CS 272

Complexity of Algorithms

Enrico Pontelli and Karen Villaverde
Introduction

**Notions of Efficiency**

The problem we are tackling in this context is how to choose one algorithm for solving a specific problem between many alternative approaches. In fact, it is not uncommon to have a huge number of alternative algorithms to solve the same problem. So, which one shall we pick?

There are many different criterion that one could apply; nevertheless, in this context we will focus on one key criteria: **efficiency**. When we typically talk about efficiency, we immediately are drawn to think about **speed**. This is not completely precise, and indeed the notion of efficiency is much more general and covers many different aspects.

To keep the discussion simple, we commonly concentrate on three notions of efficiency:

1. **Time Efficiency**: this is the most common aspect of efficiency that we encounter – it refers (intuitively) to the amount of time that the algorithm requires to solve the problem.

2. **Space Efficiency**: this notion of efficiency refers to the amount of **memory** that is required to solve the given problem.

3. **Development Efficiency**: other notions of efficiency refer not to what happens during the execution of the program, but what happens during the development of the program – i.e., how “complex” is the solution.

In this context we will occasionally talk about Space Efficiency, while in the majority of the cases we will discuss Time Efficiency.

**Measuring Efficiency**

When it comes the time to measure efficiency of an algorithm, we need to determine how to do that. What do we actually measure? And how?

When we talk about Space Efficiency, the answer to these questions is fairly simple. Since space efficiency is concerned with how much memory is employed by the program, we can simply try to count the maximum number of bytes used at any point in time by our algorithm to store any kind of data.

The situation is not as clear in the case of Time Efficiency.

**Experimental Measure**

One naïve approach to measuring time efficiency is to simply run the problem and measure the time (in seconds) that is required to obtain the answer to the problem.
Let us consider a simple example. Let us consider the problem of sorting an array of integers. There are tons of different sorting algorithms that we could use. Let us consider for example selection sort and quicksort. The following table reports the execution time for the two algorithms for different sizes of array (the array are initialized with randomly generated numbers).

<table>
<thead>
<tr>
<th>Number of Elements in the Array</th>
<th>Quicksort</th>
<th>Selection Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1 ms.</td>
<td>3 ms.</td>
</tr>
<tr>
<td>1000</td>
<td>9 ms.</td>
<td>28 ms.</td>
</tr>
<tr>
<td>10000</td>
<td>12 ms.</td>
<td>1373 ms.</td>
</tr>
<tr>
<td>100000</td>
<td>143 ms.</td>
<td>155296 ms.</td>
</tr>
</tbody>
</table>

The program used to take these measurements is the following:

```java
import java.io.*;
import java.lang.*;
import java.util.*;

class sorting
{
    public static void quicksort(int[] array, int start, int size)
    {
        int pivot;
        int left, right;

        if (size <= 1)
            return;

        pivot = array[start];
        left = start+1;
        right = start+size - 1;

        while (left <= right)
        {
            while (left <= right && array[left] < pivot)
                left++;

            while (left <= right && array[right] >= pivot)
            {
                right--;
            }
```
if (left < right)
{
    int temp = array[left];
    array[left] = array[right];
    array[right] = temp;
}
array[start] = array[right];
array[right] = pivot;
quicksort(array,start,right-start);
quicksort(array,right+1,start+size-1-right);
}
public static void selectionsort(int array[])
{
    int temp;
    for (int i=array.length; i >= 2; i--)
    {
        int max = 0;
        for (int j=0; j <i; j++)
        {
            if (array[j] > array[max])
                max = j;
        }
        temp = array[i-1];
        array[i-1] = array[max];
        array[max] = temp;
    }
}
public static void main(String args[])
{
    int i;
    long t1,t2;
    Random r = new Random();

    for (i=100; i < 1000000000; i = i*10)
    {
        int[] array1 = new int[i];
        int[] array2 = new int[i];

        for (int j =0; j < i; j++)
            array1[j] = array2[j] = r.nextInt(i)+1;

        t1 = System.currentTimeMillis();
quicksort(array1,0,i);
t2 = System.currentTimeMillis();

        System.out.print("SIZE: "+i+"    Quicksort Time: "+(t2-t1));
t1 = System.currentTimeMillis();
selectionsort(array2);
    t2 = System.currentTimeMillis();
    System.out.println("   Selection Time: "+(t2-t1));
    }
    }
    }

This approach in measuring has some major drawbacks:

- the timers used to compute execution time are often rather imprecise. In particular it
  might be difficult to guarantee that two executions are performed in exactly the same
  state, thus making it very hard to compare different solutions.

- most of the timers measure what is typically called wall-clock – i.e., the actual time
  expired from the beginning of the execution until the end; unfortunately a lot of
  things may happen in the middle of the execution, that are unrelated to the program –
  e.g., the cpu can be temporarily taken away from our program and given to a higher
  priority one.

- the execution time is computed with respect to a single execution, for one specific
  input, and might not be representative of the “typical” situation. Experiments can be
  done only with a limited set of inputs.

If we want to extrapolate some more general information about the efficiency of the program,
we need to do quite a bit of analysis and thinking. Nevertheless, experimental evaluation has
its own positive aspects. For example, if we plot the execution times from the table above we
 can get an idea of what kind of dependency exist between the size of the input and the
execution time.

The objective of the material in the successive sections is to present a methodology to discuss
about time efficiency at a higher level of abstraction, and without actually running the code.
The methodology will be used to classify solutions based on the “order of magnitude” of the
execution time.

**Computational Complexity**

**Introduction**

When we measure the time efficiency of an algorithm, we probably don’t want to write the
program and start using our watch to measure execution time. Instead of looking at actual
execution times, we prefer to use as a measure of efficiency a *count* of how many instructions
are needed to achieve the solution.

The first question we should then ask is: what do we mean by instructions? Obviously, if we
want to be very precise and ensure that the count of instruction adequately reflects the
execution time, then we should probably operate at the level of machine language instructions
and guarantee that all the machine instructions executed are counted.

This is generally also infeasible; in order to perform this type of analysis we would need to

- write the program in a high-level language (e.g., Java)
• compile the code to machine language

• determine for each machine instruction how much time it takes to executed it (not all machine instructions take the same amount of time)

• count all the instructions that are executed.

This type of analysis is too complicated, and indeed it might end up providing us with too much information (which can actually make it harder to compare different algorithms). In our analysis we are not interested in going to this level of detail, but we prefer to have a more qualitative type of analysis. For example, if we are trying to compare quicksort and selection sort, we would like to observe that, as the array size grows, the execution time of quicksort grows slower than that of selection sort. Again, our objective is not to obtain an exact measure of the execution time, instead we are interested in obtain an approximated classification of algorithms based on how quickly their execution time grows as the input size grows.

**Pseudo-code**

To accomplish this, we will make the following simplifications. We expect to have our algorithm coded in a high-level language; the language could be an actual programming language (e.g., Java), or it could simply be pseudo-code. A pseudo-code is a combination of structures from traditional programming languages (e.g., loops, conditionals) and English language. For example, the following could be an adequate pseudo-code that describes how to find the maximum in an array:

```plaintext
Algorithm: findMaximum
Input: an array of size n
Output: the maximum value in the array

currentMaximum set to array[0]
for index=1 to array length-1 do
  if currentMax < array[i] then
    currentMax set array[i]
return currentMax
```

The corresponding Java code is

```java
public class ArrayMaxProgram
{
    public static void main(String args[]) {
        int[] array = {65, 43, 75, 9, 12, 42};
        int currentMax = array[0];
        for (int i=1; i < array.length; i++)
            if (currentMax < array[i])
                currentMax = array[i];
        System.out.println("The maximum is "+currentMax);
    }
}
```
In both cases, we tend to use the same type of structures (conditionals, loops) and the same type of instructions (assignments to variables, basic expressions).

Given an algorithm, expressed either in pseudo-code or in an actual high-level language, we will refer to its time efficiency by counting how many primitive instructions are executed to solve the problem. The definition of what a primitive operation is up to the person performing the analysis – it could be the assignment operations, the calls to one or more methods, the comparison operations, etc. Thus, the only thing we make clear is that, before we perform our analysis, we set our focus on a specific class of operations and we will count only those operations.

For example, if we go back to our example of quicksort versus selection sort, then we could decide to compare the two solutions by focusing on the comparisons performed. Thus, we select the comparison operation as the primitive operation to be counted.

The core of the code of selection sort is

```java
for (int i=array.length; i >= 2; i--)
{
    int max = 0;
    for (int j=0; j <i; j++)
    {
        if (array[j] > array[max])
            max = j;
    }
    temp = array[i-1];
    array[i-1] = array[max];
    array[max] = temp;
}
```

The primitive operation we are interested in has been written in boldface. So, how many times is that instruction executed? Obviously, the number of times depends on how many times the two for loops are executed. The innermost loop executes exactly i times, and there is one comparison for each iteration of such loop. The first time we go through the innermost loop, the value of i is exactly the size of the array (let’s call it N). This means that during the first iteration of the outermost loop, we have N comparisons performed. During the second iteration of the outermost loop, the value of i is N-1, thus we have N-1 comparisons; and so on. In the end, the total number of comparisons is:

\[ N + (N-1) + (N-2) + \ldots + 2 + 1 \]

If you go back to your basic algebra, you will see that this summation can be written simply as:

\[ \frac{N \times (N + 1)}{2} \]
Examples
So let us look at some simple code fragments and measure their time efficiency by counting primitive operations.

Constant Case:
Let us consider the following pseudo-code

```plaintext
for (i=0; i<10; i++)
  operation
```

if operation is our primitive operation, then the time efficiency of this code fragment is simply 10, since that’s the number of times that operation is performed. Since the efficiency is independent from the size of the input to the problem, we typically refer to this type of situations as Constant-time Solutions.

Thus, if n is the size of the input, if we indicate with \( \text{eff}(n) \) the time efficiency of the algorithm with an input of size n, then in our case

\[
\text{eff}(n) = 10
\]

Linear Case:
Let us consider the following fragment:

```plaintext
for (i=0; i<n; i++)
  operation
```

In this case, the number of times that operation is performed depends on the value of n. In fact, operation is going to be executed exactly n times. This type of solutions is called Linear-time Solution. Using the formal notation used before, we have:

\[
\text{eff}(n) = n
\]

Quadratic Case:
Let us consider the following code:

```plaintext
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    operation
```

In this case, the operation is performed n times inside the innermost loop, and the innermost loop is itself executed n times, thus operation is executed \( n^2 \) times. This type of solutions is called Quadratic-time Solutions. Using the formal notation we have:

\[
\text{eff}(n) = n^2
\]

As another example, consider a variation of the above code:

```plaintext
for (i=0; i<n; i++)
  operation
```
for (j=0; j<i; j++)
    operation

In this case, the counting is more complex. During the first iteration of the outermost loop, the operation is executed 0 times; during the second, it’s executed one time; during the second, two times; etc. In total:

\[
eff(n) = 0+1+2+\ldots+n-1 = n(n-1)/2 = (n^2-n)/2
\]

For reasons we will explain later, we will also consider this solution as a Quadratic-time solution.

Logarithmic Case
Consider the following code fragment:

for (i=n; i>=1; i=i/2)
    operation

In this case counting the number of operations is more complex. We need to determine how many times the loop iterates. Let us look at how the value of i changes during the execution of the loop:

- at the end of the first iteration \( i=n/2 \)
- at the end of second iteration is \( i=n/4 \)
- at the end of third iteration is \( i=n/8 \)
- at the end of fourth iteration is \( i=n/16 \)

If we continue, we will see that at the iteration number \( k \) the value of \( i \) is \( n/2^k \). So when does the loop stops? The value of \( i \) has to reach 1 (or less) to stop. If we set up an equation we obtain:

\[
\frac{n}{2^k} = 1 \Rightarrow n = 2^k \Rightarrow k = \log_2 n
\]

Thus, the number of iterations is about the logarithm (base 2) of \( n \). This type of solutions are called Logarithmic-time Solutions. Formally,

\[
eff(n) = \log_2 n
\]

Exponential Case
Let us consider a much more complex example, where we have a recursive method:

```java
public static int function(int input)
{
    operation;
    if (input == 0)
Now, let us try to determine how many times operation is performed. We can set up a simple equation to help us out: note that if the input is n, then the method will execute operation 1 time and then proceed with the two recursive calls; in the first one, we will have that operation is performed eff(n-1) times, while in the second recursive call it will be executed eff(n-1) times; thus we get:

$$\text{eff}(n) = 1 + \text{eff}(n-1) + \text{eff}(n-1)$$

unless n=0, where we have eff(0) = 1. We will not really try to go into depth of what is the solution of this recursive equation, but the result is that

$$\text{eff}(n) \approx 2^n$$

This type of situations are called \textit{Exponential-time Solutions}.

\textbf{Comparing Solutions}

Let us now consider this different types of solutions that we have seen in these examples. How do they compare with each others? E.g., if we have a linear-time and a quadratic-time solution for the same problem, which one should we pick?

The answer is not immediate; clearly we want to pick the one that requires the least number of operations. And clearly, our answer will depend on what is the value of n that we are interesting to deal with (i.e., the size of the input). If we do not know n, then we will pick the solution that will give us the best result in the majority of the cases.

<table>
<thead>
<tr>
<th>n</th>
<th>constant</th>
<th>logarithmic</th>
<th>linear</th>
<th>quadratic</th>
<th>exponential</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>1.584963</td>
<td>3</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>2</td>
<td>4</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>2.321928</td>
<td>5</td>
<td>25</td>
<td>32</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>2.584963</td>
<td>6</td>
<td>36</td>
<td>64</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>2.807355</td>
<td>7</td>
<td>49</td>
<td>128</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>3</td>
<td>8</td>
<td>64</td>
<td>256</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>3.169925</td>
<td>9</td>
<td>81</td>
<td>512</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>3.321928</td>
<td>10</td>
<td>100</td>
<td>1024</td>
</tr>
</tbody>
</table>

As we can see, as we move from constant to logarithmic, to linear, to quadratic, to exponential, we have functions that grow faster as n increases. This means that, without any prior knowledge of the value of n, constant time solutions are preferable to logarithmic, that are preferable to linear, that are preferable to quadratic, that are preferable to exponential.
Asymptotic Notation and Simplifications

Up to this point, we have decided to focus our measure of efficiency on counting how many times certain operations are performed. Typically, this will give us some formula which depends on some input parameters \( n \). The problem that we might encounter is that the resulting formula can be fairly complex. E.g., we can obtain something like

\[
eff(n) = n^2 + n / 2
\]

Do we really need all this level of detail? Since our focus is really on trying to determine the growth rate of the number of operations as a function of the input size \( n \), we are allowed a number of simplifications to the formula that expresses the time efficiency of our algorithm. The result is a loss of precision in our measurement, but a simpler formula that still gives us a way of classifying our algorithm based on the rate of growth. The outcome also means that when we use our measurements to compare different solutions, we will be able to determine only major differences between the algorithms (i.e., their rates of growth are order of magnitudes different).

Mathematicians have designed a special notation to describe rates of growth of algorithms. The notation is called big-O notation. The notation is simple: it consists of the symbol \( O \) followed by a formula enclosed in parentheses. For example, \( O(n^3) \) is a valid big-O notation measurement.

When we use the big-O notation to estimate the time-efficiency, the goal is to provide a qualitative insight as to how changes in \( n \) affect the algorithm performance (as \( n \) grows larger). Because big-O notation is not intended to be a quantitative measure, it is not only appropriate but desirable to reduce the formula inside the parentheses so that it captures the qualitative behavior of the algorithm in the simplest possible form. The most common simplifications that you can make when using the big-O notation are as follows:

- Eliminate any term whose contribution to the total ceases to be significant as \( n \) becomes larger. When a formula involves many terms added together, one of those terms often grows much faster than the others and ends up dominating the entire expression as \( n \) becomes very large. For large values of \( n \), this term alone will control the running time of the algorithm, and you can ignore the other terms in the formula entirely.

  For example, the efficiency \( O(n^5+3n^3+2) \) is equivalent to \( O(n^5) \), since for large values of \( n \) the term \( n^5 \) grows much much faster than \( 3n^3 \) and 2.

- Eliminate any constant factors. When you calculate time efficiency, your main concern is how running time changes as a function of the problem size \( n \). Constant factors have no effect on the overall pattern. If you bought a machine that was twice as fast as your old one, any algorithm that you executed on your machine would run twice as fast as before for every value of \( n \). The growth pattern, however, would remain exactly the same.

  For example, the efficiency \( O(5n^4) \) is equivalent to \( O(n^4) \) since the constant factor 5 is irrelevant.
Let us consider another example. When we studied selection sort, we came up with a formula \( (n^2 + n)/2 \). Thus, if we use the big-O notation, we have

\[
O\left(\frac{n^2 + n}{2}\right) = O\left(\frac{n^2}{2} + \frac{n}{2}\right) = O\left(\frac{n^2}{2}\right) = O\left(\frac{1}{2}n^2\right) = O(n^2)
\]

This tells us that the selection sort is a Quadratic-time solution.

This type of analysis is frequently also called *asymptotic analysis*. Suppose two algorithms solving the same problem are available, one with time efficiency \( O(n) \) and one \( O(n^2) \). We know that \( n^2 \) grows much faster than \( n \), which suggests that we should pick the algorithm with time efficiency \( O(n) \) – if we have big inputs, it will take less time. In other words, the first algorithm is *asymptotically better* than the second one. Thus, the big-O notation provides a way of classifying and (asymptotically) comparing algorithms. In particular, \( O(1) \) (constant-time solution), is better than \( O(\log n) \), which is better than \( O(n) \), which is better than \( O(n^2) \), etc.

Some words of caution:

- if we have two algorithms that have the same big-O time efficiency, then we really cannot say anything about which one is better – this does not mean that they have the same computation time, it just means that asymptotically we cannot discriminate them.

- asymptotic analysis tells us what happens when we consider very big values of \( n \); thus, even though \( O(n) \) is asymptotically better than \( O(n^2) \), in practice there might be a large collection of values of \( n \) for which the second algorithm runs faster than the first on. E.g., consider an algorithm A with time efficiency \( 1000n \) and algorithm B with efficiency \( 1/1000n^2 \), then you can easily see that even for fairly large values of \( n \) algorithm B is faster than A, even though A is asymptotically better than B.

**Worst-case versus Average-case versus Best-case**

There are a number of algorithms whose running time does not only depend on the size of the input, but also on the actual values in input. Consider a very simple example: we have an array of size \( n \) and we are trying to locate one particular value (key) in this array. Standard sequential search can be described by the following code fragment

```java
for (int i=0; i<n; i++)
    if (key == array[i])
        return (i);
```

Let us consider the == comparison as our primitive operation. What is the time efficiency of this algorithm? The number of iterations of the loop depends on the location of the key inside the array. If the key is in the first position, then in one step we will stop. If the key is at the end of the array (or not present at all), then it will take \( n \) operations to complete. Which one do we pick?

When we deal with big-O notation, we automatically assume that we are looking at the *worst-case complexity*. Thus, in the example above, we look at the worst-case scenario, which is the
one where the key is not present in the array at all, and we state that the algorithm has time efficiency $O(n)$.

Since big-O notation represents a worst-case analysis, this means that the time-efficiency is really just an upper bound to the actual time efficiency.

Similarly to the case of big-O, there is a similar notation that is employed to describe best-case analysis. When we perform best-case analysis, we are providing a lower bound to the actual time efficiency. The notation we use for best-case analysis is analogous to big-O, except that instead of $O$ we use the symbol $\Omega$. Thus, the sequential search algorithm shown earlier has a best-case time efficiency $\Omega(1)$.

Occasionally we might also be interested in performing average-case analysis. In the case of the small search procedure above, the average case is obtained by averaging all the possible cases:

$$n + (n-1) + (n-2) + \ldots + 2 + 1 = \frac{n + 2}{2}$$

which gives us an asymptotic average case time of about $n/2$. Considering that $n/2$ is $O(n)$, we can see that the average case has the same order of magnitude as the worst case – an indication that this approach to search is not particularly good.

**Some Examples**

Let us consider another simple algorithm and let us try to estimate its worst case efficiency. The algorithm we will consider is the well-known binary search of an array. Let us assume that we have an array $a$ which contains numbers; let us also assume that the array is sorted (i.e., the first element is smaller than the second, which is smaller than the third, etc.). Our goal is to locate a given element $x$ in the array (or determine that the element is not present). Since the array is sorted, we can use binary search: we start by comparing $x$ with the element in the middle of the array $a$. If the two are equal, then we are done; if $x$ is smaller, then we know that our element can only be in the first half of the array (thus, we can ignore all the elements in the second half); if $x$ is bigger, than our element can only be in the second half of the array – thus the first half can be discarded. At that point the search process can be repeated in the selected half.

Let us measure the efficiency in terms of how many comparisons between $x$ and elements of the array are required.

Clearly, in the best case, the efficiency is $\Omega(1)$, and this will occur if we find the element at the first comparison (i.e., $x$ is exactly equal to the element in the middle of the array). The worst case occurs when the element $x$ is not in the array; in that case, we keep cutting the array in half until no elements are left. For example, the following figure illustrates the comparisons made when searching for 7.
So, if we start with an array of size \( n \), how many steps will be required before realizing that the element is not in the array? We need to reduce the size of the segment of the array all the way down to an array of size 1; at each step the existing size is divided by two. At the first step we have an array of size \( n \), at the second step we are left with \( n/2 \), at the next \( n/4 \), at the next \( n/8 \)… so if we repeat the process \( k \) times, at the last step we are left with \( n/2^k \). Since at the last step we expect the size of the array to be 1, then we can set up the following equation to determine what is the value of \( k \):

\[
\frac{n}{2^k} = 1 \Rightarrow n = 2^k \Rightarrow \lg n = k
\]

which allows us to conclude that binary search has a worst case efficiency of \( O(\lg n) \).

**A Profiler Class**

The following code is a class that contains the implementation of two different search strategies (sequential and binary) along with a method that allows to request timing of execution with a selected array.

```java
/**
 * This class offers two implementations of search strategies;
 * in addition the class provides methods to time the execution
 * of each search strategy as well as count the number of
 * comparisons
 *
 * @author Enrico Pontelli
 */

import java.io.*;
```

---

```
6 10 15 17 20
```

```
6 10
```

```
10
```
public class profileSearch {

    private static int countsteps;

    /*
     * Method to perform sequential search in an array
     * @param array this is an array of integer to be searched
     * @param key this is the element we are searching
     * @return the position of the element in the array if found; -1 otherwise
     */

    public static int sequentialSearch(int array[], int key) {
        countsteps = 1;
        if (array.length < 1)
            return -1;
        for (int i = 0; i < array.length; i++)
            if (array[i] == key)
                return i;
            else
                countsteps++;
        return -1;
    }

    /*
     * Method to perform binary search in an array
     * @param array this is an array of integer to be searched
     * @param key this is the element we are searching
     * @return the position of the element in the array if found; -1 otherwise
     */

    public static int binarySearch(int array[], int key) {
        int start, end, middle;
        countsteps = 1;
        if (array.length < 1)
            return -1;
        if (! sorted(array))
            return -1;
        if (! sorted(array))
            System.out.println("Cannot do binary search in a non-sorted array");
    }
return -1;
}

start = 0;
end = array.length-1;

while (start <= end)
{
    middle = (start+end)/2;
    if (array[middle] == key)
        return middle;
    else if (array[middle] > key)
    {
        end = middle-1;
    }
    else
    {
        start = middle+1;
    }
    countsteps++;
}

return -1;

/*
* Method to test whether an array is sorted
* @param array this is an array of integer
* @return true if array sorted, false otherwise
*/

public static boolean sorted(int array[])
{
    for (int i=0; i < array.length-1; i++)
        if (array[i] > array[i+1])
            return false;
    return true;
}

/*
* Method to profile execution for time
* @param array this is an array of integer to be searched
* @param key the element to be searched
* @param type this is a string indicating what search we want to perform
* @return time expired for execution
*/

public static long timeSearch(int array[], int key, String type)
{  
  long t1,t2;

  if (type.compareTo("sequential")==0)  
  { 
    t1=System.currentTimeMillis();
    sequentialSearch(array,key);
    t2=System.currentTimeMillis();
    return (t2-t1);
  }

  if (type.compareTo("binary")==0)  
  { 
    t1=System.currentTimeMillis();
    binarySearch(array,key);
    t2=System.currentTimeMillis();
    return (t2-t1);
  }

  return (0);  
}

/*  
 * Method to profile execution for number of steps  
 *  
 * @param array this is an array of integer to be searched  
 * @param key the element to be searched  
 * @param type this is a string indicating what search we want to perform  
 * @return time expired for execution  
 */

public static long stepsSearch(int array[], int key, String type)  
{  
  countsteps=0;
  if (type.compareTo("sequential")==0)  
  {  
    sequentialSearch(array,key);
  }

  if (type.compareTo("binary")==0)  
  {  
    binarySearch(array,key);
  }

  return (countsteps);  
}  
}
The following is a simple class that can be used to test the profiler class. Note that in this case the profiler class is simply a “container” that stores a number of methods. Whenever we want to use these methods from somewhere else, we can simply call them using the format

```
NameOfTheClass.NameOfTheMethods
```

For example, if we want to call the procedure stepsSearch to search for the element x in the array a using binary search, we can call it from anywhere else as:

```
profilerSearch.stepsSearch ( a, x, “binary”) 
```

The test class is:

```java
import java.io.*;

/**
 * this program has a simple main to test the profile class
 * @author Enrico Pontelli
 */

class testSearch
{
   /**
    * this is the main method
    */
   public static void main(String args[])
   {
      int size;

      System.out.print("How many elements in the array? ");
      size = SavitchIn.readInt();

      // create array
      int[] array = new int[size];

      // read the elements of the array
      for (int i=0; i<size;i++)
      {
         System.out.print("Insert element number "+i+: ");
         array[i] = SavitchIn.readInt();
      }

      // get the key
      int key;
      System.out.print("Insert the key to be searched: ");
      key = SavitchIn.readInt();

      // get the profiler to work:
```
System.out.println("The sequential Search took "+
profileSearch.timeSearch(array,key,"sequential")+
"ms and requires "+
profileSearch.stepsSearch(array,key,"sequential")+
" steps");
System.out.println("The binary search took "+
profileSearch.timeSearch(array,key,"binary")+
"ms and requires "+
profileSearch.stepsSearch(array,key,"binary")+
" steps");