Introduction to Algorithms

An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. Analyzing an algorithm has come to mean predicting the resources that the algorithm requires. The running time of an algorithm on a particular input is the number of primitive operations or steps executed. We usually distinguish between deterministic and nondeterministic algorithms. In deterministic algorithms, the number of steps needed to solve the problem is predefined and is a function of the problem size. In nondeterministic algorithms, on the other hand, running time of an algorithm depends on both size of the problem and the order in which input data is arranged. In such cases, we analyse an algorithms in terms of best-case running time and worst-case running time. The worst-case running time is an upper bound on the running time for any input. Knowing it gives us a guarantee that the algorithm will never take any longer. The best-case running time, on the other hand, is a lower bound on the running time for any input.

Asymptotic notations

**O-notation.** For a given function \( g(n) \), we denote by \( O(g(n)) \) the set of functions

\[
O(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \}.
\]

**Ω-notation.** Ω-notation provides an asymptotic lower bound. For a given function \( g(n) \), we denote by \( Ω(g(n)) \) the set of functions

\[
Ω(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \}.
\]

**Θ-notation.** For a given function \( g(n) \), we denote by \( Θ(g(n)) \) the set of functions

\[
Θ(g(n)) = \{ f(n) : \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1g(n) \leq f(n) \leq cg_2(n) \text{ for all } n \geq n_0 \}.
\]

We say that \( g(n) \) is asymptotically tight bound of \( f(n) \).
\(o\)-notation. The asymptotical upper bound provided by \(O\)-notation may or may not be asymptotically tight. We use \(o\)-notation to denote an upper bound that is not asymptotically tight. We define \(o(g(n))\) as the set

\[
o(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0 \}.
\]

\(\omega\)-notation. By analogy, \(\omega\)-notation is to \(\Omega\)-notation as \(o\)-notation is to \(O\)-notation. We use \(\omega\)-notation to denote a lower bound that is not asymptotically tight. That is,

\[
\omega(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0 \}.
\]

**Theorem** For any two functions \(f(n)\) and \(g(n)\), \(f(n) = \Theta(g(n))\) iff \(f(n) = O(g(n))\) and \(f(n) = \Omega(g(n))\).

The asymptotic notations have the following properties.

**Transitivity**

- \(\frac{f(n) = \Theta(g(n)) \quad \text{and} \quad g(n) = \Theta(h(n)) \implies f(n) = \Theta(h(n))}{f(n) = O(g(n)) \quad \text{and} \quad g(n) = O(h(n)) \implies f(n) = O(h(n))}. \)
- \(\frac{f(n) = \Omega(g(n)) \quad \text{and} \quad g(n) = \Omega(h(n)) \implies f(n) = \Omega(h(n))}{f(n) = o(g(n)) \quad \text{and} \quad g(n) = o(h(n)) \implies f(n) = o(h(n))}. \)
- \(\frac{f(n) = \omega(g(n)) \quad \text{and} \quad g(n) = \omega(h(n)) \implies f(n) = \omega(h(n))}{f(n) = \omega(g(n)) \quad \text{and} \quad g(n) = \omega(h(n)) \implies f(n) = \omega(h(n))}. \)

**Reflexivity**

- \(f(n) = \Theta(f(n))\)
- \(f(n) = O(f(n))\)
- \(f(n) = \Omega(f(n))\)

**Symmetry**

- \(f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n))\)

**Transpose symmetry**

- \(f(n) = O(f(n)) \iff g(n) = \Omega(g(n))\)
- \(f(n) = o(f(n)) \iff g(n) = \omega(g(n))\)
Problems

1. Is \(2^{n+1} = O(2^n)\)\

Solution. According to the definition of \(O\)-notation there exist positive constants \(c\) and \(n_0\) such that \(0 \leq f(n) \leq cg(n)\), where \(f(n) = 2^{n+1}\) and \(g(n) = 2^n\). Substituing \(f(n)\) and \(g(n)\) by their values, we obtain \(0 \leq 2 \cdot 2^n \leq c2^n\). Let’s divide both sides of the inequality by \(2^n\). Obviously, for any \(c \geq 2\), say for \(c = 2\), and any \(n_0 \geq 1\), say for \(n_0 = 1\), the above inequality holds. This proves the assertion.

2. Is \(2^n = O(3^n)\)?

Solution. Again we will try to prove the statement by definition. One can easily see that \(0 \leq f(n) \leq cg(n)\), where \(f(n) = 2^n\) and \(f(n) = 3^n\). Substituing \(f(n)\) and \(g(n)\) by their values, we obtain \(0 \leq 2^n \leq c3^n\). Now we divide both sides of the inequality by \(3^n\), that is \(0 \leq \left(\frac{2}{3}\right)^n \leq c\). Since \(\frac{2}{3} < 1\), the above inequality stays correct for \(c = \frac{2}{3}\) and for any \(n > n_0 = 1\).

3. Use the following algorithm to sort \(n\) numbers \(a_1, a_2, \ldots, a_n\).

We will use the following sorting scenario. We will first compare first number with the second and define the maximum. Then compare this maximum with the third number in the sequence and define maximum of the first three numbers. Repeatedly use the same procedure to define the maximum of given numbers and save it in the last position. In the second iteration we will apply exactly same method to determine the second maximum and save it in the second from the end position. Algorithm continues task up to determining the minimum of remaining two numbers.

Determine running time function in terms of \(\Theta\)-notation.

Solution. Not difficult to see that the running time of algorithm is a function

\[
T_n = (n-1) + (n-2) + \cdots + 2 + 1 = \frac{(n-1)+1}{2} (n-1) = \frac{n(n-1)}{2} = \Theta(n^2).
\]

4. Use the following algorithm to sort \(n\) numbers \(a_1, a_2, \ldots, a_n\).

Choose the first number of the sequence as pivot element.
Compare pivot with the last number of the sequence. If the pivot is less than the last number keep these numbers in their original positions. Otherwise exchange them. Then if the pivot is in its original position, compare it with the second last number of the sequence. Accordingly, if the pivot is in the last position, compare it with the second number of the sequence. By comparing the numbers (pivot and another) picked up from two ends of the sequence we define the final position of the pivot element. Its position will not be changed until end of the algorithm. Further, the same procedure will be implemented to two, then 4, then 8, etc. subsequences of unsorted numbers.

Determine running time function in terms of \( \Theta \)-notation.

Solution. The algorithm is nondeterministic algorithm. So, we will define two functions: best-case running time function and worst-case running time function.

In the best case the number of numbers must satisfy the following property:

\[
n = ((((2 + 1)2 + 1)2 + 1)2 + 1) \cdots + 1.
\]

Besides of this pivots should split the sequence into two equal sized subsequences. Hence,

\[
T_n(\text{Best - case}) = (n - 1) + 2\left(\frac{n - 1}{2} - 1\right) + 4\left(\frac{n - 1}{2} - 1\right) + \cdots =
\]

\[
= (n - 1) + (n - 3) + (n - 7) + \cdots + (n - (2^{\log_2 n} - 1)) = \Theta(n \log_2 n).
\]

In the worst case, on the other hand, running time function is of the type

\[
T_n(\text{worst - case}) = (n - 1) + (n - 2) + \cdots + 1 = \Theta(n^2).
\]

This means that \( \Theta(n \log_2 n) \leq T_n \leq \Theta(n^2) \).