DAA assignment 1

February 2020

1. (a) **Analysis**

To find the common element present in 3 given arrays the approach implemented is to start with the first elements of each array and whichever is highest, the other 2 arrays are traversed till we get an element equal to or just greater than that element in each array. If the element values pointed to at in each array are equal, then that is the common value and can be returned. Otherwise keep taking the maximum values and traverse through the arrays till result is reached. In each array we traverse all the elements before the common element once. In worst case the common element could be at the end. So the worst case time complexity will be O(p+q+r).

Algorithm

```
int findcommon( vector<int> x,vector<int> y,vector<int> z){
        for(int i=1,j=1,k=1;1;){
            if(x[i]==y[j] && y[j]==z[k]) return x[i];
            else if(x[i]>=y[j] && x[i]>=z[k]){
                while(x[i] > y[j])j++;
                while(x[i] > z[k])k++;
            }
            else if(y[j] >= x[i]&&y[j] >= z[k]){
                while(y[j] > x[i])i++;
                while(y[j] > z[k])k++;
            }
            else if(z[k]>=y[j]&&z[k]>=x[i]){
                while(z[k] > y[j])j++;
                while(z[k] > x[i])i++;
            }
        }
}
```

(b) **Analysis**

Here the only difference from the previous algorithm is that there is a possibility there is no common element in which case it is necessary to be mindful of the conditions so as not to go beyond the array. In case there are no more elements to traverse in any one of the array then this means that there is no common element.

Algorithm

```
int findcommon( vector<int> x,vector<int> y,vector<int> z){
    int p=x.size(),q=y.size(),r=z.size();
    for(int i=1,j=1,k=1;i<=p && j<=q && k<=r;){
        if(x[i]==y[j] && y[j]==z[k]) return x[i];
        else if(x[i]>=y[j] && x[i]>=z[k]){
            while(j<=q && x[i]>y[j])j++;
            while(k<=r && x[i]>z[k])k++;
        }
        else if(y[j]>=x[i] && y[j] >= z[k]){
            while(i<=p && y[j] > x[i])i++;
            while(k<=r && y[j] > z[k])k++;
        }
        else if(z[k] >= y[j] && z[k] >= x[i]){
            while(j <= q && z[k] > y[j])j++;
            while(i <= p && z[k] > x[i])i++;
        }
    }
    return -1;
}
```

2. **Analysis**

To find the common element present in n given arrays of size m each take first 2 arrays and find all the common elements in a function which is similar to the algorithm above where in we traverse each array towards the maximum element being pointed to and in case they are equal store that element in array and continue. This can be implemented with time complexity of the order o(m). Now using a loop find common element array for the next array and the common array which we got from previous iteration. The loop will run for n-1 times and each time it calls the function which is o(m). So time complexity of this algorithm will be o(n*m).

Algorithm

```
vector<int> common(vector<int> a,vector<int> b);
int findcommon(vector<vector int>> a,int m,int n){
    vector<int> v=a[1];
        for(int i = 2;i <= n; i++){
            v=common(a[i],v);
        }
        return v[0];
}
```

3. (a) **Analysis**

To find the number of common elements present in 2 given arrays the approach implemented is to start with the first elements of each array and whichever is highest, the other array is traversed till we get an element equal to or just greater than that element. If the element values pointed to at in each array are equal, then it is one of the common values. Otherwise keep taking the maximum values and traverse through the arrays till end of either one of the arrays is reached. In each array we traverse all the elements once. So the worst case time complexity will be O(p+q).

Algorithm

```
int noofcommon(vector<int> a,vector<int> b,int p,int q){
    int p=a.size(),q=b.size(),t=0;
    for(int i=1,j=1;i<=p&&j<=p;){
            if(a[i]==b[j]){
                    t++;
                    i++;
                    j++;
            }
            else if(a[i]<b[j]) i++;
            else j++;

    }
    return t;
}
```

(b) **Analysis**

Here the only difference from the previous algorithm is that there is a possibility that the same element is repeated more than once. So number of pairs of common elements will be number of times that element is repeated in first array multiplied by the same in the second array for each different common element.

Algorithm

```
int noofcommon(vector<int> a,vector<int> b,int p,int q){
    int p=a.size(),q=b.size(),t=0;
    for(int i=1,j=1;i<=p&&j<=p;){
            if(a[i]==b[j]){
                    int k=1,l=1;
                    i++;
                    j++;
                    while(i<p&&a[i]==a[i+1]) {
                            k++;
                            i++;
                    }
                    while(j<q&&b[j]==b[j+1]) {
                            l++;
                            j++;
                    }
                    t=t+k*l;

            }
            else if(a[i]<b[j]) i++;
            else j++;

    }
    return t;
}
```

4. Analysis

Taking that $F_k(x) =$ where F_k is the function after the k^{th} iteration of horner's rule while $F_k^{'}(x)$ is the derivative of $F_k(x)$.
$F_k(x) \quad = a_n x^k + a_{n-1}x^{k-1} + . \ . \ . + a_{n-k}$
$F_k^{'}(x) \quad = a_n x^{k-1} + a_{n-1}x^{k-2} + . \ . \ . + a_{n-k+1}$
$F_{k+1}(x) = F_k(x) * x + a_{n-k-1}$
$F_{k+1}^{'}(x) = F_k(x) + xF_k^{'}(x) = (a_n x^k + k a_n x^k) + . \ . \ . + a_{n-k} = \ (k+1)_n x^k \ + \ . \ . \ . \ + a_{n-k}$

Hence above statements are proved by induction and horner's rule
To find each $F_k(x)$ takes 2 operations and calculating $F_k^{'}(x)$ from previous $F_{k-1}(x)$ and $F_{k-1}^{'}(x)$ takes two more operations and total of 4 operations per stage and thus we get total of $4n$ operations

Algorithm

```
1: procedure F AND F'(a[ ], v)        ▷ Finding F and F' of coeffs a[] at value v
2:     f ← a[n]
3:     f' ← 0
4:     i ← 1
5:     while i <= n do
6:         f' ← xf' + f
7:         f ← xf + a[ n − i ]
8:         i ← i + 1
9:     return make_pair (f,f')
```

5. If n is **even** then :

$A_{even}(x) \;=\; (\ldots(a_n x^2 + a_{n-2}) + \ldots) + a_0$

$A_{odd}(x) \;=\; (\,(\ldots(a_{n-1}x^2 + a_{n-3}) + \ldots) + a_1)x$

$\therefore$ A(v) = A$_{even}(v) + A_{odd}(v)$

$\therefore$ A(-v) = A$_{even}(-v) + A_{odd}(-v) \;=\; A_{even}(v) - A_{odd}(v)$

If n is **odd** then :

$A_{even}(x) \;=\; (\ldots(a_{n-1}x^2 + a_{n-3}) + \ldots) + a_0$ because p=$\frac{n-1}{2}$

$A_{odd}(x) \;=\; (\,(\ldots(a_n x^2 + a_{n-2}) + \ldots) + a_1)x$ because $2q - 1 = n$

$\therefore$ A(v) = A$_{even}(v) + A_{odd}(v)$

$\therefore$ A(-v) = A$_{even}(-v) + A_{odd}(-v) \;=\; A_{even}(v) - A_{odd}(v)$

Thus, we have shown that the A_{even} and A_{odd} are the functions corresponding to the two functions given in the question. Therefore we can compute the A_{even} and A_{odd} by horner's rule by computing both at same time.

6. **Analysis**

Let $A(x) = a_n x^n + \cdots + a_1 x + a_0$.We have to find coefficients of A(x+c)

Let $F_k(x)$ be the coefficients of function which has been computed to x^k.

$\mathrm{F}_k(x) = f_k x^k + \ldots + f_0$

$\mathrm{F}_{k+1}(x) = F_k(x) * (x + c) + a_{n-k-1}$

Let $\mathrm{F}_{k+1}(x) = f'_{k+1} x^k + \ldots + f'_0$

$f'_k = cf_k + f_{k-1}$

$f'_{k-1} = cf_{k-1} + f_{k-2}$

.

.

.

$f'_0 = cf_0 + a_{n-k-1}$

While, $f'_{k+1} = f_k$

Thus in this way coefficients of $F_{k+1}(x)$ can be computed from previous $F_k(x)$. Time for computation of F_{k+1} from F_k is k+1.

Total Time $\sum_{i=0}^{i=n} i + 1 = O(n^2)$

Algorithm

```
1: procedure COEFF(a[ ], c, n)        ▷ Finding the coefficients of A(x+c)
2:     i ← n − 1                      ▷ Because coeffecient of x^n will be a_n
3:     while i >= 0 do
4:         j ← i
5:         while j < n do
6:             a[j] ← a[j] + c * a[j + 1]        ▷ Deduction from analysis
7:             j ← j + 1
8:         i ← i − 1
9:     return a[ ]
```

7. Analysis

The question is similar to the 8th question we will use Lagrange interpolation to find the coefficients of the polynomial, but instead of resolving each of the product term separately(which takes redundant calculations) we will resolve a complete product, i.e. we will find coefficients of

$$(x - x_1)(x - x_2)\dots(x - x_n)$$

which will take O(n^2) time according to our algorithm, from this we will find the coefficient of each product by simply dividing the product with $(x - x_i)$ which will take O(n) time along with calculating the denominator.
Overall the algorithm will take O(n^2) time.
Now that we know the coefficients of the polynomial we can calculate the coefficients of the derivative in O(n) time.

Algorithm

```
1:  procedure COEFF(x[], y[], n)
2:      Array A      ▷ Stores the coefficient if the interpolated polynomial
3:      Array D                          ▷ Stores the coefficient of the derivative
4:      Array T ← zero          ▷ Array of size n+1 to store the coefficients
5:      T[n] ← 1                              ▷ Last element of T initialized to 1
6:      i ← n − 1
7:      while i >= 0 do
8:          j ← i
9:          while j < n do
10:             T[j] ← T[j] − x[i] ∗ T[j + 1]
11:             j ← j + 1
12:         i ← i − 1
13:
14:     i ← 0
15:     while i < n do
16:         array T1 ← T        ▷ A copy of the array T and thus same size
17:         d ← 1
18:         j ← n − 1
19:         while j >= 0 do
20:             T[j] ← T[j] + x[i] ∗ T[j + 1]
21:             if i != j then
22:                 d ← d ∗ (x[i] − x[j])
23:             j ← j + 1
24:
25:         j ← 1
26:         while j <= n do
27:             A[j] ← A[j] + (T1[j + 1] ∗ y[i]/d)
28:             D[j − 1] ← (j1) ∗ A[j1]
29:             j ← j + 1
30:         i ← i − 1
31:     return A[ ]D[ ]
```

8. (a) **Analysis**

According to Lagrange interpolation the unique polynomial A(x) of degree $\leq n-1$ passing through n points (x_i, y_i) is given by

$A(x) = \sum_{1\leq i\leq n}(\prod_{i\neq j, 1\leq j\leq n} \frac{(x-x_j)}{(x_i-x_j)})y_i$

to verify that, we calculate $A(x_k)$, $0\leq k \leq n-1$

$\mathrm{A}(\mathrm{x}_k) \ = \ \sum_{1\leq i\leq n}(\prod_{i\neq j, 1\leq j\leq n} \frac{(x_k-x_j)}{(x_i-x_j)})y_i$

the product $\prod_{i\neq j, 1\leq j\leq n} \frac{(x-x_j)}{(x_i-x_j)}\ y_i$ is zero for all $i \neq k$ and at $i = k$ the product simplifies to y_i since the numerator and denominator cancel out each other.

$\therefore \mathrm{A}(\mathrm{x}_k) \ = \ y_k$

which is true for all n points.

(b) **Analysis**

While interpolating the polynomial naively through Lagrange interpolation we will calculate the coefficients of the polynomial by resolving each product term which takes n^2 time and then summing them up which gives us the total complexity to be n^3.

Assuming we have are given two arrays x and y of size n, storing the n points, we create two arrays T and A of size n-1(because the degree of the polynomial is n-1).
Array T temporarily stores the coefficients of the polynomial obtained by resolving the one of the products at a time, while A will finally store the coefficients of interpolated polynomial.Before evaluating each product term every element of T except last one is initialized to zero, the last being initialized to y_i. A is initialized to zero,
We need to solve,

$A(x) = \sum_{1\leq i\leq n}(\prod_{i\neq j, 1\leq j\leq n} \frac{(x-x_j)}{(x_i-x_j)})y_i$

Let $P_k(x)$ be a product term in above equation for which i=k,we have

$P_k(x) = y_k * \frac{(x-x_1)(x-x_2)...(x_n)}{(x_k-x_1)(x_k-x_2)...(x-x_n)}$

We have a recursion for,

$P_{k,j}(x) = \frac{P_{k,j-1}*(x-x_j)}{(x_k-x_j)} \quad j \neq k$

where $P_{k,j}$ is the interpolated polynomial till the degree j-1. Therefore to interpolate $P_{k,j}(x)$ we will have to update all the j-1 terms

in the polynomial,which will take O(j) time a. Furthermore to interpolate $P_k(x)$ we will have to loop j from 1 to n taking O(n) time, Also,

$A_k(x) = A_{k-1}(x) + P_k(x)$

We can update the final array as soon as we are finished with $P_k(x)$ which will take n steps,time complexity being $O(n(j+1))$ which equals $O(n^2)$ since worst case value of j is n.
We will repeat this step for every k from 1 to n, thus bringing the total time complexity to $O(n^3)$.

Algorithm

```
1: procedure COEFF(x[ ], y[ ], , n)
2:     array A                          ▷ Array to store final coefficients
3:     k ← 0
4:     while k < n do
5:         Array T ← 0                  ▷ Array of size n-1 initialized to zero
6:         T[n − 1] ← y[k]
7:         d ← 1                        ▷ variable that stores the denominator
8:         i ← n − 1
9:         while i >= 0 do
10:            if i = k then continue
11:            if i > k then
12:                j ← i − 1
13:            if i < k then
14:                j ← i
15:            while j <= n − 2 do
16:                T[j] ← T[j] − T[j + 1] ∗ x[i]      ▷ Deduction from
    analysis
17:                j ← j + 1
18:            d ← d ∗ (x[k] − x[i])
19:            i ← i − 1
20:        i ← 0
21:        while i < n do
22:            A[i] ← A[i] + (T[i]/d)
23:            i ← i + 1
24:        k ← k + 1
25:    return A[ ]
```

(c) Analysis

To interpolate the polynomial using Lagrange interpolation in $O(n^2)$time we have to do it in two loops. We will maintain two arrays G_j and D_j, where D_j contains the product

$(x - x_1)(x - x_2) \ldots (x - x_j)$ for $1 <= j <= n$

And G_j contains the coefficient of the interpolated polynomial up to now.

As given in the recursion formula,

$$G_j(x) \ = \ (y_j \ - \ G_{j-1}(x_j))(D_{j-1}(x)/D_{j-1}(x_j)) \ + G_{j-1}(x)$$

Proof for time complexity

To calculate G_j :

We already have D_{j-1} and G_{j-1}.

$$\begin{aligned} G_{j-1} &= a_0 + a_1 * x + ... + a_{j-1} * x^{j-1} \\ D_{j-1} &= b_0 + b_1 * x + ... + b_{j-1} * x^{j-1} \\ &= (x - x_1) * (x - x_2)..... * (x - x_{j-1}) \end{aligned} \tag{1}$$

For complexity the calculation will be

$$\begin{aligned} &O(G_{j-1}(x_j)) = O(j) \text{ (horner's rule)} \\ &O(D_{j-1}(x_j)) = O(j) \text{ (horner's rule)} \\ &O((y_j - G_{j-1}(x_j))/D_{j-1}(x_j)) = O(1) \\ &D_j(x) = D_{j-1}(x) * (x - x_j) \\ &=> O(D_j(x)) = O(j) \\ &O(c * D_j(x) + G_{j-1}(x)) = O(j) \text{ (c is a constant)} \end{aligned} \tag{2}$$

$$\begin{aligned} O(G_j(x)) &= O((y_j \ - \ G_{j-1}(x_j))(D_{j-1}(x)/D_{j-1}(x_j)) \ + G_{j-1}(x)) \\ &= O(j) \end{aligned} \tag{3}$$

To calculate G_0, G_1..... G_n the time complexity will be

$$\begin{aligned} c &= \sum_{j=0}^{n} O(j) \\ &= O(n^2) \end{aligned} \tag{4}$$

We use two functions
1)The horner.calculate takes two inputs an array of coefficient of polynomial and a value at which it has to be evaluated, outputs a number which is the evaluated polynomial.
2)The append function takes two arrays one is the destination array and the other being sources array, it adds the corresponding elements from destination array to the source array.

Algorithm

```
1: procedure COEFF(x[ ], y[ ], n)
2:     Array G
3:     Array D
4:     j ← 0
5:     while j < n do
6:         D_{j-1}(x_j) = Horner.Calculate(D_{j-1}(x), x_j)      ▷ Returns the value D_{j-1}(x_j)
7:         G_{j-1}(x_j) = Horners.Calculate(G_{j-1}(x), x_j)     ▷ Returns the value G_{j-1}(x_j)
8:         append(G_{j-1}(x), (y_j − G_{j-1}(x_j))(D_{j-1}(x)/D_{j-1}(x_j)))
9:         i ← n − 1
10:        while i > 0 do
11:            D[i − 1] ← D[i − 1] + x_{j+1} ∗ D[i]
12:            i ← i + 1
13:        j ← j + 1
14:    return G[ ]
```

9. Analysis

The minimum number of calls to be made to ensure everyone knows all the information is $2n - 4$.

To make this possible we spilt the group into two of size k and k or k and k+1 depending whether $n = 2 * k$ or $n = 2 * k + 1$, i.e. whether n is odd or even.

For n being even,
Step 1 : If n is even then we pick two person one from each group, they will call every other person in their group.
The total calls made up to now $= 2 * k - 2$.
Two people in each group know all the information about their group(the person who made the calls and the person being called last)

Step 2 : Now we make two more calls one between the callers and the one between the two being called last.
Total calls up to now $= 2 * k$

Step 3 : Now we make the callers in both group to call the remaining k-2 people again in their respective groups so that everyone gets all the information of the other group.
Total calls up to now $= 4 * k - 4$.
Substituting k with n/2 we get the total calls to be $2 * n - 4$.

For n being odd,
the groups are of size k and k+1, we follow the same procedure as done for even n.
Step 1: Two persons from each group will call all other members,the calls being made will be $= k - 1 + k$, i.e. $2 * k - 1$.

Step 2: Similarly 2 two calls will be made to interchange all the information between the groups, Total calls $= 2 * k + 1$

Step 3: We make k-2 and k-1 calls in k and k+1 group respectively to distribute the information of other group, finally the total calls are $4 * k - 2$. Substituting for n we get total calls as $2 * n - 4$.

For both odd and even cases the calls being made are $2n - 4$.

10. ## Analysis

To find number of paths from a to b in a DAG, as there are no cycles if b has k in-edges it can be said that number of paths to b is equal to sum of all the paths from s to each of the nodes connected to it through inedges. So starting from b by using dfs recursively call that function with source as a and destination as the parents of b. If source is same as the destination then number of paths will be 1.

To Prove: - Any path from a having destination as b will have the last node before bas one of its in-neighbours.

Proof by contradiction:

Assume there exists a path p such that it does not pass through any of in-neighbours of b. Let p=a p_1 p_2..... p_r b. This implies there exists an in-edge from p_r to b. This means p_r is an in-neighbour to b. But this contradicts our assumption.

Hence, Proved.

Algorithm

```
vector<vector<int>> v;
vector<int> mark;
int ways(int so,int dest){
    int k=0;
    if(mark[so]!=-1)return mark[so];
    if(so==dest){
        mark[so]=1;
        return 1;
    }
    else{
        for(int i=0;i<v[so].size();i++){
            k=(k+ways(v[so][i],dest));
        }
        mark[so]=k;
        return k;
    }
}
```

11. ## Analysis

Assuming that we are given L(u) corresponding weights as a function of u in an array and adjacency list of inedges.

From this array we will make an array l which will have vertices as a function of weights i.e l(i)=vertex with weight i. Time taken for this operation is $O(V)$.

We will perform bfs but going in order followed by l i.e first we will do bfs for the elements with smallest weight and assign it to all the vertices which are reachable to the vertex l[0] in this way we will keep doing it until we have reached the end of the array. This will take $O(V + E)$

$\therefore$ Total Time $= O(V + E)$

We define component of graph as the set of vertices with same min values.Consider G_i is the i^{th} component of Graph and V_i is the set of vertices in G_i and u_i is the vertex with weight = min value or root of this bfs tree.

To Prove: - Every vertex in G_i has min equal to u_i.
Proof:

Let There exist a vertex v in V_i thus $min(v) = u_j < u_i$(by formulation of solution) Which means that $v \in V_j$. Thus min(v) = u_j but min(v)= u_i by defination thus:

$u_i = u_j$

Which Contradicts the statement $u_j < u_i$ and **Hence, Proved.**

Algorithm

```
1: procedure MIN VERTEX PATHS(L[ ], inList[ ][ ], V, E)
2:     array l                          ▷ new array to store vertices as function of weights
3:     array b ← false                  ▷ boolean array for bfs
4:     i ← 0
5:     while i < V do
6:         l[ L[ i ] − 1 ] ← i          ▷ Here we will get all (weights − 1)
7:     i ← 0
8:     while i < V do
9:         queue q ← φ
10:        Enqueue ( q, l[ i ] )
11:        while q ! = φ do
12:            v← Dequeue( q)
13:            L[v] ← l[ i ]
14:            j ← 0
15:            while j < inList[v].size do
16:                if b[ inList[ v ][ j ] ] is false then
17:                    b[ inList[ v ][ j ] ] = true
18:                    Enqueue([ inList[ v ][ j ] ])
19:                j ← j + 1
```

```
20:             while i < V&&b[ i ] do
21:                 i ← i + 1
22:         return L[ ]
23:
```

12. Analysis

Assuming we will be given the convex polygon as a set of linked list in clockwise order and the point which we will be given be p. We will fix horizontal axis and arrange all points in increasing order of increasing angle made by segement with respect to horizontal axis while, this segment is made by joining the corner point of convex polygon to p.

For a given point y on polygon and it's neighbour being x,z. The angles of point x, y, z be α, β , γ. There are six **cases** possible:

1) If for point y, its neighbours x and z have an angle less than angle of segment y-p, then there will be decrease in no. hits by 2.

2) If for point y, and its neighbours x and z if they have an angle greater than angle of segment y-p then there will be increase in no. of hits by 2.

3) If for point y, the angle of segment y-p is in between angle of x-p and z-p then there will be no change in no. of hits.

4) If for point y, the angle of segment x-p, y-p and z-p are same then it won't be a polynomial and hence this case is not possible.

To analyse for position of point :

1) If the point is inside the polygon then there is going to be atleast one hit.

2) If the point is outside the polygon there is going to atleast zero hit.

Using a user defined class **Polygon**. It provides following functionality:

1) **Polygon(vertices,point)** : Constructor for initializing the polygon class.

2) **sort()** : This will sort the points of convex polygon in the order of increasing angle.

3) **leftNeighbour()** : This will return left neighbour (in clockwise direction) of the point y.

4) **rightNeighbour()** : This will return right neighbour (in clockwise direction) of the point y.

5) **nextPoint()** : Return the next point in order of increasing angle.

6) **isNextPoint()** : To check that is there a point after the current point in order of increasing angle.

There is another user defined class **Point** with following functionality :

1) **anlge()** : Return the angle made by segment (joining that point with p) to the horizontal axis.

There is a separate user defined function defined as **change(float,float,float)** which returns the the change in number of hits for the bullet shot from a point p to point y. This change will be calculated in the way as described above (On the basis of angles of neighbours).

Algorithm

```
1:  procedure NumberOfHits(a[ ], p)          ▷ a is the list of points
2:                                           ▷ p is the bullet firing point
3:      Polygon polygon(a,p)
4:      count ← 0
5:      maxHits← 0
6:      Point max
7:      polygon.sort()
8:      while polygon.isNextPoint() do
9:          Point y ← polygon.NextPoint()
10:         angleX ← ( polygon . leftNeighbour() ) . angle()
11:         angleY ←p.angle ()
12:         angleZ ← ( polygon . rightNeighbour() ) . angle()
13:         count ← count + change(angleX,angleY,angleZ)
14:         if count > maxHits then
15:             maxHits ← count
16:             max ← y
17:     return a
18: =0
```

Note :

The above stated algorithm won't find the exact no. of hits but still the algorithm is correct. This happens because we are assuming that for the point from which we are starting has 0 hits then we will increment or decrement depending on situation described above. In this way we will find the point(vertex) with which we will have the most no. of hits because of this reason of relative no. of hits our algorithm will find the vertex correctly.

13. ## Analysis

Since the Matrix is sorted we can find the number of occurrences if zero in O(n) time, where the size of matrix is n x n.
We proceed by entering into the matrix from left upper corner and keep moving to the right in a row until we find zero or we hit one of the edges of the matrix. If we cross zero without encountering it we move to the next row without changing the column.
According to this we will only iterate through the loop n times. Hence total time complexity will be O(n).

Algorithm

```
1: procedure COUNT(A[ ][ ], , n)
2:     count ← 0
3:     i ← 0
4:     j ← 0
5:     while i < n and j < n do
6:         if A[i][j] > 0 then i ← i + 1
7:         if A[i][j] = 0 then
8:             count ← count + 1
9:             i ← i + 1
10:        j ← j + 1
11:    return count
```

14. ## Analysis

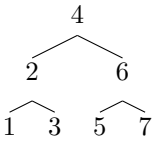
There are infinitely many possible permutations possible for a given inversion vector but there will only be a unique relative ordering between elements for a given inversion vector. So, instead we will find a permutation whose inversion vector satisfies the one given as input.

We will start to find the solution as a permutation of (1 , 2, 3, . . . , n) where n is the length of inversion vector.

We can directly use incremental design for this problem. For a given set of elements the last element in given permutation is going to be exactly determined. Assume the last element in inversion vector be x. We will be able to say that this element will be lesser than x elements in the permutation.

Let our elements to be permutted be (a_1, a_2, . . . , a_n) as our last element in inversion vector be x and so the last element in the permutation to be formed has to be the $(n - x - 1)^{th}$ smallest element or else its inversion number won't be x.In this way we will recursively find the last element of permutation until we have reached the start of inversion vector.

To implement this we will use a BST such as:



for n=7. In this way we will make bst with n nodes and $\lceil log_2\, n \rceil$ height. This can be made in O(n) time.

To find the k^{th} smallest element in bst when we are currently on node x:

$$\begin{cases} largest(k\ ,\ x{\rightarrow}\text{left}) & nodes\ in\ left\ subtree\ >\ k-1 \\ largest(k-nodes\ in\ x{\rightarrow}\text{left}\ ,\ \text{x}{\rightarrow}\text{right}) & nodes\ in\ left\ subtree\ <\ k-1 \\ x{\rightarrow}\text{value} & nodes\ in\ left\ subtree\ =\ k-1 \end{cases}$$

In this way we can get the k^{th} smallest element in O(h) time where h is the height of tree and thus in this case the maximum height that can be is O(log n). We have to this operation n-1 times and so total time = O($n \log n$).

Comment on **removal** of the found k^{th} smallest element. For this after removing that node we will backtrack the way we reached the element and decrease the number of nodes in each of the node in path by one so that we maintain proper number of nodes in one pass only.

Using a user defined class **BST** which will provide the following functionality:
1) **BST(int)** - It will create a bst tree with input n and in a way specified above.(This can be done in constructor).
2) **BST.remove(int)** - It will find the k^{th} smallest element and remove it from bst as specified and will also return this element.

Algorithm

```
1:  procedure FIND PERMUTATION(a[ ], n)      ▷ a is the inversion vector
2:                                           ▷ n is the size of inversion vector
3:      BST bst(n)
4:      i ← n − 1
5:      stack st
6:      while i >= 0 do
7:          st.push(bst.remove(i − a[i]))
8:          i ← i − 1
9:      array a
10:     a=empty(st)          ▷ empty(), empties the stack into an array and
    return it
11:     return a
12:
```

15. (a) i. For each node inserted into the heap, maximum number of comparisons will be its depth (assuming root is at depth 0). This is because in worst case it might be greater than the root and to reach the root it must make 1 comparison at each level with its immediate parent.

For a node at depth l, number of comparisons are l and at each depth value there are 2^l nodes. Let number of comparisons be c.

$$c = \sum_{l=0}^{k-1} l * 2^l \tag{5}$$

This equation is in the form of an arithmetic geometric progression

$$\begin{aligned}
c &= 1 * 2^1 + 2 * 2^2 + 3 * 2^3 ..... + (k-1) * 2^{k-1} \\
2 * c &= 1 * 2^2 + 2 * 2^3 + ... + (k-2) * 2^{k-1} + (k-1) * 2^k \\
-c &= 1 * 2 + 1 * 2^2 + 1 * 2^3 ..... + 1 * 2^{k-1} - (k-1) * 2^k \\
c &= (k-1) * 2^k - 2 * (1 + 2 + 2^2 ....2^{k-2}) \\
&= (k-1) * 2^k - 2 * (\frac{2^{k-1} - 1}{2 - 1}) \\
&= (k-1) * 2^k - 2^k + 2 \\
&= (k-2) * 2^k + 2
\end{aligned} \tag{6}$$

Here it is given that n is 2^k-1.

$$\begin{aligned}
c &= (log(n+1) - 2) * (n+1) + 2 \\
&= (n+1) * log(n+1) - 2n
\end{aligned} \tag{7}$$

ii. For each node inserted in the array which is not a child (since children are already at the bottom loop can be run from the middle to start), maximum number of comparisons will be of the order of its height assuming children are at height 0. This is because in worst case the element might be lesser than all its descendants and to reach the bottom of heap it must make 2 comparison at each level with its immediate children.

For a node at height h, number of comparisons are 2h and at each height value there are 2^{k-h-1} nodes. Let number of comparisons be c. Height of root will be k-1.

$$
\begin{aligned}
c &= \sum_{h=1}^{k-1} 2h * 2^{k-h-1} \\
&= \sum_{h=1}^{k-1} h * 2^{k-h}
\end{aligned}
\tag{8}
$$

This equation is in the form of an arithmetic geometric progression

$$
\begin{aligned}
c &= 1 * 2^{k-1} + 2 * 2^{k-2} + 3 * 2^{k-3} ..... + (k-1) * 2^1 \\
c/2 &= 1 * 2^{k-2} + 2 * 2^{k-3} + 3 * 2^{k-4} ..... + (k-2) * 2^1 + (k-1) * 2^0 \\
c - c/2 &= 1 * 2^{k-1} + 1 * 2^{k-2} + 1 * 2^{k-3} ..... + 1 * 2^1 - (k-1) * 2^0 \\
c/2 &= 2 * (1 + 2 + 2^2 ....2^{k-2}) - k + 1 \\
&= 2 * (\frac{2^{k-1} - 1}{2 - 1}) - k + 1 \\
&= 2^k - (k+1) \\
c &= 2^{k+1} - 2(k+1)
\end{aligned}
\tag{9}
$$

Here it is given that n is 2^k-1.

$$
\begin{aligned}
c &= 2 * (n+1) - 2 * (log(n+1) + 1) \\
&= 2 * (n - log(n+1))
\end{aligned}
\tag{10}
$$

(b) Heapsort is done by first building the maximum heap which can be done in the bottom-up fashion, and then retrieving the root element and exchanging with last element, and then performing heapify to the rest of the array except the last place where root was placed so that heap property is not lost.

The number comparisons for building heap will be 2*(n-log(n+1)) as calculated above. After that each time the root is removed the heapify will have number of comparisons of the order of its maximum depth assuming root is at depth 0. At each level there will be 2 comparisons with the children to decide the new parent. The number of nodes corresponding to each level will be 2^h. Let number of comparisons be c.

$$
\begin{aligned}
c &= 2 * (n - log(n+1)) + 2 * \sum_{h=0}^{k-1} h * 2^h \\
&= 2^{k+1} - 2(k+1) + 2 * \sum_{h=0}^{k-1} h * 2^h
\end{aligned}
\tag{11}
$$

This equation is in the form of an arithmetic geometric progression. Assume a variable c1.

$$c1 = \sum_{h=0}^{k-1} h * 2^h \tag{12}$$

$$\begin{aligned} c1 &= 1 * 2^1 + 2 * 2^2 + 3 * 2^3 ..... + (k-1) * 2^{k-1} \\ 2 * c1 &= 1 * 2^2 + 2 * 2^3 + ... + (k-2) * 2^{k-1} + (k-1) * 2^k \\ -c1 &= 1 * 2 + 1 * 2^2 + 1 * 2^3 ..... + 1 * 2^{k-1} - (k-1) * 2^k \\ c1 &= (k-1) * 2^k - 2 * (1 + 2 + 2^2 ....2^{k-2}) \\ &= (k-1) * 2^k - 2 * (\frac{2^{k-1} - 1}{2-1}) \\ &= (k-1) * 2^k - 2^k + 2 \\ &= (k-2) * 2^k + 2 \end{aligned} \tag{13}$$

$$\begin{aligned} c &= 2^{k+1} - 2(k+1) + (k-2) * 2^k + 2 \\ &= k * 2^k - 2k \end{aligned} \tag{14}$$

Here it is given that n is 2^k-1.

$$\begin{aligned} c &= 2 * (n - log(n+1)) + 2 * ((n+1)log(n+1) - 2n) \\ &= 2nlog(n+1) - 2n \end{aligned} \tag{15}$$

But in reality, the number of comparisons for a h depth tree need not be 2h because after removing some elements from that heap, the heap may not be complete. At the last level if there is only 1 child or no children for that particular branch then even though maximum depth is h, there no comparisons will be less than 2h. Even in the worst possible scenario there will be atleast 2 nodes with less than 2 comparisons at the last level. Considering this actual value of c1 should be 3 less than that calculated because in the worst case there will be 1 node with only 1 comparisons and another with none at the last level.

$$c1 = \sum_{h=0}^{k-1} h * 2^h - 3 \tag{16}$$

So it is not possible to build an example with number of comparisons equal to that of its complexity.

16. (a) **Analysis**

To find the transitive closure of a graph, for every 2 nodes u, v such that there is a path from u to v there must exist an edge from u to v. A method to find the transitive closure is to first get a matrix which gives all the vertices which are reachable through some path. This can be implemented for each node using DFS in O(V*(v+E)) time complexity. Using this matrix make edges for all the nodes which are reachable but are not connected. This can be achieved in $O(V^2)$ time complexity. The total time complexity will be of the order $O(V^2+V*E)$.

Algorithm

```
vector<vector<bool>> reachable[n][n];
vector<vector<int>> adjlist[n];
void makeneighbourreachable(int root,int source){
    for(int i=0;i<adjlist[root].size;i++){
        reachable[source][adjlist[root][i]]=true;
        makeneighbourreachable(adjlist[root][i],source);
    }
}
void findreachable(){
    for(int i=0;i<n;i++){
        makeneighbourreachable(i,i);
    }
}

void func(){
    for(int i=0;i<n;i++){
        findreachable();
        for(int j=0;j<n;j++){
            if(reachable[i][j]==true)
                adjlist[i].push_back(j);
        }
    }
}
```

(b) The contrapositive of the proved statement is : If there is unique transitive reduction $\rightarrow$ the graph is DAG.

To prove : If no. of transitive reductions isn't unique$\rightarrow$ the graph is not DAG.

Proof : As graph is acyclic maximum length of path can only be n-1. Define a set E_H consisting of all the edges of all possible maximal paths between two nodes u,v for all u,v $\in$ V. For an edge xy $\in E_H$ maximum path length from x to y can only be 1 otherwise it will be a contradiction as if there was a longer path between x and y then xy cannot belong to any maximal path. So E_H can be redefined as an edge xy $\in E_H$ if length of longest path from x to y is 1.
Let F be an arbitrary subset of E such that $F^* = E^*$ where H^* is defined as the closure of H. For e E_H if e does not belong to F then consequently it cant belong to F^* as the longest possible path is e itself but this is contradictyion to $F^* = E^*$. So E_H is subset of F. As E_H contains edges of maximal paths of all pairs of u,v $\in$ V ${E_H}^* = E^*$. This means that E_H is a subset of every subset of E whose closure is E^* and its closure is also E^*. Thus this is the unique minimal set which is the transitive reduction of the graph.

To prove : if the graph is not DAG $\rightarrow$ the no. of transitive reductions isn't unique

Proof : We will find the strongly connected component's of the directed graph G we will name them as SC_1, , SC_2 , SC_3 , . . . , SC_k. After finding these strongly connected components we will make a new graph with vertices as these strongly connected components. There will be an edge in between SC_i and SC_j if there is an edge from the vertices belonging to one component to another component. For every edge for the vertex to another vertex if both are in same component then we won't add the edge to it or else we will add that component to the adjacency list of new graph of the current vertex to be considered. After doing this there can be multiple edges between two components and thus we will reduce this to single edge. In this way we will construct a new graph. This graph will be a DAG. As we have already proved that DAG have a unique transitive reduction. For a given strongly connected component we can do a transitive reduction of it into a cycle. Now we will expand the transitive reduction of DAG such that each strongly connected component will be represented as the cycle. If there is an edge between two components then we will have multiple choices of vertices (i.e we will one of component with no. of vertices greater than one and so we can choose any one of them for connecting edges) between two components from the two components and then add the edge between them

and so we will have multiple transitive reductions for a graph that is not a DAG.

(c) **Analysis :**

We will find the strongly connected component's of the directed graph G by using Kosaraju's algorithm . This will take O(V+E) time. We will name them as SC_1, , SC_2 , SC_3 , . . . , SC_k. After finding these strongly connected components we will make a new graph with vertices as these strongly connected components. We will be make an edge between SC_i and SC_j if there is an edge from the vertices belonging to one component to another component. In this there will be multiple edges between components. To, remove this we will sort each edge list of vertices and then remove the repeating edges. This operation will $O(VlogE)$. In this way we will obtain a DAG.

Let us first find the transitive reduction of a directed acyclic graph. Using the 'a' part, we can find the transitive closure of the graph G. Let $M[i][j]$ be the adjacency matrix of the transitive closure of graph G, M[i][j]=1 if there exists a path i to j and A[i][j]=0 otherwise. We did this because transitive reduction of graph and transitive reduction of transitive closure of the graphs are same.

In matrix A if we find M[i][j]=1, M[j][k]=1, M[i][k]=1,then we can set M[i][k] to zero since an edge between i and k is not required as reach-ability between i and k is still maintained. Notice that since G is DAG and M[k][j]=0 otherwise G would have a cycle and path between $i \to k$ cannot be contained in path $i \to j$.
So, we check all possible V^3 combinations eliminating redundant edges.

We check for each vertex i, all vertices j which have a path to it , then for each vertex k to which there is a pth from i, we set M[j][k]=0.

Let the user defined class be **SCC** : This class provides following functionality.

1)**DAG** : This function converts the graph into DAG of strongly connected components and returns the adjacency matrix of it.

2)**Reduction** : This will do the transitive reduction of each strongly connected component to cycle.

3)**Reconvert**: For the given DAG of strongly connected component it will expand the strongly connected components graph to the normal graph and will return's its adjacency matrix.

Let the user define class be **Graph**. This class provides us with following functionality:

1)**closure** : This method takes the adjacency matrix as input and returns the adjacency matrix of graph of transitive closure of the graph.

Algorithm

```
1: procedure TransitiveReduction(X[ ][ ] , n)
2:                                  ▷ X is the adjacency matrix given in input
3:     Graph g
4:     SCC scc
5:     A← scc.DAG(X)
6:     scc.reduction()
7:     M ← g.closure(A)
8:     i ← 0
9:     while i < n do
10:        j ← 0
11:        while j < n do
12:            k ← 0
13:            while k < n do
14:                if M[ i ] [ k ] == 0 then
15:                    M[ j ] [ k ] ← 0
16:    M ← scc.reconvert(M)
17:    return M
```

17. Analysis

Let our original graph be H. Assuming that we will be given the input as adjacency list of DAG and two vertices s and t. We have to find all the min cut vertices that will make s and t disconnected.

All vertices that does not lie in any of the path between s and t are useless because their presence or absence won't contribute towards connectivity of s and t. We will do a bfs with taking s as root and remove vertices that doesn't lie in the tree and their corresponding edges also. This operation will take O(V+E) time.

Similarly we will do bfs for inedges with taking t as root and remove all the vertices not lying on its bfs tree. This operation will all the vertices that aren't reachable to t. The time for this operation is O(V+E).

Now we will have a graph which will originate from s and will reach t. That is, it will be a subgraph $G = (V, E)$ with all paths originating from s and ending at t.

Now we will do topological search and will assign the index numbers to the vertices. This operation will take O(V+E) time.

For a vertex v let $G - v$ have two components G_1 and G_2 such that G_1 have vertices with topological number less than that of v and G_2 will have vertices with topological number greater than v.

Lemma : If there is an edge G_1 to G_2 in G s and t are connected otherwise they are disconnected.

Proof : Let there be no edge from G_1 to G_2 in G and still $G - v$ have a path from s to t i.e. v is not the cut vertex in this case. This means there exist a path $p = p_1 \to p_2 \to p_3 \ldots . \to p_k$. But this path won't pass through v and thus this is going to be contradiction to statement that G_1 and G_2 doesn't have a path. Hence Similarly, we will prove converse that if G_1 and G_2 has a path in $G - v$ then v is not cut vertex.

Hence proved : G_1 and G_2 have a path in $G - v \iff$ v is cut vertex.

To accomplish this we will start checking in G from vertex s. For vertex s we will store the vertex that is adjacent to s and with highest topological number. In this way we will keep doing after a vertex is traversed. For a vertex v we will have the the vertex with highest topological number be 'y' that is adjacent from G_1. If this number is more than topological number of v then it means that there is an edge from G_1 and G_2 in $G - v$. Therefore v will not be the cut vertex otherwise if that topological number is less than v (" actually it will be equal to topological number of v") then v will be the cut vertex. Now after determining that if v is cut vertex or not, we will find the vertex with maximum topological number adjacent to v. If this is greater than out current maximum

we will update or else we won't update it.

Time Complexity analysis for this step :

$\sum_{q \in E_v} O(1)$ where E_v is set of edges of v.
$\sum v \in V \sum_{q \in E_v} = O(V + E)$.

The user defined class be **GraphOperations** with following functionality :

1)**GraphOperations() :** Constructor the adjacency list, set of vertices with vertex

2)**subgraph()** : This method vertex s and vertex t as input. This method will remove the vertices that are not reachable from s and vertices that doesn't reach v.

3)**sort** : This method doesn't require any input and does the topological sort on the subgraph and allots the topological number to the vertices.

4)**max** : This function takes vertex v as input and returns the vertex with maximum topological number.

5)**nextVertex()** : This function returns the next vertex in the topological sort performed. While the first vertex being pointed being s.

6)**topologicalNumber()** : returns the topological number of current vertex being considered.

Algorithm

```
1:  procedure MinCut(adjList[ ][ ], V[ ], s, t)        ▷ to find the min cut
2:      GraphOperations graph(adjList,V)
3:      grpah.subgraph()
4:      graph.sort()
5:      max ← graph.max(s)
6:      list l
7:      v ← graph.nextVertex()
8:      topnum ← graph.topologicalNumber()
9:      while v ! = t do
10:         if topNum > max then
11:             l.push(v)
12:         if graph.max(v) ¿ max then
13:             max=graph.max(v)
14:         v← graph.nextVertex()
15:         topnum← graph.topologicalNumber()
16:     return l
```

Refrences :

Question 4 : Website for refrence of optimal answer.

Question 16 b) : Website to prove one side of the statemant.

Question 16 c) : Website for kosaraju's algorithm and for optimal solution for DAG.