**Chapter 1: The roles of Algorithms in Computing**

**Exercise 1.1 – 1**

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

* English dictionary requires all the words to be arranged in their lexicographical order.
* Keeping track of the spatial extend of a disease outbreak could be done using the convex hull.

**Exercise 1.1 – 2**

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

* Memory can be a measure of efficiency that one might use in a real-world setting.

**Exercise 1.1 – 3**

**Select a data structure that you have seen previously, discuss its strengths and weaknesses.**

* Array is a container that can hold a fixed number of elements (variables of same data type) and where each element can be accessed using its index.
* Strengths:

Easy and efficient data access.

Can be used to implement other, more complex data structures.

* Weaknesses:

We must know the size of array in advance.

Does not support insertion or deletion of elements.

**Exercise 1.1 – 4**

**How are shortest-path and travelling-salesman problems given above similar? How are they different?**

* Shortest-path and travelling-salesman problems are similar because they both aim to find the shortest path from point A to point B.
* The difference is that, in shortest-path we only need to cover two points. We start at point A and end up at point B, after travelling the shortest possible distance between them.
* On the other hand, in travelling-salesman problem, we might need to cover more than two points and then end up at the point we began the journey from.

**Exercise 1.1 – 5**

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

* In case of calculating the trajectory needed to enter a geostationary orbit for satellite launch, only the best solution will do. On the other hand, when calculating age of a person, rounding down to the closest integer is a good enough solution.

**Exercise 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.**

* I

**Exercise 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 8n2 steps, while merge sort runs in 64nlgn steps. For which values of n does insertion sort beat merger sort?**

* Since n represents input size, it can only take integer values starting from 0. We need to prove:

8n2 < 64nlgn

= n < 8lgn

* This condition is satisfied only when n takes values from [2, 43] (brackets “[]” represent inclusiveness of boundaries and brackets “()” represent exclusiveness).

**Exercise 1.2 – 3**

**What is the smallest value of n such that an algorithm whose running time is 100n2 runs faster than an algorithm whose running time is 2n on the same machine?**

* We need to find the smallest value of n that satisfies the equation:

100n2 < 2n

* When n takes the value 0, the above equation is satisfied. But if we are told to find a non-zero value, then 15 will be the minimum value n can take to satisfy the equation.

**Problem 1.1**

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

* The values in the below table are approximations since I have assumed there are 30 days in every month of the year.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **1**  **second** | **1**  **minute** | **1**  **hour** | **1**  **day** | **1**  **month** | **1**  **year** | **1**  **century** |
| **lgn** | 21\*106 | 26\*107 | 23.6\*109 | 28.64\*1010 | 22.592\*1012 | 23.1104\*1013 | 23.1104\*1015 |
| **√n** | 1\*1012 | 6\*1013 | 3.6\*1015 | 8.64\*1016 | 2.592\*1018 | 3.1104\*1019 | 3.1104\*1021 |
| **n** | 1\*106 | 6\*107 | 3.6\*109 | 8.64\*1010 | 2.592\*1012 | 3.1104\*1013 | 3.1104\*1015 |
| **nlgn** |  |  |  |  |  |  |  |
| **n2** | 1000 | 7745 | 60000 | 293938 | 1609968 | 5577096 | 55770960 |
| **n3** | 100 |  |  |  |  |  |  |
| **2n** | 19 |  |  |  |  |  |  |
| **n!** | 9 |  |  |  |  |  |  |

**Chapter 2: Getting Started**

**Exercise 2.1 – 1**

**Using Figure 2.2 as a model, illustrate the operation of INSERTION-SORT on the array A = {31, 41, 59, 26, 41, 58}.**

* Changes made to the array in each iteration of for loop is as follows:

1. While loop not entered, hence no change in the input array.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 | 41 | 59 | 26 | 41 | 58 |

1. While loop not entered, hence no change in the input array.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 | 41 | 59 | 26 | 41 | 58 |

1. While loop entered, array changes after each iteration:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 | 41 | 59 | 59 | 41 | 58 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 | 41 | 41 | 59 | 41 | 58 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 | 31 | 41 | 59 | 41 | 58 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 26 | 31 | 41 | 59 | 41 | 58 |

1. While loop entered, array changes after each iteration:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 26 | 31 | 41 | 59 | 59 | 58 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 26 | 31 | 41 | 41 | 59 | 58 |

1. While loop entered, array changes after each iteration:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 26 | 31 | 41 | 41 | 59 | 59 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 26 | 31 | 41 | 41 | 58 | 59 |

**Exercise 2.1 – 2**

**Rewrite INSERTION-SORT procedure to sort into nonincreasing instead of non-decreasing order.**

* INSERTION-SORT(A)

for j = 2 to A.length

key = A[j]

i = j – 1

while i > 0 and A[i] < key

A[i + 1] = A[i]

i = i – 1

A[i + 1] = key

**Exercise 2.1 – 3**

**Consider a searching problem:**

**Input: A sequence of n numbers A = {a1, a2, …,an} 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 a pseudocode for linear search, which scans through the sequence, looking for v. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.**

* LINEAR-SEARCH(A, v)

for i = 1 to A.length

if A[i] = v

return i

return NIL

**Exercise 2.1 – 4**

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

* Input: Two n-sized arrays A and B containing bits of two binary integers a and b respectively.

Output: An array C of size (n + 1) containing bits of binary integer (a + b).

* ADD(A, B, C)

carryOver = 0

for i = n downto 1

C[i + 1] = (A[i] + B[i] + carryOver) mod 2;

if (A[i] + B[i] + carryOver) > 1

carryover = 1

else

carryover = 0

C[0] = carryOver

**Exercise 2.2 – 1**

**Express the function n3/1000 – 100n2 – 100n + 3 in terms of -notation.**

* In order to express the given function in terms of big theta notation, we need to get rid of lower order terms (100n2, 100n and 3) and coefficient attached to the leading term (1/1000). We are now left with n3, and therefore, the function can be expressed as (n3).

**Exercise 2.2 – 2**

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

* SELECTION-SORT(A)

for i = 1 to n – 1

smallestElementIndex = i

for j = i + 1 to n

if A[smallestElementIndex] > A[j]

smallestElementIndex = j

swap A[i] and A[smallestElementIndex]

* The algorithm maintains a loop invariant where after each iteration of the outer for loop, every element in the subarray A[1] … A[i] is in its correct sorted position.
* We only need to run the outer for loop n – 1 times, this is because after (n – 1)th iteration, the element at position n is the largest of all the elements of the array, an hence at its correct position.

**Exercise 2.3 – 1**

**Using figure 2.4 as a model, illustrate the operation of merge sort on the array A = {3, 41, 52, 26, 38, 57, 9, 49}.**

* Changes made to the array after each merge operation is shown below:

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

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

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

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

**Exercise 2.3 – 2**

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

* MERGE(A, p, q, r)

n1 = q – p + 1

n2 = r – q

let L[1 … n1] and R[1 … n2] be new arrays

for i = 1 to n1

L[i] = A[p + i – 1]

for j = 1 to n2

R[j] = A[q + j]

i = 1

j = 1

k = p

while i < n1 and j < n2

if L[i] < R[j]

A[k++] = L[i++]

else

A[k++] = R[j++]

while i < n1

A[k++] = L[i++]

while j < n2

A[k++] = R[j++]

**Exercise 2.3 – 3**

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

**T(n) = 2 if n = 2**

**2 T(n / 2) + n if n = 2k, for k > 1**

**Is T(n) = n lgn.**

* Using mathematical induction:
* Step 1 (base step):

For n = 21 = 2,

T(2) = 2 \* lg2

= 2 \* (1)

= 2

Therefore, condition holds for n = 2.

* Step 2 (inductive step):

Let us assume that the condition holds for n = 2k,

T(2k) = 2k \* lg2k

Let us now prove that the condition holds for n = 2k + 1,

T(2k + 1) = 2 \* T(2k + 1 / 2) + 2k + 1

= 2 \* T(2k) + 2k + 1

= 2 \* (2k \* lg2k) + 2k + 1

= 2k+1 \* lg2k + 2k + 1

= 2k + 1 \* (lg2k + 1)

= 2k + 1 \* (lg2k + lg2)

= 2k + 1 \* [lg(2k \* 2)]

= 2k + 1 \* lg2k + 1

**Exercise 2.3 – 4**

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

* INSERTION-SORT(A, index)

if index > 1

INSERTION-SORT(A, index – 1)

INSERT-ELEMENT(A, index)

INSERT-ELEMENT(A, index)

while index > 1 and A[index – 1] > A[index ]

swap A[index – 1] and A[index]

* Let the time taken to solve a problem of size “n” be T(n). Therefore, it would take T(n - 1) units of time to solve a problem of size “n - 1”. We also know that procedure “INSERT-ELEMENT” takes time (n). Using this data, we get the recurrence:
* T(n) = (1) if n = 1

T(n - 1) + (n) if n > 1

**Exercise 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 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 (lg(n)).**

* BINARY-SEARCH(A, v)

leftIndex = 1

rightIndex = A.length

while leftIndex <= rightIndex

middleIndex = ⌊(leftIndex + rightIndex) / 2⌋

if A[middleIndex] == v]

return middleIndex

if A[middleIndex] < v

leftIndex = middleIndex + 1

else

rightIndex = middleIndex – 1

return NIL

* In worst case, the value we are searching is either the first or the last element of the array. It takes lg(n) steps for this procedure to reach the first/last element, and therefore has the worst-case running time (lg(n)).

**Exercise 2.3 – 5**

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

* The while loop starting from line 5 to 7 inserts the element with index j in its correct sorted position in the sorted subarray A[1 … j - 1]. Replacing it with binary search won’t help, because we are not in search of a particular element in subarray A[1 … j - 1], but a position where jth fits.

**Exercise 2.3 – 6**

**Describe a (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.**

**Problem 2.1**

**Although merge sort runs in (n \* lg(n)) worst-case time and insertion sort runs in (n2) worst-case time, the constant factors in insertion sort can make it faster in practice for small problem sizes on many machines. Thus, it makes sense to coarsen the leaves of the recursion by using insertion sort within merge sort when subproblems become sufficiently small. Consider a modification to merge sort in which n / k sublists of length k are sorted using insertion sort and then merged using the standard merging mechanism, where k is a value to be determined.**

1. **Show that insertion sort can sort the n / k sublists, each of length k, in (n \* k) worst-case time.**
2. **Show how to merge the sublists in (n \* lg(n / k)) worst-case time.**
3. **Given that the modified algorithm runs in ((n \* k) + n \* lg(n / k)) worst-case time, what is the largest value of k as a function of n for which the modified algorithm has the same running time as standard merge sort, in terms of -notation?**
4. **How should we choose k in practice?**

* We have seen that it takes (k2) worst-case time for insertion sort to sort an array of length k. If we have n / k such arrays, it will take, ((n / k) \* k2) = (n \* k) worst-case time.
* If we make the suggested changes to our merge sort algorithm, we will be left with n / k subarrays, each of length k at the bottom most level. At this point when we stop dividing the subarrays further, we have a total of lg(n) – lg(k) = lg(n / k) levels instead of lg(n) + 1.

At each level, we perform merge operation that takes (n) time. Therefore our overall worst-case time complexity becomes (lg(n / k) \* n) = (n \* lg(n / k)).

* The largest value k can take is lg(n), when the complexity becomes (n \* lg(n) + n \* lg(n / k)). This is because, when we are only interested in the algorithm’s asymptotic behavior, we can neglect all the lower order terms like n \* lg(n / k) resulting in the final worst-case time complexity (n \* lg(n)).

**Problem 2.2**

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

**BUBBLESORT(A)**

**1 for i = 1 to A.length – 1**

**2 for j = A.length downto i + 1**

**3 if A[j] < A[j – 1]**

**4 exchange A[j] with A[j – 1]**

1. **Let A’ denote the output of BOBBLESORT(A). To prove that BUBBLESORT is correct, we need to prove that it terminates and that**

**A’[1] <= A’[2] <= … <= A’[n] (2.3)**

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

**The next two parts will prove the inequality (2.3).**

1. **State precisely a loop invariant for the for loop in lines 2 – 4, and prove that this loop invariant holds. Your proof should use the structure of the loop invariant proof present in this chapter.**
2. **Using the termination condition of the loop invariant proved in part (b), state a loop invariant for the for loop in lines 1 – 4 that will allow you to prove inequality (2.3). You proof should use the structure of the loop invariant proof presented in this chapter.**
3. **What is the worst-case running time of bubblesort? How does it compare to the running time of selection sort?**

* We need to prove that A’ is nothing but a permutation of A. In other words, we need to prove that it contains all the elements of the original array, just in a rearranged manner.
* Initialization: Before the first iteration, we start with an empty array A[1 … i - 1], which in itself is sorted.

Maintenance: After each iteration, the smallest element in the subarray A[i … n] is moved to the position A[i], which is its correct sorted position. Before the next iteration starts, the sorted subarray becomes A[1 … i] and unsorted subarray becomes A[i + 1 … n].

Termination: In the last iteration when i = n – 1 and j = n, we compare the second last and last element of the array to determine the smallest of them that would take the position n – 1. After this step all the elements have taken their sorted position in the array.

* Initialization:

Maintenance:

Termination:

* The inner for loop of bubblesort is executed (n – 1) + (n – 2) + … + 2 + 1 = [n \* (n - 1)] / 2 times. Therefore, the worst-case running time of bubblesort is (n2), same as insertion sort. The best-case running time of insertion sort in (n), but the best-case running time of bubblesort is still (n2).

**Problem 2.3**

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

**P(x) =**

**= a0 + x(a1 + x(a2 + … + x(an – 1 + xan) … )),**

**given the coefficients a0, a1, …, an and a value for x:**

**1 y = 0**

**2 for i = n downto 0**

**3 y = ai + x \* y**

1. **In terms of -notation, what is the running time of this code fragment for Horner’s rule?**
2. **Write pseudocode to implement the naïve 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?**
3. **Consider the following loop invariant:**

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

**y =**

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

1. **Conclude by arguing that the given code fragment correctly evaluates a polynomial characterized by the coefficients a0, a1, …, an.**

* The for loop is executed n + 1 times. Therefore, the running time of the above code fragment is (n).

**Problem 2.4**

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

1. **List the five inversions of the array {2, 3, 8, 6, 1}.**
2. **What array with elements from the set {1, 2, …, n} has the most inversions? How many does it have?**
3. **What is the relationship between the running time of insertion sort and the running time of inversions in the input array? Justify your answer.**
4. **Give an algorithm that determines the inversions in any permutation on n elements in (n \* lg(n)) worst-case time. (Hint: Modify merge sort.)**

* The five inversions of the array {2, 3, 8, 6, 1} are (3, 4), (1, 5), (2, 5), (3, 5) and (4, 5).
* An array that contains all the elements of set {1, 2, …, n} in descending order will have most inversions. It will have = (n \* (n – 1)) / 2 inversions.
* FIND-INVERSIONS(A)

numberOfInversions = 0

for i = 2 to A.length

j = i

while j > 0

if A[j – 1] > A[j]

numberOfInversions++

return numberOfInversions

The while loop is executed = (n \* (n – 1)) / 2 times, giving us the running time (n2). For each iteration of for loop, while loop is executed i – 1 times. The order of the input array has no impact on its running time and thus we conclude its best-case running time is same as its worst-case running time (n2).

This is not the case in insertion sort. The order of elements of the input array has an impact on its running time. If the input array is already sorted, it runs in (n) time, which is its best case.

**Chapter 3: Growth of Functions**

**Exercise 3.1 – 1**

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

* To show that max(f(n), g(n)) = (f(n) + g(n)), we need to show that there exists two constants c1 and c2 such that for n >= n0,

0 <= c1 \* (f(n) + g(n)) <= max(f(n), g(n)) <= c2 \* (f(n) + g(n))

The above conditions are satisfied when c2 >= 1 and when c1 <= 0.5.

* Therefore, max(f(n), g(n)) = (f(n) + g(n)).

**Exercise 3.1 – 2**

**Show that for any real constants a and b, where b > 0, (n + a)b = (nb).**

* (n + a)b = nb(1 + (a / n))b

But a / n 0 for some constant a and large values of n.

Therefore, nb(1 + (a / n))b nb

* Now, we have to show that there exists two constants c1 and c2 such that for all n >= n0,

0 <= c1 \* nb <= nb <= c2 \* nb

c1 = 1 and c2 >= 1 satisfy the above conditions.

**Exercise 3.1 – 3**

**Explain why the statement, “The running time of algorithm A is at least O(n2)”, is meaningless.**

* Let us represent running time of the algorithm A as T(n). It is given that T(n) >= O(n2). But by definition of O-notation, T(n) <= c \* n2.
* The two equations contradict each other, and hence the statement does not make sense.

**Exercise 3.1 – 4**

**Is 2n + 1 = O(2n)? Is 22n = O(2n)?**

* When we say 2n + 1 = 2 \* 2n = O(2n), we mean there exists a constant c such that when n >= n0,

c \* 2n >= 2 \* 2n

For all the values of c > 1, the above equation holds, therefore 2n + 1 = O(n2).

* Similarly, we need to prove

c \* 2n >= 22n

No value of c can satisfy the above equation and therefore 22n des not belong to the set O(n2).

**Exercise 3.1 – 5**

**Prove Theorem 3.1.**

* Theorem 3.1 says that for any two functions f(n) and g(n), we have f(n) = (g(n)) if and only if f(n) = O(g(n)) and f(n) = Ω(g(n)).
* We know f(n) = (g(n)), it means there exists a constant c1 such that for n >= n0,

c1 \* g(n) <= f(n)

Therefore, by the definition of Ω-notation f(n) = Ω(g(n)).

* There also exists a constant c2 such that for n >= n0,

f(n) <= c2 \* g(n)

Therefore, by the definition of O-notation f(n) = O(g(n)).

**Exercise 3.1 – 6**

**Prove that running time of an algorithm is (g(n)) if and only if its worst-case running time is O(g(n)) and its best-case running time is Ω(g(n)).**

* A

**Exercise 3.1 – 7**

**Prove that o(g(n)) (g(n)) is the empty set.**

* By definition, f(n) = o(g(n)) if there exists a constant c such that for all n >= n0,

f(n) < c \* g(n)

Meaning, every function in the set o(g(n)) lie above function f(n).

* On the other hand, by definition, f(n) = (g(n)) if there exists a constant c such that for all n >= n0,

c \* g(n) < f(n)

Meaning, every function in the set (g(n)) lie below function f(n).

* Therefore the two sets do not contain a single common function.

**Exercise 3.1 – 8**

**We can extend out notation tot 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 O(g(n, m)) the set of functions**

**O(g(n, m)) = f(n, m): there exists positive constants c, n0, and m0 such that 0 <= f(n, m) <= c \* g(n, m) for all n >= n0 or m <= m0 .**

**Give corresponding definitions for Ω(g(n, m)) and (g(n, m)).**

**Chapter 4: Divide-and-Conquer**

**Exercise 4.1 – 1**

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

* When all the elements of A are negative, subroutine FIND-MAXIMUM-SUBARRAY returns a tuple containing index of the smallest element in the array and its value.

**Exercise 4.1 – 2**

**Write pseudocode for the brute-force method for solving the maximum-subarray problem. Your procedure should run in (n2) time.**

* FIND-MAXIMUM-SUBARRAY