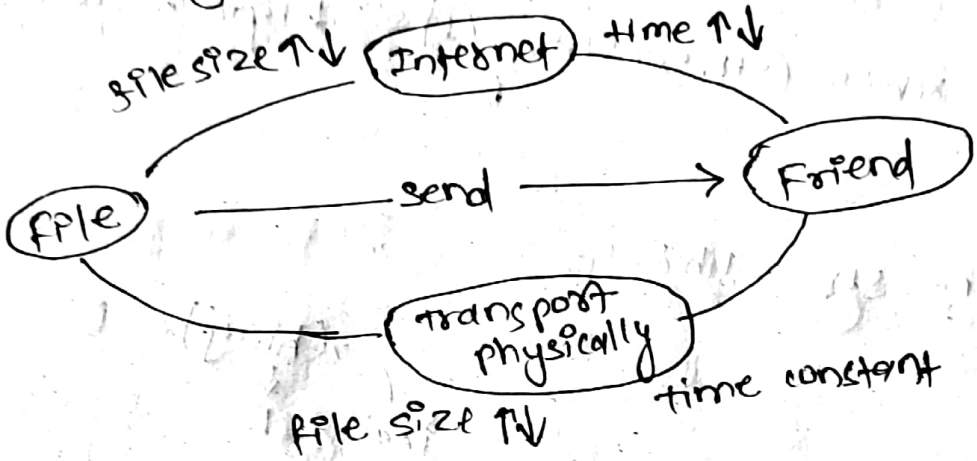
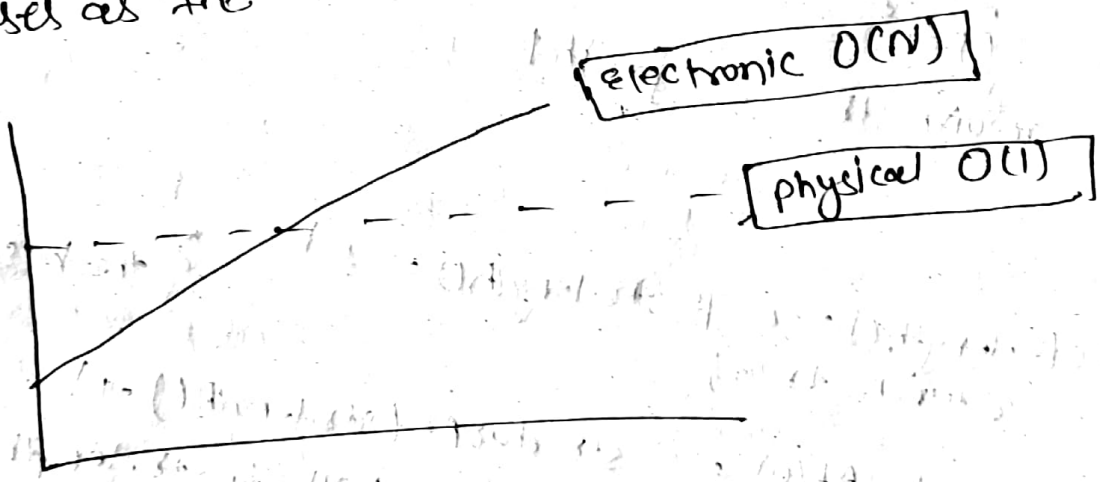
Big O notation

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

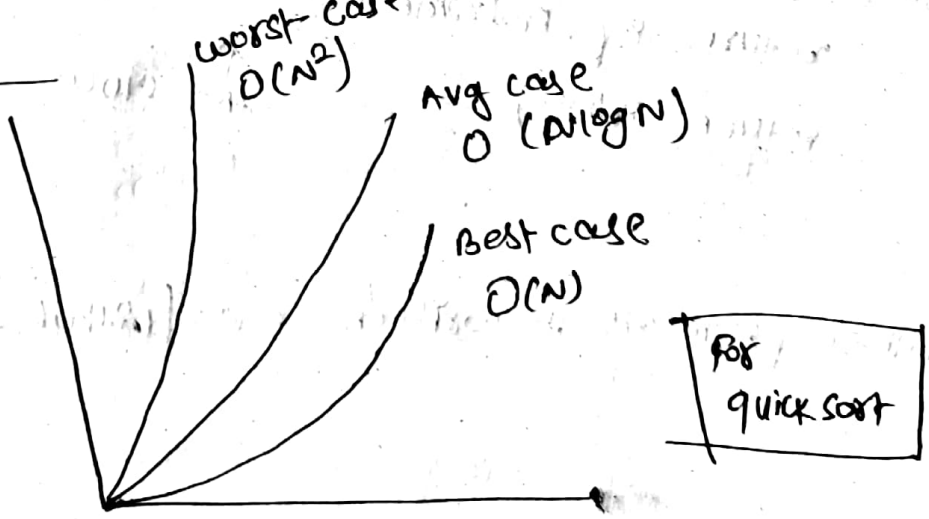


time complexity

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



Different Scenerios



Types

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

Big O - O(N) {max time}

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

Big Ω - Ω(1) {least time}

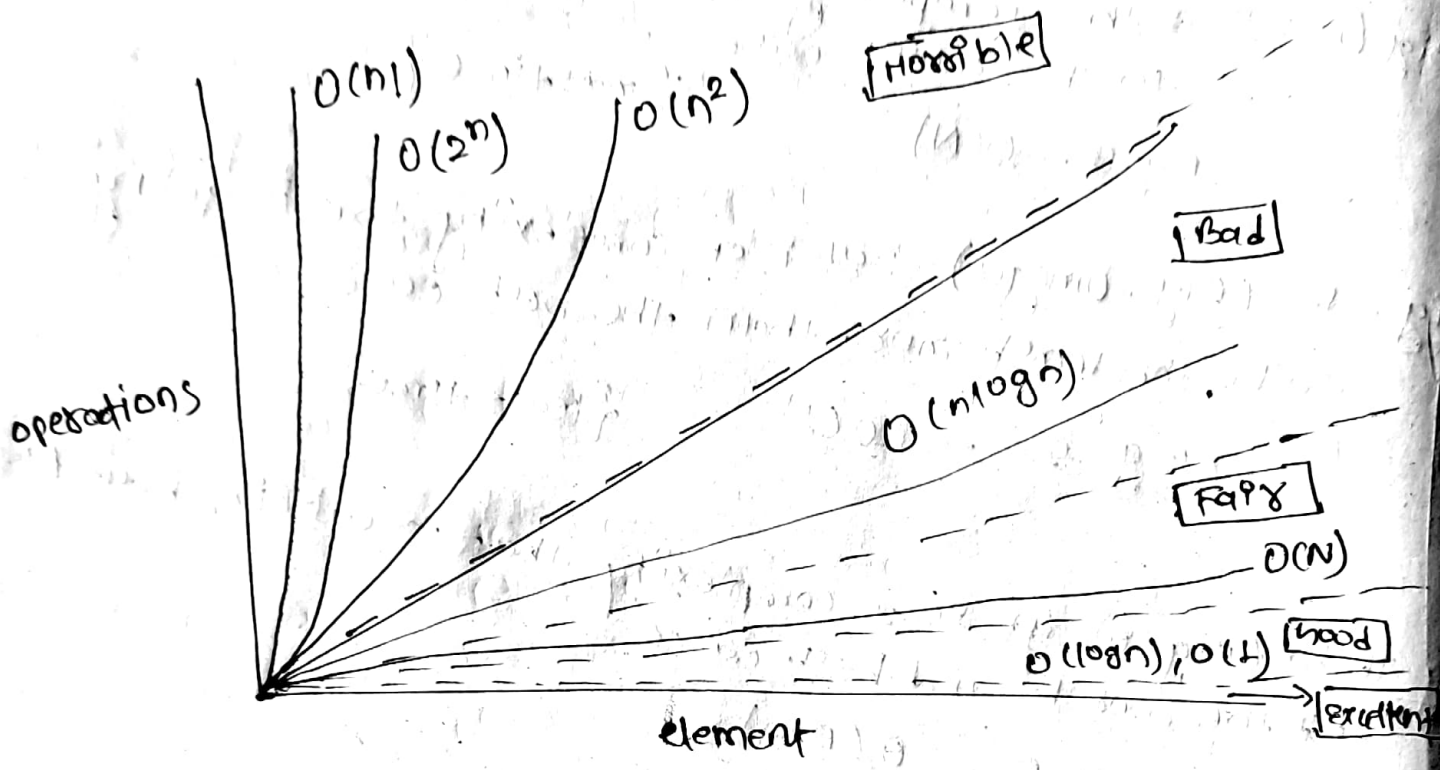
Big Theta (Big θ) It is a complexity that is within bounds of the worst and best cases.

Big θ - θ(n/2)

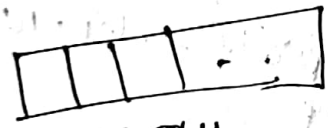
RunTime complexities

complexity	Name	sample
O(1)	constant ~~Linear~~	Accessing a specific element in array
O(N)	Linear	loop through array elements
O(logN)	logarithmic	find an element in sorted array
O(N²)	Quadratic	looking every index in the array twice
O(2^N)	exponential	Double recursion in Fibbonacci

Big-O complexity chart



#Space Complexity



linear array
SC: O(N)

2-D array
SC: O(N^2)

```
Static int sum(int n){
    if(n<=0)
        return 0;
    return n+sum(n-1);
}
```

1. sum(3)
2. sum(2)
3. sum(1)
4. sum(0)

space complexity : O(N).

#1 Drop constants and Non-Dominant Terms

Drop constant

$O(2N) \longrightarrow O(N)$

Non-Dominant terms

$O(N^2+N) \longrightarrow O(N^2)$

$O(N+\log N) \longrightarrow O(N)$

$O(2\times 2^N + 1000\, N^{100}) \rightarrow O(2\times 2^N) \rightarrow O(2^N)$

Add vs multiply

- If algorithm is in the form, 'do this, then when you are all done, do that', then you add the run times

ex

```
for (a=0; arrayA.length; a++) {
    System.out.println(arrayA[a]); }
for (b=0; arrayB.length; b++) {
    System.out.println(arrayB[b]); }
```

⇒ Add the run times : $O(A+B)$

- If algorithm is in the form, 'do this for each time you do that', then you multiply the run times.

ex

```
for(a=0; arrayA.length; a++){
  for(b=0; arrayB.length ;b++){
  System.out.println (arrayA[a] + arrayB[b]);}}
```

→ multiply the Runtimes : O(A*B)

measuring the Big O time complexities

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	O(N)
Rule 3	A nested loop of same type	O(N^2)
Rule 4	A loop, in which the controlling parameter is divided by two at each step	O(logN)
Rule 5	when dealing with multiple statements just add up	

Example

```
public static int biggestelement (int[] a){  -------> O(1)
   int big = a[0];                           -------> O(n)
 for(int i=1; i<a.length; i++){
      if (big < a[i])          -->O(1) }-->O(1) }-->O(1)
         big = a[i];           -->O(1)  }
   }                           -->O(1)
 sout ("big :"+big);
}
```

Total time complexity
$= O(1) + O(n) + O(1)$
$= O(n)$ A

Measure of Recursive Algorithm

Sample Array : | 5 | 4 | 10 | ... | 8 | 11 | ... | 6 | 68 | 87 | 10 |

```
static int Bigele ([]a, int n){        ——> m(n)
    if(n==1)                           ——> O(1)
        return a[0];                   ——> O(1)
    return max (a[n-1], Bigele (a, n-1)); ——> m(n-1)+O(1)
```

Explanation

A = | 11 | 4 | 12 | 7 |

Bigele (A,4) ——> max [A[4-1], 12)] → max(7, 12) ⇒ 12
↓ ↑
Bigele (A,3) ——> max(A[3-1], 11) → max (12, 11) → 12
↓ ↑
Bigele (A,2) ——> max (A[2-1], 11) → Max (A[1], 11) → max(4, 11) → 11
↓ ↑
Bigele (A,1) ——> a[0] = 11

Now,

$M(n) = O(1) + m(n-1)$

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

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

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

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

$m(1) = O(1)$

⇒

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

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

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

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

$= 3 + m(n-3)$

$= a + m(n-a)$

$= n-1 + m(n-(n-1))$ {a = n-1}

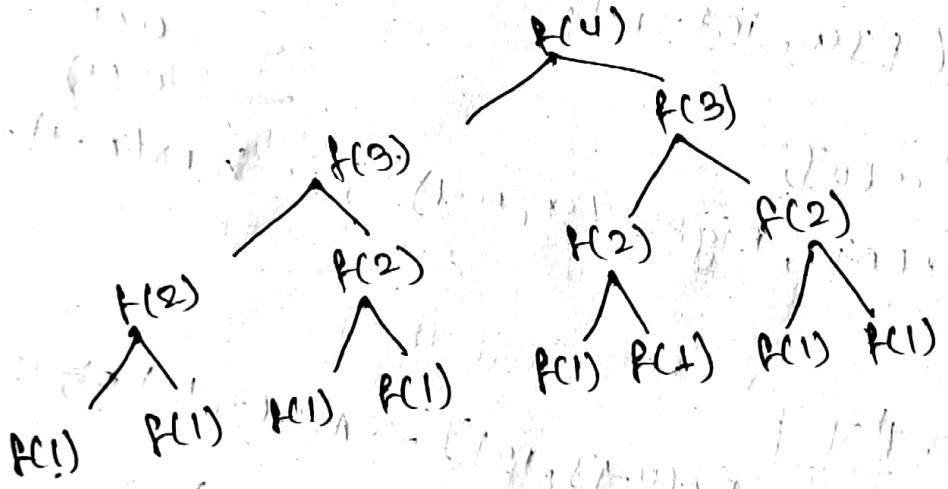
$= n-1 + m(1)$

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

measuring Recursive Algorithm with multiple calls

```
public int f(int n) {
    if (n <= 1)
        return 1;
    return f(n-1) + f(n-2);
}
```

branches



level	node
0	$= 2^0$
1	$= 2^1$
2	$= 2^2$
3	$= 2^3$
⋮	⋮
n	$= 2^n$

n ↑ depth, 2 ↑ branches

Now,

$$2^0 + 2^1 + 2^2 + \cdots + 2^n = 2^{n+1} - 1$$
$$= 2^n - 1 = 2^n \longrightarrow O(2^n).$$

In general.

Time complexity : $O(\text{branches}^{\text{depth}})$.

Q.3 Find the time complexity?

```
void printUnorderedPairs(int[] array) {
  for(int i=0; i<array.length; i++){
    for(int j=i+1; j<array.length; j++) {
      sout("a[i]+", "+a[j]);
    }
  }
}
```

1. counting iterations

1st → n-1
2nd → n-2
⋮
1

⇒ 1+2+.. +(n-2)+(n-1)

$= \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$

$= O(n^2)$.

2. Average work:
Outer loop - O(N)
inner loop !

step 1 → 10
step 2 → 9
step 3 → 8
⋮
1

} Avg = 5 times. = 10/2
n → n/2

So, outer. inner ⇒ $n * n/2 \Rightarrow \frac{n^2}{2}$

$= O(N^2)$.

Q.5 O(mN)

Q.4 O(m.N)

m = arrayA.length
N = arrayB.length

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+logN)
4. O(N+NlogN)
5. O(N+m) ✓

Q.8

Time complex of factorial code used in recursion.

$O(N)$. method → Q.3

Q.9 Fiboracci

$O(2^N) \Rightarrow$ branches (depth → N)

↓

given (2).

Q.10 power of 2

$O\log(N)$