SOLUTION FOR ASSIGNMENT 4, ALGORITHM DESIGN AND ANALYSIS

1. (25 points) A *palindrome* is a non-empty string over some alphabet that reads the same forward and backward. Examples of palindromes are all strings of length 1, civic, racecar, and aibophobia (fear of palindromes).

   Give an efficient algorithm to find the longest palindrome that is a subsequence of a given input string. For example, given the input character, your algorithm should return carac. What is the running time of your algorithm?

   Solution. Assume that the input string is $S_{1,\dots,n}$ with length n, and let $[n] := \{1,\dots,n\}$. Let $f(l,r)$ be the length of the longest sub-sequence of $S_{l,\dots,r}$ that is a palindrome (the longest sub-palindrome, for short). Then, we have the following transition:

   $$f(l,r) = \max\left\{f(l+1,r), f(l,r-1), f(l+1,r-1) + 2\cdot \mathbb{1}_{S_l=S_r}\right\}, \text{ if } l < r$$

   and the boundary condition

   $$f(i,i) = 1, \forall i \in [n]$$

   with convention $f(l,r) = 0$ for all $l > r$. The length of the longest sub-palindrome of S is $f(1,n)$. To output the answer, we just need to record the previous state we transform from and output the characters recursively. Now we show the correctness and the time complexity respectively.

   Proof of the Correctness. Firstly, for all $i \in [n]$, it is trivial that the longest sub-palindrome of S_i is exactly itself. Thus it holds that $f(i,i) = 1$ for all $i \in [n]$.

   Then, observe that, for all $l < r$, the longest sub-palindrome of $S_{l,\dots,r}$ is among the following cases:

   (a) the longest sub-palindrome of $S_{l+1,\dots,r}$. And it is of length $f(l+1,r)$.
   (b) the longest sub-palindrome of $S_{l,\dots,r-1}$. And it is of length $f(l,r-1)$.
   (c) when $S_l = S_r$, the longest sub-palindrome of $S_{l,\dots,r}$ might be

   $$S_l + \text{the longest sub-palindrome of } S_{l+1,\dots,r-1} + S_r.$$

   Its length is $f(l+1,r-1)+2$.

   Combining all above, we show the correctness of the our algorithm. □

   Time Complexity. Now we show the time complexity of our dynamic programming algorithm. Firstly, the states of our algorithm is

   $$|\{(l,r)|1 \le l \le r \le n\}| = \sum_{l=1}^{n}\sum_{r=l}^{n} 1 = \frac{n(n+1)}{2} = O\left(n^2\right).$$

   Secondly, at each step of the transform, we only need to do the operations 'max' and '+' for $O(1)$ times. Also we need $O(n)\cdot O(1) = O(n)$ to initialize all boundary conditions.

   Then the whole time of our algorithm is

   $$T(n) = O(n^2)\cdot O(1) + O(n) = O(n^2).$$

   □

   □

2. (25 points) Consider the following 3-PARTITION problem. Given positive integers $a_1,\dots,a_n$, we want to determine whether it is possible to partition $\{1,\dots,n\}$ into three disjoint subsets I,J,K such that

   $$\sum_{i\in I} a_i = \sum_{j\in J} a_j = \sum_{k\in K} a_k = \frac{1}{3}\sum_{i=1}^{n} a_i.$$

For example, for input $(1, 2, 3, 4, 4, 5, 8)$ the answer is yes, because there is the partition $(1, 8)$, $(4, 5)$, $(2, 3, 4)$. On the other hand, for input $(2, 2, 3, 5)$, the answer is no. Devise and analyze a dynamic programming algorithm for 3-PARTITION that runs in time polynomial in n and $\sum_{i=1}^{n} a_i$.

Solution. If $3 \nmid \sum_{i=1}^{n} a_i$, it is trivial that the answer is no. Then we assume that $3 \mid \sum_{i=1}^{n} a_i$. Since $I \cup J \cup K = [n]$, it makes natural sense to only consider the subsets I and J. Let $[m]$ be the set $\{1, \ldots, m\}$ for all $m \leq n$. Denote by $f(m, S_1, S_2) \in \{0, 1\}$ whether it is possible to pick two disjoint subsets $I^{(m)}$, $J^{(m)}$ of $[m]$ such that

$$\sum_{i \in I^{(m)}} a_i = S_1, \sum_{j \in J^{(m)}} a_j = S_2.$$

We have the following transition: for all $m \geq 1$, $S_1, S_2 \leq \sum_{i \in [m]} a_i$,

$$f(m, S_1, S_2) = f(m-1, S_1, S_2) \vee f(m-1, S_1 - a_m, S_2) \vee f(m-1, S_1, S_2 - a_m),$$

and the boundary condition

$$f(0, 0, 0) = 1$$

with convention $f(m, S_1, S_2) = 0$ for all $0 \leq m \leq n$, $S_1 < 0$ or $S_2 < 0$. The result of 3-PARTITION problem is exactly $f\left(n, \frac{1}{3}\sum_{i=1}^{n} a_i, \frac{1}{3}\sum_{i=1}^{n} a_i\right)$.

Now we prove the correctness of our algorithm and show the time complexity.

Proof of the Correctness. By definition, the 3-PARTITION problem is equivalent to the problem whether there exist two disjoint subsets I, J of $[n]$ such that

$$\sum_{i \in I} a_i = \sum_{j \in J} a_j = \frac{1}{3} \sum_{i=1}^{n} a_i.$$

Now we consider the problem whether there exist two disjoint subsets $I^{(m)}$, $J^{(m)}$ of $[m]$ such that

$$\sum_{i \in I^{(m)}} a_i = S_1, \sum_{j \in J^{(m)}} a_j = S_2.$$

We consider the element a_m. There are three choices for a_m:

(a) $a_m \in I^{(m)}$. Then we turn to verify the instance $(m-1, S_1 - a_m, S_2)$.
(b) $a_m \in J^{(m)}$. Then we turn to verify the instance $(m-1, S_1, S_2 - a_m)$.
(c) $a_m \notin I^{(m)}$ and $a_m \notin J^{(m)}$. Then we turn to verify the instance $(m-1, S_1, S_2)$.

Then we show our transition is correct. The correctness of the boundary condition is trivial. □

Time Complexity. By definition, we have at most

$$\left(\sum_{m=1}^{n}\left(\sum_{i=1}^{m} a_i\right)^2\right) = O\left(n\left(\sum_{i=1}^{n} a_i\right)^2\right)$$

states in all. For the transition, we only need to do the arithmetic and boolean operations for $O(1)$ times, and it takes $O(1)$ time to initialize the boundary condition (with the assumption that the whole array is set 0 before all). Then the time complexity is

$$T(n) = O\left(n\left(\sum_{i=1}^{n} a_i\right)^2\right) \cdot O(1) + O(1) = O\left(n\left(\sum_{i=1}^{n} a_i\right)^2\right).$$

□

□

3. (25 points) Recall the TSP problem we have studied in the lecture. Now we consider one of its variants. Given a non-negative weighted graph $G = (V, E, c)$, a start point $s \in V$, and a profit function $p(u)$ for each $u \in V$ (also non-negative). Unlike TSP, we are not required to go through all vertices exactly. We only need to select a subset of vertices U (other than v), design a walk from s, go through every vertex in U, and return to s. Also, different from TSP,

we can go through a vertex or an edge multiple times. Your total profit is $\sum_{u \in U} p(u)$, and your total cost is the sum of $c(e)$ on every edge e you go through. The objective is to maximize your total profit minus the total cost. Design a DP algorithm for it. (Your running time can be exponential).

Solution. Let $f(i, S)$ be the maximum net profit (the total profit minus the total cost) when we arrive at the point $i \in U$, and we have already visited $S \subseteq U$. Denote by $C(u, v)$ the minimum cost traveling from u to v in G. It is not hard to show C can be pre-processed in time $O(|V|(|V| + |E|) \log |V|)$.

Now we have the following transition:

$$f(i, S) = \begin{cases} \max\limits_{j \in S \setminus \{i\}} f(j, S \setminus \{i\}) + p(i) - C(j, i), & i \in S, \\ -\infty & i \notin S \end{cases}$$

with the boundary condition

$$f(s, \mathbb{1}_{s \in U} \{s\}) = p(s) \cdot \mathbb{1}_{s \in U}.$$

The result is

$$\max_{v \in U} f(v, U) - C(v, s).$$

To implement the dynamic programming algorithm described above, we compute f from the subsets of U with small size to the ones with large sizes. Now we show the correctness of the algorithm and analyze the time complexity.

Proof of Correctness. Firstly, observe that every trace starting from s, going through U and ending at s can be divided into two parts:
(1) the trace ℓ_1 starting from s, going through U, ending at some point $v \in U$;
(2) the trace ℓ_2 from v to s.
Note that the net profit of ℓ_2 is exactly $-C(v, s)$, and the net profit of ℓ_1 is $f(v, U)$ directly by definition. Additionally, the correctness of the boundary condition is trivial.

Now we show the correctness of the transition. The negative case is obviously correct, and we only to verify the correctness of the transition when $i \in S$. For convenience, without loss of generality we assume $s \notin U$. Consider the trace $s \to v_1 \to \ldots \to v_k \to i$ where $\{v_1, \ldots, v_k, i\} = S$ and $v_1, \ldots, v_k, i$ are different vertices. It is clear that all traces starting from s, going through U and ending at i can be expressed as $s \to v_1 \to \ldots \to v_k \to i$. Then the net profit of such a trace is

$$f(v_k, S \setminus \{i\}) + p(i) - C(v_k, i)$$

which implies the correctness of our algorithm. □

Time Complexity. We have at most

$$|V| \cdot \sum_{S \subseteq U} 1 = O\left(|V| \cdot 2^{|U|}\right)$$

states in all. For each transition, it takes $O(|V|)$ times to update $f(i, S)$ (enumerate the vertex j). To initialize the boundary condition and C, it takes $O(|V|(|V| + |E|) \log |V|)$. Combining all above, we obtain the time complexity is

$$T(n) = O\left(|V|^2 \cdot 2^{|U|} + |V|(|V| + |E|) \log |V|\right)$$

□

□

Remark. For a more precise analysis, note that we only need to consider $O\left(|U| \cdot 2^{|U|}\right)$ states. ♣

4. (25 points) We are given a sequence of integers $a_1, \ldots, a_n$, a lower bound and an upper bound $1 \le L \le R \le n$. An (L, R)-step sub-sequence is a sub-sequence $a_{i_1}, a_{i_2}, \ldots, a_{i_\ell}$, such that $\forall 1 \le j \le \ell - 1, L \le i_{j+1} - i_j \le R$. The revenue of the sub-sequence is $\sum_{j=1}^{\ell} a_{i_j}$. Design a DP algorithm to output the maximum revenue we can get from an (L, R)-step sub-sequence.
   (a) (5 points) Suppose $L = R = 1$. Design a DP algorithm in $O(n)$ to find the maximum $(1, 1)$-step sub-sequence.
   Solution. When $L = R = 1$, the sub-sequence must be continuous. Let $f(i)$ be the maximum revenue of all $(1, 1)$-step sub-sequences ending at i. Then we have the following transition
   $$f(i) = \max\{f(i-1) + a_i, a_i\}, \forall i \ge 1$$
   and the boundary condition
   $$f(0) = 0.$$
   The answer is $\max_{1 \le i \le n} f(i)$.
   It is trivial to show the time complexity of the dynamic programming algorithm is $O(n)$, since we have $O(n)$ states and the transition takes $O(1)$ operations. The correctness of the algorithm comes from the observation that: for every $(1, 1)$-step sub-sequence ending at i, there are two cases
   (1) a single element a_i;
   (2) an $(1, 1)$-step sub-sequence s' ending at $i - 1$ with a_i added to the tail of s'.
   □
   (b) (10 points) Design a DP algorithm in $O(n^2)$ to find the maximum revenue of (L, R)-step sub-sequences for any L and R.
   Solution. Similarly to (a), let $f(i)$ be the maximum revenue of all (L, R)-step sub-sequences ending at i. Then we have the following transition
   $$f(i) = \max\left\{a_i, a_i + \max_{j \in [\max\{i-R,1\}, i-L]} f(j)\right\}, \forall i \ge L$$
   and the boundary condition
   $$f(i) = a_i, \forall 1 \le i \le L.$$
   The answer is $\max_{1 \le i \le n} f(i)$.
   It is not hard to show the time complexity is
   $$T(n) = O(n \cdot (R - L)) = O(n^2).$$
   To show the correctness of the algorithm, similarly to the solution of (a), note that every (L, R)-step sub-sequence ending at i can be considered as the following two cases
   (1) a single element a_i.
   (2) an (L, R)-step sub-sequence s' ending at j for some $j \in [\max\{i - R, 1\}, i - L]$ with a_i added to the tail of s'.
   Then we can naturally show the correctness of our algorithm. □
   (c) (10 points) Design a DP algorithm in $O(n)$ to find the maximum revenue of (L, R)-step sub-sequence for any L and R. (Tips: Refer to the 'Priority Queue' technique of the k-largest number problem in the lecture)
   Solution. The solution is technical. We use a 'dequeue' Q (a kind of queue that we can pop the tail) to maintain the 'current useful information'. We modify the 'push' operation as the following: once we push an index i into Q, we pop the tail element Q_T until Q is empty or $f(Q_T) > f(i)$; then we push the index i into Q. To optimize our algorithm in (b), we firstly compute the boundary conditions and push $1, 2, \ldots, L$ into Q. When we compute $f(i)$, before update we pop the head of the dequeue Q_h until $i - Q_h \le R$, and we update $f(i) = \max\{a_i, f(Q_h) + a_i\}$. After computing $f(i)$, we push the index i into Q and compute the next index.
   To show the correctness, we only need to show, after every 'push' operation, the elements in Q are 'decreasing' with respect to the value of f. In fact, if there exists i such that

$f(Q_i) > f(Q_{i+1})$, when pushing the index Q_{i+1}, we will pop Q_i according to our modified 'push' operation. Then we can show $Q_h = \text{argmax}_{j\in[\max\{1,i-R\},i-L]} f(j)$ with the fact that if k is not in Q, it cannot be the optimal choice.

The time complexity consists of the following three parts: the time for initialization; the total update time; and the time for dequeue operations (push and pop). It is trivial to see the time for initialization is $O(L) = O(n)$ and the total update time is $O(n)$. To show the time cost for dequeue operations, note that for each index $i \in [n]$, we only pop it out of the dequeue for at most 1 time (whether it is at the tail or at the head) and push it into the dequeue for exact 1 time. And the real time cost for one pop or push is $O(1)$. Then we can show the time cost for dequeue operations is $O(n)$. Combining all above, we obtain the time complexity is $T(n) = O(n)$. □