

Complexity Analysis Problem Set Solutions

Find the Time Complexities of the following snippets of code:

(A)

```
/* Assume that rand() takes constant amount of time */
int a = 0, b = 0;
for (int i = 0; i < N; i++) { // Loop-1
    a = a + rand();
}
for (int j = 0; j < M; ++j) { // Loop-2
    b = b + rand();
}
```

Solution: **Loop-1** runs N times (as ‘*i*’ goes from 0 to $N-1$), and **Loop-2** runs M times (as ‘*j*’ goes from 0 to $M-1$). Now, both **Loop-1** and **Loop-2** run independently, on their own. So, we just simply add the number of times each loop (**Loop-1** & **Loop-2**) runs.

∴ Time Complexity is $O(N + M)$ - Linear in Nature.

(B)

```
int a = 0, b = 0;
for (int i = 0; i < N; i++) { // Loop-1
    for (int j = 0; j < N; j++) { // Loop-1.1
        a = a + j;
    }
}
for (int k = 0; k < N; k++) { // Loop-2
    b = b + k;
}
```

Solution: **Loop-1** is independent in nature and runs N times, and **Loop-1.1** runs N times too. But, for each loop of **Loop-1**, **Loop-1.1** runs N times. ∴ **Loop-1** and **Loop-1.1** combinedly run for $N \times N$ times, *i.e.*, **Loop-1** and **Loop-1.1** run for N^2 times. And now, from **Loop-2** we can see that it individually runs for N times. So, **Loop-2** runs for N times.

∴ Time Complexity is $(N^2 + N)$, which is $O(N^2 + N)$ - Quadratic in Nature.

(C)

```

int a = 0;
for (int i = 0; i < N; i++) { // Loop-1
    for (int j = N; j > i; --j) { // Loop-1.1
        a = a + i + j;
    }
}

```

Solution: **Loop-1** is an independent loop which is linear in nature, *i.e.*, it runs N times (as ‘**i**’ goes from $0 \rightarrow (N-1)$). **Loop-1.1** runs $[N \rightarrow (i+1)]$ times, *i.e.*, **Loop-1.1** is dependent on **Loop-1**. Therefore we analyse this code with the help of the following table and counting total number of iterations::

When ‘i’ is	‘j’ takes the values	#iterations
0	N, N-1, N-2, ..., 1	N
1	N, N-1, N-2, ..., 2	N-1
2	N, N-1, N-2, ..., 3	N-2
3	N, N-1, N-2, ..., 4	N-3
...
N-1	N	1

If we sum up the values in the last column, we have total iterations as:

$$\begin{aligned}
T(N) &= N + (N-1) + (N-2) + (N-3) + \dots + 1 \\
&= 1 + 2 + 3 + 4 + \dots + (N-2) + (N-1) + N
\end{aligned}$$

This is the the Sum of first ‘ N ’ Natural Numbers.

$$T(N) = \frac{N(N+1)}{2} = \frac{N^2+N}{2} = \frac{N^2}{2} + \frac{N}{2}$$

∴ Time Complexity is $O(N^2)$ or, it is Quadratic in Nature.

(D)

```
int a = 0, i = N;  
while (i > 0) {    // Loop-1  
    a += i;  
    i /= 2;  
}
```

Solution: In **Loop-1**, note that **a += i;** statement inside the loop doesn't affect '**i**'. Therefore, we don't care whatever happens to '**a**', as '**a**' is not the variable through which **Loop-1** is controlled.

We only concentrate on the values of '**i**', as the value of '**i**' determines when the loop **Loop-1** ends.

When 'i' is	#iterations	#total iterations
N	1	1
N/2	1	2
N/4	1	3
N/8	1	4
...	...	
N/2 ^K	1	K+1

We know that, **Loop-1** ends when,

i == 0 [as soon as **i** becomes 0, **Loop-1** breaks]

∴ In the last loop **i**'s value is 1. This means that **Loop-1** runs till:

$$\begin{array}{lll} i & = & 1 \\ \text{or, } & N/2^K & = 1 \quad [\because \text{in the last loop, } i = N/2^K] \\ \text{or, } & N & = 2^K \\ \text{or, } & \log_2(N) & = \log_2(2^K) \quad [\text{Applying } \log_2 \text{ on B.S}] \\ \text{or, } & \log_2(N) & = K \quad [\because \log_a(a^K) = K] \end{array}$$

We got, **K = log₂(N)**.

∴ Time Complexity is **K+1=log₂(N)+1**, which is **O(log₂(N))** - Logarithmic in Nature.

(E)

```

void fun(int N, int K) {
    for (int i = 1; i <= N; i++) { // Loop-1
        /* Assume that pow() takes constant amount of time */
        int P = pow(i, K);
        for (int j = 1; j <= P; j++) { // Loop-1.1
            /* Some constant amount of computation */
        }
    }
}

```

Solution: Loop-1 depends on N , and the Loop-1.1 depends on P , which is i^K .

∴ we can say that Loop-1 runs P times, and for each iteration of Loop-1, Loop-1.1 runs i^K times, i.e., we get $N \times i^K$. But, the given input variables are N & K , and our final time complexity should always be expressed in terms of inputs, which is N & K for this problem. So, we have to eliminate i from the expression. To do that, we need to analyse it more.

We have, $P = i^K$, with this value, we cannot get anywhere while analysing the loops. What we can do is, we can assign $K = 1, 2, 3, \dots$ to get the correct time complexity.

For $K = 1$:

When 'i' is	$P = i^1$	'j' takes the values	#iterations
1	$1^1 = 1$	1	1
2	$2^1 = 2$	1, 2	2
3	$3^1 = 3$	1, 2, 3	3
4	$4^1 = 4$	1, 2, 3, 4	4
...
$N-1$	$(N-1)^1 = N-1$	1, 2, 3, ..., $(N-1)$	$N-1$
N	$N^1 = N$	1, 2, 3, ..., N	N

When we summate the last column, our sum will be:

$$\begin{aligned}
 T(N) &= 1 + 2 + 3 + 4 + \dots + (N-1) + N \quad [\text{Sum of first 'N' Natural Numbers}] \\
 &= \frac{N(N+1)}{2} = \frac{N^2+N}{2} = \frac{N^2}{2} + \frac{N}{2}
 \end{aligned}$$

∴ when $K = 1$, the highest degree term we got is $\frac{N^2}{2}$

For K = 2:

When 'i' is	$P = i^2$	'j' takes the values	#iterations
1	$1^2 = 1$	1	1
2	$2^2 = 4$	1, 2, 3, 4	4
3	$3^2 = 9$	1, 2, 3, ..., 9	9
4	$4^2 = 16$	1, 2, 3, ..., 16	16
...
N-1	$(N-1)^2 = N^2 + 2N - 1$	1, 2, 3, ..., $(N^2 + 2N - 1)$	$N^2 + 2N - 1$
N	$N^2 = N^2$	1, 2, 3, ..., N^2	N^2

If we summate the last column, we will have the number of times both **Loop-1** and **Loop-1.1** ran, and that is:

$$\begin{aligned}
 T(N) &= 1 + 4 + 9 + 16 + \dots + (N^2 + 2N - 1) + N^2 \\
 &= 1^2 + 2^2 + 3^2 + 4^2 + \dots + (N-1)^2 + N^2 \\
 &\quad [\text{Sum of Squares of first 'N' Natural Numbers}] \\
 &= \frac{N(N+1)(2N+1)}{6} = \frac{N}{3}(N+1)\left(N+\frac{1}{2}\right) = \frac{N^2+N}{2}\left(N+\frac{1}{2}\right) = \frac{N^3}{3} + \frac{N^2}{2} + \frac{N}{6} \\
 \therefore \text{when } K = 2, \text{ the highest degree term we got is } \frac{N^3}{3}
 \end{aligned}$$

For K = 3:

When 'i' is	$P = i^3$	'j' takes the values	#times Loop-1 & Loop-1.1 runs
1	$1^3 = 1$	1	1
2	$2^3 = 8$	1, 2, 3, ..., 8	8
3	$3^3 = 27$	1, 2, 3, ..., 27	27
4	$4^3 = 64$	1, 2, 3, ..., 64	64
5	$5^3 = 125$	1, 2, 3, ..., 125	125
...
N	$N^3 = N^3$	1, 2, 3, ..., N^3	N^3

If we summate the last column, we will have the number of times both **Loop-1** and **Loop-1.1** ran, and that is:

$$\begin{aligned}
 T(N) &= 1 + 8 + 27 + 64 + \dots + N^3 \\
 &= 1^3 + 2^3 + 3^3 + 4^3 + \dots + (N-1)^3 + N^3 \\
 &\quad [\text{Sum of Cubes of first 'N' Natural Numbers}]
 \end{aligned}$$

$$T(N) = \frac{(N)^2 (N+1)^2}{(2)^2} = \frac{N^2}{4} (N^2 + 2N + 1) = \frac{N^4}{4} + \frac{N^3}{2} + \frac{N^2}{4}$$

\therefore when $K = 3$, the highest degree term we got is $\frac{N^4}{4}$

Now, when:

$$1. \text{ when } K = 1, \text{ the highest degree term we got is } \frac{N^2}{2} = \frac{N^{1+1}}{1+1}$$

$$2. \text{ when } K = 2, \text{ the highest degree term we got is } \frac{N^3}{3} = \frac{N^{2+1}}{2+1}$$

$$3. \text{ when } K = 3, \text{ the highest degree term we got is } \frac{N^4}{4} = \frac{N^{3+1}}{3+1}$$

.....

$$k. \text{ when } K = k, \text{ the highest degree term will be } \frac{N^{k+1}}{k+1}$$

\therefore Time Complexity is $O(\frac{N^{k+1}}{k+1})$.

(F)

```
int count = 0;
for (int i = N; i > 0; i /= 2) { // Loop-1
    for (int j = 0; j < i; j++) { // Loop-1.1
        count += 1;
    }
}
```

Solution: **Loop-1** is non-linear in nature and is an independent loop. **Loop-1.1** is linear in nature and depends on **Loop-1**. Therefore, we need to analyse both **Loop-1** & **Loop-1.1** iteration by iteration, i.e.,

when 'i' is	'j' takes the values	#iterations
N	0, 1, 2, 3, ..., $(N - 1)$	N
$\frac{N}{2}$	0, 1, 2, 3, ..., $(\frac{N}{2} - 1)$	$\frac{N}{2}$
$\frac{N}{4}$	0, 1, 2, 3, ..., $(\frac{N}{4} - 1)$	$\frac{N}{4}$
...
$\frac{N}{2^k}$	0, 1, 2, 3, ..., $(\frac{N}{2^k} - 1)$	$\frac{N}{2^k}$

If we summate the last column, we will have the number of times both **Loop-1** and **Loop-1.1** ran, and that is:

$$\begin{aligned} T(N) &= N + \frac{N}{2} + \frac{N}{4} + \frac{N}{8} + \dots + \frac{N}{2^k} \\ &= N \left(\left(\frac{1}{2}\right)^0 + \left(\frac{1}{2}\right)^1 + \left(\frac{1}{2}\right)^2 + \dots + \left(\frac{1}{2}\right)^{k-1} + \left(\frac{1}{2}\right)^k \right) \quad \text{----- Eqn (1)} \\ &\text{[This is a Geometric Progression]} \end{aligned}$$

$\text{Sum}_{\text{GP}} = \frac{a(1 - r^T)}{1 - r}$, where: a is first term, r is the common ratio & T is number of terms in the GP.

From *Eqn (1)*, we have, $a = (\frac{1}{2})^0 = 1$

$$r = \frac{(\frac{1}{2})^1}{(\frac{1}{2})^0} = \frac{(\frac{1}{2})^2}{(\frac{1}{2})^1} = (\frac{1}{2}) \neq 1. \text{ Therefore, the GP formula can be applied.}$$

$T \Rightarrow$ Number of terms in the GP, i.e., between $(\frac{1}{2})^0 \rightarrow (\frac{1}{2})^k$ there are $(k-0+1)$ terms, i.e., there are $k+1$ terms.

If we substitute the values of a , r & T in Sum_{GP} formula, we have:

$$\begin{aligned} \text{Sum}_{\text{GP}} &= \frac{1(1 - (\frac{1}{2})^{k+1})}{1 - \frac{1}{2}} = \frac{1(1 - (\frac{1}{2})^{k+1})}{\frac{1}{2}} \\ &= 2(1 - (\frac{1}{2})^{k+1}) \quad \text{----- Eqn (2)} \end{aligned}$$

When we substitute the value in *Eqn (2)* in *Eqn (1)*, we have:

$$T(N) = N(2(1 - (\frac{1}{2})^{k+1})) \quad \text{----- Eqn (3)}$$

In *Eqn (3)*, the term $k+1$ needs to be eliminated to get our answer in terms of our input, N . Now, we know that **Loop-1** only runs till $i > 0 \Rightarrow i = 1$ in the final loop. When i has reached the value 1, it means that, **Loop-1** is in its $(\frac{N}{2^k})^{\text{th}}$ iteration. We can use this information and equate i to:

$$\begin{array}{lcl} i & = & 1 \\ \text{or,} & \frac{N}{2^k} & = 1 \\ \text{or,} & N & = 2^k \quad [\text{Apply } \log_2 \text{ on both sides}] \\ \text{or,} & \log_2(N) & = k \quad \text{----- Eqn (4)} \end{array}$$

We can now substitute the value of k from *Eqn (4)* into *Eqn (3)*, i.e.,

$$T(N) = N(2(1 - (\frac{1}{2})^{\log_2(N)+1})) \quad \text{----- Eqn (5)}$$

Now, if we take values for N , that will approximate $T(N)$, i.e.,

$$\begin{aligned} 1. \ N = 2^{30} \Rightarrow T(N) &= N(2(1 - (0.5)^{\log_2(2^{30})+1})) \\ &= N(2(1 - (0.5)^{31})) \\ &= N(2(1 - (4.65 * 10^{-10}))) \\ \Rightarrow T(N) &\approx 2N \end{aligned}$$

$$\begin{aligned}
2. \ N = 2^{60} \Rightarrow T(N) &= N (2(1 - (0.5)^{\log_2(2^{60}) + 1})) \\
&= N (2(1 - (0.5)^{61})) \\
&= N (2(1 - (4.33 * 10^{-19}))) \\
\Rightarrow T(N) &\approx 2N
\end{aligned}$$

In both 1 & 2, we can see that as we increase the value of N , the value reaches closer and closer to $2N$ i.e.,

$$\lim_{N \rightarrow \infty} T(N) = 2N$$

\therefore Time Complexity is $O(N)$ - Linear in Nature

(G)

```

int k = 0;
for(int i = N/2; i <= N; ++i) { // Loop-1
    for (int j = 2; j <= N; j = j * 2) { // Loop-1.1
        k = k + N/2;
    }
}

```

Solution: **Loop-1** is linear in nature and it runs for values $\frac{N}{2} \rightarrow N$, i.e., **Loop-1** runs for $\frac{N}{2} + 1$ times. **Loop-1.1** is non-linear in nature and it is not dependent on **Loop-1**. Therefore, we can deduce the time complexity of **Loop-1.1** separately, i.e.,

‘j’ takes the value	2	4	8	16	...	2^k
---------------------	---	---	---	----	-----	-------

Therefore, in the final loop, ‘j’ takes the value 2^k . **Loop-1.1** ends when ‘j’ reaches $N+1$. We can use this information, and get,

$$\begin{aligned}
j &= N \\
2^k &= N && \text{(read the paragraph above)} \\
k &= \log_2(N) && \text{(Applying } \log_2 \text{ on both sides)} \\
\Rightarrow T(N) &= \log_2(N)
\end{aligned}$$

Loop-1 runs $\frac{N}{2} + 1$ times and for each loop in **Loop-1**, **Loop-1.1** runs $\log_2(N)$ times.

\therefore Time Complexity is $O(N * \log_2(N))$ - Quasilinear in Nature.

(H)

```
int j = 0;
for(int i = 0; i < N; ++i) { // Loop-1
    while(j < N && arr[i] <= arr[j]) { // Loop-1.1
        j++;
    }
}
```

Solution: **Loop-1** is linear in nature and it runs for N times (as **i** takes the values $0 \rightarrow (N-1)$). **Loop-1.1** has a conditional statement **arr[i] <= arr[j]** which, if we assume to be **true**, then, when **i** is 0 in **Loop-1**, **j** reaches N only once in **Loop-1.1**, after that, when **i** goes from $1 \rightarrow (N-1)$, **Loop-1.1**'s conditional statement “**j < N**” is always **false** and **Loop-1.1** doesn't iterate even once.

\therefore Time Complexity is $O(N)$ - Linear in Nature.

Some Important Points to Note

- When a loop is **linear in nature**, it means that the loop variable increases/decreases by only a **single unit** in every iteration of the loop.

Example-1: **for(int i = 0; i < N; ++i)** This loop is linear in nature as **i** increases from $0 \rightarrow (N-1)$ in steps of 1 each.

Example-2: **for(int i = N; i > 0; --i)** This loop is linear in nature as **i** decreases from $N \rightarrow 1$ in steps of 1 each.

Example-3: **for(int i = N; i > 0; i /= 2)** This loop is non-linear in nature as **i** decreases in divisions of 2 after every iteration.

Practice Problem: Match the following Time Complexities

- | | | |
|------------------------|--|-------|
| (1) Linear | (A) N^{K+G} | (4) |
| (2) Logarithmic | (B) $5^{N \times 2}$ | (3) |
| (3) Exponential | (C) $\frac{N}{4} \log_2(\frac{N}{4000})$ | (5) |
| (4) Polynomial | (D) $3^{20}N + 10^5$ | (1) |
| (5) Log Linear | (E) $10N + 9\frac{N}{100} + 340N^2$ | (6) |
| (6) Quadratic | (F) $10^3 \log_2(N+3N)$ | (2) |