Algorithm Homework 04

Qiu Yihang

May 2022

1 Maximum Revenues

1.1 Maximum (1,1)-Revenue

Solution. We design algorithm as follows.

Algorithm 1: Maximum (1,1)-Revenue Search

```
Function Maximum (1,1)-Revenue (a, n)
    maxrev ← 0, sum ← 0;
    for i = 1 → n do
        sum ← sum + a_i;
        if maxrev < sum then maxrev ← sum;
        if sum < 0 then sum ← 0;
    end
    Return: maxrev
end
```

The correctness of the algorithm is trivial since *sum* always holds the sum of the (1,1)-step subsequence which has the potential to have the maximum revenue. When *sum* is negative, obvious discarding the previous sequence is better since $0 > sum$ in this case. Meanwhile, *max* holds the maximum revenue of all (1,1)-step subsequence potential to have the maximum revenue.

Now we analyze the time complexity of our algorithm. Obvious it is $\underline{\boldsymbol{O(n)}}$. ■

1.2 $O(n^2)$ Algorithm for Maximum (L, R)-Revenue

Solution. Use $f[i]$ to store the maximum revenue of all $(L, R)-$step subsequences ending with a_i.

Then we have state transition equation

$$f[i] = \max\left(0, \max_{\max(1,i-R)\le k\le i-L} f[k]\right) + a_i. \quad (i = L+1, L+2, ..., n)$$

Boundaries: $f[i] = a_i, \quad i = 1, 2, .., L.$

The final result is $\max\limits_{1\le k\le n} f[k]$. Then we can solve the problem by **dynamic programming**.

The correctness of the algorithm is trivial.

(Since for any i, we list all possible previous element in the $(L, R)-$step subsequence ending with a_i, i.e. all states with potential to have maximum revenue are covered.)

Now we analyze the time complexity of algorithm above.

The number of states is $O(n)$. To update $f[i]$, we need to scan $R-L$ states, i.e. $O(n)$ states.

Thus, the time complexity of the algorithm above is $\underline{\boldsymbol{O(n^2)}}$. ■

1.3 $O(n)$ Algorithm for Maximum (L, R)-Revenue

Solution. We still use the notations in **1.2**.

In the algorithm in **1.2**, the process of finding $\max\limits_{\max(1,i-R)\leq k\leq i-L}(f[k])$ takes $O(n)$ time.

In fact, this task is to find the maximum in $(R-L)$ consecutive numbers ($f[i-R]$, $f[i-R+1], ...f[i-L]$). Thus, the process can be speed up to $O(1)$ time with the help of a monotonous priority queue which can pop out elements in the head and in the tail. This optimization has been fully covered in the class.

The algorithm is as follows.

Algorithm 2: Maximum (L, R)-Revenue Search

Function *Maximum* (L, R)*-Revenue* (a, n, L, R)

$maxrev \leftarrow 0$;

$\forall i \in [1, n], f[i] \leftarrow a_i$;

$Q \leftarrow \varnothing$;

// Q is the monotonous priority queue, in which the index is increasing and the value is decreasing.

for $i = L + 1 \to n$ **do**

  if Q.head().index $< i - R$ **then** Q.dequeue(head);

  while Q.tail().value $\leq f[i-L]$ **do** Q.dequeue(tail);

  Q.enqueue$\Big(\{\text{index} = i - L, \text{ value} = f[i-L]\}\Big)$;

  $f[i] \leftarrow \texttt{max}(f[i], Q.\text{head}().\text{value}) + a_i)$;

  $maxrev \leftarrow \texttt{max}(maxrev, f[i])$;

end

Return: $maxrev$

end

All elements have been enqueueed and dequeued for at most once during the whole process, i.e. the while-loop takes $O(n)$ time in total. Meanwhile, for-loop takes $O(n)$ time.

Thus, the time complexity of our algorithm is $\underline{\boldsymbol{O(n)}}$. ■

2 Optimal Indexing for A Dictionary

Solution. We use the abbreviation BST to denote binary searching tree.

For only one word, there exists exactly one BST, which is trivially the best BST.

For any dictionary, as long as we have the best BST of its subsets with alphabetic order, we can construct all BSTs with potentials to minimize the number of comparisons and therefore find the optimal BST.

Based on the idea above, we design the following algorithm.

Use $f[i,j]$ to denote the minimum number of comparisons of the BST for words $a_i, a_{i+1}, ...a_j$.

Use $T_{i,j}$ to denote the BST of $a_i, a_{i+1}, ...a_j$ with the minimum number of comparisons. (In fact, when coding the algorithm, we just need to record the root of $T_{i,j}$.)

We can list out all possible root of the BST for $a_i, a_{i+1}, ..., a_j$ and find the optimal one.

Thus, the state transition equation is as follows.

$$\begin{aligned} f[i,j] &= \min_{i\le k\le j}(f[i,k-1], f[k+1,j]) + \sum_{l=i}^{j} w_l \\ k^\star &= \operatorname*{argmin}_{i\le k\le j}(f[i,k-1], f[k+1,j]) \\ T_{i,j} &= \text{a tree with root } k^\star, \\ &\quad \text{whose left and right subtree is} \\ &\quad T_{i,k^*-1} \text{ and } T_{k^*+1,j} \text{ respectively.} \end{aligned}$$

Boundaries:

$$\begin{aligned} f[i,j] = 0,\ T_{i,j} = \varnothing, &\qquad \text{when } i > j. \\ f[i,j] = w_i,\ T_{i,j} = \text{the tree with one node } i \text{ only}, &\qquad \text{when } i = j. \end{aligned}$$

The final result (the best BST with minimum comparisons for the n words) is $\underline{\boldsymbol{T_{1,n}}}$.

During the dynamic programming, we compute $f[i,j]$ and $T_{i,j}$ in the increasing order of $|j-i|$.

The number of states is $O(n^2)$. For each state, we need to scan $O(n)$ states to compute the current optimal solution. Therefore, the time complexity of our algorithm is $O(n^3)$. ■

3 Palindrome Subsequence

Solution. Let the given string be $S = S_1S_2...S_n$.

Use $S(i:j)$ to denote the string $S_iS_{i+1}...S_j$.

Use $f[i,j]$ to denote the maximum length of palindrome which is a subsequence of $S(i:j)$.

Use $a[i,j]$ to denote the palindromic subsequence with maximum length of $S(i:j)$.

The state transition equation is as follows.

$$f[i,j] = \max\Big(f[i+1,j], f[i,j-1], f[i+1,j-1] + 2 \cdot \mathbb{1}\left[S_i = S_j\right]\Big),\ 1 \le i < j \le n$$

$$a[i,j] = \begin{cases} S_i + a[i+1,j-1] + S_i, & f[i,j] = f[i+1,j-1] + 2 \text{ while } S_i = S_j \\ a[i+1,j-1], & f[i,j] = f[i+1,j-1] \text{ while } S_i \neq S_j \\ a[i+1,j], & f[i,j] = f[i+1,j] \\ a[i,j-1], & f[i,j] = f[i,j-1] \end{cases}$$

for $1 \le i < j \le n$

(where $a + b$ for string or alphabet a and b means adding b to the tail of a.)

Boundaries:

$$f[i,i] = 1, a[i,i] = S_i, \qquad 1 \le i \le n$$

$$f[i,j] = 0, a[i,j] = \varnothing, \qquad 1 \le j < i \le n$$

(The second boundary is set to make sure palindromes with even length can be correctly discovered by our algorithm.)

The final result is $a[1,n]$.

Now we analyze the time complexity of our algorithm.

Let the length of the given string be n. The number of states is $O(n^2)$.

For each state, it takes $O(1)$ to determine the optimal solution.

Thus, the running time of our algorithm is $\underline{\boldsymbol{O(n^2)}}$. ■

4 Independent Sets on Tree

Notations. We use the following notations in this section.

Let $G = (V, E)$. Let the subtree of G with root u be T_u. Let the root of G be r.

Let the set of children of a node u be `children`(u).

Let the set of grandchildren of a node u be `grandchildren`(u).

4.1 The Number of Independent Sets

Solution. Use $f[u]$ to denote the number of independent sets containing u on T_u.

Use $g[u]$ to denote the number of independent sets not involving u on T_u.

Then the state transition equation is as follows.

$$f[u] = \prod_{v \in \texttt{children}(v)} g[v],$$

$$g[u] = \prod_{v \in \texttt{children}(u)} (f[v] + g[v]),$$

for $u \in V$

Boundaries:

$$f[u] = 1, g[u] = 0, \qquad \text{when } u \text{ is a leaf of } G.$$

Then the final result is $(f[r] + g[r])$.

Now we prove the correctness of the algorithm by induction.

BASE STEP. When u is a leaf, obvious there exists one independent set, i.e. $\{u\}$.

In this case, obvious $f[u] = 1, g[u] = 0$.

INDUCTION HYPOTHESIS.

For any $v \in \texttt{children}(u)$, $f[v]$ is the number of independent sets containing v on T_v, while $g[v]$ is the number of independent sets not involving v on T_v.

INDUCTIVE STEP.

CASE 01. We consider independent set A on T_u s.t. $u \in A$.

Obvious $\forall v \in \texttt{children}(u), v \notin A$.

We can choose from the independent sets not involving v of all G_vs , combine them and add u to construct such A.

Thus, there are $\prod_{v \in \texttt{children}(u)} g[v]$ such independent sets in total, i.e.

$$f[u] = \prod_{v \in \texttt{children}(u)} g[v]$$

.

CASE 02. We consider independent set A on T_u s.t. $u \notin A$.

Then we can safely choose from all independent sets of T_v for $v \in \texttt{children}(u)$ and combine them to construct such A.

Thus, there exist $\prod_{v \in \texttt{children}(u)} (f[v] + g[v])$ such independent sets in total, i.e.

$$g[u] = \prod_{v \in \texttt{children}(u)} (f[v] + g[v])$$

Therefore, for u, $f[u]$ is the number of independent sets containing u on T_u while $g[u]$ is the number of independent sets not involving u on T_u.

For G, i.e. T_r, obvious the number of all independent sets is $f[r] + g[r]$. □

Now we analyze the time complexity of our algorithm.

The number of states is exactly $|V|$. For each state, we visit several other states to reach the optimal solution of the subproblem. Since each vertex has at most one parent, we know this visiting process takes $O(|V|)$ time in total.

Moreover, since storage and multiplication only takes $O(1)$ time in this problem, we know it takes $O(1)$ time for each state to reach the optimal solution after visiting all its subproblems.

Thus, the time complexity of our algorithm is $\boldsymbol{O(|V|)}$. ■

4.2 The Number of Maximum Independent Sets

Solution. A natural idea is to record the size of the maximum independent set of the subtree.

Based on the idea above, we design the following algorithm.

Use $s[u]$ to denote the size of maximum independent sets on T_u.

Use $F[u]$ to denote the number of maximum independent sets on T_u.

Then the state transition equation is as follows.

$$s[u] = \max\left(\sum_{v\in \texttt{children}(u)} s[v], 1 + \sum_{w\in \texttt{grandchildren}(u)} s[w],\right) \qquad \text{for } u \in V.$$

$$\text{Define} \quad \begin{cases} \texttt{size}(\notin) \triangleq \sum\limits_{v\in \texttt{children}(u)} s[v] \\ \texttt{size}(\in) \triangleq 1 + \sum\limits_{w\in \texttt{grandchildren}(u)} s[w] \end{cases},$$

$$F[u] = \begin{cases} \prod\limits_{v\in \texttt{children}(u)} F[v], & \texttt{size}(\notin) > \texttt{size}(\in) \\ \prod\limits_{v\in \texttt{children}(u)} F[v] + \prod\limits_{w\in \texttt{grandchildren}(u)} F[w], & \texttt{size}(\notin) = \texttt{size}(\in) \\ \prod\limits_{w\in \texttt{grandchildren}(u)} F[w], & \texttt{size}(\notin) < \texttt{size}(\in) \end{cases}$$

for $u \in V$. (Note: if w does not exist, $F[w] = 0$.)

Boundaries:

$$s[u] = 1, F[u] = 1, \qquad \text{when } u \text{ is a leaf of } G.$$

The final result is $F[r]$.

Now we prove the correctness of the algorithm.

Use $s_{(\in)}[u]$ and $s_{(\notin)}[u]$ to denote the size of maximum independent set containing and not involving u on T_u respectively. We have $s[u] = \max\Big(s_{(\in)}[u], s_{(\notin)}[u]\Big)$.

Use $F_{(\in)}[u]$ and $F_{(\notin)}[u]$ to denote the number of maximum independent set containing and not involving u on T_u respectively. We have

$$F[u] = \begin{cases} F_{(\in)}[u], & s_{(\in)}[u] > s_{(\notin)}[u] \\ F_{(\in)}[u] + F_{(\notin)}[u], & s_{(\in)}[u] = s_{(\notin)}[u] \\ F_{(\notin)}[u], & s_{(\in)}[u] < s_{(\notin)}[u] \end{cases}$$

Similarly to **4.1**, we know if an independent set contains u, to maximize the size, we need to select all nodes in the maximum independent sets not involving v on T_v for all $v \in \texttt{children}(u)$. Thus, the size is the sum of maximum independent sets not involving v on T_v added by 1 (since u is in the set).

As for independent set not involving u on T_u, the maximum independent sets must contain all the nodes in the maximum independent set of v on T_v for all $v \in \texttt{children}(u)$.

The state transition of number of maximum independent sets is almost the same as the one in **4.1**. We just need to replace $g[v]$ with $F_{(\notin)}[v]$ and $f[v] + g[v]$ (the total number) with $F[v]$.

Therefore, we have

$$\begin{cases} s_{(\in)}[u] = 1 + \sum\limits_{v \in \texttt{children}(u)} s_{(\notin)}[v] \\ s_{(\notin)}[u] = \sum\limits_{v \in \texttt{children}(u)} s[v] \\ F_{(\in)}[u] = \prod\limits_{v \in \texttt{children}(u)} F_{(\notin)}[v] \\ F_{(\notin)}[u] = \prod\limits_{v \in \texttt{children}(u)} F[v] \end{cases} \Longrightarrow \begin{cases} s_{(\in)}[u] = 1 + \sum\limits_{w \in \texttt{grandchildren}(u)} s[w] \\ F_{(\in)}[u] = \prod\limits_{w \in \texttt{grandchildren}(u)} F[w] \end{cases}$$

Thus,

$$s[u] = \max\Big(s_{(\in)}[u], s_{(\notin)}[u]\Big) = \max\left(\sum_{v \in \texttt{children}(u)} s[v], 1 + \sum_{w \in \texttt{grandchildren}(u)} s[w]\right)$$

$$F[u] = \begin{cases} \prod\limits_{w \in \texttt{grandchildren}(u)} F[w], & s_{(\in)}[u] > s_{(\notin)}[u] \\ \prod\limits_{w \in \texttt{grandchildren}(u)} F[w] + \prod\limits_{v \in \texttt{children}(u)} F[v], & s_{(\in)}[u] = s_{(\notin)}[u] \\ \prod\limits_{v \in \texttt{children}(u)} F[v], & s_{(\in)}[u] < s_{(\notin)}[u] \end{cases}$$

which is exactly the state transition equation of our algorithm. □

Now we analyze the time complexity of our algorithm.

The number of all states is exactly $|V|$.

Similar to **4.1**, since each vertex has at most one parent and at most one grandparent, the visiting time of one's children and grandchildren is $O(|V|)$ in total. Meanwhile, since multiplication, addition and storage takes $O(1)$ time, we know for each state, reaching optimal solution after visiting all subproblems takes $O(1)$ time.

Thus, the time complexity of our algorithm is $\boldsymbol{O(|V|)}$. ■

5 Rating and Feedback

The completion of this homework takes me three days, about 24 hours in total. Still, writing a formal solution is the most time-consuming part.

The ratings of each problem is as follows.

Problem	Rating
1.1	1
1.2	1
1.3	2
2	3
3	2.5
4.1	3
4.2	4

Table 1: Ratings.

This time I finish all problems on my own.