1 Algorithmic Efficiency

There are often many different algorithms which can be used to solve the same problem. In spite of being all correct solutions of the same problem, the choice of an algorithm can have profound impact on the efficiency of the resulting program. Thus, it make sense to develop techniques that allow us to

- compare different algorithms with respect to their “efficiency”
- choose the most efficient algorithm for the problem

The question is: how do we measure the efficiency of an algorithm? There are many parameters that can be taken into account. Most important are:

- how many “operations” does the algorithm perform to obtain the solution to the problem?
- how much “space” will the algorithm require to produce the solution to the problem?

We will refer to the first aspect as time complexity of the algorithm, while we will refer to the second aspect as the space complexity of the algorithm.

In this context we will focus exclusively to the issue of time complexity. Thus, we are interested in “counting” the number of “operations” that the algorithm will perform to solve the problem. This is very vague:

- what is an “operation”? Is it an instruction in the pseudo-code of the algorithm? Or an instruction in the source language? Or an instruction in the machine program produced by the compiler? Or....

Since our analysis will be kept at the level of algorithms, we do not want to worry about machine instructions or programming languages. In our rather informal considerations we will consider algorithms written in a sufficiently precise pseudo-code;

- do we really want to count all the operations? Once we have defined what an operation is (e.g., an instruction in a sufficiently precise pseudo-code version of the algorithm), we will also need to decide whether we
are interested in counting all the operations involved in the algorithm. In most cases this is not going to be the case. It is common to identify specific classes of operations that are particularly relevant for the problem at hand, and we will thus concentrate on counting how many occurrences of such operations will be employed to obtain a solution.

From these considerations it emerges that we are going to take a very very informal view of algorithmic efficiency. Note that there is a whole research field, called algorithmic complexity, which deals (formally and precisely) with this sort of issues.

In our context we will generally discuss the algorithm’s efficiency as a function of the number of elements to be processed; thus when we write something like

\[ f(n) = \text{efficiency} \]

to express that the algorithm \( f \) applied to a problem of size \( n \) requires \( \text{efficiency} \) operations.

# 2 Examples

Let us start by considering some examples. The goal in each examples is to measure the efficiency of the solution by determining how many times the instruction \( \text{INSTR} \) is executed.

## 2.1 Linear Loop

Consider the loop:

```c
i = 1;
while ( i <= 1000 )
{
    INSTR
    i = i+1;
}
```

This algorithm has an efficiency equal to 1000, since \( \text{INSTR} \) is executed that number of times. More in general, if we consider

```c
i = 1;
while ( i <= n )
{
    INSTR
    i = i+1;
}
```

has efficiency equal to \( n \).

If we consider the loop
i = 1;
while ( i <= n )
{
    INSTR
    i = i+2;
}

then we have an efficiency equal to $\frac{n}{2}$.

All these are examples of solutions which have a linear complexity, as in all cases the efficiency is linearly proportional to the parameter $n$ (e.g., it is either $n$ or $n/2$ or, in general, of the form $a \times n$ where $a$ is a constant).

## 2.2 Logarithmic Loops

Let us now consider a different kind of loops.

```plaintext
i = 1;
while (i < n)
{
    INSTR
    i = i * 2;
}
```

In this case it is easy to see that the number of times $\text{INSTR}$ is performed as function of $n$ is $\lceil \log n \rceil$ (where the logarithm is in base 2). This is a situation that occurs quite frequently (e.g., binary search).

Algorithms which have an efficiency of the type $a \times \log n$ where $a$ is a constant, is called a logarithmic solution.

Consider two solutions for the same problem, one having efficiency $a \times n$ and the other $b \times \log n$ ($a$ and $b$ constants); which one would you choose?

## 2.3 Nested Loops

Counting the number of times a certain instruction is executed can be complex; typical example occurs when we have nested loops.

Consider the following code fragment:

```plaintext
i = 1;
while ( i < n )
{
    j = 1;
    while (j < n)
    {
        INSTR
        j=j+1;
    }
    i = i+1;
}
```
In this case the \textit{INSTR} is executed $n$ times in the innermost loop, and the innermost loop is itself repeated $n$ times because of the outermost loop. Thus the efficiency is $n^2$. This is what we will call a \textit{quadratic} solution, whenever the efficiency is of the type $a \times n^2$.

Another example:

```plaintext
i = 1;
while (i <= n)
{
    j = 1;
    while (j <= n)
    {
        \textit{INSTR}
        j = j*2;
    }
    i = i+1;
}
```

It is easy to see that the efficiency in this case is $n \times \log n$.

What happen if the two loops are not independent? Consider for example:

```plaintext
i = 1;
while (i < n)
{
    j = 1;
    while (j <= i)
    {
        \textit{INSTR}
        j=j+1;
    }
    i = i + 1;
}
```

Observe that now \textit{INSTR} is repeated $i$ times in the innermost loop, and the value of $i$ will be different at each iteration of the outermost loop. A simple calculation shows that the \textit{INSTR} is executed:

$$1 + 2 + 3 + \cdots + n = \frac{n \times (n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$$

times.

### 3 Big-O Notation

With the speed of computers today, we are not concerned with an exact measurement of an algorithm’s efficiency as much as we are with its general magnitude. If the analysis of two algorithms show that one executes 15 iterations while the other executes 25 iterations, they are both so fast that we cannot see the
difference. But if one iterates 15 times and the other 1500 times, we should be concerned.

We have previously indicated efficiency as a function of the “size” of the input (e.g., the value $n$ in the previous examples). While the equations derived for the algorithm’s efficiency may be complex, there is usually a dominant factor in the equation that determines its “order of magnitude”. Therefore, we don’t need to determine the complete measure of efficiency, but only the factor that determines the magnitude. This factor is the big-$O$, as in “On The Order Of”, and is written as $O(f(n))$ where $f(n)$ is the expression which identifies the order of magnitude of the efficiency.

For example, if we consider a linear algorithm which has efficiency $n^2$ then its order of magnitude is $O(n)$, since the constant factor is not affecting the order of magnitude. If we consider the quadratic loop seen in the previous section, we should write its efficiency as $O(n^2)$.

In determining the order of magnitude, there are various simplifications that can be applied without modifying the order of magnitude. For example:

- if we have $O(a \times f(n))$ where $a$ is a constant, then this is equivalent to $O(f(n))$ (i.e., constant factors can be removed)
- if we have $O(f(n) + g(n))$ and the order of magnitude of $f(n)$ is greater than $g(n)$ then this is equivalent to $O(f(n))$.

As an example, let us consider the previously seen nested loops where the innermost loop is dependent on the outermost one. We have seen that the efficiency is $\frac{n^2}{2} + \frac{n}{2}$. In this case the order of magnitude is $O(n^2)$ according to the described simplifications.