MODULE 1: BIG-O NOTATION

COMPLEXITY PROBLEMS

1. Introduction to Asymptotic Analysis and Big O:-

The time complexity of an algorithm can be expressed as a polynomial.

Topics:-

1. Asymptotic Analysis
2. Big O Notation
3. Simplified Asymptotic Analysis.
4. A Comparison of Some Common Functions.

(1) Asymptotic Analysis:-

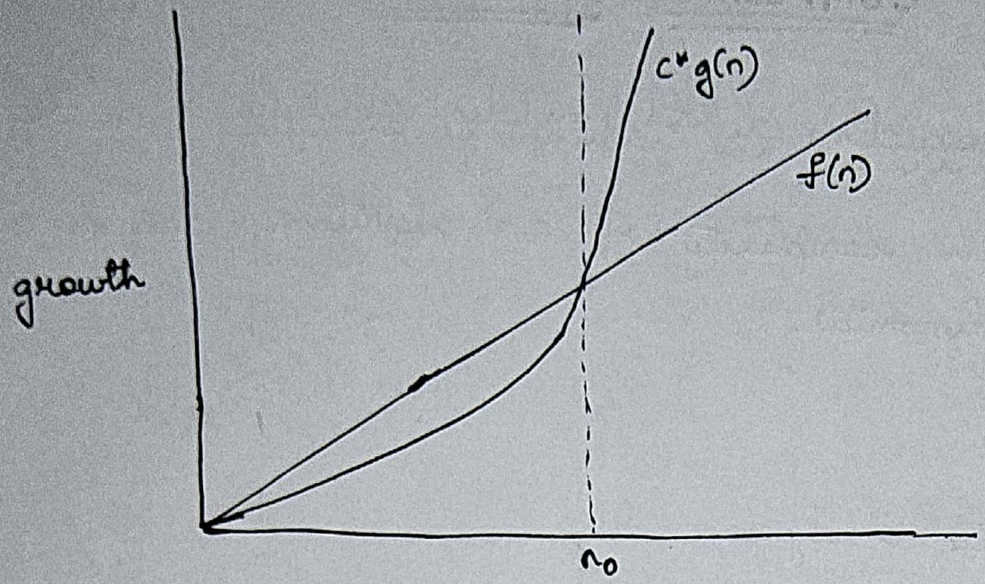
The asymptotic notation compares two functions, say, $f(n)$ and $g(n)$ for very large values of n. This fits in nicely with our need for comparing algorithms for very large input sizes.

(2) Big O Notation:-

Big O is one of the asymptotic notations.

A function $f(n)$ is considered $O(g(n))$, if there exists some positive real constant c and an integer $n_0 > 0$, such that the following inequality holds for all $n \geq n_0$:

$$f(n) \leq c\,g(n)$$



Meaning :-

For very large values of n, f(n) will be at most within a constant factor of g(n). In other words, f(n) will grow no faster than a constant multiple of g(n).

In another words, the rate of growth of f(n) is within constant factors of that of g(n)

Example :-

$f(n) = 3n^3 + 4n + 2$ → algorithm running time.

Verification of time complexity $O(n^3)$:-

Find a positive constant c and an integer $n_0 > 0$, such that for all $n \geq n_0$:

$3n^3 + 4n + 2 \leq cn^3$ →

If we prove that this condition with ~~O(g(n)) i.e.,~~ O cn^3 ($cg(n)$) is true, then the time complexity is $O(g(n)) \Rightarrow O(n^3)$.

$\Rightarrow 3n^3+4n+2 \leq 3n^3+4n^3+2n^3 = 9n^3$ → is also true.

Now we can take $c=9$.

$n_0 \rightarrow$

For what values of n is the inequality $9n^3 \leq cn^3$ satisfied? All of them actually. Therefore $n_0 = 1 (\Rightarrow n \geq n_0 \Rightarrow n \geq 1)$.

solution is: $(c, n_0) = (9, 1)$.

However $(c=9, n_0=1)$ is not unique. We could have picked any value for c that exceeds the coefficient of the highest power of n in $f(n)$. Suppose $c=4$.

$\Rightarrow 3n^3+4n+2 \leq 4n^3$

$n=1 \Rightarrow 3+4+2 \leq 4$ ✗

$n=2 \Rightarrow 24+8+2 \leq 32$ ✗

$n=3 \Rightarrow 81+12+2 \leq 108 \Rightarrow 95 \leq 108$ ✓.

$\therefore n_0 = 3$

$\Rightarrow 3n^3+4n+2 \leq 4n^3$ is true $\forall$ $n \geq 3$.

solution is : $(c, n_0) = (4, 3)$

$\therefore$ The solutions are: $(c, n_0) = (9,1), (4,3)$.

However the time complexity is within $O(n^3)$.

It is not possible to find c and n_0 to show that $f(n) = 3n^3+4n+2$ is $O(n^2)$ or $O(n)$. It is possible in mathematics to show that $f(n)$ is $O(n^4)$ or $O(n^5)$ or higher ~~but in exists~~.

but from a computer science point of view, it is not very useful.

It gives us a loose bound on the asymptotic running time of the algorithm.

When dealing with time and space complexities, we are generally interested in the tightest possible bound when it comes to the asymptotic notation.

Note :-

* Suppose algorithms A and B have running time of $O(n)$ and $O(n^2)$, respectively. For sufficiently large input sizes, algorithm A will run faster than algorithm B. That does not mean that algorithm A will always run faster than algorithm B.

(3) Simplified Asymptotic Analysis :-

Once we have obtained the time complexity of an algorithm by counting the number of primitive operations, we can arrive at the Big O notation for the algorithm simply by :

→ Dropping the multiplicative constants with all terms.
→ Dropping all but the highest order term.

Thus n^2+2n+1 is $O(n^2)$

$n^5+4n^3+2n+43$ is $O(n^5)$.

The constant coefficients have become significant in the Big O notation.

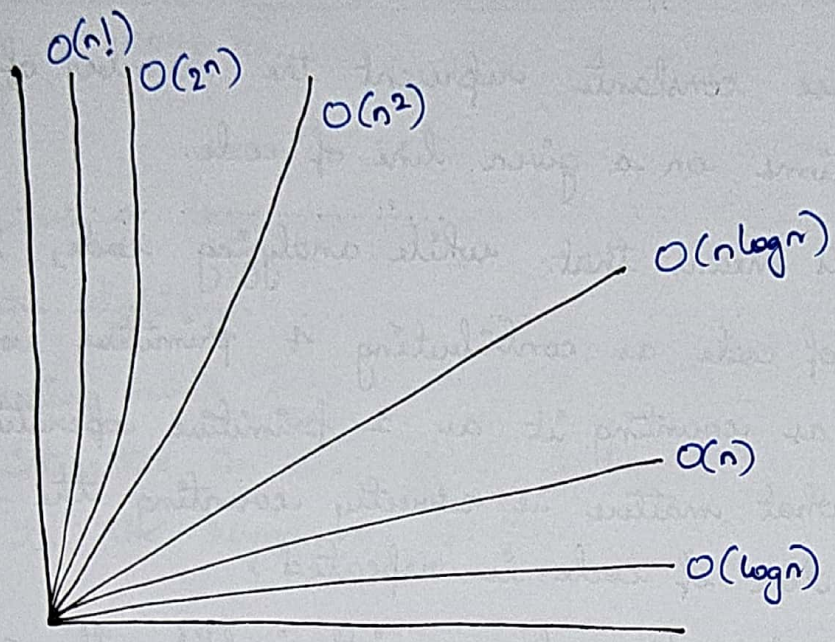
* These constants represent the number of primitive operations on a given line of code.
* This means that while analyzing code, counting a line of code as contributing 4 primitive operations is as good as counting it as 1 primitive operation.
* What matters is correctly counting the number of times each line of code is repeated.
* Moving forward, we will simplify the analysis by just counting the number of executions of each line of code, instead of the number of operations.

Note: $1000000n + 4$ and $10n + 6$ In this case, the constant coefficients do become important. Asymptotically both are $O(n)$. However, worst-case running time for algorithm B is numerically better than A.

(4) A Comparison of Some Common Functions :-

Following lists some commonly encountered functions in ascending order of rate of growth.

Function	Name.	Function	Name.
1. Any constant	Constant	7. n^2	Quadratic.
2. $\log n$.	Logarithmic	8. n^3	Cubic
3. $\log^2 n$.	Log-square.	9. n^4	Quartic.
4. $\sqrt{n}$	Root n.	10. 2^n	Exponential
5. n	Linear	11. e^n	Exponential.
6. $n \log n$.	Linearithmic.	12. $n!$	n-Factorial.



Quiz :-

(1) e^{3n} is in $O(e^n)$

$e^{3n} \leq c e^n$.

$\Rightarrow e^{3n} = e^n \cdot e^{2n} \leq c e^n$.

such a constant c cannot be found.

(2) 10^n is in $O(2^n)$.

$(2 \times 5)^n \leq c\, 2^n$.

$2^n \cdot 5^n \leq c\, 2^n$. $\quad c = 5^n$ ✗ $\quad n_0 = 1$.

no such constant c can be found.

⇑. Useful Formulae :-

* **Formulae :-**

Summation	Equation
$\sum_{i=1}^{n} c = c + c + c + c + \ldots + c$	$\Rightarrow nc$.
$\sum_{i=1}^{n} i = 1 + 2 + 3 + 4 + \ldots + n$	$\Rightarrow \frac{n(n+1)}{2}$.
$\sum_{i=1}^{n} i^2 = 1 + 4 + 9 + \ldots + n^2$	$\Rightarrow \frac{n^2(n^2+1)(2n+1)}{2}$
$\sum_{i=0}^{n} r^i = r^0 + r^1 + r^2 + \ldots + r^n$	$\Rightarrow \frac{r^{n+1}-1}{r-1}$

* **General Tips :-**

(1) Every time a list & array gets iterated over cx length times, it is most likely in $O(n)$ time.

(2) When you see a problem where the number of elements in the problem space gets halved each time, it will most probably be in $O(\log n)$ runtime.

(3) Whenever you have a single nested loop, the problem is most likely in quadratic time.

III. Common Complexity Scenarios :-

* List of Important Complexities :-

① Simple for-loop with an increment of size 1.

```
for (int x=0; x<n; x++) {
    //statement(s) that take constant time
}
```

Running Time Complexity = $T(n) = (2n+2) + cn = (2+c)n+2$

Dropping coefficients $\Rightarrow n+2$

Dropping constants & lower order terms $\Rightarrow O(n)$.

Explanation :

$n \rightarrow [0,1,2,\ldots n-1]$	$\Rightarrow$	n times.
comparison statement $x<n$	$\Rightarrow$	$n+1$ times.
initialization $x=0$	$\Rightarrow$	1 time.
overall for loop	$\Rightarrow$	$2n+2$ times
constant time statements in the loop	$\Rightarrow$	cn times

$\therefore$ Running time complexity $= (2n+2)+cn$

② For-loop with increments of size k.

```
for ( int x=0; x<n; x+=k) {
    //statement(s) that take constant time
}
```

Running Time Complexity = $2 + n\left(\frac{2+c}{k}\right) = O(n)$.

Explanation:

$x \rightarrow [0, k, 2k, \ldots\ldots, nk < n] \Rightarrow$ ~~k~~ time

$\therefore$ $x+=k \Rightarrow floor\left(\frac{n}{k}\right)$ time. $\Rightarrow \frac{n}{k}$ time.

loop no. of times

$x<n \Rightarrow \frac{n}{k}+1$ time.

$x=0 \Rightarrow$ 1 time

$2+\frac{2n}{k}$ times.

statements in the loop $\Rightarrow c \times \frac{n}{k}$ times

Total $\Rightarrow 2+\frac{2n}{k}+\frac{cn}{k}$

$= 2+(2+c)\frac{n}{k}$

$\Rightarrow O(n)$.

③ Simple nested for loop :-

```
for (int i=0; i<n; i++) {
    for (int j=0; j<m; j++) {
        // Statement(s) that take(s) constant time.
    }
}
```

Running time complexity = $nm(2+c) + 2 + 4n = O(nm)$.

Explanation :-

inner loop ⇒ $j<m \rightarrow m+1$ times.

$j++ \rightarrow m$ times

$j=0 \rightarrow 1$ time

$2m+2$ times.

statements $\rightarrow cm$ times.

outer loop ⇒ n times.

Totally ⇒ ~~$(2n+2)(2m+2)$~~ $n(2m+2+cm)$

$i=0 \rightarrow 1$ time.

$i<n \rightarrow n+1$ time.

loop $i++ \rightarrow n$ time

$2n+2$ times.

∴ Running time $= n(2m+2+cm) + 2n + 2$

$= nm(2+c) + 4n + 2$

$= O(nm)$.

④ Nested For-loop with dependent variables :-

```
for (int i=0; i<n; i++) {
    for (int j=0; j<i; j++) {
        //Statement(s) that take(s) constant time
    }
}
```

Running Time Complexity = $O(n^2)$.

Explanation :-

Outer loop runs n times and for each time the outer loop runs, the inner loop runs i times.

$i = 0 \rightarrow$ no statements run in inner loop.

$i = 1 \rightarrow$ c statements/operations

$i = 2 \rightarrow$ 2c

$i = 3 \rightarrow$ 3c

$\vdots$

$i = n-1 \rightarrow (n-1)c$.

statements $\Rightarrow c + 2c + 3c + \dots + (n-1)c = c\left(\frac{n(n-1)}{2}\right)$ times.

$j = 0 \rightarrow n$ times.

$j < i \rightarrow$

$i = 0$	1 time
$i = 1$	2 times
$\vdots$	
$i = n-1$	n times.
total $\rightarrow$	$\frac{n(n+1)}{2}$ times.

$j++ \rightarrow \frac{n(n-1)}{2}$ times

Total inner loop running time = $\frac{c\,n(n-1)}{2} + \frac{n(n+1)}{2} + \frac{n(n-1)}{2} + n$

Outer loop = $2n+2$.

Entire script running time = $2n+2+c\frac{n(n-1)}{2} + \frac{n(n+1)}{2} + \frac{n(n-1)}{2} + n$

$\therefore$ Time complexity = $\underline{O(n^2)}$

(5) Nested for loop with Index Modification :-

```
for(int i=0; i<n; i++) {
    i *= 2.
    for (int j=0; j<i; j++) {
        //Statements that take constant time
    }
}
```

i=2; j=0 → 1
i=6; j=0 → 5
i=14; j=0 → 13
⋮
i=n-1; i=2(n-1)

Running Time Complexity = $O(n)$

Explanation:

Outer Loop	Inner Loop.
$i=0$	$i=0*2=0$.
$i=1$	$i=1*2=2$
$i=3$	$i=3*2=6$
...	...
$i=n-1$	$i=(n-1)*2=2(n-1)$

To simplify, in the outer loop, i is doubled and then incremented each time. If we ignore the increment part, we will be slightly over-estimating the number of

iteration of the outer for loop. That is fine because we are looking for an upper bound on the worst-case running time (Big O).

As i keeps doubling, it will get from 1 to n-1 in roughly $\log_2(n-1)$ steps.

With this simplification, the outer loop index goes (approximately) $1, 2, 4, \ldots 2^{\log_2(n-1)}$ [ignoring increment].
[we ignored the iteration with i=0]

This can also be written as $2^0, 2^1, 2^2, \ldots 2^{\log_2(n-1)}$.

This series also gives the number of iterations of the inner for loop. Thus, the total no. of iterations of the inner loop:

$$2^0 + 2^1 + 2^2 + \ldots + 2^{\log_2(n-1)} = \frac{2^{\log_2(n-1)+1} - 1}{2-1}$$

$$= 2^{\log_2(n-1)} \cdot 2 - 1$$

$$= 2(n-1) - 1$$

inner loop statement iterations $= 2n-3$

∴ Running time of inner for loop is $2(2n-3) + 2 + c(2n-3)$

$= 2n(2+c) - 3c - 4.$

Outer for loop $\Rightarrow 2\log_2(n-1) + 2$

∴ Total running time $= 2n(2+c) - 3c - 4 + 2\log_2(n-1) + 2$

∴ Time complexity $= O(n)$.

Note:- We could have done rough approximation saying that the outer loop runs at most n times, the inner loop iterates at most n times each iteration of the outer for loop. That would lead us to conclude that the total running time is $O(n^2)$. Mathematically that is correct, but it isn't a tight bound.

⑥ **Loops with log(n) time complexity:-**

```
i = // constant
n = // constant
k = // constant
while (i<n) {
        i *= k;
        // Statement(s) that take(s) constant time
}
```

Running Time Complexity:- $\log_k(n) = O(\log_k n)$.

Explanation: A loop statement that multiplies / divides the loop variable by a constant such as the above takes $\log_k(n)$ time because the loop runs that many times.

Example:- $n=16$, $k=2$.

i	Count
1	1
2	2
4	3
8	4
loop exits ← 16	5 ($= \log_2 16 = \log_2 2^4 = 4$)

* CHALLENGE: BIG(O) OF NESTED LOOP WITH ADDITION

Code Snippet :-

```
class NestedLoop {
    public static void main (String[] args) {
        int n = 10;                                   → 1
        int sum = 0;                                  → 1
        double pie = 3.14;                            → 1
        for (int var = 0; var < n; var = var + 8) {   → n/3 times
            System.out.println ("Pie: " + pie);       → n/3
            for (int j = 0; j < n; j = j + 2) {       → n/3 * n/2
                sum++;                                → n/3 * n/2
                System.out.println ("Sum = " + sum);  → n/3 * n/2
            }
        }
    }
}
```

$$1+1+1+1+\frac{n}{3}+1+\frac{n}{3}+\frac{n}{3}+\frac{n}{3}+\frac{n}{3}\left(\frac{n}{2}+1\right)+\frac{n}{3}\cdot\frac{n}{2}+\frac{n}{3}\cdot\frac{n}{2}+\frac{n}{3}\cdot\frac{n}{2}$$

$$= O(n^2)$$

* CHALLENGE: BIG(O) OF NESTED LOOP WITH SUBTRACTION

Code Snippet:-

```
class NestedLoop {
    public static void main (String[] args) {
        int n=10;                                   → 1
        int sum=0;                                  → 1
        double pie = 3.14;                          → 1
        for(int var = n; var >=1; var = var-3) {    → 1, n/3+1, n/3
            System.out.println("Pie: " + pie);      → n/3
            for(int j=n; j>=0; j=j-1) {             → n/3, (n+2)n/3, (n+1)n/3
                sum++;                              → (n+1)n/3
            }
        }
        System.out.println("Sum: " + sum);          → 1
    }
}
```

$$1+1+1+1+\frac{n}{3}+1+\frac{n}{3}+\frac{n}{3}+\frac{n}{3}+(n+2)\frac{n}{3}+(n+1)\frac{n}{3}+(n+1)\frac{n}{3}+1$$

$$= O(n^2).$$

* CHALLENGE: BIG O OF NESTED LOOP WITH MULTIPLICATION

```
class NestedLoop {
    public static void main(String[] args) {
        int n = 10;                          → 1
        int sum = 0;                         → 1
        double pie = 3.14;                   → 1
        int var = 1;                         → 1
        while(var < n) {                     → log2(n)+1
            System.out.println("Pie: " + pie); → log2(n)
            for(int j=0; j<var; j++) {
                sum++;                       → 2^0 + 2^1 + 2^2 + ... + 2^(log2(n))
            }
            var *= 2;                        → log2(n)
        }
        System.out.println("Sum: " + sum);   → 1
    }
}
```

1, 2, $(2)^2$, 2^3, ... $2^n < n$

$m < \log_2 n$

$1 + 2 + 2^2 + \dots + 2^{\log_2(n)}$

$2^{\log_2(n)+1} - 1 = 2(n) - 1$

$= 2n - 1$

$1 + 1 + 1 + 1 + \log_2 n + \log_2 n + 1 + \log_2 n + 2n + 2n - 1 + 2n - 1 + \log_2 n + 1$

$= 4 + 4\log_2 n + 6n$

$= O(n)$.

* CHALLENGE :- NESTED LOOP WITH MULTIPLICATION (BASIC)

Problem Statement:- Compute Big O complexity.

Code Snippet:

```
class NestedLoop {
    public static void main(String[] args) {
        int n = 10;                                   — 1
        int sum = 0;                                  — 1
        double pie = 3.14;                            — 1
        int var = 1;                                  — 1
        while(var < n) {                              → log3 n + 1
            System.out.println("Pie: " + pie);        → log3 n
            for(int j = 1; j < n; j = j + 2) {        → (n/2)(log3 n)
                sum++;                                → (log3 n)(n/2 + 1)
            }                                         → (log3 n)(n/2 + 1)
            var *= 3;                                 → log3 n
        }
        System.out.println("Sum: " + sum);            → 1
    }
}
```

$1+3+3^2+\dots+3^m < n$

$\Rightarrow m = \log_3 n$

$1+2+4+6+\dots+(n*) \quad 2.m <$

$m < \frac{n}{2}$

$$1+1+1+1+\log_3 n+1+\log_3 n+\log_3 n+(\log_3 n)\left(\frac{n}{2}+1\right)+(\log_3 n)\left(\frac{n}{2}\right)+(\log_3 n)\left(\frac{n}{2}+1\right)+\log_3 n+1$$

$$= 6+6\log_3 n+\left(\frac{n}{2}\log_3 n\right)3 = O(n\log n).$$

* CHALLENGE :- NESTED LOOP WITH MULTIPLICATION (INTERMEDIATE)

Code Snippet :-

```
class NestedLoop {
    public static void main (String[] args) {
        int n = 10; — 1
        int sum = 0; — 1
        int j = 1; — 1
        double pie = 3.14; — 1
        for(int var = 1; var < n; var += 3) {
            System.out.println("Pie: " + pie); — (n-1)/3
            j = 1; — (n-1)/3
            while(j < n) {
                sum += 1; — ((n-1)/3)(log3 n)
                j *= 3; — ((n-1)/3)(log3 n)
            }
        }
        System.out.println("Sum: " + sum); — 1.
    }
}
```

for(int var = 1; var < n; var += 3) { — $\frac{n-1}{3}+1 = \frac{n+2}{3}$ — $\frac{n-1}{3}$

$1+4+7+10+\dots+m<n$

$1+(3+1)+(3\cdot2+1)+(3\cdot3+1)+\dots$

$+(3m+1)<n$

$m < \frac{n-1}{3}$

while(j < n) { — $(\log_3 n+1)\left(\frac{n-1}{3}\right)$

$3+3^2+3^3+\dots+3^m < n$

$m < \log_3 n$

$$1+1+1+1+1+\frac{n+2}{3}+\frac{n-1}{3}+\frac{n-1}{3}+\frac{n-1}{3}+\left(\frac{n-1}{3}\right)\left(\log_3 n+1\right)+2\left(\frac{n-1}{3}\right)\left(\log_3 n\right)+1$$

$$= 6 + \frac{4n-4}{3} + (n-1)\left(\log_3 n\right) + \frac{n-1}{3}$$

$$= 6 + \frac{5n-2}{3} + n\log_3 n + \log_3 n \qquad = O(n\log n)$$

* CHALLENGE : NESTED LOOP WITH MULTIPLICATION (ADVANCED)

Code Snippet :-

```
class NestedLoop {
    public static void main(String[] args) {
        int n = 10;                                   — 1
        int sum = 0;                                  — 1
        double pie = 3.14;                            — 1
        for (int var = 0; var < n; var++) {           (1 → n+1 → n)
            int j = 1;                                — n
            System.out.println("Pie: " + pie);        — n
            while (j < var) {                         (2n)
                sum += 1;                             → (2n-1)
                j *= 2;                               → (2n-1)
            }
        }
        System.out.println("Sum: " + sum);            — 1
    }
}
```

j = 1, 2, 4, 8, 16, ... 2^m < var

$m < \log_2^{var}$

$\log_2^{(n)}$

$1 + 2 + 2^2 + 2^3 + \ldots + 2^{\log_2 n}$

$\frac{2^{\log_2 n + 1} - 1}{2 - 1} = 2n - 1$

var 0, j x

var 1, j 1 x — 1 — 2

var 2, j = 1 < 2 — 1 + — 4

var 3, j = 2 — 2 ⋮

var = 4, j = 4 x

$1 + 1 + 1 + 1 + n + 1 + n + n + n + 2n + (2n-1) + (2n-1) + 1$

$4 + 10n$

$= O(n)$

Analysis :-

```
for(int var=0; var<n; var++)          —— n times
    j=1;
    while (j<var) {                   → var times
        num+=1;                       → ∴ O(log2 n)
        j*=2;                             [∵ j*=2].
    }
}.
```

(var < n) } O(n log2 n)

↑ n.

Complexity of inner loop = $O(\log_2 var)$

var varies from: 0 to n-1

$$\Rightarrow \sum_{n=0}^{n-1} O(\log_2 var) = \sum_{n=1}^{n} O(\log_2 var)$$

$$= \log_2 1 + \log_2 2 + \dots + \log_2 (n-1) + \log_2 n$$

$$= \log_2 (1.2.3.\dots.(n-1).n)$$

$$= \log_2 (n!)$$

$\log_2(n!)$ can be unfamiliar. Simplify by replacing $\log_2 1, \log_2 2, \log_2 3, \dots$ $\log_2 n$ with $\log_2 n$

$$\Rightarrow \log_2 n \cdot \log_2 n \dots \cdot \log_2 n = \sum_{k=1}^{n} \log(k) < \sum_{k=1}^{n} \log n = n \log n$$

$$\log_2(n!) \approx n \log n.$$

∴ The programs time complexity, $O(n \log n)$

* CHALLENGE: NESTED LOOP WITH MULTIPLICATION (PRO)

Code Snippet:-

```
class NestedLoop {
    public static void main(String[] args){
        int n=10;          — 1
        int sum=0;         — 1
        int j=1;           — 1
        double pie = 3.14; — 1
        for(int var = 0; var < n; var++) {
            System.out.println("Pie: " + pie);
            while(j< var) {
                sum+=1;
                j*=2;
            }
        }
        System.out.println("Sum: "+sum); — 1
    }
}
```

var: 2, 3, 4, 5, 6
j: 1 ✓ 2 ✓ 4 ✓ 8 ✓

for: 1 → n+1, n+1 → n

while: j < var → n+1; $(\log_2 var)$; n

$2^m < var$

$1+2+2^1+2^2+2^3+\ldots+n$ terms

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

$1+2+2^2+2^3+\ldots+2^m$

$5+1+(n+1)+n+\left(\log_2 n+1\right)n+\log_2 n+\log_2 n$

$= 3n+7+n\log_2 n+2\log_2 n$

$: O(n\log n)$

Analysis :-

```
class NestedLoop {
    public static void main(String[] args) {
        int n = 10;                                   → 1
        int sum = 0;                                  → 1
        int j = 1;                                    → 1
        double pie = 3.14;                            → 1
        for(int var = 0; var < n; var++) {            // var = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
                                                      (1 → n+1 → n)
            System.out.println("Pie: " + pie);        → n.
            while(j < var) {                          // j = 1, 2, 4, 8.
                sum += 1;                             } log n
                j *= 2;                               } log n
            }
        }
        System.out.println("Sum: " + sum);            → 1
    }
}
```

var	j
0	1
1	1
2	2 → 1
3	2
4	2] 2
5	4
6	4
7	4
8	4] 3
9	8

$\log_2 2^3 = 3$.

$5 + n + 1 + n + n + n + 2\log n + 1$

$= 4 + 4n + 2\log n$

$= O(n)$.

$\Rightarrow \log n$ (while loop).

j : 1 to (var − 1) times.
var can be max upto n+1 times.
∴ j : 1 to n times.
complexity = o/loop + i/loop
n+1 + n
= O(n)

PROBLEM SET-1

* If we feed an already sorted array to the following snippet, will the algorithm execute a linear number of instructions? Insertion sort's best case running time is linear (think running a single loop) and not quadratic.

```
void sort(int[] input) {
    for (int i = 1; i < input.length; i++) {
        int key = input[i];
        for (int j = i-1; j >= 0; j--) {
            if (input[j] > key) {
                int tmp = input[j];
                input[j] = key;
                input[j+1] = tmp;
            }
        }
    }
}
```

1 5 9 12 15 20 25

Time complexity is quadratic even for a sorted array.

Explanation:-

The algorithm represents insertion sort. However, the way the code is written, there's no short-circuiting for the nested loop when the values are already sorted so even in the best case, i.e., when the array is sorted the inner loops runs from the start of the array to the end.

Unlike the insertion sort in the lesson, the short-circuiting would never make the inner loop run when the input array

is already sorted. The best-case time would be linear and not quadratic. However, for the above snippet, the best case would still be quadratic.

The point to drive home is to exercise caution as slight implementation changes can change complexities for the same algorithm.

* array size is 5. Determine no. of instructions executed for the best case. The best case happens when the array is already sorted in ascending order.

```
1  for(int i=0; i< input.length; i++) {        i = 0 to 4.
2      int key = input[i];
3      j = i-1;
4      while (j>=0 && input[j] > key) {
5          if (input[j] > key) {
6              int tmp = input[j];
7              input[j] = key;
8              input[j+1] = tmp;
9              j--;
           }
       }
   }
```

(1 → 6 → 5 → 5 → 5 → 4 → 9)

i: (1,0), (2,1), (3,2), (4,3), (0,-1)

o/p: 1 + 6 + 5 + 5 + 5 + 9 = 31.

(1) $1 + n+1 + n + n + n + n + n-1 = 6n+1$.

PROBLEM SET 2 :-

Question 1 :- If someone tells that algorithm has a lower bound of $O(n^2)$. This makes no sense, why?

↳ $O(n^2)$ means Worst Case complexity.

↳ Lower bound of $O(n^2)$ ⇒ saying that lower bound can be at worst be quadratic in complexity.

↳ But by definition, lower bound means best case.

↳ So, the person meant that the algorithm he found in the best case can perform in quadratic time or better.

↳ Which means it can also perform in linear time or better, we just don't know.

↳ Hence, the person is effectively not telling you anything about the lower bound on the performance of his new found algorithm.

Question 2 :- Does $O(2^{2n})$ equal $O(2^n)$?

Time complexity → if $0 \leq f(n) \leq cg(n)$ then $O(g(n))$ is time complexity of $f(n)$.

Let's say $O(2^n) \leq O(2^{2n})$

$$2^n \leq (2^n)^2$$

$$2^n \leq 2^n \cdot 2^n$$

$$2^n \not\leq c2^n \qquad c = 2^n \text{. NOT VALID}$$

$f(n) = 2^n$

$O(g(n))$

$g(n) = O(2^{2n})$

Question 3:- Algorithm whose best case = worst case.

Counting Sort and Radix Sort.

Question 4:- Find time complexity of below.

```
1.  void averager(int[] A) {
2.      float avg = 0.0f;          —— 1
3.      int j, count;              —— 1
4.
5.  O(n) { for(j=0; j< A.length; j++) {   —— 2n+2
6.          avg += A[j];           —— n.
        }
7.      avg = avg/ A.length;       —— 1
8.      count = j = 0;             —— 1
9.      do {
10.         while (j< A.length && A[j] != avg) {   n+1.
11.             j++;                               n.
12.         }
13.         if (j < A.length) {        —— true for case 1, false for case 2.
14.             A[j++] = 0;
15.             count++;
16.         }
17.     } while (j< A.length);   → false for case 2-1
                                 → true for case 1. — O(n).
18. }
```

if avg is float, O(n) for case 2 for case 1, (10-12) will not run.

case 1: if avg occurs in array

case 2: if avg is float & don't appear in array.

Case 2:- $2 + 2n + 2 + n + 2 + n + 1 + n + 1 + 1$

$= 5n + 9. = O(n)$

Case 1:- $1 + 1 + 2n + 2 + n + 1 + 1 + 1 + n + 1 + n$

$= O(n).$

Question 5 :- Complexity ?

```
for(int i=0; i< array.length; i++) {        → 2n+2
    for(int j=0; j<10000; j++) {            → 2n+2
        // some useful work done here       + 10000(2n+2)
    }                                       + (2n+2)10000
}
```

$= 4n+4 + 20002n + 20002$

$+ 20000n + 20000$

$= \underline{O(n)}$

Question 6 :-

```
void complexMethod(int[] array) {
    int n = array.length;                   — 1
    int runFor = Math.pow(-1,n) * Math.pow(n,2);
    for(int i=0; i< runFor; i++) {
        System.out.println("Find how complex I am?");
    }
}
```

n is odd. (eg: 7)

$(-1)^{7} * 7^{2} = -49.$

for loop condition executes only once. $O(1)$

n is even (eg: 6)

runFor $= n^2$.

for loop complexity : $\underline{O(n^2)}$

Question 7:-

```
void complexMethod (int n, int m) {
    for(int j=0; j<n; j++) {
        for(int i=0; i<m%n; i++) {
            System.out.println("");
        }
    }
}
```

m = 7, 8, 9
n = 3
m%n = 1.
m%n = 2
m%n (0, n-1).

Case 1: n > m
Case 2: n = m
Case 3: m > n

O(m*n) [can also say O(n²)]
O(n) {∵ only outer loop}.
∴ inner loop executes from 0 to {1, n-1}.
[∵ m%n range {1, n-1}]

⇒ O(n²)

Question 8:-

```
void someMethod (int n) {
    for(int j=0; j<n; j++) {
        for(int i=0; i<3; i++) {
            for(int i=0; i<n; i++) {
                sout("I have 3 loops");
            }
        }
    }
}
```

3 loops of O(n) complexity.

∴ After including outermost loop. ⇒ O(n²)

Question 9:-

```
void someMethod(int n, int m) {
    for(int j=0; j<n; j++){
        for(int i=0; i<m; i++) {
            System.out.println("I have 2 loops");
        }
    }
}
```

Ans:- O(nm).

PROBLEM SET-3

Question 1 :-

In the lesson on Merge Sort, we implemented merge sort where we divided the array into two parts at each recursion step. Imagine you are asked to implement merge sort by dividing the problem into three parts instead of two. You can assume for simplicity that the input size will always be a multiple of 3. Use a priority queue to merge sub-arrays in the combining step.

a- Provide implementation for the 3-way division merge sort.

Merge Sort :-

Merge Sort is a recursive divide and conquer algorithm that essentially divides a given array into two halves, sorts those halves, and merges them together in order.

6 | 5 | 3 | 1 | 8 | 7 | 2 | 4

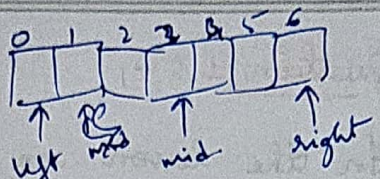
5 6 | 1 3 | 7 8 | 2 4

1 3 5 6 | 2 4 7 8

1 2 3 4 5 6 7 8.

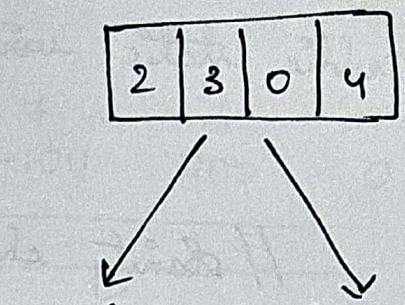
Helper.java

Main.java :-

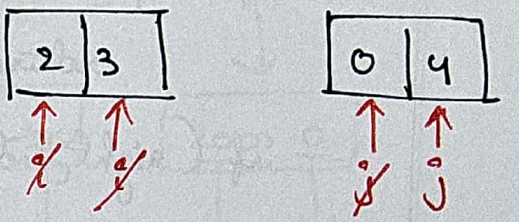


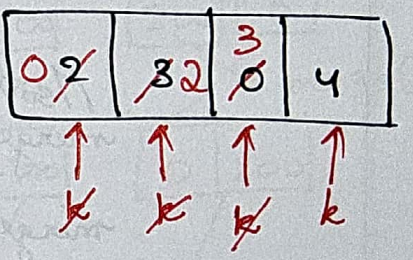

```
class Merge {

    public static void merge(int arr[], int left, int mid,
                             int right) {

        int i, j, k;

        //Initializing the sizes of the temporary arrays
        int sizeLeft = mid - left + 1;
        int sizeRight = right - mid;

        //Initializing temporary arrays
        int leftArr[] = new int[sizeLeft];
        int rightArr[] = new int[sizeRight];

        //Copying given array into the temprary arrays.
        for (i=0; i< sizeLeft; i++)
            leftArr[i] = arr[left+i];
        for (j=0; j < sizeRight; j++)
            rightArr[j] = arr[mid+1+j];

        //Merging the temprary arrays back into the given array
        i=0; //initialize index of first subarray.
        j=0; //Initialize index of second subarray.
        k= left; //initialize index of given array.
```

```
while (i< sizeLeft && j< sizeRight){
    if( leftArr[i] < rightArr[j]) {
        arr[k] = leftArr[i];
        i++;
    } else {
        arr [k] = rightArr[j];
        j++;
    }
    k++;
}

while ( i< sizeLeft) {
    arr[k] = leftArr[i];
    k++;
    i++;
}
while (j < sizeRight) {
    arr [k] = rightArr[j];
    k++;
    j++;
}
}
```



Dividing into two subarrays





l[i] > r[j] ⇒

↳ a[k] = r[j]

j++

k++

l[i] < r[j]

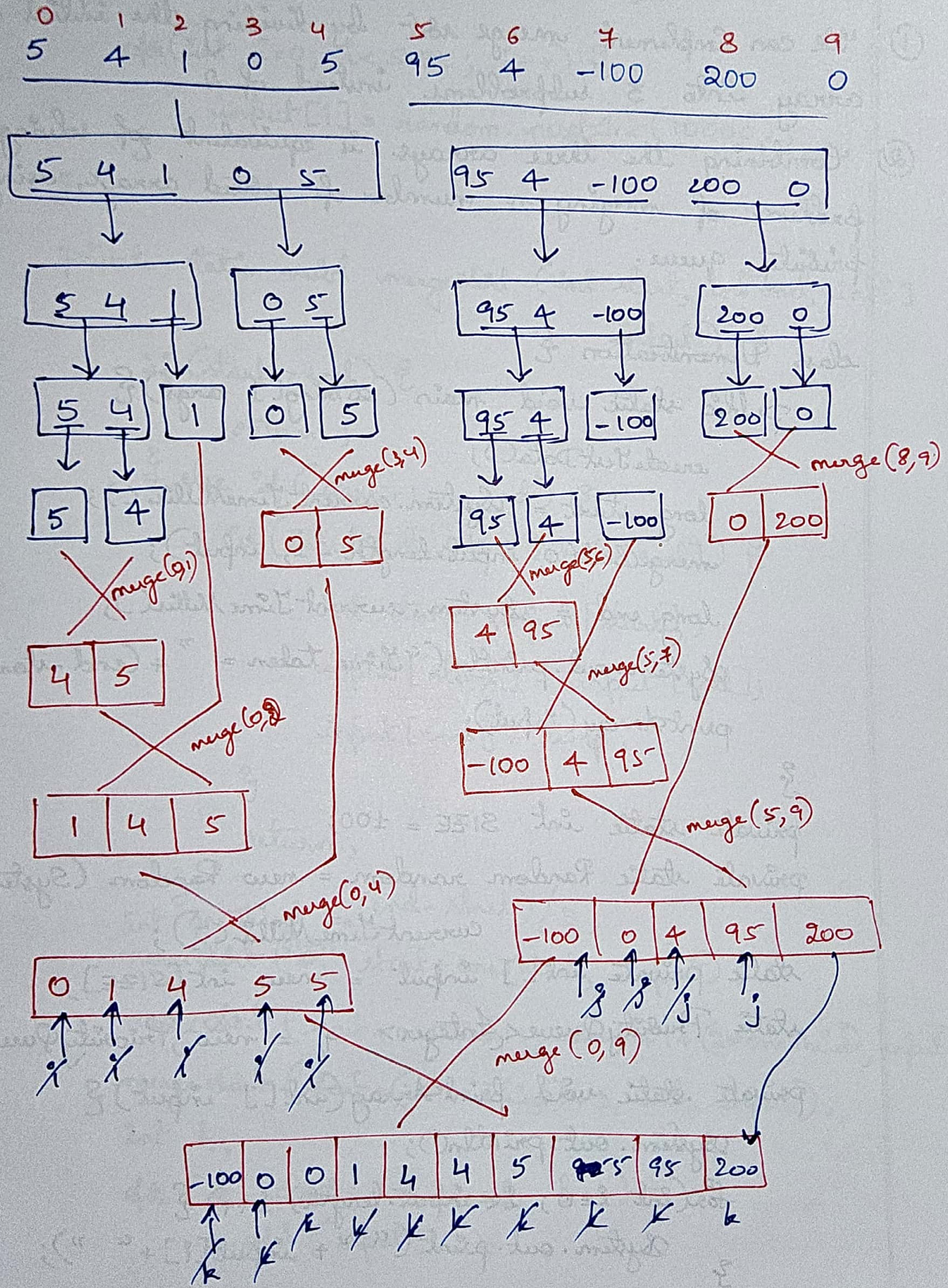
↳ a[k] = l[i]

i++

k++

```
public static void mergeSort (int arr[], int leftIndex,
                              int rightIndex) {
    // Sanity checks
    if (leftIndex < 0 || rightIndex < 0)
        return;
    if (rightIndex > leftIndex) {
        int mid = (leftIndex + rightIndex) / 2 + leftIndex;
        // sorting the first & second halves of the array.
        mergeSort (arr, leftIndex, mid);
        mergeSort (arr, mid+1, rightIndex);

        // Merging the array.
        merge(arr, leftIndex, mid, rightIndex);
    }
}

public static void main (String[] args) {
    int arr[] = { 5, 4, 1, 0, 5, 95, 4, -100, 209, 0 };
    int arrSize = arr.length;
    mergeSort (arr, 0, arrSize-1);
    for(int i : arr) {
        System.out.print (i + " ");
    }
    System.out.println();
}
```

Trace :-



Solution :-

① We can implement merge sort by dividing the initial input array into 3 subproblems instead of 2.

② Combining the three arrays is equivalent of solving the problem of merging n number of sorted arrays, using a priority queue.

```
class Demonstration {
    public static void main (String[] args) {
        createTestData();
        long start = System.currentTimeMillis();
        mergeSort(0, input.length - 1, input);
        long end = System.currentTimeMillis();
        System.out.println("Time taken = " + (end - start));
        printArray(input);
    }
    private static int SIZE = 100;
    private static Random random = new Random(System.
                                   currentTimeMillis());
    static private int[] input = new int[SIZE];
    static PriorityQueue<Integer> q = new PriorityQueue<>(SIZE);
    private static void printArray(int[] input) {
        System.out.println();
        for(int i=0; i< input.length; i++) {
            System.out.print(" " + input[i] + " ");
        }
        System.out.println();
    }
```

```
private static void createTestData() {
    for(int i=0; i< SIZE; i++) {
        input[i] = random.nextInt(10000);
    }
}

private static void mergeSort (int start, int end, int[]
                                        input) {
    if (start > end) {
        return;
    } else if (start+1 == end) {
        if (input [start] > input [end]) {
            int temp = input [start];
            input [start] = input [end];
            input [end] = temp;
        }
        return;
    }
    int oneThird = (end - start) / 3;
    mergeSort (start, start + oneThird, input);
    mergeSort (start + oneThird+1, start + 1 + (2*oneThird), input);
    mergeSort (start + (2 * oneThird) + 2, end, input);
    int k;
    for ( k = start; k <= end; k++) {
        q.add ( input [k]);
    }
    k = start;
```

```
while (!q.isEmpty()) {
        input[k] = q.poll();
        k++;
    }
}
}
```

b- How many recursion levels will be generated?

$x = \log_3 n + 1$.

c- Workout the time complexity for the 3-way merge sort.

The recurrence equation will be given as:

$T(n)$ = cost to divide into 3 subproblems + $3 * T\left(\frac{n}{3}\right)$ + cost to merge 3 subproblems.

Number of recursion levels for the 3-way merge sort will be $\log_3 n + 1$.

Next, we need to determine the cost to merge the three subproblems.

Unlike in traditional merge sort, the 3-way merge sort uses a priority queue to create a min-heap before attempting a merge of the three subproblems.