SHANGHAI JIAOTONG UNIVERSITIY

Assignment 4 Suggested Answer

Course: *AI2615 - Algorithm Design and Analysis*
Due date: *Jan, 2022*

Problem 1

Given two strings $A = a_1a_2a_3a_4 \dots a_n$ and $B = b_1b_2b_3b_4 \dots b_n$, how to find the longest common subsequence between A and B? In particular, we want to determine the largest k, where we can find two list of indices $i_1 < i_2 < i_3 < \dots < i_k$ and $j_1 < j_2 < j_3 < \dots < j_k$ with $a_{i_1}a_{i_2} \dots a_{i_k} = b_{j_1}b_{j_2} \dots b_{j_k}$. Design a DP algorithm for this task.

Answer referred by Haoxuan Xu . We use a 2d array, $LLST[n+1][n+1]$ to implement the dynamic programming, $LLST[i][j]$ means the longest common subsequence length of $A[0:i], B[0:j]$ (note that the index range grammer is same as python, i.e. "hello"[0:2] = "he")

First, if $i = 0$ or $j = 0$, then one string is empty, the LLST(length of longest common subsequence) is obviously 0. Then, for the ith character of A a_i, jth character of B B_j ($i \neq 0, j \neq 0$), there are either same or different. If they are same, they can be added to the common subsequence, the new longest common subsequence length will be $LLST[i][j] + 1$; if they are different, then the new longest common subsequence length will be $max(LLST[i-1][j], LLST[i][j-1])$. And we can see from the state transition equation that the solution of $LLST[i][j]$ need to solve $LLST[i-1][j-1], LLST[i-1][j], LLST[i][j-1]$ in advance. So we only need to solve the subproblems from $i = 0$ to n, and from $j = 0$ to n. The longest common subsequence length of A, B will be $LLST[n][n]$.

If we want to get the corresponding longest common subsequence, we can use an extra 2d array to maintain it, $LST[n+1][n+1]$, $LST[i][j]$ means the longest common subsequence of $A[0:i], B[0:j]$, and continusously update it.

Time Complexity. There are totally $(n+1)^2$ states and the update for each state takes $O(1)$. So the total time complexity is $O(n^2)$.

Correctness.

(Base case) $i = 0$ or $j = 0$, then one string is empty, the longest common subsequence length is obviously 0.

(Induction) Our inductive hypothesis is that $LLST[i][j]$ correctly represents the longest common subsequence length of $A[0:i]$ and $B[0:j]$ for $i = p-1, j = q-1$, $i = p-1, j = q$ and $i = p, j = q-1$. Then, suppose we are finding $LLST[p][q]$, if $A[p-1] = B[q-1]$, obviously they can match up, then its corresponding length is $LLST[p-1][q-1] + 1$, but we can also choose not to match them up, and try to find the LST of $A[0:p]$ and $B[0:q-1]$ or $A[0:p-1]$ and $B[0:q]$, but obviously the longest common subsequence lenght of these two pairs of strings if smaller than $A[0:p]$ and $B[0:q]$, as we have one more additional character and one more choice. So, $LLST[p][q] = LLST[p-1][q-1] + 1$. And if $A[p-1] \neq B[q-1]$, they can't match

Algorithm 1 LongestCommonSubsequenceLength Algorithm

Input: string A with size n, string B with size n

Initialize $LLST[n][n]$ with 0, initialized $LST[n][n]$ with "".
for $i \leftarrow 0$ to n **do**
  for $j \leftarrow 0$ to n **do**
    if $i = 0$ or $j = 0$ **then**
      continue
    end if
    if A[i-1] = B[j-1] **then**
      $LLST[i][j] \leftarrow LLST[i-1][j-1] + 1$
      $LST[i][j] \leftarrow LST[i-1][j-1] + A[i-1]$
    else
      $LLST[i][j] \leftarrow \max\{LLST[i-1][j], LLST[i][j-1]\}$
      $LST[i][j] \leftarrow$ corresponding $LST[i-1][j]/LST[i][j-1]$
    end if
  end for
end for
return $LLST[n][n], LST[n][n]$

up, we have to throw one of them and try to find the LST of $A[0:p]$ and $B[0:q-1]$ or $A[0:p-1]$ and $B[0:q]$, and get the maximum of them, its correctness is obvious. And the correctness of maintaining LST is obvious by the correctness of $LLST$.

(Computation Order) We solve subproblems from $i = 0$ to n, $j = 0$ to n, which guarantees that when we solve $LLST[i][j]$, $LLST[i-1][j-1]$, $LLST[i-1][j]$, $LLST[i][j-1]$ have been solved.

Problem 2

Given two teams A and B, who has already played $i+j$ games, where A won i games and B won j games. We suppose that they both have 0.5 independent probability to win the upcoming games. The team that first win $n \geq \max\{i,j\}$ games will be the final winner. Design a DP algorithm to calculate the probability that A will be the winner.

Answer referred by Yichen Tao. Define the subproblem $p_A(x,y)(i \leq x \leq n, j \leq y \leq n, x+y \neq 2n)$ as the probability that A is the final winner, provided that A has won x games and B has won y games. Three cases are considered here:

- $x = n \wedge y < n$. Since A has already won, $p_A(x,y) = 1$.
- $x < n \wedge y = n$. Here B has already won n games, so $p_A(x,y) = 0$.
- $x < n \wedge y < n$. In this case, at least one more game will be played. The possibilty that A win or lose the next game is both $\frac{1}{2}$. Hence, according to Law to Total Probability, we should have $p_A(x,y) = \frac{1}{2}p_A(x+1,y) + \frac{1}{2}p_A(x,y+1)$.

By the analysis above, we have the following recurrence relation:

$$p_A(x,y) = \begin{cases} 1, & x = n \wedge y < n \\ 0, & x < n \wedge y = n \\ \frac{1}{2}p_A(x+1,y) + \frac{1}{2}p_A(x,y+1), & x < n \wedge y < n \end{cases} \tag{1}$$

By calculating the values of $p_A(x, y)$ in a certain order (which will be shown below), the value of $p_A(i, j)$ is the answer to the problem.
The formalization of the algorithm is shown in 2:

Algorithm 2 Calculating the probability that A be the final winner.

Require: Number of games A has already won: i. Number of games B has already won: j. Number of winnings needed to be the final winner: n.
Ensure: The probability that A be the final winner.

```
for x := n to i do
  for y := n to j do
    if x = n then
      p_A(x, y) ← 1
    else if y = n then
      p_A(x, y) ← 0
    else
      p_A(x, y) ← (1/2) p_A(x + 1, y) + (1/2) p_A(x, y + 1)
    end if
  end for
end for
return p_A(i, j)
```

Proof of Correctness. The correctness of the algorithm can be proved by induction.
Base case. The value of $p_A(n, n)$ does not matter the calculation after that since it will not be involved in any further calculation. The reason that $p_A(n, y) = 1$ and $p_A(x, n) = 0$ is obvious and has already been explained above.
Induction Hypothesis. Suppose that the calculation of $p_A(x + 1, y)$ and $p_A(x, y + 1)$ (when $x, y < n$) are both correct.
Induction. Denote the event "A wins the $x + y + 1$-th game" by E; the event "A is the final winner" by F. Apparently, we should have:
$P(E) = P(\overline{E}) = \frac{1}{2}$
$P(F|E) = p_A(x + 1, y); P(F|\overline{E}) = p_A(x, y + 1)$
Therefore, $p_A(x, y) = P(F) = P(E)P(F|E) + P(\overline{E})P(F|\overline{E}) = \frac{1}{2}p_A(x + 1, y) + \frac{1}{2}p_A(x, y + 1)$, which is consistent with what we do in the algorithm.
The values of $p_A(x, y)$ is calculated in the following order: $p_A(n, n), p_A(n, n - 1), \ldots, p_A(n, j)$; $p_A(n - 1, n), \ldots, p_A(n - 1, j); \ldots; p_A(i, n), \ldots, p_A(i, j)$. So when $p_A(x, y)$ is being calculated, the two possibly needed term $p_A(x, y + 1)$ and $p_A(x + 1, y)$ should have been calculated already.

Time Complexity. Calculating each entry of p_A needs $O(1)$, and $(n - i)(n - j)$ entries are calculated. So the overall time complexity is $O((n - i)(n - j))$.

Problem 3

Formalize the improved DP algorithm for the Longest Increasing Subsequence Problem in the lecture, prove its correctness, and analyze its time complexity.

Answer referred by Haoxuan Xu. The description of the improved algorithm is explained in the lecture, so I directly give the formal algorithm.

Algorithm 3 LIS Improved Algorithm

Input: A sequence of number a with size n.

$sm \leftarrow$ '-' with size n, $maxLen \leftarrow 0$
for $i \leftarrow 0$ to $n-1$ **do**
  if $maxLen = 0$ or $a[i] > sm[maxLen-1]$ **then**
    $maxLen \leftarrow maxLen + 1$, $sm[maxLen] \leftarrow a[i]$
  else
    Binary search $sm[0 : maxLen]$ to get the largest j such that $sm[j] < a[i]$
    $sm[j+1] \leftarrow a[i]$
  end if
end for
return $maxLen$

Time Complexity. We totally do n iterations, and in each iteration, the binary search costs $O(logn)$ time, and if we append the number to the end of sm, the time cost is $O(1)$, hence, the time cost for each iteration is $O(logn)$. In conclusion, the time complexity is $O(nlogn)$.

Correctness. Here, we define the subproblems as maintain sm correctly, which means $sm[len]$ is the smallest ended number for an increasing subsequence with length len during iterations.

(Base case) The base case is that the orgin sm is empty, so when adding the first number to it, it must be the smallest ended number for an increasing subsequence with length 1, as no other number can be chosen, its correctness is obvious.

(Induction) The inductive hypothesis is that $sm[len]$ is the smallest ended number for an increasing subsequence with length len for $len < maxLen$ by using $a_1 ... a_{i-1}$.

Considering a_i, there are two cases. First, $a_i > sm[maxLen]$, we can just add $maxLen$ and append $a[i]$ to the end, which is still increasing, whose correctness is obvious. In case 2, we find the the largest number $sm[j]$ which is smaller than $a[i]$, then obviously, $sm[0 : j]$ will not be updated as they are all smaller than $a[i]$ and sm is the smallest ended number, and we update $sm[j+1]$ to $a[i]$, because we can form a new increasing subsequence with length $j+1$ by adding $a[i]$ to the previous incresing subsequence with length j. Besides, we cannot change $sm[j+2]$ or later. Take $sm[j+2]$ for example, as $a[i] <= sm[j+1]$, it's smaller than the number before ended number with length $j+2$, so we cannot change the ended number. So, the correctness of sm is guaranteed in iterations.

Hence, we have proven that the correctness of sm is maintained during iterations by induction.

(Computation Order) The order in which the subproblems are solved is from $i = 0$ to n, so we can traverse from $i = 0$ to n to solve them.

Problem 4

Optimal Indexing for A Dictionary. Consider a dictionary with n different words $a_1, a_2, \ldots, a_n$ sorted by the alphabetical order. We have already known the number of search times of each word a_i, which is represented by w_i. Suppose that the dictionary stores all words in a binary search tree T , *i.e.* each node's word is alphabetically larger than the words stored in its left subtree and smaller than the words stored in its right subtree. Then, to look up a word in the dictionary, we have to do $\ell_i(T)$ comparisons on the binary search tree, where $\ell_i(T)$ is exactly the level of the node that stores a_i (root has level 1). We evaluate the search tree by the total number of comparisons for searching the n words, *i.e.*, $\sum_{i=1}^{n} w_i \ell_i(T)$. Design a DP algorithm to find the best binary search tree for the n words to minimize the total number of comparisons.

Answer referred by Yichen Tao. First, the notations involved are stated. $cost[i][j]$ is the minimum cost of the binary search tree (BST) including words $a_i, a_{i+1}, \ldots, a_j$. $root[i][j]$ is the root of the above BST. $sum[i][j]$ is the sum of weight of words $a_i, a_{i+1}, \ldots, a_j$, *i.e.* $sum[i][j] = \sum_{x=i}^{j} w_x$.
While calculating the cost of BST involving $a_i, \ldots, a_j$, if we choose a_r ($i \leq r \leq j$) as the root node, $a_i, \ldots a_{r-1}$ will be on the left subtree, and $a_{r+1}, \ldots, a_j$ on the right subtree. Suppose that $cost[i][r-1]$ and $cost[r+1][j]$ has already been calculated, all nodes except a_r should have their levels increased by 1 in the new BST. Hence, the cost of the BST with root a_r is $sum[i][j] + cost[i][r-1] + cost[r+1][j]$. The remaining problem is finding the r that minimizes the cost, and we address it by traversing all possbile values of r.
By the analysis above, we have the following recurrence relation:

$$cost[i][j] = \begin{cases} w_i, & i = j \\ sum[i][j] + \min_r\{cost[i][r-1] + cost[r+1][j]\}, & i < j \end{cases} \tag{2}$$

$$root[i][j] = \begin{cases} i, & i = j \\ \arg\min_r\{cost[i][r-1] + cost[r+1][j]\}, & i < j \end{cases} \tag{3}$$

After that, the value of $cost[1][n]$ is the optimal cost of the BST, and the BST can be formed with $root[i][j]$. 4 shows the formalized algorithm:

Proof of correctness. We prove the correctness by induction.
Base case. When $i = j$, there is only one node a_i in the BST. So the cost is w_i, and only a_i can be the root.
Induction hypothesis. The calculation of all $cost[x][y]$ with $y - x < j - i$ is correct.
Induction. Consider calculating $cost[i][j]$ and $root[i][j]$. Since we traverse all possible roots and choose the one giving the least cost as $root[i][j]$, it should give the optimal BST including nodes $a_i, \ldots, a_j$.
By the analysis above, all values of $cost[i][j]$ and $root[i][j]$ are correct, so the value of $cost[1][n]$ is the cost of the optimal BST, and the tree formed using $root$ is also the optimal binary search tree.

Algorithm 4 Finding the optimal binary search tree.

Require: n different words a_i with weight w_i.
Ensure: The binary search tree.
  Calculate all values of $sum[i][j]$.
  $cost[i][i] \leftarrow w_i$; $root[i][i] \leftarrow i$ for all i.
  for $l := 2$**to** n **do**
    for $i := 1$**to** $n - l + 1$ **do**
      $j \leftarrow i + l - 1$
      $sum[i][j] \leftarrow sum[i][j] + \min_r\{cost[i][r-1] + cost[r+1][j]\}$
      $root[i][j] \leftarrow \arg\min_r\{cost[i][r-1] + cost[r+1][j]\}$
    end for
  end for
  Form the BST recursively with $root$.
  return the BST formed.

Time complexity. If we calculate $sum[i][j]$ following the idea of DP (which is rather simple and will not be dilated upon here), the initialization takes $O(n^2)$. For the calculation part, all $1 \leq i \leq j \leq n$ cases are considered, so there are $O(n^2)$ cases in all. Each case takes $O(n)$ as we need to traverse all possible r. So the overall time complexity is $O(n^3)$.

Problem 5

Precedence Constrained Unit Size Knapsack Problem. Let us recap the classic knapsack problem (unit size): We are given a knapsack of capacity W and n items, each item i has the same size $s_i = 1$ and is associated with its value v_i. The goal is to find a set of items S (each item can be selected at most once) with total size at most W (*i.e.*, $\sum_{i \in S} s_i = |S| \leq W$) to maximize the total value (*i.e.*, $\sum_{i \in S} v_i$). Now we are given some additional precedence constraints among items, for example, we may have that item i can only be selected if we have selected item j. These constraints are presented by a tree $G = (V, E)$, which says that a vertex can only be selected if we have selected its parent, (*i.e.*, we need to select all his ancestors). The task is also to maximize the total value we can get, but not violating the capacity constraint and any precedence constraints.

(*a*) Design a DP algorithm for it in $O(nW^2)$ time.

(*b*) Can you improve the algorithm and analysis to make it run in $O(n^2)$ time? (When W is sufficiently large like $W = \Theta(n)$, it is an improvement.)

Answer.

(*a*) referred by Xiangge Huang

Using DP algorithm, we define $F(i,j)$ is each situation to choose items. Due to the dependence between parent and children in trees, we let i represents the relationship between each item and j represents the volume. Firstly we consider the root, which must be chosen with size 1, so we should choose the biggest value in the left tree and let it add the value of the root, and we do the same thing in the following subtrees. So we have the formula $F(i,j) = \max_{0\le k\le j}(F(i,j), F(i,j-k) + F(t,k))$. And the basic step is when i is a leaf, $F(i,1) = v_i$, because each item size is 1, to make value bigger we must have only left 1 when we reach the leaf, and when $j = 0$, that is the bag is full, we have $F(i,0) = 0$. We describe our algorithm as below.

```
//precedence bag
construct f[num_of_nodes][W]
void bag(node u)
    f[u,0]=0, f[u,1]=v[u]
    for node i go through all u sons and i is not null:
        bag(i)
        for j from W to 0:
            f(i,j) = max_{0≤k≤j}(f(i,j), f(i,j-k) + f(t,k))
int main()
    bag(root)
    return f[root][W]
```

We can analyze the whole process of this algorithm, due to the precedence relationship, it is natural for us to solve this problem from the top to the bottom of this tree. And for a special node u, if our bag does not have any space, we can't add it, so $f(u,0) = 0$, and if our bag only has 1 place, we add it directly, so $f(u,1) = 1$, and this always happens when u is a leaf. And when we arrive at a node, we know its statue is decided by its sons, so we use recursion to solve its sons initially, and when we go back from sons to parent, we will choose the best subtrees through our formula. So we can get the maximal value in root finally, that is $f(root, W)$. And because the parent only focus on its son, this must be a DAG.

Obviously for all recursion part, we should go through the whole tree, so it costs $O(n)$, and in combination process, we should loop j from W to 0 and k from 0 to j separately, and it costs $O(W^2)$, so the total time complexity is $O(nW^2)$.

(*b*) referred by Xiangge Huang

We can simplify this algorithm by changing the way we explore the tree. Because we calculate all subtrees of a parent, which leads to a big cost, we optimize our algorithm is this part. We can let $G(u,j)$ means the whole forest containing u, u sons and u brothers, so the path that we loop the tree can be shorten. And the formula can be

$$G(i,j) = \max\left(G(b,j), G(i,j), \max_{x+y=j-1}(G(s,x) + G(b,y) + v_i)\right)$$

The b means the brothers of i and the s means the sons of u. Because we link a node with all its brothers and sons, obviously that each subpart of this tree is equal, and we can choose the best part among its sons and brothers as the final value. Also, x and y means the left bag size, which should be less than or equal to current subtree.

We describe our algorithm as below.

```
//improved bag
construct g[num_of_nodes][W]
void bag(node u)
    g[u,0]=0, g[u,1]=v[u]
    b=brother[u], s=son[u]
    if s is not null: bag(s)      //size[s]^2
    if b is not null: bag(b)      //size[b]^2
    size[u]=size[s]+size[b]+1
    for k from 1 to W: g[u][k]=g[b][k] //choose best brother
    for x from 0 to size[s]:    //choose both son and brother
```

for y from 0 to size[b]: //size[s]size[b]

j=x+y+1

$g(u,j) = max\ (g(u,j), g(s,x) + g(b,y) + v[u])$

int main()

bag(root)

return g[root][W]

We analyze the time complexity first. Also, we also should go through all node, but in each loop, we know the complexity is $(size[s])^2 + (size[b])^2 + (size[s]size[b])$, which is related to $bag(s)$, $bag(b)$ and the next x and y part. We know $(size[s])^2 + (size[b])^2 + (size[s]size[b]) \leq (size[s] + size[b] + 1)^2 = (size[u])^2$. Consider we start from the root, so the time complexity will not be larger than n^2. So the total time complexity is $O(n^2)$.

We reconstruct our algorithm to make it simpler.

//new improved bag

construct g[num_of_nodes][W]

void bag(node u)

g[u,0]=0, f[u,1]=v[u]

size[u]=1

for node i go through all u sons and i is not null:

bag(i)

```
        size[u]+=size[i]
        for j from size[u] to 1:
            for t from j-1 to 0:
                g(u,j) = max (g(u,j), g(u,j-t) + g(i,t))
int main()
    bag(root)
    return g[root][W]
```

Similarly, this algorithm is still $O(n^2)$ based on the former analysis. And it still builds from the bottom of the tree, but due to the truth in each layer we consider its sons and brothers together, we only run the corresponding size in each loop and recursion. And in each transformation part, the node with its brothers situates in the equal situation and is only decided by its sons. And we always choose the maximal value in each choice to fill the bag, so we can get the best option and value finally, and this is also a DAG.