CS 344 Problem Set 4: Dynamic Programming and Greedy Algorithms

Kanghwi Lee

1. (a) Brute force algorithm:

To solve this problem using a brute force algorithm, we generate all possible subsets of the given positive integers list and calculate each subset's sum. If any subset's sum equals the given integer k, we return true; otherwise, we return false. The time complexity of this algorithm is O(2^n), where n is the number of integers in the input list. This is because there are 2^n possible subsets to consider.

(b) Dynamic programming (DP) based algorithm:

- Choice of sub problem definition:

We choose the sub problem definition S(i, j), where S(i, j) is true if a subset of the first I integers adds up to j. For this choice, we create a 2D DP table with rows representing the integers (i) and columns representing the target sum (j).

- Dependency information:

To compute S(i, j), we depend on the solutions to sub problems S(i-1, j) and S(i-1, j - a[i]), where a[i] is the value of the i-th integer in the list. We are considering whether to include the i-th integer in the subset.

- Algorithm development:

We fill the DP table using the dependency information mentioned above. For each sub problem S(i, j), if S(i-1, j) is true or S(i-1, j - a[i]) is true, we set S(i, j) to true.

- Final answer computation:

After solving all sub problems, the final answer is found in the DP table’s cell S(n, k), where n is the total number of integers in the list.

-Time complexity:

The time complexity of this DP-based algorithm is O(n * k), where n is the number of integers in the list and k is the target sum. We must fill a DP table of size n x k, and each cell's computation takes constant time.

(c) False.

The DP-based algorithm is only sometimes more efficient than the brute force algorithm for this problem. While the DP algorithm provides a more efficient solution regarding time complexity, it requires additional space to store the DP table. Suppose the target sum k is relatively small, and the number of integers in the list is not too large. In that case, the brute force algorithm may perform better in practice due to its simplicity and lesser memory consumption.

2. (a) The size of the smallest vertex cover for the given tree is 3. The smallest vertex cover for this tree is {B, C, D} or {D, F, G}

Below is the original tree

Below is the vertex cover for the above tree

(b) DP-based algorithm for Minimum Vertex Cover in Trees:

- Choice of sub problems:

We choose the sub problem definition C(v, t), where C(v, t) represents the minimum vertex cover size in the subtree rooted at vertex v when the parent of v is marked as t. We consider two choices for t: 0 (unmarked) and 1 (marked).

- Dependency information:

For a general sub problem C(v, t), it depends on the solutions of its child nodes' sub problems:

- If vertex v is marked (t = 1), then the children of v can be either marked or unmarked.
- If vertex v is unmarked (t = 0), then the children of v must be marked.
- Algorithm development:

We can compute the solutions to the sub problems using a bottom-up dynamic programming approach. For each vertex v and its two possible states (t = 0 or t = 1), we consider the optimal vertex cover size by recursively considering its children's sub problems.

- Final answer computation: The final answer for the entire tree is given by the smaller of the two cases: when the root is marked (C(root, 1)) and when the root is unmarked (C(root, 0)).

-Time complexity: The time complexity of this DP-based algorithm for finding the minimum vertex cover in trees is O(n), where n is the number of vertices in the tree. This is because we compute the solutions for each vertex's sub problems only once in a bottom-up manner.

3. (a) Among the given text documents:

- "biggreyelephant" does have a valid reconstruction.
- "nopainnogain" does have a valid reconstruction.
- "commondaylotf" does not have a valid reconstruction.

(b) DP algorithm for Valid Document Reconstruction:

- Choice of subproblems:

We choose the subproblem definition S(i), where S(i) represents whether the substring s[1...i] has a valid reconstruction.

- Dependency information: For a general subproblem S(i), it depends on the solutions of its smaller subproblems:

S(i) depends on S(j) for all valid j such that j < i. This represents checking all possible prefixes of the substring s[1...i].

- Algorithm development:

We can compute the solutions to the subproblems using a bottom-up dynamic programming approach. For each index i, we iterate through all possible prefixes j and check if the substring s[j+1...i] forms a valid word according to the dictionary. If it does, we mark S(i) as true.

- Final answer computation:

The final answer is given by the value of S(n), where n is the length of the input string.

-Time complexity: The time complexity of this DP-based algorithm for determining the valid reconstruction of a given string is O(n^2), where n is the length of the input string. This is

because we compute the solutions for each index's subproblems by considering all possible prefixes of the substring.

4. To find the contiguous subsequence of the maximum sum in an array, we can use Kadane's algorithm. Kadane's algorithm is an efficient approach that works in linear time. Here's the algorithm:

Initialize two variables: maxEndingHere and maxSoFar to track the maximum sum ending at the current index and the maximum sum found so far, respectively. Also, initialize startIdx and endIdx to keep track of the indices of the maximum subsequence.

Iterate through the array from left to right:

- For each element num at index i, update maxEndingHere:
    - maxEndingHere = max(num, maxEndingHere + num)
- Update maxSoFar if maxEndingHere is greater than maxSoFar.
- If maxEndingHere is less than or equal to zero, update maxEndingHere to zero and set startIdx to i + 1.

Once the loop is done, maxSoFar will hold the maximum sum of the contiguous subsequence.

To find the indices of the subsequence, we need to backtrack from the end. Initialize tempMax as maxSoFar, tempEnd as the last index, and tempSum as zero. Traverse the array from right to left:

- Add the current element to tempSum.
- If tempSum equals tempMax, update endIdx as tempEnd and startIdx to the point where the subsequence started.

The contiguous subsequence with maximum sum will be the subarray from startIdx to endIdx.

Answer: The contiguous subsequence of maximum sum is [43, 2], and the maximum sum is 45.

5. An algorithm to solve the Longest Common Subsequence problem in O(mn) time:

1. Create a 2D array L of size (n+1) x (m+1) where L[i][j] represents the length of the longest common subsequence of the substrings x[1..i] and y[1..j].
2. Initialize the first row and first column of the array L with zeros, as the longest common subsequence with an empty string is always zero.
3. Iterate through each element of the array L in a nested loop, starting from i=1 and j=1.

4. For each i and j, if x[i] is equal to y[j], then set L[i][j] = L[i-1][j-1] + 1. This means that the current characters of both strings contribute to the longest common subsequence.
5. If x[i] is not equal to y[j], then set L[i][j] = max(L[i-1][j], L[i][j-1]). This means we either consider the longest common subsequence without the current character from x, or the current character from y, and take the maximum.
6. Once the entire array L is filled, the value at L[n][m] will represent the length of the longest common subsequence of strings x and y.
7. Return the value at L[n][m] as the result.

This algorithm has a time complexity of O(mn) because we fill a 2D array of size (n+1) x (m+1) and each cell is filled in constant time based on the values of its neighboring cells.

6. An algorithm to select the minimum number of intervals whose union is the same as the union of all intervals:

1. Sort the intervals based on their start points in increasing order.
2. Initialize an empty list selected_intervals to store the selected intervals.
3. Initialize a variable prev_end with negative infinity.
4. Iterate through the sorted intervals:
    - If the start point of the current interval is greater than or equal to prev_end, add the current interval to selected_intervals and update prev_end with the end point of the current interval.
5. Return the selected_intervals list.

This algorithm ensures that the selected intervals are disjoint and have the minimum number of intervals covering the union of all intervals.

The time complexity of the provided algorithm for selecting the minimum number of intervals whose union is the same as the union of all intervals is O(n log n), where n is the number of intervals.