Algorithms & Data Structures - 2022/2023

1. (a) Differences between Heap Array and sorted array?

→ Heap array and sorted array are two different data structures. In a sorted array elements are arranged in a particular order - such as ascending or descending order. A heap array on the other hand won't necessarily be sorted in an ascending/descending manner but instead maintains a specific order property between parent and child nodes (the relationship has to be maintained as heap is a tree-based data structure.

An insert() operation would operate as →

- Unsorted array - Adds new element at end of array
- Sorted array - Parses through array to find suitable place for the new element.
- Binary Heap ~~Unsorted Array~~ - The new element is placed at the next available location. To maintain the structure of heap the element is 'sifted up' accordingly.

Remove (n) operation would operate as →

- Unsorted array - 'N' element is found in array after parsing through array and is removed.
- Sorted array - 'N' element is found in array after parsing through array and is removed. Elements are shifted to maintain order.

- Binary Heap - The nth element (in max- or min heap) is removed and the heap property is restored by performing 'sift-down'.

Both insert and remove operations are more efficient with binary heap than with a sorted and unsorted array. Both have time complexity of $O(\log n)$ which for the array is upto $O(n)$.

1(b) Selection sort and heap sort are examples of "hard-split easy join". comment.

→ "Hard split easy join" refers to algorithms that split an array into smaller parts and merge the 'sorted parts' in a whole array.

- Selection sort divides an array into sorted and unsorted arrays. It finds the smallest element in the unsorted array and swaps it to the first place. It then places it in the sorted array and continues till the unsorted array is empty.
- Heap sort divides sorting in 2 steps. First being building a max or min heap with the parent and children nodes. After the input array is resolved into a heap structure, the second step starts. "easy-join" involves repeatedly deleting elements and placing at the end of an array - until the heap is empty. This also results in a sorted array.

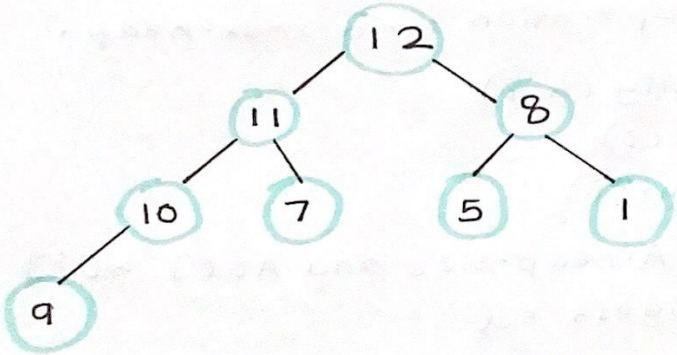
Although both these sorting algorithms are very different - they both use "hard split - easy join".

1(c) Draw heap array as binary tree. Also record hPos[] for heap.

→

k	0	1	2	3	4	5	6	7	8	9	10	11	12
a[k]		12	11	8	10	7	5	1	9				
hPos[]	null	0	1	2	3	4	5	6	7				

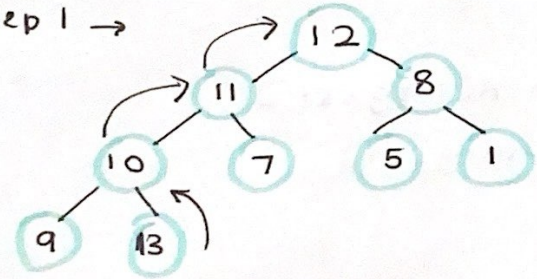
Binary Tree



This is a max heap!

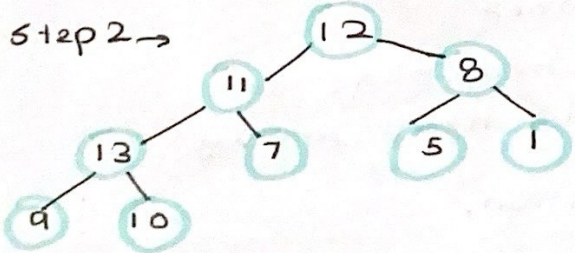
1(d) Illustrate effect of adding 13 in tree(heap).

Step 1 →



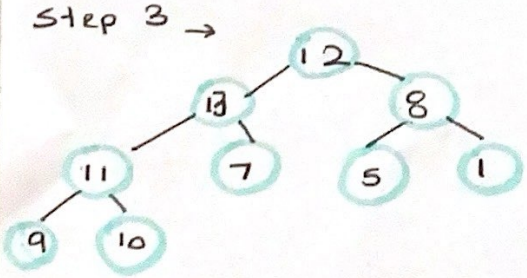
k	0	1	2	3	4	5	6	7	8	9	10	11
a[k]	12	11	8	10	7	5	1	9	13			

Step 2 →



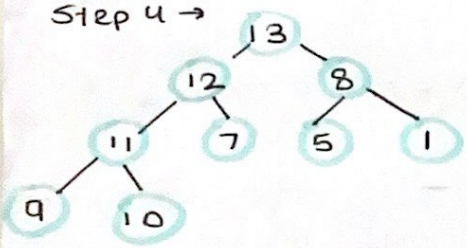
k	0	1	2	3	4	5	6	7	8	9	10	11
a[k]	12	11	8	13	7	5	1	9	10			

Step 3 →



k	0	1	2	3	4	5	6	7	8	9	10	11
a[k]	12	13	8	11	7	5	1	9	10			

Step 4 →



k	0	1	2	3	4	5	6	7	8	9	10	11
a[k]	13	12	8	11	7	5	1	9	10			

1(e) Pseudocode for maxHeapify(int k) of siftDown(int k) heap operation.

→ Max-Heapify pseudocode or essentially the SiftDown operation for a max-heap.

```
Max-Heapify (A, i)
    l = left(i)
    r = right(i)
    if l ≤ A.heap-size and A[l] > A[i]
        largest = l
    else
        largest = i
    if largest ≠ i
        swap A[i] with A[largest]
        Max-Heapify (A, largest)
```

- k = heap position
- a[k] = index in array for heap
- dist [a[k]] = priority in heap node at position k in heap array

not related to question.

2(a) Show first two passes of bubble sort for the given array.

→

6	3	1	9	8	2	4	7	0	5

First pass →

3	1	6	8	2	4	7	0	5	9

Second Pass →

1	3	6	2	4	7	0	5	8	9

As we can see bubble sort is very inefficient for longer arrays. It performs the same operation until no more swaps have to be made and that can take a lot of time especially if only 1-2 elements are out of place.

2(b) What is comb sort? Use comb sort on given array.

→ Comb sort is very similar algorithm to bubble sort. It also works by repeatedly comparing and swapping elements - the difference is that it has a gap that decreases with every iteration until it is 1 (when it does a final pass using bubble sort. This allows the sorting algorithm to have lesser passes.

Comb sort when applied →

Initial array →

6	3	1	9	8	2	4	7	0	5

First Pass →

6	0	1	9	8	2	4	7	3	5

with gap = 7

Second Pass →

2	0	1	3	5	6	4	7	9	8

with gap = 5

Third Pass →

2	0	1	3	5	6	4	7	8

gap = 3

Fourth pass →

1	0	2	3	4	6	5	7	9	8

gap = 2

Now a final pass with a gap of 1 is required

↳ This is also bubble sort.

After final pass with bubble sort →

0	1	2	3	4	5	6	7	8	9

gap = 1

2(c) Show first two passes of array from part (a) using quick sort.

→

Initial array →

6	3	1	9	8	2	4	7	0	5

First partioning with pivot as 5 →

3	1	2	4	0	[5]	6	9	8	7

Second partioning with both subarrays →

3	1	2	4	(0)

The pivot is 0

0	3	1	2	4

The pivot is 7 →

6	9	8	(7)

6	7	9	8

2(d) Simple example of quick sort at its worst. Whats the complexity?

→ There are many disadvantages of quick sort that don't allow the the algorithm to be as efficient. The worst case is when the pivot chosen is the smallest or largest element. This leads to unbalanced partitions which increase complexity of algorithm. The worst case time complexity of the algorithm is $O(n^2)$ - which is undesirable for large arrays.

For example, if we used quick sort on array A = [1, 7, 9, 2, 0] and used 0 as the pivot - it would lead to bad partitions to perform quick sort recursively on.

2(e) Complexity of heap sort? How it is arrived at? Compare with bubble and quick sort.

→ The complexity of heap sort is achieved by →

1. Creating a heap from an array } This can be called Heapify.
   $O(\log N)$ time complexity. }

2. Extract all elements and place at end of sorted array (initially empty) This has $O(N)$ time complexity for N elements

Complexity of heapsort - $O(N) \times O(\log N)$

$= O(N \log N)$

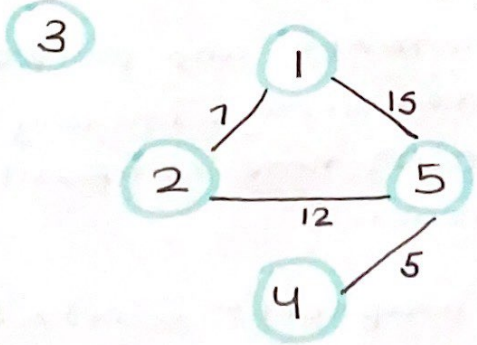
In comparison, bubble sort has a time complexity of $O(N^2)$ and is very inefficient for large arrays.

Quick sort has the time complexity of $O(N \log N)$ and is very efficient even for large arrays. But if the pivot results in unbalanced partitions, the time complexity can be upto $O(N^2)$ as well. Quick sort can also be unstable. Although heap sort is complex - it is very stable.

3(a). 2 data structures that can represent weighted graph, when it has to be stored in RAM and why one is better. Illustrate how graph would be stored both ways.

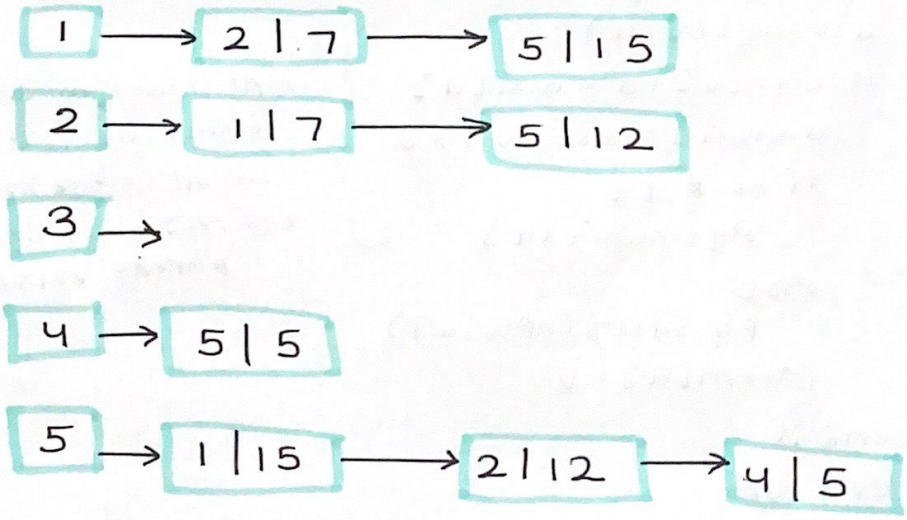
→ When weighted graph has to stored in RAM, it can be stored as an adjacency list or an adjacency matrix.

- Adjacency list is where each node in the graph stores a list of its neighbouring vertices - the edge weight is also recorded
  Efficient for sparse and dense graphs.
- Adjacency matrix is a square matrix of N x N is the number of nodes in the graph and is used to represent the connections between the edges of a graph. Adjacency matrix is inefficient for sparse graph.



Graph to be represented as adjacency list and adjacency matrix.

Adjacency List →



Adjacency Matrix →

	1	2	3	4	5
1	0	7	0	0	15
2	7	0	0	0	12
3	0	0	0	0	0
4	0	0	0	0	5
5	15	12	0	5	0

3(b) Pseudocode for Dijkstra's SPT. complexity? Add 1-2 lines to change it to Prim's MST.

→ Begin

```
// G = (V, E)
for each v ∈ V
    dist[v] = ∞
    parent[v] = null ... // have a special null vertex
    hPos[v] = 0
Pq = new Heap ... // priority queue initially empty
v = s
dist[s] = 0 // indicates that v ∈ heap
            // s will be root of SPT
```

```
while v ≠ null                                   //loop should repeat |V|-1 times
  for each u ∈ adj(v)                            //examine each neighbour u of v
      d = wgt(v,u)
      if dist[v] + d < dist[u]                   // After changing dist[u]
         dist[u] = dist[v] + d                   // either insert it or
         if u ∉ pq                               // siftUp in the heap
            pq.insert(u)                         // and record its new
         else                                    // parent vertex.
            pq.siftUp(hPos[u])
         parent[u] = v
      end if
      end for
   v = pq.remove()
 while end
 return parent
End
```

The time complexity of Dijkstra's algorithm is $O((V+E)\log_2 V)$, when the graph is sparse. This means that the graph has more vertices than edges. This means the algorithm has to update the distances less frequently which results in reduced complexity. The priority queue will also have lesser elements resulting in $O((V+E)\log_2 V)$ complexity.

The main difference between Dijkstra's algorithm and Prim's MST is that Prims only focuses on the edge weights – finding the smallest edges to connect a node in the MST to the nodes outside the tree. Both these algorithms use priority queues but the selection criteria is different. For Prim's, priority queue would select the vertex with the smallest edge weights (in that moment) instead of smallest distance.

The change would be as follows →

```
if weight(v,u) < dist[u]
    dist[u] = weight(v,u)
    parent[v] = v
```

The selection criteria changes.

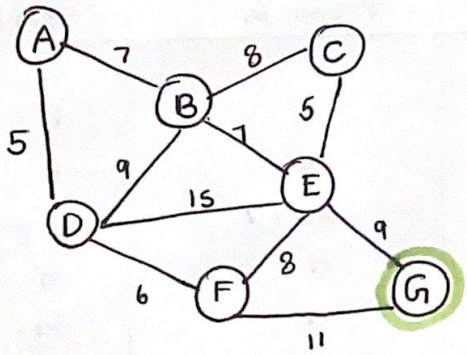
3(c). Illustrate in detail how Dijsktra's algorithm finds SPT for graph from vertex G_1. Show heap, distance[] and parent[] at each stage.

→ On next page

distance = Shortest distance from G_1

Previous vertex = Parent

Step 1 →



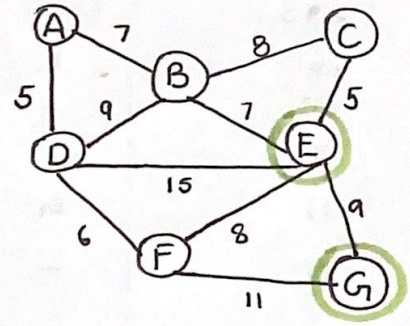
Vertex	Shortest Distance from G	Previous Vertex
A	∞	
B	∞	
C	∞	
D	∞	
E	∞	
F	∞	
G	0	

Visited = [ ]

Unvisited = [A, B, C, D, E, F, G] → current vertex

SPT = 0

Step 2 →



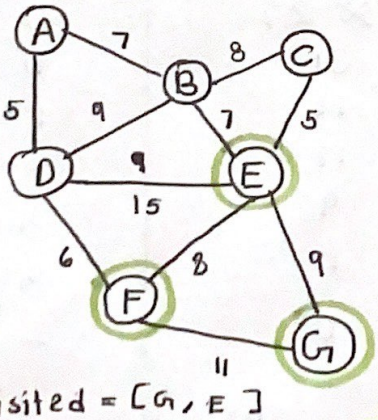
Vertex	Shortest Distance from G	Previous Vertex
A	∞	
B	∞	
C	∞	
D	∞	
E	9	G
F	11	G
G	0	

Visited = [G]

Unvisited = [A, B, C, D, E, F] → current vertex

SPT = 9

Step 3 →



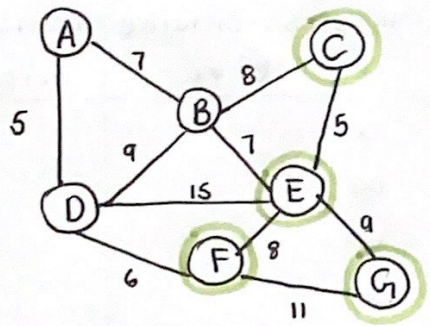
Vertex	Shortest Distance from G	Previous Vertex
A	∞	
B	~~∞~~ 16	E
C	~~∞~~ 14	E
D	~~∞~~ 24	E
E	9	G
F	11	G
G	0	

Visited = [G, E] current vertex

Unvisited = [A, B, C, D, F]

SPT = 11

Step 4 →



Visited = [G, E, F]

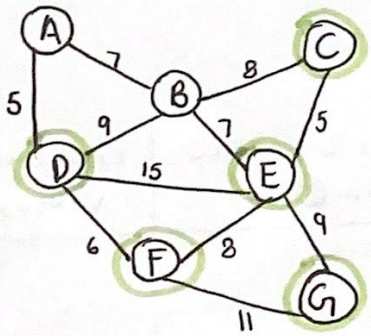
Unvisited = [A, B, C, D]

current vertex

Vertex	Shortest Distance From G	Previous Vertex
A	∞	
B	18	E
C	16	E
D	~~24~~ 17	F
E	9	G
F	11	G
G	0	–

SPT = 16

Step 5 →



Visited = [G, E, F, C]

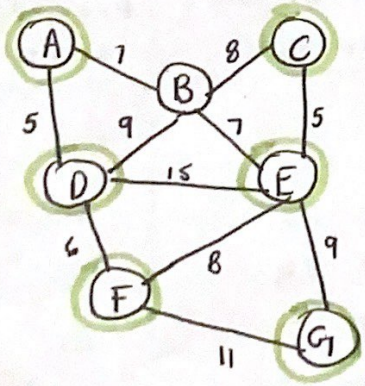
Unvisited = [A, B, D]

current vertex

Vertex	Shortest Distance From G	Previous Vertex
A	∞	
B	23	E
C	16	E
D	17	F
E	9	G
F	11	G
G	0	–

SPT = 22

Step 6 →



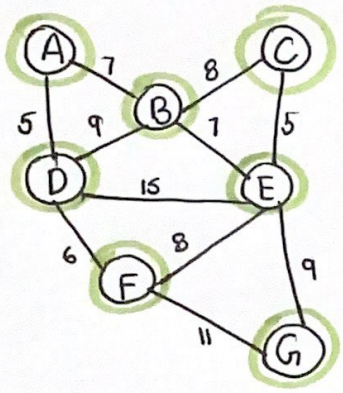
Visited = [G, E, F, C, D]

Unvisited = [A, B]

Vertex	Shortest Distance From G	Previous Vertex
A	~~∞~~ 27	D
B	31	D
C	16	E
D	17	F
E	9	G
F	11	G
G	0	–

SPT = 27

Step 7 →



Vertex	Shortest Distance From G1	Previous Vertex
A	27	D
B	34	A
C	16	E
D	17	F
E	9	G1
F	11	G1
G1	0	—

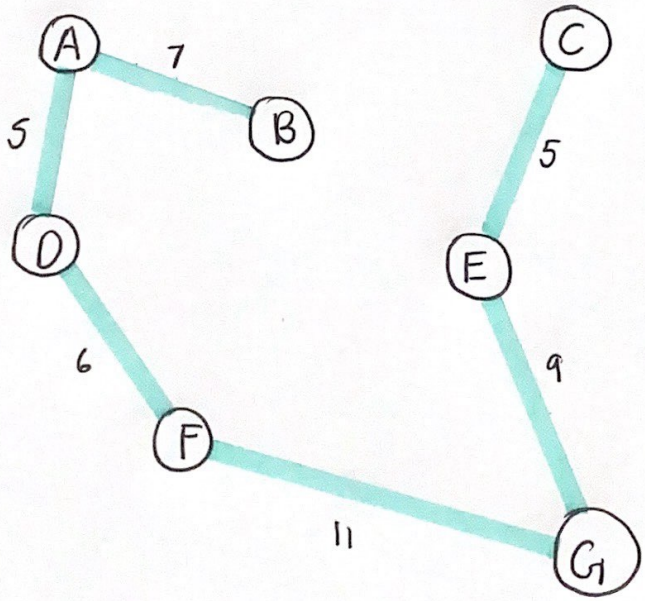
Visited = [G1, E, F, C, D, A]

Unvisited = [B]

final vertex

SPT = 34

Step 8 →



Final

SPT = 43 ?