Data Structure and Algorithm

Data Structure:

In computer science, a data structure is a data organization, management, and storage format that enable efficient access and modification. More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data. They are containers storing data in a specific memory layout.

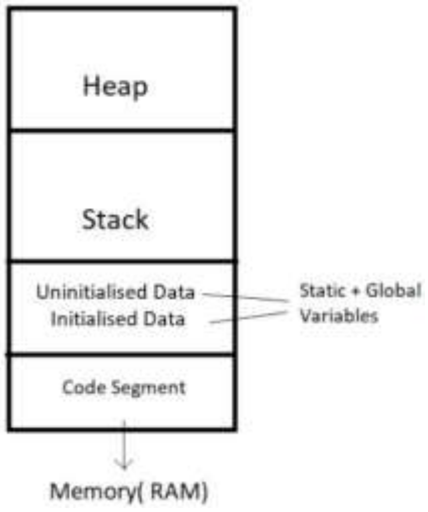
Algorithms:

Sequence of steps performed on the data using efficient data structures to solve a given problem, be it a basic or real-life-based one. Examples include: sorting an array.

Memory Layout of Program:

- When the program starts, its code gets copied to the main memory.
- The 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.
- The heap contains the data which is requested by the program as dynamic memory using pointers.
- Initialized and uninitialized data segments hold initialized and uninitialized global variables, respectively.

Take a look at the below diagram for a better understanding:



Properties of Algorithms:

- It should terminate after finite time.
- It should produce at least one output.
- It is independent of any sort of programming language.
- It should be unambiguous (Deterministic)

Deterministic - For the same input same output will come always.
Not Deterministic - For the same input different output will come. And this is not at all preferable whenever we write any sort of algorithms.

Que : whether the below program is an algorithm or not ?

```
while(true):
    print('Hello World')
```

Hello World Hello World Hello World ....infinite times. : not a algorithm

Stable Algorithm:
Algorithm which maintains the relative order of records with equal keys (i.e. values).
Stable Sorting Algorithm :

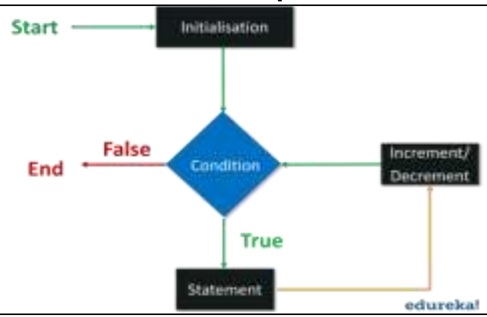
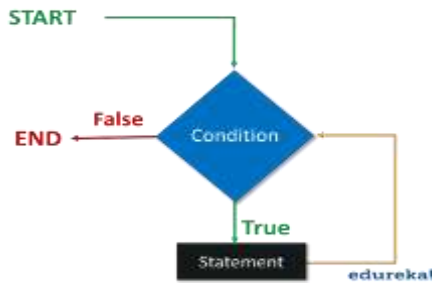
Unsorted	20	3	34	1	20'	9
	1	2	3	4	5	6
Sorted	1	3	9	20	20	34
	1	2	3	4	5	6

From above output, relative order is not maintained which represents that it is not a stable sorting algorithm.

Loops in Python

Loops : Loops are used to execute a set of instructions/functions repeatedly when some conditions become true. There are mainly three types of loops in Java.

1. **for loop** -> used to iterate a part of the program several times. If the numbers of iterations are fixed, it is recommended to go for "for loop".
2. **while loop** -> used to iterate a part of the program several times. If the numbers of iterations are not fixed, it is recommended to go for "while loop".

Loops	**for loop**	**while loop**
Flowchart	 Start Initialisation False End Condition Increment/ Decrement True Statement edureka!	 START False END Condition True Statement edureka!

Example	for x in range(3):
	    print(x)
	Output: it will print
	0 1 2
		i = 1
		while i < 6 :
		    print(i)
		    i += 1
		Output: it will print 1 2 3 4 5

Steps required for constructing an algorithm:

- **Understand Problem Definition** -> Expectations
- **Design algorithm** -> Out of existing algorithms which algorithm is more suitable to this problem statement. Existing algorithms - Divide and Conquer, Greedy Technique, Dynamic Programming, Backtracking and so on.
- **Draw flow chart** -> Visualize sudo code
- **Testing** -> For every input correct output is coming or not
- **Implementation** -> Coding Part
- **Analysis** -> Time & Space Complexity

When we start coding after problem definition it is difficult to understand where done mistake, whether in logic, syntax, problem understanding, so always follow above in constructing algorithm.

Note : Design algorithm and analysis are the two major steps.

Analysis: If any problem contains more than one solution, then best one will be decided by the analysis based on mainly two factors :

1. **Time Complexity** - CPU time
2. **Space Complexity** - Main Memory Space

Note : Time Complexity is more powerful than Space Complexity because processor cost is more costly.
Time Complexity is the study of the efficiency of algorithms. It tells us how much time is taken by an algorithm to process a given input.

Time Complexity: T(P) = C(P) + R(P)

C(P) -> Compile-time is the time at which the **source code is converted into executable code**.
R(P) -> Run time is the time at which the **executable code is started running**.

Types of analysis :

1. **Apostiary Analysis (Relative Analysis) :**
   - Dependent on language of compiler and the type of hardware
   - Exact answer
   - Different answer
   - Program run fast because of the type of hardware used
2. **Apriori Analysis (Absolute Analysis) :**
   - Independent on language of compiler and the type of hardware

- Approximate answers
- same answer
- Program run fast because of **Optimized algorithm**

Apriori Analysis: It is analysis performed prior to running it on a specific system & is a **determination of order of magnitude of a statement.**

O(magnitude) -> Big O notation used to classify algorithms according to how their run time or space requirements grow as the input size grows.

Techniques to calculate Time Complexity:
Once we are able to write the runtime in terms of the size of the input (n), we can find the time complexity. For example:
$T(n) = n^2 \rightarrow O(n^2)$
$T(n) = \log n \rightarrow O(\log n)$
Here are some tricks to calculate complexities:
Drop the constants:
Anything you might think is O(kn) (where k is a constant) is O(n) as well. This is considered a better representation of the time complexity since the k term would not affect the complexity much for a higher value of n.
Drop the non-dominant terms:
Anything you represent as $O(n^2+n)$ can be written as $O(n^2)$. Similar to when non-dominant terms are ignored for a higher value of n.

Examples for better understanding of the concepts:
Problem 1 :
```
def sum():
    x = y + z;      #O(1) - constant time - best case time complexity
```

Problem 2 :
```
def sum():
   x = y + z #O(1) constant time
   for i in range(1, n+1):
      x = y + z  #O(n) time
```

Execution of code :
Let, n = 5 = 5 times
i = 1 ; 1 <= 5 = true

```
x = y + z           1st time
i++ = i + 1 = 1 + 1 = 2 ; 2 <= 5 = true
x = y + z           2nd time
i++ = i + 1 = 2 + 1 = 3; 3 <= 5 = true
x = y + z           3rd time
i++ = i + 1 = 3 + 1 = 4; 4 <= 5 = true
x = y + z           4th time
i++ = i + 1 = 4 + 1 = 5; 5 <= 5 = true
x = y + z           5th time
i++ = i + 1 = 5 + 1 = 6; 6 <= 5 = false
```

no statement is executed and for loop is terminated here

So, for n=5, statement executed 5 times which is represented by O(5) or O(n)

Overall time complexity = O(1) + O(n) = O(n+1) = O(n)
n = 1000000000 + 1 ~ 1000000000
In Big O, Overall time complexity is calculated for worst case scenarios.
In time complexity, basically we are analyzing, in particular program or loop **number of times statement is executing**.

Problem 3 :

```
def sum():
   x = y + z                   #O(1) constant time
   for i in range(1, n+1):
      x = y + z                #O(n) time
   for i in range(1, n+1):
      for j in range(1, n+1):
         x = y + z             #O(n^2) time
```

Execution of code :

```
for i in range(1, n+1):
      for j in range(1, n+1):
         x = y + z             #O(n^2) time
```

For n=3

i = 1; 1 <= 3 = true	i = 2 ; 2 <= 3 = true	i = 3; 3 <= 3 = true

j = 1; 1 <= 3 = true
x = y + z 1st time
j++ = j+1 = 1+1 = 2 <= 3 = true
x = y + z 2nd time
j++ = j+1 = 2+1 = 3 <= 3 = true
x = y + z 3rd time
	j = 1; 1 <= 3 = true
	x = y + z 4th time
	j++ = j+1 = 1+1 = 2 <= 3 = true
	x = y + z 5th time
	j++ = j+1 = 2+1 = 3 <= 3 = true
	x = y + z 6th time
		j = 1; 1 <= 3 = true
		x = y + z 7th time
		j++ = j+1 = 1+1 = 2 <= 3 = true
		x = y + z 8th time
		j++ = j+1 = 2+1 = 3 <= 3 = true
		x = y + z 9th time

j++ = j+1 = 3+1 = 4 <= 3 = false // no statement is executed	j++ = j+1 = 3+1 = 4 <= 3 = false // no statement is executed	j++ = j+1 = 3+1 = 4 <= 3 = false // no statement is executed
i++ = i + 1 = 3 + 1 = 4 ; 4 <=3 = false //no statement is executed now		

n = 3 => 3*3 = 9 times (x = y+z)

n = 10 => 10*10 = 100times (x = y+z)

n = 100000 => 100000*100000 times (x = y + z)

So, for n=3, statement executed 9 times which is represented by O(3^2) or O(n^2)

Overall time complexity of the above code: O(1) +O(n) + O(n^2) = O(n^2) time complexity.

Problem 4 :

```
i = n
while i > 1:
   i = i-1          #O(n-1) = O(n)
```

Execution of code:

Lets, n = 5

i = 5

5 > 1 = true

i = i - 1 = 5 - 1 = 4 1st time

----

i = 4

4 > 1 = true

i = i - 1 = 4 - 1 = 3 2nd time

----

i = 3

3 > 1 = true

i = i -1 = 3 - 1 = 2 3rd time

----
i = 2
2 > 1 = true
i = i - 1 = 2 - 1 = 1   4th time
----
i = 1
1 > 1 = false   no execution inside while loop
//now it will not enter into the while loop statements

So, for n=5, statement executed 4 times which is represented by O(n-1)

n = 5 (Assume)	4 times
n = 10	9 times
n = 100	99 times

Overall time complexity of the above code: O(n-1) = O(n) time complexity.

Problem 5 :

```
i = n
while i >= 1:
   i = i-2          #O(n-1) = O(n)
```

Execution of code:

Let, n = 10
i = 10
10 >= 1true
i = 10 - 2 = 8   1st time
8 >= 1 true
i = 8-2 = 6   2nd time
6 >= 1 true
i = 6-2 = 4   3rd time
4 >= 1 true
i = 4-2 = 2   4th time
2 >= 1 true
i = 2-2 = 0   5th time
0 >= 1 false
// no statement will be executed inside the while loop

n = 10 assumption   5 times

n = 100	50 times
n = 1000	500 times
n	n/2 times

Overall time complexity of above program : O(n/2) = O(n)

Problem 6 :

```
i = n
while i >= 1:
   i = i-30
   i = i-5  # equivalent to i = i-35
```

In above program i decremented by 2 then statement executed n/2 times, so we can say if i decremented by 35 then statement will executed by n/35 times.

n   n/35 times   O(n/35) = O(n)

Problem 7 :

```
n=64
i = 1
while i < n:
   i = 2*i
   print(i)
```

Execution of code:

Lets, n = 64
i = 1 given
1 < 64 = true
i = 2 * i = 2 * 1 = 2   1st time
2 < 64 = true
i = 2 * i = 2 * 2 = 4   2nd time
4 < 64 = true
i = 2 * i = 2 * 4 = 8   3rd time
8 < 64 = true
i = 2 * i = 2 * 8 = 16   4th time
16 < 64 = true
i = 2 * i = 2 * 16 = 32   5th time
32 < 64 = true
i = 2 * i = 2 * 32 = 64   6th time

64 < 64 false
// no statement will be executed after this Conclusion :

n = 64	6 time
n = 32	log_2 (32) = log_2 (2^5)= 5*log_2 (2)=5 time
n	log_2 n or $\log_2$ n

Overall time complexity of above program : O($\log_2$ n)

Conclusion:

- Time complexity is loop only
- Not only loop but larger loop n + n^2 + n^3 = O(n^3)
- And if in a program there is no loop at all - O(1) - best case scenario

Asymptotic Notation :

Asymptotic notation gives us an idea about how good a given algorithm is compared to some other algorithm.

There are 3 types of Notations in algorithm

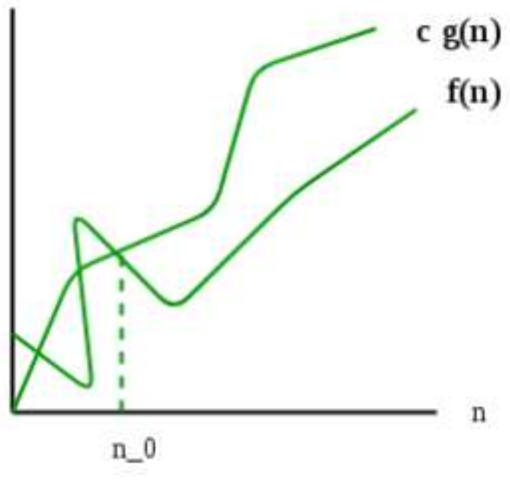
1. Big O Notation
2. Omega Notation
3. Theta Notation

1. **Big O Notation(very very important used everywhere) :** upper bound of an algorithm ie consider worst case

Definition: f(n) = O(g(n)), if f(n) <= cg(n) for all n , n >= n0 such that there exists two positive constants where c > 0 and n0 >= 1. So, g(n) function is c times greater than f(n).

So, if we say a = O(b) meaning is (b greater than a after taking c help)

In searching element in array, Bog O is upper bound or maximum position or anything which is close to last element in an array.



f(n) = O(g(n))

Problem 1 :

If f(n) = 5n, g(n) = n, What should be the value of a constant "c"?

Ans : f(n) = O(g(n)), it means f(n) <= c. (g(n))

5n <= c.n for all n, n >= 1

For, c = 5

5n <= 5n means f(n) = O(g(n))

Problem 2 :

If f(n) = n^2, g(n) = n, What should be the value of a constant "c"?

f(n) = O(g(n)), it means f(n) <= cg(n) (Mathematical definition)

n^2 <= c.n for all n, n >= 1
If we take, c = n -> n^2 <= n^2   // In this is c a constant? -> not at all because here c depends on the value of n. Bigger the value of n, bigger the value of c & smaller the value of n, smaller the value of c. c should be any constant number.
f(n) is not equal to O(g(n))

Increasing order of complexities:

1. **Constant Complexity :** O(1)
2. **Logarithmic Complexity :** O(logn)
3. **Linear Complexity :** O(n)
4. **Quadratic Complexity :** O(n^2)
5. **Cubic Complexity :** O(n^3)
6. **Polynomial Complexity :** O(n^c) ; where c is constant
7. **Exponential Complexity :** O(c^n) ; where c is constant

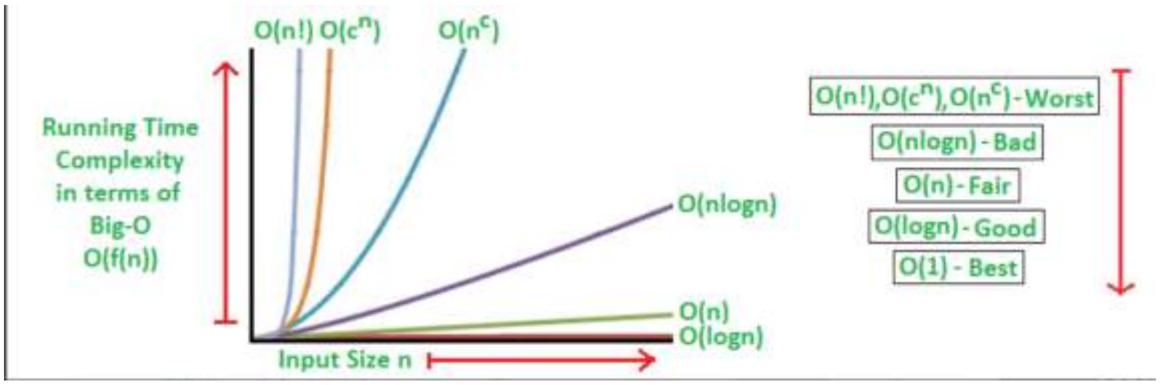
While constructing algorithm, complexity should be as minimum as possible.
n! < n^n (eg 3! < 3^3)
2^n < n^n (eg 2^3 < 3^3)
n! < 2^n (eg 2^3 < 3!)
n! < 2^n < n^n OR
n! = O(2^n) ; n! = O(n^n)
Increasing order of common runtimes:
Below mentioned are some common runtimes which you will come across in your coding career.

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

Better   Worse

Common runtimes from better to worse $\longrightarrow$

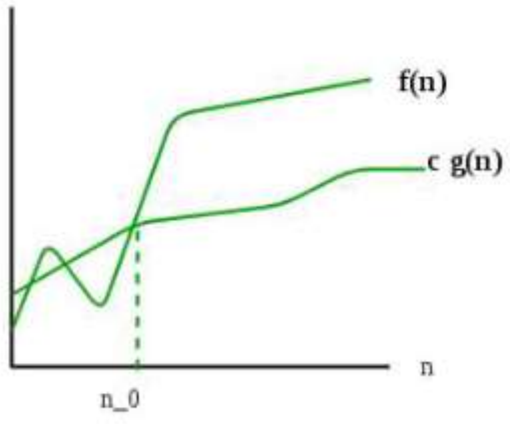
Complexity Graph:



2. Omega Notation:

Let f(n) = omega(g(n)) or f(n) = Ω(g(n)) if and only if f(n) >= c(g(n)) for all n, n>=n0 such that there exists two positive constants c > 0 and n0 >= 1.

In searching element in array, Omega is lower bound or min position or anything which is close to starting element of an array.



f(n) = Omega(g(n))

Problem 1 :

f(n) = n, g(n) = 5n

f(n) = omega(g(n))

What should be the value of c ?

Ans: f(n) >= c(g(n))

n >= c. 5n

c = 1/5 – constant

f(n) = omega(g(n))

Problem 2 :

f(n) = 5n , g(n) = n

f(n) = omega(g(n))

What should be the value of c ?

Ans: f(n) >= c. omega(g(n))

5n >= c.n

c = 1 => 5n >= 1.n

f(n) = omega(g(n))

Problem 3 :

f(n) = n^2 , g(n) = n^2 + n + 10

f(n) = omega(g(n))

What should be the value of c ?

Ans: f(n) >= c.g(n)

n^2 >= c.(n^2+n+10)

c = 1/2

n^2 >= 1/2(n^2 + n + 10)

f(n) = omega(g(n))

Problem 4 :

f(n) = n , g(n) = n^2

f(n) = omega(g(n))

What should be the value of c ?

Ans: f(n) >=c. g(n)

n >= c.n^2

c = 1/n -> means it is not a constant

That's why f(n) is not equal to omega(g(n)).

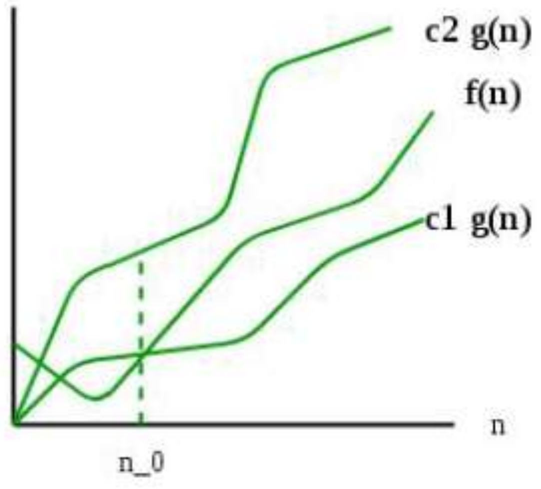
----

3. Theta Notation : Notation satisfies both bigO and omega notation

Let f(n) = theta(g(n)) if and only if f(n) >= c1g(n)(omega) and f(n) <= c2(g(n))(Big O) for all n;

n >= n0 such that there exists three constants.

c1 > 0 ; c2 > 0 ; n0 >= 1



f(n) = theta(g(n))

Problem 1 :

f(n) = n, g(n) = 5n

f(n) = theta(g(n))

1. Omega -> f(n) >= c.g(n)

n >= c. 5n

c = 1/5 – constant

f(n) = omega(g(n))

2. Big O -> f(n) <= c.g(n)

n <= c.5n

c = 1 – constant

f(n) = O(g(n))

Thus because f(n) holds true for both omega as well as O, thus f(n) = theta(g(n))

Problem 2 :

f(n) = n - 10 , g(n) = n + 10

f(n) = theta(g(n))

1. Omega -> f(n) >= c.g(n)

n-10 >= c.(n+10)

c = 1/2 – constant

f(n) = omega(g(n))

2. Big O -> f(n) <= c.g(n)

n-10 <= c.(n+10)

c = 1

f(n) = O(g(n))

We can say that f(n) = theta(g(n)) as it holds true for both omega as well as big O.

Problem 3 :

f(n) = n , g(n) = n

f(n) = theta(g(n))

1. Omega -> f(n) >= c.g(n)

n >= c.n

c = 1 - constant

Thus, f(n) = omega(g(n))

2. Big O -> f(n) <= c.g(n)

n <= c.n

c = 1 - constant Thus, f(n) = O(g(n))

And thus f(n) = theta(g(n)) as it holds true for both omega and Big O

Problem 4 :

f(n) = n , g(n) = n^2

f(n) = theta(g(n))

1. Omega -> f(n) >= c.g(n)

n >= c.n^2

c = 1/n - Not a constant

f(n) is not theta(g(n))

Recurrence Relation Solving Method

Recurrence Relation: The procedure for finding the terms of a sequence in a recursive manner is called recurrence relation.

Recursive function: Function call itself directly or indirectly with different parameters.

for example :

1. Binary Search Algorithm :

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

where T(n) is the time required for binary search in an array of size n

2. Merge Sort Algorithm :

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

3. Strassens Matrix Multiplication :

T(n) = 7T(n/2) + n^2

Now the problem is to evaluate the time complexity with the help of a given Recurrence Relation.

This can be done by 3 methods :

1. Master's Theorem
2. Substitution Method
3. Recursive Tree Method

1. Master's Theorem: Used widely, it basically take care of who is greater

T(n) = aT(n/b) + f(n) ; a and b are positive constants & a, b > 1

Simplest way to evaluate this is to compare two values:

- n^(log_b a)
- f(n)

Now if one of them is larger, then that's the solution of recurrence relation.

If both are equal the solution is T(n) = O(f(n)logn)

Problem 1 :

T(n) = 8T(n/2) + n^2

Ans:

1. n^(log_2 8) => n^3
2. f(n) = n^2

T(n) = O(n^3)

Problem 2 :

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

Ans:

1. n ^ (log_2 2) = n
2. f(n) = n^2

T(n) = O(n^2)

Problem 3 :

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

Ans:

1. n^(log_2 2) = n
2. n

1 and 2 are equal, no one is greater

T(n) = O(f(n)logn) = O(nlogn)

Problem 4 :

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

Ans:

1. n^(log_2 1) = n^0 = 1
2. c

1 and 2 both are equal, no one is greater

T(n) = O(f(n)logn) = O(c.logn) = O(logn)

2. Substitution Method:

Substitute the given function repeatedly until the given function is removed.

Problem 1 :

T(n) = 1 if n = 1

T(n-1) + n if n>1

Ans:

T(n) = T(n-1) + n   1st time

= T(n-2) + n-1 + n   2nd time

= T(n-3) + n-2 + n-1 + n   3rd time

k times = n-1

n-k = 1

n-1 = k

= T(n-k) + (n-k+1) + (n-k+2) +.........+n-2+n-1+n   k times = n-1

= T(n-(n-1)) + (n-(n-1)+1) + (n-(n-1)+2) +.........+n-2+n-1+n

= T(n-n+1) + (n-n+1+1) + (n-n+1+2) + ... + n-2+n-1+n

= T(1) + 2 + 3 + 4 + 5 + ... + n-2+n-1+n

= 1 + 2 + 3 + 4 + 5 + ... + n-2 + n-1 + n => Sum of n natural numbers

= n(n+1)/2

= (n^2 + n)/2

T(n)= O(n^2)

Problem 2 :

T(n) = 1 if n = 1

T(n-1).n if n>1

Ans:

T(n) = T(n-1).n 1st time

= T(n-2) (n-1) n 2nd time

= T(n-3) (n-2) (n-1) n 3rd time

= T(n-k) (n-k+1) (n-k+2) .... (n-2)(n-1)n k times = n-1

n-k = 1

n-1 = k

= T(n-(n-1)) (n-(n-1)+1) (n-(n-1)+2) .... (n-2)(n-1)n

= T(n-n+1) (n-n+2) (n-n+3)..........(n-2)(n-1)n

= T(1).2.3.4.........(n-1).n

= 1.2.3..........(n-1).n = n! = O(n!)

5! = 5.4.3.2.1 = 120

n^n > n!

T(n)= O(n^n)

Problem 3 :

T(n) = 1 if n = 0

T(n-2) + n^2 if n>0

Ans:

T(n) = T(n-2) + n^2 1st time

=T(n-4) + (n-2)^2 + n^2 2nd time

=T(n-6) + (n-4)^2 + (n-2)^2 + n^2 3rd time

= T(n-2k) + (n-2k+2)^2 + (n-2k+4)^2 + ... + (n-2)^2 + n^2 k times

n - 2k = 0

n = 2k

n/2 = k

= T(n-2(n/2)) + (n-2(n/2)+2)^2 + ... + (n-2)^2 + n^2

= T(0) + (2)^2 + (4)^2 + (6)^2 + ... + (n-2)^2 + n^2

= 1 + (2)^2 + (4)^2 + (6)^2 + ... + (n-2)^2 + n^2

= 1 + 2^2 ((1)^2 + (2)^2 + (3)^2 + ... + (n/2)^2)

Sum of squares of n natural numbers = (n (n+1) (2n+1))/6

Sum of squares of n/2 natural numbers = (n/2 (n/2+1) (2(n/2)+1))/6

= 1 + 2^2((n/2 (n/2+1) (2(n/2)+1))/6)

= O(n^3)

Recursive Tree Method: If two or more recursive terms occurs, then we use Recursive Tree Method.

Problem 1 :

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

Ans

T(n) = O(n.log_2 n)

Problem 2 :

T(n) = T(n/5) + T(4n/5) + n

Ans

T(n) = O(n.log_5/4 n)

Problem 3 :

T(n) = T(n/5) + T(3n/5) + n

Ans

T(n) = O(n)

Designing Of Algorithms

1. Divide And Conquer

Introduction:

Strategy of Divide and Conquer:

- Divide the big problem into some sub-problems.
- Solve the sub-problems (conquer) using **Recursion** until that particular sub-problem is solved.
- Combine the sub-problem solution so that we will be able to get final solution of the problem.

Main Point: If the problem is big divide that problem otherwise not.

Abstract Algorithm of Divide and Conquer:

i -> starting element of an array
j -> ending element of an array

```
DAC(a,i,j){
if(small(a,i,j)){
return (solution(a,i,j))          O(1)
}
else{
m = Divide(a,i,j)                 f1(n)
b = DAC(a,i,m)                    T(n/2)
c = DAC(a,m+1,j)                  T(n/2)
return (combine(b,c))             f2(n)
}
}
```

DAC - Finding of Time Complexity :

```
T(n) = O(1) ;                          if n is small
f1(n) + T(n/2) + T(n/2) + f2(n) ;      if n is large
```

So, **overall time complexity** is

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

f(n) => Divide + Combine

This is known as the **Recurrence Relation**.

So, for different problems we have different Recurrence Relation.

for example :

In QuickSort, Recurrence Relation is

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

In Strassen's Matrix Multiplication,

T(n) = 8T(n/2) + n^2

Here, 8 is the number of subproblems
T(n/2) represents size of subproblem
n^2 is the Divide + Combine function

Applications of Divide and Conquer:

There are so many applications of Divide and Conquer for example:

- Finding of Power of an Element
- Binary Search
- Merge Sort
- Quick Sort
- Selection Procedure
- Finding of inversions
- Finding of Maxima and Minima in the given array of elements
- Strassen's Matrix Multiplication and so on

Now we will explore the applications of Divide and Conquer to decide approach to a given problem and logic building behind the Divide and Conquer algorithms so that we will be able to solve any new problem using Divide and Conquer Methodology.

Linear Search

Note : Because here we are discussing about Searching. So, intention is to complete Searching concept completely.

Linear Search: (Not an application of DAC)

input: An array of n elements and what element "x" we want to search in an array

output: Position of an element x if it is found and if it is not present in the array then our function will return -1

n -> number of elements in an array

```
LinearSearch(int arr[], int x){
for(i=0;i<n;i++){                    O(n)
if(arr[i] == x){
System.out.println(i);
}
}
return -1;
}
```

Discussion about the Best case(lower bound or at least position or anything which is close to starting element of an array), Worst case (upper bound or at most position or close to last element in an array) and average case time complexity.

Best Case : O(1)

Worst Case : O(n)

Average Case : O(n)

Binary Search :

input : An array of n elements and what element "x" we want to search in an array

output : Position of an element x if it is found and if it is not present in the array then our function will return -1

Implementation of Binary Search:

Problem 1:

Eg. 6, 4, 10, 8, 2,

What is position of element 8?

[Note: Binary search can only applied to Sorted array]

Index	0	1	2	3	4
Array	2	4	6	8	10

Base address, i = 0

Maximum position, j = 4

x => That we want to search in an array = 8

mid = (0 + 4)/2 = 2

a[mid] = a[2] = 6 == 8 -> false

a[mid] = a[2] = 6 < 8 -> true

BinarySearch(a,mid+1,j,x);

i = 3

j = 4

mid = (3+4)/2 = 7/2 = 4

a[mid] = a[4] = 10 == 8 -> false

a[mid] = a[4] = 10 > 8

BinarySearch(a,i,mid-1,x);

i = 3

j = 3

i == j (3 == 3)

a[i] = a[3] = 8 == 8 -> true

Return -> 3

Python Implementation:

```
def binary_search(arr, i, j, x):
    if j >= i:
        mid = (i + j) // 2  #O(1)
        if arr[mid] == x:
            return mid    # O(1)
        elif arr[mid] > x:
            return binary_search(arr, i, mid - 1, x)   # T(n/2)
        else:
            return binary_search(arr, mid + 1, j, x)  # T(n/2)
    else:
        return -1
```

Recurrence Relation :
T(n) = T(n/2) + c -> Binary Search Algorithm
a = 1
b = 2
n^(log_b a) = n^(log_2 1) = n^0 = 1
f(n) = c
Which one is greater?
Both are equal i.e. constant
Overall time complexity is : O(f(n) logn) => O(c logn) => O(logn)
Discussion about Best case, worst case and average case time complexity
Best case : O(1)
Worst case : O(logn)
Average case : O(logn)

Some Problems asked in interview :

Problem 2:
input : An array of n elements in which until some place all values are integers and after that all values are infinite
output : Find the position of first infinite element.
Solution :
0 1 2 3 4 5 6 7 8
2 4 5 1 -1 * * * *
Position = 5
mid = (0 + 8)/2 = 4
BinarySearch(i,j)
a[mid] == *
return mid
a[mid] != *
BinarySearch(mid+1,j)
Linear Search = O(n)
BinarySearch algorithm = O(logn)

Note :
If in any problem you are able to decide whether to go to left or right just like in binary search implementation, then you can say that we can apply the binary search like implementation in that particular problem. It doesn't matter in that particular case that whether your array is sorted or not.

Problem 3:
input : An array of n elements in which until some place all are integers and afterwards all are infinite. And here you have to assume that n is unknown and after completion of array all are $
output : Find the position of first infinite

Solution :
Linear Search = O(n)
BinarySearch algorithm = O(logn)

Problem 4:
Arrange in increasing order -
1. n^2 + n
2. log^2 n
3. n/logn
4. 5n + 4
5. log logn
6. sqrt(n)
7. 2^n
8. (logn) ^ logn

f5 < f2 < f6 < f3 < f4 < f1 < f8 < f7

Sorting

Sorting - ordering a list of data items in a pre-defined sequence

Sorting algorithms are a set of instructions that take an array or list as an input and arrange the items into a particular order.

Sorts are most commonly in numerical or a form of alphabetical (called lexicographical) order, and can be in ascending (A-Z, 0-9) or descending (Z-A, 9-0) order.

There are many types of sorting algorithms: merge sort, quick sort, selection sort, bubble sort, insertion sort, heap sort, balloon sort, radix sort, etc. Not one can be considered the fastest because each algorithm is designed for a particular data structure and data set. It would depend on the data set that we want to sort.

Merge Sort : Divide and Conquer

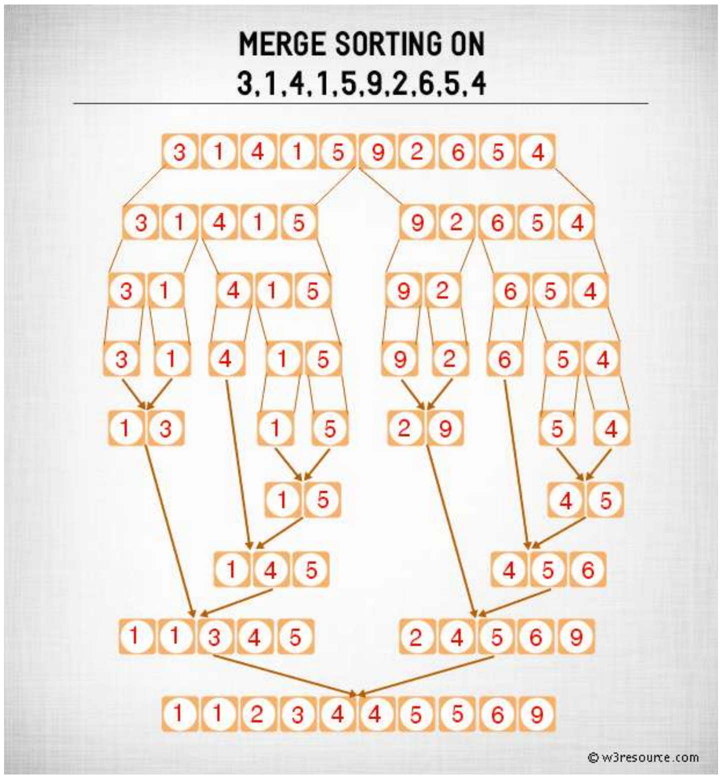
Merge sort is a sorting algorithm based on divide and conquer programming approach. It keeps on dividing the list into smaller sub-list until all sub-list has only 1 element. And then it merges them in a sorted way until all sub-lists are consumed.

input : an array of unsorted elements

output : an array of sorted elements

small problem : If an array contains only single element in an array then it is itself a sorted array and that we consider as the small problem.

Input	3	1	4	5	9	2	6	5	4
Index	1	2	3	4	5	6	7	8	9



In Stack Function call & execution happening as :

Function call -> Preorder ie Sequence is Root >Left > Right

Function execute -> Postorder ie Sequence is Left > Right > Root

Merge Procedure:

Worst case number of comparisons in Merge Procedure:

Unsorted	10	20	30	40	11	21	31	41
	1	2	3	4	5	6	7	8
Sorted	10	11	20	21	30	31	40	41
	1	2	3	4	5	6	7	8

Number of comparisons: 7

(10,11) = 10	1st time
(20,11) = 11	2nd time
(20,21) = 20	3rd time
(30,21) = 21	4th time
(30,31) = 30	5th time
(40,31) = 31	6th time
(40,41) = 40	7th time
41 = 41	No comparison

General formula for worst case scenario of merge procedure:

m + n - 1

m = number of elements in sorted subarray 1

n = number of elements in sorted subarray 2

m = 4 , n = 4

4 + 4 - 1 = 7

Best case number of comparison in Merge Procedure:

Unsorted	10	20	30	40	50	60	5	6
	1	2	3	4	5	6	7	8
Sorted	5	6	10	20	30	40	50	60
	1	2	3	4	5	6	7	8

Number of comparisons:

(10,5) = 5	1st comparison
(10,6) = 6	2nd comparison

General formula for best case scenario of merge procedure:

min(m,n)

m = number of elements in sorted subarray 1

n = number of elements in sorted subarray 2

Overall Time Complexity of Merge Procedure:

Number of moves in best and in worst case scenario = m + n

Time complexity = Number of moves + Number of comparisons

= O(m + n)

Python Implementation:

```
def merge(left,right,compare):
   result = []
   i, j = 0, 0
   while (i < len(left) and j < len(right)):
      if compare(left[i],right[j]):
         result.append(left[i])
         i += 1
      else:
         result.append(right[j])
         j += 1
   while (i < len(left)):
      result.append(left[i])
      i += 1
   while (j < len(right)):
      result.append(right[j])
      j += 1
   return result

def merge_sort(arr, compare = lambda x, y: x < y):
    #Used lambda function to sort array in both(increasing and decresing) order.
    #By default it sorts array in increasing order

    #small problem
   if len(arr) < 2:
      return arr[:]
   # big problem
   else:
      middle = len(arr) // 2                         # Divide - O(1)
      left = merge_sort(arr[:middle], compare)       # Left side tree - T(n/2)
      right = merge_sort(arr[middle:], compare)      # Right side tree - T(n/2)
      return merge(left, right, compare)             # Combine - O(n)
```

Overall Time Complexity :

Best, average and worst case scenario

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

$= O(n \log n)$

Merge sort Properties:

- Space Complexity: O(n)
- Time Complexity: O(nlog(n)).
- Stable: Yes

Quicksort:

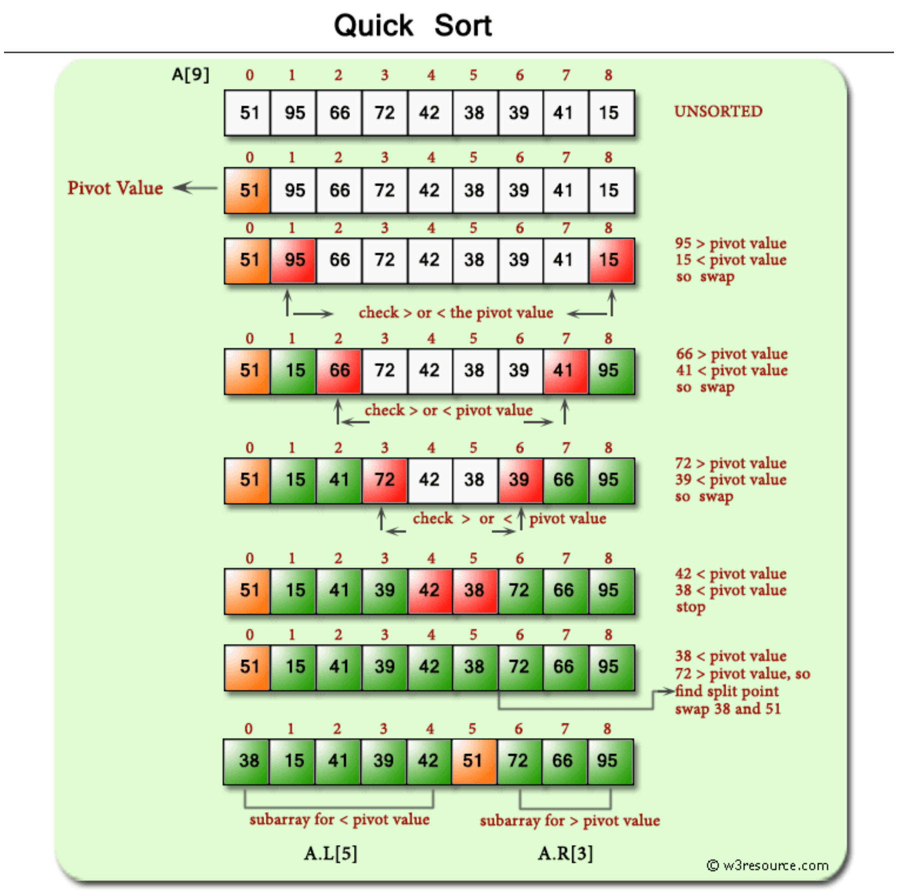
Quick sort uses divide and conquer approach. It divides the list in smaller 'partitions' using 'pivot'. The values which are smaller than the pivot are arranged in the left partition and greater values are arranged in the right partition. Each partition is recursively sorted using quick sort.
The pivot can be the first, the last, the medium or any random element on the array.

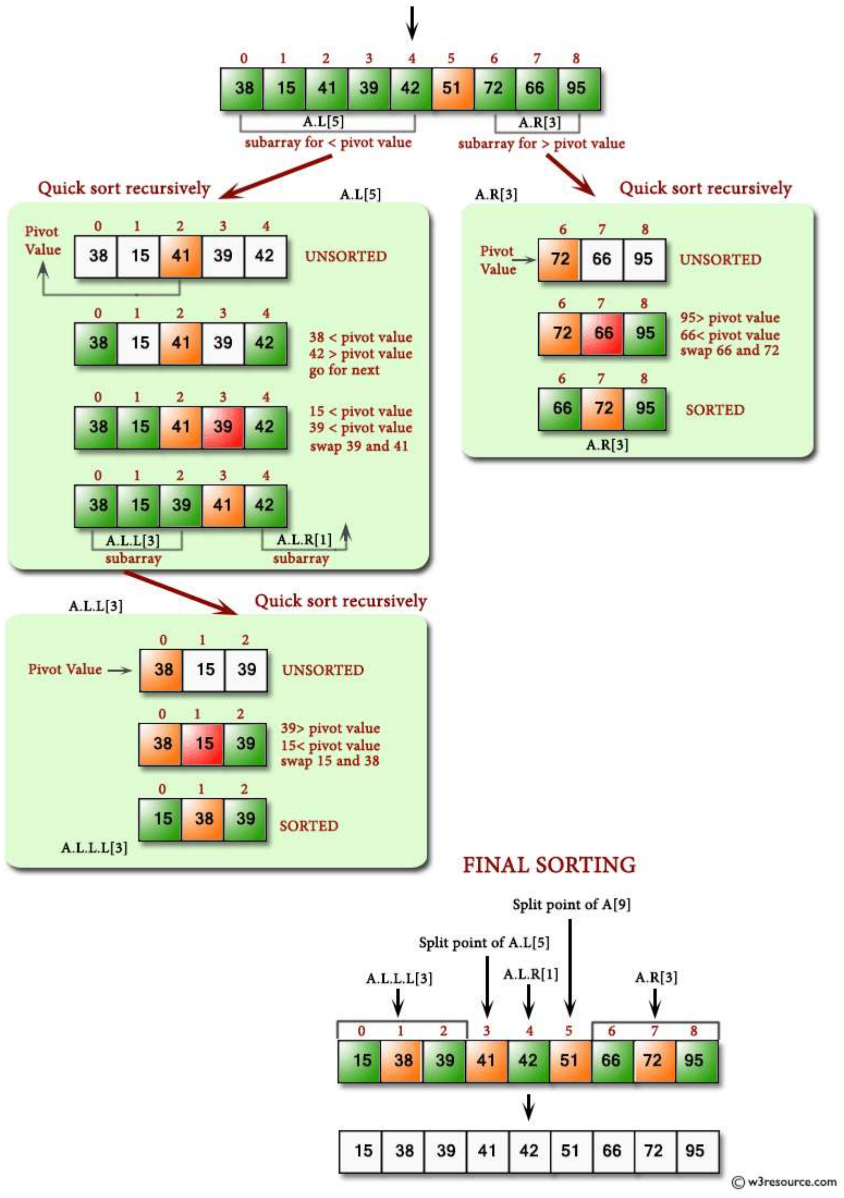
The steps involved in Quick Sort are:
Choose an element to serve as a pivot.
Partitioning: Sort the array in such a manner that all elements less than the pivot are to the left, and all elements greater than the pivot are to the right.
Call Quicksort recursively, taking into account the previous pivot to properly subdivide the left and right arrays.

Properties of Quick sort:

- Not a stable sorting algorithm
- Inplace Sorting algorithm
- Quicksort is widely used in practical/real life applications

QuickSort Implementation:





Observations:

1. **Exact location** of that **pivot element** will be returned after execution of a code.
2. **Left hand side** elements are **lesser than the pivot element** and **right hand side** elements are **greater** than the pivot elements.
3. The position that is returned for the pivot element indirectly indicates that **the pivot element is the nth smallest element in an array**.

Implementation of Partition Algorithm:

```
def partition(array,l,h):
  i = ( l-1 )
  p = array[h]
```

```
    for j in range(l, h):

        if   array[j] < p:

            i = i+1
            array[i],array[j] = array[j],array[i]

    array[i+1],array[h] = array[h],array[i+1]
    return ( i+1 )

def quickSort(a,l,h):
    if l < h:

        pin = partition(a,l,h)
        quickSort(a, l, pin-1)
        quickSort(a, pin+1, h)
```

Randomized QuickSort : When we choose the pivot element randomly from an array then it is known as Randomized Quicksort. Here we swap selected random element with starting element then start first element as pivot element. In this we give independency to pivot element which gives better results.

Recurrence Relation Of QuickSort Algorithm:
Best Case Scenario:
Best case means that the position of pivot element divides an array in such a way that there should be almost equal number of elements present in left as well as right hand side. Then, we consider that type of case as the best case scenario for a quicksort algorithm.
$T(n) = O(n) + T(n/2) + T(n/2)$
$= O(n) + 2T(n/2)$
Using master's theorem we can easily solve the above recurrence relation,
$n^{\log_b a} = n$ $f(n) = n$
$O(f(n)\log n) = O(n \log n)$
So, O(nlogn) is the best case time complexity of Quicksort algorithm.
Worst Case Scenario:
$T(n) = O(n) + T(1) + T(n-1)$
$= T(n-1) + n$
Using the substitution method, and finally showed you that overall time complexity of Quicksort in worst case scenario is:
$T(n) = O(n^2)$
Note : Whenever an array is almost sorted or completely sorted, its never recommendable to use quicksort in these type of scenarios because if we use quicksort in these situations this will give us worst case time complexity of $O(n^2)$.
If an array is almost sorted, it's recommendable to use insertion sort to sort an array completely because this will give us an overall time complexity of O(n) in that particular case.

Some of the questions asked in interviews related to QuickSort :

Problem 1 :

In quicksort sorting of n numbers the n/7th smallest element is selected as the pivot element using O(n^2) time complexity algorithm. Then what will be the worst case time complexity of the quicksort.

Solution :

T(n) = n^2 + n + T(n/7 - 1) + T(n - n/7)

= n ^ 2 + T(n/7) + T(6n/7)

Now, solve this recurrence relation with the help of an Recursive tree method and after solving the above recurrence relation with the help of recursive tree method, we get an overall time complexity of O(n^2)

Problem 2 :

In quicksort sorting of n numbers the 25th largest element is selected as the pivot element using O(n^2) as the time complexity. Then what will be the worst case time complexity of the above algorithm.

Solution :

T(n) = n^2 + n + T(n-25) + T(24)

n ^ 2 - choosing the pivot element

n - Partition of algorithm

T(n-25) - Left Part

T(24) - Right Part

T(n) = n^2 + T(n-25) (To be done by you)

After solving of this particular recurrence relation, we get an overall time complexity of O(n^3)

NOTE :

• If in question it is mentioned about the nth smallest element then take the position of pivot element from left hand side of an array just we consider in Problem number 1.

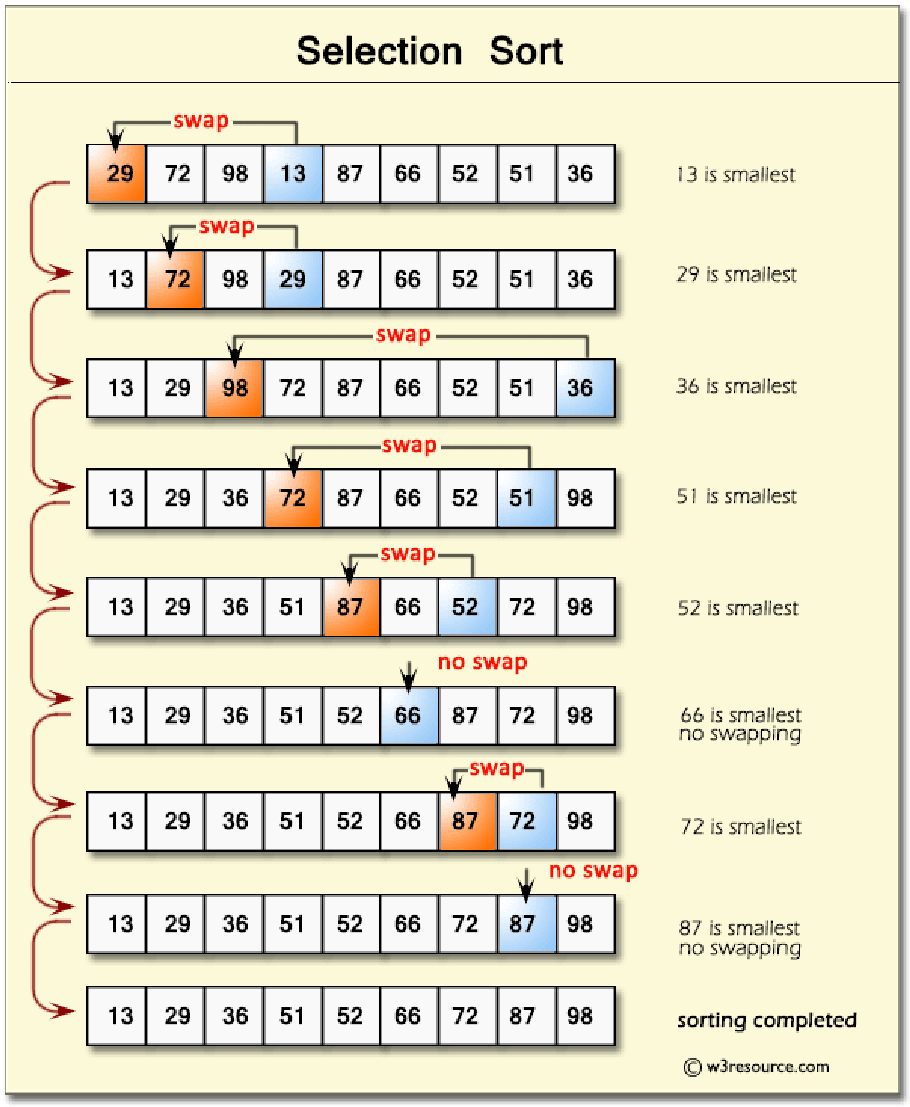
• If in question it is mentioned about the nth largest element then take the position of pivot element from right hand side of an array like we consider in Problem number 2.

Selection Sort:

Selection Sort selects the current smallest element and swaps it into place.
Selection Sort Procedure:

- Find the smallest element in the array and swap it with the first element. >2. Find the second smallest element and swap with the second element in the array.
- Find the third smallest element and swap wit with the third element in the array.
- Repeat the process of finding the next smallest element and swapping it into the correct position until the entire array is sorted.

Implementation:



Python Implementation:

```
def selectionSort(array, size):

    for step in range(size):
        min_idx = step

        for i in range(step + 1, size):
            if array[i] < array[min_idx]:
                min_idx = i

        # put min at the correct position
        (array[step], array[min_idx]) = (array[min_idx], array[step])
```

Selection sort Properties:

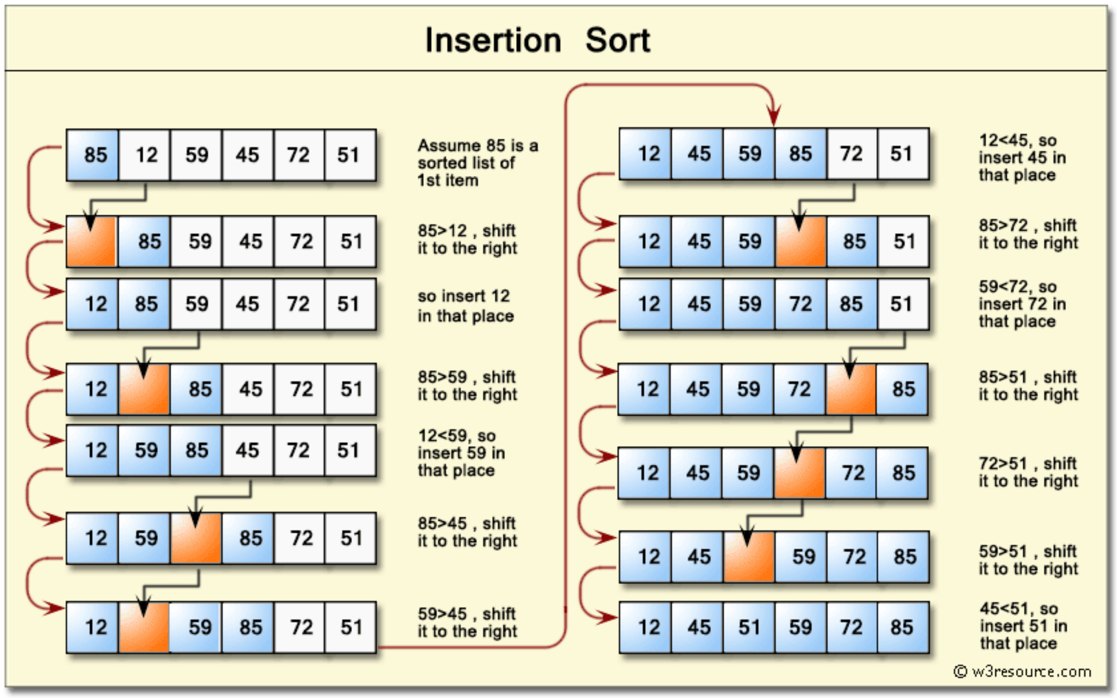
Space Complexity: O(n)
Time Complexity: O(n2)
Sorting in Place: Yes
Stable: No

Few Points to Remember about which sorting algorithm to use :
When we want minimum number of swaps - Selection Sort
When we want to sort an already almost sorted array - Insertion Sort
When you actually want to perform sort in an unsorted array - QuickSort

Insertion Sort

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.
In Insertion sort, we compare the key element with the previous elements. If the previous elements are greater than the key element, then we move the previous element to the next position.
Usually, start from index 1 of the input array.

Implementation:


Insertion Sort
Assume 85 is a sorted list of 1st item
85>12 , shift it to the right
so insert 12 in that place
85>59 , shift it to the right
12<59, so insert 59 in that place
85>45 , shift it to the right
59>45 , shift it to the right
12<45, so insert 45 in that place
85>72 , shift it to the right
59<72, so insert 72 in that place
85>51 , shift it to the right
72>51 , shift it to the right
59>51 , shift it to the right
45<51, so insert 51 in that place
© w3resource.com

Python Implementation:

```
def insertionSort(arr):
    # Traverse through 1 to len(arr)
    for i in range(1, len(arr)):
        key = arr[i]
        j = i-1
        while j >= 0 and key < arr[j] :
            arr[j + 1] = arr[j]
            j -= 1
       arr[j + 1] = key
```

Properties of Insertion Sort:

Space Complexity: O(1)

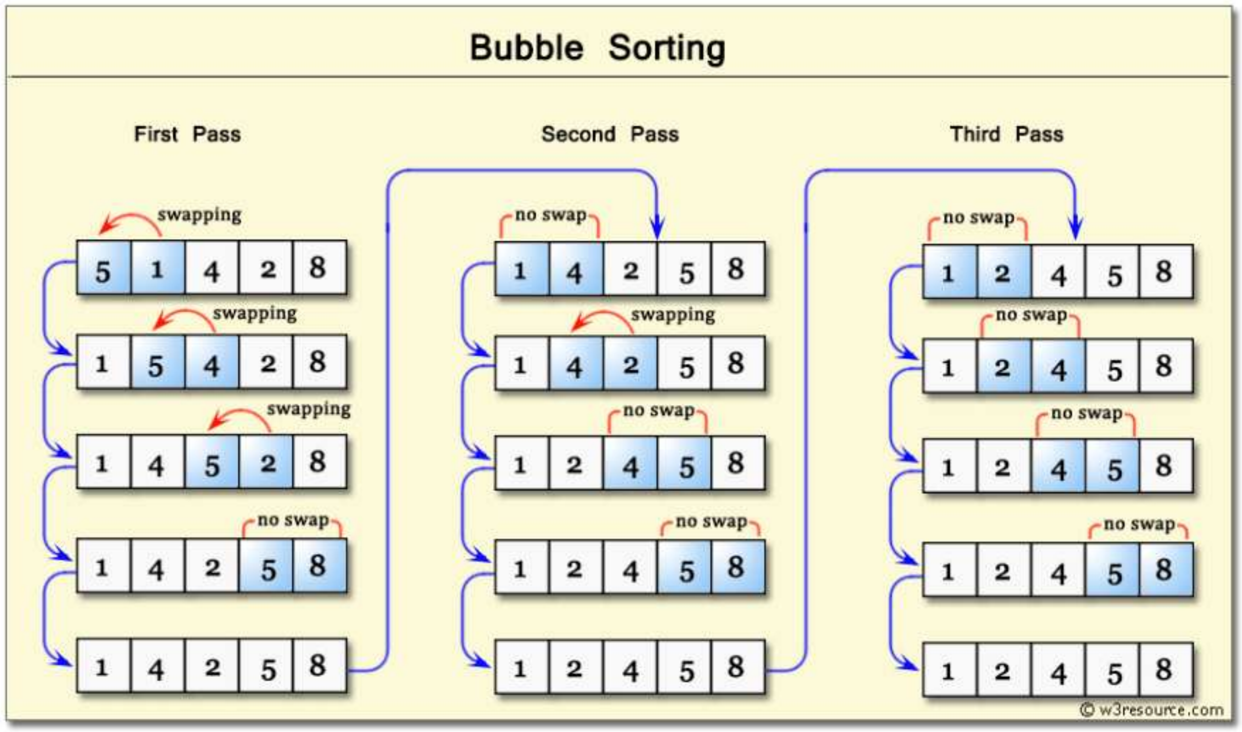
Time Complexity: O(n), O(n* n), O(n* n) for Best, Average, Worst cases respectively.

Sorting In Place: Yes
Stable: Yes

Bubble Sort:

The algorithm traverses a list and compares adjacent values, swapping them if they are not in the correct order.

Implementation:



Python implementation:

```
def bubble_sort(alist):
    n=len(alist)
    for i in range(n):
        for j in range(i):
            if alist[j]>alist[j+1]:
                temp = alist[j]
                alist[j] = alist[j+1]
                alist[j+1] = temp
    return alist
```

Bubble sort Properties:

Space complexity: O(1)

Best case performance: O(n)

Average case performance: O(n*n)

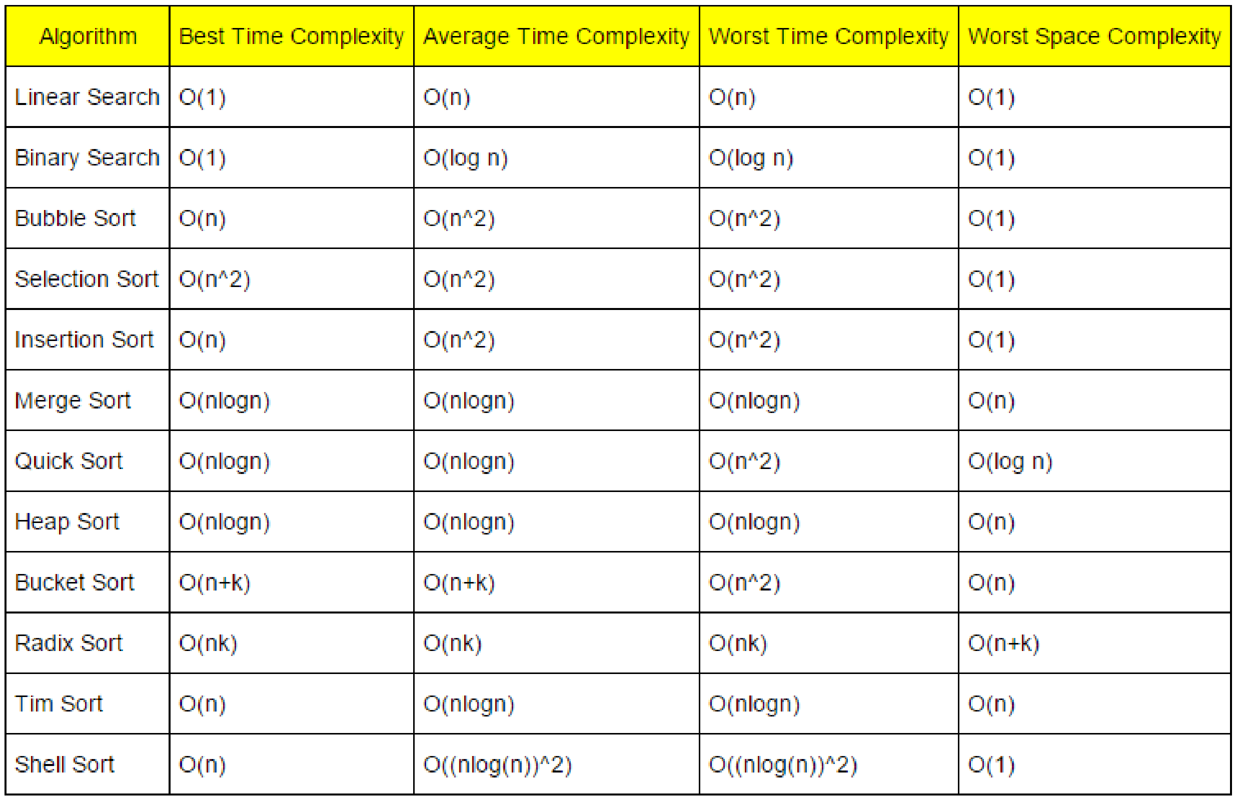
Worst case performance: O(n*n)

Stable: Yes

Pros
It is one of the easiest sorting algorithms to understand and code from scratch.
From technical perspective, bubble sort is reasonable for sorting small-sized arrays or specially when executing sort algorithms on computers with remarkably limited memory resources.

Cons
Because the time complexity is O(n2), it is not suitable for large set of data.
Bubble sort is very slow compared to other sorting algorithms like quicksort.

Algorithm comparisons:



Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Worst Space Complexity
Linear Search	O(1)	O(n)	O(n)	O(1)
Binary Search	O(1)	O(log n)	O(log n)	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)	O(log n)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)
Tim Sort	O(n)	O(nlogn)	O(nlogn)	O(n)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)	O(1)