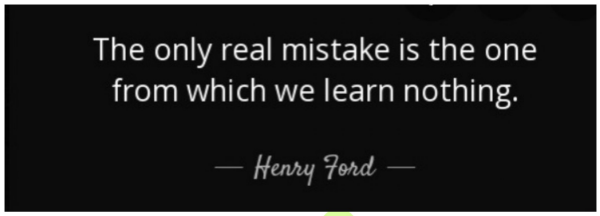
## Today Quote




{ Do mistakes → learn from them → grow. }

## Today's content.

→ Comparing two Algo's.
  a) using execution time
  b) using iterations & graphs.

→ Why Big-O needed.
  a) Why lower order terms neglected
  b) Why constant coefficients neglected
  c) Issues in Big-O
  d) Worst case.

→ Space Complexity

→ T.L.E
  a) Why TLE occurs & info about online editors.
  b) How to approach any given problem.
  c) Importance of constraints.

**Task:** Given N elements. Sort them in increasing order.

- arr = { 3 2 7 5 1 11 } ⇒ { 1 2 3 5 7 11 }
- input size [N] → $10^4$

**Algo 1** {RAM}

15 sec.

(Windows XP)

Scaler [macbook].

7 sec.

(C++)

7 sec.

[Volcano].

Antartica

5 sec.

**Algo 2** {Karunesh}

10 sec.

[latest macbook]

10 sec.

(python)

C++

5 sec.

[Antartica].

5 sec.

Execution time:

[C++ >> python]

due to temp ↑ performance ↓

**Execution time** → It depends on so many external factors → hence we can't compare performance of 2 algo's on the basis of execution time.

## iterations

```
for ( i = 1 ; i <=N ; i++) {      [iterations : N]
    print(i)
}
```

[Conclusion 1: To compare two algorithms, calculate their no. of iterations, based on input size.]

Algo 1. {Venkat}
$100 \log_2 N$

Algo-2. {Dhiraj}
$N/10$

till $N < 3500$ — Dhiraj's Algo will perform better.

$N > 3500$ — Venkat's Algo will perform better

Ind vs Aus : 1 CR

Google results : millions of results.

Baby shark : 11 Billion.

∴ Pick an algo, that will perform better for very large inputs.

⇒ In real world, data is always increasing.


Can we really draw 1000's of graph ? ⇒ No.

In order to make this comparison simpler

## Asymptotic Analysis of Algorithms

↳ Analysis of performance of algorithms for very inputs.

↳ Big-O Notation.

Algo 1. {Venkat}

$100 \log_2 N$

Algo-2. {Dhiraj}

$N/10$

Big-O. $O(\log_2 N)$ is better than $O(N)$

**Biggest Advantage of using Big-O**

We can directly tell which algorithm is better just by looking at Big-O notation.

[No need to draw graphs].

## Steps to calculate Big-O

1. Calculate no. of iterations.
2. Neglect lower order terms?
3. Neglect constant / co-efficients. ?

## Neglect lower order terms ?

Q) iterations. → $N^2 + 10N$

| i/p size. | total iterations | % of lower order terms in total iterations |
|---|---|---|
| $N = 10$ | 200 | $\frac{100}{200} \times 100\% = 50\%$ |
| $N = 100$ | $10^4 + 10^3$ | $\frac{10^3}{10^4+10^3} \times 100\% \approx 9\%$ |
| $N = 1000$ | $10^6 + 10^4$ | $\frac{10^4}{10^6+10^4} \times 100\% \approx 1\%$ |
| $N = 10^4$ | $(10^4)^2 + 10 \cdot 10^4$ $\Rightarrow 10^8 + 10^5$ | $\frac{10^5}{10^8+10^5} \times 100\% \approx 0.1\%$ |

[ Observation : As input size ↑es, contribution of lower order terms ↓es. ]

## Neglect constant co-efficients ?

Q:)

| Algo 1 (Manas) | Algo 2 (Vaibhav) | for very large i/p. |
|---|---|---|
| $10 \log_2 N$ | $N$ | Manas |
| $100 \log_2 N$ | $N$ | Manas |
| $1000 \log_2 N$ | $N/10$ | Manas |
| $10N$ | $N^2/10$ | Manas. |
| $N \log N$ | $100N$ | Vaibhav. |

{Growth rate of $N^2 >>> N$}

[* Growth rate :].

# Issues in Big-O

Q:

| | Algo 1. | Algo 2. |
|---|---|---|
| | $10^3 N$ | $N^2$ |
| Big-O. | $O(N)$ | $O(N^2)$ |

claim 1. : for every input, algo 1 will perform better : false.

| | | | |
|---|---|---|---|
| $N = 10$ | $10^4$ | $10^2$ | {Algo 2 is better} |
| $N = 100$ | $10^5$ | $10^4$ | {Algo 2 is " } |
| $N = 1000$ | $10^6$ | $10^6$ | { same } |
| $N = 10^3 + 1$ | $10^3(10^3+1)$ | $(10^3+1)(10^3+1)$ | {Algo. 1 is better} |

claim 2. : For every input $> 10^3$, algo. 1 will perform better.

Final claim: when we compare given 2-algorithms using Big-O notations, algo. 1 will perform better for all the values
> 1000.
> certain value.

⇓

Threshold value.

→ After this threshold value, Big-O holds.
→ Don't worry about threshold value.

## Issues in Big-O

Q.)

| | Algo-1 | Algo-2 | |
|---|---|---|---|
| | $2N^2 + 4N$ | $3N^2$ | {Actually, algo 1 will perform better} |
| Bigo : | $O(N^2)$ | $O(N^2)$ | {Both are same according to Big-O} |

$\boxed{2N^2} + 4N$ $\quad$ $\boxed{2N^2} + N^2$ $\quad$ growth rate of $N^2 >> N$

Q.)

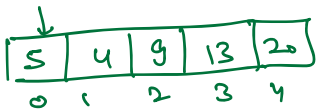
| | | | |
|---|---|---|---|
| | $5N^3 + 6N^2$ | $4N^3 + 10N$ | {Actually, algo 2 will perform better} |
| Big O : | $O(N^3)$ | $O(N^3)$ | {Both are same according to Big-O} |

Observation : If two algo's are having same Big-O notation, then we can't really compare them using Big-O notation.

{ code → iterations → Big-O }

Code: Searching for an element = k

```
boolean search (int[] arr, int k){
    int n = arr.length;
    for (int i = 0; i < n; i++){
        if (arr[i] == k) return true;
    }
    return false;
}
```




iterations.

best-case. 1

worst-case N

hire ?

Cad-1

Cad-2.

best: 7 5

worst: 8 10

Code. → Time Complexity.

Space Complexity.

int → 4 B

long → 8 B

# Space Complexity

① 4B

```
void fun (int N){
    4B : int x = N;
    4B : int y = x * x;
    8B : long z = x + y;
}
```

total memory :

4B + 4B + 4B + 8B

= 20 B

② 4B

```
void fun (int N){
    4B : int x = N;
    4B : int y = x * x;
    8B : long z = x + y;
    4N B : int[] arr = new int [N];
}
```

total memory :

(20 + 4N) B

S.C → O(N)

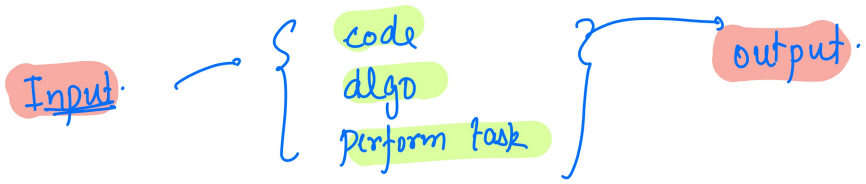
③ 4B

```
void fun (int N){
    4B : int x = N;
    4B : int y = x * x;
    8B : long z = x + y;
    4N B : int[] arr = new int [N];
    8N² B : long [][] l = new long [N][N];
}
```

total memory :

4B + 4B + 4B + 8B + 4N B + 8N² B

S.C → O(N²)




## Space Complexity Of An Algorithm :

It is amount of extra space that is needed by your algorithm, other than input and output space.

Q.) Max in an array ?

```
int maxInArray ( int[] arr) {        // i/p.
    int ans = arr[0];
    for( int i = 1 ; i < arr.length ; i++){
        ans = max( ans, arr[i])
    }
    return ans ;                     // o/p.
}
```

T.C → O(N)
S.C → O(1) → constant

D.S.A → 600 problems → T.C and S.C

Q.)

```
int task ( int[] arr, int n){
    int[] pf = new int [n]; //4N
      pf [0] = arr[0];
    for( int i = 1 ; i < n ; i++){
          pf[i] = pf[i-1] + arr[i];
    }
    // few more calculations
}
```

i/p i/p

S.C → O(N)

T.C → O(N)

[In most of the cases, first reduce the T.C]

Google → 0.1 sec.

⇓

[5 sec.] ⇒ {great losses.}

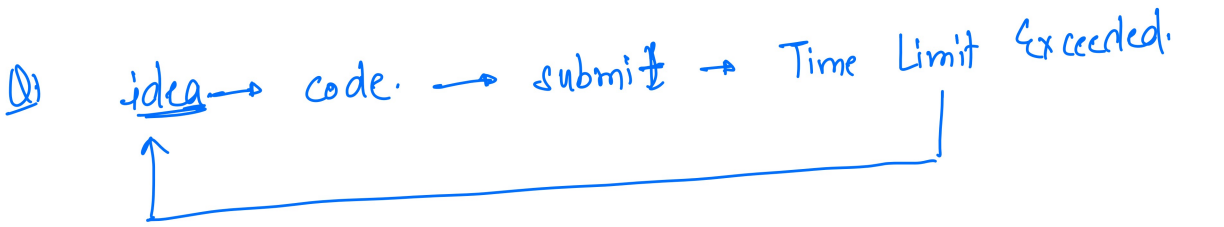
Microsoft → 0.01 sec

⇓

0.05 sec.

TLE (Time Limit Exceeded)

Shilpa → {Google} → {Hiring challenge} → 2 Qs → duration 1 hr.




Remarkable idea.:

Without even writing a single piece of code, we can tell whether this approach is going to work or not.

## Online Editor

Codes are executed on their code server.

⇓

Processing speed = 1 GHz.

$= 10^9$ instructions/sec.

⇓

⇒ Code should be executed in 1 sec

- variable declaration
- add/sub/ mult
- function calling
- comparing

observation: Our code can have at max $10^9$ instructions.

pseudo-code.

```
int countfactors ( int N){
int c=0;
for( int i=1 ; i<=N; i++){
    if(N%i == 0) { c++ }
}
return c;
}
```

iterations : N

instructions ≈ 6N

Approx. 1.

In our code, 1 iteration contains 10 instructions.

Our code can have at max = $10^9$ instructions.

= $10 \times 10^8$ instructions.

[Max iterations possible = $10^8$ iterations.]

$10^8 \times 10$ instructions;

$10^8$ iterations.

Approx: [on an avg. code is having 50-60 lines]

In our code, 1 iteration contains 100 instructions.

Our code can have at max = $10^9$ instructions.

= $10^7 \times 10^2$ instructions.

[Max iterations possible = $10^7$ iterations]

[In general, code iterations $\approx$ [$10^7 \sim 10^8$]]

Assumption → [Execution time for $10^8$ iterations = 1 sec]

# Structure to solve a question

Read Question
↓
Understand Question
↓
logic
↓
check correctness of logic
↓
code.

constraints [Most useless info]

$1 <= N <= 10^5$

idea → pseudo-code in your think

- 1 loop
- nested-loop
- sorting.

idea → T.C → $O(N^2)$

⇒ $10^{10}$ iterations.

Not going to work.

⇓

[think of the optimized approach]

Eg: $1 <= N <= 10^3$

idea → pseudo-code → T.C → $O(N^2)$

largest value, $N = 10^3$

Iterations = $(10^3)^2 = 10^6$ iterations;

⇓

[Directly go with the approach that you are thinking.]

Eg: $1 \le N \le 10^4$

idea → pseudo code → T.C → $O(N^2)$

$N = 10^4$, no. of iterations = $(10^4)^2$

$= 10^8$ iterations.

{hit / miss}

⇓

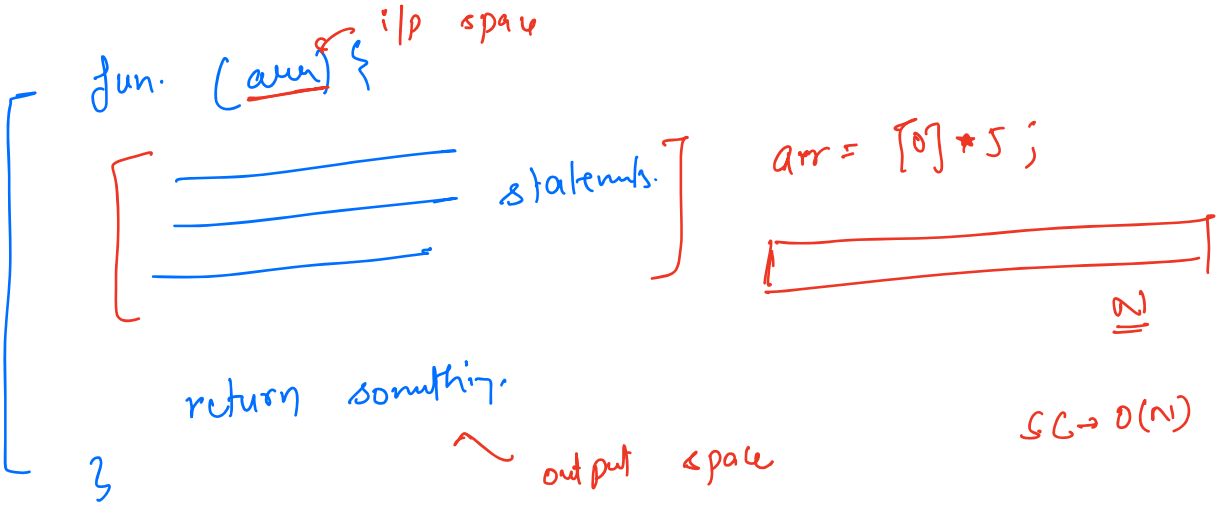
{think of optimising the code}

---

→ Modular arithmetic.

| | Algo1 | Algo2 |
|---|---|---|
| best-case | 5 | 5000 |
| avg-case | 10,000 | 11,000 |
| worst-case | $10^7$ | $10^5$ iterations. |

→ Big-O.




$5n^3 + 6n^2$

$\Downarrow$

$4n^3 + n^3 + 6n^2$

[growth rate of $n^3$ >> $n^2$]

$4n^3 + 10n^2$

$4n^3 + 10n^2$

Doubts :

# table.

```
for( i=n ; i>0 ; i/=2){
    if( i%2 == 0){
        for(j=1 ; j<=n+n ; j+=2){
        }
    }
}
```

| i | j | no. of iterations |
|---|---|---|
| N | $[1, N^2]$ | $\frac{N^2+1}{2}$ |
| $N/2$ | $[1, N^2]$ | $N^{2+1}/2$ |
| $N/4$ | $[1, N^2]$ | $N^{2+1}/2$ |
| 1 | | 1 |
| 1 | | $N^{2+1}/2$ |

$j=1 ; j<=n ; j+=2$

$\Downarrow$

$\frac{n+1}{2}$

T.C → $\frac{N^2}{2} \cdot \log_2 N$

```
for( i=1 ; j=1 ; j<=n ; i++){
    print(i+j);
    if( i%n == 0){
        j++;
    }
}
```

| j | i | iterations |
|---|---|---|
| 1 | $[1, n]$ | n |
| 2 | $[n+1, 2n]$ | n |
| 1 | | 1 |
| 1 | | 1 |
| 1 | | 1 |
| n | | n |

# no. of iterations = $n^2$

i : 1 2 3 4 —— N N+1 N+2 N+3 —— (2N) N+N

j : 1 1 1 1 -- 1 2 2 2 ——.

1. 2.

Assymptotic notations. ⇒ [ Θ → but can /avg. case
Ω → ]

```
int j = 0
for( int i = 0; i < n ; i++){
    while ( j <= i ){
        print(—)
        j++;
    }
}
```

```
for( i = 0; i < n ; i++){
    for ( j = 0; j <= i ; j++){
        print —
    }
}
```

table.

| i | j | iteration. |
|---|---|---|
| 0 | [0,0] | 1 |
| 1 | [1,1] | 1 + 1 |
| 2 | [2,2] | 1 |
| ⋮ | | ⋮ + 1 |
| N | [N,N] | 1 |

⇒ [total no. of iteration = N]

2 3

j = 1

$\{1 \le N \le 10^9\}$

$O(N^2) \Rightarrow$ iteration $= 10^{18}$

$\Downarrow$

{T.L.E}

$\{1 \le N \le 30\}$

T.C $\rightarrow O(2^N)$

$= 2^{30} = 10^9$

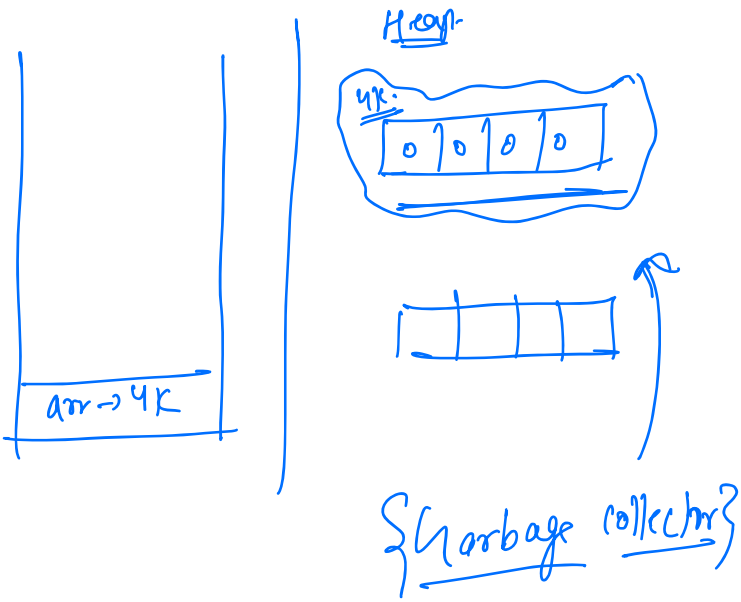
```
for( i = 1 ; i <= 100 ; i++){
    =
}
```

$i \rightarrow [1, 100]$

T.C $\rightarrow O(1)$

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

N




{Garbage collector}

At max → N space

S.C → O(N)

```
i = 1;
while ( i <= N){
    print ( name)
}
```

intermediate

↓

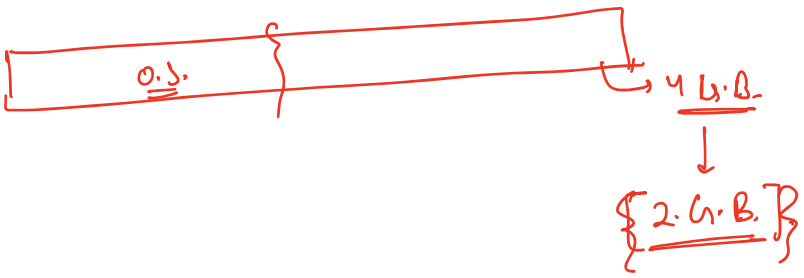
2 months

Advanced

↓

4.5 | 5 months.

→ {4 months}.




1 int → 4 Bytes.

$(x * 4)$ bytes $= 2 \times 10^9$.

$$x = \frac{2 \times 10^9}{4 \times 2} = 5 \times 10^8.$$