CS3230: Design and Analysis of Algorithms
Semester 2, 2022-23, School of Computing, NUS

Solutions of Practice Problem Set 4

March 4, 2023

Question 1: In the lecture we have seen dynamic programming algorithm that computes LCS between two input strings. Now suppose you have m input strings each of length n. Can you extend the dynamic programming algorithm covered in the lecture to find a largest subsequence that is common to all the m input strings? (Try to achieve running time of $O(mn^m)$.)

Answer: Recall that in the two input strings version of the LCS with strings A and B, we maintained a $2-Dimensional$ table L, of size n^2, where $L(i,j)$ represents the longest common subsequence between $A[1..i]$ and $B[1..j]$.

Now, we have strings of $A_1, A_2, .., A_m$. Let's try to imagine an $m-Dimensional$ table L, of size n^m, where $L(a_1, a_2, ..., a_m)$ represents the longest common subsequence between $A_1[1..a_1], A_2[1..a_2], .., A_m[1..a_m]$. From this, we can generate a solution that looks very similar to the solution of 2 strings:

$$L(a_1, a_2, ..., a_m) = \begin{cases} \emptyset & \exists j \in [1..m] \mid a_j = 0 \\ A_1[a_1] \cup L(a_1 - 1, a_2 - 1, ..., a_m - 1) & A_1[a_1] = A_2[a_2] = ... = A_m[a_m] \\ max_{j=[1...m]}(L(a_1, a_2, ...a_j - 1, ..a_m)) & \end{cases}$$

Housekeeping:

- Time complexity: We're building a n^m array where each value can be calculated in $\Theta(m)$ by max above. Thus the time complexity is trivially $\Theta(mn^m)$
- Note that the problem has optimal substructure similar to the question in the lecture notes. If the solution to the subproblems were suboptimal (smaller), the size of the resulting set would smaller and thus suboptimal.

Question 2: Given two strings x and y the *edit distance*, denoted by $ed(x,y)$, between them is the minimum number of character insertion, deletion and substitution operations (all of these operations are also known as edit operations) required to transform x into y. Modify the LCS algorithm in such a way so that given two strings x and y of length n and m respectively, your algorithm will output the value of $ed(x,y)$ in $O(mn)$ time. Can you also output the corresponding sequence of edit operations within the same time bound?

Answer:

We are going to employ a dynamic programming algorithm to solve this question. Let's model $ed(x,y)$ as $e(n,m)$, where $e(n,m)$ returns the number of edit distance between $x[1..n]$ and $y[1..m]$. Let's make a few observations about the 2 last characters of x ($x[i]$) and y ($y[j]$) we are examining:

- $e(0,j) = j$ and $e(i,0) = i$ because when x or y is empty, we can quickly see that we must delete all characters in the other string to make them equal
- If $x[i] = y[j]$, then $e(i,j) = e(i-1, j-1)$. This is because the last character for both strings are equal. Edit distance of the 2 strings will be equal to the edit distance of the 2 strings without the last character.
- Else, $x[i] \neq y[j]$, then to make them equal, we can:
  - Insert $y[j]$ after $x[i]$, thus, $e(i,j) = 1 + e(i, j-1)$. (Note that this is equivalent to deleting $y[j]$)

  - Insert $x[i]$ after $y[j]$, thus, $e(i,j) = 1 + e(i-1,j)$. (Note that this is equivalent to deleting $x[j]$)
  - Edit $x[i]$ to be $y[j]$, thus, $e(i,j) = 1 + e(i-1,j-1)$. (Note that this is equivalent to editing $y[j]$ to be $x[i]$)

From the series of observations, let's build the recursive solution:

$$e(n,m) = \begin{cases} m & n = 0 \\ n & m = 0 \\ e(n-1,m-1) & x[n] = y[m] \\ 1 + min(e(n-1,m-1), e(n-1,m), e(n,m-1)) & otherwise \end{cases}$$

Housekeeping:

- Time complexity: We're building a $n * m$ array where each value can be calculated in $\Theta(1)$ by the formula above. Thus the time complexity is trivially $\Theta(nm)$
- Note that the problem has optimal substructure. From the equation shown, we quickly see that the subproblems are $e(n-1,m-1), e(n-1,m), e(n,m-1)$, note that if they are not optimal, it would result in $e^{'}(n,m) \geq e(n,m)$.
- "Can you also output the corresponding sequence of edit operations within the same time bound" Since our algorithm will commit to an edit per value of $e(i,j)$, we can use this to output the corresponding edit operations in the same time bound.

Question 3: Suppose a seller wants to cut a rod into several pieces and then sells them to his/her customers. The main objective of the seller is to maximize his/her earning. Given a rod of length n and a list of prices of rod of length i for all $1 \leq i \leq n$, find an optimal way to cut the rod into smaller pieces in order to maximize the earning. (Try to design a dynamic programming algorithm.)

Answer: Lets define getPrice(n) to return the maximum value of a rod of length n. Given a rod we can break it into different lengths by cutting it at some location. The cut length can vary from 0 to n.

Therefore we can get the maximum price as follows, getPrice(n) = max(price[i] + getPrice(n-i-1)) for i in 0, 1 .. n-1

Looking at this recursive definition we can see the same subproblems will be called again and again, therefore we can use a dynamic programming approach.

```
def getPrice(price, n):
    val = [0 for x in range(n+1)]
    val[0] = 0

    # Build the table val[] in bottom up manner
    for i in range(1, n+1):
        max_val = INT_MIN
        for j in range(i):
            max_val = max(max_val, price[j] + val[i-j-1])
        val[i] = max_val

    # Maximum price will be stored at index n at the end
    return val[n]
```

Question 4: You are given an array of n positive integers with total sum S and a value k. Design a dynamic programming algorithm to find (if exists) k (disjoint) partitions of the input array so that the sum of numbers in each partition is S/k.

Answer:

Intuition: We try fill each partition at a time. We'll start with an empty set, then for each item not added in the chosen set, we'll try to add it into the unfilled partition:

- If the unfilled partition overflows, we'll say that set of values is impossible.
- If the unfilled partition doesn't overflow, we'll store the value of the unfilled partition in an array $dp[set]$, and we continue.

Solution: We will maintain a bit mask to indicate which elements from S are used in partitions and which are not. For example $S = [5, 3, 2, 2]$ and $mask = 0110$ means 3,2 are used in partitions.

We will maintain an array $dp[]$ and $target$ (the target size of a partition S/k). The array dp will give the output for each mask as the sum of all values in the mask modulo target, ie. $dp[\text{"0110"}] = (3 + 2)\%target = 0$. We want to find a partitioning which uses all elements in S which means the question is to find if we can build $dp[\text{"111...1"}] = 0$.

In this representation if we want to transition from mask by adding element j, it is achieved by bit wise or operation as follows, $dp[i|(1 << j)] = (dp[i] + arr[j])\%target$. Note that this can only be done if $dp[i] + arr[j] \leq target$.

We can first check for the following conditions,

if k==1: make one partition of all elements in S, return True.

if k > size of S: cannot make more paritions than elements, return False.

if (Sum of S)%k $\neq$ 0: cannot make integer sum partitions, return False.

Next we can update the dp array as follows,

initialize all dp values with -1,

```
dp[0] = 0
target = S/k
# Iterate over all subsets/masks
for mask in range((1 << N)):

    # If current mask is invalid,
    # continue
    if (dp[mask] == -1):
        continue

    # Iterate over all array elements
    for i in range(N):

        # Check if the current element
        # can be added to the current subset/mask
        #     - mask & (1 << i) == 0 ensures that the bit at i = 0 in the mask
        #     - dp[mask] + arr[i] <= target ensures that the value can be added
        #       into the unfilled partition
        if ((mask & (1 << i) == 0) and dp[mask] + arr[i] <= target):
            # Transition
            dp[mask | (1 << i)] = ((dp[mask] + arr[i]) % target)

if (dp[(1 << N) - 1] == 0):
    return True
else:
    return False
```

Question 5: In the lecture we have already discussed the problem of changing \$$n$ with smaller denomination coins, and designed a dynamic programming algorithm. Now consider the following greedy strategy for the same problem with coin denominations $\{Q, D, N, P\}$: suppose the amount left to change is n; take the largest coin that is no more than n; subtract this coin's value from n, and repeat.

Either give a counterexample, to prove that this algorithm can output a non-optimal solution, or prove that this algorithm always outputs an optimal solution.

Answer:

We prove that the greedy algorithm yields an optimal when the coin denominations are standard {Q, D, N, P}. It is easily seen that the greedy algorithm can be implemented in O(n) time.

Greedy Choice Property: We show that there exists an optimal solution for making n cents that begins with the largest coin whose value is $\leq n$ cents. Let S be the set of coins given as change by some optimal solution to the problem of making change for n cents. Suppose that the largest denomination among the coins in S is $k \in \{Q, D, N, P\}$.

Let i be the first coin chosen by the greedy algorithm. If $k = i$ then the solution begins with a greedy choice. If $k \neq i$ then we must show that there is another optimal solution S' that begins with coin i. Observe that since the greedy algorithm picks the largest denomination that can be used, it follows that $k < i$. Thus $i \neq P$. We now consider each of the remaining cases.

- **Case 1:** $i = N$. Thus the solution S must have value of 5 cents or more. Since the largest coin in S must be less than a nickel, S uses only pennies, and hence must have at least 5 pennies. Create S' from S by replacing 5 pennies by a nickel. $|S'| < |S|$ contradicting the optimality of S.

- **Case 2:** $i = D$. Thus the solution S must have value of 10 cents or more. Since the largest coin in S must be less than a dime, S uses only pennies and nickels. The only way to create 10 cents or more with pennies and nickels must use either 2 nickels, 1 nickel and 5 pennies, or 10 pennies. In all three cases we can create S' from S by replacing this subset of coins that equals 10 cents by a dime. Again, $|S'| < |S|$ contradicting the optimality of S.

- **Case 3:** $i = Q$. First suppose, that there is some combination of coins in S that sum to 25 cents. Then let $S' = S - \{\text{coins that sum to } 25\} \cup \{Q\}$ and observe $|S'| < |S|$.

  Next we consider the case when no subset of coins in S sums to 25. This can only occur if there are at least 3 dimes in S since that is required to obtain 30 cents or more without using a nickel or at least 5 pennies (in which case some subset of coins would add to 25 cents). So in this case let $S' = S - \{D + D + D\} \cup \{Q + N\}$.Again $|S'| < |S|$ contradicting the optimality of S.

Hence in all three cases we proved not only that there exist some optimal solution that picks the same first coin as the greedy algorithm, but in fact, every optimal solution must pick this largest coin.

Optimal Substructure Property: Let P be the original problem of making n. Suppose the greedy solution starts with k cent coin. Then we have the subproblem P' of making change for $n - k$ cents. Let S be an optimal solution to P and let S' be an optimal solution to P'. Then clearly $cost(S) = cost(S') + 1$, and thus the optimal substructure property clearly follows.

Question 6: You are given n events where each takes one unit of time. Event i will provide a profit of g_i dollars ($g_i > 0$) if started at or before time t_i where t_i is an arbitrary real number. (Note: If an event is not started by t_i then there is no benefit in scheduling it at all. All events can start as early as time 0.) Also only one event can be scheduled at any particular time interval.

Give as efficient algorithm as you can, to find a schedule that maximizes the total profit.

Answer:

We first argue that there always exists an optimal solution in which all of the events start at integral times. Take an optimal solution S, you can always have the first job in S start at time 0, the second start at time 1, and so on. Hence, in any optimal solution, event i will start at or before time $\lfloor t_i \rfloor$.

This observation leads to the following greedy algorithm. First, we sort the jobs according to $\lfloor t_i \rfloor$ (sorted from largest to smallest). Let time t be the current time being considered (where initially $t = \lfloor t_i \rfloor$). All jobs i where $\lfloor t_i \rfloor = t$ are inserted into a priority queue with the profit g_i used as the key. An extractMax is performed to select the job to run at time t. Then t is decremented and the process is continued.

Clearly the time complexity is $O(nlogn)$. The sort takes $O(nlogn)$ and there are at most n insert and extractMax operations performed on the priority queue, each which takes $O(logn)$ time.

We now prove that this algorithm is correct by showing that the greedy choice and optimal substructure properties hold.

Greedy Choice Property: Consider an optimal solution S in which $x + 1$ events are scheduled at times $0, 1, ..., x$. Let event k be the last job run in S. The greedy schedule will run event 1 last (at time $\lfloor t_1 \rfloor$). From the greedy choice property we know that $\lfloor t_1 \rfloor \geq \lfloor t_k \rfloor$.

We consider the following cases:

- Case 1: $\lfloor t_1 \rfloor = \lfloor t_k \rfloor$. By our greedy choice, we know that $g_1 \geq g_k$. If event 1 is not in S then we can just replace event k by event 1. The resulting solution S' is at least as good as S since $g_1 \geq g_k$. The other possibility is that event 1 is in S at an earlier time. Since $\lfloor t_1 \rfloor = \lfloor t_k \rfloor$, we can switch the times in which they run to create a schedule S' which has the same profit as S and is hence optimal.
- Case 2: $\lfloor t_1 \rfloor < \lfloor t_k \rfloor$. In this case, S does not run any event at time $\lfloor t_1 \rfloor$ since job k was its last job. If event 1 is not in S, then we could add it to S contradicting the optimality of S. If event 1 is in S we can run it instead at time $\lfloor t_1 \rfloor$ creating a schedule S' that makes the greedy choice and has the same profit as S and is hence also optimal.

Optimal Substructure Property: Let P be the original problem of scheduling events $1, ..., n$ with an optimal solution S. Given that event 1 is scheduled first we are left with the sub problem P' of scheduling events $2, ..., n$. Let S' be an optimal solution to P'. Clearly $profit(S) = profit(S') + g_1$ and hence an optimal solution for P includes within it an optimal solution to P'.

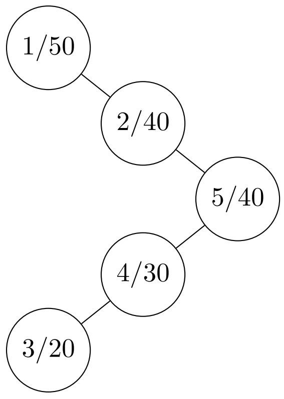
Question 7: Given a set of keys $1, 2, \cdots, n$, where the key i has weight w_i. The weight of the key reflects how often the key is accessed, and thus heavy keys should be higher in the tree. The Optimal Binary Search Tree problem is to construct a binary-search tree for these keys, in such a way that

$$wac(T) = \sum_{i=1}^{n} w_i d_i$$

is minimized, where d_i is the depth of key i in the tree (note: here we assume the root has a depth equal to one). This sum is called the *weighted access cost (wac)*. Consider the greedy heuristic for Optimal Binary Search Tree: for keys $1, 2, \cdots, n$, choose as root the node having the maximum weight. Then repeat this for both the resulting left and right subtrees.

Apply this heuristic to keys $1, 2, 3, 4, 5$ with respective weights $50, 40, 20, 30, 40$. Show that the resulting tree does not yield the minimum weighted access cost.

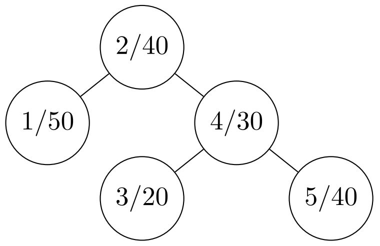
Answer:



The Binary Search Tree resulting from the greedy heuristic is given in the above figure. According to this, the total *weighted access cost(wac)* is as follows,

$$50(1) + 40(2) + 40(3) + 30(4) + 20(5) = 470$$

This is not the optimal, the following Binary Search Tree configuration will give a smaller wac of 380.



$$40(1) + 50(1) + 30(2) + 20(3) + 40(3) = 380$$

Question 8: Given a set $\{x_1 \leq x_2 \leq \cdots \leq x_n\}$ of points on the real line, determine the smallest set of unit-length closed intervals that contains all of the points x_i.

Can you design an $O(n)$ time greedy algorithm for the above problem?

Answer:

The greedy algorithm we use is to place the first interval at $[x_1, x_1+1]$, remove all points in $[x_1, x_1+1]$ and then repeat this process on the remaining points.

Clearly the above is an $O(n)$ algorithm. We now prove it is correct.

Greedy Choice Property: Let S be an optimal solution. Suppose S places its leftmost interval at $[x, x+1]$. By definition of our greedy choice $x \leq x_1$ since it puts the first point as far right as possible while still covering x_1. Let S' be the set obtained by starting with S and replacing $[x, x+1]$ with $[x_1, x_1+1]$. We now argue that all points contained in $[x, x+1]$ are covered by $[x_1, x_1+1]$. The region covered by $[x, x+1]$ which is not covered by $[x_1, x_1+1]$ is $[x, x1)$ which is the points from x up until x_1 (but not including x_1). However, since x_1 is the leftmost point there are no points in this region. (There could be additional points covered by $[x+1, x_1+1]$ that are not covered in $[x, x+1]$ but that does not affect the validity of S'). Hence S' is a valid solution with the same number of points as S and hence S' is an optimal solution.

Optimal Substructure Property: Let P be the original problem with an optimal solution S. After including the interval $[x_1, x_1+1]$, the sub problem P' is to find an solution for covering the points to the right of x_1+1. Let S' be an optimal solution to P'. Since, $cost(S) = cost(S')+1$, clearly an optimal solution to P includes within it an optimal solution to P'.