LeetCode Notes

Jason Sun

September 17, 2020

Contents

Part I.
LeetCode

LeetCode 27. Remove Element

Two Pointer Regular two pointer approach. The implementation is routine. Code

LeetCode 53. Maximum Subarray

Dynamic Programming Let $f(i)$ denote the maximum subarray sum ending with `nums[i]`,, then we have

$$f(i) = \begin{cases} \texttt{nums[0]} & \text{if } i = 0 \\ \max(f(i-1) + \texttt{nums[i]}, \texttt{nums[i]}) & \text{otherwise} \end{cases}. \tag{2.1}$$

Then the final answer is $\max_{i=0}^{n-1} f(i)$, where $n = \texttt{std::size}(\texttt{nums})$. Code

LeetCode 55. Jump Game

Dynamic Programming 1 Let $f(i)$ denote the reachability of index i. Then it follows that

$$f(i) = \begin{cases} \text{true} & \text{if } i = 0 \\ \text{true} & \text{if there exists } k \in [1, i-1] \text{ such that } f(k) = \text{true and } \texttt{nums}[k] + k \geq i \\ \text{false} & \text{otherwise.} \end{cases}. \tag{3.1}$$

The final answer is $f(i)$. The overall runtime complexity is $O(n^2)$ and memory complexity is $O(n)$. Code

Dynamic Programming 2 Let $f(i)$ denote the maximum distance that can be achieved from index i. Then we have

$$f(i) = \begin{cases} \texttt{nums}[0] & \text{if } i = 0 \\ \max(f(i-1), \texttt{nums}[i] + i) & \text{else if } f(i-1) \geq i \\ f(i-1) & \text{otherwise} \end{cases}. \tag{3.2}$$

The final answer is $\max_{i=0}^{n-1} f(i) \geq n-1$. The runtime complexity is $O(n)$ and memory complexity is $O(n)$. Code

LeetCode 66. Plus One

Simulation (Recursive) We use a stateful recursive lambda, $f_{[\texttt{carry},\texttt{D}]}(i)$, to recursively simulate the addition process, where $f(i)$ denote the process at index i, `carry` represents if a carry of one will be added in the current digit, and `D` is a mutable copy of `digits`. We start from $i = n-1$, where $n = \texttt{std::size(digits)}$. The implementation of f is routine. Code

LeetCode 88. Merge Sorted Array

Simulation (Recursive) We implement a function $f(i, n_1, n_2)$ to recursively fill `nums1` backward putting $\texttt{nums1[i]} = \max(\texttt{nums1}[n_1], \texttt{nums2}[n_2])$. The implementation is routine. The resulting runtime complexity is $O(n+m)$ and memory complexity is $O(1)$. Code

LeetCode 265. Paint House II

Dynamic Programming 1 Let $f(i, j)$ denote the minimum cost to paint all houses per the specification. Then we have

$$f(i,j) = \begin{cases} \texttt{costs}[0][j] & \text{if } i = 0 \\ \min_{k \in [0,k-1] \text{ and } k \neq j} \{\texttt{costs}[i][j] + f(i-1,k)\} & \text{o.w.} \end{cases}. \tag{6.1}$$

The time complexity if $O(nk^2)$.

Dynamic Programming 2 Let $f(i, j)$ denote the minimum cost to paint all houses per the specification. And $g(i) = \{\texttt{kth}(0, f([i][0, ..., k-1]), \texttt{kth}(1, f([i][0, ..., k-1])\}$. Then it follows that

$$f(i,j) = \begin{cases} \texttt{costs}[0][j] & \text{if } i = 0, \\ f(i, g(i-1)[1].\texttt{index}) + \texttt{costs}[i][j] & \text{else if } g(i-1)[0].\texttt{index} = j \\ f(i, g(i-1)[0].\texttt{index}) + \texttt{costs}[i][j] & \text{o.w.} \end{cases}. \tag{6.2}$$

The time complexity is $O(nk)$.

LeetCode 276. Paint Fence

Dynamic Programming Let $f(i)$ denote the total numbers of way to paint fences $[0, ..., i]$. Then we have

$$f(i) = \begin{cases} k & \text{if } i = 0 \\ k^2 & \text{else if } i = 1 \\ f(i-1) \times (k-1) + f(i-2) \times (k-1) & \text{o.w.} \end{cases}, \tag{7.1}$$

where the two parts in the last transition function is to tackle with the two cases where the ith block has the same color as the $(i-1)$th, in which case it must be difference from the $(i-2)$th fence, and different color as the $(i-1)$th fence.

LeetCode 300. Longest Increasing Subsequence

Patience Sorting There is a connection between the game of patience and and problem of LIS. The minimum number of piles formed in the game of patience is equal to the length of LIS. For more details, see Section A.1. The time complexity is $O(n \log n)$.

Dynamic Programming Let $f(i)$ denote the length of the LIS ending at `nums`$[i]$. Then it follows that

$$f(i) = \begin{cases} \max_{0 \leq k \leq i-1 \text{ and } \texttt{nums}[k] < \texttt{nums}[i]} \{1 + f(k)\} & \text{if } i > 0 \\ 1 & \text{o.w.} \end{cases}. \tag{8.1}$$

Then the final answer is $\max_{0 \leq i \leq n-1} f(i)$. The overall run time complexity is $O(n^2)$.

LeetCode 312. Burst Balloons

Dynamic Programming Let $f(i, j)$ denote the maximum coins after bursting `nums`$[i:j]$. Then we have that

$$f(i,j) = \begin{cases} \max_{i \leq k \leq k} \{\texttt{nums}[k] \cdot \texttt{nums}[i-1] \cdot \texttt{nums}[j+1] + f(i, k-1) + f(k+1, j)\} & \text{if } i < j \\ \texttt{nums}[i] \cdot \texttt{nums}[i-1] \cdot \texttt{nums}[i+1] & \text{else if } i = j \\ 0 & \text{else if } i > j \end{cases}. \tag{9.1}$$

The final answer is then $f(0, n-1)$, where $n = \texttt{std::size}(\texttt{nums})$. Note that `nums`$[-1]$ and `nums`$[n]$ is needs to be dealt with in the implementation if we don't pad `nums`. The overall running time is $O(n^2)$.

LeetCode 321. Create Maximum Number

Dynamic Programming with Greedy and Monotone Stack We can decompose this problem into three parts. The final solution of the problem is

$$\min_{(i,j) \in [0,...,n_1] \times [0,...,n_2] \text{ and } i+j=k} \texttt{merge}(g(\texttt{nums1}, i), g(\texttt{nums2}, j)). \tag{10.1}$$

There $g(A, k)$ returns the largest k element subsequence in A. And $\texttt{merge}(A_1, A_2)$ returns the largest sequentially merged array from A_1 and A_2 in term of lexicographical ordering. Note that $g(A, k)$ can be computed efficiently

using a monotone stack, see Section B.1.1. And `merge` can be implemented using a greedy idea: we with $i = 0$ and $j = 0$ and an accumulator `acc` of result. If $A_1[i:-1] \geq A_2[j:-1]$, we accumulate $A_1[i]$ into `acc` and increment i, otherwise we merge $A_2[j]$ into `acc` and increment j. The correctness of this merging algorithm can be proved using an exchange argument. The overall time complexity of this is $O(\max(n_1, n_2)^2 k)$.

LeetCode 322. Coin Change

Dynamic Programming Let $f(S)$ denote the minimum number of coins needed to reach value S. Then we have f

$$f(S) = \begin{cases} \min_{c \in \texttt{coins} \text{ and } S-c \geq 0}\{1 + f(S-c)\} & \text{if } S > 0 \\ 0 & \text{else if } S = 0 \end{cases}, \tag{11.1}$$

where we let $\min(\emptyset) := +\infty$ to handle the infeasible cases. The time complexity is $O(nS)$, where $n = \texttt{std::size}(\texttt{coins})$.

LeetCode 343. Integer Break

Dynamic Programming Let $f(i)$ denote the maximum product for number i. Then we have f

$$f(i) = \begin{cases} 1 & \text{if } i = 1 \\ \max_{1 \leq k \leq i-1}(k \cdot f(i-k), k \cdot (i-k)) & \text{o.w.} \end{cases}. \tag{12.1}$$

The final answer if then $f(n)$. The overall runtime complexity is $O(n^2)$.

LeetCode 351. Android Unlock Patterns

Backtrack

Dynamic Programming

LeetCode 624. Maximum Distance in Arrays

Prefix Max/Min First, we note that if we drop the requirement that the two elements has to be from distinct arrays then the maximum distance has to be the $\max(\texttt{flatten}(\texttt{arrays})) - \min(\texttt{flatten}(\texttt{arrays}))$. On the other hand, if we require the two elements to be from distinct arrays, then we have to exclude the some cases. One thing that is invariant is that the at least one of $\max(\texttt{flatten}(\texttt{arrays}))$ and $\min(\texttt{flatten}(\texttt{arrays}))$ will be involved in the resulting optimal distance, which one could prove using an exchange argument. So let $f(i)$ denote the maximum distance produced with one participating element in $\texttt{array}[i]$, then

$$f(i) = \begin{cases} \max(P_{\min}(i), P_{\max}(i), S_{\min}(i), S_{\max}(i)) & \text{if } 0 < i < n-1 \\ \max(S_{\min}(i), S_{\max}(i)) & \text{else if } i = 0 \\ \max(P_{\min}(i), P_{\max}(i)) & \text{else if } i = n-1 \end{cases}, \tag{14.1}$$

where as

$$P_{\min}(i) = |\max(\texttt{array}[i]) - \min(\texttt{flatten}(\texttt{array}[0:i-1]))|, \tag{14.2}$$
$$P_{\max}(i) = |\min(\texttt{array}[i]) - \max(\texttt{flatten}(\texttt{array}[0:i-1]))|, \tag{14.3}$$
$$S_{\max}(i) = |\max(\texttt{array}[i]) - \min(\texttt{flatten}(\texttt{array}[i+1:n-1]))|, \tag{14.4}$$
$$S_{\min}(i) = |\min(\texttt{array}[i]) - \max(\texttt{flatten}(\texttt{array}[i+1:n-1]))|, \tag{14.5}$$

which can be computed efficiently $O(n)$ where $n = \texttt{std::size}(\texttt{arrays})$ using prefix min and max arrays and the sorted structure of `arrays`. Code

LeetCode 723. Candy Crush

Simulation with Two Pointers Use two pointer method to mark consecutive entries in each row and each column. To do so, one can either maintain a separate matrix of boolean flags indicating whether the corresponding entry in `board` is to be deleted or negate the value in `board` directly as a flag. In our implementation, we choose the latter approach and the corresponding procedures are encapsulated in `mark_row()` and `mark_col()`. The crush procedure is implemented in `crush()`, which is routine. We will also implement a function `drop()` to simulate the gravity phenomena, which also uses a two pointer approach. Alternatively, one could also explicitly use a double ended queue to filter out the crushed candies column by column and reinsert the filtered result back into `board` starting from the bottom of each column. Code

LeetCode 746. Min Cost Climbing Stairs

Dynamic Programming Let $f(i)$ denote the minimum cost to climb to level i. Then we have n

$$f(i) = \begin{cases} \min(f(i-1) + \texttt{cost}[i-1], f(i-2) + \texttt{cost}[i-2] & \text{if } i > 1 \\ 0 & \text{else if } i \in \{0, 1\} \end{cases}. \tag{16.1}$$

Then the desired answer is then $f(n)$, where $n = \texttt{std::size}(\texttt{cost})$. Code

LeetCode 983. Minimum Cost For Tickets

Dynamic Programming Let $f(i)$ denote the minimum costs to cover $\texttt{days}[0:i]$. Then we have

$$f(i) = \begin{cases} \max(\texttt{costs}[0], \texttt{costs}[1], \texttt{costs}[2]) & \text{if } i = 0 \\ \max(\texttt{costs}[i] + f(\texttt{next\_lower}(\texttt{days}[0:i-1], \texttt{days}[i] - \texttt{pass}[i])) & \text{else if } i \neq \emptyset, i > 0 \\ 0 & \text{else if } i = \emptyset \end{cases}, \tag{17.1}$$

where $\texttt{next\_lower}(A, x)$ return last element in array A that is strictly less than x; we note that this procedure can be implemented in $O(\log n)$ times using binary search. Thus, the overall time complexity is $O(n \log n)$, where $n = \texttt{std::size}(\texttt{days})$.

LeetCode 1346. Check if N and Its Double Exist

HashSet Create a hashset `arr_set` to store non-zero unique elements in `arr`. Then we count the number of zeros in `arr`, if there are at least two zeros, we return `true` immediately; otherwise we loop through `arr_set` check if the condition is met. Code

LeetCode 1351. Count Negative Numbers in a Sorted Matrix

Binary Search Let $[R, C] = \texttt{dimension}(\texttt{grid})$. We count row by row. If $\texttt{grid}[r][0] < 0$ for some $r \in [0, R-1]$, then we add all the items in the submatrix $\texttt{grid}[r:R-1][0:C-1]$ to the final result; otherwise, we will use binary search to find the first negative entry c_N in each row and add to the final result $C - c_N$. The count process could be implemented as a function $f(r)$, where $f(i)$ means the number of negative entries when counting from row i. Then a

$$f(r) = \begin{cases} C - c_N + f(r+1) & \text{if } \min(\texttt{grid}[r]) < 0 \\ 0 & \text{else if } \min(\texttt{grid}[r]) > 0 \text{ or } r = R \\ (R-r) \cdot C & \text{else if } \texttt{grid}[r][0] < 0 \end{cases}. \tag{19.1}$$

And the desired answer is $f(0)$. Code

LeetCode 1365. How Many Numbers Are Smaller Than the Current Number

Bucket Sort with Prefix Sum Create a bucket array, `bucket`, to store all the values in `nums` and then create range query module of `bucket` using prefix sum, which we name as `RSQ`. Let A denote the desired array, then we have

$$A[i] = \begin{cases} 0 & \text{if } \texttt{nums}[i] = 0 \\ \texttt{RSQ}(0, \texttt{nums}[i] - 1) & \text{otherwise} \end{cases}. \tag{20.1}$$

The overall runtime complexity is $O(n)$ and memory complexity is $O(n)$. Code

LeetCode 1380. Lucky Numbers in a Matrix

Implementation Create two arrays, `row_min` and `col_max`, to store the minimum (maximum) of each row (column). Then we iterate through every entries `matrix` to check for lucky numbers. Code

LeetCode 1385. Find the Distance Value Between Two Arrays

Binary Search We first let `sorted_arr2` be a sorted copy of `arr2`. Then for each $i \in [0, n_1 - 1]$, where $n_2 =$ `std::size(arr1)`, we use binary search to locate l, r such that `sorted_arr2`$[l:r] \subseteq$ [`arr1`$[i] - d$, `arr1`$[i] + d$] and add the element count of `sorted_arr2`$[l:r]$ to the final result. Code

LeetCode 1389. Create Target Array in the Given Order

Simulation Just simply create a `std::vector<int>` and simulate the process using `insert`. Code

LeetCode 1394. Find Lucky Integer in an Array

Implementation Create a hashmap `freq_map` to count the occurrences of each number in `arr`. Then filter out the keys in `freq_map` that satisfies the condition into a list, and return the maximum the list if the list is non-empty or -1 if otherwise. Code

Appendices

A. Sorting Algorithm

A.1. Patient Sorting

Consider the following game: deal cards $c_1, c_2, ..., c_n$ into piles according to two rules

- Can't place a higher valued card onto a lower valued card;
- Can form a new pile and put a card onto it.

And the goal is to form as few piles as possible.

Greedy Algorithm In the natural order, place each card on the leftmost pile that fits. Note that by construction, at any stage during the greedy algorithm, top cards of piles increase from left to right.

A.1.1. Connection with LIS

Lemma A.1 (Weak Duality)**.** *In any legal game of the patience, the number of piles is larger than or equal to any increasing subsequence.*

Proof. Note that cards within a pile form a decreasing subsequence and any sequence can use at most one card from each pile. □

Lemma A.2 (Strong Duality)**.** *Min number of piles is equal to LIST. More over, the greedy algorithm finds both.*

Proof. Each card maintains a pointer to top card in previous pile. The we can follow pointers to obtain IS whose length is equal to the number of piles. The by the weak duality lemma above, the sequence if one member of the optimal solution set of the LIS problem. □

Implementations This patient sorting algorithm could be implemented in $O(n \log n)$ running time by using an array of stack to represent the piles and binary search to bind the left most pile.

B. Elementary Data Structures

B.1. Monotone Stack

B.1.1. Find k -element subsequence that is lexicographically largest

B.1.2. Find [previous/next] [greater/smaller] element