Teaching elementary algorithmics using a computer algebra system

Alasdair McAndrew
Alasdair.McAndrew@vu.edu.au
School of Engineering and Science
Victoria University
Melbourne 8001
Australia

Algorithmics *is the study of the speed, efficiency, and classification of algorithms. As such, it should form a major part of the curriculum of students studying mathematics and computer science. However, it is often difficult for students to gain an appreciation of algorithms without performing some computational experiments. These can be done in any programming language. However, many beginning students are uncomfortable with formal programming, or dislike the edit-compile-run cycle of a programming language. A CAS (computer algebra system) can ameliorate these difficulties by providing students with an interactive environment where they can explore, examine, and analyze the workings of different algorithms.*

1 Introduction

Algorithms are at the very centre of computing theory and practice, and a thorough grounding in algorithmics may be considered vital for any computing training. However, most textbooks treat algorithms as a mathematical discipline, for which a major task is to categorize and analyze algorithms. For this reason the study of algorithmics can seem daunting to the beginning student. Even if the basic ideas are taught, there still remains the problem of implementation, and not all students of computer science are proficient programmers.

In place of a programming language such as C, Java, Python or many others, we have found it much easier to use a computer algebra system (CAS) where simple code fragments can be entered and tested immediately, without any intermediate compilation stage, or without having to be embedded in a complete program. The immediate feedback and ease of editing provides an environment where students can explore types and speeds of algorithms while reducing the cognitive overheads associated with formal programming.

A course in algorithmics may consist of several topics:

- Speed of algorithms. It may not necessarily be known to students that algorithms run at different

speeds. Some algorithms, such as those for sorting a list using quicksort, will run very fast. Others, such as the travelling salesman problem, have no known fast algorithm.

Students may be given examples of algorithms which run in logarithmic, polynomial or exponential time.

- Classification of algorithms. There are various classes of algorithms: greedy, divide-and-conquer, dynamic programming, backtracking, as well as heuristic methods for solving difficult problems such as the travelling salesman problem.
- Experimentation with algorithms. There may be many different algorithms for a given problem, which may run at different speeds. Students should have the opportunity to run tests on such algorithms and see for themselves the differences in speed and efficiency.

A CAS can be used to enhance students' knowledge in any of these topics. For example, raw speed can be demonstrated by giving two algorithms for computing the Fibonacci numbers; one recursive, as shown in algorithm 1 and one using a loop, as shown in algorithm 2.

fibonacci (n);
if $n \leq 1$ **then**
  return (1)
else
  return (**fibonacci** $(n-1)$+**fibonacci** $(n-2)$)
end

Algorithm 1: Recursive algorithm for Fibonacci numbers

fibonacci (n);
if $n \leq 1$ **then**
  return (1)
else
  first $= 1$;
  previous $= 1$;
  for k *from 2 to* n **do**
    next $=$ first $+$ previous;
    first $=$ previous;
    previous $=$ next
  end
  return (previous)
end

Algorithm 2: Pseudocode for fast Fibonacci computation

Both these algorithms can be easily written in any CAS for the students to use in their explorations. They should immediately see that for large values of n, for example for $n \geq 20$ or $n \geq 30$ depending on the speed of the processor, the loop algorithm produces almost instantaneous results, while the recursive algorithm is much slower. Similar algorithms can be written for computing binomial coefficients.

A CAS can also be used to generate examples where a particular heuristic may not produce the best solution. For the travelling salesman problem (which may be described as finding the lowest weight n-vertex circuit in a complete weighted graph on n vertices) a simple heuristic is to choose the edge of lowest weight which does not form a smaller circuit. Here the CAS may be use to generate an example for which this heuristic does not in fact produce the circuit of lowest weight.

2 The local situation

At Victoria University, the student body is one of the most heterogeneous in the country. The university has the highest percentage of students from non-English speaking backgrounds of any university in the country; also the highest percentage of students who are the first in their families to study at a tertiary level.

Mathematics and computing studies at first year level are taken by students not only studying mathematics, engineering, and the sciences, but also by students of education, and of human movement studies. The range of knowledge, backgrounds, and abilities is huge, and given almost any mathematical topic, the common level of understanding and mastery can be low.

The Australian tertiary education system in general does not place a great emphasis on calculus for students of computing, and although some students studying algorithmics may have taken, or be about to take, a calculus course, many learn algorithmics independently of calculus. Students who are majoring in mathematics may take this subject as an elective.

Teaching advanced topics to such diverse students can be a daunting task, and algorithmics, with its required combination of mathematical and computational understanding, can be the toughest of all topics to master. To compound the difficulties, not every student has mastered elementary programming, and many students find programming very difficult. Hence algorithmics needs to be taught in a way which minimizes the programming requirement.

In order to allow the students a level of mastery, the computer algebra system Maple [4] has been used for some years to aid the students in their learning. Worksheets, with carefully scaffolded exercises, are provided each week, which the students carefully work through in a computer lab with the help of a laboratory demonstrator. These sheets, which are marked, allow the students to concentrate on the fundamental concepts, while the mathematical engine looks after the details. Most worksheets require the student either to load previously written procedures, or to copy and paste a procedure and make minor changes to it. Since these are all performed within Maple's interactive worksheets, the students are spared the confusion of editing and compiling program code.

We have chosen our examples to satisfy several criteria:

1. the mathematics must be discrete,
2. the implementation of the algorithm should be easy to perform using pencil and paper for small cases,
3. each algorithm should exemplify its class (for example, the subset sum procedure for backtracking),
4. students should be able to explore the algorithm with the help of Maple, using inputs of their own choice,

5. students can see, either simply by timing, or keeping track of the number of operations performed, the relative speeds and efficiencies of different algorithms.

We restrict ourselves to numerical and combinatorial algorithms, for ease of introduction and exposition. Graph algorithms, as such, are untouched in this course, although in fact they can be used very effectively to teach algorithmics [6].

3 Types of algorithms

We have adopted the classification as given by Brassard et al [2]. In our course we investigate algorithms which fall into five broad classes.

Greedy algorithms. In general, a *greedy algorithm* is one which generates the solution by choosing the best possible improvement at each stage. There are two characteristics of a greedy algorithm:

1. At each stage the best value is chosen without worrying whether it will be the best decision in the long run.
2. Once a decision is made, it is never reversed.

Greedy algorithms form one of the largest classes of algorithms; they are generally fast, easy to implement, and often provide very good results. In some cases (as we shall see) a greedy algorithm may not provide the best possible solution to a given problem, but in problems for which no simple algorithm is known, a greedy approach to a solution can be used to find a quick result, if not necessarily the best possible.

Divide-and-conquer. This is one of the most powerful approaches to algorithm design. The idea is this: the problem to be solved is divided into smaller subproblems, and the algorithm is applied to these. Then the results are (if necessary) put back together. This is shown schematically in figure 1.

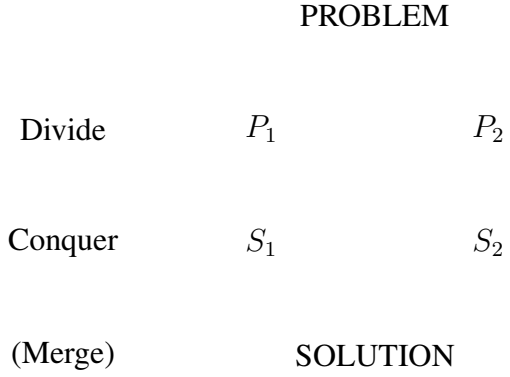

Figure 1: Divide and conquer: one level

Dynamic Programming. Dynamic programming may be considered to be an example of "bottom up" algorithm design: we start with simple cases of the problem to be solved, and build up until we reach a solution of our problem. Divide and conquer, in contrast, is a "top down" approach, where our problem is divided into smaller problems. The term "programming" in dynamic programming, has nothing to do with computer programming, but comes from an old mathematical term which means solving problems by building up information in tabular form. As we shall see, all dynamic programming algorithms work by the creation of tables.

Backtracking. One way of solving a problem is by *exhaustive search*: we enumerate all possible solutions, and see which one produces the optimum result. For example, for the knapsack problems, we could look at every possible subset of objects, and find out which one has the greatest value, and is not greater than the weight bound. *Backtracking* is a variation of exhaustive search, where the search is refined by eliminating certain possibilities. Backtracking can often be faster than an exhaustive search; for some problems no faster algorithms are known.

Heuristics. *Heuristics* may be considered as "quick and dirty" methods for providing rough solutions to generally difficult problems. This means that a heuristic might not necessarily return the *best* possible solution, but it should at least return a "reasonably good" one. What constitutes "reasonably good" depends on the problem. Many heuristics are based on greedy algorithms, with various adjustments to counteract some of their deficiencies.

Other classes of algorithms, such as probabilistic and parallel algorithms, are not covered in our course.

3.1 Greedy algorithms

Our favorite greedy algorithm is the simple algorithm for generating *egyptian fractions*:

$$\frac{p}{q} = \frac{1}{k_1} + \frac{1}{k_2} + \cdots + \frac{1}{k_n}$$

where

$$k_1 < k_2 < \cdots < k_n.$$

A sum of unit fractions in this context is called an *egyptian fraction decomposition* (EFD). A simple greedy algorithm starts by letting $1/k_1$ be the largest reciprocal less than p/q. Then update p/q by subtracting $1/k_1$ from it, so that

$$\frac{p}{q} \leftarrow \frac{p}{q} - \frac{1}{k_1} = \frac{p \cdot k_1 - 1}{q \cdot k_1}.$$

Then let $1/k_2$ be the largest reciprocal less than this new value of p/q, and so on. The values k_i can be found by noting that if k_i is the ceiling value of q/p then $1/k_i$ is the largest reciprocal less than p/q. This algorithm can be stated more precisely in algorithm 3.
This is greedy because at each step we choose as k_i the largest unit fraction $1/k_i$ which is less than p/q. It can be shown that this algorithm satisfies the requirements of EFD: each new value will be

egyptianfraction (p/q);
Set $i = 1$;
while $p > 1$ **do**
  $k[i] = \lceil q/p \rceil$;
  $p = p \cdot k[i] - q$;
  $q = q \cdot k[i]$;
  $i = i + 1$
end

Algorithm 3: Pseudocode for egyptian fraction decomposition

strictly greater than the previous, and the algorithm terminates: eventually the algorithm will produce a value $p = 1$ at which stage we have a unit fraction and can stop. The proof dates back to Fibonacci in the 13th century, and is discussed by Wagon [8].

In many ways this algorithm is a good example of a greedy algorithm, in that it always produces a result, but not necessarily the best. For example, the algorithm produces

$$\frac{11}{28} = \frac{1}{3} + \frac{1}{17} + \frac{1}{1428}$$

but in fact a simpler EFD is

$$\frac{11}{28} = \frac{1}{4} + \frac{1}{7}.$$

In Maple, an EFD procedure exactly follows the algorithm above, except that we return a list of the values of $1/k_i$, rather than a sum of fractions:

```
egypt:=proc(r::rational) local a,c,p;
a:=[];
p:=r;
while (p<>0) do
  c:=ceil(1/p);
  a:=[op(a),1/c];
  p:=p-1/c;
end do;
return(a);
end;
```

In the labs, students load the procedure, and experiment with it. They can:

- Find the EFD's of any fraction they like. Given Maple's ability to handle integers of arbitrary size, there is no restriction on the size of the numerator and denominator of r, except as limited by computer memory.
- Experiment to find an input for which a denominator in the EFD is greater than a given bound N. For example, given $N = 10,000,000$, they might find that

$$\frac{12}{43} = \frac{1}{4} + \frac{1}{35} + \frac{1}{2007} + \frac{1}{12082140}.$$

- Add two or more unit fractions to try to create fractions (such as $11/28$ above), for which the greedy EFD is not the simplest EFD, either by having a non-minimal number of terms, or containing very large integers.

Egyptian fractions are themselves the object of much study and research, and the simple greedy algorithm given here is only one of many. See for example Eppstein [3], also available online at `http://www.ics.uci.edu/~eppstein/numth/egypt/intro.html`.

3.2 Divide-and-conquer

The easiest divide-and-conquer algorithms are probably the quicksort and mergesort algorithms. Mergesort was the first to be described.

1. Divide the list into two halves.
2. Sort each half (recursively, using mergesort).
3. Merge the halves together to form a single list.

This is shown schematically in figure 2. Note the striking similarity between this figure and the generic diagram shown in figure 1.

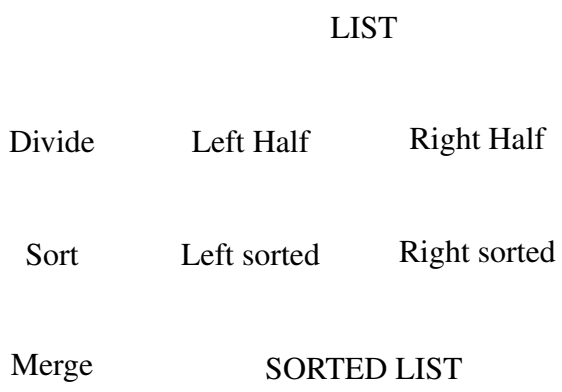

Figure 2: Mergesort: one level

In Maple, this algorithm is

```
mergesort:=proc(L)
  local n,n2,L1,L2,M1,M2,M;
n:=nops(L);
if (n=1)
  then
    return(L);
  else
    n2:=floor(n/2);
    L1:=[op(1..n2,L)];
    L2:=[op(n2+1..n,L)];
    M1:=mergesort(L1);
    M2:=mergesort(L2);
    M:=quickmerge(M1,M2);
    return(M);
end if;
end;
```

```
quickmerge:=proc(L1,L2)
  local i,j,k,m,n,P1,P2,M;
n:=nops(L1);
m:=nops(L2);
M:=[];
P1:=[op(L1),infinity];
P2:=[op(L2),infinity];
i:=1;
j:=1;
for k from 1 to m+n do
  if (P1[i]<=P2[j])
    then
      M:=[op(M),P1[i]];
      i:=i+1;
    else
      M:=[op(M),P2[j]];
      j:=j+1;
  end if;
end do;
return(M);
end;
```

The `quickmerge` procedure is taken from [2]. Contrast this with bubblesort (which simply passes over a list as many times as necessary, swapping any two consecutive elements in the wrong order, until the entire list is sorted). Students can run timing exercises on mergesort and bubblesort, and see for themselves which is fastest.

3.3 Dynamic programming

Dynamic programming is sometimes differentiated from divide and conquer in describing it as a "bottom-up" design: the solution to a problem is constructed, piece by piece, from simple instances. Divide and conquer is then called a "top-down" design. Dynamic programming can be used to provide solutions to many problems; in our course we discuss the following:

Binomial coefficients: These can be easily generated by a recursive, and slow, algorithm. But a much faster way is to use a tabular method, and keep track of previous values.

Coin change problem: Given coins of various denominations, how can we make change of any amount using the minimal number of those coins?

Zero-one knapsack: Suppose we have a "knapsack" containing articles of varying weights and values. Given a weight N, how do we choose articles from the knapsack so as not exceed the weight N but also to maximize the total value?

Longest common subsequences: Given two sequences, finding the longest subsequence which is common to both. For example, if we take the two strings "B I R T H D A Y S" and "A N N I V E R S A R Y", their longest common subsequence is "I R A Y":

BIRTHDAYS
ANNIVERSARY

The binomial coefficients can be easily generated recursively:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n}{k-1}, \qquad \binom{n}{0} = \binom{n}{n} = 1.$$

But as with the example of the Fibonacci numbers earlier, this is extremely slow. The problem with this algorithm, as with the Fibonacci numbers, is that small results need to be computed many times. A far faster algorithm uses an array $B(i,j)$ of size $n \times k$, initialized with $B(i,0) = 1$ for all $0 \le i \le n$ and $B(0,j) = 0$ for all $0 \le j \le k$. The the sum

$$B(i,j) = B(i-1,j) + B(i-1,j-1)$$

can be placed within a double loop. Here are the two programs in Maple:

```
binom:=proc(n,k)
if ((k=0) or (n=k))
  then
    return(1);
  else
    return(binom(n-1,k)
          +binom(n-1,k-1));
end if;
end;
```

```
binom2:=proc(n,k)
  local B,i,j;
B:=array(0..n,0..k);
for i from 0 to n do B[i,0]:=1; end do;
for i from 1 to k do B[0,i]:=0; end do;
for i from 1 to n do
  for j from 1 to k do
    B[i,j]:=B[i-1,j]+B[i-1,j-1];
  end do;
end do;
return(B[n,k]);
end;
```

Students can experiment with running these programs (on small values of n and k) until they begin to see a significant difference in speed between the two. To obtain a more complete analysis, students next add some global counters to these programs, to count the number of times each statement is performed:

```
binom:=proc(n,k)
 global i,j;
if ((k=0) or (n=k))
  then
   i:=i+1;
    return(1);
  else
   j:=j+1;
    return(binom(n-1,k)
          +binom(n-1,k-1));
end if;
end;
```

```
binom2:=proc(n,k)
  local B,i,j;
 global p;
B:=array(0..n,0..k);
for i from 0 to n do B[i,0]:=1; end do;
for i from 1 to k do B[0,i]:=0; end do;
for i from 1 to n do
  for j from 1 to k do
   p:=p+1;
    B[i,j]:=B[i-1,j]+B[i-1,j-1];
  end do;
end do;
return(B[n,k]);
end;
```

The programs are then run as follows:

```
> i:=0;j:=0;binom(10,5);i;j;
```

and

```
> p:=0;binom2(10,5);p;
```

The coin change problem is fine example of a problem where a greedy algorithm does not necessarily provide the best solution. For this problem, the greedy algorithm is simply to choose the coin of largest possible denomination at each step. For example, if coins of denomination 25c, 20c and 5c are required to make change for 40c, the greedy algorithm first chooses 25c (as the largest possible denomination less than 40c) and then fills up with three 5c coins for a total of four coins used. However, clearly 20c coins can be used to make the change with only two coins.

We shall create a table whose elements $c(i, j)$ give the smallest number of coins to make a value of j, *using only coins of denominations* $d_1, d_2, \ldots d_i$. So for example, $c(2, 5)$ is the smallest number of coins required to produce a change of 5 using only coins d_1 and d_2. And $c(3, 6)$ is the smallest number of coins required to produce a change of 6 using only coins d_1, d_2 and d_3. We will assume that the lowest coin has denomination $d_1 = 1$c. This is a reasonable assumption; sometimes we can scale all coins and change so that this requirement is met, or we can adjust the algorithm to allow change only for values greater than d_1.

The problem now is very similar to that of creating the table for binomial coefficients; we need to provide:

1. some initial values,
2. a rule by which a value $c(i, j)$ can be calculated from lower values.

Initial values are straightforward. For $j = 0$, we need no coins, and so $c(i, 0) = 0$ for all i. And if we are restricted to using only the first coin, with $d_1 = 1$, we are going to need j of this coin to make that value. Thus $c(1, j) = j$ for all j.

At this stage the table is:

	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0	1	2	3	4	5	6	7	8
$d_2 = 4$	0								
$d_3 = 5$	0								

Consider a general element $c(i, j)$. We have two choices: either we use a coin of denomination d_i, or we don't. If we don't use a coin of denomination d_i, then we are only using coins up to a denomination of d_{i-1}, and so $c(i, j) = c(i - 1, j)$. If we *do* use a coin of denomination d_i, then we have *one* coin of denomination d_i, and the rest of the coins for the rest of the change. But we are making up a change of value j, and if we have used a coin of denomination d_i, then there is a value of $j - d_i$ left. But this can be made up with $c(i, j - d_i)$ coins, giving a total of $1 + c(i, j - d_i)$ coins used altogether. Since we want to use the smallest number of coins, the final value of $c(i, j)$ will be whichever of these two choices produces the smallest number.

A restriction here is if $j < d_i$, in which case we can't use a coin of denomination d_i, as it would go over the value. Then $c(i,j) = c(i-1,j)$.

Putting all this together gives:

$$c(i,j) = \begin{cases} c(i-1,j) & \text{if } j < d_i, \\ \min\{c(i-1,j), 1 + c(i,j-d_i)\} & \text{otherwise.} \end{cases}$$

Now the table can be built up either row by row, or column by column. To determine which coins were used, rather than just the number, we can work backwards through the table from a give value, applying the following rules:

- if $c(i,j) \neq c(i-1,j)$ but $c(i,j) = 1 + c(i,j-d_i)$ we can assume a coin of denomination d_i was used, and go back to $c(i,j-d_i)$.
- if $c(i,j) = c(i-1,j)$ but $c(i,j) \neq 1 + c(i,j-d_i)$ we can assume no coins of value d_i were used, and go back to $c(i-1,j)$.
- if $c(i,j) = c(i-1,j)$ and $c(i,j) = 1 + c(i,j-d_i)$ it is irrelevant whether we use a coin of denomination d_i or not—the final result will still be the same whichever choice we make.

In Maple, we produce two procedures,

```
coins:=proc(denoms,value)
  local num,C,i,j;
num:=nops(denoms);
C:=array(1..num,0..value);
for i from 1 to num do
  C[i,0]:=0;
end do;
for j from 0 to value do
  C[1,j]:=j;
end do;
for i from 2 to num do
  for j from 1 to value do
    if (j<denoms[i])
      then
        C[i,j]:=C[i-1,j];
      else
        C[i,j]:=min(C[i-1,j],
             1+C[i,j-denoms[i]]);
    end if;
  end do;
end do;
return(op(C));
end;
```

```
printcoins:=proc(denoms,value)
  local i,j,C,used;
C:=coins(denoms,value);
used:=[];
i:=nops(denoms);
j:=value;
while ((i>=2) and (j>=0)) do
  if (C[i,j]<C[i-1,j])
    then
      used:=[denoms[i],op(used)];
      j:=j-denoms[i];
    else
      i:=i-1;
  end if;
end do;
while (j>0) do
  used:=[denoms[1],op(used)];
  j:=j-denoms[1];
end do;
return(used);
end;
```

Consider for example the problem of making change for 40c with 25c, 20c and 5c coins mentioned earlier. Dividing all values by 5 reduces the problem to making change for 8c with 5c, 4c, and 1c coins.

Then

```
> t:=coins([1,4,5],8):
> matrix(t);
```

produces

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 0 & 1 & 2 & 3 & 1 & 2 & 3 & 4 & 2 \\ 0 & 1 & 2 & 3 & 1 & 1 & 2 & 3 & 2 \end{bmatrix}$$

and

```
> printcoins([1,4,5],8);
```

produces

$[4, 4]$.

3.4 Backtracking

Consider the "subset sum problem": given a set S, and a number N, is it possible to find a subset $T \subseteq S$ whose elements sum to N? For example, suppose we have

$$S = \{1, 4, 7, 10, 12, 17\}$$

Can we find a subset of S which sums to 42? (No, we can't.) What about to 43? In this case yes,

$$T = \{4, 10, 12, 17\}, \qquad 4 + 10 + 12 + 17 = 43.$$

Clearly this is a problem which can easily be solved by an exhaustive search: list all the subsets of S, and find the sum of each one. Then check to see if the number N is in this list. But this would be very slow and tedious for large sets; if a set has n elements it has 2^n subsets. This would mean that for a 20 element set, we would need to check the sum of over a million subsets. Backtracking provides a more elegant approach.

Take $S = \{93, 27, 54, 42, 76, 20\}$, and $N = 157$. We shall define a subset to be *feasible* if its sum is 157 or less. The backtracking algorithm may be stated as:

Step 1: Start with an empty set.

Step 2: Add to the subset the next element in the list.

Step 3: If the subset sums to 157 then **stop** with a solution.

Step 4: If the subset is feasible, repeat step 2.

Step 5: If the subset is not feasible, or if we have reached the end of the set (no more values to choose), then *backtrack* through the subset until we find the *most recent* place at which we can change a value.

Go back to step 2.

Step 6: If we have used the entire set without finding a suitable subset, and if no more backtracking is possible, then **stop** with no solution.

We can provide a general algorithm for backtracking, based on an exhaustive search algorithm as given in algorithm 4.

```
exhaustive (Sets S, Y);
if Y satisfies condition then
    return (Y);
else
    for each element x ∈ S − Y do
        exhaustive (S, Y ∪ {x})
    end
end
```

Algorithm 4: Pseudocode for exhaustive search

If the algorithm is called with

exhaustive(S, $\emptyset$)

then all subsets will be generated, and checked for the condition. Note that if the algorithm is called with

exhaustive(S, T)

where $T \subseteq S$, then the search will only consider subsets which include T. The general backtracking algorithm is similar, except that we only check subsets if they are *feasible*; that is, if a subset may lead to a solution. In the subset sum problem, a subset is feasible if its sum is less than or equal to the required value. If a subset is feasible, we can add elements to it; if it is not feasible, we don't. The general algorithm is given is algorithm 5.

```
backtrack (Sets S, Y);
if Y is feasible then
    if Y satisfies condition then
        return (Y);
    else
        for each element x ∈ S − Y do
            if Y ∪ {x} is feasible then
                backtrack (S, Y ∪ {x})
            end
        end
    end
end
```

Algorithm 5: Pseudocode for backtracking

Note that this algorithm is very similar to that for exhaustive search; the only real difference is that we check only those subsets which are feasible. As with exhaustive search, the algorithm must be called with

backtrack$(S, \emptyset)$

to make sure that *all* possible feasible subsets are checked.

For the subset sum problem, a subset is feasible if the sum of its elements does not exceed the bound. In Maple this can be easily implemented with a recursive procedure which returns "true" if a subset can be found with the required sum, and false otherwise. In the case of "true" the subset is printed.

```
subsetsum:=proc(X,Y,N) # X is a set, N is an integer
  local y;
  global count;
count:=count+1;
if (add(x,x=Y)<=N) then
  if (add(x,x=Y)=N) then
    print(Y);
    return(true)
  else
    for y in (X minus Y) do
      if (subsetsum(X,Y union {y},N)=true) then
        return(true);
      end if;
    end do;
  end if;
return(false);
end if;
end;
```

The use of the "count" variable is simply to count the number of times the procedure is called. For example:

```
> X:={seq(ithprime(2*i),i=1..20)};
   X := {3, 7, 13, 19, 29, 37, 43, 53, 61, 71, 79, 89, 101, 107, 113, 131,
         139, 151, 163, 173}

> N:=sum(X[ithprime(i)],i=1..8);
                                    N := 574
> count:=0: subsetsum(X,{},N);count;
              {3, 7, 13, 19, 29, 37, 43, 53, 61, 71, 107, 131}
                                      true
                                       223
> N:=sum(X[i],i=1..8);
                                    N := 204

> count:=0: subsetsum(X,{},N);count;
                        {3, 7, 13, 19, 29, 37, 43, 53}
                                      true
                                        9
> N:=sum(X[i],i=11..18);
```

```
                                    N := 910

> count:=0: subsetsum(X,{},N);count;
          {3, 7, 13, 19, 29, 37, 43, 53, 61, 71, 107, 131, 163, 173}
                                    true
                                    27803
```

Given that the number of subsets to be searched through using an exhaustive search is $2^{20} = 1048576$, we see that backtracking can provide a better alternative.

3.5 Heuristics

In our subject, we understand a "heuristic" to be a quick-and-dirty method of providing a reasonable solution, if not necessarily the best. A good problem here is the travelling salesman problem: given a group of cities, and the distances between each of them, find a route through all cities which minimizes the total distance travelled. This problem can be expressed in terminology from graph theory: find the Hamiltonian circuit of least total weight from a complete weighted (undirected) graph. We can also express the problem in terms of matrices: given a symmetric square matrix M of size $n \times n$ for which $m_{ii} = 0$ for all $1 \leq i \leq n$, find the permutation p of $[1, 2, \ldots, n]$ which minimizes

$$\sum_{k=1}^{n} m_{p[i],p[i+1]}$$

where we understand $p[n+1]$ to be equal to $p[1]$. Here we can interpret M as being the adjacency matrix of the weighted graph, so that $m_{ij} = m_{ji}$ is the distance between cities i and j. This problem is important not only from a practical perspective, but also from its known difficulty: it is in fact NP-complete, and there is no known better general solution than exhaustive search. Given the speed of growth of $n!$, we see that exhaustive search is highly inefficient for all except small values of n.

Suppose an adjacency matrix M is given. The the sum of paths length for a given permutation P can be implemented by the Maple command

```
S:=add(M[P[i],P[i+1]],i=1..n-1)+M[P[n],P[1]];
```

If we generate all the permutations of $[1, 2, \ldots, n]$, we can run through them one at a time, keeping track of the smallest path length found.

The times in seconds for computing the shortest path on matrices of different sizes are:

size	time
5×5	0.020
6×6	0.076
7×7	0.320
8×8	3.828
9×9	27.037
10×10	315.384

Performing a least squares regression on the logarithms of the times and extrapolating, reveals that it would take over 2000 years to perform the calculation on a 20×20 matrix.

A simple greedy heuristic is as follows: choose a starting vertex i. At each stage, choose the path of least weight from the current vertex to an unvisited vertex.

For example, suppose we take five cities and their distances as follows:

	A	B	C	D	E
A	0	10	6	8	14
B	10	0	11	7	9
C	6	11	0	12	200
D	8	7	12	0	13
E	14	9	200	13	0

Suppose we start at A, and generate a solution according to the greedy algorithm. For the given table, we start at A, and choose to visit city C, which is the cheapest to reach from A:

$$A \xrightarrow{6} C, \qquad \text{(B, D, E unvisited)}$$

From C, the cheapest of B, D, and E to visit is B so:

$$A \xrightarrow{6} C \xrightarrow{11} B \qquad \text{(D, E unvisited)}$$

From B, the cheapest of D and E to visit is D, so:

$$A \xrightarrow{6} C \xrightarrow{11} B \xrightarrow{7} D, \qquad \text{(E unvisited)}$$

There are no more choices to be made; from D we can only visit E, and then back to A:

$$A \xrightarrow{6} C \xrightarrow{11} B \xrightarrow{7} D \xrightarrow{13} E \xrightarrow{14} A$$

The total cost is $6 + 11 + 7 + 13 + 14 = 51$.

We can't be sure if this is the shortest route or not. The greedy algorithm, although simple to apply, has a major fault (which is common to many greedy algorithms): by choosing the smallest step at each stage, we are limiting our chances later on. In particular, we may be forced into choosing a large step.

Suppose we apply the same algorithm but start at different cities:

$$B \xrightarrow{7} D \xrightarrow{8} A \xrightarrow{6} C \xrightarrow{200} E \xrightarrow{9} B \qquad \text{Total: } 230$$

$$C \xrightarrow{6} A \xrightarrow{8} D \xrightarrow{7} B \xrightarrow{9} E \xrightarrow{200} C \qquad \text{Total: } 230$$

$$D \xrightarrow{7} B \xrightarrow{9} E \xrightarrow{14} A \xrightarrow{6} C \xrightarrow{12} D \qquad \text{Total: } 48$$

$$E \xrightarrow{9} B \xrightarrow{7} D \xrightarrow{8} A \xrightarrow{6} C \xrightarrow{200} E \qquad \text{Total: } 230$$

This produces a better result: 48, and it also shows that the greedy algorithm can produce very poor results if it starts at the wrong city. Thus we can state a simple heuristic for the travelling salesman problem:

Heuristic Apply the greedy algorithm starting at each city in turn, and see which result produces the smallest value.

Other heuristics can be found in [5]. If the number of cities is very large, then this heuristic can be modified by starting at only a random sample of the cities rather than all of them.

This heuristic does not guarantee the shortest possible path—in our example we checked only 5 paths out of the 24 possible. (If we take into account the fact that in our example the cost is independent of direction of travel, then there are in fact only 12 different paths.)

And the shortest path can be found (by an exhaustive search) to be:

$$A \xrightarrow{8} D \xrightarrow{13} E \xrightarrow{9} B \xrightarrow{11} C \xrightarrow{6} A$$

The total cost is $8 + 13 + 9 + 11 + 6 = 47$. However, the heuristic does in this case provide a very close approximation.

A Maple procedure for applying this heuristic can be easily written:

```
trav_heur:=proc(M)
  local n,i,sP,iP,sV,iV;
n:=LinearAlgebra[RowDimension](M);
sP:=trav_heur_start(M,1);
sV:=path_length(M,sP);
for i from 2 to n do
  iP:=trav_heur_start(M,i);
  iV:=path_length(M,iP);
  if iV<sV then
    sP:=iP;
    sV:=iV;
  end if;
end do;
return([sP,sV])
end;
```

Here the procedure `trav_heur_start` provides the actual greedy computations, given a starting vertex:

```
trav_heur_start:=proc(M,start)
  local n,i,P,A,verts,path,recent,unseen;
n:=LinearAlgebra[RowDimension](M);
verts:=[seq(i,i=1..n)];
unseen:=subsop(start=NULL,verts);
path:=[start];
recent:=start;
for i from 2 to n do
  P:=[seq(M[recent,unseen[j]],j=1..n+1-i)];
  A:=min_list(P);
  recent:=unseen[A[1]];
  path:=[op(path),recent];
  unseen:=subsop(A[1]=NULL,unseen);
end do;
return(path);
end;
```

We also need a procedure `path_length` to compute the length of a path provided by the greedy algorithm:

```
path_length:=proc(M,P)
  local n,T;
n:=LinearAlgebra[RowDimension](M);
T:=add(M[P[i],P[i+1]],i=1..n-1)+M[P[n],P[1]];
return(T);
end;
```

The example above is pedagogically useful because it illustrates two things:

1. the heuristic is very quick to work—for n cities we only have to apply the greedy algorithm n times, rather than check $n!$ paths.
2. the heuristic does not necessarily provide the best solution.

This leads to a sequence of exercises on heuristics:

1. Given a matrix corresponding to distances between cities:
   (a) apply the heuristic to obtain a possible small value
   (b) by trial and error, or exhaustive search, find a path which has a smaller value.
2. Create a matrix for which the heuristic does not produce the smallest possible value.

This last can be done by trial and error. For example, the following distance matrix:

$$\begin{bmatrix} 0 & 44 & 17 & 7 & 68 & 46 & 2 & 21 \\ 44 & 0 & 3 & 55 & 3 & 12 & 3 & 37 \\ 17 & 3 & 0 & 91 & 46 & 24 & 42 & 16 \\ 7 & 55 & 91 & 0 & 48 & 57 & 53 & 3 \\ 68 & 3 & 46 & 48 & 0 & 67 & 35 & 29 \\ 46 & 12 & 24 & 57 & 67 & 0 & 55 & 23 \\ 2 & 3 & 42 & 53 & 35 & 55 & 0 & 17 \\ 21 & 37 & 16 & 3 & 29 & 23 & 17 & 0 \end{bmatrix}$$

has a shortest hamiltonian path of length 100, but the best that the heuristic can do is 126.

4 Conclusion

Although students may have learned that some programs, procedures or algorithms may run at different speeds, they do not often fully appreciate the scale of speed differences until formally studying algorithmics. Our approach has shown how this study can be made relatively painless. Our teaching also follows good constructivist practice [7], allowing students to learn at their own speeds.

The subject is evaluated using a standard questionnaire, on which students rate questions about the subject material, delivery, and assessment (grading) using a five point Likert scale. Students have been highly supportive of this approach using Maple, and have commented on how its use has made an abstract discipline more concrete and understandable.

It would be possible to expand on this introductory course to teach deeper material [1]; in particular to investigate complexity and data structures, and to include more rigorous mathematics. However, in its current form the course, with the aid of a CAS for exploration and enhancement, has been shown to provide a very efficient and enjoyable introduction to the discipline.

References

[1] Ricardo A. Baeza-Yates, *Teaching Algorithms*, ACM SIGACT News **26** (1995), no. 4, 51–59.

[2] Gilles Brassard and Paul Bratley, *Fundamentals of Algorithmics*, Prentice-Hall, 1996.

[3] David Eppstein, *Ten algorithms for Egyptian fractions*, Mathematica in Education and Research **4** (1995), no. 2, 5–15.

[4] *Maplesoft: Math software for engineers, educators and students*, `http://www.maplesoft.com`, 2008, Accessed August 2008.

[5] Gerhard Reinelt, *The Travelling Salesman: Computational Solutions for TSP Aplications*, Lecture Notes in Computer Science, no. 840, Springer-Verlag, 1994.

[6] Petra Scheffler, *Teaching Algorithmics—Theory and Practice*, Proc. 2nd Intern. Sc. Conf. "Informatics in the Scientific Knowledge", 2008, pp. 259–269.

[7] Geoffrey Scheurman, *From behaviorist to constructivist teaching*, Social Education **62** (1998), no. 1, 6–9.

[8] Stan Wagon, *Mathematica in action*, 2nd ed., Springer, 1999.