Time Complexity

Order Complexity Analysis:

Amount of time required or space required by an algorithm/code as a functio of input size

Not the actual time.

| 1 | 2 | 3 | 4 | 5 |

n = 6 → 2 ms
n = 100 → 100 ms

" Time for linear search in worst case will be remain same even if our array was sorted "

(worst case = O(n))

To find Largest value using.

| 1 | 2 | 3 | 4 | 5 | 6 |

Largest = arr[n-1]

sorted (Ascending order).

Case 1

n↑ Time↑

$T \propto n$

Case 2

n↑ T-const

Time is constant

Time complexity find out by :-

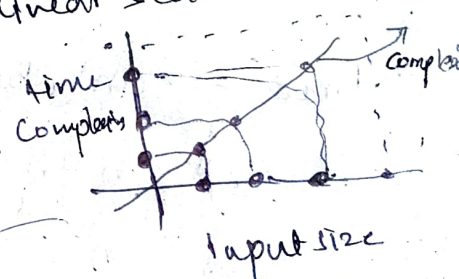
Experimental

by doing experiment

t = 0 ms
t = 1 ms
t = 5 ms.

theorital

Linear search :-

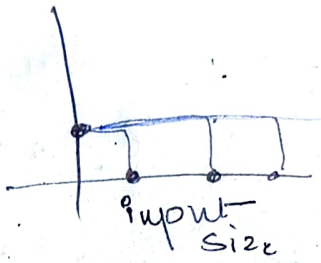


Relationship or function

$y = ax + b$
time = $an + b$
$t \propto O(n)$

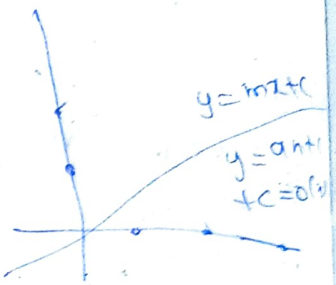
If the array is sorted and to find Largest and smallest

$TC = O(1)$



$y = const$

$TC = O(1)$



$TC = O(n)$

$TC = O(1)$

a

Always think of the worst case of time Complexity so that input size is large "

Big O Notation :- (O) Important worst

Big O → deals with the upper bound of a function

program → $O(n^2)$

$O(1)$
$O(n)$
$O(\log n)$
$O(n \log n)$

ex :-

Time ⇒ $an^2 + bn + c$

1) $n^2 + n + 1$

2) n^2

Step 1 : ignore constant

Step 2 : take largest term

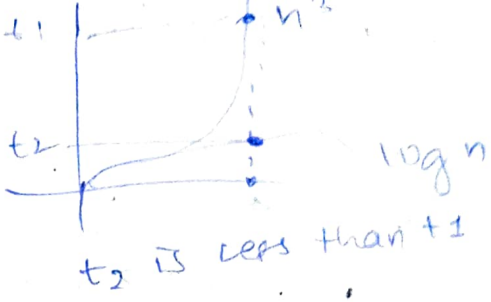
$TC = O(n^2)$

Time = $an^3 + b\log n + c$

$\Rightarrow n^3 + \log n$

largest

TC = $O(n^3)$



t_2 is less than t_1

Cormen book about Time complexity.

Time $\Rightarrow f(n)$

$f(n) =$ [ ]

$TC = O(g(n))$

$f(n) = O(g(n))$

$$\lim_{n\to\infty} \frac{|f(n)|}{g(n)} < \infty .$$

Big omega Notation :- (Ω) (best)

↓

lower bound

lower bound TC

↓

Best case TC

$\Omega(n^2)$ → will get more than n^2 but not get less than n^2.

$\Omega(n)$ ↓ n^3 n^4 n^5 ✓

$\Omega(1)$ → ✗

Big Theta (Θ) Average.

code

LB = UB

if LB & UB are same for Average

$LB = O(n^2)$

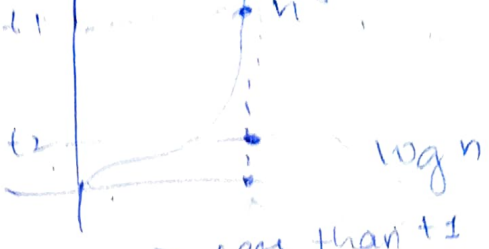
$UB = \Omega(n^2) \Rightarrow \Theta(n^2)$

Time = $an^3 + b\log n + c$

$\Rightarrow n^3 + \log n$

target

TC = $O(n^3)$



t_2 is less than t_1

Cormen book about Time complexity.

Time $\Rightarrow f(n)$

$f(n) =$ [ ]

$TC = O(g(n))$

$f(n) = O(g(n))$

$$\lim_{n\to\infty} \frac{|f(n)|}{g(n)} < \infty$$

Big omega Notation :- (Ω) (best)

↓

(lower bound

Lower bound TC

↓

Best case TC

$\Omega(n^2)$ → will get more than n^2 but not get less than n^2.

$\Omega(n)$ n^3 n^4 n^5

$\Omega(1)$ ↓

X → ✓

Big Theta (Θ) Average.

code

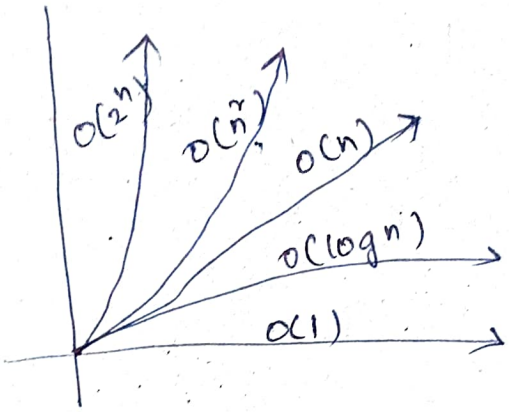
LB = UB

if LB & UB are same for Average

$LB = O(n^2)$

$UB = \Omega(n^2) \Rightarrow \Theta(n^2)$

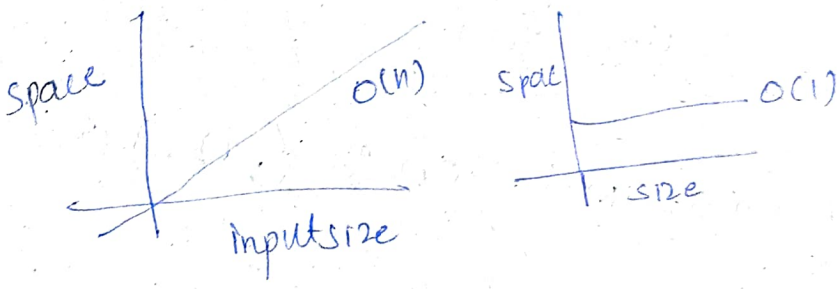
Common Complexities.



" Do not code/use for $O(2^n)$ Time Complexity"

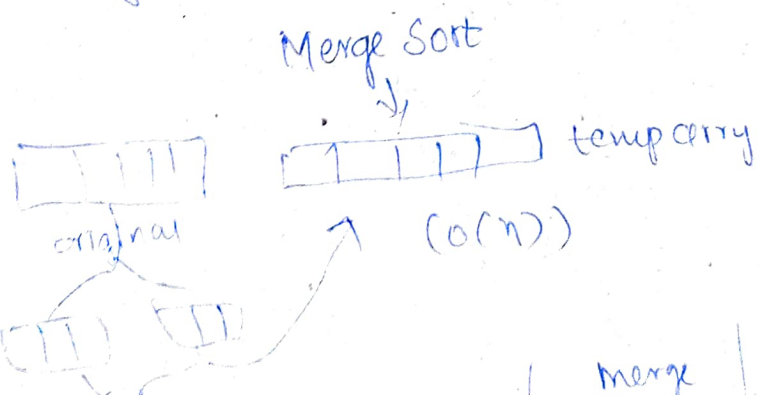
Space Complexity → memory space → heap (objects), stack (function calls)

input space + auxiliary space.



auxiliary → extra array.

Merge Sort



	Merge	Quick
Time complexity	O(n)	O(1)
Space comp	O(nlogn)	O(nlogn)

c in Loops.

mple loop:-

```
for (int i=0 ; i<n ; i++) {
    //some work
    // do constant-work. print("s");
}
```

i = iterator
n = size

constant = k (3, 5, 100 ... 10^5)
constant

→ i = 0 → K x 1
i = 1 → K x 1
i = 2 → K x 1
i = 3 → K
j = n-1 → K
i = n → Break.

I = 0 → n-1

0 → n-1
1 → n
2 → n+1
} all are same n times

$O(n \times K)$ = $O(n)$
Linear time.

i) ignore const
ii) largest term

Nested loops ② —

```
for (int i=0 ; i<n ; i++) {
    for (int j=i+1 ; j<n ; j++) {
        // some constant work
        // is done in loop
    }
}
```

} K

→

i = 0
i = 1
i = 2
break.

j = 1 to n-1 (2) (i++)
j = 2 to 2 → 1
j = 3 to 2 X — 0 times

= 2+1+0 = 3

Nested loop → 2 → total operation

worst outercase × worst innercase

i	j
i = 0	n-1
i = 1	n-2
i = 2	n-3
i = n-1	0

$(n-1)+(n-2)+(n-3)+\ldots+1+0$

0 to n sum $\frac{n(n+1)}{2}$

0 to n-1 → $\frac{n(n-1)}{2}$

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

$= O(n^2)$

Nested loop 2 :-

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

i = 0 to n-1 → n times.

j = 0 to i-1

i = 0	0 time	j = 0 to 0
i = 1	1 time	j = 0 to 1
i = 2	2	j = 0 to 2
i = 3	3	j = 0 to 3
i = 4		
⋮		
i = n-1	(n-1)	

k = 1×k

k = 2k

k = 3k

k = 4k

$k+2k+3k+\ldots+(n-1)k$

$k(1+2+3\ldots n-1)$

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

$= O(n^2)$

outer loop ⟼ $i = 0$ to n.

⟼ inner loop

0 to i

$0 + 1 + 2 + \ldots (n-1)$

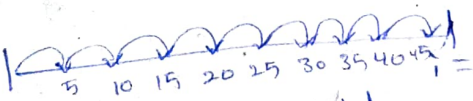
$O(n^2)$

Nested loop 3 (Important)

// some $k < n$

```
for (int i=0 ; i<n ; i = i+k ) {
    for (int j = i+1 ; j<=k ; j++){
        // some const work
```

Let $n = 50$ $k = 5$.



$i = 0$

$j = 0$ ⟼⟼⟼⟼⟼⟼⟼⟼⟼⟼

outerout loop $= \frac{n}{k} = \frac{50}{5} = 10$

Inner loop $= 0 + 0k \Rightarrow k$ jump

$= k$

$\left(\frac{n}{k} \times k \times p\right) = O(np) = 1c$

$= O(n)$

Sorting

Bubble Sort : worst & best case.

```
public static void bubbleSort (int arr[]){
    for(int i=0 ; i<arr.length-1 ; i++){
        for (int j=0 ; j<arr.length-1-i ; j++){
            if (arr[j] > arr[arr[j+1]]){
                //swap:
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
```

worst case

// [5|4|3|2] → [1|2|3|4]

worst case

i=0 (n) j = 0 to n-1
i=1 (n-1) j = 0 to n-2
i=2 (n-2) j = 0 to n-3

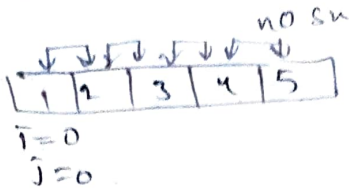
outer loop = n
inner loop = n-i-1

n x K
n-1 x K
n-2 x K
n-3 x K
.
.

K (n+n-1+n-2+...1)
K (O(n²))
= O(n²)

→ n(x)(n) = n²
= O(n²)

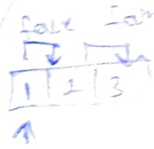
Best case → $O(n^2)$ Best case = $O(n^2)$



i = n
j = n
j × i = n^2

Modified Bubble sort :-

```
p s void. modified BS (int arr[]) {
for (int i=0 ; i< arr.length -1 ; i++)
    boolean swaped = false,
    for (int j=0 ; i< n-1-i ; j++)
        {
        if (arr[j] > arr[j+1]) {
            // swap
        int temp = arr[j]
        arr[j] = arr[j+1]
        arr[j+1] = temp;
            swapped = true; // false
```




```
    }
if (swaped = false) {
    break ; // if we break it will not
                give run n² time
}
}
```

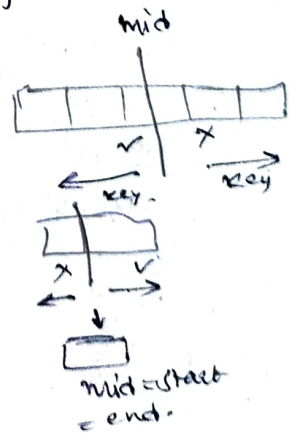
worst case = $O(n^2)$
Best case = $O(n)$

Binary search :-

```
public static int B.S (int arr[], int key) {
    int start = 0;
    int end = arr.length - 1
    while (start <= end)
        int mid = (end + start) / 2;
        // case 1
        if (arr[mid] == key) {
            return mid;
        }
        else if (arr[mid] < key) {
            start = mid + 1;
        }
        else {
            end = mid - 1;
        }
    }
    return -1;
```

Time Complexity :-

Intuitively → logically – practically.



BS

$\left[\frac{n}{2^k}\right] = 1$

$n = 2^k \times p^{(const)}$

$k = \log_2 n \times p^{(const)}$

1 →

2 — $\frac{n}{2}$

3 — $\frac{n}{4}$

$O(\log n)$ worst case

$O(1)$ Best case

	Linear search	Binary search
	$O(n)$	$\log(10^3)$
$N = 1000$	1000 ms.	$= 10$ ms
		≈ 20.
$N = 10^5$	10^5 operation	$\approx \log(10^9)$
		$= 9 \times \log 10$
$N = 10^9$	10^9	$= 9 \times 3$
		$= 27$
		≈ 30

equations

$T(n) = k + n/2$
$T(n/2) \rightarrow k + n/4$
$T(n/4) \rightarrow k + n/8$

k → log n times added.

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

$T(n) = \log n \; k$

$TC = O(\log n)$

Recursion

→ 2 type → linear
→ Divide & conquer.

1) Total workdone = (no of calls * work in each call)

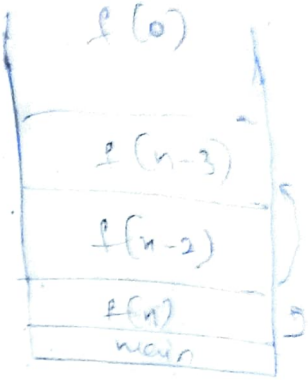
2) recursive recurrence equation

3) Space Complexity = (max depth * memory in each call)

Recursion

Factorial's

```
Public static int fact (int n) {
  if (n==0)
    return 1;
  }
  return n * fact(n-1);
```



Time complexity = no of calls x work

= n x K

= O(n)

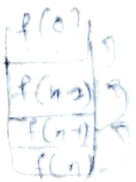
Space complexity = max depth x each length memo

O(n) x K

Sum of n :-

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

f(n) = n + f(n-1)



TC = work done = no of calls * work in each call

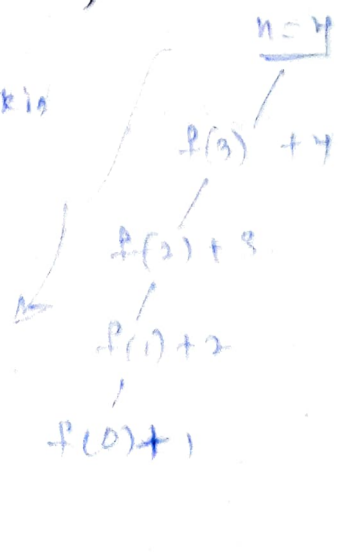
n x K

TC = O(n)

Space Comp:- depth x memory in each level

n x K

SC = O(n)



Fibonacci → Divide & conquer.

```
public class fibon {
    static int fib(int n) {
        if (n==0 || n==1) {
            return n;
        }
        return (fib(n-1) + fib(n-2));
    }
```

recurence equation / relation.

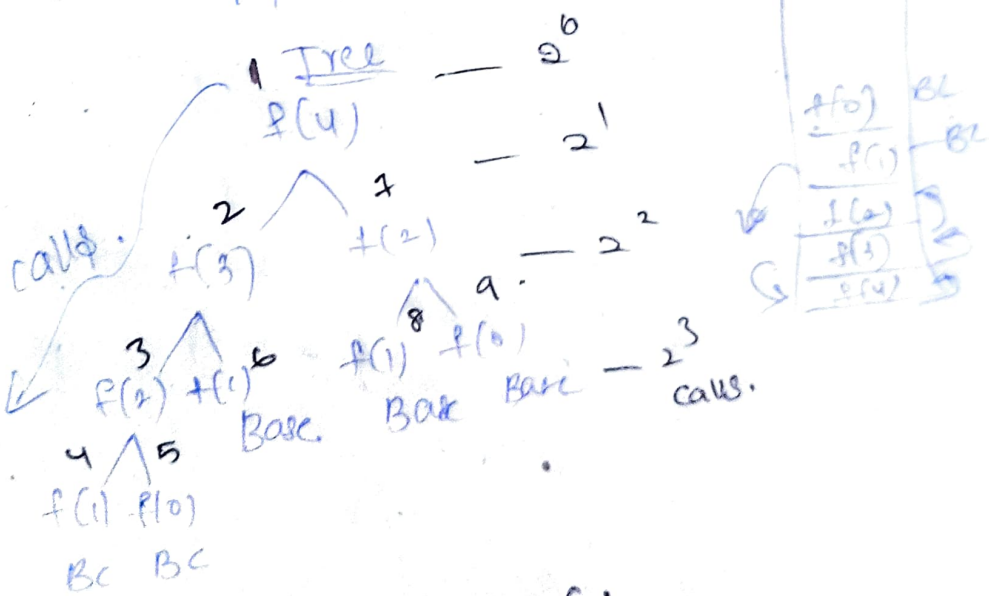
$T(n) = T(n-1) + T(n-2) + k$

$T(n-1) = T(n-2) + T(n-3) + k$

$\vdots$

$T(2) = \underbrace{T(1)}_{k_1} + \underbrace{T(0)}_{k_2} + k$

masters theorem
Golden ratio
& fabonacci



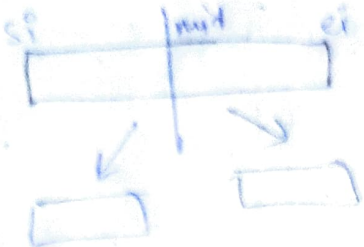
Space Complexity = $n \times O(1)$

$= O(n)$

Time Complexity = 2^n

Merge Sort :-

Divide & conquer



In code:-

while 1
while 2 } $O(n) = n$
while 3

$= O(n) + O(n)$
$= O(2n)$

4 Temp arry } - $O(n)$ TC = $O(n)$
SC = $O(n)$ - Temp array

for merge (function)

---

for (merge sort)

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

$= 2 \cdot T(n/2)$

$T(n) = 2T(n/2) + nK$ → nK

$T(n/2) = 2T(n/4) + n/2\,K$ → nK

$T(n/4)$ → nK

$T(1) = O(1)$

$T(n) = O(1) + (\log n)(nK)$

$n \to \frac{n}{2^0}$

$n/2 = \frac{n}{2^1}$ $\quad \frac{n}{2^x} = 1$

$n/2^2 = \frac{n}{2^2}$ $\quad n = 2^x$

$[\log n = x]$

$1 = \frac{n}{2^x}$

TC = O(1) + log n (n/2)

TC = n log n

(or)

n = 5

f(5) → 1/2^0

f(2) f(2) → n/2^2

f(1) f(1) f(1) f(1) → n/2^3

n/2^k = 1

n = log^k

work done x calls.

log n x n

TC = n log n

Space Complexity = O(n) (temp array)

Reccursion :-

```
public static int pow(int a, int n) {
    if (n==0) {
        return 1;
    }
    return a * pow(a, n-1);
}
```

a^n = math.pow(a, n)

f(a, n) - a^n

a x f(a, n-1) a^{n-1} 1) x f(a, 0) a^0 = 1

a x f(a, n-2) - a^{n-2}

a x f(a, n-3) - a^{n-3}

wp = no of calls x time in each call

wp = n x k

TC = O(n)

SC = calls x space

n x O(1)

each level

SC = O(n)

$a^n = a^{n/2} * a^{n/2}$ (or)

$a^n = a \times a^{n-1}$

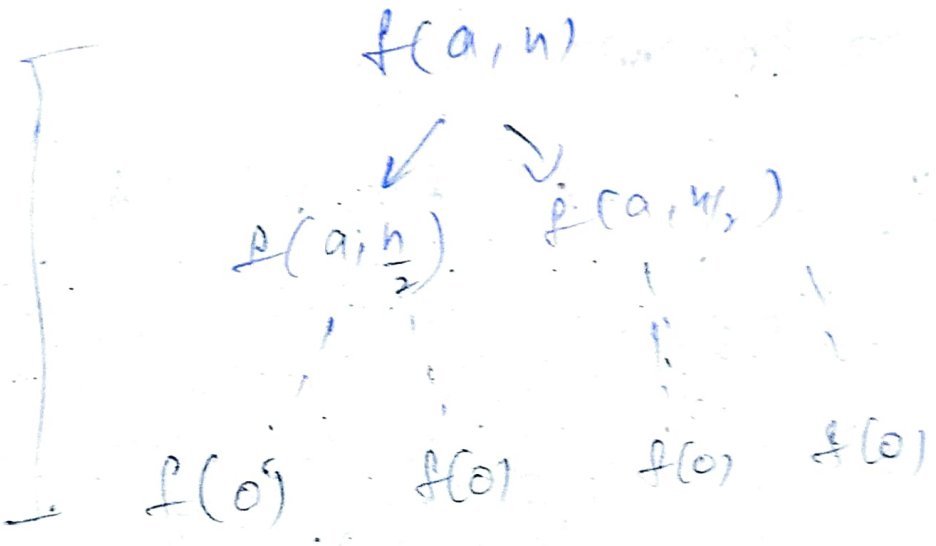
$a^{n-1} = a \times a^{n-2}$

Recursion

Power function 2 :-

```
public static int power2(int a, int n){
    if (n==0){
        return 1;
    }
int halfpowsq = power2(a, n/2) * pow2(a, n/2)

if (n%2 != 0){ // a is odd
    return a * halfpowsq;
}
return halfpowsq;
}
```



$$\frac{n}{2^x} = 1 \Rightarrow 2^x = n$$

$$x = \log_2 n$$

recurrence relation. (or)

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

$$TC = O(n)$$

Power function - 3

Power function 3 (optimised)

```
public static int power(int a, int n) {
    if (n == 0) {
        return 1;
    }
    int halfp = power(a, n/2);
    int hpsq = halfp * halfp;
    if (n % 2 != 0) { // a is odd
        return a * halfpsq;
    }
    return halfpowersq;
```

work done = totcalls x percall

$\log n \times K$

$TC = O(\log n)$

$SC = O(\log n)$

How to approach questions:-

→ 1) Bruteforce (logical all)

↓

optimized (time)

Largest search

→ 1) Brute force

↳ Linear search $O(n)$

↳ Binary search $O(\log n)$

↳ Direct search $(O(1))$

Find the time complexity of the following.

```
int i, j, k = 0.
for (int i = n/2 ; i <= n ; j++) {
    for ( j=2 ; j<=n ; j*2) {
        k = k + n/2 ;
    }
}
```

logn n/2 logn.

TC = O(n logn)

i - runs for n/2 times

j runs for logn times

$\left(\frac{n}{2} \times \log n\right)$ remove const

TC = (n logn)

A) O(n)

~~B)~~ O(n logn)

c) O(n^2)

d) O(n^2 logn)

b) for (~~k=0~~)

```
for (int i=0 ; i<n ; i++)
    i *= k ;
```

TC = (log kn) → Loop run for kn times

A and b have worst case running time of O(n) and O(logn). therefore algorithm B always runs faster than A.

) True

~~) false~~

4)

```
for (int i=0 ; i<n ; ++i)
    for (int j=n ; j>i ; --j) {
        a = a+i+j ;
    }
```

TC = O(n^2)

SC = O(1)

```
class sqrtnum {
    static int flsqrt(int x)
    {
    if (x==0 || x==1)
        return x;
    int i=1, result=1;
    while (result <= x) {
        i++;
        result = i*i;
    }
    return i-1;
    }
public static void main(String[] args)
    {
    int x=11;
    sysо(flsqrt(x));
    }
}
```

Time complexity = $O(\sqrt{n})$

Space complexity = $O(1)$