An Analysis of Algorithms Laboratory Utilizing the Maximum Segment Sum Problem

Robert McCloskey
John Beidler
University of Scranton
{mccloske, beidler}@cs.uofs.edu

Abstract

This paper describes a laboratory/homework exercise, appropriate for the traditional CS 2 or Data Structures & Algorithms course (CS 7) [1], that gives students practice in analyzing algorithms to determine their asymptotic running times as well as in recognizing the relationship between an algorithm's asymptotic running time and the execution time of a program implementing it. The exercise utilizes the maximum segment sum problem, which, we argue, is a good alternative to sorting, the problem that is probably most often used in exercises of this kind.

1 Introduction

In setting out to design a laboratory exercise intended to give students practice in analyzing algorithms, the first computational problem that usually comes to mind is sorting. This is not surprising; indeed, from a pedagogical standpoint, sorting is ideal: it is simple, has many applications, and, because it is solved by a variety of algorithms possessing several different time complexity signatures[1], it serves as a convenient vehicle for teaching a number of concepts in algorithm design and analysis.

[1]By this we mean to take into account not only worst-case complexity, but also average- and best-case complexity.

Due to its appeal, we normally spend several class periods discussing sorting in our CS 2 course. In particular, we use a number of the classic sorting algorithms to illustrate techniques by which to determine an algorithm's time complexity. We also have used a homework exercise (similar to the lab described by Epp [5]) in which the student is provided with the execution times of several sorting programs on data sets of various sizes and descriptions (e.g., random, increasing, and decreasing order) and is instructed to match each program with the sorting algorithm that it utilizes (e.g., insertion sort, heapsort, quicksort, etc.). We have found that this approach—in which the student is cast in the role of playing a game of matching—succeeds in injecting some fun into a subject that many students find rather daunting. As is mentioned in [5], this has a strong analogy to the classic chemistry exercise in which the student is given an unknown substance and is asked to identify it.

The level of intellectual effort required of the student by such an exercise depends, to a large extent, on what is covered in lectures and in the course textbook. For example, suppose that the textbook describes, for each sorting algorithm under consideration, its best-, average-, and worst-case time complexities, as well as under which conditions the best and worst cases arise. Then the student need only apply this information to determine the correct matching between the algorithms and the observed execution times of the programs. While this is instructive, insofar as it may give the student a better grasp of what is meant by complexity measures such as $O(n)$, $O(n \log n)$, and $O(n^2)$, it does not require the student to perform anything in the way of algorithm analysis.

At the opposite extreme, suppose that *no* information regarding the time complexities of the sorting algorithms has been presented to the student. Completing the exercise now becomes much more difficult, and is probably beyond the abilities of all but the best students (plus those who, unable to derive all the relevant complexity measures themselves, are resourceful enough to obtain them elsewhere, such as by perusing some algorithms texts at the library).

More likely than either of these extremes is the scenario in which the student has been informed of the average- and worst-case complexities of the majority (or perhaps all) of the sorting algorithms under consideration, but in which little or nothing has been revealed about their best cases. (Because the average and worst cases are regarded as more important than the best case, most textbooks ignore the latter.) In this scenario, the student's most difficult task may very well be to determine best-case complexities for two or more of the algorithms—specifically, those which, absent this information, cannot be conclusively matched to any of the observed execution times. For example, in order to distinguish between insertion sort (as described in [8]) and selection sort, their best-case complexities—$O(n)$ and $O(n^2)$, respectively—are required, because they agree in the average and worst cases–$O(n^2)$. While this is a valuable exercise, it seems strange to focus the student's attention on best-case analysis, not only because it is less important than the other two cases, but also because best-case analysis usually requires more in the way of application-specific reasoning, and thus tends to divert the student's attention away from the basic analysis techniques that the exercise is intended to stress.

With the above points in mind, we decided to develop a laboratory/homework exercise of the kind under discussion (i.e., requiring the student to determine the correct matching between a set of given algorithms and a set of observed execution times), utilizing some problem *not* discussed in the textbook (or in other readily accessible books) and *not* to be covered in class, other than to give its definition and to illustrate it with an example. Basing the exercise on such a problem places its emphasis squarely upon the task of analyzing the given algorithms to determine their asymptotic running times, and relegates to a secondary role the subsequent (and easier) task of matching the algorithms to the observed execution times. (Assuming that each of the algorithms can be analyzed via straightforward application of the techniques covered in the course, we find this level of difficulty appropriate for CS 2.) It is worth noting that the observed execution times play an important role not only in the matching phase of the exercise, but also in the analysis phase, because they help the students to check the accuracy of their work. For example, if the student determines one of the algorithms to have complexity $O(n^2 \log n)$, but none of the observed execution times conforms to this measure, the student can conclude that the analysis was probably in error and therefore should be repeated.

It turns out that the *maximum segment sum* problem, which is defined in the next section, serves our purposes very nicely. Like sorting, it is an easy-to-comprehend problem for which there exist several simple algorithmic solutions of various time complexities. Furthermore, one of these solutions is most appropriately expressed using recursion, which means that the exercise can be used for giving students practice in analyzing both iterative *and* recursive algorithms.

Unlike sorting, the maximum segment sum problem does not appear widely in the literature; in particular, it is not found in many freshman- or sophomore-level computer science texts. (There are exceptions, of course, including [3] and [8].) Thus, in order for the students to ascertain the complexities of the given algorithms, they will most likely have to perform the analyses themselves, which is just what we want.

Another departure from sorting is this: among the algorithms for maximum segment sum that we know about, none differs (asymptotically) between its best-case and worst-case running times. Although this precludes what could be some interesting analysis work for the student, we prefer it this way, because it allows the student to concentrate on one issue: determining the worst-case complexity of each algorithm. (Besides, how many CS 2 students are prepared to perform an average-case analysis on an algorithm whose best- and worst-case running times do not agree?)

The above explains why we like to use the maximum segment sum problem, rather than sorting, as the basis for an exercise whose intended emphasis is algorithm analysis. In what remains, we give a precise definition of this problem, describe the essential details of our lab/homework exercise, and briefly discuss the algorithms it utilizes.

2 Problem Description

The *maximum segment sum* problem is as follows: given an array `A` of numbers (integers, say), compute the maximum among the sums of its contiguous segments. That is, letting

$$f(p,q) = \sum_{p \le i < q} \mathtt{A}(i)$$

we want to find[2]

$$\mathtt{max}\{f(p,q) \mid \mathtt{A'FIRST} \le p \le q \le \mathtt{A'LAST} + 1\}$$

We use Ada's notation for arrays, in which parentheses (as opposed to square brackets) are used for subscripting and in which `A'FIRST` and `A'LAST` denote the lower and upper bounds, respectively, of the index range of `A`.

For example, taking `A` to be the array holding the sequence of values

$$(-4, 7, 0, -3, 6, -3, 5, -4, 2, -3, -1, 5)$$

we find that the maximum among `A`'s segment sums is 12, corresponding to the sum of its second through seventh elements.

3 Description of the Exercise

The student is given four algorithms (expressed as function subprograms, and identified as A, B, C, and D), each of which solves the maximum segment sum problem. These appear in the figures below. Also given are four executables, identified only as programs 1, 2, 3, and 4, each of which applies one of the subprograms to an array whose length is chosen by the user. (The data occupying the array come from a file containing several hundred integers produced by a pseudo-random number generator.) The student is instructed to run each program on arrays of various lengths between 0 and 500, and to record and plot the resulting execution times on graph paper, or, if convenient, using an appropriate software package. (Each program reports an estimated "execution time", which is simply a close approximation to the number of source code instructions executed by the subprogram it called.)

The student is instructed to find the correct matching from algorithms A, B, C, and D to

[2] This definition allows for the possibility that all values in A are negative, in which case an empty segment yields the maximum sum, zero. It also assumes, only for convenience, that A is indexed by a range of integers.

```
function Max_Seg_Sum_A
  ( A : Ary_Type ) return integer is

  Sum        : integer;
  Max_So_Far : integer := 0;

begin
  for I in A'FIRST .. A'LAST  loop
    Sum := 0;
    for J in I .. A'LAST  loop
      Sum        := Sum + A(J);
      Max_So_Far := Max(Max_So_Far, Sum);
    end loop;
  end loop;
  return Max_So_Far;
end Max_Seg_Sum_A;
```

Figure 1: Algorithm A

programs 1, 2, 3, and 4 and to supply evidence supporting his/her conclusions.

To accomplish this, the student has little choice but to apply the analysis techniques presented in the course in order to derive the asymptotic running time of each algorithm, and then to match these with the execution times obtained by running the four programs in the manner described above.

4 The Algorithms

The four algorithms given to the student are in the form of Ada function subprograms, as shown in the figures. (These are adapted from the pseudo-code in [2].) The data type `Ary_Type` of the formal parameter `A` is assumed to be defined as follows:

```
type Ary_Type is array
   ( integer range <> ) of integer;
```

For the reader not familiar with Ada, this type corresponds to the notion of an "array of integers indexed by some range of integers." (The index range of an array of this type is specified in its declaration.) For an ar-

```
function Max_Seg_Sum_B
  ( A : Ary_Type ) return integer is

  Sum        : integer;
  Max_So_Far : integer := 0;

begin
  for I in A'FIRST .. A'LAST  loop
    for J in I .. A'LAST  loop
      Sum := 0;
      for K in I .. J  loop
        Sum := Sum + A(K);
      end loop;
      Max_So_Far := Max(Max_So_Far, Sum);
    end loop;
  end loop;
  return Max_So_Far;
end Max_Seg_Sum_B;
```

Figure 2: Algorithm B

```
function Max_Seg_Sum_C
  ( A : Ary_Type ) return integer is

  Max_So_Far      : integer := 0;
  Max_Ending_Here : integer := 0;

begin
  for I in A'FIRST .. A'LAST  loop
    Max_Ending_Here :=
       Max(Max_Ending_Here + A(I), 0);
    Max_So_Far :=
       Max(Max_So_Far, Max_Ending_Here);
  end loop;
  return Max_So_Far;
end Max_Seg_Sum_C;
```

Figure 3: Algorithm C

```
function Max_Seg_Sum_D
  ( A : Ary_Type ) return integer is

  Sum    : integer;
  Mid    : integer;
  Max_TL : integer := 0; --max to left
  Max_TR : integer := 0; --max to right
  Max_WL : integer;  --max within left
  Max_WR : integer;  --max within right
  Max_C  : integer;  --max with A(Mid)

begin
  if A'LENGTH = 0  then
    return 0;
  elsif A'LENGTH = 1  then
    return Max(A(A'FIRST), 0);
  else
    Mid := (A'FIRST + A'LAST) / 2;
    --find max seg sum including A(Mid)
    --first work towards left
    Sum := 0;
    for I in reverse A'FIRST..Mid-1 loop
      Sum    := Sum + A(I);
      Max_TL := Max(Sum, Max_TL);
    end loop;
    --now work towards right
    Sum := 0;
    for I in Mid+1..A'LAST loop
      Sum    := Sum + A(I);
      Max_TR := Max(Sum, Max_TR);
    end loop;
    --now compute it
    Max_C := Max_TL + A(Mid) + Max_TR;
    --find max seg sum within left and
    --right halves, using recursion
    Max_WL :=
        Max_Seg_Sum_D(A(A'FIRST..Mid-1));
    Max_WR :=
        Max_Seg_Sum_D(A(Mid+1..A'LAST));
    --result is the maximum of the three
    return Max(Max_C,
               Max(Max_WL, Max_WR));
  end if;
end Max_Seg_Sum_D;
```

Figure 4: Algorithm D

ray `A`, `A'FIRST` and `A'LAST` denote the lower and upper bounds, respectively, of its index range, `A'LENGTH` denotes its number of elements, and `A(low..high)` (which is itself of type `Ary_Type`) denotes the "slice" of `A` with index range `low..high`, where `low` and `high` are any appropriate expressions. Also assumed to exist is a function subprogram `Max` that, in constant time, returns the larger of its two integer parameters.

Even a novice programmer—one just having completed CS 1, say—should be capable of producing the fairly obvious $O(n^3)$ solution to the problem, which is our Algorithm B. It simply computes, for each i and j satisfying `A'FIRST` $\leq i \leq j \leq$ `A'LAST`, the sum of the segment `A(i..j)`, and remembers the maximum among them. There being $O(n^2)$ such segments with an average length of $O(n)$, this yields a complexity of $O(n^3)$. Algorithm A improves on B by taking advantage of the fact that, to compute the sum of the segment `A(i..j)`, where $i \leq j$, one need only add `A(j)` to the sum of the segment `A(i..j-1)`. A running time of $O(n^2)$ is thereby achieved.

Algorithm D is based on a divide-and-conquer strategy. The crucial observation is that, for any k satisfying `A'FIRST` $\leq k \leq$ `A'LAST`, there exists a segment of `A` that yields the maximum sum and that either

(1) occurs within `A(A'FIRST .. k-1)`,
(2) occurs within `A(k+1 .. A'LAST)`, or
(3) includes `A(k)`.

Thus, to compute the maximum segment sum of `A`, it suffices to choose k, to compute the maximum sum among the segments in each of the three categories, and, finally, to take the largest of these. To cover categories (1) and (2), it is natural to use recursion. Running time is minimized when k is chosen to be midway between `A'FIRST` and `A'LAST` (for exactly the same reason that *quicksort* runs fastest when the pivot happens to partition the array into equal-sized parts). The asymptotic running time is $O(n \log n)$.

Algorithm C, which is the linear time solution first described by Gries [6] (and later covered in [2, 3, 4, 7, 8]), is derivable using the inductive methodology for algorithm design espoused by Manber [7]. The crucial observation leading to the algorithm is that, for any k satisfying `A'FIRST` $\leq k <$ `A'LAST`, if you know the maximum segment sum of `A(A'FIRST..k)` as well as the maximum sum among all segments `A(j..k)` ending at element k, you can compute, in constant time, both of these values with respect to $k+1$.

For a more detailed treatment of the development of each of the four algorithms, see [2]. For a discussion of the maximum segment sum problem emphasizing the idea that an algorithm and its correctness proof should be developed hand-in-hand, see [3, 4, 6].

5 Conclusion

The maximum segment sum problem serves very nicely as the basis of an analysis of algorithms laboratory/homework exercise in either CS 2 or CS 7. To a large extent, this is due to the fact that it is solved by several relatively simple algorithms of differing time complexities, each of which is amenable to analysis by methods usually taught in these courses. In this regard it is similar to the problem of sorting, which, it seems likely, is the one most often utilized in such exercises.

In at least two other regards, however, maximum segment sum differs from sorting in ways that cause us to prefer to use the former rather than the latter. One is that maximum segment sum, possessing neither the practical nor the historical significance of sorting, need not be a topic of discussion in the course. As a result, upon receiving the instructions for the exercise, the student will have had no prior exposure to any of the algorithms presented therein and, therefore, will be forced to analyze them "from scratch".

The second is that, in contrast to the majority of the classic algorithms for sorting, the best-case and worst-case running times of each of the algorithms for maximum segment sum are (asymptotically) the same. This simplifies the analysis of the algorithms, allowing the student to concentrate solely on applying the techniques covered in class for determining worst-case asymptotic running time without concern for best- or average-case running time (both of which can be addressed very nicely in a similar, but separate, exercise that makes use of classic sorting algorithms).

References

[1] ACM Curriculum Committee on Computer Science, Curriculum '78: Recommendations for the Undergraduate Program in Computer Science, *Comm. ACM 22,3* (March 1979), pp. 147-166.

[2] Bentley, J., *Programming Pearls*, Addison-Wesley, Reading, MA, 1986, pp. 69-80.

[3] Cohen, E., *Programming in the 1990's: An Introduction to the Calculation of Programs*, Springer-Verlag, NY, NY, 1990.

[4] Dijkstra, E.W., ed., *Formal Development of Programs and Proofs*, Addison-Wesley, Reading, MA, 1990.

[5] Epp, E.C., Yet Another Analysis of Algorithms Laboratory, *SIGCSE Bulletin* 24:4 (1992), pp. 11-14.

[6] Gries, D., A Note on the Standard Strategy for Developing Loop Invariants and Loops, *Science of Computer Programming* 2 (1984), pp. 207-214.

[7] Manber, U., *Introduction to Algorithms: A Creative Approach*, Addison-Wesley, Reading, MA, 1989.

[8] Weiss, M., *Data Structures and Algorithm Analysis in Ada*, Benjamin/Cummings, Redwood City, CA, 1993.