CS5050 Advanced Algorithms

Fall Semester, 2017

Homework Solution 5

Haitao Wang

1. The algorithm is similar as before with only slight changes. For each $0 \leq i \leq n$ and $0 \leq k \leq M$, we define the subproblem $p[i,k]$ as the number of feasible subsets of the first i items whose total size sum is equal to k.

   The dependency relation is as follows. As before, a feasible subset for $p[i,k]$ can either contain item a_i or not. Therefore, we have $p[i,k] = p[i-1,k] + p[i-1,k-a_i]$. (Recall that the dependency relation for the original knapsack problem discussed in class is $p[i,k] = \max\{p[i-1,k], p[i-1,k-a_i]\}$. Therefore, for this new problem, the only difference is to change the "max" operation to "+".)

   The base cases are the same as before. The pseudocode is given below in Algorithm 1. The running time is still $O(nM)$.

Algorithm 1: Knapsack

Input: A set of n items of sizes $\{a_1, a_2, \ldots, a_n\}$ and the size of the knapsack M.
Output: The number of feasible subsets.

for $k = 1$ *to* M **do**
    $p[0,k] = 0$;
end
for $i = 0$ *to* n **do**
    $p[i,0] = 1$;
end
for $i = 1$ *to* n **do**
    for $k = 1$ *to* M **do**
        if $k \geq a_i$ **then** $p[i,k] = p[i-1,k] + p[i-1,k-a_i]$ **else** $p[i,k] = p[i-1,k]$
    end
end
return $p[n,M]$;

---

2. We can use almost the same algorithm for the knapsack problem discussed in class. For each $0 \leq i \leq n$ and $0 \leq k \leq K$ (note that here it is K instead of M), we define the subproblem $p[i,k]$ exactly the same as before, i.e., $p[i,k] = 1$ if there is a subset of the first i items whose total size sum is equal to k and $p[i,k] = 0$ otherwise.

   The dependency relation is also the same as before, i.e., $p[i,k] = \max\{p[i-1,k], p[i-1,k-a_i]\}$.

The base cases are also the same. We run the same algorithm as before with two for-loops: i from 1 to n and k from 1 to K. Finally the values $p[n,k]$ for $k = 0, 1, 2, \ldots, K$ will all be computed.

Further, we do the following processing to determine the solution for the problem. We first check whether $p[n, M]$ is equal to 1. If yes, then we simply return M as the answer. Otherwise, starting from M, we find its left neighboring index l such that $p[n, l] = 1$ and the right neighboring index r such that $p[n, r] = 1$. In other words, l is the largest index with $l < M$ and $p[n, l] = 1$, and r is the smallest index with $r > M$ and $p[n, r] = 1$.

Once l and r are found, if $M - l < r - M$, then we return l as the answer; otherwise, we return r as the answer.

The total time of the algorithm is $O(nK)$.

3. The algorithm is still somewhat similar to the one we discussed in class.

   We first define sub-problems. For any $1 \leq i \leq n$ and $0 \leq k \leq M$, consider the sub-problem of finding a subset S' of the first i items $\{a_1, a_2, \ldots, a_i\}$ such that the sum of the sizes of all items in S' is at most k and the sum of the values of all items in S' is maximized. Here, we define $p[i, k]$ as the sum of the values of all items in the optimal solution S' of the above sub-problem.

   To find the dependency relation, as before, either the optimal solution subset for the sub-problem $p[i, k]$ contains the item a_i or not. Then, the dependency relation is $p[i, k] = \max\{p[i-1, k], p[i-1, k-a_i] + value(a_i)\}$.

   The bases cases are slightly different. The pseudo-code is given in the following Algorithm 2. The running time is $O(nM)$.

Algorithm 2: Knapsack-with-Values

Input: A set of n items of sizes $\{a_1, a_2, \ldots, a_n\}$ and the size of the knapsack M. Each item a_i has a positive value $value(a_i)$.

Output: Find a subset of items with maximum total value subject to the knapsack size M

for $k = 1$ *to* M **do**
    $p[0, k] = 0$;
end
for $i = 0$ *to* n **do**
    $p[i, 0] = 0$;
end
for $i = 1$ *to* n **do**
    for $k = 1$ *to* M **do**
        if $k \geq a_i$ **then** $p[i, k] = \max\{p[i-1, k], p[i-1, k-a_i] + value(a_i)\}$ **else** $p[i, k] = p[i-1, k]$
    end
end
return $p[n, M]$;

4. We will give two approaches for this problem. The first approach works as follows.

   We first define sub-problems: For each i, define $f(i)$ to be the number of elements in the *restricted* longest monotonically increasing subsequence (LMIS) of the first i elements of A, i.e., $A[1 \ldots i]$, subject to the constraint that the subsequence must include $A[i]$ at the end.

   Next we develop the dependency relation. In order to compute $f(i)$, we assume $f(j)$ for all $j = 1, 2, \ldots, i-1$ have been computed. Based on the definition of $f(i)$, for each $f(j)$ with $1 \leq j \leq i-1$, if $A[i] > A[j]$, then we can add $A[i]$ to the end of the restricted LMIS of the first j elements to obtain a restricted monotonically increasing subsequence for $A[1 \ldots i]$. To compute $f(i)$, we need to compute the restricted *longest* monotonically increasing subsequence, which can be done by finding the largest $f(j)$ such that $1 \leq j \leq i-1$ and $A[j] < A[i]$. Hence, we obtain the following dependency relation for computing $f(i)$:

$$f(i) = 1 + \max_{1 \leq j \leq i-1, A[j] < A[i]} f(j).$$

   In the base case, we have $f(1) = 1$. After $f(i)$ for all $i = 1, 2, \ldots, n$ are computed, the largest $f(i)$ corresponds to the LMIS of A. To report the sequence, for each $1 \leq i \leq n$, we use $pre[i]$ to record the index of the number in front of $A[i]$ in the restricted LMIS of $A[1 \ldots i]$. Specifically, when we compute $f(i)$ by using the above dependency relation, suppose $f(j)$ is the largest value, then we set $pre[i] = j$. Initially, we set $pre[i] = 0$ for each i. Finally, we will use the array $pre[1 \cdots n]$ to report the LMIS of A.

   Refer to Algorithm 3 for the pseudocode. The running time is clearly $O(n^2)$ because there are two for-loops.

   The second approach. The problem can also be solved by using the algorithm for the longest common subsequence problem discussed in class, in the following way. We first sort all numbers of A into a sorted sequence B. Then, the **key observation** is that a longest common subsequence of the original array A and the sorted sequence B is a longest monotonically increasing subsequence of A. Therefore, we can simply apply the algorithm for the longest common subsequence problem on A and B. The total running time is still $O(n^2)$.

Algorithm 3: Finding the longest monotonically increasing subsequence (LMIS)

Input: An array of n distinct numbers: $A[1 \dots n]$.
Output: The LMIS of A.

```
f[1] = 1;
for i = 1 to n do
    pre[i] = 0;
end
for i = 2 to n do
    max = 0;
    for j = 1 to i − 1 do
        if A[i] > A[j] and max < f[j] then
            max = f[j]; pre[i] = j;
        end
        f[i] = max + 1;
    end
end
/* next, we find the largest value f[i]                                    */
k = 1; max = f[1];
for i = 2 to n do
    if f[i] > max then
        max = f[i]; k = i;
    end
end
/* the above computes f[k] as the largest value, next, we report the LMIS   */
i = k;
report A[i];
while pre[i] ≠ 0 do
    i = pre[i];
    report A[i];
end
the above reports the LMIS in the inverse order, and we can reverse the list to obtain the normal order (we can use a stack to do the “reverse” operation and details are omitted);
```