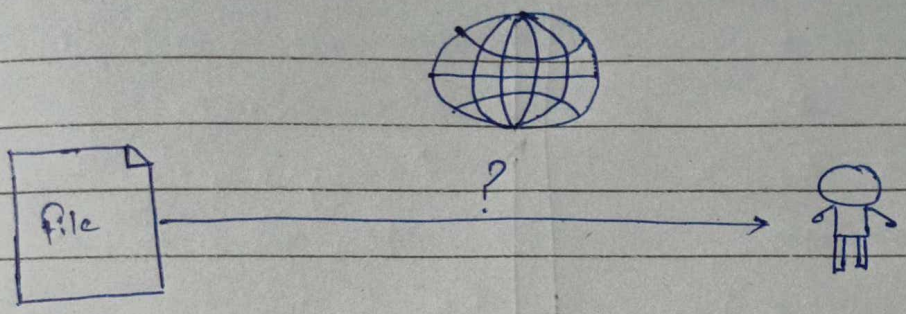
(Big O Notation)

Big O is the language and metric we use to describe the efficiency of algorithm.

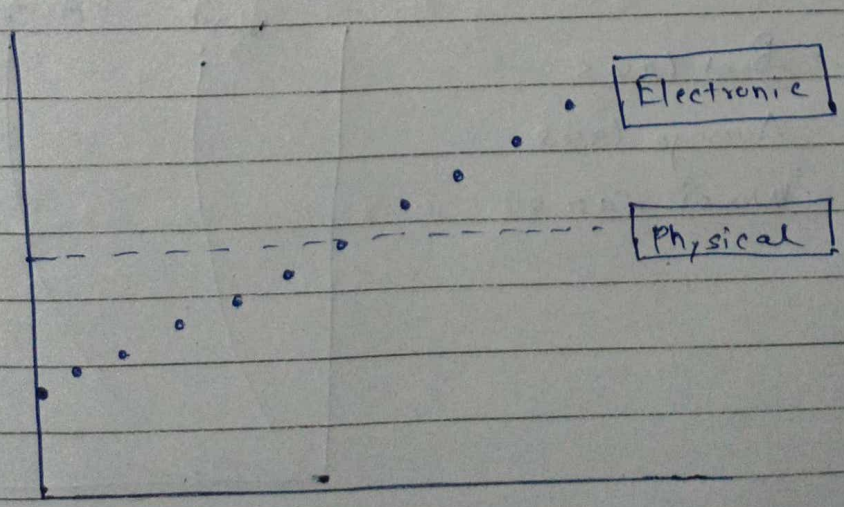


1 MB file

If we want to send 1 MB file to friend then the delivery is possible by means of mail or by the electronic communication.

But 1 TB file will take around 1 Day so instead of electronic way we can use Aeroplane.

Time Complexity - A way of showing how the runtime of a function increases as the size of input increases.

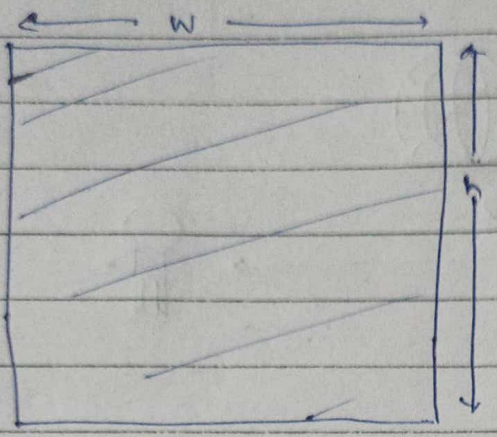


as input size increase Time increases in E-medium

But in Physical no matter how much file size is. It will always linear beos it depends on the ecroplane speed as an example.

Types of Runtimes:

$O(N)$, $O(N^2)$, $O(2^N)$



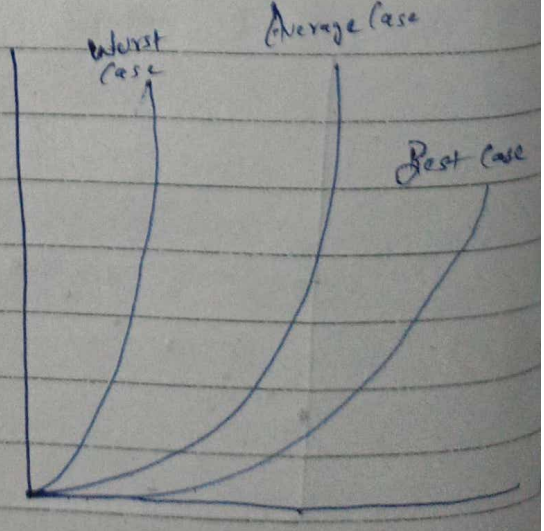
To paint on wall Time Complexity : $O(wh)$

Let Suppose a car takes petrol in 3 different conditions

→ City traffic - 20 litres
→ Highway — 10 litres
→ mixed condition - 15 litres

So by this real life example, we can consider there will be around 3 cases

- Best cases
- Average cases
- Worst cases



execution

- May be algorithm takes 1 sec in ~~run~~ under best case condition
- May be in worst case it will take 30-35 sec
- May be in Average case it will take 6 sec.

Big O: It is a complexity that is going to be less or equal to the worst case.

Big - Ω (Big-Omega) → It is a complexity that is going to be at least more than the best case.

Big Theta (Big - Θ): It is a complexity that is within bounds of the worst and the best cases.

5, 4, 10 ... , 8, 11, 68, 87, 12, ... 90, 13, 77, 55, 9

To solve this let the
Big O - O(N)
Big Ω - Ω(1)
Big Θ - Θ(n/2)

So in interview we have to only think out about the worst case.
Means always look the Big O

ANKIT
Page No.
Date

(Most Common Time Complexities)

There are many least of time complexities but we will talk about some major and important only.

Complexity	Name	Sample
$O(1)$	Constant	Accessing a specific element in array
$O(N)$	Linear	Loop through array elements
$O(\log N)$	Logarithmic	Find an element in sorted array
$O(N^2)$	Quadratic	Looking at every index in the array twice
$O(2^N)$	Exponential	Doing recursion in fibonacci

$O(1)$ – Constant time

```
int[] array = {1, 2, 3, 4, 5}
```

```
array[0] // it takes constant time to access first element
```

Here we are just accessing any 1 of the element from an array.

Like choose any card from playing card set.

O(N) - Linear time

```
int [] custArray = {1, 2, 3, 4, 5};
for (int i = 0; i<custArray.length ; i++) {
    System.out.println(custArray[i]);
}
```

// linear time since it is visiting every element of array.

Like in playing card set we have to find Jocker so we have to search all cards until jocker found.

O(Log N) - Logarithmic time

```
int [] custArray = {1, 2, 3, 4, 5}
for ( int i = 0; i<custArray.length ; i+3) {
    SOP (custArray[i]);
}
```

// logarithmic time since it is visiting only some elements.

In this example we are not doing i++ but we are doing only i+3 elements.

मतलब 13, 13, 13, 13 card मिया पहले दो में diamond है तो यहां But नीचे के 2 में diamond है। means हमें आपने work को already sort कर लिया है।

ANKIT
Page No.
Date

Binary Search

search 9 within [1, 5, 8, 9, 11, 13, 15, 19, 21]
compare 9 to 11 → smaller
search 9 within [1, 5, 8, 9]
compare 9 to 8 → bigger
search 9 within [9]
compare 9 to 9
return

N = 16
N = 8 /* divide by 2 */
N = 4 //
N = 2 //
N = 1 //

$2^k = N \rightarrow \log_2 N = k$

$O(N^2)$ → Quadratic time

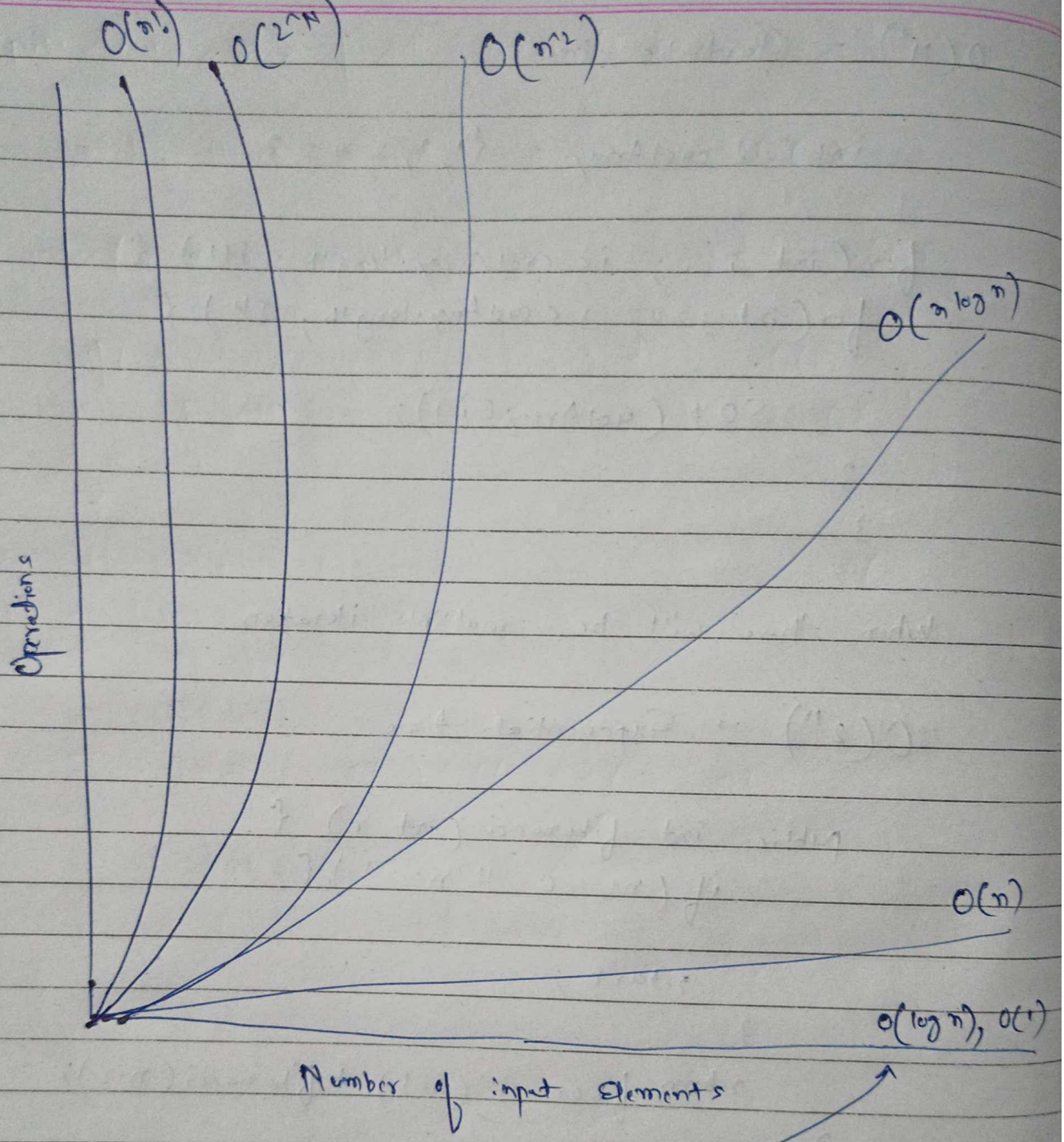
```
int [] custArray = {1, 2, 3, 4, 5}

for (int i = 0; i < custArray.length; i++) {
    for (int j = 0; j < custArray.length; j++) {
        SOP (custArray[i]);
    }
}
```

When there will be multiple iteration

$O(2^N)$ → Exponential time

```
public int fibonacci (int n) {
    if (n == 0 || n == 1) {
        return n;
    }
    return fibonacci (n-1) + fibonacci (n-2);
}
```



Below O(n) best case so O(log n), O(1) are the best time complexities

O(n) → Average case

O(n log n) → Worst case

Rest are Horrible time complexities

Means as the input increase time complexity goes high and high.

How to measure the codes using Big O?

No	Description	Complexity
Rule 1	Any assignment statements and if statements that are executed once regardless of the size of the problem	$O(1)$
Rule 2	A simple for loop from 0 to n (with no internal loops)	$O(n)$
Rule 3	A nested loop of the same type takes quadratic time complexity	$O(n^2)$
Rule 4	A loop, in which the controlling parameter is divided by two at each step	$O(\log n)$
Rule 5	When dealing with multiple statements, just add them up	

```
public static void findBiggestNumber([] sampleArray) {

    var biggestNumber = sampleArray[0]  ------> O(1)

    for(index = 1; sampleArray.length; index++) {  ---> O(n)
                                                    } -> O(n)
        if(sampleArray[index] > biggestNumber) {  ---> O(1)
            biggestNumber = sampleArray[index];  ----> O(1)
                                                    } -> O(1)
    }

    System.out.println(biggestNumber);  ------------> O(1)

}
```

Time complexity : $O(1) + O(n) + O(1) = O(n)$

By removing Non-dominating terms

(Interview Question 1)

Q:- Create a f^n which calculates the sum and products of element of array. Also find the time complexity for created method.

```
Class Main {
public static void main(String[] args){
Main main = new Main();
int[] customArray = {1,3,4,5};
main.spofArray(customArray);
}

void spofArray(int[] array) {
 int sum = 0;                                   ——— O(1)
 int product = 1;                               ——— O(1)
for(int i=0; i<array.length; i++) {             ——— O(N)
   sum += array[i];                             ——— O(1)   assignment
}
for(int i=0; i<array.length; i++) {             ——— O(N)
  product *= array[i];                          ——— O(1)
}
System.out.println(sum+", "+product);           ——— O(1)   printing on console
}
```

Eleminate Non-dominant terms

Time Complexity: O(N)

Output

13, 60

↓ sum, ↓ product

Q: 2 Create a function which prints to the console the pairs from given array. Also find the complexity for created method.

Explanation → [1, 3, 4, 5] → 11, 13, 14, 15
31, 33, 34, 35
41, 43, 44, 45
51, 53, 54, 55

```
Class Main {
public Static void main(String [] args) {
Main main = new Main();
int [] customArray = {1, 3, 4, 5};
main.printPairs(customArray);
}

void printPairs(int [] array) {
  for(int i=0; i< array.length; i++) {        → O(N) } O(N^2)
    for(int j=0; j< array.length; j++) {      → O(N)
      Sop(array[i]+", "+array[j]);            → O(1)
  }
  }
  }
  }
```

After eliminating Non dominating terms

TC: → O(N^2)

Q:3 → What is the time complexity for this method?

```
Void printUnorderedPairs( int [ ] array ) {
    for( int i=0; i<array.length; i++) {
        for( int J = i+1; J<array.length; J++) {
            System.out.println( array[i]+ ", " + array[J]);
        }
    }
}
```

Time Complexity : $O(N^2)$

Q:4

```
void printUnorderedPairs (int[] arrayA, int[] arrayB) {
    for (int i = 0; i<arrayA.length ; i++) {
        for( int J=0; J<arrayB.length ; J++) {
            if( arrayA[i] < arrayB[J]) {
                SOP( arrayA[i]+ ", "+arrayB[J]);
            }
        }
    }
}
```

Ans:- Be careful Here is 2 nested for loops but it will not $O(N^2)$ time complexity.

Here 2 array is present and we don't know how many elements are there in both array So $O(ab)$

a = arrayA.length

b = arrayB.length

So Time complexity: $O(ab)$

ANKIT
Page No.
Date

Q:5 → What is the runtime of the below code?

```
Void printUnorderedPairs (int[] arrayA, int[] arrayB) {
    for(int i=0; i<arrayA.length; i++) {                    } → O(ab)
        for(int j=0; j<arrayB.length; j++) {  ——→ B
            for(int k=0; k<1000000; k++) {  ——→ ?

                SOP(arrayA[i] + "," + arrayB[j]); ——→ O(1)
            }
        }
    }
}
```

here 2 array is there so

b = arrayB.length;

a = arrayA.length;

But in 3rd loop k will reach till 1000000 so it is a constant so time complexity will be O(1)

So overall time complexity after removing non dominating term

O(ab)

Q:6 → Create a method which takes an array as a parameter and reverse it.
Find the runtime of the created method?

Explanation :- Input: 1, 2, 3, 4, 5 → Output: 5, 4, 3, 2, 1

```
import java.util.Arrays;
  class Main {
  public static void main(String[] args) {
  Main main = new Main();
  int[] customArray = {1, 3, 4, 5};
  main.reverse(customArray);
  }
  void reverse(int[] array) {
  for(int i=0; i<array.length/2; i++) { → O(N/2) → O(N)
  int other = array.length - i - 1; → O(1)
  int temp = array[i]; → O(1)
  array[i] = array[other]; → O(1)
  array[other] = temp; → O(1)
  }
  System.out.println(Arrays.toString(array)); → O(1)
  }
  }
```

O(N/2) में constant remove और So O(N)

Time Complexity : O(N)

Q: 7 Which of the following are equivalent to O(N)? Why?

1. $O(N+P)$, where $P < N/2$
2. $O(2N)$
3. $O(N + \log N)$
4. $O(N + N \log N)$
5. $O(N+M)$

(1) $P < N/2$ means P is dominant term
So we can remove P
So overall $O(N)$ ✓

(2) $O(2N) \rightarrow O(N)$

(3) $O(N + \log N) \rightarrow O(N)$

(4) $O(N + N \log N) \rightarrow O(N \log N)$ ✗

(5) $O(N+M) \rightarrow$

Don't remove M beacs we don't know N is greater or M is greater so keep both.

Time Complexity → $O(N+M)$

9:8 Time Complexity of Factorial

```
Static int factorial (int n) {          → T(n)
    if (n<0) {
        return -1;
    }                                   → O(1)
    else if (n==0) {
    return 1;
    }
    else {
        return n * factorial (n-1);     → T(n-1)
    }
}
```

$T(n) = O(1) + T(n-1)$

$T(1) = O(1)$

$T(n-1) = O(1) + T((n-1)-1)$

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

$T(n) = 1 + T(n-1)$

$= 1 + (1 + T((n-1)-1))$

$= 2 + T(n-2)$

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

$= 3 + T(n-3)$

$\vdots$

$q + T(n-q)$

$= n + T(n-n)$

$= n+1$

$= n$

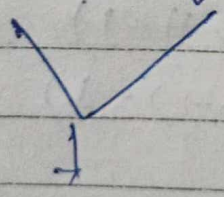
Time Complexity

$O(N)$

Q: Time Complexity of fibonacci

```
void allfib(int n) {
    for(i=0; i<n; i++) {
        fib(i);
    }
}

static int fib(int n) {
    if(n<=0) {
        return 0;
    }
    else if(n==1) {
        return 1;
    }
    else {
        return fib(n-1) + fib(n-2);
    }
}
```



Here fn call its method twice time

So

branches^depth ----> O(2^N)

depth depends on parameter

branches = 2

fib(1) → 2^1 steps

fib(2) → 2^2 steps

fib(3) → 2^3 steps

fib(4) → 2^4 steps

...

fib(n) → 2^n steps

→ Total Work = $2^1 + 2^2 + 2^3 + 2^4 + \ldots + 2^n$

$= 2^{n+1} - 2$

Time Complexity : $O(2^n)$

(Time Complexity of Power of 2)

```
static int powerof2(int n){
    if(n<1){
      return 0;
    }else if(n==1){
      sop(1);
      return 1;
    }else{
      var prev = powerof2(n/2);
      var curr = prev*2;
      sop(curr);
      return curr;
    }
}
```

→ O(1)

→ O(1)