Queue :→

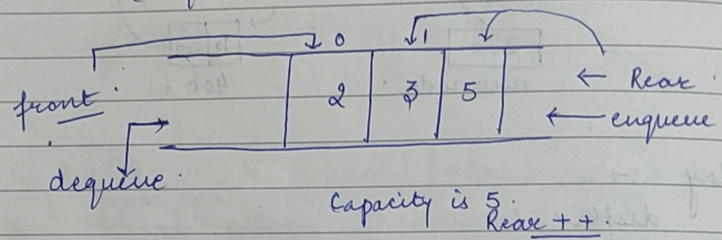
Linear Data Structure.

Abstract data type.

FIFO.

Rule → Insertion → Rear/tail (enqueue())

Deletion → Front/head. (dequeue())




Capacity is 5.

Rear++.

Operations :-

Enqueue

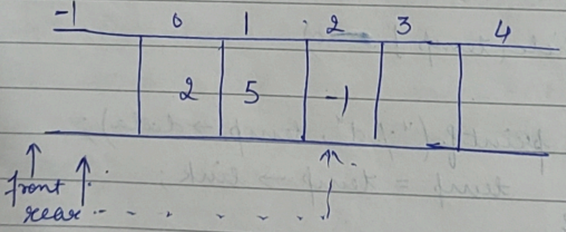
Dequeue

front() / Peek()

isfull()

isEmpty()

Using arrays.



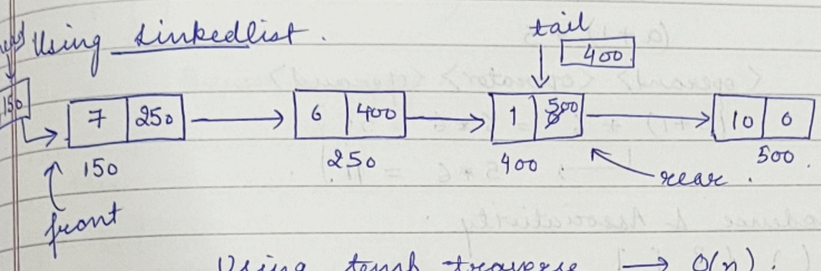

```
void dequeue()
{
  if (front == -1 && rear == -1)
    { "Underflow"
    }
  else if (front == rear)
    { front = rear = -1; }
```

```
void enqueue(x)
  if (rear == (N-1)
    { printf("Overflo
front & rear == -1
  {
    front = rear = 0
      queue[rear] = x
  }
else
  {
    rear++;
    queue[rear] = x
  }
```

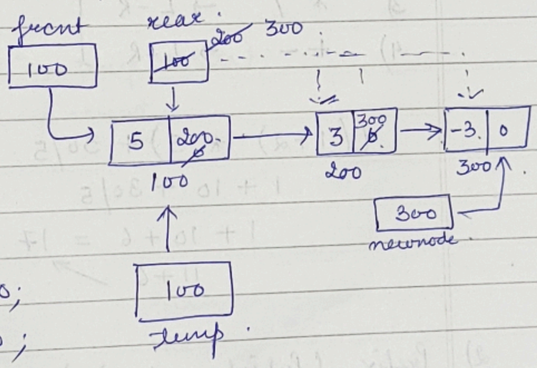
```
else
{ printf("%d", queue[front]);
  front++;
}
```

Complexity → O(1)

Using linkedlist.




Using temp traverse → O(n).
so we will use pointer to access the last node.




```
struct node
{
  int data
  struct node * next;
}
struct node * front = 0;
   "     "   " rear = 0;
void enqueue(int x)
{
  struct node * newnode;
newnode = (struct node*) malloc(sizeof(struct node));
        newnode → data = x;
        newnode → next = 0;
} if (front == 0 && rear == 0)
      {
        front = rear = newnode;
      }
  else
      {
        rear → next = newnode;
        rear = newnode;
      }
}
```

Infix, Prefix, Postfix :→

1) Infix :→ 5+1

p+q

a+5

(a+1)+5

<operand> <operator> <operand>

(5+1) * 6 = 6*6 = 36

↳ 5*6 = 41

Precedence & Associativity

1) ( ) { } [ ]
2) ^ → R-L
3) * / → L-R
4) + - → L-R

((1+2) * 5) + 30/5

1 + 10 + 30/5

1 + 10 + 6 = 17

11+6

2 ^ 2 ^ 3

2 ^ 8 = 256 ✓

4 ^ 3 = 64 ✗

2) Prefix (Polish notation)

<operator> <operand> <operand>

5+1

⇒ +51

a*b+c

⇒ *ab+c

⇒ +*abc

3) Postfix (Reverse Polish)

<operand> <operand> <operator>

5+1 = 51+

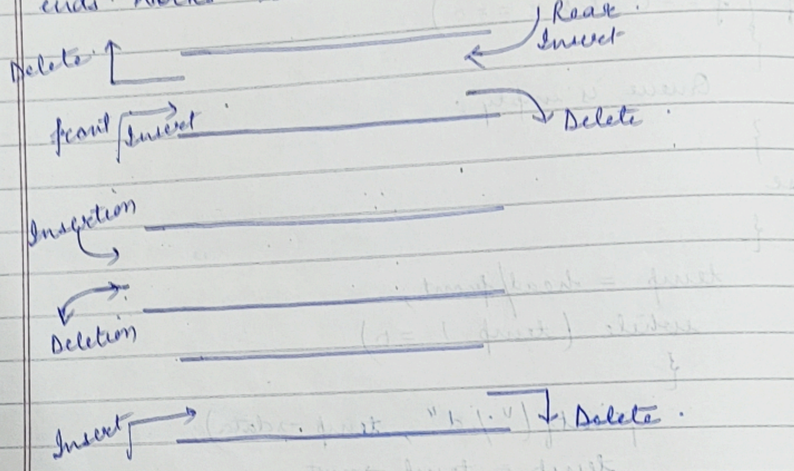
a* b+c ⇒ ab*+c

= ab*c+

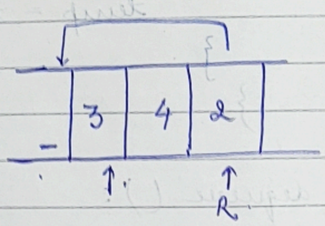
```
void display()
{
  struct node * temp;
  if (f == 0 && r == 0)
    {
      Queue is empty.
    }
  else
    {
      temp = head/front;
      while (temp != 0)
        {
          printf("%d", temp -> data);
          temp = temp -> next;
        }
    }
}

void dequeue():
{
  struct node * temp;
      temp = front;
  if (f == 0 && r == 0)
    {
      Queue empty.
    }
  else
    { printf("%d", front -> data);
      front = front -> next;
      free (temp);
    }
}
```

Dequeue :- Doubly ended queue.

↳ Insertion & deletion can take place at both the ends. Works like stack as well as queue.




Insert at front.
Delete from front
Insert at rear
Delete from rear.
get front, get rear
isempty() is full()

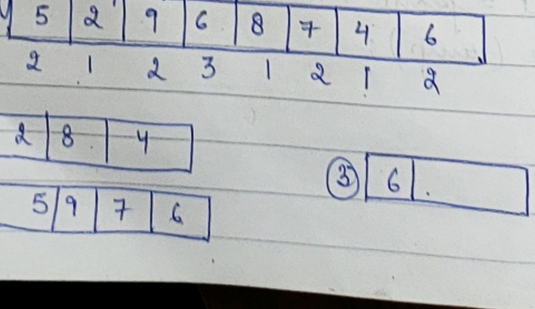



Priority Queue :→

1) Every element has a priority associated with it.
2) An element with higher priority is dequeued before an element with lower priority.
3) If two elements have the same priority, they are served according to their order in the queue.

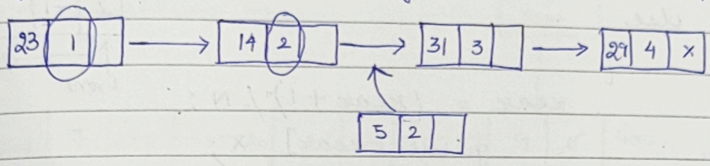
→ served first. processed first.

By using multiple queues.

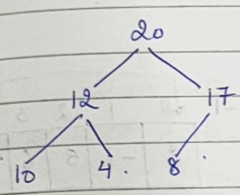


→ Deletion at the beginning

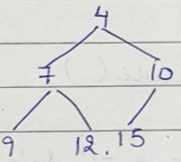
By using linked list (Ascending order acc. to their priority)




By using heap :→




Max heap.
Parent/Root > children

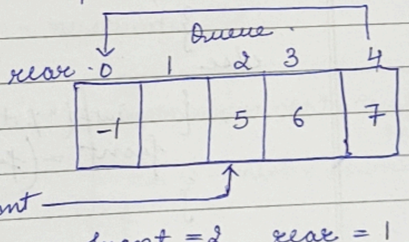



Min heap.
Parent/Root < children.

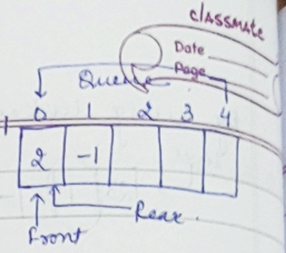
Elements will be arranged not according to their value but acc. to their priority. ~~जिसकी~~ priority सबसे ज्यादा है वो root बनेगा और कम priority वाले उसके child बनेंगे।

Circular Queue :→

```
# define N 5
int queue [N];
int front = -1;
int rear = -1;
void enqueue (int x)
{
  if (f == -1 and r == -1)
    {
      f = r = 0;
      queue [rear] = x;
    }
  else if ((rear+1) % N == front)
        { queue is full;
        }
```

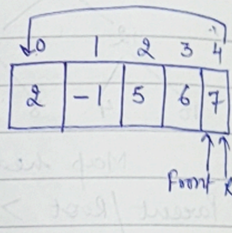



front = 2, rear = 1



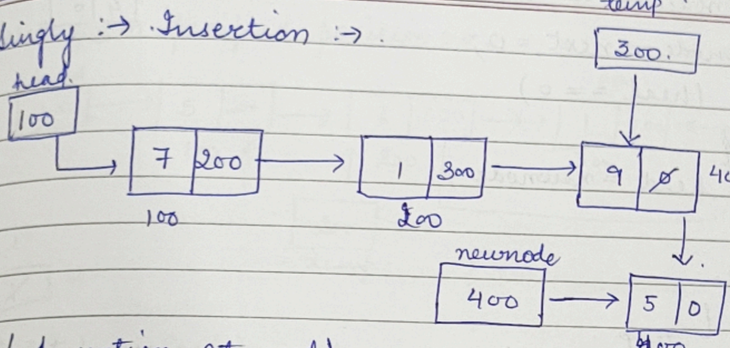

```
else
    {
        rear = (rear + 1) % N;
        queue [rear] = x;
    }
}
```

```
void dequeue()
{
  if (f == -1 && r == -1)
      {
        Queue is empty;
      }
  else if (f == r)
      {
        front = rear = -1;
      }
  else
      { printf("%d", queue[front]);
        front = (front + 1) % N;
      }
```



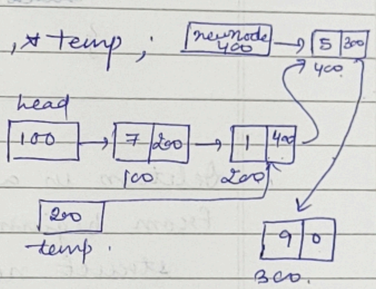

Linkedlist :→

Singly :→ Insertion :→




(Insertion at end).

```
struct node *head, *newnode, *temp;
newnode → data;
newnode → next = 0;
temp = head;
while (temp → next != 0)
{
    temp = temp → next;
}
temp → next = newnode.
```



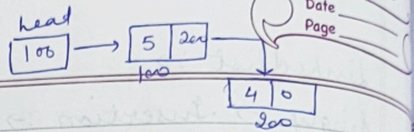

(Insert at a position) :→

position user input.

```
if (pos. > count)
    { Invalid position;
    }.

else
    { temp = head;
    while (i < pos)
    {
        temp = temp → next;
        i++;
    }.
    newnode → data;
    newnode → next = temp → next;
    temp → next = newnode;
    }.
```

At beginning :→

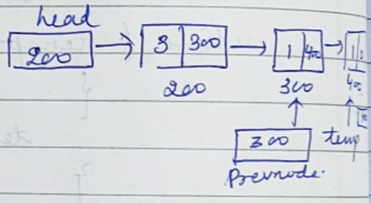
newnode → data .




```
newnode → next = 0;
if (head == 0)
  {
temp = head = newnode;
  }
else
  {
    temp
    temp → next = newnode;
    temp = newnode;
  }
```

Deletion in a linked list.

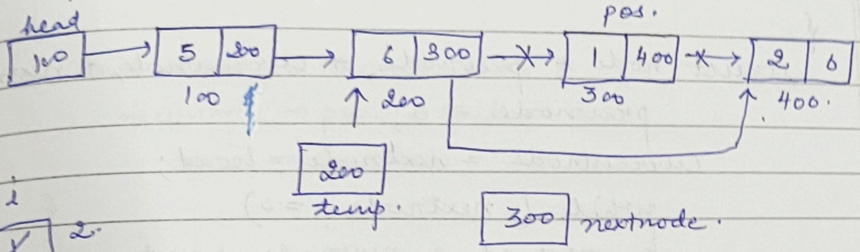
From beginning :→




```
struct node * temp;
  temp = head;
 head = head → next;
  free (temp);
```

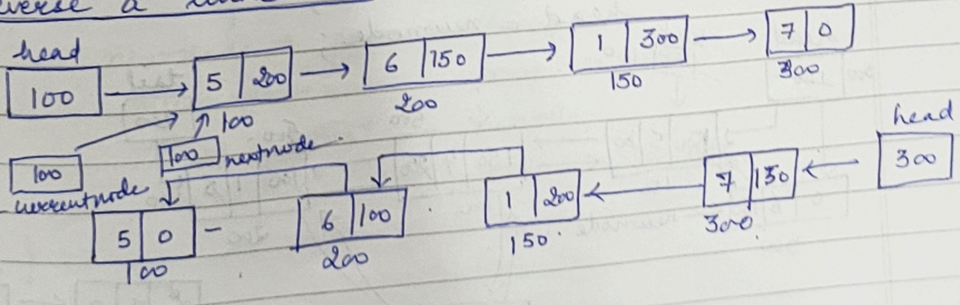
```
Delete from End ().
{
  struct node * prevnode;
    temp = head;
  while ( temp → next != 0)
    {
      prevnode = temp;
      temp = temp → next;
    }
    if (temp == head)
      {
        head = 0;
        free(temp);
      }
else
  { prevnode → next = 0;
    free (temp);
```

Delete from specified position :→.




```
DeletefromPos()
{
  struct node *nextnode;
  int pos, i = 1;
  temp = head;
  Enter pos. as input from user.
  while (i < pos-1)
  {
    temp = temp → next.
    i++;
  }
  nextnode = temp → next;
  temp → next = nextnode → next.
  free (nextnode);
}
```

Reverse a linkedlist :→

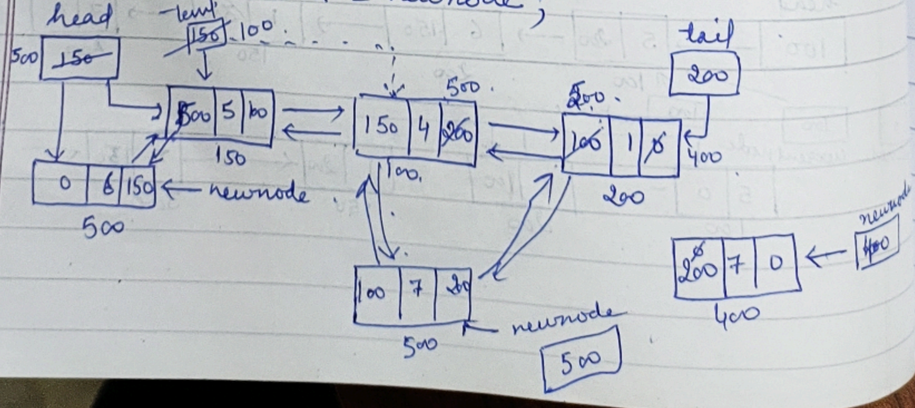


```
void reverse()
{
  struct node *prevnode, *currentnode, *nextnode;
      prevnode = 0;
    currentnode = nextnode = head;
     while (nextnode != 0)
     { nextnode = nextnode -> next;
       {currentnode -> next = prevnode};
        prevnode = currentnode;
       currentnode = nextnode;
     }
        head = prevnode;
}
```

Doubly linked list :-

Insertion :-

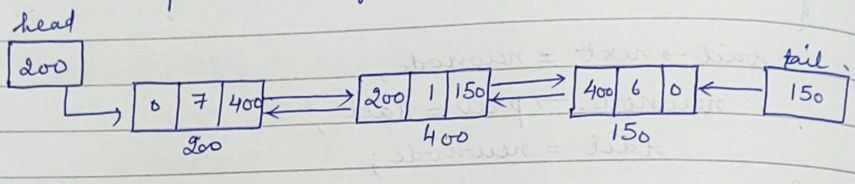
```
void insertatbeg()
  {
    struct node *newnode;
     newnode -> data;
     newnode -> prev = 0;
      newnode -> next = 0;
     head -> prev = newnode;
      newnode -> next = head;
          head = newnode;
```



```
InsertatEnd().
{
    tail → next = newnode;
    newnode → prev = tail;
    tail = newnode;
}.

insertatPos() {
        int pos;
    Enter pos. from user.
    if (pos > count).
        { invalid position.
        }
    else if (pos == 1)
        {
            insertatbeg().
        }.
    else
        { temp = head; i = 1;
          newnode → data
        while (i < pos - 1)
            {
                temp = temp → next;
                i++;
            }
        newnode → prev = temp;
        newnode → next = temp → next;
        temp → next = newnode.
        newnode → next → prev = newnode;
}.
```

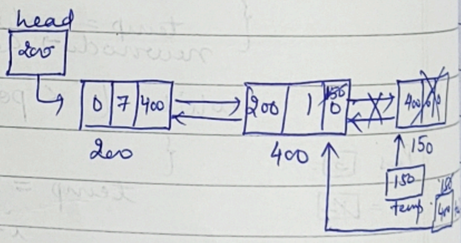
Pos 3
i = 2

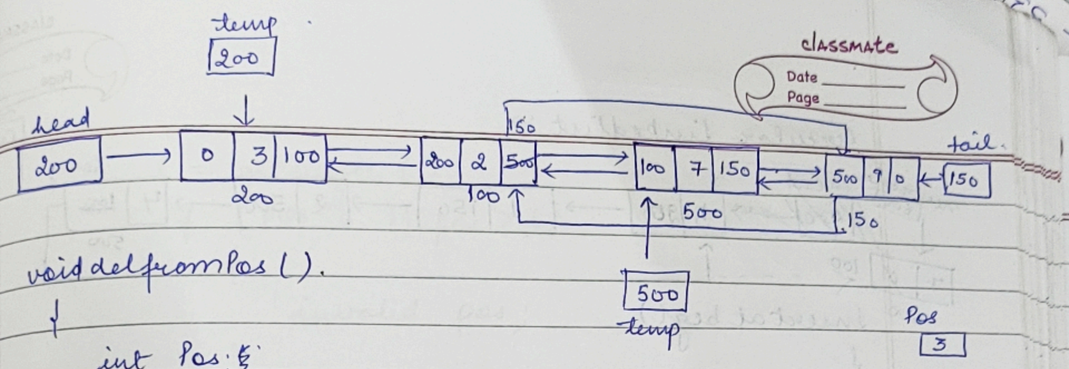
Deletion from doubly linked list :-




```
void delfrombeg()
 {
  struct node *temp;
    if (head == 0)
      {
        Empty;
      }
   temp = head;
   head = head -> next;
      head -> prev = 0;
       free (temp);
 }
```

```
void delfromEnd()
 {
  struct node *temp;
    if (tail == 0)
      {
      }
   else
      {
       temp = tail;
       tail = tail -> prev;
        tail -> next = 0;
        free (temp);
      }
```



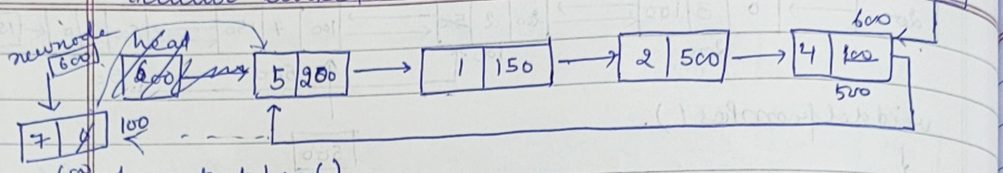



void delfromPos ().

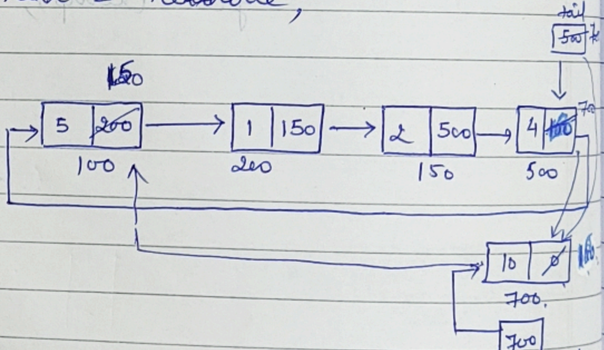
```
{
    int Pos, i;
    temp = head.
    Enter pos.
    while (i < Pos)
    {
        temp = temp → next;
        i++
    }.
temp → prev → next = temp → next;
temp → next → prev = temp → prev;
        free (temp);
```

Pos
3
i = 1

Circular linkedlist :→




```
Insert at beg()
{
  newnode → data;
  newnode → next = 0;
  if (tail == 0)
  { tail = newnode;
    tail → next = newnode;
  }
  else
  { newnode → next = tail → next;
    tail → next = newnode;
  }
}
```




```
Insert at end()
{
}
else
{  newnode → next = tail → next;
   tail → next = newnode;
   tail = newnode;
}
```

```
insert at Pos()
{
  int pos, i = 1;
  Enter from user.
```

```
l = getlength ()
if (pos<0 || pos > l)
  {
     invalid pos;
  }
else if (pos == 1)
  { insertatbeg();
  }.

  newnode → data.
   newnode → next = 0 ;
temp = tail → next;
   while (i < pos - 1);
      {
       temp = temp → next;
          i++;
      }.
newnode → next = temp → next;
    temp → next = newnode;
}-
```

Deletion :→.

```
from beg :-
   temp = tail → next
   if (tail == 0)
     {
     }.
else if (temp → next == temp)
        {
          tail = 0;
          free (temp);
        }.
else.
  {.
   tail → next = temp → next;
     free (temp);
  }.
```

```
l = getlength()
if (pos<0 || pos>l)
  {
     invalid pos;
  }
else if (pos==1)
  { insertatbeg();
  }.

   newnode → data.
   newnode → next = 0;
temp = tail → next;
  while (i<pos-1);
    {
      temp = temp → next;
        i++;
    }.
newnode → next = temp → next;
   temp → next = newnode;
}-
```

Deletion :→ .

from beg :- '

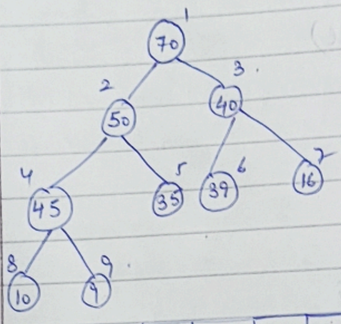
```
temp = tail → next
  if (tail==0)
    {
    }.
else if (temp → next == temp)
      {
        tail = 0;
        free(temp);
      }.
```

```
else.
  {.
    tail → next = temp → next;
      free(temp);
  }. .
```

Heaps.

Maxheap.

$A[parent(i)] \geq A[i]$.

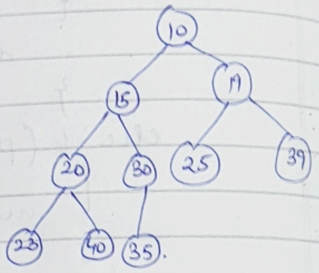



| 70 | 50 | 40 | 45 | 35 | 39 | 16 | 10 | 9 |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

$i = \lfloor \frac{i}{2} \rfloor$

Insert → 60, 5.

$A[parent(i)] \leq A[i]$




Deletion.

$i*2$.

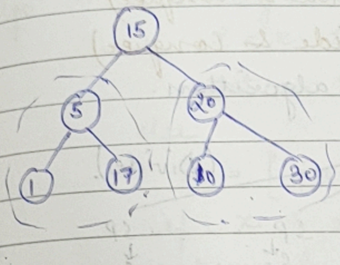
$(i*2)+1$

$O(n \log n)$

Heap Sort.

```
insertHeap (A, n, value)
{
    n = n+1;
    A[n] = value;
    i = n;
    while (i>1)
    {
        parent = ⌊i/2⌋
        if (A[parent] < A[i]);
        {
            swap (A[parent], A[i]);
            i = parent;
        }
        else
        {
            return;
        }.
    }
}
```

Heapify : O(n)



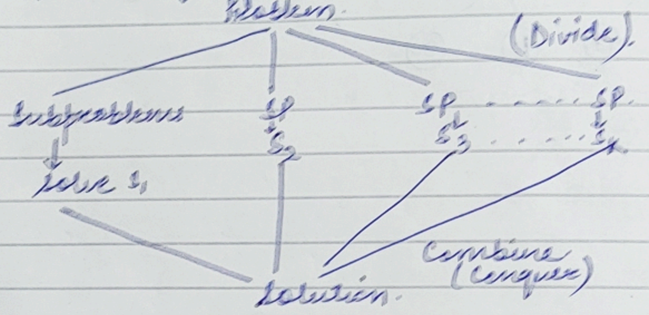

| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| 15 | 5 | 20 | 1 | 17 | 10 | 30 |

$\lfloor \frac{n}{2} \rfloor + 1$ to $n$.

Unit – II

Chapter – 4 (Divide & Conquer)

The general method (Divide & Conquer) used for designing the algorithm.




→ Parallelly can be solved on multiple machines using concept.

```
DAC(P)
{
  if (small(P))
    {
      s(P);
    }
  else
    {
      divide P into P1, P2, P3 - - - Pk.
      Apply DAC(P1), DAC(P2) - - - - DAC(Pk)
      combine (DAC(P1), DAC(P2) - - - DAC(Pk))
    }
}
```

Binary Search →

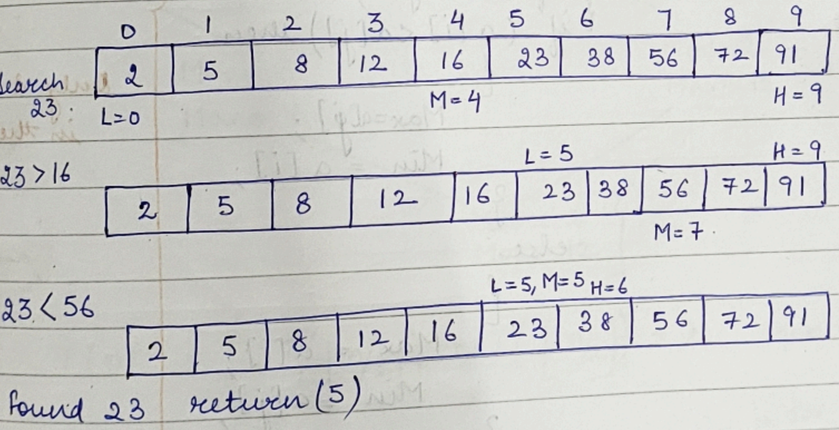
Performed on the sorted array.

In this we want to find whether element x belongs a set of no. stored in array "num[]" where 1 and n are left & right index of a subarray.

Algo :→ Binary Search (numbers[ ], x, l, r)

```
if l = r then
    return 1;
else
    m = [(l + r)/2]
if x ≤ numbers[m] then
return Binary Search (numbers[ ], x, l, m)
else
return Binary Search (numbers[ ], x, m+1, r)
```

Analysis :→ Linear Search (Unsorted array) O(n)
Binary " O(log n)




found 23 return (5)

Finding maximum & minimum of a sequence of nos.

Naive method (Basic method)

Algorithm :→ Max-Min Element (numbers[ ])

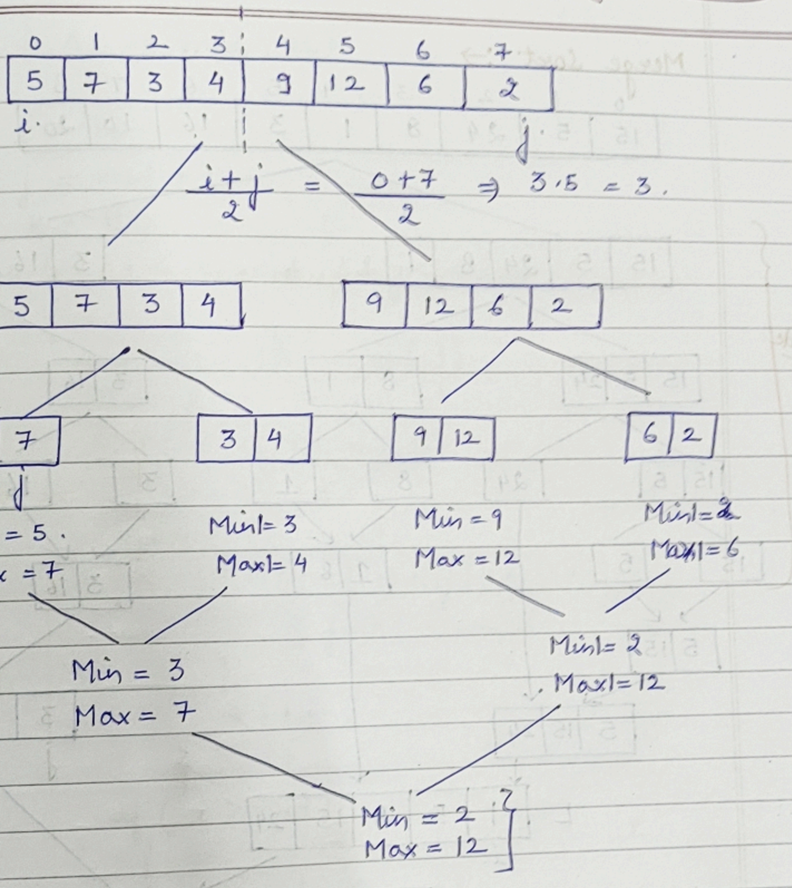
```
max = numbers[1]
min = numbers[1]
for i = 2 to n do
if numbers[i] > max then
    max = numbers[i]
```

```
if numbers[i] < min then
    min = numbers[i]
return (max, min)
```

Analysis ⇒ $2n - 2$

Using Divide & Conquer approach :→

```
Algorithm :→ MaxMin(i, j, Max, Min)
{
    if (i = j) then
    Max = a[i];
    Min = a[i];                          } One element in the array.
else if (i = j-1) then
    { if (a[i] < a[j]) then
        {
            Max = a[j];
            Min = a[i];
        }
    else
        {
            Max = a[i];
            Min = a[j];
        }
}                                        } Only 2 elements in the array
else
    {
    Mid = i+j/2;
    MaxMin(i, Mid, Max, Min);
    "      (Mid+1, j, Max1, Min1);
    if (Max < Max1) then
        Max = Max1;
    if (Min > Min1) then
        Min = Min1;
    } }                                  } More than 2 elements
```

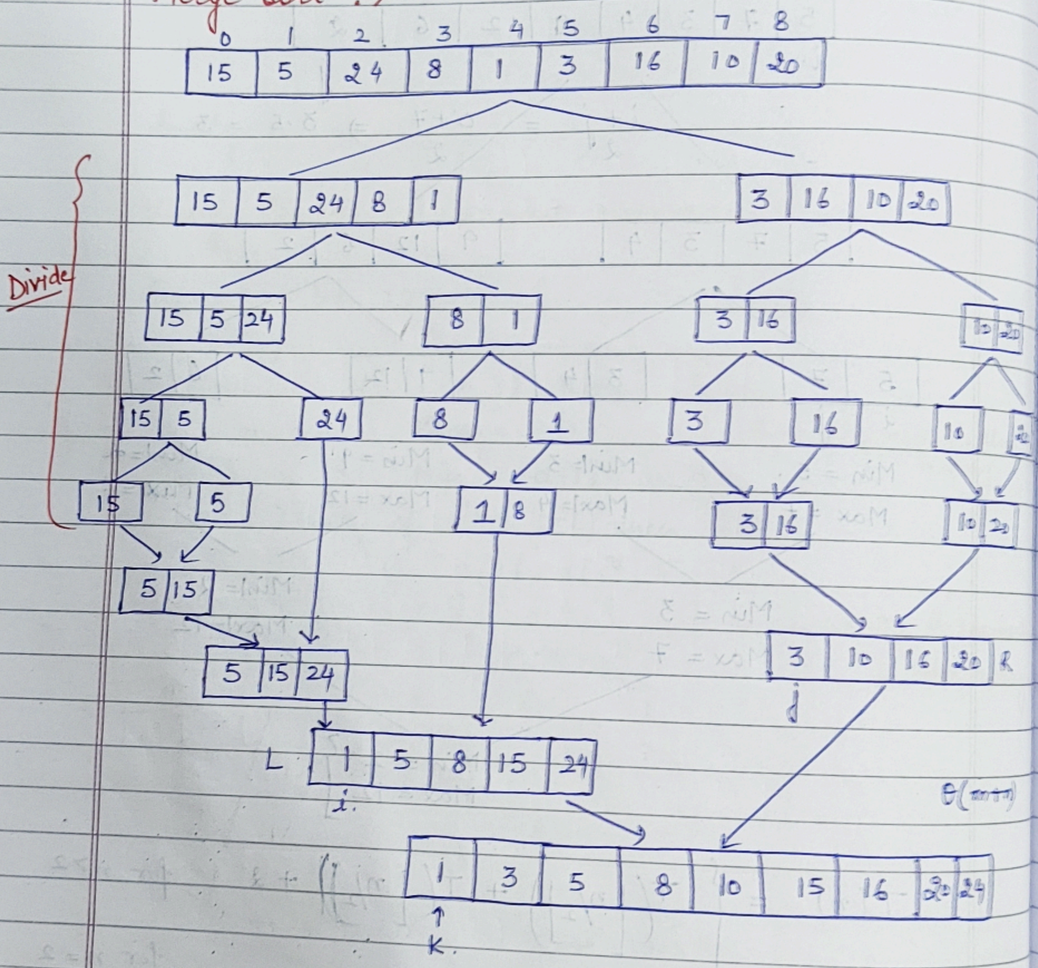



$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + T(\lfloor n/2 \rfloor) + 2 & \text{for } n > 2 \\ 1 & \text{for } n = 2 \\ 0 & \text{for } n = 1 \end{cases}$$

$T(n) = \frac{3n}{2} - 2$ (No. of comparisons less as compared to naive approach)

Solving $\Rightarrow$ $O(n)$. (Asymptotic notation).

## Merge Sort :→




Divide into two halves until we get sublist with one element then merge the two halves.

Algorithm :→

```
Mergesort (A, lb, ub)
{
    if (lb < ub)
    {
        mid = (lb + ub) / 2
    mergesort (A, lb, mid) ;
              (A, mid+1, ub) ;
    } merge ( A, lb, mid, ub);
```

```
Merge ( A, lb, mid, ub)
{
    i = lb;
    j = mid + 1;
    k = lb;
while (i <= mid && j <= ub)
    {
        if (a[i] <= a[j])
        {
            b[k] = a[i]
            i++;
        }
        else
        { b[k] = a[j];
            j++;
        }
        k++;
    }
if (i > mid)
    {
    while (j <= ub)
        {
            b[k] = a[j] ; j++;
            k++;
        }
    }
else if (j > ub)
    {
    while ( i <=mid)
        {
            b[k] = a[i];
            i++;
            k++;
        }
    };
```

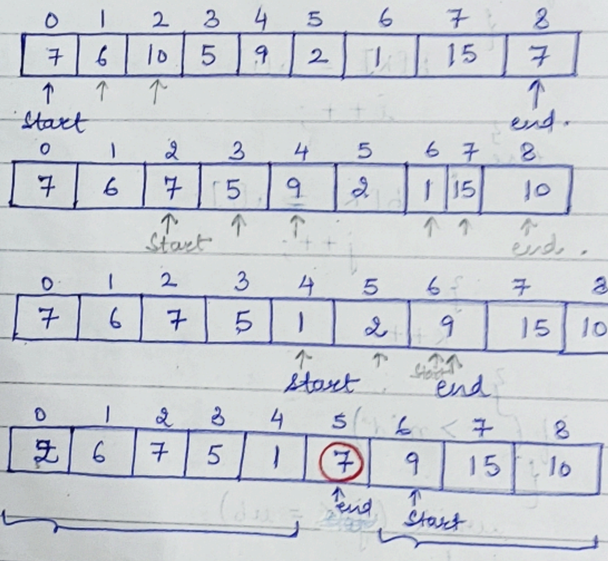
```
for (k = lb ; k <= ub ; k++)
    {
        a[k] = b[k];
    }
```

Analysis :→ $O(n \log n)$ {Best, Worst & average case}

$T(n) = 2T(n/2) + n$.

Quick-Sort :→

Pivot = 7.




```
Partition (A, lb, ub)
    {
        Pivot = a[lb];
        start = lb;
        end = ub;
    while (start < end)
        {
            while (a[start] <= Pivot)
                {
                    start++;
                }
```

```
while (a[end] > Pivot)
    {
      end--;
    }
if (start < end)
  {
    swap (a[start], a[end]);
  }
}
swap (a[lb], a[end]); return end;
}

QuickSort (A, lb, ub)
  {
    if (lb < ub)
      {
    loc = Partition (A, lb, ub);
        QuickSort (A, lb, loc-1);
        QuickSort (A, loc+1, ub);
      }
```

Analysis :- Average, Best → $O(n \log n)$

Worst → $O(n^2)$

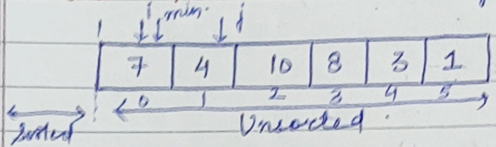
Sorted array :- 10 | 20 30 40 50 60 70

Pivot 10 : i ... j

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

$O(n^2)$

**Selection Sort :→** 2 arrays one sorted and another unsorted. Initially sorted array is empty and unsorted contains all the elements.




$n = 6$.

Pass 1 :→ | 1 | 4 | 10 | 8 | 3 | 7 |

Pass 2 :→ | 1 | 3 | 10 | 8 | 4 | 7 |

Pass 3 :→ | 1 | 3 | 4 | 8 | 10 | 7 |

Pass 4 :→ | 1 | 3 | 4 | 7 | 10 | 8 |

Pass 5 :→ | 1 | 3 | 4 | 7 | 8 | 10 |

Sorted | Unsorted

Algorithm :→

```
for (i = 0; i < n-1; i++)
{
    int min = i;
    for (j = i+1; j < n; j++)
    {
        if (a[j] < a[min])
        {
            min = j;
        }
    }
    if (min != i) {
        swap (a[i], a[min]);
    }
}
```

min [0] ✗ ✗ 5

j = ✗ ✓ ✓ ✗ 5

a[1] < a[0] ⇒ 4 < 7 ✓

a[2] < a[1] ⇒ 10 < 4 ✗

a[3] < a[1] ⇒ 8 < 4 ✗

a[4] < a[1] ⇒ 3 < 4 ✓

a[5] < a[4] ⇒ 1 < 3 ✓

5 ≠ 0

Time Complexity :→ $O(n^2)$

Both in best & worst Case.

Strassen's Matrix Multiplication :→

Naive method :→

```
void multiply (int A[][N], int B[][N], int C[][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            C[i][j] = 0;
            for (int k = 0; k < N; k++)
            {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

Time Complexity = $O(N^3)$

Divide & Conquer :→

For multiplying 2 square matrices divide matrices A & B in 4 sub matrices of size N/2 x N/2.

Then calculate following values recursively.

ae + bg, af + bh, ce + dg and cf + dh.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}_{A} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix}_{B} = \begin{bmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{bmatrix}_{C}$$

8 multiplications for matrices of size N/2 x N/2 & 4 additions. Addition of 2 matrices takes $O(n^2)$ time.

So time complexity = $8T(n/2) + O(n^2)$ or $4n^2$

so by solving using master's theorem time complexity is $O(n^3)$.

Strassen's Method :→

The idea of Strassen's method is to reduce the no. of recursive calls to 7. This " also divide matrices to submatrices of size $N/2 \times N/2$ but the result of 4 submatrices are calculated using following formula.

$p1 = a(f-h)$
$p2 = (a+b)h$
$p3 = (c+d)e$
$p4 = d(g-e)$
$p5 = (a+d)(e+h)$
$p6 = (b-d)(g+h)$
$p7 = (a-c)(e+f)$

$$\underbrace{\begin{bmatrix} a & b \\ c & d \end{bmatrix}}_{A} \times \underbrace{\begin{bmatrix} e & f \\ g & h \end{bmatrix}}_{B} = \begin{bmatrix} p5+p4-p2+p6 & p1+p2 \\ p3+p4 & p1+p5-p3-p7 \end{bmatrix}$$

$$T(N) = 7T(N/2) + O(N^2) \Rightarrow O(N^{\log 7})$$

$$= O(N^{2.8074})$$

## Greedy Algorithms :- The General Method :→

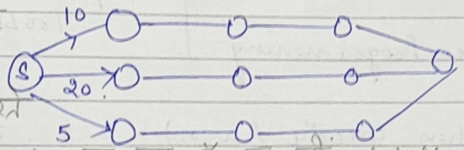
Follows local optimal choice at each stage with intend of finding global optimum.

→ Feasible solution ( based on some selection criteria )

→ Optimal solution →( Min. cost ) or →( Maximise the profit )

Offers → shop → discounts → बेशक बाद में पछताए कपड़ा अच्छा future में जाकर sol^n best ना मिले तो निकला। locally उस stage पर best sol^n देता है। →(Min. Risk)

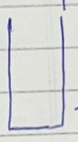
उसी stage पर find out करने की कोशिश करते हैं कि result best आ रहा है या नहीं globally आ रहा है या नहीं उसे नहीं देखते।




Fractional Knapsack ( Profit Maximise) → Constraint (Weight should be 20)

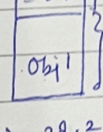
| Objects | Object 1 | Obj 2 | Obj 3 |
|---|---|---|---|
| Profit | 25 | 24 | 15 |
| Weight | 18 | 15 | 10 |
| | 1.3 | 1.6 | 1.5 |

Knapsack Capacity (M) = 20.




1) Greedy Profit :

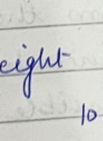
$25 + \frac{2}{15} \times 24$ ⇒ $25 + \frac{48}{15} \Rightarrow 28.2$



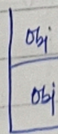

2) Greedy about Weight

$15 + \frac{10}{15} \times 24$

$\Rightarrow 15 + \frac{240}{15} \Rightarrow 31$




3) P/W both ⇒




$24 + \frac{5}{10} \times 15 \Rightarrow 24 + 7.5 \Rightarrow 31.5$

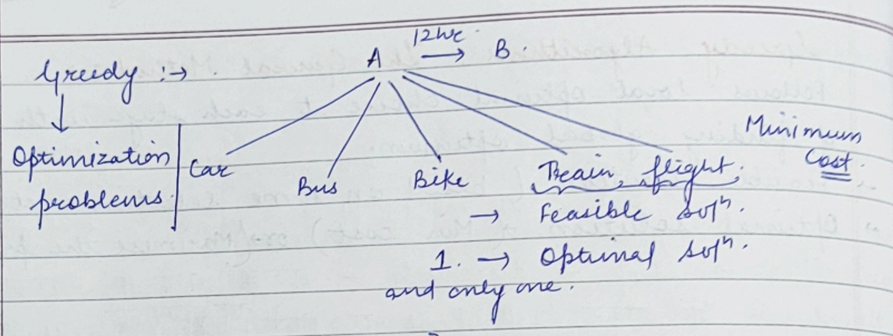
Overall complexity ⇒ O(nlogn)

O(n) [ for i = 1 to n. Calculate Profit/Weight

O(nlogn) [ Sort Objects in decreasing order of P/W Ratio.

O(n) [ for i = 1 to n.
if M > 0 and Wi ≤ M
M = M - Wi;
P = P + Pi;
else break;
if ( M > 0 )
P = P + Pi (M/Wi);
break;

Greedy :→




Optimization problems

→ Feasible soln.

1. → Optimal soln. and only one.

- → Branch & Bound.
- → Greedy.
- → Dynamic Programming.

} Optimization Problems.

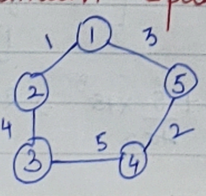
```
Algorithm Greedy (a, n)
{
    for i = 1 to n do
    {
        x = Select (a);
        if Feasible (x) then
        {
            Solution = Solution + x;
        }
    }
}
```

n = 5.

| a | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ |
|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 |

A problem is solved in stages. In each stage we will consider one input from that problem. If that I/P is feasible then it is included in the feasible soln then out of these feasible soln we will find optimal.
eg :- Car buying, Employee selection, Thief.

## Minimum Spanning Tree :→



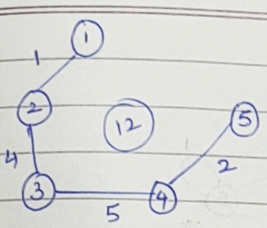

G = (V, E)

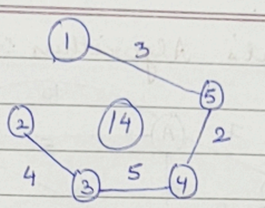
ST ⇒ G' (V', E')

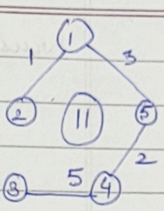
V' = V

E' ⊂ E

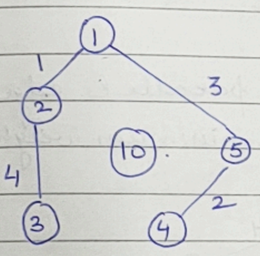
E' = |V| - 1



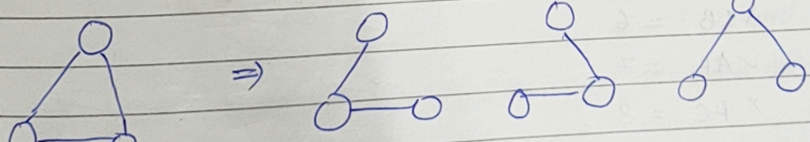

Take edges with minimum weights for constructing a MST.




⇒ MST.

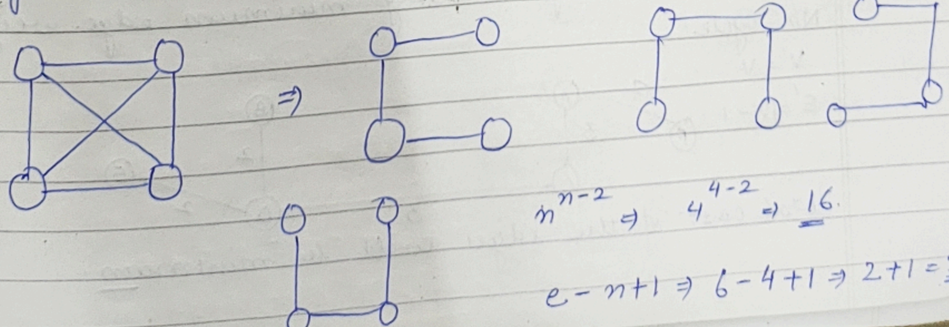
Total edge cost of that ST will be minimum.

→ Removing one edge from the ST will make it disconnected.
→ Adding one edge to the ST will create a loop.
→ If each edge has distinct weight then there will be only one and unique MST.
→ A complete undirected graph can have $n^{n-2}$ no. of ST. (no. of vertices)
→ Every connected & undirected graph has atleast one ST.
→ Disconnected graph does not have any ST.
→ From a complete graph by removing max $(e-n+1)$ edges we can construct a ST.



$3-3+1$
$\Rightarrow 1$

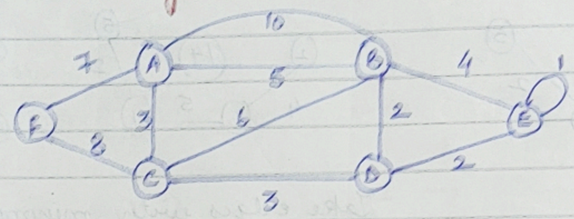
Complete Graph:→ every vertex is connected with every other vertex.



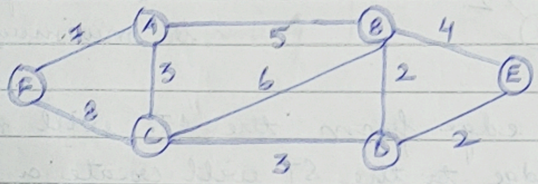
$n^{n-2} \Rightarrow 4^{4-2} \Rightarrow 16.$

$e-n+1 \Rightarrow 6-4+1 \Rightarrow 2+1 = 3$

Kruskal's Algorithm :-




Step 1:- Remove all the loops and parallel edges from the graph. Keep the edge having minimum weight.



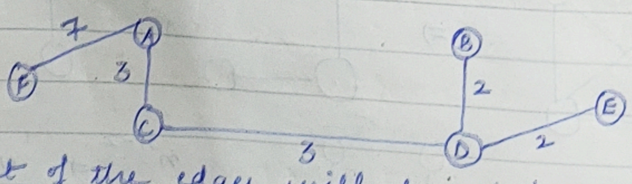

Step 2:- Arrange all the edges are in the increasing order of their edge weight/cost.

- ✓ BD = 2
- ✓ DE = 2
- ✓ AC = 3
- ✓ CD = 3
- ✗ BE = 4
- ✗ AB = 5
- ✗ CB = 6
- ✓ AF = 7
- ✗ FC = 8

Step 3:- Choose the edge having minimum edge weight. No cycle.

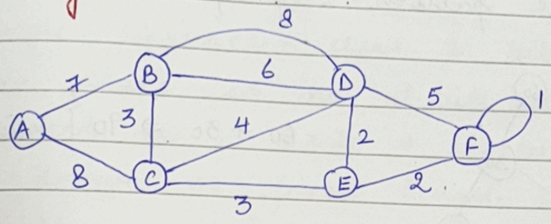
$V' = V$

$E' = |V| - 1$

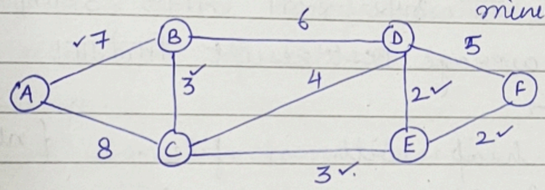



Total cost of the edges will be minimum.

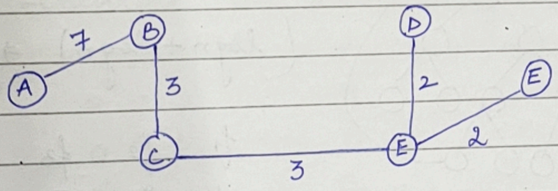
## Prim's Algorithm :→




Step 1 → Remove all the loops & parallel edges. Keep the edge having minimum weight.




Step 2 :- Choose any arbitrary vertex as root node. Choose all the outgoing edge from the root node. minimum weight




## Optimal merge patterns :→

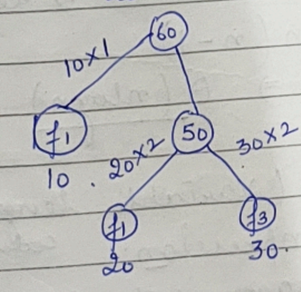
minimum होनी चाहिए

files :- f1 f2 f3.

Records :- 10 20 30.

Find no. of record movements.

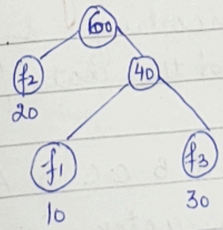
Atmax two files can be merged.




Total record movement ⇒ 60 + 50
⇒ 110.

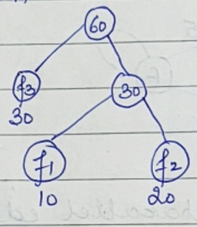
or

10×1 + 30×2 + 20×2 ⇒ 110




60 + 40 ⇒ 100.

Optimal merge pattern says pick minimum 2 then merge them.

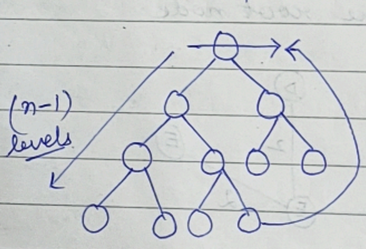



60 + 30 ⇒ 90 } min. value.

heavy files को ऊपर रखो ताकि उसकी movement कम हो। On an average Best result मिलेगा।

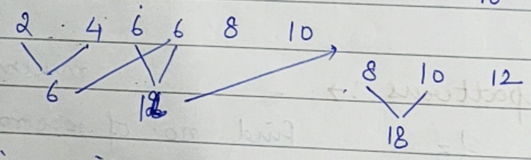
Algorithm :→

1) Create min heap with n files. {nlogn}
2) At each level remove two minimum 'a' & 'b' from min heap & put a+b again into heap.




(logn + logn) ⇒ 2logn + logn.
= 3logn.

f1 f2 f3
10 20 30 30 ( Add & then rearrange




This will be repeated till (n-1) levels if n elements are there. so 3 logn (n-1)

⇒ O(nlogn)

Huffman Coding :→ (Variable length)
Data encoding & compression coding.
Size of the data/message is to be reduced.

A B B C D B C C D A A B B E E E B E A B.

Each character is encoded in ASCII form. & then converted to binary form.

A → 65 — 01000001 (8 bits are used).
Single character uses 8 bits then here 20 characters are there so no. of bits used to send this message will be 8 × 20 ⇒ 160 bits (without compression).
Also known as fixed length coding. (Each character is simply a string of 0 & 1 4 comprising of 8 characters. B → 66 → 01000010.
After applying some compression technique we can reduce the no. of bits to be transferred or cost of this message.
2 bits → combinations possible are 4. 00, 01, 10, 11 (A B C D)
(4 diff. characters can be represented)
3 bits → $2^3 = 8$. → 4 we can represent 8 characters.
$2^4$ → 16 characters $2^8 = 256$.
In this example we have only 5 diff. characters so we the help of 3 bits we can represent these characters.

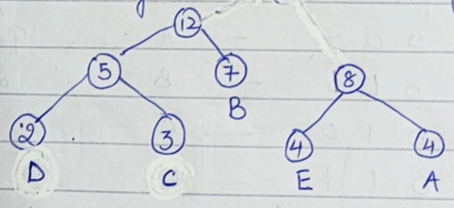
| | | |
|---|---|---|
| 0 0 0 | — A | Total bits reqd. |
| 0 0 1 | — B | 20 × 3 ⇒ 60 bits. |
| 0 1 0 | — C | |
| 0 1 1 | — D | |
| 1 0 0 | — E | |
| 1 0 1 | unused. | |
| 1 1 0 | | |
| 1 1 1 | | |

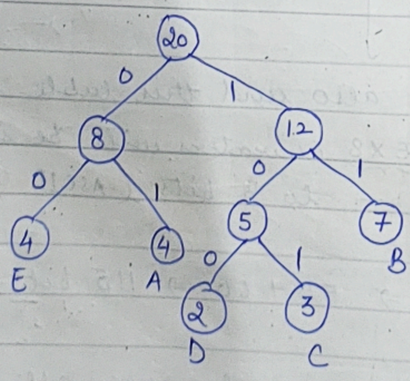
Decoding → We will also send this table with this table.
5 × 3 + 5 × 8 characters will be sent converted
15 + 40 to 8 bits (ASCII code)
⇒ 55
Total bits ⇒ 55 + 60 ⇒ 115 bits.
Huffman Coding :→
Design huffman tree - Arrange all characters acc. to increasing order of their count.

| Char | frequency/count | |
|---|---|---|
| A | 4 | 01 |
| B | 7 | (11) less no. of bits. |
| C | 3 | 101 |
| D | 2 | 100 |
| E | 4 | 00 |

D C E A B.
2 3 4 4 7

Take 2 characters with least frequency/count. or we can say priority queue because 2 is having more priority then C, E and so on. or can be implemented using min heap tree. Now in queue we have 4 4 7 & 5 and then arrange in increasing order. Now 5, 7 & 8. Min freq. count will be on left side & greater on right side. Now left only 8, 12




Characters will be at the leaf node. Left side ⇒ 0
Right child ⇒ 1

E ⇒ 00
A ⇒ 01
B ⇒ 11
D ⇒ 100
C ⇒ 101

$4 \times 2 + 7 \times 2 + 3 \times 3 \times 2 \times 3 \times 4 \times 2 \Rightarrow 45$ bits.

Decoding ⇒ 5×8 + 12 ⇒ 52.

Total ⇒ 45 + 52 ⇒ 97 bits.

Dynamic Programming.

0/1 Knapsack Problem :-

Max. Profit. (Max. size to store these items) W = ___ kg.

Items → Weights → {3, 4, 6, 5} W = 8 kg.

Profits → {2, 3, 1, 4} n = 4

0 → Not picked 1 — Picked (completely).

2kg, 3kg → 2kg left − 1kg.

1kg. Complete laptop ; Mixer.

Fractional — Greedy $x_i = \{1, 0, 0, 1\}$.

Total no. of items = 4, Total no. of combinations ⇒ $2^4 = 16$.
Check which 1's to select to get max profit.

| i ↓ / w → | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| $f_i$ $w_i$ 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 3 1 | 0 | 0 | 0 | 2 ← | 2 | 2 | 2 | 2 | 2 |
| 3 4 2 | 0 | 0 | 0 | 2 ← | 3 | 3 | 3 | 5 | 5 |
| 4 5 3 | 0 | 0 | 0 | 2 | 3 | 4 | 4 | 5 | 6 |
| 1 6 4 | 0 | 0 | 0 | 2 | 3 | 4 | 4 | 5 | 6 ← max. profit |

In 3rd row 0 to 3 copy values from above cell.

max (3+0, 2) ⇒ 3. , max (3+0, 2)

Formula → $m[i,w] = \max(m[i-1, w], m[i-1, w-w_i] + P[i])$

$x_i = \{1, 0, 0, 1\}$

Profit ⇒ 6 − 4 ⇒ 2
2 − 2 = 0

Weight ⇒ 8 − 5 = 3.
3 − 3 = 0

## Subset Sum Problem :-

$A = \{2, 3, 5, 7, 10\}$

Sum $(w) = 14$.

| (i) n ↓ \ Sum(j) → | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 2 | 1 | 0 | 1← | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 3 | 1 | 0 | 1← | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 5 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1← | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 3 7 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| 4 10 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |

Sum − A[i] ⇒ 3 − 3 = 0.

{2, 5, 7}

{ 14 − 7 = 7
7 − 5 = 2
2 − 2 = 0 }

Time complexity ⇒ $\theta(n \times Sum)$

m [i] [j] = 1

1) If A[i] = j
2) A[i − 1][j] = 1
3) When A[i − 1][j − A[i]] = 1

## Longest Common Subsequence :→

no. of characters in increasing order w.r.t. their position.

eg :→ w = abcd

= ab, bd, ~~ca~~, ac, ad, ~~db~~, abcd, bcd,

x :− a b a a b a

y :− b a b b a b

| y → / (i) → | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| | A | b | a | b | b | a | b |
| 0 A | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 a | 0 | ← 0 | ↖ 1 | ← 1 | ← 1 | ↖ 1 | ← 1 |
| 2 b | 0 | ↖ 1 | ← 1 | ↖ 2 | ↖ 2 | ← 2 | ↖ 2 |
| 3 a | 0 | 1 ↑ | ↖ 2 | ← 2 | ← 2 | ↖ 3 | ← 3 |
| 4 a | 0 | 1 ↑ | ↖ 2 | ← 2 | ← 2 | ↖ 3 | ← 3 |
| 5 b | 0 | ↖ 1 | ↑ 2 | ↖ 3 | ↖ 3 | ← 3 | ↖ 4 |
| 6 a | 0 | ↑ 1 | ↖ 2 | ↑ 3 | ← 3 | ↖ 4 | ← (4) ↑ |

Length of common subsequence is 4.

If diagonally the answer you received then we will check out the characters :- b a b a

from right to left. b a a b

Formula :-

If ( x[i] == y[j] ; c[i][j] = 1 + c[i-1][j-1]

Otherwise c[i][j] = max ( c[i][j-1], c[i-1][j] )

Coin Change Problem :-

Given :- Coins of different denomination & amount.

Problem :- Find out min. no. of coins to make change of given amount using given coins. (Infinite supply of coins).

Greedy :-

Coins → { 1, 5, 6, 9 }

W = 11. highest denomination ≤ or = 11

→ (9, 1, 1) 11 - 9 → 2

Optimal ✗ 3 coins 2 - 1 → 1

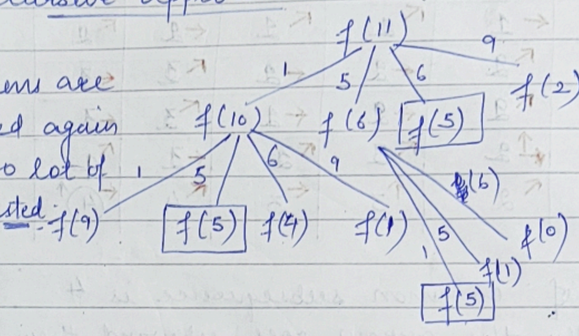
1 - 1 → 0.

(5, 6) → 11

2 coins.

Dynamic Programming will give you the optimal sol$^n$ if the sol$^n$ exists.

Recursive approach :→




Subproblems are recomputed again & again so lot of time is wasted.

Time complexity ⇓ $2^n$.

Greedy (2nd Drawback) :→ Coins = (4, 10, 25)
W = 41

(25, 10, 4, —)
↑
we cannot choose any coin < or = 2
so acc. to greedy the sol$^n$ does not exist but acc. to DP the sol$^n$ exists.

41 − 25 ⇒ 16
16 − 10 ⇒ 6
6 − 4 ⇒ 2

(25, 4, 4, 4, 4) ⇒ 5 coins.

Coins ⇒ {1, 5, 6, 9}
W ⇒ 10.

| i ↓ \ Coins | j → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 1 5 | 0 | 1 | 2 | 3 | 4 | 1 ← | 2 | 3 | 4 | 5 | 2 |
| 2 6 | 0 | 1 | 2 | 3 | 4 | 1 | 1 | 2 | 3 | 4 | 2 |
| 3 9 | 0 | 1 | 2 | 3 | 4 | 1 | 1 | 2 | 3 | 1 | 2 |
| | | | | | | | | | | | |

2 Coins → 10 − 5 ⇒ 5 (5, 5)
5 − 5 ⇒ 0.

if coin(i) > W just copy the value from above

5 > 1 (5 − 5)
↑
min (5, 1 + 0) ⇒ 1. min (6, 1 + 1) ⇒ 2

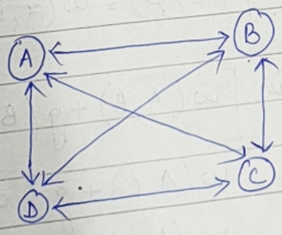
```
a[i][j]                                    a[i][0] = 0.
for (i=0 ; i <= coins.length ; i++)
    {
        for(j = 0; j <= W; j++)
        {
            if (coins[i] > j)
            {
                a[i][j] = a[i-1][j]
            }
            else {
        a[i][j] = min ( a[i-1][j], 1 + a[i][j - coins[i]])
        }}}
```

Time Complexity $\Rightarrow$ $n \times w$ (Coins) (Amount)

$O(n \times w)$

**Travelling Salesman Problem :→**

Salesman is there he starts travelling from his own place / city and then covers all the cities and returns back to ~~his~~ own city. but condition is he can visit all the cities exactly once.




Find out the root having minimum cost, distance or weight.

| | A | B | C | D |
|---|---|---|---|---|
| A | 0 | 16 | 11 | 6 |
| B | 8 | 0 | 13 | 16 |
| C | 4 | 7 | 0 | 9 |
| D | 5 | 12 | 2 | 0 |

Adjacency Matrix.

A → B → C → D → A.

A → D → C → B → A.

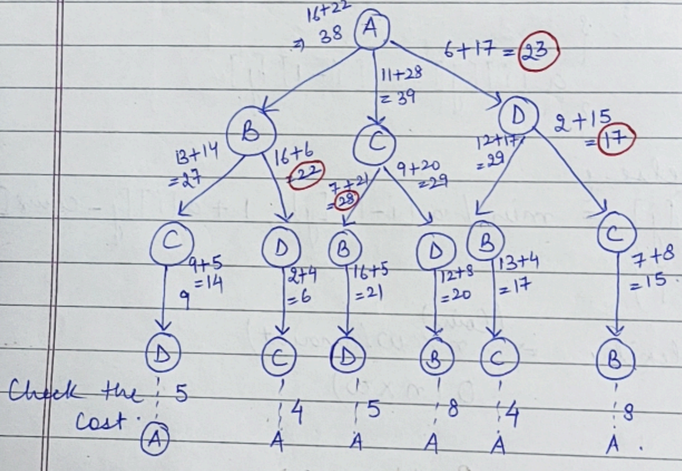
A → (D) → C → B → (D) → A.

visited twice.

(1) Brute force ~~(all possible paths)~~
Traverse each & every path & then find out the minimum path. $(n!)$

(2) Dynamic Programming.

(3) Branch & Bound.




Cost fnc.

$$g(i,S) = \min\left[w(i,j) + g(j,\{S-j\})\right]$$

$j \in S$.

$i$ = Starting vertex
$S$ → set of vertices that salesman is visiting exactly once & then coming back to its vertex.

$$g(x,\phi) = w(x,i)$$

$i = A$.

$$g(A,\{B,C,D\}) = \min\left[\begin{array}{l} w(A,B) + g(B,\{C,D\}) \Rightarrow 16+22 \\ w(A,C) + g(C,\{B,D\}) \Rightarrow 11+28 \\ w(A,D) + g(D,\{B,C\}) \Rightarrow 6+ \end{array}\right]$$

Recursive call.

$$g(B,\{C,D\}) = \min\left[\begin{array}{l} w(B,C) + g(C,\{D\}) \Rightarrow 13+14 \\ w(B,D) + g(D,\{C\}) \Rightarrow 16+6 \end{array}\right]$$

$g(C, \{D\}) \Rightarrow [w(C,D) + g(D, \phi)]$ — weight; we cannot go anywhere so we will write the starting vertex.

$= w(C,D) + w(D,A)$

$= 9 + 5 = 14.$

$g(D, \{C\}) = [w(D,C) + g(C, \phi)]$

$= [w(D,C) + w(C,A)]$

$= 2 + 4 = 6.$

$g(C, \{B,D\}) \Rightarrow \min \begin{bmatrix} w(C,B) + g(B,\{D\}) \\ w(C,D) + g(D,\{B\}) \end{bmatrix} \Rightarrow \begin{matrix} 7+21 \Rightarrow (28) \\ 9+20 \Rightarrow 29 \end{matrix}$

$g(B, \{D\}) \Rightarrow [w(B,D) + g(D, \phi)]$

$\Rightarrow 16 + 5 \Rightarrow 21$

$g(D, \{B\}) \Rightarrow [w(D,B) + g(B, \phi)]$

$= 12 + 8 \Rightarrow 20.$

$g(D, \{B,C\}) = \begin{bmatrix} w(D,B) + g(B,\{C\}) \\ w(D,C) + g(C,\{B\}) \end{bmatrix} \begin{matrix} \Rightarrow 12+17 \Rightarrow 29 \\ \Rightarrow 2+15 \Rightarrow (17) \end{matrix}$

$g(B, \{C\}) \Rightarrow [w(B,C) + g(C, \phi)]$

$= 13 + 4 = 17.$

$g(C, \{B\}) \Rightarrow [w(C,B) + g(B, \phi)]$

$= 7 + 8 \Rightarrow 15.$

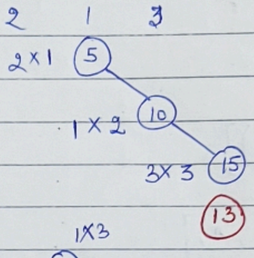
Route would be $\rightarrow$
$A \rightarrow D \rightarrow C \rightarrow B = 23.$

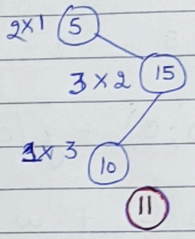
## Optimal Binary Search Tree :→

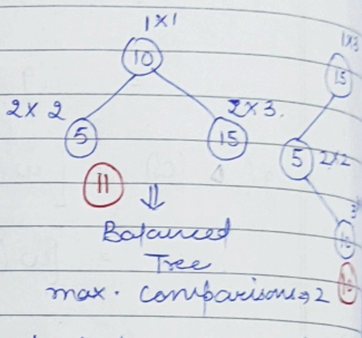
Minimum cost. (searching)

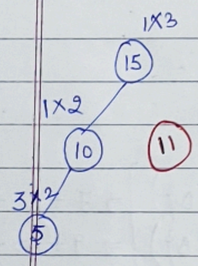
Keys : 5, 10, 15.

Cost/Freq : 2 1 3




Although balanced but not having the minimum cost.

Brute force method → Not suitable for a large tree. (Time consuming) + more calculations.

Dynamic Programming.

How to create tree / or adjust nodes so that search cost should be minimum

Vertex :→ 1 2 3 4

Keys :→ 5 10 15 20.

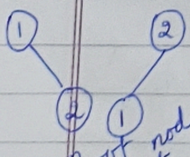
Freq. :→ 3 1 2 4

node to node cost is zero.

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 3[1] | 5[1] | 10[1] | 14[1] |
| 1 | | 0 | 1[2] | 4[3] | 4[4] |
| 2 | | | 0 | 2[3] | 8[4] |
| 3 | | | | 0 | 4[4] |
| 4 | | | | | 0 |

Cost

$$c[x,y] = \min_{(x \le k < y)} \{ c[x,k] + c[k+1,y] \} + w[x,y].$$




Root node select as 2 || 1

$$c(0,1) = \min \{ c[0,0] + c[1,1] \} + w(0,1) \Rightarrow 3$$

no vertex as 0 so taken for calculation only.

→ first value ignore always. range time 42.

$c(\phi, 2) \Rightarrow \min \{ c[1,1] + c[2,2] \}$

$+ w(1,2)$.

$\Rightarrow 1$.

$c[2,3] = c[2,2] + c[3,3] + w(2,3)$

$= 2$.

$c(3,4) = c[3,3] + c[4,4] + w(3,4)$.

$= 4$.

$c(0,2) = \min \{ c[0,0] + c[1,2], c[0,1] + c[2,2] \}$

$+ w(0,2)$

$= \min \{ 0+1, 3+0 \} + 4$

$= 1 + 4 \Rightarrow 5$.

$c(1,3) = \min \{ c(1,1) + c(\ ,\ )$

$c(1,$