CSE 6140 / CX 4140

Computational Science & Engineering Algorithms

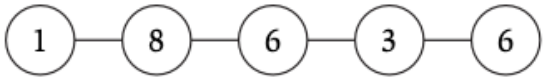
Homework 3

Please type in all answers.

1. (20 points) Let $G = (V, E)$ be an undirected graph with n nodes. Recall that a subset of the nodes is called an *independent set* if no two of them are joined by an edge. Finding large independent sets is difficult in general; but here well see that it can be done efficiently if the graph is simple enough.

Call a graph $G = (V, E)$ a *path* if its nodes can be written as $v_1, v_2, ..., v_n$, with an edge between v_i and v_j if and only if the numbers i and j differ by exactly 1. With each node v_i, we associate a positive integer weight w_i.

Consider, for example, the five-node path drawn below. The weights are the numbers drawn inside the nodes. The maximum weight of an independent set here is 14.



The goal in this question is to solve the following problem:

Find an independent set in a path G whose total weight is as large as possible.

a. Give an example to show that the following algorithm *does not* always find an independent set of maximum total weight.

```
The "heaviest-first" greedy algorithm
  Start with S equal to the empty set
  While some node remains in G
    Pick a node v_i of maximum weight
    Add v_i to S
    Delete v_i and its neighbors from G
  Endwhile
  Return S
```

Consider the weights (3) → (4) → (3) → (4) → (3)

The algorithm would pick node 2 (with weight 4) and remove node 1 and node 3 (each having weight 3), leaving us with node 4 (with weight 4) and node 5 (with weight 3). The algorithm would then pick node 4 and terminate. This gives a total weight of $4 + 4 = 8$.

However, the most optimal solution stems from choosing nodes 1, 3 and 5 (with weights of 3), and that leads to a total weight of $3 + 3 + 3 = 9$, which gives an independent set of larger weight.

Thus, the "heaviest-first" greedy algorithm does not always find an independent set of maximum total weight.

b. Given an example to show that the following algorithm also *does not* always find an independent set of maximum total weight.

```
Let S1 be the set of all vi where i is an odd number
Let S2 be the set of all vi where i is an even number
(Note that S1 and S2 are both independent sets)
Determine which of S1 or S2 has greater total weight,
  and return this one
```

Consider the weights $(2) \to (1) \to (1) \to (4) \to (1)$

The algorithm would set S_1 to be the nodes 1, 3, 5 (with weights of 2, 1, 1, respectively) and S_1 to be the nodes 2 and 4 (with weights of 1 and 4 respectively). The sum for $S_1 = 4$ and $S_2 = 5$. Since the max of 4 and 5 is 5, the algorithm would return 5.

However, the most optimal solution stems from choosing nodes 1 and 4 (with weights of 2 and 4 respectively), and that leads to a total weight of 6, which gives an independent set of larger weight.

Thus, this algorithm does not always find an independent set of maximum total weight.

c. Give an algorithm that takes an n-node path G with weights and returns an independent set of maximum total weight. The running time should be polynomial in n, independent of the values of the weights.

We define an array $dp[]$ where $dp[i]$ represents the maximum weight of an independent set considering the path up to node v_i. The key insight is that for each node v_i, we have two options:

1. Include v_i in the independent set: In this case, we cannot include its neighbor v_{i-1}, so the maximum weight will be $w_i + dp[i-2]$.
2. Do not include v_i: In this case, the maximum weight is simply $dp[i-1]$, which is the best solution considering up to the previous node.

Thus, the recurrence relation is:

$$dp[i] = \max(dp[i-1], w_i + dp[i-2])$$

We can initialize the base cases as follows:

- $dp[0] = w_0$ (the weight of the first node)

- $dp[1] = \max(w_0, w_1)$ (the maximum weight from the first two nodes)

Algorithm:

1. Initialize $dp[0] = w_0$
2. Initialize $dp[1] = max(w_0, w_1)$
3. For each node i from 2 to $n-1$, compute:

$$dp[i] = \max(dp[i-1], w_i + dp[i-2])$$

4. The maximum total weight of the independent set is stored in $dp[n-1]$

Time Complexity: This algorithm processes each node in the path exactly once, so the time complexity is $\mathcal{O}(n)$, which is polynomial in the number of nodes n.

2. (20 points) You are managing a consulting team. You need to plan their schedule for the year. For each week, you can either assign them a low-stress job or a high-stress job.

You are given arrays l and h. $l[i]$ is the revenue you make by assigning a low stress job in week i. $h[i]$ is the revenue you make by assigning a high stress job in week i.

In order for the team to take on a high-stress job in week i, its required that they do no job (of either type) in week $i-1$; they need a full week of prep time for the high-stress job. On the other hand, it is okay for them to take a low-stress job in week i even if they have done a job (of either type) in week $i-1$.

Write pseudocode of an algorithm that finds the maximum revenue you can make for the year.

Hint: Use dynamic programming. For week i, assume that you know the maximum revenue achieved up to weeks $i-1$ and $i-2$. Lets say these are $OPT(i-1)$ and $OPT(i-2)$ respectively. In week i you can either choose a low stress job (revenue $l[i]$) or high-stress job (revenue $h[i]$). You want to find a recurrence that relates $OPT(i)$ to $OPT(i-1)$ and $OPT(i-2)$.

We define an array $dp[]$ where $dp[i]$ represents the maximum revenue that can be made up to week i. The key insight is that for each week i, we have two options:

1. Choose a low stress job: In this case, we can include its neighbor $dp[i-1]$, so the maximum weight will be $l[i] + dp[i-1]$.
2. Choose a high stress job: In this case, we can include cannot include its neighbor $dp[i-1]$, so the maximum weight will be $h[i] + dp[i-2]$.

Thus, the recurrence relation is:

$$dp[i] = \max(l[i] + dp[i-1], h[i] + dp[i-2])$$

We can initialize the base cases as follows:

- $dp[0] = l[0]$ (since in the first week, we can only take the low-stress job)
- $dp[1] = max(l[1] + dp[0], h[1])$

Algorithm:

1. Initialize $dp[0] = l[0]$
2. Initialize $dp[1] = max(l[1] + dp[0], h[1])$
3. For each week i from 2 to $n-1$, compute:
$$dp[i] = \max(l[i] + dp[i-1], h[i] + dp[i-2])$$
4. The maximum total revenue in week i is stored in $dp[n-1]$

Time Complexity: This algorithm processes each week exactly once, so the time complexity is $\mathcal{O}(n)$.

3. (20 points) You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money.

Write pseudocode of a dynamic programming algorithm that returns the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return -1.

You may assume that you have an infinite number of each kind of coin.

We define an array $dp[]$ where $dp[i]$ represents the represents the fewest number of coins needed to make the amount i. For each amount i, we can try using each coin c in the coins array. If we use coin c, the number of coins needed will be $1 + dp[i - c]$ (since we use one coin c and the remaining amount is $i - c$). Thus, for each amount i, the minimum number of coins will be:

$$dp[i] = \min(dp[i], 1 + dp[i - c]) \quad \text{for each coin } c$$

We can initialize the base cases as follows:

- $dp[0] = 0$ (since no coins are needed to make an amount of 0)
- $dp[i] = \infty$ for $1 \leq i \leq amount$

Algorithm:

1. We initialize the $dp[]$ array with a large value (∞) because we are looking for the minimum number of coins. $dp[0]$ is set to 0 because 0 coins are needed to make an amount of 0
2. For each amount i from 1 to amount, we check each coin denomination. If using the coin c leads to a valid solution (i.e., $i - c \geq 0$), we update $dp[i]$ to be the minimum of its current value and $1 + dp[i - c]$.
3. After filling the $dp[]$ array, $dp[amount]$ will contain the minimum number of coins needed to make up the amount. If $dp[amount]$ is still infinity, it means the amount cannot be formed with the given coins, and we return -1.

Time Complexity: The time complexity is $\mathcal{O}(n \times m)$, where n is the amount and m is the number of coins, since for each amount, we iterate over all coins.

4. (20 points) A number of languages (including Chinese and Japanese) are written without spaces between words. You are given text in such a language, and you are required to design an algorithm to infer likely boundaries between consecutive words in the text. If English were written without spaces, the analogous problem would consist of taking a string like "meetateight" and deciding that the best segmentation is "meet at eight" (and not "me et at eight", or "meet ate aight", or any of a huge number of possibilities).

To solve the problem, you are given a function Quality that takes any string of letters and returns a number that indicates the quality of the word formed by the string. A high number indicates that the string resembles a word in the language (e.g. 'meet), whereas a low number means that the string does not resemble a word (e.g. 'eeta).

The total quality of a segmentation is determined by adding up the qualities of each of its words.

Write pseudocode of a dynamic programming algorithm that take a string y and computes a segmentation of maximum total quality. What is the running time of your algorithm? (You can treat the call to Quality as a single computational step, O(1)).

We define an array $dp[]$ where $dp[i]$ represents the maximum total quality of the segmentation of the string up to index i. For each position i, we try every possible previous position j such that the substring from j to i forms a valid word. The quality of the segmentation for index i will be the maximum quality achievable by considering segmentations that end at j plus the quality of the word formed by the substring from j to i. Thus, for each position i, the maximum quality of the segmentation is:

$$dp[i] = \max(dp[j] + \text{Quality}(y[j+1 \text{ to } i])) \quad \text{for each } j \text{ such that } 0 \leq j < i$$

We can initialize the base case as follows:

- $dp[0] = 0$ (there is no segmentation at the start of the string)

Algorithm:

1. Initialize an array $dp[]$ of length $n+1$ (where n is the length of the string y) with values $-\infty$ because we are maximizing the total quality. Set $dp[0] = 0$ since no segmentation is needed at the start
2. For each position i from 1 to n, try every possible previous position j such that the substring $y[j+1 \text{ to } i]$ forms a word, and update $dp[i]$ as:

$$dp[i] = \max(dp[i], dp[j] + \text{Quality}(y[j+1 \text{ to } i]))$$

3. After filling the $dp[]$ array, the value $dp[n]$ will contain the maximum total quality of the segmentation of the entire string
4. To reconstruct the actual segmentation, backtrack from n to 0, finding which substrings contributed to the maximum quality

The time complexity is $\mathcal{O}(n^2)$, where n is the length of the string. This is because for each position i, we iterate over all previous positions j, and the Quality function is called once for each substring, which takes constant time.

5. (20 points) A thief robbing a store wants to take the most valuable load that can be carried in a knapsack capable of carrying at most W pounds of loot. The thief can choose to take any subset of n items in the store. The i-th item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. Which items should the thief take and what is their total value? Write pseudocode to solve the problem.

We define a 2D array $dp[]$ where $dp[i][w]$ represents the maximum value the thief can obtain using the first i items with a weight limit w. The idea is to build up solutions for smaller subproblems (considering fewer items or lower weight limits) and use these solutions to construct the answer for larger subproblems. For each item i and weight limit w, we have two choices:

1. Do not take item i: The value is simply the maximum value we can obtain with the first $i-1$ items and the weight limit w, i.e., $dp[i-1][w]$
2. Take item i (if it fits within the weight limit): The value is the value of item i plus the maximum value we can obtain with the first $i-1$ items and a reduced weight limit of $w-w_i$, i.e., $v_i + dp[i-1][w-w_i]$.

Thus, the recurrence relation is:

$$dp[i][w] = \max(dp[i-1][w], v_i + dp[i-1][w-w_i]) \quad \text{if } w_i \leq w$$

Otherwise, we cannot take item i, and the recurrence simplifies to:

$$dp[i][w] = dp[i-1][w]$$

We can initialize the base case as follows:

- $dp[0][w] = 0$ for all w (since no items means zero value)

Algorithm:

1. We initialize a 2D array $dp[]$ of size $(n+1) \times (W+1)$, where n is the number of items and W is the weight limit. Each entry $dp[i][w]$ stores the maximum value obtainable with the first i items and weight limit w.
2. For each item i and weight w, we decide whether to take the item or not, based on whether it fits in the knapsack and whether taking it would provide a higher value.
3. Once we compute the maximum value, we backtrack through the $dp[]$ array to find which items were taken. If $dp[i][w] \neq dp[i-1][w]$, it means item $i-1$ was included in the optimal solution, so we add it to the list of items taken and reduce the remaining weight by the weight of the item.
4. The function returns the maximum value and the list of items taken.

The time complexity is $\mathcal{O}(n \times W)$, where n is the number of items and W is the weight limit. This is because we fill a 2D table of size $(n+1) \times (W+1)$, and for each entry, we perform a constant amount of work.