CS 141, Fall 2017 Homework 7

Posted: November 13th, 2017 Due: November 20th, 2017

Name:Zhenxiao qi

Student ID #:500654348

- You are expected to work on this assignment on your own
- Use pseudocode, Python-like or English to describe your algorithms. Absolutely no C++/C/Java
- When designing an algorithm, you are allowed to use any algorithm or data structure we explained in class, without giving its details, unless the question specifically requires that you give such details
- Always remember to analyze the time complexity of your algorithms
- Homework has to be submitted electronically on Gradescope by the deadline. No late assignments will be accepted

Problem 1. (25 points)

Let $A = \{a_1, a_2, \ldots, a_n\}$ be a set of n positive integer and let T be another integer. Design a dynamic programming algorithm that determines whether there exists a subset of A whose total sum is *exactly* T. Analyze the *time*- and *space-complexity* of your solution. For instance, if $A = \{4, 5, 17, 23, 11, 2\}$ and $T = 35$ the algorithm should return TRUE because the subset $\{5, 17, 11, 2\}$ sums to 35. For the same set of numbers if we choose $T = 31$ the problem has no solution, and the algorithm will return FALSE.

Answer:

We can use a dynamic programming table to solve this problem. First we initial the memory table with all $False$, mem[i,0]=$True$ and $mem[a_i, a_i] = True$.Then we fill the memory table from 0 to T (column) with row from a_n to a_1

We use the following rule to do so:

$mem[i, k] = mem[i + 1, k] \cup mem[i, k - a_i]$

For example, $mem[11, 13] = mem[2, 13] \cup mem[11, 13 - 11] = True$. $mem[23, 30] = mem[11, 30] \cup mem[23, 30 - 23] = False \cup False = False$

The table is of size nT, so *time-complexity*=$O(nT)$

The *space-complexity* of my solution is $O(nT)$ but actually we can store our result in one row, the *space-complexity* could be $O(T)$

Problem 2. (25 points)

You have a set of n jobs to process on a machine. Each job j has a processing time t_j, a profit p_j and a deadline d_j. The machine can process only one job at a time, and job j must run uninterruptedly for t_j consecutive units of time. If job j is completed by its deadline d_j, you receive a profit p_j, otherwise a profit of 0. You can assume that all parameters are integers, and that the jobs are sorted in increasing order of deadline. Give a dynamic programming algorithm to the problem of determining the schedule that gives the maximum amount of profit. Analyze the time- and space-complexity of your solution.

Answer:

This problem is kind of like 0-1 knapsack problem. let's define k as the start time of jobs, for the first row of the mem_table, if $k + process\_time[i] < deadline[i]$, then we can get $profit[i]$. We then use the following rule determine the maximum profit. $mem\_table[i,k] = MAX(mem\_table[i+1,k], profit[i] + mem\_table[i, k+process\_time[i+1]])$. It means if time permits, we will do job i, and if not, the profit stays the same. The algorithm is as follow.

```
def func(deadline[],profit[],process_time[]):
  for k in range(0,deadline[-1]:
    for i in range (n,0):                          #k refers to start time
      if (k+process_time[i]<deadline[i]):
        if(i==0):
          mem_table[i,n]=profit[i]
        else if(k+process_time[i]>deadline[i]): #do no take this job
          mem_table[i,n]=mem_table[i+1,n]
        else :                                   #determine weather take this job or not
          mem_table=max(mem_table[i+1,k],profit[i]+mem_table[i,k+process_time[i+1]])

  return mem_table

```

The mem_table is nxd_n, so the time-complexity is nd_n, and the space-complexity is also nd_n

Problem 3. (25 points)

Let A be a $n \times m$ matrix of 0's and 1's. Design a dynamic programming $O(nm)$ time algorithm for finding the largest square block of A that contains 1's only.

Hint: Define the dynamic programming table $l(i, j)$ be the length of the side of the largest square block of 1's whose bottom right corner is $A[i, j]$.

Answer:
Firstly, we initial the first row and column of dynamic programming table.
If $A[0, j] = 1$, $l[0, j] = 1$,if $A[0, j] = 0$, $l[0, j] = 0$.
If $A[i, 0] = 0$, $l[i, 0] = 0$.
If $A[i, 0] = 1$, $l[i, 0] = 1$.
Then we calculate the rest of dynamic programming table using the following rule.
$l[i, j] = min(l[i-1, j], l[i, j-1], l[i-1, j-1]+1)$. It means if the adjacent elements are all 1, then $l[i, j]$ should be $1 + 1 = 2$, which refers to a $2X2$ square block of 1. By doing so, we can get the final table and the largest square of block of 1.
The size of table $l(i, j)$ is of same size with $A(i, j)$, which is nXm, so the time complexity is $O(nm)$

Problem 4. (25 points)

A string y is a *palindrome* if $y^R = y$, where y^R is the reverse of y. Given a text x a partitioning of x is a *palindrome partitioning* if every substring of the partition is a palindrome. For example, `aba|bbb|a|bb|a|b|aba` and `aba|b|bbabb|ababa` are two palindrome partitioning of $x =$ `ababbbabbababa`. Design a dynamic programming algorithm to determine the coarsest (i.e., fewest cuts) palindrome partitioning of x. In the example, the second partition (3 cuts) is optimal. Analyze the time- and space-complexity of your solution.

Answer:
Let $L[i,j]$ equals to the length of the longest palindrome from index i to j.
If $i = j$, $L[i,j] = 1$.
If $x[i] = x[j]$ and $j = i+1$, $L[i,j] = 2$.
If $x[i] = x[j]$ and $j > i+1$, $L[i,j] = L[i+1, j-1] + 2$, which means in order to find L[i,j], we need the length of palindrome contained by palindrome from i to j.
Else if $x[i]! = x[j]$ and $j > i+1$, $max(L[i+1,j], L[i,j-1])$

Then how to determine the coarsest palindrome partitioning of x? In the length table we built, the top right should be the longest palindrome, and the lengths next to it should be the second longest palindromes. So what we need to do is choosing palindromes from top right until the sum of length equals to the original string. The algorithm is as follow.

```
def palindrome(string[]):
  n=len(string[])
  for i in range(0,n):
    L[i,i]=1
  for m in range (2,n+1):
    for i in range(0,n-m):
      j=i+m-1
      if(string[i]==string[j] and m==2):
        L[i,j]=2
      else if (string[i]==string[j]):
        L[i,j]=L[i+1,j-1]+2
      else
      L[i,j]=MAX(L[i,j-1],L[i+1,j])
```

The dynamic programming table is $\frac{n^2}{2}$. So the time-complexity is $\frac{n^2}{2}$ and the space-complexity is also $\frac{n^2}{2}$