# How to Estimate Time for An Algorithm – An Example

Let us assume that we are working on the following task: for a given string find all substrings that are a real word (i.e., that are present in the given dictionary of real words). For example, for a 7-letter string **`withing`** we can find substrings **`with`**, **`it`**, **`thin`**, **`within`**, **`thing`** that are real words.

We want to estimate how long it will take to do that in a brute-force manner for strings of different lengths.

First things first, we need to estimate time complexity of the brute-force algorithm. To do that we need to outline the algorithm itself. There may be multiple brute-force approaches, we will just choose one.

Let's assume that the given dictionary size is $m$ words and that the string length is $n$.

One way to brute-force this is to 1) list all possible substrings; and then 2) check each against the dictionary. We will assume that we don't optimize dictionary lookup in any way and just do a sequential search, i.e., go through the dictionary entry by entry and compare. So, the time complexity for dictionary lookup for one substring is proportional to $m$, or, as we would say in computer science, the complexity is $O(m)$.

What is the time complexity for listing all possible substrings of a string of length $n$? To answer this question, we need to outline how we are going to do that. For simplicity, let's say that notation $[i, j]$ will denote a substring that starts (= its first character is) at position $i$ of the original string and its last character is at position $j$. The substring $[1, n]$ is the original string itself.

One way to list all the substrings is to start with a first position in the string and list all substrings that start from this position. For the string **`withing`**, the first position is letter **`w`** and all the substrings that start with it are: $[1,1]$**`:w`**, $[1,2]$**`:wi`**, $[1,3]$**`:wit`**, $[1,4]$**`:with`**, $[1,5]$**`:withi`**, $[1,6]$**`:within`**, $[1,7]$**`:withing`**. Then we move to the second position and list all the substrings that start from the second position onwards: $[2,2]$**`:i`**, $[2,3]$**`:it`**, $[2,4]$**`:ith`**, $[2,5]$**`:ithi`**, $[2,6]$**`:ithin`**, $[2,7]$**`:ithing`**. We continue to do so until we reach the last position.

Is this a correct algorithm? It would be if 1) it lists every possible substring *at least* once; and 2) it lists every possible substring *at most* once. For this algorithm, it is easy to formally prove that there can't be a substring $[i, j]$, where $1 \le i \le j \le n$ such that it was either not listed or listed twice. Try to prove it as an exercise. The proof starts with words "Assume that there is a substring $[i, j]$ that was not listed / was listed twice". In your reports, I suggest that you provide at least a high-level proof of your brute-force algorithm correctness.

Now, this algorithm goes through every position in the string, and the string is of length $n$, therefore its complexity is at least $O(n)$. Now, for position 1, it will go through $n$ substrings; for position 2 it will go through $(n-1)$ substrings; for pos. 3, it'll go through $(n-2)$ substrings, etc. Finally, for position $n$, it will go through 1 substring. Overall, we will go through $n + (n-1) + (n-2) + \cdots + 1$ substrings, which evaluates to $\frac{n(n+1)}{2} = {}^{1}/_{2}\,(n^2 + n)$. Now, for the time complexity estimation, we usually drop the multiplicative factor (${}^{1}/_{2}$ in this case) and reduce it to $(n^2 + n)$. If we want to be completely formal, we can also drop the $n$ component and have the total complexity of enumerating the substrings as $O(n^2)$ since for large $n$, the $n^2$ component will dominate: the larger the $n$, the more negligible will be the relative difference between $n^2 + n$ and $n^2$; this could also be expressed as: $\lim_{n\to\infty} \frac{n^2+n}{n^2} = 1$. But we don't have to be that thorough and we plan to use this estimation for not so large values of $n$, so we can leave it at $(n^2 + n)$.

Now, for each of these substrings, we will also need to check it against the dictionary, so our overall time complexity for the whole algorithm is $m(n^2 + n)$. In most cases, dictionary size will be fixed (e.g., 100 000 words), so we can drop this constant as well and we are still left with $(n^2 + n)$.

When we say that algorithm time complexity is proportional to $(n^2 + n)$, what we are actually saying is that the time it will take to execute algorithm for an input of size $n$ is $C \times (n^2 + n)$, where $C$ is a positive constant.

So now that we know the time complexity, if we somehow know that brute-force approach for a string of length $k$ took exactly time $t$ (e.g., $t$ seconds or $t$ milliseconds), then we can use that to estimate how much time exactly in seconds it will take to process an arbitrary string of length $n$; we will first need to find $C$ by solving a simple equation $C \times (k^2 + k) = t \iff C = \frac{t}{k^2+k}$. E.g., if string of length 10 took 1 second, then $C = \frac{1}{10^2+10} = \frac{1}{110}$. From that we can estimate time required for string of any length. String of length 50 will take $\frac{50^2+50}{110} \approx 23$ seconds.

Hopefully, this example makes it clearer on what needs to be done in Assignment 2. Also, take a read here on algorithm complexity analysis: https://discrete.gr/complexity/.