CSC373 Summer '22
Tutorial 3 Solutions

June 2, 2022

Q1 Coin change

Consider the problem of making change with fewest possible coins.

Input: A positive integer A, and positive integer "denominations" $d[1] < d[2] < \ldots < d[m]$.

Output: A list of "coins" $c[1 \ldots n]$ is feasible when each $c[i]$ is in d, repeated coins are allowed (i.e. it is possible that $c[i] = c[j]$ with $i \neq j$), and $\sum_{i=1}^{n} c[i] = A$. Output the smallest n such that a feasible list of coins of size n exists. If no feasible solution exists, output $n = 0$.

Example: If we only have pennies, nickels, dimes and quarters (i.e. $d = [1, 5, 10, 25]$) to make change for 30¢ (i.e. $A = 30$), then the output should be $n = 2$ (with, e.g., $c = [5, 25]$), and not $n = 3$ (with, e.g., $c = [10, 10, 10]$). If the input was $d = [5, 10, 25]$ and $A = 52$, then the output would be $n = 0$ as no solution exists.

We want to use dynamic programming to solve this problem.

(a) Define an array which stores the necessary information needed from various subproblems. Clearly identify what each entry of your array means, and how to compute the solution to the problem given your array entries.

(b) Write a Bellman equation describing the recurrence relation for the array identified above, and briefly justify its correctness. Pay close attention to the base cases.

(c) Write a bottom-up algorithm (pseudocode) to implement the Bellman equation identified above.

(d) Analyze the worst-case running time and space complexity of your algorithm. Does it run in polynomial time? Explain.

(e) Modify your Bellman equation and bottom-up implementation to compute a smallest feasible list of coins $c[1 \ldots n]$, in addition to the minimum number of coins n needed. When $n = 0$, you can return an empty list $c = []$. When necessary, define a second array to store partial information regarding the optimal solution. Analyze the worst-case running time and space complexity of your algorithm.

Solution to Q1

(a) For $i \in \{0, 1, \ldots, A\}$ and $j \in \{1, \ldots, m\}$, let $C[i, j]$ be the minimum number of coins needed to make change for amount i using denominations $d[1], \ldots, d[j]$. If it's impossible to make change in this way, let $C[i, j] = \infty$.

The desired solution is then $C[A, m]$. *(Note: Do not forget to mention this.)*

(b) The Bellman equation is given below.

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0, \\ \infty & \text{if } i > 0 \text{ and } j = 0, \\ \min\{1 + C[i - d[j], j], C[i, j-1]\} & \text{otherwise.} \end{cases}$$

Justification:

- $C[0, j] = 0$ for all j because making change for 0¢ does not require any coins.
- $C[i, 0] = \infty$ for $i > 0$ but $j = 0$ represents the fact that no change can be made for a positive amount i when no denominations are available.
- For $C[i, j]$, there are two cases regarding an optimal solution:
  - If it contains at least one coin of denomination $d[j]$, then ignoring one such coin, the remaining solution must be the optimal change for $i - d[j]$ using denominations $d[1], \ldots, d[j]$, i.e., contain $C[i - d[j], j]$ coins. Here, denomination $d[j]$ is allowed in the remaining solution because the optimal solution may contain multiple coins of denomination $d[j]$.
  - If it does not contain any coin of denomination $d[j]$, then it must be the optimal change for i using denominations $d[1], \ldots, d[j-1]$, i.e., contain $C[i, j-1]$ coins.

Because each quantity used in the expression for $C[i, j]$ has either a smaller i term or a smaller j term, the recurrence relation is not cyclic (i.e. no $C[i, j]$ depends on itself).

(c) The bottom-up algorithm is shown below. It ensures that when $C[i, j]$ is being computed, $C[\cdot, j-1]$ column is fully computed, and $C[i', j]$ is also computed for all $i' < i$. Because $C[i, j]$ only depends on entries from $C[\cdot, j]$ and $C[\cdot, j-1]$, we can delete $C[\cdot, j-2]$ column, thus only storing two columns at a time.

Algorithm 1: Bottom-Up Coin Change

```
for j = 1, ..., m do
    If j > 2, delete the column C[·, j − 2] and free up memory
    for i = 0, ..., A do
        Compute C[i, j] using the Bellman equation above
    end
end
```

(d) The algorithm computes $\Theta(m \cdot A)$ entries of array C, and each computation takes $\Theta(1)$ time. Hence, the worst-case running time is $\Theta(m \cdot A)$.

The algorithm does not run in polynomial time because its running time is linear in A, whereas the number of bits required to represent the integer A in input is only $\log A$. Hence, the running time is not polynomial in the length of the input.

Because the algorithm only stores two columns at any given time, its space complexity is $O(A)$.

Note: You can also solve this question using a 1D array. Let $D[i]$ denote the minimum number of coins needed to make change for amount i. So we do not constrain ourselves to use only certain denominations here. Then, you can write the following Bellman equation:

$$D[i] = \begin{cases} 0 & \text{if } i = 0, \\ \infty & \text{if } 0 < i < d[1] \\ 1 + \min_{j:d[j] \leq i} D[i - d[j]] & \text{otherwise.} \end{cases}$$

However, implementing this will also lead to $O(mA)$ runtime: while there are only $O(A)$ array entries now, evaluating each takes $O(m)$ time. The space complexity will also be $O(A)$ as before. One advantage of this approach is that it naturally gives $O(A)$ space complexity, without having to use the trick of constantly "deleting" the rows/columns that are no longer needed.

(e) Corresponding to the recurrence relation for C, we can design the following recurrence relation.

$$S[i, j] = \begin{cases} \perp & \text{if } i = 0 \text{ or } j = 0, \\ L & \text{if } i > 0,\ j > 0, \text{ and } 1 + C[i - d[j], j] \leq C[i, j - 1], \\ R & \text{otherwise.} \end{cases}$$

Then, we can compute C and S together as follows. Note that we now need to remember the optimal solution can be computed as follows.

Algorithm 2: Optimal Solution for Coin Change

```
for j = 1, ..., m do
    for i = 0, ..., A do
        Compute C[i, j] using its Bellman equation
        Compute S[i, j] using its Bellman equation
    end
end
Set c ← [], i ← A, and j ← m
while i > 0 and j > 0 do
    if S[i, j] = L then
        c ← [cd[j]]
        i ← i − d[j]
    else
        j ← j − 1
    end
end
```

The running time is still $O(m \cdot A)$, but the space complexity now increases to $O(m \cdot A)$.

Q2 Longest Increasing Subsequence

Consider the following Longest Increasing Subsequence (LIS) problem:

Input: Array $A[1 \dots n]$ of integers.

Output: Length of the longest subsequence S such that each element of S is strictly larger than all the elements before it.

Example: If $A = [4, 1, 7, 3, 10, 2, 5, 9]$, then the longest subsequence is $S = [1, 3, 5, 9]$ or $S = [1, 2, 5, 9]$, so the answer is 4. Note that $[6, 7, 8]$ and $[1, 2, 3]$ are not subsequences (they either include integers not in A or include integers out-of-order); $[4, 1, 7, 10]$ is not increasing; $[1, 3, 9]$ is not the longest possible.

Our goal is to design an $O(n^2)$ time DP for this problem. The bonus question asks you to reduce the running time to $O(n \log n)$.

(a) Define an array which stores the necessary information needed from various subproblems. Clearly identify what each entry of your array means, and how to compute the solution to the problem given your array entries.

(b) Write a Bellman equation describing the recurrence relation for the array identified above, and briefly justify its correctness. Pay close attention to the base cases.

(c) Write a bottom-up algorithm (pseudocode) to implement the Bellman equation identified above.

(d) Analyze the worst-case running time and space complexity of your algorithm.

(e) (Bonus Question, Warning: Difficult) Reduce the worst-case running time of your algorithm to $O(n \log n)$.

Solution to Q2

(a) For $i \in \{1, \dots, n\}$, let $L[i]$ be the length of a longest increasing subsequence of A that ends at $A[i]$ (i.e. the longest increasing subsequence of $A[1, \dots, i]$ which contains $A[i]$).

The desired solution is then $\max_{i \in \{1,\dots,n\}} L[i]$. *(Note: Do not forget to mention this.)*

(b) The Bellman equation is as follows.

$$L[i] = \begin{cases} 1 & \text{if } i = 1, \\ 1 + \max_{j \in \{1,\dots,i-1\}: A[j] < A[i]} L[j] & \text{otherwise.} \end{cases}$$

Justification:

- $L[1] = 1$ because the longest increasing subsequence in this case simply consists of $A[1]$.
- For $i > 1$, note that we must include $A[i]$ in the subsequence. Hence, if $A[j]$ is the previous element, then the remaining subsequence must be the longest increasing subsequence of $A[1, \dots, j]$ ending at $A[j]$. Optimizing over j gives the longest possible length.

(c) We can compute L in the order of $i = 1, \ldots, n$ according to the Bellman equation above.

(d) There are $O(n)$ entries in L, and computing each takes $O(n)$ time given the previous entries. Hence, it takes a total of $O(n^2)$ time to compute L, and then finding the maximum entry in L takes $O(n)$ time. Hence, the overall running time is $O(n^2)$. The space complexity is $O(n)$.

(e) For this, we use a binary search trick on top of the DP.

The trick is to maintain another array C in the bottom-up implementation. After we have processed elements $A[1 \ldots i]$ in the bottom-up implementation, the value of $C[k]$ will be the smallest value of the last element in any increasing subsequence of length k in $A[1 \ldots i]$. Note that C will always remain sorted in a nondecreasing order. (Exercise: Prove that, in fact, C must be strictly increasing.)

When processing element i of A, we can now compute $L[i] = k^*$, where k^* is such that $C[k^* - 1] < A[i] \leq C[k^*]$. This can be computed in $O(\log n)$ time using binary search in C. After this, we need to update C. It can be checked that only $C[k^*]$ needs to be updated to $A[i]$, which can be done in $O(1)$ time. (Exercise: Why don't we need to update any other entry of C?)