CS5050 Advanced Algorithms

Spring Semester, 2018

Homework Solution 5

Haitao Wang

1. For this problem, we have unlimited supply of each item, and we only need to modify the dependency relation. The subproblem $p[i,k]$ is defined the same as before, i.e., $p[i,k]=1$ if we can pack a knapsack of size k by only using the items in $\{a_1,a_2,\ldots,a_i\}$ and each item can be used for unlimited times, and $p[i,k]=0$ otherwise.

   To find the dependency relation, observe that $p[i,k]=1$ if and only if (1) we can pack k with items in $\{a_1,a_2,\ldots,a_{i-1}\}$ (i.e., $p[i-1,k]=1$) or (2) we can pack $k-a_i$ with items in $\{a_1,a_2,\ldots,a_i\}$ (i.e., $p[i,k-a_i]=1$). Note that the above (2) is different from the one we discussed in class, because now we can use the item a_i for unlimited times. Hence, the dependency relation becomes $p[i,k]=\max\{p[i-1,k],p[i-1,k-a_i],p[i,k-a_i]\}$, which is equivalent to $p[i,k]=\max\{p[i-1,k],p[i,k-a_i]\}$ because $p[i-1,k-a_i]\leq p[i,k-a_i]$ always holds.

   The pseudocode is given below in Algorithm 1, which is the same as before except the dependency relation. The running time is still $O(nM)$.

Algorithm 1: Unlimited-Knapsack

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

Output: Determine whether we can exactly pack the knapsack, and each item can appear unlimited times

```
1  for k = 1 to M do
2      p[0,k] = 0;
3  end
4  for i = 0 to n do
5      p[i,0] = 1;
6  end
7  for i = 1 to n do
8      for k = 1 to M do
9          if k ≥ a_i then p[i,k] = max{p[i-1,k], p[i,k-a_i]};
10         else p[i,k] = p[i-1,k];
11     end
12 end
13 if p[n,M] = 1 then report "yes" else report "no" ;
```

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 $O(nM)$.

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 $f(i)$:

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

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

```
1  for k = 1 to M do
2      p[0, k] = 0;
3  end
4  for i = 0 to n do
5      p[i, 0] = 0;
6  end
7  for i = 1 to n do
8      for k = 1 to M do
9          if k ≥ 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]
10     end
11 end
12 return p[n, M];
```

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: $A[1 \ldots n]$.
Output: The LMIS of A.

```
1  f[1] = 1;
2  for i = 1 to n do
3      pre[i] = 0;
4  end
5  for i = 2 to n do
6      max = 0;
7      for j = 1 to i − 1 do
8          if A[i] > A[j] and max < f[j] then
9              max = f[j]; pre[i] = j;
10         end
11         f[i] = max + 1;
12     end
13 end
   /* next, we find the largest value f[i]                                  */
14 k = 1; max = f[1];
15 for i = 2 to n do
16     if f[i] > max then
17         max = f[i]; k = i;
18     end
19 end
   /* the above computes f[k] as the largest value, next, we report the LMIS */
20 i = k;
21 report A[i];
22 while pre[i] ≠ 0 do
23     i = pre[i];
24     report A[i];
25 end
26 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);
```