DATE

180

TIME COMPLEXITY

- What is Time Complexity?

★ Amount of time an algorithm takes to complete as a function of the size of its input.

Case 1: Unsorted

2	10	3	5	7	60

Linear Search (largest)

n ↑ operations ↑

6 6

Worst Case

∴ n ↑ T ↑

T ∝ n

Case 2: sorted, n = 6

1	2	3	4	5	6

Linear Search

Largest = arr [n-1]

n = 1000 ⟶ 1 operation

n = 10^5 ⟶ 1 operation

∴ n ↑ T constant

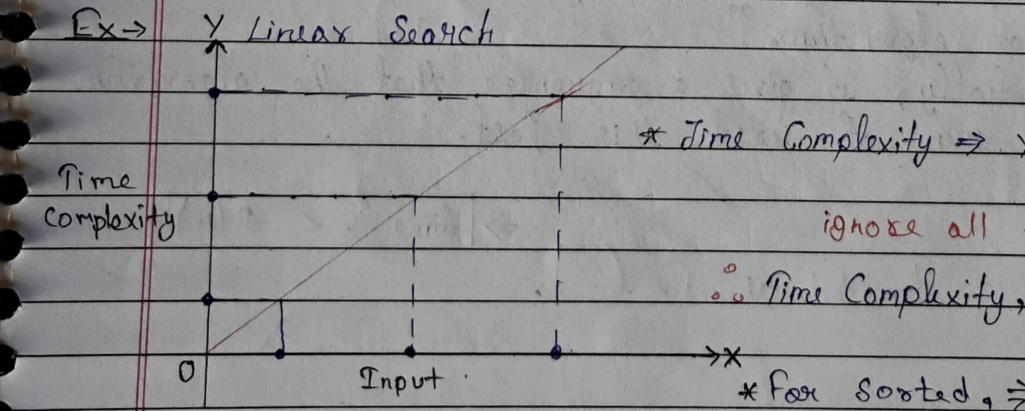
T is constant.

Hence, case 2 is more efficient code.

★ We can find out value of Time complexity by two ways :

1. Experimental Analysis
2. Theoretical Analysis (we will always use this)

Ex→



★ Time Complexity ⇒ $y = ax + b$

$ax + b$

ignore all the constants.

∴ Time Complexity, $O(n)$

★ For sorted, ⇒ $ax + b$ $O(1)$

· parallel to x-axis.

★ BIG O NOTATION

"It provides an upper bound (Worst Case) on the time or space requirements in terms of the input size, denoted as (n)."

*NOTE: We always try to find worst case Complexity.

* time => $ax^2 + bx + c$

$\downarrow$

$\cancel{a}n^2 + \cancel{b}n + \cancel{c}$

$\downarrow$

$n^2 + n + 1$

$\downarrow$

⇒ Time Complexity = $\boxed{O(n^2)}$

★
1. Ignore constants
2. Largest term

⇒ time => $f(n)$

* In mathematical form,

$\because f(n) =$ any function

→ time => $f(n)$ → n = input size

→ $f(n) = O(g(n))$

⇒ $\boxed{\lim_{n \to \infty} \frac{|f(n)|}{g(n)} < \infty}$ → Big O representation in mathematical form.

★ BIG OMEGA NOTATION

"It provides a way to expreses the best case or asymptotic lower bound for the time complexity of an algorithm."

* Essentially, it gives a guarantee that the algorithm will run at least this fast.

⇒ time =< $f(n)$

* It is denoted as $\Omega(\ )$.

★ BIG THETA NOTATION

"It provides a tight bound or average case on the growth rate of a function, meaning it gives you both an upper & a lower bound."

* It is denoted as Θ.

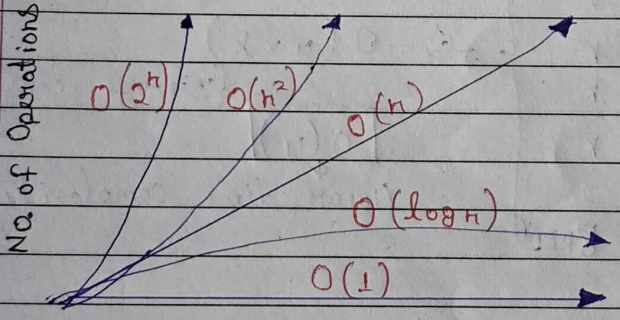
★

1.	BIG O	→ O	→ ~~Best~~ Worst Case
2.	BIG OMEGA	→ Ω	→ Best Case
3.	BIG THETA	→ Θ	→ Average Case

* Big O, Big Omega, Big theta → Tightly Packed

* Small O, small omega, small theta → Loosely Packed

* COMMON COMPLEXITIES



★ SPACE COMPLEXITY

"It refers to the amount of memory space required by an algorithm to run as a function of the size of the input to the algorithm."

* Algorithm / Code → memory / space → heap → objects
  → stack → functions
* Space Complexity → input space + auxiliary space
  (extra) → temporary

* When anyone ask about Space Complexity, we basically tell them Auxiliary Space.

NOTE: We give more priority to Time Complexity on behalf of space.

★ PROBLEMS (Theoretical Analysis)

* Simple Loop

⇒
```
for (int i=0; i<n; i++) {
    // some constant work
    // is done in this Loop.
}
```
→ k (3, 5, 100, 500).

Time Complexity → Operations → Loop run

i = 0 → k
i = 1 → k
i = 2 → k
i = 3 → k
⋮
i = n-1 → BREAK

⇒ O(n × k)

{∵ ignore constants

O(n)

Linear time complexity

* Nested Loop

⇒
```
for (int i=0; i<n; i++) {
    for (int j=i+1; j<n; j++) {
        // some constant work
        // is done in this Loop.
    }
}
```
k

n = 3 ⟹

i	j
0	1 to 2
1	2 to 2
2	3 to 2 X
3	Break.

∴ Time Complexity = Total work performed.

$i=0 \longrightarrow n-1$
$i=1 \longrightarrow n-2$
$i=2 \longrightarrow n-3$
$\vdots$
$i=n-1 \longrightarrow 0$

$\rightarrow (n-1)+(n-2)+(n-2)+\cdots+1+0$

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

$\rightarrow 0$ to $n-1 = \frac{n(n-1)}{2} = \frac{n^2-n}{2}$

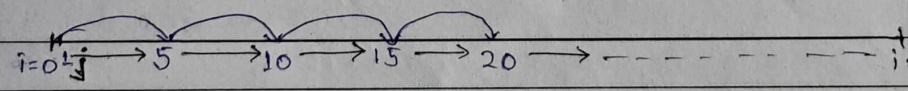
Hence, $\Rightarrow O\left(\frac{n^2}{2} - \frac{n}{2}\right)$ { ∵ ignore constant & largest one will be answer.

$\Rightarrow \boxed{O(n^2)}$

* Nested Loop 2

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

→ Let $n=50$, $k=5$



$\because \frac{n}{k} \times k = \frac{50}{5} \times 5$

$\therefore O\left(\frac{n}{k} \times k \times p\right)$

$O(np)$ { ignore constant

$\Rightarrow \boxed{O(n)}$

K is a constant.

* Bubble Sort

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

Case 1: Worst Case - | 5 | 4 | 3 | 2 | 1 | → | 1 | 2 | 3 | 4 | 5 |

* outer loop → n times kaam कर रहा है।

* inner loop → j = 0 to n-1-i →
  - i=0 → j=0 to n-1 → (n)
  - i=1 → j=0 to n-2 → (n-1)
  - i=2 → j=0 to n-3 → (n-2)
  - i=3 → j=0 to n-4 → (n-3)
  - ⋮
  - i=n → j=0 to n-n → (1)

→ $(n + (n-1) + (n-2) + \dots + 1) \cdot K$

$O(n^2 K)$

⇒ $\boxed{O(n^2)}$

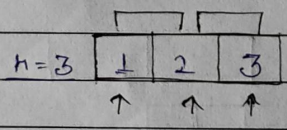
Case 2: Best Case - | 1 | 2 | 3 | 4 | 5 |

$\boxed{O(n^2)}$: because the outer & inner loop will perform & check the array so there is no change in best & worst case.

We can write Optimised code for best case.

* Bubble Sort (optimized code)

```
public static void modifiedBubbleSort (int arr[]) {
    for (int turn = 0; turn < arr.length - 1; turn++) {
        boolean swapped = false;
        for (int j = 0; j < arr.length - 1 - turn; j++) {
            if (arr[j] > arr[j+1]) {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
                swapped = true;
            }
        }
        if (swapped == false) {
            break;
        }
    }
}
```



i = 0

swapped = f

j = 0 1 2

↓

exit loop

check swapped = f

break.

∴ Time Complexity for Best Case $O(n)$

* Binary Search Analysis

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

P

0th → $\frac{n}{1} = n$

1st → $\frac{n}{2^1} = \frac{n}{2}$

2nd → $\frac{n}{2^2} = \frac{n}{4}$

K^{th} iteration → $\frac{n}{2^k} = 1$

$\therefore \frac{n}{2^k} = 1$

$n = 2^k$

$\rightarrow \boxed{k = \log_2 n}$

Loop runs = K^{th} times.

$\therefore$ Time Complexity = $O(K \times P)$ {$\because P \rightarrow$ constant work

$O(\log_2 n \times P)$

$\Rightarrow \boxed{O(\log n)}$

DATE

188

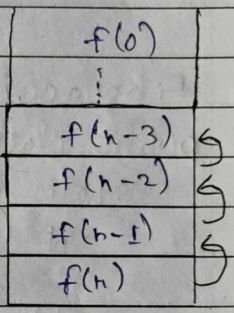
* Recursive Algorithm

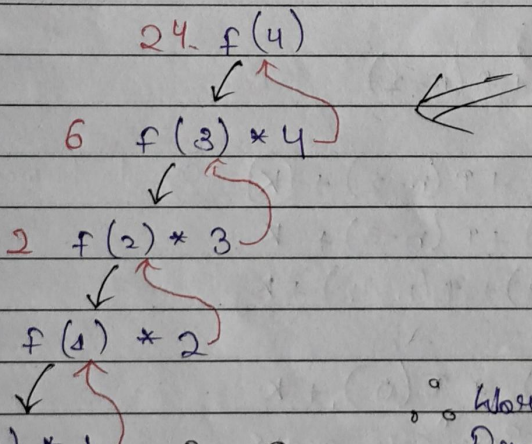
- **Linearly**: $f(n) = \square + f(n-1)$ ...
- **Divide & Conquer** (merge): $f(n) = f\left(\frac{n}{2}\right) + f\left(\frac{n}{4}\right) + \cdots$

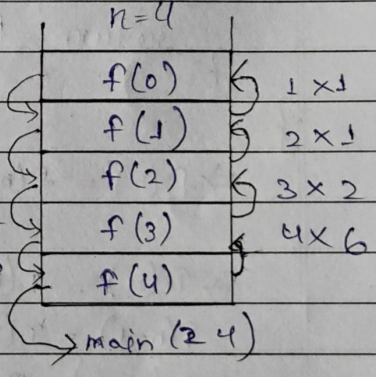
1. Total work done = no. of calls * work in each call
2. Recurrence Equation
3. Space Complexity = (max depth * memory in each call)

⇒ Factorial

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







∴ Work Done = no. of calls * work in each cell
= n * constant (k)
kaam

& Space Complexity = max depth * each level
n * k
⇒ O(n)
SC

O(nk)
TC
⇒ O(n)

⇒ Sum of n Numbers

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

#Same as factorial example.

* Time Complexity, ⇒ O(n)
* Space Complexity, ⇒ O(n)

⇒ Fibonacci

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

⇒ $f(n) = f(n-1) + f(n-2)$

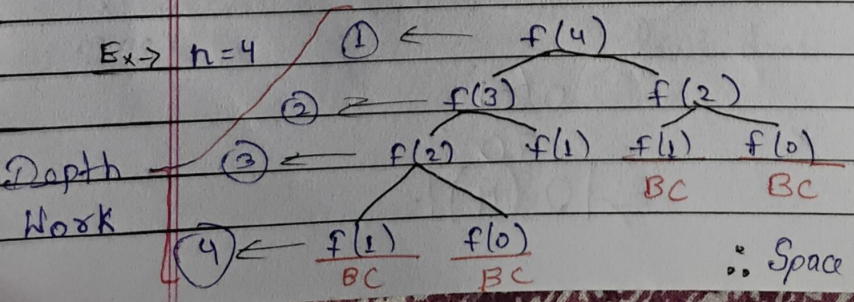
→ $T(n) = T(n-1) + T(n-2) + K$ → Constant work

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

$T(n-2) = T(n-3) + T(n-4) + K$

$T(2) = \underbrace{T(1) + T(0)}_{\text{Base case means } K_1 \& K_2 \text{ constant kaam.}} + K$

Ex→ n=4



∴ Space Complexity, ⇒ O(n)

Time Complexity = Levels ⇒ n

$2 * 2 * 2 * 2 * \ldots * 2$

∴ Time Complexity, ⇒ $O(2^n)$

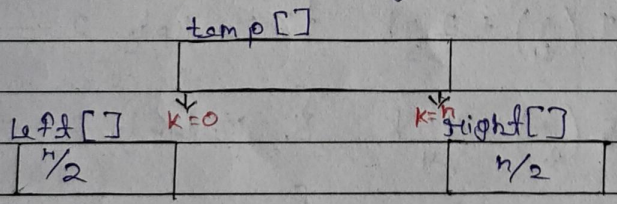
* Merge Sort

→ Recursion

```
public static void mergeSort (int arr[], int si; int ei) {
    if (si >= ei) {
        return;
    }
    int mid = si + (ei - si) /2; //finding mid value
    mergeSort (arr, si, mid); //for left part
    mergeSort (arr, mid + 1, ei); //for right part
    merge (arr, si, mid, ei); //then merge } Time Complexity
}
```

Pg no. 166 & 167

⇒ In the worst case Merge(),



∴ left[] + right[] = $\frac{n}{2} + \frac{n}{2} = n$

∴ The combining all while (3 while Loop) loops the Time Complexity will be $O(n)$

& in merge() of there is a single for loop whose Time Complexity is $O(n)$.

Hence, Time Complexity of merge() in Merge Sort is,

$O(n) + O(n)$

$O(2n)$

$\Rightarrow \boxed{O(n)}$

* My temp[] is an auxiliary[] so the Space Complexity $\Rightarrow \boxed{O(n)}$.

* Recursive function of Merge Sort is,

$\rightarrow f(n) = f(n/2) + f(n/2) + (n+k)$

merge sort — left — right — total kaam

∴ Total time complexity of Merge Sort is,

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

Let's solve.

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

$T(n) = 2T(n/2) + nk \longrightarrow nk$

$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + \frac{nk}{2}$ {∵ Divide 2 $\longrightarrow nk$

$2T\left(\frac{n}{2}\right) = 4T\left(\frac{n}{4}\right) + \frac{2nk}{2}$ {∵ Multiply 2 $\longrightarrow nk$ } ∴

$4T\left(\frac{n}{4}\right) = 8T\left(\frac{n}{8}\right) + \frac{4nk}{4}$ {∵ Multiply 2 $\longrightarrow nk$

$8T\left(\frac{n}{8}\right) = 16\left(\frac{n}{16}\right) + \frac{8nk}{8}$ {∵ Multiply 2 $\longrightarrow nk$ }

⋮

$\Rightarrow T(1) = O(1)$

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

$\downarrow$

$n/2 \to \frac{n}{2^1}$

$\downarrow$

$n/4 \to \frac{n}{2^2}$

$\downarrow$

$n/8 \to \frac{n}{2^3}$

$\vdots$

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

$\frac{n}{2^x} = 1 \implies n = 2^x \implies \log_2 n = x.$

$\therefore T(n) = O(1) + (\log n)(nk)$

ignore constant

Time Complexity = $n \times \log n$

⇒ Time Complexity of Merge Sort, $O(n \log n)$
⇒ Space Complexity of Merge Sort, $O(n)$

⇒ Power Function 1

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

a^n
Math.pow(a,n)

Let's solve the recursive function.

$f(a,n) = a^n$

$\downarrow$

$a * f(a,n-1) = a^{n-1}$

$\downarrow$

$a * f(a,n-2) = a^{n-2}$

$\downarrow$

$a * f(a,n-3) = a^{n-3}$

$\downarrow$

$\vdots$

$(1) * (f(a,0))$

Linearly

∵ Work Done = no. of calls * time in each cells

n * Constant work $O(1)$

* Time Complexity, $\Rightarrow \boxed{O(n)}$

& Space Complexity = no. of calls * ~~हर call में space~~ कितनी occupy होई

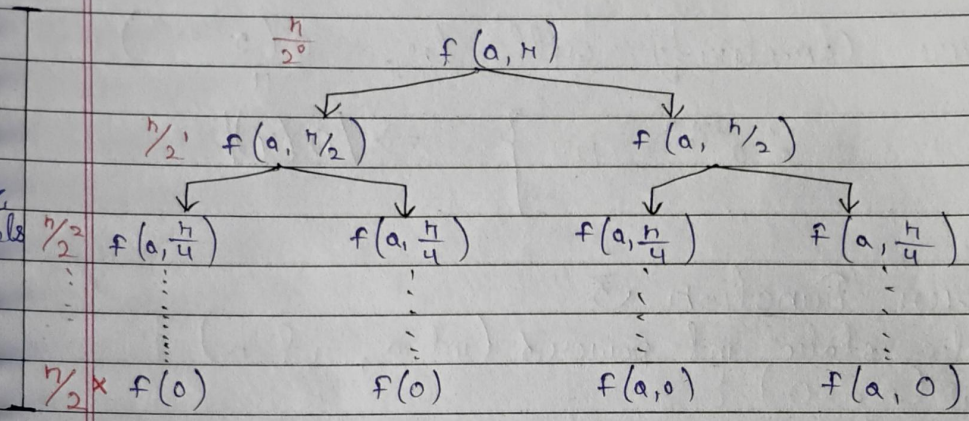
n * $O(1)$ constant space

* Space Complexity, $\Rightarrow \boxed{O(n)}$

⇒ Power Function 2

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

→ Half Power Sq = $a^{n/2} * a^{n/2} = a^{\frac{2n}{2}} = a^n$



$\because \frac{n}{2^x} = 1 \Rightarrow 2^x = n \Rightarrow x = \log_2 n = \log n$

Now, The recursive eq is → $T(n) = T(n/2) + T(n/2) + K$ (constant kaam)

$T(n) = 2 * T(n/2) + K \longrightarrow K \times 2^0$

$2.T(\frac{n}{2}) = 4 * T(\frac{n}{4}) + K.2 \longrightarrow K \times 2^1$

$4.T(\frac{n}{4}) = 8 * T(\frac{n}{8}) + K.4 \longrightarrow K \times 2^2$

$8.T(\frac{n}{8}) = 16 * T(\frac{n}{16}) + K.8 \longrightarrow K \times 2^3$

$T(1) = 2 * T(0) + K \longrightarrow K \times 2^{\log n}$

Base case → O(1)

$T(1) = 2K_2 + K$

$T(n) = K_3 + K(2^0 + 2^1 + 2^2 + 2^3 + \cdots 2^{\log n})$

$T(n) = K_3 + K\left(2\left(\frac{2^{\log n} - 1}{2-1}\right)\right) = K_3 + K\left(2^{\log n + 1} - 2\right)$

$T(n) = K_0 + K \cdot 2^{\log n + 1} - 2K$

∴ Time Complexity will be, $\rightarrow O(2^{\log_2 n})$

$\because a^{\log_a n} = n \Rightarrow \boxed{O(n)}$

⇒ Power Function 3

```
public static int power3 (int a, int n) {
    if (n == 0) {
        return 1;
    }
    int halfPower = power3 (a, n/2); // not constant work
    int halfPowerSq = halfPower * halfPower;
    if (n % 2 != 0) {
        return a * halfPowerSq;
    }
    return halfPowerSq;
}
```

constant work

constant work

```
f(a, n)
  ↓ by 2
f(a, n/2)
  ↓ by 2
f(a, n/4)
  ↓ by 2
f(a, n/8)
  ↓ by 2
  .
  .
  .
f(1)
  ↓
f(0)
```

$\log_2 n$ times

Work done = total calls or levels * kaam per levels

$= \log n * K$

∴ Time Complexity, ⇒ $\boxed{O(\log n)}$

& Space Complexity, ⇒ $\boxed{O(\log n)}$

QUESTIONS

Q1.
```
int i, j, k = 0;
for (i = n/2; i<=n; i++) {
    for (j = 2; j<=n; j = j*2) {
        k = k + n/2;
    }
}
```
→ $i = \frac{n}{2}$ to n, $O(n/2)$ $O(n)$
→ $j = 2$ to n $j = j*2$ $O(\log n)$

$\therefore T(n) = O\left(\sum_{i=n/2}^{n} \sum_{j=2}^{n} 1\right)$
$O(n) \times O(\log n)$
$\Rightarrow \boxed{O(n \log n)}$

∵ There is no additional space acquired.
⇒ Space Complexity, $\boxed{O(1)}$

A. $O(n)$
B. $O(n \log n)$
C. $O(n^2)$
D. $O(n^2 \log n)$

Q2.
```
for (int i=0; i<n; i++) {
    i *= k;
}
```
∴ Space Complexity, ⇒ $\boxed{O(1)}$

A. $O(n)$
B. $O(k)$
C. $O(\log_k n)$ for $(k > 1)$
D. $O(\log_n k)$

Q3. Algorithm A & B have a worst-case running time of $O(n)$ & $O(\log n)$ respectively. Therefore, algorithm B always runs faster than A.

TRUE
FALSE

Q4.
```
int a = 0;
for (int i=0; i<n; ++i)
    for (int j=n; j>i; --j)
        a = a + i + j;
}
```
Time Complexity, ⇒ $\boxed{O(n^2)}$
Space Complexity, ⇒ $\boxed{O(1)}$

Q5.
```
class SqrtNum {
    static int floorSqrt(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;
        System.out.print(floorSqrt(x));
    }
}
```

⇒ Time Complexity $O(n^2)$

⇒ Space Complexity $O(1)$