Cheatsheet - Time and Space Complexity

Fabio Lama – fabio.lama@pm.me

1. About

To analyze an algorithm, we must determine its processing and memory requirements. The processing requirement is the time complexity, and the memory requirement is the space complexity.

2. Counting Up Time and Space Units

2.1. Simple Algorithm

Consider:

1. **function** F1(a, b, c)
2. $\quad$ max $= a$
3. $\quad$ **if** $(b >$ max$)$
4. $\quad\quad$ max $= b$
5. $\quad$ **if** $(c >$ max$)$
6. $\quad\quad$ max $= c$
7. $\quad$ **return** max

If we analyze this function one by one:

2.1.1. Step 1.

1. $\quad$ max $= a$

1 memory read (a), 1 memory write (max), or **2 time units**.

2.1.2. Step 2.

1. $\quad$ **if** $(b >$ max$)$

2 memory reads (b, max), 1 comparison, 1 evaluation, or **4 time units**.

2.1.3. Step 3.

1. $\quad\quad$ max $= b$

1 memory read (b), 1 memory write (max), or **2 time units**.

2.1.4. Step 4.

1. $\quad$ **if** $(c >$ max$)$

2 memory reads (c, max), 1 comparison, 1 evaluation, or **4 time units**.

2.1.5. Step 5.

1. $\quad\quad$ max $= c$

1 memory read (c), 1 memory write (max), or **2 time units**.

2.1.6. Step 6.

1. $\quad$ **return** max

1 memory read (max), 1 return, or **2 time units**.

2.1.7. Total

In **total**, this function has a time complexity of **16 time units**. The only variable created is $\max$, so the space complexity is **1 space unit**.

2.2. Complex Algorithm

Consider:

1. **function** F2(A, N, x)
2. $\quad$ **for** $0 \leq i < N$
3. $\quad\quad$ **if** $(A[i] == x)$
4. $\quad\quad\quad$ **return** i
5. $\quad$ **return** -1

A loop is not a single instruction, meaning we need to analyze the inner instructions carefully. If we analyze this function one by one:

2.2.1. Step 1.

1. **for** $0 \leq i < N$

We expand this to:

1. $i = 0$
2. **if** $(i < N)$
3. `<instructions>`
4. $i = i + 1$

Respectively:

1. $i = 0$

1 memory write (i), or **1 time unit**.

1. **if** $(i < N)$

2 memory reads (i, N), 1 comparison, 1 evaluation, or 4 time units. But since this is in a loop, we can consider this as **4*(N+1) time units**. Plus one because it's executed at least once, even if N is 0 (and the if-statement is skipped).

1. $i = i + 1$

1 memory read (i), 1 numerical op (+1), 1 memory write (i), or **3*N time units**. Times N because it's executed in a loop.

For the full time complexity, excluding the inner instructions, we get:

$$
\begin{aligned}
&1 + 4(N + 1) + 3N \\
&= 1 + 4N + 4 + 3N \\
&= 7N + 5
\end{aligned}
$$

In other words, this for loop takes **7N + 5 time units**, excluding the inner instructions.

2.2.2. Step 2.

1. **if** $(A[i] == x)$

3 memory reads (x, i, $A[i]$), 1 comparison, 1 evaluation, or **5*N time units**. Times N because it's executed in a loop.

2.2.3. Step 3.

Only one of the two return statements is executed.

1. **return** i
2. **return** -1

We are going to assume the case where the number is not in the array (worst case), so the second return statement is executed which is **1 time unit**.

2.3. Total

To summarize all the steps:

$$
\begin{aligned}
&7N + 5 \\
&+5N \\
&+1 \\
&= 12N + 6
\end{aligned}
$$

In **total**, this function has a time complexity of **12*N + 6 time units**. This means it's running time depends on the size of the input array. The bigger the array, the longer the runtime. Additionally, we only create one new variable (i), so the space complexity is **1 space unit**.

2.4. Growth of Function

As we have seen, the simple algorithm has a time complexity of **16 time units**. This means that the time complexity is a constant (C_x), regardless of the size of the input.

We can specify this as:

$$T(N) = C_1$$

In comparison, the complex algorithm has a time complexity of **12*N + 6 time units**:

$$T(N) = C_1 N + C_2$$

This means that the time complexity is linearly dependent on the size of the input. Hypothetically, if we had a time complexity of **N^2**, we would have a quadratic time complexity, and so on.

Common time complexities are:

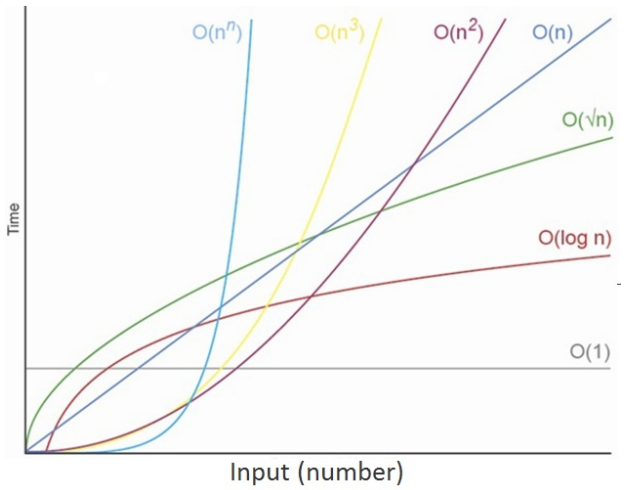

Figure 1. Source: https://www.codeproject.com/articles/1012294/algorithm-time-complexity-what-is-it

2.5. Growth of Function Without Counting

We can calculate the growth of the running time of an algorithm without counting every single time unit.

Consider:

1. **function** SumDiag(A)
2. $\texttt{sum} = 0$
3. $N = \text{length}(A[0])$
4. **for** $(0 \leq i < N)$
5. $\texttt{sum} = \texttt{sum} + A[i, i]$
6. **return** `sum`

If we analyze this function one by one:

2.5.1. Step 1.

1. $\texttt{sum} = 0$

Constant time C_0

2.5.2. Step 2.

1. $N = \text{length}(A[0])$

Here we have to analyze the length function, but let's say we already knew its time complexity of $T(N) = C_1N + C_2$

2.5.3. Step 3.

1. **for** $(0 \leq i < N)$

As we analyzed in the section on counting time units of a for loop, for example $7N + 5$ (excluding inner instructions), we hence know that $T(N) = C_3N + C_4$.

2.5.4. Step 4.

1. $\texttt{sum} = \texttt{sum} + A[i, i]$

Constant time C_5 times N because it's executed in a loop. Respectively: C_5N.

2.5.5. Step 5.

1. **return** `sum`

Constant time C_6.

2.5.6. Total

To summarize all the steps:

$$
\begin{aligned}
&C_0 \\
&+C_1N + C_2 \\
&+C_3N + C_4 \\
&+C_5N
\end{aligned}
$$

$$+C_6$$

$$= T(N) = (C_1 + C_3 + C_5)N + (C_0 + C_2 + C_4 + C_6)$$

To simplify, we can group the constants together:

$$T(N) = C_7N + C_8$$

This means that the algorithm grows linearly with the size of the input.

2.6. Worst and Best Cases

An algorithm can have different time complexities depending on the input. Generally, we're interested in the worst-case scenario, but we can also analyze the best-case scenario.

Consider:

1. **function** $\text{L\_Search}(A, x)$
2. $\quad N = \text{length}(A)$
3. $\quad$ **for** $0 \leq i < N$
4. $\quad\quad$ **if** $(A[i] == x)$
5. $\quad\quad\quad$ **return** i
6. $\quad$ **return** -1

And a given array:

$$A = (13, 8, 2, 24, 5, 17, 6, 9)$$

The **best case** is where the number we're looking for is the first element in the array, respectively $x = 13$. In such a case, the function returns immediately:

$$T(N) = C_1$$

Worst case, the number we're looking for is not in the array at all, for example $x = 7$. In that case, the function checks every element in the array (N amount), eventually returning -1:

$$T(N) = C_1N + C_2$$

We notate this as $\text{L\_Search}(A, 7)$ having a running time of $T(N)aN$ (worst case). Meanwhile, $\text{L\_Search}(A, 13)$ has a running time of $T(N)a1$ (best case).

Last updated 2024-05-21 20:23:48 UTC