Computer Science Education
2004, Vol. 14, No. 3, pp. 235–248

Taylor & Francis
Taylor & Francis Group

Teaching Algorithm Efficiency at CS1 Level: A Different Approach

Judith Gal-Ezer, Tamar Vilner, and Ela Zur
The Open University of Israel, Tel-Aviv, Israel

ABSTRACT

Realizing the importance of teaching efficiency at early stages of the program of study in computer science (CS) on one hand, and the difficulties encountered when introducing this concept on the other, we advocate a different didactic approach in the introductory CS course (CS1). This paper describes the approach as it is used at the Open University of Israel (OUI).

The OUI, a distance teaching institution with no prior educational requirements, runs a full-fledged CS program. Because of its open door policy, the dropout rate, especially in introductory courses, as well as the percentage of failures, is very high. Using the new approach has raised the percentage of students who pass the course. The new approach advocates integrating three-part questions which ask not only to identify the algorithmic problem that a given algorithm solves and to analyze its complexity, but also to design a new algorithm that performs the same task, while increasing efficiency by an order of magnitude, not only by a constant factor. The research we conducted to examine the implications of using this approach is described here.

1. BACKGROUND

The design of efficient algorithms to solve algorithmic problems is one of the most important research fields within computer science (Gal-Ezer & Zur, 2002b; Ginat, 1996, 2001; Harel, 1992). Algorithms are the spirit of computing, and good algorithm design is crucial to the performance of all software systems, as is the ability to select algorithms appropriate for specific purposes or recognizing the possibility that sometimes no suitable algorithm exists. Algorithms are therefore also central to

Address correspondence to: Judith Gal-Ezer, The Open University of Israel, 108 Ravutski St., Raanana, Israel. Tel.: +972-9-7782202. Fax: +972-9-7780642. E-mail: galezer@openu.ac.il

10.1080/0899340042000302736

computer science education. The study of algorithms gives the learner insight into the problems involved in providing techniques for solutions that are independent of programming languages, or other implementational aspects.

From *Computing Curricula 2001* (IEEE Computer Society/ACM Task Force, 2001), we learn that a large part of the core and elective course material is devoted to algorithms. Efficiency and complexity are pervasive themes throughout the study of algorithms; however, they are difficult concepts to conceive. Students first become familiar with these concepts in the introductory course. When teaching CS1 at our university, we found that students had difficulty conceiving the notion of algorithm efficiency and its implementation. We sought a new didactic approach, different to that found in existing textbooks for teaching efficiency, to provide students with better insight into the subject.

The OUI is a distance education institution, open to all those who wish to study a single course or a number of courses, or to pursue a full program of study towards a Bachelor's degree. Enrollment does not require matriculation or any other certificate from an educational institution. Because of its open admissions policy, the first courses serve as ''the proof of the pudding''; these courses actually help students to find out whether they are capable of coping with academic studies. Many students, unfortunately, fail. Computer science is known as a difficult field of study, one that requires a great deal of mathematics, and indeed the percentage of dropouts in the introductory courses is relatively high – in some courses reaching about 50%. In addition, less than 60% of those who take the final exam pass it. A great deal of effort is devoted to improving these statistics without lowering the high academic standards.

2. INTRODUCTORY COURSE IN CS: A CASE STUDY

In the OUI, CS1 is based on the book *Fundamentals of Computing I*, by Tucker et al. (1995), and on a study guide we developed. The course is similar to introductory courses given in other universities, including the topics recommended in *Computing Curricula 2001*: Basic logic, algorithms and problem solving, fundamental data structures, fundamental programming constructs, recursion, fundamental computing algorithms, basic computability,

etc. In our study guide, we added exercises, examples and explanations relating to themes that were not included in the book, the most important of which is efficiency. The language introduced in the course is C++, but mainly the procedural facet of the language, with very little space devoted to the object-oriented facet.

Algorithm complexity is measured in terms of space and time. Space complexity is measured by elements such as the number and size of the data structures used; while time complexity is measured by the number of elementary actions carried out during the execution of the algorithm.

Since the concept of complexity is essential, we recommend introducing it as early as possible (see also Ginat, 1996, 2001; Linn, 1985; Linn & Clancy, 1992). The relatively early introduction of the concept encourages students to consider alternative designs of algorithms, to analyze various algorithms, and to formulate them correctly.

However, such early introduction may lead to difficulties: the problems discussed at early stages of the introductory course are almost always toy problems, making it difficult to convince students that a more efficient algorithm is indeed needed. Also, the analysis of algorithm efficiency requires mathematics that students are not always familiar with when they take the introductory course. Misconceptions are encountered (Gal-Ezer & Zur, 2002a, 2002b) for instance, students often bring up the myth of the speed of the computer, saying that computers are so incredibly fast that there is no real time problem. This belief is, of course, groundless: time is crucial in almost every use of the computer. Many examples can be found to show that whatever the speed of computers is or will be, there is still importance in speeding up the execution of algorithms (see, e.g., Harel, 1992). Moreover, an algorithm might just be too expensive and thus unacceptable. One well-known example is that of the traveling salesman. The traveling salesman has to visit each of the cities in a given network before returning to the starting point, using the cheapest route. Though this is a very important issue in the field of computer networks, no algorithm has yet been found that solves the problem in reasonable time. This is one of the examples we use in the course to illustrate the importance of efficiency in designing algorithms.

Despite the obstacles mentioned above, we still think it is important to introduce efficiency gradually in the introductory course, thus enhancing, almost from the beginning, a deeper perspective of computer science. In our

course, we first present algorithmic problems with unreasonable algorithmic solutions, because we believe that this will increase motivation to learn the topic. Students begin to understand how important it is to be able to analyze the complexity of an algorithm and to realize that some algorithms are unreasonable even if we have a very fast computer. We then explain how to measure efficiency, and how to compare the complexity of different algorithms.

We discuss linear and binary search and introduce the big-*O* notation. We explain how critical it is to reduce the running time of algorithms by an order of magnitude and not only in terms of a constant factor. The concepts of average-case, best-case and worst-case are introduced, as well as the robustness of big-*O*. Bubble sort and merge-sort are taught and their complexity is analyzed; exponential algorithms and their inapplicability are discussed.

After teaching the course, we discovered that students are usually able to analyze the algorithm and compute its efficiency, but find it *very* difficult to design an algorithmic solution to a given problem that improves the efficiency by an order of magnitude. So we tried a new approach to teaching efficiency in CS1, without adding material to the already overloaded course. We changed the pedagogy of the subject by applying a series of carefully designed three-part exercises. The exercises vary from year to year, but all follow the same general scheme. Each exercise presents an algorithm (or a program); in the first part, we ask students to identify the algorithmic problem that the algorithm solves; in the second part, the students are asked to analyze the complexity of the algorithm; and in the third part, they are asked to design a more efficient algorithm (order-of-magnitude improvement) that performs the same task that the given algorithm performed. It is likely that this type of question has been used in CS courses occasionally, though not necessarily systematically.

For each problem, three questions are given:

(a) What task does the function perform? Explain **briefly** what the function does in general terms, not how it executes the task.
(b) What is the time complexity of the function?
(c) Write a function which performs the same task but which is an order-of-magnitude (not a constant factor) improvement in time complexity.

A function with greater (time or space) complexity will not get full credit.

Four examples of exercises are presented below:

Example 1

Suppose *a* is a given array of length *n*. Consider the following function:

```
int something (int a[n])
{
  int temp = 0, i, j;
  for (i = 0; i < n; i++)
    for (j = i + 1; j < n; j++)
      if (abs (a[j] − a[i]) > temp)
        temp = abs (a[j] − a[i]);
  return temp;
}
```

Answers, Example 1:

(a) The function finds the maximum difference between two values in the array.
(b) $O(n^2)$.
(c) Finding the maximum and minimum values in the array and calculating the difference between them. Therefore the problem will be solved in $O(n)$.

Example 2

Suppose *a* and b are given arrays of length *n*. *a* is sorted in increasing order, and *b* is not sorted. Consider the following function:

```
int what(int a[ ], int b[ ], int &i, int &j)
{
  for (j = 0; j < n; j++)
    for (i = 0; i < n − 1; i++)
      if (b[j] == a[i] + a[i+1])
        return 1;
  return 0;
}
```

Answers, Example 2:

(a) The function checks whether a value in array *b* equals the sum of two consecutive values in array *a*.
(b) $O(n^2)$.

(c) For each value in array b, a variation of a binary search needs to be carried out in array a, therefore the problem will be solved in $O(n \log n)$.

Example 3

Suppose a is a given array of length n. Consider the following function:

```
int something (int a[n])
{
  for (i = 0; i < n; i++)
    for (j = i + 1; j < n; j++)
      if (a[i] == a[j])
        return 0;
  return 1;
}
```

Answers, Example 3:

(a) The function checks whether all the values stored in the array a are different.
(b) $O(n^2)$.
(c) First, sort the array a, and then pass for finding two adjust cells with the same value. The sort takes $O(n \log n)$, and the passing $O(n)$. Therefore the problem will be solved in $O(n \log n)$.

Example 4

Suppose a is a given array of length n. Consider the following function:

```
int something (int a[n])
{
  int i, j, temp;
  for (i = 0; i < n; i++)
    if (a[i]%2 == 0)
    {
      temp = a[i];
      for (j = i; j > 0; j--)
        a[j] = a[j - 1];
      a[0] = temp;
    }
}
```

Answers, Example 4:

(a) The function rearranges the array *a* that all the even values will be at the first cells, and the odd values will be at the last cells.
(b) $O(n^2)$.
(c) Pass on the array from both sides, and make the necessary exchanges. The following function makes that in $O(n)$:

```
int something (int a[n])
{
   int head = 0, tail = n - 1, temp;
   while (head < tail)
   {
     if (a[head]%2 == 0)
       head++;
     else if (a[tail]%2! = 0)
            tail--;
          else
          {
            temp = a[head];
            a[head] = a[tail];
            a[tail] = temp;
            head++;
            tail--;
          }
   }
}
```

This kind of exercise is rarely found in textbooks (see, e.g., the excellent text, *Introduction to Algorithms*, Cormen et al., 1990). In most texts, the best algorithm is given and its complexity is analyzed, whereas in our method, the student is exposed to a number of possible solutions to the same problem. In this way, students internalize the concept of a more efficient algorithm. The study guide as well as the assignments both emphasize this approach.

After acquiring practical experience teaching the course over several semesters, we conducted a study that investigated how students internalized time and space efficiency after solving exercises of the kind described above.

We posed two main research questions:

1. To what extent are students successful in analyzing the complexity of a given function?
2. To what extent are students successful in designing a better (more efficient) algorithm in terms of order of magnitude.

3. THE STUDY

Our study was carried out with 189 CS students during the spring semester of 2002. The students took the introductory course (based on Tucker et al. as described above). On their final examination, the students were given the following problem, with questions of the type they had practiced during the course:

Suppose a is a given array of length $n - 1$. Each element of a is an integer between 1 and n. All the elements of a are different. Consider the following function:

```
int something (int a[n −1])
{
  int i, j, flag;
  for (j = 1; j <= n; j++)
  {
    flag = 0;
    for (i = 0; i < n − 1; i++)
    {
      if (a[i] == j)
      {
        flag = 1;
        break;
      }
    }
    if (!flag)
      return j;
  }
  return −1;
}
```

(a) What task does the function perform? Explain **briefly** what the function does in general terms, not how it executes the task.
(b) What is the time complexity of the function?

(c) Write a function which performs the same task but which is an order-of-magnitude (not a constant factor) improvement in time complexity.

A function with greater (time or space) complexity will not get full credit.

4. RESULTS

The given function returns a missing value (the integer between 1 and n missing in the given array a). Of 189 students, 120 gave the right answer to the first part of the question; that is, 63% correctly described the task that the function performs.

A majority of the students (168 out of 189–89%) correctly identified the complexity of the given function as $O(n^2)$. What is interesting here, is that 26% of the students who did *not* know what the function does, were able to analyze its efficiency. Our feeling is that since the complexity of "for loops" was discussed very thoroughly during the course, students were able to analyze complexity even without understanding what task the algorithm performs.

Regarding the third part of the question, things are more complicated. Here we found a variety of answers:

The best solution uses the formula for the sum of an arithmetic series (this is given in the text, and is studied in the course) in order to find the sum of the arithmetic series from 1 to $n-1$, then find the sum of the array's elements; and the difference between the two is obviously the missing integer. The time efficiency of this algorithm is $O(n)$, while the space efficiency is constant $O(1)$. This is the optimal solution:

```
int miss_num1 (int a[ ])
{
  int sum, result = 0, i;
  sum = n* (n + 1)/2;
  for (i = 0; i < n − 1; i++)
    result += a[i];
  return (sum − result);
}
```

A very similar solution is adding up the arithmetic series (without using the formula). This is still a good solution; the time complexity here is $2n$.

```
int miss_num2 (int a[ ])
{
  int sum = 0, result = 0, i;
  for (i = 1; i <= n; i++)
    sum += i;
  for (i = 0; i < n − 1; i++)
    result += a[i];
  return (sum − result);
}
```

There was also a quite different solution, using an additional array, marking the elements given in the original array. The unmarked element is the missing one. This solution reduces the time complexity to $O(n)$, but adds space requirements of $O(n)$.

```
int miss_num3 (int a[ ])
{
  int temp[n] = {0};
  int i, num;
  for (i = 0; i < n − 1; i++)
  {
    num = a[i];
    temp[num] = 1;
  }
  for (i = 1; i < n; i++)
    if (!temp[i])
      return i;
  return −1;
}
```

Another interesting solution used merge-sort, and then going through all the elements to check which is missing. This reduced the complexity to $O(n \log_2 n)$.

```
int miss_num4 (int a[ ])
{
  merge_sort(a); // calling the function
                 // which uses
                 // merge-sort to sort
                 // the given array.
```

```
    for (int i = 0; i < n - 1; i++)
    {
      if (a[i] != i + 1)
        return i + 1;
    }
    return -1;
}
```

Another variation of the solution above, but a more expensive one, was using merge-sort and then **n times** binary search, not exploiting the advantage merge-sort provided. This yielded a $2n \log_2 n$ algorithm.

Finally the solution that did not reduce time complexity at all was one that used bubble-sort and then checking which element is missing.

```
int miss_num6 (int a[])
{
  bubble_sort(a); //calling the function
                  // which uses
                  // bubble-sort, to
                  // sort the array.
     for (int i = 0; i < n - 1; i++)
     {
       if (a[i]!= i + 1)
         return i + 1;
     }
     return -1;
}
```

There were additional incorrect solutions that we will not mention here.

It is worth mentioning that in our analysis, we used item discrimination (Linn, 1989) (biserial correlation), which provides a relatively accurate estimate of how well the item can be expected to discriminate at some point on the ability scale. A biserial correlation between 0.35 and 0.7 is considered good discrimination. This question had a discrimination index of 0.6, which indicates that the question discriminated very well. The distribution of the various solutions is given in Table 1, and is shown graphically in Figure 1.

We also observed that on the exam, students used patterns or templates of known algorithmic problems, or problems they had come across during the

Table 1. Distribution of Solutions.

Percentage	Number of students	Solution
1%	2	Summing the array and using the arithmetic series formula (n)
2%	4	Summing the array and summing the arithmetic series ($2n$)
17%	33	Using of another array (n time $+ n$ space)
32%	60	Merge-sort and linear search ($n \log_2 n$)
6%	12	Merge-sort and n binary searches ($2n \log_2 n$)
5%	10	Complexity of n^2
37%	68	Incorrect solution

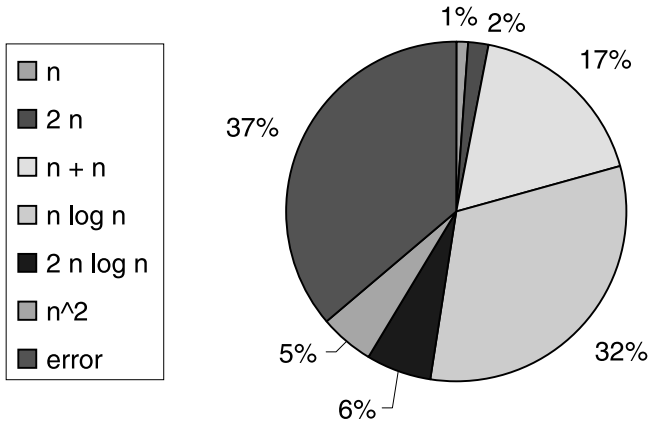

Fig. 1. Graphic distribution of solutions.

course. This may be the reason that 38% of the students used merge-sort and binary search. Indeed, Ginat (2001) advocates the use of patterns when teaching efficiency.

We were somewhat disappointed that only 3% of the students used the sum of an arithmetic series, though it is taught when dealing with bubble sort. It turns out that students avoid using mathematical analysis even when they are familiar with the material, and despite the fact that Tucker integrates mathematics when necessary, to motivate the students to learn the mathematical material needed, not many of them actually make use of this.

5. CONCLUSIONS

The answers to the two research questions we posed were:

1. Most of the students who took the exam (89%) were able to analyze the complexity of a given function (despite the fact that only 63% were able to identify the task that it performs).
2. Designing a more efficient algorithm by order of magnitude was much more difficult for them, but still, more than half of them (58%) were able to design a more efficient algorithm.

As we have noted, about 45% of the students enrolled in the course used to fail the final examination. When grading the examinations we realized that the students' Achilles Heel was the question relating to efficiency. Integrating the three-part questions into the assignments that students submit during the semester seems to improve the results on the final exam. More than 66% of the students who took the exam described here passed.

Thus we recommend using three-part questions when teaching algorithms and algorithm efficiency in CS1 in order to enable students to get more insight into and understanding of this basic issue. We plan to use this same approach in the data structures course that follows CS1.

Basically, we were pleased to see that most of the students perceived the notion of efficiency correctly. It is worth noting that the question was given in a classroom exam, not a take-home exam, which makes the students' achievements even more impressive. We believe that our approach can help students to internalize the concept of algorithm efficiency, and reduce the number of failures in the course.

REFERENCES

Cormen, T.H., Leiserson, C.E., & Rivest, R.L. (1990). *Introduction to algorithms.* Cambridge, MA: MIT Press.

Gal-Ezer, J., & Zur, E. (2002a). The concept of 'algorithm efficiency' in the high school CS curriculum. In proceedings of the *32nd ASEE/IEEE Frontiers in Education Conference*, November.

Gal-Ezer, J., & Zur, E. (2002b). The efficiency of algorithms – misconceptions. *Computers and Education*, *38*(4), 319–329.

Ginat, D. (1996). Efficiency of algorithms for programming beginners. In *Proceedings of the 27th ACM Computer Science Education Symposium* (pp. 256–260). New York: ACM Press.

Ginat, D. (2001). Early algorithm efficiency with design patterns. *Computer Science Education*, *11*(2), 89–109.

Harel, D. (1992). *Algorithmics: The spirit of computing* (2nd ed.) Reading, MA: Addison-Wesley.

IEEE Computer Society/ACM Task Force. (2001). *Computing curricula 2001* (CC-2001) [On-line]. Available: http://www.computer.org/education/cc2001/final

Linn, M.C. (1985). The cognitive consequences of programming instruction in classrooms. *Educational Researcher*, *14*(5), 29.

Linn, M.C., & Clancy, M.J. (1992). The case for case studies of programming problems. *Communications of the ACM*, *35*(3), 121–132.

Linn, R.L. (1989). *Educational measurement* (3rd ed.). New York: Macmillan.

Tucker, A.B., Bernat, A.P., Bradley, W.J., Cupper, R.D., & Scragg, G.W. (1995). *Fundamentals of computing I*. New York: McGraw-Hill.