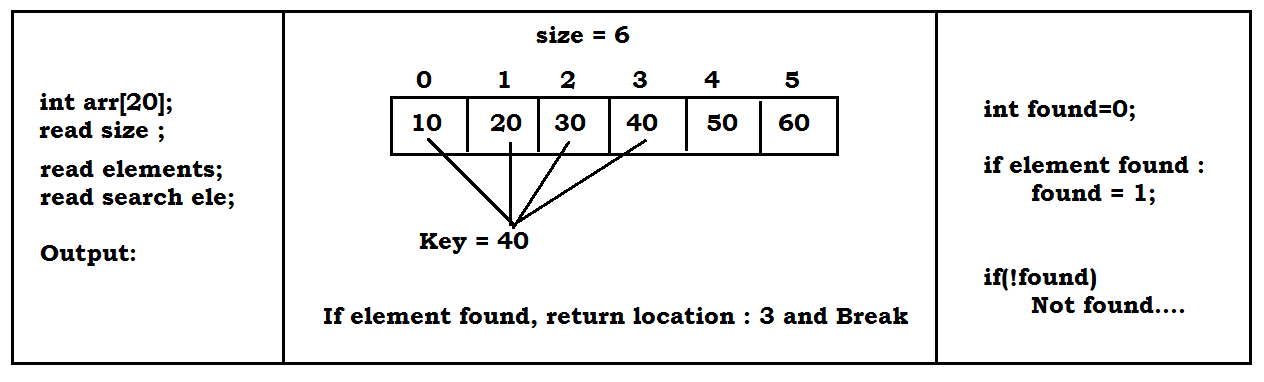
**Linear search:**

* Finding an element in the given array.
* Every time we compare the key element with index element.
* If matches, returns the location as element found.
* If not present, control returns with Error message.



**Code:**

#include<stdio.h>

int main()

{

int arr[50], size, i, key, found=0;

printf("Enter length of elements : ");

scanf("%d", &size);

printf("Enter %d elements : \n", size);

for(i=0 ; i<size ; i++)

{

scanf("%d", &arr[i]);

}

printf("Enter element to be searched :");

scanf("%d", &key);

for(i=0 ; i<size ; i++)

{

if(key == arr[i])

{

printf("Element %d found @ loc : %d \n", arr[i], i);

found=1;

break;

}

}

if(!found)

printf("Element not found \n");

return 0;

};

**Linear search using recursion:**

#include<stdio.h>

int search(int[], int, int, int);

int main()

{

int arr[50], size, i, key, found=0;

printf("Enter length of elements : ");

scanf("%d", &size);

printf("Enter %d elements : \n", size);

for(i=0 ; i<size ; i++)

{

scanf("%d", &arr[i]);

}

printf("Enter element to be searched :");

scanf("%d", &key);

found = search(arr, 0, size, key);

if(!found)

printf("Element not found \n");

return 0;

}

int search(int arr[], int cur, int end, int key)

{

if(cur==end)

{

return 0;

}

if(key==arr[cur])

{

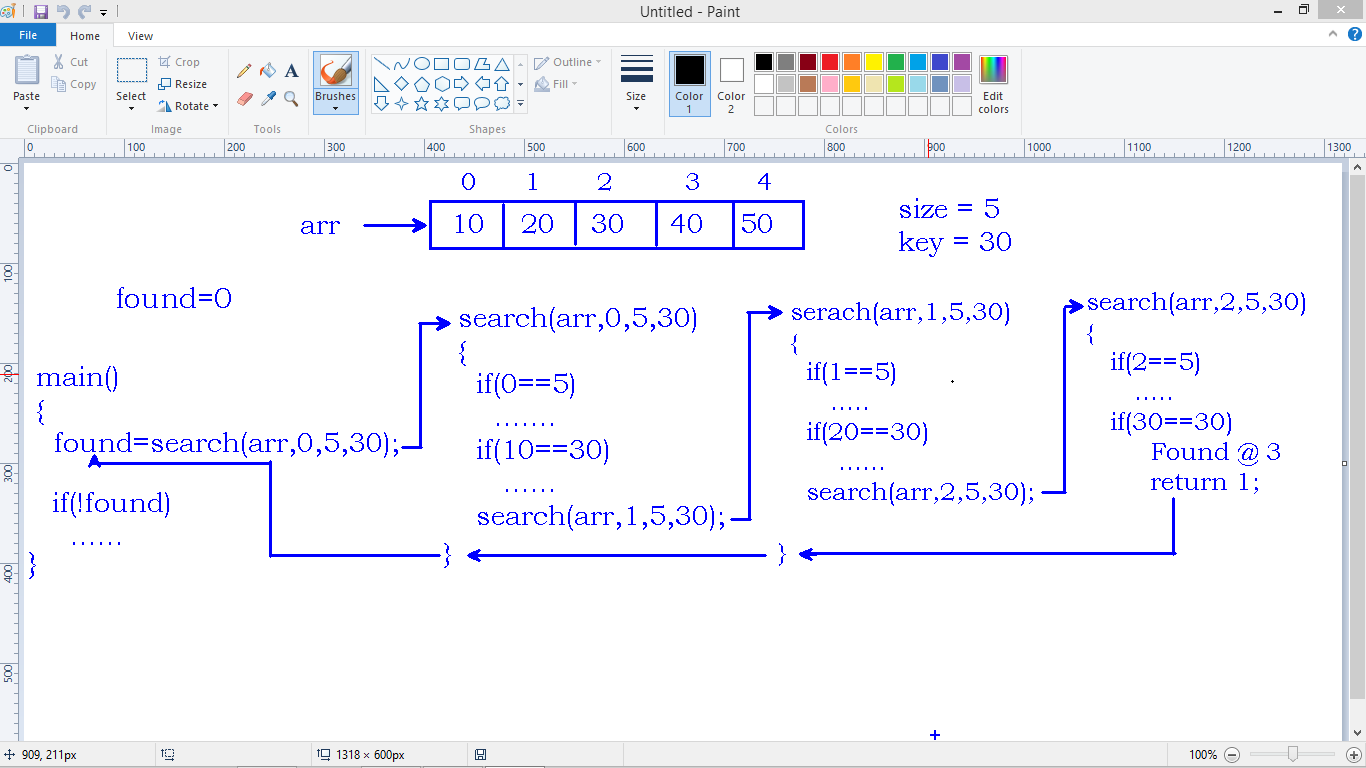
printf("Found @ loc : %d\n", cur);

return 1;

}

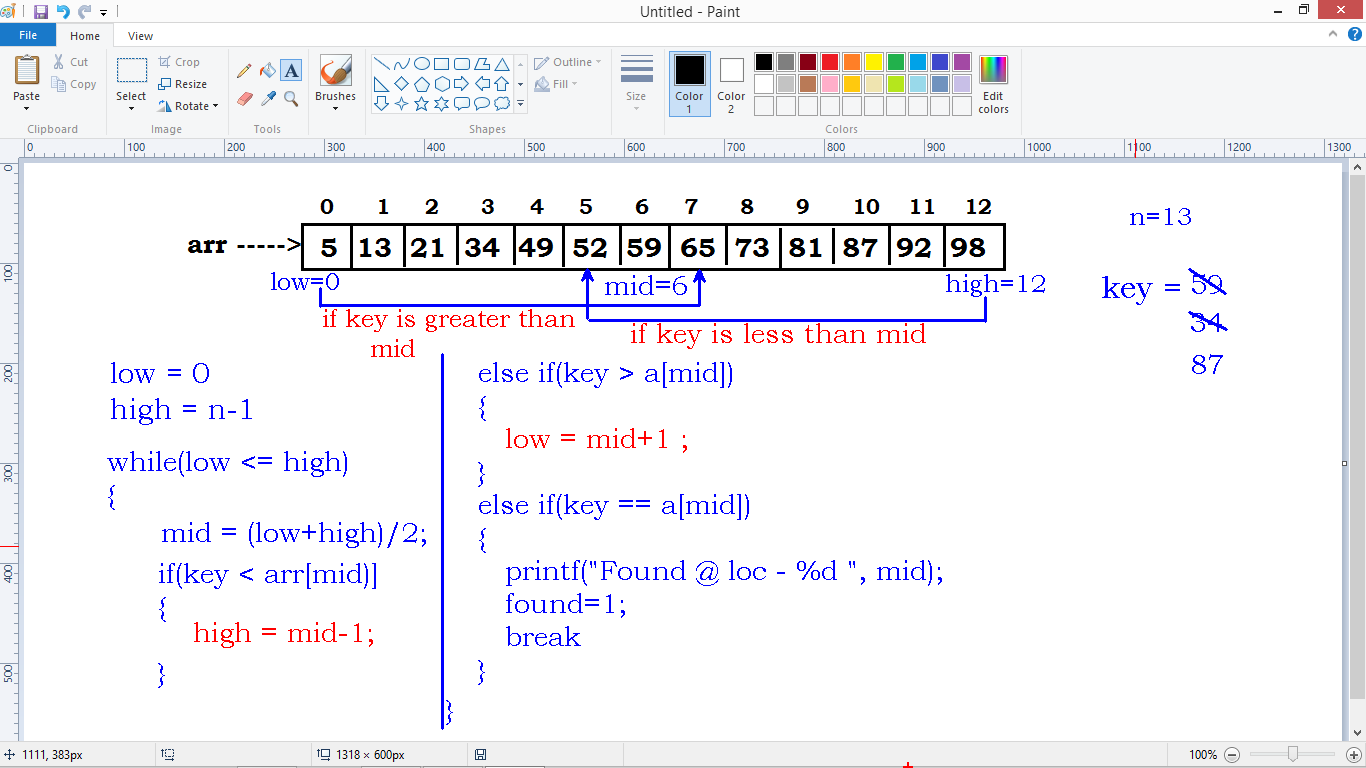
search(arr, cur+1, end, key);

}

****

**Binary Search:**

* It is another search technique to search for an element in the array.
* In binary search, every time we find the mid element and compare with the element to be searched.
* If element matches, we return mid location index as element index.
* If the element is less than the mid location element, search continues only in left part of mid location.
* If the element is higher than the mid location element, search continues in the right side part of array from mid location.
* This process continues until element search.
* **Note: We can apply binary search only on sorted array.**



#include<stdio.h>

int main()

{

int a[50], n, i, key, found=0, low, mid, high;

printf("Enter length of elements : ");

scanf("%d", &n);

printf("Enter %d elements : \n", n);

for(i=0 ; i<n ; i++)

{

scanf("%d", &a[i]);

}

printf("Enter element to be searched :");

scanf("%d", &key);

low=0;

high=n-1;

while(low<=high)

{

mid=(low+high)/2;

if(key<a[mid])

{

high=mid-1;

}

else if(key>a[mid])

{

low=mid+1;

}

else if(key==a[mid])

{

printf("Found @ loc : %d \n", mid);

found=1;

break;

}

}

if(!found)

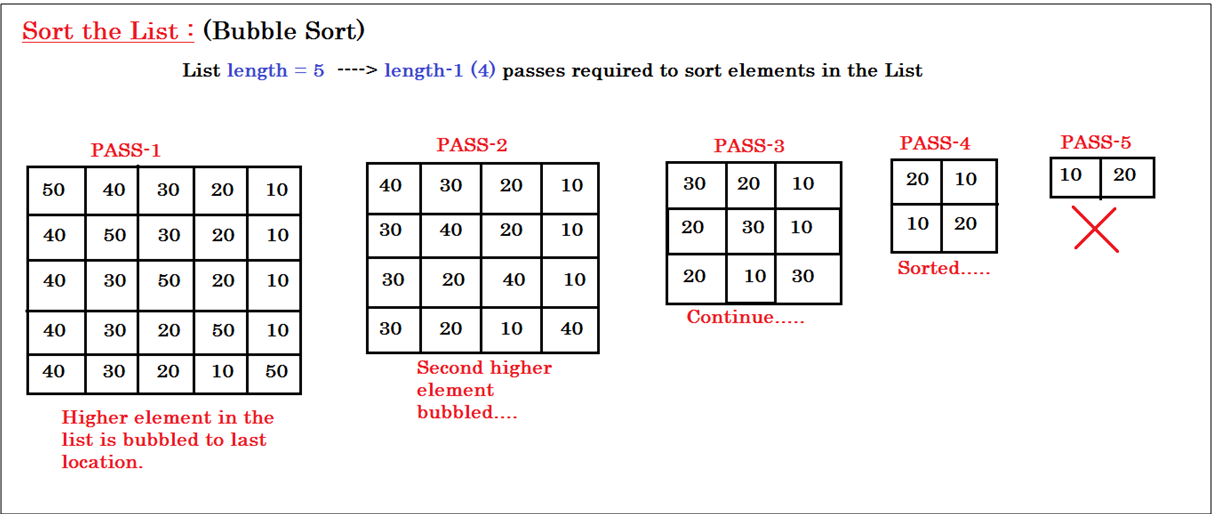
printf("Element not found \n");

return 0;

};

**Bubble sort**

* One of the easiest sorting algorithms to arrange the elements of array either in ascending or descending order.
* The concept of comparing index element with the next element and swapping them if required.
* For each pass, the highest element in the array is bubbled to the last location.
* This process will continue until all the elements get sorted.



#include<stdio.h>

#include<conio.h>

void bubble\_sort(int[] , int);

int main()

{

int arr[50] , n , i ;

printf("Enter number of elements : ");

scanf("%d", &n);

for( i=0 ; i<n ; i++ )

{

arr[i] = rand()%100 ;

}

printf("Array elements before sort : \n");

for( i=0 ; i<n ; i++ )

{

printf("%d\t",arr[i]);

}

printf("\n\n");

bubble\_sort(arr , n) ;

printf("Array elements after sort : \n");

for( i=0 ; i<n ; i++ )

{

printf("%d\t",arr[i]);

}

printf("\n\n");

return 0;

}

void bubble\_sort(int a[ ], int n)

{

int i, j, temp;

for (i=0 ; i<n-1 ; ++i)

{

for(j=0 ; j<n-1-i ; ++j)

{

if (a[j]>a[j+1])

{

temp = a[j+1];

a[j+1] = a[j];

a[j] = temp;

}

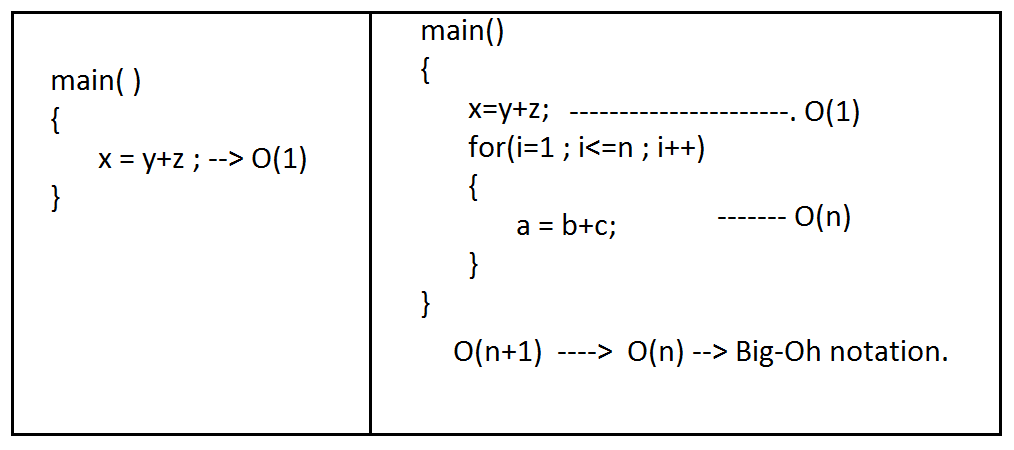
}

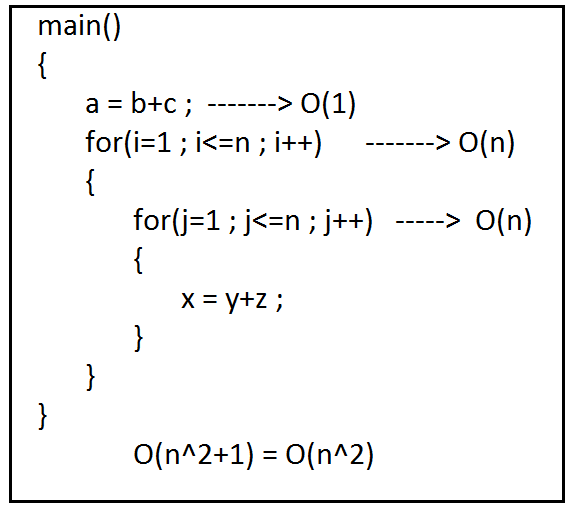
}

}

**Time complexity:**

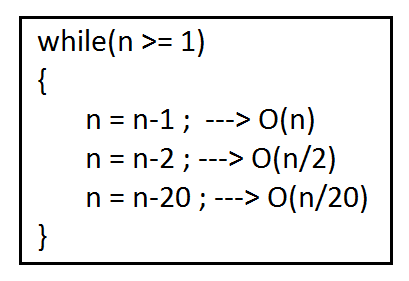
* In computer science, analysis if algorithms is crucial part.
* It is important to find the most efficient algorithm to solve a problem.
* To solve a problem, we have many algorithms. For example sorting an array of elements.
* The challenge here is to choose the most efficient one among available.
* The time complexity is the number of operations an algorithm performs to complete a task.
* The algorithm that performs the task in the smallest number of operations considered as efficient one.

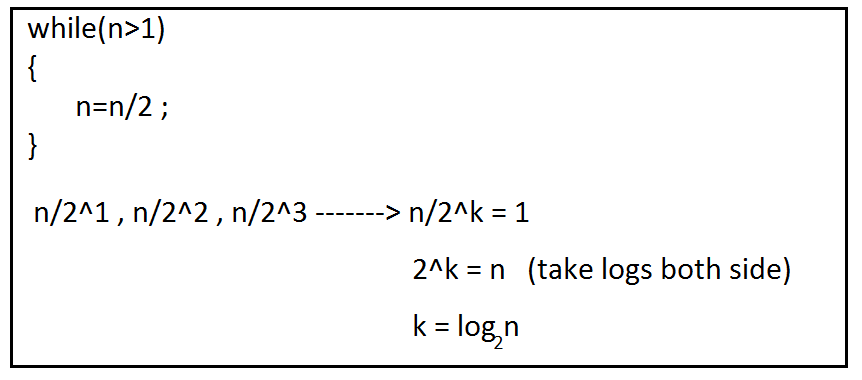


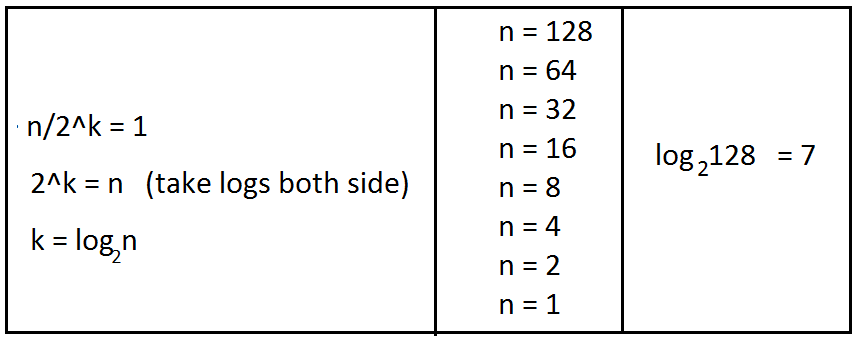


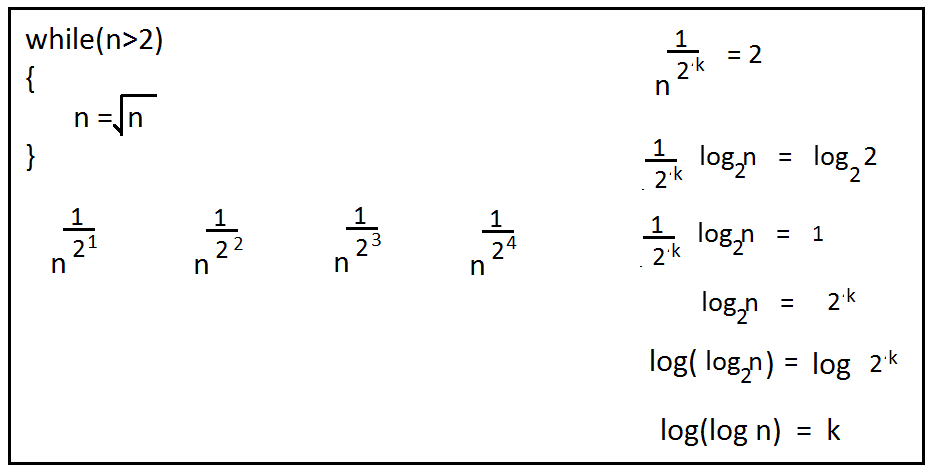
N^2 + N + 1 🡪 O(N^2)

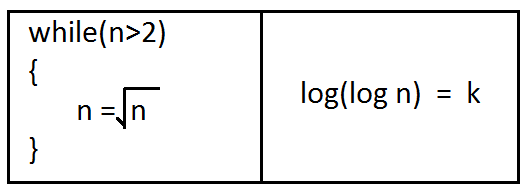
100N^2 + 50^N + 1

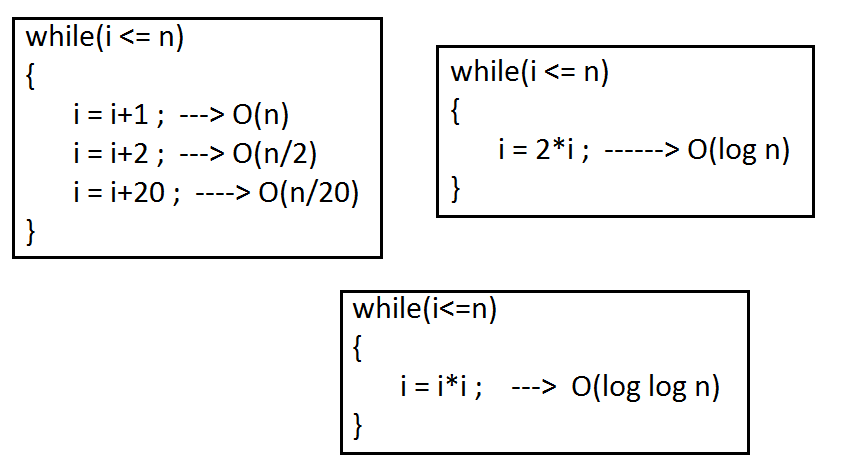


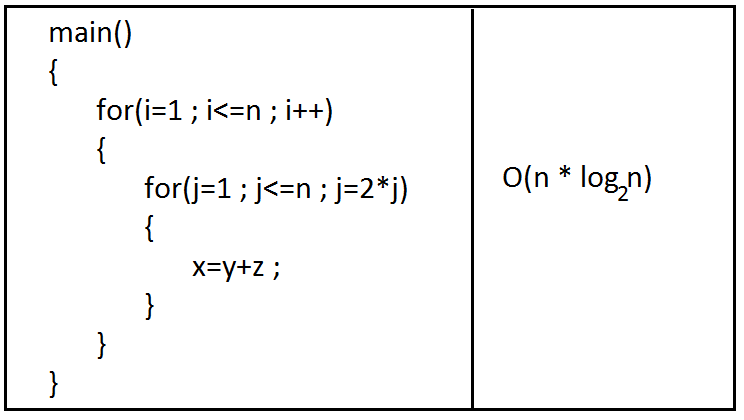


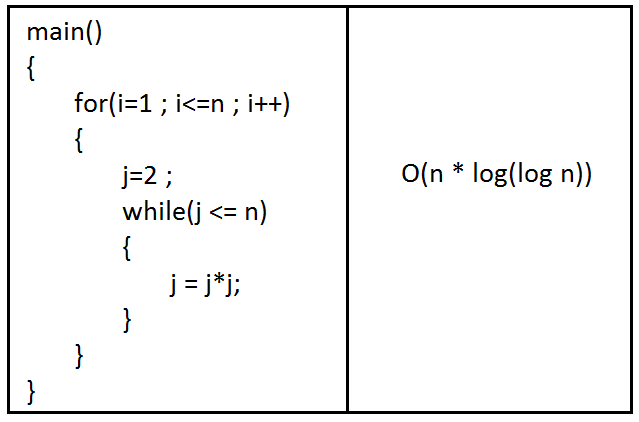


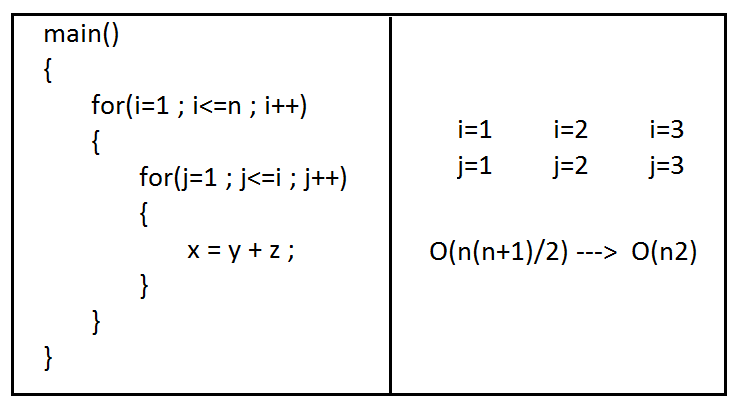


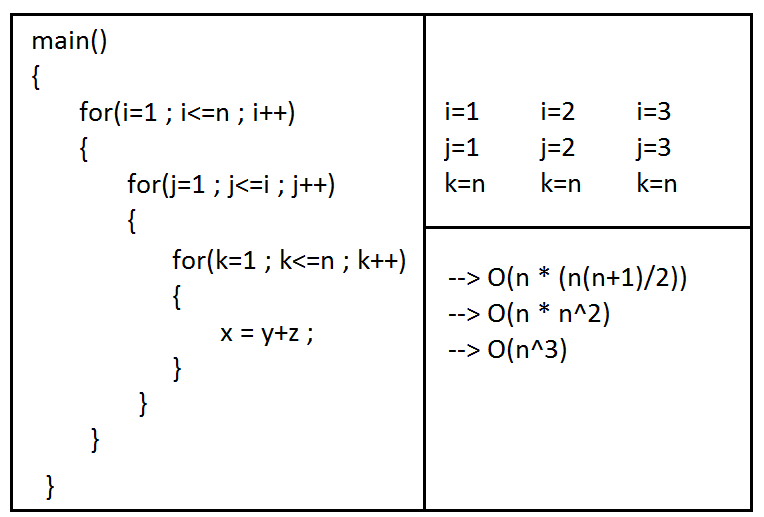


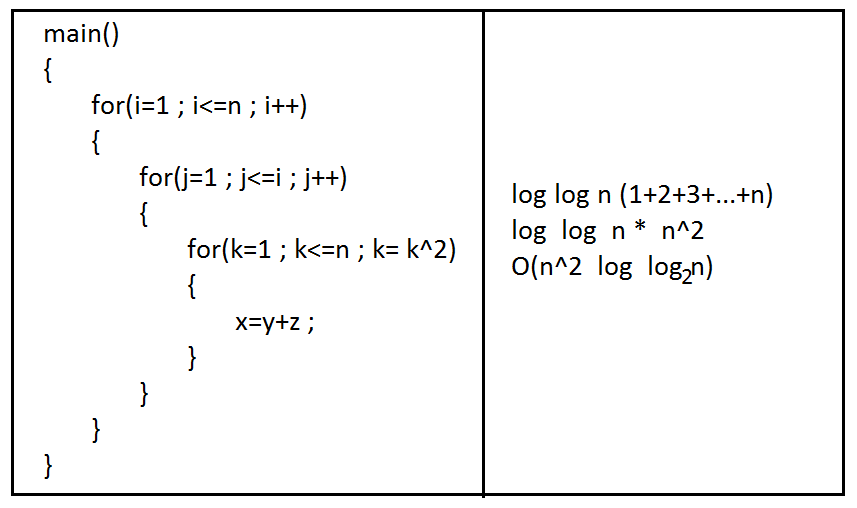


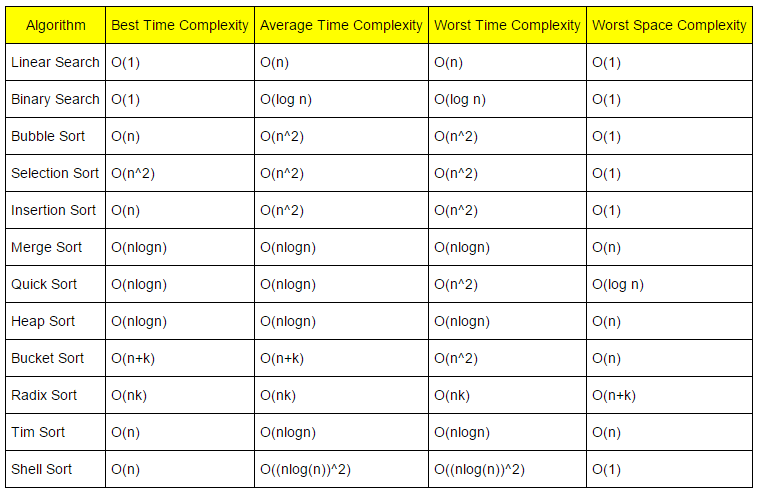












**Asymptotic Analysis**

In this tutorial, you will learn what asymptotic notations are. Also, you will learn about Big-O notation, Theta notation and Omega notation.

The efficiency of an algorithm depends on the amount of time, storage and other resources required to execute the algorithm. The efficiency is measured with the help of asymptotic notations.

An algorithm may not have the same performance for different types of inputs. With the increase in the input size, the performance will change.

The study of change in performance of the algorithm with the change in the order of the input size is defined as asymptotic analysis.

**Asymptotic Notations**

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case.

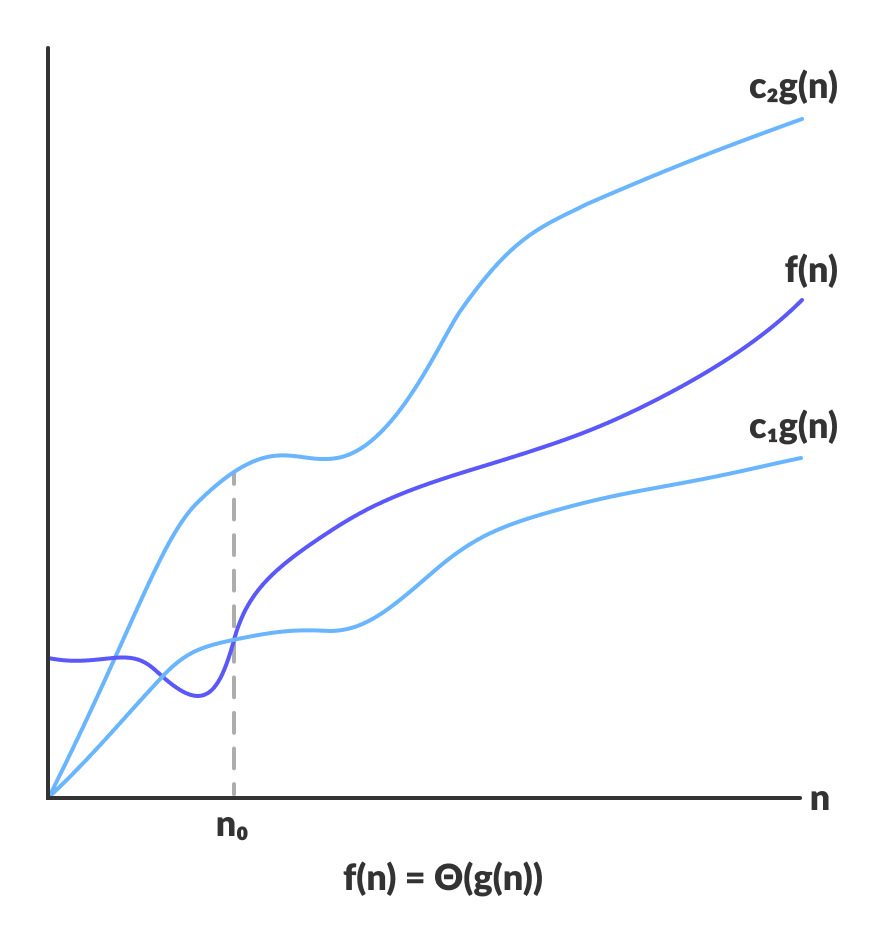
But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.

When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

There are mainly three asymptotic notations: Theta notation, Omega notation and Big-O notation.

**Theta Notation (Θ-notation)**

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average case complexity of an algorithm.

Theta bounds the function within constants factors

For a function g(n), Θ(g(n)) is given by the relation:

Θ(g(n)) = { f(n): there exist positive constants c1, c2 and n0

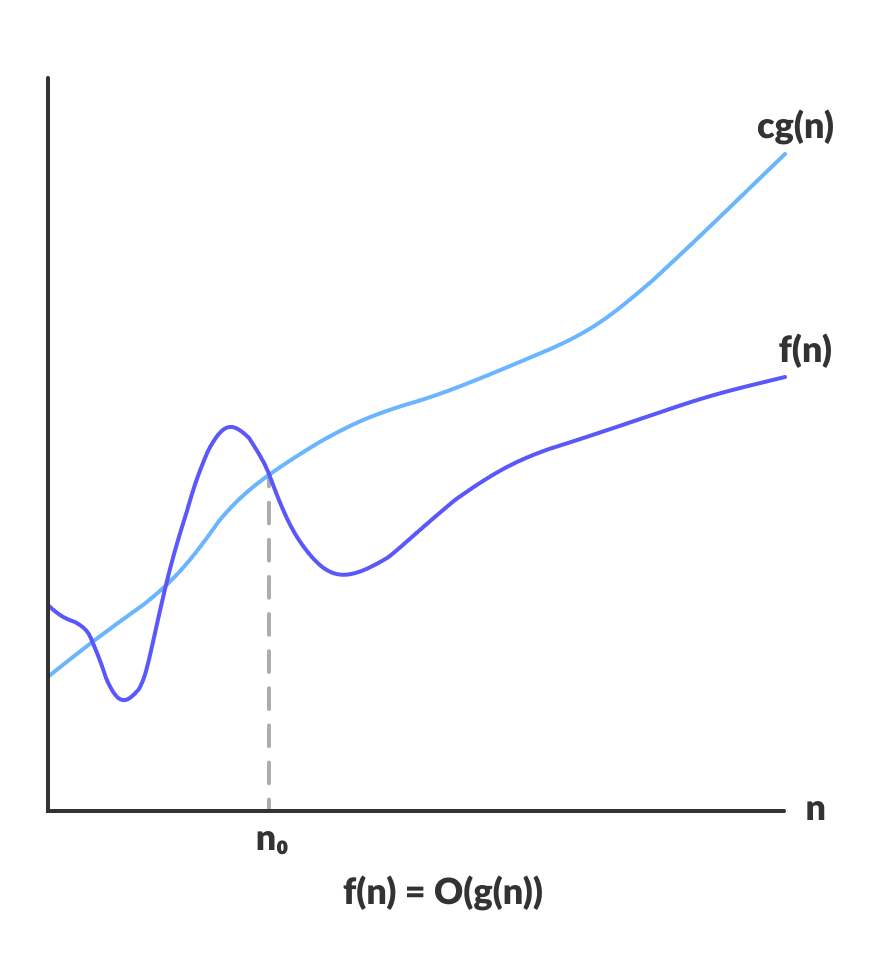
such that 0 ≤ c1g(n) ≤ f(n) ≤ c2g(n) for all n ≥ n0 }

The above expression can be described as a function f(n) belongs to the set Θ(g(n)) if there exist positive constants c1 and c2 such that it can be sandwiched between c1g(n) and c2g(n), for sufficiently large n.

If a function f(n) lies anywhere in between c1g(n) and c2g(n) for all n ≥ n0, then f(n) is said to be asymptotically tight bound.

**Big-O Notation (O-notation)**

Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst case complexity of an algorithm.

Big-O gives the upper bound of a function

O(g(n)) = { f(n): there exist positive constants c and n0

such that 0 ≤ f(n) ≤ cg(n) for all n ≥ n0 }

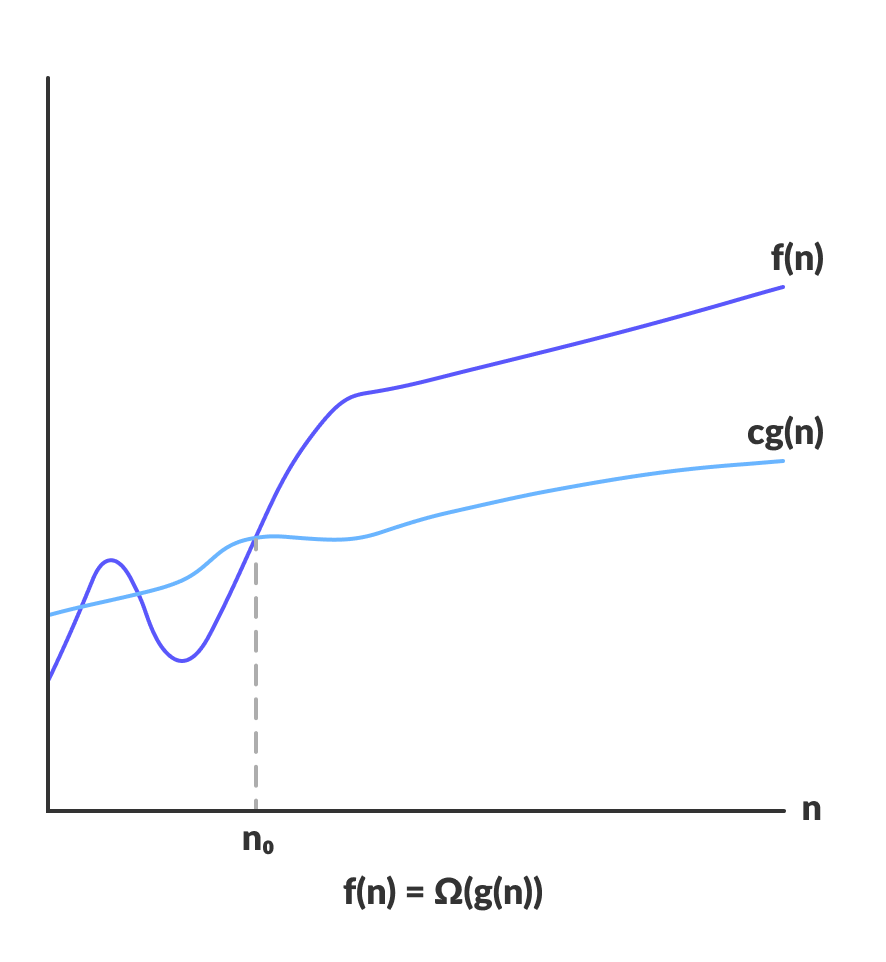
The above expression can be described as a function f(n) belongs to the set O(g(n)) if there exists a positive constant c such that it lies between 0 and cg(n), for sufficiently large n.

For any value of n, the running time of an algorithm does not cross time provided by O(g(n)).

Since it gives the worst case running time of an algorithm, it is widely used to analyze an algorithm as we are always interested in the worst case scenario.

**Omega Notation (Ω-notation)**

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides best case complexity of an algorithm.

Omega gives the lower bound of a function

Ω(g(n)) = { f(n): there exist positive constants c and n0

such that 0 ≤ cg(n) ≤ f(n) for all n ≥ n0 }

The above expression can be described as a function f(n) belongs to the set Ω(g(n)) if there exists a positive constant c such that it lies above cg(n), for sufficiently large n.

For any value of n, the minimum time required by the algorithm is given by Omega Ω(g(n))