Brigham Michaelis
November 2nd 2017
A00364835

Homework 5

Problem 1.
(20 points) In class, we have studied the following knapsack problem. We are given n items of sizes a1 , a2 , . . . , an , which are positive integers. We are also given a knapsack of size M , which is also a positive integer. We want to determine whether there is a subset S of the items such that the sum of the sizes of the items in S is exactly M . If there exists such a subset S, we call S a feasible subset.
In fact, it is possible that there are multiple feasible subsets. In this exercise, you are asked to design an O(nM) time dynamic programming algorithm to compute the number of feasible subsets.
For example, suppose the item sizes are 3, 5, 6, 2, 7, and M = 13. Then, there are two feasible subsets {6, 7} and {2, 5, 6}. Thus your algorithm should return 2.

Description
First we show the Psuedo Code for the algorithm shown in class.
Definitions:

```
1<=k<=m
0<=i<=n
P(0,k) = 0;
P(i,0) = 1;

for(i=1 to n)
  for(k=1 to m)
    if(k>=ai)
      P(i,k) = max{P(i-1,k),P(i-1,k-ai)};
    else
      P(i,k) = P(i-1,k);
return P(n,m)
```

First we discuss some preliminaries about this problem.

1. We do not need to enumerate the subsets, we just need to return the number of feasible subsets.
2. In class we discussed the algorithm to determine if there exists a feasible subset. But that is all this algorithm does.
3. For this problem we are just going to modify the algorithm discussed in class.

The algorithm discussed in class uses a boolean array to represent whether or not a sum is possible given a subset of the items. Our modification consists of making this boolean array an integer array. And changing the statement "max{P(i-1,k),P(i-1,k-ai)}" to "P(i-1,k) + P(i-1,k-ai)". Then when we return P(n,m), this will return how many feasible subsets there are in the set.

Psuedo Code
Not required

Correctness
This will work because we are simply modifying the algorithm given in class, which runs in O(n*M) time.

Runtime
The Runtime is identical to the algorithm given in class O(n*m).

Sub-Problem:
For 1<=k<=m, and 0<=i<=n.
We need to determine the total number of feasible subsets whose total sum is k.
The Goal is still to return P(n,m).

Dependency Relation:

```
    if(k>=ai)
      P(i,k) = P(i-1,k) + P(i-1,k-ai);
    else
      P(i,k) = P(i-1,k);
```

Problem 2.
(20 points) This is a problem from a student during his interview with Goldman Sachs in Salt Lake City.

Given a set A of n positive integers {a1 , a2 , . . . , an } and another positive integer M , find a subset of numbers of A whose sum is closest to M . In other words, find a subset A′ of A such that the absolute value |M − ∑a∈A ′ a| is minimized, where ∑a∈A ′ a is the total sum of the numbers of A ′ . For the sake of simplicity, you only need to return the sum of the elements of the solution subset A ′ without reporting the actual subset A ′.

For example, suppose A = {1, 4, 7, 12} and M = 15. Then, the solution subset is A ′ = {4, 12}, and thus your algorithm only needs to return 4 + 12 = 16 as the answer.

Design a dynamic programming algorithm for the problem and your algorithm should run in O(nK) time (note that it is not O(nM)), where K is the sum of all numbers of A.

Description
This is just a modification of the algorithm given in problem 1. Instead of listing the possibilities from 0 to M. We are going to list all the possibilities from 0 to K and then return the size of the solution subset who’s sum is closest to M. This is found by simply counting how close a non-zero value is to M and returning it’s associated sum.
Psuedo Code
Not required
Correctness
This will work and run in O(n*K) time because we are simply modifying the algorithm from problem 1 and instead of going from 0 to M we are going from 0 to K.
Runtime
This will run O(n*K) time just like the algorithm in problem 1 running in O(n*M) time.

Sub-Problem:
We need to find the subset whose sum is closest to M.
The Goal is to find the “feasible knapsack” size that is closest to M.
For 1 to k, and 0<=i<=n.
The Goal is still to return P(n,closest_to(m)).
Dependency Relation:
The dependency relation is the same as the previous problem. But when we are finished building the table we are going to work through the last column of the table and start at M and move up and down one at a time until we hit a non-zero value.

Problem 3.
(20 points) Here is another variation of the knapsack problem. We are given n items of sizes a1 , a2 , . . . , an , which are positive integers. Further, for each $1 \le i \le n$, the i-th item a i has a positive value value(ai) (you may consider value(ai) as the amount of dollars the item is worth). The knapsack size is a positive integer M.

Now the goal is to find a subset S of items such that the sum of the sizes of all items in S is at most M (i.e., $\sum_{ai \in S} ai \le M$) and the sum of the values of all items in S is maximized (i.e., $\sum_{ai \in S}$ value(ai) is maximized).

Design an O(nM) time dynamic programming algorithm for the problem. For simplicity, you only need to report the sum of the values of all items in the optimal solution subset S and you do not need to report the actual subset S.

Description
We are going to use the same approach for this problem as we used in problem 1. But we are just going to modify the code given in problem 1. The idea is simple, instead of keeping track of whether or not a knapsack size is possible or keeping track of how many feasible subsequences there are, we are going to keep track of the “max-value” of each possible knapsack size. Then when we are done we return the knapsack size with the greatest value that is less than or equal to M. The implementation of this idea is straightforward. We modify slightly the algorithm given in class, and when we come across any non-zero value we add to the k-value the size of the knapsack and overwrite the appropriate value only if the current value is greater than the previous value. That way even if we have multiple values for a single knapsack size only the greater value will be used.
This new code looks like the following:

```
P(0,k) = 0;
P(i,0) = 1;

m>=k>=0
0<=i<=n

for(i=1 to n) // loop from 1 to n.
  for(k=m to 0) // loop from m to zero. We are working from the top down.
    P(i,k) = P(i-1,k); // we immediately set the current value we are looking at to the value to the left.
    if(P(i-1,k)) // if the value to the left is non-zero.
      if(k == 0 && (val(ai) > P(i,ai))) // if k is equal to zero and the value of ai is greater than the current
                                                          //value then
        P(i,ai) = val(ai); // assign the new value.
      else
        possible_sum = P(i-1,k) + val(i); // assign the variable called “possible_sum”
        if((k+ai) > M) // if k + “the size” of the object is greater than “M” we do nothing.
          continue;
        if(P(i,k+ai) < possible_sum) // if our current value is less than the “possible_sum”
          P(i,k+ai) = possible_sum; // assign it.
return P(n,m);
```

Now for this code we are going to start with the same initial conditions as in problem one. But now instead of using “1’s” to indicate where possible knapsack sizes are. We are going to keep track of the sums that are happening. We are going to add the new values in reverse order and we are going to keep track of the feasible subsequence that produces that largest value for each knapsack size k. Thus when a new feasible subsequence for a knapsack size becomes possible we will first compare that value to the current value, and if it is larger it gets replaced. Other wise nothing happens.

Now when we are finished we simply scan through the last column of the matrix and return the largest value in that column.

PsuedoCode
Code is shown above
Correctness
This works because it is a modification of the algorithm given in problem 1. Namely we are using the values of the items and storing the max's instead of "1's" or "0's".
Runtime
The runtime is O(n*M), again this is a modification from the first problem with no change to the runtime.
Sub-Problem:
We need to determine which feasible subset has the highest total value whose overall size is less than or equal to M.
For 1<=k<=m, and 0<=i<=n.
The Goal is still to return P(n,greatest_value_less_than_or_equal_to_m(m)).
Dependency Relation:
This is the same as in problem 1. The crucial difference being:
We are keeping track of feasible subsequences and values. As mentioned in class the dependency relation is the code that is placed inside the for loops.

```
P(i,k) = P(i-1,k);
  if(P(i-1,k))
    if(k == 0 && (val(ai) > P(i,ai)))
      P(i,ai) = val(ai);
    else
      possible_sum = P(i-1,k) + val(i);
      if((k+ai) > M)
        continue;
      if(P(i,k+ai) < possible_sum)
        P(i,k+ai) = possible_sum;
```

The solution to this problem was discovered while working with Josh Dawson, a student in the class.
Problem 4.
4. (20 points) Given an array A[1 . . . n] of n distinct numbers (i.e., no two numbers of A are equal), design an O(n^2) time dynamic programming algorithm to find a longest monotonically increasing subsequence of A. Your algorithm needs to report not only the length but also the actual longest subsequence (i.e., report all elements in the subsequence).

Here is a formal definition of a longest monotonically increasing subsequence of A (refer to the example given below). First of all, a subsequence of A is a subset of numbers of A such that if a number a appears in front of another number b in the subsequence, then a is also in front of b in A. Next, a subsequence of A is monotonically increasing if for any two numbers a and b such that a appears in front of b in the subsequence, a is smaller than b. Finally, a longest monotonically increasing subsequence of A refers to a monotonically increasing subsequence of A that is longest (i.e., has the maximum number of elements).

For example, if A = {20, 5, 14, 8, 10, 3, 12, 7, 16}, then a longest monotonically increasing subsequence is 5, 8, 10, 12, 16. Note that the answer may not be unique, in which case you only need to report one such longest subsequence.

Description
This problem is easy once the solution is known. We are going to remove the items one by one from the set and follow the rules below.

Treat the first item as a base case and then with every subsequent item one of two things will happen.
1. It can build off a previous base case.(In which the set becomes a new base case, building off of the largest valid base case available)
2. It can't build off of any current base cases.(In which it becomes a new base case)

We will loop through the array and continue the above steps until it is apparent which set is the largest.

Psuedo-Code
Not Required.
Correctness
This algorithm runs in O(n^2) time because as it pulls an item from the set it looks at all the previous "base-cases" to see of the item can be applied to any of them. If it can it selects that biggest one available and creates a new base-case at the same time.
Run-time
This runs in O(n^2) time, as shown above.
Sub-Problem:
We need to determine the largest subset of a given set in O(n^2) time.
Our sub-problem is finding the largest subset of P(i). Where we are looking for the largest subsequence from 0 to i. And then we are looping through all the possible sizes of i up to n.
Dependency Relation:
The dependency relation consists of determining whether the current item we are looking at can be added to an existing base-case. If yes we create a new base-case and add the item to the set. If not it becomes a new base case.