Introduction to Algorithms: Third Edition

Solutions

Alexander Novotny

October 10, 2019

Contents

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 espcially true for mathmetic 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

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 then 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.

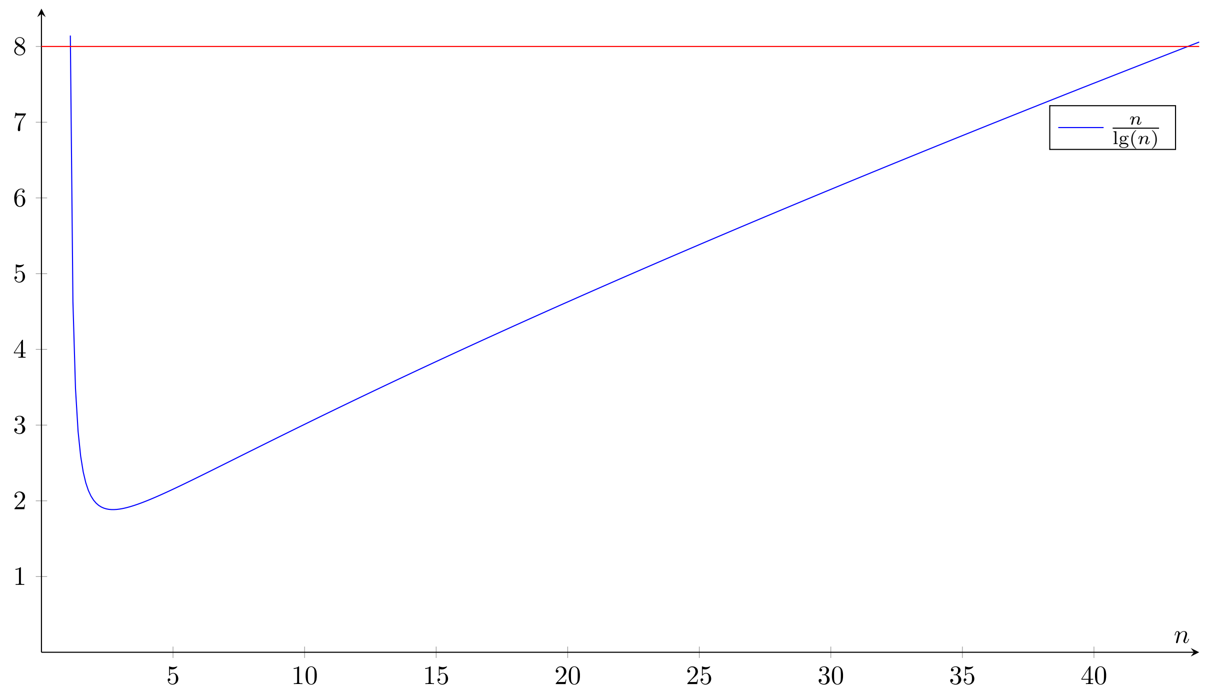
Answer. Steam, a digital retailer of video games and other software, needs algorithmic content. The platform contains tens of thousands of games and software, and if they left it up to the customer to find the games that they want, many games would go unnoticed. This means the customer doesn't get to play game 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

$$\begin{aligned} 8n^2 &< 64n\lg(n) && \Longleftarrow \\ n &< 8\lg(n) && \Longleftarrow \\ \frac{n}{\lg(n)} &< 8. \end{aligned}$$

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



Then, we can verify:

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

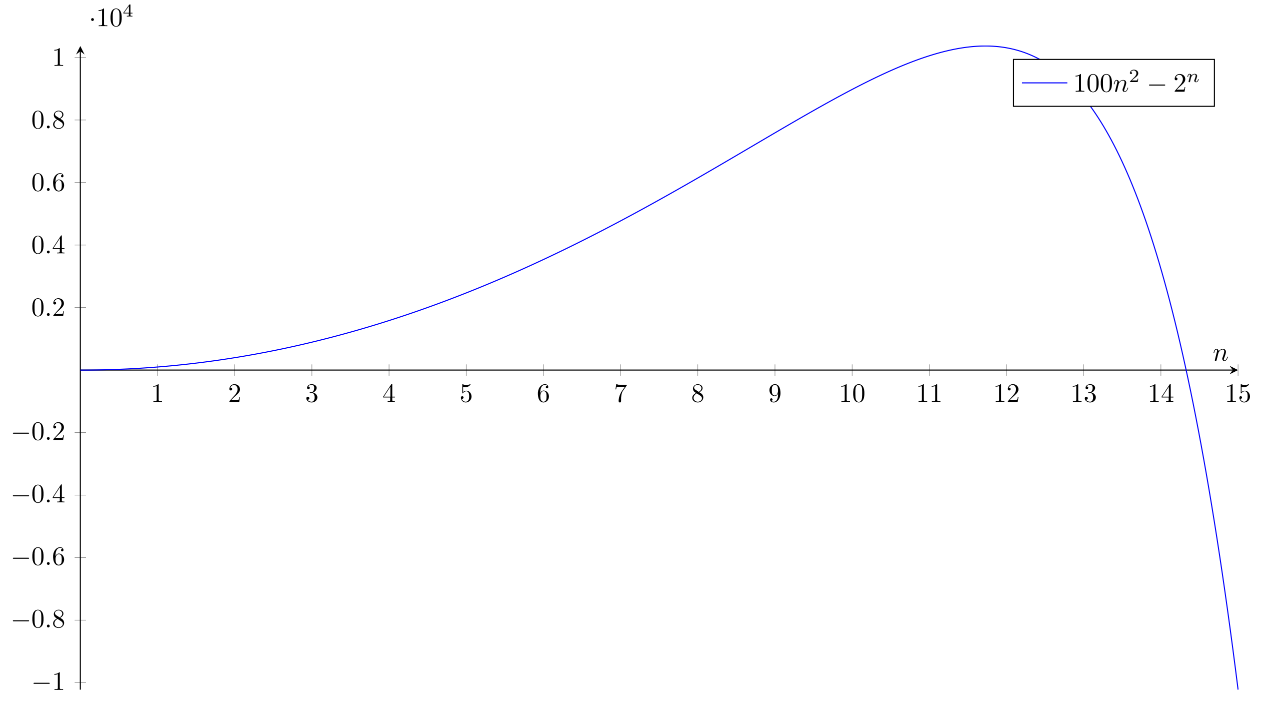
As well, we can check the in-between values by analyzing critical points. We have

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

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
$\lg(n)$	$9.9 \times 10^{301,029}$	$5.5 \times 10^{18,061,799}$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
$\sqrt{n}$	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}

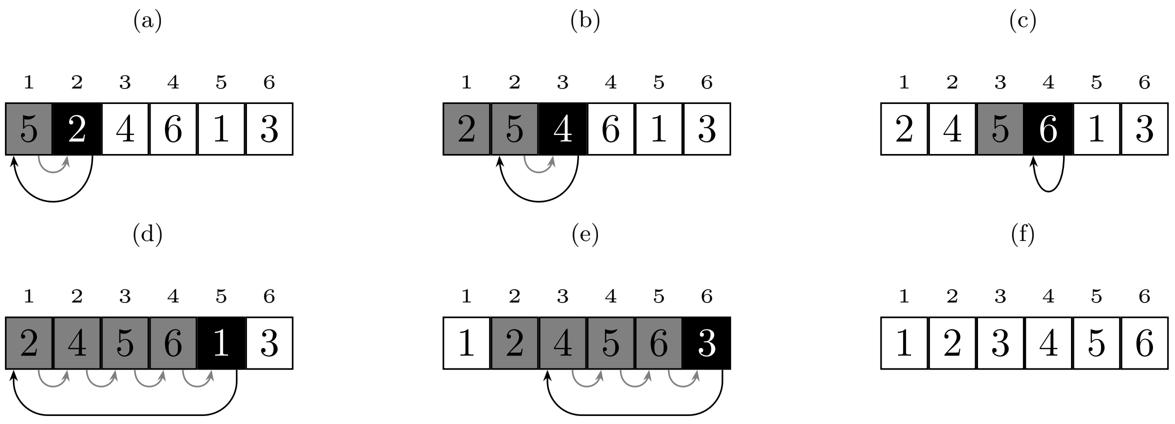
Table 1: The rest of the first row has been left out, as the numbers are too large to consider

Answer. See Table 1.

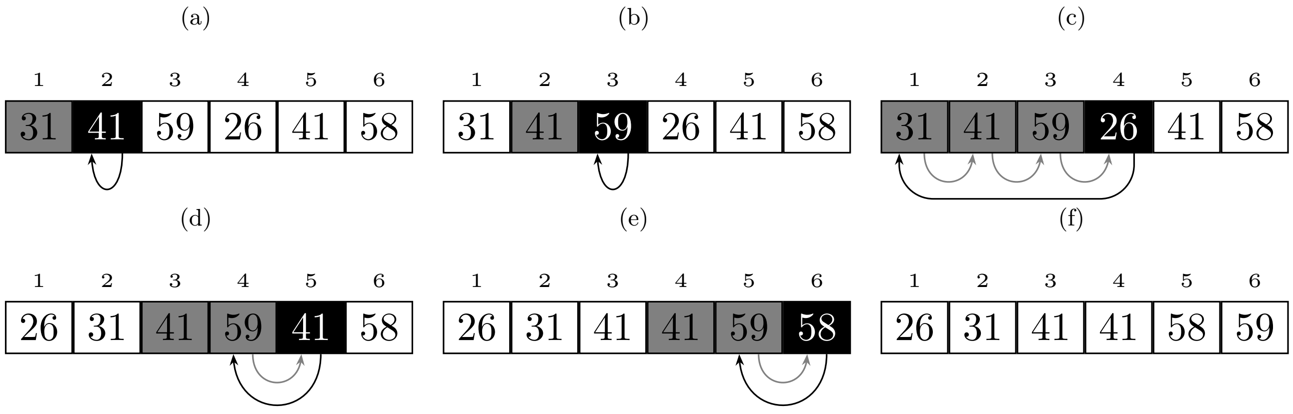
2 Getting Started

2.1 Insertion Sort

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



Answer.



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:

```
def Insertion-Sort($A[1 \dots n]$):
    for $j = n - 1$ to 1 do
        $key \leftarrow A[j]$
        $i \leftarrow j + 1$
        while $i \leq n$ and $A[i] < key$ do
            $A[i-1] \leftarrow A[i]$
            $i \leftarrow i + 1$
        end
        $A[i-1] \leftarrow key$
    end
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 $i = 1$ 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 $A[1 \ldots i-1]$ are v, and the only additional element in $A[1 \ldots i]$ 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 $A[1 \ldots i]$ does not contain v.

Termination: The loop terminates when $i = n+1$, so $A[1 \ldots i-1] = A[1 \ldots n]$ 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 $A = \langle a_1, a_2, \ldots, a_n \rangle$ and $B = \langle b_1, b_2, \ldots, b_n \rangle$ which represent binary numbers, $a_i, b_i \in \{0, 1\}$ for all $1 \leq i \leq n$.

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

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

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($A[1 \ldots n], B[1 \ldots n]$):
       // We only carry from a previous addition, which does not exist at the beginning of
          the algorithm
2      $carry \leftarrow 0$
3      for $i = 1$ 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 $C[1 \ldots (n+1)]$
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 $0 + (carry_1)2^0 = 0 + 0$, which is true, since $carry_1 = 0$.

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

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

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

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

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

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, $c_{n+1} = carry_n \in \{0, 1\}$. Then we have

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

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

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

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 $i = 1$ to $n-1$ do                              // $n$
3          $min \leftarrow i$                                // $n-1$
4          for $j = i+1$ 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. For an element in index i, there are $i-1$ possible elements to compare to (all previous elements in the array). With an equally likely chance of stopping at any one of these elements ($\frac{1}{n}$ for each), the expected number number of comparisons is $\frac{1+(i-1)}{2} = \boxed{\frac{i}{2}}$, as a uniformly distributed random variable. In the worst case, we compare against all $\boxed{i-1}$ elements. Then our expected average-case running time is

$$\begin{aligned}
n + (n-1) + (n-1) + \sum_{i=1}^{n-1} \frac{i}{2} + (n-1) &= \Theta(n) + \frac{(n-1)n}{4} \\
&= \mathcal{O}(n^2) + \Theta(n^2) \\
&= \boxed{\Theta(n^2)}.
\end{aligned}$$

Our expected worst-case running time is

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

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 $\Theta(n)$.