Sample Solution for Problem Set 12

Data Structures and Algorithms, Fall 2020

December 29, 2020

Contents

1 Problem 1

Algorithm Overview: Use dynamic programming. Let $dp[i][j] = True$ means wee can parenthesize $S[i....j]$ to make the value be $True$. Then we have the transition function:

$$dp[i][j] = \bigvee_{i \leq k \leq j, S[k] \in \{\wedge, \vee, \oplus\}} dp[i][k-1]S[k]dp[k+1][j] \tag{1}$$

Implementation: We use a simple down-top dynamic programming. Initialize $dp[i][i] = S[i]$ for all even i. Then do the following loop

- For $k = 2, 4, 6, 8, ...$ until $k = 2n$, do
  - For $i = 0, 2, 4, 6, ...$ until $i + k = 2n$, let $j = i + k$, do
    - $dp[i][j] = \bigvee_{i \leq k \leq j, S[k] \in \{\wedge, \vee, \oplus\}} dp[i][k-1]S[k]dp[k+1][j]$

Complexity: $O(n^3)$. **Recall that the complexity of dynamic programming is the number of states ($O(n^2)$) times the complexity for transition function ($O(n)$).**

2 Problem 2

Let $f_{max}(i)$ represent the largest product in the subarrays ending with i, and $f_{min}(i)$ represent the smallest product in the subarrays ending with i.

Algorithm 1: LARGESTPRODUCT(A, i, j)

```
f_max = f_min = ans = 1;
for i = 1 to n do
    temp = f_max;
    f_max = max(A[i], f_max * A[i], f_min * A[i]);
    f_min = min(A[i], temp * A[i], f_min * A[i]);
    ans = max(ans, f_max) ;
end
return ans;
```

3 Problem 3

First sort all the points based on their x coordinate. To index our subproblem, we will give the rightmost point for both the path going to the left and the path going to the right. Then, we have that the desired result will be the subproblem indexed by v, where v is the rightmost point. Suppose by symmetry that we are further along on the left-going path, that the leftmost path is going to the ith one and the right going path is going until the jth one. Then, if we have that $i > j + 1$, then we have that the cost must be the distance from the $i - 1$st point to the ith plus the solution to the subproblem obtained where we replace i with $i - 1$. There can be at most $O(n^2)$ of these subproblem, but solving them only requires considering a constant number of cases. The other possibility for a subproblem is that $j \leq i \leq j + 1$. In this case, we consider for every k from 1 to j the subproblem where we replace i with k plus the cost from kth point to the ith point and take the minimum over all of them. This case requires considering $O(n)$ things, but there are only $O(n)$ such cases. So, the final runtime is $O(n^2)$.

4 Problem 4

(a) To index the subproblem, we will give the first k items from the item list and the maximum weight of current knapsack. There can be at most $O(nW)$ of these subproblem. For each item, we can decide to put it in or not. If we put the ist item in, we need to prepare w_i weight for it and earn v_i value. The value is v_i plus the solution to the subproblem with the first $i-1$ items, and $W - w_i$ capacity. Otherwise, the value is as same as the solution to the subproblem with the first $i-1$ items, and W capacity. We take the maximum value from the two choices for each item.

(b) No since W can increase exponential as input length.

5 Problem 5

From the definition, there is an algorithm A_1 solves L_1 in $O(n^{k_1})$, and A_2 solves L_2 in $O(n^{k_2})$.

5.1 $L_1 \cup L_2$

Algorithm 2: $A_3(x)$

if $A_1(x)$ **then**
    return true;
end
if $A_2(x)$ **then**
    return true;
end
return false;

A_3 solves $L_1 \cup L_2$ in $O(n^{\max(k_1,k_2)})$, so $L_1 \cup L_2 \in P$.

5.2 $L_1 \cap L_2$

Algorithm 3: $A_3(x)$

if $A_1(x)$ **then**
    if $A_2(x)$ **then**
        return true;
    end
end
return false;

A_4 solves $L_1 \cap L_2$ in $O(n^{\max(k_1,k_2)})$, so $L_1 \cap L_2 \in P$.

5.3 $L_1 L_2$

Algorithm 4: $A_5(x)$

for $i = 1$ *to* $(n-1)$ **do**
    if $A_1(x_1 \ldots x_i)$ **then**
        if $A_2(x_{i+1} \ldots x_n)$ **then**
            return true;
        end
    end
end
return false;

A_5 solves $L_1 L_2$ in $O(n^{\max(k_1,k_2)+1})$, so $L_1 L_2 \in P$.

5.4 $\overline{L}_1$

A_6 solves $\overline{L}_1$ in $O(n^{k_1})$, so $\overline{L}_1 \in P$.

```
Algorithm 5: A_6(x)
if A_1(x) then
    return false;
end
return true;
```

5.5 L_1^*

Let $f[i][j]$ indicate whether $x_i \dots x_j$ is belong to L_1^*.

$\forall x \in L_1^*, x \in L_1$ or $\exists k, x_1 \dots x_k \in L_1^* \wedge x_{k+1} \dots x_n \in L_1^*$.

Therefore, we can construct a dynamic programming algorithm A_7. A_7 solves L_1^* in $O(n^{k_1+2})$, so $L_1^* \in P$.

```
Algorithm 6: A_7(x)
for i = 1 to n do
    for j = i + 1 to n do
        f[i][j] = false;
    end
end
for i = 1 to n do
    for j = i to n do
        if A_1(x_i ... x_j) then
            f[i][j] = true;
        end
    end
end
for i = 1 to n do
    for j = i + 1 to n do
        for k = i to (j - 1) do
            if f[i][k] and f[k + 1][j] then
                f[i][j] = true;
            end
        end
    end
end
return f[1][n];
```

6 Problem 6

(a)

Consider (deterministic polynomial time) Turing Machine M defined as follows: Let $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$, Turing Machine M accept input (G_1, G_2, π) if and only if π is a bijection between G_1 and G_2, and $\pi(u)\pi(v) \in E_2$ if and only if $uv \in E_1$.

(b)

Note that $\overline{TAUTOLOGY}$ is the language of boolean formulas which can be false for some input.

7 Problem 7

(a)

for any language $L \in P$, there exists a (deterministic polynomial time) Turing Machine M defined as follows: $M(x) = 1$ if and only if $x \in L$. Therefore, we may define a (deterministic polynomial time) Turing Machine M' in the following way.

- for any x and proof w, $M'(x, w) = 1 - M(x)$.

It can be seen that $\bar{L} \in NP$. Hence, $L \in coNP$.

(b)

If not, $L \in NP \Rightarrow L \in P \Rightarrow \bar{L} \in P \Rightarrow \bar{L} \in NP \Rightarrow L \in coNP$. Similarly, we have $L \in coNP \Rightarrow L \in NP$, leading to contradiction.