Computer Science 1FC3

Lab 9 – Recurrence Relations (Computational)

Author – Paul Vrbik – vrbikp@mcmaster.ca

This lab will review methods of solving recurrence relations in Maple.

RECURRENCE RELATIONS

Simply put, a recurrence relation is just a sequence where any given element is defined using one or more of the previous elements. As you may recall we defined simple sequences in maple as functions:

```
]>a:=n->n;
```

However we may use a similar syntax to define recurrence relations. For instance consider the sequence of factorials $fact_n = n!$ this would be a recurrence relation defined as $fact_0=1$, $fact_n=n*fact_{n-1}$ implemented like:

```
]>fact:=n->n*fact(n-1);
                                fact:=n->n fact(n-1)
]>fact(0):=1;
                                   fact(1):=1;
]>fact(4);
                                      24;
```

Question 1:

What happens if you define `fact:=1;` before `fact:=n->n*fact(n-1);` in the above scheme. Can you provide any reason as to why this may happen?

Question 2:

The Fibonacci Numbers are given as $f_1=1$, $f_2=1$, $f_n=f_{n-1}+f_{n-2}$. Define this sequence in maple and determine the following:

$f_2=$

$f_{10}=$

$f_{50}=$

$f_{100}=$

SOLVING RECURRENCE RELATIONS

Although the definition given by a recurrence relation is elegant and easy to give to a computer, evaluating an arbitrary term may be hard and time consuming. For example, in order to evaluate $fact_{12}$ (by the definition given on the first page) would require us to first determine $fact_{11}$ and correspondingly $fact_{10}$ all the way down to $fact_0$. It would be undoubtedly easier to just evaluate `n!` since $fact_n = n!$, where `n!` is called the solution to the recurrence relation $fact_n$.

In fact this is the motivation for the strategies given in section 6.2 of the Rosen. But we will not be covering these today; instead we will use Maple to solve the recurrence relations for us. However let us first give a strict definition of a recurrence relation

solution to a recurrence relation

Given any recurrence relation a_n, a *solution* to the recurrence relation a_n is an explicit function (non-recursive function) `f(n)`, such that $f(n)=a_n$ for all `n`.

Consider our first example:

```
(1)    ]>temp:=rsolve({f(n)=n*f(n-1),f(0)=1},f);
                                    temp := Γ(n + 1)
(2)    ]>g:=x->expand(eval(temp,n=x));
                                    g := x -> expand(eval(temp, n = x))
(3)    ]>g(3);
                                    6
(4)    ]>g(4);
                                    24
```

Note that Γ (the gamma function) is defined as $\Gamma(n+1)=n!$.

Lets investigate the syntax used above.

In `(1)`

`rsolve` is a function which takes the list `{recurrence relation, base case(s)}` and the name of the relation being solved for (in our case its `f`) and returns the solution to the recurrence relation.

In `(2)`

This is a way to take the function outputted by `rsolve` and assign it to a function `g` that we can then use.

In `(3)-(4)`

We simultaneously demonstrate that `rsolve` and our definition for `g` are valid.

Question 3:

Use `rsolve` to determine a solution to the Fibonacci sequence given on the first page. Assign the solution to the function `h(x)` and record the following values.

```
h(2)=                         h(50)=

h(10)=                        h(100)=
```

Do your answers match those given in Question 2? Should they?

Question 4:

Solve the recurrence relation

$$a_n = 2 * a_{n-1}$$
$$a_0 = 3$$

and verify that this is the correct solution by comparing to values given by the definition.

DIVIDE AND CONQUER ALGORITHMS

Divide and conquer is an important strategy in computer science. The idea of the strategy is to take a big problem and divide into many sub problems that are more easily solved. If we allow our recurrence relations to do more robust things they may be considered a form of divide and conquer.

For example, consider the problem of finding the largest number in a set of integers. We could easily find it by doing a linear search but a recurrence relation may be devised to solve it more intelligently by doing the following:

```
\\single element set
maxInList({x})=x;

\\if a set has more then one element then it can be broken into two
non-empty sets which allows us to write
maxInList(A union B) = max( maxInList(A), maxInList(B))
```

This definition is a little bit different then what we are used to, but looking at it indeed has a base case and recursive step fulfilling our definition of recurrence relation.

Question 5:

Like the above example, create a loosely defined recurrence relation `element` to test weather or not a given element is a member of a set. That is:

```
element(1,{1,2,3})=true
element(7,{1,2,3})=false
```