CS5050 Advanced Algorithms

Spring Semester, 2018

Assignment 5: Dynamic Programming

Student: Tanner Kvarfordt - A02052217

Due Date: 3:00 pm, Tuesday, Apr. 3, 2018 (**at the beginning of CS5050 class**)

Note: For each of the following problems, you will need to design a dynamic programming algorithm. When you describe your algorithm, please explain clearly the *subproblems* and the *dependency relation* of your algorithm.

1. **(20 points)** The knapsack problem we discussed in class is the following. Given an integer M and n items of sizes $\{a_1, a_2, \ldots, a_n\}$, determine whether there is a subset S of the items such that the sum of the sizes of all items in S is exactly equal to M. We assume M and all item sizes are positive integers.

   Here we consider the following *unlimited version* of the problem. The input is the same as before, except that there is an unlimited supply of each item. Specifically, we are given n item sizes $a_1, a_2, \ldots, a_n$, which are positive integers. The knapsack size is a positive integer M. The goal is to find a subset S of items (to pack in the knapsack) such that the sum of the sizes of the items in S is exactly M and each item is allowed to appear in S multiple times.

   For example, consider the following sizes of four items: {2, 7, 9, 3} and $M = 14$. Here is a solution for the problem, i.e., use the first item once and use the fourth item four times, so the total sum of the sizes is $2 + 3 \times 4 = 14$ (alternatively, you may also use the first item 4 times and the fourth item 2 times, i.e., $2 \times 4 + 3 \times 2 = 14$).

   Design an $O(nM)$ time dynamic programming algorithm for solving this unlimited knapsack problem. For simplicity, you only need to determine whether there exists a solution (namely, if there exists a solution, you do not need to report the actual solution subset).

   Solution: This problem (and solution) differs only slightly from the knapsack problem and solution discussed in class. If the last item, a_n, is in S, then there is a subset from the remaining $n-1$ items whose sum is $M - a_n$. If a_n is not in S, then there is a set of the remaining $n-1$ items whose sum is M. Note also though, that if a_n is evenly divisible by M, then a solution exists purely using a_n. So overall, that is,

$$P(n, M) = \begin{cases} P(n-1, M) & \text{if } a_n \in S \\ P(n-1, M - a_n) & \text{if } a_n \notin S \\ 1 & \text{if } M \bmod a_n = 0 \end{cases}$$

   Where the result of P is either a 1 indicating 'true' or a 0 indicating 'false.' So we have the following subproblems:

For each $0 \le i \le n$, $0 \le k \le M$, $P(i,k)$ denotes whether or not there is a subset from the first i items whose sum is k. Hence the dependency relation is

$$P(i,k) = \max\{P(i-1,k), P(i-1,k-a_i), -(a_i \bmod k)+1\}$$

Therefore our pseudocode will be much the same as the original knapsack problem from class. However, the key difference is that items may be repeated. To account for this, notice that if an item a_i is evenly divisible by a knapsack size k, then a solution exists purely using that one item. Therefore, anytime an item is evenly divisible by the knapsack size, we must mark it as having a solution. Therefore the pseudocode is as follows below:

```
for k = 1 to M
    P[0, k] = 0
for i = 0 to n
    P[i, 0] = 1
for i = 1 to n
    for k = 1 to M
        if (k % a_i) == 0 // these lines are all that differ between
            P[i, k] = 1   // this algorithm and the one discussed in class
        else if k >= a_i
            P[i, k] = max{P[i - 1, k - a_i], P[i - 1, k]}
        else P[i, k] = P[i - 1, k]
return P[n, M]
```

This algorithm clearly runs in $O(nM)$ time.

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 $\{a_1, a_2, \ldots, a_n\}$ 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 $|M - \sum_{a \in A'} a|$ is minimized, where $\sum_{a \in A'} a$ 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.

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

   Solution: To ensure an $O(nK)$ time complexity, the first thing to do is sum up all elements of A and see if that sum is less than or equal to M. This takes $O(n)$ time, and if the sum of the elements of A is less than or equal to M, then we already know the solution is simply that sum. Now when we program our dynamic solution, our for loops will loop over A and some $K \le M$. To develop a dynamic programming solution, examine a single element, a_i. a_i is either in A' or it is not. If it is not, the problem size has decreased by one, and M remains the same. if it is in A', then the problem size has decreased by one, and M has decreased by

a_i. This gives us the following subproblems:
For $0 \le i \le n$ and $0 \le j \le m$, $P(i,j)$ = the sum K of $a_1 \dots a_i$ nearest to j. Therefore our dependency relation is

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } a_i \notin A' \\ P(i-1,j) & \text{if } |j - P(i-1,j)| = B \\ a_i & \text{if } |j - a_i| = B \\ a_i + P(i-1, j-a_i) & \text{if } |j - (a_i + P(i-1, j-a_i))| = B \end{cases}$$

where

$$B = \min\{|j - a_i|,\ |j - (a_i + P(i-1, j-a_i))|,\ |j - P(i-1,j)|\}$$

So the general pseudocode is as follows:

```
total = 0
for a_i in A
    A_sum = A_sum + a_i
if A_sum <= M then return total
for i from 0 to n
    P[i][0] = 0
for j from 0 to M
    P[0][j] = 0
for i from 0 to n
    for j from 0 to M
        if a_i <= j // a_i is shorthand for A[i]
            B = min{|j-a_i|, |j-(a_i + P(i-1, j-a_i))|, |j-P(i-1, j)|}
            if (j - a_i| == B) then P[i][j] = a_i
            else if (|j - P[i-1][j]| == B) then P[i][j] = P[i-1][j]
            else P[i][j] = a_i + P[i-1][j-a_i]
        else
            B = min{|j-a_i|, |j-P(i-1, j)|}
            if (|j - a_i| == B) then P[i][j] = a_i
            else P[i][j] = P[i-1][j]
return P[n][M]
```

Note that the indices of the pseudocode may vary depending on if you index arrays beginning at 0 or 1. This solution is $O(n)$ when M is greater than or equal to the sum of the elements of A, and $O(nM)$ otherwise. In the latter case, note that K is generally very near to M since we first checked that M was less than the total sum of the elements of A.

3. **(20 points)** Here is another variation of the knapsack problem. We are given n items of sizes $a_1, a_2, \dots, a_n$, which are positive integers. Further, for each $1 \le i \le n$, the i-th item a_i has a positive value $value(a_i)$ (you may consider $value(a_i)$ 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., $\sum_{a_i \in S} a_i \le M$) and the sum of the values of all items in S is **maximized** (i.e., $\sum_{a_i \in S} value(a_i)$ 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.

Solution: This dynamic programming solution is easier to reason about than some. Consider an element, a_i. If a_i can fit in the knapsack, then we have a value of $value(a_i)$ at a minimum, with $M - a_i$ room left in the knapsack. Consider $w \le M - a_i$. At maximum, we have value $value(a_i) + value(w)$. So we have the following subproblems:
For $0 \le i \le n$ and $0 \le j \le M$, $P(i,j)$ gives the maximized value of an ideal packing of a knapsack of size j. Therefore our dependency relation is

$$P(i,j) = \begin{cases} \max\{P(i-1,j),\ value(a_i) + P(i-1, j-a_i)\} & \text{if } a_i \in S \\ P(i-1,j) & \text{if } a_i \notin S \end{cases}$$

From here, we can develop the pseudocode:

```
for j from 0 to M
    P[0][j] = 0
for i from 1 to n
    for j from 0 to M
        if (j >= A[i])
            P[i][j] = max{P[i-1][j], value(A[i]) + P[i-1][j-A[i]]}
        else
            P[i][j] = P[i-1][j]
return P[n][M]
```

The algorithm is clearly $O(nM)$.

4. **(20 points)** Given an array $A[1 \dots n]$ 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.

Solution: The LMIS [1] can be found by iterating over A and calculating the LMIS for each element a_i of A, considering only elements up to a_i. Backtracking can be used (by storing sequence indices in a separate array as they are discovered) to report the actual subsequence, while another array of length n can be used to track the length of the LMIS for each element a_i of A, considering only the elements up to a_i.

```
// Let LMIS denote Longest Monotonically Increasing Subsequence
// Let MIS denote Monotonically Increasing Subsequence
A is as given in the problem
L[0..n-1] = array storing the lengths of each MIS
S[0..n-1] = array storing the indices in A of the LMIS
for k from 0 to n-1
    L[k] = 1
for i from 1 to n-1
    for j from 0 to i-1
        if A[i] > A[j]
            L[i] = max{L[i], L[j] + 1}
            if L[j] + 1 == max{L[i], L[j] + 1}
                S[i] = j
len = largest element of L // O(n) to find this
report len as the length of the LMIS
i = index of the largest element in L // O(n) to find this
R[0..len-1] = set of indices in A that make up the LMIS // O(n) at worst
R[len-1] = i
for i = len - 1; i > 0; --i:
    R[i-1] = S[R[i]]
// Now report the LMIS
for i = 0 to len - 1:
    report A[R[i]]
```

Examining the for loops, we can see that this algorithm is $n + n^2 + n + n + n + n = O(n^2)$ at worst.
LMIS implemented in C++11:

[1] Longest Monotonically Increasing Subsequence

```
#include <iostream>
#include <cstdlib>
#include <algorithm>

const int A_SIZE = 9;
int max(int const & a, int const & b) {
    if (a > b) return a;
    return b;
}
int main() {
    int A[A_SIZE] = {20, 5, 14, 8, 10, 3, 12, 7, 16}; int L[A_SIZE]; int S[A_SIZE];
    S[0] = -1; // A garbage/debugging value. This element should never be accessed
    for (int k = 0; k < A_SIZE; ++k) {
        L[k] = 1;
    }
    for (int i = 1; i < A_SIZE; ++i) {
        for (int j = 0; j < i; ++j) {
            if (A[i] > A[j]) {
                L[i] = max(L[i], L[j] + 1);
                if (L[j] + 1 == max(L[i], L[j] + 1)) {
                    S[i] = j;
                }
            }
        }
    }
    auto max_length = std::max_element(L, L + A_SIZE);
    auto max_length_index = std::distance(L, max_length);
    std::cout << "Max length is " << *max_length << std::endl;
    int R[*max_length];
    R[*max_length - 1] = max_length_index;
    for (int i = *max_length - 1; i > 0; --i) {
        R[i - 1] = S[R[i]];
    }
    std::cout << "Subsequence is: ";
    for (int i = 0; i < *max_length; ++i) {
        std::cout << A[R[i]] << ", ";
    }
    std::cout << std::endl;
    return EXIT_SUCCESS;
}
```

Total Points: 80