The Australian National University Semester 2, 2024
School of Computing Tutorial 11

Convenor(s): Yun Kuen Cheung (Marco) & Ahad N. Zehmakan

Concrete version: Thursday 17th October, 2024, 14:17 (Compilation date and time)

Algorithms (COMP3600/6466)

Problems marked with (∗) are challenge exercises. They will not be discussed in tutorials, and solutions will not be released. Once confident in your solution for a challenge exercise, you are welcome to discuss it with your tutor during the consultation period in the tutorial session, or schedule a time with Marco to present it.

Exercise 1 ***Longest Increasing/Common/Alternating Subsequence Problems***

1. In the lecture, we discuss a DP algorithm for computing the length of longest increasing subsequence (LIS) of a given array. Now, instead of computing the length of LIS, we want to compute a LIS. Design a DP algorithm that computes any LIS of a given array. For instance, if $A = [1, 2, 6, 3]$, the two LIS are $[1, 2, 6]$ and $[1, 2, 3]$. Your algorithm is correct if it computes either of the two LIS.

   Hint: In the lecture, we discuss a DP algorithm for computing a longest common subsequence (LCS) of two given arrays, which uses *pointers* to help computing a LCS. This should give you useful ideas on how to design an algorithm that computes a LIS.

   Solution

   We can reuse the algorithm for the LIS problem in the lecture, and make a few modifications. The key idea is when computing
   $$L[i] \;=\; 1 + \max_{\substack{j: i+1 \leq j \leq n-1, \\ A[j] \geq A[i]}} L[j] \;,$$
   using a double-for loop, we simultaneously record the j which attains the above maximum, and save it as $ptr[i]$. After computing the arrays L and ptr, we find out i^* that attains the maximum $L[i^*]$, and starting from $D[i^*]$, we trace the pointers to recover a LIS. The code is given below.

```
def longest_increasing_subsequence(A):
    n = len(A)
    L = [1 for i in range(n)]
    ptr = [-1 for i in range(n)]
    for i in range(n − 2,-1-1):
        runningmax = 1
        for j in range(i + 1,n):
            if A[j] >= A[i]:
                if 1+L[j] > runningmax:
                    runningmax = 1+L[j]
                    ptr[i] = j
        L[i] = runningmax

    istar = 0
    for i in range(1,n):
        if L[i] > L[istar]:
            istar = i

    lis = [ A[istar] ]
    while ptr[istar] != -1:
        istar = ptr[istar]
        lis.append(A[istar])
    return lis
```

2. Given an array A, $\langle A[i_0], A[i_1], \ldots, A[i_{\ell-1}]\rangle$ is an *alternating subsequence* of A if either of the following is true:

- for even j, $A[i_j] > A[i_{j+1}]$, **and** for odd j, $A[i_j] < A[i_{j+1}]$;
- for even j, $A[i_j] < A[i_{j+1}]$, **and** for odd j, $A[i_j] > A[i_{j+1}]$.

For instance, if $A = [8, 1, 2, 7, 6, 3, 9]$, two alternating subsequences of A are $[1, 7, 3, 9]$ and $[8, 2, 6, 3, 9]$.

We consider the following problem: given an array A or integers, find the length of the longest alternating subsequence (LAS).

The following notions will be useful for formulating a DP algorithm for the LAS problem:

- We say an alternating subsequence is initially-raising if its length is 1, or its length is more than 1 and its second number is larger than the first number;
- We say an alternating subsequence is initially-dropping if its length is 1, or its length is more than 1 and its second number is smaller than the first number;
- Let $R[i]$ denote the length of longest initially-raising alternating subsequence that starts with $A[i]$;
- Let $D[i]$ denote the length of longest initially-dropping alternating subsequence that starts with $A[i]$.

(a) The boundary cases are $R[n-1]$ and $D[n-1]$. What are their values?

(b) For $0 \le i \le n-2$, let $J = \{j \mid j > i \text{ and } A[j] > A[i]\}$. Explain why

$$R[i] \;=\; \begin{cases} 1 + \max_{j \in J} D[j], & \text{if } J \text{ is not empty;} \\ 1, & \text{if } J \text{ is empty.} \end{cases}$$

(c) Following the ideas in (b), formulate a recurrence of $D[i]$ in terms of $R[k]$ for some $k > i$.

(d) Using (a), (b) and (c), design a DP algorithm for the LAS problem. Write down pseudocode, or code in your favorite PL.

(e) Explain why the algorithm you design in (d) has running time $\mathcal{O}(n^2)$.

(f) (*) Design an $\mathcal{O}(n)$-time algorithm for the LAS problem. Explain why your algorithm is correct.

Solution

(a) $R[n-1] = D[n-1] = 1$.

(b) The key "optimal sub-structure" observation is given a longest initially-raising alternating subsequence that starts with $A[i]$, after removing its first number $A[i]$, the remaining subsequence must be a longest initially-dropping alternating subsequence that starts with $A[j]$ for some $j > i$. Also, by the definition of initially-raising, we have $A[j] > A[i]$. Thus we have the recurrence.

(c) Let $K = \{k \mid k > i \text{ and } A[k] < A[i]\}$. Then

$$D[i] \;=\; \begin{cases} 1 + \max_{k \in K} R[k], & \text{if } K \text{ is not empty;} \\ 1, & \text{if } K \text{ is empty.} \end{cases}$$

(d) After initializing $R[n-1] = D[n-1] = 1$, for i from $n-2$ down to 0, compute $R[i]$ and $D[i]$ using the recurrence in (b) and (c). And then the length of LAS is simply $\max_{0 \le i \le n-1} \max\{R[i], D[i]\}$.

(e) For each i from $n-2$ down to 0, computing $R[i]$ amounts to using a for loop of $n-i$ iterations to iterate over all $D[j]$ for j specified in (b). Analogously is true for computing $D[i]$. Thus the runtime is $\mathcal{O}(n) \cdot \mathcal{O}(n) = \mathcal{O}(n^2)$, where the first $\mathcal{O}(n)$ accounts for the iterations over i, and the second $\mathcal{O}(n)$ accounts for the iterations over j.

3. (*) As noted in lecture, LIS problem admits a faster $\mathcal{O}(n \log n)$-time algorithm. In this question, we will help to understand the key ideas behind the improved running time.

(a) Suppose $A = [x, 7, 10, 3, 15, 5, 9]$, where x is an integer and its value is not yet revealed. List all LIS in $[7, 10, 3, 15, 5, 9]$.

(b) Among the LIS listed in (a), suppose you can only save one of them for latter use (i.e., after x is revealed), and you have to forget the other LIS. Which LIS will you save? Why?

(c) Let $A'[i]$ denote the tail of an array A from its i-th position till its end. For instance, if $A = [4, 7, 10, 3, 11, 5, 9]$, then $A'[2] = [10, 3, 11, 5, 9]$.

A key notion for the $\mathcal{O}(n \log n)$-time algorithm is $B[i, k]$, which is the "best" length-k increasing subsequence in $A'[i]$. We have not formally described what is meant by the "best", but if you have done (b) correctly you should understand what we mean.

When $A = [4, 7, 10, 3, 11, 5, 9]$, what are $B[1, 1]$, $B[1, 2]$ and $B[1, 3]$?

(d) For any i, let $H[i, k]$ denote the first number of $B[i, k]$. Explain why $H[i, 1] \geq H[i, 2] \geq H[i, 3] \geq \ldots$.

(e) Hopefully, you may now have some ideas on how the $\mathcal{O}(n \log n)$-time algorithm works, via suitable binary search and update of $H[i, 1], H[i, 2], \ldots$. Write down pseudocode, or code in your favorite PL, that implements the $\mathcal{O}(n \log n)$-time DP algorithm.

Exercise 2 — *Game Dynamic Programming*

4. *Grasshopper game* is a two-player game specified by a positive integer n and a set S of positive integers. Alice and Bob face each other on a rod, and they are separated by a distance of $(n + 0.5)$ units. Each player takes term to jump towards the other player for s units, where s must be chosen from S. The first player that jumps *across* the other player wins the game.

   For instance, suppose $n = 5$ and $S = \{1, 2, 4\}$. Alice makes the first move by jumping 1 unit towards Bob. Then Bob jumps 4 units toward Alice. Now Alice and Bob are separated by a distance of 0.5 unit, so Alice can win the game by jumping 1 unit towards Bob.

   Following the ideas behind the two-player jump game and two-player coin game discussed in lecture, design a DP algorithm that, given n and S, determines which player will win the game, assuming both players are clever.

   Solution

   Let $B[i]$ be a boolean variable which is true if the two players are separated by $(i + 0.5)$ units, and the player that makes the next move will win, assuming both players are clever.

   We first cover the boundary cases. For $i = 0, 1, 2, \ldots, \max(S) - 1$, we can easily deduce that $B[i]$ is true, because the next player can choose to jump $\max(S)$ units so as to jump across the other player.

   For $i \geq \max(S)$, following the logic we discuss in the lecture, we have

   $$B[i] \;=\; \vee_{s \in S} \left(\neg B[i - s]\right) ,$$

   where $\vee$ means "or" and $\neg$ means "not" (hope you still remember these from your year-1 courses...). This provides sufficient ingredients for you to implement the DP algorithm.

5. A *Nimux game* is a two-player game specified by four positive integers n_1, n_2, n_3, k. At the beginning of the game, there are three piles of coins, where the i-th pile has n_i coins. Alice and Bob take term to make moves. A valid move can be one of the following two types:

   - Choose one non-empty pile, take at least one but at most k coins from the pile.
   - Choose two non-empty piles, and take one coin from each of the two piles.

   The player who is the first to fail to make a valid move loses the game.

   Design a DP algorithm that, given n_1, n_2, n_3, k, determines which player will win the game, assuming both players are clever. Analyze the runtime of the algorithm.

Solution

We use a three-dimensional vector $[n_1, n_2, n_3]$ to represent the state that the i-th pile has n_i coins, for $i = 1, 2, 3$. After fixing k, let $B[n_1, n_2, n_3]$ be a boolean variable which is true if starting from state $[n_1, n_2, n_3]$, the player that makes the next move will win, assuming both players are clever. The boundary case is $B[0, 0, 0] = \text{false}$.

At $[n_1, n_2, n_3]$, the next player can move to the following states:

(i) $[q, n_2, n_3]$ for $\max\{0, n_1 - k\} \leq q \leq n_1 - 1$;

(ii) $[n_1, q, n_3]$ for $\max\{0, n_2 - k\} \leq q \leq n_2 - 1$;

(iii) $[n_1, n_2, q]$ for $\max\{0, n_3 - k\} \leq q \leq n_3 - 1$;

(iv) if $n_1, n_2 \geq 1$, $[n_1 - 1, n_2 - 1, n_3]$;

(v) if $n_1, n_3 \geq 1$, $[n_1 - 1, n_2, n_3 - 1]$;

(vi) if $n_2, n_3 \geq 1$, $[n_1, n_2 - 1, n_3 - 1]$.

Following the logic we discuss in the lecture, $B[n_1, n_2, n_3] = \vee(\neg B[\bullet, \bullet, \bullet])$, where $[\bullet, \bullet, \bullet]$ covers all possible states listed above.

Notice that after each move, the total number of coins in the three piles drop by at least 1. This suggests that when performing DP iterations, we should start with states with small number of coins, and gradually compute for states with larger number of coins. The code is quite long, so I have implemented it in a separate nimux.py file, uploaded to Wattle.

For runtime analysis, note that there are $(n_1 + 1)(n_2 + 1)(n_3 + 1) = \mathcal{O}(n_1 n_2 n_3)$ states for which we compute the corresponding value in B. And for each state, there are at most k possible next states in each of cases (i), (ii) and (iii), and at most 1 possible next state in each of cases (iv), (v) and (vi). Overall, the runtime is at most $\mathcal{O}(n_1 n_2 n_3) \cdot (3k + 3) = \mathcal{O}(n_1 n_2 n_3 k)$.

Exercise 3 *Sum Dynamic Programming*

6. In the lecture, we discuss a DP algorithm for solving the subset sum problem. Now we consider a variant of the subset sum problem: given a set $S = \{s_0, s_1, \ldots, s_{k-1}\}$ of positive integers, and two positive integers n and j, determine if there exists a subset of S with size at most j, such that the sum of the subset is exactly n. Let Q be the sum of all integers in S. Define $C[i, q]$ as follows.

   - If there does not exist a subset of $\{s_0, s_1, \ldots, s_i\}$ whose sum is exactly equal to q, then $C[i, q] = +\infty$.
   - If there exists a subset of $\{s_0, s_1, \ldots, s_i\}$ whose sum is exactly equal to q, then $C[i, q]$ is the size of the smallest such subset.

   (a) What are the values of $C[0, q]$ for $0 \leq q \leq Q$?

   (b) Derive a recurrence of $C[i, q]$ for $1 \leq i \leq k - 1$ and $0 \leq q \leq Q$.

   (c) Write down pseudocode, or code in your favorite PL, that implements a DP algorithm for the problem. Analyze the runtime of the algorithm.

Solution

(a) $C[0, 0] = 0$, $C[0, s_0] = 1$, and for any other q, $C[0, q] = +\infty$.

(b) Similar to the logic that we compute $B[i, q]$ for the original subset sum problem, we have the following recurrence:

$$C[i, q] \;=\; \begin{cases} C[i-1, q], & \text{if } q < s_i; \\ \min(C[i-1, q], C[i-1, q - s_i] + 1), & \text{if } q \geq s_i. \end{cases}$$

(c) By using (a) and (b), you should now be capable to implement the DP algorithm. The runtime is $\mathcal{O}(|S|Q)$, by essentially the same analysis as for the DP algorithm of original subset sum problem.

Exercise 4 ***Bonus exercise***

(*) Log into leetcode and attempt the following problem related to dynamic programming:

1. Cat and Mouse