Time complexity of a computer program-

In an interview,

let's say you are able to come up with a solution.

You will be asked the following questions-

1. Time complexity of solution?
2. Can you improve the time complexity of the solution?

Given that in any interview (or) in real life situations, you are gonna face this question many times, it looks like something important

How to calculate running time?

depends upon

- processors ✗
- read and write speed ✗
- architecture ✗
- configuration of the machine ✗
- Input ✓

We need to evaluate time as a function of input.

Model machine-

Assuming that there is a common machine that all of us have and then we are trying to run program on it.

- Single processor
- 32 bit
- Sequential execution
- 1 unit of time for arithmetic and logical operations
- 1 unit of time for assignment and return statements

function that calculates the sum of two numbers

```
sum(num1, num2) {
    return num1 + num2;
}
```

1 + 1 = 2

It will always take 2 units of times.

Constant time...

function that calculates the sum of all elements in list

```
sum ( list ) {
    total = 0 → 1 unit of time
    for ( item in list ) { → 2 units of time   (n+1)
        total += item; → 2 units of time       (n)
    }
    return total; → 1 unit of time
}
```

Time taken = $1 + 2(n+1) + 2(n) + 1$

$= 1 + 2n + 2 + 2n + 1$

$T(n) = 4n + 4 \Rightarrow a_1 n + a_2$

Linear Function

Square matrix - calculate sum of all elements

```
int sum( int[][] arr) {
1   <-  int total = 0;
(n+1)  2  <-  for (int i=0; i< arr.length; i++){
    (n+1)  2  <-  for (int j=0; j< arr[0].length;
                      j++) {
        (n)  2  <-  total += arr[i][j];
                3
            3
    1  <-  return total;
}
```

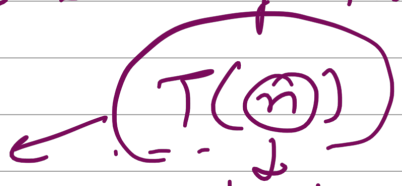
Total time = $1 + (2n)(n) + 2 + 2 + 1$

$= 1 + 2n^2 + 5$

$= 2n^2 + 6$

$T(n) = a_1 n^2 + a_2$ <Quadratic function>

We are trying to see how can we calculate the running time as a function of input.



running time

input

We will map our running time into categories-

$T_1(n) = 6n^2 + 3n + 4$

$T_2(n) = 3n^2 + 2$

Let's say when $n = 100$;

$T_1(100) = 6 \cdot (100)^2 + 3 \cdot 100 + 4$

$= 60000 + 300 + 4$

$= 60304$

$T_2(100) = 3n^2 + 4 = 3 \cdot (100)^2 + 4$

$= 30004$

$n = 10000$,

$T_1(10000) = 6(10000)^2 + 3 \cdot 10000 + 4$

$= 600000000 + 30000 + 4$

$= 600030004$

$T_2(10000) = 3(10000)^2 + 2$

$= 300000002$

$T_1(10,00,000) = 6n^2 + 3n + 4$

$= 2600000000000000 + 3.1000000 + 4$

$= 6000003000004$

$T_2(10,00,000) = 3n^2 + 2$

$= 300000600000002$

- As we can see that effect of lower degree terms is getting lesser and lesser in a way that we can ignore the lower terms as when $n \to \infty$

When $n \to \infty$,

$T_1(n) \simeq T_2(n)$

So, we just got the idea that there are many functions which will behave in similar way when $n \to \infty$

So, why not keep all those functions in the same category...

O Notation-

(Big O notation, Asymptotic notation)

$O(g(n)) = \{ f(n) :$ there exists some constant c and n_0 such that

$f(n) \leq c.g(n)$ for

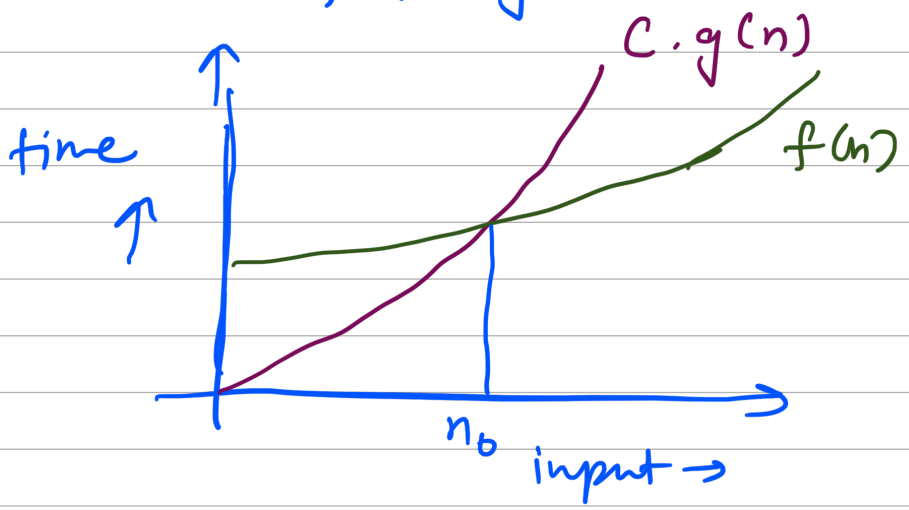
$n \geq n_0$

$f(n) = 5n^2 + 2n + 1$

$g(n) = n^2$

$[n_0 = 1, c = 8]$

$f(n) \leq c \cdot g(n)$



O → upper bound of rate of growth of time

Omega notation (Ω)

$$\Omega(g(n)) = \{ f(n) : f(n) \geq c \cdot g(n)$$

for some constant

c and n_0

$n \geq n_0$

Ω → Lower bound of rate of growth of time

$f(n) = 5n^2 + 2n + 1$

$g(n) = n^2$

$[c \geq 5;\ n_0 \geq 0]$

$f(n) \geq 5n^2$

Θ notation –

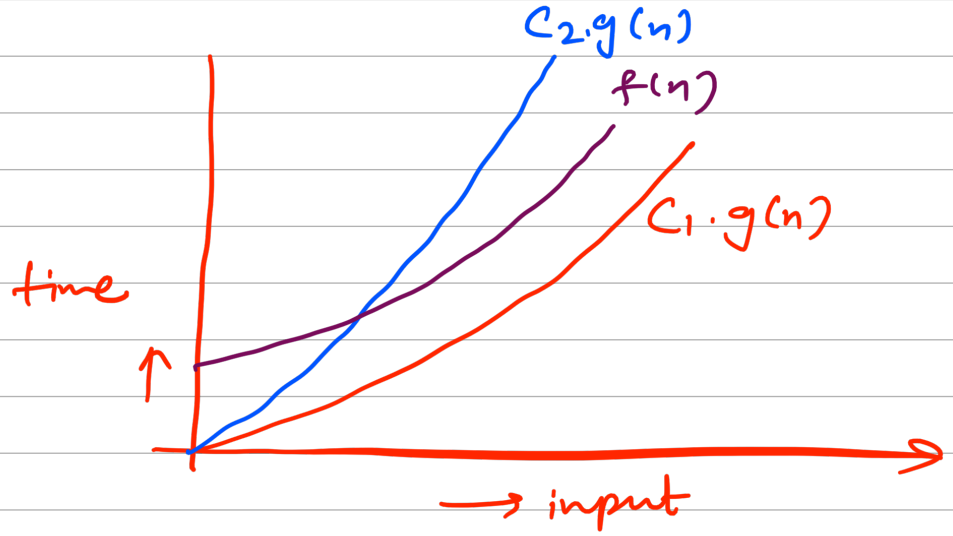
$$\Theta(g(n)) = \{ f(n) : c_1, c_2$$
$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

$f(n) = 5n^2 + 2n + 1$

$g(n) = n^2$

$[c_1 = 5,\ c_2 = 8,\ n_0 = 1]$

$c_1 g(n) \leq f(n) \leq c_2 \cdot g(n)$



Θ notation → tight bound...

– Always try to come up with a Θ notation for an algorithm

Some general rules –

Time complexity analysis –

→ very large input size
→ worst case scenario

$T(n) = n^3 + 3n^2 + 4n + 2$

for very large input size,

$\simeq n^3 \quad (n \to \infty)$

$O(n^3)$

Rules –

- Drop the lower order terms
- Drop the constant multipliers

$T(n) = 17n^4 + 3n^3 + 4n + 8 = O(n^4)$

Rule –

Running time $\Rightarrow \sum$ Running time of all fragments

```
int a;
a=5;
a++;
```
} $O(1)$ Simple statements

```
for (int i=0;  i< n; i++) {
        // some work
}
```

O(n)

```
for (int i=0;  i< n;  i++) {
        for (int j=0;  j< n; j++) {
            // some work;
        }
}
```

$O(n^2)$

```
fun() {
```

O(1)
```
{ int a;
  a=5;
  a++;
```

O(n)
```
{ for (i=0; i<n ; i++) {
        // some work;
  }
```

O(n²)
```
{ for(i=0; i<n; i++) {
        for (j=0; j<n; j++) {
            // some work
        }
  }
```

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

```
fn() {
    if (some_condition) {
        for (i=0; i<n; i++) {
            // some work
        }
    } else {
        for (int i=0; i<n; i++) {
            for (int j=0; j<n; j++) {
                // simple work
            }
        }
    }
}
```

$O(n)$ — the `for` loop in the `if` branch

$O(n^2)$ — the nested `for` loops in the `else` branch

Worse case: $O(n^2)$

$T(n) = O(n^2)$

Rule-
→ Pick complexity of condition which is worse case

Space Complexity-

Measure of how efficient your code <function> is in terms of memory used...

Space Complexity analysis happens in the same way time complexity analysis happens

```
fn(n) {
    int[] arr = new int[n];
    for(int i=0; i<n; i+=1) {
        arr[i] = i;
    }
}
```

This code ends up creating an array of size n. So the space complexity is $O(N)$.

Additional space / memory is measured in terms of largest memory used by the program when it runs.

If your program creates an matrix of $O(n^2)$ and later deletes it, the space complexity would be $O(n^2)$ and not $O(1)$.

Relevance of time complexity

- Real life examples
- Prime number...

A → Linear algorithm
B → $\sqrt{n}$ ⟶ Chances of B getting thread is more

Examples

①
```
int a=0, b=0;
for(i=0; i<n; i++){
        //...
}
for(j=0; j<m; j+=1){
        //...
}
```

Time → O(N+M)
Space → O(1)

②
```
int a=0, b=0;
for(i=0; i<N; j+=1){
        for(j=0; j<N; j+=1){
                a=a+j;
        }
}
```

```
for (k=0; k<N; k+=1) {
        b=b+1;
}
```

Time = O(N2)
Space = O(1)

③
```
int a=0;
for (i=0; i<N; i+=1) {
        for (j=N; j>i; j--) {
                // ...
        }
}
```

T = O(N²)
S = O(1)

④ When we say an algorithm X is asymptotically better than Y?

X will be a better choice for _____ inputs

- large

⑤

```
int a = 0, i = N;
while (i > 0) {
    a += i;
    i /= 2;
}
```

T = O(log N)

⑥

```
int count = 0;
for (i = N; i > 0; i /= 2) {
    for (j = 0; j < i; j++) {
        //...
    }
}
```

T = O(N)

$N + N/2 + N/4 + \cdots$

$N\left[(1)\left(\frac{1-r^n}{1-r}\right)\right] \rightarrow N\left[(1)\frac{(1-(1/2)^{\infty})}{(1-1/2)}\right]$

$= 2N$

(7) → for (int i = n/2; i <= n; i++) {

for (j = 2; j <= n ; j++) {

//

}

}

T = O(n . log (n))

(8) → for (i = 0; i < n; i++)

for (i = 0; i < n; i += 2)

for (i = 1; i < n; i *= 2)

for (i = n; i > -1; i /= 2)

(C)

(9) → 15^ 10 * n + 12099

n^ 1.98

$n^3/\sqrt{n}$

(2^20) * n

which is not bounded by O(n^2)

(C)

(10) →

2^n

n^(3/2)

n.logn

n^logn

3, 2, 4, 1

(11) f(a, k, 0, N-1)

```
int f( int[] a, int k, int s, int e){
    if(s > e) {
        return 0;
    }
    int mid= (s+e)/2;
    if( a[mid] < k ) {
        return f(a, k, mid+1
            end);
    }
    if( a[mid] > k ) {
        return f(a, k, start,
            mid-1);
    }
    return f(a, k, start, mid-1) +
        1 + f(a, k, mid+1, end);
}
```

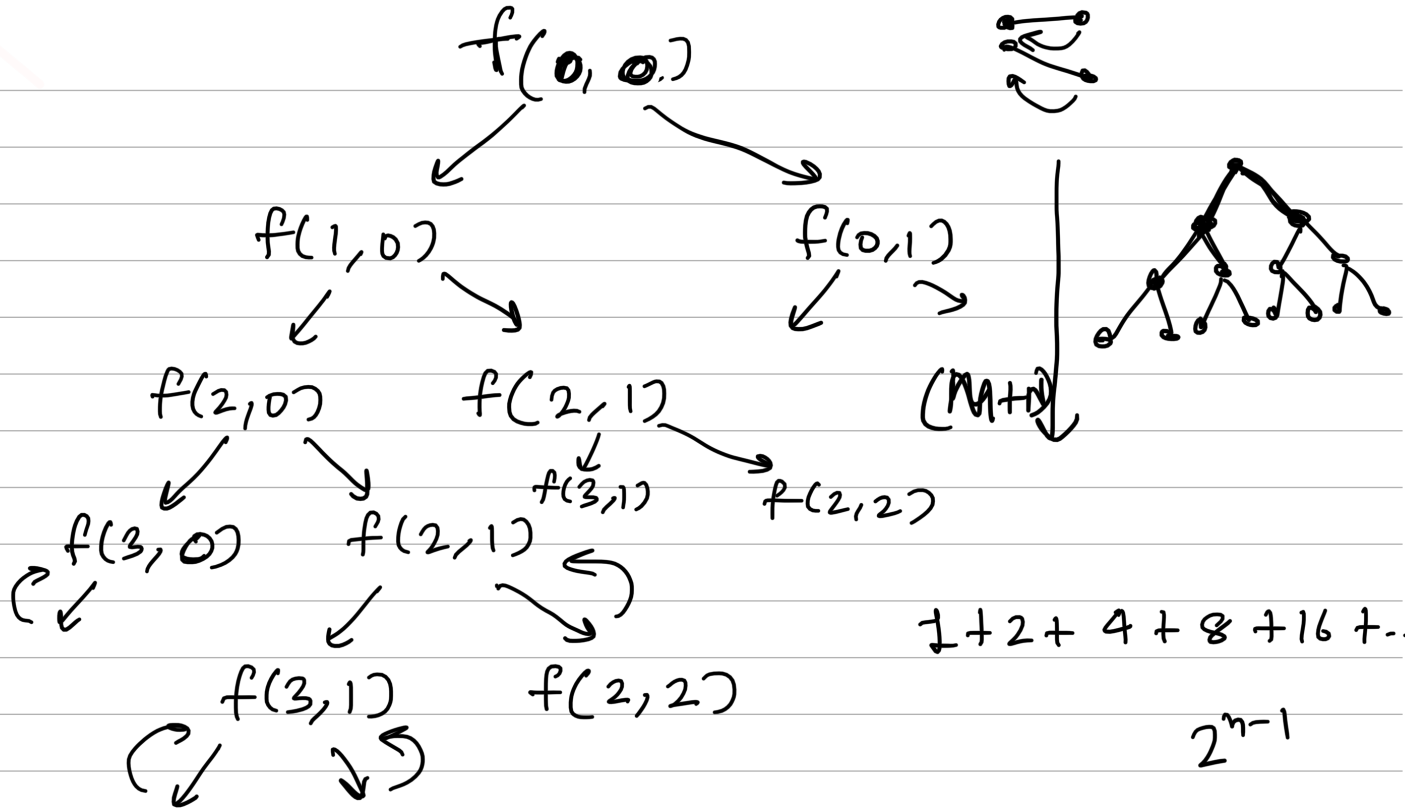
$T(n) = 1 + T(n/2) + T(n/2)$

(12) →

```
int f( int[] a, int r, int c ) {
    int R = a.length,
        C = a[0].length;
    if (r >= R || c >= C ) {
        return 0;
    }
    if ( r == R-1 || c == C-1 ) {
        return 0;
    }
    return 1 + f(a, r+1, c) +
               f(a, r, c+1);
}

f(a, 0, 0);
```

$T(R, C) = O(2^{(R+C)})$



$1+2+4+8+16+\ldots$

2^{n-1}

$1\left(\frac{2^n-1}{2-1}\right) \rightarrow 2^n$

(13) Memoization ?

O(RXC)

H.W.

(14) int j=0;

for(i=0; i<N; i++){

while(j<N && arr[i]<arr[j]){

j++;

}

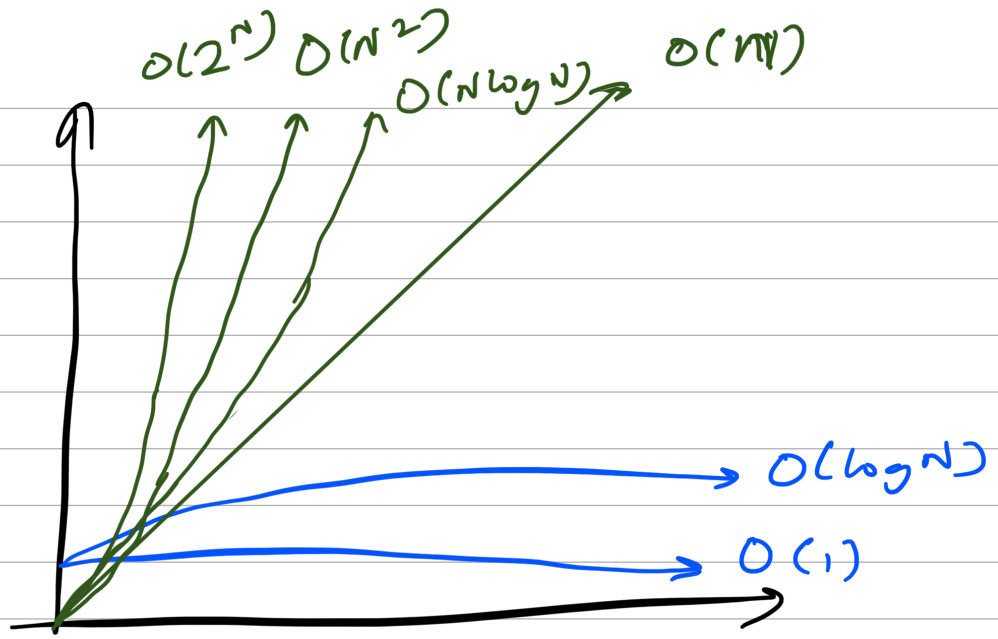
}

O(N)

[3, 6, 2, 4, 5, 1]

i i i

j


O(2^N)
O(N^2)
O(N log N)
O(N!)
O(log N)
O(1)