Homework 14

Brian Knotten, Brett Schreiber, Brian Falkenstein

September 28, 2018

19

a

A recursive algorithm for this problem is as follows: In short, keep track of the current indices of T and P to find the optimal solution of either taking one from T to match with P or not. After exhausting all possible options to take, return the choices that yield the maximum cost.

```
T[n]
C[n]
P[k]

A(i, j):
    if j == 0 or i == 0: return 0 # No more opportunities

    if T[i] == P[j]:
        return max(
            A(i - 1, j - 1) + C[i], # Either take the opportunity and use T[i] for P[j], therefore moving t
            A(i - 1, j) # Or, skip this T[i], and look for a better one to fill P[j]
        )

    else:
        return A(i - 1, j)


Output A(n, k)
```

To convert this into a dynamic programming algorithm, use a 2-dimensional array to keep track of the indices i and j where $A[i][j]$ means: The maximum cost achievable considering the first i items of T and the first j items of P.

Filling this array A requires time $O(nk)$, where n is the number of items in T and k is the number of items in P. Since $k \leq n$, the time complexity is therefore $O(n^2)$ in the worst case.

Here is the code for the dynamic solution:

```
# Base case: with none of T or P to consider, the only available cost is 0.
A[0][0] = 0

for i = 1 to n do:
    for j = 1 to k do:

if T[i] == P[j]:
    A[i][j] = max(
                A[i - 1][j - 1] + C[i],
                A[i - 1][j]
            )
        else:
```

```
            A[i][j] = A[i - 1][j]


Output A[n][k]
```

b

You can alternatively enumerate all the subsequences of T as a binary tree, where, given a vertex at depth i, the left child represents omitting $T[i+1]$ and the right child represents appending $T[i+1]$ with cost $C[i+1]$ into the solution set. Since it is a binary tree, it has 2^n leaves representing solutions, so the following pruning rules must be applied.

1. For any node v, if the sequence in v is not a prefix of P, (for example, $v = XYY$ and $P = XYXX$, then prune v. This also applies to any v larger than P, since a string a cannot be a prefix of b if $|a| > |b|$. v cannot possibly be a solution since the children of v only append to the end of v's sequence, and cannot therefore rectify any differences between v's sequence and P. This limits the depth of the tree from n to k, since any sequence larger than k cannot be a prefix of P since $|P| = k$.

2. For any two nodes u and v at the same level where the string u = string v, if the cost of u is greater than or equal to the cost of v, then prune the tree rooted at v. Any item v can append in the future, u can also append and still have a greater or equal total cost. So v's tree does not contain the solution.

Applying both pruning rules leaves at most i nodes at depth i, rather than 2^i nodes, since there can only be one node representing a substring of size j where $0 \leq j \leq i$, since there is only one valid prefix of length j and only one prefix that has the highest cost. So there are i nodes at depth i: the maximum cost prefix of length 0, the maxmimum cost prefix of length 1, ... the maximum cost prefix of length $i-1$, and the maximum cost prefix of length i. This is i strings in total at depth i.

Since the tree is of depth k and breadth at most n, we can make a polynomial algorithm using a 2d array $A[k][n]$, where $A[i][j]$ represents the maximum aggregate cost of the prefix of the first i elements of P of length j. Here is the dynamic algorithm:

```
for i = 1 to k do: # For each level of the tree
    for j = 1 to i do: # For each node at depth i (containing a prefix of size j)
        if T[i + 1] == P[i + 1]: # Only consider adding a new element if it matches the prefix
            A[i + 1][j + 1] = max(
                A[i + 1][j + 1] # Compete with what's currently in the cell
                A[i][j] + C[i + 1]
            )

Output A[k][n]
```

c

Consider a tree of height k where each node has n children. At depth i, we are considering the jth child where $1 \leq j \leq n$ such that the child represents appending $T[j]$ to the solution, (that is, matching $P[i+1]$ with $T[j]$). The solution would be at the leaves, where a certain node will have matched P from 0 to k. But the tree has a branching factor of n, so it needs to be pruned down from n^k leaves. Here are the pruning rules:

1. Given a node v at depth i, prune v if the last character in the sequence of v is not equal to $P[i]$. This means that the matching is not correct, and it cannot be corrected by adding more onto it. So the solution will not be in the tree rooted at v.

2. Given a node u at depth i, if u is the jth child of his parent node v, prune the $0..j$th child of u. These children are adding characters which are not in increasing order, so they cannot be a part of the solution. So prune these children of u.

22

Pruning rule:

1. If two nodes at the same depth have purchased the same amount of cards, prune the one that has the higher cost.

This pruning rule alone is enough to bring the size of the tree down to $n \times n$. The height of the tree will be n, as at each level i, we decide whether to purchase a card at d_i, and there are n dates in D. Further, at each level i, there will be at max i purchased cards, and we will only have one node corresponding to each possible number of cards purchased.
Thus, the tree can be constructed with a table of size $n \times n$.

$A[i][j]$ = the lowest cost of the first j trips having had the opportunity to purchase i Bahncards. Output the minimum value over the last column (considered all n trips).

```
A[0][0] = 0 # With no tickets and no trips, the total cost is 0

# Fill in the first row
for j = 1 to n do:
    A[0][i] = f[i] + A[0][j - 1] # The sum of all the trips (without any discounts) is the sum of the previ

for j = 1 to n do: # Going down each row of the tree for each trip j
    for i = j to n do: # Choosing to purchase i tickets at each level
        A[i + 1][j + 1] = A[i][j] + f[i] + min(
    0, # Not purchasing a ticket
    A[i][j] + B + f[i] - (sum of the costs of trips from d[j] to d[j] + L) / 2 # Choosing to buy a ticket f
)


Output A[n][n] # Output the cost of taking n trips, having had the opportunity to purchase n Bahncards
```

The result will be in the bottom right, since it will have had n opportunities to purchase Bahncards, but the algorithm can choose not to purchase the ith Bahncard. To retrieve the dates of purchase, keep track of the dates when a Bahncard is worth it to purchase (this can be tracked in a boolean 1-dimensional array where each index i indicates purchasing a ticket at date d_i).