CS 119(g) Information, Calcul et Communication

Notes on Computing the Time Complexity of an Algorithm

November 29, 2024

1 Recall

- The resource requirements of an algorithm can be analyzed with respect to time and space, i.e., (i) how long does it take to run this algorithm on a given input and (ii) how much memory/storage we need to run this algorithm on a given input. We focus on the **time complexity.**
- The complexity is always expressed in terms on the size of the inputs, e.g., if an algorithm has two inputs n and m, then the complexity is a function of the size of n and m. For simplicity we focus on functions with **one input**.
- The actually running time of an algorithm can vary a lot depending on the problem instance. For example, suppose that we run a linear search algorithm on a list to find an element x, starting the search from the first element of the list and moving to later ones. The algorithm runtime clearly depends on the position where x is in the list. If x is at the first position in the list, the algorithm will find it immediately. If x is not in the list at all, the algorithm must go through all elements of the list. Therefore, it make sense to analyze algorithms with respect to the best-case, average-case and worst-case input that one can give to them. We focus on the **worst-case behavior,** i.e., the input is chosen such that the algorithm needs the most number of steps (e.g., finding an element in a list that does not contain this element).
- For small input values usually all algorithms are fast. E.g., if you would like to find an element in a list of 10 elements, linear search will need in the worst case 10 comparisons, and binary search will need at most 4 comparisons. However, 10 or 4 comparisons makes little difference given that a modern computer can execute over 10^9 instructions (like a comparison) within a second. However, if the list has 10^9 elements, then linear search will need in the worst case 10^9 comparisons but binary search will need at most 31. Therefore, we are interested in the **asymptotic complexity**, i.e., the complexity when the size of the input goes towards infinity. We use the **Big O** or the **Big Theta (Θ)** notation to express this complexity.

2 Recipe

In this section we present a recipe to compute the asymptotic complexity of simple algorithms. In order to determine the (asymptotic) complexity of an algorithm, we first have to determine for each line l

1. the **cost** c_l of executing Line l, and
2. the **number of repetitions** r_l of Line l (i.e., how many times this line is executed).

Then,

3. we sum the product of costs and repetitions over all lines, i.e., $T(n) = \sum_{l=1\ldots m} c_l \cdot r_l$, where m is the number of lines in the program and n is the size of the input.
4. Finally, we approximate $T(n)$ using the big O notation.

2.1 Cost of a line

The cost of a line depends on our choice of basic operators, which in turn depends on the choice of computing device. E.g., if we use a laptop, addition ($+$) of two numbers represented with 32 bits has a constant cost. If we choose a human as computing device, we could say that our basic operation is adding two single digit numbers, e.g., 5+4, which has constant cost. Adding two numbers with n digits would then have cost linear in n.

Unless stated otherwise we assume that our computing device is a computer with the following basic operations: addition ($+$), subtraction ($-$), multiplication ($*$), division ($/$), assignment ($\leftarrow$), comparisons ($<, \leq, =, \geq, >$), accessing element i in a list ($a[i]$). All these operations are assumed to have cost 1. Using this assumption, Table 1 gives a few examples of source code lines and their corresponding costs. All lines expect the last one (Line 8) have a constant cost, e.g., they are in $O(1)$ (or $\Theta(1)$). The cost of Line 8 depends on the cost of executing the function size on the input l.

Table 1: Example of expressions and their costs

Line	**Expression**	**Cost**
1	$a \leftarrow 0$	1
2	$a \leftarrow a + 2$	2
3	$a + b + c$	2
4	$l[5]$	1
5	$l[a]$	1
6	$l[a] + 4$	2
7	$2 * (l[a] + 4)$	3
8	$2 * \text{size}(l)$	1 + the cost of executing the function size on input l

2.2 Number of repetitions

The number of repetitions of a line depends on its context. If a line is in the body of a loop, then it is executed (in the worst-case) as many times as the loop is executed. Below are a few examples showing how to compute an upper bound on the number of repetitions of a line.

2.2.1 Example 1

Assume we are given a list l with n elements and the following code snippet:

```
1: a ← 0
2: b ← 0
3: for e in l do
4:     a ← e
5:     b ← e + 1
```

Line 1 and 2 are only executed a single time. Line 3-5 are executed as many times as there are elements in the list l, i.e., n times.

2.2.2 Example 2

Assume we are given an integer n and the following code snippet:

```
1: s ← 0
2: for i from 1 to 2 · n do
3:     if i is even then
4:         s ← s + i
```

In this example, Line 1 is again executed only a single time. Lines 2 and 3 are executed for all the numbers from 1 to $2 \cdot n$, e.g., if $n = 3$, then these lines are executed once for the number $1, 2, 3, 4, 5, 6$, which means they are executed 6 times. In the general case, Line 2 and 3 are executed $2 \cdot n$ times. Finally, Line 4 is executed for all the even numbers between 1 and $2 \cdot n$, i.e., if $n = 3$, then Line 4 is executed for the three numbers $2, 4, 6$. So, Line 4 is executed at most $2 \cdot n/2 = n$ times.

2.2.3 Example 3

Assume we are given an integer n and the following code snippet:

1: $i \leftarrow n$
2: **while** $i > 0$ **do**
3: $\quad i \leftarrow i - 3$

Line 1 is executed one time. Line 2 and 3 are executed as many times as one can subtract 3 from n. What is the smallest x such that $n - 3 * x < 0$? We conclude that x is larger than $n/3$. Let us consider several examples for n:

- If $n = 6$, Line 2 is executed with $i = 6, i = 3, i = 0$ (3 times) and Line 3 is executed with $i = 6$ and $i = 3$ (2 times).
- If $n = 7$, Line 2 is executed with $i = 7, i = 4, i = 1, i = -2$ (4 times) and Line 3 is executed with $i = 7$, $i = 4$, and $i = 1$ (3 times).
- If $n = 8$, Line 2 is executed with $i = 8, i = 5, i = 2, i = -1$ (4 times) and Line 3 is executed with $i = 8$, $i = 5$, and $i = 2$ (3 times).

So, Line 2 is executed $\lfloor n/3 \rfloor + 1$ times and Line 3 is executed $\lfloor n/3 \rfloor$ times.

2.2.4 Example 4

Assume we are given an integer n and the following code snippet:

1: $s \leftarrow 0$
2: **for** i from 1 to n **do**
3: $\quad$ **for** j from i to n **do**
4: $\quad\quad s \leftarrow s + i \cdot j$

Line 1 is executed a single time.
Line 2 is executed n times.
In Iteration i of the outer loop (Line 2), Line 3 is executed $(n - i + 1)$ times. So, if we sum over all the iterations of the outer loop, i.e., $\sum_{i=1..n}(n-i+1)$, we obtain the number of times Line 3 and 4 are executed.

$$\sum_{i=1..n} (n - i + 1) = n + \ldots + 1 \;=\; 1 + \ldots + n = \frac{n \cdot (n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$$

2.2.5 Example 5

Assume we are given an integer n and the following code snippet:

1: $s \leftarrow 0$
2: $i \leftarrow n$
3: **while** $i > 0$ **do**
4: $\quad i \leftarrow i/3$ $\quad\quad \triangleright$ where $i/3$ denotes the integer division of i by 3, e.g., $7/3 = 2$ and $2/3 = 0$

Line 1 and 2 are executed one time.
Line 3 and 4 are executed as many times as one can divide n by 3. Note that we use an integer division here, i.e., as soon as the division by 3 is smaller than 1, the result will be 0, and the while loop stops. Therefore, we search for the smallest x such that $n/3^x < 1$. We can multiply this inequality on both sides with 3^x and we get $n < 3^x$, and applying $\log_3$ on both sides gives $\log_3(n) < \log_3(3^x) = x$. So, x has to be larger than

$\log_3(n)$. E.g., if $n = 8$, then Line 3 is executed three times for $i = 8, i = 2$, and $i = 0$ (and Line 4 two times for $i = 8$ and $i = 2$). The same holds for $3 \leq n < 9$. If $n = 9$, then Line 3 is executed four times for $i = 9, i = 3, i = 1$, and $i = 0$ (and Line 4 three times). The same holds for $9 \leq n < 27$.
So, Line 3 is executed $\log_3(n) + 2$ times and Line 4 is executed $\log_3(n) + 1$ times.

2.2.6 Example 6

Assume we are given the following recursive algorithm that takes an integer n as input.

```
1: function ALGOR(n)
2:     if n ≤ 0 then                                              ▷ Cost = 1
3:         return 0                                               ▷ Cost = 1
4:     s ← ALGOR(n − 1)      ▷ Cost = 3 (assignment to s, subtraction of 1, call to a recursive function)
5:     return s + 1                                               ▷ Cost = 2 (addition, return)
```

First, note that the computation for recursive algorithms proceeds slightly differently as for iterative algorithms. In an iterative algorithm we use loops to execute a line more than once. In a recursive algorithm, we use recursive call to execute a line more than once. E.g., in ALGOR above, Line 1 is executed as many times as we call ALGOR recursively. Assume $n = 4$, then we compute ALGOR(4) by calling ALGOR(3), which calls ALGOR(2), which calls ALGOR(1), which calls ALGOR(0). Line 1 is executed in every instance of ALGOR, i.e., it is execute 5 times if $n = 4$. Line 2 is only executed if $n \leq 0$ and therefore only once. Line 4 and 5 are execute as many times as Line 1, which mean $n + 1$ times (the height of the recursive call stack).

Note that using this recipe we denote the cost of calling a recursive function by 1 but the cost of calling an iterative function by the number of steps it takes to execute the iterative functions (cf. Table 1).

The overall complexity of ALGOR (cf. Section 2.3) is

$$T(n) = \sum_{l=2..5} c_l \cdot r_l = 1 \cdot (n+1) + 1 + 1 + 3 \cdot (n+1) + 2 \cdot (n+1) = 6 \cdot (n+1) + 1 = 6 \cdot n + 8$$

2.3 Sum of all lines

Once we have determined the costs c_l and the number of repetitions r_l of each line l in an algorithm, we can compute the overall complexity of the algorithm as the sum of the product of costs and repetitions, i.e.,

$$T(n) = \sum_{l=1..m} c_l \cdot r_l, \text{where } n \text{ is the size of the input and } m \text{ is the size of the algorithm.}$$

Consider the following code snippet taken from Example 2.2.4. The comments on the right show the different costs per line.

```
1: s ← 0                                                          ▷ Cost=1
2: for i from 1 to n do          ▷ Cost=3 (addition of 1, assignment to i, comparison with n)
3:     for j from i to n do                  ▷ Cost=3 (addition, assignment, comparison)
4:         s ← s + i · j          ▷ Cost=3 (multiplication, addition, and assignment)
```

The overall complexity of this code snippet can be computed with the following sum:

$$T(n) = \sum_{l=1..4} c_l \cdot r_l = 1 \cdot 1 + 3 \cdot n + 6 \cdot (\frac{n^2}{2} + \frac{n}{2}) = 1 + 6 \cdot n + 3 \cdot n^2$$

2.4 Approximation with the Big O notation and the Big Theta notation

In the final step of this recipe we approximate the complexity of an algorithm given by $T(n)$.

Therefore, we will use the Big O or the Big Theta notation, which allow on to describe the limiting behavior of a function when the argument tends towards infinity. That implies that we do not differentiate between function $f(n)$ and the functions such as $3 \cdot f(n) + 100$ that differ from $f(n)$ only in the multiplicative factors (such as 3) and additive factors (such as 100 in the previous example). More precisely, we say that $f \in O(g)$ if and only if

$$\exists c > 0 \, \exists n_0 \, \forall n > n_0 : |f(n)| \leq c \cdot g(n).$$

This means that from some point onwards ($n > n_0$, for large enough n), $c \cdot g(n)$ is an upper bound for $f(n)$ for some constant c. For example, the function $f(n) = 4 \cdot n^2 + n + 100$ is in $O(n^2)$ because $\forall n > 11 : f(n) \leq 5 \cdot n^2$ (i.e., $c = 5$ and $n_0 = 11$). Note that $f(n)$ is also in $O(n^3)$ and in $O(2^n)$ because both functions (n^3 and 2^n) grow asymptotically as fast as or faster than $4 \cdot n^2 + n + 100$.

Therefore, we also use the Big Theta Notation (Θ), which bounds the asymptotic behavior of another function from above and below. More precisely, we say that $f \in \Theta(g)$ if and only if

$$\exists c_1 > 0 \, \exists c_2 \, \exists n_0 \, \forall n > n_0 : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n).$$

This means that from some point onwards ($n > n_0$, for large enough n), $g(n)$ is a lower and an upper bound for $f(n)$ for some constants c_1 and c_2.

Table 2 gives a few examples of f and g.

Table 2: Examples of Big O and Big Theta Notation

f(n)	$O(g(n))$	$\Theta(g(n))$
3	$O(1)$	$\Theta(1)$
1000	$O(1)$	$\Theta(1)$
$33 \cdot n + 10$	$O(n)$	$\Theta(n)$
$1 + 6 \cdot n + 3 \cdot n^2$	$O(n^2)$	$\Theta(n^2)$
$-10 \cdot n^5 + \frac{n^3}{5} - 10$	$O(n^5)$	$\Theta(n^5)$
$\frac{4}{3} \cdot 2^n + n - n^2$	$O(2^n)$	$\Theta(2^n)$
2^{n+100}	$O(2^n)$	$\Theta(2^n)$
$2^{3 \cdot n}$	$O(8^n)$	$\Theta(8^n)$
$\log_2(n) + n^2$	$O(n^2)$	$\Theta(n^2)$
$10 + \log_2(3 \cdot n^5)$	$O(\log_2(n))$	$\Theta(\log_2(n))$
$\log_{10}(n)$	$O(\log_2(n))$	$\Theta(\log_2(n))$

Note that 1000 is in $O(1)$ but also in $O(n)$, $O(n^2)$, etc. In order to approximate the complexity of an algorithm, we search for the smallest function g, such that $f \in O(g)$.

Regarding the last line in Table 2 recall the following logarithm rules:

- **Logarithm product rule** $\log_b(x \cdot y) = \log_b(x) + \log_b(y)$
- **Logarithm quotient rule** $\log_b(x/y) = \log_b(x) - \log_b(y)$
- **Logarithm power rule** $\log_b(x^y) = y \cdot \log_b(x)$
- **Logarithm change of base** $\log_b(x) = \log_c(x) \ / \ \log_c(b)$

From the above logarithm rules it follows that $f(n) = 10+\log_2(3\cdot n^5) = 10+\log_2(3)+5\cdot\log_2(n) \in O(\log_2(n))$.

Because of the change of base rule, if $f \in \log_b(x)$ then also $f \in \log_c(x)$ for the other constants c because $\log_c(b)$ is a constant. Thus, the particular base of the logarithm is often not important for the asymptotic complexity.