

# 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

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

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

[Code](#)

### LeetCode 55. Jump Game

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

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

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

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

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,  $f_{[\text{carry}, D]}(i)$ , 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  $n = \text{std::size}(\text{digits})$ . 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  $\text{nums1}[i] = \max(\text{nums1}[n_1], \text{nums2}[n_2])$ . The implementation is routine. The resulting runtime complexity is  $O(n + m)$  and memory complexity is  $O(1)$ .

Code

## LeetCode 233. Number of Digit One

## 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

$$f(i, j) = \begin{cases} \text{costs}[0][j] & \text{if } i = 0 \\ \min_{k \in [0, k-1] \text{ and } k \neq j} \{ \text{costs}[i][j] + f(i-1, k) \} & \text{o.w.} \end{cases} \quad (7.1)$$

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

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

$$f(i, j) = \begin{cases} \text{costs}[0][j] & \text{if } i = 0, \\ f(i, g(i-1)[1].\text{index}) + \text{costs}[i][j] & \text{else if } g(i-1)[0].\text{index} = j. \\ f(i, g(i-1)[0].\text{index}) + \text{costs}[i][j] & \text{o.w.} \end{cases} \quad (7.2)$$

The time complexity is  $O(nk)$ .

## LeetCode 276. Paint Fence

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

$$f(i) = \begin{cases} k & \text{if } i = 0 \\ k^2 & \text{else if } i = 1, \\ f(i-1) \times (k-1) + f(i-2) \times (k-1) & \text{o.w.} \end{cases} \quad (8.1)$$

where the two parts in the last transition function is to tackle with the two cases where the  $i$ th 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 ???. 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

$$f(i) = \begin{cases} \max_{0 \leq k \leq i-1 \text{ and } nums[k] < nums[i]} \{1 + f(k)\} & \text{if } i > 0 \\ 1 & \text{o.w.} \end{cases} \quad (9.1)$$

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

$$f(i, j) = \begin{cases} \max_{i \leq k \leq j} \{nums[k] \cdot nums[i-1] \cdot nums[j+1] + f(i, k-1) + f(k+1, j)\} & \text{if } i < j \\ nums[i] \cdot nums[i-1] \cdot nums[i+1] & \text{else if } i = j \\ 0 & \text{else if } i > j \end{cases} \quad (10.1)$$

The final answer is then  $f(0, n-1)$ , where  $n = std::size(nums)$ . 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

$$\min_{(i,j) \in [0, \dots, n_1] \times [0, \dots, n_2] \text{ and } i+j=k} \text{merge}(g(\text{nums1}, i), g(\text{nums2}, j)). \quad (11.1)$$

There  $g(A, k)$  returns the largest  $k$  element subsequence in  $A$ . And  $\text{merge}(A_1, A_2)$  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 ???. And  $\text{merge}$  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

$$f(S) = \begin{cases} \min_{c \in \text{coins} \text{ and } S-c \geq 0} \{1 + f(S - c)\} & \text{if } S > 0 \\ 0 & \text{else if } S = 0 \end{cases}, \quad (12.1)$$

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

## LeetCode 338. Counting Bits

**Dynamic Programming** Let  $f(i)$  be the number of 1's in  $i$ 's binary representation. Then we have that

$$f(i) = \begin{cases} 0 & \text{if } i = 0 \\ 1 & \text{else if } i = 1 \\ f(i/2) & \text{else if } i \text{ is even} \\ f(i/2) + 1 & \text{else if } i \text{ is odd} \end{cases} \quad (13.1)$$

## LeetCode 343. Integer Break

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

$$f(i) = \begin{cases} 1 & \text{if } i = 1 \\ \max_{1 \leq k \leq i-1} (k \cdot f(i-k), k \cdot (i-k)) & \text{if } i \geq 2 \end{cases} \quad (14.1)$$

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

## LeetCode 351. Android Unlock Patterns

**Backtrack** Let  $f(i, k)$  denote the number of possible patterns with length  $k$  that starts from  $i$ . Then the final answer is  $\sum_{i=1}^9 \sum_{k=1}^n f(i, k)$ . We note that  $f(i, k)$  can be computed using exhaustive search (backtrack). We will also create a list *blocks* to keep track of the jumping across cases for every key; for example,  $blocks[1][\{3, 9, 7\}] = \{2, 5, 4\}$  and  $blocks[1][k] = \emptyset$  for all other  $k \in [1, \dots, 9] \setminus \{3, 9, 7\}$ .

**Dynamic Programming** Using the similar idea as before, we note that  $f$  can actually be memoized. Let  $f(S, k, l)$  denote the number of possible patterns of length  $l$  starting from  $k$ .

## LeetCode 354. Russian Doll Envelopes

**Patience Sorting** Convert this problem into a LIS problem on the last coordinate. Note that LIS problem can be solved using the patience sorting algorithm in  $O(n \log n)$  run time, see ??.

## LeetCode 357. Count Numbers with Unique Digits

**Dynamic Programming** Let  $f(i)$  denote the counts of numbers with unique digits such that  $0 \leq x \leq 10^n$ . Then we have that

$$f(i) = \begin{cases} 10 & \text{if } i = 1 \\ 9 \cdot 9 & \text{else if } i = 2 \\ f(i-1) \cdot (10-i+1) & \text{else if } i > 2 \end{cases} \quad (17.1)$$

Then the final answer is then  $\sum_{i=1}^n f(i)$ . We note that the special case of  $n = 0$  is to be dealt with separately as a special case.

## LeetCode 361. Bomb Enemy

**Dynamic Programming** Let  $f(i, j, d)$  denote the number of enemies that can be killed in direction  $d$  if the bomb is dropped in  $grid[i][j]$ . Then we have

$$f(i, j, d) = \begin{cases} 0 & \text{if } grid[i][j] = 'E' \\ 1 + f(i + dr[d], j + dc[d], d) & \text{else if } grid[i][j] = 'E' \\ f(i + dr[d], j + dc[d], d) & \text{else if } grid[i][j] = 0 \end{cases} \quad (18.1)$$

Then the final solution is  $\max_{i,j} \sum_d f(i, j, d)$ .

## LeetCode 368. Largest Divisible Subset

**Dynamic Programming** First we sort the  $nums$ . Let  $f(i)$  denote the maximum divisible set size ending with  $nums[i]$ . Then it follows that

$$f(i) = \begin{cases} 1 & \text{if } i = 0 \\ \max_{0 \leq j \leq i-1 \text{ and } nums[i] \% nums[j] == 0} \{f(j) + 1\} & \text{o.w} \end{cases} \quad (19.1)$$

Then we can use backtrace to recursively build up the answer.

## LeetCode 375. Guess Number Higher or Lower II

**Dynamic Programming** Let  $f(i, j)$  denote the minimum number of amount one need a guarantee a win for number to be picked in range  $[i, j]$ . Then we have that

$$f(i, j) = \begin{cases} 0 & \text{if } i = j \\ k + \min_{i \leq k \leq j} \{f(i, k-1) + f(k+1, j)\} & \text{else if } i < j \end{cases} \quad (20.1)$$

The final answer is then  $f(1, n)$ . The overall run time complexity is  $O(n^2)$ .

## LeetCode 392. Is Subsequence

**Greedy with two pointer** Just match  $s$  with  $t$  from the beginning.

## LeetCode 376. Wiggle Subsequence

**Dynamic Programming** Let  $f(i)$  denote the length of the longest wiggle subsequence ending with  $nums[i]$  and the difference between last two elements is positive and  $g(i)$  denote the length of the longest wiggle subsequence ending with  $nums[i]$  and the difference between the last two elements is negative. Then we have

$$f(i) = \begin{cases} 1 & \text{if } i = 0 \\ \max_{0 \leq j < i, nums[i] < nums[j]} \{1 + g(j)\} & \text{o.w.} \end{cases}, \quad (22.1)$$

and

$$g(i) = \begin{cases} 1 & \text{if } i = 0 \\ \max_{0 \leq j < i, nums[i] > nums[j]} \{1 + f(j)\} & \text{o.w.} \end{cases}. \quad (22.2)$$

Then the final answer is  $\max_{0 \leq i \leq n-1} \{\max(f(i), g(i))\}$ . And the overall running time complexity is  $O(n^2)$ .

## LeetCode 403. Frog Jump

**Dynamic Programming** Let  $f(x, y)$  denote the reachability to  $x$  with step  $y$ . Then it follows that

$$f(x, y) = \begin{cases} true & \text{if } x = 1, y = 1 \\ f(x - y, y) \vee f(x - y, y - 1) \vee f(x - y, y + 1) & \text{o.w.} \end{cases}. \quad (23.1)$$

Since there is not limit on steps, we have that  $\bigvee_{i=1}^{1001} f(stones[-1], i)$ .

## LeetCode 410. Split Array Largest Sum

**Dynamic Programming** Let  $f(i, j)$  denote the largest sum splitting array  $nums[0 : i]$  into  $j$  parts. Then we have

$$f(i, j) = \begin{cases} RMQ(0, i) & \text{if } j = 1 \\ \min_{j-1 \leq k \leq i} \{RMQ(k, i) + f(k-1, j-1)\} & \text{o.w.} \end{cases} \quad (24.1)$$

The final answer is  $f(n-1, m)$ . The overall runtime complexity is  $O(n^2m)$ .

**Binary Search** Let  $f(x)$  denote the minimum number of contiguous subarrays that can be splitted with sum less than or equal to  $x$ . Then we note that  $f$  is a decreasing function. The desired answer is then  $\sup\{x : f(x) \leq m\}$ . This can be queried using binary search on  $f$  on range  $[\max_i \{nums[i]\}, \sum_i nums[i]]$ . The runtime complexity is  $O(n \log n)$ .

## LeetCode 413. Arithmetic Slices

**Dynamic Programming** Let  $f(i)$  denote the number arithmetic slices ending with  $A[i]$ . Then we have

$$f(i) = \begin{cases} 0 & \text{if } i = 0 \\ 1 + f(i-1) & \text{else if } A[i] - A[i-1] = A[i-1] - A[i-2] \\ 0 & \text{else if } A[i] - A[i-1] \neq A[i-1] - A[i-2] \end{cases} \quad (25.1)$$

The final answer is  $\sum_{i=0}^{n-1} f(i)$ , where  $n = \text{std}::\text{size}(A)$ .

## LeetCode 418. Sentence Screen Fitting

**Memoization** We design a function  $fill(i)$  which fills a row starting from  $sentence[i]$  and stop until this row cannot be filled. It returns the number of words filled and the word next to the last filled word. Note that the sentences can be filled multiple times in one row if the column size is sufficiently large. Then we fill the matrix row by row and accumulate the total number of words filled, which we denote as  $acc\_cnt$ . The final result is then  $acc\_cnt/n$ , where  $n = (sentence)$ . To speed things up, we note that  $fill$  is a pure function which does not depend on other states. Therefore, we can memoize it to save unnecessary computation.

## LeetCode 446. Arithmetic Slices II

**Dynamic Programming** Let  $f(i, d)$  denote the number of arithmetic slices ending with  $A[i]$  with difference  $d$ . Then we have

$$f(i, d) = \begin{cases} \sum_{0 \leq j < i, A[i] - A[j] = d} f(j, d) + 1 & \text{if } i > 1 \\ 1 & \text{else if } i = 1 \text{ and } A[i] - d = A[i-1] \\ 0 & \text{else if } i = 1 \text{ and } A[i] - d \neq A[i-1] \end{cases} \quad (27.1)$$

We need to note that these sequences contains arithmetic sequences with length 2, which doesn't satisfy the requirement given in the problem. Hence we need to subtract them off. There are two ways to subtract them:

1. Let  $x$  denote the accumulated sum for all  $i$  and  $d$ . Then the final answer is  $x - n(n-1)/2$ , where  $n = (A)$ .
2. Let  $x$  denote the final answer. Then  $x = \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} f(j, A[i] - A[j])$ . By doing so, we are excluding out the two element sequence since we are starting from  $j$  instead of  $i$ . By doing so we are guaranteeing the first element any arithmetic slices therefore guaranteeing  $x$  counts all the arithmetic slices with at least three elements.

We also note that there some int max and int min issues to be dealt with separately. Also, to speed things up, we use a *value\_index\_map* to keep track of all the indices.



## LeetCode 459. Repeated Substring Pattern

**String Prefix Function** To solve this, we note the following proposition:

**Proposition 28.1.** *Let  $s$  be a string. If  $s$  can be constructed by concatenating multiples of its proper substrings, then  $n|(n_s - \pi(n_s - 1)) = 0$ , where  $\pi$  is the string prefix function.*

*Proof.* Suppose  $s = [t]_k$ , where  $t$  is a proper substring of  $s$  and  $k \geq 2$ . Without loss of generality, we assume that  $t$  is the minimal proper substring that has the property. We let  $n_t = (t)$ . Then it follows that  $\pi(n_s - 1) = (k - 1) \cdot n_t$ . Then it follows that  $n - \pi(n_s - 1) = n_t$ . which is divisible by  $n_s = kn_t$ . On the other hand, suppose  $s$  cannot be written in the form of  $[t]_k$ .  $\square$

With this in mind, it suffices to compute the prefix function,  $\pi$ , and check.

## LeetCode 464. Can I Win

**Dynamic Programming with BitSet** Let  $f(S, x)$  denote the possibility to guarantee a win for the first player with remaining available number set  $S$  and target  $x$ . Then we have that

$$f(S, x) = \begin{cases} false & \text{if } S = \emptyset \text{ or } \sum_{x \in S} S < x \\ (\max(S) \geq x) \text{ or } \bigvee_{x \in S} (not f(S \setminus \{x\} - x - i)) & \text{else} \end{cases}. \quad (29.1)$$

The final answer is then  $f([1, \dots, n], desiredTotal)$ . We note that we don't need to memoize both  $S$  and  $x$  since once we know  $S$ ,  $x$  can be automatically calculated.

## LeetCode 466. Count The Repetitions

**Dynamic Programming with Doubling** Let  $f(i, k)$  denote the number of characters of the infinite stream of  $s_i$ 's starting from  $s_1[i]$  needed to cover  $2^k$  copies of  $s_2$ . Then it follows that

$$f(i, k) = \begin{cases} single\_match\_count(i) & \text{if } k = 0 \\ f(i, k - 1) + f((i + f(i, k - 1)) \bmod \ell_1, k - 1) & \text{o.w.} \end{cases}, \quad (30.1)$$

where  $single\_match\_count(i)$  is the base case, which could be implemented as a simple accumulation. Then we just need to fill using double technique.

## LeetCode 467. Unique Substrings in Wraparound String

**Sliding Window** Let  $f(c)$  denote the length of the longest increasing contiguous subarray that ends with  $c$  in  $p$ . Then the final answer is  $\sum_{c \in [a, \dots, z]} f(c)$ . We also note the following:

- The contiguous subarray condition check needs to include the check to  $'za'$  case.

- $f(c)$  can be precomputed using a sliding window.

The overall runtime is then  $O(n)$ , where  $n = (p)$ .

## LeetCode 471. Encode String with Shortest Length

**Dynamic Programming with KMP** Let  $f(i, j)$  denote the shortest encoded string for  $s[i : j]$ . Then it follows that

$$f(i, j) = \begin{cases} & \text{if } j - i + 1 \leq 4 \\ ((j - i + 1)/p) \oplus [\oplus f(i, i + p - 1) \oplus] & \text{else if } p := \text{compress}(s[i : j]) > 0 . \\ \min_{i \leq k < j} \{f(i, k) \oplus f(k + 1, j)\} & \text{else if } \text{compress}(s[i : j]) = -1 \end{cases} \quad (32.1)$$

Note that the function  $\text{compress}(i, j)$  returns the number of multiples of substrings that are used to encode  $s[i : j]$ . This can be solved efficiently using KMP algorithm like in [LeetCode 459. Repeated Substring Pattern](#). We will also use a more generalized form of the prefix function  $\pi_1$ , where  $\pi_1(i, j)$  denote equals  $\pi_0(j)$  for  $s[i : j]$ , where  $\pi_0$  is the original vanilla prefix function.

## LeetCode 472. Concatenated Words

**Fold over Trie** We first build a Trie for *words*. Then we implement a fold operation to check if a given word is a concatenated word. Let  $f_s(n, i, k)$  denote the fold function representing if a given word  $s$  is a concatenated word when we start checking from trie node  $n$  and index  $i$  of  $s$ . Then we have

$$f_s(n, i, k) = \begin{cases} n.is\_word \text{ and } cnt \geq 1 & \text{if } i = (s) \\ false & \text{else if } s[i] \notin n.next \\ f_s(n.next[s[i]], i + 1, k) \text{ or } f(n, i + 1, k + 1) & \text{else if } s[i] \in n.next \text{ and } n.next[s[i]].is\_word = true \\ f_s(n.next[s[i]], i + 1, k) & \text{else if } s[i] \in n.next \text{ and } n.next[s[i]].is\_word = false \end{cases} \quad (33.1)$$

The final answer is  $\{s : s \in \text{words and } f_s(\text{root}, 0, 0)\}$ .

## LeetCode 474. Ones and Zeros

**Dynamic Programming** Let  $f(i, r_0, r_1)$  denote the maximum number of strings can be made with  $r_0$  0's and  $r_1$  1's for  $strs[0 : i]$ . Then we have that

$$f(i, r_0, r_1) = \begin{cases} 1 & \text{if } i = 0 \text{ and } costs[i] \leq (r_0, r_1) \\ 0 & \text{else if } i = 0 \text{ and } costs[i] > (r_0, r_1) \\ \max(f(i-1, r_0 - costs[i][0], r_1 - costs[i][1]), f(i-1, r_0, r_1)) & \text{else if } i > 0 \text{ and } costs[i] \leq (r_0, r_1) \\ f(i-1, r_0, r_1) & \text{else if } i > 0 \text{ and } costs[i] > (r_0, r_1) \end{cases} \quad (34.1)$$

Then the final solution is  $f(\ell - 1, m, n)$ , where  $\ell = (strs)$ .

## LeetCode 486. Predict the Winner

**Dynamic Programming** Let  $f(i, j)$  be the difference of score between player1 and player2 if they are to pick numbers between  $[i, \dots, j]$  with player 1 pick first. Note that since both players are playing optimally,  $f(i, j)$  also represents the difference player2 and player1 if they are to pick numbers between  $[i, \dots, j]$  with player 2 pick first. Then we have that

$$f(i, j) = \begin{cases} nums[i] & \text{if } i = j \\ \max(nums[i], nums[j]) - \min(nums[i], nums[j]) & \text{else if } i + 1 = j \\ \max(nums[i] - f(i+1, j), nums[j] - f(i, j-1)) & \text{o.w.} \end{cases} \quad (35.1)$$

The final answer is then  $f(0, n-1) \geq 0$ , where  $n = (nums)$ .

## LeetCode 514. Freedom Trail

**Dynamic Programming** Let  $f(i, j)$  denote the minimum number to open the door given that is key is  $key[j : -1]$  and the starting position of the ring is  $i$ . Then we have that

$$f(i, j) = \begin{cases} 0 & \text{if } ring[i] = key[j] \text{ and } j = n_{key} - 1 \\ f(i, j+1) & \text{else if } ring[i] = key[j] \text{ and } j < n_{key} - 1 \\ \min(first\_ll(i, key[j]).steps, first\_rr(i, key[j]).steps) & \text{else if } ring[i] \neq key[j] \text{ and } j = n_{key} - 1 \\ \min(G_{ll}, G_{rr}) & \text{else if } ring[i] \neq key[j] \text{ and } j < n_{key} - 1 \end{cases}, \quad (36.1)$$

where

$$G_u(i, j) = first\_ll(i, key[j]).steps + f(first\_ll(i, key[j]).id, key[j + 1]), \quad (36.2)$$

$$G_{rr}(i, j) = first\_rr(i, key[j]).steps + f(first\_rr(i, key[j]).id, key[j + 1]). \quad (36.3)$$

We note that  $first\_ll(i, c)$  and  $first\_rr(i, c)$  returns the first match on left (right) of the ring for character  $c$ .

### LeetCode 516. Longest Palindromic Subsequence

**Dynamic Programming** Let  $f(i, j)$  denote the longest palindrome for  $s[i : j]$ . Then we have

$$f(i, j) = \begin{cases} 1 & \text{if } i = j \\ 2 & \text{else if } i + 1 = j \text{ and } s[i] = s[j] \\ 1 & \text{else if } i + 1 = j \text{ and } s[i] \neq s[j] \\ 2 + f(i + 1, j - 1) & \text{else if } i + 1 < j \text{ and } s[i] = s[j] \\ \max(f(i + 1, j) + f(i, j - 1)) & \text{else if } i + 1 < j \text{ and } s[i] \neq s[j] \end{cases} \quad (37.1)$$

Then the final answer is  $f(0, n - 1)$ , where  $n = (s)$ . The overall running time is  $O(n^2)$ .

### LeetCode 517. Super Washing Machines

**Greedy** First, we need to check if the total number of machines is divisible by  $n$ . If not, then it is not possible to produce the desired outcome and therefore return  $-1$ . Otherwise, we first compute the average number of clothes that are supposed to be in each machine,  $avg$ , keep track for each machine  $i$ ,

$$L_i = \max(0, (i + 1) \cdot avg + RMQ(0, i - 1)), \quad (38.1)$$

$$R_i = \max(0, (n - (i + 1)) \cdot avg + RMQ(i + 1, n - 1)), \quad (38.2)$$

$$T_i = R_i + L_i. \quad (38.3)$$

Note that  $L_i$  represents the total number of clothes needed to be transferred from  $machines[i + 1 : -1]$  to  $machines[0 : i - 1]$  if any to balance the clothes count and similarly  $R_i$  represents the total number of clothes needed to be transferred from  $machines[0 : i - 1]$  to  $machines[i + 1, -1]$  to balance the clothes. We claim that the final answer must be one of the  $T_i$ 's. This is because suppose otherwise there doesn't exist a washing machines whose total transfer  $T_i$  is not the answer, then there must be the cases where some machine  $A$  moved once machine  $B$  is fixed and then machine  $B$  moved once while  $A$  is fixed. This is a contradiction since  $A$  and  $B$  can be chosen at the same time. Hence, therefore the final answer is then  $\max_i\{T_i\}$ .

## LeetCode 523. Continuous Subarray Sum

**Prefix Sum** Note that

$$\left( \sum_{p=i}^j nums[p] \right) \mod k = \left( \sum_{p=0}^j nums[p] - \sum_{p=i-1}^j nums[p] \right) \mod k \quad (39.1)$$

$$= \left[ \left( \sum_{p=0}^j nums[p] \right) \mod k \right] - \left[ \sum_{p=i-1}^j nums[p] \mod k \right]. \quad (39.2)$$

Therefore, it follows that

$$\sum_{p=i}^j nums[p] \mid k \iff \left[ \left( \sum_{p=0}^j nums[p] \right) \mod k \right] = \left[ \sum_{p=i-1}^j nums[p] \mod k \right]. \quad (39.3)$$

So we need to compute the prefix sum mod of  $k$  of  $nums$  and use a hashmap to check.

## LeetCode 600. Non-negative Integers without Consecutive Ones

**Dynamic Programming** Let  $f(k)$  denote the count of numbers without consecutive ones for  $k$  digits number that starts with 0. Then take  $k = 6$  for example, there are only two portions are valid choices, namely  $000000 \sim 011111$  and  $010000 \sim 010111$ . This is because any number that starts with  $11\dots$  does not meet the condition. Note that the count of numbers that satisfy the condition in  $000000 \sim 011111$  is essentially  $f(5)$  because the first zero is fixed and similarly the count for  $010000 \sim 010111$  is  $f(4)$ . Therefore, it follows that

$$f(k) = \begin{cases} 1 & \text{if } k = 0 \\ 2 & \text{else if } k = 1 . \\ f(k-2) + f(k-1) & \text{o.w.} \end{cases} \quad (40.1)$$

Then let  $g(i)$  denote the count of numbers that satisfy the condition for  $nums[i : -1]$ ; we have that

$$g(i) = \begin{cases} 1 & \text{if } i = 0 \text{ and } binary[i] = 0 \\ 2 & \text{else if } i = 0 \text{ and } binary[i] = 1 \\ f(i) + g(i-1) & \text{else if } i > 0 \text{ and } binary[i] = 1 \text{ and } binary[i-1] = 0, \\ f(i) + f(i-1) & \text{else if } i > 0 \text{ and } binary[i] = 1 \text{ and } binary[i-1] = 1 \\ g(i-1) & \text{else if } i > 0 \text{ and } binary[i] = 0 \end{cases} \quad (40.2)$$

where  $binary$  is the binary representation of  $num$ . Then final answer is then  $g(n-1)$ , where  $n = (binary)$ .

## 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  $\max(\text{flatten}(\text{arrays})) - \min(\text{flatten}(\text{arrays}))$ . 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  $\max(\text{flatten}(\text{arrays}))$  and  $\min(\text{flatten}(\text{arrays}))$  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  $array[i]$ , then

$$f(i) = \begin{cases} \max(P_{\min}(i), P_{\max}(i), S_{\min}(i), S_{\max}(i)) & \text{if } 0 < i < n-1 \\ \max(S_{\min}(i), S_{\max}(i)) & \text{else if } i = 0 \\ \max(P_{\min}(i), P_{\max}(i)) & \text{else if } i = n-1 \end{cases}, \quad (41.1)$$

where as

$$P_{\min}(i) = |\max(array[i]) - \min(\text{flatten}(array[0 : i-1]))|, \quad (41.2)$$

$$P_{\max}(i) = |\min(array[i]) - \max(\text{flatten}(array[0 : i-1]))|, \quad (41.3)$$

$$S_{\max}(i) = |\max(array[i]) - \min(\text{flatten}(array[i+1 : n-1]))|, \quad (41.4)$$

$$S_{\min}(i) = |\min(array[i]) - \max(\text{flatten}(array[i+1 : n-1]))|, \quad (41.5)$$

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

Code

## LeetCode 629. K Inverse Pairs Array

**Dynamic Programming** Let  $f(i, j)$  be the number of arrays from 1 to  $i$  that contains  $j$  inverse pairs. Then by noting that placing  $(i + 1)$  on  $x$  positions away from the right of  $[1, \dots, i]$  creates  $x$  more inverse pairs, we have that

$$f(i, j) = \begin{cases} 1 & \text{if } i = 1 \text{ and } j = 0 \\ 0 & \text{else if } i = 1 \text{ and } j > 0 \\ 1 & \text{else if } j = 0 \\ \sum_{m=0}^{\min(i-1, j)} f(i-1, j-m) & \text{o.w.} \end{cases}. \quad (42.1)$$

Such an algorithm has a running time complexity of  $O(n^2k)$ , which is unacceptable for the size of the input of the problem. So it is necessary to come up with an optimization scheme. Note that when  $1 < i \leq j$ , we have that

$$f(i, j) = \sum_{m=0}^{\min(i-1, j)} f(i-1, j-m) = \sum_{m=0}^{i-1} f(i-1, j-m), \quad (A)$$

$$f(i, j-1) = \sum_{m=0}^{\min(i-1, j-1)} f(i-1, j-1-m) = \sum_{m=1}^i f(i-1, j-m). \quad (B)$$

Subtracting (B) from (A) yields

$$f(i, j) - f(i, j-1) = \sum_{m=0}^{i-1} f(i-1, j-m) - \sum_{m=1}^i f(i-1, j-m) = f(i-1, j) - f(i-1, j-i), \quad (42.2)$$

which implies that

$$f(i, j) = f(i-1, j) + f(i, j-1) - f(i-1, j-i). \quad (42.3)$$

On the other hand, if  $i > j > 0$ , we have that

$$f(i, j) = \sum_{m=0}^{\min(i-1, j)} f(i-1, j-m) = \sum_{m=0}^j f(i-1, j-m), \quad (C)$$

$$f(i, j-1) = \sum_{m=0}^{\min(i-1, j-1)} f(i-1, j-1-m) = \sum_{m=1}^j f(i-1, j-m). \quad (D)$$

Subtracting (D) from (C) yields that

$$f(i, j) - f(i, j - 1) = f(i - 1, j), \quad (42.4)$$

which implies that

$$f(i, j) = f(i - 1, j) + f(i, j - 1). \quad (42.5)$$

Combining together, we have that

$$f(i, j) = \begin{cases} 0 & \text{if } i = 1 \text{ and } j > 0 \\ 1 & \text{else if } i = 1 \text{ and } j = 0 \\ 1 & \text{else if } j = 1 \\ f(i - 1, j) + f(i, j - 1) & \text{else if } i > j \\ f(i - 1, j) + f(i, j - 1) - f(i - 1, j - i) & \text{else if } i \leq j \end{cases}. \quad (42.6)$$

Then the running time complexity is reduced to  $O(nk)$ .

## LeetCode 639. Decode Ways II

**Dynamic Programming** Let  $f(i)$  denote the number of ways to decode  $s[0 : i]$ . Then

$$f(i) = \begin{cases} 9 & \text{if } i = 1 \text{ and } s[i] = * \\ 1 & \text{else if } i = 1 \text{ and } s[i] \in [1, \dots, 9] \\ 1 & \text{else if } i = 0 \\ 9 \cdot f(i - 1) + 15 \cdot f(i - 2) & \text{else if } s[i] = * \text{ and } s[i - 1] = * \\ 9 \cdot f(i - 1) + 9 \cdot f(i - 2) & \text{else if } s[i] = * \text{ and } s[i - 1] = 1 \\ 9 \cdot f(i - 1) + 6 \cdot f(i - 2) & \text{else if } s[i] = * \text{ and } s[i - 1] = 2 \\ 9 \cdot f(i - 1) & \text{else if } s[i] = * \text{ and } s[i - 1] \in \{0\} \cup \{3, \dots, 9\} \\ 2 \cdot f(i - 2) & \text{else if } s[i] = 0 \text{ and } s[i - 1] = * \\ f(i - 2) & \text{else if } s[i] = 0 \text{ and } s[i - 1] \in \{1, 2\} \\ f(i - 1) + 2 \cdot f(i - 2) & \text{else if } s[i] \in \{1, \dots, 6\} \text{ and } s[i - 1] \in \{*, 1, 2\} \\ f(i - 1) & \text{else if } s[i] \in \{1, \dots, 6\} \text{ and } s[i - 1] \in \{0\} \cup \{3, \dots, 9\} \\ f(i - 1) + f(i - 2) & \text{else if } s[i] \in \{7, \dots, 9\} \text{ and } s[i - 1] \in \{*, 1\} \\ f(i - 1) & \text{else if } s[i] \in \{7, \dots, 9\} \text{ and } s[i - 1] \in \{0\} \cup \{2, \dots, 9\} \end{cases}. \quad (43.1)$$



The final answer is  $f(n - 1)$ , where  $n = (s)$ . The total running complexity is  $O(n)$ . Note that there are cases in which  $s$  is not a valid input. We deal with these cases by exceptions.

### LeetCode 646. Maximum Length of Pair Chain

**Dynamic Programming** First, we sort  $pairs$  by lexicographical order. Let  $f(i)$  denote the maximum length of the pair chained ending with  $pairs[i]$ . Then we have

#### Patience Sorting

### LeetCode 647. Palindromic Substrings

**Dynamic Programming** Let  $f(i, j)$  denote if  $s[i : j]$  is palindrome or not. Then we have

$$f(i, j) = \begin{cases} true & \text{if } i = j \\ true & \text{else if } i + 1 = j \text{ and } s[i] = s[j] \\ false & \text{else if } i + 1 = j \text{ and } s[i] \neq s[j] \\ f(i + 1, j - 1) & \text{else if } s[i] = s[j] \\ false & \text{else if } s[i] \neq s[j] \end{cases} \quad (45.1)$$

Then the final answer is  $\sum_{(i,j) \in [0, \dots, n-1]^2} \mathbb{I}(f(i, j) = true)$ .

### LeetCode 650. 2 Keys Keyboard

**Dynamic Programming** Let  $f(i)$  denote the minimum number of operations to reach  $i$  copies of  $A$ . Then it follows that

$$f(i) = \begin{cases} 0 & \text{if } i = 1 \\ i & \text{else if } i \text{ is prime} \\ \min_{j \in [1, \dots, i] \text{ and } i|j} f(i/j) + j & \text{else if } i \text{ is not prime} \end{cases}, \quad (46.1)$$

where  $f(i/j) + j$  comes from  $f(i/j) + \underbrace{1}_{copy} + \underbrace{j-1}_{paste}$ . The overall running time complexity is  $O(N^2)$ .

## LeetCode 651. 4 Keys Keyboard

**Dynamic Programming** Let  $f(i)$  denote the maximum number of characters with a max of  $i$  operations. Note that to achieve maximum characters the last operations must be a pressing  $A$  or pressing  $Ctrl + V$ . Then it follows that

$$f(i) = \begin{cases} 1 & \text{if } i = 1 \\ 2 & \text{else if } i = 2 \\ \max(f(i-1) + 1, \max_{2 \leq j \leq i-2} \{f(j) \cdot (i - (j+2) + 1)\}) & \text{o.w.} \end{cases} \quad (47.1)$$

The final answer is  $f(N)$ . The overall running time complexity is  $O(N^2)$ .

## LeetCode 656. Coin Path

**Dynamic Programming** Let  $f(i)$  denote the minimum path to reach  $n$  from  $A[i]$ . Then

$$f(i) = \begin{cases} A[i] & \text{if } i = n - 1 \\ \min_{j \in \{i+1, \dots, i+B\}} \{A[i] + f(j)\} & \text{o.w.} \end{cases} \quad (48.1)$$

The final answer is then constructed by reconstructing from  $f$ . The running time complexity is  $O(nB)$ .

## LeetCode 664. Strange Printer

**Dynamic Programming** Let  $f(i, j)$  denote the minimum operations to print  $s[i : j]$ . Note that there are two options for the first step of  $s[i : j] : 1$ . just print one letter; 2. print all occurrences of  $s[i]$  up to some  $k \in [i+1, j]$ . Then, it we have that

$$f(i, j) = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{else if } i > j \\ 1 & \text{else if } i + 1 = j \text{ and } s[i] = s[j] \\ 2 & \text{else if } i + 1 = j \text{ and } s[i] \neq s[j] \\ \min(1 + f(i+1, j), \min_{k \in [i+1, \dots, j] \text{ and } s[i]=s[k]} \{f(i, k-1) + f(k+1, j)\}) & \text{o.w.} \end{cases} \quad (49.1)$$

## 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

$$f(i) = \begin{cases} \min(f(i-1) + \text{cost}[i-1], f(i-2) + \text{cost}[i-2]) & \text{if } i > 1 \\ 0 & \text{else if } i \in \{0, 1\} \end{cases} . \quad (51.1)$$

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

[Code](#)

## LeetCode 788. Rotated Digits

## LeetCode 877. Stone Game

**Dynamic Programming** Let  $f(i, j)$  denote the difference between the numbers of stones between Alex and Lee if they are to pick from  $piles[i : j]$  with Alex picks first. Then note that because each player is playing optimally,  $f(i, j)$  is also equal to the difference when Lee picks first. Then it follows that

$$f(i, j) = \begin{cases} \max(piles[i], piles[j]) - \min(piles[i], piles[j]) & \text{if } i + 1 = j \\ \max(piles[i] - f(i + 1, j), piles[j] - f(i, j - 1)) & \text{o.w.} \end{cases} . \quad (53.1)$$

Then the final answer is  $f(0, n - 1) > 0$ , where  $n = \text{size}(piles)$ .

## LeetCode 902. Number At Most $N$ Given Digit Set

## LeetCode 983. Minimum Cost For Tickets

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

$$f(i) = \begin{cases} \max(costs[0], costs[1], costs[2]) & \text{if } i \text{ then } -1 \\ \max(costs[i] + f(next\_lower(days[0 : i - 1], days[i] - pass[i])) & \text{else if } i \neq \emptyset, i > 0 \\ 0 & \text{else if } i = \emptyset \end{cases} \quad (55.1)$$

where  $next\_lower(A, x)$  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  $n = std :: size(days)$ .

## LeetCode 1012. Numbers With Repeated Digits

## 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  $[R, C] = \text{dimension}(\text{grid})$ . We count row by row. If  $grid[r][0] < 0$  for some  $r \in [0, R - 1]$ , then we add all the items in the submatrix  $grid[r : R - 1][0 : C - 1]$  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

$$f(r) = \begin{cases} C - c_N + f(r + 1) & \text{if } \min(grid[r]) < 0 \\ 0 & \text{else if } \min(grid[r]) > 0 \text{ or } r = R \\ (R - r) \cdot C & \text{else if } grid[r][0] < 0 \end{cases} \quad (58.1)$$

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

$$A[i] = \begin{cases} 0 & \text{if } \text{nums}[i] = 0 \\ \text{RSQ}(0, \text{nums}[i] - 1) & \text{otherwise} \end{cases}. \quad (59.1)$$

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  $n_2 = \text{std}::\text{size}(\text{arr1})$ , we use binary search to locate  $l, r$  such that  $\text{sorted\_arr2}[l : r] \subseteq [\text{arr1}[i] - d, \text{arr1}[i] + d]$  and add the element count of  $\text{sorted\_arr2}[l : r]$  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](#)

## LeetCode 1397. Find All Good Strings