Algorithms

Paper - 2019

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(n/2) + n & n > 1 \end{cases}$$

Q1 (a) input of n equal numbers

(i) Merge sort : $O(n \log n)$ →

(ii) Heap sort : $O(n)$, Heapify will take $O(1)$.

(b) Give Recurrence

(i) Max heapify

$$T(n) <= T(2n/3) + \Theta(1)$$

(ii) Quick sort

$T(n) = 2T(n/2) + \Theta(n)$ Best case

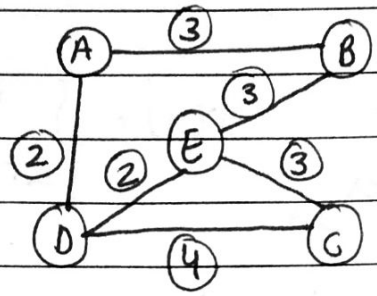
$T(n) = T(n-1) + \Theta(n)$ Worst case

(c) Radix sort can use counting sort, insertion sort, bubble sort or bucket sort. Insertion sort is used because it is a stable sorting algorithm.

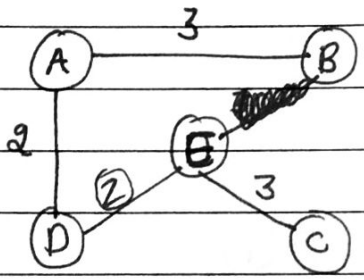
(d) Compare worst case running times

	BST (Best)	BST (worst)	Red-Black (Avg, worst)
Search	$O(\log n)$	$O(n)$	$O(\log n)$
Insert	$O(\log n)$	$O(n)$	$O(\log n)$
Delete	$O(\log n)$	$O(n)$	$O(\log n)$
	$O(\log n)$	$O(n)$	$O(\log n)$

(g)

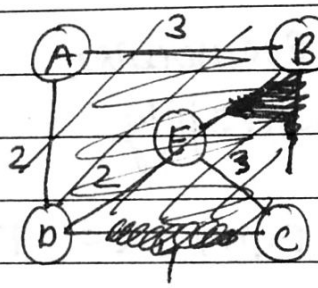


Prims from D.

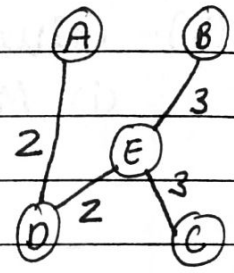


Cost = 10

Krusgal's



~~Cost = 11~~



Cost = 10

(h) Adjacency list would be more suitable.

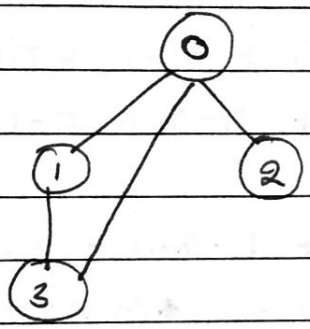
1) Adjacency matrix requires $O(V^2)$ space where as Adjacency list requires $O(V+E)$ space, so in a sparse graph most of the entries for a would be zero so it is better to go with adjacency list.

2) Traversing may take order of (V^2) time in adj. mat. whereas $O(V+2E)$ or $O(E)$ time is taken in adj. list.

3) Quering can be done in $O(1)$ time in adj mat. but it may take $O(V)$ time in worst case in adj list. in a case when all vertices are connected to that vertex, we have to check all of them so time would be $O(V)$.

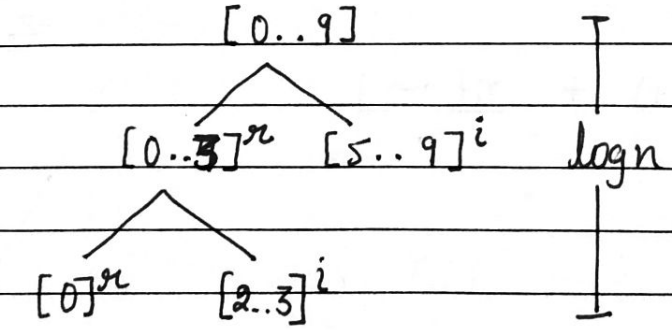
(i) NO, DFS cannot be used to determine shortest path

DFS starts exploring a vertex as soon as it visits it so, the first choice may not be the best choice.

Consider 

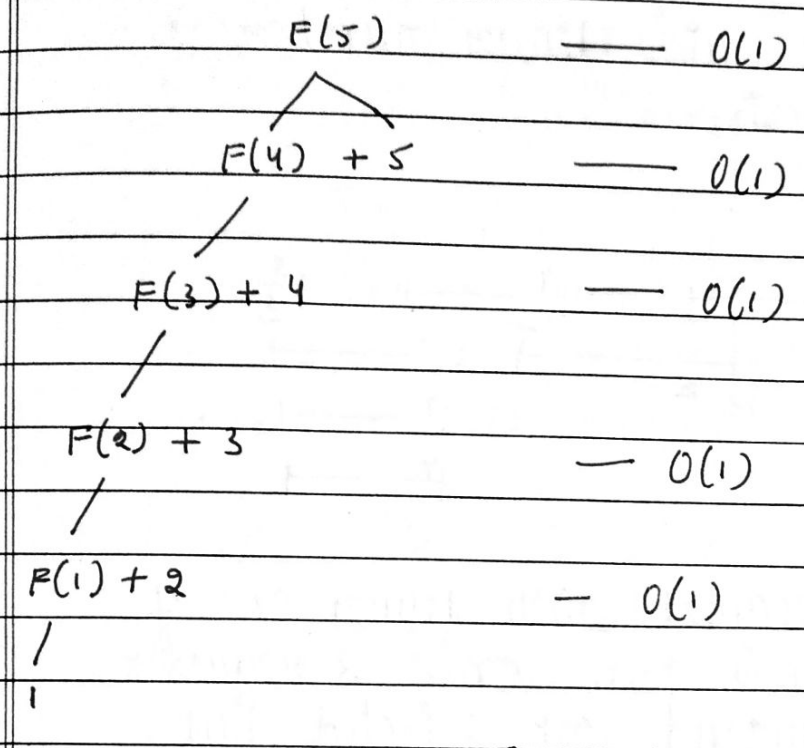
the DFS would give 0-1-3 as a path from 0 to 3 where as shortest path is available i.e 0-3.

(ii) Assuming that the array is always partitioned in middle, in that case one part will be solved by recursion and larger subarray will be solved by iterative module. Hence depth of recursion would still be O(logn) because half of the elements still required to be solved recursively.

eg 

In case of worst case partitioning, larger subarray (which has (n-1) elements, would be solved by iterative module and smaller subarray has 0 elements so recursion will simply terminate.

(g)



Time = no. of levels × time spent at each level

$T(n) = n \times O(1)$

$T(n) = O(n)$

Alternatively, $T(n) = T(n-1) + O(1)$ — ①

$T(n-1) = T(n-2) + O(1)$ — ②

$T(n-2) = T(n-3) + O(1)$ — ③

⋮

$T(n) = T(n-1) + O(1)$

$= T(n-2) + O(1) + O(1)$

$= T(n-3) + O(1) + O(1) + O(1)$

⋮

$T(n) = T(n-K) + K.O(1)$

We know, $T(1) = 1$ so $T(n-K) = T(1)$, $n-K = 1$

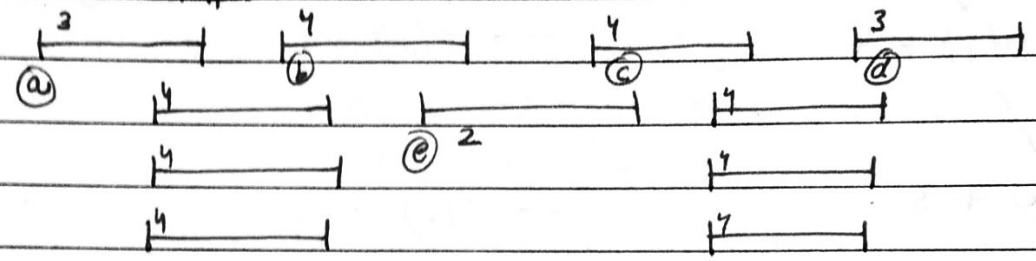
or $K = n-1$

$T(n) = T(1) + (n-1).O(1)$ or $T(n) = 1 + O(n-1)$

$\boxed{T(n) = O(n)}$, Memoization will not improve the running time because there are no repeating subproblems.

(j) Select the request with fewer number of incompatible requests.

Counter example:



If we select the request with fewer no. of incompatible requests then only 3 requests i.e a, e, and d would get selected but this is not the optimal solution because if we select a, b, c and d then 4 request are selected which are also compatible.

Q2.(c) Subset sum problem: Given a set of non-negative integers, and a value sum, determine if there is a subset of the given set with sum equal to given sum.

Refer to Q1(c) of 2017

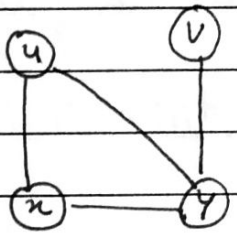
```
bool isSubsetSum( int arr[] , int n, int sum) {
    if ( sum == 0 ) {
        return true;
    }
    if ( n == 0) {          ( n==0 && sum != 0)
        return False;
    }
    if( arr[n-1] > sum) {
        return isSubsetSum( arr, n-1, sum);
    }
```

return (isSubsetSum(set, n-1, sum) ||
isSubsetSum (set, n-1, sum - arr[n-1]);

Recurrence Relation:

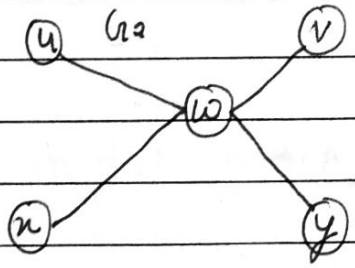
isSubsetSum (set, n, sum) =
(isSubsetSum(set, n-1, sum) ||
isSubsetSum (set, n-1, sum - arr[n-1]);

Q3. (a) G_1



G_1 is not bipartite because it contains an odd length cycle.

G_2



G_2 is bipartite
$V_1 = \{u, v, x, y\}$, $V_2 = \{w\}$.

(b) a) There is an optimal schedule with no idle time.

Proof: A schedule having idle time is not optimal because by definition a job can be started as soon as the previous job finishes i.e machine can only be idle once all the jobs are finished.

b) All schedules with no inversions and no idle time have the same maximum lateness.

Proof: If two different schedules have neither inversions nor idle time, then order of jobs may differ, but they can only differ in the order in which jobs with identical deadlines are scheduled.

In both schedules all the jobs with deadline d are scheduled consecutively (after all jobs with earlier deadlines and before all jobs with later deadlines). Among the jobs with deadline d, the last one has greater lateness but this lateness does not depend on the order of jobs.

c) There is an optimal schedule that has no inversions and no idle time.

Proof. By (a), there is an optimal schedule O with no idle time

(i) If O has an inversion, then there is a pair of jobs i and j such that i is scheduled before j and has $d_j < d_i$

(ii) After swapping i and j we get a schedule with one less inversion.

(iii) The new swapped schedule has a maximum lateness no larger than that of O.

Proof. Assuming that each request r is scheduled for the time interval $[s(r), f(r)]$ and has lateness l_r. Let $L' = \max_r l'_r$ denote max. lateness of this schedule. Let $\bar{O}$ denote the swapped schedule and $\bar{s}(r), \bar{f}(r), \bar{l}_r$ denote corresponding quantities in swapped schedule.

Job j will get finished earlier in the new schedule, and hence the swap does not increase the lateness of job j. After swap, job i finishes at time $f(j)$, lateness of $i = \bar{l}_i = \bar{f}(i) - d_i = f(j) - d_i$ and we know $d_i > d_j$
Hence $\bar{l}_i = f(j) - d_i < f(j) - d_j = l'_j$. Hence $l' \geq l'_j > \bar{l}_i$
Hence swapping does not increase max lateness of schedule.

Q4 (a)
```
MaxHeapify(A, largest)
for(i = ~~length~~(A.heapsize/2); i>=1; i--){
    MaxHeapify(A, i); (OR) → for i = (A.length/2) down to 1
}
for i = A.length down to 2
    exchange A[1] with A[i]
A.heapsize = A.heapsize - 1
Max-Heapify(A, 1);
```

b) NO, it is not correct.

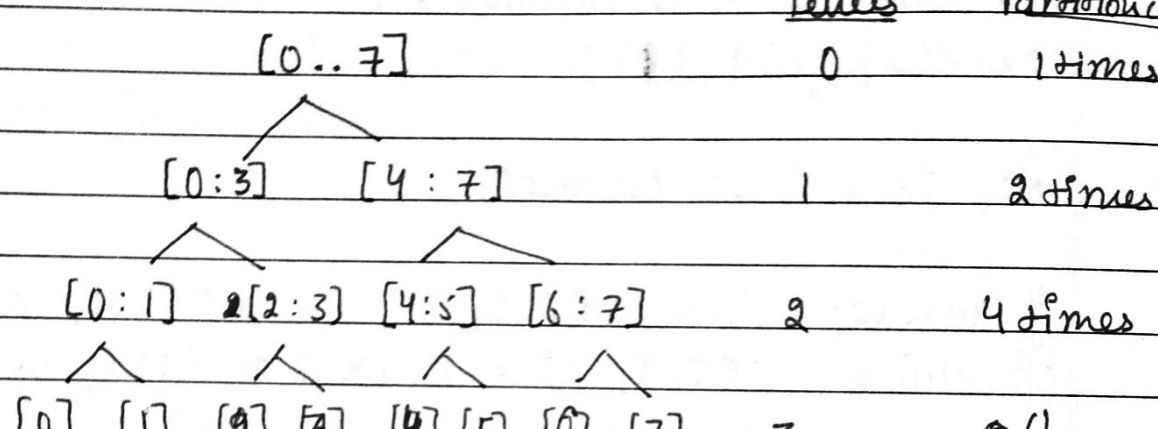
If $w < w_i$ then $OPT(i, w) = OPT(i-1, w)$
Otherwise, $OPT(i, w) = max(OPT(i-1, w), v_i + OPT(i-1, w - w_i))$

In a case when $w_i > w$ then object cannot be included but otherwise there are two possibilities, including the object or not including it. In a case when it is included, v_i is added to the optimal solution that must be computed from $(i-1)$ elements with maximum allowable weight of $(w - w_i)$.

(c)
```
BFS(s):
    Set visited[s] = true and visited[v] = false for all other v
    Push s in queue
    while(!q.empty()):
        Pop u from queue
        Consider each edge (u,v) incident to u.
            If visited[v] == false
                Set visited[v] = true
                Push v to queue
            EndIf
        EndFor
    EndWhile
```

Q5(a) No it won't be O(1) because in a case when n Multi-push() operations are performed, and each Multi-push() operation ~~has~~ takes O(n) time in worst-case. Hence overall time complexity becomes $n \times O(n) = O(n^2)$, which gives amortized cost as $O(n^2)/O(n) = O(n)$.

(b)



Levels	Partition called
0	1 times
1	2 times
2	4 times
3	0 (base condition)

The no. of times partition function is called is equal to the no. of non leaf nodes of the binary tree. i.e equal to no. of internal vertices.

We know that no. of leaves (L) is equal to = no. of internal vertices (I) + 1

i.e $L = I + 1$, So, Total nodes = $N = L + I$

$N = I + 1 + I$

$N = 2I + 1$ OR $\boxed{I = \frac{N-1}{2}}$

So, no. of times partition func called = $\left\lfloor \frac{N-1}{2} \right\rfloor$ or $O(n)$

Verification: For $n = 8$, $i = \frac{8-1}{2} = \left\lfloor \frac{7}{2} \right\rfloor =$

no. of internal vertices = sum of all nodes till second last level

we know that last level (l) = $\log_2(n)$
second last level = $l-1$ = $\log_2(n)-1$

no. of internal vertices = $2^0 + 2^1 + 2^2 + \cdots + 2^{\log_2(n)-1}$

this is a Geometric progression, so sum of GP is given by $\boxed{GP = \frac{a(r^k-1)}{r-1}}$ where k is the no. of terms in G.P.

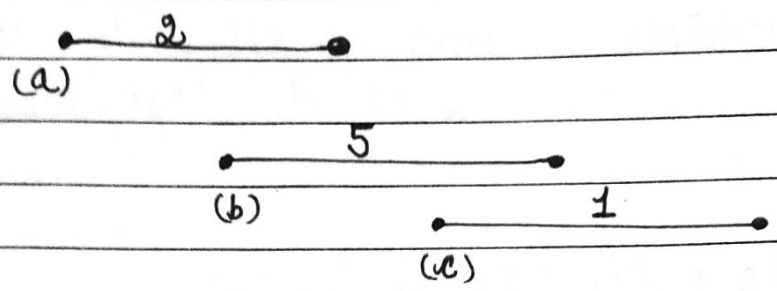
So, sum = $\frac{1 \cdot (2^{(\log_2(n)-1)+1} - 1)}{(2-1)} = 2^{\log_2(n)} - 1$

no. of internal vertices(i) = $2^{\log_2 n} - 1$
no. of times partition function called = $i = 2^{\log_2 n} - 1$

$i = 2^{\log_2 n} - 1$
$\Rightarrow i + 1 = 2^{\log_2 n}$
Taking log on both sides
$\log(i+1) = \log_2(2^{\log_2 n})$
$\log_2(i+1) = \log_2 n \times \log_2 2$
$\log_2(i+1) = \log_2 n \times 1$
So, $i + 1 = n$
i.e $\boxed{i = n-1}$ or $i = O(n)$

Verify: For $n=8$, $i = 8-1 = 7$ i.e the no. of internal vertices (see the figure).

Q5(c)



No, this greedy approach will not work. Consider a scenario where three intervals are there with weights mentioned on them.

If we go by greedy strategy then c interval would be selected and we cannot take b because it is overlapping so we took a.

Total weight = $w_a + w_c = 2 + 1 = 3$
But clearly this is not the optimal solution because if we would have selected interval b then weight would be 5.

Q6 (a) If we do not use randomized-partition then the probability of worst case partitioning increases.

(b) To prove: If G has a topological ordering, then G is a DAG.
Suppose by way of contradiction, that G has a topological ordering $v_1, v_2 \dots v_n$ (i.e for every edge (v_i, v_j), we have $i < j$) and also has a cycle C. Let v_i be the lowest-indexed node on C and let v_j be the node of C just before v_i — thus (v_j, v_i) is an edge. But by our choice of i, we have $i < j$, which contradicts our assumption that $v_1, v_2 \dots v_n$ was a topological ordering.

Q7 (a) $(3K-1)$, $3K$, $3K+1$

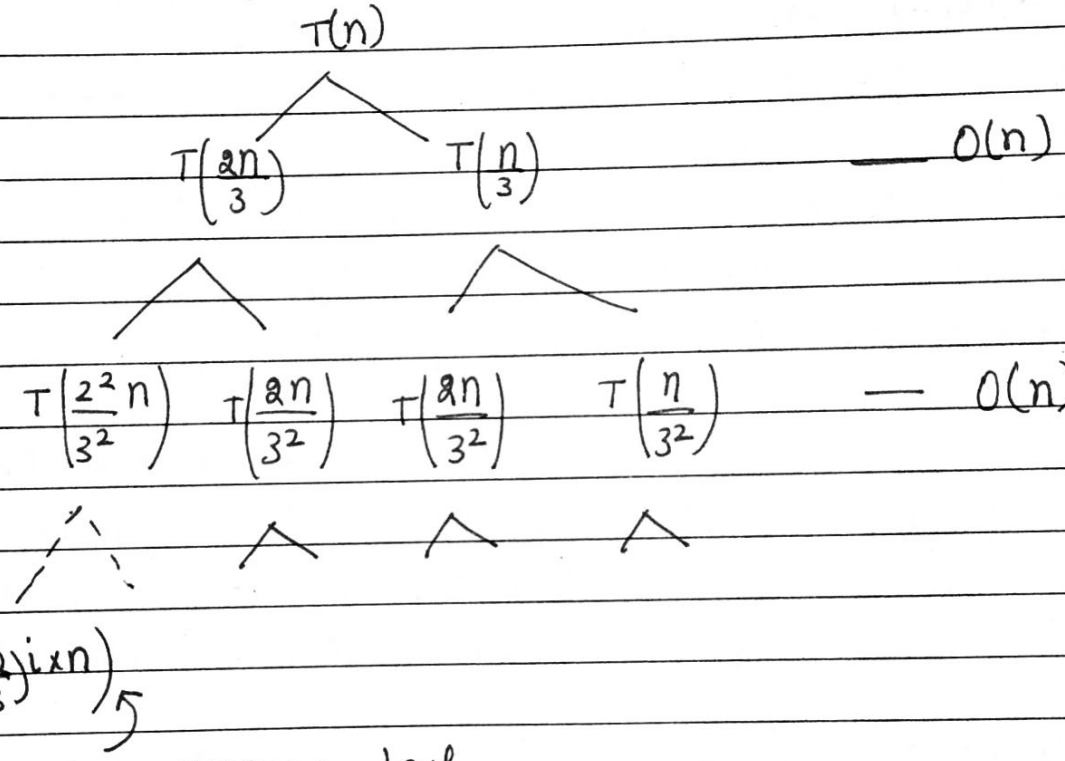
(b) Sort sacks in decreasing order of capacity and ~~first take~~ take sacks one by one in decreasing order of capacity.

(c) $O(n^2)$

PAPER - 2018

Q1. (a) $n^2 \log(n) < n^{\log(n)} < 2^n < 2^{2^n}$

(b) $T(n) = T\left(\frac{2n}{3}\right) + T\left(\frac{n}{3}\right) + \Theta(n)$



Following greatest path

$n \rightarrow \frac{2n}{3} \rightarrow \frac{2^2 n}{3^2} \rightarrow \frac{2^3 n}{3^3} \dashrightarrow \left(\frac{2}{3}\right)^i n$

Let $\left(\frac{2}{3}\right)^i n = 1$

$n = \left(\frac{3}{2}\right)^i$

Taking $\log_{3/2}$ both sides.

$\log_{3/2}(n) = \log_{3/2}(3/2)^i = i \log_{3/2}(3/2) = i$

So, $i = \log_{3/2}(n)$

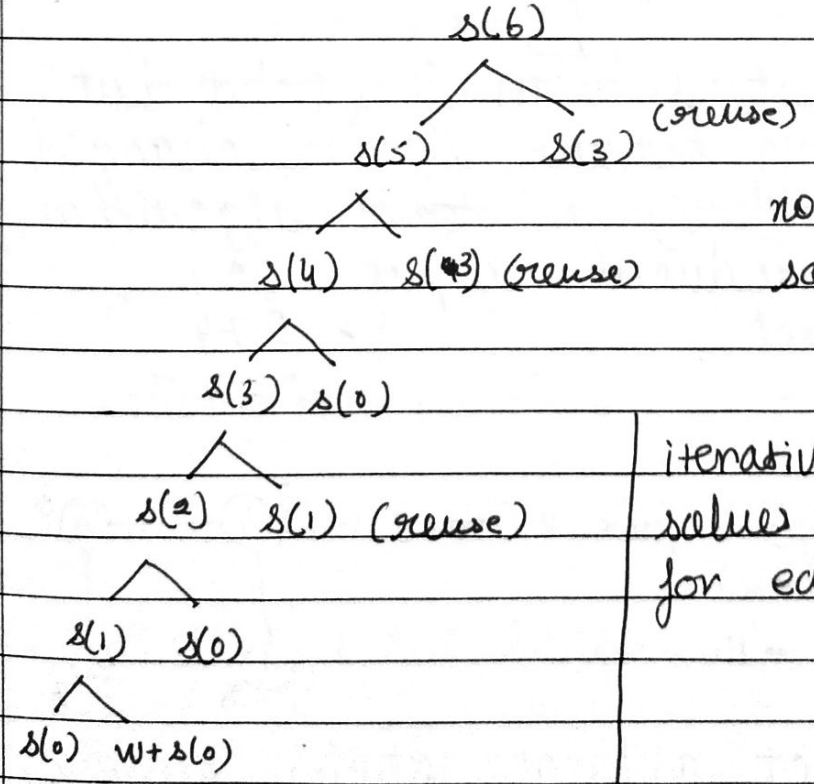
Sum of work done $= n + n + n \cdots \log(n)$ terms

$= O(n \log(n))$

(c) sort sacks in decreasing order of capacities

```
int count = 1;
for(int i = 0; i < n ; i++) {
    if( W >= cap[i] ) {
        W -= cap[i];
        count++;
    } else {
        count++;
        break;
    }
}
return count; (no. of sacks used).
```

(d) Solved by memoized recursive algorithm



no. of subproblems solved = 5

iterative algorithm also solves 5 subproblems, one for each interval.

(f) Algorithm should be stable for it to be usable as an intermediate sort in radix sort.

Consider a scenario where there are three numbers and we call ~~radix sc~~ intermediate sorting algorithm on hundred'th place (leftmost).

0	24
2	34
0	74

Now if intermediate sorting algorithm is not stable then it might change the relative ordering of 0's. i.e

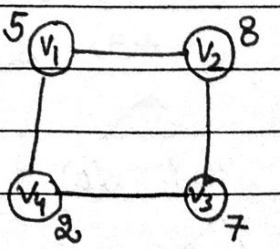
0	74
0	24
2	34

Although leftmost column is sorted but output is wrong because it has changed the relative ordering. A stable algorithm would have produced output 024, which is correct.

↳ 074

234

(g) Acc. to algorithm, first pick V_2 (wt = 8)
then pick V_4 (wt = 2)
Total wt = 8 + 2 = 10



Clearly this is not optimal solution because if we would have picked V_1 and V_3 then total weight would be = 5 + 7 = 12

(h) Naive string matching algorithm
Best Case: $O(n)$
The first character of the pattern is not present in text at all.

Worst case: $O(m * (n-m+1))$
All characters of the text and pattern are same. or only the last character is different.

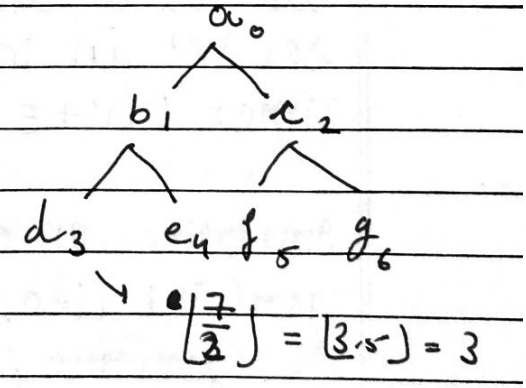
(i) BFS can be used, Time: $O(|V| + |E|)$

(j) We only need to search for leaf nodes.

```
int findMax(int heap[], int n) {
    int s = ⌊n/2⌋;
    int max = heap[s];
    for(int i = s+1; i < n; i++) {
        if(heap[i] > max) {
            max = heap[i];
        }
    }
    return max;
}
```

clear writing

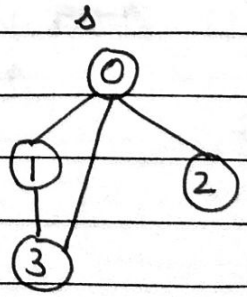
```
{ int s = ⌊n/2⌋;
  int max = heap[s]; }
```



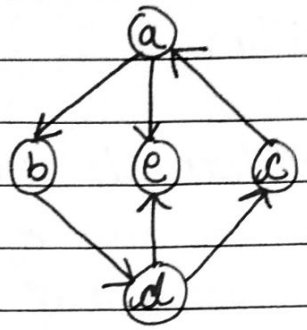
Exact running time: $O(n/2)$

(K) No DFS can not be used.

DFS may give path from 0 to 3 as $0 \to 1 \to 3$ but a shorter path exists i.e $0 \to 3$.



Q2 (a)



a — b, e
b — d
c — a
d — c, e
e — x

In degree of a vertex 'v' = no. of times it appears on right side of adjacency list.

If we search all the list for each vertex, time to compute the in-degree of every vertex is ~~of~~ O(V.E)

But we can create an array of size V and initialize it's entries to zero - O(V)
Now, we need to traverse through the list and incrementing arr[v] when we see 'v' in the lists.
Time: O(V+E) with O(V) additional storage.

Initialize an array ~~with~~ of size v with entries = 0.

```
for( ini i =0; i <n; i++) {
    for( int j = 0; j< Adj[i].size(); j++) {
        ~~arr~~ v = Adj[i][j];
            arr[v]++;
    }
}
```

arr[] now contains the indegree of every vertex.

(b)

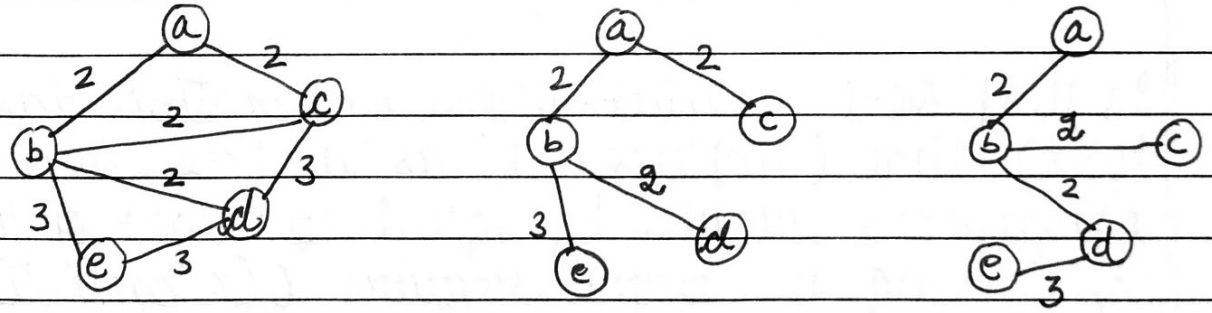
Q3 (a) Worst case running time of quick sort : $O(n^2)$

In worst case one call to binary search takes $O(\log_2 n)$ time.

for loop call binary search for n time.
So total time = $O(n \log n)$

Total time for FindPair = $O(n^2) + O(n \log n)$
$= O(n^2)$

(b) Yes, in a Graph G which has edges of same weight, have more than one minimum spanning tree.



G(V, E) MST-1 MST-2

(c) (i) In place : No, it uses extra arrays to merge

(ii) Stable : Yes, if "L <= R" condition must be used. It does not change the relative ordering of the same elements if '<=' is used rather than just '<'.

```
if (arr[l] <= arr[r]){
    arr[k++] = arr[l++];
}
else{
}
```

Red Wings
Page No.__ Date __/__/20__

Q4. (a) In a sequence of n operations although a single multipop operation can be expensive but any sequence of n Push, pop and multipop operations on initially empty stack can cost at most O(n) because ~~we~~ can no. of times pop operation can be called upon stack, including calls within multipop, is at most the no. of push operations, which is at most n. So, total time = O(n).
The average cost of an operation is $\frac{O(n)}{n} = O(1)$.

Q5(b) Selection sort: Here time required to find the maximum element and placing it in ~~curre~~ correct position is O(n). {Traversing the array}. Sorting n elements require $n \times O(n)$ i.e $O(n^2)$ time.

In Heap sort extract-max() operation takes log(n) time (implemented as a tree so maximum comparisons would be equal to height of the tree) So, Sorting n elements require O(nlogn) time.

Q4 (b)

(i) Merge sort: 1 2 3 4 5 (already sorted)

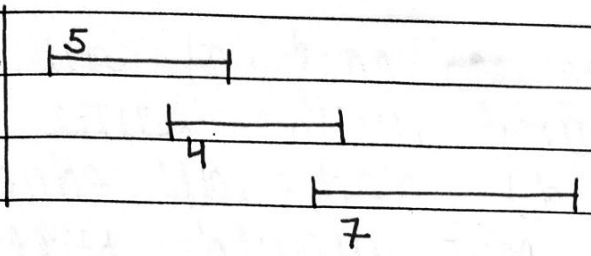
(ii) Quick sort: 1 2 3 4 5 (already sorted)

or

5 4 3 2 1

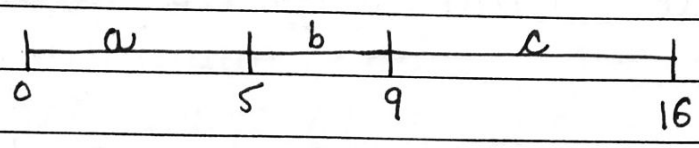
Red Winge
Page No.__ Date _/_/_20_

Q4 (c)



	length	deadline
a	5	6
b	4	6
c	7	10

Case I:

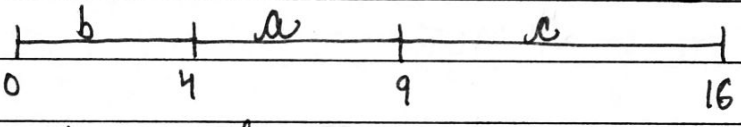


$l_a = 0$ $l_b = 9-6$ $l_c = 16-10$

$l_b = 3$ $= 6$

Total lateness $= 3+6 = 9$

Case II:

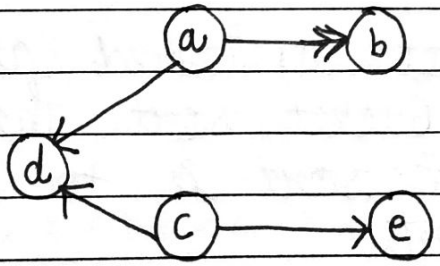


$l_b = 0$ $l_a = 9-6$ $l_c = 16-10$

$l_a = 3$ $= 6$

Total lateness $= 3+6 = 9$

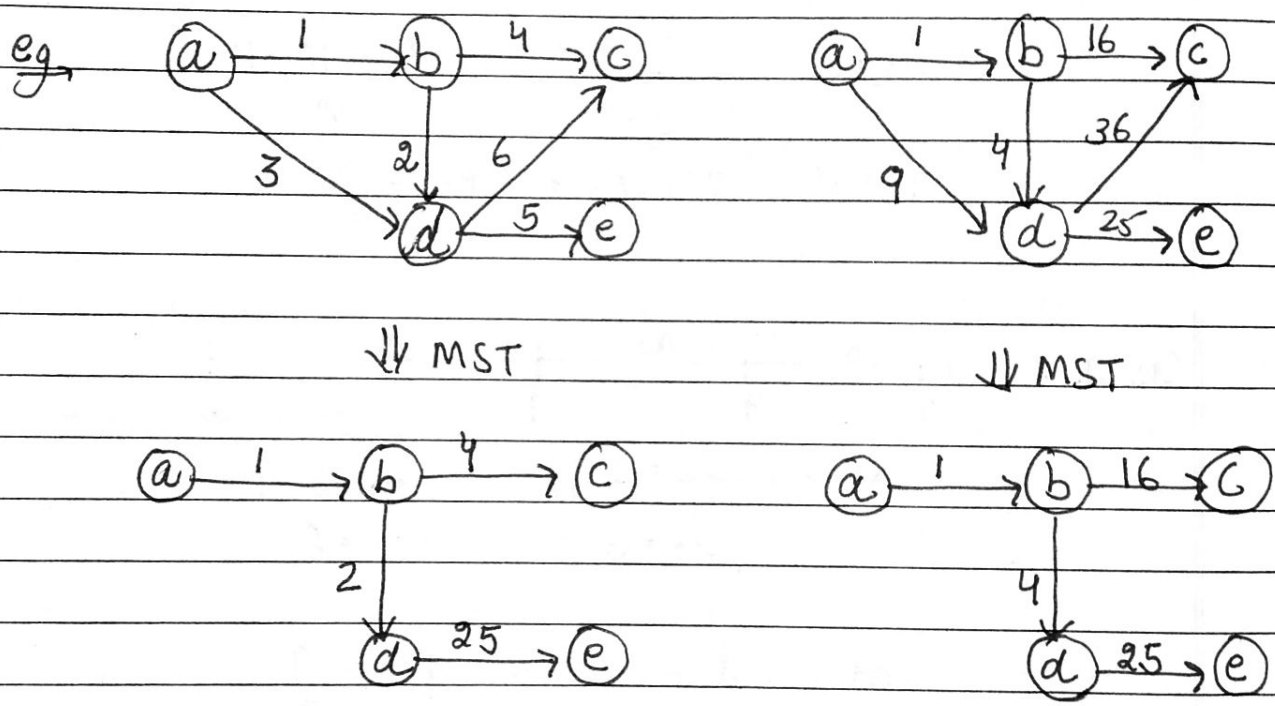
Q5(a)



(i) a, b, c, d, e

(ii) c, e, a, b, d

Q6 (a) Yes, ~~it~~ because as cost of all the edges are increased with same proportion (squared) and all edges are distinct, the MST would still be same but with different cost of edges.



(b) Insertion-sort is not considered as an integral part of Bucket sort. The main function of bucket sort is to distribute elements into buckets.

(c) (i) Best case: ascending order input
$O(n)$ → (one comparison for each element)

(ii) Worst case: descending order input
$O(n^2)$ → [For each element (i) all the elements on left side of i have to be shifted ~~forward~~ rightward]

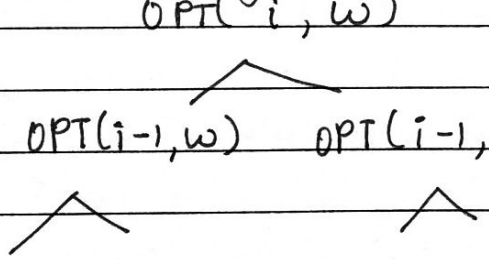
Placing 1 element : $O(n)$
Placing n elements : $O(n^2)$

Red Wings
Page No.__ Date __/__/__20__

07 (a) We can find both minimum and maximum using at most $3\lfloor n/2 \rfloor$ comparisons. Here we process elements in pairs. We compare the elements from the input first with each other, and then we compare smaller with the current minimum and larger with the current maximum, at a cost of 3 comparisons for every two elements.

If n is odd, we set both the minimum and maximum to the value of first element, If n is even then we perform 1 comparison on first two elements to determine the initial values of minimum and maximum.

(b) (i) For each object we have two possibilities of either including it or not including it.



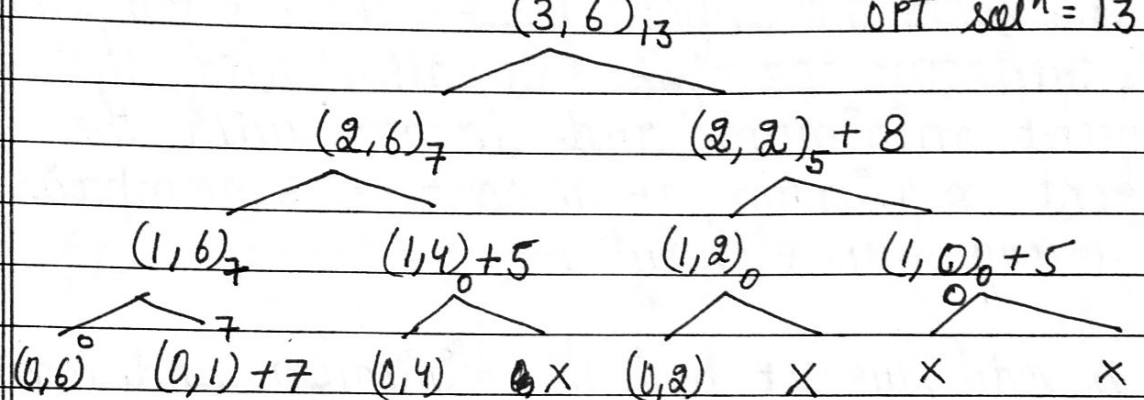
Hence for n objects
Time = 2^n (exponential).

(ii) We can use memoization technique, where we store the value of subproblems and when a subproblem appears again we use the value stored rather than re-computing. This decreases the running time from exponential to polynomial.

(iii) Yes, because at each stage we explore two possibilities (inclusion & exclusion) and take optimal (or max) one.
(Give example).

	1	2	3
w	5	2	4
p	7	5	8

Capacity = 6



Hence we can see that optimal solution of (3,6) contains within it the optimal solution of (2,6) and (2,2)

Paper - 2017

Q1(a) $n^2 \log(n) < 2^{n^2} < 2^{2^n} < n^{2^n}$

(b) $T(n) = T(n/3) + O(1)$

By Back Substitution:

$T(n) = T(n/3) + \cancel{O(1)}\, c$ — ①

$T(n/3) = T(n/9) + \cancel{O(1)}\, c$ — ②

$T(n/9) = T(n/27) + \cancel{O(1)}\, c$ — ③

So, $T(n) = T(n/3) + \cancel{O(1)}\, c$

$T(n) = T(n/9) + \underset{c}{\cancel{O(1)}} + \underset{c}{\cancel{O(1)}}$ — from ②

$= T(n/9) + \cancel{O(1)}\, 2c$

$T(n) = T\left(n/27\right) + \underset{3c}{\cancel{O(1)}}$ — from ③

$\vdots$

$T(n) = T(n/3^i) + \cancel{O(1)}\, ic$ — ④ → at i^{th} level

Put $\frac{n}{3^i} = 1 \Rightarrow n = 3^i \Rightarrow \log_3(n) = \log_3(3^i)$

$\Rightarrow \log_3(n) = i \log_3(3) \Rightarrow \log_3(n) = i$

Put $i = \log_3(n)$ in eqn ④

$T(n) = T(1) + \log_3(n) \cdot c$

$T(n) = 1 + c \cdot \log_3(n)$

$T(n) = O(\log_3(n))$

By Masters Theorem

$T(n) = T(n/3) + O(1)$

Compare with

$T(n) = aT(n/b) + f(n)$

$a = 1$, $b = 3$, $f(n) = O(1)$

$\log_b a = \log_3 1 = 0 \quad \because [3^0 = 1]$

$n^{\log_b a} = n^0 = 1$

So Here $f(n) = \theta(n^{\log_b a}) \Rightarrow O(1) = \theta(1)$

So, case II is applied

~~$f(n) = \theta$~~

(case 2: If $f(n) = \theta(n^{\log_b a})$

then $T(n) = \theta(n^{\log_b a} \lg n)$

$T(n) = \theta(1 \cdot \lg n)$

$T(n) = \theta(\lg n)$

Q1(c) Subset-Sum problem: Here we are given n items and each has a non-negative weight w_i. The goal is to find a subset S of items so that sum of their weights should ~~me~~ be maximum but less than W (Total capacity).

$\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} w_i$ is as large as possible.

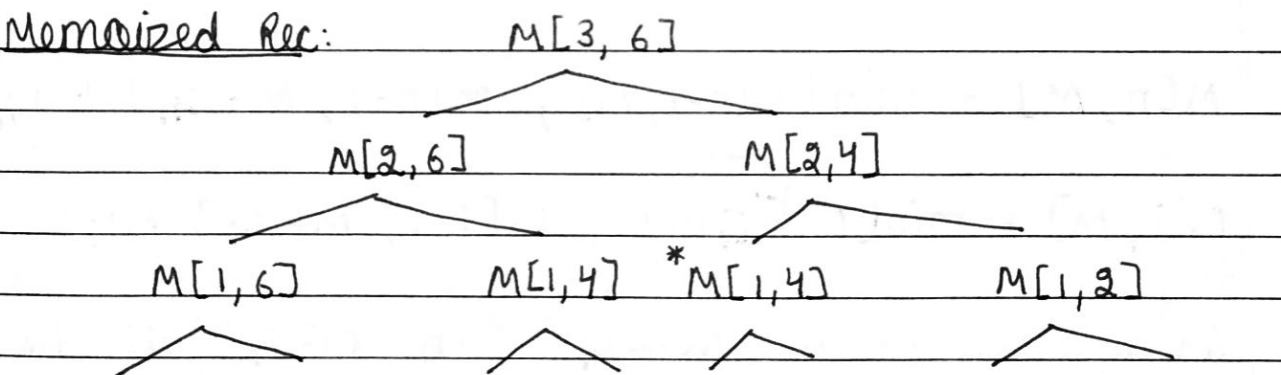
Recurrence:

If $w_i > w$

then $OPT(i, w) = OPT(i-1, w)$, otherwise

$OPT(i, w) = \max(OPT(i-1, w), w_i + OPT(i-1, w-w_i))$

Memoized Rec:



Memoized rec. solves: M[1,6], M[1,4], M[1,2], M[2,6], M[2,4], M[3,6]

DP solves (n x w) problems i.e ~~O(3x6)~~ 3x6

M[1,0], M[1,1], M[1,2] ---------- M[1,6]

M[2,0], M[2,1], M[2,2] ---------- M[2,6]

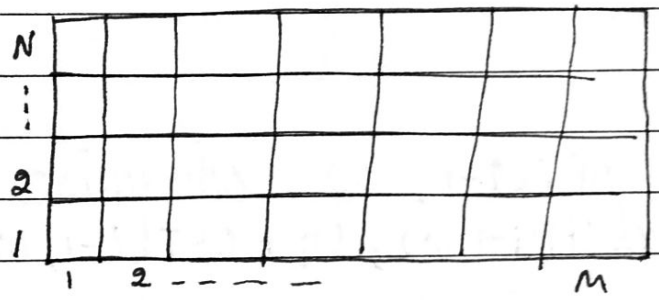
M[3,0], M[3,1], M[3,2] ---------- M[3,6]

Clearly memoized recursive solves lesser no. of subproblems.

Red Wings
Page No.__ Date __/__/__20__

Q1 (e) (i) Here greedy approach can be used, first arrange the bottles in decreasing order of capacity say $b_1, b_2 \dots b_n$ and fill the bottles in above order.

(ii) Here Dynamic Programming can be used. For each bottle there are two possibilities, i.e it can either be included or not included.



M : Total marbles
n : Total bottles.

$$M[n, M] = \min(M[n-1, M],\ M[n-1, M-c_n] + p_n)$$

OR

$$M[i, M] = \min(M[i-1, M],\ M[i-1, M-c_i] + p_i)$$

In a case when $M < c_i$ then $M - c_i$ will be negative which can fail the algorithm. So Assuming that bottle can either be fully filled or not included. ie no partial filling.

```
if (M < ci)
    M[n, M] = M[n-1, M]    // It cannot be filled fully
else
    M[n, M] = min(M[i-1, M], M[i-1, M-ci] + pi)
```

(f) (i) Yes, the output will be correct.
(ii) The running time may worsen. In the worst case all the elements may fall into the same bucket and time complexity may become equal to that of intermediate sort used.

(g) Best Case: The first character of the pattern is not present in the text at all.

Time Complexity: $O(n)$ $\quad |Text| = n$, $|pattern| = m$

Worst Case: All the characters of the text and pattern are same. or only the last character of window (pattern) is different.

Time Complexity $O((n-m+1)*m)$ or $O(nm)$.

Algo: For every position in text, check if the string begins at it. This check may take m comparisons each.
Hence $O(mn)$

(h) edges = m, vertices = n
Adjacency matrix: $O(n^2)$ or $O(|V|^2)$ for both directed and undirected, independent of edges

Adjacency list: Directed: $O(|V|+|E|)$, Undirected: $O(|V|+2|E|)$ } $\Theta(|V|+|E|)$ or $O(m+n)$

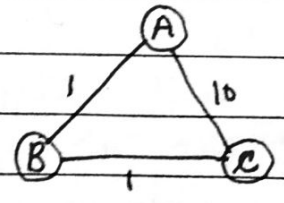
(i) The min element of max heap must be at leaf. leaf elements begin at $= \lfloor n/2 \rfloor$

```
int findMin (int arr[], int n) {
    int loc = 0;
    int s = ⌊n/2⌋;
    int min = arr[s];
    for i = s to n-1
    {
        if ( arr[i] < min) {
            min = arr[i];
        } loc = i
        }
    }
}
```

Exact Runtime : $\lceil n/2 \rceil$ since there $\lceil n/2 \rceil$ leaves.

(j) No, it is not advisible to use BFS because the graph is weighted.

eg,



Acc. to BFS, dist(A, C) = 10 where as the optimal dist(A, C) solution would be 2, by going through B.

Q2(a) On running BFS / DFS on a graph if an already discovered node is visited again, then a cycle exists.

Make an visited array and mark all vertices as not visited
queue.push(s);
visited[s] = True
while (!queue.empty()) {
    u = queue.~~pop()~~ front();
    queue.popfront();
    for all vertices v that are adjecent to u {
        if (visited[v] == True) {
            return True; // Graph contains a cycle
        } else {
            visited[v] = ~~False~~ True; // Mark v as visited
            queue.push(v);
        }
}
}
return ~~return~~ False; // Cycle not present
}

Q3(a) (i) Finding a maximum element
Heap : O(1)] → First element is largest element
Array : O(1)]

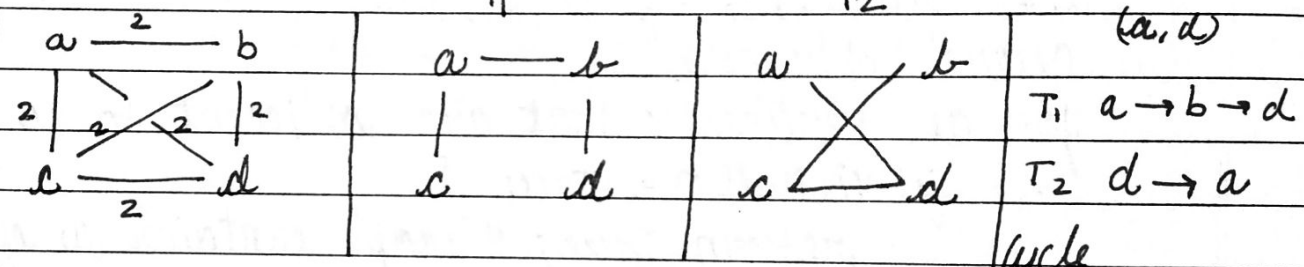
(ii) Deleting the maximum element
Heap O(log n) → Extract-max(), the element from last leaf will move to top and heapify should be called.
Array: O(n) → Shift all the n-1 elements to left.

(iii) Increase the priority of a certain element.

Heap: $O(\log n)$: Increase key operation

Array: $O(n)$: In worst case lowest priority would be increased and become the highest, in that case all the $(n-1)$ elements have to be shifted right.

Q3 (b)



Graph G has two edge disjoint spanning trees T_1 and T_2. T_1 covers all vertices and hence any two vertices u and v have a path.

Graph T_2 covers all vertices (with different edge from T_1) and hence u and v have a different path in T_2

So for any two nodes, there are two ways to reach each other, one via path in T_1 and one via path in T_2.
cycle = path in T_1 + path in T_2.

Q4(a) (i) Since it makes exponential many recursive calls. For each objects / intervals we have two possibilities (recursive calls) (either including it or not including it).

For each interval we have two possibilities or recursive calls and there are n such intervals, so time complexity would be (2^n).

(ii) Memoization (since the no. of distinct subproblems solved is only polynomial)

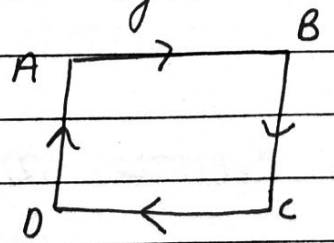
or

Dynamic program (iterative)

Q4(b) For each row find the first zero, i.e, a zero preceded by one. Use binary search to locate 0 with an additional check for the preceeding element to be 1. this takes $O(\lg n)$ time.

There are 'n' such rows ⇒ Total time $O(n \lg n)$.

Q5(a) Ensure that every node has an incoming edge, or a cycle



Q5 (b) Insertion sort, since only 10 (few) elements are to be inserted in their right position in the otherwise sorted array. this leads to best case behaviour of insertion sort - linear time

Inserting 10 elements would take ≈ 10n time (linear)

(c) 3, 2, 9, 0, 7

9 is selected as the pivot and exchanged with 7.

3, 2, 7, 0, 9

Recurse on | 3, 2, 7, 0 |

7 appears as pivot and exchanged with 0

3	2	0	7

Recurse on | 3 | 2 | 0 |

3 appears as pivot and exchanged with 0

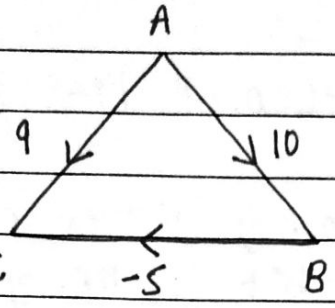
0	2	3

Recurse on | 0 | 2 |

2 appears as pivot, Recurse on | 0 |

Red Wings
Page No. ___ Date _/_/_20_

Q6 (b) No,



By Dijkstra algo,
dist (a, c) = 9

opt = 5.

or counting sort

(c) Radix sort can be used.
Radix sort internally uses counting sort which has certain assumptions.

① the elements must be integers in range 0 to K.
② where K should be O(n).

Time complexity of counting sort = O(n+K)
Space : O(n+K)
Time comp. for radix sort = d*(n+K)
worst case = O(nK)

```
Counting-SORT (A, B, K)
 let C[0...K] be a new array
 for i = 0 to K
    C[i] = 0
 for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
 for i = 1 to K
    C[i] = C[i] + C[i-1];
 for j = A.length to down to 1
    B[C[A[j]]] = A[j];
    C[A[j]] = C[A[j]] - 1;
```

Red Wings
Page No.__ Date __/__/20__

Q7(a) cost of increment = θ(no. of bits flipped)

over n increments, no. of times each bit flips is given as follows

bit	no. of flips
0	n
1	$\lfloor n/2 \rfloor$
2	$\lfloor n/2^2 \rfloor$
⋮	⋮
i	$\lfloor n/2^i \rfloor$
k	

$$\text{Total flips} = \sum_{i=0}^{k-1} \lfloor n/2^i \rfloor < n \cdot \sum_{i=0}^{\infty} 1/2^i$$

$$= n \times \left(\frac{1}{1 - 1/2} \right) \quad [\text{GP}]$$

$$= 2n$$

cost of n increments = $O(n)$ ⇒ Avg. cost per operation $= O(1)$ $\left[\frac{O(n)}{n}\right]$

Increment(A)
i = 0
while i < A.length and A[i] == 1
    A[i] = 0
    ~~A[i]~~ i = i + 1
if i < A.length
    A[i] = 1

(b) $T(n) = 2T(n/2) + O(n)$

Time : $O(n \log n)$

In another words merging takes $O(n)$ time, (scan the arrays) and there would be $O(\lg n)$ calls to merge function.
So time would be $O(n \lg n)$.

(c) Heap → $O(n \lg n)$
Each call to binary search → $O(\lg n)$
n calls to binary search → $O(n \lg n)$

Total : $O(n \lg n)$.

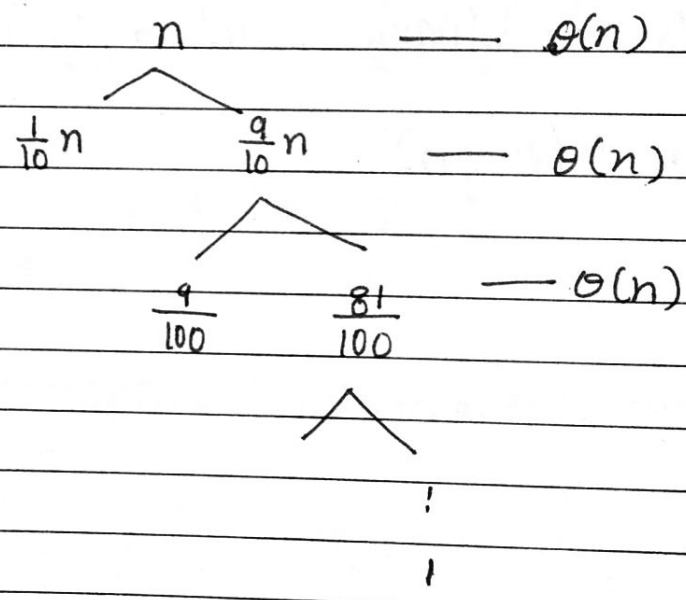
PAPER 2016

Q1. (a) Insertion sort is best when the list is already sorted. This leads to best case behaviour of insertion sort i.e $O(n)$.

(c) Assuming that the partitioning algorithm always produces 9-to-1 proportional split.

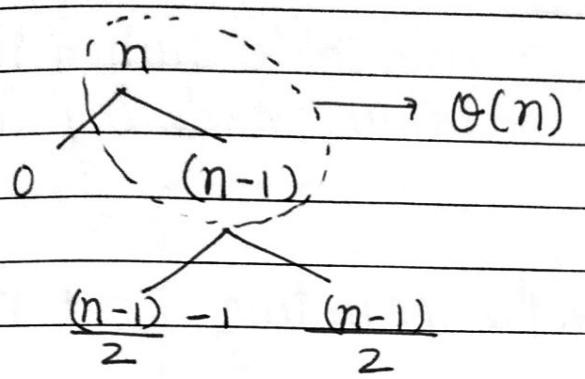
Recurrence

$$T(n) = T(9n/10) + T(n/10) + \Theta(n)$$

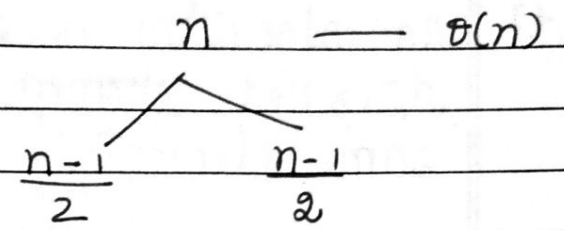


If we see the recurrence tree, the cost at each level is $\Theta(n)$ and recursion reaches a boundary condition at depth $\log_{10} n = \Theta(\lg n)$.

Total cost = $\Theta(n \lg n)$



(a)



(b)

The average case, produces a mix of "good" and "bad" splits. Suppose that these good and bad splits occur at alternate levels.

In Fig(a), first worst case split occurs at ~~the~~ root with cost $\Theta(n)$. At second level best case split occurs where $(n-1)$ is partitioned into two subarrays of size $\left(\frac{n-1}{2}-1\right)$ and $\left(\frac{n-1}{2}\right)$.

The combination of bad split followed by a good split produces three subarrays of size 0, $\frac{n-1}{2}-1$ and $\frac{n-1}{2}$ at total cost of = $\Theta(n) + \Theta(n-1) = \Theta(n)$ which is same as that of best case partitioning.

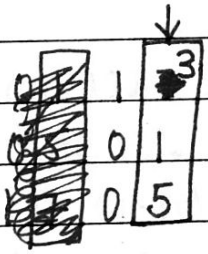
Hence the average case also has a running time of $O(n \log n)$ but with a slightly larger constant hidden in O-notation.

d) An algorithm is said to be stable when it does not change the relative ordering of same elements.

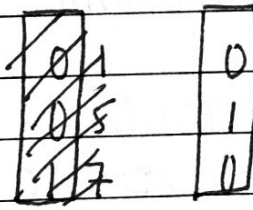
Yes, it is necessary for the counting sort to be stable.

If counting sort has to be used as an intermediate sort in radix sort, then it has to be stable otherwise it may not give sorted output.

eg

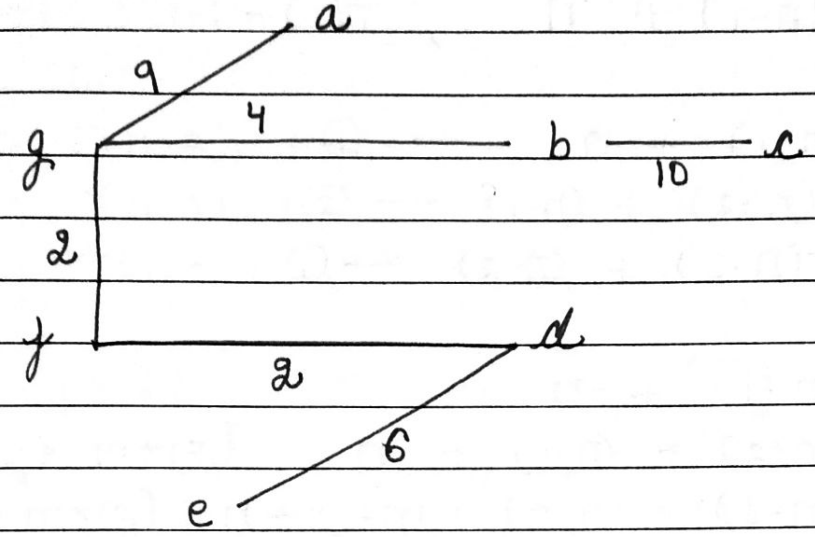


First count sort is called at one's position. It will sort digits at one's position.



Now, count sort is called at ten's digits. Now if count sort is not stable then it may interchange the relative position of digit '0' Hence the output will be <05, 01, 13> which it incorrect. Hence it needs to be stable to give correct output i.e <01, 05, 13>.

(e)



(f) { .79 , .13 , .16 , .64 , .56 , .69 , .79 , .47 }

0	→
1	→ .13 → .16
2	
3	
4	→ .47
5	→ .56
6	→ .64 → .69
7	→ .79 → .79
8	
9	

each number is multiplied by 10 and digit before decimal gives the index of the bucket.

eg. .79 x 10 = 7.9

Index = 7

Now, these buckets are individually sorted and combined to give output as.

< .13 , .16, .47, .56, .64, .69, .79, .79 >

Note: It is more appropriate to take n=8, because n elements are there.

g) $T(n) = T(n-1) + n$, $T(1) = 1$

$T(n) = T(n-1) + n$ — ①
$T(n-1) = T(n-2) + (n-1)$ — ②
$T(n-2) = T(n-3) + (n-2)$ — ③

$T(n) = T(n-1) + n$
$T(n) = T(n-2) + (n-1) + n$ [From eq 2]
$T(n) = T(n-3) + (n-2) + (n-1) + n$ [From eq 3]

Above recurrence can also be written as
$T(n) = T(n-3) + 3n - (1+2)$
or in general
$T(n) = T(n-k) + kn - (1+2+3 \dots (k-1))$ — ⑤

We know that $T(1) = 1$ [given]
So, $T(n-k) = 1$ if $n-k = 1$ or $\boxed{k = n-1}$
Put value of k in eqn s.

$T(n) = T(n-n+1) + (n-1)(n) - (1+2+3 \dots n-2)$
$T(n) = T(1) + n(n-1) - \left(\frac{(n-2)(n-1)}{2}\right)$
$T(n) = 1 + \frac{2n(n-1)}{2} - \left(\frac{n^2-3n+2}{2}\right)$
$T(n) = 1 + \frac{2n^2-2n-n^2+3n-2}{2}$
$T(n) = 1 + \frac{n^2+n-2}{2} = \frac{n^2+n}{2}$
$T(n) = O(n^2)$

Alternate: (Easy)

$T(n) = T(n-1) + n$

$T(n) = T(n-2) + n-1 + n$

$T(n) = T(n-3) + n-2 + n-1 + n$

$T(n) = T(n-K) + n-(K-1) + n-(K-2) \cdots n$

$T(n-K) = 1$ if $n-K=1$, so, $\boxed{K = n-1}$.

$T(n) = T(n-(n-1)) + (n-(n-1-1)) + n-(n-1-2) \cdots$
$\cdots n$

$T(n) = T(1) + 2 + 3 + \cdots n$

$T(n) = 1 + 2 + 3 + \cdots + n$

$T(n) = \frac{n(n+1)}{2}$ [Sum of first n natural numbers]

$T(n) = O(n)$

(h) Knapsack(n, W)

Array M[0..n][0..W]

Initialize M[0,w] = 0 for each w = 0, 1 ..., W

For i = 1, 2, --- n

For w = 0, 1, 2 --- W

~~M[i,w]~~

If $w_i > w$

~~OR IF~~ M[i,w] = M[i-1, w]

else

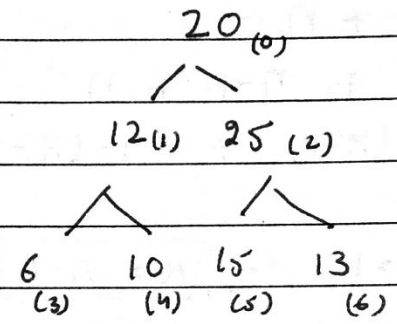
M[i,w] = max(M[i-1,w], v_i + M[i-1, w-w_i])

EndFor

EndFor

Return M[n,W]

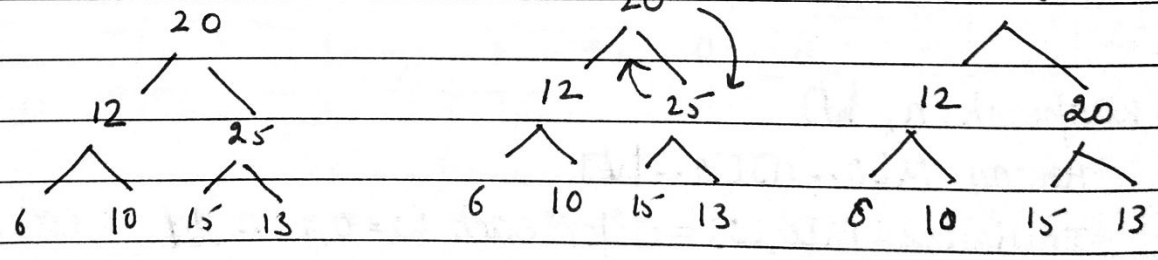
Q2(a) 20, 12, 25, 6, 10, 15, 13



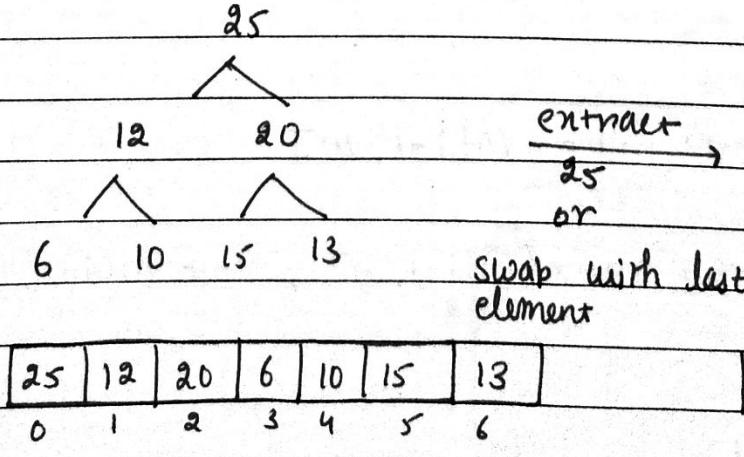
Build max heap: starting with $\lfloor \frac{n}{2} \rfloor - 1$ to 0

call max-heapify. $\left[\lfloor \frac{n}{2} \rfloor - 1 = \lfloor \frac{7}{2} \rfloor - 1 = 3 - 1 = 2 \right]$

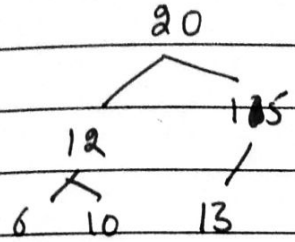
call max heapify on 2 on 12 on 20



Step 1



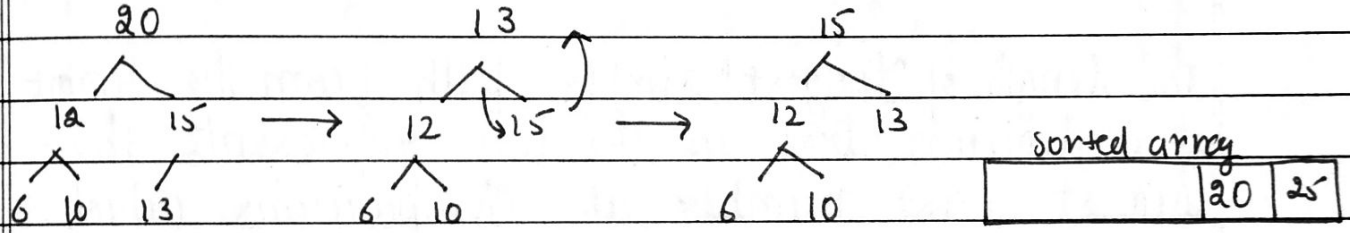
Call max heapify on root i.e 13



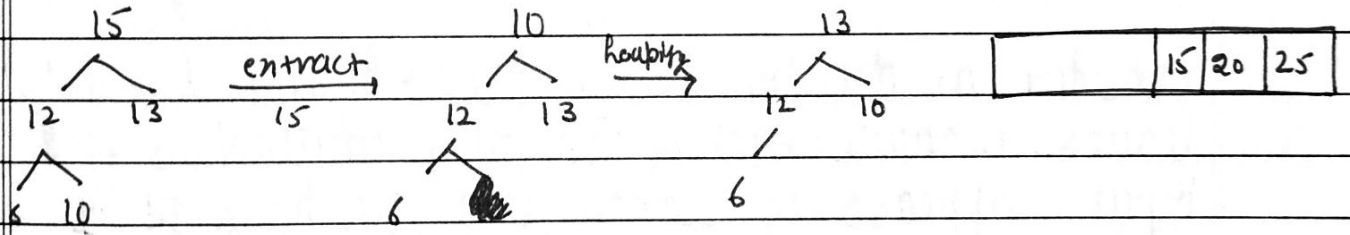
Algo:

```
HeapSort(A)
    Build-max-Heap(A)
    for i = A.length-1 down to 0
    exchange A[0] with A[i];
    A.heapsize = A.heapsize - 1;
    Max-heapify(A, 0);
```

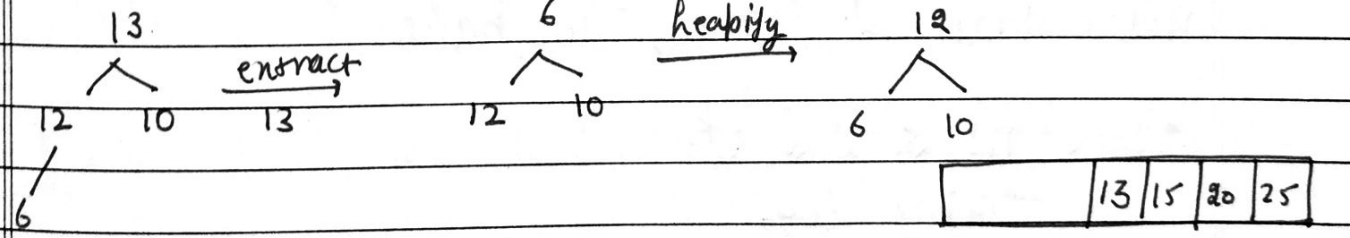
Step 2



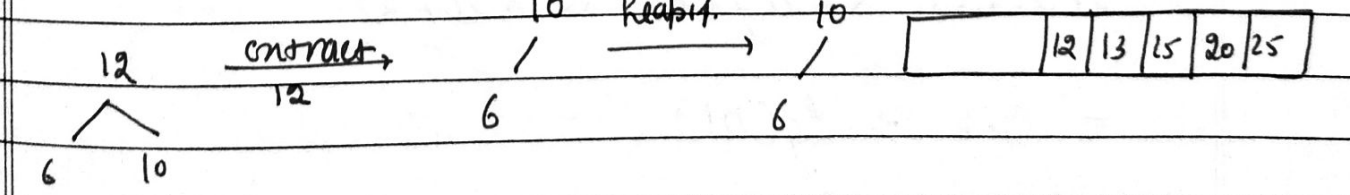
Step 3



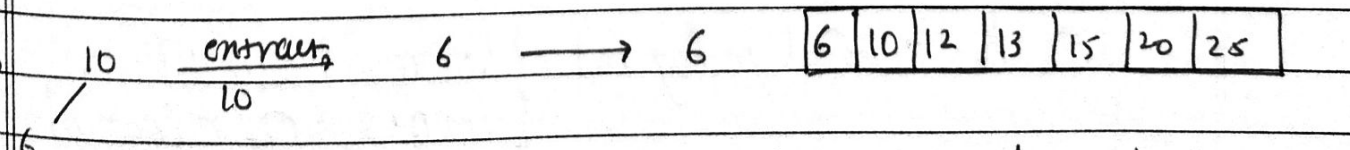
Step 4



Step 5



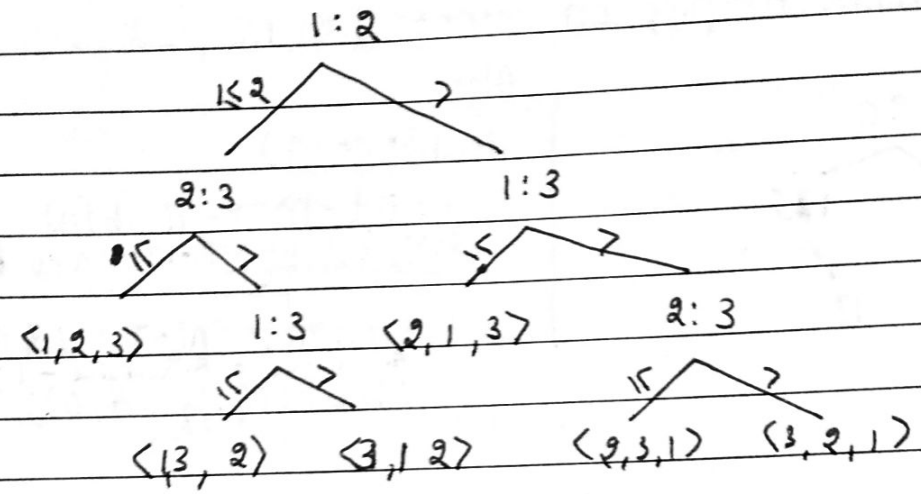
Step 6



Step 6 Last swap places ~~to~~ remaining two elements in their correct position.

Time: O(nlogn)

Q2 (b)



The length of "longest" simple path from the root of a decision tree to the leaf represents the worst case number of comparisons which in turn depend on maximum height of the tree.

Consider a decision tree of height h and l reachable leaves. Because each of the $n!$ permutations of the input appears as some leaf, we have $n! \leq l$. Since a binary tree of height h has no more than 2^h leaves, we have

$$n! \leq l \leq 2^h$$

Taking log

$$\log(n!) \leq \log(l) \leq h \log(2)$$

$$\Rightarrow h \cdot 1 \geq \lg(n!)$$

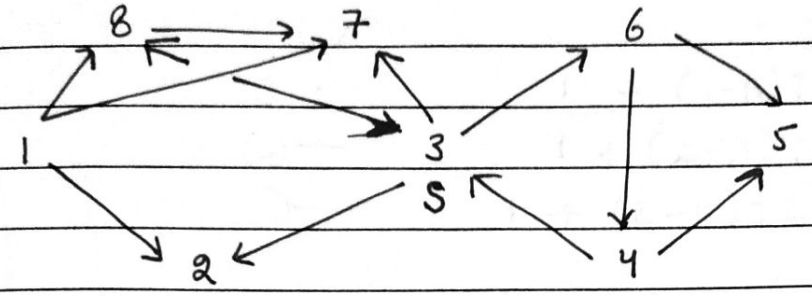
$$\Rightarrow h = \Omega(n \log n) \quad \text{[lower bound]}$$

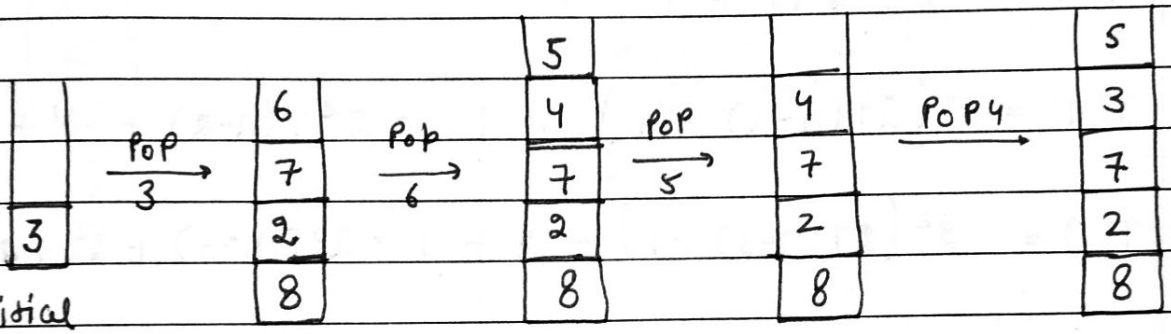
$$[\lg(n!) = \Omega(n \log n)]$$

* It shows that Heapsort and mergesort are asymptotically optimal comparison sorts.

$n! = n \times (n-1)(n-2)(n-3) \cdots$
$\leq n \times n \times n \times \cdots$ n times
$= n^n$
$\log(n!) = n \log n$ approx

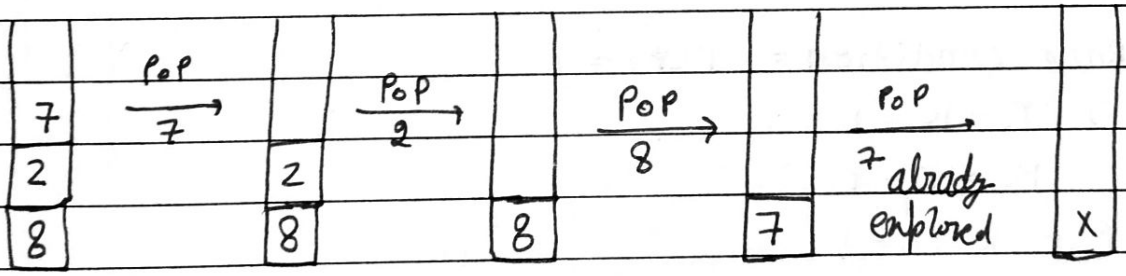
Q5(b)





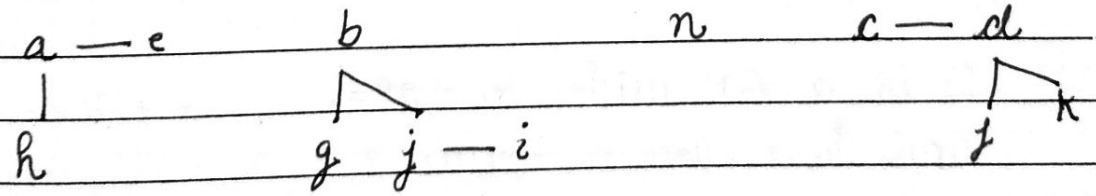
5 and 3 are already visited

DFS: 3, 6, 5, 4, 7, 2, 8



1 can not be explored because it has no incoming edge.

Q6(b)



Q7(b) The Prims algorithm is a greedy because every time we select the minimum weight edge ~~from~~ with one end in ~~either other in~~ S and other in V-S. At each step we are making a greedy decision.

$O(|V|\log|V|) + O(|E|\log|V|) = O((|V|+|E|)\log|V|) = O(|E|\log|V|)$

PAPER - 2015

Q1. (d)

$$T(n) = 2T(n-1) + 1 \quad \text{— ①}$$
$$T(n-1) = 2T(n-2) + 1 \quad \text{— ②}$$
$$T(n-2) = 2T(n-3) + 1 \quad \text{— ③}$$

$T(n) = 2T(n-1) + 1$

$T(n) = 2(2T(n-2) + 1) + 1 = 2^2 T(n-2) + 2^1 + 1$

$T(n) = 2^2(2T(n-3) + 1) + 2 + 1 = 2^3 T(n-3) + 2^2 + 2^1 + 1$

$T(n) = 2^3(2T(n-4) + 1) + 2^2 + 2^1 + 2^0 = 2^4 T(n-4) + 2^3 + 2^2 + 2^1 + 2^0$

$\vdots$

$T(n) = 2^k T(n-k) + 2^{k-1} + 2^{k-2} + 2^{k-3} \text{ ---- } + 2^2 + 2^1 + 2^0$

Base condition = $T(1) = 1$

$\Rightarrow n - k = 1$

$k = n - 1$

Put $k = n - 1$

$T(n) = 2^{n-1} T(1) + 2^{n-2} + 2^{n-3} + 2^{n-4} + \text{---} + 2^2 + 2^1 + 2^0$

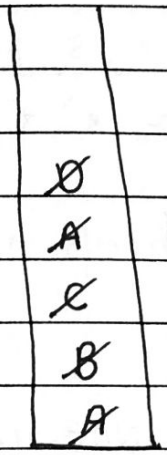
$T(n) = 2^0 + 2^1 + 2^2 \text{ ------ } + 2^{n-4} + 2^{n-3} + 2^{n-2} + 2^{n-1}$

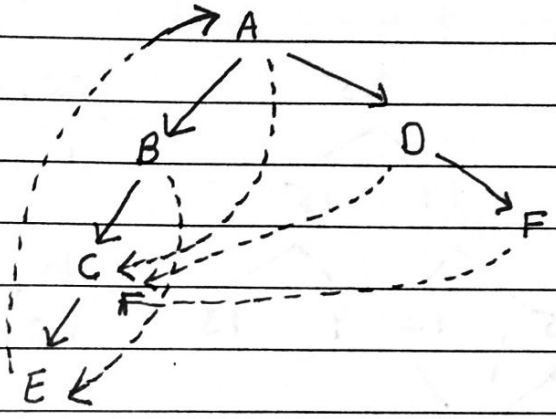
It is a G.P. with $a = 2^0 = 1$, $r = 2$ and there are n terms

$$T(n) = \frac{a(r^n - 1)}{r - 1} = \frac{1 \cdot (2^n - 1)}{2 - 1} = 2^n - 1$$

$T(n) = O(2^n)$, which is exponential

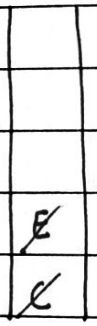
(e) Using DFS Recursive



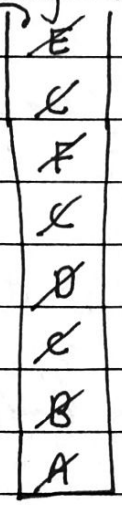


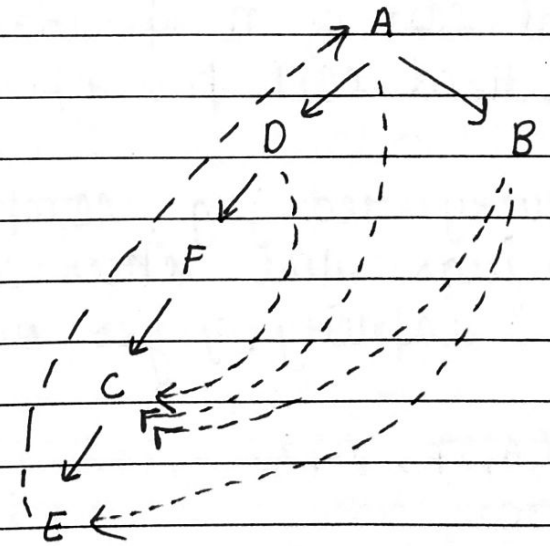
DFS: A, B, C, E, D, F,

G and H cannot be explored because G has no incoming edge.



Using DFS Iterative



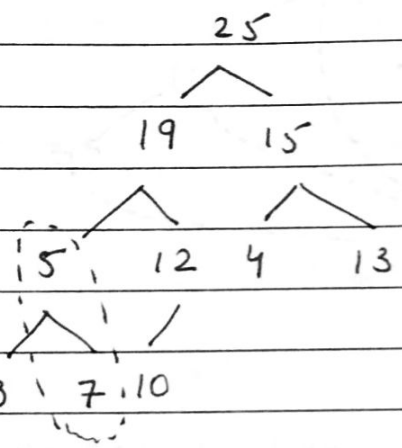


DFS: A, D, F, C, E, B

Forward edge (u, v) → v is descendant of u eg, (D, E), (A, C)
Back edge (u, v) → v is ancestor of u eg (E, A)
Cross edge (u, v) → No ancestor, descendant relationship eg, (B, C) (B, E)

(f) No, sorted array in quicksort gives worst case behaviour.

(g)



25, 19, 15, 5, 12, 4, 13, 3, 7, 10

1 2 3 4 5 6 7 8 9 10

No, it is not max-heap neither min-heap.

7 is a child of 5. Not a max-heap.

(h) Suppose in a worst case every i^{th} operation costs i. So a sequence of n operations will cost $1+2+3+\cdots+n$ i.e $\frac{n(n+1)}{2}$.

So, Total cost of n operations in worst case $= O(n^2)$

& Amortized cost per operation $= \frac{O(n^2)}{n} = O(n)$

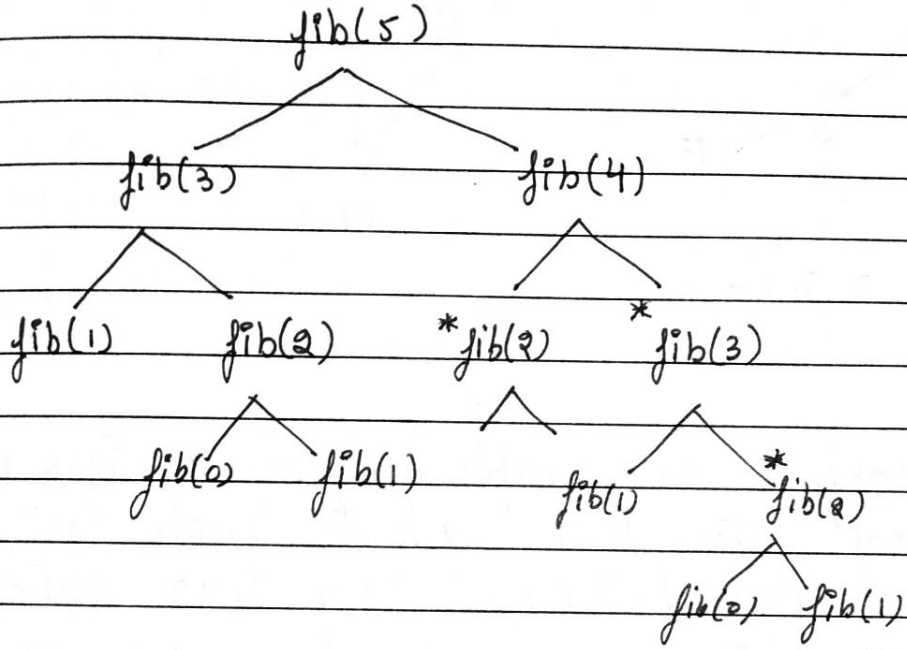
Q2(a) For every red jug, compare it with all blue jugs and check if they are of same capacity. if yes make them as pairs.

Q3(a)

```
Rselect(A, p, r, i)
if (p==r)
    return A[p]
q = randomized-partition(A, p, r)
K = q - p + 1
if i==K
    return ~~elements from p to q~~ A[q]
else if i<K, return Rselect(A, p, q-1, i)
else return Rselect(A, q+1, r, i-K)
```

Rselect will give i^{th} smallest element. Rescaan the original array to find all elements <= i^{th} smallest element returned by Rselect

Q5(a) Memoization is a technique of storing the values of solution of subproblems so that if a subproblem repeats is doesn't need to be solved again.



$$fib(n) = \begin{cases} 0 & \text{if } n=0 \\ 1 & \text{if } n=1 \\ fib(n-2) + fib(n-1) & \text{if } n>1 \end{cases}$$

```
int fib(int n){
  if (n<=1)
    return n;
  return fib(n-2) +
         fib(n-1);
}
```

Recurrence → $T(n) \geq 2T(n-1)+1$

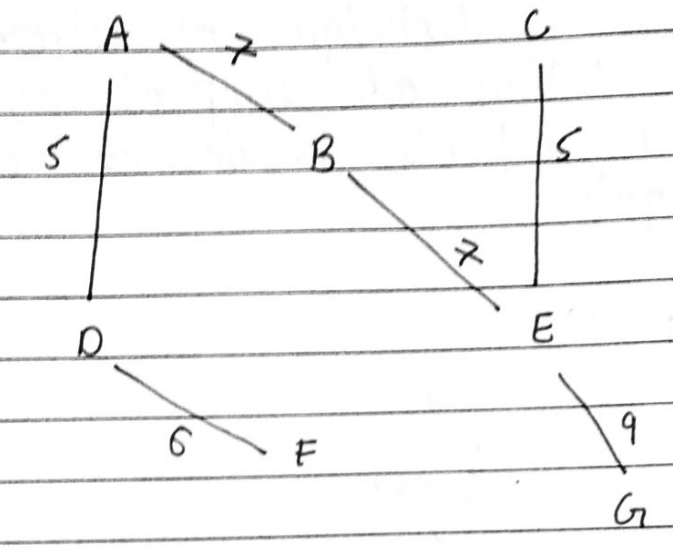
Time → $= O(2^n)$

Here we have repeating subproblems such as fib(0), fib(1), fib(2) and fib(3), they can be solved once and their value is stored in an array for later use.

Memoization decreases the no. of recursive calls from 15 to 6 (including fib(5)). For fib(5) it makes six calls. For fib(n) it will make $n+1$ calls which is $O(n)$ while $O(2^n)$ without memoization.

It reduces exponential time to polynomial

Q6 (a)



$E = V - 1 = 7 - 1 = 6$

Q6 (b) A bipartite graph is a graph whose vertices may be partitioned into two subsets such that there is no edge between any two vertices in the same set.

[-1 → no color
1 → red color
0 → blue color]

```
bool isBipartite( ~~int G~~ G , ins src) {
    declare a color array and initialize all elements to -1,
    assign starting vertex as red color [color[src] = 1;]
    q.push( src);
    while (!q.empty()) {
     int u = q.front(); q.popfront();
     For all vertices 'v' that are having an edge incident to u.
        if ( color[v] == -1 ) {
            color[v] = 1 - color[u]; //Assign alternate color
            q.push(v);
        } else if ( color[v] == color[u] ) {
            return False;
        }
}
```

[A graph is not bipartite if it contains an odd length cycle].

Paper - 2014

Q1(a) TEXT : A A A A A A

Pattern: A A A

(b) $T(n) = T(n-1) + \theta(n)$

$T(n) = T(n-1) + O(n)$

$T(n) = T(n-2) + 2O(n)$

$T(n) = T(n-3) + 3O(n)$

⋮

$T(n) = T(n-k) + k.O(n)$

when $n-k=1$, $k = n-1$

$T(n) = T(1) + (n-1).O(n)$

$T(n) = 1 + O(n^2)$

$T(n) = O(n^2)$ (worst case)

The cost of solving a problem of n size is equal to the cost of sorting (n-1) elements recursively + $O(n)$ [because we have to place n in it's correct position which is taking $O(n)$ time in worst case (for loop is there).

(c) ~~Each iterati~~

~~one iteration of inner for loops take time = O(1)~~

(c) Time taken by one iteration of inner for loop = $O(1)$
Time taken by n iterations of inner for loop = $O(n)$
Time taken by one iteration of outer for loop = $O(n)$
Time taken by n iterations of outer for loop = $O(n^2)$.

(e) Inplace : insertion, quick sort, heapsort
Not inplace : count sort

Q1 Similar to Q2(a) 2017

(k)

operations	No. of times the operation is performed	Total time for the operation.
Build max Heap	1	O(n)

n: no. of vertices
m: no. of edges

(h)

operations	Time taken by the operation	No. of times the operation is performed	Total time for the operation
1. Build min heap	O(n)	1	O(n)
2. Extract-min	O(log n)	n (vertices)	O(n log n)
3. Decrease-key	O(log n)	m (edges)	O(m log n)
Total time:			

$O(n) + O(n \log n) + O(m \log n)$
$= O(n \log n) + O(m \log n)$
$= O((m+n) \log n)$
$\simeq O(m \log n)$ [m > n]
OR $O(|E| \log |V|)$ edges

(i) Same as Q7(a) 2017

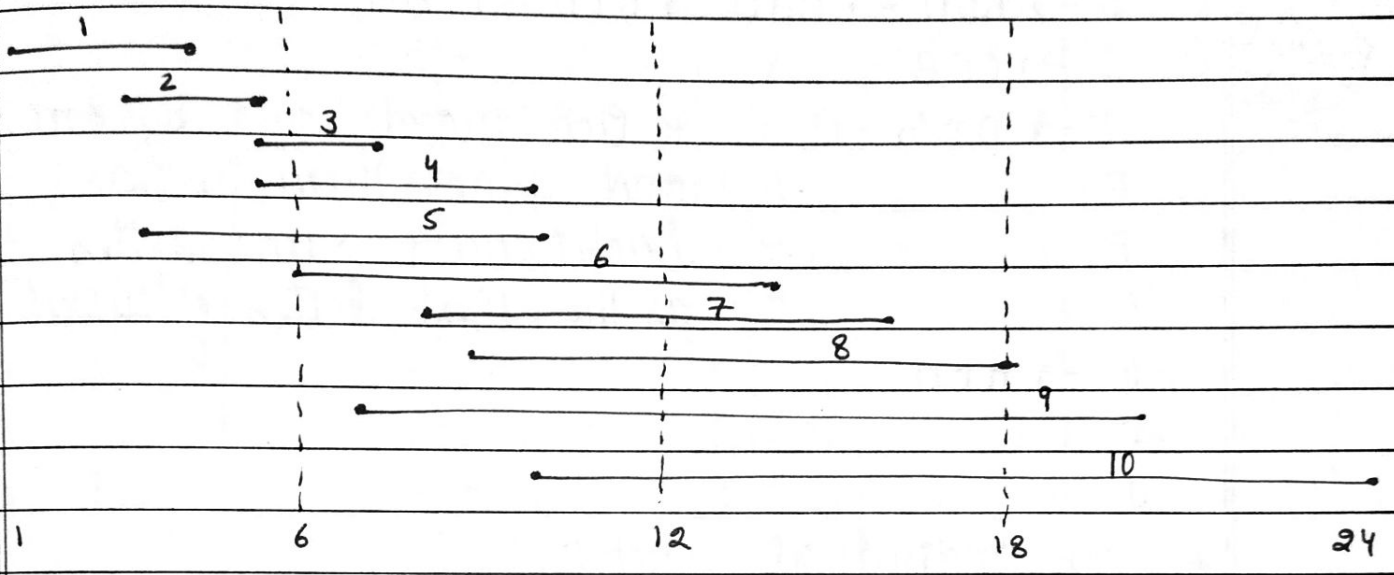
(j)

eg.	obj 1	obj 2	obj 3	
Profit	2	5	8	
weight	1	2	4	
P/w	2	2.5	2	W = 4

If we go by greedy strategy.
1) select obj 2 (max P/w), cost = 5, Rem. cap = 4-2 = 2
2) select obj (1), cost = 5 + 2 = 7, Rem. cap = 2 - 1 = 1
3) obj 3 cannot be selected.
So profit comes out to be 7, but it is not optimal because if we had selected obj 3, the profit would be 8.

Q2. (a) Same as Q2(a) 2018

(b)



select the activity with smallest finish time first.
$S = \{1, 3, 7\}$

Q3 (a) max-heapify : $O(\log n)$
build max heap : $O(n)$
Heapsort : $O(n \log n)$

In Heapsort we extract the root (max) vertex and swap it with the last element. Because of swapping the max-heap property may get disturbed so we call max-heapify on root vertex which may take $O(\log n)$ time in worst case because we may have to go all the way down to the height of tree.

So extract-max() take $O(\log n)$ time and it is called n times (for every element)
Total time : $O(n \log n)$.

Q4 (a)

Bucket sort

A → AID
B → BED → BAD → BID → BAG
C → CAB
D → DIG
E
F
G
H → HID
I
J

* Pick words one by one and place them in a bucket with same letter as the first letter of word.

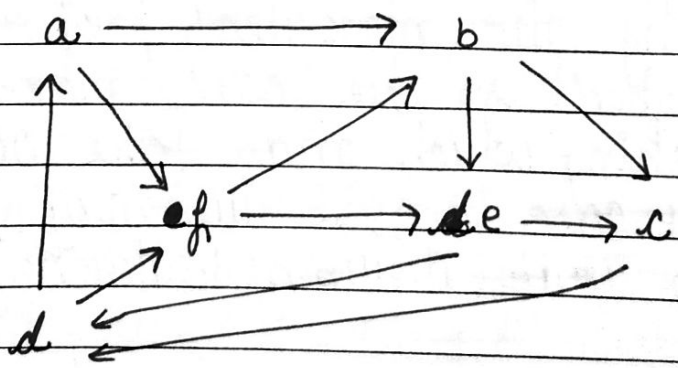
* Sort individual buckets

A → AID
B → BAD → BAG → BED → BID
C → CAB
D → DIG
H → HID

* Concatenate buckets top to down

AID, BAD, BAG, BED, BID, CAB, DIG, HID

(b)



Next page →

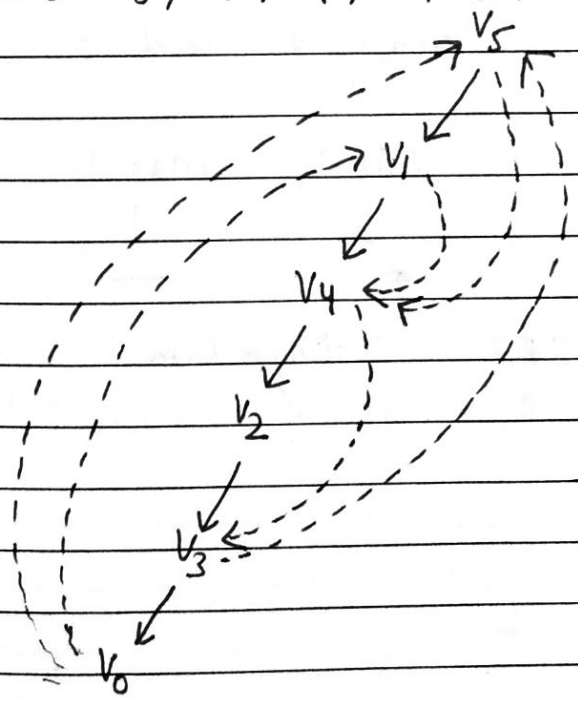
Red Wings
Page No.__ Date __/__/__20__

Q4(a) Radix sort

B E D		C A B		C A B		A I D
B A D		B E D		B A D		B A D
H I D		B A D		B A G		B A G
B I D	→	H I D	→	B E D	→	B E D
D I G		B I D		H I D		B I D
B A G		A I D		B I D		C A B
C A B		D I G		A I D		D I G
A I D		B A G		D I G		H I D

(b) DFS Recursive : starting with V_5 (f)

DFS: $V_5, V_1, V_4, V_2, V_3, V_0$



~~V_0~~
~~V_3~~
~~V_2~~
~~V_4~~
~~V_1~~
~~V_5~~

Tree edges
~~Forward~~: $(V_5, V_1)(V_1, V_4)(V_4 - V_2)(V_2 - V_3)(V_3 - V_0)$

Back edges: ~~$(V_5, V_0)(V_1, V_0)$~~ $(V_0, V_5)(V_0, V_1)(V_3, V_5)$

Forward edges: $(V_5, V_4)(V_1, V_4)(V_4, V_3)$

cross edges : No.

Q4 (c) Comp 1 → Take any two elements say a, b and compare them, its maximum is max1 and minimum is min1.

Comp 2 → Take other two elements say c, d and comare them, its maximum is max2 and minimum is min2.

Comp 3 → Compare Max1 and Max2 to get ultimate max element.

Comp 4 → Compare min1 and min2 to get ultimate min element.

Comp 5 → Compare remaining elements and place accordingly.

eg, 2 3 1 0

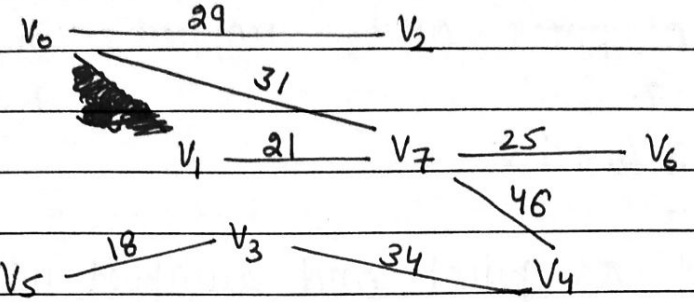
Comp 2 and 3 | Comp 1 and 0

Min1	Max1	Min2	Max2
2	3	0	1
*	—	*	—

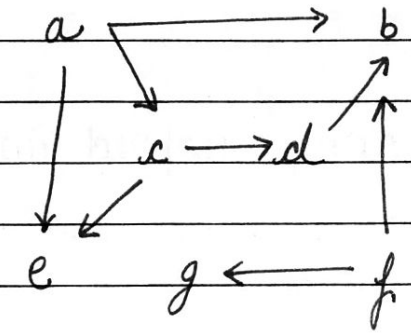
Ultimate min: 0 | Comp 1 & 2: 1, 2 | Ultimate max: 3

order : 0, 1, 2, 3

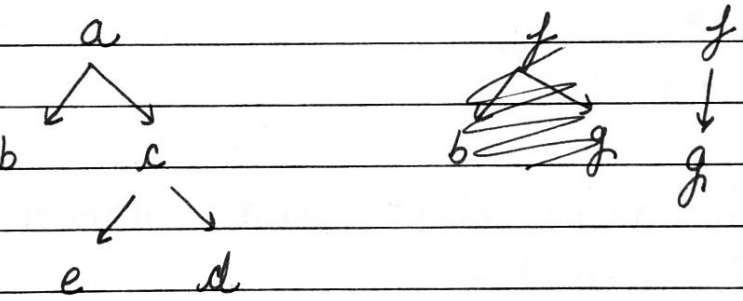
Q5(a)



Q6 (a)



DFS on A :



(b) $A = \langle 3, 2, 9, 0, 7, 5, 4, 8 \rangle$

1) 9 is selected as pivot and swapped with 8
3 2 8 0 7 5 4 9
Recurse on 3 2 8 0 7 5 4

2) 8 is selected as pivot and swapped with 4
3 2 4 0 7 5 8
Recurse on 3 2 4 0 7 5

3) 7 is selected as pivot and swapped with 5
3 2 4 0 5 7
Recurse on 3 2 4 0 5

4) 5 is selected as pivot and swapped with 5
3 2 4 0 5
Recurse on 3 2 4 0

5) 4 is selected as pivot and swapped with 0
3 2 0 4
Recurse on 3 2 0

6) 3 is selected as pivot and swapped with 0
0 2 3
Recurse on 0 2

7) 2 is selected as pivot and swapped with 2
0 2
Recurse on 0

8) Base condition (ans = 0)

Worst case: Always select largest element as pivot.

(c)
```
int fib(int n) {
  if(n <= 1)
      return n;
  if(m[n] != -1) {
     return m[n];
  } else {
     m[n] = fib(n-2) + fib(n-1);
  }
}
```

Q7(b) Aggregate: We can pop as much as elements that we push on the stack, which in worst case is O(n).
Average cost per operation = $\frac{O(n)}{n} = O(1)$.

Accounting method:
Let us assign the amortized cost as follows
Push 2
Pop 0
Multipop 0

We charge the push operation 2 units (eg 2 dollars) out of which 1 dollar is used to pay the actual cost and remaining 1 dollar is saved as a credit, which can be used to pay for future operations.

Since we can pop as many elements, we pushed onto the stack. Therefore we always have enough credit to pay for pop operations.
amortized For a sequence of n operations.
Total cost is O(n) and amortized cost per operation: O(1)

(c) (i) there are only a polynomial no. of subproblems.
(ii) the solution to the original problem can be easily computed from the solutions to the subproblems.
(iii) there is a natural ordering of subproblems from "smallest" to "largest".
(iv) It is generally used when there are repeating subproblems.

Red Wings
Page No.___ Date _/_/_20_

Interval Scheduling: $\underbrace{O(n \log n)}_{\text{sorting}} + \underbrace{O(n)}_{\text{selecting intervals}} = O(n \log n)$

~~Insertion-sort(A)~~
~~key =~~

```
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
```

Time:
Best: $O(n)$
Worst: $O(n^2)$

```
Merge(A, p, q, r)
  n1 = q - p + 1
  n2 = r - q
  Let L[1.. n1+1] and R[1.. n2+1] be new arrays
  for(i = 1 to n1)
    L[i] = A[p+i-1];
  for j = 1 to n2
    R[j] = A[q+j];
  L[n1+1] = ∞ , R[n2+1] = ∞
  i = 1 , j = 1
  for K = P to r
    if L[i] ≤ R[j]
      A[K] = L[i]
      i = i+1
    else A[K] = R[j]
         j = j+1
```

Time: $O(n)$
Space: $O(n)$

```
Merge-sort (A, p, r)
  if p < r
    q = ⌊(p + r) / 2⌋
    Merge-sort (A, p, q)
    Merge-sort (A, q+1, r)
    Merge (A, p, q, r)
```

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Time: $O(n \log n)$

```
Bubblesort (A)
  for i = 1 to A.length - 1
    for j = A.length down to i+1
      if (A[j] < A[j-1])
        exchange A[j] with A[j-1]
```

```
Max-heapify (A, i)
  l = left(i)
  r = right(i) ; largest = i;
  if l ≤ A.heapsize and A[l] > A[i]
      largest = l
  if r ≤ A.heapsize and A[r] > A[largest]
      largest = r
  if largest != i
      exchange A[i] with A[largest]
      Max-heapify (A, largest)
```

Time: $O(\log n)$
$O(\log n)$ comparisons

$T(n) \leq T(2n/3) + \Theta(1)$

```
Build-max-heap (A)
  A.heapsize = A.length
  for i = ⌊A.length / 2⌋ downto 1
    Max-heapify (A, i)
```

Time: $O(n)$

```
Heap-sort(A)
  Build-max-heap(A)                    Time = O(nlogn)
  for i = A.length down to 2
      exchange A[i] with A[1]
      A.heapsize = A.heapsize - 1
      Max-heapify(A, 1)
```

```
Quicksort(A, p, r)                     Time = O(nlogn) [Best case]
  if (p < r)
     q = Partition(A, p, r)
     Quicksort(A, p, q-1)
     Quicksort(A, q+1, r)
```

```
Partition(A, p, r)                     Time = O(n)
  x = A[r]
  i = p-1
  for j = p to r-1
     if A[j] ≤ x
        i = i+1
        exchange A[i] with A[j]
  exchange A[i+1] with A[j]
  return(i+1)
```

Time:

Worst case : $O(n^2)$ $\quad T(n) = T(n-1) + \theta(n)$

Best case : $O(n\log n)$ $\quad T(n) = 2T(n/2) + \theta(n)$

Avg case : $O(n\log n)$ $\quad T(n) = T(9n/10) + T(n/10) + \theta(n)$

```
Randomized-partition(A, p, r)
  i = random(p, r)
  exchange A[i] with A[r]
  return Partition(A, p, r)
```

```
Counting-sort(A, B, K)
  let C[0..K] be a new array
  for i = 0 to K
    C[i] = 0
  for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
  for i = 1 to K
    C[i] = C[i] + C[i-1]
  for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

Time: $\Theta(n+K)$
If $K = O(n)$
then time $O(n)$

It is stable

```
Radix-sort(A, d)
  for i = 1 to d
    use a stable sort
    to sort array A
    on digit i
```

Time: $\Theta(d(n+K))$

```
Bucket-sort(A)
  let B[0..n-1] be a new array
  n = A.length
  for i = 0 to n-1
    make B[i] an empty list
  for i = 1 to n
    insert A[i] into list B[⌊n * A[i]⌋]
  for i = 0 to n-1
    sort list B[i] with insertion sort
  concatenate the lists B[0], B[1] --- B[n-1] together in order
```

Time: $O(n)$

Randomized-select(A, p, r, i) Time : O(n)

```
if p == r
    return A[p]
q = randomized-partition(A, p, r)
K = q - p + 1
if (i == K)
    return A[q]
else if i < K
    Randomized-select(A, p, q-1, i)
else
    Randomized-select(A, q+1, r, i-K)
```

Mst-Kruskal(G, w)

```
A = ϕ
for each vertex V ∈ G.V
    Make-set(V)
sort the edges of G.E in increasing order by weight w
for each edge(u,v) ∈ G.E taken in increasing order by w.
    if ( findset(u) != findset(v))
        A = A U {(u,v)}
        union(u, v)
return A
```

Time: O(E log V)

```
Mst-prim (G, w, s)
  for each u ∈ G.V
     u.key = ∞
     u.π = NIL
  s.key = 0
  Q = G.V
  while Q ≠ ∅
     u = extractmin(Q)
     for each v ∈ G. Adj[u]
        if v∈Q and v.key > w(u,v)
           v.key = w(u,v)
           v.π = u                    Time O(E log V)
```

```
Dijkstra(G, w, s)
  for each u ∈ G.V
     u.d = ∞
     u.π = NIL
  s.d = 0
  S = ∅
  Q = G.V
  while Q != ∅
     u = extract-min(Q)
     S = S ∪ {u}
     for each v ∈ G.Adj[u]
        if( v.d > u.d + w(u,v))
           v.d = u.d + w(u,v)
           v.π = u                    Time = O(E log V)
```

quick sort

Best case : $T(n) = 2T(n/2) + \theta(n) \rightarrow O(n \log n)$

Worst case : $T(n) = T(n-1) + \theta(n) \rightarrow O(n^2)$

Heapify

$T(n) <= T(2n/3) + \theta(1)$

Mergesort

$T(n) = 2T(n/2) + \theta(n) \rightarrow O(n \log n)$

weighted interval scheduling

$OPT(i) = \max(OPT(i-1), OPT(P(i)) + v_i)$

Subset-sum

If $w_i > W$

then $OPT(i, w) = OPT(i-1, w)$, otherwise

$OPT(i, w) = \max(OPT(i-1, w), w_i + OPT(i-1, W-w_i))$

0-1 knapsack

If $w_i > w$

then $OPT(i, w) = OPT(i-1, w)$. otherwise

$OPT(i, w) = \max(OPT(i-1, w), v_i + OPT(i-1, w-w_i))$

	Best	Average	Worst	Aux. Space	Stable	In place
Selection Sort	$\Omega(n^2)$	$\theta(n^2)$	$O(n^2)$	$O(1)$	NO but can be made	Yes
Bubble	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$	$O(1)$	Yes	Yes
Insertion	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$	$O(1)$	Yes	Yes
Heap Sort	$\Omega(n\log n)$	$\theta(n\log n)$	$O(n\log n)$	$O(1)$	No	Yes.
Quick Sort	$\Omega(n\log n)$	$\theta(n\log n)$	$O(n^2)$	$O(1)$	No but can be made	Yes
Merge sort	$\Omega(n\log n)$	$\theta(n\log n)$	$O(n\log n)$	$O(n)$	Yes	NO
Count sort	$\Omega(n+k)$	$\theta(n+k)$	$O(n+k)$	$O(n+k)$	Yes	NO
Radix sort	$\Omega(nk)$	$\theta(nk)$	$O(nk)$ $O(d*(n+k))$	$O(n+k)$	Yes	NO
Bucket sort	$\Omega(n+k)$	$\theta(n+k)$ $\theta(n)$ Linear	$O(n^2)$	$O(n)$	Yes	NO

Comparison of various time complexities

$O(c) < O(\log(\log n)) < O(\log n) < O(n^{1/2}) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(n^k) < O(2^n) < O(n^n) < O(2^{2^n})$

Sum of first n natural numbers $\frac{n(n+1)}{2}$

Sum of GP : $S_n = \frac{a(r^n - 1)}{r - 1}$

$$S_\infty = \frac{a}{1 - r} \quad \text{or} \quad \frac{a}{r - 1}$$

Masters Theorem

$T(n) = aT(n/b) + f(n) \quad a \geq 1 ; b > 1$

Case 1: If $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$
then $T(n) = \Theta(n^{\log_b a})$

$[n^{\log_b a} > f(n)]$

Case 2: If $f(n) = \Theta(n^{\log_b a})$
then $T(n) = \Theta(n^{\log_b a} \lg n)$

Case 3: If $f(n) = \Omega(n^{\log_b a + \epsilon}) \; ; \epsilon > 0$

and $af\left(\frac{n}{b}\right) \leq c \cdot f(n) \; ; c < 1 \; \forall n$

then $T(n) = \Theta(f(n))$