CS5050 Advanced Algorithms

Philip Nelson

Spring Semester, 2018

Assignment 5: Dynamic Programming

Worked with: Ammon Hepworth, Raul Ramirez, Hailey Maxwell, and Jack Kiefer

1. **(20 points)** Here we consider the *unlimited version* of the knapsack problem. The input is an array A of n items with sizes $A = \{a_1, a_2, \ldots, a_n\}$, except that there is an unlimited supply of each item, 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.

   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: The optimal solution structure for this *unlimited version* of the knapsack problem is an array $z = [z_1, z_2 \ldots z_n]$. Let f be the unlimited knapsack problem such that $f(n, M)$ is true if a subset of the first n elements of A can be add up to M with unlimited repetition and false if not. At the end, the solution will be $f(n, M)$. We start by looking at the elements of A in reverse order and evaluate three subproblems:

   if $a_n = z_k$ and $a_n \neq z_{k-1}$: $f(n-1, M-a_n)$

   (ie. a_n is in the solution and is the last instance of a_n)

   if $a_n = z_k$ and $a_n = z_{k-1}$: $f(n, M-a_n)$

   (ie. a_n is in the solution and is not the last instance of a_n)

   if $a_n \neq z_k$: $f(n-1, M)$

   (ie. a_n is not in the solution)

   Then the dependency relationship for $f(i,j) = \{f(i-1, j-a_i) \text{ or } f(i, j-a_i) \text{ or } f(i-1, j)\}$ where or is the logical or. The base cases are $f(0, M) = \text{false}$ and $f(n, 0) = \text{true}$ because filling any sized knapsack with no elements is impossible and filling a knapsack of size zero with any element is trivially possible.

   The solution is $O(nM)$ because it constructs a 2D table size n x M which requires $O(nM)$ work. Although the computational complexity is $O(nM)$ the space complexity can be reduced to linear space if only the current and previous columns are kept.

2. **(20 points)** 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'. 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 $O(nM)$). 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'.

   Solution: The optimal solution structure for this version of the knapsack problem is an array $z = [z_1, z_2 \ldots z_n]$. Let f be the Goldman Sachs knapsack problem such that $f(n, M)$ is the sum of $A' \subseteq A$ whose sum is closest to M. We start by looking at the elements of A in reverse order and evaluate two subproblems:

   if $a_n = z_k$: $f(n-1, K - a_n)$
   (ie. a_n is in the solution)

   if $a_n \neq z_k$: $f(n-1, K)$
   (ie. a_n is not in the solution)

   Then the dependency relationship is

$$f(i,j) = \begin{cases} f(i-1, j-a_i), & \text{if } |M - f(i-1, j-a_j)| < |M - f(i-1, j)|. \\ f(i-1, j), & \text{otherwise.} \end{cases} \tag{1}$$

   The base cases are $f(0, M) = 0$ and $f(n, 0) = 0$ because filling any sized knapsack with no elements will be filled to zero and filling a knapsack of size zero with any element is also size zero.

   The solution is $O(nK)$ because it constructs a 2D table size n x K which requires $O(nM)$ work. It needs to go all the way to K because that is the worst case scenario for getting close to M. The solution will be the value in the last column that is closest to M. Although the computational complexity is $O(nM)$, the space complexity can be reduced to linear space if only the current and previous columns are kept.

3. **(20 points)** Here is another variation of the knapsack problem. We are given n items of sizes $a_1, a_2, \ldots, a_n$, which are positive integers. Further, for each $1 \leq i \leq n$, the i^{th} item a_i has a positive value $value(a_i)$. The knapsack size is a positive integer M. Now the goal is to find a subset Z of items such that the sum of the sizes of all items in Z is **at most** M (i.e., $\sum_{a_i \in Z} a_i \leq M$) and the sum of the values of all items in Z is **maximized** (i.e., $\sum_{a_i \in Z} value(a_i)$ 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 Z and you do not need to report the actual subset Z.

   Solution: The optimal solution structure for this version of the knapsack problem is an array $z = [z_1, z_2 \ldots z_n]$. Let f be the Maximum Value knapsack problem such that $f(n, M)$ is the sum of the values of Z for a given n and M. We start by looking at the elements of A in reverse order and evaluate two subproblems:

if $a_n = z_k$: $f(n-1, K-a_n)$

(ie. a_n is in the solution)

if $a_n \neq z_k$: $f(n-1, K)$

(ie. a_n is not in the solution)

Then the dependency relationship for $f(i,j) = max\{f(i-1, M-a_i)+value(a_i), f(i-1, M)\}$. The base cases are $f(0, M) = 0$ and $f(n, 0) = 0$ because the value of any sized knapsack with no elements will be zero and the value of any knapsack of size zero is also size zero.

The solution is $O(nM)$ because it constructs a 2D table size n x M which requires $O(nM)$ work. Although, the computational complexity is $O(nM)$ the space complexity can be reduced to linear space if only the current and previous columns are kept.

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

   Solution: In order to find the larges monotonically increasing subsequence, we will use a dynamic programming algorithm. Let $f(j)$ be the length of the longest monotonically increasing subsequence ending in the j^{th} element of A. Then the longest monotonically increasing subsequence would have the length $max\{f(1),\ f(2)\ \ldots\ f(n)\}$. Assuming at any point that $f(1),\ f(2)\ \ldots\ f(j-1)$ is known, then $f(j)$ can be determined by looking at $a_1\ \ldots\ a_{j-1}$ and seeing which elements are less than a_j. If an element, a_i, is less than a_j, then ending it's sequence with a_j would increase it by one. If an element, a_i, is larger than a_j, then the largest monotonically increasing sequence ending with a_j is one. The value $f(j)$ is then the maximum of all these values. For each $f(j)$ to compute, All previous values, $f(1)\ \ldots\ f(j-1)$ must be visited, therefore the time complexity is $O(\frac{n^2}{2}) \Rightarrow O(n^2)$

   Then to recover the subsequence, we backtrack from $max\{f(1),\ f(2)\ \ldots\ f(n)\}$ and when an $f(j)$ has a value one less than the max length it is the next element. Then that element's value is taken and we continue back tracking to find one less than that. The process is repeated until $f(j) = 1$ which is the first element int the subsequence. A visual representation is shown in Table 1.

A	20	5	14	8	10	3	12	7	16
$f(j)$	1	1	2	2	3	1	4	2	5
sub sequence		x		x	x		x		x

Table 1: Back Tracking

Total Points: 80