**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.**