Maple Transactions

Analysis of algorithms as a teaching experience

DONATELLA MERLINI, Dipartimento di Statistica, Informatica, Applicazioni - Università di Firenze, Italy

Abstract. Teaching analysis of algorithms to students in Computer Science degrees, using the approach popularized by Knuth in his series of books "The Art of Computer Programming" and later by Sedgewick and Flajolet in the book "An Introduction to the Analysis of Algorithms", is not a simple task since, in general, these students are more interested in the implementation of an algorithm than in the corresponding theoretical aspects. This approach concentrates on precisely characterizing the performance of algorithms by determining their best, worst and average case performance using a methodology based on symbolic tools such as recurrence relations and generating functions. The most difficult aspect is to understand the average case since this corresponds to studying the algorithm as its possible inputs vary: this represents the most important goal since generally students have no difficulty in understanding the best and worst cases, corresponding to particular input configurations. A compromise that has been successful over the years consists in teaching students the analytical aspects of the problem and then organize a simulation of the algorithm with a system of symbolic computation in order to exhaustively check the theoretical results.

CCS Concepts: • **Theory of computation → Design and analysis of algorithms**; • **Mathematics of computing → Mathematical software**.

Additional Key Words and Phrases: analysis of algorithms, algorithms simulation, Quicksort, symbolic computation

Recommended Reference Format:
Donatella Merlini. 2023. Analysis of algorithms as a teaching experience. *Maple Trans.* 3, 2, Article 15664 (August 2023), 16 pages. https://doi.org/10.5206/mt.v3i2.15664

1 Introduction

The author of this paper has a quite long research and teaching experience in the context of the analysis of algorithms, as popularized by Knuth [5, 6, 7] and Sedgewick and Flajolet [14]; this approach to the analysis of algorithms concentrates on precisely characterizing the performance of algorithms by determining their best, worst and average case performance using a methodology that can be refined to produce increasingly precise answers when desired.
The *Analysis of Algorithms* teaching experience concerns the Computer Science degree at the University of Florence and, in general, involves students more interested in the implementation of an algorithm than in the corresponding theoretical aspects. A compromise that was successful over the years consists in teaching students the analytical aspects of the analysis of an algorithm and then asking them to implement the algorithm in order to precisely check the theoretical results. This is in general possible for small sizes of the problem under consideration while for larger sizes it is necessary to set up a simulation by executing the algorithm on a sufficiently large sample of inputs. In this latter case, obviously, the theoretical results cannot be checked precisely and some statistical test must be used to conclude that the simulation agrees with the theory. In other words, during the course, the analysis of an algorithm or a data structure is accompanied by its simulation with a system of symbolic computation, like `Maple`.

Author's address: Donatella Merlini, donatella.merlini@unifi.it, Dipartimento di Statistica, Informatica, Applicazioni - Università di Firenze, viale Morgagni 65, Firenze, Italy, 50134.

Permission to make digital or hard copies of all or part of this work for any use is granted without fee, provided that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to Maple Transactions, under Creative Commons CC-BY 4.0 License.
https://doi.org/10.5206/mt.v3i2.15664

In the field of algorithms and data structures, simulation is often used to determine whether an algorithm or a program is correct or not. When a program has to act on many data, and the real ones on which it will work are not available, random generators of input data are used to reproduce the characteristics of the real data. This is common practice and knowledge of the correct methods of data generation gives immediate help for a good setting. In a course about analysis of algorithms various mathematical techniques are typically illustrated to determine the complexity of an algorithm and usually the problem is studied in the best, worst and average cases. The most interesting mathematical problems arise in the study of the average case because this corresponds to running the algorithm on all its possible inputs or to consider all possible data structures of a certain size.

The greater intrinsic difficulty in studying the average case translates into a greater difficulty for students in understanding what a study of this type really means. In our teaching experience we have often see students present the results of an algorithm experimentation sold as an average case study, when instead it was simply the arithmetic mean of the result of a few executions of the algorithm involved. This also happens in second and third level university courses.

Teaching a course on algorithm analysis gave us the opportunity to highlight what could be the best strategy to make students learn how to set up the simulation of an algorithm, using a system for symbolic computation in order to fully understand the theoretical results.

In this paper we illustrate the problem by analyzing the well-known `Quicksort` algorithm. In particular, we perform an average case analysis of the algorithm in terms of comparisons and exchanges executed to sort an array and set up the corresponding simulation by using `Maple 2020` [16]. This is an extended version of the presentation made during the `Maple Conference 2022`, in the section `Maple in Education`. The approach illustrated in this paper can be easily adopted for simulating other algorithms and analysing characteristics of data structures.

2 Analyzing QuickSort

`Quicksort` [4] is one of the most studied sorting algorithms. This algorithm sorts an array V of n elements by rearranging V around one of its elements, the pivot, so that the elements smaller than the pivot are to its left and larger elements are to its right. During this phase, two pointers are used to scan the array from left to right and from right to left, respectively. The choice of the pivot is essential, because the more central it is in the array, the more efficient is the partitioning. The simplest idea is the one adopted by standard `Quicksort` which takes an arbitrary element as the pivot, for example the first or the last element of the array. However, as is well-known, there are more sophisticated choices of the pivot that are to be preferred in order to avoid inefficient situations that arise when the input is already sorted, or nearly so. After the array has been partitioned, `Quicksort` is recursively applied to the subarrays on either side of the pivot. Excellent sources for background information, implementation, variants, and their analysis include [1, 7, 8, 9, 10, 12, 13, 17].

Listing 1 is a `Quicksort` implementation in `Maple` that sorts an array V of length n and performs $n + 1$ comparisons during the partitioning stage by using the last element of the array as pivot: in fact, the pivot is compared with all the remaining $n - 1$ elements and two of them are compared twice to allow pointers to cross each other, as will be discussed below. Moreover, we assume for convenience that the array to be sorted is filled with the values $1, \ldots, n$ and that the possible inputs of the algorithm are the $n!$ permutations of these values. The reason why the variant of the algorithm illustrated in Listing 1 is considered, is that for small arrays of length n it is quite easy to exhaustively obtain the cost of the algorithm over all $n!$ permutations of length n, comparing it with the mathematical analysis of the algorithm, as will be illustrated in Section 3. Different choices of the pivot would give rise to both more complex mathematical relations and more complex algorithm simulations in the exhaustive case. The input arguments ℓ and r bound the sub-array to

be sorted and the algorithm has to be called with $\ell = 1$ and $r = n$ to sort an array V of length n. As for the two inner while loops, observe that the increment of pointer i stops thanks to the presence of the pivot, while the decrement of j needs the additional check $\ell \leq j$ to guarantee stopping.
As an example, let us consider the array $V = [5, 6, 2, 1, 7, 3, 8, 4]$ of length $n = 8$ and see what happens during the first partitioning stage of the algorithm. The pivot is $V[8] = 4$, and since $V[1] = 5 > 4$ the pointer i stops in position $i = 1$; on the other end, since $V[7] = 8 > 4$ and $V[6] = 3 < 4$ the pointer j stops in position $j = 6$ and the elements 5 and 3 are swapped. Then, since $i < j$, the pivot is compared with $V[2] = 6 > 4$ and the pointer i stops at $i = 2$, while the comparisons $V[5] = 7 > 4$ and $V[4] = 1 < 4$ stop pointer j at position $j = 4$, hence, the elements 6 and 1 are swapped. We still have $i < j$ and the pivot is compared with $V[3] = 2 < 4$ and with $V[4] = 6 > 4$, so that the pointer i stops at $i = 4$; moreover, since $V[3] = 2 < 4$ pointer j stops at $j = 3$. Now we have $i > j$, that is, the pointers cross each other and the partitioning step ends with the final swap between the pivot $V[8] = 4$ and $V[4] = 6$. The final configuration after this phase is $V = [3, 1, 2, 4, 7, 5, 8, 6]$, obtained with $9 = 7 + 2$ comparisons between elements of the array (note that elements 2 and 6 were compared twice with the pivot) and 4 swaps, 3 of which before the pointers cross each other plus the final swap to put the pivot in the definitive position. An important observation is that for an array of length n, this version of the algorithm always employs $n + 1$ comparisons during the partitioning phase while the number of exchanges depends on the initial configuration of the array, as will be explained in detail in Section 2.2.

Listing 1. An implementation of Quicksort in Maple

```
Quicksort := proc(v::Array, ℓ::nonnegint, r::nonnegint)
  local p, i, j;
  if 0 <= r - ℓ then
    p := v[r];
    i := ℓ - 1;
    j := r;
    while i < j do
      i := i + 1;
      while v[i] < p do i := i + 1; end do;
      j := j - 1;
      while ℓ <= j and p < v[j] do j := j - 1; end do;
      if i < j then
        (v[i], v[j]) := (v[j], v[i]);
      end if;
    end do;
    (v[i], v[r]) := (p, v[i]);
    Quicksort(v, ℓ, i - 1);
    Quicksort(v, i + 1, r);
  end if;
  return v;
end proc:
```

2.1 Best and worst cases

The best case is when at each step we have perfectly balanced partitions of dimensions $\lfloor (n-1)/2 \rfloor$ and $\lceil (n-1)/2 \rceil$. If we denote by C_n the number of comparisons done to sort an array of size n, with

$C_0 = 0$, we have:

$$C_n^{\min} = C_{\lfloor (n-1)/2 \rfloor}^{\min} + C_{\lceil (n-1)/2 \rceil}^{\min} + n + 1; \tag{1}$$

the presence of the floor and ceiling functions complicates the exact solution of the problem for general n but if we assume that $n = 2^m - 1$ the recurrence relation can be easily solved. In fact, with this choice we have $C_{2^m-1} = 2C_{2^{m-1}-1} + 2^m = m2^m$ which gives $C_n^{\min} = (n+1)\log_2(n+1)$. Obviously, the $O(n \log_2 n)$ bound still holds when n is not of that form, in fact, each recursive call processes an array of nearly half the size and we can make only $\log_2 n$ nested calls before we reach an array of size 1.
The worst case is when at each step the partitions degenerate, one is empty and the other contains $n - 1$ elements. In this case we have:

$$C_n^{\max} = C_{n-1}^{\max} + n + 1; \tag{2}$$

this is a simple recurrence relation with solution $C_n^{\max} = n(n+3)/2 = O(n^2)$

Listing 2. Returns the number of comparisons in the best case corresponding to relation (1)

```
Cmin := proc(n::nonnegint)
  option remember;
  if n = 0 then
    0;
  else
    n + 1 + Cmin(floor(1/2*n - 1/2)) + Cmin(ceil(1/2*n - 1/2));
  end if;
end proc:
```

Listing 3. Returns the number of comparisons in the worst case corresponding to relation (2)

```
Cmax := proc(n::nonnegint)
  option remember;
  if n = 0 then
    0;
  else
    n + 1 + Cmax(n - 1);
  end if;
end proc:
```

Both recurrence relations can be implemented as Maple procedures, as shown in Listings 2 and 3.

2.2 Average case

For what concerns the average case, we assume that the probability distribution is uniform on all the permutations of the values $1, \cdots, n$. We denote by C_n the average number of comparisons done to sort an array of size n, and $\pi_{n,j}$ the probability that the jth element is the chosen pivot in an array of size n. The recursive design of the algorithm is translated into the following recurrence relation:

$$C_n = n + 1 + \sum_{j=1}^{n} \pi_{n,j} \left(C_{j-1} + C_{n-j} \right); \tag{3}$$

in fact, if j is the pivot, after the partition phase the two resulting sub-arrays have dimensions $j - 1$ and $n - j$ and give rise to C_{j-1} and C_{n-j} comparisons, respectively. In this variant, the pivot is chosen with a uniform probability, that is $\pi_{n,j} = 1/n$.

Listing 4. Returns the number of comparisons in the average case corresponding to relation (3)

```
Cproc := proc(n::nonnegint)
  local k;
  option remember;
  if n = 0 then
    0;
  else
    n + 1 + 2*add( Cproc(k), k = 0 .. n - 1 )/n;
  end if;
end proc:
```

As before, the recurrence relation can be implemented as the `Maple` procedure in Listing 4, however, in a course on Analysis of Algorithm it is instructive to solve recurrence (3) in terms of *generating functions* and the extraction of their coefficients. In fact, generating functions have emerged as one of the most popular approaches to the analysis of algorithms (see, for example, [3, 5, 6, 7, 14, 15, 18]). This elegant technique is described in Merlini, Sprugnoli and Verri [11] as the *method of coefficients* and for the sake of clarity we summarize here the main concepts of the method; more details and properties can be found in the original paper.

Let us consider a sequence of numbers $F = (f_0, f_1, f_2, \ldots) = (f_n)_{n\in\mathbb{N}}$; the *generating function* for the sequence F is defined as the formal power series $f(t) = f_0 + f_1 t + f_2 t^2 + \cdots$, where the indeterminate t is arbitrary. Given the sequence $(f_n)_{n\in\mathbb{N}}$, we introduce the *generating function operator* $\mathcal{G}$, which applied to $(f_n)_{n\in\mathbb{N}}$ produces the generating function for the sequence, i.e., $\mathcal{G}(f_n) = f(t)$. A more accurate notation would be $\mathcal{G}_t\,(f_n)_{n\in\mathbb{N}} = f(t)$, this notation is essential when $(f_n)_{n\in\mathbb{N}}$ depends on some parameter. The operator $\mathcal{G}$ is clearly linear. The function $f(t)$ can be shifted or differentiated and two functions $f(t)$ and $g(t)$ can be multiplied. This leads to the following properties for the operator $\mathcal{G}$:

$$
\begin{array}{lrcl}
\text{(linearity)} & \mathcal{G}(\alpha f_n + \beta g_n) & = & \alpha\mathcal{G}(f_n) + \beta\mathcal{G}(g_n) \\
\text{(shifting)} & \mathcal{G}(f_{n+1}) & = & \dfrac{\mathcal{G}(f_n) - f_0}{t} \\
\text{(differentiation)} & \mathcal{G}(nf_n) & = & tD\mathcal{G}(f_n) \\
\text{(convolution)} & \mathcal{G}\left(\displaystyle\sum_{k=0}^{n} f_k g_{n-k}\right) & = & \mathcal{G}(f_n) \cdot \mathcal{G}(g_n)
\end{array}
$$

The inverse operator $[t^n]$ is called the *coefficient operator* and applied to a generating function $f(t)$ gives the coefficient of t^n from $f(t)$, that is, $[t^n]f(t) = f_n$. This operator has properties similar to the previous ones:

$$
\begin{array}{lrcl}
\text{(linearity)} & [t^n](\alpha f(t) + \beta g(t)) & = & \alpha[t^n]f(t) + \beta[t^n]g(t) \\
\text{(shifting)} & [t^n]tf(t) & = & [t^{n-1}]f(t) \\
\text{(differentiation)} & [t^n]f'(t) & = & (n+1)[t^{n+1}]f(t) \\
\text{(convolution)} & [t^n]f(t)g(t) & = & \displaystyle\sum_{k=0}^{n}[t^k]f(t)[t^{n-k}]g(t)
\end{array}
$$

The operator $\mathcal{G}$ can be used to transform a recurrence relation for a sequence $(f_n)_{n\in\mathbb{N}}$ into an equation defining the corresponding generating function $f(t)$; a formula for f_n can then be found by extracting the coefficient $[t^n]f(t)$. In particular, we can state the following *principle of identity*: *Given two sequences* $(f_n)_{n\in\mathbb{N}}$ *and* $(g_n)_{n\in\mathbb{N}}$, *then* $\mathcal{G}(f_n) = \mathcal{G}(g_n)$ *if and only if for every* $n \in \mathbb{N}$ $f_n = g_n$.

The principle states the condition under which we can pass from an identity about elements to the corresponding identity about generating functions. In fact, for two generating functions to be unequal it is sufficient that their coefficient sequences differ by a single element.
In `Maple`, the well-known `gfun` package provides tools for determining and manipulating generating functions, together with many other native functions, as will be shown below.
Coming back to `Quicksort`, let us denote by $C(t) = \sum_{n\geq 0} C_n t^n = \mathcal{G}(C_n)$ the generating function of the sequence $(C_n)_{n\in\mathbb{N}}$. `Maple` can be used at this stage to verify the various steps, some of which are shown below. From recurrence relation (3), we have

$$nC_n = n(n+1) + 2\sum_{k=0}^{n-1} C_k$$

hence

$$\mathcal{G}(nC_n) = \mathcal{G}(n(n+1)) + 2\mathcal{G}\left(\sum_{k=0}^{n-1} C_k\right).$$

Using the `rectodiffeq` function of the `gfun` package to find the generating functions (13):
> *with*(*gfun*):
> *rectodiffeq*($\{f(0) = 1, f(n) = 1\}, f(n), f(t)$);

$$(1-t)\,f(t) - 1 \tag{4}$$

> *solve*((**4**), $f(t)$);

$$-\frac{1}{-1+t} \tag{5}$$

> *series*((**5**), t);

$$1 + t + t^2 + t^3 + t^4 + t^5 + \mathrm{O}\left(t^6\right) \tag{6}$$

> *rectodiffeq*($\{f(0) = 0, f(n) = n\}, f(n), f(t)$);

$$\left(t^2 - 2t + 1\right) f(t) - t \tag{7}$$

> *factor*(*solve*((**7**), $f(t)$));

$$\frac{t}{(-1+t)^2} \tag{8}$$

> *series*((**8**), t);

$$t + 2t^2 + 3t^3 + 4t^4 + 5t^5 + \mathrm{O}\left(t^6\right) \tag{9}$$

> *rectodiffeq*($\{f(0) = 0, f(n) = n^2\}, f(n), f(t)$);

$$\left(-t^3 + 3t^2 - 3t + 1\right) f(t) - t^2 - t \tag{10}$$

> *factor*(*solve*((**10**), $f(t)$));

$$-\frac{t\,(t+1)}{(-1+t)^3} \tag{11}$$

> *series*((**11**), t);

$$t + 4t^2 + 9t^3 + 16t^4 + 25t^5 + \mathrm{O}\left(t^6\right) \tag{12}$$

By applying the properties of the operator and the well known generating functions (as seen above, where the `Maple` package `gfun` is used)

$$\mathcal{G}(1) = \frac{1}{1-t}, \quad \mathcal{G}(n) = \frac{t}{(1-t)^2}, \quad \mathcal{G}\left(n^2\right) = \frac{t+t^2}{(1-t)^3}, \tag{13}$$

we have:

$$tC'(t) = \frac{t+t^2}{(1-t)^3} + \frac{t}{(1-t)^2} + \frac{2t}{1-t}C(t)$$

$$C'(t) = \frac{2}{(1-t)^3} + \frac{2}{1-t}C(t). \tag{14}$$

In order to solve the differential equation (14) we proceed by finding a solution of the associated homogeneous equation:

$$\rho'(t) = \frac{2\rho(t)}{1-t}, \qquad \frac{\rho'(t)}{\rho(t)} = \frac{2}{1-t}$$

and we have

$$\ln\rho(t) = -2\ln(1-t), \qquad \rho(t) = \frac{1}{(1-t)^2}.$$

Then we differentiate the ratio between $C(t)$ and $\rho(t)$:

$$((1-t)^2C(t))' = (1-t)^2C'(t) - 2(1-t)C(t) =$$
$$= (1-t)^2\left(C'(t) - 2\frac{C(t)}{1-t}\right) = \frac{2}{1-t}.$$

Finally we have:

$$(1-t)^2C(t) = -2\ln(1-t) + k, \quad C(0) = C_0 = 0,$$

therefore we have to choose $k = 0$ and

$$C(t) = \frac{2}{(1-t)^2}\ln\frac{1}{1-t} =$$
$$= 2\,t + 5\,t^2 + \frac{26}{3}t^3 + \frac{77}{6}t^4 + \frac{87}{5}t^5 + \frac{223}{10}t^6 + O\left(t^7\right).$$

In `Maple`, equation (14) can be solved by using the `dsolve` function, but care must be taken in interpreting the result, since $I\pi = \ln(-1)$ appears in the result of the call to **`dsolve`**. One may clean the result up by using **`evalc`** and the assumption that $0 < t < 1$. Furthermore, the solution can be developed by using the `Maple` function `series`.

> $dsolve(\{C(0) = 0, diff(C(t), t) = \frac{2}{(1-t)^3} + \frac{2\cdot C(t)}{1-t}\}, C(t))$ assuming $t < 1$;

$$C(t) = \frac{-2\ln(-1+t) + 2\,\mathrm{I}\pi}{(-1+t)^2} \tag{15}$$

> $map(evalc, \%)$ assuming $t :: RealRange(Open(0), Open(1))$;

$$C(t) = -\frac{2\ln(1-t)}{(-1+t)^2} \tag{16}$$

The value of C_n can be found by extracting the n-th coefficient from $C(t)$. We have:

$$C_n = [t^n]\frac{2}{(1-t)^2}\ln\frac{1}{1-t} =$$
$$= 2[t^n]\frac{1}{(1-t)}\mathcal{G}(H_n) = 2\sum_{k=0}^{n}H_k, \quad H_0 = 0,$$

where

$$H_n = \sum_{k=1}^{n}\frac{1}{k} \approx \ln n + \gamma,$$

with $\gamma = 0.577215\cdots$ the Euler's constant, denotes the nth harmonic number having generating function

$$\mathcal{G}(H_n) = \frac{1}{(1-t)} \ln \frac{1}{1-t};$$

in fact, $\mathcal{G}(H_n)$ can be computed by the convolution property of the operator $\mathcal{G}$ applied to the sequences $f_n = \frac{1}{n}$ and $g_n = 1$, with $\mathcal{G}(f_n) = \ln \frac{1}{1-t}$ and $\mathcal{G}(g_n) = \frac{1}{1-t}$. On the other side:

$$D\mathcal{G}(H_n) = \frac{1}{t}\mathcal{G}(nH_n) = \frac{1}{(1-t)^2} \ln \frac{1}{1-t} + \frac{1}{(1-t)^2}$$

and consequently

$$C_n = 2[t^n]\frac{1}{t}\mathcal{G}(nH_n) - 2[t^n]\frac{1}{(1-t)^2}.$$

Therefore, the solution of the recurrence relation (3) with $C_0 = 0$ is:

$$C_n = 2(n+1)(H_{n+1} - 1), \quad n > 0 \tag{17}$$

To solve recurrence (3) directly in `Maple`, it is convenient to compute the difference $nC_n - (n-1)C_{n-1}$ which yields the alternative formula

$$nC_n = (n+1)C_{n-1} + 2n, \tag{18}$$

which can be solved by the `rsolve` procedure (see below; as before, care must be taken in interpreting the result, which involves the Ψ function satisfying the relation $\Psi(n+1) = H_{n+1} - 1/(n+1) - \gamma$).
> *rsolve*$(\{n \cdot C(n) = (n+1) \cdot C(n-1) + 2 \cdot n, C(0) = 0\}, C(n))$;

$$2\,(n+1)\,\Psi(n+1) + 2\gamma n + 2\gamma - 2n \tag{19}$$

> *simplify*(*subs*(Psi$(n+1) =$ harmonic$(n+1) - 1/(n+1) -$ gamma, **(19)**));

$$2\,(\text{harmonic}(n+1) - 1)\,(n+1) \tag{20}$$

The average number S_n of exchanges before the pointers i and j intersect in Listing 1 satisfies a similar recurrence relation:

$$S_n = \frac{n-2}{6} + \frac{1}{n}\sum_{j=1}^{n}(S_{j-1} + S_{n-j}), \quad n \geq 2, \quad S_0 = S_1 = 0. \tag{21}$$

The reasoning is similar to the one for comparisons, but the computation of the average number of exchanges during the partitioning stage is less trivial. In this case we have to prove that, if j is the pivot, then $\frac{n-j}{n-1}(j-1)$ exchanges are made on the average and consequently $\frac{1}{n}\sum_{j=1}^{n}\frac{n-j}{n-1}(j-1) = \frac{n-2}{6}$ in total. In order to prove this result, let p_k^j be the probability to make k exchanges during the partitioning stage when the pivot is j (in particular, before the pointers intersect). The value p_k^j can be found by observing that the number of permutations having the value j in the last position, k elements greater than j in the first $j-1$ positions and k elements smaller than j in the other $n-j$ positions, is given by:

$$\binom{n-j}{k}\binom{j-1}{k}(j-1)!\,(n-j)!$$

Finally, dividing by $(n-1)!$ we find the value of p_k^j. Therefore the average number of exchanges when the pivot is j is given by $\sum_{k\geq 0} kp_k^j$ and

$$\sum_{k\geq 0} k\binom{j-1}{k}\binom{n-j}{k}\binom{n-1}{j-1}^{-1} = \binom{n-1}{j-1}^{-1}\sum_{k\geq 0} k\binom{j-1}{k}\binom{n-j}{k} =$$

$$= \binom{n-1}{j-1}^{-1}\sum_{k\geq 0}\binom{n-j}{k}\binom{j-2}{k-1}(j-1) =$$

$$= (j-1)\binom{n-1}{j-1}^{-1}\sum_{k\geq 0}\binom{n-j}{k}\binom{j-2}{j-1-k} = (j-1)\binom{n-1}{j-1}^{-1}\binom{n-2}{j-1} =$$
$$= (j-1)\frac{(j-1)!\,(n-j)!}{(n-1)!}\frac{(n-2)!}{(j-1)!\,(n-j-1)!} = \frac{(n-j)(j-1)}{n-1},$$

where we used the Vandermonde formula:

$$\sum_{k\geq 0}\binom{r}{k}\binom{s}{n-k} = \binom{r+s}{n} \tag{22}$$

The average number of exchanges, before the pointers intersect, during the partitioning stage can finally be determined as follows:

$$\frac{1}{n}\sum_{j=1}^{n}\frac{n-j}{n-1}(j-1) = \frac{1}{n(n-1)}\sum_{j=1}^{n}(nj-j^2-n+j) = \frac{n-2}{6};$$

this result can be easily checked by using the `Maple` function `sum`.
Finally, observe that in order to have the average number of exchanges performed by the algorithm, before and after the pointers intersect, the n exchanges that are carried out to put the pivot in its definitive position during the n partitioning must be added to the value S_n.

Listing 5. Returns the number of exchanges in the average case, before the pointers intersect, corresponding to relation (21)

```
Sproc := proc(n::nonnegint)
  local k;
  option remember;
  if n <= 1 then
    0;
  else
    1/6*n - 1/3 + 2*add(Sproc(k), k = 0 .. n - 1)/n;
  end if;
end proc:
```

In order to solve recurrence relation (21) with generating functions, we need to take care of the initial condition in $n = 1$ with a Kronecker delta and consequently consider the relation

$$6nS_n = n(n-2) + 12\sum_{k=0}^{n-1}S_k + \delta_{n,1};$$

in fact, in this way, the *principle of identity* stated before can be applied and the recurrence relation can be transformed into the following differential equation

$$S'(t) = \frac{t^2(3-t)}{6(1-t)^3} + \frac{2}{1-t}S(t) \tag{23}$$

which has the solution

$$S(t) = \frac{1}{3}\frac{1}{(1-t)^2}\ln\frac{1}{1-t} + \frac{t(t^2-3t-6)}{18(1-t)^2}.$$

This can be easily checked with `Maple`, by considering as before $I\pi = \ln(-1)$ (see below). Therefore we have:

$$S_n = \frac{1}{3}(n+1)(H_{n+1}-1) + \frac{1}{18}\left([t^{n-2}]\frac{t}{(1-t)^2} - 3[t^{n-1}]\frac{t}{(1-t)^2} - 6[t^n]\frac{t}{(1-t)^2}\right) =$$
$$= \frac{1}{3}(n+1)(H_{n+1}-1) + \frac{1}{18}\left(n-2-3(n-1)-6n\right) = \frac{1}{3}(n+1)H_{n+1} - \frac{7}{9}n - \frac{5}{18},$$

that is,

$$S_n = \frac{1}{3}(n+1)(H_{n+1} - \frac{7}{3}) + \frac{1}{2}. \tag{24}$$

Using Maple to solve equation (23):

> $dsolve(\{S(0) = 0, diff(S(t), t) = \frac{t^2 \cdot (3-t)}{6 \cdot (1-t)^3} + \frac{2 \cdot S(t)}{1-t}\}, S(t));$

$$S(t) = \frac{t^3 + 6\,\mathrm{I}\pi - 3t^2 - 6\ln(-1+t) - 6t}{18\,(-1+t)^2} \tag{25}$$

> $map(evalc, \mathbf{(25)})$ assuming $t < 1$;

$$S(t) = \frac{\frac{t^3}{18} - \frac{t^2}{6} - \frac{\ln(1-t)}{3} - \frac{t}{3}}{(-1+t)^2} \tag{26}$$

The solutions (17) and (24) can be implemented as `Maple` functions (see Listing 6).

Listing 6. Maple functions for formulas (17) and (24)

```
C := n ->
if n = 0 then
    0;
else
    2*(n + 1)*(harmonic(n + 1) - 1);
end if;

S := n ->
if n = 0 then
    0;
elif n = 1 then
    0;
else
    1/3*(n + 1)*(harmonic(n + 1) - 7/3) + 1/2;
end if;
```

Finally, we can observe that the ratio $\frac{C_n}{S_n+n}$ between the average number of comparison and the average number of exchanges tends to 6 as n goes to infinity; however, the convergence to this value is very low, for example it is ≈ 5.9568 for $n = 10^{100}$.

3 Simulating Quicksort

If we want to check formulas (3) and (21) for small values of n, we can execute `QuickSort` over all the $n!$ permutations of the elements $1, \cdots, n$ and use a counter to record the number of key comparisons and exchanges. The average values can then be found by summing the results of these executions and then by dividing by $n!$. This exhaustive execution of the algorithm we believe is important to clarify to students what it really means to study an algorithm in the average case. Listing 7 illustrates a `Maple` implementation of the algorithms `Quicksort` that sort an array V from index ℓ to index r and, at the same time, computes the number of comparisons and exchanges performed during execution. These values are stored in the two variables `comp` and `exch`. Although Listing 7 contains small changes compared to Listing 1, it is important from an educational point of view to show students where the counters must be inserted, so that the simulation in the exhaustive case gives the same results as the mathematical formulas. For example, from the author's experience,

a mistake students often make, is forgetting to increment the counters before the two inner `while` loops.

Listing 7. The modified version of Quicksort in Maple

```
MQuicksort := proc(v::Array, ℓ::nonnegint, r::nonnegint)
  local comp, exch, i, j, p;
  comp := 0;
  exch := 0;
  if 0 <= r - ℓ then
    p := v[r];
    i := ℓ - 1;
    j := r;
    while i < j do
        i := i + 1;
        comp := comp + 1;
        while v[i] < p do
            i := i + 1;
            comp := comp + 1;
        end do;
        j := j - 1;
        comp := comp + 1;
        while ℓ <= j and p < v[j] do
            j := j - 1;
            comp := comp + 1;
        end do;
        if i < j then
            (v[i],v[j]) := (v[j],v[i]);
            exch := exch + 1;
        end if;
    end do;
    (v[i],v[r]) := (p,v[i]);
    exch := exch + 1;
    (comp,exch) := (comp,exch) + MQuicksort(v, ℓ, i - 1);
    (comp,exch) := (comp,exch) + MQuicksort(v, i + 1, r);
  end if;
  return (comp,exch);
end proc:
```

For small dimensions, we can use an exhaustive strategy and use algorithm in Listing 7 over all the $n!$ permutations of the numbers $1, \cdots, n$ in order to check formulas (3) and (21). Procedure `exhaustive_test` in Listing 8 does exactly this for arrays of length n, by using the `permute` procedure of the `Maple` package `combinat`. Alternatively, instead of generating all the permutations together, the `nextperm` procedure of the same package could be used to generate the lexicographic successor of a given permutation.

Listing 8. Returns the average value of the comparisons and exchanges performed by `QuickSort` over all the $n!$ permutations.

```
exhaustive_test := proc(n::nonnegint)
  local permutations, p, cs, c, s, i, mean;
```

```
  permutations := combinat[permute](n);
  for i to n! do
    p := Array(permutations[i]);
    cs := MQuicksort(p, 1, n);
    c[i] := cs[1];
    s[i] := cs[2];
  end do;
  mean[1] := add(c[i], i = 1 .. n!)/n!;
  mean[2] := add(s[i], i = 1 .. n!)/n!;
  return mean[1], C(n), Cproc(n), mean[2], S(n)+n, Sproc(n)+n;
end proc:
```

A final numerical check is illustrated below.

```
for i from 0 to 6 do
    i, exhaustive_test(i);
end do;
```

$$0, 0, 0, 0, 0, 0, 0$$
$$1, 2, 2, 2, 1, 1, 1$$
$$2, 5, 5, 5, 2, 2, 2$$
$$3, \frac{26}{3}, \frac{26}{3}, \frac{26}{3}, \frac{19}{6}, \frac{19}{6}, \frac{19}{6}$$
$$4, \frac{77}{6}, \frac{77}{6}, \frac{77}{6}, \frac{53}{12}, \frac{53}{12}, \frac{53}{12}$$
$$5, \frac{87}{5}, \frac{87}{5}, \frac{87}{5}, \frac{86}{15}, \frac{86}{15}, \frac{86}{15}$$
$$6, \frac{223}{10}, \frac{223}{10}, \frac{223}{10}, \frac{1279}{180}, \frac{1279}{180}, \frac{1279}{180} \tag{27}$$

The procedure can easily be modified so that it also returns the permutations that correspond to the minimum and maximum number of comparisons, thus allowing students to understand which configurations correspond to the results of relations (1) and (2). This is done in Listing 9 and the corresponding execution with $n = 4$ is shown below, in agreement with the values $C_4^{\min} = 12$ and $C_4^{\max} = 14$.

Listing 9. A modified version of the exhaustive_test procedure

```
exhaustive_test1 := proc(n::nonnegint)
  local permutations, i, cs, c, s, cm, sm, ma, mi, p, Pmax, Pmin;
  permutations := combinat[permute](n);
  for i to n! do
    p := Array(permutations[i]);
    cs := MQuicksort(p, 1, n);
    c[i] := cs[1];
    s[i] := cs[2];
  end do;
  cm := add(c[i], i = 1 .. n!)/n!;
  sm := add(s[i], i = 1 .. n!)/n!;
  ma := max([seq(c[i], i = 1 .. n!)]);
```

```
  mi := min([seq(c[i], i = 1 .. n!)]);
  Pmax := Array(0..-1); Pmin := Array(0..-1);
  for i to n! do
    if c[i]=ma then
         Pmax ,= permutations[i];
    elif c[i]=mi then
         Pmin ,= permutations[i];
    end if;
  end do;
  return ['Cmean' = cm, 'Smean' = sm], ['Max' = ma, Pmax],
  ['Min' = mi, Pmin];
end proc:
```

> *exhaustive_test1*(4);

$$\left[\mathit{Cmean} = \frac{77}{6}, \mathit{Smean} = \frac{53}{12}\right], [\mathit{Max} = 14, \\ [[1, 2, 3, 4], [2, 1, 3, 4], [2, 3, 1, 4], [2, 3, 4, 1], [2, 4, 3, 1], \\ [3, 2, 1, 4], [4, 2, 3, 1], [4, 3, 2, 1], \cdots\ 0 .. 7\ \text{Array}]], [\mathit{Min} = 12, \\ [[1, 2, 4, 3], [1, 3, 4, 2], [1, 4, 2, 3], [1, 4, 3, 2], [2, 1, 4, 3], \\ [2, 4, 1, 3], [3, 1, 4, 2], [3, 4, 1, 2], [4, 1, 2, 3], [4, 1, 3, 2], \\ [4, 2, 1, 3], [4, 3, 1, 2], \cdots\ 0 .. 11\ \text{Array}]] \tag{28}$$

When n is large, we cannot generate all the $n!$ permutations of the input array and therefore, in order to check formulas (3) and (21), we have to execute the algorithm over a sample of N permutations. Many important questions arise during the organization of the simulation. First of all, we have to choose the size of the problem: we cannot consider a single value n but different values in a quite large range. A second important point concerns the number of tests: once the dimension of the problem has been fixed, we have to repeat the simulation several times in order to increase our confidence on the results. The third fundamental point concerns data: we have to make precise assumptions on the input data, closely related to the theoretical formula we want to check.

Listing 10. Returns the average value of comparisons and exchanges performed by `QuickSort` over N random permutations.

```
random_test := proc(n::nonnegint, N::nonnegint)
local permutations, p, cs, c, s, i, mean;
for i to N do
    p := Array(combinat[randperm](n));
    cs := MQuicksort(p, 1, n);
    c[i] := cs[1];
    s[i] := cs[2];
end do;
mean[1] := add(c[i], i = 1 .. N)/N;
mean[2] := add(s[i], i = 1 .. N)/N;
map(evalf[5], [mean[1], C(n), Cproc(n), mean[2], S(n)+n, Sproc(n)+n]);
end proc:
```

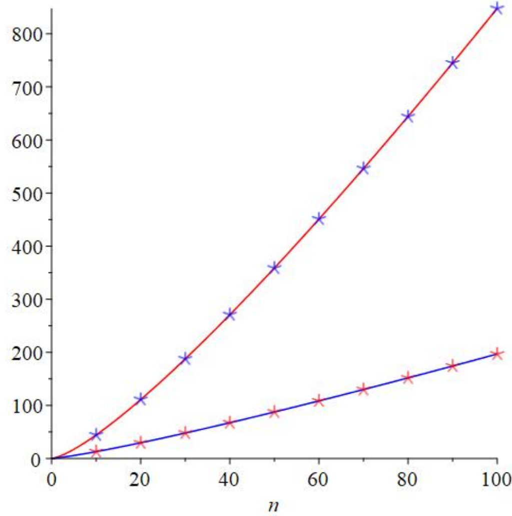

Fig. 1. Average number of comparisons (blue stars) and exchanges (red stars) found with the simulation, compared with the theoretical values (red and blue curves).

In particular, the procedure `random_test` in Listing 10 applies algorithm `QuickSort` to N uniformly generated random permutations of the elements $1, \ldots, n$. The uniform random generation of the permutations is obtained by using the `randperm` procedure of the `Maple` package `combinat`, although we could have used the classical shuffling algorithm, as described by Knuth [7]. The procedure returns the average value of comparison and exchange operations performed by `QuickSort` over N random permutations, compared to values of formulas (3), (17), (21) and (24). A final numerical check is mandatory and is illustrated in Listing (11).

Listing 11. The random test for $n = 10, 20, 30, 40, 50$

```
STEP := 10:
DIMENSION := 50:
for i from STEP by STEP to DIMENSION do
    i, random_test(i, 10*i*STEP);
end do;
```

$$10, [44.416, 44.437, 44.437, 13.022, 13.017, 13.017]$$

$$20, [111.01, 111.11, 111.11, 29.681, 29.684, 29.684]$$

$$30, [187.48, 187.69, 187.69, 47.983, 48.004, 48.004]$$

$$40, [270.87, 270.84, 270.84, 67.388, 67.418, 67.418]$$

$$50, [358.88, 358.92, 358.92, 87.651, 87.653, 87.653] \tag{29}$$

Finally, Figure 1 illustrates, with blue and red stars, the results of several simulations performed with $n = 10, 20, \cdots, 100$ and $N = 10 \cdot n$, compared with the theoretical results given by formulas (17) and (24), evidenced in red and blue, respectively. The figure is obtained with the `Maple` function `plot`.

We wish to conclude this paper by observing that the results of the simulation could also be checked by using a statistical test, based for example on the *central limit theorem* (see, e.g., [2, 3]). This theorem asserts that if we have a sum of random variables X_i independent and identically distributed, with mean μ and variance σ^2, then, regardless of the distribution, when the sample size goes to infinity the sum tends to be distributed as a normal random variable. In formulas:

$$\frac{\overline{X}_N - \mu}{\sigma}\sqrt{N} \sim \mathrm{Normal}(0, 1) \tag{30}$$

where $\overline{X}_N = \sum_{i=1}^{N} X_i/N$ is the sample mean. In the simulation of `Quickort`, X_i is the number of interesting operations, i.e., comparisons and exchanges, during the i-th execution and consequently $\overline{X}_N$ is one of the average values found by the `random_test` procedure. On the other hand, μ corresponds to the average theoretical value and σ to the corresponding standard deviation. If the standard deviation is not known we can use the estimated sample by suitably modifying the procedure `random_test` so that in addition to the mean it also returns the variance.

4 Conclusions

In this work it has been shown how `Maple` can be used to teach analysis of algorithm in a Computer Science degree course, both to verify theoretical results and to show, on concrete examples, the meaning of an average case analysis. In fact, according to the experience of the author of this paper, the average case is not easily understood, while the approach presented in this work is appreciated by Computer Science students who, in addition to learning theoretical concepts, learn to implement the algorithms, or data structures, with the aim to perform both an exhaustive and a random simulation of the average case under examination. The example shown in this work was `Quicksort`, a well-known sorting algorithm that is very interesting because its average-case analysis requires a non-trivial use of the mathematical tools typically used in this context, such as generating functions. More sophisticated variants of the algorithm could also be shown, for example, the variant that chooses the pivot as the median of three randomly chosen elements of the array; or, the variant of the algorithm that uses a different sorting algorithm, for example `Insertion Sort`, for sub-arrays of size $n \le m$, thus avoiding recursive calls on small length arrays. The recurrence relations for the average number of comparisons would become

$$C_n = n + 1 + \sum_{j=1}^{n} \frac{(n-j)(j-1)}{\binom{n}{3}} \left(C_{j-1} + C_{n-j}\right)$$

and

$$C_n = \begin{cases} n + 1 + \frac{1}{n}\sum_{j=1}^{n}(C_{j-1} + C_{n-j}) & n > m \\ \frac{1}{4}n(n-1) & n \le m \end{cases},$$

in the first and second case, respectively (see [14]). In general, complex recurrence relations correspond to simulations whose set up requires great attention.

Acknowledgements

The author wishes to thank the reviewers for their very helpful comments. A special thanks to prof. R. M. Corless for his great work editing the `Maple` code.

References

[1] J. Bentley and M. McIlroy. Engineering a sort function. *Software—Practice and Experience*, 23:1249—-1265, 1991.
[2] W. Feller. *An introduction to probability theory and its applications, third ed.* John Wiley, 1968.
[3] P. Flajolet and R. Sedgewick. *Analytic combinatorics*. Cambridge University Press, 2009.
[4] C. A. R. Hoare. Quicksort. *Computer Journal*, 5:10–15, 1962.

[5] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms, Vol. 1, 3rd ed.* Addison-Wesley, Reading, MA, 1997.
[6] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms, Vol.2, 3rd ed.* Addison-Wesley, Reading, MA, 1998.
[7] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching, Vol. 3, 2nd ed.* Addison-Wesley, Reading, MA, 1998.
[8] H. Mahmoud. *Sorting: A Distribution Theory*. John Wiley and Sons, New York, 2000.
[9] C. Martínez and H. Prodinger. Moves and displacements of particular elements in quicksort. *Theor. Comput. Sci.*, 410(21-23):2279–2284, 2009.
[10] C. Martínez and S. Roura. Optimal sampling strategies in quicksort and quickselect. *SIAM J. Comput.*, 31(3):683–705, 2001.
[11] D. Merlini, R. Sprugnoli and M.C. Verri. The Method of Coefficients. *Amer. Math. Monthly* 114 (1), 40–57 (2007).
[12] R. Sedgewick. The analysis of quicksort programs. *Acta Informatica*, 7:327—-355, 1976.
[13] R. Sedgewick. Implementing quicksort programs. *Comm. ACM*, 21:847—-856, 1978.
[14] R. Sedgewick and P. Flajolet. *An Introduction to the Analysis of Algorithms*. Addison-Wesley, Reading, MA, 1996.
[15] R. P. Stanley. *Enumerative Combinatorics*, vol. 1, Wadsworth, Cambridge (1986).
[16] Official site of Maple. http://www.maplesoft.com.
[17] B. Vallèe, J. Clèment, J. A. Fill, and P. Flajolet. The number of symbol comparisons in quicksort and quickselect. In *ICALP*, volume 1, pages 750–763, 2009.
[18] H. Wilf. *Generatingfunctionology*, Academic Press, San Diego (1990).