Time & Space Complexity

Ideal :→ Time ↓ and Space ↓ (most optimized)

Time ↑ Space ↓ vs Time ↓ Space ↑

Algo 1 Algo 2 ✓

Time Complexity (1) Runtime → Machine Dependent

eg leetcode → similar machines

② Growth of runtime over input size (Asymptotic Analysis)

input size: N = 100 students → (*10) → N = 1000 students → (*10) → n = 10^4 students

runtime: 1 ms → (*10) → 10 ms → (*10) → 100 ms — Machine 1 (mac)

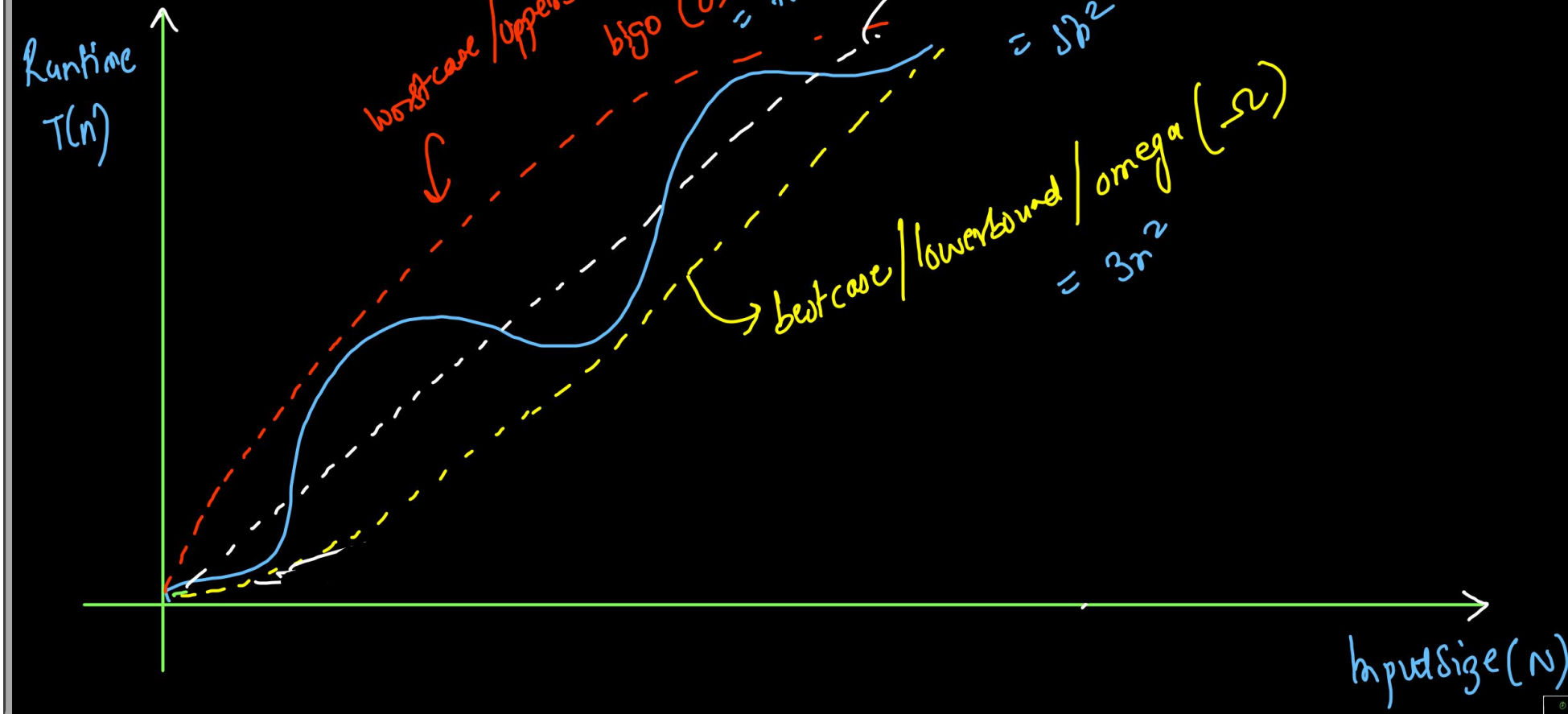
5 ms → (*10) → 50 ms → (*10) → 500 ms — Machine 2 (i3 window)

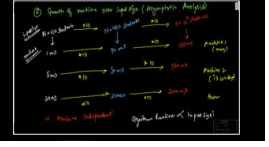
20 ms → (*10) → 200 ms → (*10) → 2000 ms — Pentium

⇒ Machine Independent

Algorithm Runtime ∝ Input size?

Growth of runtime ⇒ $T(n) = 5n^2 + 3n + 6$





$T(n) = 5n^2 + 3n + 6$ → Ignore smaller terms

$O(7n^2) \approx O(n^2)$ (Ignore constant terms)

$7n^2 > 5n^2 + 3n + 6$

$n = 1$, $n = 2$

$7 \not> 5 + 3 + 1$

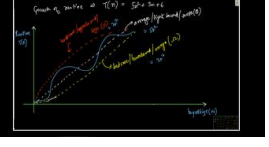
$7 \times 4 = 28 \not> 5 \times 4 + 3 \times 2 + 1$

Ignore Smaller Terms

$n = 3$ $\quad 7 \times 3^2 > 5 \times 3^2 + 3 \times 3 + 1$

$= 63 \quad = 60$

$n \geq 3$ ⇒ Good!



Runtime $T(n)$ | Worst case

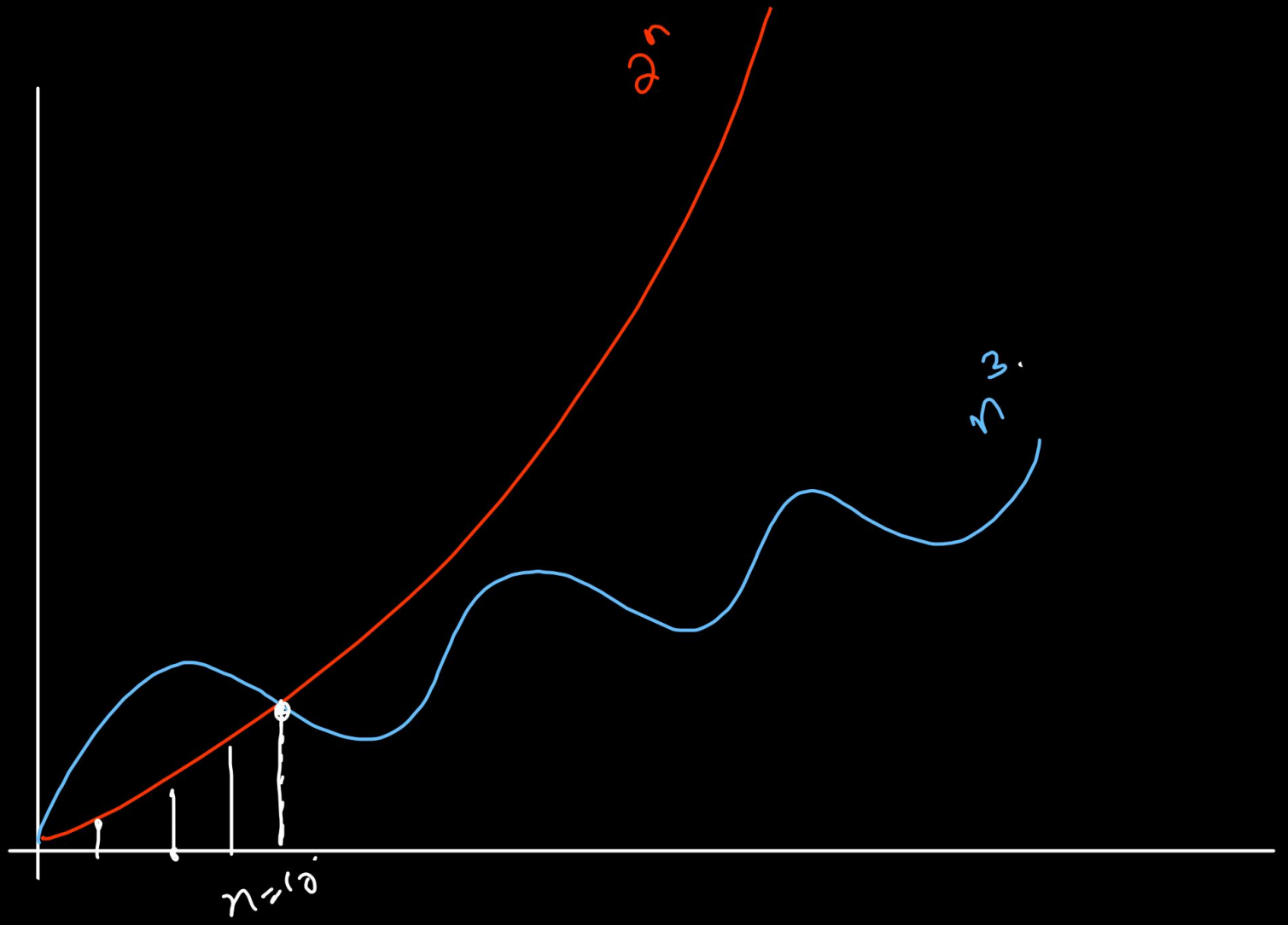
Q1 $4n^3 + 100n + 200$ ⇒ $O(n^3)$ = cubic

Q2 $100n + 200n + 300n$
$= 600 * n$ ⇒ $O(n)$ = Linear

Q3 $200 + 100 + 400$
$= 700 n^0$ ⇒ $O(n^0) = O(1)$ = constant

Q4 $n^5 + n^3 + n^1 + 8$ ⇒ $O(n^5)$

Q5 $2^n + n^{10} + n^9 + n^8$ ⇒ $O(2^n)$ ⇒ exponential


2^n
n^3
n=10

Time Complexity Chart

$O(1) < O(\log_2 n) << O(\sqrt{N}) < O(N) < O(N \log N)$

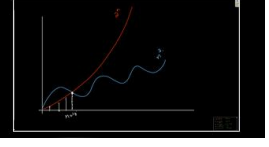
poly

Constant — Logarithmic — Square root — Linear

$< O(N^2) < O(n^3) <<< O(2^n) < O(3^n) < O(n!)$

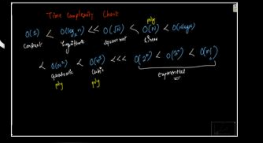
Quadratic — Cubic — exponential

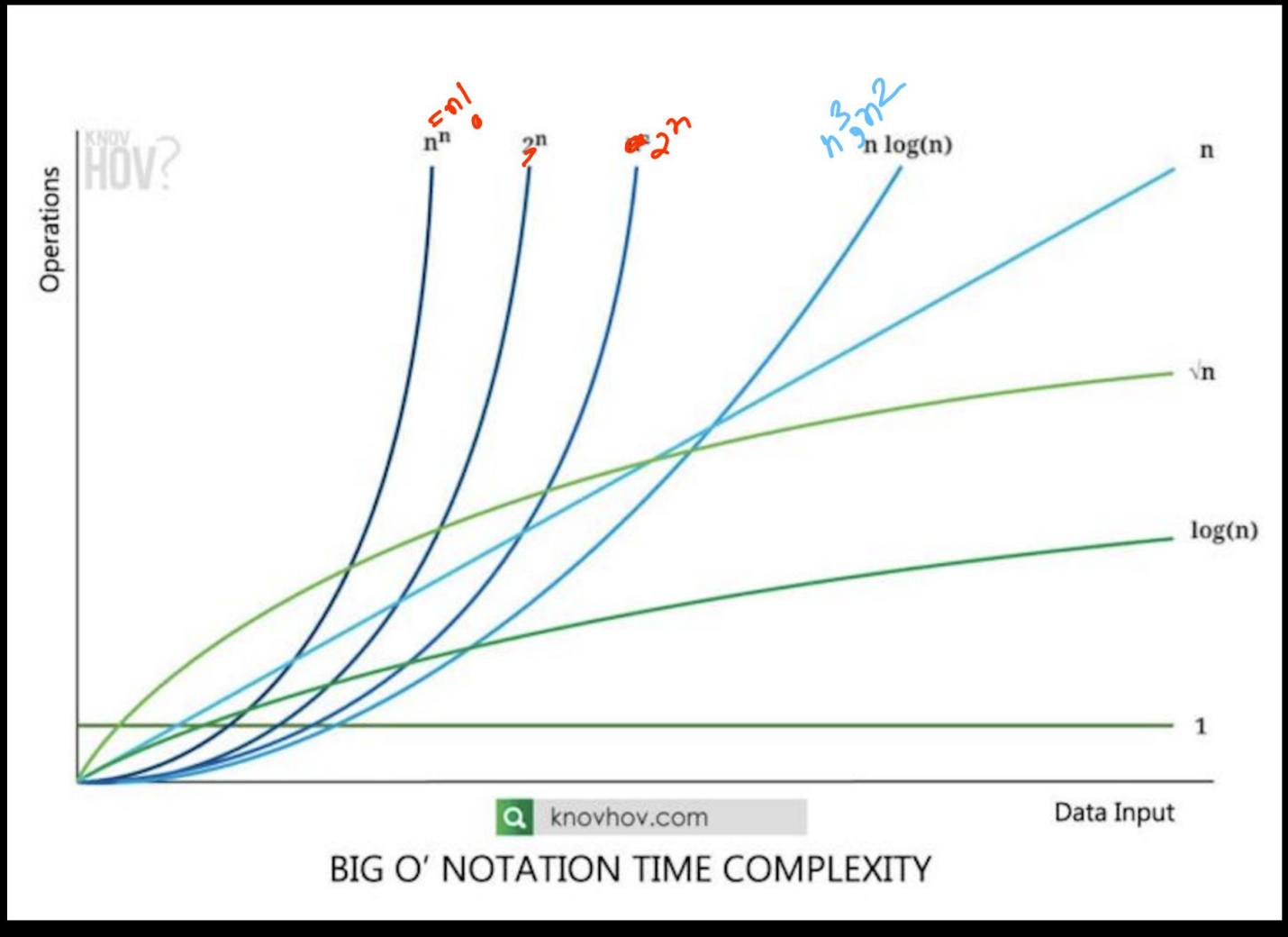
poly — poly



O(1) Constant Time Complexity

1. System.out.print() — Printing
2. Scanner scn = new Scanner(System.in);
   int marks = scn.nextInt(); } Inputs for variable
3. logical Operators (&&, ||, !) , Comparison operators (==, !=, >, >=, <, <=) , arithmetic operations (+, -, /, %, *)
4. Assignment operation (=)
5. if-else conditions / switch
6. Function call
   - Passing Parameter
   - return Statement
7. Array Indexing (arr[idx])
8. Bitwise operators (&, |, ^, ~, <<, >>)
9. ternary operators (?:)




KNOV HOV?
Operations
n^n
2^n
n log(n)
n
√n
log(n)
1
knovhov.com
Data Input
BIG O' NOTATION TIME COMPLEXITY

for loops time complexity.

(1)

```
for(int idx = 1; idx <= n; idx++){
    System.out.print(idx + " ");
}
```

$n = 5$

idx = 1 — k ms = O(1)

idx = 2 — k ms = O(1)

idx = 3 — k ms = O(1)

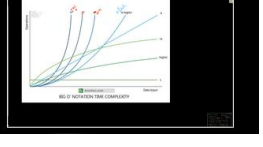
idx = 4 — k ms = O(1)

idx = 5 — k ms = O(1)

k + k + k + k + k = 5k

k + k + ... n times = n * k

O(n) = linear



(2)

```
int n = scn.nextInt();

for(int idx = 1; idx <= n; idx++){
    System.out.print(idx + " ");
}

int m = scn.nextInt();
for(int idx = 1; idx <= m; idx++){
    System.out.print(idx + " ");
}
```

$n = 5$, $m = 3$

idx = 1 — k ms
idx = 2 — k ms
idx = 3 — k ms
idx = 4 — k ms
idx = 5 — k ms

} 5 * k ms = n * k ms

+

idx = 1 — k ms
idx = 2 — k ms
idx = 3 — k ms

} 3 * k ms = m * k ms

$(n+m) * k$ ms

$\Rightarrow O(n+m)$

(linear)

Two loops which are independent → time complexity will add up !

Time Complexity will multiply if both loops are dependent!

③ Nested Loops

```
int n = scn.nextInt();

for(int i = 1; i <= n; i++){
    for(int j = 1; j <= n; j++){
        System.out.print(i + " " + j);
    }
}
```

O(1)

n = 4

i=1 → n times

j=1 kms
j=2 kms
j=3 kms
j=4 kms

i=2

j=1 kms
j=2 kms
j=3 kms
j=4 kms

i=3

j=1 kms
j=2 kms
j=3 kms
j=4 kms

i=4

j=1 kms
j=2 kms
j=3 kms
j=4 kms

$n \times k + n \times k + n \times k + n \times k$

$= 4nk$ ms $= n \cdot n \cdot k$ ms

$\Rightarrow O(n^2)$ quadratic

④

```
int n = scn.nextInt();
int m = scn.nextInt();

for(int i = 1; i <= n; i++){
    for(int j = 1; j <= m; j++){
        System.out.print(i + " " + j);
    }
}
```

$n=5$, $m=3$

$i=1$ $j=1, j=2, j=3$ mk +

$i=2$ $j=1, j=2, j=3$ mk +

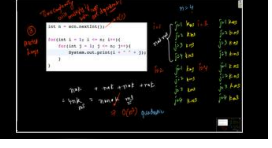
$i=3$ $j=1, j=2, j=3$ mk +

$i=4$ $j=1, j=2, j=3$ mk +

$i=5$ $j=1, j=2, j=3$ mk

$= n \times m \times k$

$O(n \times m)$



(5)

```
int n = scn.nextInt();

for(int i = 1; i <= n; i++){
    for(int j = 1; j <= i; j++){
        System.out.print(i + " " + j);
    }
}
```

n = 5

i = 1 → j = 1 → k ms = O(1) ⇒ 1 * k ms

i = 2 → j = 1 → k ms
j = 2 → k ms
} ⇒ 2 * k ms

i = 3 → j = 1 → k
j = 2 → k
j = 3 → k
} 3 * k ms

i = 4 → j = 1, 2, 3, 4 → 4k ms

i = 5 → j = 1, 2, 3, 4, 5 → 5 * k ms

$1k + 2k + 3k + 4k + 5k$

$= k(1 + 2 + 3 + \cdots n)$

$= k * \left(\frac{n*(n+1)}{2}\right) = \frac{kn^2}{2} + \frac{kn}{2}$

$\Rightarrow O(n^2)$

6

```
int n = scn.nextInt();

for(int i = 1; i <= n; i++){
    for(int j = i; j <= n; j++){
        System.out.print(i + " " + j);
    }
}
```

$n = 5$

$i = 1$ $j = 1, 2, 3, 4, 5$

$i = 2$ $j = 2, 3, 4, 5$

$i = 3$ $j = 3, 4, 5$

$i = 4$ $j = 4, 5$

$i = 5$ $j = 5$

$5k + 4k + 3k + 2k + 1k$ ms

$$\frac{n * (n+1)}{2} * k$$

$\Rightarrow O(n^2)$

(7)

```
for(int i = 1; i < n; i++){
    System.out.print(i + " " );
}
```

$1, 2, 3, 4, \ldots n-1 \Rightarrow n-1$ times

$nk - k \Rightarrow O(n)$

(8)

```
for(int i = 1; i <= n / 2; i++){
    System.out.print(i + " " );
}
```

$1, 2, 3, 4 \ldots n/2 \Rightarrow n/2$ times

$\Rightarrow \frac{n}{2} * k \Rightarrow O(n)$

n is +ve integer

```
int n = scn.nextInt();

for(int idx = n; idx <= 0; idx++){
    System.out.println(idx);
}

for(int idx = n; idx >= 0; idx++){
    System.out.println(idx);
}
```

(9) 0 iteration loops

n = 5
idx = 5
idx <= 0 5 <= 0 false

k ops
k ops

O(1) Constant

(10) infinite loop

n = 5
idx = 5 5 >= 0 k ops
6 6 >= 0 k ops
7 7 >= 0 k ops
8 8 >= 0 k ops
9

=> infinite time non terminating

```java
int n = scn.nextInt();

for(int idx = 1; idx * idx <= n; idx++){
    System.out.println(idx);
}
```

(11)

(or)

```java
int root = (int)Math.sqrt(n);
for(int idx = 1; idx <= root; idx++){
    System.out.println(idx);
}
```

$n = 4 \Rightarrow 2$

idx = 1 $\quad 1*1 \leq 4$ ✓

idx = 2 $\quad 2*2 \leq 4$ ✓

$n = 9 \Rightarrow 3$

idx = 1 $\quad 1*1 \leq 9$ ✓

idx = 2 $\quad 2*2 \leq 9$ ✓

idx = 3 $\quad 3*3 \leq 9$ ✓

$n = 16 \Rightarrow 4$

idx = 1 $\quad 1*1 \leq 16$

idx = 2 $\quad 2^2 \leq 16$

idx = 3 $\quad 3^2 \leq 16$

idx = 4 $\quad 4^2 \leq 16$

$n = 25 \Rightarrow 5$

idx = 1 $\quad 1^2 \leq 25$

idx = 2 $\quad 2^2 \leq 25$

idx = 3 $\quad 3^2 \leq 25$

idx = 4 $\quad 4^2 \leq 25$

idx = 5 $\quad 5^2 \leq 26$

$\Rightarrow O(\text{root } N)$

$= O(\sqrt{N})$

$= O(N^{1/2})$

(12)

```
int n = scn.nextInt();

for(int idx = 1; idx <= n; idx = idx * 2){
    System.out.println(idx);
}
```

$n = 2^x \Rightarrow \log_2 n = x$

loop is running $x+1$ time

$= O(\log_2 n)$

$n = 4 = 2^2$ ($2 = \log_2 4$)

idx = 1 $\quad 1 \leq 4$

idx = 2 $\quad 2 \leq 4$

idx = 4 $\quad 4 \leq 4$

3 times

$n = 8 = 2^3$ ($3 = \log_2 8$)

idx = 1 $\quad 1 \leq 8$

idx = 2 $\quad 2 \leq 8$

idx = 4 $\quad 4 \leq 8$

idx = 8 $\quad 8 \leq 8$

4 times

$n = 16 = 2^4$ ($4 = \log_2 16$)

idx = 1 $\quad 1 \leq 16$

idx = 2 $\quad 2 \leq 16$

idx = 4 $\quad 4 \leq 16$

idx = 8 $\quad 8 \leq 16$

idx = 16 $\quad 16 \leq 16$

5 times

$n = 32 = 2^5$ $\quad 5 = \log_2 32$

6 times

$n = 64 = 2^6$ $\quad 6 = \log_2 64$

7 times

$n = 128 = 2^7$ $\quad 7 = \log_2 128$

8 times

13

```
for(int idx = n; idx >= 1; idx = idx / 2){
    System.out.println(idx);
}
```

$\Rightarrow O(\log_2 N)$

exponential → as N increases, 2^N grows rapidly

$N = 10 \rightarrow 20 \rightarrow 30 \rightarrow 40$

$2^{10} \rightarrow 2^{20} \rightarrow 2^{30} \rightarrow 2^{40}$

$1024 \rightarrow 1048576 \rightarrow 1073741824 \rightarrow 1.09 * 10^{12}$

logarithmic → as N grows, $\log_2 N$ grows very slowly

$N = 2^i \rightarrow 2^{10} \rightarrow 2^{100} \rightarrow 2^{1000}$

$\log n : \{ \rightarrow 10 \rightarrow 100 \rightarrow 1000$ → almost constant

14)
```
for(int idx = 0; idx * idx <= n; idx++){
    System.out.println(idx);
}
```
$O(\sqrt{N})$

idx = 0, 0*2, 0*3, ... infinite

15)
```
for(int idx = 0; idx <= n; idx = idx * 2){
    System.out.println(idx);
}
```
runtime error

16)
```
for(int idx = 1; idx <= n; idx = idx * 3){
    System.out.println(idx);
}
```
$O(\log_3 N)$

17)
```
for(int idx = 1; idx <= n; idx = idx * 10){
    System.out.println(idx);
}
```
$\log_{10} N$

18)
```
for(int idx = 1; idx <= n; idx = idx + 2){
    System.out.println(idx);
}
```
$\frac{n}{2} = O(n/2)$

19)
```
for(int idx = 1; idx <= n; idx = idx + 3){
    System.out.println(idx);
}
```
$\frac{n}{3} = O(n)$

20)
```
for(int idx = 1; idx <= n; idx++){
    for(int j = 0; j < 5; j++){
        System.out.println(idx);
    }
}
```
$\rightarrow O(5) = O(1)$

$\rightarrow n * O(1) = O(n)$

21)
```
for(int idx = 1; idx <= n; idx++){
    for(int j = 1; j <= n; j = j * 2){

    }
}
```
n

$\hookrightarrow \log_2 N$

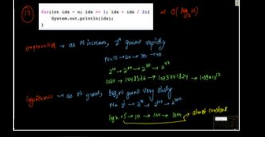
$= O(n * \log_2 N)$

$n = 10$
idx = 1, 3, 5, 7, 9

$n = 20$
idx = 1, 3, 5, 7, 9, 11, 13, 15, 17, 19

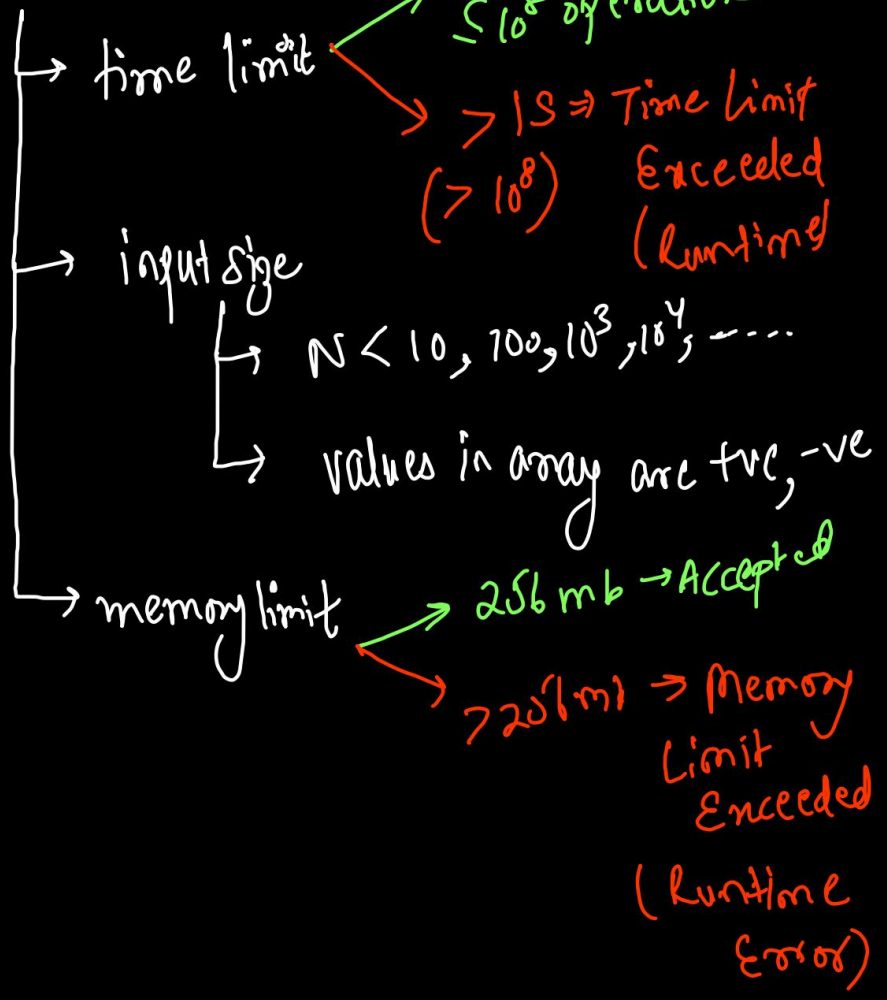
$n/2$ times

$\Rightarrow O(n)$



Competitive Programming Chart (Constraints)

1s = 10^8 operations per second



Time Complexity Chart

1 second = 10^8 operations
time limit = 1s

N_{max}	Time Complexity	Examples
18	$O(2^n)$ or $O(n!)$	Recursion & Backtracking
22	$O(2^n * n^2)$ } exponential	Travelling Salesman (DP with Bitmasking)
$4 * 10^2$	$O(N^3)$ cubic	3 nested loops (print all subarrays) Matrix Multiplication (3DPP)
10^4	$O(N^2)$ quadratic	Nested loops (Bubble/Insertion/Selection), matrix (2D DP)
10^6	$O(N \log N)$	Arrays.sort (Merge Sort / Quick Sort / Heap Sort) BS on Answer
10^8	$O(N)$ Linear	Linear search, max^m of Array, merge 2 sorted arrays (1D DP)
10^{16}	$O(\sqrt{N})$	Checking prime.
10^{18} or 2^{63} Long.MAX_VALUE	$O(1)$ or $O(\log_2 N)$ Constant / logarithmic	Digit Traversal, Binary No's, Binary Search

① Print Hello world → O(1) Constant time complexity ⇒ never give TLE
=) always run

Check odd or even, print a to z, print table of 4

② linear loop → O(n) linear time complexity

```
public static void main(String[] args) {
    Scanner scn = new Scanner(System.in);
    int n = scn.nextInt();

    for(int idx=0; idx<=n; idx++){
        System.out.println(idx);
    }
}
```

⇒ $N_{max} = 10^8$

Array Printing, N^{th} fibonacci

Integer.MAX_VALUE
$= 2.7 * 10^9$
↳ TLE

(3) Logarithmic loops ⇒ $O(\log N)$ ⇒ It will never give TLE

```
Scanner scn = new Scanner(System.in);
int n = scn.nextInt();

for(int i = n; i > 0; i = i/3){
    System.out.println(i);
}
```

} $\log_3 N$

N_{max} = no limit

$\log N = 18$

$N = 3^{18}$ = very big value = $+\infty$

$N = \underbrace{10^{18}}_{\text{long}} \Rightarrow$ digits = 18

```
for(int i = n; i > 0; i = i/10){
    System.out.println(i % 10);
}
```

} Print Digits of a No ⇒ $O(\log_{10} N)$

```
for(int idx=n; idx>=1; idx=idx/2){
    step=step+1;
}
```

} $\log_2 N$

(4) Quadratic Time (nested loops)

Grid/Box patterns, Bubble sort, insertion Sort, selection sort, Print Pairs

$= O(n^2)$

```
int n = sc.nextInt();
for (int i = 0; i<n; i++){
    for (int j = 0; j<n; j++){
        System.out.print("*");
    }
    System.out.println();
}
```

$N_{max} = 10^4$

$O(n^2) = 10^4 * 10^4$

$= 10^8 \leq 10^8$

Accepted

$N = 10^8$

$O(n^2) = O(10^8 * 10^8)$

$= O(10^{16})$

$> 10^8$

TLE

$N = 10^4$

$O(n^2) = 10^4 * 10^4$

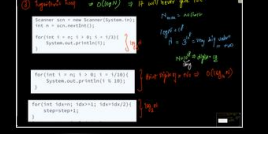
$= 10^{14}$

$> 10^8$

TLE

```
for(int i=0 ; i<n ; i++){
    for(int j=0 ; j<=i ; j++){
        System.out.print("* ");
    }
    System.out.println();
}
```

↳ Printing Upper/Lower Triangles



⑤ GCD (Euclid Algorithm) Long Division

```java
public static int gcd(int a, int b){
    int numerator = a, denominator = b;

    while(numerator % denominator != 0){
        int remainder = numerator % denominator;

        numerator = denominator;
        denominator = remainder;
    }

    return denominator;
}
```

$O(\log(\min(a,b)))$

$= \log N$

Check No is Prime

Approach (1)

```java
public static boolean isPrime1(int n){
    if(n == 1 || n == 2) return true; // 1 and 2 are prime

    for(int idx = 2; idx <= n - 1; idx++){
        // is idx a factor of n OR is n a multiple of idx
        if(n % idx == 0) return false;
    }

    return true;
}
```

$O(n)$ → linear

$N_{max} = 10^8$ (int)

Brute force

$N = 10^9, 10^{11} \cdots 10^{16}$ ⇐ time limit exceeded

Approach (2)

```java
public static boolean isPrime2(int n){
    if(n == 1 || n == 2) return true; // 1 and 2 are prime

    int sqrt = (int)Math.sqrt(n);

    for(int idx = 2; idx <= sqrt; idx++){ // idx * idx ≤ n
        // is idx a factor of n OR is n a multiple of idx
        if(n % idx == 0) return false;
    }

    return true;
}
```

$O(\sqrt{N}) = O(N^{1/2})$

$N_{max} = (10^8)^2 = 10^{16}$ long

Optimized approach

Linear Search Algorithm

```java
public static String linearSearch(int[] arr, int target){
    for(int idx = 0; idx < arr.length; idx++){
        if(arr[idx] == target) {
            return "True";
        }
    }

    return "False";
}
```

$N_{max} = 10^8$

Worst case → $O(n)$ linear

Average case → $\Theta(n)$ linear

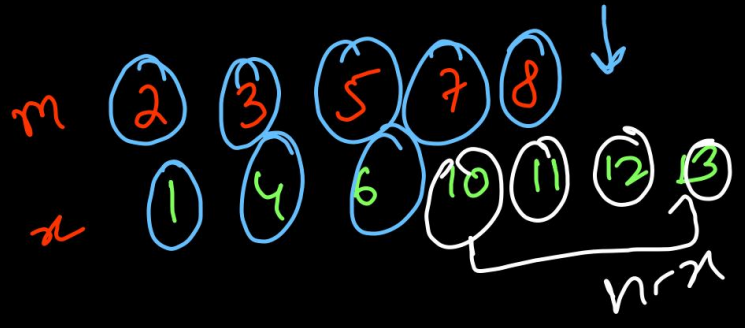
Best case → $\Omega(1)$ constant

Merge 2 Sorted Arrays

```java
public static int[] merge(int[] arr1, int[] arr2){
    int first = 0, second = 0, third = 0;
    int[] res = new int[arr1.length + arr2.length];

    while(first < arr1.length && second < arr2.length){
        if(arr1[first] <= arr2[second]){
            res[third] = arr1[first];
            first++;
            third++;
        } else {
            res[third] = arr2[second];
            second++;
            third++;
        }
    }

    while(first < arr1.length){
        res[third] = arr1[first];
        first++;
        third++;
    }

    while(second < arr2.length){
        res[third] = arr2[second];
        second++;
        third++;
    }

    return res;
}
```

m = arr1.length

n = arr2.length



O(m+n)

Print all subarrays

```
public static void printSubarrays(int[] arr){
    for(int start = 0; start < arr.length; start++){
        for(int end = start; end < arr.length; end++){
            for(int idx = start; idx <= end; idx++){
                System.out.print(arr[idx]+ " ");
            }
            System.out.println();
        }
    }
}
```

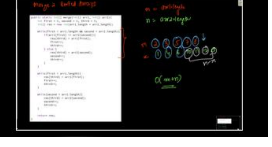
n → n → n

$n \times n \times n = O(n^3)$

cubic

$N_{max} = 500$

$= 10^2 - 10^3$



Target Sum Subarray

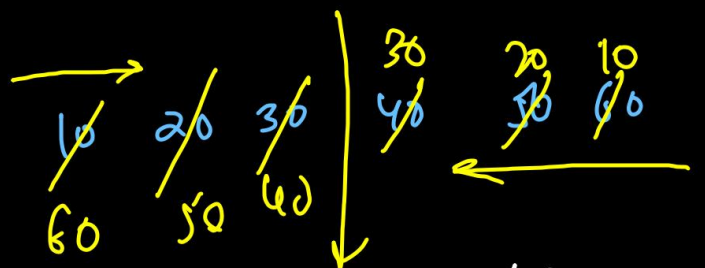
```java
public static boolean targetSumSubarray(int[] arr, int target){
    for(int st = 0; st < arr.length; st++){
        int sum = 0;
        for(int end = st; end < arr.length; end++){
            sum = sum + arr[end];
            if(sum == target) return true;
        }
    }
    return false;
}
```

$O(n^2)$ quadratic

$N_{max} = 10^4$

Reverse Array

```java
public static void reverseArray(int[] arr){
    int left = 0, right = arr.length - 1;

    while(left < right){
        int temp = arr[left];
        arr[left] = arr[right];
        arr[right] = temp;

        left++; right--;
    }
}
```



$O(n/2) + O(n/2)$

$= O(n)$ linear

$N_{max} = 10^8$

Sort 01

```
public static void sortBinary(int[] arr){
    int left = 0, right = 0;
    while(right < arr.length){
        if(arr[right] == 1){
            right++;
        } else {
            int temp = arr[left];
            arr[left] = arr[right];
            arr[right] = temp;

            left++; right++;
        }
    }
}
```

O(n)

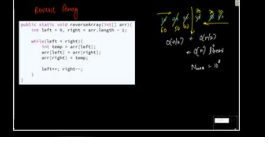
Two Pointer

Most optimized

Sort 012

```
public static void sort012(int[] arr){
  int left = 0, mid = 0, right = arr.length-1;
    while(mid <= right){
        if(arr[mid] == 0){
            swap(arr,left,mid);
            left++;
            mid++;
        }
        else if(arr[mid] == 1){
            mid++;
        }
        else{
            swap(arr,right,mid);
            right--;
        }
    }
}
```

O(n)



Space Complexity

1. Input Space complexity (Parameters)
2. Output Space Complexity (Return type)
3. Recursion call stack space (Recursion)
4. * Extra or Auxiliary Space (Data Structures to perform Algo)
   - No extra space soln is in-place algorithm

O(1) space

normal primitive Variables

O(n) space

1D Array / String

O(n²) space

2D Array / matrix