# Fundamental of Algorithm

1. Trace the brute force maxima and plane sweep algorithm with suitable example.

```
MAXIMA (int n, print (1…….. n)) // maximum of p(0…. n-1)
{
    for I = 1 to n
    {
        maximal = TRUE // p[i] is maximum by default
        for j = 1 to n
        {
            If(I != j) and (p[i].X <= p[j].X) and (p[i].Y <= p[j].Y)
            {
                maximal = FALSE // p[i] is dominated by p[j]
                Break;
        }}
        If (maximal) output p[i] // no one dominate p[i]
}}
```

Trace the brute force maxima:

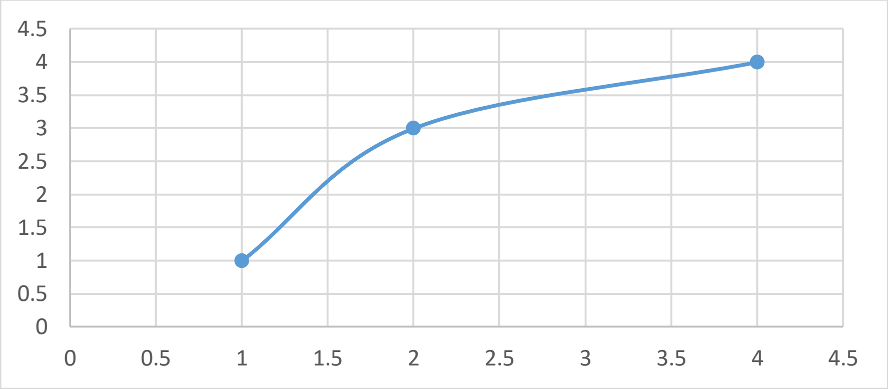
Brute-Force maxima Algorithm given 3 points:

p ((1,1), (2,3), (4,4))

Now, n = 3

for i = 1 to n(3)

If we plot the given number in graph then,



Here, point (6,6) is the maximum set.

Now, finding out by using Brute force maxima.

```
p ((1,1), (2,3), (4,4))

for i = 1 to 3, i = 1

Maximal = (1,1)

for j = 1

if (I != j) i.e (1 != 1) which is False

        Loop not executed.

for j = 2

if (1 != 2)  && (1 <= 2) && (1 <= 3)

        Maximal = False

        Break;


p ((1,1), (2,3), (4,4))

                for i = 2

                Maximal = (2,3)

                for j = 1

                if (2 != 1) && (2 <= 1) Which is False

                        Loop not executed.
```

```
    for j = 2
    if (2 != 2) which is False
        Loop not executed.
    for j = 3
    if (2 != 3) && (2 <=4) && (3 <= 4)
        Maximal = False
        Break;

p ((1,1), (2,3), (4,4))
    for i = 3
    Maximal = (4,4)
    for j = 1
    if (3 != 1) && (4 <= 1) which is False
        Loop not executed.
    for j = 2
    if (3 != 2) && (4<= 2) which is False
        Loop not executed.
    for j = 3
    if ( 3 != 3) Which is False
        Loop not executed.
```

Now, loop end for both i and j.

Now, print latest maximal i.e. (4, 4)

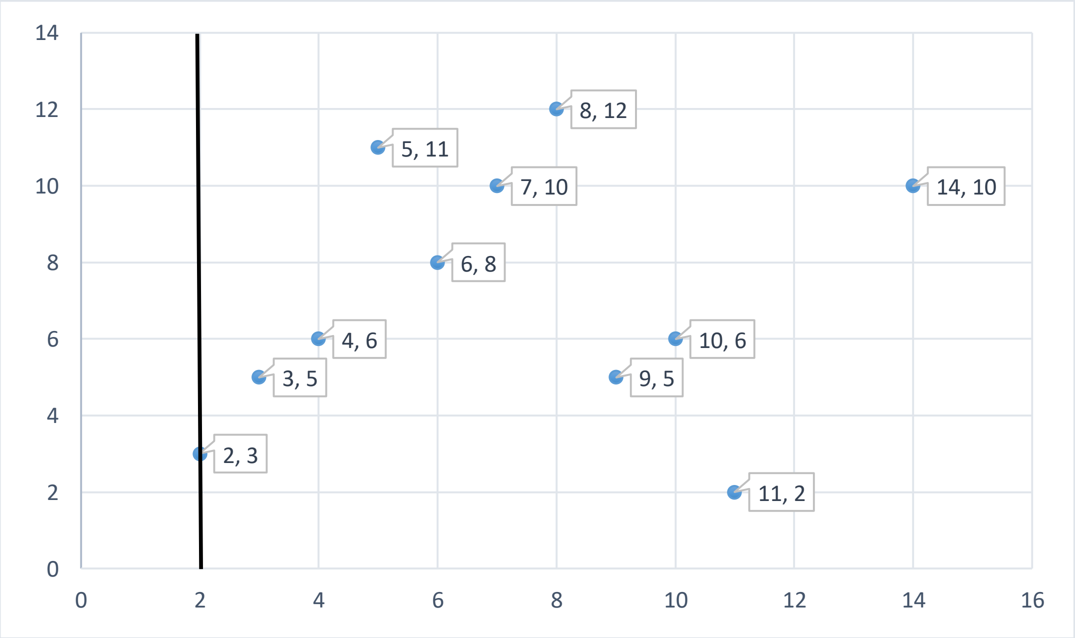
Output = (4, 4)

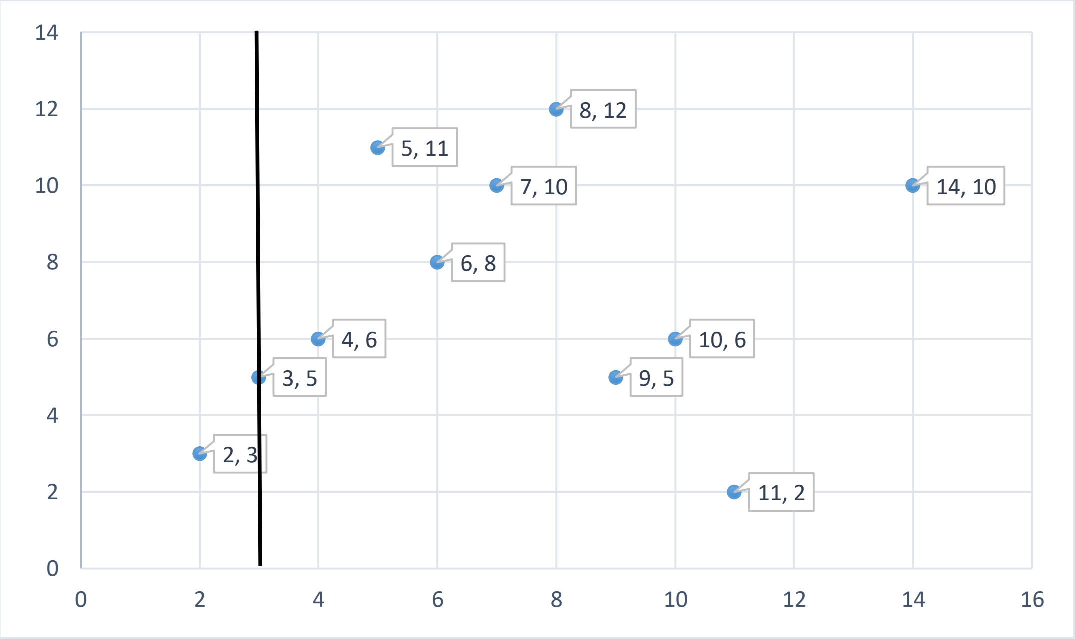
We know the highest/maximum number is (4, 4) from the given set p ((1, 1), (2, 3), (4, 4))

Algorithm

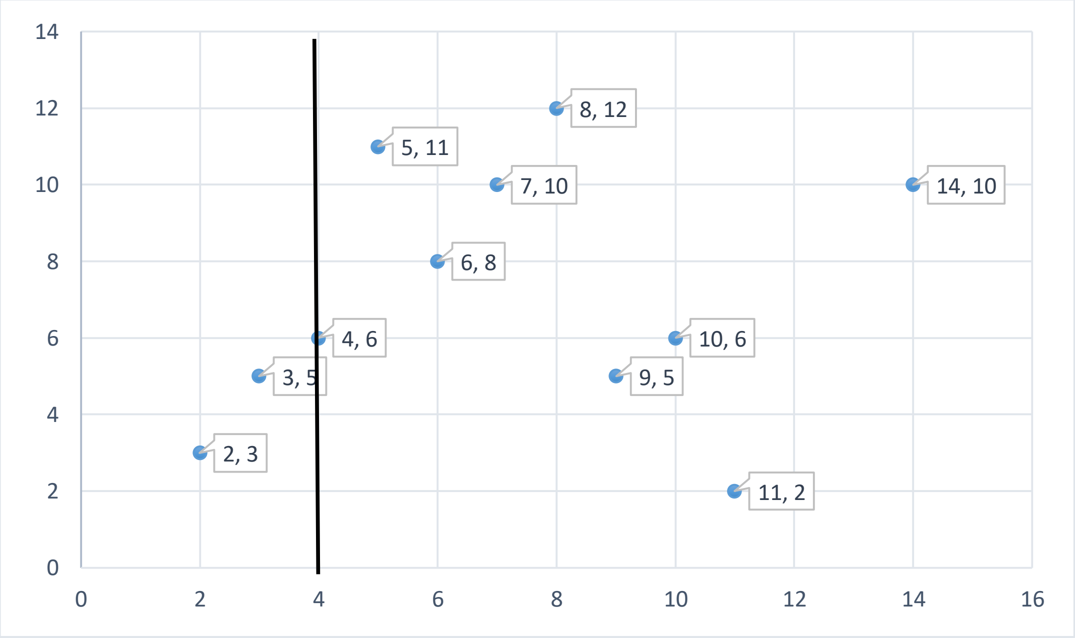
```
Plane-sweep–maxima (n,p[1,…….., n])
Sort the p in increasing order of x.
Initialize stack s.
For i=1 to n
DO While (S.notempty() && s.top.Y< p[i].Y)
DO S.POP()
Do S.PUSH(p[i]
Output The content of STACK
```

(2,3), (3,5), (4,6), (5,11), (6,8), (7,10), (8,12), (9,5), (10, 6), (11,2), (14, 10), (15, 7)

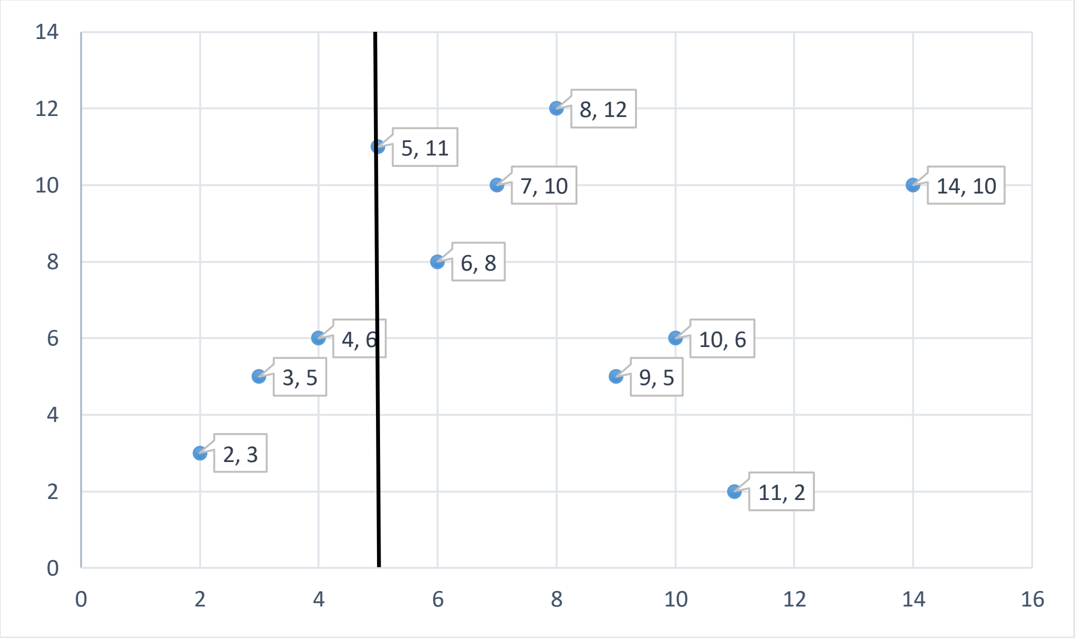


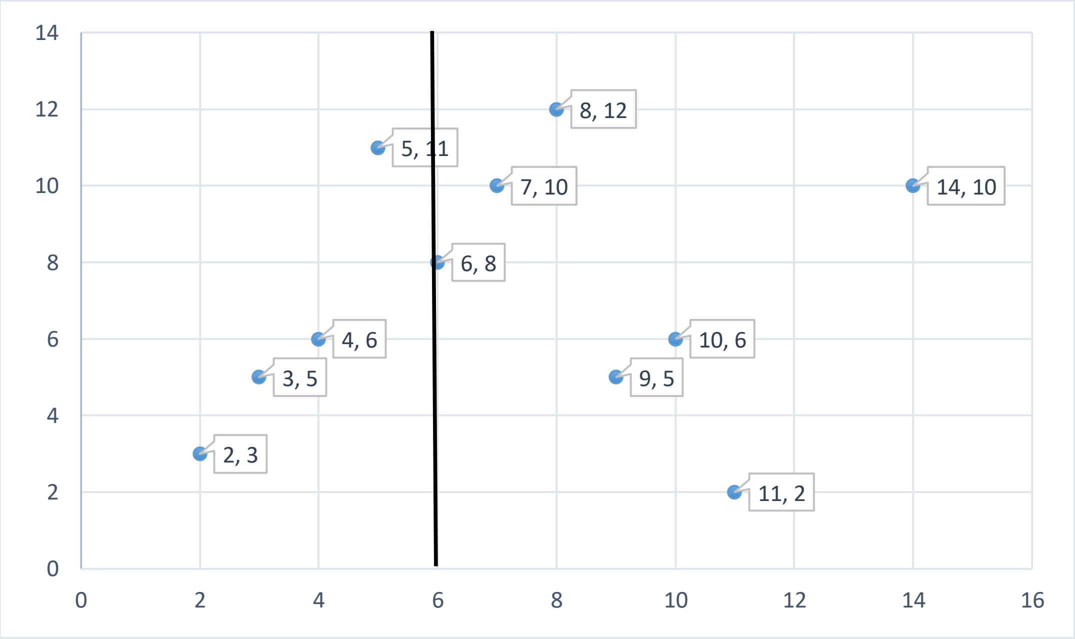


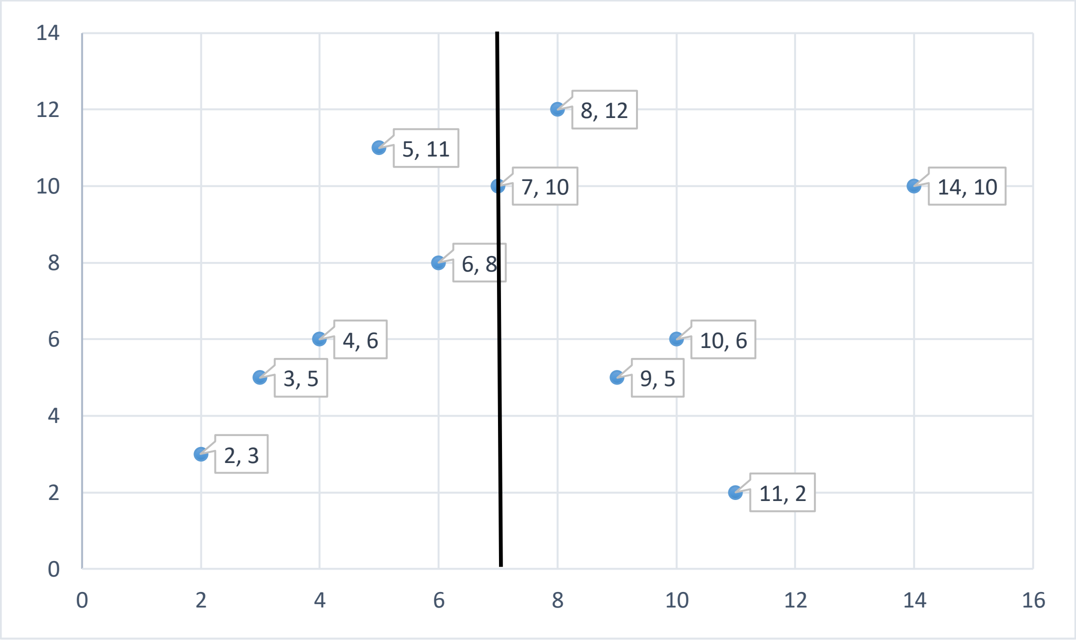
0
2
4
6
8
10
12
14
0
2
4
6
8
10
12
14
16
2, 3
3, 5
4, 6
5, 11
6, 8
7, 10
8, 12
9, 5
10, 6
11, 2
14, 10

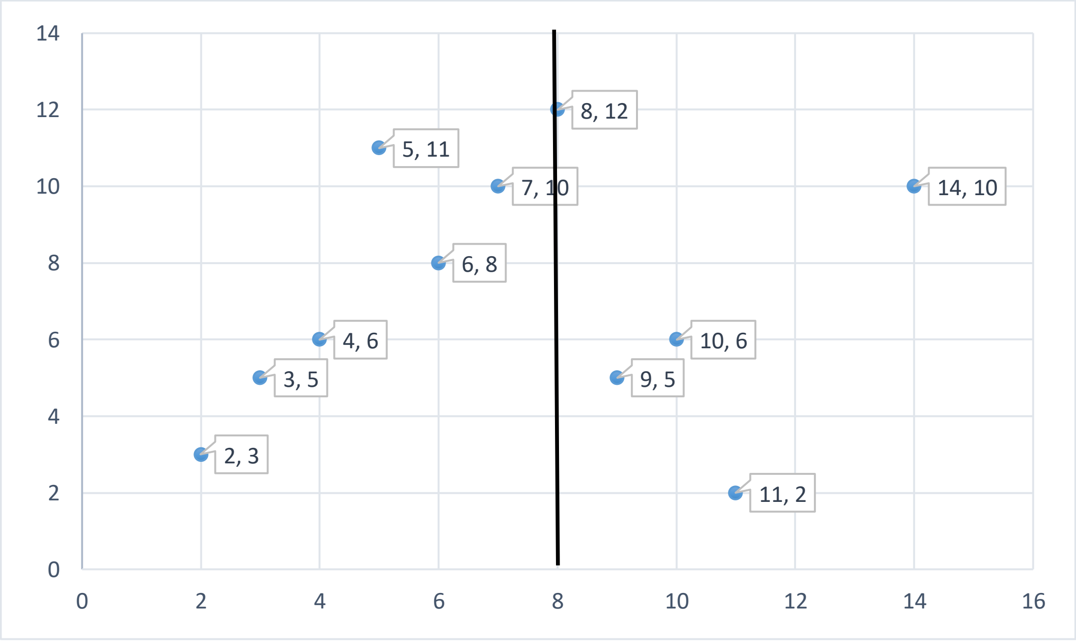


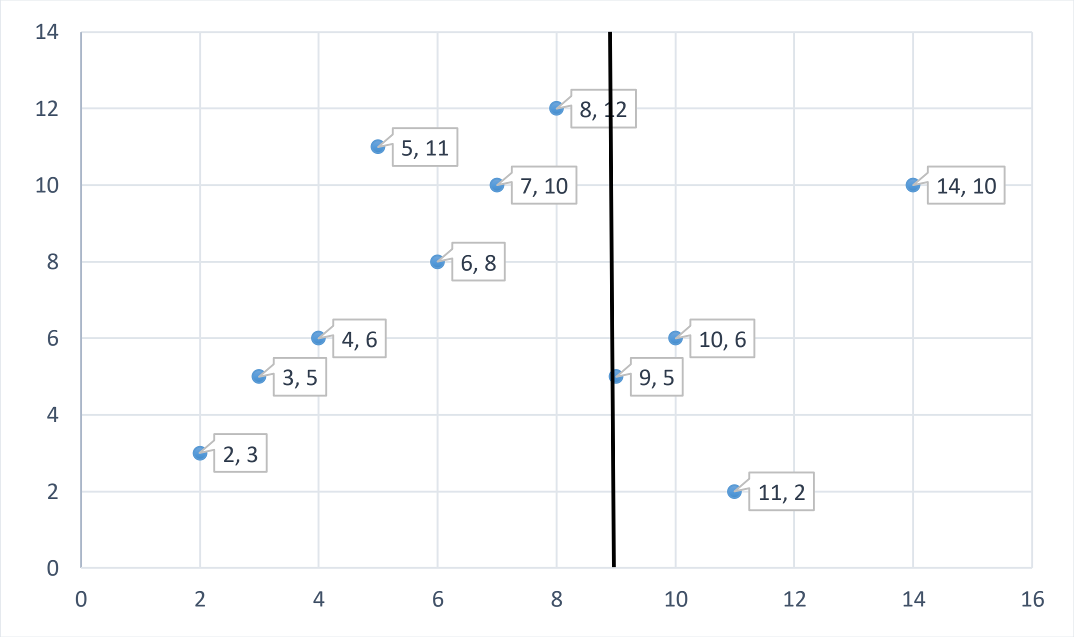
0
2
4
6
8
10
12
14
0
2
4
6
8
10
12
14
16
2, 3
3, 5
4, 6
5, 11
6, 8
7, 10
8, 12
9, 5
10, 6
11, 2
14, 10

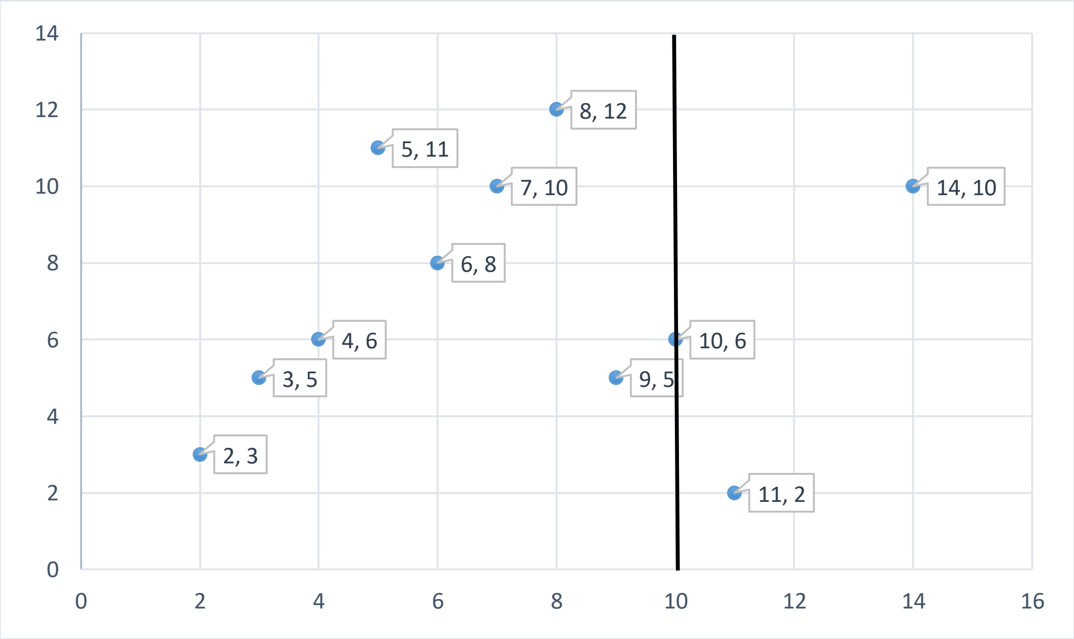

14
12
10
8
6
4
2
0
0
2
4
6
8
10
12
14
16
8, 12
5, 11
7, 10
14, 10
6, 8
4, 6
10, 6
3, 5
9, 5
2, 3
11, 2

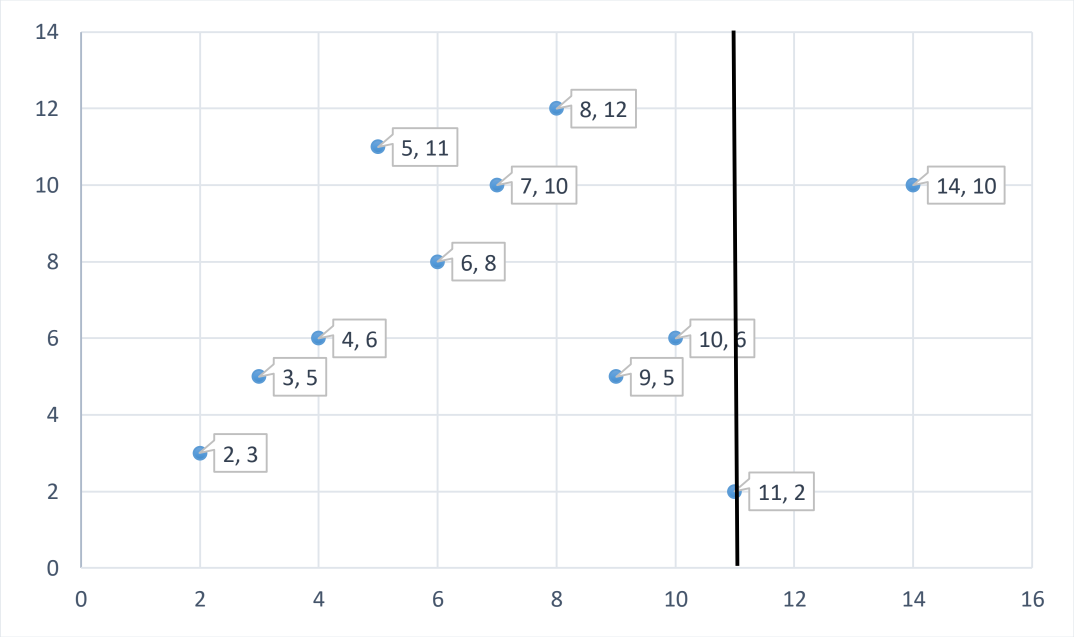

14
12
10
8
6
4
2
0
0
2
4
6
8
10
12
14
16
8, 12
5, 11
7, 10
14, 10
6, 8
4, 6
10, 6
3, 5
9, 5
2, 3
11, 2

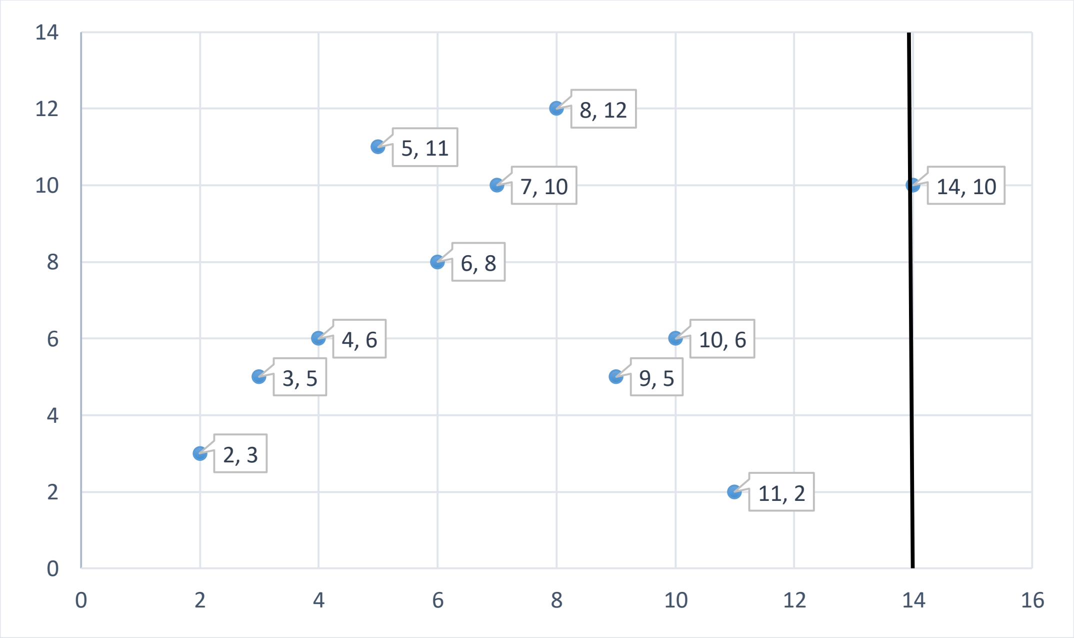

14
12
10
8
6
4
2
0
0
2
4
6
8
10
12
14
16
8, 12
5, 11
7, 10
14, 10
6, 8
4, 6
10, 6
3, 5
9, 5
2, 3
11, 2


14
12
10
8
6
4
2
0
0
2
4
6
8
10
12
14
16
8, 12
5, 11
7, 10
14, 10
6, 8
4, 6
10, 6
3, 5
9, 5
2, 3
11, 2


14
12
10
8
6
4
2
0
0
2
4
6
8
10
12
14
16
8, 12
5, 11
7, 10
14, 10
6, 8
4, 6
10, 6
3, 5
9, 5
2, 3
11, 2


14
12
10
8
6
4
2
0
0
2
4
6
8
10
12
14
16
8, 12
5, 11
7, 10
14, 10
6, 8
4, 6
10, 6
3, 5
9, 5
2, 3
11, 2


14
12
10
8
6
4
2
0
0
2
4
6
8
10
12
14
16
8, 12
5, 11
7, 10
14, 10
6, 8
4, 6
10, 6
3, 5
9, 5
2, 3
11, 2


14
12
10
8
6
4
2
0
0
2
4
6
8
10
12
14
16
8, 12
5, 11
7, 10
14, 10
6, 8
4, 6
10, 6
3, 5
9, 5
2, 3
11, 2

2. How algorithm is analyzed? Explain the different algorithm analysis techniques.

The analysis of algorithm gives good insight of the algorithm.

It basically deals with the process of analyzing time and space complexity, after designing the algorithm with the capability of handling large amount of data.

Then programmer must answer the following questions:

Is the algorithm correct?

The algorithm generates the required result or not?/ Does the algorithm terminates for all the inputs under problem domain?

How long will my program take to execute completely?

Why my program runout memory?

To answer these questions there is scientific method/model which includes following operations:

Observe some features generally with precise measurement.

Hypothesize a model that is consistent with the observation.

Predict event with hypothesis.

Verify the predictions by making further observations.

Validate by repeating until the hypothesis and observation agree.

The scientific model can be analyzed in java programming by analyzing the time complexity by built in function stopwatch.java , which evaluates the complete runtime of program.

```
Public class stopwatch {

Stopwatch() // creates stopwatch

Double elapsedTime() // return elapsed time
```

*The time is measured by threesum.java function*

The Mathematical model

In mathematical model we basically focus on two terms :

Tilde Approximation

Order of growth classification.

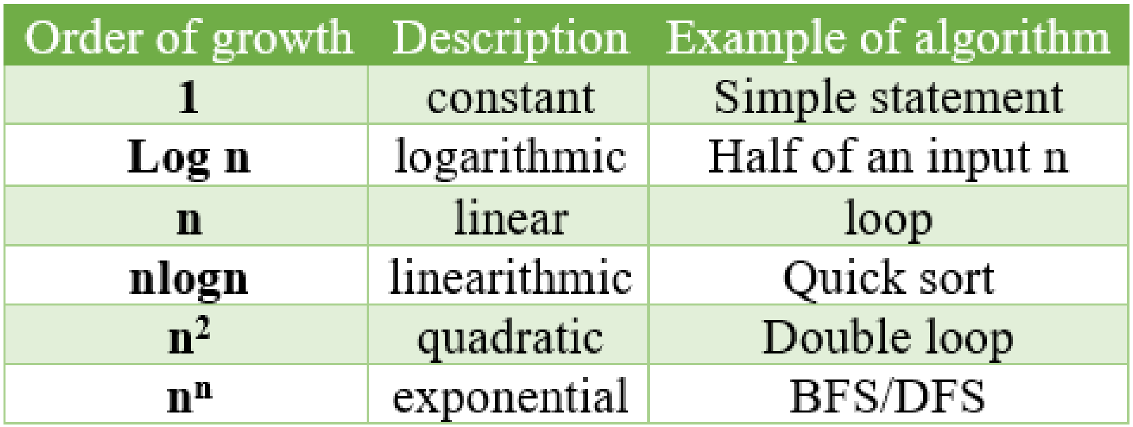
Tilde approximation discards lower order term of a complexity function to simplify the problem and order of growth is calculated by analyzing the power associated with variable.

The Mathematical  model

For eg:

| Complexity Function | Tilde Approximation | Order of growth |
|---|---|---|
| $4n^3 + 5n^2 +5$ | $4n^3$ | $n^3$ |
| $3n^2/4 – 7n$ | $3n^2/4$ | $n^2$ |
| Log  n – 3 | Log n | Log n |
| 7 | 7 | 1 |

Different types of order of growth:



| Order of growth | Description | Example of algorithm |
|---|---|---|
| **1** | constant | Simple statement |
| **Log n** | logarithmic | Half of an input n |
| **n** | linear | loop |
| **nlogn** | linearithmic | Quick sort |
| **$n^2$** | quadratic | Double loop |
| **$n^n$** | exponential | BFS/DFS |

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

Theory of algorithm

GOAL : establish difficulty to the problem and develop optimal algorithm.

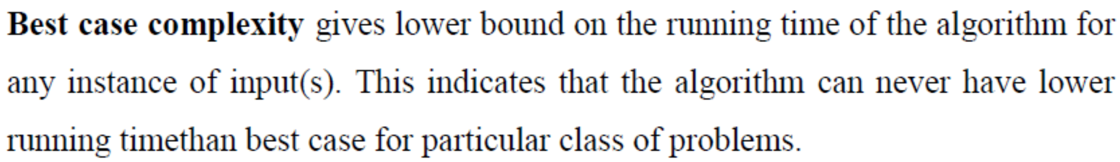
Where the analysis of algorithm is done  in three ways:

Best case

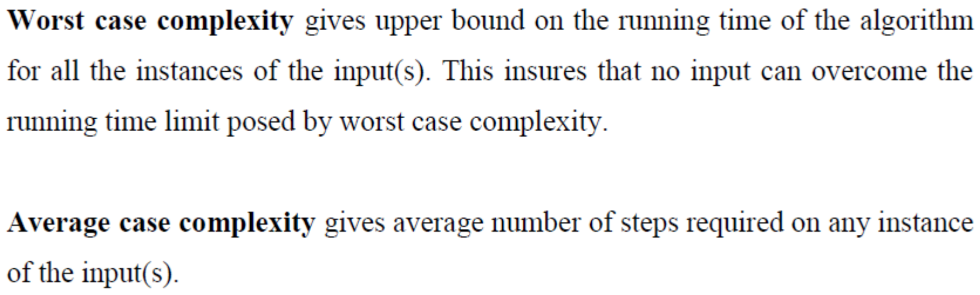
Average case

Worst case

Theory of algorithm

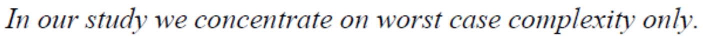


**Best case complexity** gives lower bound on the running time of the algorithm for any instance of input(s). This indicates that the algorithm can never have lower running timethan best case for particular class of problems.

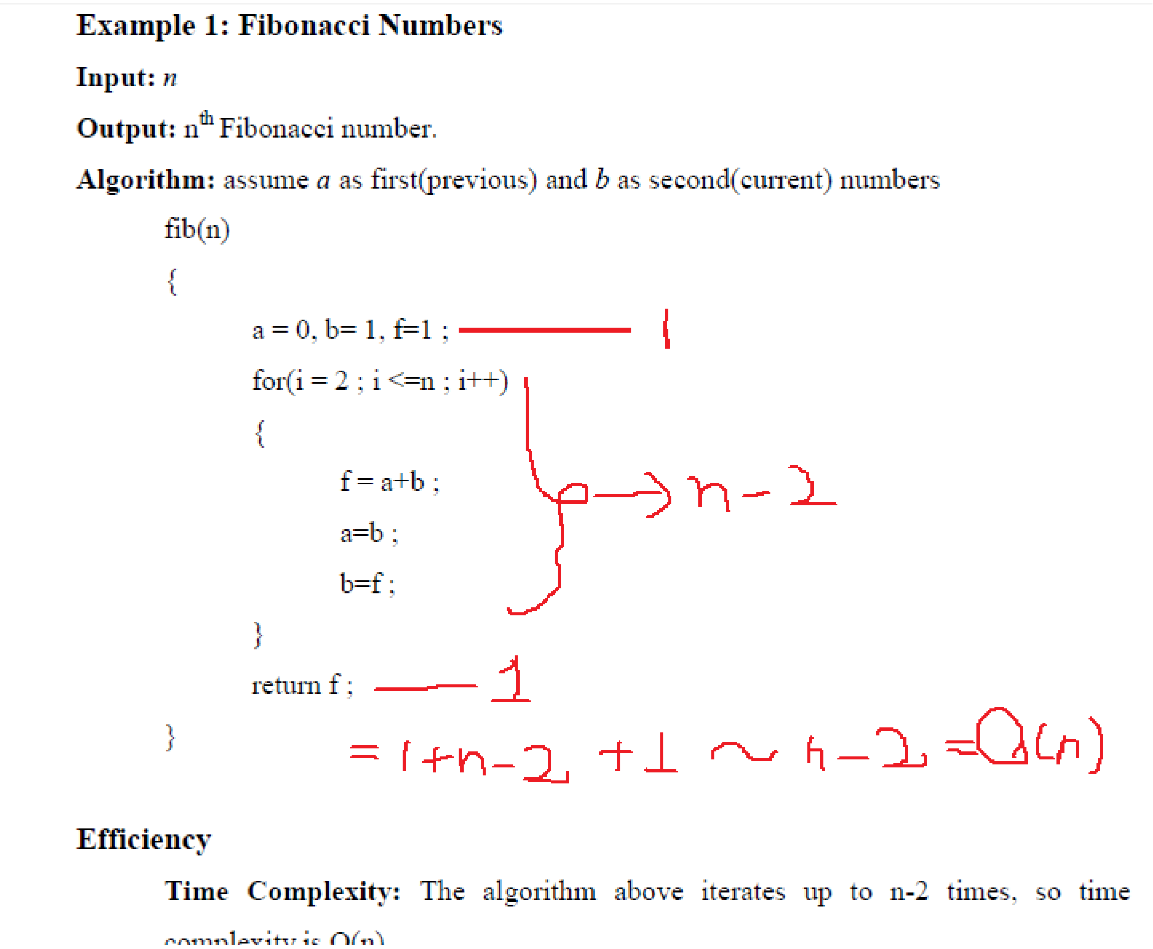


**Worst case complexity** gives upper bound on the running time of the algorithm for all the instances of the input(s). This insures that no input can overcome the running time limit posed by worst case complexity.

**Average case complexity** gives average number of steps required on any instance of the input(s).



*In our study we concentrate on worst case complexity only.*



**Example 1: Fibonacci Numbers**

**Input:** *n*

**Output:** n$^{th}$ Fibonacci number.

**Algorithm:** assume *a* as first(previous) and *b* as second(current) numbers

```
fib(n)
{
        a = 0, b= 1, f=1 ;  ————  1
        for(i = 2 ; i <=n ; i++)
        {
                f = a+b ;          } → n-2
                a=b ;
                b=f ;
        }
        return f ;  ——  1
}             = 1+n-2 +1 ~ n-2 = O(n)
```

**Efficiency**

**Time Complexity:** The algorithm above iterates up to n-2 times, so time complexity is O(n).

Theory of algorithm

The analysis of algorithm can be done by using asymptotic notation to get mathematical function of best, worst and average case.the asymptotic notations are:

Big O (worst case)

Big Omega (Best Case)

Big Theta (average case)