of the asymptotic notations to prove the following properties.
>
> **a.** If $k \ge d$, then $p(n) = O(n^k)$.
>
> **b.** If $k \le d$, then $p(n) = \Omega(n^k)$.
>
> **c.** If $k = d$, then $p(n) = \Theta(n^k)$.
>
> **d.** If $k > d$, then $p(n) = o(n^k)$.
>
> **e.** If $k < d$, then $p(n) = \omega(n^k)$.

Let's see that $p(n) = O(n^d)$. We need do pick $c = a_d + b$, such that

$$\sum_{i=0}^{d} a_i n^i = a_d n^d + a_{d-1} n^{d-1} + \cdots + a_1 n + a_0 \le cn^d.$$

When we divide by n^d, we get

$$c = a_d + b \ge a_d + \frac{a_{d-1}}{n} + \frac{a_{d-2}}{n^2} + \cdots + \frac{a_0}{n^d}.$$

and

$$b \ge \frac{a_{d-1}}{n} + \frac{a_{d-2}}{n^2} + \cdots + \frac{a_0}{n^d}.$$

If we choose $b = 1$, then we can choose n_0,

$$n_0 = \max(da_{d-1}, d\sqrt{a_{d-2}}, \ldots, d\sqrt[d]{a_0}).$$

Now we have n_0 and c, such that

$$p(n) \le cn^d \quad \text{for } n \ge n_0,$$

which is the definition of $O(n^d)$.

By chosing $b = -1$ we can prove the $\Omega(n^d)$ inequality and thus the $\Theta(n^d)$ inequality.

It is very similar to prove the other inequalities.

Problem 3-2 Relative asymptotic growths

> Indicate for each pair of expressions (A, B) in the table below, whether A is O, o, Ω, ω, or Θ of B. Assume that $k \ge 1$, $\epsilon > 0$, and $c > 1$ are constants. Your answer should be in the form of the table with "yes" or "no" written in each box.

A	B	O	o	Ω	ω	Θ
$\lg^k n$	n^ϵ	yes	yes	no	no	no
n^k	c^n	yes	yes	no	no	no
$\sqrt{n}$	$n^{\sin n}$	no	no	no	no	no
2^n	$2^{n/2}$	no	no	yes	yes	no
$n^{\lg c}$	$c^{\lg n}$	yes	no	yes	no	yes
$\lg(n!)$	$\lg(n^n)$	yes	no	yes	no	yes

Problem 3-3 Ordering by asymptotic growth rates

a. Rank the following functions by order of growth; that is, find an arrangement $g_1, g_2, \ldots, g_{30}$ of the functions $g_1 = \Omega(g_2), g_2 = \Omega(g_3), \ldots, g_{29} = \Omega(g_{30})$. Partition your list into equivalence classes such that functions $f(n)$ and $g(n)$ are in the same class if and only if $f(n) = \Theta(g(n))$.

$\lg(\lg^* n)$	$2^{\lg^* n}$	$(\sqrt{2})^{\lg n}$	n^2	$n!$	$(\lg n)!$
$(\frac{3}{2})^n$	n^3	$\lg^2 n$	$\lg(n!)$	2^{2^n}	$n^{1/\lg n}$
$\lg\lg n$	$\lg^* n$	$n \cdot 2^n$	$n^{\lg\lg n}$	$\lg n$	1
$2^{\lg n}$	$(\lg n)^{\lg n}$	e^n	$4^{\lg n}$	$(n+1)!$	$\sqrt{\lg n}$
$\lg^*(\lg n)$	$2^{\sqrt{2\lg n}}$	n	2^n	$n \lg n$	$2^{2^{n+1}}$

b. Give an example of a single nonnegative function $f(n)$ such that for all functions $g_i(n)$ in part (a), $f(n)$ is neither $O(g_i(n))$ nor $\Omega(g_i(n))$.

$$
\begin{aligned}
&2^{2^{n+1}}\\
&2^{2^n}\\
&(n+1)!\\
&n!\\
&e^n\\
&n \cdot 2^n\\
&2^n\\
&(3/2)^n\\
&(\lg n)^{\lg n} = n^{\lg\lg n}\\
&(\lg n)!\\
&n^3\\
&n^2 = 4^{\lg n}\\
&n \lg n \text{ and } \lg(n!)\\
&n = 2^{\lg n}\\
&(\sqrt{2})^{\lg n} (= \sqrt{n})\\
&2^{\sqrt{2\lg n}}\\
&\lg^2 n\\
&\ln n\\
&\sqrt{\lg n}\\
&\ln\ln n\\
&2^{\lg^* n}\\
&\lg^* n \text{ and } \lg^*(\lg n)\\
&\lg(\lg^* n)\\
&n^{1/\lg n} (= 2) \text{ and } 1
\end{aligned}
$$

b. For example,

$$
f(n) = \begin{cases} 2^{2^{n+2}} & \text{if } n \text{ is even,} \\ 0 & \text{if } n \text{ is odd.} \end{cases}
$$

for all functions $g_i(n)$ in part (a), $f(n)$ is neither $O(g_i(n))$ nor $\Omega(g_i(n))$.

Problem 3-4 Asymptotic notation properties

Let $f(n)$ and $g(n)$ by asymptotically positive functions. Prove or disprove each of the following conjectures.

a. $f(n) = O(g(n))$ implies $g(n) = O(f(n))$.

b. $f(n) + g(n) = \Theta(\min(f(n), g(n)))$.

c. $f(n) = O(g(n))$ implies $\lg(f(n)) = O(\lg(g(n)))$, where $\lg(g(n)) \geq 1$ and $f(n) \geq 1$ for all sufficiently large n.

d. $f(n) = O(g(n))$ implies $2^{f(n)} = O(2^{g(n)})$.

e. $f(n) = O((f(n))^2)$.

f. $f(n) = O(g(n))$ implies $g(n) = \Omega(f(n))$.

g. $f(n) = \Theta(f(n/2))$.

h. $f(n) + o(f(n)) = \Theta(f(n))$.

a. Disprove, $n = O(n^2)$, but $n^2 \neq O(n)$.

b. Disprove, $n^2 + n \neq \Theta(\min(n^2, n)) = \Theta(n)$.

c. Prove, because $f(n) \geq 1$ after a certain $n \geq n_0$.

$$
\begin{aligned}
&\exists c, n_0 : \forall n \geq n_0, 0 \leq f(n) \leq cg(n)\\
\Rightarrow\ &0 \leq \lg f(n) \leq \lg(cg(n)) = \lg c + \lg g(n).
\end{aligned}
$$

We need to prove that

$$
\lg f(n) \leq d \lg g(n).
$$

We can find d,

$$
d = \frac{\lg c + \lg g(n)}{\lg g(n)} = \frac{\lg c}{\lg g(n)} + 1 \leq \lg c + 1,
$$

where the last step is valid, because $\lg g(n) \geq 1$.

d. Disprove, because $2n = O(n)$, but $2^{2n} = 4^n \neq O(2^n)$.

e. Prove, $0 \leq f(n) \leq cf^2(n)$ is trivial when $f(n) \geq 1$, but if $f(n) < 1$ for all n, it's not correct. However, we don't care this case.

f. Prove, from the first, we know that $0 \leq f(n) \leq cg(n)$ and we need to prove that $0 \leq df(n) \leq g(n)$, which is straightforward with $d = 1/c$.

g. Disprove, let's pick $f(n) = 2^n$. We will need to prove that

$$
\exists c_1, c_2, n_0 : \forall n \geq n_0, 0 \leq c_1 \cdot 2^{n/2} \leq 2^n \leq c_2 \cdot 2^{n/2},
$$

which is obviously untrue.

h. Prove, let $g(n) = o(f(n))$. Then

$$
\exists c, n_0 : \forall n \geq n_0, 0 \leq g(n) < cf(n).
$$

We need to prove that

$$
\exists c_1, c_2, n_0 : \forall n \geq n_0, 0 \leq c_1 f(n) \leq f(n) + g(n) \leq c_2 f(n).
$$

Thus, if we pick $c_1 = 1$ and $c_2 = c + 1$, it holds.

Problem 3-5 Variations on O and Ω

Some authors define Ω in a slightly different way than we do; let's use Ω^∞ (read "omega infinity") for this alternative definition. We say that $f(n) = \Omega^\infty(g(n))$ if there exists a positive constant c such that $f(n) \geq cg(n) \geq 0$ for infinitely many integers n.

a. Show that for any two functions $f(n)$ and $g(n)$ that are asymptotically nonnegative, either $f(n) = O(g(n))$ or $f(n) = \Omega^\infty(g(n))$ or both, whereas this is not true if we use Ω in place of Ω^∞.

b. Describe the potential advantages and disadvantages of using Ω^∞ instead of Ω to characterize the running times of programs.

Some authors also define O in a slightly different manner; let's use O' for the alternative definition. We say that $f(n) = O'(g(n))$ if and only if $|f(n)| = O(g(n))$.

c. What happens to each direction of the "if and only if" in Theorem 3.1 if we substitute O' for O but we still use Ω?

Some authors define $\tilde{O}$ (read "soft-oh") to mean O with logarithmic factors ignored:

$$
\begin{aligned}
\tilde{O}(g(n)) = \{f(n) : &\text{there exist positive constants } c, k, \text{ and } n_0 \text{ such that}\\
&0 \leq f(n) \leq cg(n)\lg^k(n) \text{ for all } n \geq n_0.\}
\end{aligned}
$$

d. Define $\tilde{\Omega}$ and $\tilde{\Theta}$ in a similar manner. Prove the corresponding analog to Theorem 3.1.

a. We have

$$
f(n) = \begin{cases} O(g(n)) \text{ and } \Omega^\infty(g(n)) & \text{if } f(n) = \Theta(g(n)), \\ O(g(n)) & \text{if } 0 \leq f(n) \leq cg(n), \\ \Omega^\infty(g(n)) & \text{if } 0 \leq cg(n) \leq f(n), \text{ for infinitely many integers } n. \end{cases}
$$

If there are only finite n such that $f(n) \geq cg(n) \geq 0$. When $n \to \infty$, $0 \leq f(n) \leq cg(n)$, i.e., $f(n) = O(g(n))$.

Obviously, it's not hold when we use Ω in place of Ω^∞.

b.

- Advantages: We can characterize all the relationships between all functions.
- Disadvantages: We cannot characterize precisely.

c. For any two functions $f(n)$ and $g(n)$, we have if $f(n) = \Theta(g(n))$ then $f(n) = O'(g(n))$ and $f(n) = \Omega(g(n))$ and $f(n) = \Omega(g(n))$.

But the conversion is not true.

d. We have

$$
\begin{aligned}
\tilde{\Omega}(g(n)) = \{f(n) : &\text{there exist positive constants } c, k, \text{ and } n_0 \text{ such that}\\
&0 \leq cg(n)\lg^k(n) \leq f(n) \text{ for all } n \geq n_0.\}
\end{aligned}
$$

$$
\begin{aligned}
\tilde{\Theta}(g(n)) = \{f(n) : &\text{there exist positive constants } c_1, c_2, k_1, k_2, \text{ and } n_0 \text{ such that}\\
&0 \leq c_1 g(n)\lg^{k_1}(n) \leq f(n) \leq c_2 g(n)\lg^{k_2}(n) \text{ for all } n \geq n_0.\}
\end{aligned}
$$

For any two functions $f(n)$ and $g(n)$, we have $f(n) = \tilde{\Theta}(g(n))$ if and only if $f(n) = \tilde{O}(g(n))$ and $f(n) = \tilde{\Omega}(g(n))$.

Problem 3-6 Iterated functions

We can apply the iteration operator * used in the $\lg^*$ function to any monotonically increasing function $f(n)$ over the reals. For a given constant $c \in \mathbb{R}$, we define the iterated function f_c^* by $f_c^*(n) = \min\{i \geq 0 : f^{(i)}(n) \leq c\}$ which need not be well defined in all cases. In other words, the quantity $f_c^*(n)$ is the number of iterated applications of the function f required to reduce its argument down to c or less.

For each of the following functions $f(n)$ and constants c, give as tight a bound as possible on $f_c^*(n)$.

$f(n)$	c	f_c^*
$n-1$	0	$\Theta(n)$
$\lg n$	1	$\Theta(\lg^* n)$
$n/2$	1	$\Theta(\lg n)$
$n/2$	2	$\Theta(\lg n)$
$\sqrt{n}$	2	$\Theta(\lg\lg n)$
$\sqrt{n}$	1	does not converge
$n^{1/3}$	2	$\Theta(\log_3 \lg n)$
$n/\lg n$	2	$\omega(\lg\lg n), o(\lg n)$

4 Divide-and-Conquer

4.1 The maximum-subarray problem

4.1-1

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

It will return the greatest 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.

```
BRUTE-FORCE-FIND-MAXIMUM-SUBARRAY(A)
    n = A.length
    max-sum = -∞
    for l = 1 to n
        sum = 0
        for h = l to n
            sum = sum + A[h]
            if sum > max-sum
                max-sum = sum
                low = l
                high = h
    return (low, high, max-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?

On my computer, n_0 is 37.

If the algorithm is modified to used divide and conquer when $n \geq 37$ and the brute-force approach when n is less, the performance at the crossover point almost doubles. The performance at $n_0 - 1$ stays the same, though (or even gets worse, because of the added overhead).

What I find interesting is that if we set $n_0 = 20$ and used the mixed approach to sort 40 elements, it is still faster than both.

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?

If the original algorithm returns a negative sum, returning an empty subarray instead.

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 $A[1..j]$, extend the answer to find a maximum subarray ending at index $j+1$ by using the following observation: a maximum subarray $A[i..j+1]$, is either a maximum subarray of $A[1..j]$ or a subarray $A[i..j+1]$, for some $1 \le i \le j+1$. Determine a maximum subarray of the form $A[i..j+1]$ in constant time based on knowing a maximum subarray ending at index j.

```
ITERATIVE-FIND-MAXIMUM-SUBARRAY(A)
    n = A.length
    max-sum = -∞
    sum = -∞
    for j = 1 to n
        currentHigh = j
        if sum > 0
            sum = sum + A[j]
        else
            currentLow = j
            sum = A[j]
        if sum > max-sum
            max-sum = sum
            low = currentLow
            high = currentHigh
    return (low, high, max-sum)
```

4.2 Strassen's algorithm for matrix multiplication

4.2-1

> Use Strassen's algorithm to compute the matrix product
>
> $$\begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix} \begin{pmatrix} 6 & 8 \\ 4 & 2 \end{pmatrix}.$$
>
> Show your work.

The first matrices are

$$\begin{aligned} S_1 &= 6 & S_6 &= 8 \\ S_2 &= 4 & S_7 &= -2 \\ S_3 &= 12 & S_8 &= 6 \\ S_4 &= -2 & S_9 &= -6 \\ S_5 &= 6 & S_{10} &= 14. \end{aligned}$$

The products are

$$\begin{aligned} P_1 &= 1 \cdot 6 = 6 \\ P_2 &= 4 \cdot 2 = 8 \\ P_3 &= 6 \cdot 12 = 72 \\ P_4 &= -2 \cdot 5 = -10 \\ P_5 &= 6 \cdot 8 = 48 \\ P_6 &= -2 \cdot 6 = -12 \\ P_7 &= -6 \cdot 14 = -84. \end{aligned}$$

The four matrices are

$$\begin{aligned} C_{11} &= 48 + (-10) - 8 + (-12) = 18 \\ C_{12} &= 6 + 8 = 14 \\ C_{21} &= 72 + (-10) = 62 \\ C_{22} &= 48 + 6 - 72 - (-84) = 66. \end{aligned}$$

The result is

$$\begin{pmatrix} 18 & 14 \\ 62 & 66 \end{pmatrix}.$$

4.2-2

> Write pseudocode for Strassen's algorithm.

```
STRASSEN(A, B)
    n = A.rows
    if n == 1
        return a[1, 1] * b[1, 1]
    let C be a new n x n matrix
    A[1, 1] = A[1..n / 2][1..n / 2]
    A[1, 2] = A[1..n / 2][n / 2 + 1..n]
    A[2, 1] = A[n / 2 + 1..n][1..n / 2]
    A[2, 2] = A[n / 2 + 1..n][n / 2 + 1..n]
    B[1, 1] = B[1..n / 2][1..n / 2]
    B[1, 2] = B[1..n / 2][n / 2 + 1..n]
    B[2, 1] = B[n / 2 + 1..n][1..n / 2]
    B[2, 2] = B[n / 2 + 1..n][n / 2 + 1..n]
    S[1] = B[1, 2] - B[2, 2]
    S[2] = A[1, 1] + A[1, 2]
    S[3] = A[2, 1] + A[2, 2]
    S[4] = B[2, 1] - B[1, 1]
    S[5] = A[1, 1] + A[2, 2]
    S[6] = B[1, 1] + B[2, 2]
    S[7] = A[1, 2] - A[2, 2]
    S[8] = B[2, 1] + B[2, 2]
    S[9] = A[1, 1] - A[2, 1]
    S[10] = B[1, 1] + B[1, 2]
    P[1] = STRASSEN(A[1, 1], S[1])
    P[2] = STRASSEN(S[2], B[2, 2])
    P[3] = STRASSEN(S[3], B[1, 1])
    P[4] = STRASSEN(A[2, 2], S[4])
    P[5] = STRASSEN(S[5], S[6])
    P[6] = STRASSEN(S[7], S[8])
    P[7] = STRASSEN(S[9], S[10])
    C[1..n / 2][1..n / 2] = P[5] + P[4] - P[2] + P[6]
    C[1..n / 2][n / 2 + 1..n] = P[1] + P[2]
    C[n / 2 + 1..n][1..n / 2] = P[3] + P[4]
    C[n / 2 + 1..n][n / 2 + 1..n] = P[5] + P[1] - P[3] - P[7]
    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 $\Theta(n^{\lg 7})$.

We can just extend it to an $n \times n$ matrix and pad it with zeroes. It's obviously $\Theta(n^{\lg 7})$.

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 is time $o(n^{\lg 7})$? What would the running time of this algorithm be?

Assume $n = 3^m$ for some m. Then, using block matrix multiplication, we obtain the recursive running time $T(n) = kT(n/3) + O(1)$.

By master theorem, we can find the largest k to satisfy $\log_3 k < \lg 7$ is $k = 21$.

4.2-5

> V. Pan has discovered a way of multiplying 68×68 matrices using 132464 multiplications, a way of multiplying 70×70 matrices using 143640 multiplications, and a way of multiplying 72×72 matrices using 155424 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?

Using what we know from the last exercise, we need to pick the smallest of the following

$$\begin{aligned} &\log_{68} 132464 \approx 2.795128 \\ &\log_{70} 143640 \approx 2.795122 \\ &\log_{72} 155424 \approx 2.795147. \end{aligned}$$

The fastest one asymptotically is 70×70 using 143640.

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.

- $(kn \times n)(n \times kn)$ produces a $kn \times kn$ matrix. This produces k^2 multiplications of $n \times n$ matrices.
- $(n \times kn)(kn \times n)$ produces an $n \times n$ matrix. This produces k multiplications and $k-1$ additions.

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 three matrices are

$$\begin{aligned} A &= (a+b)(c+d) = ac + ad + bc + bd \\ B &= ac \\ C &= bd. \end{aligned}$$

The result is

$$(B - C) + (A - B - C)i.$$

4.3 The substitution method for solving recurrences

4.3-1

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

We guess $T(n) \le cn^2$,

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

where the last step holds for $c > \frac{1}{2}$.

4.3-2

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

We guess $T(n) \le c \lg(n-a)$,

$$\begin{aligned} T(n) &\le c \lg(\lceil n/2 \rceil - a) + 1 \\ &\le c \lg((n+1)/2 - a) + 1 \\ &= c \lg((n+1-2a)/2) + 1 \\ &= c \lg(n+1-2a) - c \lg 2 + 1 && (c \ge 1) \\ &\le c \lg(n+1-2a) && (a \ge 1) \\ &\le c \lg(n-a), \end{aligned}$$

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 $\Theta(n \lg n)$.

First, we guess $T(n) \le cn \lg n$,

$$\begin{aligned} T(n) &\le 2c \lfloor n/2 \rfloor \lg \lfloor n/2 \rfloor + n \\ &\le cn \lg(n/2) + n \\ &= cn \lg n - cn \lg 2 + n \\ &= cn \lg n + (1-c)n \\ &\le cn \lg n, \end{aligned}$$

where the last step holds for $c \ge 1$.

Next, we guess $T(n) \ge c(n+a) \lg(n+a)$,

$$\begin{aligned} T(n) &\ge 2c(\lfloor n/2 \rfloor + a)(\lg(\lfloor n/2 \rfloor + a) + n \\ &\ge 2c((n-1)/2 + a)(\lg((n-1)/2 + a)) + n \\ &= 2c\frac{n-1+2a}{2} \lg \frac{n-1+2a}{2} + n \\ &= c(n-1+2a) \lg(n-1+2a) - c(n-1+2a) \lg 2 + n \\ &= c(n-1+2a) \lg(n-1+2a) + (1-c)n - (2a-1)c && (0 \le c < 1, n \ge \tfrac{(2a-1)c}{1-c}) \\ &\ge c(n-1+2a) \lg(n-1+2a) && (a \ge 1) \\ &\ge c(n+a) \lg(n+a), \end{aligned}$$

where the last step holds for $c \ge 1$.

4.3-4

> Show that by making a different inductive hyptohesis, 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.

We guess $T(n) \le n \lg n + n$,

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

where the last step holds for $c \ge 1$.

This time, the boundary condition is

$$T(1) = 1 \le cn \lg n + n = 0 + 1 = 1.$$

4.3-5

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

The recurrence is

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) \tag{4.3}$$

To show Θ bound, separately show O and Ω bounds.

- For $O(n \lg n)$, we guess $T(n) \le c(n-2) \lg(n-2) - 2c$,

$$\begin{aligned} T(n) &\le c(\lceil n/2 \rceil - 2) \lg(\lceil n/2 \rceil - 2) + c(\lfloor n/2 \rfloor - 2) \lg(\lfloor n/2 \rfloor - 2) + dn \\ &\le c(n/2 + 1 - 2) \lg(n/2 + 1 - 2) - 2c + c(n/2 - 2) \lg(n/2 - 2) - 2c + dn \\ &\le c(n/2 - 1) \lg(n/2 - 1) + c(n/2 - 1) \lg(n/2 - 1) + dn \\ &= c\frac{n-2}{2} \lg \frac{n-2}{2} + c\frac{n-2}{2} \lg \frac{n-2}{2} - 4c + dn \\ &= c(n-2) \lg \frac{n-2}{2} - 4c + dn \\ &= c(n-2) \lg(n-2) - c(n-2) - 4c + dn \\ &= c(n-2) \lg(n-2) + (d-c)n + 2c - 4c \\ &\le c(n-2) \lg(n-2) - 2c, \end{aligned}$$

  where the last step holds for $c > d$.

- For $\Omega(n \lg n)$, we guess $T(n) \ge c(n+2) \lg(n+2) + 2c$,

$$\begin{aligned} T(n) &\ge c(\lceil n/2 \rceil + 2) \lg(\lceil n/2 \rceil + 2) + c(\lfloor n/2 \rfloor + 2) \lg(\lfloor n/2 \rfloor + 2) + dn \\ &\ge c(n/2 + 2) \lg(n/2 + 2) + 2c + c(n/2 - 1 + 2) \lg(n/2 - 1 + 2) + 2c + dn \\ &\ge c(n/2 + 1) \lg(n/2 + 1) + c(n/2 + 1) \lg(n/2 + 1) + 4c + dn \\ &\ge c\frac{n+2}{2} \lg \frac{n+2}{2} + c\frac{n+2}{2} \lg \frac{n+2}{2} + 4c + dn \\ &= c(n+2) \lg \frac{n+2}{2} + 4c + dn \\ &= c(n+2) \lg(n+2) - c(n+2) + 4c + dn \\ &= c(n+2) \lg(n+2) + (d-c)n - 2c + 4c \\ &\ge c(n+2) \lg(n+2) + 2c, \end{aligned}$$

  where the last step holds for $d > c$.

4.3-6

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

We guess $T(n) \le c(n-a) \lg(n-a)$,

$$\begin{aligned} T(n) &\le 2c(\lfloor n/2 \rfloor + 17 - a) \lg(\lfloor n/2 \rfloor + 17 - a) + n \\ &\le 2c(n/2 + 17 - a) \lg(n/2 + 17 - a) + n \\ &= c(n + 34 - 2a) \lg \frac{n + 34 - 2a}{2} + n \\ &= c(n + 34 - 2a) \lg(n + 34 - 2a) - c(n + 34 - 2a) + n && (c > 1, n > n_0 = f(a)) \\ &\le c(n + 34 - 2a) \lg(n + 34 - 2a) && (a \ge 34) \\ &\le c(n-a) \lg(n-a). \end{aligned}$$

4.3-7

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

We guess $T(n) \le cn^{\log_3 4}$ first,

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

We stuck here.

We guess $T(n) \le cn^{\log_3 4} - dn$ again,

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

where the last step holds for $d \ge 3$.

4.3-8

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

First, let's try the guess $T(n) \le cn^2$. Then, we have

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

We can't proceed any further from the inequality above to conclude $T(n) \le cn^2$.

Alternatively, let us try the guess

$$T(n) \le cn^2 - cn,$$

which subtracts off a lower-order term. Now we have

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

where the last step holds for $c \ge 1/2$.

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.

First,

$$\begin{aligned} T(n) &= 3T(\sqrt{n}) + \lg n \qquad \text{let } m = \lg n \\ T(2^m) &= 3T(2^{m/2}) + m \\ S(m) &= 3S(m/2) + m. \end{aligned}$$

Now we guess $S(m) \le cm^{\lg 3} + dm$,

$$\begin{aligned} S(m) &\le 3\left(c(m/2)^{\lg 3} + d(m/2)\right) + m \\ &\le cm^{\lg 3} + (\frac{3}{2}d + 1)m \qquad (d \le -2) \\ &\le cm^{\lg 3} + dm. \end{aligned}$$

Then we guess $S(m) \ge cm^{\lg 3} + dm$,

$$\begin{aligned} S(m) &\ge 3\left(c(m/2)^{\lg 3} + d(m/2)\right) + m \\ &\ge cm^{\lg 3} + (\frac{3}{2}d + 1)m \qquad (d \ge -2) \\ &\ge cm^{\lg 3} + dm. \end{aligned}$$

Thus,

$$\begin{aligned} S(m) &= \Theta(m^{\lg 3}) \\ T(n) &= \Theta(\lg^{\lg 3} n). \end{aligned}$$

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.

- The subproblem size for a node at depth i is $n/2^i$.

  Thus, the tree has $\lg n + 1$ levels and $3^{\lg n} = n^{\lg 3}$ leaves.

  The total cost over all nodes at depth i, for $i = 0, 1, 2, \ldots, \lg n - 1$, is $3^i(n/2^i) = (3/2)^i n$.

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

- We guess $T(n) \le cn^{\lg 3} - dn$,

$$\begin{aligned} T(n) &= 3T(\lfloor n/2 \rfloor) + n \\ &\le 3 \cdot (c(n/2)^{\lg 3} - d(n/2)) + n \\ &= (3/2^{\lg 3})cn^{\lg 3} - (3d/2)n + n \\ &= cn^{\lg 3} + (1 - 3d/2)n, \end{aligned}$$

  where the last step holds for $d \ge 2$.

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.

- The subproblem size for a node at depth i is $n/2^i$.

  Thus, the tree has $\lg n + 1$ levels and $1^{\lg n} = 1$ leaf.

  The total cost over all nodes at depth i, for $i = 0, 1, 2, \ldots, \lg n - 1$, is $1^i(n/2^i)^2 = (1/4)^i n^2$.

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

- We guess $T(n) \le cn^2$,

$$\begin{aligned} T(n) &\le c(n/2)^2 + n^2 \\ &= cn^2/4 + n^2 \\ &= (c/4 + 1)n^2 \\ &\le cn^2, \end{aligned}$$

  where the last step holds for $c \ge 4/3$.

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.

- The subproblem size for a node at depth i is $n/2^i$.

  Thus, the tree has $\lg n + 1$ levels and $4^{\lg n} = n^2$ leaves.

  The total cost over all nodes at depth i, for $i = 0, 1, 2, \ldots, \lg n - 1$, is $4^i(n/2^i + 2) = 2^i n + 2 \cdot 4^i$.

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

- We guess $T(n) \le c(n^2 - dn)$,

$$\begin{aligned} T(n) &= 4T(n/2 + 2) + n \\ &\le 4c[(n/2 + 2)^2 - d(n/2 + 2)] + n \\ &= 4c(n^2/4 + 2n + 4 - dn/2 - 2d) + n \\ &= cn^2 + 8cn + 16c - 2cdn - 8cd + n \\ &= cn^2 - cdn + 8cn + 16c - cdn - 8cd + n \\ &= c(n^2 - dn) - (cd - 8c - 1)n - (d - 2) \cdot 8c \\ &\le c(n^2 - dn), \end{aligned}$$

  where the last step holds for $cd - 8c - 1 \ge 0$.

4.4-4

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

- The subproblem size for a node at depth i is $n - i$.

  Thus, the tree has $n + 1$ levels ($i = 0, 1, 2, \ldots, n$) and 2^n leaves.

  The total cost over all nodes at depth i, for $i = 0, 1, 2, \ldots, n - 1$, is 2^i.

  The n-th level has 2^n leaves each with cost $\Theta(1)$, so the total cost of the n-th level is $\Theta(2^n)$.

  Adding the costs of all the levels of the recursion tree we get the following:

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

- We guess $T(n) \le c2^n - d$,

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

  Where the last step holds for $d \ge 1$. Thus $T(n) = O(n^2)$.

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.

This is a curious one. The tree makes it look like it is exponential in the worst case. The tree is not full (not a complete binary tree of height n), but it is not polynomial either. It's easy to show $O(2^n)$ and $\Omega(n^2)$.

To justify that this is a pretty tight upper bound, we'll show that we can't have any other choice. If we have that $T(n) \le cn^k$, when we substitue into the recurrence, the new coefficient for n^k can be as high as $c(1 + \frac{1}{2^k})$ which is bigger than c regardless of how we choose the value c.

- We guess $T(n) \le c2^n - 4n$,

$$\begin{aligned} T(n) &\le c2^{n-1} - 4(n - 1) + c2^{n/2} - 4n/2 + n \\ &= c(2^{n-1} + 2^{n/2}) - 5n + 4 \qquad (n \ge 1/4) \\ &\le c(2^{n-1} + 2^{n/2}) - 4n \qquad (n \ge 2) \\ &= c(2^{n-1} + 2^{n-1}) - 4n \\ &\le c2^n - 4n \\ &= O(2^n). \end{aligned}$$

- We guess $T(n) \ge cn^2$,

$$\begin{aligned} T(n) &\ge c(n - 1)^2 + c(n/2)^2 + n \\ &= cn^2 - 2cn + c + cn^2/4 + n \\ &= (5/4)cn^2 + (1 - 2c)n + c \\ &\ge cn^2 + (1 - 2c)n + c \qquad (c \le 1/2) \\ &\ge cn^2 \\ &= \Omega(n^2). \end{aligned}$$

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 the recursion tree.

We know that the cost at each level of the tree is cn by examining the tree in figure 4.6. To find a lower bound on the cost of the algorithm, we need a lower bound on the height of the tree.

The shortest simple path from root to leaf is found by following the leftest child at each node. Since we divide by 3 at each step, we see that this path has length $\log_3 n$. Therefore, the cost of the algorithm is

$$cn(\log_3 n + 1) \ge cn \log_3 n = \frac{c}{\log_3} n \log n = \Omega(n \log n).$$

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 answer with the substitution method.

- The subproblem size for a node at depth i is $n/2^i$.

  Thus, the tree has $\lg n + 1$ levels and $4^{\lg n} = n^{\lg 4} = n^2$ leaves.

  The total cost over all nodes at depth i, for $i = 0, 1, 2, \ldots, \lg n - 1$, is $4^i(cn/2^i) = 2^i cn$.

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

- For $O(n^2)$, we guess $T(n) \le dn^2 - cn$,

$$\begin{aligned} T(n) &\le 4d(n/2)^2 - 4c(n/2) + cn \\ &= dn^2 - cn. \end{aligned}$$

- For $\Omega(n^2)$, we guess $T(n) \ge dn^2 - cn$,

$$\begin{aligned} T(n) &\ge 4d(n/2)^2 - 4c(n/2) + cn \\ &= dn^2 - cn. \end{aligned}$$

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 $a \ge 1$ and $c > 0$ are constants.

- The tree has $n/a + 1$ levels.

  The total cost over all nodes at depth i, for $i = 0, 1, 2, \ldots, n/a - 1$, is $c(n - ia)$.

$$\begin{aligned} T(n) &= \sum_{i=0}^{n/a} c(n - ia) + (n/a)ca \\ &= \sum_{i=0}^{n/a} cn - \sum_{i=0}^{n/a} cia + (n/a)ca \\ &= cn^2/a - \Theta(n) + \Theta(n) \\ &= \Theta(n^2). \end{aligned}$$

- For $O(n^2)$, we guess $T(n) \le cn^2$,

$$\begin{aligned} T(n) &\le c(n - a)^2 + ca + cn \\ &\le cn^2 - 2can + ca + cn \\ &\le cn^2 - c(2an - a - n) \qquad (a > 1/2, n > 2a) \\ &\le cn^2 - cn \\ &\le cn^2 \\ &= \Theta(n^2). \end{aligned}$$

- For $\Omega(n^2)$, we guess $T(n) \ge cn^2$,

$$
\begin{aligned}
T(n) &\geq c(n-a)^2 + ca + cn \\
&\geq cn^2 - 2acn + ca + cn \\
&\geq cn^2 - c(2an - a - n) \qquad (a < 1/2, n > 2a) \\
&\geq cn^2 + cn \\
&\geq cn^2 \\
&= \Theta(n^2).
\end{aligned}
$$

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.

We can assume that $0 < \alpha \leq 1/2$, since otherwise we can let $\beta = 1 - \alpha$ and solve it for β.

Thus, the depth of the tree is $\log_{1/\alpha} n$ and each level costs cn. And let's guess that the leaves are $\Theta(n)$,

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{\log_{1/\alpha} n} cn + \Theta(n) \\
&= cn \log_{1/\alpha} n + \Theta(n) \\
&= \Theta(n \lg n).
\end{aligned}
$$

We can also show $T(n) = \Theta(n \lg n)$ by substitution.

To prove the upper bound, we guess that $T(n) \leq dn \lg n$ for a constant $d > 0$,

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

where the last step holds when $d \geq \frac{-c}{\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)}$.

We can achieve this result by solving the inequality

$$
\begin{aligned}
& dn \lg n + dn(\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)) + cn \leq dn \lg n \\
\Rightarrow\ & dn(\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)) + cn \leq 0 \\
\Rightarrow\ & d(\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)) \leq -c \\
\Rightarrow\ & d \geq \frac{-c}{\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)},
\end{aligned}
$$

To prove the lower bound, we guess that $T(n) \geq dn \lg n$ for a constant $d > 0$,

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

where the last step holds when $0 < d \leq \frac{-c}{\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)}$.

We can achieve this result by solving the inequality

$$
\begin{aligned}
& dn \lg n + dn(\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)) + cn \geq dn \lg n \\
\Rightarrow\ & dn(\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)) + cn \geq 0 \\
\Rightarrow\ & d(\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)) \geq -c \\
\Rightarrow\ & 0 < d \leq \frac{-c}{\alpha \lg \alpha + (1-\alpha)\lg(1-\alpha)},
\end{aligned}
$$

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

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. $\Theta(n^{\log_4 2}) = \Theta(\sqrt{n})$.

b. $\Theta(n^{\log_4 2} \lg n) = \Theta(\sqrt{n} \lg n)$.

c. $\Theta(n)$.

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

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 has running time of $\Theta(n^{\lg 7})$.

The largest integer a such that $\log_4 a < \lg 7$ is $a = 48$.

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.)

$$
\begin{aligned}
& a = 1, b = 2, \\
& f(n) = \Theta(n^{\lg 1}) = \Theta(1), \\
& T(n) = \Theta(\lg n).
\end{aligned}
$$

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.

With $a = 4$, $b = 2$, we have $f(n) = n^2 \lg n \neq O(n^{2-\epsilon}) \neq \Omega(n^{2+\epsilon})$, so we cannot apply the master method.

We guess $T(n) \leq cn^2 \lg^2 n$, subsituting $T(n/2) \leq c(n/2)^2 \lg^2(n/2)$ into the recurrence yields

$$
\begin{aligned}
T(n) &= 4T(n/2) + n^2 \lg n \\
&\leq 4c(n/2)^2 \lg^2(n/2) + n^2 \lg n \\
&= cn^2 \lg(n/2) \lg n - cn^2 \lg(n/2) \lg 2 + n^2 \lg n \\
&= cn^2 \lg^2 n - cn^2 \lg n \lg 2 - cn^2 \lg(n/2) \lg 2 + n^2 \lg n \\
&= cn^2 \lg^2 n + (1 - c \lg 2) n^2 \lg n - cn^2 \lg(n/2) \lg 2 \qquad (c \geq 1/\lg 2) \\
&\leq cn^2 \lg^2 n - cn^2 \lg(n/2) \lg 2 \\
&\leq cn^2 \lg^2 n.
\end{aligned}
$$

Exercise 4.6-2 is the general case for this.

4.5-5 ⋆

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

$a = 1$, $b = 2$ and $f(n) = n(2 - \cos n)$.

If we try to prove it,

$$
\begin{aligned}
& \frac{n}{2}(2 - \cos\frac{n}{2}) < cn \\
& \frac{1 - \cos(n/2)}{2} < c \\
& 1 - \frac{\cos(n/2)}{2} \leq c.
\end{aligned}
$$

Since $\min \cos(n/2) = -1$, this implies that $c \geq 3/2$. But $c < 1$.

4.6 Proof of the master theorem

4.6-1 ⋆

Give a simple and exact expression for n_j in equation (4.27) for the case in which b is a positive integer instead of an arbitrary real number.

We state that $\forall j \geq 0, n_j = \left\lceil \frac{n}{b^j} \right\rceil$.

Indeed, for $j = 0$ we have from the recurrence's base case that $n_0 = n = \left\lceil \frac{n}{b^0} \right\rceil$.

Now, suppose $n_{j-1} = \left\lceil \frac{n}{b^{j-1}} \right\rceil$ for some $j > 0$. By definition, $n_j = \left\lceil \frac{n_{j-1}}{b} \right\rceil$.

It follows from the induction hypothesis that $n_j = \left\lceil \frac{\left\lceil \frac{n}{b^{j-1}} \right\rceil}{b} \right\rceil$.

Since b is a positive integer, equation (3.4) implies that $\left\lceil \frac{\left\lceil \frac{n}{b^{j-1}} \right\rceil}{b} \right\rceil = \left\lceil \frac{n}{b^j} \right\rceil$.

Therefore, $n_j = \left\lceil \frac{n}{b^j} \right\rceil$.

P.S. n_j is obtained by shifting the base b representation j positions to the right, and adding 1 if any of the j least significant positions are non-zero.

4.6-2 ⋆

Show that if $f(n) = \Theta(n^{\log_b a} \lg^k n)$, where $k \geq 0$, then the master recurrence has solution $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$. For simplicity, confine your analysis to exact powers of b.

$$
\begin{aligned}
g(n) &= \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) \\
f(n/b^j) &= \Theta\left((n/b^j)^{\log_b a} \lg^k(n/b^j)\right) \\
g(n) &= \Theta\left(\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a} \lg^k\left(\frac{n}{b^j}\right)\right) \\
&= \Theta(A) \\
A &= \sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a} \lg^k \frac{n}{b^j} \\
&= n^{\log_b a} \sum_{j=0}^{\log_b n - 1} \left(\frac{a}{b^{\log_b a}}\right)^j \lg^k \frac{n}{b^j} \\
&= n^{\log_b a} \sum_{j=0}^{\log_b n - 1} \lg^k \frac{n}{b^j} \\
&= n^{\log_b a} B \\
\lg^k \frac{n}{d} &= (\lg n - \lg d)^k = \lg^k n + o(\lg^k n) \\
B &= \sum_{j=0}^{\log_b n - 1} \lg^k \frac{n}{b^j} \\
&= \sum_{j=0}^{\log_b n - 1} \left(\lg^k n - o(\lg^k n)\right) \\
&= \log_b n \lg^k n + \log_b n \cdot o(\lg^k n) \\
&= \Theta(\log_b n \lg^k n) \\
&= \Theta(\lg^{k+1} n) \\
g(n) &= \Theta(A) \\
&= \Theta(n^{\log_b a} B) \\
&= \Theta(n^{\log_b a} \lg^{k+1} n).
\end{aligned}
$$

4.6-3 ⋆

Show that case 3 of the master method is overstated, in the sense that the regularity condition $af(n/b) \leq cf(n)$ for some constant $c < 1$ implies that there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$.

$$
\begin{aligned}
& af(n/b) \leq cf(n) \\
\Rightarrow\ & f(n/b) \leq \frac{c}{a} f(n) \\
\Rightarrow\ & f(n) \leq \frac{c}{a} f(bn) \\
& \quad = \frac{c}{a}\left(\frac{c}{a} f(b^2 n)\right) \\
& \quad = \frac{c}{a}\left(\frac{c}{a}\left(\frac{c}{a} f(b^3 n)\right)\right) \\
& \quad = \left(\frac{c}{a}\right)^i f(b^i n) \\
\Rightarrow\ & f(b^i n) \geq \left(\frac{a}{c}\right)^i f(n).
\end{aligned}
$$

Let $n = 1$, then we have

$$
f(b^i) \geq \left(\frac{a}{c}\right)^i f(1) \quad (*).
$$

Let $b^i = n \Rightarrow i = \log_b n$, then substitue back to equation $(*)$,

$$
\begin{aligned}
f(n) &\geq \left(\frac{a}{c}\right)^{\log_b n} f(1) \\
&\geq n^{\log_b \frac{a}{c}} f(1) \\
&\geq n^{\log_b a + \epsilon} \qquad \text{where } \epsilon > 0 \text{ because } \frac{a}{c} > a \text{ (recall that } c < 1) \\
&= \Omega(n^{\log_b a + \epsilon}).
\end{aligned}
$$

Problem 4-1 Recurrence examples

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

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

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

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

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

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

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

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

a. By master theorem, $T(n) = \Theta(n^4)$.

b. By master theorem, $T(n) = \Theta(n)$.

c. By master theorem, $T(n) = \Theta(n^2 \lg n)$.

d. By master theorem, $T(n) = \Theta(n^2)$.

e. By master theorem, $T(n) = \Theta(n^{\lg 7})$.

f. By master theorem, $T(n) = \Theta(\sqrt{n} \lg n)$.

g. Let $d = m \mod 2$,

$$
\begin{aligned}
T(n) &= \sum_{j=1}^{j=n/2} (2j + d)^2 \\
&= \sum_{j=1}^{n/2} 4j^2 + 4jd + d^2 \\
&= \frac{n(n+2)(n+1)}{6} + \frac{n(n+2)d}{2} + \frac{d^2 n}{2} \\
&= \Theta(n^3).
\end{aligned}
$$

Problem 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 subrage that might be accessed by the called procedure. Time $= \Theta(q - p + 1)$ if the subarray $A[p..q]$ 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 problems and n be the size of a subproblem.

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

a.

1. $T(n) = T(n/2) + c = \Theta(\lg n)$. (master method)
2. $\Theta(n \lg n)$.

$$
\begin{aligned}
T(n) &= T(n/2) + cN \\
&= 2cN + T(n/4) \\
&= 3cN + T(n/8) \\
&= \sum_{j=0}^{\lg n - 1} (2^j cN/2^j) \\
&= cN \lg n \\
&= \Theta(n \lg n).
\end{aligned}
$$

3. $T(n) = T(n/2) + cn = \Theta(n)$. (master method)

b.

1. $T(n) = 2T(n/2) + cn = \Theta(n \lg n)$. (master method)
2. $\Theta(n^2)$.

$$\begin{aligned} T(n) &= 2T(n/2) + cn + 2N = 4N + cn + 2c(n/2) + 4T(n/4) \\ &= 8N + 2cn + 4c(n/4) + 8T(n/8) \\ &= \sum_{i=0}^{\lg n - 1} (cn + 2^i N) \\ &= \sum_{i=0}^{\lg n - 1} cn + N \sum_{i=0}^{\lg n - 1} 2^i \\ &= cn \lg n + N \frac{2^{\lg n} - 1}{2 - 1} \\ &= cn \lg n + nN - N = \Theta(nN) \\ &= \Theta(n^2). \end{aligned}$$

3. $\Theta(n \lg n)$.

$$\begin{aligned} T(n) &= 2T(n/2) + cn + 2n/2 \\ &= 2T(n/2) + (c+1)n \\ &= \Theta(n \lg n). \end{aligned}$$

Problem 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. By master theorem, $T(n) = \Theta(n^{\log_8 4})$.

b.

By the recursion-tree method, we can guess that $T(n) = \Theta(n \log_3 \log_3 n)$.

We start by proving the upper bound.

Suppose $k < n \implies T(k) \le ck \log_3 \log_3 k - k$, where we subtract a lower order term to strengthen our induction hypothesis.

It follows that

$$\begin{aligned} T(n) &\le 3(c\frac{n}{3}\log_3 \log_3 \frac{n}{3} - \frac{n}{3}) + \frac{n}{\lg n} \\ &\le cn \log_3 \log_3 n - n + \frac{n}{\lg n} \\ &\le cn \log_3 \log_3 n, \end{aligned}$$

if n is sufficiently large.

The lower bound can proved analogously.

c. By master theorem, $T(n) = \Theta(n^{2.5})$.

d. It is $\Theta(n \lg n)$. The subtraction occurring inside the argument to T won't change the asymptotics of the solution, that is, for large n the division is so much more of a change than the subtraction that it is the only part that matters. once we drop that subtraction, the solution comes by the master theorem.

e. By the same reasoning as part (b), the function is $O(n \lg n)$ and $\Omega(n^{1-\epsilon})$ for every ϵ and so is $\tilde{O}(n)$, see Problem 3-5.

f. We guess $T(n) \le cn$,

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

where the last step holds for $c \ge 8$.

g. Recall that χ_A denotes the indicator function of A. We see that the sum is

$$T(0) + \sum_{j=1}^{n} \frac{1}{j} = T(0) + \int_1^{n+1} \sum_{j=1}^{n+1} \frac{\chi_{[j,j+1)}(x)}{j} dx.$$

Since $\frac{1}{x}$ is monatonically decreasing, we have that for every $i \in \mathbb{Z}^+$,

$$\sup_{x \in (i,i+1)} \sum_{j=1}^{n+1} \frac{\chi_{[j,j+1)}(x)}{j} - \frac{1}{x} = \frac{1}{i} - \frac{1}{i+1} = \frac{1}{i(i+1)}.$$

Our expression for $T(n)$ becomes

$$T(N) = T(0) + \int_1^{n+1} \left(\frac{1}{x} + O(\frac{1}{\lfloor x \rfloor (\lfloor x \rfloor + 1)}) \right) dx.$$

We deal with the error term by first chopping out the constant amount between 1 and 2 and then bound the error term by $O(\frac{1}{x(x-1)})$ which has an anti-derivative (by method of partial fractions) that is $O(\frac{1}{n})$,

$$\begin{aligned} T(N) &= \int_1^{n+1} \frac{dx}{x} + O(\frac{1}{n}) \\ &= \lg n + T(0) + \frac{1}{2} + O(\frac{1}{n}). \end{aligned}$$

This gets us our final answer of $T(n) = \Theta(\lg n)$.

h. We see that we explicity have

$$\begin{aligned} T(n) &= T(0) + \sum_{j=1}^{n} \lg j \\ &= T(0) + \int_1^{n+1} \sum_{j=1}^{n+1} \chi_{[j,j+1)}(x) \lg j dx. \end{aligned}$$

Similarly to above, we will relate this sum to the integral of $\lg x$.

$$\sup_{x \in (i,i+1)} \sum_{j=1}^{n+1} \chi_{[j,j+1)}(x) \lg j - \lg x = \lg(j+1) - \lg j = \lg\left(\frac{j+1}{j}\right).$$

Therefore,

$$\begin{aligned} T(n) &\le \int_1^n \lg(x+2) + \lg x - \lg(x+1) dx \\ &(1 + O(\frac{1}{\lg n}))\Theta(n \lg n). \end{aligned}$$

i. See the approach used in the previous two parts, we will get $T(n) = \Theta(\frac{n}{\lg n})$.

j. Let i be the smallest i so that $n^{\frac{1}{2^i}} < 2$. We recall from a previous problem (3-6.e) that this is $\lg \lg n$ Expanding the recurrence, we have that it is

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

Problem 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***) $\square$ as

$$\begin{aligned} \square(z) &= \sum_{i=0}^{\infty} F_i z^i \\ &= 0 + z + z^2 + 2z^3 + 3z^4 + 5z^5 + 8z^6 + 13z^7 + 21z^8 + \cdots, \end{aligned}$$

where F_i is the ith Fibonacci number.

a. Show that $\square(z) = z + z\square(z) + z^2\square$.

b. Show that

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

where

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

and

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

c. Show that

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

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.

$$\begin{aligned} z + z\square(z) + z^2\square(Z) &= z + z\sum_{i=0}^{\infty} F_i z^i + z^2 \sum_{i=0}^{\infty} F_i z^i \\ &= z + \sum_{i=1}^{\infty} F_{i-1} z^i + \sum_{i=2}^{\infty} F_{i-2} z^i \\ &= z + F_1 z + \sum_{i=2}^{\infty} (F_{i-1} + F_{i-2}) z^i \\ &= z + F_1 z + \sum_{i=2}^{\infty} F_i z^i \\ &= \square(z). \end{aligned}$$

b. Note that $\phi - \hat{\phi} = \sqrt{5}$, $\phi + \hat{\phi} = 1$ and $\phi\hat{\phi} = -1$.

$$\begin{aligned} \square(z) &= \frac{\square(z)(1 - z - z^2)}{1 - z - z^2} \\ &= \frac{\square(z) - z\square(z) - z^2\square(z) - z + z}{1 - z - z^2} \\ &= \frac{\square(z) - \square(z) + z}{1 - z - z^2} \\ &= \frac{z}{1 - z - z^2} \\ &= \frac{z}{1 - (\phi + \hat{\phi})z + \phi\hat{\phi}z^2} \\ &= \frac{z}{(1 - \phi z)(1 - \hat{\phi} z)} \\ &= \frac{\sqrt{5}z}{\sqrt{5}(1 - \phi z)(1 - \hat{\phi} z)} \\ &= \frac{(\phi - \hat{\phi})z + 1 - 1}{\sqrt{5}(1 - \phi z)(1 - \hat{\phi} z)} \\ &= \frac{(1 - \hat{\phi} z) - (1 - \phi z)}{\sqrt{5}(1 - \phi z)(1 - \hat{\phi} z)} \\ &= \frac{1}{\sqrt{5}}\left(\frac{1}{1 - \phi z} - \frac{1}{1 - \hat{\phi} z}\right). \end{aligned}$$

c. We have $\frac{1}{1-x} = \sum_{k=0}^{\infty} x^k$, when $|x| < 1$, thus

$$\begin{aligned} \square(n) &= \frac{1}{\sqrt{5}}\left(\frac{1}{1 - \phi z} - \frac{1}{1 - \hat{\phi} z}\right) \\ &= \frac{1}{\sqrt{5}}\left(\sum_{i=0}^{\infty} \phi^i z^i - \sum_{i=0}^{\infty} \hat{\phi}^i z^i\right) \\ &= \sum_{i=0}^{\infty} \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i) z^i. \end{aligned}$$

d. $\square(z) = \sum_{i=0}^{\infty} \alpha_i z^i$ where $\alpha_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}}$. From this follows that $\alpha_i = F_i$, that is

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

For $i = 1$, $\phi/\sqrt{5} = (\sqrt{5}+5)/10 > 0.5$. For $i > 2$, $|\frac{\hat{\phi}^i}{\sqrt{5}}| < 0.5$.

Problem 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 more than $n/2$ 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 $n/2$ chips are good. Give and solve the recurrence that describes the number of tests.

a. Lets say that there are $g < n/2$ good chips and $n - g$ bad chips.

From this assumption, we can always find a set of good chips G and a set of bad chips B of equal size g since $n - g \ge g$.

Now, assume that chips in B always conspire to fool the professor in the following:

"for any test made by the professor, chips in B declare chips in B as 'good' and chips in G as 'bad'."

Since the chips in G always report correct answers thus there exists symmetric behaviors, it is not possible to distinguish bad chips from good ones.

b.

Generalize the original problem to: "Assume there are more good chips than bad chips."

Algorithm:

1. Pairwise test them, and leave the last one alone if the number of chips is odd.
   - If the report says at least one of them is bad, throw both chips away;
   - otherwise, throw one away from each pair.
2. Recursively find one good chip among the remaining chips. The recursion ends when the number of remaining chips is 1 or 2.
   - If there is only 1 chip left, then it is the good chip we desire.
   - If there are 2 chips left, we make a pairwise test between them. If the report says both are good, we can conclude that both are good chips. Otherwise, one is good and the other is bad and we throw both away. The chip we left alone at step 1 is a good chip.

Explanation:

1. If the number of chips is odd, from assumption we know the number of good chips must be greater than the number of bad chips. We randomly leave one chip alone from the chips, in which good chips are not less than bad chips.
2. Chip pairs that do not say each other is good either have one bad chip or have two bad chips, throwing them away doesn't change the fact that good chips are not less than bad chips.
3. The remaining chip pairs are either both good chips or bad chips, after throwing one chip away in every those pairs, we have reduced the size of the problem to at most half of the original problem size.
4. If the number of good chips is n ($n > 1$) more than that of bad chips, we just throw away the chip we left alone when the number of chips is odd. In this case, the number of good chips is at least one more than that of bad chips, and we

can eventually find a good chip as our algorithm claims.

5. If the number of good chips is exactly one more than that of bad chips, there are 2 cases.
   - We left alone the good chip, and remaining chips are one half good and one half bad. In this case, all the chips will be thrown away eventually. And the chip left alone is the one we desire.
   - We left alone the bad chip, there are more good chips than bad chips in the remaining chips. In this case, we can recursively find a good chip in the remaining chips and the left bad chip will be thrown away at the end.

c. As the solution provided in (b), we can find one good chip in

$$T(n) \le T(\lceil n/2 \rceil) + \lfloor n/2 \rfloor.$$

By the master theorem, we have $T(n) = O(n)$. After finding a good chip, we can identify all good chips with that good chip we just found in $n - 1$ tests, so the total number of tests is

$$O(n) + n - 1 = \Theta(n).$$

Problem 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 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 seperately 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 in 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 its solution is $O(m + n \log m)$.

a. The "only if" part is trivial, it follows form the definition of Monge array.

As for the "if" part, let's first prove that

$$\begin{aligned} &A[i,j] + A[i+1,j+1] \le A[i,j+1] + A[i+1,j] \\ \rightarrow\ &A[i,j] + A[k,j+1] \le A[i,j+1] + A[k,j], \end{aligned}$$

where $i < k$.

Let's prove it by induction. The base case of $k = i + 1$ is given. As for the inductive step, we assume it holds for $k = i + n$ and we want to prove it for $k + 1 = i + n + 1$. If we add the given to the assumption, we get

$$\begin{aligned} &A[i,j] + A[k,j+1] \le A[i,j+1] + A[k,j] && \text{(assumption)} \\ &A[k,j] + A[k+1,j+1] \le A[k,j+1] + A[k+1,j] && \text{(given)} \\ \rightarrow\ &A[i,j] + A[k,j+1] + A[k,j] + A[k+1,j+1] \le A[i,j+1] + A[k,j] + A[k,j+1] + A[k+1,j] \\ \rightarrow\ &A[i,j] + A[k+1,j+1] \le A[i,j+1] + A[k+1,j] \end{aligned}$$

b.

37	23	**24**	32
21	6	7	10
53	34	30	31
32	13	9	6
43	21	15	8

c. Let a_i and b_j be the leftmost minimal elements on rows a and b and let's assume that $i > j$. Then we have

$$A[j,a] + A[i,b] \le A[i,a] + A[j,b].$$

But

$$\begin{aligned} A[j,a] &\ge A[i,a] && (a_i \text{ is minimal}) \\ A[i,b] &\ge A[j,b] && (b_j \text{ is minimal}) \end{aligned}$$

Which implies that

$$\begin{aligned} A[j,a] + A[i,b] &\ge A[i,a] + A[j,b] \\ A[j,a] + A[i,b] &= A[i,a] + A[j,b] \end{aligned}$$

Which in turn implies that either:

$$\begin{aligned} A[j,b] < A[i,b] &\rightarrow A[i,a] > A[j,a] \rightarrow a_i \text{ is not minimal} \\ A[j,b] = A[i,b] &\rightarrow b_j \text{ is not the leftmost minimal} \end{aligned}$$

d. If μ_i is the index of the i-th row's leftmost minimum, then we have

$$\mu_{i-1} \le \mu_i \le \mu_{i+1}.$$

For $i = 2k + 1, k \ge 0$, finding μ_i takes $\mu_{i+1} - \mu_{i-1} + 1$ steps at most, since we only need to compare with those numbers. Thus

$$\begin{aligned} T(m,n) &= \sum_{i=0}^{m/2-1} (\mu_{2i+2} - \mu_{2i} + 1) \\ &= \sum_{i=0}^{m/2-1} \mu_{2i+2} - \sum_{i=0}^{m/2-1} \mu_{2i} + m/2 \\ &= \sum_{i=1}^{m/2} \mu_{2i} - \sum_{i=0}^{m/2-1} \mu_{2i} + m/2 \\ &= \mu_m - \mu_0 + m/2 \\ &= n + m/2 \\ &= O(m+n). \end{aligned}$$

e. The divide time is $O(1)$, the conquer part is $T(m/2)$ and the merge part is $O(m + n)$. Thus,

$$\begin{aligned} T(m) &= T(m/2) + cn + dm \\ &= cn + dm + cn + dm/2 + cn + dm/4 + \cdots \\ &= \sum_{i=0}^{\lg m-1} cn + \sum_{i=0}^{\lg m-1} \frac{dm}{2^i} \\ &= cn \lg m + dm \sum_{i=0}^{\lg m-1} \\ &< cn \lg m + 2dm \\ &= O(n \lg m + m). \end{aligned}$$

5 Probabilistic Analysis and Randomized Algorithms

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.

A total order is a partial order that is a total relation $(\forall a, b \in A : aRb \text{ or } bRa)$. A relation is a partial order if it is reflexive, antisymmetric and transitive.

Assume that the relation is good or better.

- **Reflexive:** This is a bit trivial, but everybody is as good or better as themselves.
- **Transitive:** If A is better than B and B is better than C, then A is better than C.
- **Antisymmetric:** If A is better than B, then B is not better than A.

So far we have a partial order.

Since we assume we can compare any two candidates, then comparison must be a total relation and thus we have a total order.

5.1-2 ⋆

> 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?

As $(b - a)$ could be any number, we need at least $\lceil \lg(b-a) \rceil$ bits to represent the number. We set $\lceil \lg(b-a) \rceil$ as k. Basically, we need to call RANDOM(0, 1) k times. If the number represented by binary is bigger than $b - a$, it's not valid number and we give it another try, otherwise we return that number.

```
RANDOM(a, b)
    range = b - a
    bits = ceil(log(2, range))
    result = 0
    for i = 0 to bits - 1
        r = RANDOM(0, 1)
        result = result + r << i
    if result > range
        return RANDOM(a, b)
    else return a + result
```

The expectation of times of calling procedure RANDOM(a, b) is $\frac{2^k}{b-a}$. RANDOM(0, 1) will be called k times in that procedure.

The expected running time is $\Theta(\frac{2^k}{b-a} \cdot k)$, k is $\lceil \lg(b-a) \rceil$. Considering 2^k is less than $2 \cdot (b - a)$, so the running time is $O(k)$.

5.1-3 ⋆

> 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?

There are 4 outcomes when we call BIASED-RANDOM twice, i.e., 00, 01, 10, 11.

The strategy is as following:

- 00 or 11: call BIASED-RANDOM twice again
- 01: output 0
- 10: output 1

We can calculate the probability of each outcome:

- $\Pr\{00|11\} = p^2 + (1-p)^2$
- $\Pr\{01\} = (1-p)p$
- $\Pr\{10\} = p(1-p)$

Since there's no other way to return a value, it returns 0 and 1 both with probability $1/2$.

The pseudo code is as follow:

```
UNBIASED-RANDOM
    while true
        x = BIASED-RANDOM
        y = BIASED-RANDOM
        if x != y
            return x
```

This algorithm actually uses the equivalence of the probability of occurrence of 01 and 10, and subtly converts the unequal 00 and 11 to 01 and 10, thus eliminating the probability that its probability is not equivalent.

Each iteration is a Bernoulli trial, where "success" means that the iteration does return a value.

We can view each iteration as a Bernoulli trial, where "success" means that the iteration returns a value.

$$\begin{aligned} \Pr\{\text{success}\} &= \Pr\{0 \text{ is returned}\} + \Pr\{1 \text{ is returned}\} \\ &= 2p(1-p). \end{aligned}$$

The expected number of trials for this scenario is $1/(2p(1-p))$. Thus, the expected running time of UNBIASED-RANDOM is $\Theta(1/(2p(1-p)))$.

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 you hire exactly n times?

You will hire exactly one time if the best candidate is at first. There are $(n-1)!$ orderings with the best candidate being at first, so the probability that you hire exactly one time is $\frac{(n-1)!}{n!} = \frac{1}{n}$.

You will hire exactly n times if the candidates are presented in increasing order. There is only an ordering for this situation, so the probability that you hire exactly n times is $\frac{1}{n!}$.

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?

Note that

- Candidate 1 is always hired
- The best candidate (candidate whose rank is n) is always hired
- If the best candidate is candidate 1, then that's the only candidate hired.

In order for HIRE-ASSISTANT to hire exactly twice, candidate 1 should have rank i, where $1 \le i \le n-1$, and all candidates whose ranks are $i+1, i+2, \ldots, n-1$ should be interviewed after the candidate whose rank is n (the best candidate).

Let E_i be the event in which candidate 1 have rank i, so we have $P(E_i) = 1/n$ for $1 \le i \le n$.

Our goal is to find for $1 \le i \le n-1$, given E_i occurs, i.e., candidate 1 has rank i, the candidate whose rank is n (the best candidate) is the first one interviewed out of the $n - i$ candidates whose ranks are $i+1, i+2, \ldots, n$.

So,

$$\sum_{i=1}^{n-1} P(E_i) \cdot \frac{1}{n-i} = \sum_{i=1}^{n-1} \frac{1}{n} \cdot \frac{1}{n-i}.$$

5.2-3

> Use indicator random variables to compute the expected value of the sum of n dice.

Expectation of a single dice X_i is

$$\begin{aligned} E[X_k] &= \sum_{i=1}^{6} i \Pr\{X_k = i\} \\ &= \frac{1+2+3+4+5+6}{6} \\ &= \frac{21}{6} \\ &= 3.5. \end{aligned}$$

As for multiple dices,

$$\begin{aligned} E\Bigg[\sum_{i = 1}^n X_i\Bigg] &= \sum_{i = 1}^n \text E[X_i] \\ &= \sum_{i = 1}^n 3.5 \\ &= 3.5 \cdot n. \end{aligned}$$

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 hat?

Let X be the number of customers who get back their own hat and X_i be the indicator random variable that customer i gets his hat back. The probability that an individual gets his hat back is $\frac{1}{n}$. Thus we have

$$E\Bigg[\sum_{i = 1}^n X_i\Bigg] = \sum_{i = 1}^n E[X_i] = \sum_{i = 1}^n \frac{1}{n} = 1.$$

5.2-5

> Let $A[1..n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an ***inversion*** of A. (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 $X_{i,j}$ for $i < j$ be the indicator of $A[i] > A[j]$. We have that the expected number of inversions

$$\begin{aligned} E\left[\sum_{i<j} X_{i,j}\right] &= \sum_{i<j} E[X_{i,j}] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{A[i] > A[j]\} \\ &= \frac{1}{2}\sum_{i=1}^{n-1} n - i \\ &= \frac{n(n-1)}{2} - \frac{n(n-1)}{4} \\ &= \frac{n(n-1)}{4}. \end{aligned}$$

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.

Modify the algorithm by unrolling the $i = 1$ case.

```
swap(A[1], A[RANDOM(1, n)])
for i = 2 to n
    swap(A[i], A[RANDOM(i, n)])
```

Modify the proof of Lemma 5.5 by starting with $i = 2$ instead of $i = 1$. This resolves the issue of 0-permutations.

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
>         swap A[i] with A[RANDOM(i + 1, n)]
> ```
>
> Does this code do what Professor Kelp intends?

The code does not do what he intends. Suppose $A = [1, 2, 3]$. If the algorithm worked as proposed, then with nonzero probability the algorithm should output $[3, 2, 1]$. On the first iteration we swap $A[1]$ with either $A[2]$ or $A[3]$. Since we want $[3, 2, 1]$ and will never again alter $A[1]$, we must necessarily swap with $A[3]$. Now the current array is $[3, 2, 1]$. On the second (and final) iteration, we have no choice but to swap $A[2]$ with $A[3]$, so the resulting array is $[3, 1, 2]$. Thus, the procedure cannot possibly be producing random non-identity permutations.

5.3-3

> Suppose that instead of swapping element $A[i]$ with a random element from the subarray $A[i..n]$, we swapped it with a random element from anywhere in the array:
>
> ```
> PERMUTE-WITH-ALL(A)
>     n = A.length
>     for i = 1 to n
>         swap A[i] with A[RANDOM(1, n)]
> ```
>
> Does this code produce a uniform random permutation? Why or why not?

Consider the case of $n = 3$ in running the algorithm, three RID choices will be made, and so you'll end up having 27 possible end states each with equal probability. There are $3! = 6$ possible orderings, these should appear equally often, but this can't happen because 6 does not divide 27.

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
>         dest = i + offset
>         if dest > n
>             dest = dest - n
>         B[dest] = A[i]
>     return B
> ```
>
> 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.

Fix a position j and an index i. We'll show that the probability that $A[i]$ winds up in position j is $1/n$. The probability $B[j] = A[i]$ is the probability that $dest = j$, which is the probability that $i + offset$ or $i + offset - n$ is equal to j, which is $1/n$.

This algorithm can't possibly return a random permutation because it doesn't change the relative positions of the elements; it merely cyclically permutes the whole permutation.

For instance, suppose $A = [1, 2, 3]$,

- if $offset = 1$, $B = [3, 1, 2]$,
- if $offset = 2$, $B = [2, 3, 1]$,
- if $v = 3$, $B = [1, 2, 3]$.

Thus, the algorithm will never produce $B = [1, 3, 2]$, so the resulting permutation cannot be uniformly random.

5.3-5 ⋆

> 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 $\Pr\{j\}$ be the probability that the element with index j is unique. If there are n^3 elements, then the $\Pr\{j\} = 1 - \frac{j-1}{n^3}$.

$$\begin{aligned} \Pr\{1 \cap 2 \cap 3 \cap \ldots\} &= \Pr\{1\} \cdot \Pr\{2 \mid 1\} \cdot \Pr\{3 \mid 1 \cap 2\} \cdots \\ &= 1(1 - \frac{1}{n^3})(1 - \frac{2}{n^3})(1 - \frac{3}{n^3}) \cdots \\ &\geq 1(1 - \frac{n}{n^3})(1 - \frac{n}{n^3})(1 - \frac{n}{n^3}) \cdots \\ &\geq (1 - \frac{1}{n^2})^n \\ &\geq 1 - \frac{1}{n}, \end{aligned}$$

where the last step holds for $(1 - x)^n \geq 1 - nx$.

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.

```
PERMUTE-BY-SORTING(A)
    let P[1..n] be a new array
    for i = 1 to n
        P[i] = i
    for i = 1 to n
        swap P[i] with P[RANDOM(i, n)]
```

5.3-7

> Suppose we want to create a ***random sample*** of the set $\{1, 2, 3, \ldots, n\}$, that is, an m-element subset S, where $0 \leq m \leq n$, such that each m-subset is equally likely to be created. One way would be to set $A[i] = i$ for $i = 1, 2, 3, \ldots, n$, call RANDOMIZE-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 much 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)
>     if m == 0
>         return ∅
>     else S = RANDOM-SAMPLE(m - 1, n - 1)
>         i = RANDOM(1, n)
>         if i ∈ S
>             S = S ∪ {n}
>         else S = S ∪ {i}
>         return S
> ```

We prove that it produces a random m subset by induction on m. It is obviously true if $m = 0$ as there is only one size m subset of $[n]$. Suppose S is a uniform $m - 1$ subset of $n - 1$, that is, $\forall j \in [n - 1]$, $\Pr[j \in S] = \frac{m-1}{n-1}$.

If we let S' denote the returned set, suppose first $j \in [n - 1]$,

$$\begin{aligned} \Pr[j \in S'] &= \Pr[j \in S] + \Pr[j \notin S \wedge i = j] \\ &= \frac{m-1}{n-1} + \Pr[j \notin S] \Pr[i = j] \\ &= \frac{m-1}{n-1} + \left(1 - \frac{m-1}{n-1}\right)\frac{1}{n} \\ &= \frac{n(m-1) + n - m}{(n-1)n} \\ &= \frac{nm - m}{(n-1)n} = \frac{m}{n}. \end{aligned}$$

Since the constructed subset contains each of $[n - 1]$ with the correct probability, it must also contain n with the correct probability because the probabilities sum to 1.

5.4 Probabilistic analysis and further uses of indicator random variables

5.4-1

> How many people must there be in a room before the probability that someone has the same birthday as you do is at least $1/2$? How many people must there be before the probability that at least two people have a birthday on July 4 is greater than $1/2$?

The probability of a person not having the same birthday as me is $(n - 1)/n$. The probability of k people not having the same birthday as me is that, squared. We apply the same approach as the text - we take the complementary event and solve it for k,

$$\begin{aligned} 1 - \left(\frac{n-1}{n}\right)^k &\geq \frac{1}{2} \\ \left(\frac{n-1}{n}\right)^k &\leq \frac{1}{2} \\ k \lg\left(\frac{n-1}{n}\right) &\geq \lg\frac{1}{2} \\ k = \frac{\log(1/2)}{\log(364/365)} &\approx 253. \end{aligned}$$

As for the other question,

$$\begin{aligned} \Pr\{2 \text{ born on Jul 4}\} &= 1 - \Pr\{1 \text{ born on Jul 4}\} - \Pr\{0 \text{ born on Jul 4}\} \\ &= 1 - \frac{k}{n}\left(\frac{n-1}{n}\right)^{k-1} - \left(\frac{n-1}{n}\right)^k \\ &= 1 - \left(\frac{n-1}{n}\right)^{k-1}\left(\frac{n+k-1}{n}\right). \end{aligned}$$

Writing a Ruby programme to find the closest integer, we get 115.

5.4-2

> Suppose that we toss balls into b bins until some bin contains two balls. Each toss is independent, and each ball is equally likely to end up in any bin. What is the expected number of ball tosses?

This is just a restatement of the birthday problem. I consider this all that needs to be said on this subject.

5.4-3 ⋆

> For the analysis of the birthday paradox, is it important that the birthdays be mutually independent, or is pairwise independence sufficient? Justify your answer.

Pairwise independence is enough. It's sufficient for the derivation after (5.6).

5.4-4 ⋆

> How many people should be invited to a party in order to make it likely that there are three people with the same birthday?

The answer is 88. I reached it by trial and error. But let's analyze it with indicator random variables.

Let X_{ijk} be the indicator random variable for the event of the people with indices i, j and k have the same birthday. The probability is $1/n^2$. Then,

$$\begin{aligned} E[X] &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \sum_{k=j+1}^{n} X_{ijk} \\ &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \sum_{k=j+1}^{n} \frac{1}{n^2} \\ &= \binom{n}{3} \frac{1}{n^2} \\ &= \frac{k(k-1)(k-2)}{6n^2}. \end{aligned}$$

Solving this yields 94. It's a bit more, but again, indicator random variables are approximate.

Finding more commentary online is tricky.

5.4-5 *

> What is the probability that a k-string over a set of size n forms a k-permutation? How does this question relate to the birthday paradox?

$$\begin{aligned} \Pr\{\text{k-perm in n}\} &= 1 \cdot \frac{n-1}{n} \cdot \frac{n-2}{n} \cdots \frac{n-k+1}{n} \\ &= \frac{n!}{n^k(n-k)!}. \end{aligned}$$

This is the complementary event to the birthday problem, that is, the chance of k people have distinct birthdays.

5.4-6 *

> Suppose that n balls are tossed into n bins, where each toss is independent and the ball is equally likely to end up in any bin. What is the expected number of empty bins? What is the expected number of bins with exactly one ball?

Let X_i be the indicator variable that bin i is empty after all balls are tossed and X be the random variable that gives the number of empty bins. Thus we have

- [x] = \sum_{i = 1}^n E[X_i] = \sum_{i = 1}^n \bigg(\frac{n - 1}{n}\bigg)^n = n\bigg(\frac{n - 1}{n}\bigg)^n.$$

Let X_i be the indicator variable that bin i contains exactly 1 ball after all balls are tossed and X be the random variable that gives the number of bins containing exactly 1 ball. Thus we have

- [x] = \sum_{i = 1}^n E[X_i] = \sum_{i = 1}^n \binom{n}{1}\bigg(\frac{n - 1}{n}\bigg)^{n - 1} \frac{1}{n} = n\bigg(\frac{n - 1}{n}\bigg)^{n - 1},$$

because we need to choose which toss will go into bin i, then multiply by the probability that that toss goes into that bin and the remaining $n - 1$ tosses avoid it.

5.4-7 *

> Sharpen the lower bound on streak length by showing that in n flips of a fair coin, the probability is less than $1/n$ that no streak longer than $\lg n - 2 \lg\lg n$ consecutive heads occurs.

We split up the n flips into n/s groups where we pick $s = \lg(n) - 2\lg(\lg(n))$. We will show that at least one of these groups comes up all heads with probability at least $\frac{n-1}{n}$. So, the probability the group starting in position i comes up all heads is

$$\Pr(A_{i,\lg n - 2\lg(\lg n)}) = \frac{1}{2^{\lg n - 2\lg(\lg n)}} = \frac{\lg n^2}{n}.$$

Since the groups are based of of disjoint sets of IID coin flips, these probabilities are independent. so,

$$\begin{aligned} \Pr(\bigwedge \neg A_{i,\lg n - 2\lg(\lg n)}) &= \prod_i \Pr(\neg A_{i,\lg n - 2\lg(\lg n)}) \\ &= \left(1 - \frac{\lg n^2}{n}\right)^{\frac{n}{\lg n - 2\lg(\lg n)}} \\ &\le e^{-\frac{\lg n^2}{\lg n - 2\lg(\lg n)}} \\ &= \frac{1}{n} e^{\frac{-2\lg(\lg n)\lg n}{\lg n - 2\lg(\lg n)}} \\ &= n^{-1 - \frac{2\lg(\lg n)}{\lg n - 2\lg(\lg n)}} \\ &< n^{-1}. \end{aligned}$$

Showing that the probability that there is no run of length at least $\lg n - 2\lg(\lg n)$ to be $< \frac{1}{n}$.

Problem 5-1 Probabilstic counting

> With a b-bit counter, we can ordinarily only count up to $2^b - 1$. With R. Morris'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 that 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 $1/(n_{i+1} - n_i)$, and it leaves the counter unchanged with probability $1 - 1/(n_{i+1} - n_i)$.
>
> If we select $n_i = i$ for all $i \ge 0$, 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 that 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 $i \ge 0$. Estimate the variance in the value represented by the register after n INCREMENT operations have been performed.

a. To show that the expected value represented by the counter after n INCREMENT operations have been performed is exactly n, we can show that each expected increment represented by the counter is 1.

Assume the initial value of the counter is i, increasing the number represented from n_i to n_{i+1} is with a probability of $\frac{1}{n_{i+1} - n_i}$ and leaving the value not changed otherwise.

The expected increase:

$$\frac{n_{i+1} - n_i}{n_{i+1} - n_i} = 1.$$

b. For this choice of n_i , we have that at each increment operation, the probability that we change the value of the counter is $\frac{1}{100}$. Since this is a constant with respect to the current value of the counter i, we can view the final result as a binomial distribution with a p value of 0.01. Since the variance of a binomial distribution is $np(1 - p)$, and we have that each success is worth 100 instead, the variance is going to be equal to $0.99n$.

Problem 5-2 Searching an unsorted array

> The problem examines three algorithms for searching for a value x in an unsorted array A consisting for 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 $k \ge 1$ 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 search 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 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 $k \ge 1$ 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 permuting array.
>
> **h.** Letting k be the number of indices i such that $A[i] = x$, give the worst-case and expected running time of SCRAMBLE-SEARCH for the cases in which $k = 0$ and $k = 1$. Generalizing your solution to handle the case in which $k \ge 1$.
>
> **i.** Which of the three searching algorithms would you use? Explain your answer.

a.

```
RANDOM-SEARCH(x, A, n)
    v = Ø
    while |Ø| != n
        i = RANDOM(1, n)
        if A[i] = x
            return i
        else
            v = v ∩ i
    return NIL
```

v can be implemented in multiple ways: a hash table, a tree or a bitmap. The last one would probably perform best and consume the least space.

b. RANDOM-SEARCH is well-modelled by Bernoulli trials. The expected number of picks is n.

c. In similar fashion, the expected number of picks is n/k.

d. This is modelled by the balls and bins problem, explored in section 5.4.2. The answer is $n(\ln n + O(1))$.

e. The worst-case running time is n. The average-case is $(n + 1)/2$ (obviously).

f. The worst-case running time is $n - k + 1$. The average-case running time is $(n + 1)/(k + 1)$. Let X_i be an indicator random variable that the ith element is a match. $\Pr\{X_i\} = 1/(k + 1)$. Let Y be an indicator random variable that we have found a match after the first $n - k + 1$ elements ($\Pr\{Y\} = 1$). Thus,

$$\begin{aligned} E[X] &= E[X_1 + X_2 + \ldots + X_{n-k} + Y] \\ &= 1 + \sum_{i=1}^{n-k} E[X_i] = 1 + \frac{n-k}{k+1} \\ &= \frac{n+1}{k+1}. \end{aligned}$$

g. Both the worst-case and average case is n.

h. It's the same as DETERMINISTIC-SEARCH, only we replace "average-case" with "expected".

i. Definitelly DETERMINISTIC-SEARCH. SCRAMBLE-SEARCH gives better expected results, but for the cost of randomly permuting the array, which is a linear operation. In the same time we could have scanned the full array and reported a result.