
HOMEWORK 4

Solutions

1. (15 points) Given a set of n points in the plane \mathbb{R}^2 , say $(x_1, y_1), \dots, (x_n, y_n)$. Assume $x_1 < x_2 < \dots < x_n$. Let e_{ij} be the least square error of any line for approximating points $(x_i, y_i), \dots, (x_j, y_j)$, i.e.,

$$e_{ij} = \min_{a, b \in \mathbb{R}} \sum_{q=i}^j (y_q - ax_q - b)^2.$$

Design an $O(n^2)$ time algorithm to compute e_{ij} for all $1 \leq i < j \leq n$.

Hint. You can assume the minimum value of a two-variable quadratic function $f(x, y) = ax^2 + by^2 + cxy + dx + ey + g$ where coefficients $a, b, c, d, e, g \in \mathbb{R}$ and $a, b > 0$ can be computed in a constant time.

Answer. We want to compute the least square error e_{ij} by the given equation, where a and b are the coefficients of the best-fit line for the interval (i, j) , and e_{ij} can be minimized when a and b are optimal. To do that, we can further simplify the equation for e_{ij} and take the partial derivative with respect to a and b :

$$e_{ij} = \sum_{q=i}^j (y_q^2 + a^2 x_q^2 + b^2 - 2ax_q y_q - 2by_q + 2abx_q)$$

$$e_{ij} = \sum_{q=i}^j y_q^2 + a^2 \sum_{q=i}^j x_q^2 + b^2 l - 2a \sum_{q=i}^j x_q y_q - 2b \sum_{q=i}^j y_q + 2ab \sum_{q=i}^j x_q \quad (1)$$

$$\frac{\partial e_{ij}}{\partial a} = 2a \sum_{q=i}^j x_q^2 - 2 \sum_{q=i}^j x_q y_q + 2b \sum_{q=i}^j x_q = 0 \quad (2)$$

$$\frac{\partial e_{ij}}{\partial b} = 2bl - 2 \sum_{q=i}^j y_q + 2a \sum_{q=i}^j x_q = 0 \quad (3)$$

where l is $(j - i + 1)$.

By solving equation (2) and (3), we can find the optimal value for a and b . However, to calculate the minimum e_{ij} in the given time constraint, we need to know all the coefficients to compute the quadratic function of e_{ij} in a constant time. We can pre-compute these coefficients to achieve that.

To do that, first, we need to compute the cumulative sum of each coefficient. The

cumulative sum $C_{y2}[k]$ defined as:

$$C_{y2}[k] = \sum_{q=1}^k y_q^2$$

Where $C_{y2}[k]$ includes the sum of all the elements of coefficient y_q^2 from the start of the array up to the k-th element. Similarly, for other coefficients:

$$C_{x2}[k] = \sum_{q=1}^k x_q^2$$

$$C_{xy}[k] = \sum_{q=1}^k x_q y_q$$

$$C_y[k] = \sum_{q=1}^k y_q$$

$$C_x[k] = \sum_{q=1}^k x_q$$

After calculating all the cumulative sums from 1 up to n, and storing all the values. For each pair of (i, j) , we can use these cumulative sums to calculate the summation of each coefficient in constant time. We define the summation of each coefficient as:

$$S_{y2} = \sum_{q=i}^j y_q^2 = C_{y2}[j] - C_{y2}[i - 1]$$

$$S_{x2} = \sum_{q=i}^j x_q^2 = C_{x2}[j] - C_{x2}[i - 1]$$

$$S_{xy} = \sum_{q=i}^j x_q y_q = C_{xy}[j] - C_{xy}[i - 1]$$

$$S_y = \sum_{q=i}^j y_q = C_y[j] - C_y[i - 1]$$

$$S_x = \sum_{q=i}^j x_q = C_x[j] - C_x[i - 1]$$

We can then compute the optimal a and b by substitute these summations into equation (2) and (3):

$$\frac{\partial e_{ij}}{\partial a} = 2aS_{x2} - 2S_{xy} + 2bS_x = 0$$

$$\frac{\partial e_{ij}}{\partial a} = aS_{x2} - S_{xy} + bS_x = 0 \quad (4)$$

$$\frac{\partial e_{ij}}{\partial b} = 2bl - 2S_y + 2aS_x = 0$$

$$\frac{\partial e_{ij}}{\partial b} = bl - S_y + aS_x = 0 \quad (5)$$

We can isolate b from equation (5) and substitute into equation (4) to get the optimal value for a , and then use a to get the optimal value of b :

$$b = \frac{S_y - aS_x}{l}$$

$$a = \frac{lS_{xy} - S_yS_x}{lS_{x2} - S_x^2}$$

Now we can rewrite the equation of e_{ij} as all the terms can be simplified into summations:

$$e_{ij} = S_{y2} + a^2S_{x2} + b^2l - 2aS_{xy} - 2bS_y + 2abS_x \quad (6)$$

where l is $(j - i + 1)$.

Since all the summations $S_{y2}, S_{x2}, S_{xy}, S_y, S_x$ can be computed in constant time by using the precomputed cumulative sums, the value of a , b and $e_{i,j}$ can also be computed in constant time.

Now we iterate through all the possible pairs of (i, j) using a nested loop, and for each pair, we compute the summations of each coefficient, then use it to compute a , b and e_{ij} . After the loop is finished, we have computed e_{ij} for all $1 \leq i \leq j \leq n$.

Algorithm Summary

1. Compute cumulative sums for the coefficients $x_q^2, y_q^2, x_qy_q, x_q, y_q$ over the array.
2. Use nested loop to generate all pairs of (i, j) where $1 \leq i \leq j \leq n$.
3. For each interval (i, j) :
 - 2.1. Compute Summation $S_{y2}, S_{x2}, S_{xy}, S_y, S_x$ for current interval using corresponding Cumulative sums
 - 2.2. Compute the optimal value of a and b using the Summations of current interval
 - 2.3. Compute the least squares error using a, b and Summations of current interval
 - 2.4. Store e_{ij} of current interval.
4. Return the e_{ij} for all $1 \leq i \leq j \leq n$

Correctness Analysis

Claim 1: The algorithm computes the coefficient Summations $S_{y2}, S_{x2}, S_{xy}, S_y, S_x$ correctly for any interval (i, j) .

Proof: In our algorithm, the cumulative sum C_{y2} are precomputed as:

$$C_{y2}[k] = \sum_{q=1}^k y_q^2$$

Similarly for C_{x2}, C_y, C_{xy}, C_x .

And the summation for any interval (i, j) are computed as:

$$S_{y2} = C_{y2}[j] - C_{y2}[i - 1]$$

Similarly for S_{x2}, S_{xy}, S_y, S_x .

The cumulative sum C_{y2} stores the total summations from the start of the array up to index k . For any interval (i, j) , the summation $S_{y2} = \sum_{q=i}^j y_q^2$ is computed as:

$$S_{y2} = C_{y2}[j] - C_{y2}[i - 1]$$

This computation is mathematically equivalent to directly summing the coefficient values in the interval (i, j) since:

$$\begin{aligned} C_{y2}[j] &= \sum_{q=1}^j y_q^2 \\ C_{y2}[i - 1] &= \sum_{q=1}^{i-1} y_q^2 \end{aligned}$$

Subtracting these two will result in the total summation from index i to index j :

$$C_{y2}[j] - C_{y2}[i - 1] = \sum_{q=1}^j y_q^2 - \sum_{q=1}^{i-1} y_q^2 = \sum_{q=i}^j y_q^2 = S_{y2}$$

This is the same for S_{x2}, S_{xy}, S_y, S_x .

Therefore our algorithm correctly computes the coefficient summation $S_{y2}, S_{x2}, S_{xy}, S_y, S_x$.

Claim 2: The algorithm computes the optimal value of a and b for minimizing e_{ij} .

Proof: The error equation for an interval (i, j) is given by:

$$e_{ij} = \sum_{q=i}^j (y_q - ax_q - b)^2.$$

This can be expanded as equation (1) we computed:

$$e_{ij} = \sum_{q=i}^j y_q^2 + a^2 \sum_{q=i}^j x_q^2 + b^2 l - 2a \sum_{q=i}^j x_q y_q - 2b \sum_{q=i}^j y_q + 2ab \sum_{q=i}^j x_q$$

Using each coefficient summation we define, we can rewrite this equation as:

$$e_{ij} = S_{y2} + a^2 S_{x2} + b^2 l - 2a S_{xy} - 2b S_y + 2ab S_x$$

where l is $(j - i + 1)$.

Our algorithm minimized e_{ij} by solving:

$$\frac{\partial e_{ij}}{\partial a} = 0, \frac{\partial e_{ij}}{\partial b} = 0$$

Using the second derivative:

$$\frac{\partial^2 e_{ij}}{\partial^2 a} = 2S_{x2} = 2 \sum_{q=i}^j x_q^2$$

$$\frac{\partial^2 e_{ij}}{\partial^2 b} = 2l$$

Notice that both partial derivatives are always greater or equal to 0, both of them are convex. This means setting their first derivative gives us the minimum value of the error function.

That means by setting the first derivatives respect to both a and b to zero, our algorithm will find the correct value for a and b that minimized e_{ij} :

$$b = \frac{S_y - a S_x}{l}$$

$$a = \frac{l S_{xy} - S_y S_x}{l S_{x2} - S_x^2}$$

These formulas for a and b are derived from the normal equations for least squares error and by minimizing the quadratic equation using derivative. Thus, the algorithm correctly computes the optimal a and b that can result in minimum e_{ij} .

Claim 3: The algorithm correctly computes e_{ij} for all intervals.

Proof: Once a and b are computed using summations $S_{y2}, S_{x2}, S_{xy}, S_y, S_x$, we can compute e_{ij} using the formula we derived:

$$e_{ij} = S_{y2} + a^2 S_{x2} + b^2 l - 2a S_{xy} - 2b S_y + 2ab S_x \quad (6)$$

where l is $(j - i + 1)$.

By substituting the optimal value of a and b to this expanded error function, the algorithm ensures e_{ij} is minimized. The Algorithm iterates over all pairs of (i, j) using a nested loop, this ensures that the minimal e_{ij} is computed for all possible intervals.

Time Complexity Analysis

In this algorithm, we:

1. precomputed the cumulative sums $C_{y2}, C_{x2}, C_y, C_{xy}, C_x$ from the start of the array to n , where:

$$C[k] = \sum_{q=1}^n \text{value at index } q$$

During the cumulative calculation, we only need to sum up the previous cumulative value to the current index value to get the current index cumulative value, this takes constant time, for example:

$$C_y[k] = C_y[k-1] + y_k$$

Then for each array, we iterate through n points to compute the cumulative sums for each coefficient. Since each cumulative sum requires n iterations, and there are 5 cumulative sums, the total time for this step is: $O(n) + O(n) + O(n) + O(n) + O(n) = O(n)$

2. Iterating over all possible intervals In this step, we try to compute the least square error e_{ij} for all intervals. There are $\binom{n}{2}$ possible intervals, which is about n^2 intervals.

For each interval, we first calculate the summations $S_{y2}, S_{x2}, S_{xy}, S_y, S_x$. Since:

$$S = C[j] - C[i-1]$$

Therefore each summation can be computed using the corresponding cumulative sums in constant time $O(1)$.

Then we can compute the least square error e_{ij} using the equation:

$$e_{ij} = S_{y2} + a^2 S_{x2} + b^2 l - 2a S_{xy} - 2b S_y + 2ab S_x \quad (6)$$

Noticed that we derived the optimal a and b using summations:

$$a = \frac{l S_{xy} - S_y S_x}{l S_{x2} - S_x^2}$$

$$b = \frac{S_y - a S_x}{l}$$

Therefore all the terms of the e_{ij} function can be represent using summation $S_{y2}, S_{x2}, S_{xy}, S_y, S_x$. Since we computed the Summations in constant time $O(1)$, each terms in e_{ij} involves only basic arithmetic operations, so e_{ij} also takes $O(1)$ to compute.

Hence, iterating through all intervals and compute fore e_{ij} takes $O(n^2)$.

Thus, the total time complexity is $O(n + n^2) = O(n^2)$, which satisfies the requirement of the problem.

-
2. (20 points) **Longest palindrome subsequence.** A palindrome is a nonempty string over some alphabet that reads the same forward and backward. For example, *aaaabaaaa*, *00000*, *abddcba* are all palindromes. Give an efficient algorithm to find the longest palindrome that is a subsequence of a given input string s of input length n with 2 character-usage constraints. The character-usage constrain at index i means that index i have to be selected and $s[i]$ have to be appeared in the palindrome. Your algorithm should run in time $O(n^2)$. Note that we do not assume the alphabet is of size 26, or 10; it can be an alphabet of any size.

Input Example. Given a string $s_e = axbbybaea$ (one indexed) with 2 character-usage constraints at index 8 and 9, the longest palindrome subsequence is *aea* since the subsequence *abbba* doesn't include $s_e[8]$.

We used the nested loop to implement the dynamic programming algorithm. First, we will initialize a 2D array $dp[i][j]$. For example $dp[i][j]$ represents the length of the longest palindrome from index i to index j .

Base case:

- (1) For single character substrings:

$$dp[i][i] = 1$$

- (2) For two character substrings:

If $s[i] = s[i + 1]$:

$$dp[i][i + 1] = 2$$

else:

$$dp[i][i + 1] = 1$$

Inductive step

Assume that the dp table are correctly computed for the longest palindrome with length 1 and 2.

Then, for substrings of length $L > 3$: Consider index i and j where $|i - j| \geq 2$:

If $s[i] = s[j]$, then:

$$dp[i][j] = dp[i + 1][j - 1] + 2$$

else if $s[i] \neq s[j]$:

$$dp[i][j] = \max(dp[i + 1][j], dp[i][j - 1])$$

Then, run this transition steps for the nested loop, we will get a complete dp table. Also, we need to keep track of the longest palindrome in each iteration and make sure the constraints are in the substring. We just need to initialize a variable to store the max length of the palindrome, each time the max length is updated, we update the longest palindrome $s(i,j)$ where $dp[i][j]$ is the max length of palindrome in that iteration. Make sure we update longest palindrome $s(i,j)$ only when the max length is updated and constraints (c_1, c_2) are in the range.

After all the iteration, if the max length is still the same as the initialized value, then it means there is no palindrome contains the given constraints since we never find one in each iteration.

Pseudocode

```
Function LongestPalindromeSubsequence(s, c1, c2):
```

```
    If c1 or c2 out of bound:
        return ""
```

```
    # Step 1: Initialize DP Table
```

```
    Initialize dp[n][n] as a 2D array with all values = 0
```

```
    # Base cases: single-character substrings are palindromes of length 1
```

```
    For i from 0 to n-1:
```

```
        dp[i][i] = 1
```

```
    # Base cases: two-character substrings
```

```
    For i from 0 to n-2:
```

```
        If s[i] == s[i+1]:
```

```
            dp[i][i+1] = 2
```

```
        Else:
```

```
            dp[i][i+1] = 1
```

```
    # Step 2: Fill DP Table for substrings of length >= 3
```

```
    maxLen = 1
```

```
    longestPalindrome = empty string
```

```
    For length from 2 to n-1:
```

```
        For i from 0 to n-length-1:
```

```
            j = i + length
```

```
            If s[i] == s[j]:
```

```
                dp[i][j] = dp[i+1][j-1] + 2
```

```
            Else:
```

```
                dp[i][j] = max(dp[i+1][j], dp[i][j-1])
```

```
    # Step 3: Store the longest iteration
```

```
    If dp[i][j] > maxLen and (c_1,c_2) in range (i,j):
```

```
        maxLen = dp[i][j]
```

```
        longestPalindrome = s(i,j)
```

```
    If maxLen = 1: #the maxLen does not change:
```

```
        return "" #no such palindrome contains the given constraints
```

```
    else
```

```
        #return the longest palindrome with the constraints
```

```
return longestPalindrome
```

Runtime

(1) In step 1 , we are handling base cases with loops, it takes $O(n)$ time.

(2) In step 2, we used nested loop to filling the dp table, it obviously takes $O(n^2)$ time.

As a result, the algorithm takes $O(n^2)$ time in total.

-
3. (20 points) **Minimum cycle.** A cycle in a directed graph is defined as a sequence of vertices $u_1 u_2 \dots u_m$ where every adjacent $\langle u_i, u_{i+1} \rangle$ vertex and $\langle u_m, u_1 \rangle$ are connected. Every edge has a positive cost c_e . The fractional cost of a cycle $u_1 \dots u_m$ is defined as: the sum of their c cost divided by the cycle length m .

Algorithm 1 Minimum Fractional Cycle Cost

Require: Adjacency matrix G of size $n \times n$ with edge costs $c_e > 0$, precision ϵ , largest edge cost C

Ensure: Minimum fractional cost of a cycle up to precision ϵ

```

1: Initialize  $\mu_{\text{low}} \leftarrow 0, \mu_{\text{high}} \leftarrow C$ 
2: while  $\mu_{\text{high}} - \mu_{\text{low}} > \epsilon$  do
3:   Set  $\mu \leftarrow \frac{\mu_{\text{low}} + \mu_{\text{high}}}{2}$ 
4:   Define new edge weights  $w_e \leftarrow c_e - \mu$  for all edges  $e$ 
5:   Run Bellman-Ford algorithm on the graph with weights  $w_e$  to detect negative-weight
   cycles
6:   if a negative-weight cycle is found then
7:     Set  $\mu_{\text{high}} \leftarrow \mu$  ▷ Decrease upper bound
8:   else
9:     Set  $\mu_{\text{low}} \leftarrow \mu$  ▷ Increase lower bound
10:  end if
11: end while
    return  $\mu_{\text{low}}$ 

```

Problem Restatement

Given a directed graph $G = (V, E)$ with n nodes and n^2 edges (in an adjacency matrix form), each edge $e \in E$ has a positive cost $c_e > 0$. We are required to find the cycle with the minimum fractional cost, defined as:

$$\frac{\sum_{e \in \text{cycle}} c_e}{\text{length of cycle}}.$$

This quantity is often referred to as the *minimum mean cycle cost*.

Key Idea

To find the minimum mean cycle cost, we employ binary search on a parameter μ which represents a candidate mean cycle cost. For a given μ , we transform the edge weights as $w_e = c_e - \mu$. This transformation allows us to convert the minimum mean cycle problem into a negative cycle detection problem.

Algorithm Outline

Initialize $\mu_{\text{low}} = 0$ and $\mu_{\text{high}} = C$, where C is the maximum edge cost. While $\mu_{\text{high}} - \mu_{\text{low}} > \epsilon$, where ϵ is the desired precision: Set $\mu = \frac{\mu_{\text{low}} + \mu_{\text{high}}}{2}$. Transform edge weights as $w_e = c_e - \mu$. Run the Bellman-Ford algorithm to detect any negative-weight cycles.

If a negative cycle is found, set $\mu_{\text{high}} = \mu$. Otherwise, set $\mu_{\text{low}} = \mu$. Return μ_{low} as the minimum mean cycle cost.

Proof of Correctness

To prove the correctness of this algorithm, we need to show that:

1. If there exists a cycle with a mean cost less than μ , it appears as a negative cycle after transforming the edge weights to $w_e = c_e - \mu$.
2. The binary search converges to the minimum mean cycle cost within the desired precision ϵ .
3. The monotonicity for such problem.

Step 1: Cycle Transformation

For a given μ , we define transformed edge weights as:

$$w_e = c_e - \mu \quad \text{for all } e \in E.$$

Let $C = (v_1, v_2, \dots, v_m, v_1)$ be a cycle in G of length m . The total weight of the cycle C under the transformed weights is:

$$\sum_{e \in C} w_e = \sum_{e \in C} (c_e - \mu) = \sum_{e \in C} c_e - m \cdot \mu.$$

If the mean cycle cost $\frac{\sum_{e \in C} c_e}{m}$ is less than μ , then:

$$\sum_{e \in C} c_e < m \cdot \mu \Rightarrow \sum_{e \in C} w_e < 0.$$

Thus, any cycle with mean cost less than μ appears as a *negative cycle* in the graph with transformed edge weights w_e .

Conversely, if no negative-weight cycle exists for the transformed edge weights, it implies that all cycles have a mean cost greater than or equal to μ .

Step 2: Binary Search on μ

The algorithm uses binary search to converge on the minimum mean cycle cost. We start with an interval $[\mu_{\text{low}}, \mu_{\text{high}}] = [0, C]$, where C is the largest edge cost. At each step, we set $\mu = \frac{\mu_{\text{low}} + \mu_{\text{high}}}{2}$ and transform the edge weights.

1. **If a negative cycle is detected** using the Bellman-Ford algorithm with the transformed weights, it implies that there exists a cycle with mean cost less than μ . Thus, we set $\mu_{\text{high}} = \mu$, as the minimum mean cycle cost must be less than or equal to μ .
2. **If no negative cycle is detected**, it implies that all cycles have mean costs greater than or equal to μ . In this case, we set $\mu_{\text{low}} = \mu$.

The binary search continues until $\mu_{\text{high}} - \mu_{\text{low}} < \epsilon$, at which point μ_{low} approximates the minimum mean cycle cost to within the desired precision.

Step 3: Monotonicity Proof

To establish this monotonicity property, we define the transformed edge weights as:

$$w_e = c_e - \mu \quad \text{for all } e \in E,$$

where c_e is the original edge cost in G , and μ is the candidate mean cycle cost in the binary search.

Let $C = (v_1, v_2, \dots, v_m, v_1)$ be a cycle in G with length m . The total weight of cycle C under the transformed weights is:

$$\sum_{e \in C} w_e = \sum_{e \in C} (c_e - \mu) = \sum_{e \in C} c_e - m \cdot \mu.$$

Define the mean cost of cycle C as:

$$\text{Mean}(C) = \frac{\sum_{e \in C} c_e}{m}.$$

Case 1: If a Negative Cycle Exists for a Given μ

Suppose a negative cycle exists for a given μ , which means there exists a cycle C such that:

$$\sum_{e \in C} w_e = \sum_{e \in C} c_e - m \cdot \mu < 0.$$

Rearranging, we find:

$$\text{Mean}(C) = \frac{\sum_{e \in C} c_e}{m} < \mu.$$

Now consider a larger value $\mu' > \mu$. For the same cycle C , the transformed total weight becomes:

$$\sum_{e \in C} w'_e = \sum_{e \in C} c_e - m \cdot \mu' = \sum_{e \in C} w_e - m \cdot (\mu' - \mu).$$

Since $\mu' - \mu > 0$, it follows that:

$$\sum_{e \in C} w'_e < \sum_{e \in C} w_e < 0.$$

Thus, if a negative cycle exists for μ , it will also exist for any $\mu' > \mu$.

Case 2: If No Negative Cycle Exists for a Given μ

Suppose no negative cycle exists for a given μ , which means for all cycles C :

$$\sum_{e \in C} w_e = \sum_{e \in C} c_e - m \cdot \mu \geq 0.$$

Now consider a smaller value $\mu'' < \mu$. For any cycle C , the transformed total weight becomes:

$$\sum_{e \in C} w_e'' = \sum_{e \in C} c_e - m \cdot \mu'' = \sum_{e \in C} w_e + m \cdot (\mu - \mu'').$$

Since $\mu - \mu'' > 0$, it follows that:

$$\sum_{e \in C} w_e'' > \sum_{e \in C} w_e \geq 0.$$

Thus, if no negative cycle exists for μ , no negative cycle will exist for any $\mu'' < \mu$.

Monotonicity Conclusion

Based on the above two cases, we can conclude the following:

If a negative cycle exists for a particular μ , it will also exist for any larger value $\mu' > \mu$.
 If no negative cycle exists for a particular μ , it will also not exist for any smaller value $\mu'' < \mu$.

Therefore, **the existence of a negative cycle in the transformed graph is a monotonic property with respect to μ** . This monotonicity guarantees that binary search can effectively converge to the minimum mean cycle cost.

Step 4: Convergence and Complexity

Each iteration of the binary search reduces the interval $[\mu_{\text{low}}, \mu_{\text{high}}]$ by half. Thus, the binary search performs $O(\log(C/\epsilon))$ iterations to reach a precision of ϵ .

For each iteration, we run the Bellman-Ford algorithm for each vertex, which has a time complexity of $O(n^3)$ on a dense graph. Therefore, the overall complexity is:

$$O(n^3 \log(C/\epsilon)),$$

as required by the problem statement.

-
4. (20 points) **Knapsack on a tree.** You are given a tree of n vertices. Every vertex has a positive weight w_i (a real number). Your goal is to choose a subset of vertices such that no two vertices are connected by an edge, and maximize the weight of the vertices you choose. Give a linear time $O(n)$ algorithm for that.

Answer:

Meaning of DP state: We use dynamic programming with a depth-first search (DFS). Define two states for each vertex v :

- $\text{dp_include}(v)$: Maximum weight of the subtree rooted at v when v is included.
- $\text{dp_exclude}(v)$: Maximum weight of the subtree rooted at v when v is excluded.

Transition Functions:

$$\begin{aligned}\text{OPT_include}(i) &= w[i] + \sum_{c \in \text{children}(i)} \text{OPT_exclude}(c) \\ \text{OPT_exclude}(i) &= \sum_{c \in \text{children}(i)} \max(\text{OPT_include}(c), \text{OPT_exclude}(c))\end{aligned}$$

The optimal solution for the entire tree rooted at 0 is given by:

$$\text{OPT}(0) = \max(\text{OPT_include}(0), \text{OPT_exclude}(0))$$

The transition function is saying that, essentially, there are two cases to consider for each node:

- Node is included in the result set:** If a node is included in the final result set, none of its children can be included. Therefore, the best contribution from the subtree rooted at this node is the sum of the `OPT_exclude` values of all its children. Since each node has a positive weight, there is no reason to skip any child.
- Node is not included in the result set:** If a node is not included in the final result set, its children can be considered for inclusion. For each child, the best contribution will be the maximum of its `OPT_include` and `OPT_exclude` values. Thus, the total contribution from the subtree rooted at this node is the sum of the maximum values from each child's include and exclude states.

The final result is obtained by evaluating the two possible states for the root node and taking the maximum of these values:

$$\text{OPT}(\text{root}) = \max(\text{OPT_include}(\text{root}), \text{OPT_exclude}(\text{root})).$$

This is because the optimal solution for the entire tree must either include the root (and exclude its children) or exclude the root (and allow the best contributions from its children). By considering both cases and selecting the maximum value, we ensure the overall weight of the result set is maximized.

Algorithm 2 Knapsack on a Tree

```
1: procedure DFS( $v$ , parent)
2:   if dp_include[ $v$ ]  $\neq$   $-1$  and dp_exclude[ $v$ ]  $\neq$   $-1$  then
3:     return dp_include[ $v$ ], dp_exclude[ $v$ ]
4:   end if
5:   include  $\leftarrow w[v]$ 
6:   exclude  $\leftarrow 0$ 
7:   for each child  $c$  of  $v$  do
8:     if  $c \neq$  parent then
9:       (child_include, child_exclude)  $\leftarrow$  DFS( $c$ ,  $v$ )
10:      include  $\leftarrow$  include + child_exclude
11:      exclude  $\leftarrow$  exclude + max(child_include, child_exclude)
12:    end if
13:  end for
14:  dp_include[ $v$ ]  $\leftarrow$  include
15:  dp_exclude[ $v$ ]  $\leftarrow$  exclude
16:  return (include, exclude)
17: end procedure

18: procedure KNAPSACKONTREE( $w$ , adj)
19:   Initialize dp_include and dp_exclude arrays of size  $n$  with value  $-1$ 
20:   (include, exclude)  $\leftarrow$  DFS( $0$ ,  $-1$ )
21:   return max(include, exclude)
22: end procedure
```

Proof of Correctness: We prove the correctness using induction on the size of the subtree rooted at each vertex v .

Base Case: For a leaf node v , the subtree rooted at v contains only v itself. If we include v , the maximum weight is $w[v]$, so $\text{dp_include}(v) = w[v]$. If we exclude v , the maximum weight is 0, so $\text{dp_exclude}(v) = 0$. The computation for $\max(\text{dp_include}(v), \text{dp_exclude}(v))$ is clearly correct.

Inductive Step: Assume the algorithm correctly computes dp_include and dp_exclude for all subtrees of size $k \geq 1$. Now, consider a subtree of size $k + 1$ rooted at vertex v .

- If we include vertex v , we cannot include its children. Thus, the total weight is:

$$\text{dp_include}(v) = w[v] + \sum \text{dp_exclude}(c) \quad \text{for all children } c \text{ of } v.$$

- If we exclude vertex v , we can either include or exclude its children to maximize the total weight. Thus, the total weight is:

$$\text{dp_exclude}(v) = \sum \max(\text{dp_include}(c), \text{dp_exclude}(c)) \quad \text{for all children } c \text{ of } v.$$

By the inductive hypothesis, the algorithm correctly computes the dp_include and dp_exclude values for all children of v . Therefore, the values for v are also computed correctly.

Final Result: The algorithm computes `dp_include` and `dp_exclude` for the root of the tree. To obtain the maximum weight of the independent set for the entire tree, we take:

$$\max(\text{dp_include}(\text{root}), \text{dp_exclude}(\text{root})).$$

This is correct because the two states, `dp_include(root)` and `dp_exclude(root)`, represent all possible configurations: one where the root is included, and one where it is excluded. Taking the maximum ensures that the best configuration is selected.

Time Complexity Analysis:

The algorithm runs in $O(n)$, where n is the number of vertices in the tree, due to the following reasons:

- The tree is traversed using depth-first search (DFS). Each node is visited exactly once, and each edge is traversed at most twice (once when descending and once when backtracking). Since a tree with n nodes has $n - 1$ edges, the total traversal cost is $O(n)$.
- At each node v , the algorithm computes `dp_include[v]` and `dp_exclude[v]` by accessing precomputed DP values of its children in $O(1)$ time per child. The total work across all nodes is proportional to the number of edges, which is $O(n)$.
- Memoization ensures that each node's DP values are computed only once, eliminating redundant computation. Once a node's DP values are stored, future accesses return the result in $O(1)$.

-
5. (25 points) **Number of spanning trees.** You are given an undirected graph of n vertices. For two vertices labeled with i and j , there is an edge (i, j) if and only if $|i - j| \leq 5$. Give an algorithm that computes the number of spanning trees of this graph. Your algorithm should run in linear time $O(n)$. Note that a spanning tree $T = (V, E)$ differs from another $T' = (V, E')$ if and only if there exists some edges $e \in E$ but $e \notin E'$.

Hint 1. The time complexity may have $5!$ and/or 2^5 as constant terms.

Hint 2. This problem is not related to Kirchhoff's theorem (the matrix tree theorem). For that, the time complexity is $O(n^3)$ – for computing determinant.

State:

The core idea of this question is to use dynamic programming to select the appropriate combination of edges to form our spanning trees and compute the total number.

We define the meaning of the state the connectivity of a subset of vertices at a certain point in the algorithm.

Before I explain why we do the define, we emphasize a specific condition of the question: a vertex is only directly connected to its immediate neighbors within a distance of 5 because of the condition $|i - j| \leq 5$. This means when we are at the stage of a certain point, we can capture all the necessary information about the connectivity through looking back to 5 vertices and there is totally 6 vertices including this one.

Let us get back to the definition :)

We explain or say give the state a better definition. A state at stage i encodes the partitioning of the set of vertices $V_i = \{i - 5, i - 4, i - 3, i - 2, i - 1, i\}$ into connected components (trees), considering all edges and connections formed up to vertex i . We actually label the vertices in this intermediate stage with the current connected components they are in.

To write the state in coding, we can say $dp[i]$ [the labels of each 6 vertices in the current i -th stage], the label in the second param means that which connected components the vertex is in. In the real coding, we can do normalization for every i -th stage, which does not affect the final answer.

Induction:

The final goal is to get the number of spanning trees in this graph. We can think of that we collect all possibilities that can finally go to the final goal and accumulate their numbers in this *path*. So it is easily to be seen that the final state is all vertices are labeled by a same label, which means they are in a big connected components, which is the definition of spanning tree.

However, we only record the last 6 vertices and the main power we rely on is the constraints in the transition process (I will mention it later). We can say that the last 6 vertices are in the same connected component is equals to the fact that all the vertices

are in a connected component now. So the final answer is obvious, we go the way down to n -th stage and select the dp array element that represents all the last 6 vertices are the same label, which is $dp[n][\{1,1,1,1,1,1\}]$.

Base case:

The base case is really simple. We have initial state with no vertices and of course, the number of spanning trees is 0.

Then, we look at the first 6 vertices. The reason why I put these 6 processes into base case is because they have some difference between the following transitions, they do not have some constraints to do the stage transition. We can see that we can accumulate the possible path down to the state of 6 vertices(which actually the connected components circumstance now) by iterating all possible forwarding path in the dp process.

Transition & Correctness:

First, it is really dangerous to continue the above process without adding some constraints because the possible states will boom within nearly less than 10 steps. But actually we do not need to do that because we have a really good property that a vertex is only directly connected to its immediate neighbors within a distance of 5 because of the condition $|i - j| \leq 5$ as we mentioned before.

When we look at this property in a transition perspective, for example, when we iterate to i , we only need to care about $V_i = \{i - 5, i - 4, i - 3, i - 2, i - 1, i\}$ and $i - 6$ has nothing to do with our current process because anyway, the connection of i and following vertices will not influence the state of $i - 6$ now.

Let us look at a new question in transition. The reason we do not need to bring all the states in the previous process, say $V_{i-1} = \{i - 6, i - 5, i - 4, i - 3, i - 2, i - 1\}$ all the way down to current process is that we only care about a particular possibility, which are the states that can go the way down to the final answer, which means all the vertices must be in one big connected components. From the thinking, we can say that only if $i - 6$ is connected to one of the vertices in $\{i - 5, i - 4, i - 3, i - 2, i - 1\}$, we bring this state to the next stage, otherwise, we have no way to affect $i - 5$ anymore in the following processes.

Finally, we can get to the true transition process, in the i stage, we consider the possible connections between i to $\{i - 5, i - 4, i - 3, i - 2, i - 1\}$ (attention! we have to filter the states first, to make sure $i - 6$ is connected to one of the vertices here):

$$dp[i][new_state] += dp[i - 1][prev_state] * ways_to_transition$$

The transition is really simple. If i is connected to any existed connected components(just one), we just label it as the label of the connected component. And if not, we add a new label to represent a new connected component and assign it to i .

If i is connected to multiple existed connected components, the actual meaning is to

merge these connected components together and then add i into the new connected component, we label all these vertices with a new same connected component label. There is lots of possibilities here because we can link i to any one of each connected components, so the way we calculate *ways_to_transition* is to multiply the number of elements in each of the connected components.

It is worth thinking of how many transition will occur in one process, which help with our calculation of time complexity. The states in the previous stage that satisfy our requirement are less than the number of possible 203 combinations (we filter some that $i - 6$ is not connected with others :)). The transition itself represents that putting i into one of the connected components of the 5 vertices $\{i - 5, i - 4, i - 3, i - 2, i - 1\}$ or a new connected component, which will result at most 6 possibilities. All in all, the transition is actually a constant time operation, which may have $5!$ and/or 2^5 terms. *Really thx to the greed condition :)*

Conclusion:

While the design of the state and transition is complicated but the impl is actually simple. We just use a loops to iterate each stage from 1 to n . In the loop, we first filter the possible states in the previous stage and then trying to put the new vertex into different components in different states and do the counting / transition. In the last, we will fetch the final answer by indexing by the n -th stage and the final state, which means the labels of all the last 6 vertices are the same.

As for time complexity, we can see that the states of a particular stage is at most 203, which is the number of all possible partitions of connected components within 6 vertices. So the overall states is $O(\text{constant} \times N)$. And we already know that the transition is $O(\text{constant})$. So the overall time complexity is $O(\text{constant} \times N \times \text{constant}) = O(N)$.

Coding Problem Record

Yuchen:

DescriptionEditorialSolutionsDebugger

23. Merge k Sorted Lists

Solved

HardTopicsCompanies

You are given an array of `k` linked-lists `lists`, each linked-list is sorted in ascending order.

Merge all the linked-lists into one sorted linked-list and return it.

Example 1:

```
Input: lists = [[1,4,5],[1,3,4],[2,6]]
Output: [1,1,2,3,4,4,5,6]
Explanation: The linked-lists are:
1->4->5,
1->3->4,
2->6
```

19.8K21470 Online

Submissions

Status	Language	Runtime	Memory	Notes
Accepted	Python3	8 ms	20.2 MB	

DescriptionEditorialSolutionsDebugger

2365. Task Scheduler II

Solved

MediumTopicsCompaniesHint

You are given a **0-indexed** array of positive integers `tasks`, representing tasks that need to be completed in order, where `tasks[i]` represents the **type** of the `ith` task.

You are also given a positive integer `space`, which represents the **minimum** number of days that must pass **after** the completion of a task before another task of the **same** type can be performed.

Each day, until all tasks have been completed, you must either:

- Complete the next task from `tasks`, or
- Take a break.

Return the **minimum number of days needed to complete all tasks**.

5671712 Online

Submissions

Status	Language	Runtime	Memory	Notes
Accepted	Python3	143 ms	37.3 MB	

DescriptionEditorialSolutionsDebugger

158. Read N Characters Given read4 II - Call Multiple Times

Solved

HardTopicsCompanies

Given a `file` and assume that you can only read the file using a given method `read4`, implement a method `read` to read `n` characters. Your method `read` may be called **multiple times**.

Method read4:

The API `read4` reads **four consecutive characters** from `file`, then writes those characters into the buffer array `buf4`.

The return value is the number of actual characters read.

Note that `read4()` has its own file pointer, much like `FILE *fp` in C.

Definition of read4:

872192 Online

Submissions

Status	Language	Runtime	Memory	Notes
Accepted	Python3	36 ms	16.7 MB	

DescriptionEditorialSolutionsDebugger

1392. Longest Happy Prefix

Solved

HardTopicsCompaniesHint

A string is called a **happy prefix** if is a **non-empty** prefix which is also a suffix (excluding itself).

Given a string `s`, return the **longest happy prefix** of `s`. Return an empty string `""` if no such prefix exists.

Example 1:

```
Input: s = "level"
Output: "l"
Explanation: s contains 4 prefix excluding itself ("l", "le", "lev", "leve"), and suffix ("l", "el", "vel", "evel"). The largest prefix which is also suffix is given by "l".
```

1.4K256 Online

Submissions

Status	Language	Runtime	Memory	Notes
Accepted	Python3	126 ms	21.8 MB	

[illegible]

Problem List
<
>
?

Run
Submit

v4 Code

Description

295. Find Median from Data Stream

Test
Hints
Compare

The median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value, and the median is the mean of the two middle values.

- For example, for `arr = [2,3,4]`, the median is `3`.
- For example, for `arr = [2,3,4]`, the median is $(2 + 3) / 2 = 2.5$.

Implement the `MedianFinder` class:

- `MedianFinder()` initializes the `MedianFinder` object.
- `void addNum(int num)` adds the integer `num` from the data stream to the data structure.
- `double findMedian()` returns the median of all elements so far. Answers within 10^{-5} of the actual answer will be accepted.

Example 1:

```

Input
[MedianFinder, addNum, addNum, findMedian]
[[], [1], [3], []]

Output
[null, null, null, 2.0]

```

Editorial
Solutions

Test Result
Testcase
Submissions

Status
Language
Runtime
Memory
Notes

Accepted Nov 9, 2024
Python3
420 ms
35 MB
+ Notes

Problem List
<
>
?

Run
Submit

v4 Code

Description

2601. Prime Subtraction Operation

Test
Hints
Compare

You are given a **0-indexed** integer array `nums` of length `n`.

You can perform the following operation as many times as you want:

- Pick an index `i`, such that you haven't picked before, and pick a prime `p` which is **strictly less than** `nums[i]`, then subtract `p` from `nums[i]`.

Return `true` if you can make `nums` a strictly increasing array using the above operation and `else` otherwise.

A **strictly increasing array** is an array where each element is strictly greater than its preceding element.

Example 1:

```

nums = [4,6,1,3]
// 4 is not prime, so we can subtract 3 from it to get 1.
// 6 is not prime, so we can subtract 5 from it to get 1.
// 1 is prime, so we can subtract 1 from it to get 0.
// 3 is prime, so we can subtract 2 from it to get 1.
// Now the array is [0,1,1,1], which is not strictly increasing.
// So the answer is false.

```

Editorial
Solutions

Test Result
Testcase
Submissions

Status
Language
Runtime
Memory
Notes

Accepted Nov 9, 2024
Python3
122 ms
11.7 MB
+ Notes

Yuheng:

DescriptionAccepted xEditorialSolutionsSubmissions

167. Two Sum II - Input Array Is SortedSolved

MediumTopicsCompanies

Given a 1-indexed array of integers `numbers` that is already **sorted in non-decreasing order**, find two numbers such that they add up to a specific `target` number. Let these two numbers be `numbers[index1]` and `numbers[index2]` where $1 \leq \text{index}_1 < \text{index}_2 \leq \text{numbers.length}$.

Return the *indices of the two numbers, `index1` and `index2`, added by one as an integer array `[index1, index2]` of length 2.*

The tests are generated such that there is **exactly one solution**. You **may not** use the same element twice.

Your solution must use only constant extra space.

38. Count and SaySolved

MediumTopicsCompaniesHint

The **count-and-say** sequence is a sequence of digit strings defined by the recursive formula:

- `countAndSay(1) = "1"`
- `countAndSay(n)` is the run-length encoding of `countAndSay(n - 1)`.

Run-length encoding (RLE) is a string compression method that works by replacing consecutive identical characters (repeated 2 or more times) with the concatenation of the character and the number marking the count of the characters (length of the run). For example, to compress the string `"3322251"` we replace `"33"` with `"23"`, replace `"222"` with `"32"`, replace `"5"` with `"15"` and replace `"1"` with `"11"`. Thus the compressed string becomes `"23321511"`.

Given a positive integer `n`, return the n^{th} element of the **count-and-say** sequence.

Example 1:

DescriptionAccepted xEditorialSolutionsSubmissions

295. Find Median from Data StreamSolved

HardTopicsCompanies

The **median** is the middle value in an ordered integer list. If the size of the list is even, there is no middle value, and the median is the mean of the two middle values.

- For example, for `arr = [2,3,4]`, the median is 3.
- For example, for `arr = [2,3]`, the median is $(2 + 3) / 2 = 2.5$.

Implement the `MedianFinder` class:

- `MedianFinder()` initializes the `MedianFinder` object.
- `void addNum(int num)` adds the integer `num` from the data stream to the data structure.
- `double findMedian()` returns the median of all elements so far. Answers within 10^{-5} of the actual answer will be accepted.

48. Rotate ImageSolved


MediumTopicsCompanies

You are given an $n \times n$ 2D *matrix* representing an image, rotate the image by 90 degrees (clockwise).

You have to rotate the image **in-place**, which means you have to modify the input 2D matrix directly. **DO NOT** allocate another 2D matrix and do the rotation.

Example 1:

1	2	3
4	5	6



7	4	1
8	5	2

Peng Yao:

2246. Longest Path With Different Adjacent Characters

Solved

Hard Topics Companies Hint

You are given a **tree** (i.e. a connected, undirected graph that has no cycles) **rooted** at node 0 consisting of n nodes numbered from 0 to $n - 1$. The tree is represented by a **0-indexed** array `parent` of size n , where `parent[i]` is the parent of node i . Since node 0 is the root, `parent[0] == -1`.

You are also given a string `s` of length n , where `s[i]` is the character assigned to node i .

Return the length of the **longest path** in the tree such that no pair of **adjacent** nodes on the path have the same character assigned to them.

2.4K 102 2 Onls

Submissions

Status	Language	Runtime	Memory	Notes
Accepted	C++	843 ms	218 MB	
a few seconds ago				

662. Maximum Width of Binary Tree

Solved

Medium Topics Companies

Given the `root` of a binary tree, return the **maximum width** of the given tree.

The **maximum width** of a tree is the maximum **width** among all levels.

The **width** of one level is defined as the length between the end-nodes (the leftmost and rightmost non-null nodes), where the null nodes between the end-nodes that would be present in a complete binary tree extending down to that level are also counted into the length calculation.

It is **guaranteed** that the answer will fit in the range of a **32-bit** signed integer.

8.9K 116 19 Online

Submissions

Status	Language	Runtime	Memory	Notes
Accepted	C++	1 ms	18.7 MB	
a few seconds ago				

543. Diameter of Binary Tree

Solved

Easy Topics Companies

Given the `root` of a binary tree, return the length of the **diameter** of the tree.

The **diameter** of a binary tree is the **length** of the longest path between any two nodes in a tree. This path may or may not pass through the `root`.

The **length** of a path between two nodes is represented by the number of edges between them.

14.2K 258 70 Onls

Submissions

Status	Language	Runtime	Memory	Notes
Accepted	C++	0 ms	22.5 MB	
a few seconds ago				

42. Trapping Rain Water

Solved

Hard Topics Companies

Given `n` non-negative integers representing an elevation map where the width of each bar is `1`, compute how much water it can trap after raining.

32.8K 310 206 Online

Submissions

Status	Language	Runtime	Memory	Notes
Accepted	C++	1 ms	25.6 MB	
a few seconds ago				

Bowen Yu:

11. Container With Most Water

已解答

算术难度: 4 同步题目状态

中等 相关标签 相关企业 显示

You are given an integer array `height` of length `n`. There are `n` vertical lines drawn such that the two endpoints of the i^{th} line are $(i, 0)$ and $(i, \text{height}[i])$.

Find two lines that together with the x-axis form a container, such that the container contains the most water.

Return the maximum amount of water a container can store.

Notice that you may not slant the container.

5.2K 3.3K 56.9 MB 50 人在读

提交记录

所有状态	所有语言	执行用时	消耗内存	备注
通过	Java	6 ms	56.9 MB	

2024.11.11

代码

智能模式

```
1 class Solution {
2     public int maxArea(int[] height) {
3         int ans = 0;
4         int i = 0;
5         int j = height.length - 1;
6         while (i < j) {
7             int s = (j - i) * Math.min(height[i], height[j]);
8             ans = Math.max(s, ans);
9             if (height[i] < height[j]) {
10                 i++;
11             } else {
12                 j--;
13             }
14         }
15         return ans;
16     }
17 }
```

行 1, 列 1 已存稿

运行 提交

测试用例

测试结果

Case 1 Case 2 +

height =

[1, 8, 6, 2, 5, 4, 8, 3, 7]

</> Source

42. Trapping Rain Water


已解答

算术难度: 6 同步题目状态

困难 相关标签 相关企业

Given `n` non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it can trap after raining.

Example 1:



5.4K 2.7K 453 MB 84 人在读

提交记录

所有状态	所有语言	执行用时	消耗内存	备注
通过	Java	0 ms	453 MB	

九秒前

代码

智能模式

```
1 class Solution {
2     public int trap(int[] height) {
3         int ans = 0;
4         int left = 0;
5         int right = height.length - 1;
6         int preMax = 0;
7         int sufMax = 0;
8         while (left < right) {
9             preMax = Math.max(preMax, height[left]);
10            sufMax = Math.max(sufMax, height[right]);
11            ans += preMax < sufMax ? preMax - height[left++] : sufMax - height[right--];
12        }
13        return ans;
14    }
15 }
```

行 6, 列 25 已存稿

运行 提交

测试用例

测试结果

Case 1 Case 2 +

height =

[0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]

</> Source

923. 3Sum With Multiplicity

已解答

算术难度: 5 第 106 场周赛 Q3 同步题目状态

中等 相关标签 相关企业

Given an integer array `arr`, and an integer `target`, return the number of tuples (i, j, k) such that $i < j < k$ and $\text{arr}[i] + \text{arr}[j] + \text{arr}[k] == \text{target}$.

As the answer can be very large, return it modulo $10^9 + 7$.

Example 1:

Input: `arr = [1, 1, 2, 2, 3, 3, 4, 4, 5, 5], target = 8`

140 120 39 ms 43.3 MB 2 人在读

提交记录

所有状态	所有语言	执行用时	消耗内存	备注
通过	Java	39 ms	43.3 MB	

2024.11.11

代码

智能模式

```
1 class Solution {
2     public int threeSumMulti(int[] arr, int target) {
3         Arrays.sort(arr);
4         long ans = 0;
5         int n = arr.length;
6         for (int i = 0; i < n - 2; i++) {
7             int x = arr[i];
8             int j = i + 1;
9             int k = n - 1;
10            while (j < k) {
11                int sum = arr[j] + arr[k] + x;
12                if (sum == target) {
13                    j++;
14                } else if (sum < target) {
15                    j++;
16                } else {
17                    k--;
18                }
19            }
20        }
21        return (int) ans;
22    }
23 }
```

行 1, 列 1 已存稿

运行 提交

测试用例

测试结果

Case 1 Case 2 Case 3 +

arr =

[1, 1, 2, 2, 3, 3, 4, 4, 5, 5]

target =

8

</> Source

请输入题号或关键字

题库

948. Bag of Tokens

已解答

题目描述

题解

算术难度: 4 第 112 场周赛 Q4 同步题目状态

1792 相似标签 相关企业

You start with an initial **power** of `power`, an initial **score** of `0`, and a bag of tokens given as an integer array `tokens`, where each `tokens[i]` denotes the value of token.

Your goal is to **maximize** the total **score** by strategically playing these tokens. In one move, you can play an **unplayed** token in one of the two ways (but not both for the same token):

- **Face-up**: If your current **power** is **at least** `tokens[i]`, you may play token, losing `tokens[i]` **power** and gaining `1` **score**.
- **Face-down**: If your current **score** is **at least** `1`, you may play token, gaining `tokens[i]` **power** and losing `1` **score**.

Beware the **maximum** possible score you can achieve after playing **some** number of tokens

106 153 3 人在线

代码

智能模式

```
1 class Solution {
2     public int bagOfTokensScore(int[] tokens, int power) {
3         Arrays.sort(tokens);
4         int i = 0;
5         int j = tokens.length-1;
6         int ans = 0;
7         while(i <= j && power >= tokens[i]){
8             while(i <= j && power >= tokens[i]){
9                 power -= tokens[i++];
10                ans++;
11            }
12            if(i < j) {
13                power += tokens[j--];
14                ans--;
15            }
16        }
17        return ans;
18    }
19 }
```

行 1, 列 1 已存稿

运行 提交

提交记录

所有状态

执行用时

消耗内存

备注

通过

2024.11.11

Java

2 ms

41.9 MB

测试用例

测试结果

Case 1

Case 2

Case 3

+

tokens =

[100]

power =

50

</> Source