CS2010 Semester 1 2012/2013

Data Structures and Algorithms II

Tutorial 08 - Dynamic Programming - The Basics

For Week 10 (22 October - 26 October 2012)

Released: Wednesday, October 17, 2012

Document is last modified on: October 17, 2012

1 Introduction and Objective

This tutorial marks the transition to the third part of CS2010: DP, although DP is not yet examinable in this Saturday's Quiz 2. In this tutorial, we will look at two classic DP problems: Computing nth Fibonacci number and Coin Change in a detailed manner. We will also discuss PS6 Subtask 1 briefly.

DP can be challenging to master but it is an important algorithm design strategy as it can solve certain problems much more efficiently than using Complete Search. Do not hesitate to ask the teaching staffs if you encounter difficulties with this topic.

This tutorial will be shorter than usual to allow students to ask questions regarding Quiz 2, if any.

2 Tutorial 08 Questions

Dynamic Programming - Basic Ideas

Q1. What are the important properties of a problem that allows for a DP solution?

Ans:
1. Optimal Substructure. (refer to shortest/longest/counting path on a DAG examples in Lec09)
If you can write a recurrence that describe the problem, this part is satisfied.
You will strengthen your recursive skills by solving these DP problems.

2. Overlapping Subproblems. (refer to shortest/longest/counting path on a DAG examples in Lec09)
Certain subproblems are revisited more than once, usually many times.
This is something that we want to avoid, by using DP table.
This is the key part of the last topic in CS2010 syllabus: Space-Time trade-off.

Keep this in mind and see if other DP problems discussed in this tutorial have these two properties.

Q2. You are given the following implementation for finding the n-th Fibonacci number. Is it a DP solution? If yes, explain! If no, how to to write a DP solution (either bottom-up or top-down) for finding the n-th Fibonacci number?

Algorithm 1 fib(n)

```
if n == 0 or n == 1 then
    return n
end if
return fib(n-2) + fib(n-1)
```

Ans: The first criteria of optimal substructure is satisfied as fib(n) is a recursive function. However, the second criteria is not satisfied. Even though there are overlapping subproblems, the solutions to such subproblems are re-computed again and again. e.g. look at the computation graph for fib(5). Notice that computations for fib(0), fib(1), fib(2) and fib(3) are repeated many times. Only fib(4) and fib(5) that are only computed once in this figure.

There are two ways to write DP solution to avoid re-computations of overlapping sub problems.

The first solution is called: top-down DP where we use memo table to help the complete search recursion to check if the current computation is the first one or if it is a repeat one. If it is the first one, the recursion will proceed as per normal and store the result in memo table. If it is a repeat one, the recursion will not continue and simply report the result already stored in memo table previously.

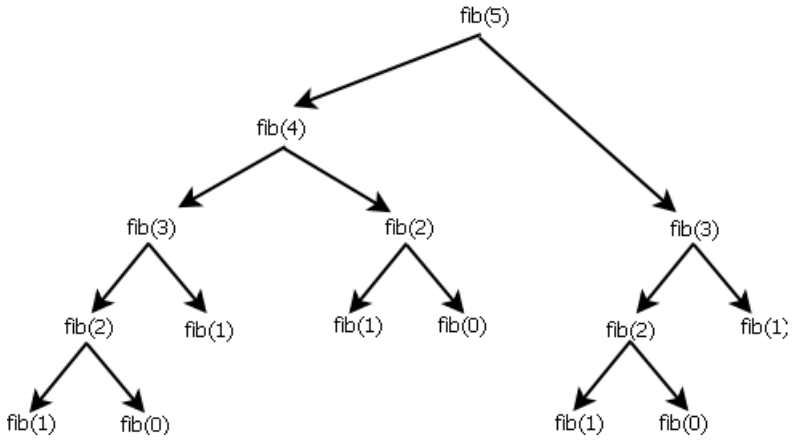

Figure 1:

// initially, the memo table is set to all -1
memo[0] = -1, memo[1] = -1, memo[2] = -1, ..., memo[n] = -1

Algorithm 2 fib_dp(n)

if $n == 0$ or $n == 1$ **then** ▷ The base cases (the easy cases) remain as they are
  return n
end if
if memo[n] != -1 **then** ▷ If this subproblem has been computed before (the value is no longer -1)
  return memo[n] ▷ Do not compute again, simply report this value
end if
return memo[n] = fib_dp(n-2) + fib_dp(n-1) ▷ Compute and store the result in memo table.

The computation graph for fib_dp(5) is as shown in Figure 2. There is no redundant computation as we memoize the solutions of subproblems. In fact, do you see that the computation graph is a DAG? Furthermore, can you see that fib_dp(n) actually computes the number of paths to fib(1) = 1?

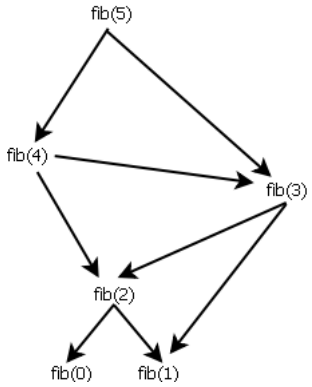

Figure 2:

We will also use this opportunity to show the second solution: bottom-up DP. This version does not use recursion, but just table (we do not call it memo table, but usually just dp table) and usually

some loops. We start from known base cases and then use the information that we already know to compute the optimal value for other subproblems according to a certain topological order of the (computation) DAG.

Algorithm 3 bottomup_fib_dp(n)

```
fib[0] = 0, fib[1] = 1                          ▷ These are the two known base cases
for int i = 2; i <= n; i++ do
    fib[i] = fib[i - 1] + fib[i - 2]   ▷ with the previous two values, we can compute the current value
end for
```

Now, do you realize that in bottom-up DP, the computation DAG is the same as in Figure 2 but with ALL arrow direction reversed, i.e. you start from fib(0) and fib(1), then point arrows to fib(2). Then, fib(1) and fib(2) point arrows to fib(3), and so on.

Both DP runs in $O(n)$. Top-down analysis: There are $O(n)$ states, each state is computed two times. So overall time complexity: $O(2n) = O(n)$. Bottom-up analysis: It is clear that the for loop runs from 2 to n (inclusive), or $O(n-1) = O(n)$.

Q3. Given a target value n in cents, and a list C of k different coin denominations (each coin denomination is also given in terms of cents), what is the minimum number of coins that we must use in order to obtain the amount n? You can assume we have an unlimited supply of coins of any type, and the smallest coin denomination is a 1 cent coin.

e.g. Given $n = 10$, $k = 2$ and $C = \{1,5\}$, we can use
1). Ten 1 cent coins = 10x1 = 10. Total coins used = 10
2). One 5 cents coin + five 1 cent coins = 1x5 + 5x1 = 10. Total coins used = 6
3). **Two 5 cents coins = 2x5 = 10. Total coins used = 2** → Optimal

Now your tasks:
A). Model the coin change problem above as a DAG and show how to solve it with standard graph algorithm that you have learned in Lecture05!
B). Write both top-down and bottom-up DP solution for this problem!

Ans:

A). We can model this coin change problem as a DAG (see Figure 3):
1.) We represent each value from 0 to n as a node and label them as such.
2.) We link node u to v if value(u)-{some coin denomination i} == value(v)

Then solving this coin change is actually a SSSP problem on an unweighted DAG!
We can use a simple BFS to solve this.

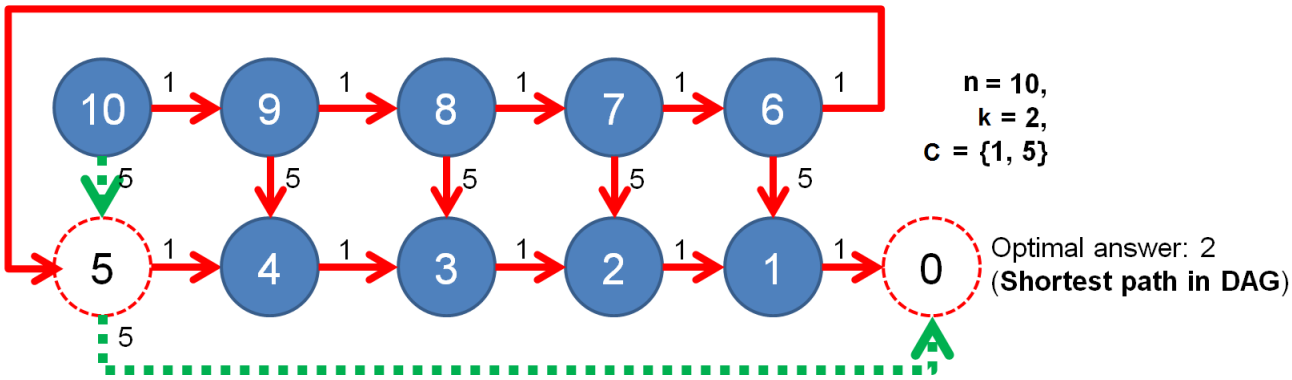

Figure 3:

B). Solving coin change with DP:

The major drawback of the graph solution in part A). is that you have to build the graph (the DAG) explicitly. We do not have to do that. We can let recursion (top-down DP) or carefully constructed loops (bottom-up DP) to achieve the same feat.

Top-down DP: Let v be the remaining cents we want to change and change(v) be the minimum number of coins needed to change v.

Base cases:
change(0) = 0
change(<0) = ∞

Recurrence relation:
change(v) = min(1 + change(v - $C[i]$)) for all i in [0..k-1]

// memo is an array of size N+1 (because we need index 0 to N, inclusive)
// we initialize this memo table with all -1, because this value -1 is not used in the problem (there is nothing as -1 number of coin)
memo[0] = -1, memo[1] = -1, ..., memo[N] = -1.
Call change(n) in main program. If change(n) reports ∞, then there is no way to make change for N cents; otherwise, report the value of change(n). This top-down approach runs in $O(nk)$, because there are $O(n)$ states and each state needs a loop that runs k iterations, so the overall time complexity is $O(nk)$.

Algorithm 4 change(v)

```
if v == 0 then                                                    ▷ Base case
    return 0
end if
if v < 0 then                                                     ▷ Invalid value
    return ∞          ▷ Return a very large number so that this is not selected as the answer.
end if
if memo[v] != -1 then                                  ▷ Already computed before
    return memo[v]                                          ▷ Do not recompute
end if
memo[v] = ∞
for i = 0 to k − 1 do
    memo[v] = min(memo[v], 1 + change(v-C[i]))                  ▷ The recursive case
end for
return memo[v]
```

The bottom-up DP again has similar structure as the top-down version, just that this time we use DP table and loops. This one is clearly $O(nk)$ but maybe not intuitive for students who are more inclined towards recursion.

Algorithm 5 bottom up DP for coin change

```
for i = 1 to n do                                                 ▷ Initialization
    change[i] = ∞
end for
change[0] = 0                                                        ▷ Base case
for i = 1 to n do     ▷ For the next coin values, from 1 to n, because that is the topological order
    for j = 0 to k − 1 do
        if i − C[j] >= 0 then                                  ▷ To prevent negative index
            change[i] = min(change[i], 1 + change[i − C[j]])
        end if
    end for
end for
the answer is at change[n]
```

Problem Set 6

Q4. Discussion of PS6 Subtask 1 (a quick one)

Ans for this standard longest path in DAG problem:
In this subtask, basically the *topological order* is already fixed: {0, 1, 2, ..., V-1}. We can just relax the edges based on this fixed topological order to get the longest path on this DAG from source vertex 0 to vertex V-1. Note that since the input graph is very small, finding neighbors of a vertex using Edge List is still acceptable.

For the other subtask, you should know what to do :). Yes, PS6 is another easy PS to score well.