TUTORIAL

BY MAI ELBAABAA

Exercise 2

Write True or False :

$$T(n) = 5n^3 + 2n^2 + 4 \log n$$

1. $T(n) \in O(n^4)$
2. $T(n) \in O(n^2)$
3. $T(n) \in \Theta(n^3)$
4. $T(n) \in O(\log n)$
5. $T(n) \in \Theta(n^4)$
6. $T(n) \in \Omega(n^2)$

- Rules of thumb
- in describing the asymptotic complexity of an algorithm:
  - If the running time is the sum of multiple terms, keep the one with the largest growth rate and drop the others, since they will not have an impact for large
  - If the remaining term is a product, drop any multiplicative constants

$f(n) \approx 1000 + n + 2n^2 \mapsto f(n) \approx n^2$

$f(n) \approx 2^n + 2^{n+1} \mapsto f(n) \approx 2^{n+1}$

$f(n) \approx 2 + 400n + 2^n \mapsto f(n) \approx 2^n$

$f(n) \approx n + 500 \log n \mapsto f(n) \approx n$

$f(n) \approx 2\sqrt{n} + 500 \log n \mapsto f(n) \approx \sqrt{n}$

$f(n) \approx \sqrt{n} + n + n^3 \mapsto f(n) \approx n^3$

MATH BACKGROUND: EXPONENTS

- Some useful identities:
  - $X^A \cdot X^B = X^{A+B}$
  - $X^A / X^B = X^{A-B}$
  - $(X^A)^B = X^{AB}$
  - $X^N + X^N = 2X^N$
  - $2^N + 2^N = 2^{N+1}$

MATH BACKGROUND: LOGARITHMS

- Logarithms
  - *definition*: $X^A = B$ if and only if $\log_X B = A$
  - *intuition*: $\log_X B$ means:
    "the power X must be raised to, to get B"
  - In this course, a logarithm with no base implies base 2.
    log B means $\log_2 B$
- Examples
  - $\log_2 16 = 4$ (because $2^4 = 16$)
  - $\log_{10} 1000 = 3$ (because $10^3 = 1000$)

- **O(1):** Time complexity of a function (or set of statements) is considered as O(1) if it doesn't contain loop, recursion, and call to any other non-constant time function.
- **O(n):** Time Complexity of a loop is considered as O(n) if the loop variables are incremented/decremented by a constant amount. For example following functions have O(n) time complexity.

```
// Here c is a positive integer constant
for (int i = 1; i <= n; i += c) {
    // some O(1) expressions
}
```

- **$O(n^c)$**: Time complexity of nested loops is equal to the number of times the innermost statement is executed. For example, the following sample loops have $O(n^2)$ time complexity

```
for (int i = 1; i <=n; i += c) {
      for (int j = 1; j <=n; j += c) {
        // some O(1) expressions
      }
  }

  for (int i = n; i > 0; i -= c) {
      for (int j = i+1; j <=n; j += c) {
        // some O(1) expressions
  }
```

- **O(Logn)** Time Complexity of a loop is considered as O(Logn) if the loop variables are divided/multiplied by a constant amount.

```
for (int i = 1; i <=n; i *= c) {
      // some O(1) expressions
  }
  for (int i = n; i > 0; i /= c) {
      // some O(1) expressions
  }
```

- What is the exact runtime and complexity class (Big-Oh)?

```
int sum = 0;
for (int i = 1; i <= N; i += c) {
    sum++;
}
```

- Runtime = N / c = O(N).

```
int sum = 0;
for (int i = 1; i <= N; i *= c) {
    sum++;
}
```

- Runtime = logc N = O(log N).

Call this number of multiplications "*x*".

$2^x = N$

$x = \log_2 N$

```
for (int i = 1; i < n; i = i*2)
{
    // some O(1) operation
}
```

Here loop is running in the range of 1 to n, and the loop variable increases or decreases by a factor of 2 at each step. So we need to count the total number of iterations performed by the loop to calculate the time complexity.

Let's assume the loop will terminate after k steps where the loop variable increases or decreases by a factor of 2. Then 2^k must be equal to the n i.e.

```
for (int i = 1; i < n; i *= k) {
    // some constant time operations
}
```

Here, the loop starts at $i = 1$ and increments by multiplying i by a constant k after each iteration, so the sequence of values for i would be $1, k, k^2, k^3, \ldots$.

The key is to determine how many times the loop executes.

- In each iteration, i is multiplied by a constant factor k.
- The loop stops when i reaches or exceeds n.
- After t iterations, the value of i will be $i = k^t$, where t is the number of iterations.

So, the condition for the loop to stop is:

$$k^t \geq n$$

Taking the logarithm of both sides:

$$t \cdot \log k \geq \log n \qquad O(\log n)$$

EXAMPLE: O(N²)

```
public static void main(String[] args) {
    int n = 100;
    for (int i =1 ; i <= n/3; i++) {
        for (int j = 1; j < n; j=j+4) {
            System.out.println("*");
            break;
        }
    }
}
```

outer loop will run n/3 times
inner loop will run n/4 times
so total time complexity is (n/3)*(n/4)=n²/12=O(n²)

- *What is the exact runtime complexity (Big-Oh)?*

```
int sum = 0;
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= 2*N; j++) {
        sum++;
    }
}
```

Runtime = N · 2N = O(N^2).

```
int sum = 0;
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= i; j++) {
        sum++;
    }
}
```

- *Runtime = N (N + 1) / 2 = O(N^2).*

- **Arithmetic series:** $\sum_{k=1}^{n} k = 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$

```
for i in range(n):
  for j in range(i):
    # some constant time operation
```

- Here, the outer loop performs n iterations, and the inner loop performs i iterations for each iteration of the outer loop, where i is the current iteration count of the outer loop. The total number of iterations performed by the inner loop can be calculated by summing the number of iterations performed in each iteration of the outer loop, which is given by the formula sum(i) from i=1 to n, which is equal to n * (n + 1) / 2. Hence, the total time complex

```
for(i=0;i<n;i++)
{
   for(j=i;j<n;j++)//this for loop is based on previous 'i'
   {
      foo();
   }
}
```

```
i = 0, 1, 2, 3, 4, ... , n
j = i = 0, 1, 2, 3, 4, ... n

first time: i = 0
inner loop will execute from 0 to n ( n times )

second time , i = 1
j = 1, 2, 3, 4, .... , n ( n-1 times)

third time : i = 2
j = 2, 3, 4, ... , n

i = 0 : j = 0 -> n (n)
i = 1 : j = 1 -> n (n-1)
i = 2 : j = 2 -> n (n-2)
i = n - 1 (i < n) : j = (n - 1) -> n ( 1)

sum( 1 + 2 + 3 + 4 + ..... + n) = n*(n + 1)/2 = O(n^2)
```

```
if (value % 2 == 0){
    return true;
  }
  else
    return false;
}
```

Answer: O(1). Constant run time complexity.

Because you're only ever taking one value, there is no "loop" to go through.

```
for (let i=0; i<array.length; i++) {
        if (array[i] === item) {
            return i;
        }
    }
```

Answer: O(n). Linear run time complexity.

HOW TO FIND COMPLEXITY?

Some rules of thumb

Basically just count the number of statements executed:

- If there are only a small number of simple statements in a program — **O(1)**
- If there is a 'for' loop dictated by a loop index that goes up to *n* — **O(*n*)**
- If there is a nested 'for' loop with outer one controlled by *n* and the inner one controlled by *m* — **O(*n***m*)**
- For a loop with a range of values *n*, and each iteration reduces the range by a fixed constant fraction (eg: ½) — **O(log *n*)**