

Introduction to Algorithms: Third Edition
Solutions

Alexander Novotny

February 23, 2021

Contents

Introduction	ii
Notation	ii
Solutions	iii
1 The Role of Algorithms in Computing	iii
1.1 Algorithms	iii
1.2 Algorithms as a Technology	iv
Problems	vi
2 Getting Started	vi
2.1 Insertion Sort	vi
2.2 Analyzing algorithms	x
2.3 Designing algorithms	xi
Problems	xv
3 Growth of Functions	xxii
3.1 Asymptotic Notation	xxii
3.2 Standard notations and common functions	xxiii
4 Divide-and-Conquer	xxvi
4.1 The maximum-subarray problem	xxvi
4.2 Strassen's algorithm for matrix multiplication	xxvii
Appendix	xxx
A Summations	xxx
A.1 Summation formulas and properties	xxx
A.2 Bounding Summations	xxxiii
Problems	xxxvii

Introduction

This is a list of solutions to the third edition of *Introduction to Algorithms* by Thomas H Cormen, Charles E Leiserson, Ronald L. Rivest, and Clifford Stein. The solutions are all my own, done in my free time as a hobby. As such, the solution presented for problem may be only *one* of the solutions to that problem.

Notation

The notation used in this document vary based on my own personal preferences to the notation used in the book. This is especially true for mathmatic notation not introduced by the book. As such, a list of different notation is available below.

Solutions	Book	Definition
$\mathbb{E}[X]$	$\mathbf{E}[X]$	Expected Value of X
$\mathcal{P}(X)$	$\Pr\{X\}$	Probability of X
$\mathcal{O}(f(x))$	$O(f(x))$	Big-O of $f(x)$
φ	ϕ	Golden Ratio

Solutions

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.

Answer. A company might keep a database of all customers who have purchased products or services from them. At the end of the year, this company might offer a certain sale to its customers, but can only afford to offer a certain number of these sales. They would prefer to offer them to their best customers first, but if those customers aren't interested in accepting the deal, they would offer it to the next best customer. This would require sorting the customers based on how much money they have spent.

A game theorist formulates a real-world problem (such as bidding for project against rival companies) as a game. The company would like to know whether a particular payoff is possible if they have a particular mixed strategy. The game theorist knows that the possible payoffs for this game when using mixed strategies is the convex hull of the payoffs possible when only using non-mixed strategies, which are known. They would simply compute this convex hull, and check whether or not the given payout is contained within.

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

Answer. Storage space required and power used could be useful measures.

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

Answer. Linked lists are useful when the length of the list changes often, as new links can be inserted and removed without re-allocating every other link. However, it can take a long time to reach a particular link, as other links may have to be traversed to reach it. As well, the construction requires storing elements in the heap (instead of the stack), which is slower to access.

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

Answer. Both problems require the use of a “road map” and minimize the distance traveled in completing some task. In the shortest-path problem, we don't require that every destination is visited (and, in fact, they probably aren't), but we do require this of the traveling salesman problem. This means that adding additional roads and destinations to the map might not change the shortest path (and it won't, unless this introduces a shortcut), but will always change the solution to the traveling salesman problem.

- 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.

Answer. At the end of an Olympic event, a list of competitors in that event each have a certain number of points. Medals are given to the three competitors who have more points than anyone else. The problem of finding these three competitors is exact - if someone did better than everyone else and did not get a medal, the problem would not be solved appropriately.

If we would analyze the traveling salesman problem talked about in the chapter, however, we would notice that the exact solution isn't needed. It's worth finding an approximate solution, as the amount of distance saved over an entire route could be tens or hundreds of miles, but the difference between an approximate solution and the exact solution would only be a few miles (the cost of which being negligible).

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.

Answer. Steam, a digital retailer of video games and other software, needs algorithmic content. The platform contains tens of thousands of games and pieces of software, and if they left it up to the customer to find the games that they want, many games would go unnoticed. This means the customer wouldn't get to play games they might want to play, and Steam doesn't get to see the profit from those users buying those games. So instead, Steam has an algorithm which takes as input that customer's previous purchases and playtime and shows them games that they may be interested in.

- 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?

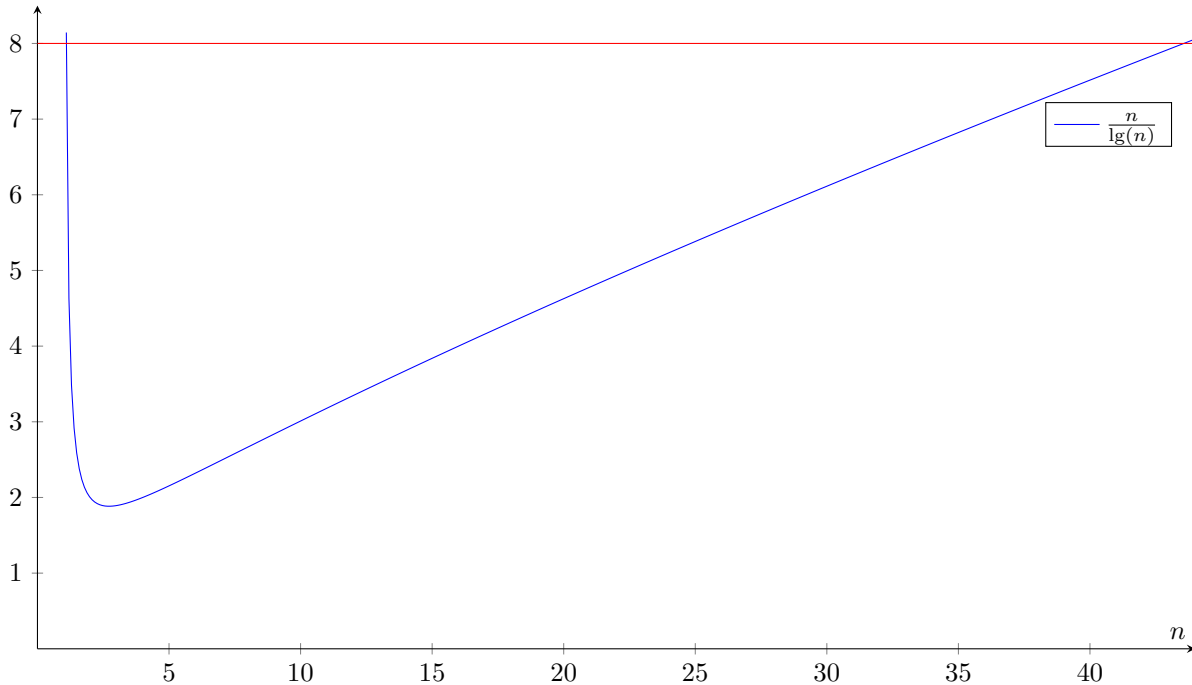
Answer. Note that n is positive. We wish to have

$$8n^2 < 64n \lg(n) \quad \Leftarrow$$

$$n < 8 \lg(n) \quad \Leftarrow$$

$$\frac{n}{\lg(n)} < 8.$$

This can't be solved algebraically, so we analyze the graph and approximate solutions:



Then, we can verify:

$$\begin{aligned} \lim_{x \rightarrow 1^+} \frac{x}{\lg(x)} &= \infty, \\ \frac{2}{\lg(2)} &= 2, \\ \frac{43}{\lg(43)} &\approx 7.92, \\ \frac{44}{\lg(44)} &\approx 8.06. \end{aligned}$$

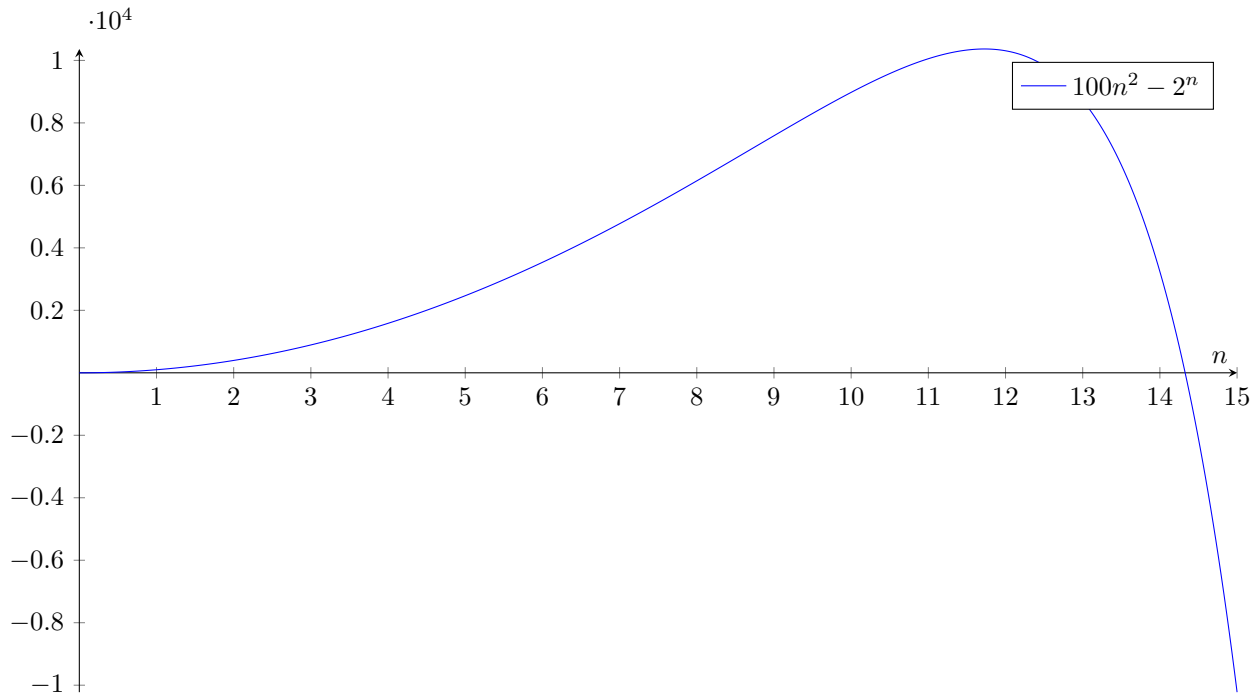
As well, we can check the in-between values by analyzing critical points. We have

$$\frac{d}{dx} \left[\frac{x}{\lg(x)} \right] = \ln(2) \frac{\ln(x) - 1}{\ln^2(x)},$$

which vanishes or is undefined only when $x = 1, e$ and $\frac{e}{\lg(e)} \approx 1.88 < 8$. Therefore insertion sort beats merge sort for all $2 \leq n \leq 43$.

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?

Answer. We wish to find the first value of n such that $100n^2 < 2^n$. Similarly to above, this is unsolvable algebraically, so we again refer to a graph:



We can then verify that $100(14)^2 = 19,600 > 16,384 = 2^{14}$ and $100(15)^2 = 22,500 < 32,768 = 2^{15}$. Therefore, 15 is the smallest value of n that satisfies the given conditions.

Problems

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

	1 second	1 minute	1 hour	1 day	1 month	1 year	1 century
$\lg(n)$	$9.9 \times 10^{301,029}$	$5.5 \times 10^{18,061,799}$
\sqrt{n}	1.0×10^{12}	3.6×10^{15}	1.3×10^{19}	7.5×10^{21}	6.7×10^{24}	9.9×10^{26}	9.9×10^{30}
n	1.0×10^6	6.0×10^7	3.6×10^9	8.6×10^{10}	2.6×10^{12}	3.2×10^{13}	3.2×10^{15}
$n \lg(n)$	6.3×10^4	2.8×10^6	1.3×10^8	2.8×10^9	7.2×10^{10}	8.0×10^{11}	6.7×10^{13}
n^2	1.0×10^3	7.7×10^3	6.0×10^4	3.0×10^5	1.6×10^6	5.6×10^6	5.6×10^7
n^3	1.0×10^2	3.9×10^2	1.5×10^3	4.4×10^3	1.3×10^4	3.1×10^4	1.5×10^5
2^n	1.9×10^1	2.5×10^1	3.1×10^1	3.6×10^1	4.1×10^1	4.4×10^1	5.1×10^1
$n!$	9	1.1×10^1	1.2×10^1	1.3×10^1	1.5×10^1	1.6×10^1	1.6×10^1

Table 1: The rest of the first row has been left out, as the numbers are too large to consider

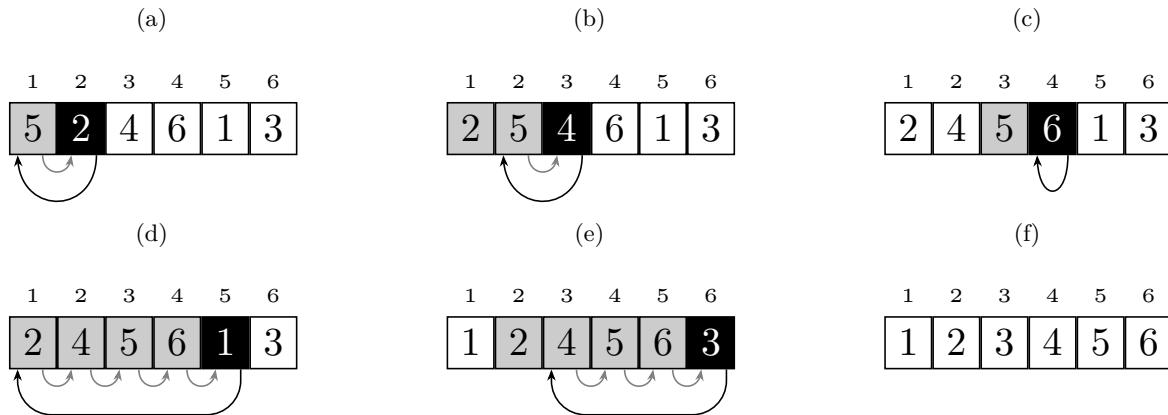
Answer. See Table 1.

2 Getting Started

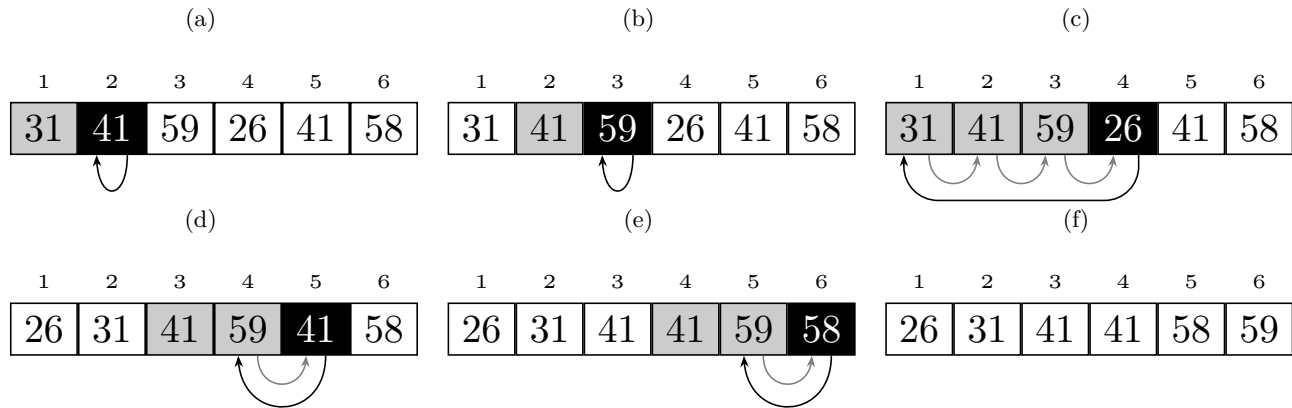
2.1 Insertion Sort

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

Figure 1: Figure 2.2 from the book, included for reference.



Answer. See fig. 3.

Figure 3: Sorting A using INSERTION-SORT.

2.1-2 Rewrite the INSERTION-SORT procedure to sort into nonincreasing instead of nondecreasing order.

Answer. We simply reverse everything: start at the end of the array and go backwards instead of at the start and forwards, and search forwards for a larger element instead of searching backwards for a smaller element:

```

1 def Insertion-Sort( $A[1 \dots n]$ ):
2   for  $j \leftarrow n - 1$  to 1 do
3      $key \leftarrow A[j]$ 
4      $i \leftarrow j + 1$ 
5     while  $i \leq n$  and  $A[i] < key$  do
6        $A[i - 1] \leftarrow A[i]$ 
7        $i \leftarrow i + 1$ 
8     end
9      $A[i - 1] \leftarrow key$ 
10  end
11 end

```

2.1-3 Consider the *searching problem*:

Input: A sequence of n numbers $A = \langle a_1, a_2, \dots, a_n \rangle$ 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.

Answer.

```

1 def Linear-Search( $A[1 \dots n], v$ ):
2   for  $i \leftarrow 1$  to  $n$  do
3     if  $A[i] = v$  then
4       return  $i$ 
5     end
6   end
7   return NIL
8 end

```

We use the following loop invariant to show that the algorithm returns NIL if and only if v is not present in A :

At the start of each iteration of the **for** loop of lines 2-6, the sub-array $A[1 \dots i - 1]$ does not contain v .

We then check that the loop invariant holds.

Initialization: This is trivially true for $i = 1$, as the sub-array $A[1 \dots 0]$ is empty.

Maintenance: We know that none of the elements in $A[1 \dots i - 1]$ are v , and the only additional element in $A[1 \dots i]$ is $A[i]$, which cannot be v , otherwise we would have exited from the loop with the **if** statement on lines 3-5. Therefore the sub-array $A[1 \dots i]$ does not contain v .

Termination: The loop terminates when $i = n + 1$, so $A[1 \dots i - 1] = A[1 \dots n]$ does not contain v . Therefore, we can safely return NIL.

Since NIL is returned if and only if v is not in A , then an index i must be returned if v is in A - this can only happen when $A[i] = v$ (as seen in lines 3-5). Therefore 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.

Answer. We define the *binary addition problem*:

Input: Two n -element sequences $A = \langle a_1, a_2, \dots, a_n \rangle$ and $B = \langle b_1, b_2, \dots, b_n \rangle$ which represent binary numbers, $a_i, b_i \in \{0, 1\}$ for all $1 \leq i \leq n$.

Output: An $(n + 1)$ -element sequence $C = \langle c_1, c_2, \dots, c_{n+1} \rangle$ which represents the binary sum of the two binary numbers given above, i.e. $c_i \in \{0, 1\}$ for all $1 \leq i \leq n + 1$ and

$$\sum_{i=1}^{n+1} c_i 2^{i-1} = \left(\sum_{i=1}^n a_i 2^{i-1} \right) + \left(\sum_{i=1}^n b_i 2^{i-1} \right).$$

We then note a key fact about binary addition, which lets us “carry” in a similar way to normal base 10 addition:

$$\begin{aligned} 1 \cdot 2^i + 1 \cdot 2^i &= 2 \cdot 2^i \\ &= 1 \cdot 2^{i+1} + 0 \cdot 2^i. \end{aligned} \tag{1}$$

With this in mind, we can write pseudocode for an algorithm BINARY-ADD to solve the *binary addition problem*:

```

1 def Binary-Add( $A[1 \dots n], B[1 \dots n]$ ):
    // We only carry from a previous addition, which does not exist at the beginning of
    // the algorithm
2      $carry \leftarrow 0$ 
3     for  $i \leftarrow 1$  to  $n$  do
4          $C[i] \leftarrow A[i] + B[i] + carry$ 
5         if  $C[i] > 1$  then
6              $C[i] \leftarrow C[i] - 2$ 
7              $carry \leftarrow 1$ 
8         else
9              $carry \leftarrow 0$ 
10        end
11    end
12     $C[n + 1] \leftarrow carry$ 
13    return  $C[1 \dots (n + 1)]$ 
14 end

```

We then use the following loop invariant to show the correctness of BINARY-ADD:

At the start of each iteration of the **for** loop of lines 3-11, the values $c_j \in \{0, 1\}$ for $1 \leq j \leq i - 1$ and

$$\sum_{j=1}^{i-1} c_j 2^{j-1} + (carry_{(i-1)}) 2^{i-1} = \left(\sum_{j=1}^{i-1} a_j 2^{j-1} \right) + \left(\sum_{j=1}^{i-1} b_j 2^{j-1} \right).$$

Initialization: This is trivially true, as $i - 1 = 0$, so we don't have to worry about any elements of C yet. As well, the sum evaluates to $0 + (\text{carry}_0)2^0 = 0 + 0$, which is true, since $\text{carry}_0 = 0$.

Maintenance: Observe that $\text{carry} \in \{0, 1\}$ for the entire algorithm. Since $a_i, b_i, \text{carry}_{(i-1)} \leq 1$, then $a_i + b_i + \text{carry}_{(i-1)} \leq 3$. Then we have two cases:

Case 1: $c_i = a_i + b_i + \text{carry}_{(i-1)} \leq 1$. Then $c_i \in \{0, 1\}$, the next $\text{carry}_i = 0$, and

$$\begin{aligned}
 \sum_{j=1}^i c_j 2^{j-1} + (\text{carry}_i) \cdot 2^i &= \sum_{j=1}^{i-1} c_j 2^{j-1} + c_i 2^{i-1} \\
 &= \sum_{j=1}^{i-1} c_j 2^{j-1} + (a_i + b_i + (\text{carry}_{(i-1)})) 2^{i-1} \\
 &= \left(\sum_{j=1}^{i-1} c_j 2^{j-1} + (\text{carry}_{(i-1)}) 2^{i-1} \right) + (a_i + b_i) 2^{i-1} \\
 &= \left(\sum_{j=1}^{i-1} a_j 2^{j-1} \right) + \left(\sum_{j=1}^{i-1} b_j 2^{j-1} \right) + (a_i + b_i) 2^{i-1} \\
 &= \left(\sum_{j=1}^{i-1} a_j 2^{j-1} + a_i 2^{i-1} \right) + \left(\sum_{j=1}^{i-1} b_j 2^{j-1} + b_i 2^{i-1} \right) \\
 &= \left(\sum_{j=1}^i a_j 2^{j-1} \right) + \left(\sum_{j=1}^i b_j 2^{j-1} \right).
 \end{aligned}$$

Case 2: $2 \leq a_i + b_i + \text{carry}_{(i-1)} \leq 3$. Then $c_i = a_i + b_i + \text{carry}_{(i-1)} - 2 \in \{0, 1\}$, the next $\text{carry}_i = 1$, and

$$\begin{aligned}
 \sum_{j=1}^i c_j 2^{j-1} + (\text{carry}_i) \cdot 2^i &= \sum_{j=1}^{i-1} c_j 2^{j-1} + c_i 2^{i-1} + 2^i \\
 &= \sum_{j=1}^{i-1} c_j 2^{j-1} + (a_i + b_i + \text{carry}_{(i-1)} - 2) 2^{i-1} + 2^i \\
 &= \left(\sum_{j=1}^{i-1} c_j 2^{j-1} + \text{carry}_{(i-1)} 2^{i-1} \right) + (a_i + b_i) 2^{i-1} - \cancel{2^i} + \cancel{2^i} \\
 &= \left(\sum_{j=1}^{i-1} a_j 2^{j-1} \right) + \left(\sum_{j=1}^{i-1} b_j 2^{j-1} \right) + (a_i + b_i) 2^{i-1} \\
 &= \left(\sum_{j=1}^{i-1} a_j 2^{j-1} + a_i 2^{i-1} \right) + \left(\sum_{j=1}^{i-1} b_j 2^{j-1} + b_i 2^{i-1} \right) \\
 &= \left(\sum_{j=1}^i a_j 2^{j-1} \right) + \left(\sum_{j=1}^i b_j 2^{j-1} \right).
 \end{aligned}$$

Termination: The loop terminates when $i = n + 1$, so $c_j \in \{0, 1\}$ for all $1 \leq j \leq i - 1 = n$. As well, $c_{n+1} = \text{carry}_n \in \{0, 1\}$. Then we have

$$\begin{aligned}
 \sum_{j=1}^{n+1} c_j 2^{j-1} &= \sum_{j=1}^n c_j 2^{j-1} + c_{n+1} 2^n \\
 &= \sum_{j=1}^{i-1} c_j 2^{j-1} + \text{carry}_{(i-1)} 2^{i-1}
 \end{aligned}$$

(Since the algorithm terminates by assigning $C[n+1] \leftarrow \text{carry}$)

$$\begin{aligned}
 &= \left(\sum_{j=1}^{i-1} a_j 2^{j-1} \right) + \left(\sum_{j=1}^{i-1} b_j 2^{j-1} \right) \\
 &= \left(\sum_{j=1}^n a_j 2^{j-1} \right) + \left(\sum_{j=1}^n b_j 2^{j-1} \right).
 \end{aligned}$$

Therefore, our algorithm is correct.

2.2 Analyzing algorithms

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

Answer. We have

$$\begin{aligned}
 n^3/1000 - 100n^2 - 100n + 3 &= \Theta(n^3) + \Theta(n^2) + \Theta(n) + \Theta(1) \\
 &= \Theta(n^3) + \mathcal{O}(n^3) + \mathcal{O}(n^3) + \mathcal{O}(n^3) \\
 &= \boxed{\Theta(n^3)}.
 \end{aligned}$$

2.2-2 Consider sorting n numbers stored in an 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.

Answer.

```

1 def Selection-Sort( $A[1 \dots n]$ ):
2   for  $i \leftarrow 1$  to  $n-1$  do                                     //  $n$ 
3      $min \leftarrow i$                                               //  $n-1$ 
4     for  $j \leftarrow i+1$  to  $n$  do                                   //  $\sum_{i=1}^{n-1} (n-i)$ 
5       if  $A[j] < A[min]$  then                                       //  $\sum_{i=1}^{n-1} (n-i-1)$ 
6          $min \leftarrow j$ 
7       end
8     end
9     Swap( $A[i], A[min]$ )
10  end
11 end

```

This algorithm maintains the following loop invariant:

At the start of each iteration of the **for** loop of lines 2-10, the sub-array $A[1 \dots i-1]$ is sorted, and every element in $A[1 \dots i-1]$ is less than or equal to every element in $A[i \dots n]$.

The key to why the algorithm only needs to run for the first $n-1$ elements lies in the termination of this loop invariant:

Initialization: The loop invariant is trivially satisfied, as $A[1 \dots 0]$ is empty.

Maintenance: Since all of the elements in the sub-array $A[1 \dots i-1]$ are smaller than (or equal to) all of the elements in $A[i \dots n]$, by swapping any element into $A[i]$ from $A[i \dots n]$ means $A[1 \dots i]$ is sorted. By making this element be the smallest element in $A[i \dots n]$, we maintain that all of the elements in $A[1 \dots i]$ are smaller than or equal to $A[i+1 \dots n]$.

Termination: The loop terminates when $i = n$. Therefore, the sub-array $A[1 \dots n-1]$ is sorted, and every element in $A[1 \dots n-1]$ is smaller than (or equal to) $A[n]$. Therefore $A[1 \dots n]$ is sorted.

No loop terminates early under any circumstance and all **if** statements contain only $\Theta(1)$ -time statements, so the best-case and worst-case running-times of SELECTION-SORT are the same. We wish to expand out some sums seen above:

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

Then the best-case and worst-case running-times are

$$\begin{aligned}
 n + (n-1) + \Theta(n^2) + \Theta(n^2) &= \mathcal{O}(n^2) + \mathcal{O}(n^2) + \Theta(n^2) + \Theta(n^2) \\
 &= \boxed{\Theta(n^2)}.
 \end{aligned}$$

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

Answer. The number of elements that need to be checked is between 1 (best case) and n (worst-case). If the element is equally likely to be any element in the array, then the number of elements which need to be checked in the array, X , is distributed uniformly across $n = \{1, 2, \dots, n\}$. Then the expected number of elements which need to be checked in the average-case is $\mathbb{E}[X] = \boxed{\frac{1+n}{2}}$. The number of elements that need to be checked in the worst-case is \boxed{n} , since after checking all elements, we have either found the sought-after item, or could conclude that it is not in the array. In either case, the running times are $\boxed{\Theta(n)}$.

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

Answer. We can simply pre-compute the correct output for some set of inputs, and modify our algorithm to first check if our input is one of the pre-computed ones. Then the best-case running time of the algorithm is simply the time it takes to check if the input is one of the pre-computed ones. For input structures such as arrays, the running time of this check is $\Theta(n)$.

2.3 Designing algorithms

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

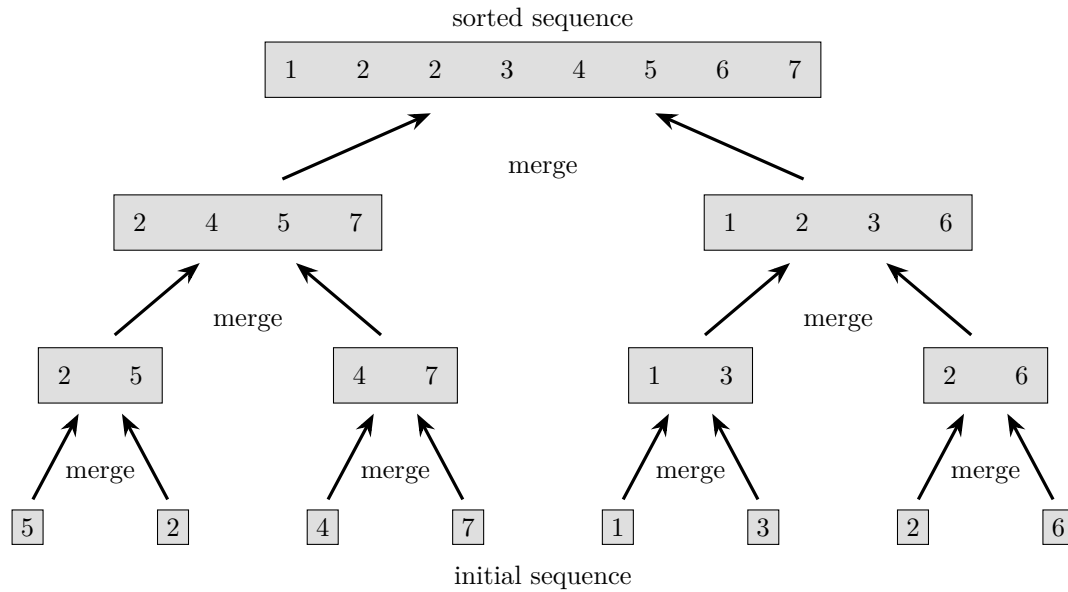
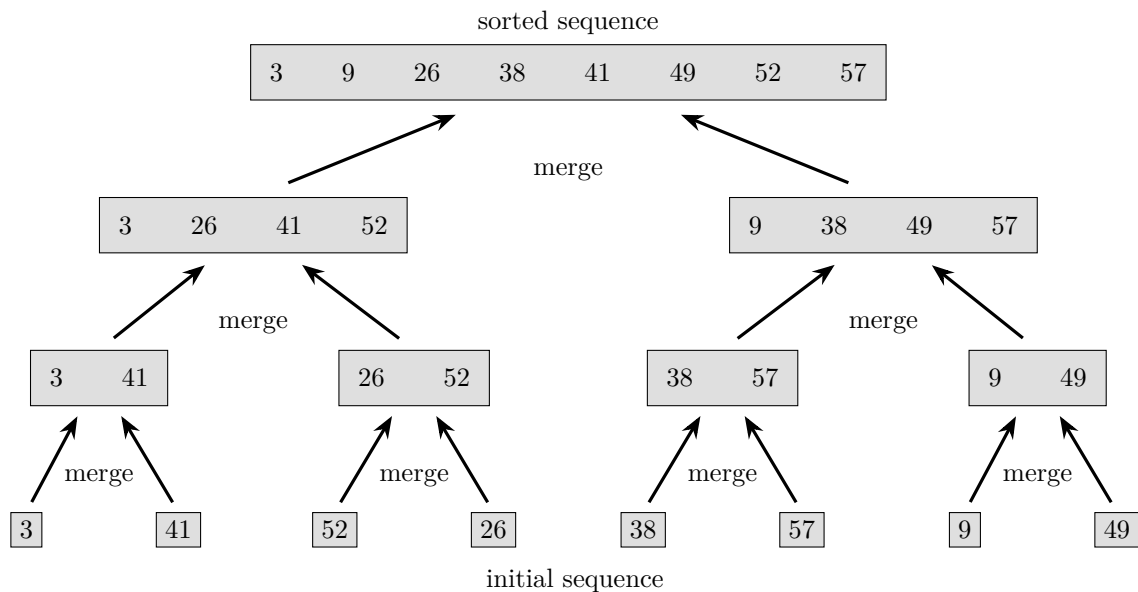


Figure 5: Figure 2.4 from the book, included for reference.

Figure 6: Merging A .

Answer. See Figure 6.

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 remained of the other array back into A .

Answer. See algorithm 1.

```

1 def Merge( $A[p \dots q \dots r]$ ):
2    $n_1 \leftarrow q - p + 1$ 
3    $n_2 \leftarrow r - q$ 
4   for  $i \leftarrow 1$  to  $n_1$  do
5      $L[i] \leftarrow A[p + i - 1]$ 
6   end
7   for  $j \leftarrow 1$  to  $n_2$  do
8      $R[j] \leftarrow A[q + j]$ 
9   end
10   $i \leftarrow 1$ 
11   $j \leftarrow 1$ 
12   $k \leftarrow 1$ 
13  while  $i \leq n_1$  and  $j \leq n_2$  do
14    if  $L[i] \leq R[j]$  then
15       $A[k] \leftarrow L[i]$ 
16       $i \leftarrow i + 1$ 
17    else
18       $A[k] \leftarrow R[j]$ 
19       $j \leftarrow j + 1$ 
20    end
21     $k \leftarrow k + 1$ 
22  end
23  for  $i$  to  $n_1$  do
24     $A[k] \leftarrow L[i]$ 
25     $k \leftarrow k + 1$ 
26  end
27  for  $j$  to  $n_2$  do
28     $A[k] \leftarrow R[j]$ 
29     $k \leftarrow k + 1$ 
30  end
31 end

```

Algorithm 1: MERGE redesigned to not use sentinels.

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

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

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

Answer. For $n = 2^1$, we have $T(2) = 2 = 2 \cdot 1 = 2 \lg(2)$, so the solution works for $k = 1$. Then, assume $T(2^k) = 2^k \lg(2^k)$ for some $k \geq 1$. We have

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

Therefore, by induction, the solution works for all $k \geq 1$.

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

Answer. Since the worst-case running time of inserting an element into a sorted array of size n is n (as we might have to insert the element at the front of the sorted array), our recurrence is

$$T(n) = T(n-1) + (n-1).$$

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

Answer. See algorithm 2.

```

1 def Binary-Search( $A[a \dots b], v$ ):
2   if  $b < a$  then
3     return NIL
4   end
5    $m \leftarrow \lfloor \frac{a+b}{2} \rfloor$ 
6   if  $A[m] = v$  then
7     return  $m$ 
8   else if  $v > A[m]$  then
9     return Binary-Search( $A[m+1 \dots b], v$ )
10  else
11    return Binary-Search( $A[a \dots m-1], v$ )
12  end
13 end

```

Algorithm 2: BINARY-SEARCH

The worst-case scenario for this algorithm happens when it terminates due to $b < a$ - this happens when there are only one or two elements left in A , and neither one of them are v . Since the algorithm effectively halves the size of A each time it runs, we have a recurrence for the worst-case runtime $T(n)$ of the algorithm in the worst-case on an array of size n :

$$T(n) = \begin{cases} c, & n \leq 2 \\ T(n/2) + c & n > 2 \end{cases}.$$

Then we have

$$\begin{aligned} T(n) &= \sum_{i=1}^{\lg(n)} c \\ &= c \lg(n) \\ &= \Theta(\lg(n)). \end{aligned}$$

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

Answer. No - while the amount of time it takes to find the correct element (in the worst-case) is i and we can improve this to $\lg(i)$, it still takes i swaps to get the element into the correct position.

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 .

Answer.

```

1 def Sums-To( $A[1 \dots n], x$ ):
2   Merge-Sort( $A[1 \dots n]$ )
3    $l \leftarrow 1$ 
4    $r \leftarrow n$ 
5   while  $l < r$  do
6     if  $A[l] + A[r] = x$  then
7       return true
8     else if  $A[l] + A[r] < x$  then
9        $l \leftarrow l + 1$ 
10    else
11       $r \leftarrow r - 1$ 
12    end
13  end
14  return false
15 end

```

We wish to show the correctness of SUMS-TO. Obviously if SUMS-TO returns **true**, then there actually is such a pair, due to the preceding **if**-statement. To show that if SUMS-TO returns **false** then there are no such elements which sum to x , we maintain the following loop invariant:

At the start of each iteration of the **while** loop of lines 5-13, none of the elements in the subarrays $A[1 \dots l - 1]$ and $A[r + 1 \dots n]$ can be used to sum up to x .

We then check that the loop invariant holds.

Initialization: This is trivially true for $l = 1, r = n$, as the subarrays $A[1 \dots l - 1]$ and $A[r + 1 \dots n]$ are empty.

Maintenance: We have two cases:

Case 1: $A[l] + A[r] < x$. Since we have identified that the elements in $A[r + 1 \dots n]$ cannot be used to sum up to x , then $A[r]$ is the largest value which we can still use to sum to x (since A is sorted). Then $A[l] + A[i] < A[l] + A[r] < x$ for any $l < i < r$, so $A[l]$ cannot be used to sum up to x and none of the elements in $A[1 \dots l]$ can be used to sum up to x .

Case 2: $A[l] + A[r] > x$. Since we have identified that the elements in $A[1 \dots l - 1]$ cannot be used to sum up to x , then $A[l]$ is the smallest value which can still be used to sum to x (since A is sorted). Then $A[i] + A[r] > A[l] + A[r] > x$ for any $l < i < r$, so $A[r]$ cannot be used to sum up to x and none of the elements in $A[r \dots n]$ can be used to sum up to x .

Termination: Since $l = r$, then the only element left that can be used to sum to x is $A[l]$, but we need a pair to sum to x . Therefore, there are no pairs of numbers which can sum to x .

We know that MERGE-SORT runs in $\Theta(n \lg(n))$ worst-case time, so what remains is to check that the rest of the algorithm doesn't affect this run-time. Since, in the worst-case, the loop ends each iteration by incrementing l by 1 or decrementing r by 1 (but not both), the number of iterations that have run is $(l - 1) + (n - r)$. In the worst-case scenario, the loop terminates when $l = r$, so the number of times the loop has run is $(l - 1) + (n - l) = n - 1$. Then the number of extra operations we perform after MERGE-SORT is $c_1 + n + c_2(n - 1) = \Theta(n)$. Therefore the overall runtime of the algorithm is $\Theta(n \lg(n)) + \Theta(n) = \Theta(n \lg(n)) + \mathcal{O}(n \lg(n)) = \Theta(n \lg(n))$.

Problems

2-1 Insertion sort on small arrays in merge sort. Although merge sort runs in $\Theta(n \lg(n))$ worst-case time and insertion sort runs in $\Theta(n^2)$ worst-case time, the constant factors in insertion sort can make it faster in practice for small problem sizes on many machines. Thus, it would make sense to *coarsen* the leaves of 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.

Answer. Since each sublist is of size k , we know sorting each sublist will take $\Theta(k^2)$ worst-case time. Since there are $\frac{n}{k}$ such sublists, the total worst-case sorting time will be $\frac{n}{k}\Theta(k^2) = \Theta(nk)$.

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

Answer. Following our original analysis of MERGE-SORT, we know that the $\lg(n)$ factor comes from the height of the recursion tree. Starting from the bottom of the tree, we cut the number of un-merged arrays (represented by the number of nodes on each level, starting with the leaves) in half by merging them. In the worst-case scenario, the recursion tree is complete, so the number of leaves is a power of 2, and the number of times we can cut a number of nodes n in half is $\lg(n)$. So if we start with n/k un-merged arrays (leaves), the number of times we can cut the number of un-merged arrays in half is $\lg(n/k)$. Therefore, we can 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 times, 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?

Answer. If we choose $k = \Theta(\lg(n))$, then the run time is $\Theta(n \lg(n) + n \lg(n/\lg(n))) = \Theta(n \lg(n))$, since $n/\lg(n) < n$ for $n > 2$. If we choose k any (asymptotically) larger, then our runtime would be worse.

- d. How should we choose k in practice?

Answer. In practice, we shouldn't choose k to be an asymptotic function of n , since we are arguing that that INSERTION-SORT is better than MERGE-SORT for *small* values of k , based on the constants which are hidden by the asymptotic notation. In practice, we should pick a constant number for k such that the constants of INSERTION-SORT allow better run-time than the constants of MERGE-SORT. This can be figured out with a more in-depth analysis of the two algorithms, without asymptotic notation, or through trial-and-error on an actual machine.

2-2 Correctness of bubblesort. Bubblesort is a popular, but inefficient, sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order.

```

1 def Bubblesort( $A[1 \dots n]$ ):
2     for  $i \leftarrow 1$  to  $n - 1$  do
3         for  $j \leftarrow n$  to  $i + 1$  do
4             if  $A[j] < A[j - 1]$  then
5                 Swap( $A[j], A[j - 1]$ )
6             end
7         end
8     end
9 end

```

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

$$A'[1] \leq A'[2] \leq \dots \leq A'[n], \quad (2)$$

where n is the length of A . In order to show that BUBBLESORT actually sorts, what else do we need to prove?

Answer. We need to prove that A' is a permutation of A .

The next two parts will prove ineq. (2).

- b. State precisely a loop invariant for the **for** loop in lines 3-7, and prove that this loop invariant holds. Your proof should use the structure of the loop invariant proof presented in this chapter.

Answer. We use the following loop invariant:

At the start of each iteration of the **for** loop of lines 3-7, the element $A[j] \leq A[m]$ for all $j + 1 \leq m \leq n$.

We then prove this loop invariant using the structure presented in this chapter:

Initialization: This is trivially true, since $j + 1 = n + 1$, so there is no such m .

Maintenance: We have two cases:

Case 1: $A[j-1] > A[j]$. Then we swap elements, and we know now that $A[j-1] < A[j]$. So $A[j-1] < A[m]$ for $j \leq m \leq n$.

Case 2: $A[j-1] \leq A[j]$. Then $A[j-1] \leq A[j] \leq A[m]$ for all $j \leq m \leq n$.

Termination: Since the loop terminates when $j = i$, then $A[i] \leq A[m]$ for all $i+1 \leq m \leq n$. In other words, $A[i]$ is the minimum element of the sub-array $A[i \dots n]$.

- c. Using the termination condition of the loop invariant proved in Problem 2-7.b, state a loop invariant for the **for** loop in lines 1-8 that will allow you to prove ineq. (2). Your proof should use the structure of the loop invariant proof presented in this chapter.

Answer. We use the following loop invariant:

At the start of each iteration of the **for** loop of lines 2-8,

$$A'[1] \leq A'[2] \leq \dots \leq A'[i] \leq A'[m], \quad (3)$$

for all $i+1 \leq m \leq n$.

We then prove this loop invariant using the structure presented in this chapter:

Initialization: This is trivially true, since $i = 0$, so ineq. (3) is empty.

Maintenance: From ineq. (3) above, we have for $m = i+1$,

$$A'[1] \leq A'[2] \leq \dots \leq A'[i] \leq A'[i+1],$$

and from the loop invariant in Problem 2-7.b above, we have $A'[i+1] \leq A[m]$ for all $i+2 \leq m \leq n$.

Termination: Since the loop terminates when $i = n$, ineq. (2) follows directly from ineq. (3).

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

Answer. In this version of bubblesort, the worst-case running time occurs every time, since there is no early loop termination, and it is

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

This is the same worst case running time of insertion sort, but insertion sort performs better in the best case.

2-3 Correctness of Horner's rule. The following code fragment implements Horner's rule for evaluating a polynomial

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

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

```

1 y ← 0
2 for i ← n to 0 do
3   | y ← ai + x · y
4 end

```

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

Answer. The running time for this code fragment is $\Theta(n)$.

- 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?

Answer.

```

1  $y \leftarrow 0$ 
2 for  $i \leftarrow 0$  to  $n$  do
3    $t \leftarrow a_i$ 
4   for  $j \leftarrow 1$  upto  $i$  do
5      $t \leftarrow t \cdot x$ 
6   end
7    $y \leftarrow y + t$ 
8 end

```

The running time of this algorithm is

$$\begin{aligned}
 (n+2) + (n+1) + \left(\sum_{i=0}^n (i+1) \right) + \left(\sum_{i=0}^n i \right) + (n+1) &= 3n+4 + 2 \sum_{i=0}^n i + \sum_{i=0}^n 1 \\
 &= 3n+4 + n(n+1) + n \\
 &= n^2 + 5n + 4 \\
 &= \Theta(n^2).
 \end{aligned}$$

- c. Consider the following loop invariant:

At the start of each iteration of the **for** loop of lines 2-4,

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

Interpret a summation with no terms as equaling 0. Following the structure of the loop invariant proof presented in this chapter, use the loop invariant to show that, at termination,

$$y = \sum_{k=0}^n a_k x^k.$$

Answer. We prove the loop invariant by following the structure presented in this chapter:

Initialization: At the start of the loop, $y = 0$ and $i = n$, so we have

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

Maintenance: The next y is now

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

Termination: The loop terminates when $i = -1$, so we have

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

which is the correct value of $P(x)$.

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

Answer. See the termination step of 2-7.c above.

2-4 Inversions. Let $A[1 \dots n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an ***inversion*** of A .

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

Answer. The five inversion are $(1, 5), (2, 5), (3, 5), (4, 5), (3, 4)$.

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

Answer. The array $A = \langle n, (n-1), \dots, 2, 1 \rangle$ has the most inversions, since $A[i] > A[j]$ whenever $i < j$. In other words, every element is in an inversion pair with every succeeding element, so the number of inversions is

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

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


Answer. The number of inversions in the input array is the number of times the **while** loop on lines 5-7 repeats, so the running time of insertion sort is $\Theta(n + k)$, where n is the size of the input array and k is the number of inversions. This is due to the loop invariant discussed in the chapter:

At the start of each iteration of the **for** loop of lines 1-8, the subarray $A[1 \dots j - 1]$ consists of the elements originally in $A[1 \dots j - 1]$, but in sorted order.

This loop invariant implies another loop invariant:

At the start of each iteration of the **for** loop of lines 1-8, the subarray $A[1 \dots j]$ consists of the same number of inversions (i, j) for $1 \leq i < j$ as the original array A . As well, if (a, j) is an inversion, then (b, j) is an inversion for all $1 \leq a < b < j$. Finally, the number of shifts required to insert $A[j]$ into $A[1 \dots j]$ is equal to the number of inversions (i, j) in the original array A .

We can skip the initialization and maintenance step of proving this loop invariant if we show that this loop invariant is a consequence of the previous loop invariant, which we do now.

Proof. Whenever we insert an element $A[j]$ into $A[1 \dots j]$, the new index $m \leq j$. Therefore, if (i, j) is an inversion in the original array A , then $A[i] < A[j]$ in the original array and now $A[m] < A[j]$ in the new subarray, so (i, m) is an inversion in $A[1 \dots j]$. Since $A[1 \dots j - 1]$ consists of the elements originally in $A[1 \dots j - 1]$, then there cannot be any “new” inversions. Therefore, the number of inversions (i, j) in $A[1 \dots j]$ is equal to the number in the original array A . Since $A[1 \dots j - 1]$ is sorted, then $1 \leq a < b < j$ implies that $A[a] \leq A[b]$, so if (a, j) is an inversion, then $A[b] \geq A[a] \geq A[j]$, so (b, j) is also an inversion. Finally, we wish to insert $A[j]$ into the minimum index a such that (a, j) is an inversion (or j otherwise, in which case there are no shifts). Since all elements $A[b]$ where $a < b < j$ are shifted, and these are all of the inversions (a, j) , then the number of shifts required to insert $A[j]$ into $A[1 \dots j]$ is equal to the number of such inversions. 

Then, we note the termination step of this new loop invariant:

Termination: The total number of swaps that have been performed is the sum over the number of inversions (i, j) for $2 \leq j \leq n$, which is the sum of all inversions.

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

Answer. See algorithm 3.

```

1 def Count-Inversions( $A[a \dots b]$ ):
2   if  $a < b$  then
3      $m \leftarrow \lfloor (a + b) / 2 \rfloor$ 
4      $c \leftarrow \text{Count-Inversions}(A[a \dots m])$ 
5      $c \leftarrow \text{Count-Inversions}(A[m + 1 \dots b]) + c$ 
6      $c \leftarrow \text{Merge-Inversions}(A[a \dots m \dots b]) + c$ 
7     return  $c$ 
8   end
9   return 0
10 end
11 def Merge-Inversions( $A[a \dots m \dots b]$ ):
12    $n_1 \leftarrow m - a + 1$ 
13    $n_2 \leftarrow b - m$ 
14   for  $i \leftarrow 1$  to  $n_1$  do
15      $L[i] \leftarrow A[a + i - 1]$ 
16   end
17   for  $j \leftarrow 1$  to  $n_2$  do
18      $R[j] \leftarrow A[m + j]$ 
19   end
20    $i \leftarrow 1$ 
21    $j \leftarrow 1$ 
22    $k \leftarrow 1$ 
23    $c \leftarrow 0$ 
24   while  $i \leq n_1$  and  $j \leq n_2$  do
25     if  $L[i] \leq R[j]$  then
26        $A[k] \leftarrow L[i]$ 
27        $i \leftarrow i + 1$ 
28     else
29        $A[k] \leftarrow R[j]$ 
30        $j \leftarrow j + 1$ 
31       /* Whenever we insert an element from the right sub-array before finishing
32          the left sub-array, we resolved a number of inversions equal to the
33          remaining un-inserted elements in the left sub-array. */
34        $c \leftarrow c + n_1 - i + 1$ 
35     end
36      $k \leftarrow k + 1$ 
37   end
38   for  $i$  to  $n_1$  do
39      $A[k] \leftarrow L[i]$ 
40      $k \leftarrow k + 1$ 
41   end
42   for  $j$  to  $n_2$  do
43      $A[k] \leftarrow R[j]$ 
44      $k \leftarrow k + 1$ 
45   end
46   return  $c$ 
47 end

```

Algorithm 3: COUNT-INVERSIONS, which runs in $\Theta(n \lg(n))$ worst-case time. Uses modified version of MERGE from algorithm 1.

3 Growth of Functions

3.1 Asymptotic Notation

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

Answer. Let $n_0 = 1$ and $n \geq n_0$. Without loss of generality, assume $f(n) \geq g(n)$. Then we have

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

and

$$\begin{aligned}\max\{f(n), g(n)\} &= f(n) \\ &= \frac{1}{2}(f(n) + f(n)) \\ &\geq \frac{1}{2}(f(n) + g(n)),\end{aligned}$$

so for $c_1 = \frac{1}{2}$ and $c_2 = 1$, $\max\{f(n), g(n)\} = \Theta(f(n) + g(n))$.

3.1-2 Show that for any real constants a and b , where $b > 0$,

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

Answer. Let $n_0 = 1$ and $n \geq n_0$. When $a \geq 0$, we have

$$\begin{aligned}(n + a)^b &\leq (n + an)^b \\ &= (1 + a)^b n^b,\end{aligned}$$

and

$$\begin{aligned}(n + a)^b &\geq (n + 0)^b \\ &= n^b,\end{aligned}$$

so for $c_1 = 1$ and $c_2 = (1 + a)^b$, $(n + a)^b = \Theta(n^b)$. When $a < 0$, then all of the above inequalities are exactly flipped, so $(n + a)^b = \Theta(n^b)$ still.

3.1-3 Explain why the statement, “The running time of algorithm A is at least $\mathcal{O}(n^2)$,” is meaningless.

Answer. Since \mathcal{O} -notation gives an upper bound, it implies upper bounds of anything greater than n^2 by itself.

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

Answer. Yes, $2^{n+1} = \mathcal{O}(2^n)$, since $2^{n+1} = 2 \cdot 2^n$, so for $c = 2$ we have $2^{n+1} = \mathcal{O}(2^n)$. However, $2^{2n} \neq \mathcal{O}(2^n)$. If given $c > 0$, then $2^{2n} > c2^n$ for all $n > \lg(c)$, so $2^{2n} = \omega(2^n)$.

3.1-5 Prove Theorem 3.1.

Theorem (Theorem 3.1). *For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$.*

Answer. \Rightarrow Let f, g be functions such that $f(n) = \Theta(g(n))$. Then there exist constants $n_0, c_1, c_2 > 0$ such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ whenever $n \geq n_0$. The second inequality tells us that $f(n) = \Omega(g(n))$, and the third tells us that $f(n) = \mathcal{O}(g(n))$.

\Leftarrow Let f, g be functions such that $f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$. Then there exist constants n_1, n_2, c_1, c_2 such that $0 \leq f(n) \leq c_1 g(n)$ whenever $n > n_1$ and $0 \leq c_2 g(n) \leq f(n)$ whenever $n > n_2$. Then $0 \leq c_2 g(n) \leq f(n) \leq c_1 g(n)$ whenever $n \geq \max\{c_1, c_2\}$. Therefore, $f(n) = \Theta(g(n))$.

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

Answer. \Rightarrow Assume that the running time of an algorithm is $\Theta(g(n))$. Then the worst-case running time and best-case running times are also $\Theta(g(n))$. The rest follows from Theorem 3.1.

\Leftarrow Let $f(n)$ represent the running time of an algorithm, $f_{\text{worst}}(n)$ represent the running time of the algorithm in worst-case, and $f_{\text{best}}(n)$ represent the running time of the algorithm in the best-case. It should be intuitive that $f_{\text{best}}(n) \leq f(n) \leq f_{\text{worst}}(n)$. Assume that $f_{\text{worst}}(n) = \mathcal{O}(g(n))$ and $f_{\text{best}}(n) = \Omega(g(n))$. Then there exist constants $n_1, n_2, c_1, c_2 > 0$ such that $0 \leq f_{\text{worst}} \leq c_1 g(n)$ whenever $n \geq n_1$ and $0 \leq c_2 g(n) \leq f_{\text{best}}(n)$ whenever $n \geq n_2$. Then we have

$$0 \leq c_2 g(n) \leq f_{\text{best}}(n) \leq f(n) \leq f_{\text{worst}}(n) \leq c_1 g(n)$$

whenever $n \geq \max\{n_1, n_2\}$. Therefore the running time of the algorithm is $\Theta(g(n))$.

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

Answer. By contradiction, assume $o(g(n)) \cap \omega(g(n))$ is nonempty. Then there exists a function f and constants $n_1, n_2 > 0$ such that $0 \leq f(n) < cg(n)$ whenever $n \geq n_1$ and $0 \leq cg(n) < f(n)$ whenever $n \geq n_2$ for all $c > 0$. But then $g(n) < f(n) < g(n)$ for all $n \geq \max\{n_1, n_2\}$ and $0 < 0$ (by subtracting $g(n)$ from both sides), which is a contradiction. Therefore $o(g(n)) \cap \omega(g(n))$ 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 by $\mathcal{O}(g(n, m))$ the set of functions

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

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

Answer. We define $\Omega(g(n, m))$ and $\Theta(g(n, m))$ as

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

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 $f(n)g(n)$ is monotonically increasing.

Answer. Let f, g be monotonically increasing functions and $a, b \in \mathbb{Z}$ such that $a < b$. Then $f(a) \leq f(b)$ and $g(a) \leq g(b)$. Then $f(a) + g(a) \leq f(b) + g(b)$ by adding the previous two inequalities together, so $f(n) + g(n)$ is monotonically increasing. As well, since $g(a) \leq g(b)$ and f is monotonically increasing, then $f(g(a)) \leq f(g(b))$, so $f(g(n))$ is monotonically increasing. Finally, if $g(a), f(b) \geq 0$, then $f(a)g(a) \leq f(b)g(a) \leq f(b)g(b)$, so $f(n)g(n)$ is monotonically increasing.

- 3.2-2** Prove eq. (3.16).

$$a^{\log_b(c)} = c^{\log_b(a)} \tag{3.16}$$

Answer. We have

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

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

$$\lg(n!) = \Theta(n \lg(n)) \quad (3.19)$$

Answer. By properties of logarithms, we have

$$\begin{aligned}
 \lg(n!) &= \lg\left(\prod_{i=1}^n i\right) \\
 &= \sum_{i=1}^n \lg(i) \\
 &= \Theta(n \lg(n)),
 \end{aligned}$$

which follows from Problem A-5.b.

We would like to show that $n! \geq n^{n/2}$ or alternatively $(n!)^2 \geq n^n$. We can show this with induction, using this fact:

$$(1+x)^n \geq 1+nx, \quad x \geq 0 \quad (4)$$

(which we will also prove) and noting that $(1!)^2 = 1 \geq 1 = 1^1$, $(1+x)^1 \geq 1+1x$, and $(1+x)^2 = 1+2x+x^2 \geq 1+2x$. Then for some $n \geq 1$ assume that $(n!)^2 \geq n^n$ and $(1+x)^n \geq 1+nx$ for all $x \geq 0$. Then we have

$$\begin{aligned}
 ((n+1)!)^2 &= (n!)^2(n+1)^2 \\
 &\geq n^n(n+1)^2 \\
 &= nn^{n-1}(n+1)^2 \\
 &= n\left(\frac{n-1}{n}\right)^{n-1}\left(\frac{n}{n-1}\right)^{n-1}n^{n-1}(n+1)^2 \\
 &= n\left(1-\frac{1}{n}\right)^{n-1}\frac{n^{2(n-1)}(n+1)^2}{(n-1)^{n-1}} \\
 &\stackrel{(4)}{\geq} n\left(1-\frac{n-1}{n}\right)\frac{n^{2(n-1)}(n+1)^2}{(n-1)^{n-1}} \\
 &= (1)\frac{n^{2(n-1)}(n+1)^2}{(n-1)^{n-1}} \\
 &\geq \left(1-\frac{1}{n^2}\right)^{n-1}\frac{n^{2(n-1)}(n+1)^2}{(n-1)^{n-1}}
 \end{aligned}$$

(with $x = -\frac{1}{n}$)

$$\begin{aligned}
& \text{(since } 0 < \frac{1}{n^2} \leq 1, \text{ then } 0 \leq (1 - \frac{1}{n^2})^{n-1} < 1) \\
&= \left(\frac{n^2 - 1}{n^2} \right)^{n-1} \frac{n^{2(n-1)}(n+1)^2}{(n-1)^{n-1}} \\
&= ((n-1)(n+1))^{n-1} \frac{(n+1)^2}{(n-1)^{n-1}} \\
&= (n+1)^{n+1},
\end{aligned}$$

$$\begin{aligned}
(1+x)^{n+2} &= (1+x)^n(1+x)^2 \\
&\geq (1+xn)(1+2x+x^2) \\
&= (1+2x+x^2) + xn + 2x^2n + x^3n \\
&= 1 + (n+2)x + (2n+1)x^2 + nx^3 \\
&\geq 1 + (n+2)x
\end{aligned}$$

Then by induction $(n!)^2 \geq n^n$ and we have

$$\begin{aligned}
n! &> c2^n && \Leftarrow \\
n^{n/2} &> c2^n && \Leftarrow \\
16^{n/2} &> c2^n && \Leftarrow \\
\text{(if } n > 16) &&& \\
4^n &> c2^n && \Leftarrow \\
2^n &> c && \Leftarrow \\
n &> \lg(c), &&
\end{aligned}$$

so $n! = \omega(2^n)$.

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

Answer.

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

Answer. Since $\lg^*(\lg(n)) = \lg^*(n) - 1$ (we just pre-apply a single \lg before counting the number of \lg), and $\lg(n) = o(n-1)$, then $\lg(\lg^*(n)) = o(\lg^*(n) - 1)$.

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

Answer.

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

$$F_i = \frac{1}{\sqrt{5}}(\varphi^i - \hat{\varphi}^i),$$

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

Answer.

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

Answer.

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?

Answer. The singleton subarray $A[i \dots i]$ where $A[i] = \max_j \{A[j]\}$. This is due to the base case returning a singleton array.

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.

Answer. See algorithm 4.

```

1 def Maximum-Subarray( $A[1 \dots n]$ ):
2    $sum_{max} \leftarrow -\infty$ 
3   for  $i \leftarrow 1$  to  $n$  do
4      $sum \leftarrow 0$ 
5     for  $j \leftarrow i$  to  $n$  do
6        $sum \leftarrow sum + A[j]$ 
7       if  $sum > sum_{max}$  then
8          $sum_{max} \leftarrow sum$ 
9          $i_{max} \leftarrow i$ 
10         $j_{max} \leftarrow j$ 
11      end
12    end
13  end
14  return ( $i_{max}, j_{max}, sum_{max}$ )
15 end

```

Algorithm 4: A brute-force method of solving the maximum-subarray problem.

Our running time should be

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

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?

Answer.

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?

Answer. Before returning the result, we can check if the sum is negative. If so, simply return an empty sub-array and a sum of 0.

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

Answer. See algorithm 5. The algorithm uses the fact that a subarray ending at index $j + 1$ is either the singleton subarray at $j + 1$ or a subarray ending at j concatenated with $j + 1$. Then the maximum subarray is either the singleton subarray or the maximum subarray ending at j concatenated with $j + 1$. The singleton subarray is greater only when the maximum subarray ending at j sums to a negative number.

```

1 def Linear-Maximum-Subarray( $A[1 \dots n]$ ):
2    $sum_{max} \leftarrow -\infty$ 
3    $sum_{end} \leftarrow -\infty$ 
4   for  $j \leftarrow 1$  to  $n$  do
5     if  $sum_{end} < 0$  then
6        $i \leftarrow j$ 
7        $sum_{end} \leftarrow A[j]$ 
8     else
9        $sum_{end} \leftarrow sum_{end} + A[j]$ 
10    end
11    if  $sum_{end} > sum_{max}$  then
12       $sum_{max} \leftarrow sum_{end}$ 
13       $i_{max} = i$ 
14       $j_{max} = j$ 
15    end
16  end
17 end

```

Algorithm 5: A nonrecursive, linear-time algorithm for the maximum-subarray problem.

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.

Answer. We have

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

Then

$$\begin{array}{lll} P_1 & = & 1 \cdot 6 = 6, \\ P_2 & = & 4 \cdot 2 = 7, \\ P_3 & = & 12 \cdot 6 = 72, \\ P_4 & = & 5 \cdot -2 = -10, \\ P_5 & = & 6 \cdot 8 = 48, \\ P_6 & = & -2 \cdot 6 = -12, \\ P_7 & = & -6 \cdot 14 = -84. \end{array}$$

Finally,

$$C = \begin{pmatrix} 48 + (-10) - 8 + (-12) & 6 + 8 \\ 72 + (-10) & 48 + 6 - 72 - (-84) \end{pmatrix} = \begin{pmatrix} 18 & 14 \\ 62 & 66 \end{pmatrix}$$

4.2-2 Write pseudocode for Strassen's algorithm.

Answer. See algorithm 6.

```

1 def Strassen-Multiply(A,B):
2   if Size(A) = 1 then
3     return  $a_{11} \cdot b_{11}$ 
4   else
5      $S_1 \leftarrow B_{12} - B_{22}$ 
6      $S_2 \leftarrow A_{11} + A_{12}$ 
7      $S_3 \leftarrow A_{21} + A_{22}$ 
8      $S_4 \leftarrow B_{21} - B_{11}$ 
9      $S_5 \leftarrow A_{11} + A_{22}$ 
10     $S_6 \leftarrow B_{11} + B_{22}$ 
11     $S_7 \leftarrow A_{12} - A_{22}$ 
12     $S_8 \leftarrow B_{21} + B_{22}$ 
13     $S_9 \leftarrow A_{11} - A_{21}$ 
14     $S_{10} \leftarrow B_{11} + B_{12}$ 
15     $P_1 \leftarrow \text{Strassen-Multiply}(A_{11}, S_1)$ 
16     $P_2 \leftarrow \text{Strassen-Multiply}(S_2, B_{22})$ 
17     $P_3 \leftarrow \text{Strassen-Multiply}(S_3, B_{11})$ 
18     $P_4 \leftarrow \text{Strassen-Multiply}(A_{22}, S_4)$ 
19     $P_5 \leftarrow \text{Strassen-Multiply}(S_5, S_6)$ 
20     $P_6 \leftarrow \text{Strassen-Multiply}(S_7, S_8)$ 
21     $P_7 \leftarrow \text{Strassen-Multiply}(S_9, S_{10})$ 
22     $C_{11} \leftarrow P_5 + P_4 - P_2 + P_6$ 
23     $C_{12} \leftarrow P_1 + P_2$ 
24     $C_{21} \leftarrow P_3 + P_4$ 
25     $C_{22} \leftarrow P_5 + P_1 - P_3 - P_7$ 
26  end
27 end

```

Algorithm 6: Strassen's algorithm.

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 $\Theta(n^{\lg(7)})$.

Answer.

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

Answer.

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

Answer.

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.

Answer. If A is a $kn \times n$ matrix and B is an $n \times kn$ matrix, then we can write

$$AB = \begin{pmatrix} A_1 \\ A_2 \\ \vdots \\ A_k \end{pmatrix} (B_1 \quad B_2 \quad \dots \quad B_k) = \begin{pmatrix} A_1 B_1 & A_1 B_2 & \dots & A_1 B_k \\ A_2 B_1 & A_2 B_2 & \dots & A_2 B_k \\ \vdots & \vdots & \ddots & \vdots \\ A_k B_1 & A_k B_2 & \dots & A_k B_k \end{pmatrix},$$

where each of A_i, B_i are $n \times n$ submatrices for $i = 1 \dots k$. There are k^2 entries in AB , each with an $n \times n$ multiplication. Using Strassen's algorithm to multiply each of these entries would take $\boxed{\Theta(k^2 n^{\lg(7)})}$ time.

To contrast, if A is a $n \times kn$ matrix and B is an $kn \times n$ matrix, then we can write

$$AB = \begin{pmatrix} A_1 & A_2 & \dots & A_k \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_k \end{pmatrix} = A_1 B_1 + A_2 B_2 + \dots + A_k B_k,$$

where each of A_i, B_i are $n \times n$ submatrices for $i = 1 \dots k$. There are k terms in AB , each with an $n \times n$ multiplication. Using Strassen's algorithm to multiply each of these terms and then sum would take $\Theta(k n^{\lg(7)}) +$

$\Theta((k-1)n^2) = \boxed{\Theta(k n^{\lg(7)})}$ time.

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 components $ac - bd$ and the imaginary component $ad + bc$ separately.

Answer. Note that the matrix product

$$\begin{pmatrix} a & b \\ a & b \end{pmatrix} \begin{pmatrix} c & d \\ -d & c \end{pmatrix} = \begin{pmatrix} ac - bd & ad + bc \\ ac - bd & ad + bc \end{pmatrix}.$$

Using Strassen's algorithm, we can compute the two entries in the anti-diagonal using four multiplications. However, we can observe that $P_2 = P_3$, so we can avoid recalculating it. We let

$$\begin{aligned} P_1 &= a(d - c), \\ P_2 &= c(a + b), \\ P_3 &= c(a + b), \\ P_4 &= -b(d + c). \end{aligned}$$

Then

$$\begin{aligned} \text{Re}((a + bi)(c + di)) &= P_3 + P_4 = P_2 + P_4 \\ &= c(a + b) - b(d + c) \\ &= ca + cb - bd - bc \\ &= ac - bd, \\ \text{Im}((a + bi)(c + di)) &= P_1 + P_2 \\ &= a(d - c) + c(a + b) \\ &= ad - ac + ca + cb \\ &= ad + bc. \end{aligned}$$

Appendix

A Summations

A.1 Summation formulas and properties

A.1-1 Find a simple formula for

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

Answer. By linearity, we have

$$\begin{aligned}\sum_{k=1}^n (2k - 1) &= 2 \sum_{k=1}^n k - \sum_{k=1}^n 1 \\ &= 2 \left(\frac{n(n+1)}{2} \right) - n \\ &= n(n+1) - n \\ &= n(n+1-1) \\ &= n^2.\end{aligned}$$

A.1-2 ★ Show that

$$\sum_{k=1}^n \frac{1}{2k-1} = \ln(\sqrt{n}) + \mathcal{O}(1)$$

by manipulating the harmonic series.

Answer. Note that $2k-1$ for $k = 1 \dots n$ assumes only all of the odd numbers between 1 and $2n$. This observation, combined with the observation that $2k$ for $k = 1 \dots n$ assumes only all of the even numbers between 1 and $2n$ leads us to the following:

$$\begin{aligned}\sum_{k=1}^n \frac{1}{2k-1} &= \sum_{k=1}^n \frac{1}{2k-1} + \sum_{k=1}^n \frac{1}{2k} - \sum_{k=1}^n \frac{1}{2k} \\ &= \sum_{k=1}^{2n} \frac{1}{k} - \frac{1}{2} \sum_{k=1}^n \frac{1}{k} \\ &= \ln(n) + \mathcal{O}(1) - \frac{1}{2}(\ln(n) + \mathcal{O}(1)) \\ &= \frac{1}{2} \ln(n) + \mathcal{O}(1) \\ &= \ln(\sqrt{n}) + \mathcal{O}(1).\end{aligned}$$

A.1-3 Show that

$$\sum_{k=0}^{\infty} k^2 x^k = \frac{x(1+x)}{(1-x)^3}$$

for $|x| < 1$.

Answer. Since $|x| < 1$, we know that the sum

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

converges. Then, we take the derivative of each side, and find

$$\begin{aligned} \sum_{k=0}^{\infty} k^2 x^{k-1} &= \frac{d}{dx} \left[\frac{x}{(1-x)^2} \right] \\ &= \frac{1(1-x)^2 - x(-2(1-x))}{(1-x)^4} \\ &= \frac{(1-2x+x^2) + (2x-2x^2)}{(1-x)^4} \\ &= \frac{1-x^2}{(1-x)^4} \\ &= \frac{(1-x)(1+x)}{(1-x)^4} \\ &= \frac{1+x}{(1-x)^3}. \end{aligned}$$

Then we have

$$\begin{aligned} \sum_{k=0}^{\infty} k^2 x^k &= x \sum_{k=0}^{\infty} k^2 x^{k-1} \\ &= x \left(\frac{1+x}{(1-x)^3} \right) \\ &= \frac{x(1+x)}{(1-x)^3}. \end{aligned}$$

A.1-4 ★ Show that

$$\sum_{k=0}^{\infty} \frac{k-1}{2^k} = 0.$$

Answer. As the sum of convergent known geometric-like series, the given series converges. Then we have

$$\begin{aligned} \sum_{k=0}^{\infty} \frac{k-1}{2^k} &= \sum_{k=0}^{\infty} \frac{k}{2^k} - \sum_{k=0}^{\infty} \frac{1}{2^k} \\ &= \sum_{k=0}^{\infty} k \left(\frac{1}{2} \right)^k - \sum_{k=0}^{\infty} \left(\frac{1}{2} \right)^k \\ &= \frac{\frac{1}{2}}{\left(1 - \frac{1}{2}\right)^2} - \frac{1}{1 - \frac{1}{2}} \\ &= 2 - 2 \\ &= 0. \end{aligned}$$

A.1-5 ★ Evaluate the sum

$$\sum_{k=1}^{\infty} (2k+1)x^{2k}$$

for $|x| < 1$.

Answer. Note that for $|x| < 1$, we also have $|x^2| < 1$. Then we have

$$\begin{aligned}
 \sum_{k=1}^{\infty} x^{2k+1} &= x \sum_{k=1}^{\infty} x^{2k} \\
 &= x \sum_{k=1}^{\infty} (x^2)^k \\
 &= x \left(\sum_{k=0}^{\infty} (x^2)^k - 1 \right) \\
 &= x \left(\frac{1}{1-x^2} - 1 \right) \\
 &= x \frac{1 - (1-x^2)}{1-x^2} \\
 &= \frac{x^3}{1-x^2}.
 \end{aligned}$$

Then, we can take derivatives of both sides, and get

$$\begin{aligned}
 \sum_{k=1}^{\infty} (2k+1)x^{2k} &= \frac{d}{dx} \left[\frac{x^3}{1-x^2} \right] \\
 &= \frac{3x^2(1-x^2) - x^3(-2x)}{(1-x^2)^2} \\
 &= \frac{3x^2 - 3x^4 + 2x^4}{(1-x^2)^2} \\
 &= \frac{x^2(3-x^2)}{(1-x^2)^2}.
 \end{aligned}$$

A.1-6 Prove that

$$\sum_{k=1}^n \mathcal{O}(f_k(i)) = \mathcal{O}\left(\sum_{k=1}^n f_k(i)\right)$$

by using the linearity property of summations.

Answer. For some sequence $\{c_k\}_{k \leq n}$, we have

$$\begin{aligned}
 \sum_{k=1}^n \mathcal{O}(f_k(i)) &\leq \sum_{k=1}^n c_k f_k(i) \\
 &\leq \sum_{k=1}^n \max_{k \leq n} \{c_k\} f_k(i) \\
 &= \max_{k \leq n} \{c_k\} \sum_{k=1}^n f_k(i),
 \end{aligned}$$

$$\text{so } \sum_{k=1}^n \mathcal{O}(f_k(i)) = \mathcal{O}\left(\sum_{k=1}^n f_k(i)\right).$$

A.1-7 Evaluate the product

$$\prod_{k=1}^n 2 \cdot 4^k.$$

Answer. As a finite product, we can “reorder” the factors using commutativity of multiplication so that all of the 2 factors come before all of the 4^k factors. This gives an identity similar to linearity of the sum:

$$\begin{aligned}
 \prod_{k=1}^n 2 \cdot 4^k &= \prod_{k=1}^n 2 \cdot \prod_{k=1}^n 4^k \\
 &= 2^n \prod_{k=1}^n 2^{2k} \\
 &= 2^n \exp\left(\ln\left(\prod_{k=1}^n 2^{2k}\right)\right) \\
 &= 2^n \exp\left(\sum_{k=1}^n \ln(2^{2k})\right) \\
 &= 2^n \exp\left(\sum_{k=1}^n 2k \ln(2)\right) \\
 &= 2^n \exp\left(2 \ln(2) \sum_{k=1}^n k\right) \\
 &= 2^n \exp(\ln(2)n(n+1)) \\
 &= 2^n \cdot 2^{n(n+1)} \\
 &= 2^{n+n(n+1)} \\
 &= 2^{n(n+2)}.
 \end{aligned}$$

A.1-8 ★ Evaluate the product

$$\prod_{k=2}^n \left(1 - \frac{1}{k^2}\right).$$

Answer. We have

$$\begin{aligned}
 \prod_{k=2}^n \left(1 - \frac{1}{k^2}\right) &= \prod_{k=2}^n \frac{k^2 - 1}{k^2} \\
 &= \prod_{k=2}^n \frac{(k-1)(k+1)}{k^2}.
 \end{aligned}$$

This is a sort of “telescoping” product, which cancels out in a similar way to a telescoping sum. We can observe this by writing out a few of the first factors and a few of the last factors:

$$\begin{aligned}
 \prod_{k=2}^n \frac{(k-1)(k+1)}{k^2} &= \left(\frac{1(3)}{2^2}\right) \left(\frac{2(4)}{3^2}\right) \left(\frac{3(5)}{4^2}\right) \cdots \left(\frac{(n-3)(n-1)}{(n-2)^2}\right) \left(\frac{(n-2)n}{(n-1)^2}\right) \left(\frac{(n-1)(n+1)}{n^2}\right) \\
 &= \left(\frac{1(\cancel{3})}{2^{\cancel{2}}}\right) \left(\frac{\cancel{2}(4)}{\cancel{3}^2}\right) \left(\frac{\cancel{3}(\cancel{5})}{\cancel{4}^2}\right) \cdots \left(\frac{(\cancel{n-3})(\cancel{n-1})}{(\cancel{n-2})^2}\right) \left(\frac{(\cancel{n-2})n}{(\cancel{n-1})^2}\right) \left(\frac{(\cancel{n-1})(n+1)}{n^{\cancel{2}}}\right) \\
 &= \frac{n+1}{2n}.
 \end{aligned}$$

A.2 Bounding Summations

A.2-1 Show that

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

is bounded above by a constant.

Answer. We wish to show that not only is the given sum bounded above by 2, but that the difference $2 - \sum_{k=1}^n \frac{1}{k^2} \geq \frac{1}{n}$. We show this by induction. We can see that for $n = 1$, we have

$$\sum_{k=1}^1 \frac{1}{k^2} = 1$$

$$\leq 2,$$

and

$$2 - 1 \geq 1.$$

Then, assume these facts are true for some n . We have

$$\begin{aligned} \sum_{k=1}^{n+1} \frac{1}{k^2} &= \sum_{k=1}^n \frac{1}{k^2} + \frac{1}{(n+1)^2} \\ &\leq \sum_{k=1}^n \frac{1}{k^2} + \frac{1}{n} \\ &\leq 2. \end{aligned}$$

As well,

$$\begin{aligned} 2 - \sum_{k=1}^{n+1} \frac{1}{k^2} &= 2 - \sum_{k=1}^n \frac{1}{k^2} - \frac{1}{(n+1)^2} \\ &\geq \frac{1}{n} - \frac{1}{(n+1)^2} \\ &= \frac{(n+1)^2 - n}{n(n+1)^2} \\ &= \frac{n^2 + n + 1}{n(n+1)^2} \\ &\geq \frac{n^2 + n}{n(n+1)^2} \\ &= \frac{n(n+1)}{n(n+1)^2} \\ &= \frac{1}{n+1}. \end{aligned}$$

Therefore, by induction, the given sum is bounded from above by 2.

A.2-2 Find an asymptotic upper bound on the summation

$$\sum_{k=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^k} \right\rceil.$$

Answer. Since $\lfloor \lg n \rfloor \leq n$ and $n \leq 2^n$, we have

$$\begin{aligned}
 \sum_{k=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^k} \right\rceil &\leq \sum_{k=0}^n \left\lceil \frac{2^n}{2^k} \right\rceil \\
 &= \sum_{k=0}^n 2^{n-k} \\
 &= 2^n \sum_{k=0}^n \left(\frac{1}{2}\right)^k \\
 &= 2^n \left(\frac{1 - \frac{1}{2^{n+1}}}{1 - \frac{1}{2}} \right) \\
 &= 2^n \left(2 - \frac{1}{2^n} \right) \\
 &= 2^{n+1} - 1.
 \end{aligned}$$

Therefore, the given sum is $\boxed{\mathcal{O}(2^n)}$. However, this is a *really* bad upper bound, and we can do better.

$$\begin{aligned}
 \sum_{k=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^k} \right\rceil &\leq \sum_{k=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{2^{\lfloor \lg n \rfloor}}{2^k} \right\rceil \\
 &= \sum_{k=0}^{\lfloor \lg n \rfloor} \frac{2^{\lfloor \lg n \rfloor}}{2^k} \\
 &= 2^{\lfloor \lg n \rfloor} \sum_{k=0}^{\lfloor \lg n \rfloor} \left(\frac{1}{2}\right)^k \\
 &= 2^{\lfloor \lg n \rfloor} \frac{1 - \frac{1}{2^{\lfloor \lg n \rfloor + 1}}}{1 - \frac{1}{2}} \\
 &= 2^{\lfloor \lg n \rfloor + 1} - 1 \\
 &\leq 2^{\lg n + 2} - 1 \\
 &= 4n - 1.
 \end{aligned}$$

Therefore, the given sum is $\boxed{\mathcal{O}(n)}$.

A.2-3 Show that the n -th harmonic number is $\Omega(\lg n)$ by splitting the summation.

Answer. We split the summation in a similar way to showing the upper bound, but we instead split the range 1 to n into $\lfloor \lg(n) \rfloor$ pieces and upperbound the contribution of each piece by $\frac{1}{2}$. See the following table for a comparison of the terms of these sums:

n	1	2	3	4	5	6		
original terms	1	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$	$\frac{1}{5}$	$\frac{1}{6}$		
upper sum	1	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$
lower sum	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{4}$					

Table 2: A comparison of the sums used to bound H_n .

Note how, since there are only $\lfloor \lg(n) \rfloor$ pieces, we aren't adding any additional terms to the sum. Then we have

$$\begin{aligned}
 \sum_{k=1}^n \frac{1}{k} &\geq \sum_{i=0}^{\lfloor \lg(n) \rfloor - 1} \sum_{j=0}^{2^i - 1} \frac{1}{2^i + j} \\
 &\geq \sum_{i=0}^{\lfloor \lg(n) \rfloor - 1} \sum_{j=0}^{2^i - 1} \frac{1}{2^{i+1}} \\
 &= \sum_{i=0}^{\lfloor \lg(n) \rfloor - 1} \frac{1}{2} \\
 &\geq \frac{1}{2}(\lg(n) - 2) \\
 &= \frac{1}{2} \lg(n) - 1.
 \end{aligned}$$

Therefore, the n -th harmonic number is $\Omega(\lg n)$.

A.2-4 Approximate

$$\sum_{k=1}^n k^3$$

with an integral.

Answer. We have

$$\begin{aligned}
 \int_0^n x^3 dx &\leq \sum_{k=1}^n k^3 \leq \int_1^{n+1} x^3 dx && \implies \\
 \frac{1}{4}n^4 &\leq \sum_{k=1}^n k^3 \leq \frac{1}{4}((n+1)^4 - 1).
 \end{aligned}$$

Therefore, the given sum is $\Theta(n^4)$.

A.2-5 Why didn't we use the integral approximation (A.12) directly on

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

to obtain an upper bound on the n -th harmonic number?

Answer. Since there is an asymptote at $x = 0$ in the function $f(x) = \frac{1}{x}$, the integral needed for the upperbound,

$$\int_0^n \frac{1}{x} dx$$

would be improper. The improper integral does not converge:

$$\begin{aligned}
 \int_0^n \frac{1}{x} dx &= \lim_{t \rightarrow 0^+} \int_t^n \frac{1}{x} dx \\
 &= \lim_{t \rightarrow 0^+} \ln(x) \Big|_{x=t}^n \\
 &= \lim_{t \rightarrow 0^+} (\ln(n) - \ln(t)) \\
 &= \infty.
 \end{aligned}$$

An upper bound of ∞ is useless.

Problems

A-1 Bounding summations. Give asymptotically tight bounds on the following summations. Assume $r \geq 0$ and $s \geq 0$ are constants.

a. $\sum_{k=1}^n k^r.$

Answer. Note that for $r \geq 0$, k^r is monotonically increasing. Then we have

$$\int_a^b x^r dx = \frac{1}{r+1} (b^{r+1} - a^{r+1}),$$

so

$$\frac{1}{r+1} n^{r+1} \leq \sum_{k=1}^n k^r \leq \frac{1}{r+1} ((n+1)^{r+1} - 1).$$

Therefore, $\sum_{k=1}^n k^r = \Theta(n^{r+1})$.

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

Answer. We can arrive at a good upper bound by simply “promoting” k to n :

$$\begin{aligned} \sum_{k=1}^n \lg^s(k) &\leq \sum_{k=1}^n \lg^s(n) \\ &= n \lg^s(n). \end{aligned}$$

So $\sum_{k=1}^n \lg^s(k) = \mathcal{O}(n \lg^s(n))$. Then, for a lower bound, we have

$$\begin{aligned} \sum_{k=1}^n \lg^s(k) &\geq \sum_{k=\lfloor n/2 \rfloor}^n \lg^s(k) \\ &\geq \sum_{k=\lfloor n/2 \rfloor}^n \lg^s\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \\ &\geq \frac{n}{2} \lg^s\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \\ &\geq \frac{n}{2} \left(\lg\left(\frac{n}{2}\right) - 1\right)^s \\ &\geq \frac{n}{2} (\lg(n) - 2)^s, \end{aligned}$$

so $\sum_{k=1}^n \lg^s(k) = \Omega(n \lg^s(n))$. By Theorem 3.1 in the book, $\sum_{k=1}^n \lg^s(k) = \Theta(n \lg^s(n))$.

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

Answer.