Lecture 1: Introduction to Dynamic Programming (PART 1)

1) Revision Question[1]

(RQ1) What is a recurrent relation? Give an example.

(RQ2) What is a recursive algorithm? Give an example.

These topics were covered in Lecture 3 of the AAD course. We review parts of that lecture below.

1.1. Recurrence Relations

Recurrent relations are mathematical equations that are defined in terms of themselves. These relations constitute the basis of mathematical induction.

Examples

Polynomial linear recurrence relation: a_n = a_{n-1} +1 where a_1=1 what is the value of where a_n

Polynomial linear recurrence relation: a_n = $1/(1 + a_{n-1})$

Exponential recurrence relation: $a_n = 2 * a_{n-1}$ where a_n

The recursion stops for a base case, e.g., a_1=1.

Recurrence relations are used to express the complexity of recursive functions.

Exercise 1. Solve the recurrence $na_n = (n-2)a_{n-1} + 2$ for n> 1 with $a_1 = 1$

Exercise 2. Solve the recurrence: $a_n = 2a_{n-1}$+1 for n>1 with $a_1 = 1$

Exercise 3. Solve the recurrence: $a_n = \frac{n}{n+1}a_{n-1} + 1, n > 0$ with $a_0 = 1$

Recurrence relations are useful to help us understand recursive function and are also useful when calculating the complexity of recursive algorithms.

Recursive algorithms can use the following techniques:

[1] Shorthand RQ

1.2. Simple recursion

Example 1. Counting elements in a list.
If the list is empty, return zero.
Otherwise,
Step past the first element and count the remaining
Add one to the result.

Example 2. Factorial
The iterative version would be:

```
 8 def iterative_factorial(n):
 9     result = 1
10     for i in range(2,n+1):
11         result *= i
12     return result
13
14 def factorial(n):
15     if n == 1:
16         return 1
17     else:
18         return n * factorial(n-1)
19
```

```
 8
 9 def fib(n):
10     if n == 0:
11         return 0
12     elif n == 1:
13         return 1
14     else:
15         return fib(n-1) + fib(n-2)
16
17 def fibi(n):
18     a, b = 0, 1
19     for i in range(n):
20         a, b = b, a + b
21     return a
22
23
24 memo = {0:0, 1:1}
25 def fibm(n):
26     if not n in memo:
27         memo[n] = fibm(n-1) + fibm(n-2)
28     return memo[n]
29
```

Example 3: The Fibonacci Numbers

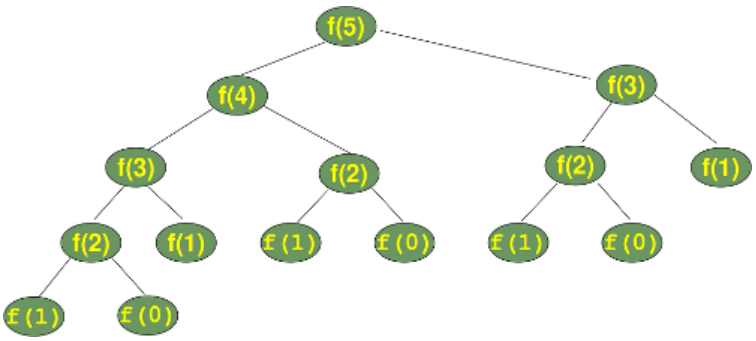
In the figure on the left we have three implementations.

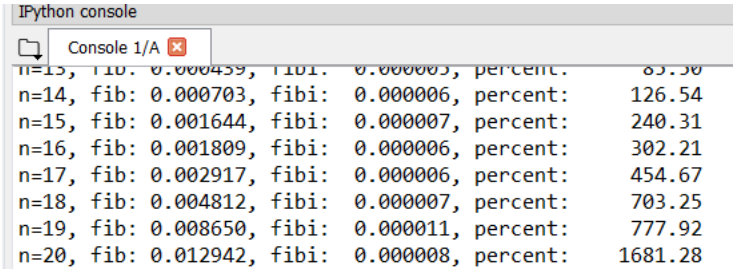
Fib – is the recursive one; Fibi – the iterative one and Fibm is a recursive version that uses a memory to remember previous calculations. We will explain below.

The recursive definition of Fibonacci is much slower than the iterative one. For n= 15 the iterative solution is 240 times faster than the recursive solution**.**

The reason for such a bad performance of the recursive algorithm is that it repeats

calculations several times. For example for fib(5), the calculations of fib(3) is done twice and the calculation of fib(2) is done 3 times. For n sufficiently large these repetition will slow down the performance dramatically. (See the graph below).





You can measure the time each algorithm takes to run for n up to 20.

```
 7
 8 from timeit import Timer
 9
10 t1 = Timer("fib(10)","from fibonacci import fib")
11
12 for i in range(1,20):
13     s = "fib(" + str(i) + ")"
14     t1 = Timer(s,"from fibonacci import fib")
15     time1 = t1.timeit(3)
16     s = "fibi(" + str(i) + ")"
17     t2 = Timer(s,"from fibonacci import fibi")
18     time2 = t2.timeit(3)
19     s = "fibm(" + str(i) + ")"
20     t3 = Timer(s,"from fibonacci import fibm")
21     time3 = t3.timeit(3)
22     print("n=%2d, fib: %8.6f, fibi:  %7.6f, fibm:  %7.6f, \
23         percent: %10.2f, percent: %10.2f" % \
24         (i, time1, time2, time3, time1/time2,time3/time2))
25
```

The recursive algorithm can be improved by remembering the calculations made for previous steps. This is done in the version of the suction **fibm** above.

Is there a faster algorithm that calculates the factorial? Perhaps an improved iterative algorithm?

The answer is yes, using what is called dynamic programing which is a kind of reclusiveness that reuses values, except that works backwards using an iterative methods.

1.3. The Fibonacci Using Dynamic Programming

```
def dp_fib(n):
    fib={}
    for k in range(n):
        if k<=2:
            f=1
        else:
            f=fib[k-1]+fib[k-2]
        fib[k]=f
    return fib[n-1]
```

Notice that this algorithm is not recursive but that it simulates the calculation of a recurrent relation using a table **`fib`** to memorize the values that were calculated.

We will show other examples of using dynamic programming but before we do that lets attempt to define dynamic programming technique.

2) Definition of the Dynamic Programming (DP)

A **DP** is an algorithmic technique used in optimization problems. It is usually based on a recurrent formula and one (or some) starting states. A sub-solution of the problem is constructed from previously found ones.

DP solutions have a polynomial complexity which assures a much faster running time than other techniques like recursion, backtracking, brute-force etc.

The following steps are part of an algorithm using dynamic programming:

i) Find the patters/ structure of the problem (or the recurrent relation)
ii) Formulate conditions that are satisfied by an optimal solution.
iii) A recursive **value** that characterizes an optimal solution, i.e., how the different values of the optimal sub-problems combined into the overall value of the optimal solution
iv) Calculate this **value** from bottom up, or backwards.
v) Construct the solution along the steps described above.

In the calculation of the factorial, these steps are clear.

We will now look at another example.

The algorithm is given on the next page.

Example 4 (Longest ordered string)

Given a string, what is the longest sub-string whose characters are ordered.

For example, in ABC the longest sub-string that is ordered is ABC. In DBCAEFG, the longest substring is BCEFG of length 5. The sub-strings of length 4 are DEFG and AEFG.

Let us review the components of an algorithm that uses dynamic programming for this problem.

i) The structure/ pattern is the string containing letters of the alphabet
ii) The condition is that all characters are ordered w.r.t. a given ordering.
iii) This value will be the length of an ordered sequence

We will now discuss the last steps iv and v.

Suppose we have the optimal sequence O, i.e., the sub-sequence that is ordered and of maximum length. Suppose that the first element of this sequence is on position i. Suppose that the length is L(O)=x.

Two conditions are satisfied:

A) The optimal sub-string of the string to the right if i cannot have a length greater than x.
B) The element on position i is smaller than the first element of any subsequence of length x-1 to the right of i.

So get the longest sequence starting from i, we would need to first find the optimal sub-sequence of length x-1 to the right of i, whose first element is less or equal to the element on position i. This leads to the following recursive formula:

$$\text{length(i)} = Max_{j=i+1, S[i] \leq S(j)}^{n-1} length(j) + 1$$

where S is the sequence for which we need to find the longest ordered sub-string. To calculate the length of the longest ordered sub-sequence we need to calculate the lengths of the longest sub-string that starts with each element of the sequence. These are stored in an array called ***maxims.***

We have the basic case: ***maxims[n-1]=1***

```
maxims[i]=1
next[i]=-1
for (j = i+1; j < n; j++)
   if (S[i] <= S[j] && maxims[i] <= maxims[j]){
         maxims[i] = maxims[j] + 1
         next[i] = j
   }
}


maxims[n-1] = 1
next[n-1] = -1
for (i= n-2; i>=0; i--)
      maxims[i]=1
      next[i]=-1
      for (j = i+1; j < n; j++)
         if (S[i] <= S[j] && maxims[i] <= maxims[j]){
               maxims[i] = maxims[j] + 1
               next[i] = j
         }
      }
}
```

For example the algorithm above will generate the following arrays maxims and next for S:

S	= { 77,	1,	34,	2,	3,	545,	5464,	32,	34,	5,	654,	43,	22,	42}
maxims	= { 3,	6,	3,	5,	4,	2,	1,	3,	2,	3,	1,	1,	2,	1}
next	= { 5,	3,	5,	4,	7,	6,	-1,	8,	10,	12,	-1,	-1,	13,	-1}

The longest increasing sub-sequence is {1,2,3,32,34,654}

To find out the length of the maximum sub-string we calculate the maximum of the array maxims and the start position of that sub-string is read from the array next.

Practice Problems:

P1) Find the longest sub-sequence common to two sequences.
P2) Find the longest increasing sub-sequence that is common to two given sequences.
P3) Find the longest sub-string common to two sequences.
P4) Find the longest increasing sub-string common to two sequences.
P5) We read **n** natural numbers from the keyboard. Write a program that prints the highest sum that can be formed using those numbers (each number is included once in the sum) which is divisible to n. The program should also print the numbers that make up the sum.