**Assignment**

**Course Name: Design and Analysis of Algorithms**

**Course Code : CSE-2201**

**Topic : Algorithm**

**Submitted By: Abu Sayeed Md Afridi**

**Roll : 18CSE015**

**Session : 2017-18**

**Semester : 4th**

**Date : 02/02/2020**

**CHAPTER-1**

**The Role of Algorithms in Computing**

**Summary 1.1**

An algorithm is a computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. There are many types of problems that can be solved efficiently by algorithms. It is the roadmap of a code.

**Exercise 1.1**

1.1-1) An example of a real world situation that would require sorting would be if you wanted to keep track of a bunch of people’s file folders and be able to look up a given name quickly. A convex hull might be needed if you needed to secure a wildlife sanctuary with fencing and had to contain a bunch of specific nesting locations.

1.1-2) One might measure memory efficiency of an algorithm.

1.1-3) An array. It has the limitation of requiring a lot of copying when resizing, inserting, and removing elements.

1.1-4) They are similar since both problems can be modeled by a graph with weighted edges and involve minimizing distance, or weight, of a walk on the graph. They are different because the shortest path problem considers only two vertices, whereas the traveling salesman problem considers minimizing the weight of a path that must include many vertices and end where it began.

1.1-5) If you were for example keeping track of terror watch suspects, it would be unacceptable to have it occasionally bringing up a wrong decision as to whether a person is on the list or not. It would be fine to only have an approximate solution to the shortest route on which to drive, an extra little bit of driving is not that bad.

**summary-1.2**

An algorithm is very useful because it is the procedure of a computer program regardless of capability of our computer. As our computer is not infinitely fast and does not have infinite memory, algorithm is an asset as technology to solve some problems through limited resources. It makes the use of hardware easier.

**Exercise 1.2**

1.2-1) A program that would pick out which music a user would like to listen to next. They would need to use a bunch of information from historical and popular preferences in order to maximize.

1.2-2) We wish to determine for which values of n the inequality 8(n^2)< 64nlog2(n) holds. This happens when n<8log2(n), or when n ≤ 43. In other words, insertion sort runs faster when we’re sorting at most 43 items. Otherwise merge sort is faster.\

**Exercise 1.2-3**

We want that 100n2 < 2n. note that if n = 14, this becomes 100(14)2 = 19600 > 214 = 16384. For n = 15 it is 100(15)2 = 22500 < 215 = 32768. So, the answer is n = 15.

**Summary-2.1**

Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands.

Likewise :

// We have to sort an arr[] of size n

insertionSort(arr, n)

Loop from i = 1 to n-1.

Now, pick element arr[i] and insert it into sorted sequence arr[0…i-1]

12, 11, 13, 5, 6

* Let us loop for i = 1 (second element of the array) to 4 (last element of the array)
* i = 1. Since 11 is smaller than 12, move 12 and insert 11 before 12

11, 12, 13, 5, 6

* i = 2. 13 will remain at its position as all elements in A[0..I-1] are smaller than 13

11, 12, 13, 5, 6

* i = 3. 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.

5, 11, 12, 13, 6

* i = 4. 6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.

5, 6, 11, 12, 13

So the array is sorted!

**Exercise 2.1-1**

1. 31 41 59 26 41 58
2. 31 41 59 26 41 58
3. 31 41 59 26 41 58
4. 26 31 41 59 41 58
5. 26 31 41 41 59 58
6. 26 31 41 41 58 59

**Exercise 2.1-2**

Algorithm 1 Nonincreasing Insertion-Sort(A)

1: for j = 2 to A.length do

2: key = A[j]

3: // Insert A[j] into the sorted sequence A[1..j −1].

4: i = j −1

5: while i > 0 and A[i] < key do

6: A[i + 1] = A[i]

7: i = i−1

8: end while

9: end for

10: A[i + 1] = key

**Exercise 2.1-3**

On each iteration of the loop body, the invariant upon entering is that there is no index k < j so that A[k] = v. In order to proceed to the next iteration of the loop, we need that for the current value of j, we do not have A[j] = v. If the loop is exited by line 5, then we have just placed an acceptable value in i on the previous line. If the loop is exited by exhausting all possible values of j, then we know that there is no index that has value j, and so leaving NIL in i is correct.

1: i = NIL

2: for j = 1 to A.length do

3: if A[j] = v then

4: i = j

5: return i

6: end if

7: return i

8: end for

**Exercise 2.1-4**

Input: two n-element arrays A and B containing the binary digits of two numbers a and b. Output: an (n + 1)-element array C containing the binary digits of a + b.

Algorithm 2 Adding n-bit Binary Integers

1: carry = 0

2: for i=1 to n do

3: C[i] = (A[i] + B[i] + carry) (mod 2)

4: if A[i] + B[i] + carry ≥ 2 then

5: carry = 1

6: else

7: carry = 0

8: end if

9: end for

10: C[n+1] = carry

**Summary-2.2**

Analyzing insertion sort, we get these in brief :

Time Complexity: O(n\*2)

Auxiliary Space: O(1)

Boundary Cases: Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.

Algorithmic Paradigm: Incremental Approach

Sorting In Place: Yes

Stable: Yes

Online: Yes

Uses: Insertion sort is used when the number of elements is small. It can also be useful when input array is almost sorted, only few elements are misplaced in complete big array.

**Exercise 2.2-1**

n3/1000−100n2 −100n + 3 ∈ Θ(n3)

**Exercise 2.2-2**

Input: An n-element array A.

Output: The array A with its elements rearranged into increasing order. The loop invariant of selection sort is as follows:

At each iteration of the for loop of lines 1 through 10, the subarray A[1..i−1] contains the i−1 smallest elements of A in increasing order. After n−1 iterations of the loop, the n−1 smallest elements of A are in the ﬁrst n−1 positions of A in increasing order, so the nth element is necessarily the largest element. Therefore we do not need to run the loop a ﬁnal time. The best-case and worst-case running times of selection sort are Θ(n2). This is because regardless of how the elements are initially arranged, on the ith iteration of the main for loop the algorithm always inspects each of the remaining n−i elements to ﬁnd the smallest one remaining.

Algorithm 3 Selection Sort

1: for i = 1 to n−1 do

2: min = i

3: for j = i + 1 to n do

4: // Find the index of the ith smallest element

5: if A[j] < A[min] then

6: min = j

7: end if

8: end for

9: Swap A[min] and A[i]

10: end for

This yields a running time Θ(n2).

**Exercise 2.2-4**

For a good best-case running time, modify an algorithm to the ﬁrst randomly produce output and then check whether or not it satisﬁes the goal of the algorithm. If so, produce this output and halt. Otherwise, run the algorithm as usual. It is unlikely that this will be successful, but in the best-case, the running time will only be as long as it takes to check a solution. For example, we could modify selection sort to ﬁrst randomly permute the elements of A, then check if they are in sorted order. If they are, output A. Otherwise run selection sort as usual. In the best case, this modiﬁed algorithm will have running time Θ(n).

**Summary-2.3**

Divide and Conquer is an algorithmic paradigm. A typical Divide and Conquer algorithm solves a problem using these following three steps.

1. Divide: Break the given problem into subproblems of the same type.

2. Conquer: Recursively solve these subproblems

3. Combine: Appropriately combine the answers

Likewise merge sort is a sorting algorithm that divides the array into two halves, recursively sorts them and finally merges the two sorted halves.

**Exercise 2.3-1**

If we start with reading across the bottom of the tree and then go up level by level.

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**

The following is a rewrite of MERGE which avoids the use of sentinels. Much like MERGE, it begins by copying the subarrays of A to be merged into arrays L and R. At each iteration of the while loop starting on line 13 it selects the next smallest element from either L or R to place into A. It stops if either L or R runs out of elements, at which point it copies the remainder of the other subarray into the remaining spots of A.

Algorithm 4 Merge(A,p,q,r)

1: n1 = q−p + 1

2: n2 = r−q

3: let L[1,..n1] and R[1..n2] be new arrays

4: for i = 1 to n1 do

5: L[i] = A[p + i−1]

6: end for

7: for j = 1 to n2 do

8: R[j] = A[q + j]

9: end for

10: i = 1

11: j = 1

12: k = p

13: while i 6= n1 + 1 and j 6= n2 + 1 do

14: if L[i] ≤ R[j] then

15: A[k] = L[i]

16: i = i + 1

17: else A[k] = R[j]

18: j = j + 1

19: end if

20: k = k + 1

21: end while

22: if i == n1 + 1 then

23: for m = j to n2 do

24: A[k] = R[m]

25: k = k + 1

26: end for

27: end if

28: if j == n2 + 1 then

29: for m = i to n1 do

30: A[k] = L[m]

31: k = k + 1

32: end for

33: end if

**Exercise 2.3-4**

Let T(n) denote the running time for insertion sort called on an array of size n. We can express T(n) recursively as T(n) = Θ(1) if n ≤ c T(n−1)I(n) otherwise where I(n) denotes the amount of time it takes to insert A[n] into the sorted array A[1..n−1]. As seen in exercise 2.3-5, I(n) is Θ(logn).

1: Use Merge Sort to sort the array A in time Θ(nlg(n))

2: i = 1

3: j = n

4: while i < j do

5: if A[j] + A[j] = S then

6: return true

7: end if

8: if A[i] + A[j] < S then

9: i = i + 1

10: end if : if A[i] + A[j] > S then

12: j = j −1

13: end if

14: end while

15: return false

**Summary-3.1**

The main idea of asymptotic analysis is to have a measure of efficiency of algorithms that doesn’t depend on machine specific constants, and doesn’t require algorithms to be implemented and time taken by programs to be compared. Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis. The following 3 asymptotic notations are mostly used to represent time complexity of algorithms.

* Θ Notation: The theta notation bounds a functions from above and below, so it defines exact asymptotic behavior.
* Big O Notation: The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in the best case and quadratic time in the worst case. We can safely say that the time complexity of Insertion sort is O(n^2). Note that O(n^2) also covers linear time.

The Big O notation is useful when we only have upper bound on time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

* Ω Notation: Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.

Ω Notation can be useful when we have a lower bound on the time complexity of an algorithm. As discussed in the previous post, the best case performance of the algorithm is generally not useful, the Omega notation is the least used notation among all three.

**Exercise 3.1-1**

Since we are requiring both f and g to be asymptotically non-negative, suppose that we are past some n1 where both are non-negative (take the max of the two bounds on the n corresponding to both f and g). Let c1 = .5 and c2 = 1. 0 ≤ .5(f(n) + g(n)) ≤ .5(max(f(n),g(n)) + max(f(n),g(n))) = max(f(n),g(n)) ≤ max(f(n),g(n)) + min(f(n),g(n)) = (f(n) + g(n))

**Exercise 3.1-3**

There are a tons of diﬀerent functions that have growth rate less than or equal to n2. In particular, functions that are constant or shrink to zero arbitrarily fast. Saying that you grow more quickly than a function that shrinks to zero quickly means nothing.

**Exercise 3.1-4**

2n+1 ≥ 2·2n for all n ≥ 0, so 2n+1 = O(2n). However, 22n is not O(2n). If it were, there would exist n0 and c such that n ≥ n0 implies 2n·2n = 22n ≤ c2n, so 2n ≤ c for n ≥ n0 which is clearly impossible since c is a constant.

**Exercise 3.1-6**

Suppose the running time is Θ(g(n)). By Theorem 3.1, the running time is O(g(n)), which implies that for any input of size n ≥ n0 the running time is bounded above by c1g(n) for some c1. This includes the running time on the worst-case input. Theorem 3.1 also imlpies the running time is Ω(g(n)), which implies that for any input of size n ≥ n0 the running time is bounded below by c2g(n) for some c2. This includes the running time of the best-case input. On the other hand, the running time of any input is bounded above by the worst-case running time and bounded below by the best-case running time. If the worst-case and best-case running times are O(g(n)) and Ω(g(n)) respectively, then the running time of any input of size n must be O(g(n)) and Ω(g(n)).

**Exercise 3.1-8**

Ω(g(n,m)) = {f(n,m) : there exist positive constants c,n0, and m0 such that f(n,m) ≥ cg(n,m) for all n ≥ n0 or m ≥ m0}

Θ(g(n,m)) = {f(n,m) : there exist positive constants c1,c2,n0, and m0 such that c1g(n,m) ≤ f(n,m) ≤ c2g(n,m) for all n ≥ n0 or m ≥ m0}

**Exercise 3.2-1**

Let n1 < n2 be arbitrary. From f and g being monotonic increasing, we know f(n1) < f(n2) and g(n1) < g(n2). So f(n1) + g(n1) < f(n2) + g(n1) < f(n2) + g(n2)

Since g(n1) < g(n2), we have f(g(n1)) < f(g(n2)).

Lastly, if both are non negative, then, f(n1)g(n1) = f(n2)g(n1) + (f(n2)−f(n1))g(n1) = f(n2)g(n2) + f(n2)(g(n2)−g(n1)) + (f(n2)−f(n1))g(n1)

Since f(n1) ≥ 0, f(n2) > 0, so, the second term in this expression is greater than zero. The third term is nonnegative, so, the whole thing is< f(n2)g(n2).

**Exercise 3.2-5**

Note that lg∗(2n) = 1 + lg∗(n), so,

lim n→∞ lg(lg∗(n))/lg∗(lg(n))

= lim n→∞ lg(lg∗(2n))/lg∗(lg(2n))

= lim n→∞ lg(1 + lg∗(n)) /lg∗(n)

= lim n→∞ lg(1 + n)/ n

= lim n→∞ 1/1 + n

= 0

So, we have that lg∗(lg(n)) grows more quickly.

**Exercise 3.2-6**

φ2 = 1 +√5 2 !2 = 6 + 2√5 4

= 1 +1 +√5 2

= 1 + φˆ φ2 = 1−√5 2 !2 = 6−2√5 4

= 1 +1−√5 2

= 1 + ˆ φ

**Exercise 3.2-7**

First, we show that 1 + φ = 6+2√5 4 = φ2. So, for every i, φi−1 + φi−2 = φi−2(φ + 1) = φi.

Similarly for ˆ φ. For i = 0, φ0−ˆ φ0 √5 = 0.

For i = 1, 1+√5 2 −1−√5 2√ 5 = √5 √5 = 1.

Then, by induction, Fi = Fi−1 + Fi−2 = φi−1+φi−2−(ˆ φi−1+ˆ φi−2) √5 = φi−ˆ φi √5 .