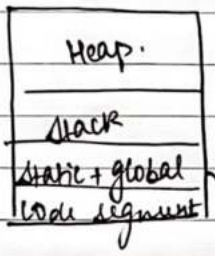
PAGE No.
DATE / /

DATA STRUCTURES

• Introduction to data structures and algorithm

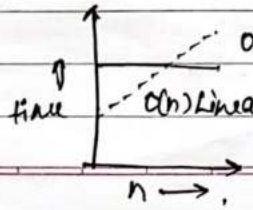
* **Data structures:** way to arrange data in main memory (RAM) for efficient usage.
  Eg. Arrays, Queues, linked list.
* **Algorithm:** Sequence of steps to solve a given problem.
* **Database:** collection of information in permanent storage for faster retrival and updation. [Hard disk]
* **Data warehouse:** Managment of huge amount of legacy data for better analysis.
* **Big Data:** Analysis of too large / complex data which cannot be dealt with traditional data processing application.



* **Stack** holds the memory occupied by functions. It stores the activation records of the functions used in the program. and erases them as they get executed.
* **Static + global** holds initialized and uninitialized global variables.
* **Heap** ~~holds memory occupied by functions~~ contains the data which is requested by the program as dynamic memory using pointers.

• Time complexity and Big O Notation

'O' → Big 'O'
→ Order of
→ In Mathematics :- all those complexities our programs run in.
→ In industry → Minimum of all.



PAGE No.
DATE / /

- Asymptotic Notations : Big O, Big Omega and Big theta.

Big 'O' notation :- asymptotic upper bound

f(n) describes the running time of an algorithm.

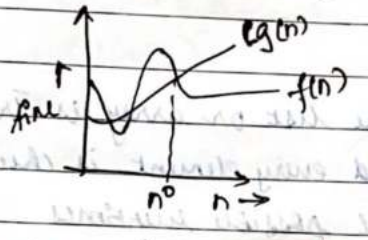
f(n) is O(g(n)) if and only if there exist positive constants c and n^0 such that

$0 \leq f(n) \leq c\,g(n)$ for all $n \geq n^0$.

n = input size

g(n) = any complexity function eg n, n^2

(used to give upper bound on a function).

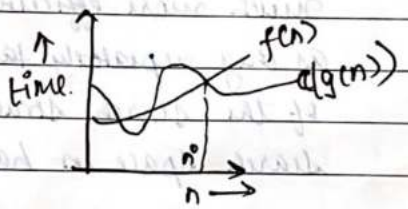


Big Omega notation (Ω) :- asymptotic lower bound.

f(n) describes the running time of an algorithm.

f(n) is said to be $\Omega(g(n))$ if and only if there exists positive constants c & n^0 such that.

$0 \leq c\,(g(n)) \leq f(n)$ for all $n \geq n^0$

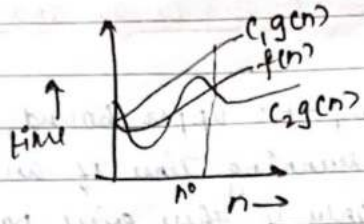


Big theta notation (Θ) :-

F(n) is said to be $\Theta(g(n))$ if f(n) is O(g(n)) and f(x) is $\Omega(g(n))$ both.

$0 \leq c_2(g(n)) \leq f(n) \leq c_1(g(n)) \;\forall\; n \geq n^0$.

there simply exist positive constants c_1 & c_2 such that f(n) is between $c_2(g(n))$ and $c_1(g(n))$.



$1 < \log n < n < n\log n < n^2 < n^3 < 2^x < n^x$

↑ better. ↑ worst

- **Best case, worst case and Average case. Analysis of an algorithm**

ANALYSING ALGORITHM

Linear Search.

TIME COMPLEXITY

Best case = $O(1)$
worst case = $O(n)$
Average case = $O\left[\leq \frac{\text{all possible runtimes}}{\text{total no. of possibilities}}\right]$

The list or array is traversed sequentially and every element is checked.

Binary Search.

Best case = $O(1)$
worst case = $O(\log n)$.

These algorithms are specifically designed for searching in sorted data structures. These type of searching algorithms are much more efficient than Linear Search as they repeatedly target the center of the search structure and divide the search space in half.

SPACE COMPLEXITY

→ Creating an array of size n (size of the input) = $O(n)$ space.
→ If a function calls itself recursively n times, its space complexity is $O(n)$

✱ We cannot calculate time complexity in seconds because not everyone's computer is equally powerful. Asymptotic analysis is the measure of how time (runtime) grows with input.

PAGE No.
DATE / /

How to calculate time complexity of an algorithm

Step 1:- Drop the non dominant terms

Step 2:- Drop the constant term.

Step 3:- Break the code into fragments

```
int i;          ]
int n=0         ] k1

for (i=0; i<n; i++){
    k = i;      ]
    k++;        ] k2
    x = y+1;    ]
}
for (j=0; j<n; i++){
    k = i+1;    ]
    k--;        ] k3
}
```

$T(n) = k_2 n + k_1$

$T(n) = k_2 n$

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

$T(n) = k_1 + k_2 n + k_3 n$

$= n(k_2 + k_3)$

$= n(k_4)$

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

$\leftrightarrow (k_1) \leftrightarrow$

Eg-

```
for (i=0; i<n; i++){
    for (j=0; j<n; j++){
        k = i+1;
        k++;
    }
}
```

i	j
0	0
0	1
0	2
0	3
:	:
1	0
1	1
1	2
1	3
:	:
n	0
n	1
n	2
:	:

$T(n) = k_1 + k_2 n^2 + n k_3$

$= k_2 n^2$

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

PAGE No.
DATE / /

• Arrays and Abstract data Types in Data structure.

ADT → Minimal requirement functionalities : [MRF]
→ operations.

ARRAYS → get (i) } MRF Methods
Set (i, num) } → Insert → Delete → Add → Resize.

collection of elements accessible by an index.

* Abstraction :- Implementation is hiden but still you can use it.
Static arrays → size cannot be changed
Dynamic arrays → size can be changed.

Memory representation of arrays :-

Index →	0	1	2	3
	7	9	13	2
address →	10	14	18	22 26

⇒ Array of size 4.

Time complexity O(1).

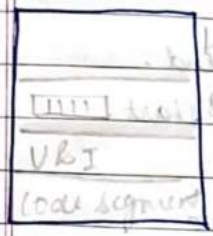
• Arrays as An abstract Data type in Data structure.

ADTs = set of values + set of operations.
int → 9, 10, 12 9+12=21
↑ operator.
but how the + sign works is hidden
∴ Details are abstracted.

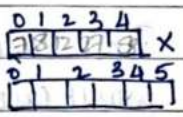
myArray → total_size. max() &.
used_size. + get(i)
base address set (i, num)
representation operation.

To initialize an array.

ptr [int * arr = new int [ ]
↳ Size of (int)



↳ dynamic memory
↳ static memory



Cannot extend

Have to create new arr.

int total_size;
int used_size;

Allocating memory of int size to an int pointer :-

int* ptr = (int*) malloc (sizeof (int));

QP
- *malloc function allocates a block of uninitialized memory to a pointer.
- *It is defined in <cstdlib>

• CODE [Implementation of array as an ADT]

1] Allocating memory and data in an array :-

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
    int* ptr = (int)* malloc (4 * sizeof (int));
    for (i=0 ; i<4; i++){
        ptr[i] = (i*8)-2;
    }
    for(i=0; i<4; i++){
        cout << ptr[i] <<endl;
    }
    free (ptr); → dellocated memory
    return 0;
}
```

output = -2
6
14
22

→ *(ptr+i)

PAGE No.
DATE / /

- operations on Array : Traversal, Insertion, Deletion and Searching.

Traversal :- visiting every element of an array once is known as Traversing the array.

Insertion :- Best case : O(1) Inserting at end.
worst case : O(n) Inserting at start.

Deletion :- Best case : O(1) Deleting at end.
worst case : O(n) Deleting at start.

Searching :- Best case : O(log n) Sorted array [Binary search]
worst case : O(n) unsorted array [Linear search]

- Code for insertion.

```
#include <iostream>
using namespace std;
int insertion (int arr[], int size, int element, int capacity, int index){
if (size >= capacity){
  return -1;
}
for (int i= size-1 ; i >= index ; i--){
    arr[i+1] = arr[i];
}
arr[index] = element;
return 1;
}
int main (){
 int arr[100] = {13, 23, 33, 43};
 int size = 4, element = 36, index = 3;
 insertion (arr, size, element, 100, index);
 size += 1;
for (int i=0; i<5; i++){
  cout << arr[i] << " "; } return 0; }.
```

Output : 13, 23, 33, 36, 43

PAGE No.
DATE / /

- **Code for deletion**

```
#include <iostream>
using namespace std;
int deletion (int arr [], int size, int index)
{
  for (int i = index ; i < size -1 ; i++){
      arr [i] = arr [i+1];
  }
  return 1;
}
int main (){
    int arr [100] = {13, 23, 33, 43};
    int size = 4, element = 36, index = 1;
    deletion (arr, size, index);
    size - = 1;
    for (int i = 0, i < 3; i++){
        cout << arr [i] << " ";
}
return 0;
}
```

Output : 13, 33, 43

- **Linear Search Vs Binary Search.**

Linear search ⇒ Traversing all elements one by one and finding the element.
⇒ Elements need not be sorted for this search.
⇒ time complexity = O(1).

Binary search ⇒ Diving the elements by its mid element and searching whether the given element is lower / higher than mid and repeating the process until the element is found.
⇒ elements need to be sorted.
⇒ time complexity = O(log n).

CODE FOR LINEAR SEARCH

```
#include <iostream>
using namespace std;
int linearSearch (int arr[], int size, int element){
  for (arr[i]==element int i=0; i<size; i++){
      if (arr[i]==element){
          return i;
      }
  }
  return 0;
}
int main(){
  int arr[] = { 5, 6, 7, 3545, 45, 642, 3245346, 2323, 5678789, 1
  int size = sizeof(arr)/sizeof(int);
  int element = 5678789;
  int searchIndex = LinearSearch(arr, size, element);
  cout << "The element " << element << "was found at index" <<
                                    searchIndex << endl;
  return 0;
  }
```

Output: The element 5678789 was found at index 8.

CODE FOR BINARY SEARCH

```
#include <iostream>
using namespace std;
int binarysearch (int arr[], int size, int element){
    int low, mid, high;
    low = 0;
  high = size-1;
  while (low <= high){
    mid = (low + high)/2;
    if (arr[mid]==element){
      return mid;
    }
    if (arr[mid] < element){
      low = mid+1;
    }
```

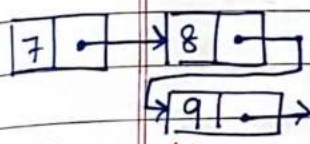
```
        else {
            high = mid - 1;
        }
    }
    return -1;
}
int main() {
    int arr[] = {23, 45, 67, 2, 12, 24, 90, 68, 4};
    int size = sizeof(arr) / sizeof(int);
    int element = 90;
    int SearchIndex = binarySearch(arr, size, element);
    cout << "The element " << element << " was found at index" <<
                                                SearchIndex;
    return 0;
}
```

Output: The element 90 was found at index 6.

• Introduction to Linked Lists

Linked List: Every element in a linked list is called a node and consists of two parts, the data part and the pointer part.

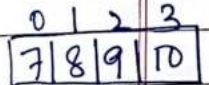
The data part stores the ~~data~~ value. While pointer stores the ~~pointer~~ address of next node of which it is pointing.



Linked List

Similarities: Both Linked List and arrays are linear data structure.

Differences: Arrays, capacity is fixed and insertion & deletion is difficult. Vice versa in Linked List. But in Linked list ~~and~~ random access ~~and~~ is not available and extra space is occupied by the pointer.



Arrays.

```
class node{
  public:
    int value;
    Node *next;
};
int main(){

}
```

→ Implementing Linked List.

```
struct Node{
  int data;
  struct Node * Next;
};
```

- Creation and Traversal.

```
struct Node{
  int data;
  struct Node *Next;
};
struct Node * head = Null;
void insert (int new_data){
  struct Node * new_node = (struct Node *) malloc (size of (struct Node));
  new_node → data = new data;
  new_node → next = head;
  head = new_node
}
void display(){
  struct Node * ptr;
  ptr = head;
  while (ptr != NULL){
    cout << ptr → data << " ";
    ptr = ptr → next;
  }
}
```

```
#include <iostream>
#include <cstdlib>
using namespace std;
class node{
public:
    int data;
    Node * next;
};
void printList (node* n){
    while (n != NULL){
        cout << n->data<< " ";
        n = n->next;
    }
}
int main(){
    node* head = NULL;
    node* second = NULL;
    node* third = NULL;
    head = new node();
    second = new node();
    third = new node();
    head -> data = 1;
    head -> next = second;
    second -> data = 2;
    second -> next = third;
    third -> data = 3;
    third -> next = NULL;
    printList (head);
    return 0;
```

Traversing and printing the list

initializing linked list with three nodes.

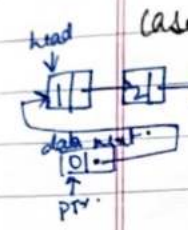
Allocating 3 nodes in the heap.

Assigning data and moving the pointer to point next node.

DATE / /

• Insertion in a linked List.

Case 1: Insert at the beginning

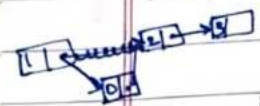



```
struct Node *ptr = (struct Node *) malloc (sizeof (struct node));
ptr -> next = head;
head = ptr;
return head;
```

Make the next of new node to point the old head and move the pointer to old head.

Time complexity = O(1). worst case only.

Case 2: Insert after head.




```
ptr -> next = p -> next;
p -> next = ptr;
```

Make p as head and move p pointer until it reaches the given index - 1 and let p's next = next P's position in the loop. then ptr->data = ptr -> data p->next becomes ptr's next ptr = p -> next. return head;

Time complexity = O(n).

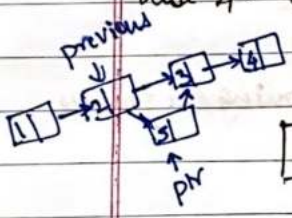
Case 3: Insert at the end


```
p->next = ptr;
ptr -> next = NULL;
```

while p = p -> next.
where p -> next is not Null.
then p -> next = ptr;
ptr -> next = NULL;
return head.

Time complexity = O(n).

Case 4: Insert after a node




```
ptr -> next = q -> next
q -> next = ptr.
```

given data = ptr's data.
previous nodes next = ptr's next.
ptr becomes previous node next

Time complexity = O(1).

↳ * take values * such as head, prevnode, data.

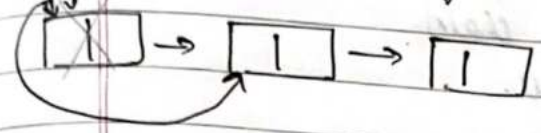
PAGE No.
DATE / /

• Deletion in a linked List.

both pointer.

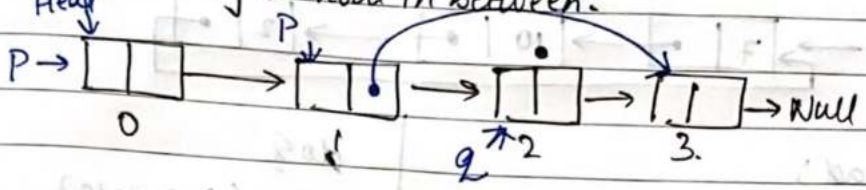
Head ptr

Case 1: Deleting the first node



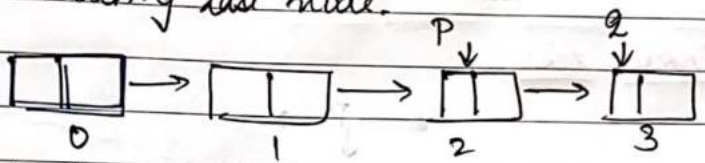
free ptr's memory.
head = head → next;

Case 2: Deleting a node in between.



p = head.
while (__ ind-1)
p = p → next;
}
initialize new pointer q where p → next
p → next = q → next.
free (q).

Case 3: Deleting last node.



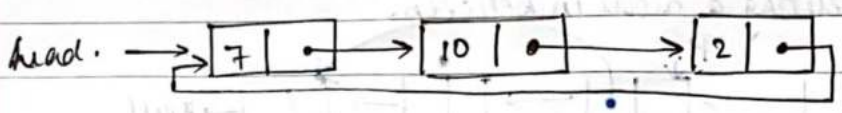
p → next = Null
free (q);

Case 4: Deleting a node with given value
[just the first occurance]

PAGE No.
DATE / /

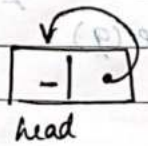
• Circular Linked List

→ A Linked List where the last element points to the first element (head) hence forming a circular chain.
→ There is no node pointing to null.



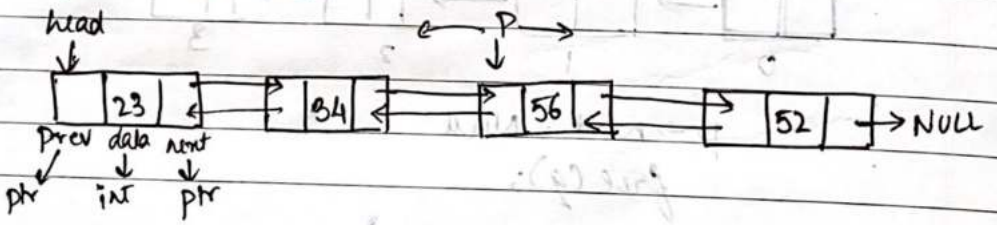

```
p = head;
while (p -> next != head) {
    print (p -> data);
    p = p -> next;
}
print (p -> data);
```

```
do {
print (p -> data)
    p = p -> next;
} while (p != head);
```



head is a node. } alternate representation.

• Doubly linked list



Each node contains a data part and two pointers in a doubly-linked list, one for the previous node and the other for the next node.

- SCLL Insertion / Deletion
- DLL Insertion

PAGE No.
DATE / /

★ Diffrence between singly LL and doubly LL:-

→ A doubly linked list allows traversal in both directions. We have the addresses of both the next node and the previous node.
→ A node comprises of three parts, the data, a pointer ~~and~~ to next node and a pointer to previous node.
→ Head node has the pointer to the previous node pointing to ~~NULL~~ NULL

IMPLEMENTATION:-

```
Struct Node {
    int data;
    Struct Node *next;
    Struct Node *prev;
};
```

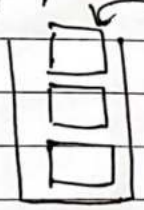
C programming

```
Class Node {
public:
    int data;
    Node *next;
    Node *prev;
};
```

CPP programming.

- Stack in Data structures.

→ A stack is a linear data structure.
→ Any operation performed is LIFO.



Insertion / Deletion can happen only in this end.

Applications of stack.

★ used in function calls.
★ Infix to postfix conversion (and other similar conversions).
★ parenthesis matching and more.

Stack ADT :

In order to create a stack we need a pointer to the topmost element along with other elements which are stored inside the stack.

Some stack operations of ADT are :-

1] push() → push an element into the stack.

2] pop() → remove the topmost element from the stack.

3] peek(index) → value at a given position is returned.

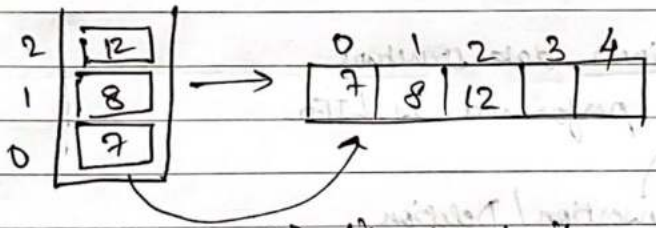
4] isempty() / isfull → Determine whether stack is empty or full.

Implementation.

* stack is a collection of elements with certain operations following LIFO (Last In First Out) discipline.

A stack can be implemented using an array or a linked list.

- ### Implementing stack using array.



O(1). time complexity

* fixed size array creation.
* Top element.

PAGE No.
DATE / /

Push, pop and other operations in stack implemented using an array.

→ Cannot push if full.
Cannot pop if empty.

PUSH OPERATION

1. create a defined stack. if we have creating stack and declaring its fundamentals part done, then pushing an element requires to first check if there's space left
2. call the isFull function which we did. If its full, then we cannot push anymore elements. This is stack overflow. Otherwise, increase the variable top by 1 and insert the element at the index top of the stack.

POP OPERATION

1. popping an element requires you to first check if there is any element left in the stack to pop.
2. call is Empty function. If its empty, then we cannot pop any element. Since its empty this is called stack underflow. Otherwise, store the topmost element in a temporary variable, Decrease the variable Top by 1 and return the temporary variable which stored the popped element.

PAGE No.
DATE / /

- Pseudo code for implementing stack and using array and its operations.

```
#include <iostream>
#include <cstdlib>
using namespace std;
#define MAX 1000
class stack {
    int top;
  public:
    int a[MAX];
  stack() { top = -1; }
    bool push(int x);
    int pop();
    int peek();
    bool isEmpty();
};
```

```
push =
    top >= MAX-1 ?
yes →
    cout << stack overflow
NO →
    a[++top] = x;
    cout << pushed;
    return true;
```

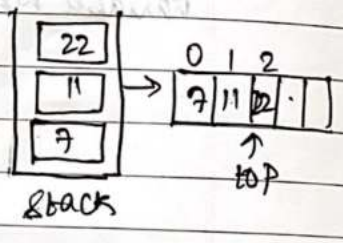
```
pop =
    top<0 ?
yes →
    cout << stack underflow
NO →
    int x = a[top--]
    cout << popped
    return x;
```

PEEK :- returns the value of the top ("front") of the collection without removing the element from the collection.

- Stack top and stack Bottom

Stack top :-

→ This operation is responsible for returning the topmost element in a stack, ~~Retrieving the topmost element~~ we just use the stack member top to fetch the topmost index and its corresponding element.



PAGE No.
DATE / /

Stack Bottom :-

→ Responsible for returning bottom most element in stack
→ Returns element at index 0.

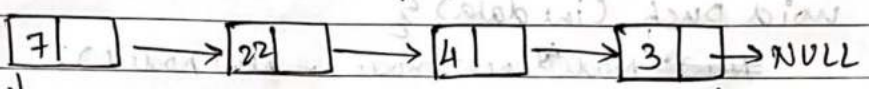
Both stackUp and stackBottom work in constant runtime i.e O(1)

Stack operations and time complexities :-

isEmpty() :- O(1)
isFull() :- O(1)
push() :- O(1)
pop() :- O(1).
peek() :- O(1).

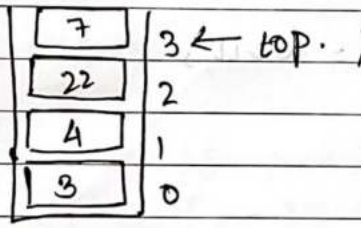
All have constant time complexities.

- **Stacks using linked Lists**



side 1

side 2.



the head becomes the top as the insertion/Deletion of a node at head happens to a function in constant time complexity O(1)

- Condition for stack to be full ⇒ when heap memory is exhausted.
- Condition for stack to be empthy ⇒ top == NULL.

Stack linked list has no upper limit to its size, but still we can to custom size to it.

- pseudocode for implementing stacks using linked list

```
#include <iostream>
using namespace std;
class Node {
  public
    int data;
    Node * next;
    }
Node * top;
(1) void push (int data) {
    ~~New~~ Node* new_node = new node();
        new_node -> data = data;
        new_node -> next = top;
            top = new_node;
(2) is Empty() {
        return top == NULL;
    }
(3) int peek() {
    if (! is Empty) {
        return top => data;
    }
    else {
        stack underflow. / exit (1);
    }
```

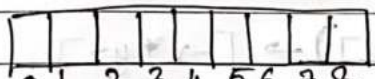
PAGE No.
DATE / /

④ void pop(){
Node* new_node;
if (top == NULL)
{
stack underflow;
}
else {
new_node = top;
top = top->next;
free (new_node);
}
}
void display(){

}

- parenthesis matching.

((3*2) - 1(8-2)) . ✓ parenthesis matching.

Eg- 

Expression :- 3*2-(8+1).

time complexity :- O(N)
[Both Best case and worst case]

(? → NO → if yes push into the stack.
)? → NO → if yes pop out of the stack.
↓
ignore

Condition for a balanced exp:-

① while popping, stack should not underflow ⇒ if it happens then unbalanced expression.

② At end of expression, stack must be empty ⇒ if it happens then unbalanced expression.

PAGE No.
DATE / /

Infix, prefix and postfix

Infix:- Here the operator comes in between two operands.

2+3, a*b, 6/3

<operand1> <operator> <operand2>

Prefix: Here the operator comes before the two operands.

+68, *xy, -32

<operator> <operand1> <operand2>

postfix:- Here the operator comes after the two operands.

57+, ab*, 126/

<operand1> <operand2> <operator>

Infix :- a* (b+c)*d.

postfix :- abc+*d*

Infix to prefix.

x-y*z.

(x-(y*z)) → (x-[*yz]) → [-x*yz].

Infix to postfix.

x-y*z

(x-(yz*)) → (x-[yz*]) → [xyz*-]

p-q-r/a = (p-q) -(r/a) = ([-pq] - [/ra]) = -- pq/ra.

(p-q)*(m-n) = ((p-q)*(m-n)) = ([pq-][mn-]) → pq-mn-

PAGE NO.
DATE / /

- Infix To postfix using stack

Infix	postfix	stack
x-y/z-k*a		
x-y/z-k*a	x	
x-y/z-k*a	xy	
x-y/z-k*a	xy/ ~~z*~~ xyz/-	expressions popped out because of division is greater than -
x-y/z-k*a	xyz/-k	
x-y/z-k*a	xyz/-ka*-	

✱ Multiplication, Division, remainder > addition, subtraction > relational operators

- Queue in Data structure.

First in first out.

- Queue ADT

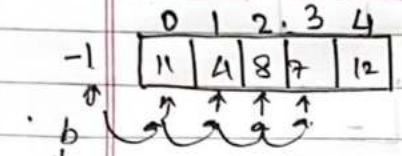
Data : ① Storage.
② Insertion end.
③ Deletion end.

Methods: ① enqueue. ④ last val.
② Dequeue. ⑤ peek (pos)
③ first val. ⑥ isEmpty.
⑦ isFull.

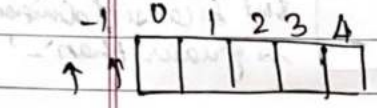
PAGE No.
DATE / /

→ Queue are not only Limited to counters.
→ Queue can be implemented in various ways.
① Arrays ② Linked Lists ③ other ADT's.

• Queue Implementation: Array Implementation of Queue.



Insertion :- Increament back end] O(1)
Insert at back end.
Deletion :- Remove element at index.] O(n)
shift all elements



Insert :- Increament b and insert at b.
Remove :- Increment f and Remove element
element at f.
first element → front index + 1
Rear element → Back index.
Queue empty → f = b.
Queue full → b = size - 1.

```
struct Queue {
    int size;
    int f;
    int r;
    int *arr;
};
```

```
int main {
    struct Queue q;
    q.size = 10;
    q.f = q.r = -1;
    q.arr = (int*) malloc (q.size * sizeof(int));
```

→ Void enque ( )

```
if (isfull q)) {
    cout << "Queue overfull" << endl;
}
else {
    q->r = q->r + 1;
    q->arr[q->r] = val;
```

→ int deque (Queue *q) {
int a = -1;
if (q→f == q→r).
cout << No element to deque << } is empty().

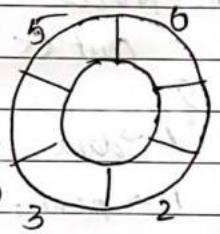
else {
q→f++
a = q→arr[q→f];
}
return a;
}

[if (q→r == q→size - 1)
return 1;] ↓ is full();

• Circular Queue.

Drawbacks of Queue using Arrays.
* Space is not used efficiently.

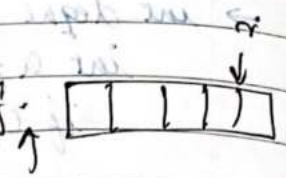
Circular increament
i = i+1; → Linear increament
* i = (i+1) % size. → Very Imp.



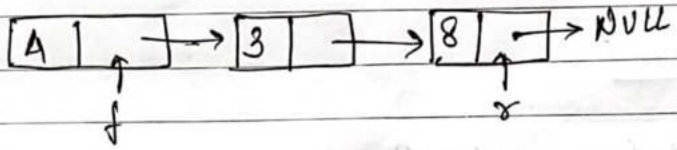
size = 5

void enqueue (struct Queue *q, int val)
if ((q→r+1) % q→size. == q→f.
cout << "Queue overflow" << endl;
else
q→r = (q→r+1) % q→size; → circular increament.
q→arr[q→r] = val;
}
}

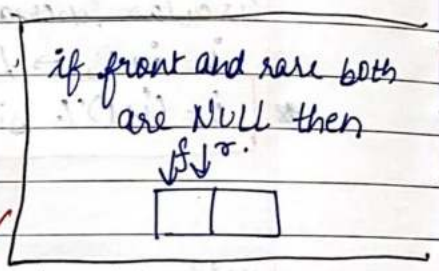
```
int dequeue (struct queue *q){
    int val = -1;
    if (q->r == q->f){
        cout << "Empty queue";
    else {
        q->f = (q->f + 1) % q->size;
        val = q->arr[q->f];
        }
    return val;
```



- Queue using linked lists



~~void enqueue (struct Queue *q, int val, struct N~~
~~struct N~~
```
void enqueue (N *f, N *r, int val).
N *n = (N*) malloc (sizeof (N));
if (n == NULL)
        cout << "Queue full" <<
else {
    n->data = val;
    n->next = NULL;
    ~~r->next = n;~~
    if (f == NULL){
        f = r = n;
    }
    else {
        r->next = n;
        r = n;
    }
}
```

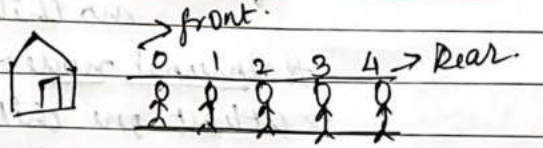


[A] Condition for Queue Empty
f == NULL
[B] Condition for Queue Full
r == NULL.

```
int dque (N *f, N *r) {
    int val = -1;
    N * ptr = f;
    if (q->r == q->f) {
        cout << "Empty queue";
    }
    else {
        f = f->next;
        val = ptr->data;
        free (ptr); }
    return val;
```

Dequeue

- Double ended Queue (DE Queue)



* insertion & Deletion can be done from front & Rear.
* FIFO NOT FOLLOWED.

DEQueue

Restricted Input
↓
Insertion from front not allowed

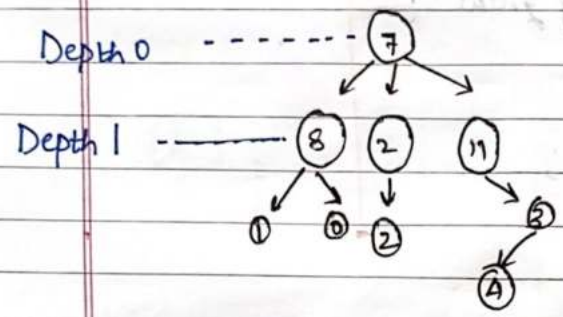
Restricted ~~DEQ~~ output.
↓
Deletion from rare not allowed.

PAGE No.
DATE / /

DEQueue ADT

1] Data → same as Queue.
2] Operations → isEmpty() enqueue(F)
isFull() dequeue(R)
initialize() enqueue(R).
print() dequeue(F)

• Tree Data structure.



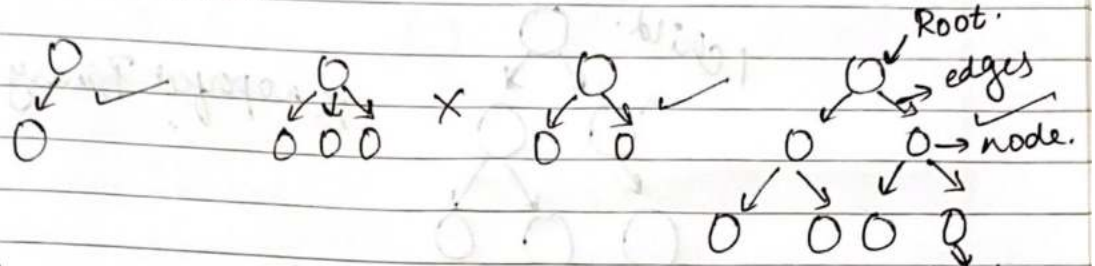
★ Root :- topmost node.
★ Parent :- Node which connects to the child.
★ Child :- Node which is connected by another node is its child.
★ Leaf / External node :- Node with no children.
★ Internal node :- Nodes with atleast one child.

★ Depth :- No. of edges from root to the node.
★ Height :- No. of edges from node to the deepest leaf.
★ Sibling :- Nodes belonging to the same parent.
★ Ancestors / Descendents.

PAGE No.
DATE / /

• Binary tree.

Binary tree is a tree which has atmost 2 children for all the nodes.



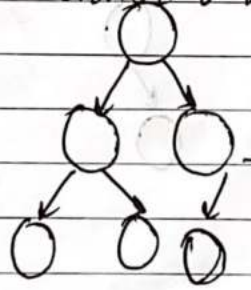
① Tree is made up of nodes and edges.
② n nodes has n-1 edges.
③ Degree ⇒ no. of direct children (for a node).
④ Degree of a tree is the highest degree of a node among all the nodes present in the tree.
⑤ Binary Tree = Tree of degree 2.
node can have 0, 1 or 2 childeren.

• Types of Binary Tree

classification
Tree → non linear → Ideal for representing hierarchial data.
Array, Stack, Queue, Linked List → Linear

① FULL / STRICT BINARY TREE.

* All nodes have either 0 or 2 children.

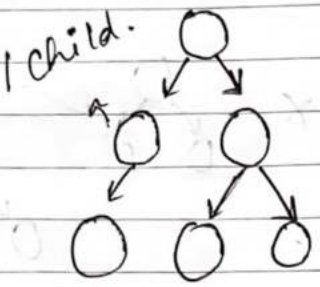


Hence, not a full/strict Binary tree.

PAGE No.
DATE / /

(2) PERFECT BINARY TREE.

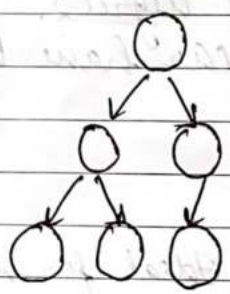
Internal nodes have 2 children + all leaf nodes are on same level.



× no perfect Binary tree

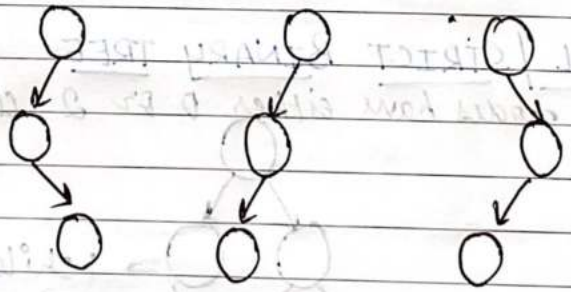
(3) COMPLETE BINARY TREE.

All levels are completely filled except possibly the last level + last level must have its keys as left as possible.



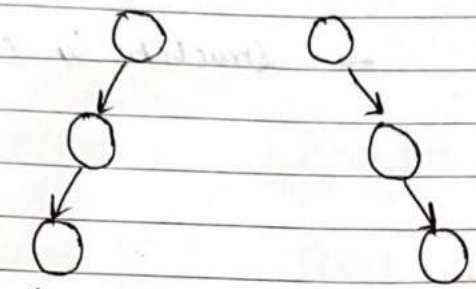
(4) DEGENERATE TREE

Every parent node has exactly one child.



PAGE No.
DATE / /

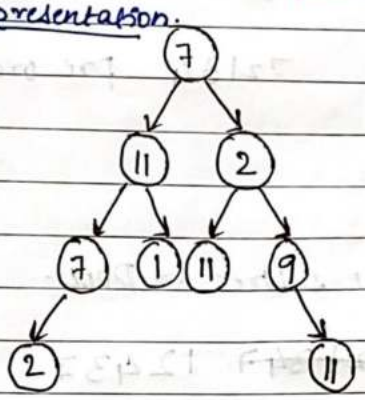
(5) ~~Skewed~~ SKEWED TREES



Left Skewed Right Skewed.

- ~~Array~~ ~~representation~~. Representation of a binary tree

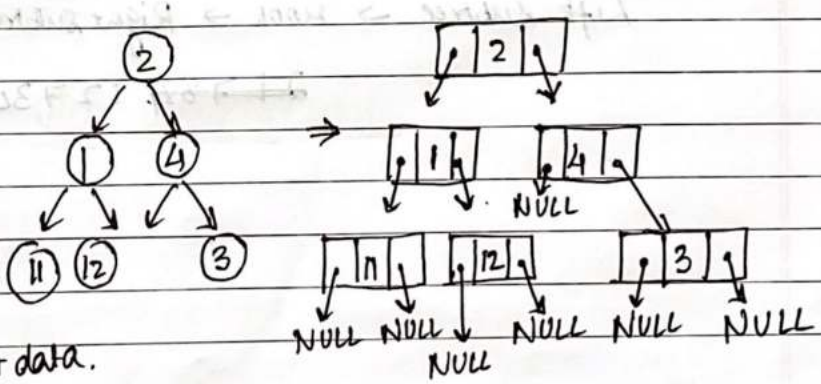
(1) Array Representation.

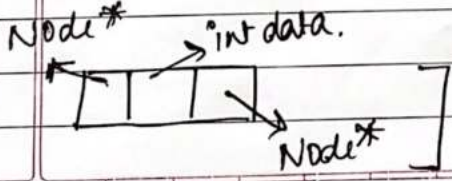


Size 15.
→ Array is needed to store this tree!

Array representation is inefficient.

(2) Linked Representation.





71 - 78 79 - 82 83 - 89

PAGE No.
DATE / /

```
struct node {
   int data;
   struct node * left;
   struct node * Right;
   };
```

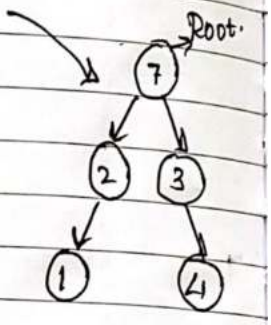
→ Structure in C language.

- Traversal in Binary Tree.

(1) PRE ORDER TRAVERSAL.

Root → left subtree → Right subtree.

12134 pre order traversal



(2) POST ORDER TRAVERSAL

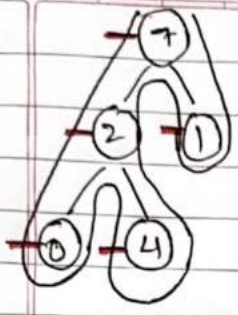
Left subtree → Right subtree → Root.

~~8 2 1 3 4 7~~. 12437

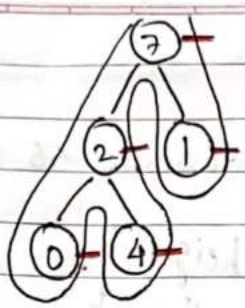
(3) INORDER TRAVERSAL.

Left subtree → root → Right subtree.

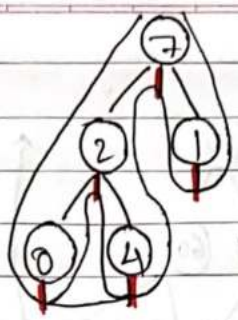
~~2 1 7 3 4~~ 12734



7 2 0 4 1
PRE ORDER



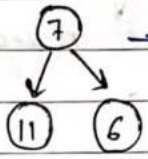
0 4 2 7 1
POST ORDER



0 2 4 7 1
IN ORDER

- Binary Search Tree

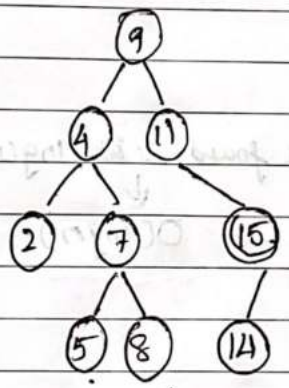
It is a type of Binary tree



→ NOT A BINARY SEARCH TREE

Properties of a BST

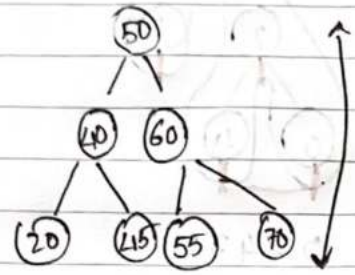
1. All nodes of the left subtree are lesser.
2. All nodes of the right subtree are greater.
3. Left and Right subtrees are also BST.
4. There are no duplicate nodes.
5. In order traversal of a BST gives an ascending sorted array.



→ IS A BINARY SEARCH TREE

PAGE No.
DATE / /

- Searching in Binary search tree



→ Binary search tree is efficient in searching elements.
→ If in array we need to search ~~for all the ele~~ the whole array i.e n searches but in BST we only need to do $n/2$ searches.

For above example :
Is 55 > 50 ? → yes.
Search in ~~Left~~ Right subtree.
Is 55 > 60 ? → NO.
Search in Left subtree.
55 found ? → yes.

* Best case :- if ~~the~~ element to be found is at log(n) height.
* Worst case :- if element to be found is at ~~dept~~ last step.
O(n).

↓
O(log n).

PAGE No.
DATE / /

C code for searching in a BST =

```
Node * search (Node * root, int key){
    if (root == NULL).
        return NULL;
    if (root → data == key)
        return root;
    else if (root → data > key).
        return search (root → left, key).
    else (root → data < key).
    return search (root → right, key);
    }.
```

Recursive Method.

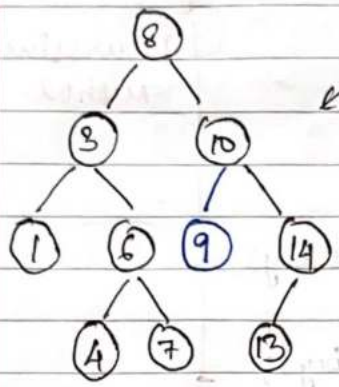
```
Node * search (Node * root, int key){
  while (root != NULL){
      if (key == root → data){
        return root;
      }
      else if (key < root → data){
          root = root → left;
      }
      else {
          root = root → right;
      }
  }
  return NULL;
}.
```

Iterative method.

PAGE No.
DATE / /

• Insertion in Binary Search tree

BST → NO Duplicates allowed.



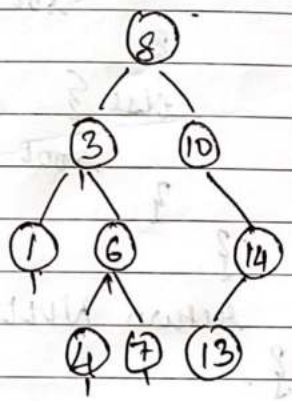

```
Void incert (Node *root, int key){
  Node* prev = NULL;
  Node * ptr ;
  while (root != NULL){
      prev = root ;
  if (key == root → data)
      return ;
  else if (key < ~~x~~ → ~~do~~ root → data)
  return (root = root → left);
    else
      return (x = x → right);
  }.
```

• Delution in Binary Search tree

★ Case: 1 – The node is a leaf node :-

Step1 :- Search the node and delete the node.

★ Case : 2 – The node is a non leaf node :-



Step1 :- Search the node.

Step2 :- Replace that element with its inorder pre or ~~po~~ inorder post.

1 3 4 6 7 → Inorder ~~tr~~.

PAGE No.
DATE / /

★ Case: 3– The node is the root node.

Step 1 :- Search the node. removing 8

Step 2 :- Do inorder traversal

1 3 4 6 ~~7~~ ~~8~~ ~~10~~ 13 14

Step 3 :- Either replace 8 by 7 or 10.

Step 4 :- if replaced by 10 then replace the previous place of 10. by ~~pre~~ inorder pre of 10 or inorder post of 10.

But in this case 8 has already been deleted. Hence 13 would now be the root node.

• AVL Trees. [AVL - Adelson - Velskii and Landis]

What is an AVL Tree?

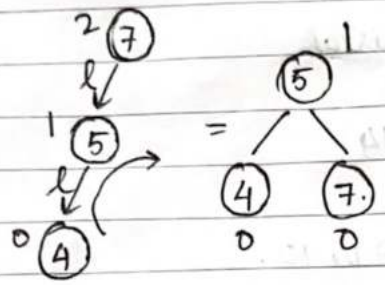
- Height ~~bla~~ balanced binary search trees ?.
- Height difference between heights of left and right subtrees for every node is less than or equal to 1.
- Balanced factor : Height of right subtree - Height of left subtree.
- Can be -1, 0 or 1 for a node to be balanced in a Binary search tree.
- Can be -1, 0 or 1 for all nodes of an AVL trees.

For a Balanced tree $|BF| \leq 1$

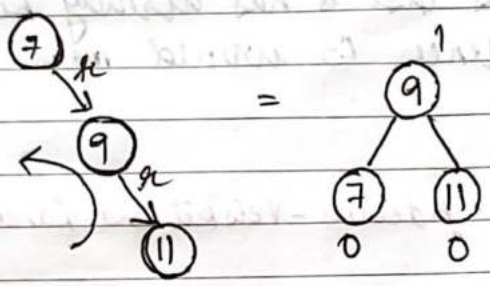
PAGE No.
DATE / /

- Insertion and Rotation in AVL Trees.

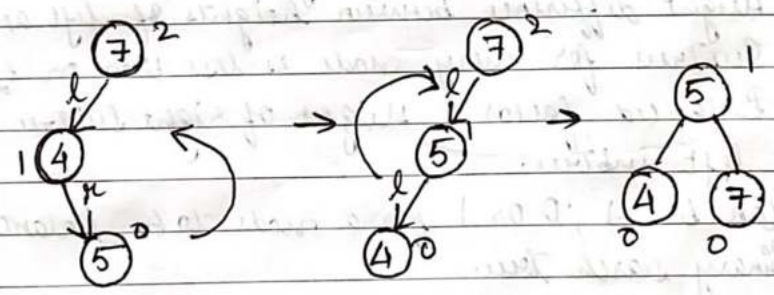
→ LL ROTATION.



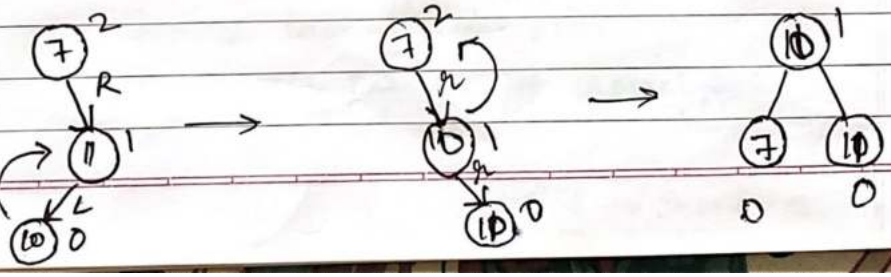
→ RR ROTATION



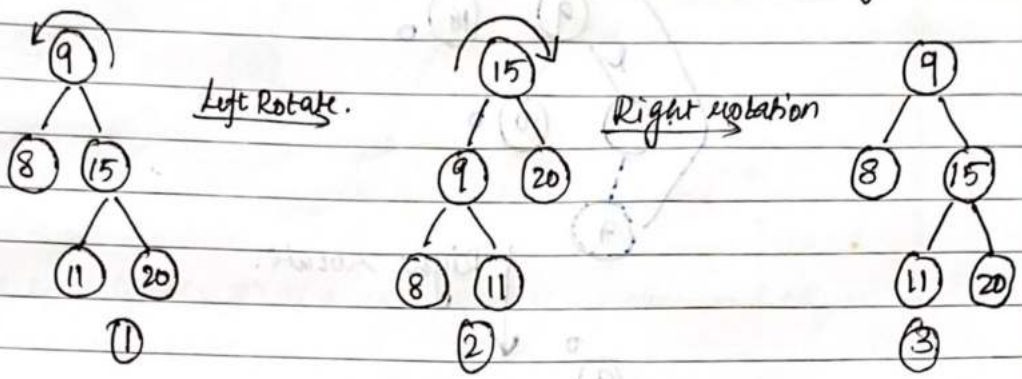
→ LR ROTATION



→ RL ROTATION



→ Left rotation wrt a node – Node is moved towards the left.
→ ~~Rotati~~ Right rotation wrt a node – Node is moved towards the right.



(1) & (3) are the same trees.

Balancing a AVL Tree after insertion.

In order to balance an AVL tree after insertion, we can follow the following rules:-

(1) For a ~~right~~ Left - Left insertion –
Right Rotate once wrt the first imbalanced node.

(2) For a right - right insertion –
Left rotate once wrt the first imbalanced node.

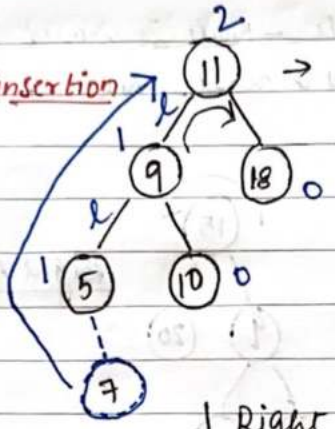
(3) For a left - right insertion –
Left rotate once and then right rotate once.

(4) For a right - Left insertion –
Right rotate once and then left rotate once.

first rotation is made with respect to child of first imbalanced node

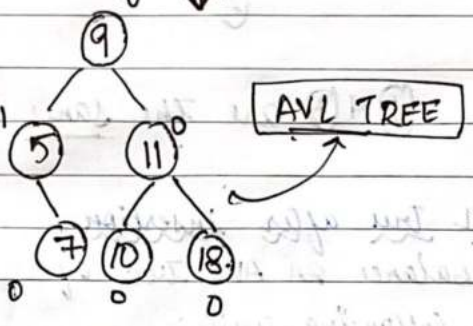
PAGE No.
DATE / /

* Left - Left insertion



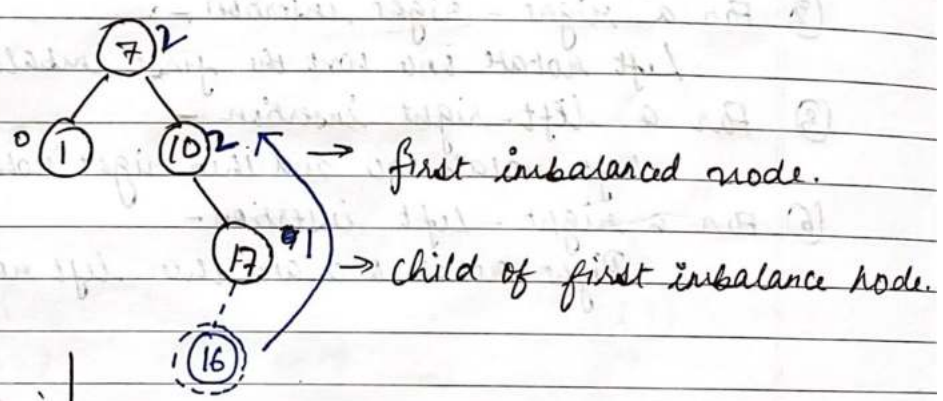
→ first imbalanced node

Right rotate.



AVL TREE

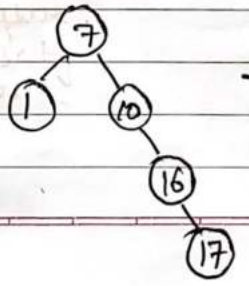
* Right - ~~Right~~ Left insertion.



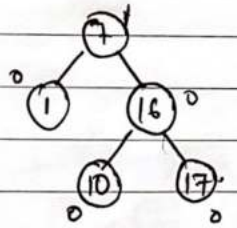
→ first imbalanced node.

→ child of first imbalance node.

Right rotate

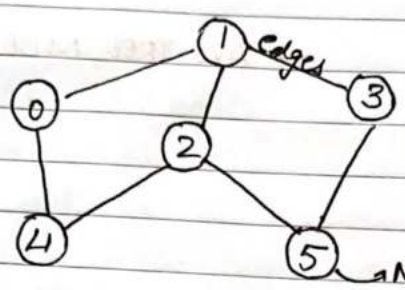


Left rotate.



PAGE No.
DATE / /

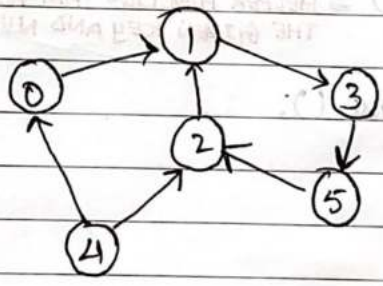
- Graph data structures.



vertices = {0, 1, 2, 3, 4, 5}
edges = {(0,1); (1,3); (1,2) ... }.

→ A graph G = (V,E) is a collection of vertices and edges connecting these vertices.
→ used to model paths in a city, social networks, website backlinks, internal employee network, etc.
→ A vertex / node is one fundamental unit / entity of which graphs are formed.
→ An edge is uniquely defined by its 2 endpoints.
→ directed edge - one way connection.
→ undirected edge - two way connection.
→ Directed graph - All directed edges.
→ Undirected graph - All undirected edges

Indegree and outdegree of a node.



Indegree - No. of edges going out of the node.
Out degree - No. of edges coming into the node.

PAGE No.
DATE / /

• Implementation of AVL Tree.

```
class node                                  → AVL TREE NODE
{
 public:
  int key;
  Node * left;
  Node * right;
  int height;
};
int max (int a, int b);  → A UTILITY FUNCTION TO GET MAX OF 2 INTEGERS
int height (Node * N)    → A UTILITY FUNCTION TO GET THE HEIGHT OF THE TREE
{
  if (N == NULL).
    return 0;
  return N → height;
}
int max (int a, int b) → A UTILITY FUNCTION TO GET MAX OF 2 INT
{
  return (a>b)? a:b;
}
Node * newNode (int key) → HELPER FUNCTION THAT ALLOCATES A NEW NODE WITH THE GIVEN KEY AND NULL LEFT AND RIGHT POINTER
{
  Node * node = new Node ();
  node → key = key;
  node → left = NULL;
  node → right = NULL;
  node → height = 1; → NEW NODE IS INITIALLY ADDED AT LEAF.
  return (node);
}
```

PAGE No.
DATE / /

```
Node * rightRotate (Node *y).   → AUTILITY FUNCTION TO RIGHT ROTATE SUBTREE ROOTED WITH y.
{
  Node *x = y->left;
  Node *T2 = x->right;
  x->right = y;                 → PERFORM ROTATION
  y->left = T2;
  y->height = max(height(y->left), height(y->right))+1;   ] → UPDATE HEIGH
  x->height = max(height(x->left), height(x->right))+1;
  return x;                     → RETURN NEW ROOT.
}
Node * LeftRotate (Node *x)     → AUTILITY FUNCTION TO LEFT ROTATE SUBTREE ROOTED WITH x.
{
  Node *y = x->right;
  Node *T2 = y->left;
  y->left = x;                  → PERFORN ROTATION
  x->right = T2;
  x->height = max(height(x->left), height(x->right))+1;   ] → UPDAT HEIGH
  y->height = max(height(y->left), height(y->right))+1;
  return y;                     → RETURN NEW ROOT
}
int getBalance (Node *N).       → GETTING BALANCE FACTOR OF A NODE
{
  if (N == NULL)
    return 0;
  return height(N->left) - height(N->right);
}
Node * insert (Node *node, int key)   → RECURSIVE FUNCTION TO INSERT A KEY IN THE SUBTREE ROOTED WITH NODE AND RETURNS THE NEW ROOT OF THE SUBTREE.
{
  if (node == NULL)             → PERFORMING NORMAL BST INSERTION
    return (newNode(key));
  if (key < node->key)
    node->left = insert(node->left, key);
  else if (key > node->key)
    node->right = insert(node->left, key);
  else return node;             → EQUAL KEYS ARE NOT ALLOWED IN BST
```

```
node -> height = 1 + max ( height (node -> left), height (node -> right));
```
↳ UPDATE HEIGHT OF THIS ANCESTOR NODE

```
int balance = getBalance (Node);
```
→ GET THE BALANCE FACTOR OF THIS ANCESTOR NODE TO CHECK WHETHER THIS NODE BECAME UNBALANCED

If this node becomes unbalanced, then there are 4 cases.

```
if (balance > 1 && key < node -> left -> key)
    return rightRotate (node);
```
→ LEFT LEFT CASE

```
if (balance < -1 && key > node -> right -> key)
    return leftRotate (node);
```
→ RIGHT RIGHT CASE

```
if (balance > 1 && key > node -> left -> key)
{
    node -> left = leftRotate (node -> left);
    return rightRotate (node);
}
```
→ LEFT RIGHT CASE

```
if (balance < -1 && key < node -> right -> key)
{
    node -> right = rightRotate (node -> right);
    return leftRotate (node);
}
```
→ RIGHT LEFT CASE

```
return node;
```
→ RETURN UNCHANGED NODE POINTER.

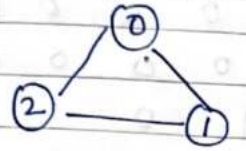
```
}
```

```
void preOrder (Node *root)
{
    if (root != NULL)
    {
        cout << root -> key << " ";
        preOrder (root -> left);
        preOrder (root -> right);
    }
}
```
→ UTILITY FUNCTION TO PRINT PREORDER TRAVERSAL OF THE TREE. THE FUNCTION ALSO PRINTS HEIGHT OF EVERY NODE.

PAGE No.
DATE / /

- **Representation of graphs - Adjacency List, Adjacency Matrix & Other**

→ Adjacency list - Mark the nodes with the list of its neighbors.
→ Adjacency matrix - $A_{ij} = 1$ for an edge between i and j, 0 otherwise



0 → 1 → 2
1 → 0 → 2
2 → 0 → 1

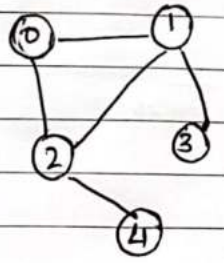
Adjacency List.

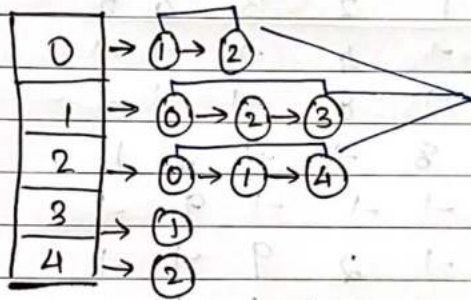
	0	1	2
0	0	1	1
1	1	0	1
2	1	1	0

→ Adjacency Matrix.

→ Edge set - store the pair of nodes / vertices connected with an edge. Eg - {(0,1), (1,2), (0,2)}.
→ Other implementations to represent a graph also exists. For eg. compact list, cost adjacency list, cost adjacency matrix

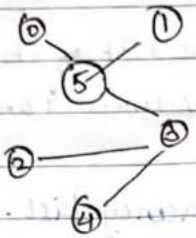
* ADJACENCY LIST





pointer of 0 will store head as 1 and makes a trailing LL.
pointer of 1 will store head as 0. and makes a trailing list.

★ ADJACENCY MATRIX



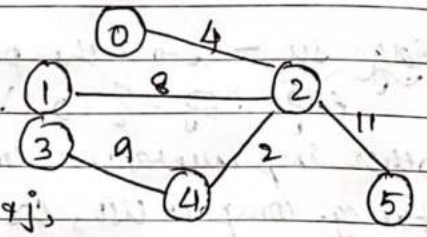
	0	1	2	3	4	5
0	0	0	0	0	0	1
1	0	0	0	0	0	1
2	0	0	0	0	0	0
3	0	0	1	0	1	1
4	0	0	0	1	0	0
5	1	1	0	1	0	0

A_{ij} = 1 for an edge between i and j,
0 otherwise.

★ COST ADJACENCY MATRIX

→ A_{ij} = cost for an edge between i and j, 0 otherwise.

→ If the cost can be 0:
A_{ij} = cost for an edge between i & j,
-1 otherwise



	0	1	2	3	4	5
0	-1	-1	4	-1	-1	-1
1	-1	-1	8	-1	-1	-1
2	4	8	-1	-1	2	11
3	-1	-1	-1	-1	9	-1
4	-1	-1	2	9	-1	-1
5	-1	-1	11	-1	-1	-1

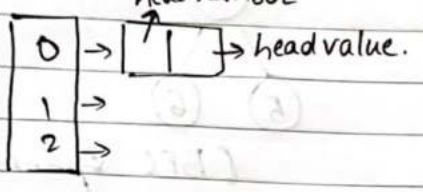
PAGE No.
DATE / /

* EDGE SET

Store the pair of nodes/vertices connected with an edge eg - {(0,1), (0,4), (1,4)}

* COST ADJACENCY LIST

Cost is also stored along with the links.



* COMPACT LIST REPRESENTATION:

Entire graph is stored in 1D array.

• Graph traversal and graph traversal algorithm

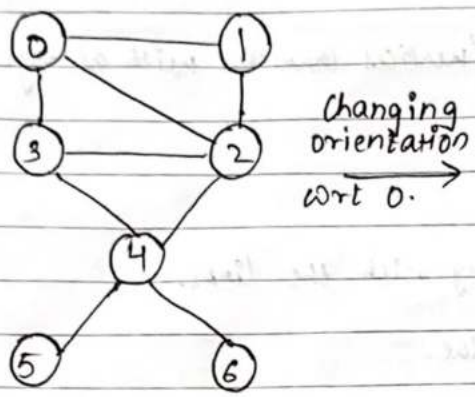
→ Graph traversal refers to the process of visiting (checking and/or updating) each vertex (node) in a graph.

→ Two algorithms which can be used.
(1) Breadth first search (BFS) [Queue is used]
(2) Depth first search (DFS). [Stack is used].

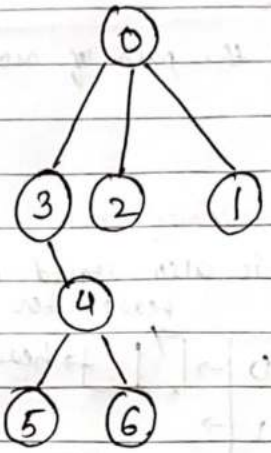
• Breadth first Search.

→ In BFS, we start with a node and start exploring its connected nodes. The same process is repeated with all the connecting nodes until all nodes are visited.

Method 1:-



Changing orientation wrt 0.

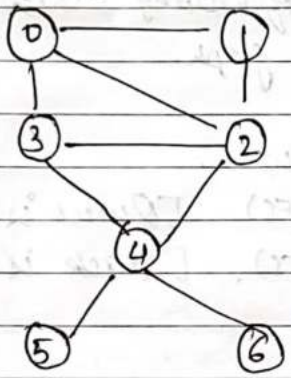


(BFS Spanning Tree).

LOT - Level order Traversal (Same level nodes from left to right).

[0 3 2 1 4 5 6] → BFS.

Method 2:-



BFS.

V → visited: 0 1 2 3 4 5 6

EQ → Exploration Queue: ~~0 X Z 3 4 5 6~~

Explored :- the connected nodes are visited.

PAGE No.
DATE / /

ALGORITHM : BFS.

Input : A graph $G = (V, E)$ and source node in V.

Algorithm:

```
Mark all nodes v in V as visited.
Mark source node s as visited
Enqueue (Q, s) // FIFO (Q)
while (Q is not empty)
    {
        v := deq(Q);
        for each unvisited neighbour v of v{
            mark v as visited;
            enqueue (Q, v);
        }
    }.
```

→ We can start with any vertex.
→ There can be multiple BFS results for a given graph.
→ Order of visiting vertices ~~may~~ can be anything.

- Implementation of Adjacency list and BFS

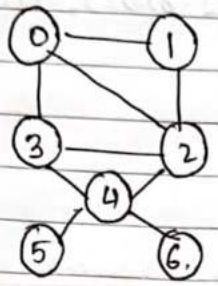
```
class graph {                    → This class represents a directed
                                   graph using adjacency list representation
  int V;  → No. of vertices.
  vector <list <int>> adj;  → pointer to an array containing adjacency lists.
  public:
    graph (int V);  → constructor.
    void ~~edge~~ addedge (int v, int w);  → func. to add an edge to graph.
    void BFS (int s);  → prints BFS traversal from s source.
};
```

PAGE No.
DATE / /

```
Graph : graph (int v)
{
   this -> V = V;
   adj.resize (V);
}
void graph:: addEdge (int v, int w).
  {
    adj [v]. push_back (w); → Add w to v's list.
  }
void graph:: BFS (int s)
  {
    vector <bool> visited; → All vertices not visited yet.
    visited. resize (V, false);
    list <int> queue; → Creating a queue for BFS.
    visited [s] = true; → mark current node as visited & enqueue it.
    queue. push_back (s);
    while (! queue. empty ()).
     {
       s = queue. front (); → Dequeue a vertex from queue and print it.
       cout << s << " ";
       queue. pop_front ();
       for (auto adjacent : adj [s]) → Get all adjacent vertices of the dequeued vertex s. If a adjacent has not been visited, then mark it visited & enqueue it.
       {
         if (! visited [adjecent]).
           {
             visited [adjacent] = true;
             queue. push_back (adjecent);
           }
       }
  }
}
```

PAGE No.
DATE / /

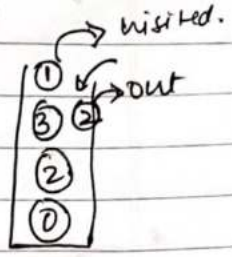
- Depth first search.



V:- 0 1 2 3 4 5 6 2

0, 1, 2, 4, 6, 5, 3

DFS Traversal.



(1) Start by putting any one of the graph's vertices on top of a stack.
(2) Take the top item of the stack and add it to the visited list.
(3) Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
(4) Keep repeating until the stack is empty.

* ALGORITHM : DFS.

```
DFS (G, v)
v.visited = true.
  for each v ∈ G.adj[v]
    if v.visited == false.
      DFS (G, v)
init () {
    for each v ∈ G
      v.visited = false.
    For each v ∈ G
    DFS (G, v).
```

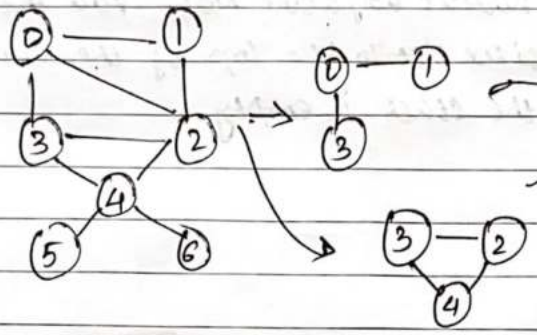
- Implementation of DFS

```
void Graph :: DFS (int v)
{
  visited [v] = true;   → Mark the current node not visited and print it
  cout << v << " ";
```

```
list <int> :: iterator i;  → Recur for all the vertices adjacent to this vertex.
for (i = adj[v].begin(); i != adj[v].end(); ++i).
    if (!visited[*i])
        DFS(*i);
}
```

- **Spanning Trees and maximum no. of possible spanning trees**

→ A subgraph of a graph G is a Graph whose vertices and edges are subsets of original graph G.



→ A connected graph is a graph that is connected in the sense of a Topological space i.e, there is a path from any point to any other point in the graph. A graph that is not connected is called disconnected.

→ A complete graph is a simple undirected graph in which every pair of distinct vertices in connected by a unique edge

→ A connected subgraph 'S' of graph G(V,E) is said to be a spanning tree of graph G if (if and only if):

1. All vertices of G must be present in S.
2. No of edges S should be V-1
3. Cannot be cyclic.

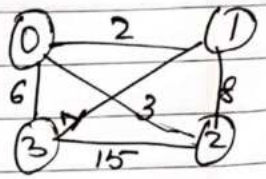
→ A complete graph has n^{n-2} spanning trees where n is the number of vertices in the graph.

Minimum spanning tree = [MST]
which takes the least cost.

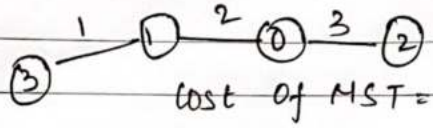
• prim's minimum spanning Tree algorithm.

→ prim's algorithm uses greedy approach to find the MST.

→ We start with any node and start creating a MST



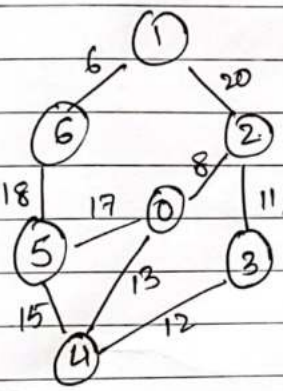
→ In prim's algorithm we grow the spanning tree from a starting position until n-1 edges are formed (or n nodes are covered).



cost of MST = 1+2+3 = 6.

MST of above graph.

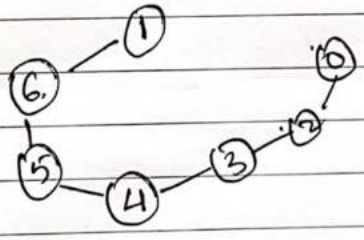
eg-



Not included MST
→ V = {0, 2, 3, 4, 5, 6} } initially.
A = {1}
Included in MST

options?
1 → 2 = 20 ✗
1 → 6 = 6 ✓

MST Tree making example.



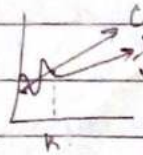
cost.
= 6+18+15+12+11+8
= 70

V = {0, 2, 3, 4, 5} A = {1, 6}
V = {0, 2, 3, 4} A = {1, 5, 6}.
V = {0, 2, 3} A = {1, 4, 5, 6}
V = {0, 2} A = {1, 3, 4 5 6}
V = {0} A = {1 2 3 4 5 6
V = { } A = {0, 1, 2, 3, 4 5 6}

Asymptotic notation, functions and running times.

* Asymptotically positive function is one that is positive for all sufficiently large n. $\Theta(g(n))$ → asymptotically non negative.
* O notation - asymptotic upper bound. - Worst case.
* Ω notation - asymptotic lower bound. - Best case.

→ O notation - $f(n) = O(g(n))$



$f(n) \leq c \cdot g(n)$

$c > 0$

$n \geq k$

$k \geq 0$.

$f(n) = O(\quad)$

$f(n) = 2n^2 + n$

$2n^2 + n \leq c \cdot g(n)$.

$2n^2 + n \leq c\, g(n^2)$

↳ lower upperbound

$2n^2 + n \leq 3n^2$

↳ (if 2 then +n would not be it impossible).

$n \leq n^2$.

$1 \leq n$

→ upper lower bound only.

→ Big omega (Ω).

$f(n) = \Omega\, g(n)$.

$f(n) \geq c \cdot g(n)$.

$2n^2 + n \geq c \cdot g(n)$.

$2n^2 + n \geq c \cdot n^2$.

$2n^2 + n \geq 2n^2$.

$n \geq 0$.

→ theta

→ little o $f(n) < c \cdot g(n)$

→ little Ω $f(n) > cgn$

		Reflexive	Symmetric	Transi
Big (O)	$f(n) \leq c \cdot g(n)$	✓	×	✓
Big (Ω)	$f(n) \geq c \cdot g(n)$	✓	×	✓
theta (Θ)	$c_1 g(n) \leq f(n) \leq c_2 g(n)$	✓	✓	✓
Small (o)	$f(n) < c \cdot g(n)$	×	×	✓
Small (ω)	$f(n) > c \cdot g(n)$	×	×	✓

ANALYSIS OF ALGORITHM

MODULE 1:

→ Data structure is a way to store and organize data in order to facilitate access and modifications.

Insertion sort - n^2 time to sort n items
Merge sort - $n \log n$ time to sort n items
($\log n$ = $\log_2 n$)

Instructions

* Comparisons of various time complexities.

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

Binary search - $\log_2 n$
Sequential search - $O(n)$
Quick sort - $O(n \log n)$
Merge sort - $O(n \log n)$.
Insertion sort - $O(n^2)$
Bubble sort - $O(n^2)$
Heap sort - $O(n \log n)$
Selection sort - $O(n^2)$

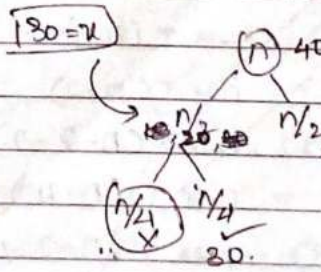
Binary Search.

```
Bs(a,i,j,x).
  mid = (i+j)/2
  if (a[mid] == x)
   return (mid);
  else
    if (a[mid] > x)
    Bs(a, i, mid-1, x).
    else
      Bs(a, mid+1, I, x)
```

Recurrance Relation

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

10, 20, 30, (40), 50, 60, 70
I — mid — I



Substitution Method

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

$T(n) = T(n/2) + C$ ①

$T(n/2) = T(n/4) + C$ ②

$T(n/4) = T(n/8) + C$

$T(n) = T(n/4) + C + C$

$T(n/2^2) + 2C$ — A.

$T(n/8) + 3C$

$T(n/2^3) + 3C$ — B

⋮

k steps

$n = 2^k$

$T\left(\frac{n}{2^k}\right) + kC$

$T(n/n) + kC$

$T(1) + kC$.

$1 + kC$.

but time complexity should be in terms of n.

$\log n = \log 2^k$

$\log n = k \log 2 \Rightarrow \log n = k$.

$\therefore 1 + \log n C$

$= \boxed{O(\log_2 n)}$ → time complexity of Binary search.

2] $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ n * T(n-1) & \text{if } n > 1 \end{cases}$

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

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

$= n-1 * T(n-2)$ — ①

$T(n-2) = n-2 * T(n-2-2)$

$= n-2 * T(n-4)$ — ②.

$T(n-3) = n-3 * T(n-3-1)$

$n-3 * T(n-4)$ — ③

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

$T(n-1) = n-1 * T(n-1-1)$ — ①

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

$T(n-2) = n-2 * T(n-3)$ — ②

$T(n-3) = n-3 * T(n-4)$. — ③

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

$= n * (n-1) * n-2 * T(n-3) \cdots$

$= n * n\left(1-\frac{1}{n}\right) * \left(1-\frac{2}{n}\right) \cdots$ (steps)

$= n \cdot n \cdot n \cdot n = n^n$

$n\left(\frac{1}{n}\right) = T(1)$

$T(n-(n-1))$

3] $T(n) = \begin{cases} 1 & \text{if } n=1 \\ 2T(n/2)+n \text{ -(1)} & \text{o/w} \end{cases}$

$T(n/2) = 2T\left(\frac{n}{4}\right)+\frac{n}{2}$ -(2)

$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right)+\frac{n}{4}$ -(3)

$T(n) = 2\left[2T\left(\frac{n}{4}\right)+\frac{n}{2}\right]+n$

$2^2 T\left(\frac{n}{4}\right)+n+n$

$= 2^2 T\left(\frac{n}{4}\right)+2n$

$= 2^2\left[2T\left(\frac{n}{8}\right)+\frac{n}{4}\right]+2n$

$= 2^3 T\left(\frac{n}{8}\right)+3n$

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

$= 2^4 T\left(\frac{n}{2^4}\right)+4n$ .... k times.

$T(n) = 2\times 2T\left(\frac{n}{4}\right)+\frac{n}{2}+n$

$= 2\times 2\times 2T\left(\frac{n}{8}\right)+\frac{n}{4}+\frac{n}{2}+n$

$= 2\times 2\times 2\times 2T\left(\frac{n}{16}\right)+\frac{n}{8}+\frac{n}{4}+\frac{n}{2}+n$

$= 2T\left(\frac{n}{4}\right)+n$

$= 2\left[2T\left(\frac{n}{8}\right)+\frac{n}{4}\right]+n$

$= 2^k T\left(\frac{n}{2^k}\right)+kn$

$2^k T(1)+n\log n$

$n(1)+n\log n$

$n+n\log n$

$O(n\log n)$ ↓ Time complexity.

$\frac{n}{2^k}=1$

$n=2^k$

$\log n = \log 2^k$

$\log n = k\log 2$ (log 2 = 1)

$\log n = k$

4] $T(n) = \begin{cases} 1 & \text{if } n=1 \\ T(n-1)+\log n, & \text{o/w} \end{cases}$

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

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

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

$= T(n-3)+\log(n-2)$

$T(n) = T(n-2)+\log(n-1)+\log n$.

$= T(n-3)+\log(n-2)+\log(n-1)+\log n$ .... k times.

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

$\log m+\log n = \log mn$

$\log(n)$

$n-k=1$

$n=k$

$1+\log(n-(n-1))+\log(n-(n-2))\cdots$

$= 1+\log(1)+\log(2)\cdots\log(n)$

$= 1+\log(1\cdot 2\cdot 3\cdots n)$

$n\times(n-1)(n-2)\cdots$

$n\times n\times n\cdots$

$= 1+\log(n!) = 1+\log n^n \Rightarrow O[n\log n]$

Master theorem

Substitution — slower but all problems can be solved.
Master — faster but few problems only can be solved.

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

$[a \geq 1]$, $[b > 1]$

$T(n) = n^{\log_b a}\,[U(n)]$ $\quad U(n)$ depends on $f(n)$

$H(n) = \dfrac{f(n)}{n^{\log_b a}}$.

Relation between $h(n)$ & $U(n)$ →

$h(n)$	$U(n)$
n^r, $r>0$	$\Theta(n^r)$
n^r, $r<0$	$O(1)$
$(\log n)^i$, $i \geq 0$	$\dfrac{(\log_2 n)^{i+1}}{i+1}$

1] $T(n) = 8T(n/2) + n^2$.

$a = 8 \quad b = 2 \quad f(n) = n^2$.

$T(n) = n^{\log_b a}$.

$= n^{\log_2 8}\, U[n]$

$= n^3\, U[n]$. $\qquad n^3 (1) = O(n^3)$

$h(n) = \dfrac{f(n)}{n^{\log_b a}} = \dfrac{n^2}{n^3} = \dfrac{1}{n} = n^{-1} \rightarrow n^r\ r<0 \Rightarrow O(1)$

2] $T(n) = T(n/2) + c$

$a = 1 \quad b = 2 \quad f(n) = c$.

$T(n) = n^{\log_b a}$.

$= n^{\log_2 1}\, U[n]$.

$= n^0\, U[n] = n^0 (\log_2 n) = \boxed{O[(\log_2 n)]}$

$h(n) = \dfrac{f(n)}{n^{\log_b a}} = \dfrac{c}{n^0} = \dfrac{c}{1}$ $\qquad \dfrac{(\log_2 n)^0 \cdot c}{1} = \dfrac{(\log_2 n)^1 \cdot c}{1}$

$= \log_2 n \cdot c$.

$= O(\log_2 n)$.

3] $T(n) = \begin{cases} T(\sqrt{n}) + \log n & \text{if } n \geq 2 \\ O(1) & \text{o/w} \end{cases}$

Master Method :- $T(n) = aT\left(\frac{n}{b}\right) + n^k \log^p n$

$T(n) = T(\sqrt{n}) + \log n$

$n = 2^m = (n)^{1/2} = (2)^{m/2} = 2^{m/2}$ $\log 2^m$

$= m \log_2 2 = m$

$T(2^m) = T\,2^{m/2} + m \log_2 2$

$= T\,2^{m/2} + m$

$T(2^m) = S(m) \rightarrow$ (assume)

$S(m) = S(m/2) + m$

$a = 1 \quad b = 2 \quad k = 1 \quad p = 0$

$a < b^k$

$1 < 2^1 \quad 1 < 2$

$n^k \log^p n = m^1 \log^0 n$

$= m$

$= \log n$ $\rightarrow$ $n = 2^m$

$\log n = \log 2^m$

$\log n = m \log 2$

$\log n = m$

Recurssive tree method.

(Only used in Divide and Conquer).

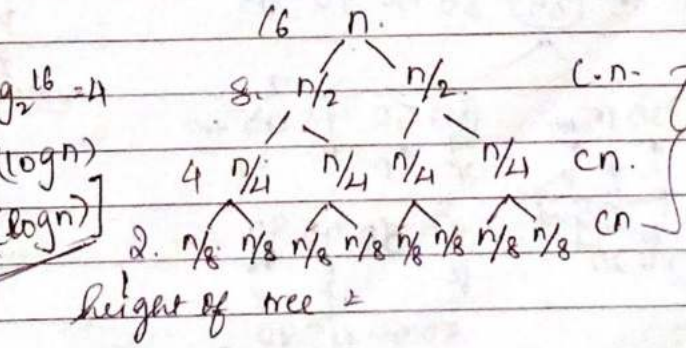
$T(n) = 2T(n/2) + cn$

if $n = 16$

$\log_2 16 = 4$

$cn(\log n)$

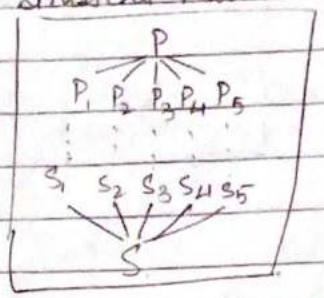
$T(n) = O[n(\log n)]$



Height of tree =

DIVIDE AND CONQUER.

Binary search, Find Maximum and Minimum., Quick Sort, Merge Sort, Strassen's Matrix Multiplication.



Quick Sort.

(Best case). when ve is exactly falls in mid after 1st pass.

Avg case.

35 50 15 25 80 20 90 45

↑ stop ← Swap. → stop

ve = 35

P will move unless and until it gets an element greater than ve

q will move unless and until it gets an element lesser than or equal.

for sorting in ascending order.

rule

rule 1: Swap p & q if p & q not crossed each other.

rule 2: if crossed Swap q with pivot element.

rule 3: compare p & q with ve and p should be > ve, q < ve

rule 4: position of element matters not the value.

1st pass

n/2 n/2 n/4 n/4 n/4 n/4

n/1 → 1 p.

n/2 → n/2 - 1

n log n = T(n)

2T(n/2) + n.

35 20 15 25 80 50 90 45

p q. stop crossed.

25 20 15 35 80 50 90 45

2nd pass.

25 20 15 ∞ | 80 50 90 45 ∞

ve p p q

swapped

15 20 25

80 50 45 90 ∞

ve p q

80 50 45 90

q p

45 50 80 90

q p

crossed

= 15 20 25 35 45 50 80 90 → sorted

→ Sorting in Linear Time
→ hiring problem
→ Indicator Random vars
→ sorting in linear (9)
→ counting sort
→ radix sort.

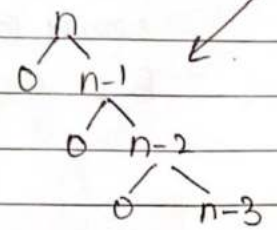
Performance of Quick Sort.

(Worst case) 1 2 3 4 5

Completly completely sorted ~~or~~ and descending sorted both are worst case at the partitioning in array becomes ~~worst case~~ unbalanced.

~~T(n)~~ 10 20 30 40 50 60 70

v p q

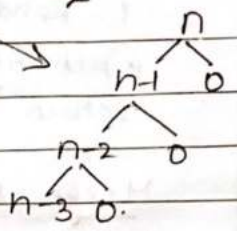


70 60 50 40 30 20 10

v p q

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

$O(n^2)$.



~~Insertion Sort~~ Algorithm for quick sort.

```
Quick sort (array A, start, end).
{
  if (start < end)
  {
    p = partion (A, start, end)
    Quicksort (A, start, ~~end~~ p-1)
    Quicksort (A, p+1, end)
  }
}
```

→ Partioning

```
partition (array A, start, end)
  pivot = A[end]
  I = start - 1
  for j = start to end - 1
  : do if A[j] < pivot.
    {
      ~~then~~ i = i+1
      Swap A[i] with A[j]
    }}.
  Swap A[i+1] with A[end]
  return i+1
}.
```

Randomised Quick Sort

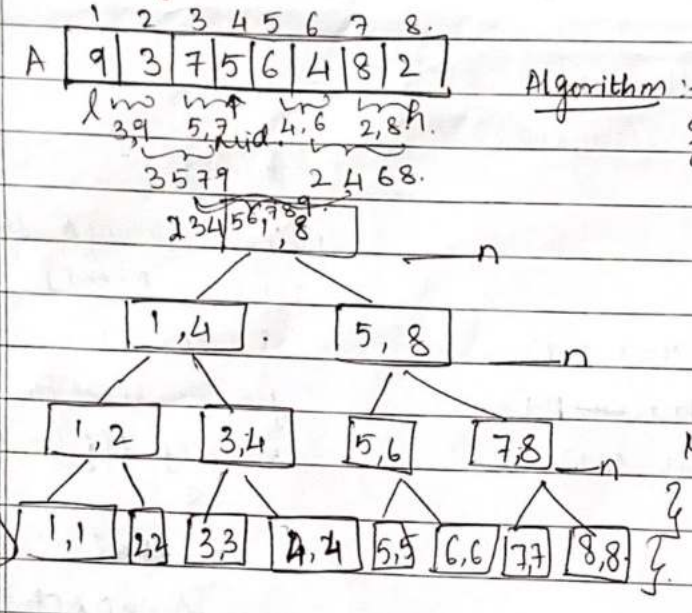
1	2	3	4

Randomised QS (A, p, r).
  if p < r.
    q = Randomised partition. (A, p, r)
    Randomised Quicksort (A, p, q-1)
    Randomised Quicksort (A, q+1, r).

Randomised partion (A p r)
  i = Random (p, r).
  exchang A[r] with A[i].
  return partition (A, p, r).

Worst case - (n^2)
Best case - nlogn.

Merge sort.



Algorithm :- Mergesort (l, h) → T(n)
{
  if (l < h)
  {
    mid = $\frac{(l+h)}{2}$; — ①
    Merge sort (l, mid); — T(n/2)
    Merge sort (mid+1, h); T(n/2)
    Merge sort (l, mid, h); n
  }
}

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

Strassen's Matrix Multiplication.

$$C_{ij} = \sum_{k=1}^{n} A_{ik} * B_{kj}$$

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times B \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = C = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

$m \times n$.

$2 \times 2 \quad 2 \times 2 \quad 2 \times 2$.

Algorithm:-

```
for (i=0; i<n; i++)
{
  for (j=0; j<n; j++)
  { c[i,j]=0;
    for (k=0; k<n; k++)
    {
      c[i,j] += A[i,k] * B[k,j]; ---- O(n^3)
    }
  }
}.
```

$$c_{11} = a_{11} * b_{11} + a_{12} * b_{21}$$
$$c_{12} = a_{11} * b_{12} + a_{12} * b_{22}$$
$$c_{21} = a_{21} * b_{11} + a_{22} * b_{21}$$
$$c_{22} = a_{21} * b_{12} + a_{22} * b_{22}$$

Strassen's Multiplication

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \quad B = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

$$T(n) \rightarrow \begin{cases} 1 \\ 8(T/2) + n^2 \end{cases}$$

$a = 8 \quad f(n) = n^2$

$b = 2$

$\log_b a = \log_2 8 = 3$

$n = 2$

$O(n^3)$

Hiring

Algorithm

```
MM(A,B,n)
{
  if (n<=2)
  {
    C = 4 formulas
  }
  else
  {
    mid = n/2.
    MM(A11, B11, n/2) (+) MM(A12, B21, n/2)      (+ : matrix addition, n^2)
    MM(A11, B12, n/2) + MM(A12, B22, n/2)
    MM(A21, B11, n/2) + MM(A22, B21, n/2)
    MM(A21, B12, n/2) + MM(A22, B22, n/2)
```

$P = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$

$Q = (A_{21} + A_{22}) \cdot B_{11}$

$R = A_{11} (B_{12} - B_{22})$

$S = A_{22} (B_{21} - B_{11})$

$T = (A_{11} + A_{12}) \cdot B_{22}$

$U = (A_{21} - A_{11}) \cdot (B_{11} + B_{12})$

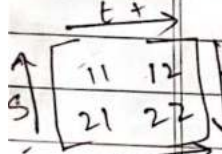
$V = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$

$C_{11} = P + S - T + V$

$C_{12} = R + T$

$C_{21} = Q + S$

$C_{22} = P + R - Q + U$



BAAB
11 11 22 22

$P = (A_{11} + A_{22}) + B_{11} \cdot B_{12}$ (illegible)

or $U = (A_{21} - A_{11})(B_{11} + B_{12})$.

$V = (A_{12} - A_{22})(B_{21} + B_{22})$.

$Q = B_{11} (A_{21} + A_{22})$

$R = A_{11} (B_{12} - B_{22})$

$S = A_{22} (B_{11} - B_{11})$

$T = B_{22} (A_{11} + A_{12})$

$C_{11} = P$ $C_{12} = R$ (+) T $C_{21} = Q + S$ $C_{22} = P$

(+) $= R + T$

Hiring problem

```
Hire assistant(n)
{
 best <- 0
 for i <-1 to n do
      if candidate [i] is better than candidate [best]
      best <- i
      hire candidate i
 }.
```

hire candidate

$C_i \ll C_h$

Cost $C_i \times n + C_h \times m$ ← first candidate interview candidate

↳ no. of hirings

Best cost :- $C_i \times n + C_h$.

worst case - $C_i \times n + C_h \times n$.

Indicator Random var

$I[A] = 1 \rightarrow$ If A occurs

$I[A] = 0 \rightarrow$ If A does not occur.

$X_H = I\{y = H\} = 1$, if y = head

$X_H = I\{y = T\} = 0$, if y = tail.

$X_H = I\{y = H\}$

$X_A = Pr(A)$.

Mean → $E[X_H] = E[I\{y = H\}]$

$= 1 * Pr\{y = H\} + 0 * Pr\{y = T\}$

$= 1 * 1/2 + 0 * 1/2$

$= 1/2$

$E[X] = E\left[\sum_{i=1}^{n} X_i\right]$

$E[X] = E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i]$.

X = no of candidates hired.

$X_i = I\{$ candidate i is hired $\}$

Finding min and max.

① Iterative Method.

```
Alg Iterative (a, n, Max, min)
{
max = a[0];
Min = a[0];
for i=1 to n do
{
  if (a[i] > max) then
    Max = a[i];
  if (a[i] < Min) then
    Min = a[i];
}
}
```

eg:-

0	1	2	3	4	5	6	7
5	7	3	4	9	12	6	2

1] max = 5 ← compare 7 i→i
min = 5;

2] max = 7. compare 3
min = 5

3] max = 7 compare 4
min = 3.

4] max = 7 compare 9
min = 3

5] max = 9 compare 12
min = 3.

6] Max = 12 compare 6
min = 3.

7] Max = 12 Compare 2.
min = 2.

② Divide and Conquer.

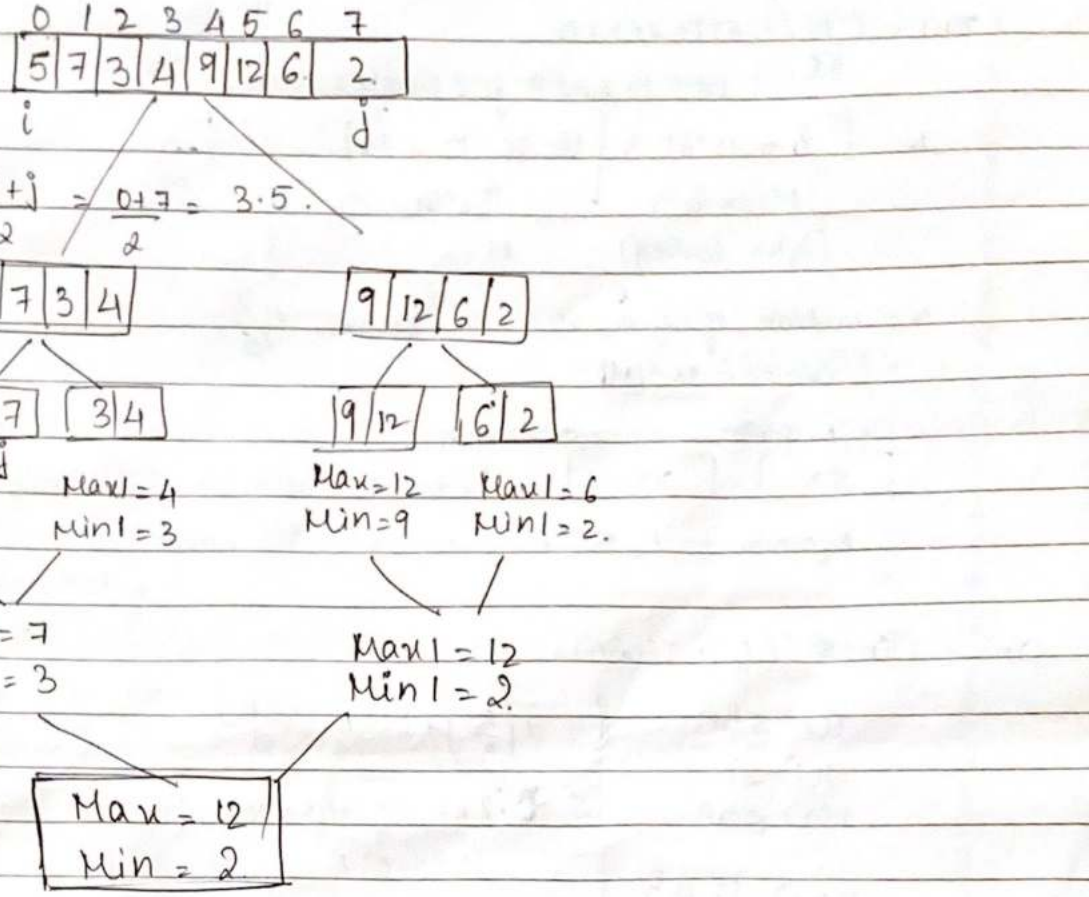
```
Alg MaxMin (i, j, Max, Min)
                first last.
{
if (i=j) then
Max = a[i];
Min = a[i];
else if (i=j-1) then
{ if [a[i] < a[j]) then
  { Max = a[j];
    Min = a[i];
  }
  else
  { Max = a[i];
    Min = a[j];
  }
}
else
{ mid = i+j/2
MaxMin (i, mid, Max, Min);
MaxMin (mid+1, j, Max1, Min1)
if (Max < Max1) then
  Max = Max1)
if (Min > Min1) then
  Min = Min;
}
}
```

one element.

two elements.

more than 2 elements.

eg.



Selection problem (Median of Medians)

Input: an array A[1,n] and k ∈ [1,n]

output: the k^{th} smallest element in A

Select (A,k)

~~finds~~ - finding a pivot x∈A,

Such that A can be partitioned into

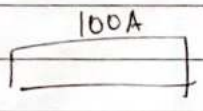
$A_1 = \{A[i] \le x\}$ and $A_2 = \{A[i] > x\}$

- if $k \le |A_1|$, Select (A_1, k).

else $k > |A_1|$, Select $(A_2, k-|A_1|)$

Return.

$|A_1|, |A_2| \le 0.7n$

eg] 

k=25, Select (A,25) = Select $(A_1, 25)$

k=64, Select (A,64) = Select $(A_2, 16)$

64 - 40 = 16.

$T(n) = ( ) + T(0.7n) + Cn$

$T(n) \leq \left(\frac{n}{5}\right) + T(0.7n) + cn$

$\frac{n}{5}$ ← time to find a good pivot(x).

A [8 5 12 8 1 3 | 18 36 12 7 14 | ..... | ...

$m_1 = 8$ $m_2 = 14$ m_3 $m_4 =$

(after sorting) M.

x = median of $\{m_1, m_2, m_3, \ldots \frac{n}{5}\}$

~~Select $(M, \frac{|M|}{2})$~~

$A_1 \geq 0.3n$ $A_2 \geq 1 - 0.3n$.

$\leq x$ | $> x$.

$A_1 \geq 0.3n$ $A_2 \leq 0.7n$.

if $m_i \leq x$, the i^{th} block contribute 3 element

$m_i > x$ i^{th} block will contribute 3 elements to A_1 to A_2

$C(n) = T(n) \leq T\left(\frac{n}{5}\right) + T(0.7n) + c \cdot n$.

$T(n) \leq Bn$.

$T(1) \leq B$

$T(2) \leq 2B$

$\vdots$

$T(k-1) \leq (k-1)B$

$T\left(\frac{k}{5}\right) + T(0.7k + ck)$.

$\leq 0.2kk + 0.7Bk + ck$.

$= (0.9B + c)k$ | $B = 10c$.

$= Bk$

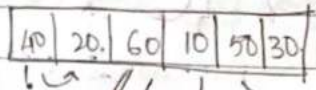
$T(k) \leq RHS \leq Bk$.

$T(1) \leq B(1)$

$= O(n)$

min 5 element partition

Insertion Sort



20 40 ~~60~~

20 40 60

20 40 60 = ~~20 40 10 60~~ = ~~20 40 10 50~~

10 20 40 60 = 10 20 40 50 60 =

0 20 30 40 50 60

Algorithm

for j = 2 to A. length

key = A[j].

i = j - 1

while i > 0 A[i] > key

initial → ~~while i > 0~~ { → value

A[i+1] = A[i] } exchanging.

i = i - 1 → position

}

A [i+1] = key.

Best case (Ascending order)

10 20 30 40 50 60.

Comparison: 0 0 / 1 0 / 1 0 / 1 0 (n-1).

~~Swaps~~

∴ Time complexity = O(n)

Worst case (Descending order)

60 50 40 30 20 10

Comparison	Swaps
0	0
1	1
2	2
3	3
(n-1)	(n-1)

$\frac{n(n-1)}{2} = \frac{n^2-n}{2}$

$n^2 \leftarrow$ $n^2 \leftarrow$

Time complex = O(n^2)

Insertion sort is stable.

Insertion sort is In place.

Bubble
~~Selection~~ Sort.

```
for i ← 1 to n-1 do
    minj ← i;
    minx ← A[i]
    for j ← i+1 to n do
        if A[j] < minx then
            minj ← j
            minx ← A[j]
    A[minj] ← A[i]
    A[i] ← minx.
```

eg → | 40 | 20 | 60 | 10 | 50 | 30 |

n-1 ↓ ← I will look

i=1, j

min = 40

20 < 40?

A[j] = min = 20

60 < 20 ✗

10 < 20 ✓

A[j] = min = 10

10 < 50 ✗

10 < 30 ✗

10 20 60 40 50 30 } 1st pass.

i=2

min = 20

20 < 60 ✓

20 < 40 ✓

20 < 50 ✓

20 < 30 ✓

10 20 60 40 50 30 } 2nd pass

⋮

(n-1) pass.

Best case (Ascending order).

10 20 30 40 50 60

0+1+2+3+ ··· (n-1)

n-1
n-2
n-3
⋮
0

$T(n) = O(n^2)$.

Worst case (Descending Order).

60 50 40 30 20 10

(n-1) — 1 swap.
(n-2) — 1 swap.
(n-3) .. — 1
⋮
0 — 1

$\frac{n(n-1)}{2} = n^2$

$T(n) = O(n^2)$.

Selection sort is not stable.
Selection sort is in place.

Selection Sort.

| 2 | 3 | 5 | 4 | 7 |

min = 2.

| 10 | 9 | 8 | 7 | 6 |

min = 6

| 6 | 9 | 8 | 7 | 10 |

sorted | unsorted

min = 7

| 6 | 7 | 8 | 9 | 10 |

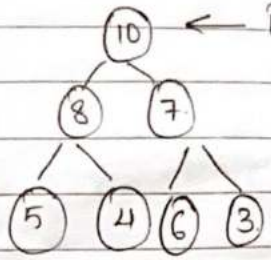
min = 8

$O(n^2)$ — Best case
$O(n^2)$ — worst case.

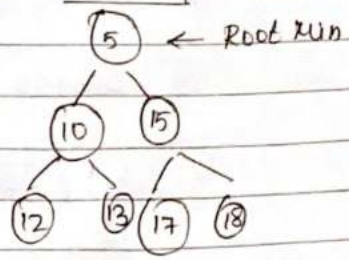
Heap sort.

1. Almost complete binary tree.
2. Left child first then right child.

Max heap



Min heap.

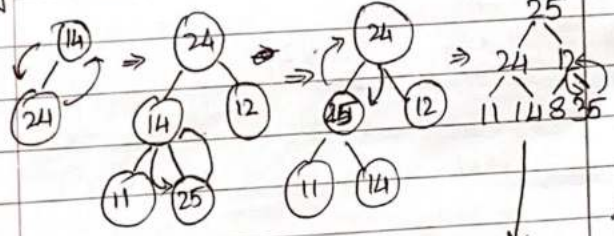


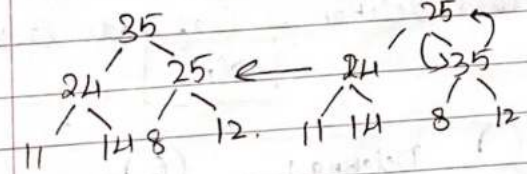
Heap tree construction.

Insert key one by one in the given order.

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

Eg - 14, 24, 12, 11, 25, 8, 35.





(Max height = $\log n$.

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

↓ Time complexity.

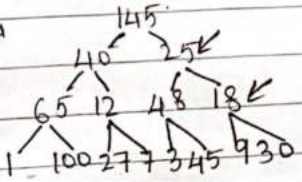
Best case

↓ Worst case ← $n(\log n)$.

Heapify Method

$T(n) = O(n)$.

Eg - 145, 40, 25, 65, 12, 48, 18, 1, 100, 27, 7, 3, 45, 9, 30.

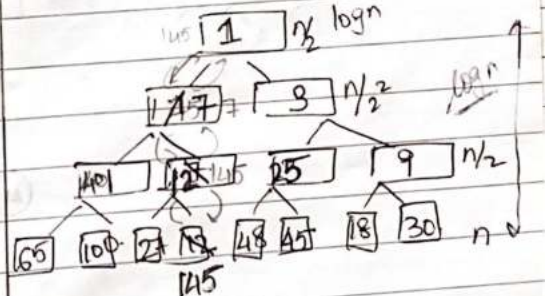


Min heap

Leaf nodes = ignored or no swapping req.

$n/2$ leaf elements are present out of n.

Start checking from 2nd last right node.



Total swaps $S = \frac{n}{2^0} * 0 + \frac{n}{2^1} * 1 + \frac{n}{2^2} * 2 + \frac{n}{2^3} * 3 \ldots + \frac{n}{2^{\log n}} * \log n$ — (1)

$$S = n\left[\frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \frac{4}{2^4} + \cdots + \frac{\log n}{2^{\log n}}\right] \quad -(1) \times \frac{1}{2}$$

$$\frac{S}{2} = n\left[\frac{1}{2^2} + \frac{2}{2^3} + \frac{3}{2^4} + \frac{4}{2^5} + \cdots + \frac{\log n - 1}{2^{\log n}} + \frac{\log n}{2^{\log n + 1}}\right] \quad -(2)$$

$S - \frac{S}{2}$ Subtract (2) from (1).

$$\frac{S}{2} = n\left[\frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^4} + \frac{1}{2^{\log n}}\right] - \frac{\log n}{2^{\log n + 1}}$$

Use GP formula $r < 1$

$$S = \frac{a(1 - r^n)}{1 - r}$$

$$\frac{S}{2} = n\left[\frac{\frac{1}{2}\left(1 - \frac{1}{2^{\log n}}\right)}{1/2}\right] - \frac{\log n}{2^{\log n + 1}}$$

$$\frac{S}{2} = n\left[\left(\frac{2^{\log n} - 1}{2^{\log n}}\right) - \frac{\log n}{2^{\log n + 1}}\right]$$

$$= \left(n\left[\frac{n-1}{n}\right] - n\left(\frac{\log n}{2n}\right)\right]$$

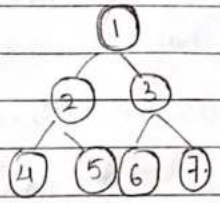
$$S/2 = \left((n-1) - \frac{\log n}{2}\right) \quad = [S\ 2n - 2 - \log n \times 2$$

$\frac{\log n}{2^{\log n + 1}}$

$(2n - 2 - \log n)$.

$= O(n)$

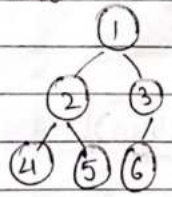
Deletion in heap sort.



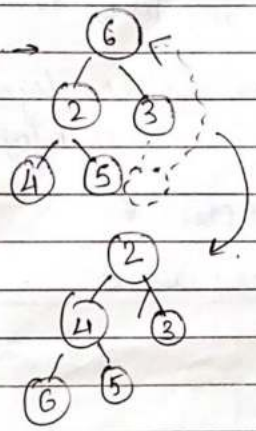
Best case deletion — leaf node last element.

$O(n)$.

But



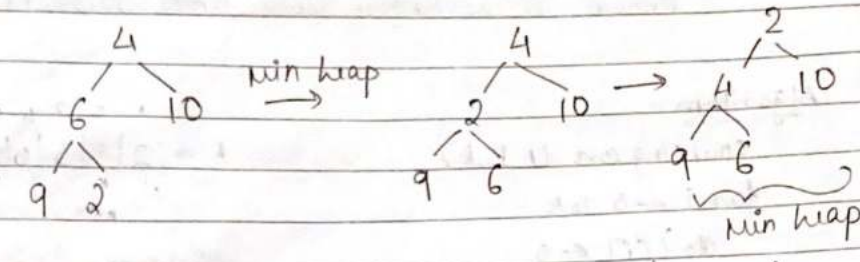
Deleting 1 →



Worst case deletion — Deleting Root node

$O(\log n)$

eg] 4 6 10 9 2 → 2 4 6 9 10

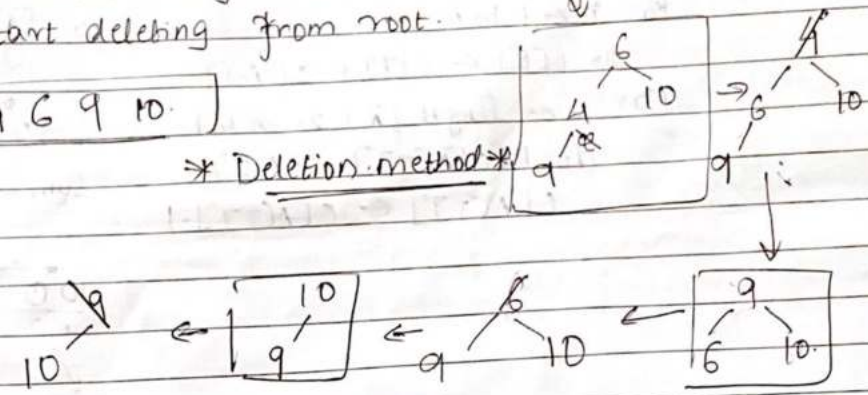


Now, inserting it into array

2. Start deleting from root.

[2. 4 6 9 10]

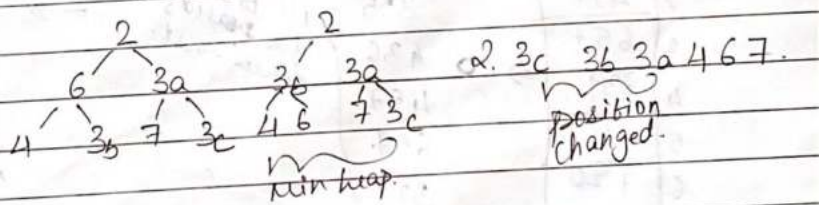
* Deletion method *



O(n) + n log n ⇒ [O(n log n)] → Heap sort.
Time complexity

heapsort is in place → Does not take any extra space.
heapsort is unstable → unstable.

2 6 3a 4 3b 7 3c.

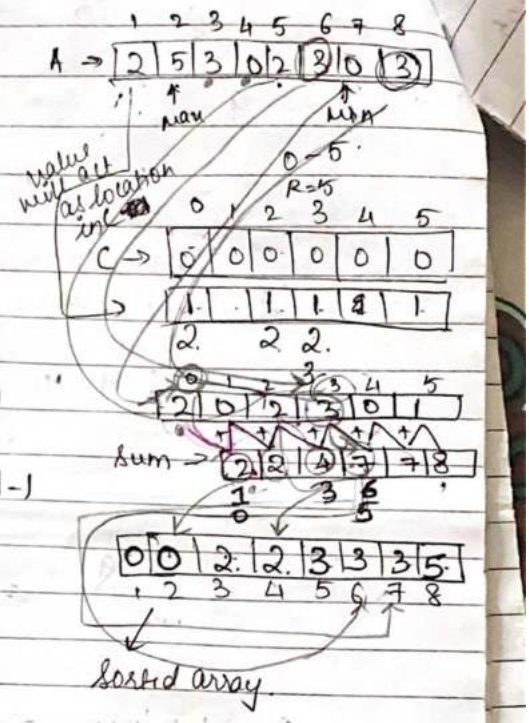


Counting Sort.

known as sorting in linear time because $T(n) = O(n)$

Algorithm

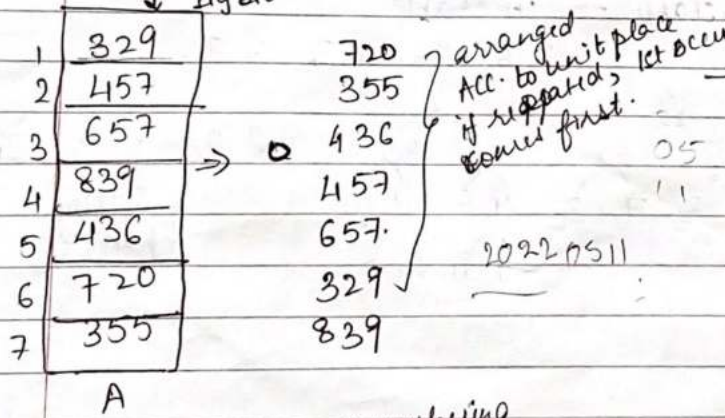
```
Counting sort (A, B, k)
for i ← 0 to k
    do C[i] ← 0.
for J ← 1 to length [A]
    do C[A[J]] ← C[A[J]] + 1
For i ← 1 to k
    do C[i] ← C[i] + C[i-1]
For J ← length [A] down to 1
    do. B[C[A[J]] ← A[J]
        C[A[J]] ← C[A[J]] - 1
```



Radix Sort.

$T(n) = O(d(n+k))$

↳ used for numbers and alphabets.



$T(n) = O(d(n+k))$

numbering

digit of number

Greedy Technique. (Algorithm).

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

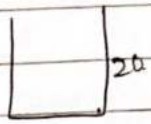
→ feasible solution
→ optimal solution

* ~~Minimum~~ cost
* Max profit
* Min Risk

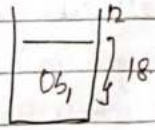
Knapsack problem.

objects	ob_1	ob_2	ob_3
profit	25	24	15
weight	18	15	10
	25/18 = 1.3	24/15 = 1.6	15/10 = 1.5

knapsack capacity (M) = 20



1] greedy about profit

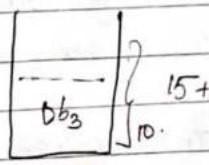


$25 + \frac{2}{15} \times 24 = 28.2$

$25 + \frac{2}{18} \times 24 = 25 + \frac{8}{3}$

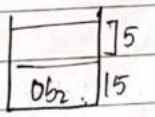
$= 25 + \frac{8}{3} = 28.2$

2] greedy about weight.



$15 + \frac{10}{15} \times 24 = 31$

3] keeping both in mind.
only correct answer.



$24 * \frac{5}{15} \times 15 = 31.5$

$24 + \frac{15}{15} \times 5 = 31.5$

O(n) → for i=1 to n.
Calculate profit/weight.
O(nlogn) → sort objects in decreasing order of P/W Ratio.
→ for i=1 to n
if M>0 and $w_i \le M$)
$M = M - w_i$;
$P = P + P_i$;
O(n) else break
if (M>0)
$P = P + P_i \left(\frac{M}{w_i}\right)$;

O(n) + O(nlogn) + O(n)

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

Job sequencing (Greedy Technique)

	J_1	J_2	J_3	J_4
profit	50	15	10	25
Deadline	2	1	2	1

* 1 → man doing all jobs.
* No guiding in between.
* 1 unit of time for each job

1	2	3
50	25	15	10
2	1	1	2

J_1 | J_3 (0 1 2) → 50+10=60 ✗

J_4 | J_1 (0 1 2) → 25+50=75 ✓

Algorithm

1] Arrange all jobs in decreasing order of profit.

2] For each job (m,i) do linear search to find particular slot in array of size (n), where n = maximum dead line.

m = Total jobs

time for doing the job = n × m.

Worst case when no of jobs = max dead line

n^2.

$T(n) = O(n \log n) + O(n^2)$

$= O(n^2)$

Spanning tree.

A connected subgraph 's' of graph G (V, E) is said to be spanning if,

1] 's' should contain all vertices of 'G'

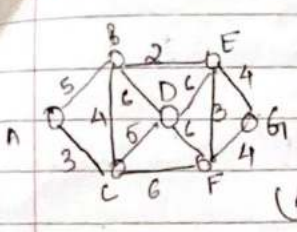
2] 's' should contain (|V|-1) edges.

n^{n-2} n = no. of vertex.

↓ when a complete graph is given.

no. of spanning tree possible.

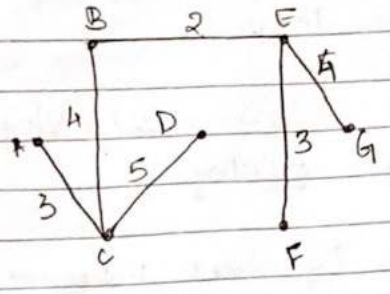
Kruskal's algorithm for minimum spanning



$(n-1)$
$7-1=6$

1] Construct Min heap with 'e' edges. - $O(n \log n)$
2] Take one by one edge and add in spanning tree. (cycle should not be created)
Best Case $(n-1)$ edges
Worst case e edges.

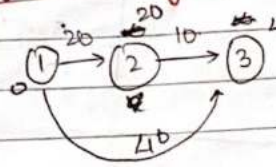
2 - BE
3 - AC
3 - EF
4 - BC
4 - EG
~~4 - FG~~
~~5 - AB~~
5 - CD
6 - BD
6 - DE
6 - DF
~~6 - CF~~



21 - minimum spanning.

$T(n) = e \log e$ (no. of edges)

Dijkstra's Algorithm (single source shortest path).



Relaxation

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

$d(v) = d(u) + C(u,v)$.

$d(v) = 0 + 20 < \infty$ -(1)

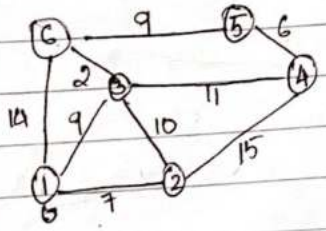
$d(u) + C(u,v) < d(v)$

$0 + 40 < \infty$ -(2).

$20 + 10 < 40$

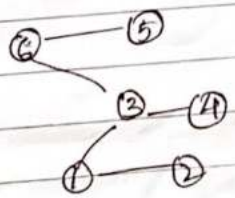
$30 < 40$

$\therefore$ 3 = 30



Shortest path to be chosen.

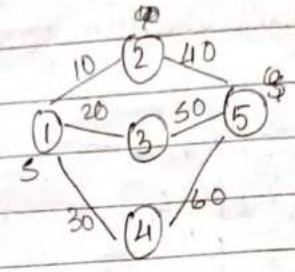
Source	Destination				
1	2	3	4	5	6
	∞	∞	∞	∞	∞
1	(7)	9	∞	∞	14
1,2	(7)	(9)	22	∞	14
1,2,3	7	9	20	∞	(11)
1236	7	9	(20)	20	11
12364	7	9	20	20	11
123645	7	9	20	20	11



Create vertex set Q
for each vertex v in graph.
dist[v] = ∞
add v to Q
dist[source] = 0

$\Theta(V)$

$O(V)$



while Q is not empty.
v = extract-min[Q] $O(\log V)$
for each neighbour v of u. $\log V$
Relax(u,v)

$$O(V) + O(V) + V\log V + \log V \cdot E$$

$$T(n) = O[E \log V]$$

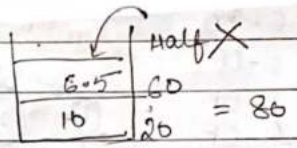
0/1 Knapsack problem. (0 - absent, 1 - present)

Object	ob_1	ob_2	ob_3
weight	2	4	8
profit	20	25	60

knapsack (M) = 12

P/W = 10 6.2 7.5

Greedy approach.



~~Not used~~ Greedy approach not used because either 0/1 property is only possible.

ob_1	ob_2	ob_3	
0	0	0	- NP
0	0	1	- 60
0	1	0	- 25
1	0	0	- 20
1	0	1	- 80
1	1	0	- 45
0	1	1	- 85
1	1	1	- NP.

Recursive equation.

0/1 Knapsack (n, m) =

if n = 3

0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1

$n = p_n, w_n$

Now: { 01 KS(n-1), (m-w_n) + p_n → considered
01 K(n-1), m → not considered. }

01 KS(n-1, m) $w_n > m$.

0 n=0 OR m=0

2^0 01 KS(4,4) n×m = 16 → unique problems.

2^1 01 KS(3,3) 01KS(3,4)

2^2 01KS(2,2) 01KS(2,3) 01KS(3,3) 01KS(2,4)

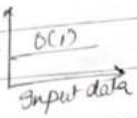
01(1,1) 01(1,2) 0(1,2) (1,3) (1,2) (1,3) (1,3) (1,4)

$O(2^n)$ ↓ → $O(n \cdot m)$

table size (n+1) (m+1)

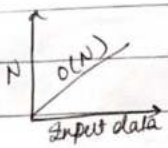
→ constant time complexity [O(1)]

* Complexity is constant, the size of input (n) doesn't matter.
* Algorithms with constant time complexity takes a constant amt of time to run and are independent of size of n.



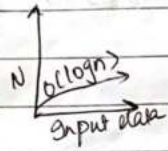
→ Linear time complexity [O(n)]

* Time complexity grows in direct proportion to the size of the input
* The algorithms will perform process the input (n) in 'n' number of operations.



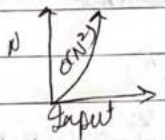
→ Logarithmic time complexity [O(logN)]

* Makes computation really fast.
* An algorithm is said to have logarithmic time complexity if its time execution is proportional to the logarithmic input size.
* Instead of increasing the time to execute, the time is decreased that inversely proportional to the input 'n'



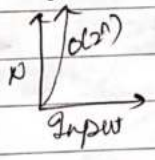
→ Quadratic Time Complexity [O(n²)]

* This type of algorithms, the time it takes to run grows directly proportional to the square of the size of the input.

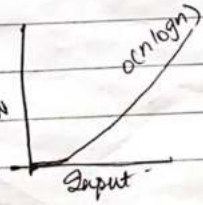


→ Exponential time complexity [O(2ⁿ)]

The growth rate doubles with each addition to the input (n), often iterating through all subsets of the input elements.



→ Linearithmic time complexity [O(nlogn)]



→ Slightly slower than a linear algorithm but better than quad algo.

ANALYSIS OF ALGORITHM

Module 3.2 :- Dynamic programming and ~~optimization problems~~ optimal binary search trees, ~~Floyd warshall~~ Algorithm for all pair shortest path, ~~longest~~ common subsequence, Travelling ~~salesman~~ problem.

Module 4.

4.1 The Backtracking Technique, ~~N-Queens problem~~, ~~Hamiltonian Circuit problems~~, ~~Sum of subsets problem~~.

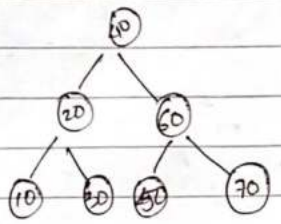
4.2 Travelling ~~salesman problem~~, ~~15 puzzle problem~~ & 0/1 ~~knapsack~~ using ~~branch and bound~~.

Module 5

5.1 NP and N.P complete.

5.2 NP reducibility.

★ Optimal binary search trees.. $C[i,j] = \min_{i<k\le j} \{ C[i,k-1] + C[k,j] \} + w(i,j)$



$\log n$.

$\frac{2^n C_n}{n+1}$

nodes.

Cost = $\frac{\text{height sum}}{\text{total no of nodes.}}$

optimal sol^n = $\frac{\text{frequency} \times \text{height} + \text{freq} \ldots}{\text{total no. of nodes.}}$

eg-

	1	2	3	4
keys →	10	20	30	40
freq →	4	2	6	3

$C[0,0]$

i \ j	0	1	2	3	4
0	0	4	8	20^3	26^3
1		0	2	10^3	16^3
2			0	6	12^3
3				0	3
4					0

$l = j - i = 0$

$0-0=0$

$1-1=0$

$2-2=0$

$3-3=0$

$4-4=0$

$l = j - i = 1$

$1-0=1\ (0,1)$

$2-1=1\ (1,2)$

$3-2=1\ (2,3)$

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

$C[0,1] = 4$

$C[1,2] = 2$

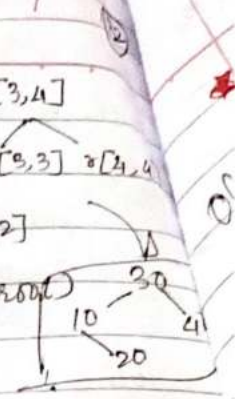
$C[2,3] = 6$

$C[3,4] = 3$

$^5C_3 = \frac{3\times2\times3}{3!(n-r)!} = \frac{3\times3\times3}{3\times3\times3\times0}$

$^{20}C_{11} = \frac{20}{11} = 5$

r[0,4]



$l = j - i = 2$

$= 2-0 = 2 \quad (0,2)$

$= 3-1 = 2 \quad (1,3)$

$= 4-2 = 2 \quad (2,3)$

c [0,2]	
10	20
4	2

$4\times1 + 2\times2 = 8 \rightarrow$ optimal. (10 as root)

$2\times1 + 4\times2 = 10$

$w(0,4) = \sum_{i=1}^{4} f(i)$

c [1,3]	
20	30
2	6

$2\times1 + 6\times2 = 14$

$6\times1 + 2\times2 = 10$

c [2,4]	
30	40
6	3

$6\times1 + 3\times2 = 12$

$6\times2 + 3\times1 = 15$

$l = j - i = 3$

$3-0 = (0,3)$

$4-1 = (1,4)$

$4\times1 + 2\times2 + 6\times3 =$

$4\times2 + 2\times3 + 6\times1$

$4\times3 + 2\times1 + 6\times2$

$4\times1 + 2\times2 + 6\times2$

$4\times1 + 6\times2 + 2\times3$

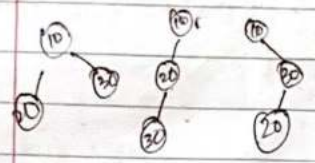
$w[0,3] = \sum_{i=1}^{3} f(i)$

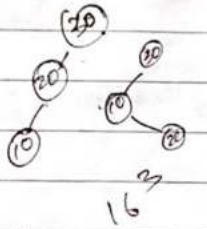
$= 12$

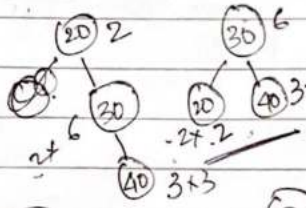
$c[0,3] = \min\{c[0,0]+c[1,3]+12,$
$c[0,1]+c[2,3]+12,$
$c[0,2]+c[3,3]+12\}$

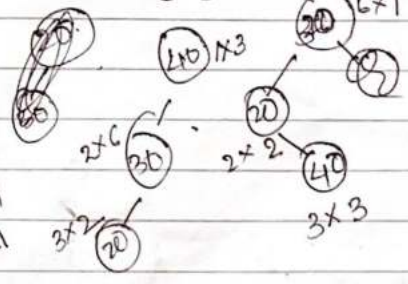
22 22 20

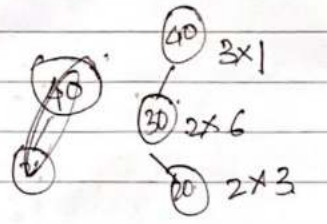
20, 30, 40.











$= 2+12+9 = 23$

$= 6+4+6 = 16$

$= 3+12+6 = 21$

$= 6+4+9 = 19$

$= 3+12+6 = 21$

$l = j - i = 4$

$= c[0,4] = c[0,0] + c[1,4] + 15 = 0 + 16 + 15 = 31$

$c[0,1] + [2,4] + 15 = 4 + 12 + 15 = 31$

$c[0,2] + [3,4] + 15 = 8 + 3 + 15 = 26$

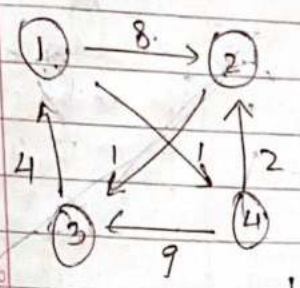
$(0,3) + [4,4] + 15 = 20 + 0 + 15 = 35$

Camlin Page
Date /

★ Floyd marshall Algorithm.

$O(n^3)$ { Best case – Vellagy.
worst case – ~~Vellagy~~ $n^2 \times n$? n times $= n^3$ (matrix change)

Space complexity $= O(n^2)$



$$D^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & \infty & 0 & \infty \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{array} n^2$$

via vertex 1

$$D^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{array} n^2$$

$9+4=13$
$2+8+4=7$
$4+1+7=$

$$D^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & 9 & 10 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 3 & 0 \end{bmatrix} \end{array} n^2$$

$$D^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix} \end{array} n^2$$

$$D^4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix} \end{array}$$

$A^k[i,j] = \min\{A^{k-1}[i,j], A^{k-1}[i,k] + A^{k-1}[k,j]\}$

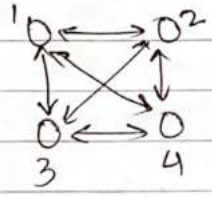
$i=4 \quad j=3$

$k=2$

$A^2[4,3] = \min\{9, \cancel{2+8}\} = \{9, 3\}$

★ Travelling salesman problem.

$O(n!)^{2,3} = O(n^2) \Rightarrow$ Brute force.

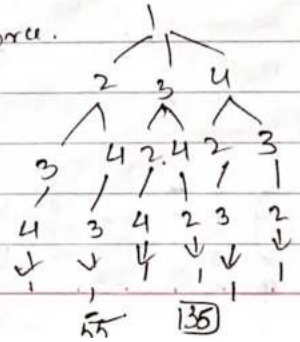


$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 25 & 10 \\ 15 & 30 & 0 & 5 \\ 15 & 10 & 20 & 0 \end{bmatrix} \end{array}$$

greedy approach – 1 to 2 = 10
– 2 to 4 = 10
– 4 to 3 = ~~20~~ 5
3 to 1 = 15
= 55

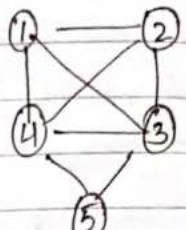
Cozent 3
Only not visited

Brute force.



greedy failed

(Branch and ~~Bond~~ Round)



	1	2	3	4	5
1	∞	10	20	0	1
2	13	∞	14	2	0
3	1	3	∞	0	2
4	18	3	15	∞	0
5	12	0	3	12	∞

→ 1 0 3 0 0 = 4

① ∴ 21+4 = 25

reduced cost (min).

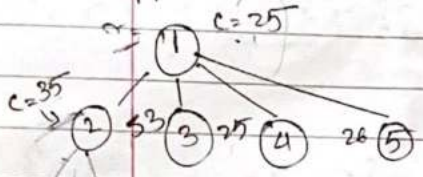
10 – 1
2 – 2
2 – 3
3 – 4
– 5
4
21

	1	2	3	4	5
1	∞	20	30	10	11
2	15	∞	16	4	2
3	3	5	∞	2	4
4	19	6	18	∞	3
5	16	4	7	16	∞

	1	2	3	4	5
1	∞	10	17	0	1
2	12	∞	11	2	0
3	0	3	∞	0	2
4	15	3	12	∞	0
5	11	0	0	12	∞

→ A
↓ C

upper = ∞



	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	∞	∞	11	2	0
3	0	∞	∞	0	2
4	15	∞	12	∞	0
5	11	∞	0	12	∞

c(1,2) + r + r̂
10 + 25 + 0 = 35

continue the same for all.

(1,3)

→	∞	∞	∞	∞	∞
	12	∞	∞	2	0
	∞	3	∞	0	2
	15	3	∞	∞	0
	11	0	∞	12	∞

c(1,3) = 17 + 25 + 11 = 53

11

14
23
37

(1,4)

→	∞	∞	∞	∞	∞
	12	∞	11	∞	0
	0	3	∞	∞	2
	∞	3	12	∞	0
	11	0	0	∞	∞

c(1,4).
= 0 + 25 + 0 = 25

→	∞	∞	∞	∞	∞
	10	∞	9	2	∞
	0	3	∞	0	∞
	15	3	9	∞	∞
	∞	0	0	12	∞

2 (1,5)
= 1 + 25 + 5 = 31
23

0/1 Knapsack problem using Branch Bound.

upper bound = $\sum_{i=1}^{n} p_i x_i$

$C = \sum_{i=1}^{n} p_i x_i$ (with fraction)

	1	2	3	4
P	10	10	12	18
W	2	4	6	9

$M = 15$

{1, 1, 0, 1}
10+10+0+18 = 38
2+4+0+9 = 15

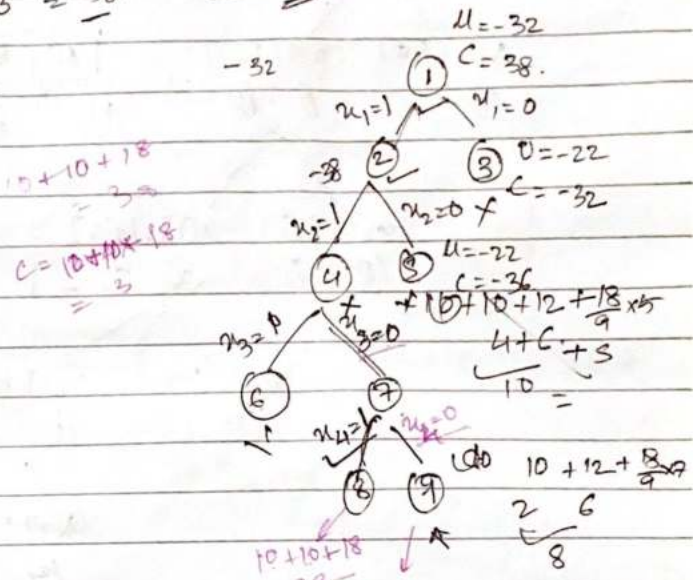
LC-BB
↳ Jiska bhi sabse kam cost

upper = ∞.

$C = 10 + 10 + 12 + \frac{6}{9} \times 3 = 38$

2+4+6 → 12

15 − 12 = 3



Longest common subsequence (LCS)

String 1: a b c d e f g h i j

String 2: c d g i

ex

cdgi → There in string 1 in same order but break doesn't matter.

String match but not long. egix

cdgi ✓

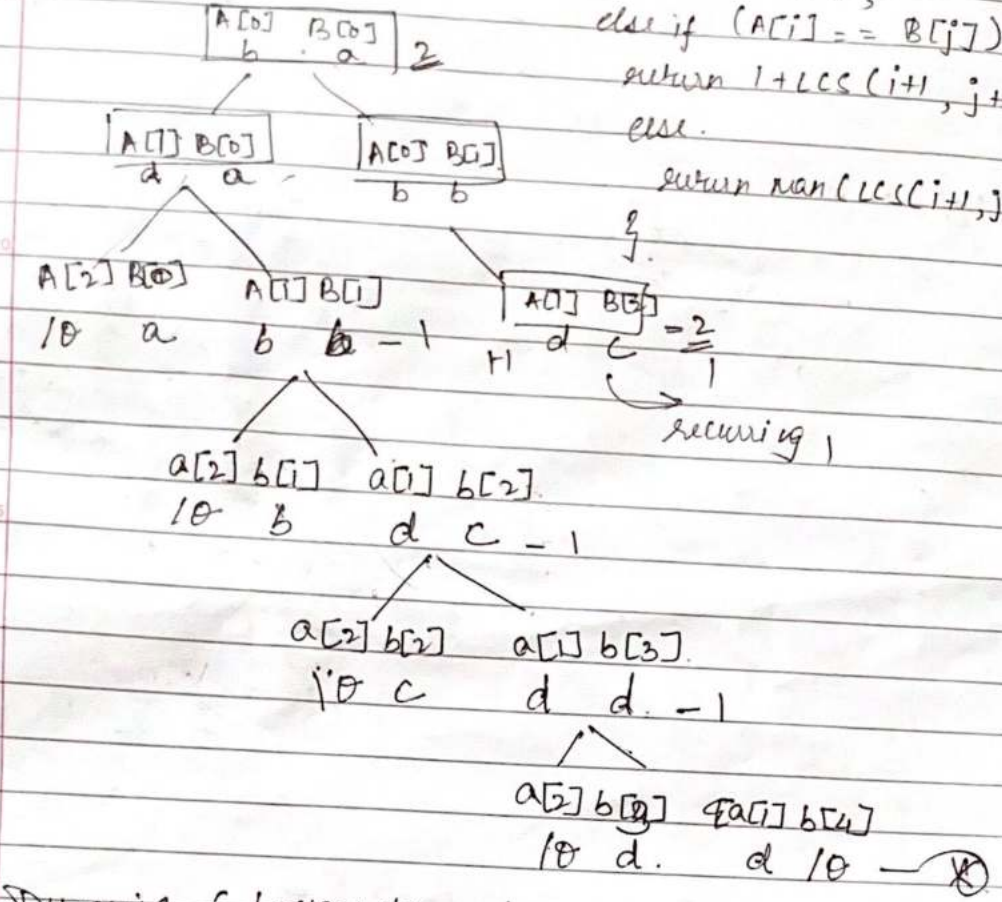
time (O(m×n))

Recursion (bottom top but table bottom down)

```
int LCS(i,j)
{
  if (A[i]=='\0' || B[j]=='\0')
      return 0;
  else if (A[i] == B[j])
      return 1+LCS(i+1, j+1);
  else
      return max(LCS(i+1,j), LCS(i,j+1));
}
```

A | b | d | \0 | (0 1 2)

B | a | b | c | d | \0 | (0 1 2 3 4)



Dynamic (bottom down last table top to up)

Algo:- if (A[i] = B[j])

LCS[i,j] = 1+LCS[i-1, j-1]

↓ diagonal element.

else,

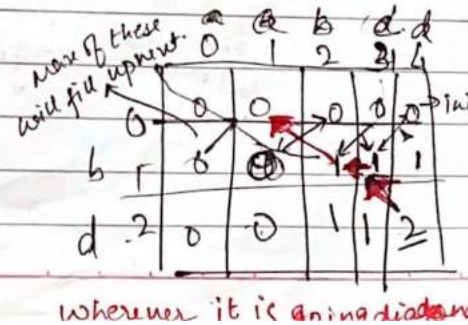
LCS[i,j] = max(LCS[i-1,j], LCS[i,j-1])

O(m×n).

[If the tables are not matching then take left bottom & right top's max element otherwise add 1 to left top and write in right bottom]

A | b | d | (0 1 2) m

B | a | b | c | (0 1 2) n.



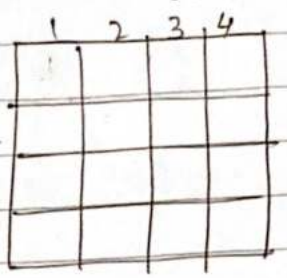
bd.

wherever it is a in a diagonal.

★ N-Queens. Chessboard Queen.

$O(n^2)$

Q_1, Q_2, Q_3, Q_4



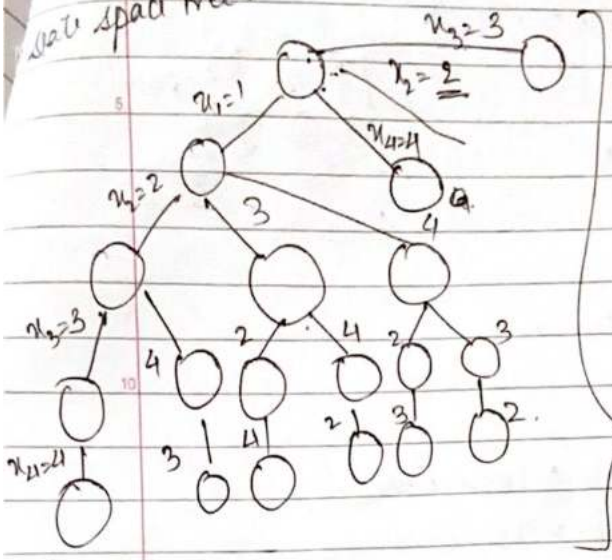
Same: row, column, diagonal

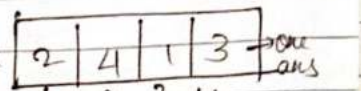
↓

Attack

(which we have to avoid)

State space tree.





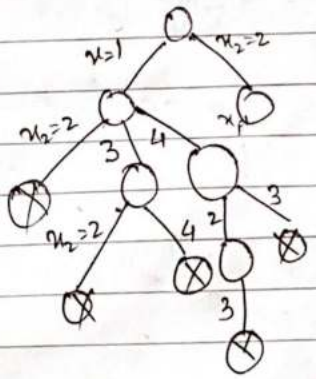
Soln
2, 4, 1, 3
3, 1, 4, 2

$$1 + \sum_{i=0}^{3} \left[\prod_{j=0}^{i} (4-j) \right] = 65$$

max nodes.

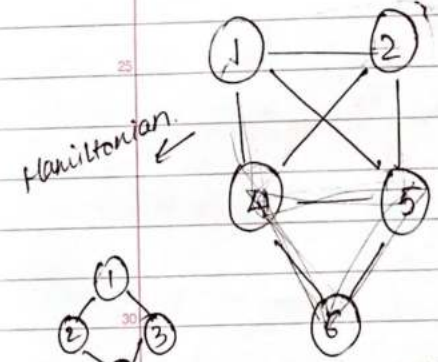
product of.

Bounding function = ✗ row ✗ col ✗ dia.



→ keep on backtracking until the one in attack.

★ Hamiltonian cycles/ circuits.

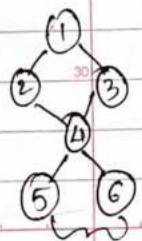


Hamiltonian.

Articulation point – wherein a node acts as a junction and we have to pass through it 2 times.

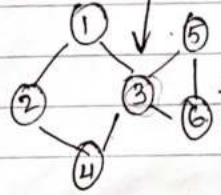
↓

here it does not form cycle.

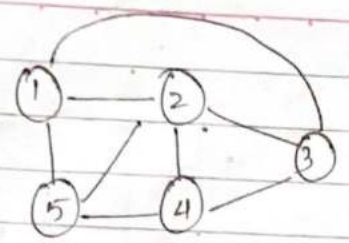


→ Non hamiltonian

pendant vertices.



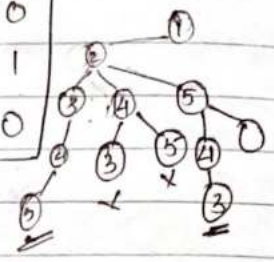
→ Non hamiltonian.



G =

	1	2	3	4	5
1	0	1	1	0	1
2	1	0	1	1	1
3	1	1	0	1	0
4	0	1	1	0	1
5	1	1	1	1	0

x	1	2	3	4	5
	1	2	3	4	5



1 2 3 4 5
1 2 3 5 4
:
:

```
Algo (k)
{
do
{
Next vertex (k);
if (x[k] == 0)
    return;
if (k == n)
    print (x[1:n]);
else
    Hamiltonian (k+1);
} while (true);
}
```

```
Algo next vertex.
{
do
{
x[k] = (x[k]+1) Mod (n+1);
if (x[k] == 0) return;
if (G[x[k-1], x[k]] != 0)
    for j = 1 to k-1 do if (x[j] == x[k]) break;
if (j == k)
    if ((k < n) OR ((k == n) AND G[x[n], x[1]] != 0))
        return;
} while (true);
}
```

edge to previous fn

checking for duplicate

if last than connection to 1st vertex

① no duplicate

② edge from previous fn.

③ last vertex to 1st vertex connection to be there

Time complexity $O(n^n)$

★ Sum of subsets. using Least cost Branch and Bound. $O(2^n)$

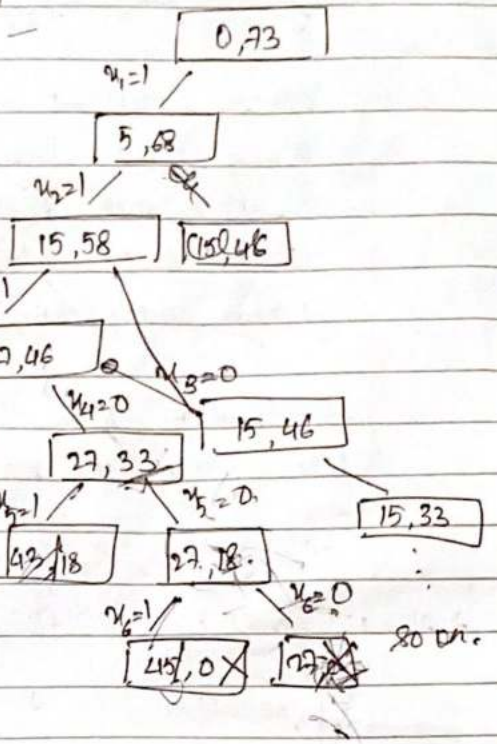
$w[1:6] = \{5, 10, 12, 13, 15, 18\}$

$n = 6 \quad m = 30.$

	1	0	1	0	1	0
x	1	1	0	0	1	0
	1	2	3	4	5	6

$\sum_{i=1}^{k} w_i x_i + w_{k+1} \le m$? Bounding functions.

$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i > m$ — DFS is followed



★ 15 puzzle problem.

1	3	4	15
2		5	12
7	6	11	14
8	9	10	13

⇒

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

$\hat{c}(x) = f(x) + \hat{g}(x).$

$\hat{c}(x)$ = is the estimated min cost to reach the goal node.

$f(x)$ = no. of move from initial state.

$g(x)$ = No. of non blank tiles that are not in their goal position.

→ From initial problem move the blk up / down / left / right then calculate cost and the whichever has less cost branch out that

→ ~~One of~~ which direction is used in step 1 ~~for~~ opposite of that move be used again step 2.

★ NP and NP complete.

Polynomial time. (P) (Deterministic)	Exponential time.
Linear search – n	0/1 knapsack – 2^n
Binary search – log n.	Travelling SP – 2^n
Insertion sort – n^2.	Sum of subsets – 2^n
Merge sort – n log n.	Hamilton circuits – 2^n.
Matrix multiplication – n^3	

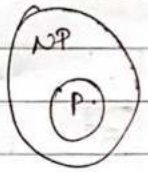
Deterministic Algo – when algos are known.
Non-deterministic Algo – when we dont know the algo.
↳ Asign in some places are kept blank.

↳ Algo NSearch (A, n, key). (Binary search).

```
{
  j = choice();  — 1
  if (key = A[j]).  (← non deterministic)
  {
    write (j);
    success();  — 1
  }
  write (0);
  failure();  — 1
}.
```

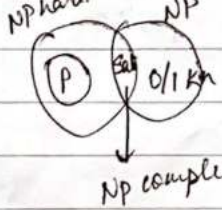
O(1).

P — Deterministic polynomial algorithm.
NP — Non Deterministic polynomial algorithm.



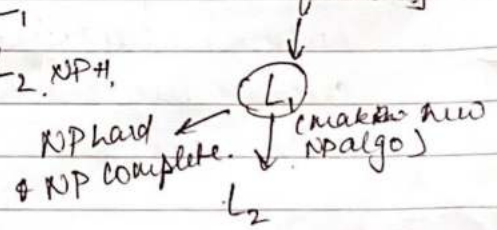
CNF – Satisfiability – 2^n.

Reduction.
NP Hard. —poly→ NP Hard.
Sat. ∝ 0/1 knapsack.



P ⊆ NP
P = NP ↳ Cook's theorem

∀ NPH Sat ∝ L_1
$L_1 \propto L_2$ NPH.



Satisfiability will come in P if and only if P = NP