Polynomial notation :-
Polynomial algorithms include quadratic algorithms $O(n^2)$, cubic algorithms $O(n^3)$.
represents an algorithm whose performance is directly proportional to square of size of the data set.

Exponential notation :- $O(2^n)$ describes an algorithm whose growth doubles with each addition to the data.

Exp. 9ces more (doubles) than poly.

n	n^2	2^n
1	1	2
2	4	4
3	9	8
4	16	16
5	25	32

2^n increases more.

Avg., Best, worst case complexities :-
Time complexity is dependent on parameter ~~with~~ associated with the I/O instances of the problem.

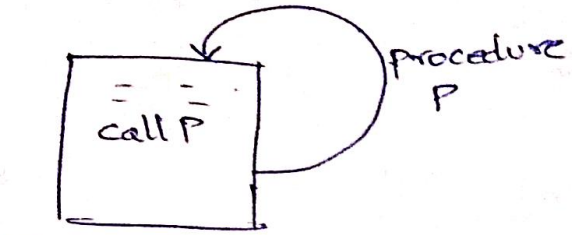
Eg:- According to no. of instances, time complexities 9ces.

I/P 1 :- -1, -23, -11, -2, -4, -6, -7, -8, -9, 2.
I/P 2 :- 1, -7, -11, -2, -6, -8, -7, -9.

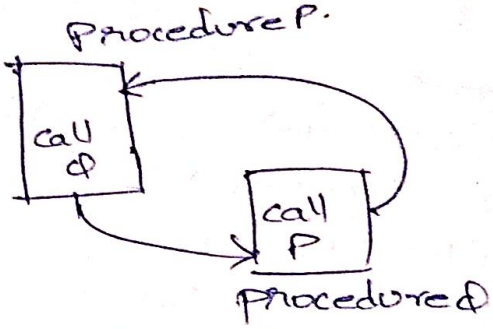
Search for the first occuring even no. in the list.
In case of input 1, it takes 10 comparisons to find out.
In case of input 2, the first element is +ve.
Running time of algorithms are not just dependent on the size of the I/p but also on its nature.

Analyzing Recursive algorithms:-

If 'p' is a procedure containing a call statement to itself or, to another procedure that results in a call to itself. the procedure P is said to be a recursive procedure



Direct Recursion



Indirect Recursion.

Q:- Recursion is used to solve Towers of Hanoi problems.

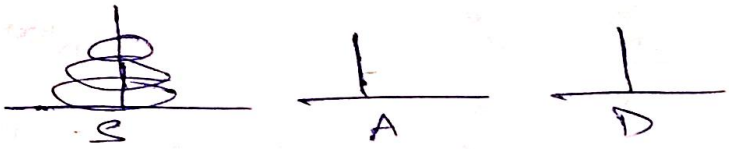
Towers of Hanoi Problems:-

It is a mathematical puzzle where we have three rods and 'n' disks.

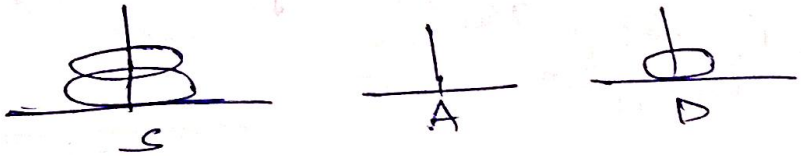
Its objective is to move the entire stack to another rod.

Rules :-

(1) Only one disk can be moved at a time.
(2) Larger disk can't be placed on the top of smaller disk.
(3) We can use auxilary tower for temporary storage of disks.



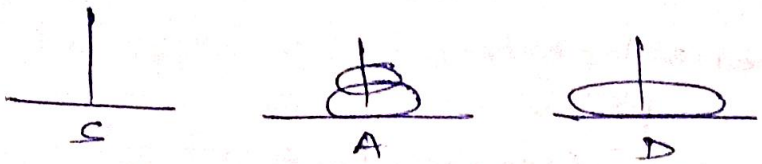
Step-1:- Move disk from S to D



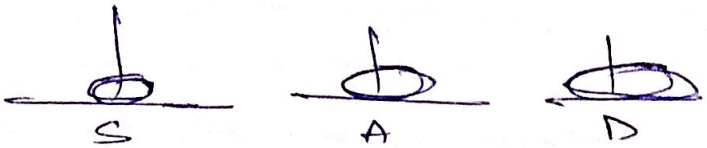
Step 2:- Move disk from S to A

(3) Move disk from D to A.

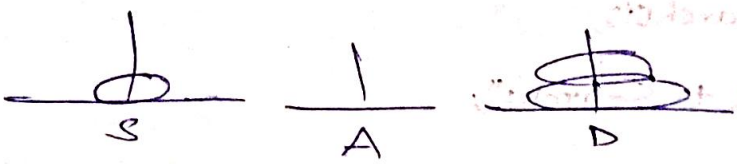
(4) Move disk from S to D



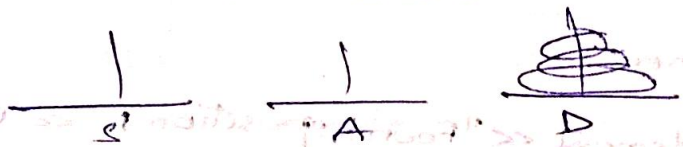
(5) Move disk from A to S.



(6) Move disk from A to D



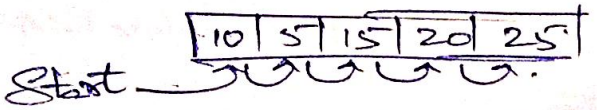
(7) Move disk from S to D



No. of disks = 3
No. of moves required
$= 2^n - 1$
Here $n = 5$
$2^n - 1 = 32 - 1 = 31$ moves required.

Searching :- It is the process of finding a given value position in a list of values.
It decides whether a search key is present in the data or not
It is the algorithmic process of finding a particular item in a collection of items.
It can be done on internal ~~data~~ or external D.S.
To search an element in a given array, can be done in :-
(1) Sequential Search (2) Binary search.

(1) Sequential Search :- also called as Linear Search.
Starts at the beginning of the list and checks every element of the list. It is a basic & simple search algorithm.
It compares the element with all the other elements given in the list. If the element is matched, it returns the value index, else it returns -1.



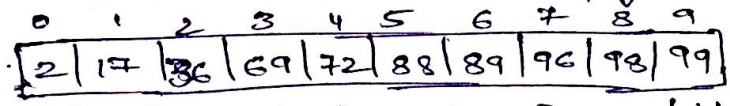
Suppose, there are 100 elements & the req. element is at 100th position, then it executes 100 times - Time req. is more.

```
#include <stdlib.h>
#define max_size 5
using namespace std;
int main()
{
int arr_search[max_size], i, element;
cout << "Linear search example\n";
cout <<"\n Enter" << max_size << "Elements for searching";
for (i=0; i< max_size ; i++)
    cin >> arr_search[i];
cout << "\n Your Data :";
for ( i=0; i< max_size; i++)
{
   cout<< "\t" << arr_search[i];
}
cout << "\n Enter Element to Search:";
cin >> element;
for (i=0; i< max_size ; i++)
{
   if (arr_search[i] == element)
   {
     cout <<"Element" << element <<"Found :position i "<< i+1;
     break;
   }
}
if (i==max_size)
    cout << "Element" << element << "Not found";
 getch();
}
```

Sample I/O:-

```
Linear Search Example
Enter 5 elements for searching
```

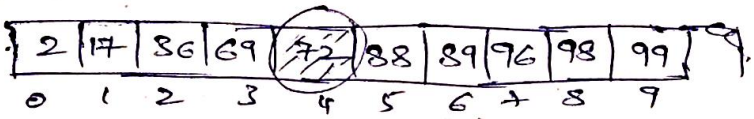
Binary Search:- Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.



Assume low=0, high=9.

(1) Calculate Mid, Mid = $\frac{low+high}{2} = \frac{0+9}{2} = 4.5 \simeq 4$

(2)



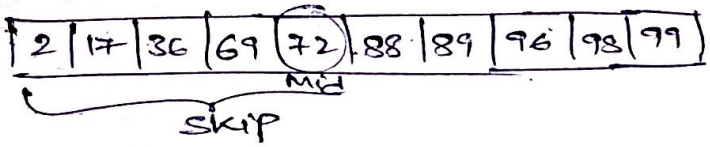
Mid=72.

Search Element=89

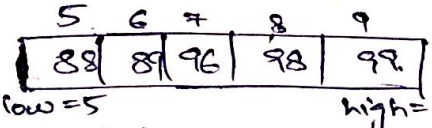
72 < 89 → key to be searched.

↙ Midvalue

NOTE:- Element is greater than the mid value, so we can skip left part of the data.



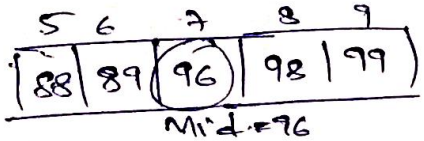
(3) Consider new data.



(4) Calculate Mid

$= \frac{5+9}{2} = \frac{14}{2} = 7$

(5)



Search element = 89

96 > 89

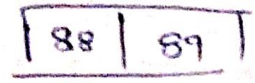
↙ Midvalue ↘ Search element.

(89) < (96) → Midvalue.

↙ Key to be Searched

NOTE:- Mid value is greater than the search element we can skip right part of the Mid.

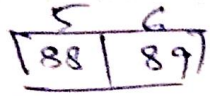
(6)



low=5 high=6

Calculate mid = $\frac{low+high}{2} = \frac{5+6}{2} = \frac{11}{2} = 5.5 \simeq 5$

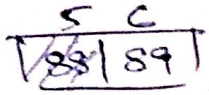
(7)



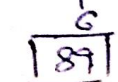
88 < Searchelement

88 < 89 (Skip the left part)

(8)



(9)



L=6 h=6

when low == high, we can say that Element found.

H.W:-

(1) 7, 34, 56, 67, 72, 87, 92, 94, 111, 115 ; Search element = 92.

0	1	2	3	4	5	6	7	8	9
7	34	56	67	72	87	92	94	111	115

low=0 {Search element = 92}. high=9.

(2) Calculate Mid, Mid = $\frac{low+high}{2} = \frac{0+9}{2} = \frac{9}{2} = 4.5 \simeq 4$.

(2)

7	34	56	67	72	87	92	94	111	115
0	1	2	3	4	5	6	7	8	9

Mid = 72.

Search element = 92.

92 < 92 → Search element

↙ Mid value

Element is greater than mid value, so skip the left part of mid.

0	1	2	3	4	5	6	7	8	9
7	34	56	67	72	87	92	94	111	115

skip

(3) Consider new data.

5	6	7	8	9
87	92	94	111	115

low=5 high=9.

(4) Calculate mid, Mid = $\frac{5+9}{2} = \frac{14}{2} = 7$.

(5)

87	92	94	111	115
5	6	7	8	9

Mid = 94.

92 < 94 ⇒ Mid value.

↙ Search element

Mid value is greater than the search element, we can skip the right part of the Mid.

5	6	7	8	9
87	92	94	111	115

Skip

(6)

5	6
87	92

low=5 high=6

Mid = $\frac{low+high}{2} = \frac{5+6}{2} = \frac{11}{2} = 5.5 \simeq 5$.

(7)

87	92
5	6

Mid

87 < Search element.

87 < 92 (Skip the left part)

(8)

6
92

low=6 high=6

(9) ⇒ When low=high, we can say that Element found.

Index = 6, Position = 7.

Binary search program:-

```
#include<iostream>
#include<conio.h>
#include<stdlib.h>
#define max_size 5
using namespace std;
int main()
{
int arr_search[max_size],i,element;
int low=0, high=max_size, mid;
cout<<"Simple C++ Binary search example-array\n";
cout<<"\nEnter"<<max_size<<"Elements for Searching:"<<endl;
for(i=0;i<max_size;i++)
    cin>>arr_search[i];
cout<<"\nYour Data:";
for(i=0;i<max_size;i++)
{
  cout<<"\t"<arr_search[i];
}
cout<<"\nEnter Elements to Search:";
cin>>element;
while(low<=high)
{
  mid=(low+high)/2;
  if(arr_search[mid]==element)
  {
    cout<<"\nSearch element:"<<element<<":found:position:"
    <<mid+1<<"\n";
    break;
  }
  else if(arr_search[mid]<element)
    low=mid+1;
  else
    high=mid-1;
}
if(low>high)
  cout<<"\nSearch element:"<<element<<":Not found\n";
getch();
}.
```

Fibonacci Search:-

H.W Ex:- 10, 22, 34, 42, 46, 57, 78, 82, 89, 93, 99. Search element = 99

Fibonacci search uses Fibonacci numbers to search an element in a sorted array.

Fibonacci numbers:- The first two numbers are '0' and '1' and each subsequent numbers in the series is equal to the sum of the previous two numbers.

$F(n) = n$ ——— when $n <= 1$

$F(n) = F(n-1) + F(n-2)$ —— when $n > 1$.

Fibonacci numbers are 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Step-1:- Find the smallest number >= n. let the number be fibm. Let the two Fibonacci numbers preceding it be m_1, m_2.

Step 2:- While the array has elements

(1) Compare x with last element of the range covered by m_2

-> Else if 'x' less than the element ~~more than~~ I move the 3 Fibonacci variables two Fibonacci down, indicating elimination of approximately rear two-third of the remaining array.

-> Else 'x' is greater than the element, move the three fibonacci variables one Fibonacci down. It indicates the elimination of the front one-third of the remaining array. Reset offset to Index.

(3) Since there might be single element remaining for comparison. Check if m_1 is 1. If yes, compare x with that remaining element. If match return index.

Ex:-

(1) 0 1 2 3 4
10, 20, 30, 40, 50, Search element = 20.

n = no. of elements = 5.

Now, write Fibonacci series upto n. => 0 1 1 2 3

Smallest fibonacci number >= n.

0 1 1 2 3 (5) <- fibm = 5.

m_2 m_1

preceding no.'s are m_1, m_2.

$m_1 = 3, m_2 = 2$, offset = 0

Calculate index = $i = \min(\text{offset} + m_2, n)$

fibm = 5, $m_1 = 3$, $m_2 = 2$.

i = min(0+2, 5)
= min(2, 5)
i = 2.

Now, compare a[2] with search element.
x = 20, a[2] = 30.
'20', x is less than indexed element.
Move two fibonacci variables down, it means:

0 1 1 2 3 5

Now fibm = 2.

0 1 1 2 3 5
fibm = 2, m1 = 1, m2 = 1

Calculate index, i = min(0+1, 5) = min(1, 5) = 1
a[i] = a[1] = 20
x = 20, a[i] = 20.
Element is found at index = 1.

(2) 16, 25, 39, 47, 86, 92, 98, 99
0 1 2 3 4 5 6 7.
n = 8, search element = 47.

Fibonacci sequence, 0 1 1 2 3 5 8 13.
Find smallest fibonacci num >= n.
fibm = 13, m1 = 8, m2 = 5.
i = min(offset + m2, n).
= min(0+5, 13) = min(5, 13) = 5.

a[i] = a[5] = 92, x = 47.
'47' is smaller than a[i], so down '2' variables.

0 1 1 2 3 5 8 13.
m2 m1 fibm

0 1 1 2 3 5 8 13.
m2 m1 M=5

i = min(offset+m2, n) = min(0+2, 13) = min(2, 13) = 2.
a[2] = 39.
x = 47, a[2] = 39.
47(x) is greater than a[2], down only one variable.

0 1 1 2 3 5
m2 m1 M=3

i = min(2+1, 13) = 3.
a[3] = 47.
So, element found at index 3.

offset = index (reset)
i = 2, offset = 2

0 1 2 3 4 5 6 7 8 9 10

10, 22, 34, 42, 46, 57, 78, 82, 89, 93, 99. (99).

n=11.

0, 1, 1, 2, 3, 5, 8, 13.

m_2 m_1 fibm.

i = min(5, 11) = 5.

a[5] = 57 < 99.

offset = 5.

0 1 1 2 3 5 8 13

m_2 m_1 fibm.

i = min(8, 11) = 8.

a[8] = 89 < 99.

offset = 8.

0 1 1 2 3 5 8 13.

m_2 m_1 fibm

i = min(10, 11) = 10.

a[10] = 99.

Bubble Sort:-

Bubble Sort:- In bubble sorting, consecutive adjacent pairs of elements in the array are compared with each other.

→ If the element at the lower index is greater than the element at the higher index, the two elements are interchanged so that the element is placed before the bigger one.

→ This process will continue till the list of unsorted elements exhausts.

→ This procedure is called Bubble sorting because elements 'bubble' to the top of the list.

Technique:-

(a) In pass 1, [A0] and [A1] are compared, then [A1] is compared with [A2], [A2] is compared with [A3] ... Finally A[N-2] is compared with A[N-1], Pass 1 involves n-1 comparisons and places the biggest element at the highest index of the array.

(b) In pass 2, [A0] and [A1] compared, then [A1] is compared with [A2], [A2] is compared with [A3], so on. Finally A[N-3] is compared with A[N-2]. Pass 2 involves n-2 comparisons and places the second highest element.

(c) In pass 3, [A0] and [A1] compared, then [A1] compared with [A2], so on. Finally A[N-4] is compared with A[N-3]. Pass 3 involves n-3 comparisons and places the third biggest element at the third highest index of the array.

(d) In pass n-1, [A0] and [A1] are compared, so that [A0] < [A1]

After this step, all the elements of the array are arranged in ascending order.

Ex:- A[] = {30, 52, 29, 87, 63, 27, 19, 54}.

Pass 1

(a) Compare 30 and 52, since 30 < 52, no swapping.

(b) Compare 52 and 29, since 52 > 29, swapping req.

30, 29, 52, 87, 63, 27, 19, 54.

(c) Compare 52 and 87, since 52 < 87, no swapping.

(d) Compare 87 and 63, since 87 > 63, swapping req.

30, 29, 52, 63, 87, 27, 19, 54.

(e) Compare 87 and 27, since 87 > 27, swapping req.

30, 29, 52, 63, 27, 87, 19, 54.

(f) Compare 87 and 19, since 87 > 19, swapping req.

30, 29, 52, 63, 27, 19, 87, 54.

(g) Compare 87 and 54, since 87 > 54, swapping req.

30, 29, 52, 63, 27, 19, 54, 87.

(Largest element placed at the highest index of the array)

Elements are still unsorted.

Pass 2

(a) Compare 30 and 29, since 30 > 29, swap. req.

29, 30, 52, 63, 27, 19, 54, 87.

(b) Compare 30 and 52, since 30 < 52, no swap.

(c) Compare 52 and 63, since 52 < 63, no swap.

(d) Compare 63 and 27, since 63 > 27, swap. req.

29, 30, 52, 27, 63, 19, 54, 87.

(e) Compare 63 and 19, since 63 > 19, swap. req.

29, 30, 52, 27, 19, 63, 54, 87.

(f) Compare 63 and 54, since 63 > 54, swap req.

29, 30, 52, 27, 19, 54, 63, 87.

Second largest element placed at 2nd highest position.

Pass3:- 29, 30, 52, 27, 19, 54, 63, 87.

(a) Compare 29 and 30, since 29<30, no swap.

(b) Compare 30 and 52, since 30<52, no swap.

(c) Compare 52 and 27, since 52>27, swap req
29, 30, 27, 52, 19, 54, 63, 87.

(d) Compare 52 and 19, since 52>19, swap req.
29, 30, 27, 19, 52, 54, 63, 87.

(e) Compare 52 and 54, since 52<54, no swap.
29, 30, 27, 19, 52, 54, 63, 87.
↳ Third highest element placed.

Pass4:- 29, 30, 27, 19, 52, 54, 63, 87.

(a) Compare 29 and 30, since 29<30, no swap.

(b) Compare 30 and 27, since 30>27, swap req.
29, 27, 30, 19, 52, 54, 63, 87.

(c) Compare 30 and 19, since 30>19, swap req.
29, 27, 19, 30, 52, 54, 63, 87.

(d) Compare 30 and 52, since 30<52, no swap.
29, 27, 19, 30, 52, 54, 63, 87.
↳ 4th highest placed.

Pass5:-

(a) Compare 29 and 27, since 29>27, swap req.
27, 29, 19, 30, 52, 54, 63, 87.

(b) Compare 29 and 19, since 29>19, swap req.
27, 19, 29, 30, 52, 54, 63, 87.

(c) Compare 29 and 30, since 29<30, no swap.
27, 19, 29, 30, 52, 54, 63, 87.
↳ 5th element placed.

Pass6:-

(a) Compare 27 & 19, since 27>19, swap req.
19, 27, 29, 30, 52, 54, 63, 87.

(b) Compare 27 and 29, since 27<29, no swap.
19, 27, 29, 30, 52, 54, 63, 87.
↳ 6th element placed.

Pass7:- Compare 19 and 27, since 19<27, no swap.
19, 27, 29, 30, 52, 54, 63, 87.
↳ 7th element placed.

19, 27, 29, 30, 52, 54, 63, 87.
Sorted List.

Bubble sort:-

5	4	3	2	1
0	1	2	3	4

n=5

```
for(i=0;i<n-1;i++)
{
for(j=0;j<n-1-i;j++)
{
 if(a[j]>a[j+1])
 {
  temp=a[j];
  a[j]=a[j+1]
  a[j+1]=temp
```

Insertion sort:-

45, 23, 99, 12, 38, 67, 74, 10, 22, 55, 9.

(1) 23, 45, 99, 12, 38, 67, 74, 10, 22, 55, 9

(2) 23, 45, 99, 12, 38, 67, 74, 10, 22, 55, 9

(3) ~~23 45 99~~ 12, 23, 45, 99, 38, 67, 74, 10, 22, 55, 9

(4) 12, 23, 38, 45, 99, 67, 74, 10, 22, 55, 9

(5) 12, 23, 38, 45, 67, 99, 74, 10, 22, 55, 9

(6) 12, 23, 38, 45, 67, 74, 99, 10, 22, 55, 9

(7) 10, 12, 23, 38, 45, 67, 74, 99, 22, 55, 9.

(8) 10, 12, 22, 23, 38, 45, 67, 74, 99, 55, 9

(9) 10, 12, 22, 23, 38, 45, 55, 67, 74, 99, 9

(10) 9, 10, 12, 22, 23, 38, 45, 55, 67, 74, 99.

Insertion Sort:- The insertionsort works very well when the no. of elements are very less.

→ This technique is similar to the way a librarian keeps the books in the shelf.

Initially all the books are placed in shelf according to their access number.

When a student returns the book to the librarian, he compares the access number of this book with all other books of access numbers and inserts it into the correct position so that all books are arranged in order w.r.to access numbers.

Method:-

Pass1:- The 1st element in the array is itself is sorted.

Pass2:- 2nd element is inserted either before or after 1st element, so that 1st and 2nd elements are sorted.

Pass3:- 3rd element is inserted into the proper place, i.e, before 1st & 2nd (or) bln 1st & 2nd (or) after 1st and 2nd elements.

Pass4:- Element is inserted into its proper place in 1st, 2nd, 3rd, so that 1st, 2nd, 3rd, 4th elements are sorted.

Eg:- a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9]
39, 9, 45, 63, 18, 81, 108, 54, 72, 36. n=10.
0 1 2 3 4 5 6 7 8 9.

compare (39<9) not true, so interchange.

Pass1:- 9 39 (45) 63 18 81 108 54 72 36
sorted. greater (so no prob).

Pass2:- 9 39 45 (63) 18 81 108 54 72 36.
greater (so no prob).

Pass3:- 9 39 45 63 (18) 81 108 54 72 36.
(greater than 9) (less than 39) (less than 45) (less than 63)

So, '18' will be placed bln 9 & 39

Pass4:- 9 18 39 45 63 (81) 108 54 72 36
(greater, so no prob).

Pass5:- 9 18 39 45 63 81 (108) 54 72 36.
(greater, so no prob).

Pass6:- 9 18 39 45 63 81 108 (54) 72 36
(greater than 45) (less than 63) (less than 81) (less than 108)

So '54' placed bln 45 & 63.

Pass7:- 9 18 39 45 54 63 81 108 (72) 36
(greater than 63.) (less than 81) (less than 108)

So '72' placed bln 63 & 81.

Pass8:- 9 (18) 39 45 54 63 72 81 108 (36)
greater than 18. less than 39 less than 45 less than 54 less than 63 less than 72 less than 81 less than 108

So '36' placed bln 18 & 39.

Pass9:- 9 18 36 39 45 54 63 72 81 108.

Elements in sorted order.

Insertion sort

```
for (K=1; K<n; K++)
{
 key = arr[K];
 ptr = K-1;
 while (ptr >= 0 && arr[ptr] > key)
 {
   arr[ptr+1] = arr[ptr]
   ptr = ptr-1;
 }
 arr[ptr+1] = key;
}
```

Hash :- Structuring the data using tables.
Index is calculated by Hash % Tablesize
collision happens if 22%10=2, 12%10=2.
then some resolution techniques.

=

Selection Sort:- It is a simple sorting algorithm.
Consider an array with N elements, first find the smallest value in the array and place it in the first position. Repeat this procedure until the entire array is sorted.

Method:-
In pass 1, find the smallest value position in the array and then swap a[pos] and a[0], Thus a[0] is sorted.
In pass 2, find the position of the second smallest value in the array and then swap a[pos] and a[1], Thus a[1] is sorted.
This procedure is repeated untill all the elements get sorted.

Ex:- 39 (9) 81 45 90 27 72 18
small.

Pass 1:- 9 39 81 45 90 27 72 (18) ← next smallest element.
Make a comparison.
& then swap (39>18)

Pass 2:- 9 18 81 45 90 (27) 72 39.
Swap
next smallest element (81>27)

Pass3:- 9 18 27 45 90 81 72 (39) ← next smallest element.

Swap

Pass4:- 9 18 27 39 90 81 72 (45) ← next smallest element.

Swap

Pass5:- 9 18 27 39 45 81 (72) 90

Swap

Pass6:- 9 18 27 39 45 72 (81) 90

(✓)

Pass7:- 9 18 27 39 45 72 81 90

(Sorted).

Hashing

Hashing is the process of mapping large amount of data items to a

Swap

Pass6:- 9 18 27 39 45 72 (81) 90

(✓)

Pass7:- 9 18 27 39 45 72 81 90

(Sorted).

Hashing

Hashing is the process of mapping large amount of data item to a smaller table with the help of hashing function.

Adv:- This method is used to handle the vast amount of data.

Hash table:- Hash table is a data structure used for storing and retreiving data very quickly.

Insertion of data in the hash table is based on the key value.

Every entry in the hash table is associated with some key

Eg:- Storing an employee record in the hash table, employee id will work as a key.

Using the hash key the required piece of data can be searched in the hash table by few (or) more comparisons. The searching time dependent upon the size of the hash table.

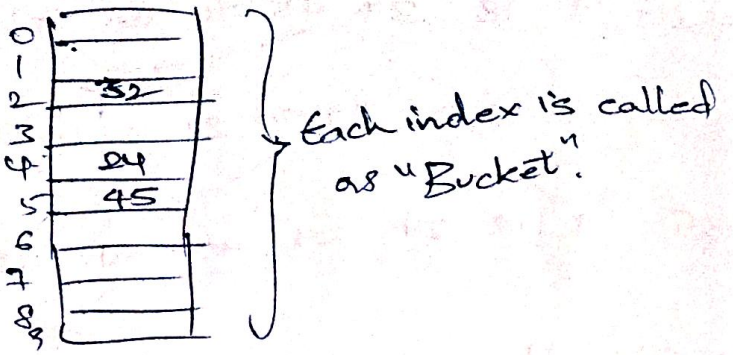
Hash Function:- Hash function is a function which is used to put the data in the hash table. Hence one can use the same function to retreive the data from the hash table. Thus hash function is used to implement the hash table. The integer returned by the hash function is called hash key.

Hash key(H) = Key % Table size.

Eg:- 24, 32, 45 :-

Table size = 10.

Hash key (H) = 24 % 10 = 4
Hash key (H) = 32 % 10 = 2
Hash key (H) = 45 % 10 = 5



Types of Hash Functions :- There are various types of hash function that are used to place the record in the hash table.

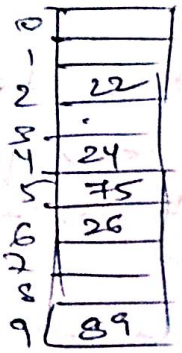
(1) Division Method (2) Mid-square Method.
(3) Multiplicative Hash function (4) Digit folding.

Division Method :- The hash function depends on the remainder of division.

Eg:- 22, 24, 26, 89, 75, ... Table size = 10.

H(key) = key % Tablesize.

H(22) = 22 % 10 = 2
H(24) = 24 % 10 = 4
H(26) = 26 % 10 = 6
H(89) = 89 % 10 = 9
H(75) = 75 % 10 = 5



Mid-square method :- In mid square method, the key is squared and middle (or) mid part of the result is used as index.

Consider key = 3111 then, find square of a given number.

$(3111)^2 = 9684321$

↳ Middlenum.

The Hash index is 483, then place 3111 into 483 position.

Multiplicative Hash Function :- In this function, the given record is multiplied with some constant value.

H(key) = floor(p * (key * A))
↳ (Author donald Knuth suggested A = 0.618033...)

if Key = 107 & P = 50 then.

H(key) = floor(50(107 * 0.618033))

= floor(3306.481)

H(key) = 3306

At location 3306, the record 107 will be placed.

Digit Folding :- The key is divided into separate parts and using simple function. These parts are combined to procedure the hash key.

For ex:- num = 1237654.

first divide the given number into parts 123, 765, 4

Now combine, 123+765+4 = 892.

'892' will be the index to place the record i.e, 1237654

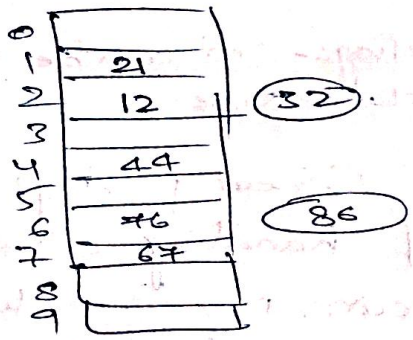
Collision :- The hash function is a function that returns the key value, using which the record can be placed in the hash table.

The function needs to be designed very carefully and it should not return the same hash key address for two different records.

Def:- The hash function returns the same hash key for more than one record is called collision and two same hash keys returned for different record is called collision.

Ex:- 12, 21, 32, 44, 76, 67, 86, ...

Assume Table Size = 10.



To avoid the collision, we have some resolution techniques.

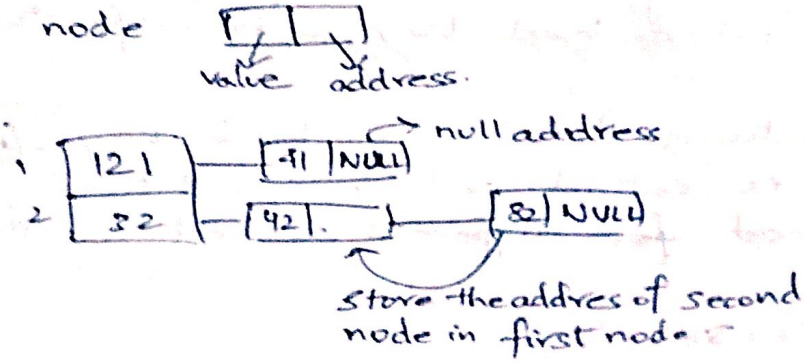
Collision Resolution Techniques :- If collision occurs then it should be handled by some applying some techniques.

(1) Chaining.
(2) Open addressing (Linear Probing).
(3) Quadratic probing.
(4) Double hashing.

Chaining:- In collision handling method, chaining is a concept which introduces an additional field with data. i.e; chain.

Separate chain table is maintained for colliding data. When collision occurs then a linked list (chain) is maintained at the home bucket.

Linked lists used to connect values.



In Quadratic probing:- 'm' can be table size (or) nearest prime number less than table size.

Open-addressing:- (Linear probing).

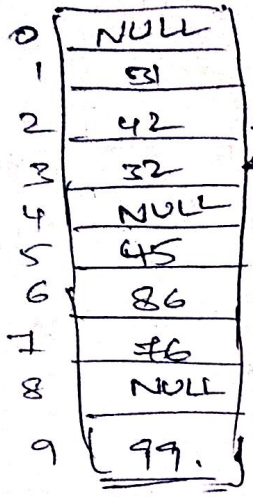
Easiest method of handling collision.

When collision occurs, i.e; when two records demand for the same bucket in the hash table then collision can be solved by placing "the second record linearly down whenever the empty bucket is found".

Eg: Records = 31, 42, 86, 99, 45, 32, 76, ...

H(key) = key % Tablesize.

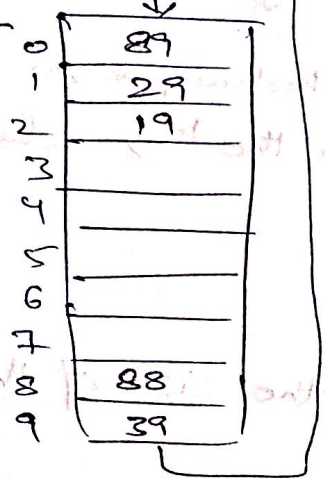
H(31) = 31 % 10 = 1
H(42) = 42 % 10 = 2
H(86) = 86 % 10 = 6
H(99) = 99 % 10 = 9
H(45) = 45 % 10 = 5
H(32) = 32 % 10 = 2
H(76) = 76 % 10 = 6



"32" but already filled with some other num. So go linearly down, next bucket is free, 32 into 3rd bucket.

Problem with Linear problem is primary clustering. It is a process in which a block of data is formed in the hash table when collision resolved.

Eg:- 89, 89, 29, 19, 88, ... Table size = 10



Quadratic Probing:- Quadratic probing operates by taking the original hash value and adding recursive hash values of an arbitrary quadratic polynomial to the starting value.

$$H(key) = (Hash(key) + i^2) \% m$$

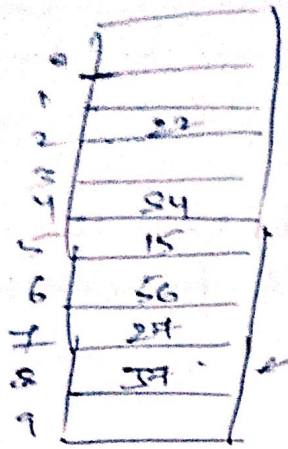
Eg:- 15, 22, 27, 84, 66, 37, ... Table size = 10

H(15) = 15%10 = 5
H(22) = 22%10 = 2
H(27) = 27%10 = 7
H(84) = 84%10 = 4
H(66) = 66%10 = 6
H(37) = 37%10 = 7 (Collision).

7th bucket has already an element. Hence apply quadratic probing.

$H(key) = (H(key) + i^2) \% m$ → 'M' can be table size or nearest prime number less than table size.

Consider i=0, then H(key) = (37+0²) % 10 = 7.
i=1, H(key) = (37+1²)%10 = 38 % 10 = 8.
8th bucket is empty so, simply insert 37 into 8th bucket.
If 8th bucket is not empty, then continue with i=2,3,...



'37' is inserted into 8th bucket by using "Quadratic Probing".

Double Hashing:- Double Hashing is technique in which a second hash function is applied to the key when a collision occurs.

$H_1(key)$ = Key % Tablesize.
$H_2(key)$ = M − (key % M).

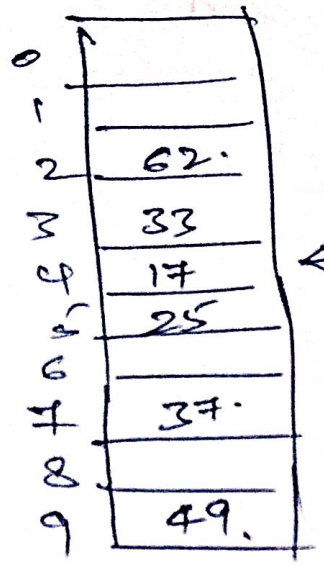
M is a prime number smaller than the size of the table.

Eg:- 37, 49, 25, 62, 33, 17, ...
Table size = 10.

H(37) = 37 % 10 = 7.
H(49) = 49 % 10 = 9
H(25) = 25 % 10 = 5.
H(62) = 62 % 10 = 2
H(33) = 33 % 10 = 3
H(17) = 17 % 10 = 7 ← Collision occured

$H_2(key) = M - (key \% M)$.
↓
Table size 10, Take 'M' as nearest prime no. to the table size, M=7.

M = 7.
$H_2(17) = 7 - (17 \% 7) = 7 - 3 = 4$.
$H_2(17) = 4$.
Insert ~~into~~ '17' into 4th bucket.



← '17' inserted in 4th bucket by applying Double hashing.

Rehashing:- It

Quick Sort:- also known as partition exchange sort.

Quick sort algorithm works by divide and conquer strategy. to divide a single unsorted array into two smaller sub-arrays.

Algorithm:- (1) Select an element pivot from the array elements

(2) Rearrange the elements in the ~~ar~~ array in such a way that elements that are less than the pivot appear before the pivot and all the elements greater than the pivot element come after it.

(3) After such a partitioning, the pivot is placed in its final position. This is called partition operation.

(4) Recursively sort the two sub-arrays & thus obtained one with sublist of values smaller than that of the pivot element & the other having higher value elements.

Technique:-

(1) Set the index of the first element in the array to loc and left variables. Also set the index of the last elements of the array to the right variable.

loc = 0, left = 0, right = n-1.

(2) Start from the element pointed by right and scan the array from right to left, comparing each element on the way with the element pointed by the variable loc.

i.e., a[loc] should be less than a[right].

→ If this is the case, then simply continue comparing until right becomes equal to loc. Once right = loc., it means the pivot has been placed in its correct position.

→ However, if at any point, we have a[loc] > a[right], then interchange the two values & jump to step 3.

→ set loc = left.

Eg:

27	10	36	18	25	45	n=6
0	1	2	3	4	5	
loc, left					right	

←

→ Initially scan from right to left. a[loc] < a[right]

27	10	36	18	25	45
loc, left				right	

decrement the value of right.

→ a[loc] > a[right], interchange two values, loc = right

25 10 36 18 27 45.
left → right loc.

→ From left to right a[loc] > a[left]

25 10 36 18 27 45 — increment the value of left
left → right loc.

25 10 36 18 27 45
left right loc

a[loc] < a[left], interchange values set loc = left.

25 10 ~~27~~ 18 36 45.
left loc. right

Scan from R to L, since a[loc] < a[right] — decrement the value of right.

25 10 27 18 36 45.
left right.
loc

a[loc] > a[right], interchange & loc = right.

25 10 ~~18~~ ~~27~~ 36 45.
left right
loc.

Scan from L to R, a[loc] > a[left], increment the value of left.

25 10 18 (27) 36 45.
left sub array — left loc right. — right sub array.
sorted position.

25 10 18
left right. a[loc] < a[right], interchange.
loc

18 10 25.
left right.
loc. → a[left] < a[loc], increment.

18 10 25
left right
loc. → a[left] < a[loc]. increment

18 10 (25) → sorted position.
left sub array — left right loc

18 10
left right a[loc] < a[right], interchange
loc
←

10 18 a[left] > a[loc], increment left.
left loc right

(10) (18) ← Sorted position
left loc right
left loc right

36 45 a[loc] < a[right], decrement right.
left loc right

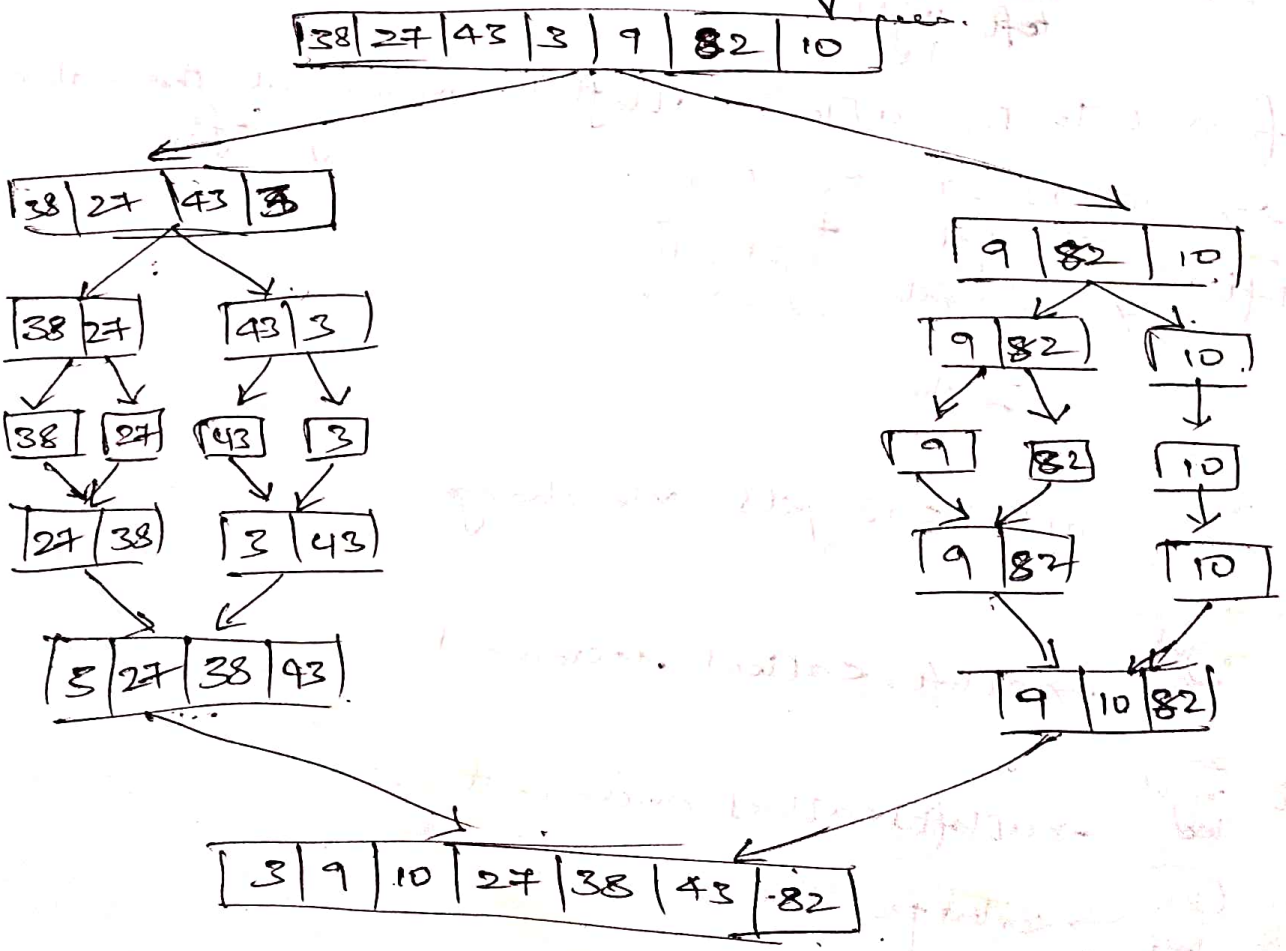
(36) ← Sorted position.
left loc right

45
left right loc

10	18	25	27	36	

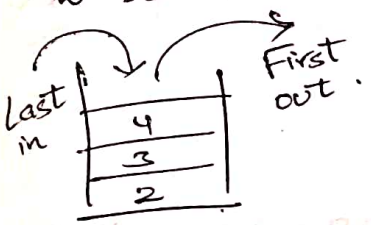
10	18	25	27	36	45

Merge sort:- Once the size becomes 1, the merge process comes into action and starts merging arrays back till complete array is merged.



Stack :- UNIT2

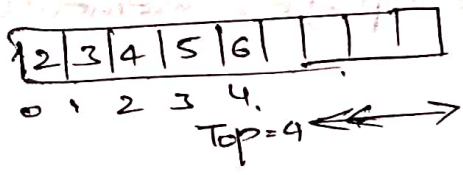
Stack is a linear data structure which stores elements in an ordered manner like a pile of plates.
The elements in a stack are added and removed only from one end which is called the "Top".
Stack is called a "LIFO" (Last in First out) data structure, as the element that was inserted last is the first one to be taken out.



Uses of Stack :- In order to keep track of returning point of each active function.

Array Representation of a stack :-

→ Stack can be represented as a linear array.
→ Every stack has a variable called 'Top' associated with it which is used to store the address of the top most element of the stack.
→ There is another variable 'MAX' to store maximum no. of elements that the stack can hold.
If TOP=NULL, it indicates that the stack is empty and if TOP = MAX-1, means stack is full.



⟷ We can insert 4 more elements.

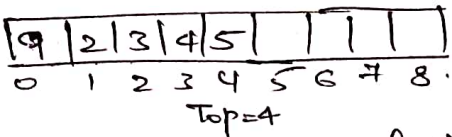
Operations of a Stack :-

(1) push() :- push operation adds an element to the top of the stack.
(2) pop() :- pop operation removes the element from the top of the stack.
(3) peek() :- peek operation returns the value of top most element of the stack.

Push operation:- It is used to insert an element into the stack. The new element is added to the topmost position of the stack. ~~Beforing~~

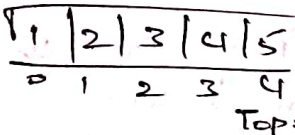
Before inserting the value, we must first check if TOP=MAX-1 because in this case, stack is full and no more insertions can be done.

If the stack is full, in that case if ~~you~~ an attempt is made to insert a value in a stack that is already full, an overflow occured.

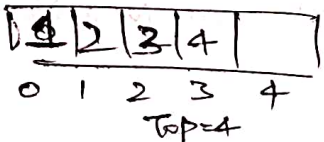


To insert an element "9", we first check TOP=MAX-1, if it is true means stack is full we can't insert, if the condition false, we increment "top" from 4 to 5th index, then we simply insert the element into the stack.

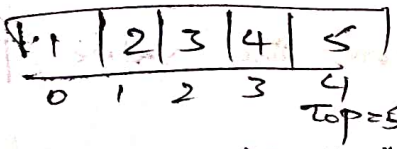
Pop operation:- It is used to delete the topmost element from the stack. Before deleting the value, we must first check if TOP=NULL, in this case stack is empty. If an attempt is made to delete the element from stack, stack underflow occurs.



To delete the topmost element, we first check if TOP=NULL, condition false means, there are some elements in stack, then decrement the top, i.e., elements gets deleted from stack.



Peek Operation:- It returns the value of the topmost element of the stack without deleting it from the stack. In peek operation first check if the stack is empty i.e., TOP=NULL, means no elements in stack



Peek operation will return "5", it is the value of the topmost element of the stack.

Applications of stack :-

(1) Reversing a list.
(2) Parentheses checker.
(3) Conversion of an infix expression into a postfix expression.
(4) Evaluation of a postfix expression.
(5) Conversion of an infix expression into a prefix expression.
(6) Evaluation of a prefix expression.
(7) Recursion.