Recursion and the MATLAB profile tool

This is outside of textbook

Disclaimer: *These abbreviated notes DO NOT substitute the textbook for this class. They should be used IN CONJUNCTION with the textbook and the material presented in class. If there is a discrepancy between these notes and the textbook, ALWAYS consider the textbook to be correct. Report such a discrepancy to the instructor so that he resolves it. These notes are only distributed to the students taking this class with A. Gerbessiotis in Fall 2011 ; distribution outside this group of students is NOT allowed.*

S1. Recursion: A function calling itself.

S1. Recursive functions. A (mathematical) function is allowed to call itself. Such a function is called a `recursive function`. **Recursion** refers to the use of functions that call themselves in a recursive fashion.

S1a. An example. Consider function $f(n)$ defined on positive integers such that $f(n) = n \times f(n-1)$. This is a recursive function. The value at n is defined in terms of the value at $n-1$. From $f(n) = nf(n-1)$, if we substitute $n-1$ for n we get that $f(n-1) = (n-1)f(n-2)$. Similarly, if we substitute $n-2$ for n we get $f(n-2) = (n-2)f(n-3)$ and so on. Using the expression for $f(n-1), f(n-2)$ in $f(n)$ we can unfold the recursion as shown below. Then we obtain the following, after unfolding the recursion 4 times.

$$\begin{aligned} f(n) &= n \times f(n-1) \\ &= n \times (n-1) \times f(n-2) \\ &= n \times (n-1) \times (n-2) \times f(n-3) \\ &= n \times (n-1) \times (n-2) \times (n-3) \times f(n-4) \end{aligned}$$

If this is continued, eventually we will reach after $n-1$ unfolding operations

$$\begin{aligned} f(n) &= n \times f(n-1) \\ &\dots \\ &= n \times (n-1) \times (n-2) \times (n-3) \times f(n-4) \\ &\dots \\ &= n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times f(1) \end{aligned}$$

Then, if this unfolding continues, we need to retrieve $f(1), f(0)$ and then negative valued f values such as $f(-1), f(-2), \ldots$ and this might continue ad infinitum. For this reason, for every recursive function we define a **base case** or **basis** or **boundary case** that terminates the recursion. For our case we set $f(1) = 1$ (an $f(0) = 1$ would also have worked). Then

$$\begin{aligned} f(n) &= n \times f(n-1) \\ &\dots \\ &= n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times f(1) \\ &= n \times (n-1) \times \ldots \times 3 \times 2 \times 1 \\ &= n! \end{aligned}$$

(c) Copyright A. Gerbessiotis. All rights reserved.

S2. A recursive MATLAB function: Factorial. Function `sub9fact.m` calls itself in line 5 below. It takes one argument n. If the value of n is equal to 1 this value is returned. This is what we call the **base case** or **basis** of the **recursion**. If the value of n is not equal to 1 (i.e. it is not covered by the base case), then it is computed recursively by line 5. This is what we call a `recursive call`, where a function calls itself but with an argument value smaller than that of the enclosing function. The argument/parameter of `sub9fact` in line 1 is n but that of the recursive call in line 5 is $n-1$ and $n-1 < n$. What this tells us is that `sub9fact(n) = n * sub9fact(n-1)`. In order to find `sub9fact(n)` for a given n, we must unfold the recursion i.e. call repeatedly until the execution terminates promptly (this happens when the base case is reached).

```
1 function z= sub9fact(n)
2  if (n==1)
3    z=1;
4  else
5    z= n * sub9fact(n-1);
6  end
7 end
```

Thus if we want to compute `sub9fact(5)` this will be equal, by way of line 5, to `5* sub9fact(4)`, but then `sub9fact(4)` will be equal to `4* sub9fact(3)`, and `sub9fact(3)` will be equal to `3* sub9fact(2)`, and `sub9fact(2)` will be equal to `2* sub9fact(1)`.

The recursion then terminates since `sub9fact(1)` does not require a recursive call: line 3 is executed to provide the value for $n = 1$. The return value is the value of z obtained through line 3 and thus `sub9fact(1)` returns 1.

Up to that time the recursion went deeper by unfolding previous recursive calls. As soon as the base is reached this unfolding stops, and we can just start moving up again by combining the available results. Thus since `sub9fact(2)` is equal to `2* sub8fact(1)` we obtain that `sub9fact(2)=2`. Similarly `sub9fact(3) =3 * sub9fact(2)=6` and `sub9fact(4)=4*sub9fact(3)=4*6=24` and finally `sub9fact(5)=5*sub9fact(4)=120`. Thus in a recursive call going down means unfolding the recursion and going up means combining available results to obtain finally what we are looking for.

The following page shows these steps.

```
sub9fact(5) =                                                  Function sub9fact(5)
              5  *                sub9fact(4)      =          1st recursive call: Depth 1 : Down
                   ------------------------------
              5  * (4 *           sub9fact(3) )    =          2nd recursive call: Depth 2 : Down
                        -------------------------
              5  * (4 * (3 *      sub9fact(2) ))   =          3rd recursive call: Depth 3 : Down
                             ---------------------
              5  * (4 * (3 *(2 *  sub9fact(1))))   =          4th recursive call: Depth 4 : Base
              5  * (4 * (3 *(2 *      1      )))   =          Base case value returned    : Base
              5  * (4 * (3 *(         2      )))   =          Value computed    : Depth 3 : Up
              5  * (4 *               6       ))   =          Value computed    : Depth 2 : Up
              5  * (                 24        )   =          Value computed    : Depth 1 : Up
                             120                   =          Function sub9fact(5) evaluated!
```

(c) Copyright A. Gerbessiotis. All rights reserved.

S4. Fibonacci sequence. In another example, we revisit the Fibonacci sequence problem by providing a recursive formulation for it.A recursive formulation is implicit to the definition of the problem: $f(n) = f(n-1) + f(n-2)$. The code of function `sub9fib` that implements a recursive Fibonacci sequence computation, uses this in step 5. The values $f(1) = 1$ and $f(2) = 2$ become the base case. The base case is described in lines 2-4 of `sub9fib`, and the value provided is that in line 3.

```
1 function z= sub9fib(n)
2  if (n==1) | (n==2)
3    z=1;
4  else
5    z= sub9fib(n-1)+sub9fib(n-2);
6  end
7 end
```

We show how the recusion unfolds and the results combined for $f(50)$ i.e. `sub9fib(5)`.

```
sub9fib(5)=
          (                       sub9fib(4)            +            sub9fib(3)       ) =
                                /         \                        /     \
          ((         sub9fib(3)       +   sub9fib(2)   ) + (sub9fib(2)   +  sub9fib(1))) =
                      /   \
          (((sub9fib(2) + sub9fib(1)) +      1         ))+(    1         +      1      ) =
          (((   1       +    1      ) +      1         ))+               2               =
          ((            2             +      1          )+               2               =
          (                           3                  +               2            ) =
                                                         5
```

Function `sub9fib` is a **divide-and-conquer** algorithm that can be used to compute $f(n)$, the n-th Fibonacci number. A divide-and-conquer algorithm is an algorithm that solves a problem of some size n by splitting this problem into smaller subproblems (say for Fibonacci of sizes $n-1$ and $n-2$), then solves the subproblems the same way (i.e. recursively) and then combines the solutions of the subproblems (i.e. by adding `sub9fib(n-1), sub9fib(n-2)`) to obtain the solution of the original problem (i.e. `sub9fib(n)`).

A divide-and-conquer algorithm or solution to a problem can be very nicely expressed in the form of a recursive program. Function `sub9fib` is such a formulation. It is elegant simple and easily derived.

However one need to be alert of the pitfalls of a recursive solution. An recursive program when activated, maintains multiple copies of itself active. This activation consumes resources.

One might observe that line 5 can be a cause of signficant problems. Suppose we are interested in determining `sub9fib(8)`. This would require finding `sub9fib(7)` and `sub9fib(6)` respectively: this is the first level of recursion. However, `sub9fib(7)` requires finding `sub9fib(6)` and `sub9fib(5)`, and `sub9fib(6)` requires finding `sub9fib(5)` and `sub9fib(4)` in the second level of recursion. We have too much duplication of work.

For example, with reference to the Figure 1, the first 3 levels of recursion require finding `sub9fib(4)` a total of four times. In addition `sub9fib(2)` and `sub9fib(1)` will be evaluated a total of 13 and 8 times respectively (if we did not lose track of the evaluations!). So elegance comes at a cost of redundancy for naive implementation of recursive functions.

(c) Copyright A. Gerbessiotis. All rights reserved.

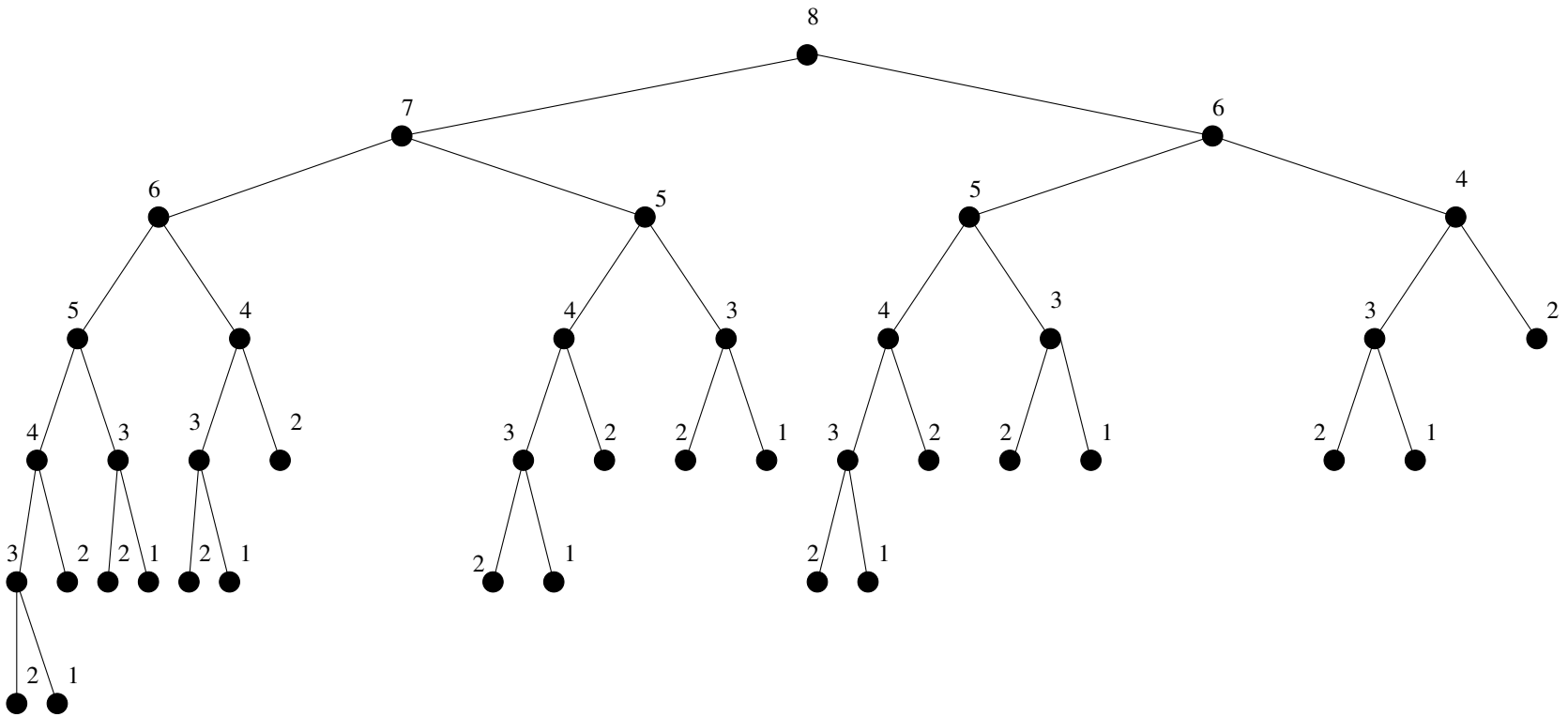


Figure 1: Fibonacci tree for recursively evaluating `sub9fib(8)`; node shows the index of the Fibonacci number computed at that node

We observe that the longest branch in the tree of Figure 1 for `sub9fib(8)` is the left-most branch of length 6 (equal to the root value minus 2) and the shortest is the right-most branch of length 3 (equal to the root value minus 2 and divided by 2). Had the root value been n instead of 8, the left-branch (the deepest) would have been of length $n-2$ and the rightmost branch (the shallowest) would have been of length $(n-2)/2 \approx n/2$, with the total number of nodes in the tree between $2^{n/2}$ and 2^{n-2}.

(c) Copyright A. Gerbessiotis. All rights reserved.

Because of these problems we want to make sure that `sub9fib(k)` for any $k < n$ will be computed just once and not several times. In fact one can show that in computing `sub9fib(n)` the number of evaluations of the various `sub9fib(k)` is a number between roughly $2^{n/2}$ and 2^n or 2^{n-2}, i.e. an exponential number of evaluations. We want to perform just $n-2$ evaluations and no more than that!

2^n grows much faster than n (or $n-2$). For $n = 2$, we have that $2^2 = 4$ and $4 > n = 2$, but for $n = 10$ we have $2^n = 1024$ and n is still 10 but now $1024 \gg 10$. For $n = 32$, we have $2^{32} \approx 4,000,000,000$! way much larger than 32.

Thus a recursive `sub9fib` uses exponential space (number of calls as depicted in the figure) and time (to complete those calls). The solution of Subject 8 is much more efficient. It uses only n space and $n-2$ time for the same computation. The latter is preferable to the former.

The question that then arises is whether we can reduce the amount of computation performed in a recursive solution.

How can we reduce the amount of evaluations? We use a technique called **memoization**. (Note that an `r` is missing intentionally.) We keep track of which evaluations have been performed and which have not been performed in the form of an array similarly to the incremental method. For the case of the Fibonacci sequence we just need two values, that can be stored into two variables.

```
1 function [fn fn_1] = mfibonacci(n)
2   if n < 3
3      fn=1; fn_1=1;
4   else
5      [ fn_1 fn_2] = mfibonacci(n-1);
6      fn = fn_1 + fn_2;
7 end
```

(c) Copyright A. Gerbessiotis. All rights reserved.

The memoized version `mfibonacci` returns two values for `mfibonacci(n)`, these are `mfibonacci(n)` and `mfibonacci(n-1)`, the newly and previously computed values. Why would one do so? The answer is simple: those two values are needed to add them up and return their sum as the value of `mfibonacci(n+1)`, the next value.

Thus lines 2-4 provide the first two values $f(1), f(2)$ stored in `fn_1 , fn` respectively.

When `mfibonacci(n)` is to be computed, we issue just one recursive call `mfibonacci(n-1)`. This will return not onlu `mfibonacci(n-1)` but also `mfibonacci(n-2)`. The return values are stored in `fn_1 , fn_2` in line 5, which are then added in line 6 to provide for `mfibonacci(n)`.

If you need to be convinced of the merits of having `mfibonacci` over `sub9fib`, there is one easy way to convince yourselves. On the AFS lab machines, run `mfibonacci(25)` and then run `sub9fib(25)` respectively. The former will complete and print the value of the 25-th Fibonacci number almost immediately whereas the other one will drag for a while and it will take several seconds before it prints anything. Notwithstanding the merits of memoization, a non-recursive approach such as that of `sub8fibonacci` is still more efficient!

(c) Copyright A. Gerbessiotis. All rights reserved.

S7. A simple example: Decimal to binary conversion

Function `convert.m` shows how one can convert a decimal into a binary through a recursive function. (Function `iconvert.m` eliminates recursion altogether and is available through the protected area of the course web-page.)

The idea is simple. For an input decimal number m, if m is odd then its binary representation ends with an 1, otherwise it ends with a zero. Oddness or evenness is determined by `mod(m,2)` being 1 or 0 respectively. Thus `convert(mod(m,2))` prints the last digit, according to the obvious rules of lines 6-9.

To that partial results (rightmost bit) we just need to attach the binary representation of the remaining part of m which is `floor(m/2)`. To get the whole we need to concatenate both parts. This is the effect of lines 10-12. The result is `convert.m`.

```
1  function s=convert(m)
2  %assuming m is a nonnegative integer
3  %returns its binary represenation as
4  %a string
5
6  if(m==0)
7      s='0';
8  elseif(m==1)
9      s='1';
10 else
11     s=[convert(floor(m/2)),convert(mod(m,2))];
12 end
```

(c) Copyright A. Gerbessiotis. All rights reserved.

S8. Profiling MATLAB programs

MATLAB allows us to `profile` a piece of MATLAB code and figure out how much time is spent on each line of a MATLAB program or function. The command `profile on` activates the profiler, `profile off` deactivates it, and `profile clear` clears all collected results to start a new profiling. A report on the data collected can be issued by a `profile report`. Thus a sequence `profile on` , `profile clear`, follow by a MATLAB M-file program execution and concludes with a `profile report` is the standard sequence of using the profiler.

Consider function `test1.m` that fills an array `my` with x elements. The ii-th element of x would be ii^2. The problem with this piece of code is the oft-warned comment: a dynamically allocated array/vector is a bad programming practice in MATLAB. You can observe for example that the elapse time for say `test1(1000)` is considerable. You can record elapsed time by enclosing the function call in a pair of `tic` (starting) and `toc` (ending) calls. A `profile report` will in fact alert you of the problem of dynamic array allocation of line 4. You can speed up the computation by a factor of 10 or more by rewriting the code as in `test2.m` where one preallocates array `my`.

```
1 function y= test1(x)
2
3 for ii=1:x
4   my(ii) = ii ^ 2 ;
5 end
6 y=my;
7 end
```

```
1 function y= test2(x)
2
3 my =zeros(x,1);
4 for ii=1:x
5   my(ii) = ii ^ 2 ;
6 end
7 y=my;
8
9 end
```

(c) Copyright A. Gerbessiotis. All rights reserved.

An even more efficient approach is to vectorize all relevant computations. This is shown in `test3.m`

```
1 function y= test3(x)
2
3 my = zeros(x,1);
4 my = 1:x;
5 my = my .^ 2;
6 y=my;
7
8 end
```

As a final note, one might observe that the copy of `my` to `y` is redundant! We could return `my` instead of `y` instead to avoid this copy.

(c) Copyright A. Gerbessiotis.