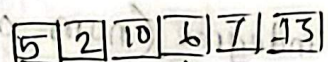
(*) Linear Search:

n=6



- If found → print location
- If not found → data not found.

```
int a[] = {-------}
int found = 0
for (int i=0; i<n; i++){
    if (a[i] == s_data){
        printf ("data found %d ", i); found++;
        break;
    }
}
if (found == 0){
    printf ("data not found!");
}
```

Complexity:
Best: O(1)
Worst: O(n)
Average: $\frac{\Sigma \text{All case}}{\text{no of case}}$

$= \frac{n(n+1)/2}{n}$

$= n+1/2$

$O((n+1)/2)$

Binary Search: Data must be sorted.

(i) data == A[mid] → return mid
(ii) data > A[mid] → l = mid+1
(iii) data < A[mid] → r = mid-1

$mid = l+r/2$

```
int binary_search (int A[], int n, int data){
    int l, r, mid;
    l = 0;
    while (l<r){ r = n-1; }
        mid = (l+r)/2;
    If (data == A[mid])
        return mid;
    else if (data > A[mid])
        l = mid+1;
    else
        r = mid-1;
    }
    return -1;
}
```

Time complexity:

Best case: O(1).

Worst case: $n, n/2^1, n/2^2, n/2^3 \dots n/2^K$

$n/2^K = 1$

$\Rightarrow n = 2^K$

$\Rightarrow \log n = \log_2 K$

$\Rightarrow \log n = K \log_2 [\log_2 n = K \log_2 2]$

$\Rightarrow K = \log n.$

$O(\log n).$

Average case: $O(\log n)$.

Topological Sort:

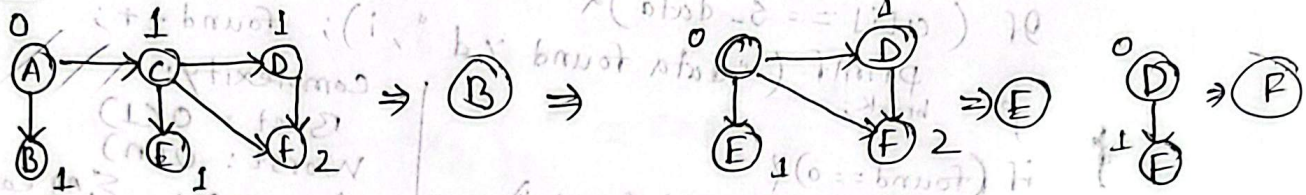
- graph directed হবে।
- graph এ কোনো cycle থাকবে না।
- topological sort of a directed graph is a linear Ordering of its vertices such that every directed edge uv from vertices u to v, where u comes before v.

Method: ① সবচেয়ে Vertices এর indegree (num of incoming edge) বের করতে হবে।

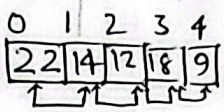
② যে vertics print করতে হবে আগে, indegree = 0.

print করার পর তাকে remove করতে হবে যত গুলো edge আছে

Print: A B C E D F



(*) Bubble Sort:



n = 5

it-1:

22 14 12 18 9

14 22 12 18 9

14 12 22 18 9

14 12 18 22 9

14 12 18 9 22

it-2:

14 12 18 9 22

12 14 18 9 22

12 14 18 9 22

12 14 9 18 22

it-3:

12 14 9 18 22

12 14 9 18 22

12 9 14 18 22

it-4:

12 9 14 18 22

9 12 14 18 22

Sorted

```
Void bubble_sort (int A[], int n){
    for (int i=0; i<n-1; i++){
        flag=0;
        for (int j=0; j<n-1-i; j++){
            if (A[j] > A[j+1]){
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
                flag=1;
            }
        }
        if (!flag) break;
    }
```

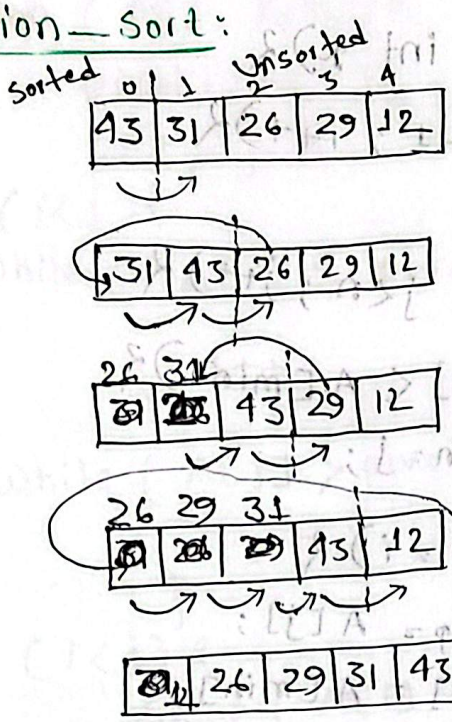
time complexity:

Best: O(n)

Worst: $O(n^2)$

Average: $O(n^2)$

⑧ Insertion_Sort:



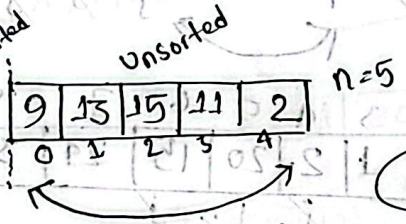
time complexity:

$j = 0 \longrightarrow 1$
$j = 1 \longrightarrow 2$
$j = n-1 \longrightarrow n$
$1+2+3+4+\cdots+n$
$\Rightarrow \frac{n(n+1)}{2} = \frac{n^2+n}{2}$
$\therefore O(n^2)$ upper bound

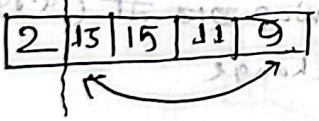
- Best case : $O(n)$
- Worst case : $O(n^2)$
- Average case : $O(n^2)$

```
Void insertion_sort (int A[], int n){
    for (i=1; i<n; i++){
        temp = A[i];
        j = i-1;
        while (j>=0 && A[j] > temp){
            A[j+1] = A[j];
            j--;
        }
        A[j+1] = temp;
    }
```

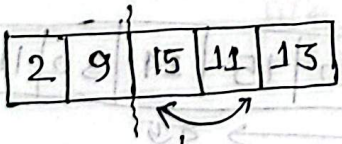
⑨ Selection_Sort:



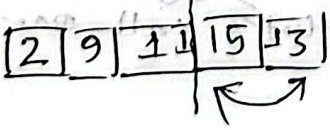
iteration 1:



iteration 2:



iteration 3:



iteration 4:

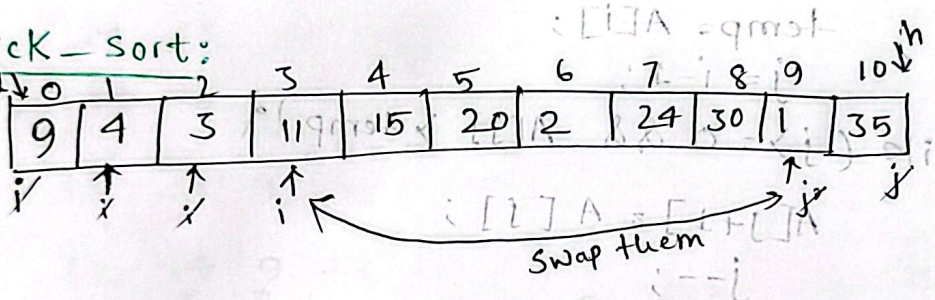
2 9 11 13 15
Sorted.

```
void selection_sort (int A[], int n){
    for (i = 0; i < n-1; i++){
        min = i;
        for (j = i+1; j < n; j++){
            if (A[j] < A[min]){
                min = j;
            }
        }
        if (min != i){
            temp = A[j];
            A[j] = A[min];
            A[min] = temp;
        }
    }
}
```

time complexity:

Best: $O(n^2)$
Worst: $O(n^2)$
Average: $O(n^2)$
Space: $O(1)$

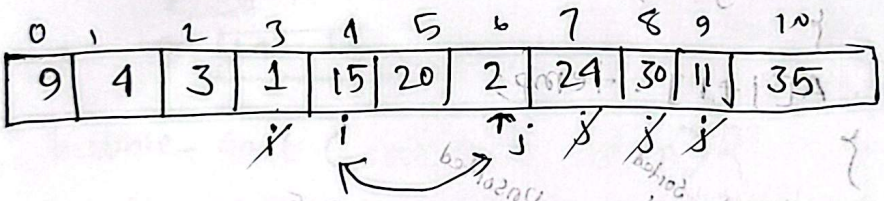
⊛ Quick_sort:



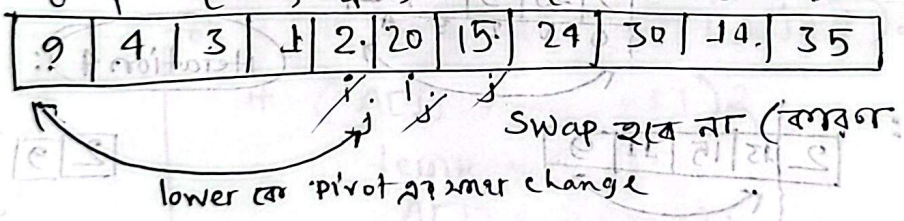
pivot = 9

A[i] ≤ pivot
i++

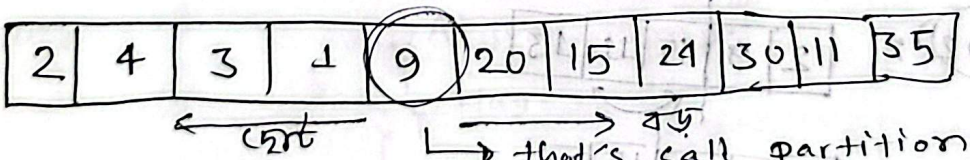
A[j] > pivot
j--



if (i<j)
swap(a[i], a[j])
else
swap(pivot, a[j])



Swap হবে না (কারণ i > j হয়ে গেছে)

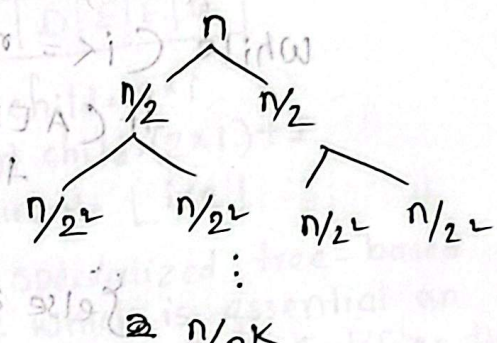


↳ that's call partition

```
partition (A[], l, h){
    pivot = A[l];
    i = l, j = h;
    While (i<j){
        while (A[i] <= pivot){
            i++;
        }
        while (A[j] > pivot){
            j--;
        }
        if (i<j){
            Swap (A[i], A[j]);
        }
    }
    swap (A[l], A[j]);
    return j;
}

Quicksort (A, l, h){
    if (l<h){
        j = partition (A, l, h);
        Quicksort (A, l, j-1);
        Quicksort (A, j+1, h);
    }
}
```

time complexity:



$\Rightarrow n = 2^K$

$\Rightarrow \log n = \log 2^K$

$\Rightarrow \log n = K \log 2$

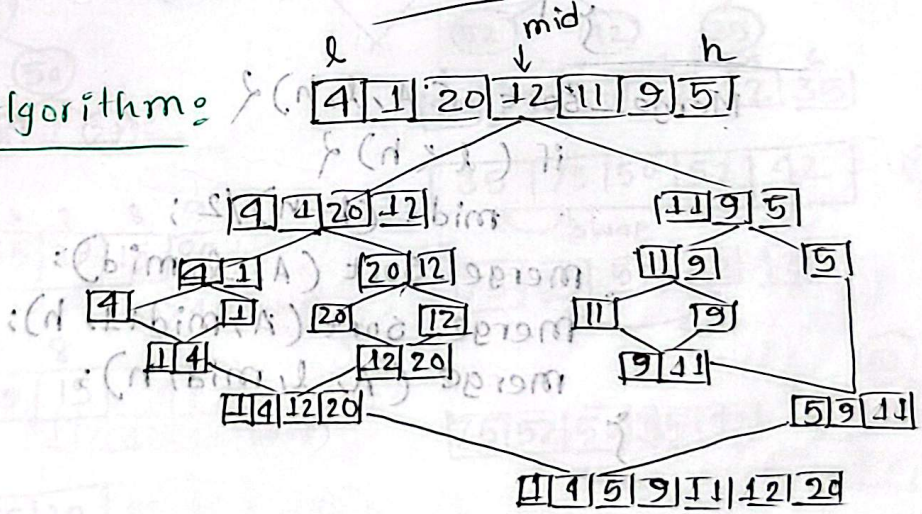
$\Rightarrow K = \log n$

→ Best case: $O(n \log n)$

→ Worst case: $O(n^2)$

→ Average case: $O(n \log n)$.

(7) Merge_Sort Algorithm:



```
merge (A, l, mid, h) {
    i = l;
    j = mid + 1;
    k = l;
    while (i <= mid && j <= h) {
        if (A[i] <= A[j]) {
            temp[k] = A[i];
            i++;
            k++;
        } else {
            temp[k] = A[j];
            j++;
            k++;
        }
    }
    if (i > mid) {
        while (j <= h) {
            temp[k] = A[j];
            j++;
            k++;
        }
    } else {
        while (i <= mid) {
            temp[k] = A[i];
            i++;
            k++;
        }
    }
    for (k = l; k <= h; k++) {
        A[k] = temp[k];
    }
}
```

time complexity:

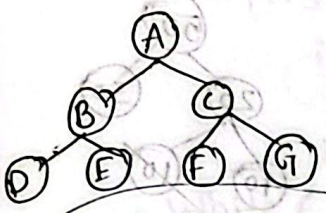
Best, Worst, Average;

→ O(nlogn)

```
Merge_sort (A, l, h) {
    if (l < h) {
        mid = (l+h)/2;
        merge_sort (A, l, mid);
        merge_sort (A, mid+1, h);
        merge (A, l, mid, h);
    }
}
```

Heap:

Array representation of Binary tree:



0	1	2	3	4	5	6
A	B	C	D	E	F	G

index → i

left child → (2*i) + 1

Right child → (2*i) + 2

parent → ⌊(i−1)/2⌋

If index started from zero (0) index

if index started from one(1) index:

1	2	3	4	5	6	7
A	B	C	D	E	F	G

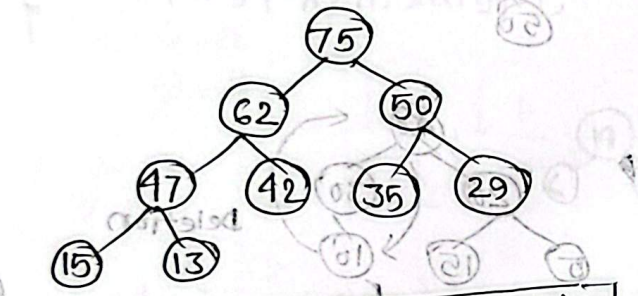
left child = 2*i

right child = (2*i)+1

parent = ⌊i/2⌋

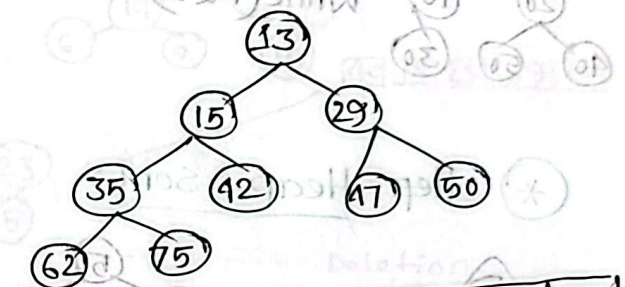
⇒ Heap is a specialized tree-based data structure which is essential an almost complete tree that satifies the heap property.

Max heap (parent ≥ child)



75	62	50	47	42	35	29	15	13

Min heap (parent ≤ child)



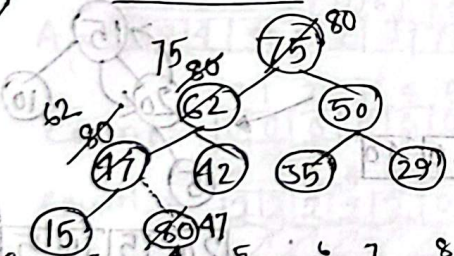
13	15	29	35	42	47	50	62	75

Max heap Insertion & deletion:

best: O(1)
worst: O(logn)

O(logn)

Insertion (80)



1	2	3	4	5	6	7	8	9
75	62	50	47	42	35	29	15	80

80 > 47 (swap)

1	2	3	4	5	6	7	8	9
75	62	50	80	42	35	29	15	47

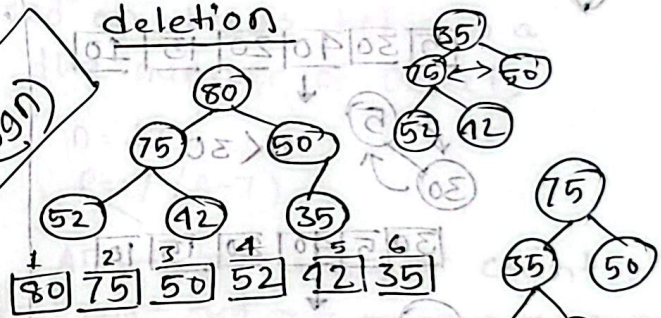
62 < 80 (swap)

75	80	50	62	42	35	29	15	47

80 > 75 (swap)

80	75	50	62	42	35	29	15	47

deletion



1	2	3	4	5	6
80	75	50	52	42	35

35	75	50	52	42

swap

75	35	50	52	42

swap

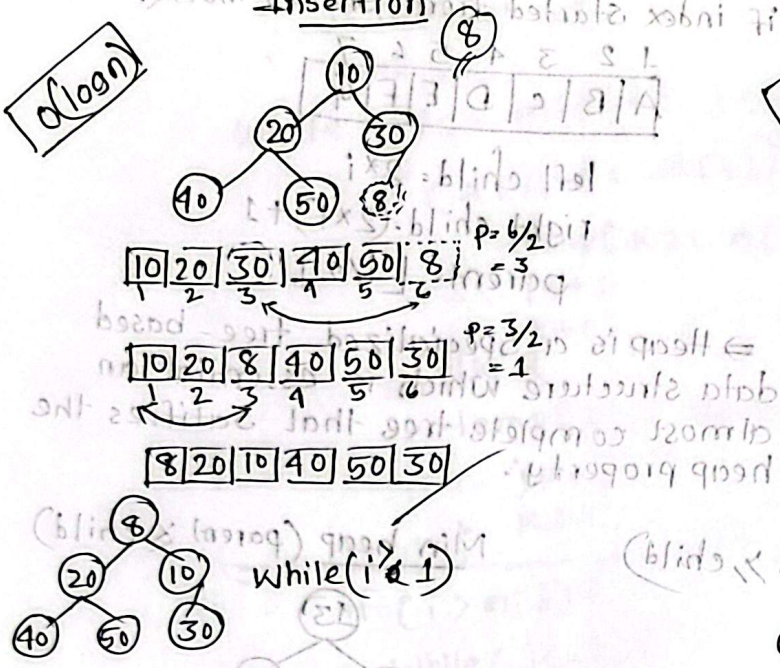
75	52	50	35	42

Sorted

⊛ Min Heap Insertion Deletion:

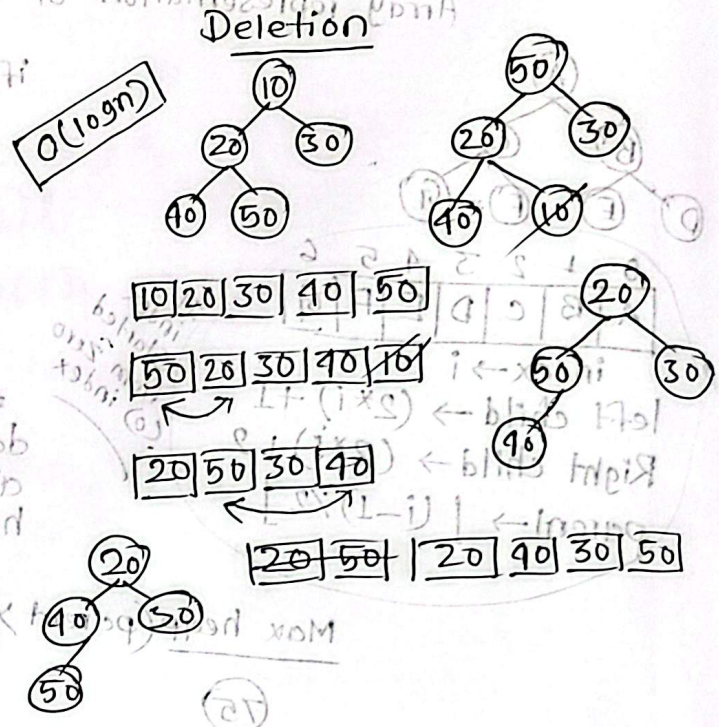
Insertion

O(logn)



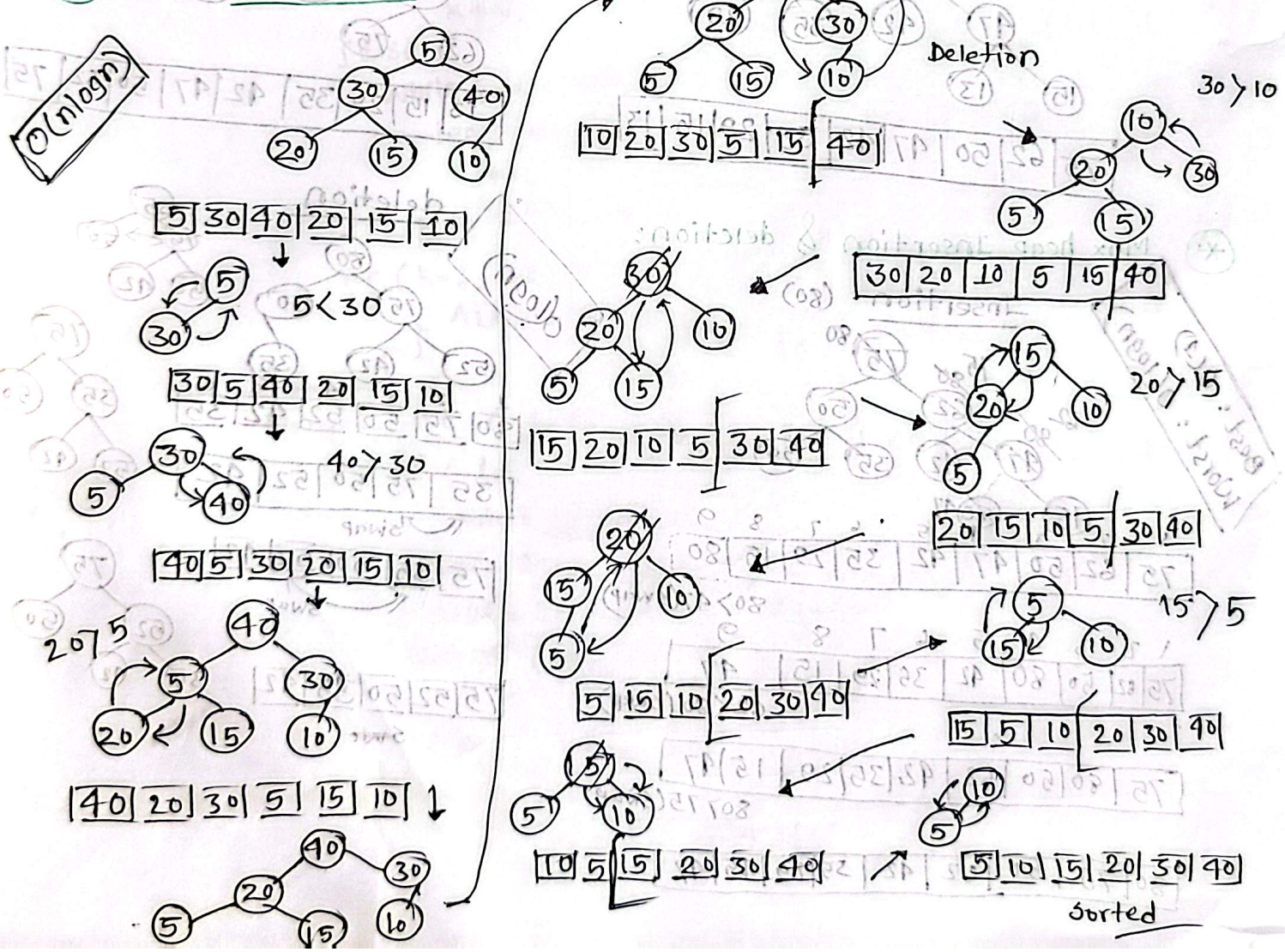
Deletion

O(logn)



⊛ Hep Heap sort:

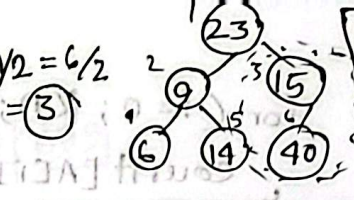
O(nlogn)



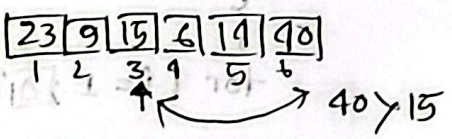
✳ Heapify method:

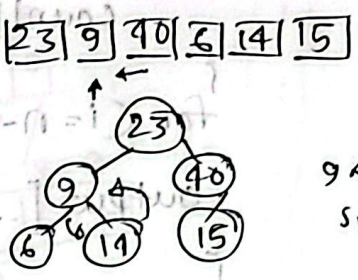
```
Heapify (A, n, i){
    int large = i;
    int l = 2*i;
    int r = (2*i)+1;
While (l<=n && A[l] > A[large]){
    large = l;
}
While (r<=n && A[r] > A[large]){
    large = r;
}
if (large != i){
  swap (A[large], A[i]);
  heapify (A, n, large);
}
```

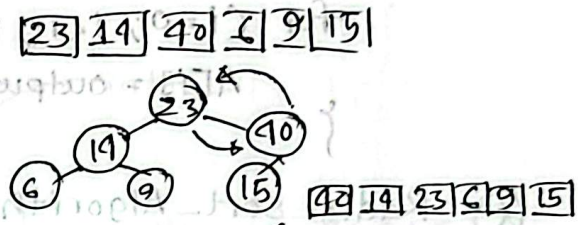
n/2 = 6/2 = 3

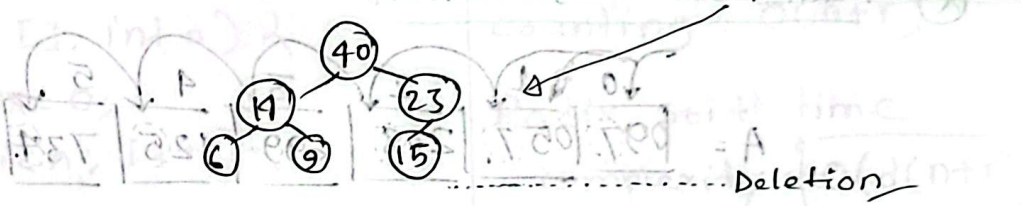


leaf node নিয়ে কাজ করে না।









✳ Counting-Sort Algorithm:

counting sort is a sorting technique based on keeys between a specific range. It works by counting the number of objects having distinct key values.

i = 0 1 2 3 4 5 6 7 8 9 10 11

A = 1 2 4 3 0 2 1 7 1 4 3 0

n = 12

R = 7 (0-7)

count: 0 0 0 0 0 0 0 0

count = 2 2 2 2 2 0 0 1

count = 2 5 7 9 11 11 11 12

A[i] → এর value যতো count এর সেই সব বারগুলোর থেকে 1 করে decrement করতে হবে।

initial এ আছে সেই বারগুলো '0' index এ, তারপর এগুলো count এর '1' index এর মানে সেই count এর '0' index এগুলো যোগ করতে থাকবে।

Output: 0 0 1 1 1 2 2 3 3 4 4 7

main Array এর right to left এ যাবো- index এর মান যেটা last count এর সেই index এ বসিয়ে 1 decrement করবো।

1 decrement করার পর যে value হবে output Array এর সেই সব index এ বসিয়ে main Array এর value বসাতে হবে।

```
for (i=0; i<n; i++){
    count[A[i]]++;
}
for (i=1; i<=R; i++){
    count[i] += count[i-1];
}
for (i=n-1; i>=0; i--){
    output[... count[A[i]] = A[i];
}
for (i=0; i<n; i++){
    A[i] = output[i];
}
```

Time complexity:

$O(n+R)$

Radix_sort_Algorithm:

n = 6
R = 9

A = | 097 | 057 | 208 | 699 | 125 | 734 |

(for last digit)

count = | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

count = | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 2 | 1 | 1 |

count = | 0 | 0 | 0 | 0 | 1 | 2 | 2 | 4 | 5 | 6 |

output: | 734 | 125 | 097 | 057 | 208 | 699 |

Now, for 2nd last digit:

A: | 734 | 125 | 097 | 057 | 208 | 699 |

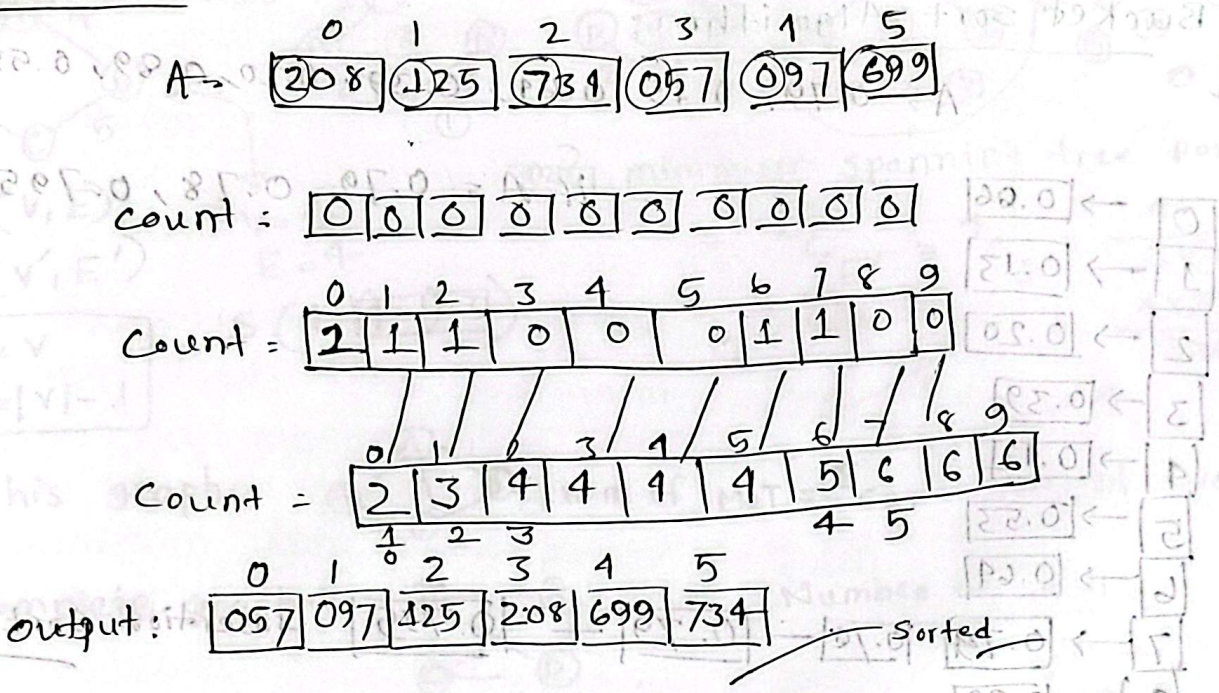
count = | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

count = | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 2 |

count = | 1 | 1 | 2 | 3 | 3 | 4 | 4 | 4 | 4 | 6 |

output: | 208 | 125 | 734 | 057 | 697 | 699 |

for fiisl element:




```
int getmax (int A[], int n) {
    int i, max = 0;
    for (i=0; i<n; i++) {
        if (A[i] > max) {
            max = A[i];
        }
    }
    return max;
}
```

```
void radixsort (int A[], int n) {
    int m = getmax (A, n);
    for (pos=1; m/pos > 0; pos*=10)
        countsort (A, n, pos);
}
```

```
void countsort (int A[], int n, int pos) {
    int output[n];
    int count[10] = {0};
    int i;
    for (i=0; i<n; i++)
        count[(A[i]/pos)%10]++;
    for (i=0; i<r; i++)
        cout[i] = Count[i] + Count[i-1];
}
```

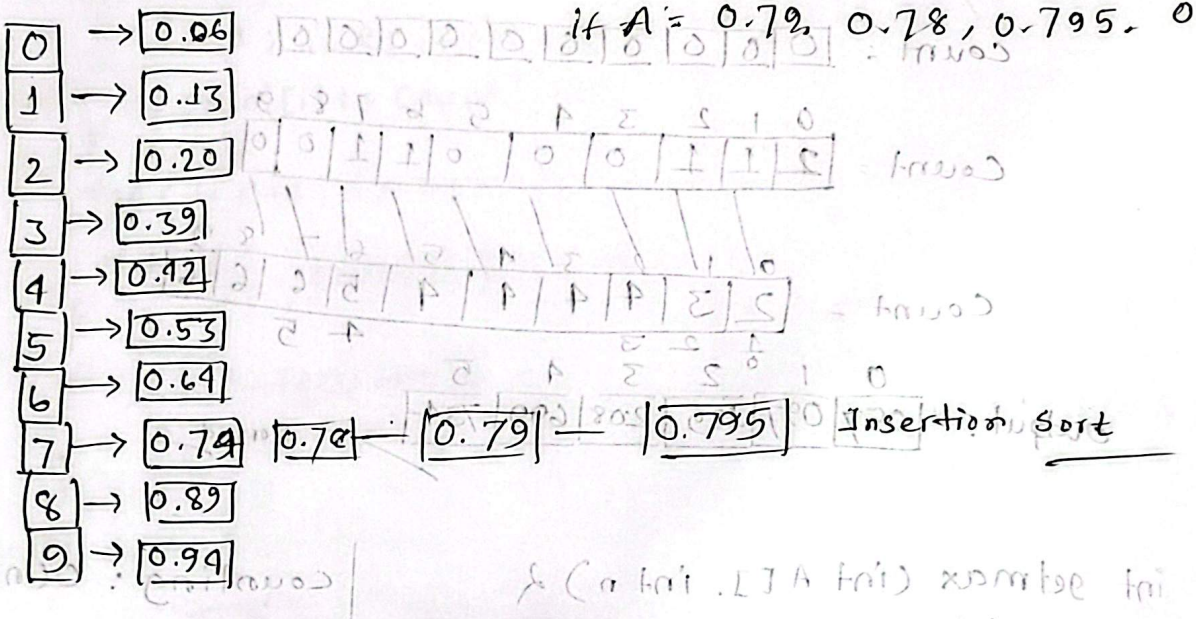
Counting : O(n+r)

Radix sort time complexity : O(d(n+r))

✳ Bucket sort Algorithm:

best: O(n)

A = 0.79, 0.13, 0.64, 0.39, 0.20, 0.89, 0.53, 0.12, 0.06, 0.94

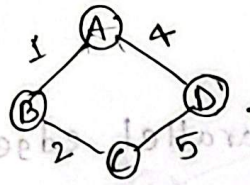


if A = 0.79, 0.78, 0.795, 0.74

```
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 together
      B[0], B[1] ... B[n-1] (in order).
```

✳ Minimum spanning tree:

- subset of a graph
- Cycle থাকবে না
- Vertices disconnected থাকবে না,
- ~~one~~ নতুন কোনো edge add/delete করলেও না,

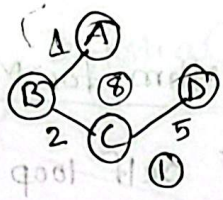


$G(V, E)$

$S(V', E')$

$V' = V$

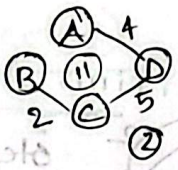
$E' = |V| - 1$

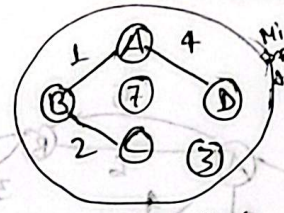


$V = 4$

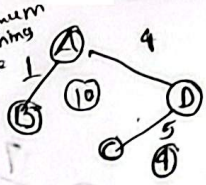
$E = 4$

$S(4, 4-1=3)$





Minimum Spanning Tree

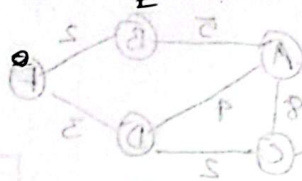


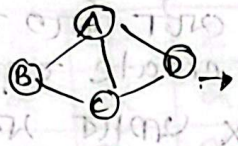
মোট minimum spanning tree possible

$= {}^{E}C_{E'} = {}^{4}C_{3} = \frac{4!}{3!(4-3)!}$

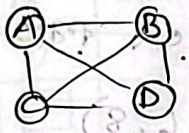
$= \frac{4 \times 3 \times 2 \times 1}{3 \times 2 \times 1}$

$= 4$



* for this graph: 

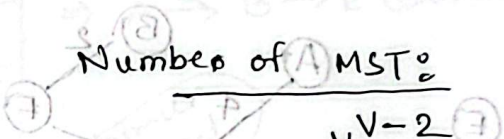
→ num of MST = ${}^{E}C_{E'}$ − num of cycle

* for complete graph: 

remove edge: $E - V + 1$

$= 6 - 4 + 1$

$= 3$ (যত edge remove করতে পারবো maximum)



Number of MST:

V^{V-2}

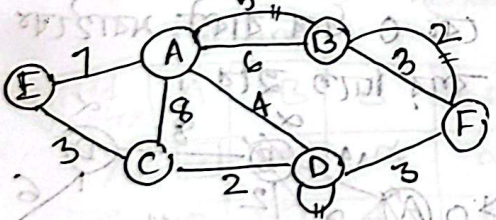
$\Rightarrow 4^{4-2}$

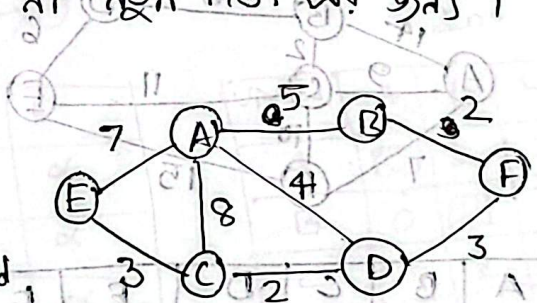
$= 4^2$

$= 16$

Kruskal's Algorithm for MST

- graph থেকে loop delete করতে হবে।
- graph থেকে parallel edge delete করতে হবে।
- cycle হওয়া যাবে না নতুন MST এর জন্য।





Step 1: loop delete করতে হবে And, parallel edge delete করতে হবে।

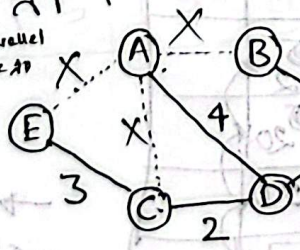
Step 2: ছোট থেকে বড় ক্রমে edge নিতে হবে। cycle হলে আর নেওয়া যাবে না।

$G(V, E) = (6, 8)$ → loop ও parallel remove এর পর

$S(V', E') = (6, 5)$

$E' = |V| - 1$

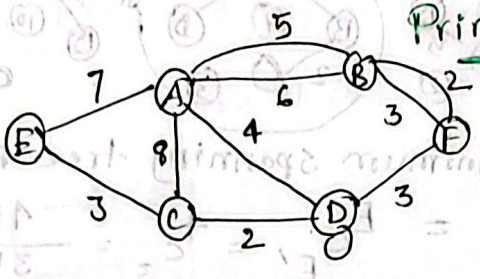
→ Correct



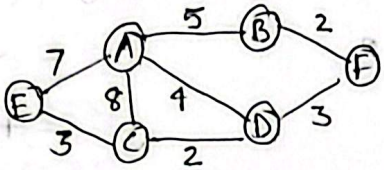
BF → 2
CD → 2
DF → 3
CE → 3
AD → 4
AB → 5 ✗
AE → 7 ✗
AC → 8 ✗

এখন দেখতে হবে এটা MST এর সর্ব পুরণ করে কিনা?

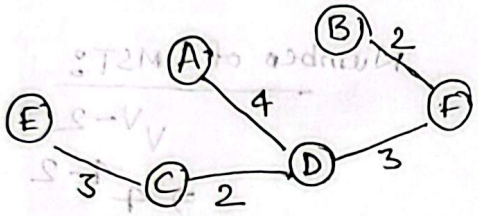
Prims Algorithm for MST



Step 1: self loop ও parallel edge remove করতে হবে।



Step 2: প্রত্যেকটা vertex এর outgoing/incoming edge check করে যেটা সবচেয়ে ছোট সেটা নিতে হবে এবং কোনভাবেই cycle create করা যাবে না। (যতগুলো vertex থাকবে সবার minimum edge check করতে সর্বনিম্ন টা নিতে হবে)



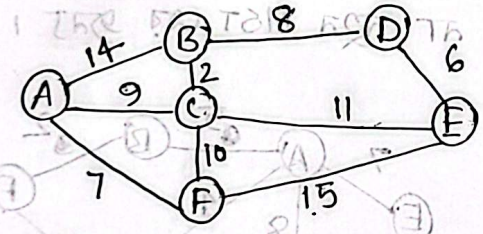
G1 (6, 8)
S (6, 5) ✓

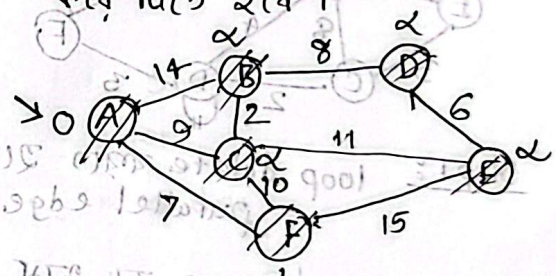
Dijkstra (Undirected)

If $(d(u) + c(u,v) < d(v))$

$d(v) = d(u) + c(u,v)$

→ initially যেকোনো (যার থেকে traverse শুরু করবে) vertex কে 0 ধরে বাকী সবাইকে ∞ করে দিতে হবে।





Visited Vertex	A	B	C	D	E	F
A	[0]	∞	∞	∞	∞	∞
F		14	9	∞	∞	[7]
C		14	[9]	∞	22	
B		[11]		∞	20	
D				[19]	20	
E					[20]	

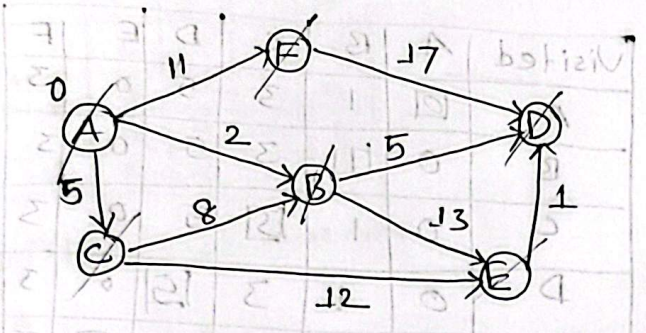
E, C, A

A, C, E

→ condition check করতে থাকবো— যার value সবথেকে ছোট থাকবে সে visited হবে এবং সেখান থেকে আবার traverse করতে হবে।

→ backtracking করে Shortest path বের করতে হবে।

Dijkstra (directed)



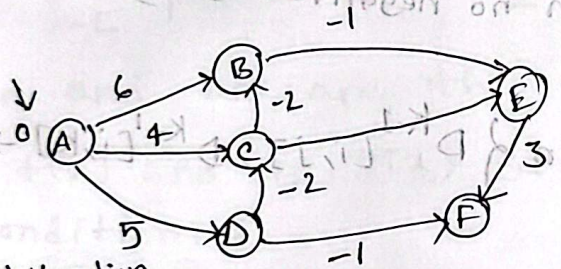
Visited Vertics	A	B	C	D	E	F
A	[0]	α	α	α	α	α
B		[2]	5	α	α	11
C			[5]	7	15	11
D				[7]	15	11
F					15	[11]
E					[15]	

minimum select

E, B, A

⇒ A → B → E (shortest path)

Bellman-ford Algorithm



Iteration (n-1) times
= (6-1)
= 5 times

1st iteration

Visited Vertex	A	B	C	D	E	F
A	[0]	α	α	α	α	α
B	0	[6]	4	5	α	α
C	0	6	[4]	5	5	α
D	0	2	4	[5]	5	α
E	0	2	3	5	[5]	4
F	0	2	3	5	5	[4]

2nd

Visited	A	B	C	D	E	F
A	[0]	2	3	5	5	4
B	0	[2]	3	5	5	4
C	0	2	[3]	5	1	4
D	0	1	3	[5]	1	4
E	0	1	3	5	[1]	4
F	0	1	3	5	1	[4]

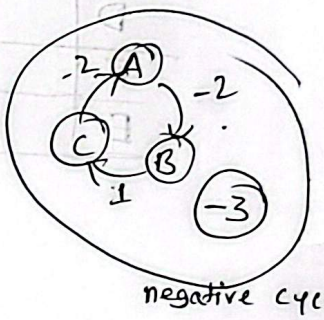
3rd

Visited	A	B	C	D	E	F
A	[0]	1	3	5	1	4
B	0	[1]	3	5	1	4
C	0	1	[3]	5	0	4
D	0	1	3	[5]	0	4
E	0	1	3	5	[0]	4
F	0	1	3	5	0	[3]

4th

Visited	A	B	C	D	E	F
A	[0]	1	3	5	0	3
B	0	[1]	3	5	0	3
C	0	1	[3]	5	0	3
D	0	1	3	[5]	0	3
E	0	1	3	5	[0]	3
F	0	1	3	5	0	[3]

A → 0 D → 5
B → 1 E → 0
C → 3 F → 3



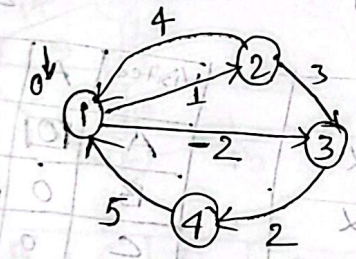
negative cycle

Floyd Warshall Algorithm

(1) Works with positive and negative edges (but with no negative cycle)

Formula:

$$D^K[i,j] = \min\{D^{K-1}[i,j],\ D^{K-1}[i,K] + D^{K-1}[K,j]\}$$



D^0 Distance matrix

	1	2	3	4
1	0	1	-2	∞
2	4	0	3	∞
3	∞	∞	0	2
4	5	∞	∞	0

D^1

	1	2	3	4
1	0	1	-2	∞
2	4	0	2	∞
3	∞	∞	0	2
4	5	6	3	0

D^0 থেকে [1,1] fill up করে নিতে হবে।

K=1

$$D^1[2,3] = \min\{D^0[2,3],\ D^0[2,1] + D^0[1,3]\}$$
$$= \min\{3,\ 4+(-2)\}$$
$$= \min\{3, 2\}$$
$$= 2$$

D^2

	1	2	3	4
1	0	1	-2	∞
2	4	0	2	∞
3	∞	∞	0	2
4	5	6	3	0

D থেকে (2,2) Value আসবে

D^3

	1	2	3	4
1	0	1	-2	0
2	4	0	2	4
3	∞	∞	0	2
4	5	6	3	0

D^4

	1	2	3	4
1	0	1	2	~~∞~~
2	4	0	2	∞
3	7	8	0	2
4	5	6	3	0

Shortest path