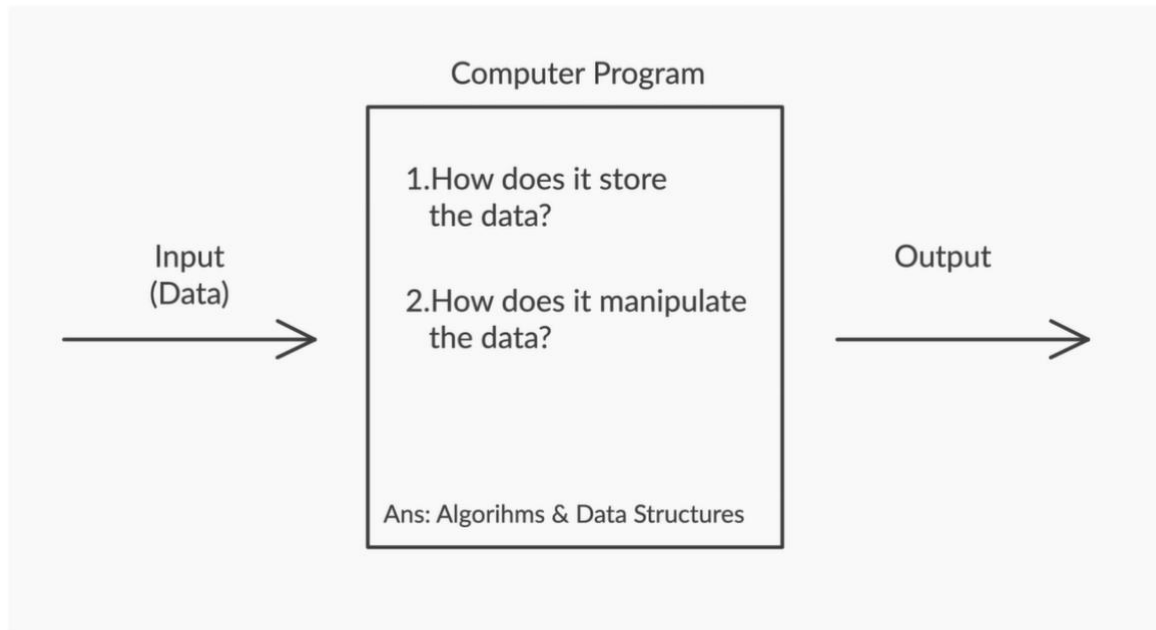# *Algorithm Analysis*

Nemish Zalavadiya

1: Algorithms

Introduction to algorithms

you have seen the importance of algorithms in a computer program. Specifically, Computer Program = Algorithms + Data structures
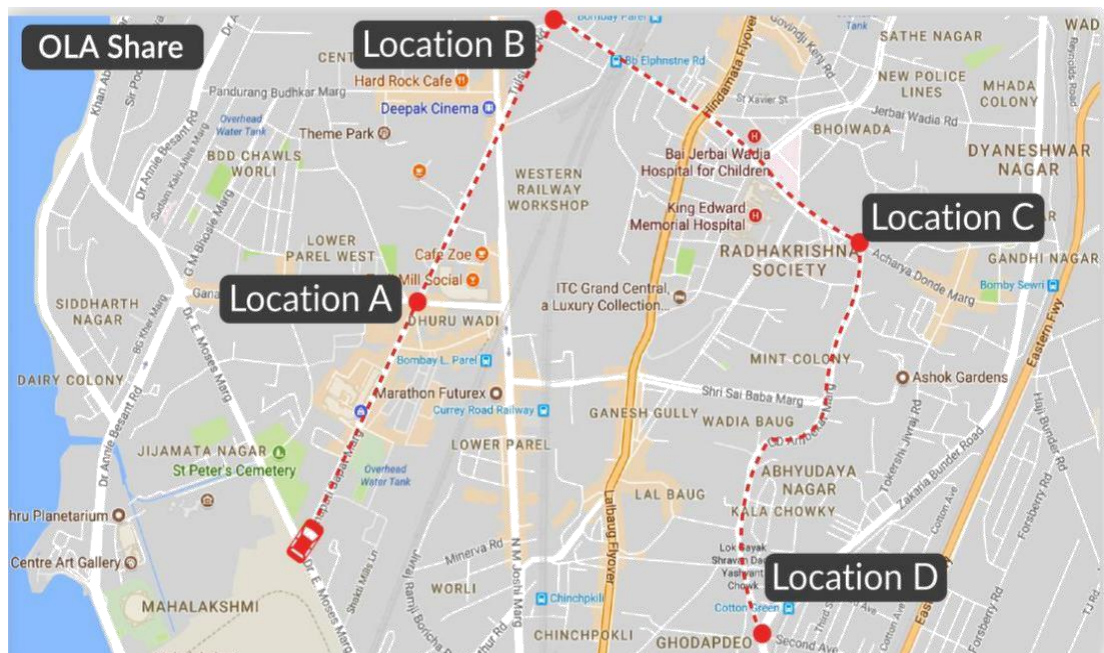

What is an algorithm?

An algorithm is a method of solving a problem through a set of sequential instructions.

Example 1: Ola app, on your request to pick from location A and drop at location B, an algorithm running behind the screen helps to locate a cab nearby and find the shortest route to destination.
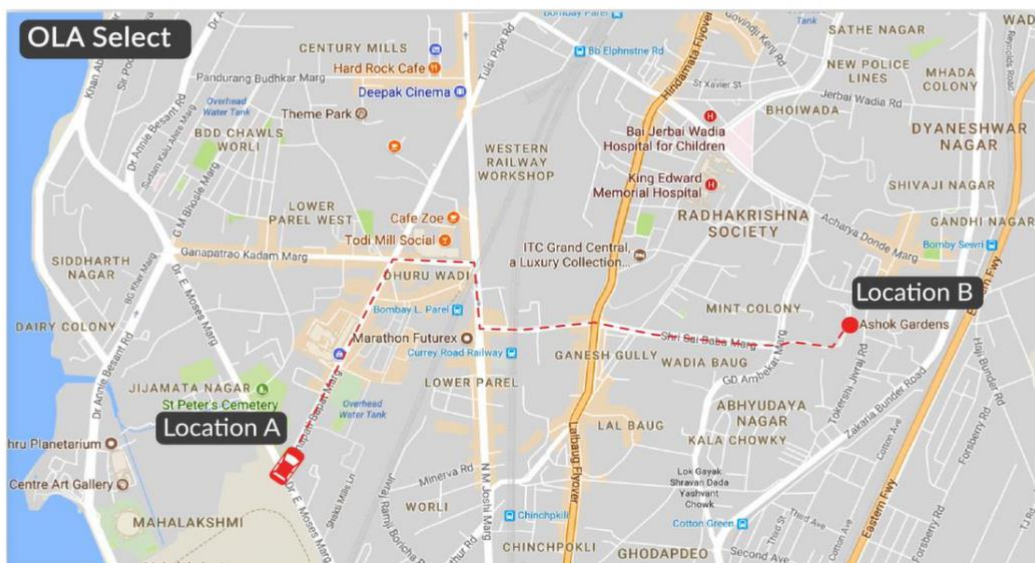
Example 2: In a similar way, Ola share has another algorithm which helps to identify the multiple requests from different people on route to the destination and find out the shortest distance to pick and drop in their respective locations.

Algorithm 1

We discussed a practical scenario where students can register for a course both online mode & offline at academic office and certain students registered for the same course in both the modes.

an algorithm to find out the student ID's registered twice for the same course.
Algorithm 1

```java
public void findDuplicates(int[] id) {
    System.out.println("Duplicate student id : ");
    for (int i = 0; i < id.length; i++) {
        for (int j = i+1; j < id.length; j++) {
            if (id[i] == id[j]) {
                System.out.print(id[i] + " ");
                break;
            }
        }
    }
}
```

We have assumed that the university consists of not more than 10000 students and student ID's are a set of integers between 1 – 10000.

The combined data of student ids are stored in an array variable **id [ ]**

				......	

i = 0 1 2 3   n - 1
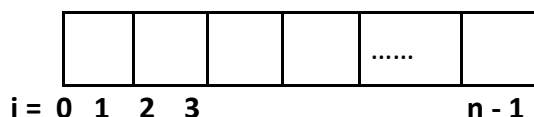
id [i] : i^th student id registered
   n : length of the array

The nested for loops in algorithm 1 helps to iterate across the data and compare each student id with other already registered ids to find duplicate student IDs.

For the first iteration of outer loop(i) i.e., when i = 0, the inner loop(j) iterates from 1 (i+1) to n - 1 and compares the first student id (id [0]) with other student IDs on right-hand side till it finds the duplicate student ID in id [0].

When i = 0

0   1   2   .....   n-1
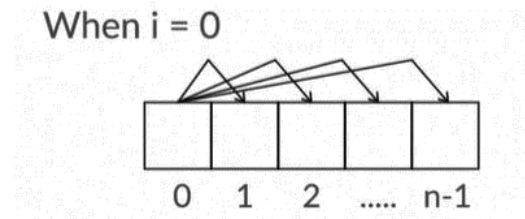
For the second iteration of outer loop(i) i.e., when i = 1, the inner loop(j) iterates from 2 (i + 1) to n − 1 and compares the second student id (id [1]) with other student IDs on the right-hand side till it finds the duplicate student ID of id [1].

When i = 1

0   1   2   .....   n-1
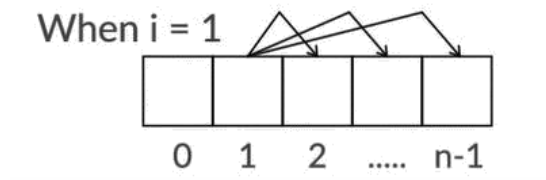
For each iteration of inner loop(j), instruction set will compare id[i] & id[j], if it is true then prints the duplicate ID and breaks out of inner loop(j).

In this way the outer loop(i) is iterated from 0 to n − 1 and for each iteration of outer loop(i), inner loop(j) instruction set is executed for n − (i + 1) and prints all the duplicate IDs found.
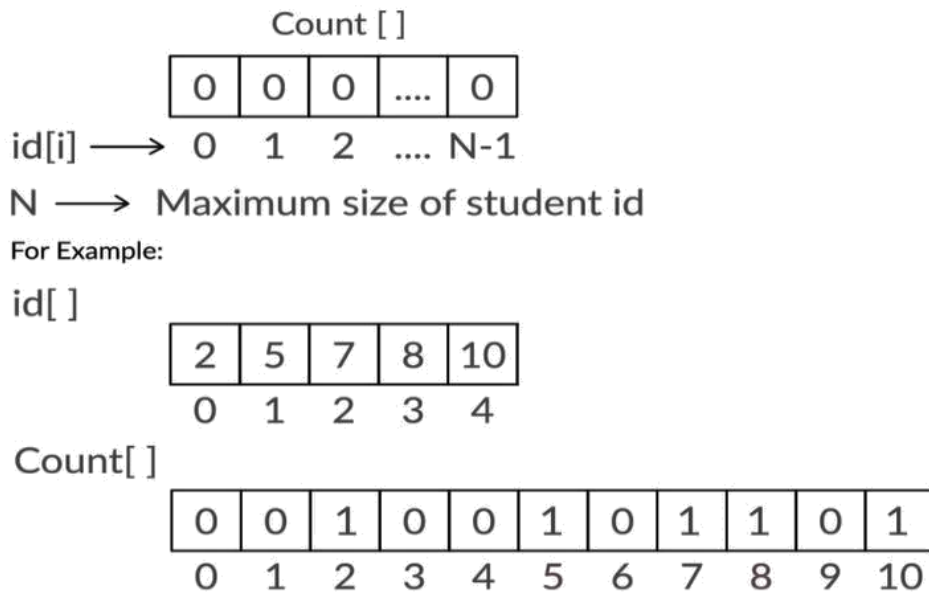
Algorithm 2

As you know, typically, there is more than one way to solve the same problem and you have seen another approach in finding the duplicate student IDs from given data.

Algorithm 2

```java
public void findDuplicates(int[] id) {
        System.out.println("Duplicate data : ");
        int count[] = new int[10000];
        for (int i = 0; i < id.length; i++) {
            count[id[i]]++;
            if (count[id[i]] == 2)
                System.out.print(id[i] + " ");
        }
        System.out.println();
}
```

In algorithm 2, declare an extra array variable count [ ] with the maximum size of student id i.e. 10000 besides an array id [ ] for student registrations data.

Count array index values refers to corresponding student IDs. For id[i] (i^{th} student), count[id[i]] increments to 1, if same id appears again then the same cell is incremented to 2.

Count [ ]

0	0	0	....	0
0	1	2	....	N-1

id[i] ⟶ 0 1 2 .... N-1

N ⟶ Maximum size of student id

For Example:

id[ ]

2	5	7	8	10
0	1	2	3	4

Count[ ]

0	0	1	0	0	1	0	1	1	0	1
0	1	2	3	4	5	6	7	8	9	10

As you can observe, there are no duplicates in the above given data and so the corresponding cells of student ID in the count array are incremented to 1.

Whereas if there is a duplicate student ID in the given data,

For Example:

id[ ]

2	5	7	8	10	5	8
0	1	2	3	4	5	6

Count[ ]

0	0	1	0	0	2	0	1	2	0	1
0	1	2	3	4	5	6	7	8	9	10
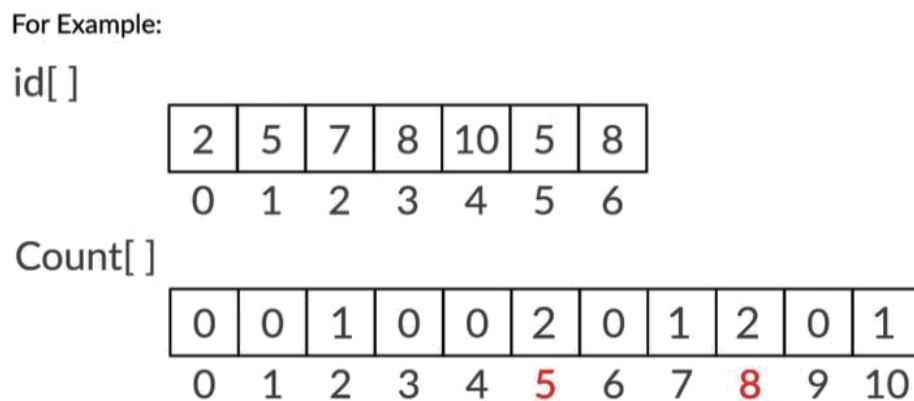
Here, Student ID – 5 & 8 are repeated in the given data, so you can see that the arrays cells corresponding to the duplicate student ID (5 & 8) are incremented to 2.

When count [id[i]] is equal to 2 then prints the duplicate student ID.

Now, you need to analyze the above two algorithms and find the efficient algorithm. For which we discussed that the parameters based on which an algorithm is analyzed are

- How long does an algorithm takes to process output?
- How much memory space is required to execute an algorithm?

Both execution time and memory space are calculated as a function of input size(n).

In general, to analyze an algorithm you need to calculate the no of times certain instruction set is executed rather than the exact time values as it depends on various external factors such as processor speed, the compiler etc. Also, while analyzing an algorithm, we consider worst case possible.

To understand worst case, you saw an analogy of unlocking a lock when you have ten different keys, of which you are not aware of the right key. When you try to unlock on a trial and error method, the worst case is that the lock is unlocked on your tenth attempt and the best case is that it unlocked on your first attempt.
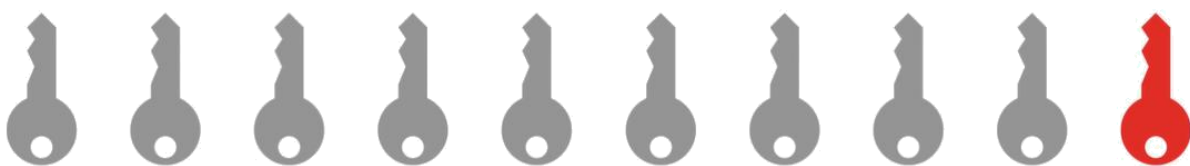
WORST CASE

Time complexity

The worst case to be considered in algorithm 1 is when there is no duplicate student IDs and for every i^{th} iteration of outer loop(i), inner loop instruction set is executed $n - (i + 1)$ times.
Therefore, in total the no of times inner loop instruction set is executed = $(- 1) + (- 2) + \ldots\ldots. + 1$

In a similar way, algorithm 2 executes certain instruction set to find the duplicate student IDs n times

On assuming constant times:

1. C_1 for rest of the instruction set like declaring variables, passing data to functions, etc.,
2. C_2 for the instruction set inside the loop to find duplicate student IDs.

Therefore, the total time taken,

Algorithm 1 – $T(n) = C_1 + \frac{(n-1)}{2} * C_2$

Algorithm 2 – $T(n) = C_1 + n* C_2$

The total time taken to execute an algorithm as a function of input size(n) is called time complexity of an algorithm and is represented by $T(n)$.

The other parameter considered to analyse an algorithm is memory space required to execute an algorithm.

The total memory space required to execute an algorithm as a function of input size(n) is called space complexity of an algorithm and is represented by $S(n)$. In general, you only calculate the extra memory required, not including the memory needed to store the input.

As you have learnt, algorithm 1 uses a constant memory space besides an array variable id [ ] to store student IDs and algorithm 2 uses an extra array variable count [ ], the size of count array variable increases linearly with the increase in the student strength of the university i.e., when student ID exceeds 10000, then the maximum student ID is considered as the size of count array.

Therefore,
Algorithm 1 – $S(n)$ - Constant space, memory space does not depend on the input size

Algorithm 2 – $S(n)$ is linearly proportional to the number of possible students in the university

After obtaining the complexity functions of both algorithms, you have learnt mathematical notations like Big O, Big omega (Ω), Big theta (Θ) called as asymptotic notations which help us to compare the functions of two different algorithms.

Big O

Big O indicates the upper bound (worst case) of the running time or space complexity of an algorithm.

To calculate the Big O of any function, we discussed certain simplification rules as,

- Drop the constant multiplier in a function as it depends on hardware like processor speed, on which the program is run.

  For example,

  i) $f(n) = 2 \Rightarrow f(n) \in O(n)$
  ii) $f(n) = 5 n^2 \Rightarrow f(n) \in O(n^2)$

- Drop less significant terms in a polynomial function. Except for higher order terms, rest of the terms relatively contribute very less in explaining the growth of a function.
  For example,

  $T(n) = 10n^3 + n^2 + 4n + 800. \Rightarrow T(n) \in O(n^3)$

  If n = 1000, then T(n)=10,001,040,800.

  If you drop all the less significant terms except for $10n^3$, then the error rate is 0.01%, which is very minimal.

Definition: Big O is "bounded above by" (upper bound), there exists constants $c > 0$ and $n_0 > 0$ $f(n) \le c g(n)$ for all $n > n_0$, $f(n) \in O(g(n))$

If an algorithm takes the time complexity function as T(n) = $2n^2 - 3n + 6$

When calculating Big O, the time complexity function of an algorithm is **less than or equal** to upper bound and input size(n) is a positive value, as n value increases $-3n + 6 < 0$.

So, $T(n) \le 2n^2$, $n > 2$ $\Rightarrow -3n + 6 < 0$

$\frac{n > 6}{3}$ $\frac{3n > 6}{n > 2}$

Therefore, $T(n) \le 2n^2 \Rightarrow T(n) \in O(n^2)$

Big Omega

Big Omega indicates the lower bound of running time or space complexity of an algorithm. In the scenario of opening a lock, if you dealt with the best case, i.e. on your first attempt itself, you could unlock.

Definition: Big Omega (Ω) is "bounded below by" (lower bound), there exists constants $c > 0$ and $n_0 > 0$ $f(n) \ge c g(n)$ for all $n > n_0$, $f(n) \in \Omega(g(n))$

If an algorithm takes the time complexity function as $T(n) = 2n^2 - 3n + 6$,

When calculating Big Omega, the time complexity function of an algorithm is **greater than or equal** to lower bound.

$T(n) = 2n^2 - 3n + 6$ $\ge$ $2n^2 - 3n$ (because 6 > 0) $\ge n^2 \Rightarrow 2n^2 - 3n \ge n^2$ $\frac{n}{} $ $2n^2 - n^2 \ge 3n$ $\frac{n \ge 3}{n \ge 3}$

So, $T(n) \ge n^2$, for all $n \ge 3$
Therefore, $T(n) \in \Omega(n^2)$

Big Theta

Big Theta represents an in-between case, bounded both above and below the running time or space complexity of an algorithm.

Definition: Big Theta (Θ) is "bounded above and below", there exists constants $c_1 > 0$, $c_2 > 0$ and $n_0 > 0, c_1 . g(n) \leq f(n) \leq c_2 . g(n)$, for all $n > n_0$, $f(n) \in \Theta(g(n))$

If an algorithm takes the time complexity function as $f(n) = 2n^2 - 3n + 6$, As already discussed,

Upper Bound: $\quad 2n^2 - 3n + 6 \leq 2n^2$
Lower Bound: $\quad n^2 \leq 2n^2 - 3n + 6$
So, $\quad n^2 \leq 2n^2 - 3n + 6 \leq 2n^2$, $f(n) \in \Theta(n)$

Among all the three asymptotic notations, the most used notation to compare algorithms is Big O. Therefore, you always tend to find the worst case of an algorithm with respect to the input size(n).

You have learnt two rules which are used in general while analysing for time complexity of algorithms,

Rule of sums

In an algorithm, when two for loops are one after another as follows

```java
for(int i = 0;i < n;i++){
    //instruction set
}
for(int j = 0;j < m;j++){
    //instruction set
}
```

Then the first for loop(i) instruction set is executed n times and second for loop(j) instruction set is executed m times.

So, in total n + m steps taken in executing this kind of an algorithm. If you consider the time complexity of this algorithm, $f(n) = n + m$ (neglecting the constant time taken by instruction set) If $n > m \Rightarrow n + m \leq 2n$ then you can say that $f(n) = n + m$

On dropping the constant multiplier 2, $f(n) \in O(n)$

Rule of products

In an algorithm, when two for loops are nested as follows

```java
for(int i = 0;i < n;i++){
    for(int j = 0;j < m;j++){
        //instruction set
    }
}
```

Then the outer loop(i) is iterated n times and for each iteration of outer loop(i), inner loop(j) instruction set is executed for m times. In total, the instruction set inside the inner loop is executed n * m times, if you consider the time complexity function of algorithm, $T(n) = n*m$ (neglecting the constant time taken by instruction set).

if $n > m \Rightarrow$ then you can say that $T(n) = $

$m \leq n$ $n*n$

So, $T(n) \in O(n^2)$

You have learnt different functions encountered when analysing algorithms as follows,

Constant function	- T(n) = C (C is a constant)
	T(n) $\in$ O(1)
Linear function	- T(n) = $C_1 n + C_2$ (C_1 , C_2 are Constants)
	T(n) $\in$ O(n)
Quadratic function	- T(n) = $C_1 n^2 + C_2 n + C_3$ (C_1 , C_2 , C_3 are Constants)
	T(n) $\in$ O(n²)
Cubic function	- T(n) = $C_1 n^3 + C_2 n^2 + C_3 n + C_4$ (C_1 , C_2 , C_3 are Constants)
	T(n) $\in$ O(n³)

To find out the relative efficiency among the different functions, the following table has been discussed in detail on how it helps you to give an idea on the different time taken for different input sizes.

To understand the below table, we have assumed that a computer performs 1 billion (10^9) operations per second.

Input size(n)	O(1)	O(log n)	O(n)	O(n log n)	O(n^2)	O(n^3)	O(2^n)
10		3 ns	0.01 µs	0.03 µs	0.1 µs	1 µs	1 µs
20		4 ns	0.02 µs	0.09 µs	0.4 µs	8 µs	1 ms
30		5 ns	0.03 µs	0.15 µs	0.9 µs	27 µs	1 s
40	Running	5 ns	0.04 µs	0.21 µs	1.6 µs	64 µs	18 min
50	time	5 ns	0.05 µs	0.28 µs	2.5 µs	125 µs	13 days
100	does not depend	6 ns	0.1 µs	0.66 µs	10 µs	1 ms	4×10^3 years
10^3	on input size(n)	9 ns	1 µs	9.96 µs	1 ms	1 s	32×10^{283} years
10^4		13 ns	10 µs	130 µs	100 ms	16.67 min	
10^5		16 ns	100µs	1.66 ms	10 s	11.57 days	
10^6		19 ns	1ms	19.92 ms	16.67 min	31.17 years	

Hence, the efficiency order of different functions

(1)> (   )> ( )> (   )> (2) >   (3)

Using this, you can compare different complexity functions of algorithms and find the efficient algorithm.

After finding the relative efficiency of different functions, you have learnt how to compare algorithms to find duplicate student IDs and find the efficient algorithm

Time complexity:

Algorithm 1 – T(n) = C_1 + $(^{-1})$ * C_2 ⇒ T(n) ∈ O(^{2}) – Quadratic function
Algorithm 2 – T(n) = C_1 + n* C_2 ⇒ T(n) ∈ O( ) – Linear function
you can observe that algorithm 2: O( ) is more efficient when compared to algorithm 1: O(^{2})

Space complexity:

Algorithm 1 – S(n) ∈ O(1) – constant function
Algorithm 2 – S(n) ∈ O(n) – linear function

You can observe that algorithm 1: O(1) is more efficient when compared to algorithm 2: O(n)

Time vs Space Complexity Trade-off:

As you've seen with the algorithms for identifying duplicated student ids, the two algorithms have made different trade-offs in terms of time and space.

Specifically:
- Algorithm 1 runs slower, but uses less memory
- Algorithm 2 runs faster, but uses more memory

As a software developer, you'll often face this kind of dilemma while designing programs and software. Do you write programs that runs fast but uses lots of memory space? Or do you write program that runs slower but uses less memory space?

The answer is it depends.

For example, if you are writing software to do high-frequency stock trading where every microsecond can be the difference between earning or losing hundreds of thousands of dollars, you will likely want to design programs can execute very quickly at the expense of using lots of memory space.

On the other hand, you may be writing software that runs on smartphones where the memory available to the software is limited. In this situation, you may want to write software that uses less memory but runs a bit more slowly.

Therefore, use your best judgement when it comes to Time vs Space Complexity trade-off. Identify your business needs or constraints, and then decide if you should trade space for time or vice versa.

In this session, you have been introduced to a mathematical function called Fibonacci sequence, which is defined as:

F(0) = 0

F(1) = 1

F(n) = F(n - 1) + F(n - 2)

We made a small change in the above function as F(n) = [F(n - 1) + F(n - 2)]%10, i.e.by dividing the function with modulo 10 in order to avoid integer overflow error for higher input(n) values. We then discussed on calculating n^{th} number of function i.e., n = 0, 1, 2, 3, 4, 5 …

F(0) = 0

F(1) = 1

F(n) = [F(n-1) + F(n-2)]%10

If n = 0, then F(0) = 0

If n = 1, then F(1) = 1

If n = 2, then F(2) = [F(2 - 1) + F(2 - 2)]%10

= [F(1) + F(0)]%10

= [0 +1]%10 = 1

If n = 3, then F(3) = [F(2) + F(1)]%10

= [1 + 1]%10

= 2

If n = 4, then F(4) = [F(3) + F(2)]%10

= [2 + 1]%10

= 3

If n = 5, then F(5) = [F(4) + F(3)]%10

= [3 + 2]% 10

= 5

If n = 6, then F(6) = [F(5) + F(4)]%10

= [5 + 3]%10

= 8

If n = 7, then F(7) = [F(6) + F(5)]%10

= [8 + 5]%10

= 13%10

= 3

You have learnt algorithm 1 to generate n^{th} number of the function F(n) = [F(n - 1) + F(n - 2)]%10.

Algorithm 1

```java
public int fibonacci(int n){
    if (n < 2)
        return n;
    else
        return (fibonacci(n - 1) + fibonacci(n - 2))%10;
}
```

As you have learnt, the function fibonacci() is called inside the same function which is called as recursive function

Recursive function

Recursive function is a function which calls itself during its execution, and a recursive function typically needs to define two cases:

1. the base case that returns a definitive value and
2. the recursive case where the recursive function calls itself and tries to solve smaller parts of the problem at hand

In algorithm 1, the if condition acts as a terminating or base condition which returns the definite values to end the recursive calls of the function fibonacci() and else condition acts as a recursive case, which calls the same function fibonacci() again till the passing argument satisfies the base condition. Now to understand how exactly the recursion function generates the Fibonacci number for a given input(n), we discussed a recursion tree to generate 5th Fibonacci number.

RECURSION

```
                        F(5) = 3 + 2
                             = 5

        F(4) = 1 + 2                      F(3) = 1 + 1
             = 3                               = 2

 F(3) = 1 + 1        F(2) = 1 + 0    F(2) = 1 + 0      F(1) = 1
      = 2                = 1             = 1

F(2) = 1 + 0   F(1) = 1   F(1) = 1  F(0) = 0  F(1) = 1   F(0) = 0
     = 1

F(1) = 1       F(0) = 0
```
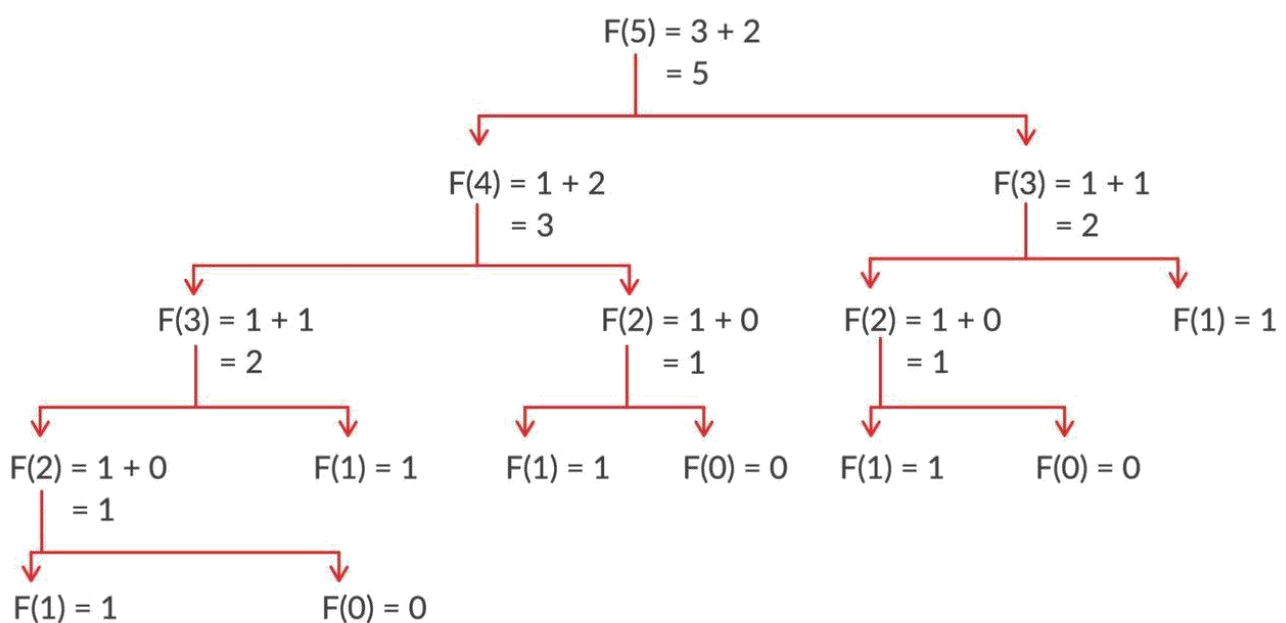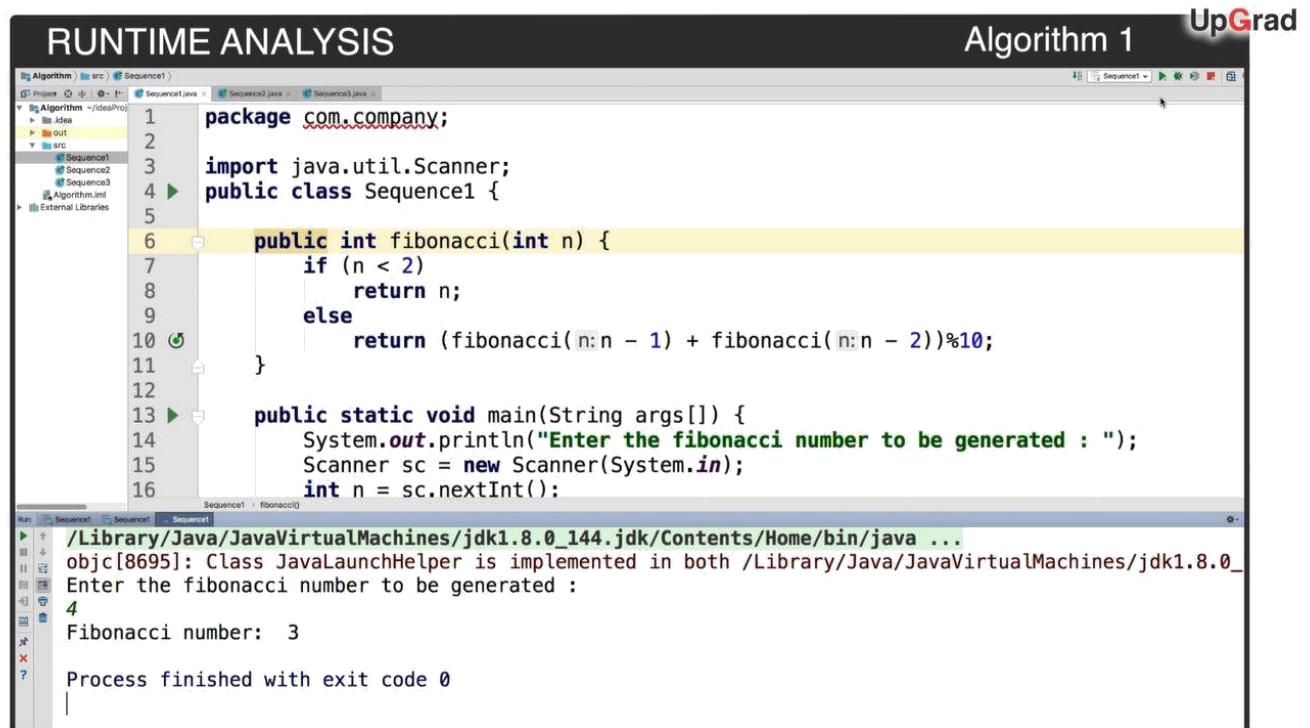
In which F(0) & F(1) are the terminating conditions and helps to find rest of the values and print final output.

Then, we have demonstrated the code of algorithm 1 for different input size like n = 4, n =42

But when the input given is n = 100, then the compiler took time to process and you never saw the output. As suggested, If you have run this code in your system for the input n = 100 even after waiting for hours the compiler must not have given any output and still processing.

We considered that,

T(n) = No of additions required to compute F(n)

No of additions required to calculate F(0)& F(1) = 0, as the if condition returns the same value without any calculations(additions) required.

$$T(n) = \text{No. of additions to compute } F(n)$$

$$T(n-1) = \text{No. of additions to compute } F(n-1) \quad + \quad \text{No. of additions to compute } F(n-2) = T(n-2)$$

Therefore, $T(n) = T(n-1) + T(n-2) + 1$

Then, we have calculated the upper bound for the time complexity function as follows,

UPPER BOUND

$T(n) = T(n - 1) + T(n - 2) + 1$
$\quad \leq T(n - 1) + T(n - 1) = 2T(n - 1)$
$\quad \leq 2T(n - 1)$
$\quad \leq 2*2T(n - 2)$
$\quad \leq 2 *2*2T(n - 3)$
$\quad \leq 2^k T(n - k)$
$n - k = 2 \rightarrow k = n - 2$
So, $T(n) \leq 2^{n-2} T(2)$
$\quad \leq 2^{n-2}$ (Because $T(2)= 1$)
$\quad \leq 2^n$
Therefore, $T(n) \in O(2^n)$

$T(n - 1) = T(n - 2) + T(n - 3) + 1$
$\quad \geq T(n - 2) + 1$ (Because $T(n - 3) \geq 0$)
$\quad \rightarrow T(n - 2) + 1 \leq T(n - 1)$

$F(2) = F(0) + F(1)$
$\quad = 0 + 1$
$\quad = 1$
One addition, so $T(2) = 1$

So, the time complexity of algorithm 1 is an exponential function with O(2^n), which is really slow, and this is why algorithm 1 is unable to produce an output when n = 100.

With respect to the memory space i.e., space complexity required for algorithm 1, we have again discussed with an example of generating 5th Fibonacci number and how the memory space is occupied on each recursive call?

Think of memory space as partitions as shown and each partition is occupied as the function is called and the same function pops out of memory space when the process is done and returns the value.

Space Complexity
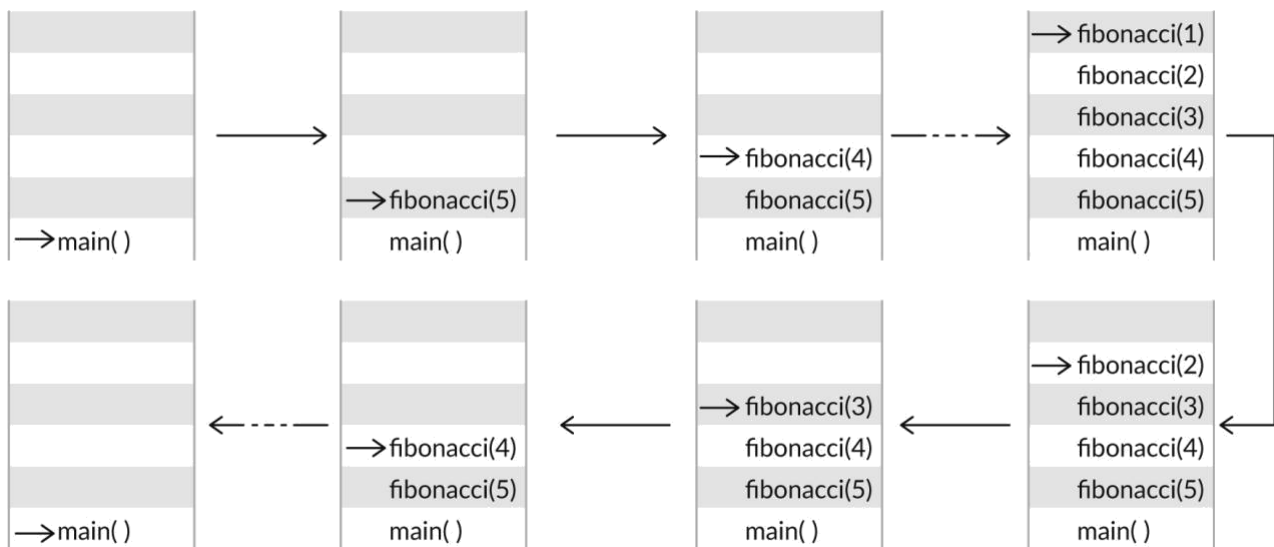
As you can observe, the maximum memory space required is proportional to the Fibonacci number generated. So, space complexity S(n) ∈ O(n) – linear function

We have discussed a real-world problem "file search" in a laptop using recursion,

The file search process as follows,

Consider each folder is a directory and it consists of many sub directories(folders), in order to search a specific filename, pass in the filename that you are looking for and the file directory path(folder path) that you want to start the search to the recursive function.

Then follow certain steps inside the recursive function as
1. List all the contents inside the file directory passed to the function
2. Loop through the contents inside the file directory,
    - If a specific content is another directory, recursively call the function and pass in the file directory and its path.

- If the content is a file, check and see if the file matches with the filename you are searching for. If it matches, then return "file is found" with the file path.

This has been demonstrated using java code as follows,

Algorithms > src > sde > program > analysis > FileFinder   FileFinder ▾ ▶ ☀

```java
8        * @param fileName name of the file to be searched
9        * @param inputPath path of the directory in which the file is to be sea
10       */
11      public void findFile(String fileName, String inputPath)
12      {
13          File directory = new File(inputPath);
14          if (directory.isDirectory()){
15              File[] list = directory.listFiles();
16              for (File file : list){
17                  if (file.isDirectory()){
18                      findFile(fileName, file.getAbsolutePath());
19                  }else if (fileName.equals(file.getName())){
20                      System.out.println(inputPath + "/" + fileName + " found'
21                  }
22              }
23          }
24      }
25
26      public static void main(String args[]) {
27          String projectPath =  System.getProperty("user.dir");
28          FileFinder fileFinder = new FileFinder();
29          fileFinder.findFile( fileName: "Algorithms.iml", projectPath);
30      }
31  }
32
```

Algorithm 2

Now, coming back the function F(n) = [F(n - 1) + F(n - 2)]%10,

In algorithm 1, we have not stored the values F(2) & F(3) and had to calculate each time whenever required

UpGrad

RECURSION
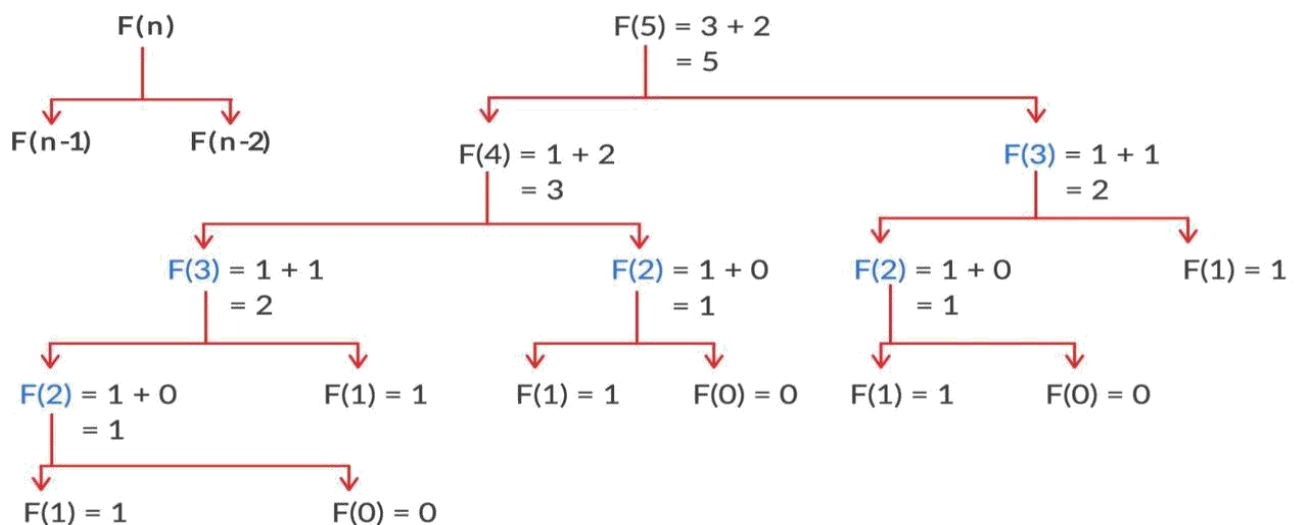
We try and overcome the redundant calculations in algorithm 2 by storing all the values calculated in an array variable f [ ]. Therefore, if we need to recall a Fibonacci value that has been previously calculated, we can simply refer back to the values stored in f[ ] rather than recalculating the value again.

Algorithm 2

```java
public void fibonacci(int n){
    int[] f = new int[n];
    f[0] = 0;
    f[1] = 1;
    for(int i = 2;i <= n;i++)
        f[i] = (f[i - 1] + f[i - 2])%10;
    System.out.println("Fibonacci number : "+f[n]);
}
```
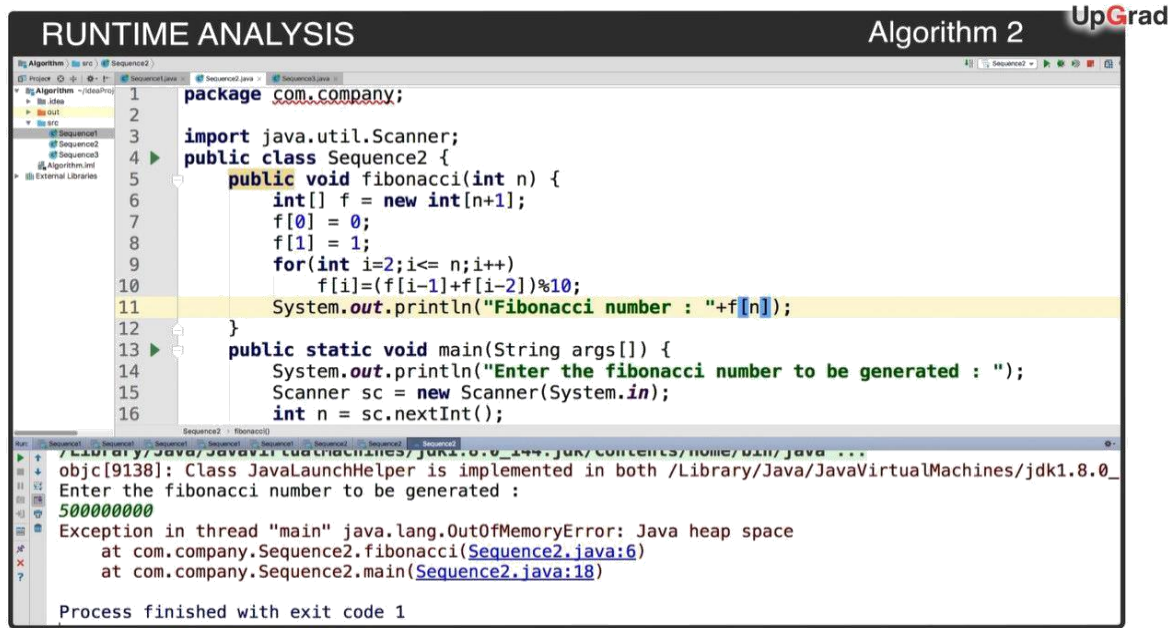
As you can observe, there is only one for loop iterating from 2 to n i.e., n – 1 times executes certain instructions to generate n^{th} number of the function F(n) = [F(n - 1) + F(n - 2)]%10

Therefore,

Time Complexity:

T(n) = No. of additions to compute F(n)

So, T(n) = n – 1

Therefore, T(n) ∈ O(n), linear time
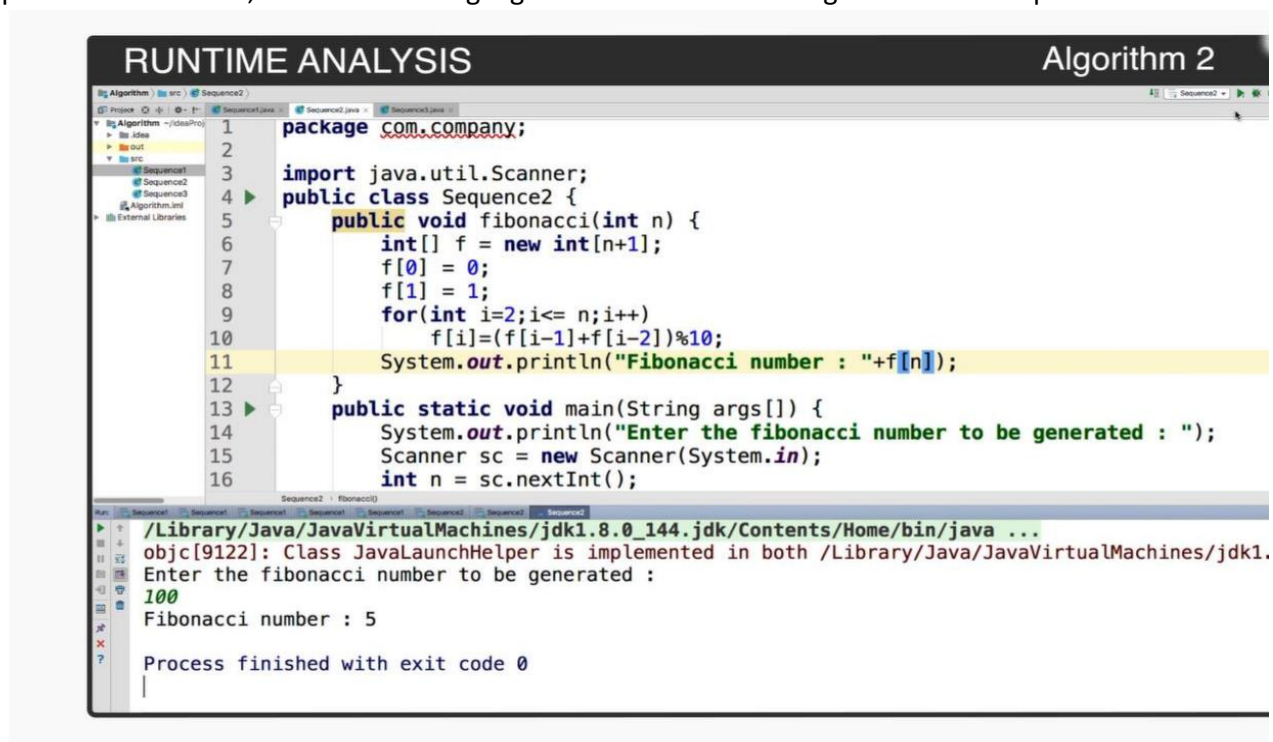
With respect to the memory space, besides declaring variable n for input, you need to create an array variable f[ ] of size n in order to store all the calculated values of the function F(n) = [F(n - 1) + F(n - 2)]%10

Therefore,

Space Complexity:

An extra array variable f[] is defined, whose size is dependent on input variable n So, S(n) ∈ O(n), linear in memory space

After analysing algorithm 2 with respect to time taken and memory space required, we have run the java code for input values like n = 4, n = 100 and using algorithm 2 we are able to generate the output for n = 100.

So, algorithm 2 overcomes the constraint of algorithm 1 and process the nth number of the function when n =100.

But, for an input n = 5 x 10^8, the compiler gives an error as follows,

The compiler displays an out of memory error as the memory needed to create an array of size 5 x 10^8 is much larger than the total memory available to our entire program. Therefore, algorithm 2 cannot process the output when $= 5 \times 10^8$.

If you compare algorithm 2 with algorithm 1,

Algorithm 1 – T(n) ∈ O(2) – Exponential time

S(n) ∈ O(n) – Linear space

Algorithm 2 – T(n) ∈ O(n) – Linear time

S(n) ∈ O(n) – Linear space

With respect to the execution time, algorithm 2 with O(n) is better than algorithm 1 with (2) and so is able to process the Fibonacci number for n = 100 in no time.

Whereas algorithm 2 must be improved with respect to the memory space required. Otherwise it can't store and process Fibonacci numbers for large input values such as $= 5 \times 10^8$.

Algorithm 3

To overcome the memory space constraint in algorithm 2, you have learnt algorithm 3, a clever technique that calculates the Fibonacci sequence by using three different variables a, b & c.

Algorithm 3

```java
public int fibonacci(int n){
    int a = 0, b = 1, c = n;
    for (int i = 2;i <= n;i++){
        c = (a + b)%10;
        a = b;
        b = c;
    }
    return c;
}
```

The variables are initialized as
a = 0, b = 1, c = n

and then for each iteration of for loop(i), variables are assigned as follows
c = (a + b)%10;
a = b;
b = c;

The values are listed below for first two iterations and for n^{th} iteration, variable c stores n^{th} number of the function $F(n) = [F(n - 1) + F(n - 2)]\%10$

i / Variable	2	3	....	n
a	1	1		F(n-2)
b	1	2		F(n-1)
c	1	2		F(n)

The java code of algorithm 3 is executed for different values of n and in specific when n = 10^9, output is processed and displayed as follows,

So, with this we are able to confirm that algorithm 3 overcomes the memory space constraint of algorithm 2. More specifically, unlike algorithm 2, the memory required for algorithm 3 is constant and independent of the input n. On analysing algorithm 3 with respect to time taken and memory space required,

The instruction set which help in generating n^{th} number of the function is executed n – 1 times, So, the time complexity of algorithm 3, T(n) = n – 1 ⇒ T(n) ∈ O(n) – linear time

Memory space required is constant, as only three variables are used to process the output irrespective of the input size(n), so the space complexity of algorithm 3, S(n) ∈ O(1) – constant space

Algorithm 1 – ( ) ∈ (2) – Exponential time
( ) ∈ ( ) – Linear space
Algorithm 2 – ( ) ∈ ( ) – Linear time
( ) ∈ ( ) – Linear space
Algorithm 3 – ( ) ∈ ( ) – Linear time
( ) ∈ (1) – Constant space

The runtime and space complexity of algorithm 3 is O(n) and O(1) respectively, which is much more efficient than the other two algorithms. Therefore, algorithm 3 can process the nth number in the Fibonacci sequence for large values of n = 10^9, a value that would otherwise be too large for algorithm 1 and algorithm 2.