Time & Space Complexity Analysis

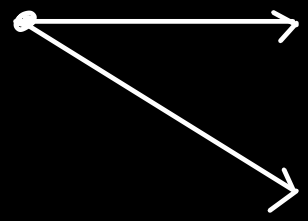
computer resources :→

- CPU (OS)
- memory (RAM) → space (stack, heap, static)
  → time (execution/runtime)
- disk (HDD or SDD) (DBMS)
- network (CN)

ideal solution :→ Minimum execution time (time ↓)

Minimum space allocation (space ↓)

tradeoff



time ↓ ⇒ space ↑ **preferred**

space ↓ ⇒ time ↑

Time Complexity Analysis

① Actual runtime

→ no relation with input size

→ system constraints

→ language constraints

Order of execution

C < C++ << Java < Python < Javascript

2s 4s 10s

machine dependency

faster machine ⇒ less runtime

Solution

- Asymptotic analysis ⇒ rate of growth of runtime wrt input size
- Amortized analysis ⇒ mean of all actual asymptotic analysis.

Asymptotic Analysis

linear search
liner growth

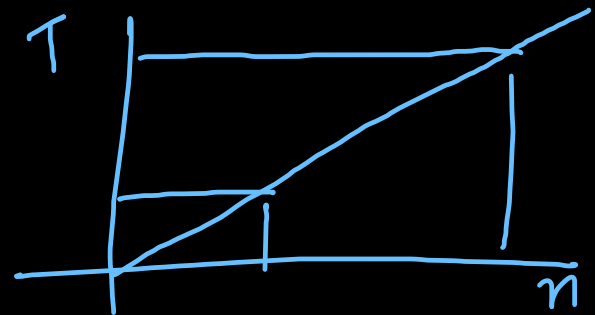
machine m2 macbook

windows i3

$n = 10$
↓ *10
$n = 100$
↓ *10
$n = 1000$
↓ *10
$n = 10000$

1ms
↓ *10
10 ms
↓ *10
100 ms
↓ *10
1 s

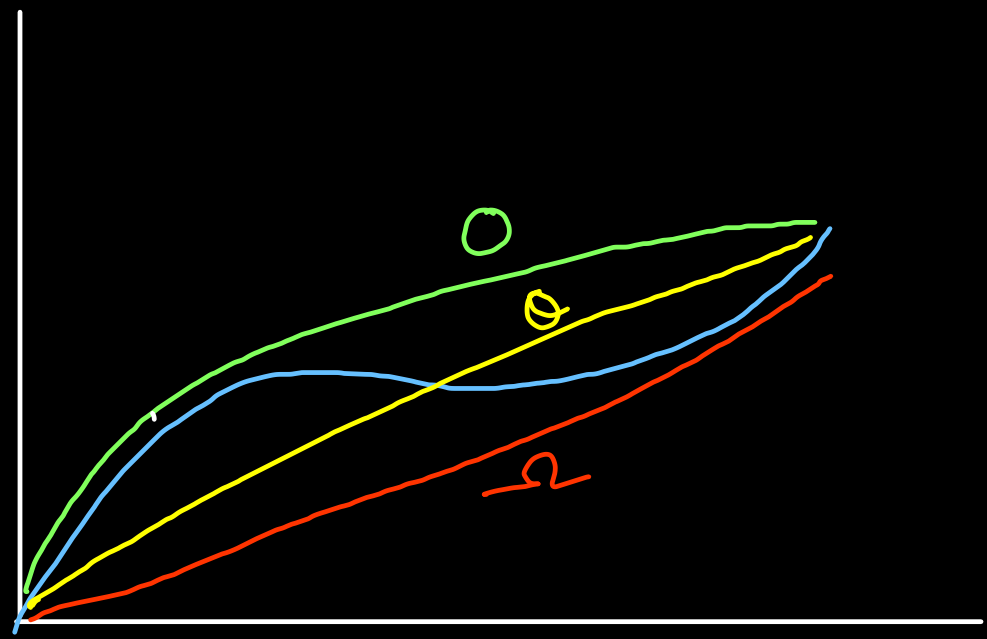
2 ms
↓ *10
20 ms
↓ *10
200 ms
↓ *10
2s



① best case analysis (atleast time) ⇒ omega $\Omega(\ )$ lower bound

② average case analysis ⇒ theta $\Theta(\ )$ tight bound

③ worst case analysis (atmost time) ⇒ bigo $O(\ )$ upper bound



$T(n) = 3n^2 + 5n + 6$

$O(3n^2)$
$\Theta(3n^2)$
$\Omega(3n^2)$

$\Rightarrow O(n^2)$

Ignore smaller terms (small power of n)

Ignore constants

eg $T(n) = 3n^2 + 5n + 6 \Rightarrow O(n^2)$

$T(n) = 5n^3 + 2n^3 + 10n^3 \Rightarrow O(n^3)$

$T(n) = n^2 + 1000n + 10^6 \Rightarrow O(n^2)$

$T(n) = 10 + 200 + 450 + 6000 \Rightarrow O(1)$ constant

Time Complexity Chart

almost constant

$O(1)$ < $O(\log n)$ <<< $O(\sqrt{n})$ or $O(n^{1/2})$ <

$O(1)$ constant

$O(\log n)$ logarithmic

$O(\sqrt{n})$ or $O(n^{1/2})$ root

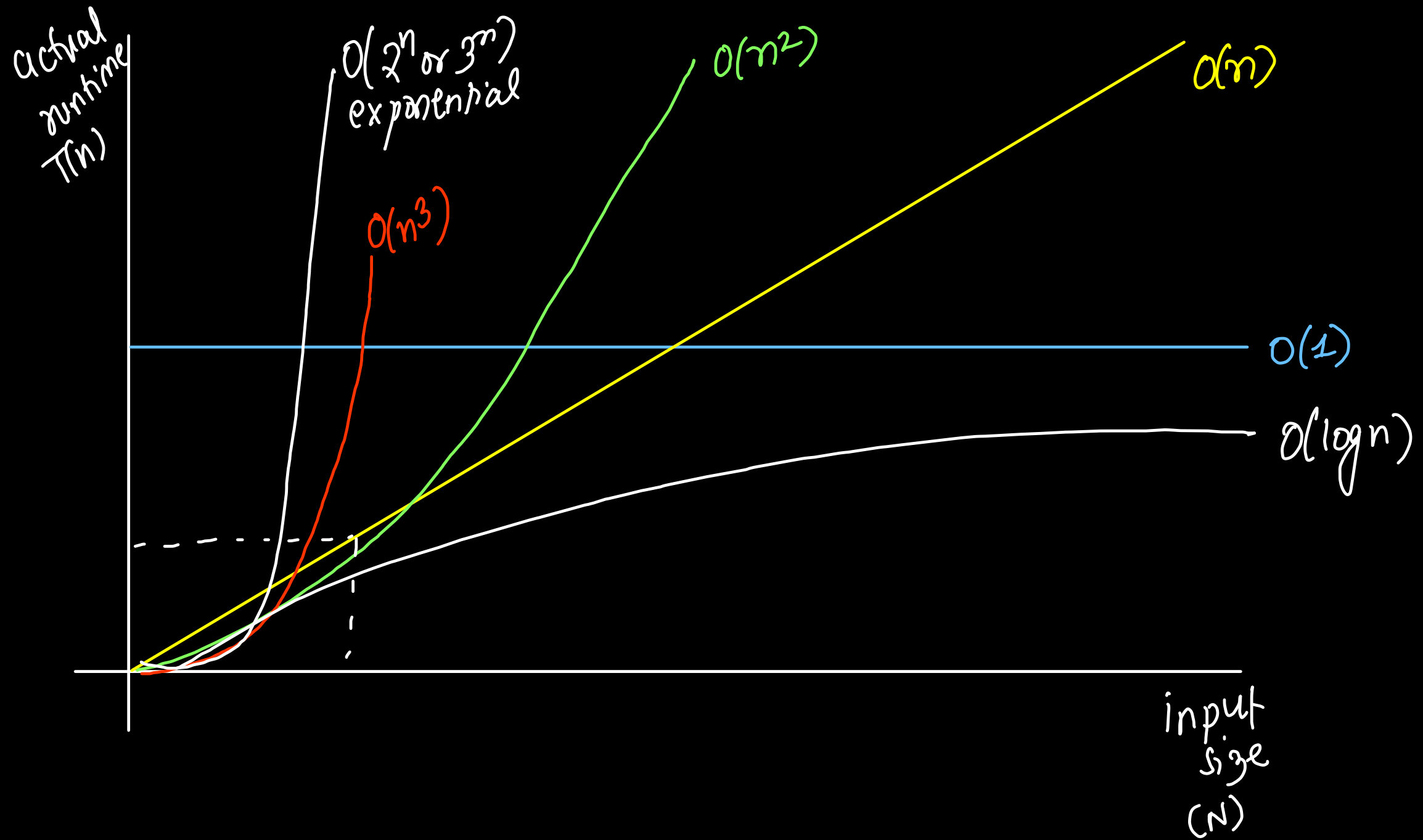
$O(n)$ < $O(n \log n)$ < $O(n^2)$ < $O(n^3)$ — polynomial

$O(n)$ linear

$O(n^2)$ quadratic

$O(n^3)$ cubic

<<< $O(2^n)$ < $O(3^n)$ < $O(n^n)$ or $O(n!)$ — exponential


actual runtime T(n)
O(2^n or 3^n) exponential
O(n^2)
O(n)
O(n^3)
O(1)
O(log n)
input size (N)

Ignore smaller values of n

eg 1

$O(n^2)$ Insertn Sort

better

$O(n \log n)$ mergesort

$n < 16$

$O(n \log n)$ mergesort

better

$O(n^2)$ insertn sort

$n > 16$

eg 2

$O(n^2) > O(2^n)$

if $n = 1$

Constant Time Complexity O(1)

print or println primitive variables

assignment operation (=) (shallow copy for reference)

input (Scanner) nextInt(), ... funcns

Conditional statements (if-else, switch, ternary operator)

Operators → arithmetic (+, -, %, /, *)
→ logical (&&, ||, !)
→ Comparison (<, >, ==, !=, <=, >=)
→ bitwise (!, &, |, ^)

function call or return
↓ Pass by value
↓ Shallow copy

Array Indexing (random access)

Iterator (loops) time complexity

(1)

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

$n = 5$

k ms — idx = 1 — $1 \leq 5$ — (1)

k ms — idx = 2 — $2 \leq 5$ — (2)

k ms — idx = 3 — $3 \leq 5$ — (3)

k ms — idx = 4 — $4 \leq 5$ — (4)

k ms — idx = 5 — $5 \leq 5$ — (5)

5k ms or nk ms

$O(n)$

(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 + " ");
}
```

O(n) time

+

O(m) time

O(n+m)

(= O(n) linear)

independent loops

20 sal
↓ *10
200 sal

10 sal
↓ *10
100 sal

Square or grid

(3)

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

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

$n = 4$

$i = 1$ $j = 1,2,3,4 \Rightarrow O(n)$

$+$

2 $j = 1,2,3,4 \Rightarrow O(n)$

$+$

3 $j = 1,2,3,4 \Rightarrow O(n)$

$+$

4 $j = 1,2,3,4 \Rightarrow O(n)$

$= O(n*n)$

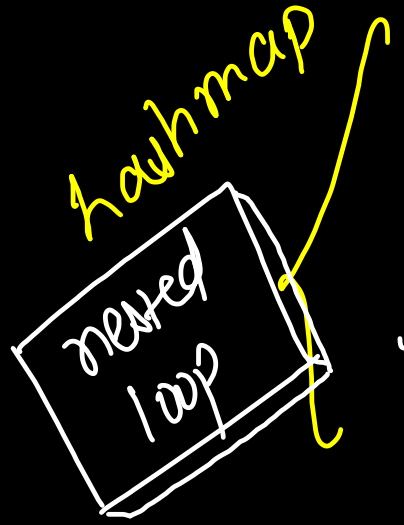
$= O(n^2)$

quadratic

dependent loops

eg hashmap, sliding window, monotonic stack, DFS/BFS

{ 1, 2, 3, 4, 1, 2, 3, 3, 4, 5 }



i = distinct numbers

j = frequency (count)

i = 1, 2, 3, 4, 5

j = 1 1 1 1 1

2 2 2 1

2 2 3 2

2k + 2k + 3k + 2k + 1k

= 10k ⇒ O(n)

Corner case ①

{1,2,3,4,5} outer n

inner 1

Corner case ②

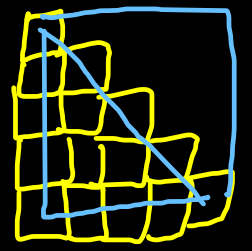
{1,1,1,1,1} outer 1

inner n

④

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

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



Lower Left Triangle

$n = 5$

$i=1 \Rightarrow j=1 \Rightarrow 1k$

$i=2 \Rightarrow j=1,2 \Rightarrow 2k$

$i=3 \Rightarrow j=1,2,3 \Rightarrow 3k$

$i=4 \Rightarrow j=1,2,3,4 \Rightarrow 4k$

$i=5 \Rightarrow j=1,2,3,4,5 \Rightarrow 5k$

overall time : $1k + 2k + 3k + \cdots nk$

$= k(1+2+3+\cdots n) = k * \frac{n*(n+1)}{2} = \frac{kn^2}{2} + \frac{kn}{2}$

$\Rightarrow O(n^2)$

(5)

```
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	5k +
2	j = 2, 3, 4, 5	4k +
3	j = 3, 4, 5	3k +
4	j = 4, 5	2k +
5	j = 5	k

total time = $\frac{n*(n+1)}{2}$ k ms

$\Rightarrow O(n^2)$ quadratic

upper left triangle

(6)

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

n = 10

i = 1, 2, 3, 4, 5, 6, 7, 8, 9

(n-1)x ⇒ O(n)

(7)

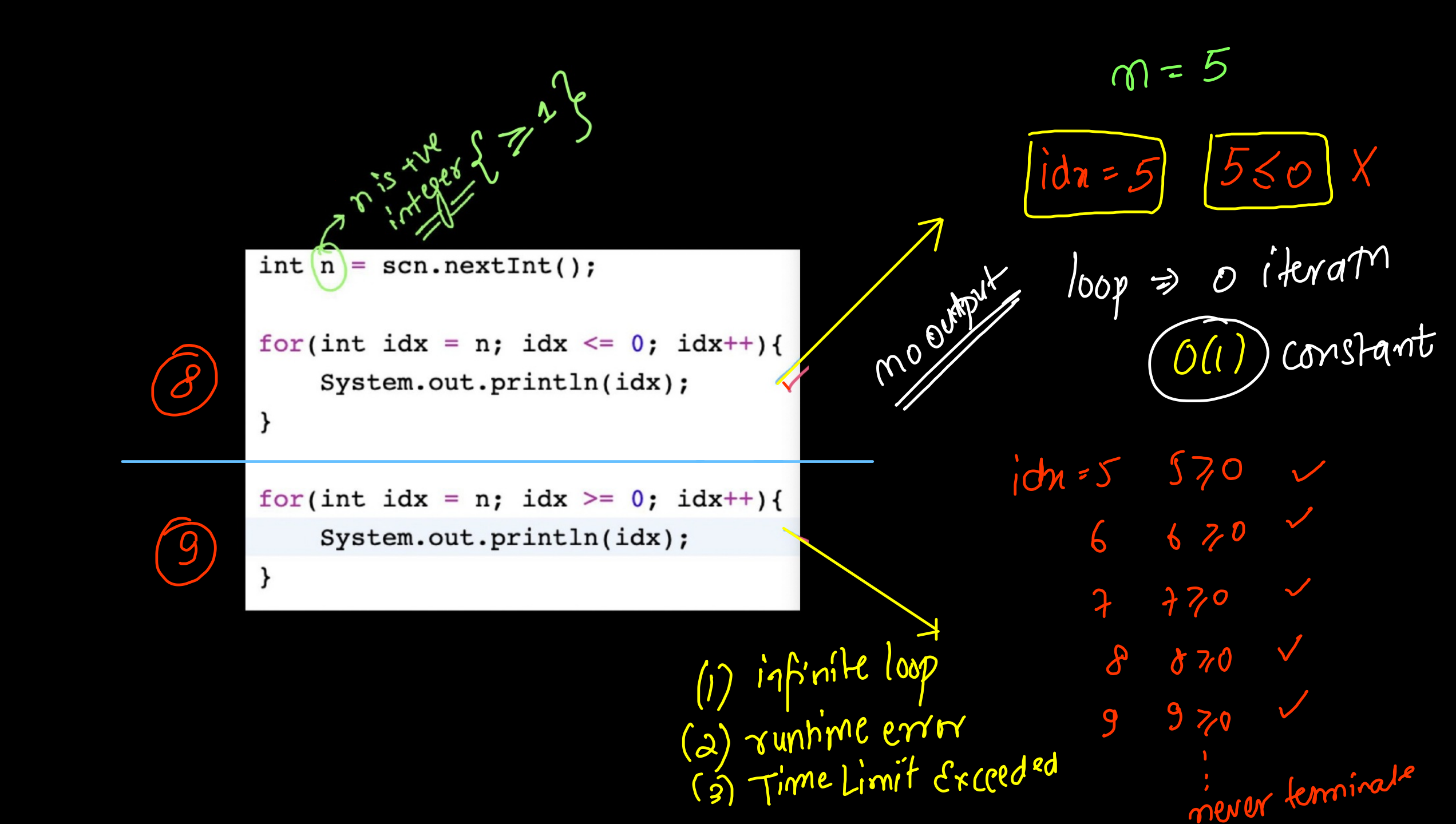
```
for(int i = 1; i <= n / 2; i++){
    System.out.print(i + " " );
}
```

n = 10

i = 1, 2, 3, 4, 5

(n/2)x ⇒ O(n)

linear O(n) = O(n-1) = O(n+1) = O(2n) = O(n/2)


n = 5
n is +ve integer {≥1}
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);
}
8
9
idx = 5
5 ≤ 0 X
no output
loop ⇒ 0 iteration
O(1) constant
idx = 5 5 ≥ 0 ✓
6 6 ≥ 0 ✓
7 7 ≥ 0 ✓
8 8 ≥ 0 ✓
9 9 ≥ 0 ✓
(1) infinite loop
(2) runtime error
(3) Time Limit Exceeded
never terminate

eg Check prime

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

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

(10)

(or)

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

$n = 16$

4 times

idx = 1 : $1*1 \le 16$ ✓

idx = 2 : $2*2 \le 16$ ✓

idx = 3 : $3*3 \le 16$ ✓

idx = 4 : $4*4 \le 16$ ✓

idx = 5 : $5*5 > 16$ ✗

$n = 100$

$1^2, 2^2, 3^2, \ldots 10^2 \le 100$

$O(\sqrt{n})$ or $O(n^{1/2})$ time

$root = \sqrt{n} = 4$

$1 \le 4$ ✓ $3 \le 4$ ✓

$2 \le 4$ ✓ $4 \le 4$ ✓

$5 > 4$ ✗

O(?) time complexity

(11)

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

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

$n = 2^x$

loop is running for x times

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

$\Rightarrow \boxed{\log_2 n = x}$

$\Downarrow$

O(logarithmic)

$n = 16 = 2^4$

idx = 1 < 16 ✓

2 < 16 ✓

4 < 16 ✓

8 < 16 ✓

16 < 16 ✗

$n = 128 = 2^7$

idx = 1 < 128 ✓

2 < 128 ✓

4 < 128 ✓

8 < 128 ✓

16 < 128 ✓

32 < 128 ✓

64 < 128 ✓

128 < 128 ✗

(almost constant)

input size (N)	logarithmic $O(\log_2 N)$	<<<	linear $O(N)$
$n = 2$	$\log_2 2 = 1$		2
$n = 2^2 = 4$	$\log_2 4 = 2$		4
$n = 2^3 = 8$	$\log_2 8 = 3$		8
$n = 2^{10}$	$\log 2^{10} = 10$		1024
$n = 2^{31}$	$\log 2^{31} = 31$		$2^{31} = 2 \times 10^9$

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

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

Observam ⇒ Iterator / Pointer jump by *2, *3, *10

↳ O(logarithmic) /2, /3, /10, etc

n = 16

16 → 8 → 4 → 2 → 1

(12)

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

n = 128

128 → 64 → 32 → 16

1 ← 2 ← 4 ← 8

13

```
for(int idx = 1; idx <= n; idx = idx * 3){
    System.out.println(idx);
}
```

$O(\log_3 N)$

$3 \rightarrow 9 \rightarrow 27 \rightarrow 81$

$\hookrightarrow 3^4$

$4 = \log_3 3^4$

14

```
for(int idx = 1; idx <= n; idx = idx * 10){
    System.out.println(idx);
}
```

$O(\log_{10} N)$

$1 \rightarrow 10 \rightarrow 100 \rightarrow 1000 \rightarrow 10000$

$\Downarrow$

10^4

$4 = \log_{10} 10^4$

```
int res = 1;                          ⇒ O(1)
for (int idx=1; idx <= b; idx++){     ⇒ O(b)
    res = res * a;                    ⟶ O(1)
}
```

(15)

Time ⇒ O(b)

linear

Power Function { a^b }

16

```java
// Function to check if given number n is a power of two.
public static boolean isPowerofTwo(long n) {
    long power = 1;
    while (power < n) {
        power = power * 2;
    }

    if (power == n)
        return true;
    else
        return false;
}
```

for(long p=1; p<n; p*=2)

↳ $O(\log_2 N)$

logarithmic

int range: -2^{31} to $2^{31}-1$

long range: -2^{63} to $2^{63}-1$ → 63 times

Nth Fibonacci (Two Pointer Technique)

```java
public int fib(int n) {
    if (n <= 1)
        return n;
    int a = 0, b = 1, c = 1;
    for (int i = 2; i < n; i++) {
        a = b;
        b = c;
        c = a + b;
    }
    return c;
}
```

O(1)

O(1)

O(1)

O(n-2) = O(n) linear

(77)

Nth Fibonacci No ⇒ O(N) linear time

Fibonacci

① Recursion
- → Time : $O(2^N)$
- → Space = O(N)

② DP (memoization)
- → Time = O(N)
- → Space = O(N)

③ DP (Tabulation)
- → Time = O(N)
- → Space = O(1)

Print Lowercase English Alphabets

12

```java
public static void main(String[] args) {
    for (char ch = 'a'; ch <= 'z'; ch++) {
        System.out.println(ch);
    }

    // OR
    for (int idx = 0; idx < 26; idx++) {
        char ch = (char) (idx + 'a');
        System.out.println(ch);
    }
}
```

→ O(26) = O(1) constant

{ 256 characters asci }

FizzBuzz

```java
public static void main(String[] args) {
    int fizz = 0, buzz = 0;
    for (int idx = 1; idx <= n; idx++) {
        fizz++;
        buzz++;
        if (fizz == 3 && buzz == 5) {
            System.out.println("FizzBuzz");
            fizz = buzz = 0;
        } else if (fizz == 3) {
            System.out.println("Fizz");
            fizz = 0;
        } else if (buzz == 5) {
            System.out.println("Buzz");
            buzz = 0;
        } else {
            System.out.println(idx);
        }
    }
}
```

O(1)

O(n) time

Digit Traversal

eg 12345

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

    while (n != 0) {
        int digit = n % 10;
        System.out.println(digit);
        n /= 10; // n = n / 10
    }

    scn.close();
}
```

(14)

5, 4, 3, 2, 1

⇒ for(int idx = n; idx > 0; idx /= 10)

↳ number of digit traversals

$O(\log_{10} N)$ logarithmic

$n_{max} = 2^{31} - 1 = 10^9$ ⇒ max digit ≤ 9

⇒ $O(9) = O(1)$

Time Complexity Chart

Verdict :→

Complete marks ① Accepted (All testcases passed)

Partial marks ② Partially Accepted (some testcases passed)

↳ corner cases

- integer overflow
  - long
  - modular arithmetic
- negative elements or zero value
- input array is empty or only 1 element

0 marks

③ Wrong answer → Sample test cases wrong output
or → logical error (Index out of bound, Runtime Error, null pointer Except^n, etc)
compilation error

→ error at line no —

④ Time Limit Exceeded (TLE)

(or)

Memory limit Exceeded (MLE) ⇒ Stack Overflow

eg recursion

Constraints :→

worst case time complexity which can be afforded

1s C++ ⇒ 10^8 operations.

2s Java

↳ constant time O(1)

Code $\leq$ 1/2s ⇒ Accepted

Code $>$ 1/2s ⇒ TLE

Max Input Limit	Worst Case Time Complexity	Examples
-	O(1) constant	If-Else, Bit Manipulation/Bitmasking
-	O(logN) logarithmic	Digits, Power, Binary Search, Divide & Conquer
$N_{max} = 10^{16}$ (long)	$O(n^{1/2})$ or $O(\sqrt{n})$	Check Prime Number
$N_{max} = 10^8$	O(n) linear	Linear Search, Prefix/Suffix, 1D DP, 2 Pointers, BFS/DFS
$N_{max} = 10^5 - 10^6$	O(nlogn)	Merge/Quick/Heap Sort, Binary Search on Answer
$N_{max} = 10^4$	$O(n^2)$ quadratic	Grid/Matrix, Subarrays, 2D DP, Basic Sorting
$N_{max} = 400$	$O(n^3)$ cubic	3D DP, Floyd Warshall, Bellmon Ford, Matrix multiplication
$N_{max} = 25$	$O(2^N)$ exponential	Recursion & Backtracking (Subsets)
$N_{max} = 12$	O(N!) or $O(N^N)$	DP with bitmasking (Travelling salesman)

LC 169 Majority Element - I

Constraints:

- `n == nums.length`
- `1 <= n <= 5 * 10^4`
- `-10^9 <= nums[i] <= 10^9`

$N_{max} = 5 * 10^4$

Worst time Complexity

$= O(n \log n)$ ✓

① Each ele : freq
brute force
$\# O(n^2)$
$= O((5 \times 10^9)^2)$
$= O(25 * 10^8)$
$= 25s$
TLE

② Sorting ⇒ better soln ✓ Accepted
$O(n \log n)$

③ Boyer Moore Voting algorithm ⇒ Most optimized
⇒ $O(n)$

Space Complexity

~~actual ram space~~

rate of growth of space(ram) with respect of input size

① primitive variable ⇒ O(1) constant space

char	int	long
2byte	4bytes	8byte

② String or Array or Arraylist ⇒ O(n) linear space

- String → no of characters → .length()
- Array → array length → .length
- Arraylist → arraylist size → .size()

③ LL, Stack, Queue, Priority Queue, Trees

$\Rightarrow$ O(n) Space {number of nodes}

④ matrix or 2D array $\Rightarrow$ O(rows * columns)

$= O(n^2)$ {quadratic space}

⑤ Graph $\Rightarrow$ $O(V+E)$ $\Rightarrow$ adjacency list / edge list.

V: no of vertices

E: no of edges

Space Complexity

changes in input & no extra space
⇒ Inplace soln

(1) Input Space → Parameters

(2) Output Space → Return type

(*) (3) Extra Space or auxilliary data structure → inside fn to solve the problem

(4) Recursion call stack space → overhead (recursive calls → depth)

Linear Search

```
static int search(int arr[], int N, int target) {
    for (int idx = 0; idx < arr.length; idx++) {
        if (arr[idx] == target) {
            return idx;
        }

    }
    return -1;

}
```

Input ⇒ O(n) space array

output space ⇒ O(1) integer

extra space ⇒ O(1) inplace

time = O(n) linear

first & last occurence

```java
public int[] searchRange(int[] arr, int target) {
    int firstIndex = -1, lastIndex = -1;
    for (int idx = 0; idx < arr.length; idx++) {
        if (arr[idx] == target) {
            lastIndex = idx;
            if (firstIndex == -1) {
                firstIndex = idx;
            }
        }
    }
    int[] ans = { firstIndex, lastIndex };
    return ans;
}
```

2 elements = 8 bytes = O(1)

input space
⇒ O(n) array

output space
⇒ O(1) constant
(ans ⇒ size = 2)

extra space ⇒ O(1)
integers
inplace

Time = O(n) linear

Product of Array Except Self

```java
public int[] productExceptSelf(int[] nums) {
    int[] prefix = getPrefix(nums);
    int[] suffix = getSuffix(nums);

    int n = nums.length;
    int[] answer = new int[n];
    for (int idx = 0; idx < n; idx++) {
        int left = (idx > 0) ? prefix[idx - 1] : 1;
        int right = (idx < n - 1) ? suffix[idx + 1] : 1;
        answer[idx] = left * right;
    }
    return answer;
}
```

extra (auxiliary)

O(n)

O(n)

O(n)

input (parameter)

⇒ output (return)

input space
⇒ O(n) array

output space
⇒ O(n) array

# extra space ⇒
O(2n) = O(n)
due to prefix & suffix array

Time = O(3n) = O(n)
linear

Indexed Based HashMap

find duplicates

```java
public List<Integer> findDuplicates(int[] nums) {
    int n = nums.length;

    // visited mark
    for (int idx = 0; idx < n; idx++) {
        int original = nums[idx] % (n + 1);
        nums[original - 1] += (n + 1);
    }

    List<Integer> duplicate = new ArrayList<>();

    for (int idx = 0; idx < n; idx++) {
        int freq = nums[idx] / (n + 1);
        if (freq > 1)
            duplicate.add(idx + 1);

    }
    return duplicate;
}
```

O(n)

O(n)

input space
⇒ O(n) array

output space
⇒ O(n) woost case
(all duplicate)

extra space ⇒ O(1)
integers
inplace

Time : O(n) linear

```java
public static void main(String[] args) {
    // APPROACH 1 : 3 NESTED LOOPS
    for (int left = 0; left < arr.length; left++) {
        for (int right = 0; right < arr.length; right++) {
            for (int idx = left; idx <= right; idx++) {
                System.out.println(arr[idx] + " ");
            }
            System.out.println();
        }
    }

    // APPROACH 2 : 2 NESTED LOOPS
    for (int left = 0; left < arr.length; left++) {
        ArrayList<Integer> subarray = new ArrayList<>();
        for (int right = left; right < arr.length; right++) {
            subarray.add(arr[right]);
            System.out.println(subarray);

        }
    }
}
```

arr

{ 1, 2, 3, 4, 5 }

{1} 1

{1,2} 2

{1,2,3} 3

{1,2,3,4} 4

{1,2,3,4,5} 5

{2} 1

{2,3} 2

{2,3,4} 3

{2,3,4,5} 4

1 + 2 + 3

1 + 2

1

total time = $(1+2+3+\cdots n) + (1+2+\cdots(n-1)) + (1+2+\cdots(n-2)) + 1$