ADVANCED ALGORITHM ANALYSIS

Final Term Exam

Q1. Analyze Bubble Sort.

Answer:

The definition of Bubble sort is a really weird name but this algorithm actually bubbles up the largest element at the end after each iteration and that's why the name is Bubble Sort. The following is the analyze bubble sort.

Non-optimized version of the Bubble sort.

Regardless of the input, the two loops and the if statements are going to execute every time. Only the execution of the swap statement will depend upon the input.

the first for loop (for i in 1 to A.length) is going to execute n+1 times (n is the size of the array 'A' and 1 extra time when the condition of the loop will be check and failed) and the second for loop is going to execute (n-i)+1 times for every iteration of the outer loop. For example, it is going to execute (n-1) +1 times for the first iteration of the outer loop (when i is 1), (n-2) +1 times in the second iteration of the outer loop (i is 2) and so on. So, it will run a total of (n) + (n−1) +...+1=(n)(n+1)2 time.

Now, the if statement is going to run every time the statements of inner loop are going to be executed i.e., (n−1)+(n−2)+...+1=(n−1)(n)2.

Till now, we know that our running time is going to be quadratic regardless of the input. The swap statement can run a minimum of 0 times when the array is already sorted or a maximum of (n−1)(n)2

Times (equal to the number of times if is executed). In both cases, the rate of the growth of our function will be quadratic because we already have quadratic terms in our function ((n)(n+1)2 and (n−1)(n)2). So, the running time of this algorithm is Θ(n2)

.

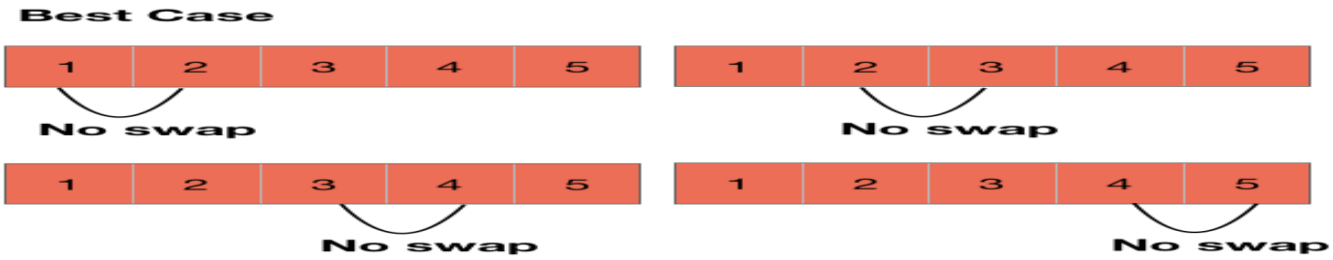
Optimized Bubble Sort

In the worst case, there won't be any difference between the optimized and the non-optimized versions because the worst case will be when the break statement never gets executed and this is going to occur when the array is sorted in the reverse order.

Thus, the optimized version has O (n2)

Worst case running time.

Now, let's talk about the best case. The best case would be when the outer for loop (for i in 1 to A.length) just breaks after its first iteration. And this can occur when there is nothing swapped inside the second loop which means the array is already sorted.



In this case, the outer for loop will run only one time and the inner for loop will run n times and then the algorithm will stop. So, the best case running time of this algorithm is going to be a linear one and thus this algorithm has Ω (n)

Thus, we have seen two sorting algorithms till now. Both of these algorithms are coding implementations of how we basically sort in our real life. And this is exactly what we need to develop our algorithms.

Complexity Analysis of Bubble Sort

In Bubble Sort, n-1 comparisons will be done in the 1st pass, n-2 in 2nd pass, n-3 in 3rd pass and so on. So the total number of comparisons will be,

Out put

(n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1

Sum = n(n-1)/2

i.e $O(n^2)$

Hence the **time complexity** of Bubble Sort is **$O(n^2)$**.

The main advantage of Bubble Sort is the simplicity of the algorithm.

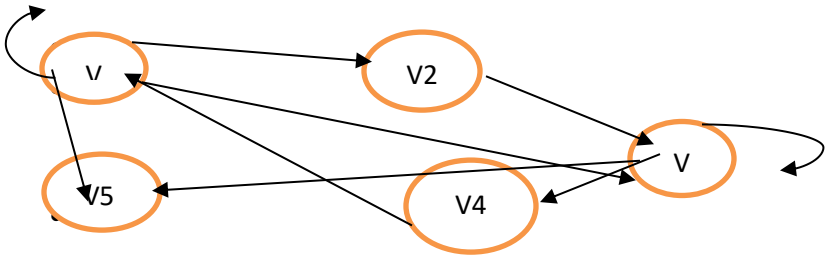
The **space complexity** for Bubble Sort is **O(1)**, because only a single additional memory space is required i.e. for temp variable.

Also, the **best case time complexity** will be **O(n)**, it is when the list is already sorted.

Following are the Time and Space complexity for the Bubble Sort algorithm.

- Worst Case Time Complexity [Big-O]: **$O(n^2)$**
- Best Case Time Complexity [Big-omega]: **O(n)**
- Average Time Complexity [Big-theta]: **$O(n^2)$**
- Space Complexity: **O(1)**

Q2. Design an Adjacency Matrix for the given graph. (12 Marks)



Answer:

Adjacency Matrix

In graph theory and computer science, an **adjacency matrix** is a square **matrix** used to represent a finite graph. The elements of the **matrix** indicate whether pairs of vertices are adjacent or not in the graph. In the special case of a finite simple graph, the **adjacency matrix** is a (0,1)-**matrix** with zeros on its diagonal.

How to create an adjacency matrix from a graph?

The **graph** family argues that one of the best ways to represent them into a **matrix** is by counting the number of edge between two adjacent vertices. Two vertices is said to be adjacent or neighbor if it support at least one common edge. To fill the **adjacency matrix**, we look at the name of the vertex in row and column.

For the above

	V1	V2	V3	V4	V5
V1	0	1	0	1	0
V2	0	0	0	0	0
V3	0	1	0	0	0
V4	0	0	0	0	0
V5	0	0	0	1	0

The above matrix indicates that we can go from vertex v1 to vertex v2, or from vertex v1 to vertex v4 in two moves. In fact, if we examine the graph, we can see that this can be done by going through vertex v5 and through vertex v2 respectively. We can also reach vertex v2 from v3, and vertex v4 from v5, all in two moves.

In general, to generate the matrix of path of length n, take the matrix of path of length n-1, and multiply it with the matrix of path of length 1.

Q3. Consider the given Adjacency Matrix and design the graph. (12 Marks)

$$A = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{bmatrix}$$

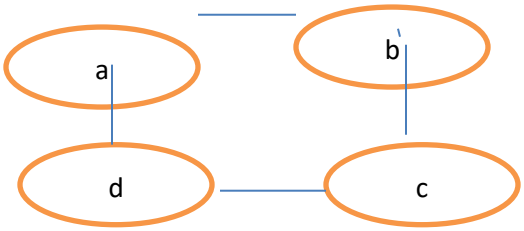
Answer:

Adjacency Matrix of a Graph

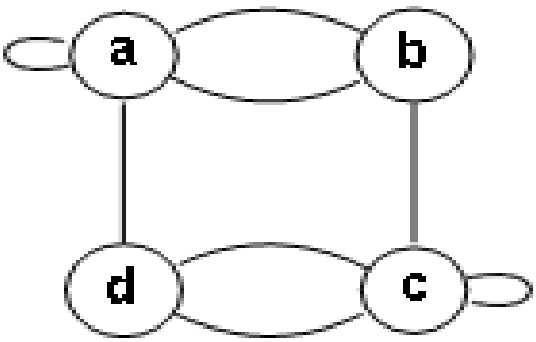
The adjacency matrix, sometimes also called the connection matrix, of a simple labeled graph is a matrix with rows and columns labeled by graph vertices, with a 1 or 0 in position according to weather and. are adjacent or not. For a simple graph with no self-loops, the adjacency matrix must have 0s on the diagonal.

Two vertices is said to be adjacent or neighbor if it support at least one common edge. To fill the adjacency matrix, we look at the name of the vertex in row and column. If those vertices are connected by an edge or more, we count number of edges and put this number as matrix element.

Let the *adjacency matrix AG* = [*aij*] of a graph *G* is the *n* × *n* (*n* = *jV j*) zero-one matrix, where *aij* = 1 if *fvi; vjg* is an edge of *G*, and is 0 otherwise



Can extend to graphs with loops and multiple edges by letting each matrix elements be number of links (possibly > 1) between the nodes.



Q4. Design a Heap and then Sort using Heap Sort.

10, 5, 31, 7, 11, 0

Answer:

Heap Sort is a popular and efficient sorting algorithm in computer programming. Learning how to write the heap sort algorithm requires knowledge of two types of data structures - arrays and trees.

Design a Heap

Step 1 − Create a new node at the end of heap.

Step 2 − Assign new value to the node.

Step 3 − Compare the value of this child node with its parent.

Step 4 − If value of parent is less than child, then swap them.

Heap Sort

1. Algorithm :
2. Build a min heap from the input data.
3. At this point, the smallest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
4. Repeat above steps while size of heap is greater than 1.

   The initial set of numbers that we want to sort is stored in an array e.g.

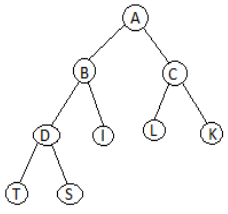
   10, 5, 31, 7, 11, 0

   and after sorting, we get a sorted array

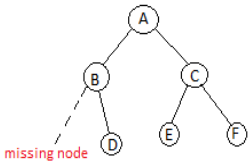
   0,5,7,10,11,31

   Heap sort works by visualizing the elements of the array as a special kind of complete binary tree called a heap.

Heap as a Tree root of tree:

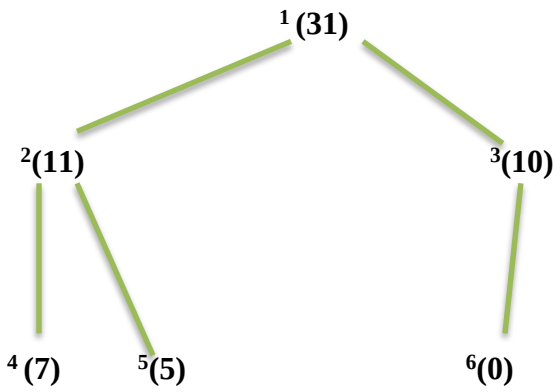


Complete Binary Tree



In-Complete Binary Tree

first element in the array, corresponding to i = 1 parent(i) =i/2: returns index of node's parent left(i)=2i: returns index of node's left child right(i)=2i+1: returns index of node's right child



……..