CSES Problem Set Editorial

0−jij−0

Contents

1 Introductory Problems

Contents

1.1 Weird Algorithm

This is a simple simulation problem. Just simulate the process as described and you are done.
For more info about why the process always terminates with the number eventually reaches 1 and the length of such a sequence, see Collatz Conjecture.

1.2 Missing Number

This problem can be easily solved with extra space using a set or a frequency array. Here we describe an linear solution using constant additional space

We know that the XOR operation is associative, commutative and that $x \oplus x = 0$. Let A be the XOR of all numbers from 1 to n, ans B be the XOR of all elements of the list. We have $A = B \oplus x$ where x is the missing number. Hence $x = A \oplus B$. Note that we can find A in constant time but this won't affect our solution much since we will still need linear time to find B.

Time Complexity: $O(n)$ (and $O(1)$ Space Complexity)

1.3 Repetitions

This is a classical Two-Pointers exercise. Let l be the left endpoint of our current DNA segment and r be its right endpoint, keep extending r while you can then update your current result if you get a larger segment. Then set $l = r = r + 1$ and continue until you traverse all the DNA segment.

Time Complexity: $O(n)$

1.4 Increasing Array

We can use the following strategy: on a list of size N, make the prefix of size $N - 1$ increasing using the minimum number of operations, then while the last element of the list is less than the one before it increment it by 1. (You can prove correctness of this method by induction)

To implement it iteratively, traverse the list from left to right while keeping a variable *cur* which is the last element in your current increasing list. Now to add an element x, if $x > cur$ then you add it without operations and update *cur*, otherwise you increase x until it reaches *cur* with $cur - x$ operations and continue.

Time Complexity: $O(n)$

1.5 Permutations

If $n = 1$ then the only permutation (1) works vacuously. If $n = 2$ or 3, you can check that no permutations is "beautiful".

If $n > 3$ then printing all even integers $\leq n$ in increasing order then all odd integers $\leq n$ in increasing order yields a "beautiful" permutation because in each group adjacent differences are exactly 2 and when we switch the difference is $d = |x - 1| \geq 3$ where $x \geq 4$ is the greatest even number $\leq n$.

Time Complexity: $O(n)$

1.6 Number Spiral

Find the side-length n of the largest square with vertex $(0, 0)$ where your point is not included. Now we know that our number $n^2 < a < (n+1)^2$. Note that $n = \max(x, y) - 1$. Rest is casework locating a in the spiral depending on whether n, x, y are even or odd.

Time Complexity: $O(1)$

1.7 Two Knights

Some observations to make:

- Two knights that can attack each other form a 2×3 or 3×2 rectangle in the grid
- Given such rectangles there are exactly 2 ways to place knights that can attack each other, and that rectangle will uniquely determine those 2 positions.

Building from here we can calculate the number of those recatngles in the given grid, multiply by 2 to account for the knights positioning and then subtract this number from the number of all possible placements of the knights.

The total number of knights placements in a $k \times k$ grid is $k^2(k^2-1)/2$. Now note that by symmetry, the number of 2×3 rectangles is equal to the number of 3×2 rectangles. This number is equal to $(k-1)(k-2)$ since we only need to choose the left column and top row of the rectangle. So the final answer for a fixed k is $k^2(k^2-1)/2 - 4(k-1)(k-2)$

Time Complexity: $O(n)$ (To find the answer for all $k \leq n$).

1.8 Two Sets

First let us note that if the sum of the first n integers is odd then there's obviously no solution. We claim that otherwise a solution always exists.

Start with an empty set A and a set B containing all integers from 1 to n. While the sum of elements in B is less than those in A remove the smallest element in B and place it in A. The procedure terminates with the first k integers in A and the rest in B for some k. Now if both sets have equal sum we are done otherwise the difference of their sums is in the set A because otherwise the last step in the procedure wouldn't have been necessary. So we just move it back to B and we are done.

Time Complexity: $O(n \log n)$ or $O(n)$ depending on the implementation.

1.9 Bit Strings

This is a simple counting problem. Each bit has 2 possibilities (0 or 1) so by the product rule the final answer is $2^n \pmod{10^9+7}$ and can be calculated via Fast Modular Exponentiation.

Time Complexity: $O(\log n)$

1.10 Trailing Zeroes

$10 = 2 \times 5$ and we can clearly see that in the prime factorization of $n!$ we have more 2's than 5's, so the problem now boils down to counting the largest power of 5 that divides $n!$ and this can be done via Legendre's Formula.

Time Complexity: $O(\log n)$

1.11 Coin Piles

WLOG assume $a < b$, if $2a < b$ then clearly we cannot reach two empty piles even if we always remove 1 coin from the first pile and 2 from the second. Otherwise we can do this move until both piles have the same number of coins, then if that number is a multiple of 3 we can alternate moves until we reach two empty piles. Otherwise we cannot reach empty piles.

Time Complexity: $O(1)$ per test case.

1.12 Palindrome Reorder

If the string length is even all occurring characters must have even frequency, and if the length is odd all characters but exactly one must have even frequency, otherwise no solution exists.

If that's the case we can count the occurrence of each character and then build a string containing half (with floor) as much of every character in the original string, then reverse it and append it to the end (with an edge case when n is odd and we have to add the odd-occurring character beforehand) and we have our palindrome.

Time Complexity: $O(A + n)$ where A is the size of the alphabet.

1.13 Gray Code

We can use the following recursive procedure to generate the Gray Code of length n

- if $n = 1$ return the list { '0', '1' }
- Generate Gray Code of length $n - 1$ ans store them in a list A
- Append to the list L all elements of A in the order they appear with a '0' bit appended in the beginning.
- Append to the list L all elements of A in reverse order with a '1' bit appended in the beginning.
- Return the list L

The correctness of this procedure follows by induction from the fact that when switching between the two steps of appending A to L we have the same string but each with a different character appended to it in the beginning.

Time Complexity: $O(2^n)$

1.14 Tower of Hanoi

This is a classical recursion exercise and has the following recursive solution given towers $(start, mid, end)$ and n disks:

- if $n = 1$ move the only disk from $start$ to end
- Move the smallest $n - 1$ disks on towers $(start, end, mid)$
- Move the largest disk from $start$ to end
- Move the smallest $n - 1$ disks on towers $(mid, start, end)$

Correctness and the fact that the minimum number of moves is 2^n can be proven by induction.

Time Complexity: $O(2^n)$

1.15 Creating Strings

We only need to print all distinct permutations of the given string. This can be done by sorting the string initially and keep finding its next permutation (using for example *next_permutation* in C++) until we reach the final one.

Time Complexity: $O(n!)$

1.16 Apple Division

Given the constraints, we can iterate over all subsets of apples and find the difference of its sum and the sum of its complement set. This can be done efficiently using recursion.

Time Complexity: $O(2^n)$

1.17 Chessboard and Queens

Recursively place queens on each row while keeping track of the columns/diagonals that are now under some queen's reach.

Time Complexity: $O(n!)$

1.18 Digit Queries

First locate your digit in the corresponding decimal range. To do so fix pointers $l = 1$, $r = 9$ and $d = 1$, where d represents the number of digits in this range and l, r are the limits of the range. Keep decrementing k by the size of the range (Namely $d(r - l + 1)$) until k becomes smaller than the size of the current range.

Now we need to locate our digit in the corresponding number. From our previous work we have the index of our digit in the corresponding range and we know that the first number in the range is l and each number contains d digits. Hence our number is $l + \left\lfloor \frac{k}{d} \right\rfloor$ and our digit is located at the $(k \pmod d)$-th digit of that number from the right.

Time Complexity: $O(\log n)$

1.19 Grid Paths

We are going to try to move to every possible direction when encountered with a '?' sign, but this method will time out. To optimize it we will stop when we reach a state where two opposite neighbors are visited and the other two are not because then no sequence of moves will visit the two cells.

Time Complexity: Unknown

2 Searching and Sorting

Contents

2.1 Distinct Numbers

We can either insert all elements in a set and return its size or insert them in a list, remove duplicates and then return its size.

Time Complexity: $O(n \log n)$

2.2 Apartments

There is an optimal solution where the apartments given to two the applicants are sorted in increasing order if we sort the applicants by their size requirement. Consider any solution where applicants i and j where $a[i] < a[j]$ get apartments x and y respectively where $b[x] > b[y]$ then we can swap the apartments of both applicants and we would still get a valid solution.

We get the following greedy algorithm: Sort the applicants by their size requirement and insert apartments' size in a multiset. Now iterate over the applicants and greedily select the smallest apartment available withing the applicant's tolerance and remove it from the multiset, or don't assign an apartment to the applicant and skip.

Time Complexity: $O(n \log n)$

2.3 Ferris Wheel

Sort the children by their weights, then using the Two-Pointers technique, always assign the heaviest child available to a gondola with the lightest child available if possible, or alone otherwise and continue.

Time Complexity: $O(n \log n)$

2.4 Concert Tickets

Insert the tickets' price in a multiset then simulate the process of buying tickets as described using *lower_bound* in C++ for example.

Time Complexity: $O(n \log n)$

2.5 Restaurant Customers

Compress the arrival and leaving times of customers and then build an array A such that $A[i] =$ number of customers in the restaurant at time i. Answer is then the maximum element in this array.

To build this array in linear time we can use Difference Arrays to perform range increments in constant time.

Time Complexity: $O(n \log n)$ (From compression)

2.6 Movie Festival

Given a list of movies it is always beneficial to watch the one that ends the soonest because it will discard the minimum number of remaining movies and we will still have the maximum number of option.

From here we get the following greedy algorithm: sort movies by ending time, then iterate over movies while keeping track of the last ending time you selected, updating the first time you get a movie with start time later than this stored ending time.

Time Complexity: $O(n \log n)$

2.7 Sum of Two Values

This is the classical 2SUM Problem. To solve it first sort your list of integers and have two pointers l at the beginning of the list and r at the end of the list.

If $a[l]+a[r] < x$ then no possible solution could include $a[l]$ and we can discard it by setting $l = l+1$
If $a[l]+a[r] > x$ then no possible solution could include $a[r]$ and we can discard it by setting $r = r-1$
We keep repeating this until our list is empty or we reach $a[l] + a[r] = x$.

Time Complexity: $O(n \log n)$ (Because of sorting)

2.8 Maximum Subarray Sum

We will solve this problem with the Two-Pointers technique. Keep a pointer l to the beginning of your subarray and r to its end (Initially both start at the beginning of the list). While your subarray sum is positive we keep extending it ($r = r + 1$), keeping track of the current sum and the best sum, and when the sum becomes negative we discard the whole subarray and continue ($l = r = r + 1$).

Time Complexity: $O(n)$

2.9 Stick Lengths

Let x be the final stick length, we need to minimize $\sum_{i=1}^{n} |p[i] - x|$. over all possible values of x.

We know that the optimal value of x is the median of the list (see 1D Geometric Median). So we need to find the median (using *kth_element* in C++ for example) then compute the summation above.

Time Complexity: $O(n)$

2.10 Missing Coin Sum

First, sort the integers in the list. Currently the smallest number we cannot create is 1. Assume at any point it is k, then if the number we're looking at is $x[i] > k$ then k is the answer to our problem because all numbers processed before give us the numbers less than k and all remaining numbers are greater than k. Otherwise from our assumption we can generate all numbers from 1 to $k + x[i] - 1$ so we update our k to $k + x[i]$.

Time Complexity: $O(n \log n)$ (Because of sorting)

2.11 Collecting Numbers

Let us construct a list A where $A[i]$ = index of number i in the list (takes linear time).

Now traverse this list, each time we get $A[i + 1] < A[i]$ it means that we will have to do another round to get numbers $i + 1$ and above. So we just count the number of such indices + 1 to account for the first round.

Time Complexity: $O(n)$ (Because of sorting)

2.12 Collecting Numbers II

Computing the initial result and intermediate results relies on the solution described in the previous section so make sure to read it.

We have seen that the value in a number only affects its check and the check of the next integer. So to handle the swap queries we can just subtract the contributions of the two numbers that have to be swapped, swap them and add their new contribution again.

Note that the case where the two numbers to be swapped are consecutive must be handled separately.

Time Complexity: $O(n + m)$ (Because of sorting)

2.13 Playlist

We will solve this problem using the Two-Pointers technique. Keep a pointer l to the beginning of your considered subarray and a pointer r to its end and a set S containing the elements in your subarray. While you can keep extending your subarray ($r = r + 1$) without adding an element already in your set do it and then update your answer accordingly. Then keep shrinking your subarray ($l = l + 1$) until you remove the next element that must be added and continue.

Time Complexity: $O(n \log n)$

2.14 Towers

It is always optimal to place a cube on the smallest cube that can handle it, assuming there is an extra cube with infinite size representing the addition of a tower. We can implement this method with a set using $upper_bound$ in C++ for example.

Time Complexity: $O(n \log n)$

2.15 Traffic Lights

Let us have a set S of positions of traffic lights + the numbers 0 and x to limit our segments, and a multiset L of lengths of passages without traffic lights which initially contains x (the whole passage has no lights).

At each element $p[i]$ find the biggest a and smallest b in S such that $a < p[i] < b$, then remove $b - a$ from L, add $p[i] - a$ and $b - p[i]$ to L and $p[i]$ to S. The answer after that addition will be the greatest element in L.

Time Complexity: $O(n \log n)$

2.16 Josephus Problem I

Have a set containing all numbers from 1 to n. Keep track of the current child number that is not removed from the circle. The next child to be removed is either the smallest number greater than this current number or the smallest number in the set in case the current number is the greatest, and the next current is the number that comes after the one we removed. All these can be maintained using a set.

Time Complexity: $O(n \log n)$

2.17 Josephus Problem II

This problem has the same structure and solution as the one before it, but now we need a data structure other than a normal set that can give us the index of a key in the sorted order and the key at a specific index. Treaps or C++ Policy-Based Data Structures do that job efficiently.

Time Complexity: $O(n \log n)$

2.18 Nested Ranges Check

Lets sort ranges $[a, b]$ by increasing a and decreasing b in case of ties in a. This way given a range, all ranges that might cover it are to its left and all ranges that this range can cover are to its right.

To know if a range cover the current range $\{a[i], b[i]\}$ we need to find if the maximum b among ranges to its left is $\geq b[i]$. To know if a range is covered by the current range $\{a[i], b[i]\}$ we need to find if the minimum b among ranges to its right is $\leq b[i]$.

Data Structures that does the job here are RMQ's or Segment Trees.

Time Complexity: $O(n \log n)$

2.19 Nested Ranges Count

Building on our previous work, we need to count for every range $\{a[i], b[i]\}$ how many ranges $\{a[j], b[j]\}$ to its left have $b[j] \geq b[i]$ and how many ranges $\{a[k], b[k]\}$ to its right have $b[k] \leq b[i]$.

A very elegant Data Structure that does the job efficiently here is the Wavelet Tree that answers each of those queries online in $O(\log n)$ time.

Time Complexity: $O(n \log n)$

2.20 Room Allocation

We will simulate the process. First we sort customers by their arrival time, then have a set $freeRooms$ containing the indices of the unoccupied rooms, and a set $occupiedRooms$ containing the leaving time of present customers along with the room they currently occupy.

When processing a customer, free all the rooms occupied that have leaving time less than the current customer's arrival time, then assign this customer to the free room of smallest index. This way the minimum number of rooms used will be the maximum index assigned to a customer.

Time Complexity: $O(n \log n)$

2.21 Factory Machines

Let us fix the working time T of the machines. Then this time is feasible if and only if $\sum_{i=1}^{n} \lfloor \frac{t}{k[i]} \rfloor \geq t$

Knowing that, we can binary search for the answer, since if time t is not feasible then any smaller time won't be feasible and if time t is feasible then any greater time is feasible.

Time Complexity: $O(n \log A)$ where A is the maximum possible answer.

2.22 Tasks and Deadlines

The final answer will be $\sum_{i=1}^{n} d[i] - f[i] = \sum_{i=1}^{n} d[i]$ - $\sum_{i=1}^{n} f[i]$

The first summation is constant, so we have to minimize the second summation. To do so we can take the tasks by increasing order of duration so that at each step $f[i] = \sum_{k=1}^{i} a[k]$ is minimum possible.

Time Complexity: $O(n \log n)$ (Because of sorting)

2.23 Reading Books

If $2 \max_{i=1}^{n} t[i] > \sum_{i=1}^{n} t[i]$ then clearly the answer is $2 \max_{i=1}^{n} t[i]$ because while each is reading that book the other can read the other ones.

Otherwise the answer is $\sum_{i=1}^{n} t[i]$ and we can achieve this time using the following strategy: let Kotivalo read the books by increasing order of duration to read it, and Justiina read first the longest one then the others by increasing order also.

Time Complexity: $O(n)$

2.24 Sum of Three Values

This is the classical 3SUM problem. To solve it sort the list of integers and then for each index i find whether the subarray $a[i+1...n]$ contains two values whose sum is $x-a[i]$ using the 2SUM algorithm.

Time Complexity: $O(n^2)$ (It is conjectured that no algorithm can solve 3SUM in $O(n^{2-\epsilon})\forall\epsilon > 0$)

2.25 Sum of Four Values

This is the classical 4SUM problem. To solve it first generate a list S of sums of all possible pairs in the list and sort that list, now the problem boils down to 2SUM and we give a different approach.

For each element at index i search for $x - S[i]$ in the subarray to the right of our element (we can use binary search since the list is sorted). Make sure both pairs selected do not share a common index.

Time Complexity: $O(n^2 \log n)$

2.26 Nearest Smaller Values

We will solve this problem using a stack. Start with a stack S containing the first element of the list. Now for each element $x[i]$ if it's bigger than the top the stack then the result for this element is the element at the top of the stack and then we push that element to the stack. Otherwise (if smaller) we keep popping the top of the stack since it won't be needed anymore until we reach an element smaller than our number (result for our current number) or the stack becomes empty (result is 0) and then we push our element to the stack.

Time Complexity: $O(n)$

2.27 Subarray Sums I

Find the prefix sums of the the array a. Now since $a[i] > 0$ this prefix sum array will be increasing. Hence for each index i we need to search to its right for the value $x + pref[i]$ (using binary search)

Time Complexity: $O(n \log n)$

2.28 Subarray Sums II

We will build up on our previous solution. We will add a map m that will count how many prefixes have sum k for every k that appears. Now for each index i we first decrement the value of $pref[i]$ in m and then add $m[x + pref[i]]$ to our result.

Time Complexity: $O(n \log n)$

2.29 Subarray Divisibility

The only modification to the previous solution that we need to do in this problem is to store $pref[i] \pmod n$ in m. Now for each index i we first decrement the value of $pref[i] \pmod n$ in m and then add $m[pref[i] \pmod n]$ to our result.

Time Complexity: $O(n \log n)$

2.30 Subarray Distinct Values

We will solve this problem using the Two-Pointers technique. Have pointers l pointing to the left endpoint of your current subarray and r pointing to its right endpoint, and a map m counting the frequency of each element appearing in your subarray. While you can extend your subarray without having more than k elements in m do it, then shrink your subarray by one ($l = l + 1$) and continue.

Time Complexity: $O(n \log n)$

2.31 Array Division

Fix the subarray sum upper bound a, now we can easily check feasibility by greedily taking largest subarray whose sum is $\leq a$ and then splitting it, in the end checking if we end up with $\leq k$ subarrays.

Knowing this fact we can now binary search for the answer since the solution space satisfies the binary search structure.

Time Complexity: $O(n \log A)$ where A is the sum of elements in the array.

2.32 Sliding Median

Have 2 multisets $low, high$ that stores the smallest and greatest half of the window respectively. Now the median is either the biggest element in low or the smallest element in $high$ depending on the parity of k and we can slide our window(move from $[i, i+k-1]$ to $[i+1, i+k]$) easily by performing $O(1)$ insertions and deletions to our multisets. Note that each element will be added once and removed at most once.

Time Complexity: $O(n \log n)$

2.33 Sliding Cost

Given an array of elements the minimum cost of making them equal is by making them all equal to the median of the list (see 1D Geometric Median). Hence solving this problem is essentially the same as solving the previous problem while also keeping track of the sum of the window which can be easily done and updated when sliding.

Time Complexity: $O(n \log n)$

2.34 Movie Festival II

We know how to solve the problem when $k = 1$, we will use a similar strategy for multiple members.

Sort the movies by increasing ending time and have a multiset s with k elements store the ending time of the movie that the ith member is watching. Given a movie if all elements in the set are greater than its starting time then discard it, otherwise assign it greedily to the member with the closer ending time less than it.

Time Complexity: $O(n \log n)$

2.35 Maximum Subarray Sum II

First find the prefix sums of the current array. Now at any index i the maximum subarray of length between a and b starting at i has sum $\max\{pref[i+a-1], ..., pref[i+b-1]\} - pref[i]$

To implement this efficiently we should be able to find the maximum of every window of size $b-a+1$ (see Sliding Window Maximum for a linear solution). Finally note that for the last few indices where no full window fits, we should take suffix maximums instead.

Time Complexity: $O(n)$

3 Dynamic Programming

Contents

3.1 Dice Combinations

Let $dp[i]$ = number of ways to construct sum i by throwing a die one or more times.
Final answer is $dp[n]$ and base case is $dp[0] = 1$.

Each die throw gives us a transition so $dp[i] = \sum_{j=1}^{\min(i,6)} dp[i-j] \pmod{10^9+7}$

We can optimize our code to use only constant extra space and remove the loop for every die throw by noticing that $dp[i]$ is the sum of the 6 values to its left in the dp array so we can precompute the values for dp[1...6] using the above method and then keep track of the sum of the previous 6 values and storing the result for i in $dp[i\%6]$. FInal answer is $dp[n\%6]$

Time Complexity: $O(n)$

3.2 Minimizing Coins

Let $dp[i]$ = Minimum number of coins needed to get a sum equal to i. Final answer is $dp[x]$ and base case id $dp[0] = 0$.

Each coin gives us one transition so $dp[i] = 1 + \min_{j=1}^{n} dp[i - c[j]]$

Time Complexity: $O(nx)$

3.3 Coin Combinations I

Let $dp[i]$ = Number of distinct ways we can produce a money sum equal to i. Final answer is $dp[x]$ and base case is $dp[0] = 1$.

Each coin gives us one transition so $dp[i] = \sum_{c[j]<=i} dp[i - c[j]] \pmod{10^9+7}$

Time Complexity: $O(nx)$

3.4 Coin Combinations II

Let $dp[i][j]$ = number of distinct ordered ways we can produce sum i using the first j coins only. Final answer is $dp[x][n]$ and base case is $dp[a][0] = [a == 0]$

At state $dp[i][j]$ we have two possible transitions: (result is their sum)
Do not use the j-th coin $\longrightarrow dp[i][j-1]$
Use the j-th coin if possible $\longrightarrow dp[i - c[j]][j]$

We can also optimize memory by compressing the j state into 2 because $dp[i][j]$ only depends on the previous value of j. So replace j anywhere above by $j\&1$ and $j-1$ by $(j\&1) \oplus 1$. Final answer is $dp[x][n\&1]$. Don't forget to take modulo.

Time Complexity: $O(nx)$

3.5 Removing Digits

Let $dp[i]$ = Minimum number of operations needed to make number i equal to 0. Final answer is $dp[n]$ and base case is $dp[0] = 0$.

Each digit in i gives us a transition so $dp[i] = \min_{d \in i} dp[i-d]$

We can also optimize memory since a digit cannot get greater than 9 so $dp[i]$ will only depend on the previous 9 values at most. So we compress this state to 10 and replace i by $i\%10$ and $i-d$ by $(i-d)\%10$. Final answer id $dp[n\%10]$

Time Complexity: $O(n \log n)$(We need $\log n$ to iterate over the digits of a number)

3.6 Grid Paths

Let $dp[i][j]$ = the number of ways to get from cell $(0,0)$ to (i,j). Final answer is $dp[n-1][n-1]$ and base case is $dp[0][0] = 1$.

At state $dp[i][j]$ if the corresponding cell is a trap then $dp[i][j] = 0$, otherwise we could've arrived from only 2 other cell so $dp[i][j] = (dp[i-1][j] + dp[i][j-1]) \pmod{10^9+7}$.

We can also optimize memory by compressing any one state into 2 (Let's say i) since it only depends on the previous and current value of i. So we replace i by $i\&1$ and $i-1$ by $(i\&1) \oplus 1$. Final answer is $dp[(n\&1) \oplus 1][n-1]$

Time Complexity: $O(n^2)$

3.7 Book Shop

This is the classical Knapsack-Problem. Let $dp[i][t]$ = Maximum number of pages we can read from the first i books without paying more than t units of money. Final answer is $dp[n][x]$ and base case is $dp[0][t] = 0$.

At state $dp[i][t]$ we have three possible transitions: (result is the maximum of them)
Do not buy the i-th book $\longrightarrow dp[i-1][t]$
Buy the i-th book if possible $\longrightarrow s[i-1] + dp[i-1][t-h[i-1]$
Read the maximum number of pages with strictly less than t units of money $\longrightarrow dp[i][t-1]$

We can also optimize for memory since state i depends only on state $i-1$ so we can compress the i state to 2 by replacing i by $i\&1$ and $i-1$ by $(i\&1) \oplus 1$. Final answer is $dp[n\&1][x]$.

Time Complexity: $O(nx)$

3.8 Array Description

Let $dp[i][v]$ = The number of ways to fill up the array up til index i such that the i-th value is v.

Final answer is $\sum_{v=1}^{m} dp[n-1][v]$ and base case is $dp[0][v] = 1$ if $x[0] = 0$ or $dp[0][x[i]] = 1$ and $dp[0][v] = 0$ otherwise.

On state $dp[i][v]$ we have two possibilities: $x[i] \neq 0$ and $x[i] \neq v$ then $dp[i][v] = 0$
Otherwise $dp[i][v] = dp[i-1][v-1] + dp[i-1][v] + dp[i-1][v+1] \pmod{10^9+7}$

We can also optimize for memory since state i depends only on state $i-1$ so we can compress the i state to 2 by replacing i by $i\&1$ and $i-1$ by $(i\&1) \oplus 1$.

Final answer is $\sum_{v=1}^{m} dp[(n\&1) \oplus 1][v]$.

Time Complexity: $O(nm)$

3.9 Counting Towers

Let $dp[i][b]$ = number of ways we can get a tower of height i knowing that below us we have 2 blocks of width 1 if $b = 0$ or 1 block of width 2 if $b = 1$. Final answer is $dp[n][0] + dp[n][1] \pmod{10^9+7}$ and base case is $dp[1][0] = dp[1][1] = 1$.

Assume $b = 0$ there are three states that lead us to $dp[i][0]$:
$b = 0$ and we extended both our blocks of width 1 $\longrightarrow dp[i-1][0]$
$b = 0$ and we extended one of our blocks(and place a new one the other place) $\longrightarrow 2dp[i-1][0]$
$b = 0$ or $b = 1$ and we extended none of our blocks(then added 2 of width 1) $\longrightarrow dp[i-1][0]+dp[i-1][1]$
Hence $dp[i][0] = 4dp[i][0] + dp[i][1]$

Assume $b = 1$ there are two states that lead us to $dp[i][1]$:
$b = 1$ and we extended our block of width 2 $\longrightarrow dp[i-1][1]$

$b = 0$ or $b = 1$ and we extended none of our blocks(then added 1 of width 2) $\longrightarrow dp[i-1][0]+dp[i-1][1]$
Hence $dp[i][1] = dp[i][0] + 2dp[i][1]$

You could also optimize for memory by compressing the i state to 2 since the i-th state only depends on the state before it. You could also compute the values in dp for all possible n's and then answer test cases in constant time.

Time Complexity: $O(n)$...WAIT!

We can optimize further! We have a small number of states(0 and 1) and constant transitions between states so we can use Matrix Exponentiation. Let's build a matrix M where $M[i][j]$ = Number of transitions from state i to state j. Then from our previous analysis, $M = \left(\begin{smallmatrix} 4 & 1 \\ 1 & 2 \end{smallmatrix}\right)$ and to solve our problem we just need to compute M^{n-1} and sum the entries of the matrix!

Time Complexity: $O(\log n)$

3.10 Edit Distance

Let $dp[i][j]$ = Minimum edit distance between $s[0...i-1]$ and $t[0...j-1]$. Final answer is $dp[n][m]$ and base case is $dp[0][0] = 0$.

At state $dp[i][j]$ we have four possibilities: (Result is minimum of those)
If $s[i] = t[j]$ we can simply discard them $\longrightarrow dp[i-1][j-1]$
We can add $s[i]$ to t or remove $s[i]$ from s $\longrightarrow 1 + d[i-1][j]$
We can add $t[j]$ to s or remove $t[j]$ from t $\longrightarrow 1 + dp[i][j-1]$
We can replace $t[j]$ by $s[i]$ or vice-versa $\longrightarrow 1 + dp[i-1][j-1]$

We can optimize memory by compressing the i-th (or the j-th) state to 2 since it only depends on the previous i-th state. We replace i by $i\&1$ and $i-1$ by $(i\&1) \oplus 1$. Final answer is $dp[n\&1][m]$

Time Complexity: $O(nm)$

3.11 Rectangle Cutting

Let $dp[i][j]$ = Minimum number of operations needed to cut an $i \times j$ rectangle into squares. Final answer is $dp[a][b]$ and base case is $dp[x][x] = 0$ and $dp[1][x] = dp[x][1] = x - 1$.

We will try every possible cut we have while making use of symmetry:

$$dp[i][j] = \min(\min_{k=1}^{\frac{i}{2}}\{1 + dp[k][j] + dp[i-k][j]\}, \min_{k=1}^{\frac{j}{2}}\{1 + dp[i][k] + dp[i][j-k]\})$$

Time Complexity: $O(a^2b + b^2a)$

3.12 Money Sums

Let $dp[i] = true$ if there's a subset with sum i and $false$ otherwise. Final answer is $i > 0\ \forall dp[i] = true$ and base case is $dp[0] = true$.

We iterate over every number x in the list and set $dp[i + x] = true\ \forall dp[i] = true$.

We can optimize time and memory by a high constant factor if we use a bitset instead of an array.

Time Complexity: $O(n^2A)$ where A is the maximum number in the list.

3.13 Removal Game

Let $dp[d][i]$ = Maximum difference the first player can get on a game in the segment $x[i...i+d-1]$.
Final answer is $(dp[n][0] + \sum_{i=0}^{n-1} x[i])/2$ and base case is $dp[1][i] = x[i]$
Since both players are playing optimally, after Player 1 makes a move, Player 2 can be considered Player 1 on the new configuration of the game.
Hence $dp[d][i] = \max(x[i] - dp[d-1][i+1], x[j] - dp[d-1][i])$

We can also compress our d state to 2 since it only depends on its previous value.
We replace d by $d\&1$ and $d-1$ by $(d\&1) \oplus 1$. Final answer is $dp[n\&1][0]$.

Time Complexity: $O(n^2)$

3.14 Two Sets II

First of all if $n = 1$ or $2 \pmod 4$ then the answer is 0 since the sum of integers from 1 to n is odd. Otherwise let $dp[x]$ = number of subsets of $\{1, 2, ..., n\}$ with sum x. Final answer is $dp[n(n+1)/4]$ and base case is $dp[0] = 1$.

We iterate over all numbers i from 1 to n and increment $dp[x+i] = dp[x+i] + dp[x] \pmod{10^9+7}$.

Note that when iterating over i we should iterate over x in reverse order, to avoid counting the use of a number i more than once.

Time Complexity: $O(n^3)$

3.15 Increasing Subsequence

Let $dp[i]$ = minimum last element in an increasing subsequence of size i. Final answer is maximum i such that $dp[i] \neq \infty$ and base case is $dp[0] = -\infty$ and $dp[i > 0] = \infty$

We iterate over the sequence by order and we assign $x[i]$ to the first position p such that $dp[p] > x[i]$.

Notice that from our initial assignment of the dp array and from our method of updating, the dp array will remain increasing throughout the procedure, so we can use binary search to speed up the updating phase. (*lower_bound* in C++ for example, or *upper_bound* if the sequence was only required to be non-decreasing.)

Time Complexity: $O(n \log n)$

3.16 Projects

We only care about days where we either start or finish a project, so we first compress all the days in the input. We get days on a smaller range $[1...A]$ where $A < 4 \times 10^5$. Now let $dp[i]$ = Maximum profit we can get by the end of the i-th day. Final answer is $dp[A]$ and base case is $dp[0] = 0$.

On state $dp[i]$ we have 2 possible transitions: (result is max of those)
On the i-th day we may have done nothing $\longrightarrow dp[i-1]$
Or we could have completed some project $\longrightarrow \max_{b[j]=i}\{p[j] + dp[a[j]-1]\}$

To optimize the second case, sort the projects by ending time and keep track of the nearest project with $b[j] <= i$ so we will check every project only once which is when we're computing $dp[b[j]]$.

Time Complexity: $O(n \log n)$ (Because of compression)

3.17 Elevator Rides

There's a natural bitmask DP solution that comes to mind but has time complexity $O(3^n)$ so it will not pass the constraints. We describe a faster $O(n2^n)$ bitmask solution.

Let $dp[mask]$ be a pair of integers (A, B) where A is the minimum number of elevator rides needed to get the subset $mask$ to the end of the building, and B the minimum sum of weights in the last ride of that subproblem. Final answer is $dp[2^n - 1].A$ and base case is $dp[0] = (0, x)$

Given a subset $mask$ of people, iterate over each person b in $mask$, and given $dp[mask \oplus 2^b]$ if we can fit this person in the last ride we do so otherwise we start a new ride with only him inside. Thus:

$$dp[mask] = \min_{b \in mask} \begin{cases} (dp[mask \oplus 2^b].A, \ \ dp[mask \oplus 2^b].B + w[b]), & \text{if } dp[mask \oplus 2^b].B + w[b] \leq x \\ (dp[mask \oplus 2^b].A + 1, \ \ w[b]), & \text{otherwise} \end{cases}$$

Time Complexity: $O(n2^n)$

3.18 Counting Tillings

Let us first generate all possible single rows (There are $\approx 3^n$ different rows, constrained combinations of $\sqcap, \sqcup, \sqsupset$ and $\sqsubset$) Now assign each row a mask where the i-th bit is 1 if the i-th symbol is $\sqcup$ and 0 otherwise. We will also store our transitions, when transforming each row to a mask each time we encounter a $\sqcap$ symbol we set the i-th bit in $trMask$ to 1 and in the end we add $trMask$ to the list of transitions of $mask$.

Then let $dp[i][mask]$ = number of ways to build a tower of height i such that the last row is $mask$. Final answer is $dp[m][0]$ and base case is $dp[0][0]$ = number of row without a $\sqcap$ symbol.

Then $dp[i][mask] = \sum_{trMask \in transitions[mask]} dp[i-1][trMask]$

Time Complexity: $O(m3^n)$

3.19 Counting Numbers

Let $dp[i][d][low][high][zero]$ = number of integers between $a[0...i-1]$ and $b[0...i-1]$ with no equal consecutive digits such that the last digit is d, with low being a flag denoting if our number is still equal to a, $high$ denoting if our number is still equal to b, and $zero$ denoting if our number is still equal to 0.

A dp recurrence will be messy to write, but all we have to do now is add digits to our number with cases depending on the flag configurations, and updating the flags accordingly.

Time Complexity: $O(\log b)$

4 Graph Algorithms

Contents

4.1 Counting Rooms

The problem essentially deals with counting the number of connected components of a given graph. Build an undirected graph G where each cell is represented by a node and two nodes are connected by an edge if they share a side (so we can go from one to the other in one move) and both of them are floors '.' and finally run DFS/BFS on each non-visited floor node and each BFS will cancel one connected component.

Time Complexity: $O(nm)$

4.2 Labyrinth

Build a graph G where cells are represented by nodes and two nodes share an edge if they share a side and both of them are not walls. Also, have each edge store the direction it goea to ('L', 'R', 'U', 'D').

Now run BFS/DFS on the graph G from the starting node while saving for each node the edge that connects it to its parent in the traversal. If we don't visit the end node in our traversal the answer is "NO" otherwise the answer is "YES" and we can reconstruct the path iteratively using a stack.

Time Complexity: $O(mn)$

4.3 Building Roads

Build the graph G given in the problem. Consider one connected component of G, every city is reachable from every other city in it so we don't need to add edges between them, and connecting any new city to any city in that component will make that city reachable from any node in our component.

Hence we can consider each connected component as a single node, and we will be left with k nodes with no edges between them, where k is the number of connected components in G. Thus the minimum number of new roads we need to add is $k-1$ (To build a tree out of the k nodes)

To find the roads, store one vertex for each connected component you visit while doing DFS/BFS on G, and then add edges between every consecutive vertex.

Time Complexity: $O(n+m)$

4.4 Message Route

This is a shortest path problem, and shortest path problems on unweighted graphs can be solve via BFS. Build the graph G and run BFS from node 0, keeping track of the distance $d[v]$ of each node v from node 0 and of the parent $par[v]$ of each node v. If node $n-1$ is not reachable then the answer is "IMPOSSIBLE" otherwise the answer is $d[n-1]$ and the path can be reconstructed using a stack.

Time Complexity: $O(n+m)$

4.5 Building Teams

Build an undirected graph G where pupils are nodes and friendship relations are edges. Under this convention, we are asked to partition the graph into 2 independent sets or say that it is not possible. This is exactly checking if the graph is bipartite and can be done greedily with BFS/DFS.

Treat each component individually, take any node and put it in the first set and then in the traversal put every non-visited node in the opposite set as its parent. If you reach a visited node that has the same color as its parent, this is an odd length cycle and hence the answer is "IMPOSSIBLE", otherwise repeat for every component and you're done.

4.6 Round Trip

Build the graph G given in the problem, then we are asked to check whether G contains a cycle of length 3 or more. This can be done using DFS.

Treat each component individually, while traversing the graph if we find an edge going from a node to a visited node other than the parent, then we have found our cycle, otherwise there's no cycle.

To extract our cycle keep track of the parent of each node while doing DFS, and when we find our required edge uv we set $parent[v] = u$ and then we can extract our cycle using a stack.

Time Complexity: $O(n + m)$

4.7 Monsters

We will use multi-source BFS to solve this problem, first push into the BFS queue every node corresponding to a monster cell, and in the end push the node corresponding to your starting cell.

When we reach our source in the queue, we would have traversed every node reachable by monsters using at most 1 move, then the remaining nodes are the nodes that are safe to be visited using at most 1 move. Now we assume all our visited nodes are new sources, and all previously visited nodes are new monsters. If some boundary square becomes a source node at some point we are done, otherwise the answer is "NO".

To get the valid path, keep track of the parent of each node and then we can reconstruct our path with a stack.

Time Complexity: $O(n + m)$

4.8 Shortest Routes I

This is the classical Single Source Shortest Path problem. The graph is weighted but all weights are positive, hence we can use Dijkstra's Algorithm to solve the problem.

Time Complexity: $O((n + m) \log n)$

4.9 Shortest Routes II

The graph size is relatively small, hence we can find the shortest path between every pair of vertices and then answer queries in constant time. This is the classical All Pairs Shortest Path problem and we can use Floyd-Warshall Algorithm to solve the problem.

Time Complexity: $O(n^3 + q)$

4.10 High Score

We will first multiply all edge weights by -1. Now the problem boils down into finding the shortest path between node 0 and node $n - 1$, or detect a negative weight cycle which means our score can get arbitrarily large.

We will use Bellman-Ford Algorithm, and we will run the outer loop one extra iteration, recording all nodes that have been relaxed during that final round. If no nodes have been relaxed during that round, then no negative weight cycles are reachable from the source and hence the answer is $-dist[n - 1]$. Otherwise the nodes we recorded are nodes reachable from a negative weight cycle reachable from the source.

We're not done yet! Because the negative weight cycle might not reach node $n - 1$. To solve this issue we run multi-source BFS with all recorded nodes in our queue and we try to reach node n. If we succeed the answer is -1, otherwise $-dist[n - 1]$.

Time Complexity: $O(nm)$

4.11 Flight Discount

This is a problem where we have to modify Dijkstra's Algorithm in order to solve it. We will have a 2D array $dist$ where $dist[u][b]$ = Shortest path from node 0 to node u where b is a flag keeping track of whether we have used the discount option (if $true$) or not (if $false$)

At a state $dist[u][0]$ we can relax using an edge uv both states $dist[v][0]$ and $dist[v][1]$
$dist[v][0] = \min(dist[v][0], dist[u][0] + e_{uv}.w)$ $\quad dist[v][1] = \min(dist[v][1], dist[u][0] + e_{uv}.w/2)$

Time Complexity: $O((m+n)\log n)$

4.12 Cycle Finding

We simply have to check if a negative weight cycle exists using Bellman-Ford Algorithm. We can then construct the cycle if we keep track of the parents and then use a stack.

We will run Bellman Ford from node 0, and in case some nodes can reach a negative weight cycle but are not reachable from 0, we will add edges from node 0 to node u for every u, with weight ∞ to make sure every node is reachable from 0 and that those edges won't induce a negative weight cycle.

Time Complexity: $O(mn)$

4.13 Flight Routes

We will modify Dijkstra's Algorithm to solve this problem. Have an array $visited$ such that $visited[u]$ = Number of times node u has been popped out the priority queue. The i-th time node u is popped out the priority queue corresponds to the i-th shortest path from node 0 to node u. We keep relaxing using nodes in the priority queue until $visited[u] = k$, then we discard every occurrence of u.

Time Complexity: $O(k(m+n)\log n)$

4.14 Round Trip II

We have to solve the directed version of the cycle finding problem. To do so we will mark nodes as 1 if our DFS is still inside its subtree in the DFS-tree and 0 otherwise. Whenever we get an edge to a node that is marked 1 (We call this a back-edge) we have found our cycle, otherwise no cycle exists.

To extract the cycle we can keep track of parents, then when we find our back-edge uv we set $parent[v] = u$ and then we can reconstruct our cycle using a stack.

Time Complexity: $O(m+n)$

4.15 Course Schedule

Build a directed graph G where courses are represented by nodes and prerequisite relations are directed edges. Then the problem reduces to finding a topological ordering of the nodes in G.

To check whether a solution exists, we need to check if our graph contains cycle (solution described in the previous problem). If G has a cycle the answer is "IMPOSSIBLE", otherwise we can run sort our nodes using any efficient Topological Sorting algorithm.

Time Complexity: $O(m+n)$

4.16 Longest Flight Route

This is the classical problem of finding the longest path in a DAG. First sort the nodes in reverse topological order. Then define $dp[i]$ = Longest path from node $order[i]$ to node $order[n-1]$. Now $dp[i] = 1 + \max_{iv \in E}\{dp[order[v]]\}$ and by the topological property every neighbour of i will have its answer calculated before i. Final answer is $dp[order[0]]$ and base case is $dp[order[n-1]] = 1$.

Time Complexity: $O(m+n)$

4.17 Game Routes

Build a directed graph G where nodes are levels and directed edges are teleporters, the problem now reduces to count the number of path between two nodes in a DAG.

Sort the nodes using any efficient Topological Sort Algorithm, then traverse the nodes in reverse topological order and compute $dp[i]$ = Number of path from node $order[i]$ to node $order[n-1]$. $dp[i] = \sum_{iv \in E} dp[order[v]]$. Final answer is $dp[order[0]]$ and base case is $dp[order[n-1]] = 1$.

Time Complexity: $O(m+n)$

4.18 Investigation

We will modify Dijkstra's Algorithm in order to solve this problem. Since we only care about shortest paths, we keep considering each node only the first time it is popped out the priority queue.

We will keep track of the number of shortest paths, length of longest shortest paths and shortest shortest path in terms of edges for each node. When relaxing, if a node u gives a shorter path than already saved then we override v's data with u's, and if it gives an equal path weight then we merge the two nodes' data (sum of number of path, min/max of shortest/longest shortest path).

Time Complexity: $O((m+n)\log n)$

4.19 Planet Queries I

We will use Binary Lifting to solve the problem. Let $dp[i][u]$ = the node that we reach if we start from node u after using 2^i teleporters. Base case is $dp[0][u] = t[u]$ and the recurrence is $dp[i][u] = dp[i-1][dp[i-1][u]]$ (We use 2^{i-1} teleporters 2 times)

Now given x and k we iterate over the '1' bits of k and for each bit i change x to $dp[i][x]$ (simulating the use of 2^i teleporters) and we get our answer.

Time Complexity: $O(n \log K + q \log K)$ where K is the maximum possible value of k.

4.20 Planet Queries II

The graph we have to deal with is a functional graph, a directed graph where each node has out-degree exactly 1. This graph is a set of disjoint cycles with paths that eventually leads to a cycle.

Run DFS on the nodes and mark the nodes that are part of a cycle, and for each cycle choose a node u and let $len[v]$ = distance of every node v that can reach u to u. Note that to get this node for a particular node u we have to find the $len[u]$-th descendant of node u which can be done using the method described above. Now to find the length of the path between node a and node b we have to deal with many cases:

If cycle representative of nodes u and v are not equal, this means each node reaches a different cycle and hence no path between the 2 nodes exists.
If node u is in the cycle and node v is not then there's also no way to go from node u to node v
If both nodes are in the cycle then the answer is $len[u] - len[v]$ (+ cycleLength if $len[u] < len[v]$)
Otherwise v is either unreachable, or at a distance of $len[u] - len[v]$ or at a distance of $len[u]+$ cycleLength $-len[v]$. We can try each of the two distances with the descendant function.

Time Complexity: $O(n \log n + q \log n)$

4.21 Road Reparation

This is the classical Minimum Spanning Tree Problem. We can solve it using any efficient MST Algorithm (Prim, Kruskal, Boruvka, ...)

Time Complexity: $O(m \log n)$

4.22 Road Construction

We will use a DSU to keep track of the components and their sizes, each time construct a road between nodes u and v we merge u's and v's components. If the merge is successful (the two nodes were initially in different components) the number of components decreases by 1 and the size of the maximum component is either the previous one or the size of the new component.

Time Complexity: $O(m\alpha(n))$ Where α is the Inverse Ackermann Function.

4.23 Flight Routes Check

We need to check if the given directed graph is strongly connected or not, to do so we can find the strongly connected components of the graph using DFS. If the graph has only one SCC then it is strongly connected, otherwise any node in the last SCC cannot reach any node in the first SCC (in topological order)

Time Complexity: $O(n + m)$

4.24 Planets and Kingdoms

This problem also deals with strongly connected components, find them using DFS, then give each node the index of the component it belongs to. (the number of kingdoms will be the number of SCC)

Time Complexity: $O(n + m)$

4.25 Giant Pizza

We reduce this problem to 2-SAT. Have a variable for each pizza topping and a clause for each person. We negate a clause variable if the corresponding person finds the corresponding topping bad otherwise we don't. We then have to find a topping assignment that satisfy all the requirements, with a person being satisfied if any of his requirements is fulfilled. This is then the classical 2-SAT problem and can be solved using DFS

Time Complexity: $O(n + m)$

4.26 Coin Collector

Being at a node u, we can visit all nodes in its strongly connected component and return back to u. From this observation we conclude that SCC, that we can find using DFS, can be compressed into a single node.

Now we have a DAG with weights on each node and we need to find a path of maximum weight, this is the classical Longest Path in a DAG problem and can be solved using Topological Sort and Dynamic Programming.

Time Complexity: $O(n + m)$

4.27 Mail Delivery

We are given an undirected graph G and are asked to find a path that goes through every edge exactly once. This is exactly the Eulerian Path problem, which exists if and only if every vertex except two or none have even degrees and the graph is connected up to addition of isolated vertices. We can find eulerian paths in linear time w.r.t number of edges

Time Complexity: $O(m)$

4.28 De Bruijn Sequence

Construct a directed graph with 2^{n-1} nodes labeled in binary from 0 to $2^{n-1}-1$. Let $mask = 2^{n-1}-1$, add a directed edge between node x and nodes $(x << 1)\&mask$ with label 0 and $(x << 1|1)\&mask$ with label 1. Notice that every n-bit sequence occurs if we traverse each edge exactly once and return to the starting point. So all we need to do now is find an Eulerian Cycle (that exists since every node has in-degree = out-degree = 2), print the starting node and the labels on the edges in the order chosen by the Euler Cycle.

Time Complexity: $O(m)$

4.29 Teleporters Path

This is a constrained version of the normal Euler Path problem, we are given two nodes a and b and must find an Euler Path starting at a and ending at b. So we should only add a check that the node u that has outDegree(u) - inDegree(u) = 1 must be node a and the node v that has inDegree(v) - outDegree(v) = 1 must be node b and then check for an Euler Path.

Time Complexity: $O(m)$

4.30 Hamiltonian Flights

This is the classical Hamiltonian Path Problem. We will solve it using bitmask DP. Let $dp[i][mask]$ = Number of path that goes through every vertex in $mask$ exactly once and end at vertex i. Final answer is $dp[n-1][2^n-1]$ and base case is $dp[0][1] = 1$.

To compute the DP at a given state $dp[i][mask]$, iterate over all neighbors e of i not in mask and set $dp[e][mask|(1 << e)] = dp[e][mask|(1 << e)] + dp[i][mask] \pmod{10^9+7}$

Time Complexity: $O(n^2 2^n)$

4.31 Knight's Tour

The Knight's Tour problem is a classical mathematical problem in computer science. We have to find a Hamiltonian path in the graph built by the knight's possible moves on a chessboard.

There's no known polynomial time algorithm to find Hamiltonian paths in general graphs, but luckily enough, the knight tour problem on an 8x8 chessboard has a very efficient Heuristic called the Warnsdorff's rule.

We will brute force every possible move a knight can do from his position by order of fewest onward moves.

Time Complexity: Unknown.

4.32 Download Speed

Build a network where nodes are computers and connections are edges and each edge is given capacity c where c is the speed of the connection between both computers.

Now, considering node 0 as a source and node $n-1$ as a sink we get the standard Maximum Flow Problem. We can solve this problem using any efficient algorithm (Edmond-Karp, Dinic, ...)

Time Complexity: $O(mn^2)$ (Using Dinic) (Much faster in practice)

4.33 Police Chase

Build a network where nodes are crossings and edges are street, and give each edge unit capacity. Looking at node 0 as source and node $n-1$ as sink, the problem reduces to the standard Minimum Cut Problem.

By the Max-Flow Min-Cut Theorem it is equal to the Maximum Flow in this network and we can find it using any efficient algorithm (Edmond-Karp, Dinic, ...)

To retrieve the answer, perform BFS/DFS from the source, moving only along edges where flow was pushed. The desired streets are exactly those whose left endpoint is visited in our traversal but whose right endpoint is not.

Time Complexity: $O(mn^2)$ (Using Dinic) (Much faster in practice)

4.34 School Dance

Build a bipartite graph $G = (V_L, V_R, E)$ where V_L is the set of boys and V_R is the set of girls and we connect nodes $i \in V_L$ and $j \in V_R$ if and only if boy i and girl j are willing to dance together.

The problem reduces now to the standard Bipartite Matching Problem. We can solve it and retrieve the answer using any efficient algorithm (Kuhn, Hopcroft-Karp, ...)

Time Complexity: $O(m\sqrt{n})$ (Using Hopcroft-Karp) (Much faster in practice)

4.35 Distinct Routes

Build a network where nodes are rooms and edges are teleporters, and give each edge unit capacity. Looking at node 0 as source and node $n-1$ as sink, the problem reduces to the standard Path-Cover problem.

This problem is equivalent to the Maximum Flow Problem on unit-capacitaded networks and hence we can solve it using any efficient algorithm (Edmond-Karp, Dinic, ...) and retrieve the answer by extracting all paths from source to sink using DFS for example.

Time Complexity: $O(mn^2)$ (Using Dinic) (Much faster in practice)

5 Range Queries

Contents

5.1 Static Range Sum Queries

Compute the prefix sums of the array $pref[k] = \sum_{i=0}^{k} x[i]$. Now we can answer queries in O(1) since $\sum_{i=a}^{b} x[i] = \sum_{i=0}^{b} x[i] - \sum_{i=0}^{a-1} x[i] = pref[b] - pref[a-1]$.

Time Complexity: $O(n+q)$

5.2 Static Range Minimum Queries

We can solve this problem using a segment tree. Alternatively we can use the RMQ Data Structure to do so since the queries are static.

Time Complexity: $O(n + q \log n)$ (Using Segment Tree) or $O(n \log n + q)$ (Using RMQ)

5.3 Dynamic Range Sum Queries

We need a data structure that answers the sum queries while being able to handle update queries. We can use a Segment Tree where a node stores the sum of the corresponding subarray, or a Binary Indexed Tree (BIT).

Time Complexity: $O(n + q \log n)$

5.4 Dynamic Range Minimum Queries

We need a data structure that answers the minimum queries while being able to handle update queries. We can use a Segment Tree where a node stores the minimum of the corresponding subarray.

Time Complexity: $O(n + q \log n)$

5.5 Range XOR Queries

We will use the fact that XOR is associative and commutative. Calculate the prefix XOR of the array $pref[k] = \bigoplus_{i=0}^{k} x[i]$. Now we can answer queries in O(1) since $\bigoplus_{i=a}^{b} x[i] = \bigoplus_{i=0}^{b} x[i] \oplus \bigoplus_{i=0}^{a-1} x[i] = pref[b] \oplus pref[a-1]$.

Time Complexity: $O(n+q)$

5.6 Range Update Queries

We need to perform range increment updates while being able to find thee value at a specific index efficiently. We can use a Segment Tree with Lazy Propagation where a node in the tree stores sum/min/max or any subarray operation which is the identity operator when applied to a segment of size 1. The lazy value should store the value to increment the corresponding subarray.

Time Complexity: $O(n + q \log n)$

5.7 Forest Queries

We will use 2D Prefix Sums to solve the problem since queries are static. Calculate $pref[i][j]$ = Number of trees in the rectangle with opposite corners $(0, 0)$ and (i, j). For the first row/column this is standard prefix sums and then $pref[i][j] = pref[i-1][j] + pref[i][j-1] - pref[i-1][j-1] + v[i][j]$.

Now given a query (a, b) (c, d) we can find the answer in O(1) which is equal to
$pref[c][d] - pref[a-1][d] - pref[c][b-1] + pref[a-1][b-1]$.

Time Complexity: $O(n^2 + q)$

5.8 Hotel Queries

We need to find for each room requirement $r[i]$ the first hotel having the required number of rooms, or report that no such hotel is available. Build a Segment Tree on the array of available hotels where a node stores the maximum number of rooms in the corresponding subarray.

We can now model each group as the query described above. We can do binary search on the prefix length to find the required index in $O(\log^2 n)$, but we can do better.

First if the maximum of the whole array is less than $r[i]$ then there's no such hotel. Otherwise if the left child of the current node has maximum $>= r[i]$ then we proceed to that node since a suitable hotel exists in that region, otherwise we proceed to the right child.

After finding our index we should perform a single point update on our Segment Tree to decrement the number of available rooms in the corresponding hotel. This way we solve the problem by traversing the height of the segment tree at most 2 times.

Time Complexity: $O(n + q \log n)$

5.9 List Removals

Build a segment tree on the initial list of elements, where a node stores any function of the subarray which is identity on a subarray of size 1, and the number of elements not removed from the list.

On a removal query we can do binary search on the prefix length to find the desired element in $O(\log^2 n)$, but we can do better.

Assume we need to remove the current k-th element of the sublist represented by a node in the segment tree. If its left child has k or more elements that are still in the list, we proceed to that node since our desired element is in that part of the list. Otherwise we proceed to delete the $(k - \alpha)$-th element in the right node where α is the number of available elements in the left node.

After finding our index we should perform a single point update on our Segment Tree to decrement the number of elements in the corresponding sublists. This way we solve the problem by traversing the height of the segment tree at most 2 times.

Time Complexity: $O(n + q \log n)$

5.10 Salary Queries

The problem can be solved offline using compression and Segment Tree or BIT, but we describe an online approach.

We need to find how many elements of the array are in a given range while also being able to perform point updates. Since each query asks about the entire array, we can build a balanced BST (Treap or Order-Statistics Tree in C++ for example) so that we can find how many values in it are in a given range in logarithmic time.

To perform updates, simply remove any occurrence of the old value and insert the new value.

Time Complexity: $O(n \log n + q \log n)$

5.11 Prefix Sum Queries

Build a Segment Tree where each node stores the sum and maximum prefix sum of corresponding subarray. To merge two nodes L, R we set $sum = L.sum + R.sum$ and $prefMax = \max(L.prefMax, L.sum + R.prefMax)$. Single point updates can be performed as in any Segment Tree.

Time Complexity: $O(n + q \log n)$

5.12 Pizzeria Queries

Build 2 Segment Trees $st1$, $st2$ that store minimums of subarrays. Initialize $st1$ with $x[i] + i$ and $st2$ with $x[i] + n - 1 - i$. Single point updates can be handled as in any segment tree. As for the other query we have to query the minimum value to the left of the building from $st1$ (subtract i from it) and the minimum value to the right of the building from $st2$ (subtract $n - 1 - i$ from it) and take the minimum of both.

Time Complexity: $O(n + q \log n)$

5.13 Subarray Sum Queries

We will build a Segment tree based on the divide and conquer algorithm for Maximum Sum Subarray. Build a Segment tree that stores the maximum prefix sum and suffix sum, sum, and maximum sum subarray of corresponding subarray. Single point updates can be handled as in any Segment Tree.

To merge two nodes L, R we do the following: $sum = L.sum + R.sum$
$maxSum = \max(L.maxSum, R.maxSum, L.sufMax + R.prefMax)$
$prefMax = \max(L.prefMax, L.sum + R.prefMax)$
$sufMax = \max(R.sufMax, R.sum + L.sufMax)$

Time Complexity: $O(n + q \log n)$

5.14 Distinct Values Queries

We describe an offline solution that can be made online using Persistent Segment Trees. First compress the numbers into a small range instead of 10^9. Now sort the queries by right index and build an array $last$ where $last[i]$ = index of last occurrence of i in the original array so far. Build a Binary Indexed Tree (BIT) or Sum Segment Tree of size n that initially contains all 0's.

Traverse the array from left to right, At each index increment the index i and decrement the index $last[x[i]]$ (which would've been incremented once previously). This way at each index i every element has it's rightmost occurrence set to 1 and all other occurrences set to 0. We also answer at index i every query with right index equal to i.

Time Complexity: $O(n \log n + q \log n)$

5.15 Increasing Array Queries

We describe an offline solution. Sort the queries by decreasing left index, and build a segment tree of size n that stores sum, max and minimum number of operations to make corresponding subarray increasing. Initially all values in it are 0's.

Traverse the array from left to right. For each element $x[i]$, find the index idx of the last first element greater than $x[i]$ in the current array (Which can be done with one traversal of the tree depth described in previous solutions). What we have to do now is set all elements in the range $(i + 1, idx - 1)$ to $x[i]$ (Increasing the number of operations needed), set and then we answer all queries having i as left endpoint.

Notice that the way we constantly update the array ensures that it will always remain increasing. Updates can be handled by standard lazy propagation.

Time Complexity: $O(n \log n + q \log n)$

5.16 Forrest Queries II

We need a data structure that can support 2D range sums and point updates efficiently. We can use a 2D Binary Indexed Tree that answers all those queries in $O(\log^2 n)$

Time Complexity: $O(n^2 \log^2 n + q \log^2 n)$

5.17 Range Updates and Sums

All queries can be handled by a Segment Tree with lazy propagation, but we need to be careful since we have two types of update queries, Range Add and Range Set. When merging lazy values we should ensure that 1) A Range Set update overrides anything below it, and 2) A Range Add update adds to anything below it.

Time Complexity: $O(n + q \log n)$

5.18 Polynomial Queries

We want to solve this problem online using Segment Tree with lazy propagation. To do so, the segment tree's nodes will store the sum of corresponding subarrays and the lazy value will be a pair (a, r) which tells us to increment the corresponding range by an arithmetic sequence with first term a and common difference r.

Notice that the pair (a, r) uniquely defines an arithmetic sequence. To merge lazy values, we simply have to add the a's and the r's but we should find the correct value of a to be merged with subsequent children (an (a, r) pair propagates down as (a, r) to the left child and (b, r) to the right child where b is *roughly* the middle value of the previous arithmetic sequence, which can be found in $O(1)$). Applying lazy changes can also be done in $O(1)$ since arithmetic sequences have nice formulas for their sums.

Time Complexity: $O(n + q \log n)$

6 Tree Algorithms

Contents

6.1

7 Mathematics

Contents

8 String Algorithms

Contents

9 Geometry

Contents

10 Advanced Techniques

Contents

11 Additional Problems

Contents