CS3110: Assignment 7 Solutions (Dynamic Programming) July 12 2017

Q1. The last step is either A, B or C. This means that the number of moves is T(n-1) followed by A, T(n-2) followed by B or T(n-3) followed by C:

1) we can formulate the recurrence: T(n)=T(n-3)+T(n-2)+T(n-1)

2)

```
Def T(n):
     if n < 0:
          return 0
     else if n == 0:
          return 1;
     else:
          return T(n - 1) + T(n - 2) + T(n - 3)
```

We can draw the recursion tree and calculate T(n)=O(3^n)

3.)

```
Def T(n, table):
     if n < 0:
          return 0
     else if n == 0:
          return 1;
     else if table[n] is not None:
          return table[n]
     else:
          table[n] = T(n - 1) + T(n - 2) + T(n - 3)
     return table[n]
```

4) The complexity will be O(n) (the size of the table)

Q2. Reconstructing the path backward, at every position we can either go left or up, so the recurrence would be:

Part 1

1) T(n) = t(x-1,y)+T(x,y-1)

We only need to take care of the marginal situations. Let's start with a simple version:

```
%%time
def count_path(x,y):
    if x<0 or y<0:
        return 0
    if (x==1 and y==0) or (x==0 and y==1):
        return 1
    return (count_path(x-1,y )  #Go Left
            +count_path(x,y-1)) #Go up
print count_path(20,10)
```

Output:

```
30045015
CPU times: user 28.9 s, sys: 80 ms, total: 28.9 s
Wall time: 28.8 s
```

Now let's add memoization:

```
%%time
table={}
def count_path(x,y):
    if (x,y) in table:
        return table[(x,y)]
    if x<0 or y<0:
        return 0
    if (x==1 and y==0) or (x==0 and y==1):
        return 1
    table[(x,y)]= (count_path(x-1,y )  #Go Left
            +count_path(x,y-1)) #Go up
    return table[(x ,y)]
print count_path(20,10)
```

Output:

```
30045015
CPU times: user 0 ns, sys: 0 ns, total: 0 ns
Wall time: 1.49 ms
```

Compare the performance time of the two approaches!

3) The computation time is again the size of the table O(xy)

Part 2)

```
def get_path(x, y, path) {
    path.append((x,y));
    if (x == 0 && y == 0):
        return true
    success = false;
    if (x >= 1 && isLegal(x - 1, y)):  #Try left
        success = get_path(x - 1, y, path); // # Go left
    if (not success and y >= 1 and isLegal(x, y - 1)): # Try up
        success = get_path(x, y - 1, path); #Go up
    if success:
        path.append(p)
    return success;
```

3) The computation time is still the size of the table O(xy)

Q3). **[note: I provided an implementation in an IPython notebook. Read the following explanation fiest and then check it out at: https://github.com/asajadi/snippets/blob/master/cfparser.ipynb]**

Let’s replace all the shapes and emojis by uppercase and lowercase letters (S is the starting symbol). For those of you with a background in formal language theory, this type of rules are called “**Chomsky Normal Form**”:

R0: S → AB
R1: S → AA
R2: A → BC
R3: A → a
R4: A → b

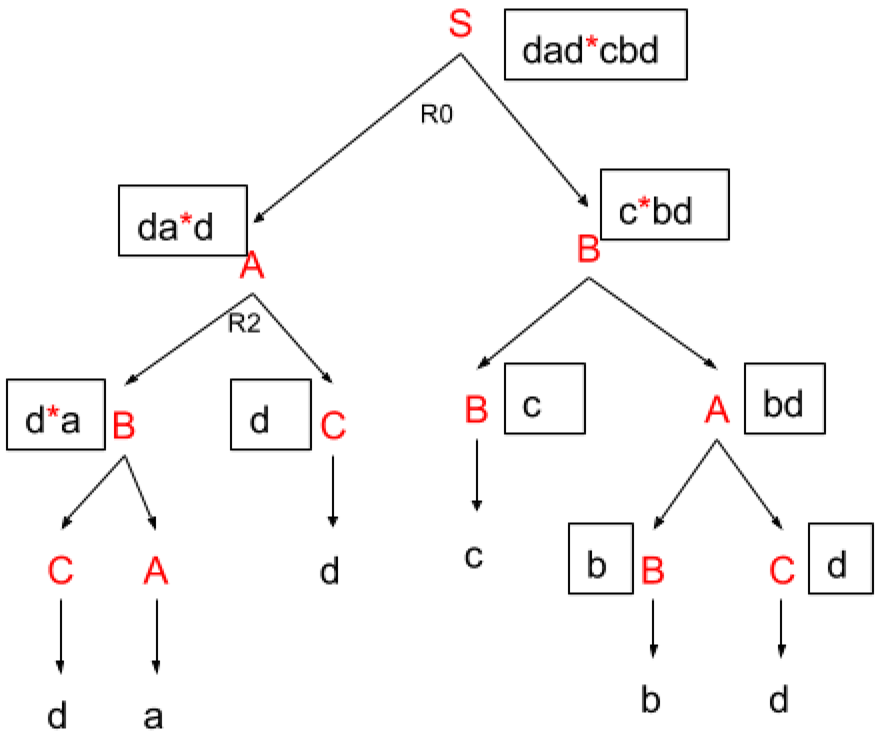
R5: B →BA
R6: B→ CA
R7: B → c
R8: B→ b

R9: C → BB
R10: C → d

And the string to generate (parse) will be: dadcbd

A) Let’s take a look at how we can generate the sentence by merely guessing:

My first guess is if I break it from the third position using R0,(dad*cbd), the left half (dad) can be generated from A, and the right half (cbd) from B. Then for the left half, we can split from the second position and use R2, in other words, we guess that the left part (da) can be generated by B and the second part by C, and so on until we hit a terminal (a.k.a alphabet), which can be simply checked. The complete tree looks like this:



B) Here the optimal substructure refers to the process of obtaining the solution for a problem by combining solutions for its subproblems. In this case, we have: ***A string w can be generated from S iff there is a way to split w to xy such that S → XY is among the rules and X can generate x and y can generate Y.***

C) Let’s define our variables first:
R: List of the rules. Every r in R is in this form : A → BC. We save A in r.lhs and BC in r.rhs
w: given string
Chart: This is the dynamic programming table. Every entry of it has a list of pairs. For example
`chart['abcd']= [(1,2),(2,3)]`
denotes that you can generate this string from rule number 1 if you split it at the 2nd position, or rule no 3 if you split it at 3’rd

```
def generate(R,w, chart):
    if w is a single emoji::
        For all r in R such that r.rhs = w
           #r_index is the number of the rule, Rr_index: A --> w
            chart[e].append(r_index,1)

    for every possible way of splitting w to x and y
        let k be the splitting point
        generate(R, x, chart)
        generate(R, y, chart)
        for every combination of elements ((rx_index,kx),
(ry_index_ky)) in  (xlist and ylist):
        If there exist a rule r such that  r.rhs=rx.lhs ry.lhs:
            charte[w].append((r,k))
```

At the end, we can generate the list of the rules by looking at the entry at *chart[string]*, finding the rule starting from S with a splitting point k and continue breaking the sentence until we reach to a terminal (an emoji, or a lowercase letter)
D) The iterative implementation is called CYK algorithm (https://en.wikipedia.org/wiki/CYK_algorithm)

Q4)

- Longest path problem [page 382 of CLRS]
    - There you can find a nice explanation of optimal substructure assumption. The argument, in summary, is: Let’s assume longest part from s to p passes from x (s → x → p), this doesn’t mean that s → x is the longest path from s to x, as there might be a longer path that goes through a node on x → p path.
- Least-cost airline fare [Wikipedia] . It’s even easier, even if you find a cheap ticket from A to B through C, there is no mathematical reason for this A-C flight to be the cheapest