USING SYMBOLIC COMPUTATION
FOR
TEACHING DATA STRUCTURES AND ALGORITHM ANALYSIS

Joan Krone
Department of Mathematics and Computer Science
Denison University
Granville, Ohio 43023
krone@denison.edu

INTRODUCTION
When students are introduced for the first time to the concepts in algorithm analysis, they are faced, among other things, with understanding the concept of big-O and the need for setting up and solving recurrence relations. We will see here how the use of symbolic computation can aid in mastering both tasks.

Moreover, symbolic computation can help in setting up test data sets when students implement programs, in particular those in which they define abstract data types.

At Denison University in our Data Structures and Algorithms course, we use the symbolic computation package called MAPLE. However, the ideas presented here can probably be achieved with other such packages as well.

COMPARING GROWTH RATES

Once students have been introduced to the concept of big-O, a common assignment is one in which students are asked to put several functions in order according to growth. For example, log(n) comes before n, which in turn comes before nlog(n). Even though it is obvious to students that the values associated with log(n) are smaller than those associated with n for any given n greater than or equal to one, what is really important is that these functions grow at different rates, one growing much faster than the other as n grows.

For this reason it is beneficial for students to see the graphs of these functions. Text books frequently include such graphs for a limited number of functions, and, of course, students can plot such graphs for themselves. However, MAPLE can plot the curves associated with several of these functions on a single graph, making the differences obvious. MAPLE permits users to adjust X-axis and Y-axis values in order to display the function behavior in the most revealing regions; i.e., students are able to see where such functions intersect, if they do, and where each dominates other functions.

Moreover, not all functions are so easy to compare as those listed above. For example, log(n) compared to square root of n, or more complicated functions such as $2^n$ and n! make immediate graphing and comparing tedious.
In MAPLE it is easy to get any single function or combination of functions presented on a single graph with different colors designating which function is which.

The following MAPLE code is a procedure that compares as many as five graphs at a time to see how they grow as n increases:

```maple
# Graph Comparisons
> GraphCompare:=proc(S,domain)
> local temp, i, colorlist, P;
> with(plots,display):
> colorlist:=[red,blue,green,magenta,black];
> temp:=nops(S);
> for i from 1 by 1 to temp do
> P.i:=plot(op(i,S),x=0..domain,color=colorlist[i]);
> od;
> display([seq(P.i,i=1..temp)], title=`Graph Comparison`);
> end:
```

In some cases instructors may want students to learn to program in MAPLE, whereas in other situations instructors may consider that students should just use programs they themselves have written or ones that are available through other sources. Still a third possibility is for students to learn just enough so they can take existing programs and make modifications such as changing titles or changing ranges or domains of graph exhibits. In the case where MAPLE is to be considered strictly as a tool, students can simply make calls to appropriate code. At the end of this document is a web site with all of the programs mentioned here and several others appropriate for a data structures and algorithms course.

A help feature in MAPLE permits the writer of any code to include text explaining how to use that code. For example, a student who wants to use the GraphCompare function can type

```maple
> ? GraphCompare
```

and the text will appear explaining how to interpret the colors, what the domain is, and other points that will help the student to use the function to its best advantage. The following is the code that places the help text for GraphCompare in a file that will then be available upon using the ? command given above.

```maple`
\textquotesingle help/text/GraphCompare\textquotesingle := TEXT(
```
FUNCTION: GraphCompare - Returns the graphical comparison among a list of graphs.

CALLING SEQUENCE:
    GraphCompare(L, domain)

PARAMETERS:
  L - A list of expressions to be plotted and compared,
  domain - The max X value for which you would like to compare

NOTE: the F[i] on the title for the plot stands for the ith expression entered into the function

COLOR SCHEME:
  Red = Function 1
  Blue = Function 2
  Green = Function 3
  Magenta = Function 4
  Black = Function 5

As part of work supported by a FIPSE grant, the code above and several other MAPLE labs, were developed and now reside in a special web site so that they can be shared with others who teach data structures and algorithm analysis. Help files are available with that code.

With a few simple commands, a student can see at a glance what happens to the functions of his choice as their argument increases. Here are maple commands to exhibit the graphs of four functions together and the resulting graph that is produced:

> f1 := log(x);
> f2 := sqrt(x);
> f3 := x;
> f4 := x*log(x);

As one can probably guess, these four statements define four functions in MAPLE.

The next command calls the procedure shown above to exhibit all four functions on the same graph. Students need not see that code, but rather may simply use the command.

> GraphCompare({f3,f4,f2,f1},10);

The screen then displays in color the following:
As students become more sophisticated in writing and analyzing algorithms, they frequently find it necessary to represent the time complexity of a given algorithm recursively. For example, in doing binary search, one may express complexity \( T(n) = \frac{1}{2} T(n) + C \) where \( n \) represents the number of data points, and \( C \) is some constant. In order to solve this recurrence students learn a variety of techniques.

Although the recurrence relation for binary search is simple to solve, more complicated recurrences may require so much tedium that the emphasis becomes disproportionately focused on solving the recursive equation than on the purpose for solving it.

MAPLE provides the capability to solve such equations. The following shows several recurrences and the MAPLE code needed for solving them, together with the solutions provided by MAPLE. Note that MAPLE is capable of solving general cases.
A Maple Session

Although students learn to solve several kinds of recurrences, once they understand the methods, there is no need for them to solve every recurrence they encounter when analyzing an algorithm. In fact, students can focus on finding the correct function to represent the complexity of a given algorithm, knowing that once they have found this function, MAPLE can provide the solution.
TEST DATA

One of the challenges to students in any data structures and algorithms course is that of generating appropriate test data to use for trying out their programs. MAPLE can help students with this problem in a variety of ways.

If the input to a given program is supposed to be numeric, say integer or real, a good set of test data might be a set of numbers chosen at random. Using MAPLE, one can make a text file filled with any number of random values. The following example shows code for setting up a file of 10,000 integers chosen at random.

```maple
> Generate:=proc(range)
> local i,j,arr;
> arr:=array(1..10000);
> for i from 1 by 1 to 10000 do
> j:=rand(1..range);
> arr[i]:=j;
> od;
> RETURN(arr);
> end:
# This procedure can be passed a range for the random numbers
```

This procedure is one example of how such data sets can be generated. The number 10,000 is arbitrary and can certainly be changed to any positive integer value to produce a text file of the given size. Such a file would be excellent test data for sorting programs. Having large data sets so easily available encourages students to carry out better test procedures and to notice differences among algorithms when such large data sets are used. For example, sorting 10,100 data points with bubble sort takes enough more time than using heap sort for students to notice the difference.

Having large sets of numeric test data generated so easily is useful, but one might argue that with just a bit more code, one can generate such sets in most languages. However, MAPLE goes beyond just the numeric sets. For example, one can set up graphs that can be used to test programs written to solve such problems as minimal spanning tree and shortest path.

In fact, in any algorithms course, a great deal of attention is directed toward problems involving graphs, all pairs shortest path, single source shortest path, traveling salesperson problem, critical path problems, map coloring, hamiltonian and eulerian cycles, and many others.

MAPLE has the capability to generate graphs given vertices and directions. Such graphs can be used to test programs written to solve such problems as minimal spanning tree and shortest path.

CONCLUSIONS AND A LOOK TO THE FUTURE
Having students invest a small amount of time learning to use a symbolic computation package such as MAPLE can provide a useful tool to support the understanding of certain concepts typically introduced in a data structures and algorithms analysis course.

Students can see the graphs of appropriate functions without the tedium of computing points and sketching by hand. They can solve recurrence relations and are thereby likely to give more detailed analysis of their recursive algorithms knowing that there is an easy way to get a solution to the resulting recurrence relation.

At present FIPSE is supporting research into the uses of symbolic computation in a variety of courses in science and mathematics. Faculty involved are preparing a variety of programs that can be used as course materials for those who want them. These materials are now available on the internet at the address

http://www.denison.edu/fipse/

Supported by grants from the Fund for the Improvement of PostSecondary Education (FIPSE grant P116B30079) and the Keck Foundation