培正 喇沙 編程挑戰賽

LA SALLE – PUI CHING PROGRAMMING CHALLENGE

6TH · 2022

AUG 13, 2022 (SATURDAY)

LA SALLE COLLEGE

SOLUTIONS

ID		NAME	AUTHOR / PREPARER	DIFFICULTY
A	🟩	Advertere Augmento	Yuen Lok Kan Ethen Yuen Lok Kan Ethen	★★☆☆☆
B	🟧	Balanced Splitting	Xie Lingrui Xie Lingrui	★★★★☆
C	🟦	Carpark	Wong Man Hang Wai Ka Hei	★★☆☆☆
D	⬛	Delivery	Lee Kai Fung Kevin Lee Kai Fung Kevin	★★★☆☆
E	🟦	Exclusive-or Merging	Cheng Hei Chit Cheng Hei Chit	★☆☆☆☆
F	🟪	Fall Guys	Wong Man Hang Wong Man Hang	★★★☆☆
G	🟨	Great Plummet	Ng Yau Fu Ng Yau Fu	★★★☆☆
H	🟥	HDL Shipping Service	Chung Wai Jit Chung Wai Jit	★★★★★★
I	🟥	I want to buy games!	Lee Ching Hei Yuen Lok Kan Ethen	★★★☆☆
J	🟧	Just Another FFT Problem	Wai Ka Hei Lee Cheuk Kit	★★★☆☆
K	🟦	Karaoke	Lee Ching Hei Chiu Long Hin Vincent	★☆☆☆☆
L	🟩	Linear Game	Yeung Man Tsung Yeung Man Tsung	★★☆☆☆

The authors expect that the problems were to be solved by

the following percentage of teams

From 1 to 6 stars: >75%, 50-75%, 25-50%, 5-25%, 1-5%, <1%

A - Advertere Augmento

If the gates only contain addition and subtraction. It is easily seen that you should always pick the gate that increases your current value the most. Such a greedy solution will also work when there only exists gates for multiplying positive numbers.

However, considering the following case:

```
Gate 1: + 1 + 1
Gate 2: * -2 + 1
Gate 3: * -2 + 1
```

A greedy solution would take `+ 1 + 1 + 1`, yielding the value of `+3`. But the optimal solution is to take `+ 1 * -2 * -2`, yielding the value of `+4`. Sometimes, the maximum value comes from multiplying the minimum value with a negative value!

You may have observed that the maximum and minimum achievable values after passing through each pair of gates must come from the maximum and minimum achievable values in the previous step.

So, for each step, simply maintain both the maximum and minimum achievable values, and output the maximum value after the last step.

B – Balanced Splitting

Solution

It is obvious that when the number of 0s or 1s is odd, then there is no solution. We claim that there is always a solution otherwise.

Let the queried substring $s[L..R]$ contains $2x$ occurrences of 0 and has length $2l$. It would be sufficient to find a substring $s[a..b]$ with $L \leq a \leq b \leq R$ of length l and containing x copies of 0.

We first define $f(a)$ as number of occurrences of 0 in $s[a..a+l-1]$, we want to find an index i such that $f(i) = x$.

We also define $M = \dfrac{L+R+1}{2}$ such that $s[L..M-1]$ and $s[M..R]$ both has length l.

We first consider $f(L)$ and $f(M)$. Since the two substrings are non-overlapping and their union is the whole queried substring, we get $f(L) + f(M) = 2x$.

Lemma 1. *For any i that satisfies $L \leq i < M$, we have $|f(i+1) - f(i)| \leq 1$*

Proof. We have $f(i+1) = f(i) + (s[i+L] \text{ is } 0) - (s[i] \text{ is } 0)$. Since $(s[i+L] \text{ is } 0) - (s[i] \text{ is } 0)$ can only take values of $-1, 0, 1$, the lemma must hold. □

Lemma 2. *For any i, j that satisfies $L \leq i \leq j < M$ and $f(i) \leq x \leq f(j)$, there exists a k with $i \leq k \leq j$ such that $f(k) = x$.*

Proof. Following lemma 1, this is equivalent to the discrete case of intermediate value theorem when we consider $g(a) = f(a) - x$. □

Without loss of generality, we assume that $f(L) \leq x \leq f(M)$, by lemma 2 we see that there exists an i with $L \leq i \leq M$ such that $f(i) = x$ and $s[i..i+l-1]$ would be the desired substring.

Such an index can be found using a method similar to the bisection method but for discrete functions.

Assuming that the search space is $[L..R]$ and that $f(L) \leq x \leq f(R)$, with initially the search space being $[L..M]$. In each step we check $m = \left\lfloor \dfrac{l+r}{2} \right\rfloor$ and the value of $f(m)$.

- If $f(m) = x$ then we are done.
- If $f(m) < x$ then we update the search range to $[m+1, R]$.
- If $f(m) > x$ then we update the search range to $[L, m-1]$.

The time complexity is analogous to that of binary search and is therefore $\mathcal{O}(\lg N)$ per query and therefore $\mathcal{O}(Q \lg N)$ in total.

Comments
This task was initially given a difficulty rating of three stars, but only two teams managed to solve this task in contest, and is the only task not solved by the champion team.

The core takeaway from this task is to avoid delving into standard algorithms and data structures too quickly but instead think of simpler methods first. In contest, there were unsuccessful long submissions (100+ lines) to this task using complex data structures, whereas the model solution is barely 20 lines and without anything more advanced than binary search and partial sum.

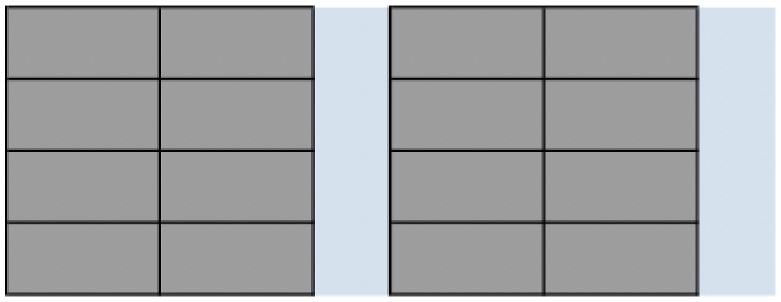
Extension
This task was inspired by a YouTube video by 3Blue1Brown.

In fact, as mentioned in the video, if the each character in the string can take on k different symbols with an even number of occurrences of each in the whole string, it is always possible to split the string into $k + 1$ substrings such that there exists a subset of the substrings which contains exactly half of each symbol. I could not find a generalised algorithm to solve this generalised problem, if you find one please message me on Discord @ `kevinxiehk`.

C - Carpack

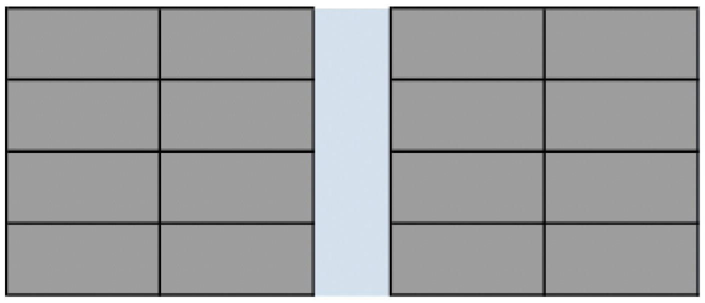
There are many possible solutions. One of them is as follows:

Firstly, we can consider only putting the parking spots horizontally (i.e. 1 × 2 regions). For each row, let’s try to put the parking spots from left to right. After we put two consecutive parking spots, we need to leave one extra cell to form internal roads. For example, the construction for a 4 × 10 grid is as follows:

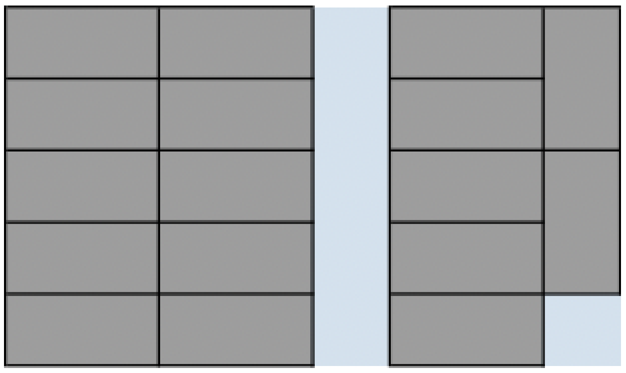


As you can see, the utilization is 80%. If *M* is divisible by 5, the problem is already solved.

Next, we need to consider the construction for *M* that is not divisible by 5. For *M* mod 5 = 2 and *M* mod 5 = 4, we can add some extra horizontal parking spots for each row. The utilization for the last 2 or 4 columns will be 100%. For example, the construction for a 4 × 9 grid is as follows:



For *M* mod 5 = 1 and *M* mod 5 = 3, the utilization for the last 1 and 3 columns will be 0% and 66.7% respectively, if we only put horizontal parking spots. So we need to utilize the remaining column by putting vertical parking spots (i.e. 2 × 1 regions). Following is the construction for a 5 × 8 grid:



If *N* is an even number, the utilization for the remaining column will be 100%. Otherwise, it will be (*N* - 1) / *N*, which is enough for most cases. The only invalid cases are when *N* = 3 and *M* mod 5 = 1. For such cases, the easiest way to handle them is to swap *N* and *M* before doing the above construction, since *N* won't be 3 anymore after swapping.

D — Delivery

Simplified Problem Statement

Given integers D and R, determine if there exists an undirected, unweighted graph with diameter D and radius R, and if so, output a possible graph.

Solution

Claim. *A solution exists if and only if* $\frac{D}{2} \leq R \leq D$.

Proof. First we show that any graph with diameter D and radius R satisfies the inequalities $\frac{D}{2} \leq R \leq D$. Note that $R \leq D$ is obviously true from the definitions of diameter and radius.

Define $d(x, y)$ to be the length of the shortest path from vertex x to vertex y. By definition, the following inequality holds for all x, y, z:

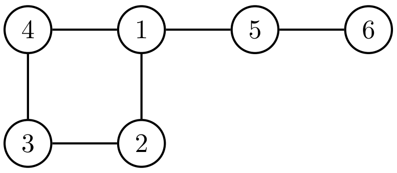
$$d(x, y) \leq d(x, z) + d(z, y)$$

Let u and v be two vertices such that $d(u, v) = D$, and w be a vertex such that $\text{inconvenience}(w) = R$. Now we can deduce

$$\begin{aligned} D = d(u, v) &\leq d(u, w) + d(w, v) \\ &= d(w, u) + d(w, v) \\ &\leq R + R \\ &= 2R \end{aligned}$$

Next we prove the converse by illustrating an explicit construction.

Consider a graph with a cycle of $2R$ vertices with a straight line of $(D - R)$ vertices attached to one of them. The cycle guarantees that the radius is exactly R (because $D - R \leq R$) and the straight line fulfills the requirement of the diameter being exactly D. $\square$



The figure above shows the construction for the sample case $D = 4, R = 2$.

Comments

This constructive task was designed to be accessible to the strongest teams but at the same time appealing to every team. The mathematical aspect of the task prevented a number of middle-ranking teams from solving it but the author deemed the purely constructive version (where teams need not determine if a solution exists) to be much less interesting.

The main challenges of this task are figuring out the correct inequalities and writing the seemingly straightforward solution very carefully. Indeed, all 8 teams who solved the task during the contest submitted at least one wrong attempt. Two common pitfalls are outputting duplicate edges for the cases with $R = 1$ and writing `if (D/2 <= R && R <= D)` (which is incorrect due to integer division) instead of `if (D <= 2*R && R <= D)`.

E - Exclusive-or Merging

First, since the xor operation is associative, the order of merging will not affect the result. Therefore, the problem can be seen as converting string S to string T by replacing some substrings in string S with their xor sum.

Since the order of merging doesn't matter, we can convert S to T from left to right using a greedy approach. For each digit in S, if it is the same as the corresponding digit of T, we can go to the next digit. Otherwise, we can try merge the current digit and the next digit in S, and check if they are the same again. If we can convert S to T by keep doing this, we can output "Yes". And if S becomes shorter than T before we have finished or there are some extra digits that cannot be merged after we have finished, we can output "No".

Note that even if there are some extra digits after we converted S to T, the answer isn't always "No". If the xor sum of the extra digits are 0, we can just merge all of them onto the last matched digit of S and the matched string will still be the same as T.

F - Fall Guys

To solve the main problem, we first need to solve a different problem: What is the current set of qualifying teams (according to the input). It can be solved using the following steps:

1. Calculate the score of each squad
2. Calculating the ranking
3. Determine the cut-off score
4. Obtain the set of qualifying squads

Extra care should be taken when determining the cut-off score. As specified in the problem statement, the cut-off score (minimum qualifying score) is

- The top score, if the top $M+1$ squads have the same score
- (Score of $M+1$ th squad) $+ 1$; otherwise

Once the cut-off score is determined, the set of qualifying squads is simply all the squads that have a score greater than or equal to the cut-off score.

Now let's solve the original problem. The intuition is that, as a player, you want to maximize your squad's score and hope that all other players fall into the slime.

Define X = current set of qualifying squads if no more player reach the finish line
Define Yi = new set of qualifying squads if the remaining players (if any) in Squad i are the next players that reach the finish line and all other remaining players in other squads fall into the slime.

The set of qualifying teams has been determined iff $X = Yi$ for all $i = 1..N$.

The task can also be solved by handling the various cases separately. However it is easy to miss some cases and receive Wrong Answer, this approach may not be ideal for an ICPC style contest.

G - Great Plummet

Suppose we know the answer for the interval (l, r), and we maintain the information about the interval in some data structure. To answer the next interval (l+1, r+1), we need to remove A[l] and insert A[r+1].

As we are interested in continuous decline, we may treat one range of continuous decline as one element. Suppose there are two ranges of them within the interval (l, r), let say (l1, r1) and (l2, r2), where l <= l1 <= r1 < l2 <= r2 <= r. If the sum of (l2, r2) is not less than (l1, r1), then for all intervals afterward, we can choose (l2, r2) instead of (l1, r1). Otherwise, we need to keep (l1, r1) and remove A[l1] from the sum when l1 < l.

We may store the ranges using a monotone queue. For every new range, i.e. A[r+1] is the first number of a range, we add one element equal to A[r+1] to the end of the queue. If A[r+1] is a subsequent number of the added range, we update the value of the element at the end of the queue. We also need to update the value of the front element if A[l] is negative, i.e. a number in the range we maintain. Finally, if we maintain the monotonicity of the queue, we can answer every interval with the value of the front element.

H - HDL Shipping Service

Notice that P and Q are at most 50, and the original graph is actually not important, as we don't care how the gifts are delivered. So, we could build a new table stating the shortest distance between the headquarter and gift's locations to each of the receive locations by running dijkstra (Careful that the distance from headquarter has to be multiplied by 2). In this way, we could shrink the graph into about P+Q nodes as these are the information that matters, and we could also treat the headquarter as a normal gift's location after the modification.

How can we determine if there is a way to deliver all gifts within K days? We can know by using a flow model: we can build a bipartite graph between the gift's locations to receive locations, and add an edge with infinite capacity between those two if their distance is less than K. Then, we could connect the source node and gift's locations with capacity A_i, and the sink node and receive locations with capacity B_i. If the max flow of the model equals the sum of A_i, then we know there is a way to deliver all gifts with K days. Therefore, we could just binary search on K by building a new bipartite graph each time. The time complexity is $O(V^2 * E * \log(D))$ by running max flow algorithm each time.

I - I want to buy games!

Notice that **game i** will be discounted on the **i-th day**. Also, we can never buy game i at a discounted price if i > D.

Let’s say we have decided which games we will buy, it is always optimal to buy Game i on the i-th day (if i <= D), and fill in the gap by buying games that are not going to get discounted.

Therefore, we can treat every game i where i <= D to have price P[i], while game i where i > D to have price A[i]. Then, we just need to simulate buying the game with the min(N, D) lowest price, and break whenever we already run out of money.

This can be easily achieved by sorting with ascending price and iterating through it.

Time complexity: O(N log N)

J - Just Another FFT Problem

Although the title contains FFT and the formula looks like something that requires using FFT to calculate, you don’t actually need to use FFT to solve this problem. In fact, you can’t use FFT (or NTT) to solve this problem.

A value of 1 will be added to A_i only if $S_k = T_{i-k+1}$, so we can consider each English letter separately. When considering only one English letter, you will discover that the task is equivalent to applying HPA multiplication to multiply two 0-1 strings to get a new array. The final array *A* will be the sum of the resulting arrays of all 26 letters.

HPA multiplication can be done in *O(N log N)* time (where *N* is the maximum length of the arrays) by using FFT or NTT. By applying it 26 times, we can solve the problem in *O(26N log N)* time, which is not enough given that *N* is large and the time limit is tight in this problem.

Observe that you only need to output one integer but not the whole array. Instead of compressing the array *A*, we can actually compress the 0-1 strings before doing multiplication, and multiply the two compressed numbers directly. The result will be the same since what we’ve done previously is just simple multiplication. The time complexity will be *O(26N)*.

K - Karaoke

The constraints of this problem are set to such that it can be solved by exhausting with four for-loops in O(T^4/(768ABC)) time. Note that *Golden Songs* is a song by itself, so there are four songs in total to choose from the screen (before it is locked), and ALL songs (A, B, C & *Golden Songs*) can be played an unlimited number of times, so it is possible to sing *Golden Songs* more than once for the singing time to be maximised.

Alternatively, one can solve this unbounded knapsack problem with Dynamic Programming in O(4T) time. Let dp[i] indicate whether it is possible to sing for exactly i seconds. Then dp[0] = 1, and for $0 \leq i \leq (T - 1)$ if dp[i] = 1 then the following will be set true: dp[i + 768], dp[i + A], dp[i + B], and dp[i + C]. The answer is 768 added to the highest satisfiable i ($0 \leq i \leq (T - 1)$).

L — Linear Game

Solution

The situation when there are no draws is fairly simple to handle – we could simply simulate the result of the game in $O(n)$ time by maintaining two pointers, the rightmost surviving player from team A and the leftmost surviving player from team B. However, problem arises when there are draws, as the randomly decided result would affect subsequent matches.

To solve this problem, we will need the following observation:

Observation 1. *If we assume that team A always wins when there are draws, the results would be optimal for all team A players (that is, the decision is "no worse" than letting team B win when there are draws).*

Proof. Consider the state when there is a draw. Let the players be $A_1, A_2, \ldots, A_N, B_1, B_2, \ldots, B_M$ from left to right (the first N players are from team A and the last M players are from team B). The draw occurs between A_N and B_1.

Assume A_x eventually survives after B_1 wins in the draw. We can infer that there exists a sequence $1 = s_N \leq s_{N-1} \leq s_{N-2} \leq \ldots \leq s_x = M$ such that

- A_{N-1} wins $s_N + 1, s_N + 2, \ldots, s_{N-1}$.
- A_{N-2} wins $s_{N-1} + 1, s_{N-1} + 2, \ldots, s_{N-2}$.
- $\ldots$
- A_x wins $s_{x+1} + 1, s_{x+1} + 2, \ldots, s_x$.

Now, assume A_N wins in the draw instead. We consider two cases:

1. A_N **never loses in subsequent games:** As A_N survives, we can obviously know that A_x survives as well.
2. A_N **loses to a player** B_y**:** Find the greatest i such that $s_i \geq y$. By the definition of s_i above, we know that players $A_i, A_{i-1}, \ldots, A_x$ would be able to beat players $B_y, B_{y+1}, \ldots, B_M$ (details are left as exercise). Hence, A_x can survive in the game.

This shows that allowing A_N to win in the draw would be "no worse" than allowing B_1 to win in the draw. $\square$

Note that the same observation holds for team B by symmetry. Thus, we can simulate the game two times, assuming that team A or team B wins in all draws in each respective simulation. Combine the two results to obtain the number of players who can possibly win (Note: some players may win in both simulations). The solution takes $O(n)$ time as only two simulations are involved.

Comments

The task is rated three stars at the initial stage. As a task that requires more about observations rather than implementation, the performance in general is somewhere around expected. However, it is observed that some teams did not think carefully enough before implementing their greedy solutions, resulting in incorrect attempts. When approaching observation tasks, teams are encouraged to informally justify the correctness of their observations or approaches by coming up with hack cases or sketching proof ideas.