Practice with time complexity

Alaa Sheta

Time complexity: Nested Loop

```
def sum_pairs(arr):
    total = 0                        # Line 1
    for i in range(len(arr)):        # Line 2
        for j in range(len(arr)):    # Line 3
            total += arr[i] + arr[j]  # Line 4
    return total                     # Line 5
```

Time Analysis Table

Line Number	Code	Time Taken (Assume each line takes `c` time)
1	`total = 0`	`c`
2	`for i in range(len(arr)):`	`n * c` (outer loop runs `n` times)
3	`for j in range(len(arr)):`	`n * n * c` (inner loop runs `n` times for each `i`)
4	`total += arr[i] + arr[j]`	`n * n * c` (runs `n` times for each `i` and `j`)
5	`return total`	`c`

Final Time Calculation

- Total Time `T(n)`:

$$T(n) = c + n \times c + n \times n \times c + n \times n \times c + c$$

Simplifying:

$$T(n) = c + n \times c + 2 \times n^2 \times c + c = (2n^2 + n + 2) \times c$$

Simplified Time Complexity

- The time complexity is `O(n^2)`, where `n` is the number of elements in the array.

Matrix Multiplication: Time complexity

```
MatrixMultiply(A, B)
1. Initialize matrix C of size n x n with zeros
2. for i = 0 to n-1:
3.     for j = 0 to n-1:
4.         C[i][j] = 0
5.         for k = 0 to n-1:
6.             C[i][j] = C[i][j] + A[i][k] * B[k][j]
7. return C
```

Line	Operation	Time Complexity
1	Initialize matrix C of size $n \times n$	$O(n^2)$
2	Outer loop: Iterate over i from 0 to $n-1$	$O(n)$
3	Inner loop: Iterate over j from 0 to $n-1$	$O(n)$
4	Initialize $C[i][j] = 0$	$O(1)$
5	Inner loop: Iterate over k from 0 to $n-1$	$O(n)$
6	Update $C[i][j] = C[i][j] + A[i][k] * B[k][j]$	$O(1)$
7	Return matrix C	$O(1)$

Time complexity: sequential search

```python
def sequential_search(arr, key):
    for i in range(len(arr)):     # Line 1
        if arr[i] == key:         # Line 2
            return i             # Line 3
    return -1                    # Line 4
```

Time Analysis Table

Line Number	Code	Time Taken (Assume each line takes `c` time)
1	`for i in range(len(arr)):`	`n * c` (where `n` is the length of `arr`)
2	`if arr[i] == key:`	`n * c` (runs `n` times, checking each element)
3	`return i`	`c` (only executes if the key is found)
4	`return -1`	`c` (executed if the key is not found)

Code complexity

```
for (i = 1; i < n; i = i * 2) {
    // code
}
```

Total Number of Iterations

- The value of `i` follows the sequence: $1, 2, 4, 8, 16, \ldots$

  This means that after k iterations, $i = 2^k$.
- The loop condition is $i < n$, so $\boxed{2^k < n.}$
- Solving for k, we get $k < \log_2(n)$.
- Thus, the number of iterations is $\log_2(n)$.

Overall Time Complexity

- **Overall Time Complexity:** $\boxed{O(\log n)}$

Code complexity

```
for (i = n; i >= 1; i = i / 2) {
    // code
}
```

Total Number of Iterations

- The value of `i` follows the sequence: $n, \frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \ldots$. After k iterations, $\boxed{i = \frac{n}{2^k}}$.
- The loop condition is $i \geq 1$, so $\boxed{\frac{n}{2^k} \geq 1}$.
- Solving for k, we get $2^k \leq n$.
- Taking the logarithm base 2, we get $k \leq \log_2(n)$.
- Thus, the number of iterations is $\log_2(n)$.

Overall Time Complexity

- **Overall Time Complexity:** $O(\log n)$

Code complexity

```
for (i = 0; i * i < n; i++) {
    // code
}
```

Total Number of Iterations

- The loop continues as long as $i^2 < n$.
- The largest value of `i` will be approximately $\sqrt{n}$, because when i is around $\sqrt{n}$, i^2 will be close to n.
- Therefore, the number of iterations is approximately $\sqrt{n}$.

Overall Time Complexity

- **Overall Time Complexity** $\boxed{O(\sqrt{n})}$

Code complexity

```
for (i = 0; i < n; i++) {
    // code
}

for (j = 0; j < n; j++) {
    // code
}
```

Combined Time Complexity

Since these loops are not nested but sequential:

- First Loop: $O(n)$
- Second Loop: $O(n)$

When combined, the overall time complexity is the sum of both loops:

Overall Time Complexity: $O(n) + O(n) = O(n)$

Code complexity

```
for (i = 0; i < n; i = i * 2) {
    p++;
}

for (j = 0; j < p; j = j * 2) {
    // code
}
```

Time Complexity for First Loop:

$p = O(\log n)$

Total Iterations:

$\log_2(p)$ where p is approximately $\log_2(n)$, so this becomes:

$\log_2(\log_2(n))$

Time Complexity for Second Loop:

$O(\log \log n)$

Exercises

Compute the time complexity

```cpp
void fun(int n) {
    int count = 0;
    for (int i = n / 2; i <= n; i++) {           // Outer loop
        for (int j = 1; j + n / 2 <= n; j++) {  // Middle loop
            for (int k = 1; k <= n; k = k * 2) { // Inner loop
                count++;  // Increment count
            }
        }
    }
    cout << "Count: " << count << endl;
}
```

Time Complexity Calculation:

Let's break down the time complexity for each loop.

1. **Outer Loop** (`for (i = n / 2; i <= n; i++)`):
    - The loop runs from `i = n / 2` to `i = n`, inclusive.
    - Therefore, the outer loop runs approximately **n / 2** times.

2. **Middle Loop** (`for (j = 1; j + n / 2 <= n; j++)`):
    - This loop runs as long as `j + n / 2 <= n`.
    - This means `j` runs from 1 to `n / 2`, so the middle loop also runs **n / 2** times.

3. **Inner Loop** (`for (k = 1; k <= n; k = k * 2)`):
    - The value of `k` starts at 1 and doubles each time (`k = k * 2`).
    - The number of iterations for this loop is logarithmic, i.e., **O(log n)**. The loop runs until `k` exceeds `n`, so it runs **$\log_2(n)$** times.

```cpp
void fun(int n) {
    int count = 0;
    for (int i = n / 2; i <= n; i++) {          // Outer loop
        for (int j = 1; j + n / 2 <= n; j++) {  // Middle loop
            for (int k = 1; k <= n; k = k * 2) { // Inner loop
                count++;  // Increment count
            }
        }
    }
    cout << "Count: " << count << endl;
}
```

j <= n - n/2

j <= n/2

Total Time Complexity:

To calculate the total time complexity, we multiply the complexities of all loops:

- **Outer loop**: runs **n / 2** times → $\boxed{O(n)}$
- **Middle loop**: runs **n / 2** times → $\boxed{O(n)}$
- **Inner loop**: runs $\boxed{\mathbf{O(\log n)}}$ times

Thus, the total time complexity is:

$$\boxed{O\left(n \times n \times \log n\right) = O(n^2 \log n)}$$

Compute the time complexity

```
void fun(int n) {
    if (n <= 1) return;

    int i, j;
    for (i = 1; i < n; i++) {
        for (j = 1; j <= n; j++) {
            cout << "Hello" << endl;
            break;  // Break immediately after printing "Hello"
        }
    }
}
```

Time Complexity Analysis:

- The outer loop (`for (i = 1; i < n; i++)`) runs **n - 1** times (approximately **O(n)**).
- The inner loop (`for (j = 1; j <= n; j++)`) runs only once for each iteration of the outer loop because of the `break` statement. Even though the condition would allow it to run up to `n` times, the `break` ensures that the inner loop terminates after one iteration.

  Thus, the inner loop effectively contributes **O(1)** to the complexity for each iteration of the outer loop.

Total Time Complexity:

- The outer loop runs **O(n)** times.
- The inner loop runs **O(1)** times for each iteration of the outer loop.

Compute the time complexity

```
for (int i = 3; i < n; i++) {  // Outer loop
    for (int j = 0; j < i; j++) {  // Inner loop
        count++;  // Increment count
    }
}
```

Time Complexity Analysis:

1. **Outer Loop** (`for (i = 1; i < n / 3; i++)`):
   - The outer loop runs from `i = 1` to `i < n / 3`.
   - The number of iterations of the outer loop is approximately **n / 3**, which simplifies to **O(n)**.
2. **Inner Loop** (`for (j = 1; j <= n; j += 4)`):
   - The inner loop starts from `j = 1` and increments by 4 in each iteration until `j > n`.
   - The number of iterations of the inner loop is approximately **n / 4**, which simplifies to **O(n)**.

Total Time Complexity:

To find the total time complexity, multiply the complexity of the outer loop and the inner loop:

$$O\left(\frac{n}{3} \times \frac{n}{4}\right) = O(n \times n) = O(n^2)$$

Compute the time complexity

```python
def fun(n):
    count = 0
    for i in range(3, n):  # Outer loop starts from i=3 and goes up to n-1
        for j in range(i):  # Inner loop runs from j=0 to j=i-1
            count += 1
    return count
```

Time Complexity Analysis:

1. **Outer Loop** (`for (int i = 3; i < n; i++)`):
   - The outer loop runs from `i = 3` to `i = n - 1`.
   - Therefore, the number of iterations of the outer loop is **n - 3**, which is **O(n)**.
2. **Inner Loop** (`for (int j = 0; j < i; j++)`):
   - The inner loop runs `i` times for each value of `i` from the outer loop. So:
     - When `i = 3`, the inner loop runs 3 times.
     - When `i = 4`, it runs 4 times.
     - When `i = 5`, it runs 5 times, and so on up to `i = n - 1`.

3. **Total Iterations:**

- The total number of iterations of the inner loop across all iterations of the outer loop is the sum of the values from `3` to `n - 1`:

$$\text{Total Iterations} = 3 + 4 + 5 + \cdots + (n - 1)$$

- This is an arithmetic series, and the sum of the first `k` integers is given by:

$$\sum_{i=3}^{n-1} i = \frac{(n-1)(n)}{2} - \frac{2(2+1)}{2}$$

- Simplified, this becomes:

$$\text{Total Iterations} = O(n^2)$$

Compute the time complexity

```
// First loop
for (int i = 0; i < n; i++) {
    sum += i;
}

// Second nested loop
for (int j = 0; j < n; j++) {
    for (int k = 0; k < n; k++) {
        sum += j * k;
    }
}

cout << "Sum: " << sum << endl;
return 0;
```

Complexity Analysis

1. **First Loop:**
   - The first loop runs **n** times, contributing **O(n)** to the complexity.
2. **Second Nested Loop:**
   - The nested loop runs **n** times for `j` and **n** times for `k`, resulting in $\mathbf{n^2}$ iterations.
   - This contributes $\mathbf{O(n^2)}$ to the complexity.

Total Complexity

Combining both parts:

- First loop: **O(n)**
- Second nested loop: $\mathbf{O(n^2)}$

Compute the time complexity

```
for(int i = 0; i < 100; i++)
    for(int j = N; j >= 0; j--)
        print("bark");
```

1. **Outer Loop:**
    - The outer loop runs **100** times, which is a constant and does not depend on **N**.
2. **Inner Loop:**
    - The inner loop runs from **N** down to **0**, which means it runs **N + 1** times.

Total Complexity

To find the total number of iterations:

- The outer loop contributes **100** iterations.
- For each iteration of the outer loop, the inner loop contributes **N + 1** iterations.

So, the total number of iterations is:

$$100 \times (N + 1)$$

This simplifies to **O(N)** since the constant factor (100) is ignored in Big O notation.

Compute the time complexity

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

1. **Outer Loop:**
    - The outer loop starts at `i = 0` and increments by `2` each time until it reaches `N`.
    - This means the outer loop runs approximately $\frac{N}{2}$ times (or $O(N)$ in Big O notation).
2. **Inner Loop:**
    - The inner loop starts at `j = 1` and multiplies `j` by `2` each iteration until it reaches or exceeds `N`.
    - The inner loop runs as follows:
        - 1, 2, 4, 8, ..., up to the largest power of `2` less than `N`.
    - The number of iterations for the inner loop can be expressed as $\log_2(N)$ since it doubles `j` each time.

Total Complexity

To find the total number of iterations:

- The outer loop runs $O(N)$ (specifically $\frac{N}{2}$).
- The inner loop runs $O(\log N)$.

Combining both parts:

$$\text{Total Iterations} = O(N) \times O(\log N) = O(N \log N)$$

Final Complexity

Thus, the time complexity of the code is:

$$\mathrm{O}(N \log N)$$