JONATHAN PETERSEN
A01236750
CS 5050 - ADVANCED ALGORITHMS

Homework 5

Dynamic Programming

1. Find an O(nM) algorithm that calculates the number of feasible subsets of the knapsack problem studied in class.

Given:

a knapsack with size M,

an ordered set of n positive integers A,

an integer i such that $0 \leq i \leq n$,

an integer k such that $0 \leq k \leq M$,

a_i as the ith element of A.

This solution is very similar to the solution to the problem discussed in class. We define a subproblem to be a function f that reports the number of feasible solutions for a given i and k and relates those subproblems such that a subproblem for a given i and k is equal to a sum of the number of feasible subsets to reach the same k value with fewer elements from A plus the number of ways the given k value may be reached using the given element. In other words, that the value of f(i, k) is equal to the number of ways k can be reached with or without a_i. When this is run with f(n, M) it will return the total number of ways M can be reached using all n numbers.

Pseudocode is as follows:

```
for (i = 0 to n)
     f(i, 0) = 1;
for (k = 1 to M)
     f(0, k) = 0;
for (i = 1 to n)
     for (k = 1 to M)
          f(i, k) = f(i-1, k-ai) + f(i-1, k);
return f(n, M);
```

2. Given a value M and a set of positive integers A, design an algorithm to find the subset of A whose sum is closest to M in O(nK) time, where K is the sum across A.

We'll define the subproblem as a function f that reports the sum of the first i elements of A which is closest to a given value k. Each subproblem can be defined in terms of smaller subproblems by comparing the closest sum of i - 1 elements with the closest-sum of the first i elements, and whichever is closer becomes the value of the function f for those parameters. To find the result, simply locate the minimum value of f when i = n.

Pseudocode for the problem is as follows:

```
for (i = 1 to n)
      f(i, 0) = 0
for (k = 0 to K)
      f(0, k) = -M;
for (i = 1 to n)
      for (k = 1 to M)
            if (abs(f(i-1, k-ai)+ai - M) < abs(f(i-1, k) - M) || k-ai == -M)
                  f(i, k) = f(i-1, k-ai) + ai;
            else
                  f(i, k) = f(i-1, k);
minDistance = -M;
for (j = 1 to K)
      if (abs(f(n, j)) < minDistance)
            minDistance = f(n, j);
return minDistance;
```

2.1. Modify the Knapsack problem such that every item has an additional value attribute. Construct an O(nM)-time algorithm to report the total value of a feasible subset of items, where the total value is maximized.

Given the same setup as in problem #1:

Once more, this solution is similar to the knapsack problem we solved in class. It has the same subproblems, but their relations have changed and the base cases have changed. For the base cases, every entry where k or i is zero will also have a value of zero. For the dependency relation, we will still relate the subproblems based on the result of the subproblem for the same k with or without the current a_i, but instead of checking if that subset of A contains a feasible subset for M or finding the sum of the number of ways that combination can be reached, we will instead return whichever of the two has a larger value, adding a special case that if k - a_i is exactly zero (adding a_i to the current subset is exactly equal to k), we add the value of a_i to the total of the subset without ai.

For example, if we had a sequence of numbers whose combined sizes were 5 and whose combined values were 18, and if we were trying to reach a k value of 9, then if we were computing the result for an a_i with size 4 and value 6, we would find that 9 - 5 = 4, therefore we would compare 18 + 6 = 24 with the highest combined value for this k score without a_i = 4.

Pseudocode for the problem is as follows:

```
for (i = 1 to n)
     f(i, 0) = 0
for (k = 0 to M)
     f(0, k) = 0;
for (i = 1 to n)
     for (k = 1 to M)
          if (f(i-1, k-ai) > 0 || k-ai == 0)
               prevTotal = f(i-1, k-ai) + ai.value;
          else
               prevTotal = 0;
          f(i, k) = max(prevTotal, f(i-1, k) );
return f(n, M);
```

3. Design an algorithm that reports the longest monotonically increasing subsequence of a given set of numbers in $O(n^2)$ time.

Given:

An array A of positive integers,

We define a subproblem of this to be a function L that takes a single parameter b, and returns the length of the longest monotonically increasing subsequence ending at b. We will define the function L(b) as one plus the largest L(a), where $A[a] < A[b]$ and $a < b$. The longest monotonically increasing subsequence is the largest value of L(b), and the base case is $L(1) = 1$.

In order to recreate the longest subsequence, we will keep track of which a each L(b) came from, such that we can backtrack up from the greatest L(b) to its first ancestor, adding each step on the way to the front of a list. When we have reached the first ancestor, the list will contain the subsequence in order.

Pseudocode for this solution is as follows:

```
L(1) = 1
maxB = 0;
for (b from 2 to n)
    int value = 0;
    int parent = -1;
    for (a from 1 to b - 1)
        if (A[a] < A[b] && L(a) > value)
            value = L(a);
            b.parent = a;
    L(b) = 1 + value;
    if (L(b) > maxB)
        maxB = b;
while (b.parent != -1)
    result.push_front(b);
    b = b.parent;
return result;
```