The CENTRE for EDUCATION in MATHEMATICS and COMPUTING

cemc.uwaterloo.ca

2023 CCC Senior Problem Commentary

This commentary is meant to give a brief outline of what is required to solve each problem and what challenges may be involved. It can be used to guide anyone interested in trying to solve these problems, but is not intended to describe all the details of a correct solution. We encourage everyone to try and implement these details on their own using their programming language of choice.

S1 Trianglane

To help Bocchi determine how much tape she needs, we need to do some careful counting.

For the first subtask, we can count B, the number of black triangles, which is the number of times 1 appears in the input. The correct answer will be $3 \times B$. Note that we can ignore the second line of input which is also true for the second subtask. However, for this second subtask, there could be adjacent black triangles and Bocchi does not need to place warning tape on the common side in these cases. Notice that each of these common sides contributes two to the value of $3 \times B$. This means we obtain the correct final answer by subtracting two from $3 \times B$ for each pair of adjacent black triangles.

The same principle applies to the third subtask but we also have to look for black triangles that are adjacent but in different rows. These can only occur at even-numbered positions (assuming we start counting at zero).

It is possible to solve this problem without making the observation that you can subtract from $3 \times B$. This approach involves looking for all the places where warning tape needs to be placed. An algorithm is needed which includes sides of triangles on the perimeter of the pathway as well as sides between adjacent triangles which are not the same colour.

Regardless of which approach is taken, many solutions to the first three subtasks will be quite efficient. However, the fourth subtask is meant to ensure this by disallowing overly slow solutions. It specifically requires that not too many passes are made over either row. More formally, only a constant number of passes of each row is allowed. Solutions that miss this requirement are probably too slow because they use a built-in function within a loop which itself loops through a row of triangles.

S2 Symmetric Mountains

To solve this problem, we will find the asymmetric value for each crop and compute the minimum asymmetric value for each crop's corresponding length. For every subtask, we find different ways to compute the asymmetric value of each crop.

Subtask 1

Considering iterating over all crops. A crop is defined by its left endpoint and right endpoint. Hence, we can use a nested for loop to first fix the left endpoint and then fix the right endpoint. There are other ways to loop through crops, such as looping through its left endpoint and its length and vice versa. To find the asymmetric value of a crop, we can have an accumulator and sum up the absolute difference for every such pair by looping over i defined in the problem statement. There are $N \cdot (N-1)/2 = N^2/2 - N/2$ such crops, and we need about N calculations on average to to compute the asymmetric value of each crop; hence, it runs in time that is roughly cubic in the number of mountains.

Subtask 2

For this subtask, we note that if we were to find the asymmetric value of each crop, then we have to do it faster than roughly N time. Now consider a crop with a left endpoint of l and a right endpoint of r. We will take a closer look at the formula to compute the asymmetric value:

$$|h_l - h_r| + |h_{l+1} - h_{r-1}| + |h_{l+2} - h_{r-2}| + \ldots + |h_{l+t} - h_{r-t}|$$

where $t = \lfloor \frac{r-l}{2} \rfloor$. Note that because the array is non-decreasing in this subtask, then it follows that for mountains with indices $i \leq j$, we have that $|h_i - h_j| = h_j - h_i$. Hence the above formula is equivalent to

$$(h_r - h_l) + (h_{r-1} - h_{l+1}) + (h_{r-2} - h_{l+2}) + \ldots + (h_{r-t} - h_{l+t}).$$

Let $m = \frac{l+r}{2}$, which represents the midpoint of the mountains (it may not be an integer, but that does not matter). Notice that for every term whose index is greater than m, it is being added, and for every term whose index is less than m it is being subtracted. We can then rearrange the formula to become

$$h_r + h_{r-1} + h_{r-2} + \ldots + h_{r-t} - (h_l + h_{l+1} + h_{l+2} + \ldots + h_{l+t}).$$

Note that we have simplified this problem into static range sum queries. We can compute them quickly by building a prefix sum array of the mountains' heights (i.e., an array where the kth element is the sum of the first k elements of h) before iterating through all the crops. Note that to calculate the asymmetric crop, in this case, it will be constant time (i.e., independent of N), so this runs in quadratic time in the number of mountains.

Subtask 3

We will now let $[l, r]$ represent a crop where $l \leq r$ are the left and right endpoints, respectively. Now for this subtask, let us take a closer look at the formula for the crop $[l, r]$:

$$|h_l - h_r| + (|h_{l+1} - h_{r-1}| + |h_{l+2} - h_{r-2}| + \ldots + |h_{l+t} - h_{r-t}|).$$

Notice that the terms in brackets are the asymmetric value of the crop $[l + 1, r - 1]$. If we have a way to iterate through the crops so that crop $[l + 1, r - 1]$ comes right before $[l, r]$, we can add $|h_l - h_r|$ directly to the accumulator to get the asymmetric value of crop $[l, r]$. This motivates us to cleverly iterate over the crops by fixing the midpoint and expanding outwards while maintaining an accumulator for each midpoint. Note that the midpoint will not be an actual mountain for crops with even length. This runs in time which is the square of the number of mountains. As we move from crop $[l + 1, r - 1]$ to $[l, r]$, we only add a term to the accumulator. Note that there exist dynamic programming solutions, but these were not necessary to receive full marks.

S3 Palindromic Poster

Subtask 1

For this subtask, since $R = 1$ and $C = 1$, we can try and make only the first row and column a palindrome. One way to do this is to fill the first row and column with the letter `a` and fill the rest of the grid with the letter `b`.

Subtask 2

Because there are so few cases, it suffices to solve this subtask on paper and hardcode the answers. To reduce the amount of casework, notice that if we have a solution for (R, C), we can get a solution for (C, R) by flipping the grid.

La version française figure à la suite de la version anglaise.

Another possible approach is to write a brute force since there are only four important possibilities for each cell.

Subtask 3

This subtask is intended to aid students in thinking about the problem.

Subtask 4

The solutions to subtasks 1 and 2 should give some intuition here. From subtask 1, we propose the following general solution:

Fill the first R rows and the first C columns with the letter `a` and fill the rest of the grid with the letter `b`.

We notice that this fails when $R = 0$, $C = 0$, $R = N$ or $C = M$. We can handle these in pairs since we can flip the grid.

If $R = 0$, we can fill the first $M-1$ columns with the letter `a`, and the last column with the letter `b`, and then increment the last $M - C$ characters of the last row. For instance, for the input `4 4 0 2` we can output:

```
aaab
aaab
aaab
aabc
```

If $R = N$, every row must be a palindrome. Because of this, starting from a full grid of `a` characters, we can easily reduce the number of palindromic columns by two at a time by making matching columns of the first row `b` characters. For instance, for the input `4 4 4 2`, we can output:

```
baab
aaaa
aaaa
aaaa
```

However, this fails if C is not the same parity as M, that is, if we cannot get C by subtracting 2 an integer number of times from M.

In this case, it depends: if M is odd, we can use the middle column. For instance, for the input `5 5 5 2`:

```
babab
aaaaa
aaaaa
aaaaa
aaaaa
```

However, the input `4 4 4 1` is impossible since there is no middle column.

We can handle $C = 0$ and $C = M$ symmetrically by flipping the inputs, processing, and then flipping the output.

S4 Minimum Cost Roads

Subtask 1

This subtask can be solved by running an MST (Minimum Spanning Tree) algorithm that only considers

the cost of each edge. Note that we may have multiple components, so our final answer will be the sum of costs of the MST of each component. Luckily, an algorithm like Kruskal's handles this problem without any additional complexity.

Subtask 2

The constraints of this subtask allow us to focus on an invariant that is very useful in the long run. Consider a single edge $e = (a, b, l, c)$. We get the following cases:

- If e is the unique shortest path from a to b, we definitely want to take e;
- If there exists a path from a to b with length $l' < l$, we definitely don't want to take e;
- If there exists another path from a to b with length $l' = l$ (but no paths that are shorter), we still don't want to take e. However, this time the proof is much more complex:

  Suppose p is a path from a to b consisting of edges $\{p_1, p_2, \ldots, p_k\}$ such that

  $$\sum_{i=1}^{k-1} \text{length}(p_i) = l$$

  Now, suppose that $\forall i, p_i$ is the unique shortest path between its endpoints $a(p_i)$ and $b(p_i)$. If this is not the case for some i, then we can find another path from $a(p_i)$ to $b(p_i)$ with length exactly length(p_i) and replace p_i with that path (if that path had length $<$ length(p_i) then we would have a path from a to b with length $< l$, which is a contradiction). Observe that this process can only be repeated a finite number of times.

  In the end, we have a path from a to b that has length l, does not take the edge e, and every edge in that path is the unique shortest path between its endpoints. We'll call the edges on this path $q_1, \ldots, q_m$.

  Now consider some optimal answer that contains the edge e. Clearly, $q_1, \ldots, q_m$ must be in the answer. However, this means that we can remove e as any time we need to traverse from a to b, we can take the edges $q_1, \ldots, q_m$ instead. This violates the optimality of the answer and is thus a contradiction.

This results in a very simple solution for this subtask: we loop over every edge $e = (a, b, l, c)$ and check whether it's the unique shortest path between a and b. One way to do this is to first run Dijkstra's algorithm n times to find the shortest path between every pair of nodes (and store it in a 2-D array *dist*). Then, for each edge we check if

$$l < \min_{1 \le i \le n, i \ne a, i \ne b} (\text{dist}[a][i] + \text{dist}[i][b])$$

Remark: Notice that we don't even care about costs at all! In fact, the original problem didn't have costs ($l_i = c_i$), but we thought it was an interesting extension :)

Subtask 3

Unfortunately, we lose the guarantees that allow our solution from subtask 2 to work. Fortunately, we can restore these guarantees with some clever reductions.

First, let's look at the graph formed by all of the 0-length edges. If any pair of nodes (a, b) are connected in this graph, then our answer must contain a path of length 0 between a and b. Thus, we must choose a

subset of the 0-length edges that has the same connectivity properties as the original graph (i.e. if (a, b) are connected by 0-length edges in the original graph, then they must be connected by 0-length edges in our answer). Trying to do so while also minimizing cost is analogous to finding the minimum spanning forest of the graph, which is the same as our solution in subtask 1.

Unfortunately, the 0-length edges are still part of our graph. In order to remove them for good, we must make the following observation: we can treat each component in our spanning forest as a single node as we're able to travel between any two nodes in that component with a path of length 0. This motivates us to compress our initial graph by these components to remove all 0-length edges.

Note: Be careful, the compressed graph may also contain self-edges so make sure to ignore them.

Next, to remove multiedges notice that we only ever want at most one edge between any pair of nodes. Thus, we take the one with lowest length, breaking ties by cost.

Finally, we run our solution from subtask 2 and sum the answer we obtained from that solution with the one from computing the minimum spanning forest.

Alternative Solution

There is also an alternative solution much closer to Kruskal's algorithm for computing MST: we loop over the edges in order of length, breaking ties by cost and adding it to the graph if it improves the shortest path between its endpoints. This solution also has the added benefit of not worrying about 0-length edges or multi-edges separately.

S5 The Filter

Subtask 1

The answers for $N = 3$ are $0, 1, 2, 3$.

Suppose the answers for $N = 3^k$ are $a_1, a_2, \ldots, a_M$. The answers for the first third of $N = 3^{k+1}$ are

$$a_1, a_2, \ldots, a_M$$

The Cantor set has copies of the same structure. In particular, the first third of the Cantor set is identical to the last third. The remaining answers for $N = 3^{k+1}$ are

$$2 \times 3^k + a_1, 2 \times 3^k + a_2, \ldots, 2 \times 3^k + a_M$$

These observations are enough to generate the answer for any power of 3.

Subtask 2

Note that $\frac{34676}{51169}$ is covered by the 49^{th} filter, but not any earlier filter. This is the maximum record for $N \leq 10^5$. A naive approach is expected to fail on $N = 51169$.

This subtask requires deeper observations about Alice's filters.

Property 1 (Filters are symmetric). If r is covered by the k-th filter, then $1 - r$ is covered by the k-th filter. Likewise, if the k-th filter does not cover r, then $1 - r$ is not covered by the k-th filter.

Property 2 (The k-th filter and $(k + 1)$-th filter are related). Let k be a positive integer, and let $0 \leq r \leq \frac{1}{3}$. If r is covered by the $(k + 1)$-th filter, then $3r$ is covered by the k-th filter. Likewise, if r is not covered by the $(k + 1)$-th filter, then $3r$ is not covered by the k-th filter.

La version française figure à la suite de la version anglaise.

The proof of these 2 properties is left as an exercise. From these properties, it becomes natural to define the function $f(r) = 3 \times \min(r, 1 - r)$. Here are a few theorems about f.

Theorem 1. If r is not covered by any filter, then $f(r)$ is not covered by any filter.
Proof. Apply Property 1 and Property 2.

Theorem 2. Suppose r is covered by the k-th filter. Then either $k = 1$, or $f(r)$ is covered by the $(k-1)$-th filter.
Proof. Apply Property 1 and Property 2.

Here is the approach to solving subtask 2:

- Let $r = \frac{x}{N}$. Construct this sequence: $r, f(r), f(f(r)), f(f(f(r))), \ldots$
- If r is in the Cantor set, then by Theorem 1, every term in the sequence is in the Cantor set. Since every term is a multiple of $\frac{1}{N}$, the sequence will enter a cycle.
- If r is not in the Cantor set, then r is covered by a filter. By Theorem 2, a term in the sequence will be covered by the first filter.

There are 2 outcomes. Either the sequence will enter a cycle, or a term will be covered by the first filter. The algorithm needs to handle both outcomes. The intended time complexity is $O(N)$ or slightly slower.

Alternatively, it is enough to ignore cycle detection and construct the first 49 terms. In the worst case (i.e. $r = \frac{34676}{51169}$), the 49-th term is covered by the first filter.

Subtask 3

Note that $\frac{222534799}{867579680}$ is covered by the 130^{th} filter, but not any earlier filter. The author believes this is the maximum record for $N \leq 10^9$.

Firstly, use the first 18 filters to remove most of the numbers. After this step, less than 10^6 numbers remain.

Next, apply the algorithm from subtask 2 on each of the remaining numbers. It may be a good idea to use memoization to improve time complexity. There are other tricks to improve performance.

The intended time complexity is $O(N^{\log_3 2} \log N)$ because there are $O(N^{\log_3 2})$ numbers to consider, and each number takes roughly $O(\log N)$ to consider.

The CENTRE for EDUCATION
in MATHEMATICS and COMPUTING
cemc.uwaterloo.ca

Commentaire sur le CCI de niveau sénior

L'objectif de ce commentaire est de donner un bref aperçu des éléments nécessaires pour résoudre chaque problème et des défis à relever. Les notes ci-dessous peuvent être utilisées pour guider toute personne qui souhaite tenter de résoudre ces problèmes. Or, elles n'ont pas pour but de décrire tous les détails d'une solution correcte. Nous encourageons toute personne intéressée à essayer d'élaborer une solution correcte en utilisant le langage de programmation de son choix.

S1 Allée de triangles

Pour aider Bocchi à déterminer le nombre de mètres de ruban de signalisation dont elle aura besoin, il faut que l'on procède à un comptage minutieux.

Pour la première sous-tâche, on peut compter B, qui correspond au nombre de fois où 1 parait dans l'entrée. La réponse correcte sera $3 \times B$. Remarquons que l'on peut ignorer la deuxième ligne des données d'entrée, ce qui est également vrai pour la deuxième sous-tâche. Cependant, pour cette deuxième sous-tâche, il pourrait y avoir des triangles noirs adjacents. Dans ce cas, Bocchi n'a pas besoin de placer du ruban de signalisation sur le côté commun. Remarquons que chacun de ces côtés communs augmente de 2 la valeur de $3 \times B$. Cela signifie qu'on obtient la réponse finale correcte en soustrayant deux de $3 \times B$ pour chaque paire de triangles noirs adjacents.

Le même principe s'applique pour la troisième sous-tâche, mais on doit également rechercher les triangles noirs qui sont adjacents mais dans des rangées différentes. Ces triangles peuvent uniquement se trouver dans les positions paires (en supposant que l'on commence à compter à partir de zéro).

On peut résoudre ce problème sans nécessairement remarquer qu'on peut effectuer des soustractions à $3 \times B$. Cette approche consiste à rechercher tous les endroits où il faut placer du ruban de signalisation. Il faut utiliser un algorithme qui inclut les côtés des triangles qui sont situés sur les bords de l'allée ainsi que les côtés entre les triangles adjacents qui ne sont pas de la même couleur.

Quelle que soit l'approche adoptée, il existe plusieurs solutions aux trois premières sous-tâches qui sont très efficaces. Cependant, la quatrième sous-tâche a pour but de garantir cette efficacité en interdisant les solutions trop lentes. Elle impose en particulier que le nombre de passages sur chaque rangée ne soit pas trop élevé et précise que seul un nombre constant de passages est autorisé. Les solutions qui ne respectent pas cette contrainte sont probablement trop lentes car elles utilisent une fonction intégrée dans une boucle qui elle-même passe en boucle sur une rangée de triangles.

S2 Montagnes symétriques

Pour résoudre ce problème, on va déterminer la valeur asymétrique de chaque photo recadrée et on va calculer la valeur asymétrique minimale pour la longueur correspondante de chaque photo recadrée. Pour chaque sous-tâche, on va trouver différentes façons de calculer la valeur asymétrique de chaque photo recadrée.

Sous-tâche 1

Considérons le fait d'effectuer des itérations sur toutes les photos recadrées. Une photo recadrée est définie par son extrémité gauche et son extrémité droite. Ainsi, on peut utiliser une boucle "for" imbriquée pour

fixer d'abord le point d'extrémité gauche, puis le point d'extrémité droit. Il existe d'autres façons de boucler à travers les photos recadrées, comme boucler à travers son extrémité gauche et sa longueur et vice versa. Pour trouver la valeur asymétrique d'une photo recadrée, on peut utiliser un accumulateur et additionner la différence absolue pour chaque paire en bouclant sur i défini dans l'énoncé du problème. Il y a $N \cdot (N-1)/2 = N^2/2 - N/2$ telles photos recadrées et on a besoin d'environ N calculs en moyenne pour calculer la valeur asymétrique de chaque photo recadrée; par conséquent, cela s'exécute en un temps qui est approximativement cubique en fonction du nombre de montagnes.

Sous-tâche 2

Pour cette sous-tâche, on doit remarquer que s'il faut trouver la valeur asymétrique de chaque photo recadrée, alors on doit le faire plus rapidement qu'en un temps d'environ N. Considérons une photo recadrée ayant une extrémité gauche de l et une extrémité droite de r. On va examiner de plus près la formule pour calculer la valeur asymétrique:

$$|h_l - h_r| + |h_{l+1} - h_{r-1}| + |h_{l+2} - h_{r-2}| + \ldots + |h_{l+t} - h_{r-t}|$$

où $t = \lfloor \frac{r-l}{2} \rfloor$. Remarquons que puisque le tableau est non décroissant dans cette sous-tâche, il en découle que pour les montagnes avec les indices $i \leq j$, on a que $|h_i - h_j| = h_j - h_i$. Donc, la formule ci-dessus est équivalente à

$$(h_r - h_l) + (h_{r-1} - h_{l+1}) + (h_{r-2} - h_{l+2}) + \ldots + (h_{r-t} - h_{l+t}).$$

Soit $m = \frac{l+r}{2}$, ce qui représente le milieu des montagnes (et qui ne peut être un entier quoique cela n'a pas d'importance). Remarquons que pour chaque terme dont l'indice est supérieur à m, il est ajouté, et pour chaque terme dont l'indice est inférieur à m, il est soustrait. On peut donc récrire la formule de la manière suivante:

$$h_r + h_{r-1} + h_{r-2} + \ldots + h_{r-t} - (h_l + h_{l+1} + h_{l+2} + \ldots + h_{l+t}).$$

Remarquons que l'on a simplifié ce problème en requêtes de somme de plage statique. On peut les calculer rapidement en construisant un tableau de sommes préfixes des hauteurs des montagnes (c'est-à-dire un tableau où le $k^{\text{ième}}$ élément est la somme des k premiers éléments de h) avant d'itérer sur toutes les photos recadrées. Remarquons que pour calculer la photo recadrée asymétrique dans ce cas, cela prendra un temps constant (c'est-à-dire indépendant de N) et donc cela s'exécute en un temps quadratique par rapport au nombre de montagnes.

Sous-tâche 3

Pour cette sous-tâche, soit $[l, r]$ une photo recadrée où $l \leq r$ sont respectivement l'extrémité gauche et l'extrémité droite. Pour cette sous-tâche, examinons de plus près la formule pour la photo recadrée $[l, r]$:

$$|h_l - h_r| + (|h_{l+1} - h_{r-1}| + |h_{l+2} - h_{r-2}| + \ldots + |h_{l+t} - h_{r-t}|).$$

Remarquons que les termes entre parenthèses sont la valeur asymétrique de la photo recadrée $[l+1, r-1]$. Si on peut itérer à travers les photos recadrées de manière que la photo recadrée $[l+1, r-1]$ soit juste avant $[l, r]$, on peut ajouter $|h_l - h_r|$ directement à l'accumulateur pour obtenir la valeur asymétrique de la photo recadrée $[l, r]$. Cela nous pousse à itérer astucieusement sur les photos recadrées en fixant le point central et en élargissant vers l'extérieur tout en maintenant un accumulateur pour chaque point central. Remarquons que le point central ne sera pas une montagne pour les photos recadrées de longueur paire. Ceci s'exécute en temps qui est égal au carré du nombre de montagnes. Lorsqu'on passe de la photo recadrée $[l+1, r-1]$ à $[l, r]$, on n'ajoute qu'un terme à l'accumulateur. Remarquons qu'il existe des solutions qui emploient la programmation dynamique, mais elles n'étaient pas nécessaires pour obtenir la note maximale.

S3 Affiche palindromique

Sous-tâche 1

Pour cette sous-tâche, étant donné que $R = 1$ et $C = 1$, on peut essayer de faire en sorte que seule la première rangée et la première colonne soient un palindrome. Pour ce faire, on peut remplir la première rangée et la première colonne avec la lettre `a` et remplir le reste de la grille avec la lettre `b`.

Sous-tâche 2

Comme il y a très peu de cas, il suffit de résoudre cette sous-tâche sur papier et de coder en dur les réponses. Pour réduire la quantité de cas, remarquons que si on a une solution pour (R, C), on peut obtenir une solution pour (C, R) en inversant la grille.

Une autre approche possible est d'employer une méthode force brute car il n'y a que quatre possibilités importantes pour chaque cellule.

Sous-tâche 3

Cette sous-tâche vise à aider les étudiants à réfléchir sur le problème.

Sous-tâche 4

Les solutions aux sous-tâches 1 et 2 devraient susciter l'intuition. À partir de la sous-tâche 1, on propose la solution générale suivante:

Remplir les R premières rangées et les C premières colonnes avec la lettre `a` et remplir le reste de la grille avec la lettre `b`.

On remarque que cela échoue lorsque $R = 0$, $C = 0$, $R = N$ ou $C = M$. On peut les traiter par paires puiqu'on peut inverser la grille.

Si $R = 0$, on peut remplir les $M - 1$ premières colonnes avec la lettre `a` et la dernière colonne avec la lettre `b`, puis incrémenter les $M - C$ derniers caractères de la dernière rangée. Par exemple, pour l'entrée `4 4 0 2`, on peut avoir comme sortie:

```
aaab
aaab
aaab
aabc
```

Si $R = N$, chaque rangée doit être un palindrome. Pour cette raison, en partant d'une grille complète de caractères `a`, on peut facilement réduire le nombre de colonnes palindromiques de deux à la fois en faisant des colonnes correspondantes de caractères `b` de la première rangée. Par exemple, pour l'entrée `4 4 4 2`, on peut avoir comme sortie:

```
baab
aaaa
aaaa
aaaa
```

Cependant, cela échoue si C n'a pas la même parité que M, c'est-à-dire si on ne peut pas obtenir C en soustrayant 2 un nombre entier de fois de M.

Dans ce cas, cela dépend: si M est impair, on peut utiliser la colonne du milieu. Par exemple, pour l'entrée `5 5 5 2`:

```
babab
aaaaa
aaaaa
aaaaa
aaaaa
```

Cependant, l'entrée `4 4 4 1` est impossible car il n'y a pas de colonne du milieu.

On peut traiter $C = 0$ et $C = M$ symétriquement en retournant les entrées, en les traitant, puis en retournant la sortie.

S4 Routes à coût minimum

Sous-tâche 1

Cette sous-tâche peut être résolue en exécutant un algorithme ACM (Arbre Couvrant Minimum) qui ne prend en compte que le coût de chaque arête. Remarquons que l'on peut avoir plusieurs composantes, donc notre réponse finale sera la somme des coûts de l'arbre ACM de chaque composante. Heureusement, l'algorithme de Kruskal peut être employé pour résoudre ce problème sans complexité supplémentaire.

Sous-tâche 2

Les contraintes de cette sous-tâche nous permettent de nous concentrer sur un invariant qui sera très utile. Considérons une seule arête $e = (a, b, l, c)$. On a les cas suivants:

- Si e est le chemin le plus court unique de a à b, on veut définitivement emprunter e;
- S'il existe un chemin de a à b avec une longueur $l' < l$, on ne veut définitivement pas emprunter e;
- S'il existe un autre chemin de a à b avec une longueur $l' = l$ (mais pas de chemins plus courts), on ne veut toujours pas emprunter e. Cependant, cette fois, la démonstration est beaucoup plus complexe:

  Supposons que p est un chemin de a à b composé des arêtes $\{p_1, p_2, \ldots, p_k\}$ tel que

  $$\sum_{i=1}^{k-1} \text{longueur}(p_i) = l$$

  Supposons maintenant que $\forall i, p_i$ est le chemin le plus court unique entre ses extrémités $a(p_i)$ et $b(p_i)$. Si ce n'est pas le cas pour certains i, alors on peut trouver un autre chemin de $a(p_i)$ à $b(p_i)$ avec une longueur exactement égale à longueur(p_i) et remplacer p_i par ce chemin (si ce chemin avait une longueur $<$ longueur(p_i), alors on aurait un chemin de a à b avec une longueur $< l$, ce qui est une contradiction). Remarquons que ce processus ne peut être répété qu'un nombre fini de fois.

  Enfin, on a un chemin de a à b qui a une longueur de l, qui n'emprunte pas l'arête e, et chaque arête dans ce chemin est le plus court chemin entre ses extrémités. Soit $q_1, \ldots, q_m$ les arêtes sur ce chemin.

  Considérons maintenant une réponse optimale qui contient l'arête e. Clairement, $q_1, \ldots, q_m$ doivent faire partie de la réponse. Cependant, cela signifie que l'on peut supprimer e car à chaque fois que l'on doit aller de a à b, on peut emprunter les arêtes $q_1, \ldots, q_m$ à la place. Cela nuit à l'optimalité de la réponse et constitue donc une contradiction.

On a donc une solution très simple pour cette sous-tâche: on boucle à travers chaque arête $e = (a, b, l, c)$ et on vérifie si c'est le chemin le plus court unique entre a et b. Une façon de le faire consiste à d'abord exécuter l'algorithme de Dijkstra n fois pour trouver le chemin le plus court entre chaque paire de nœuds (et le stocker dans un tableau bidimensionnel *dist*). Ensuite, pour chaque arête, on vérifie si

$$l < \min_{1 \le i \le n, i \ne a, i \ne b} (\text{dist}[a][i] + \text{dist}[i][b])$$

Remarque: Remarquons que l'on ne se soucie pas du tout des coûts! En fait, le problème original n'avait pas de coûts ($l_i = c_i$), mais on a pensé que c'était une extension intéressante :)

Sous-tâche 3

Malheureusement, on perd les garanties qui permettent à notre solution de la sous-tâche 2 de fonctionner. Heureusement, on peut rétablir ces garanties grâce à quelques réductions astucieuses.

Tout d'abord, examinons le graphe formé par toutes les arêtes de longueur 0. Si une paire de nœuds (a, b) sont connectés dans ce graphe, alors notre réponse doit contenir un chemin de longueur 0 entre a et b. Donc, on doit choisir un sous-ensemble des arêtes de longueur 0 qui ont les mêmes propriétés de connectivité que le graphe d'origine (c'est-à-dire si (a, b) sont connectés par des arêtes de longueur 0 dans le graphe d'origine, alors ils doivent être connectés par des arêtes de longueur 0 dans notre réponse). Essayer de le faire tout en minimisant le coût équivaut à trouver la forêt couvrante de poids minimal du graphe, ce qui est pareil à notre solution dans la sous-tâche 1.

Malheureusement, les arêtes de longueur 0 font toujours partie de notre graphe. Pour les supprimer définitivement, on doit faire l'observation suivante: on peut traiter chaque composante de notre forêt couvrante comme un seul nœud car on peut voyager entre deux nœuds dans cette composante en empruntant un chemin de longueur 0. Cela nous pousse à compresser notre graphe initial par ces composantes pour supprimer toutes les arêtes de longueur 0.

Remarque: Attention, le graphique compressé peut également contenir des auto-arêtes, veillez à les ignorer.

Ensuite, pour supprimer les multi-arêtes, remarquons que l'on ne veut jamais avoir plus d'une arête entre une paire de nœuds. Donc, on choisit celle avec la plus petite longueur et, en cas d'égalité, on choisit celle avec le coût le plus faible.

Enfin, on exécute notre solution de la sous-tâche 2 et on additionne la réponse que l'on a obtenue avec celle du calcul de la forêt couvrante de poids minimal.

Solution alternative

Il existe également une solution alternative qui se rapproche beaucoup plus de l'algorithme de Kruskal pour le calcul du ACM: on boucle à travers les arêtes dans l'ordre de leur longueur, en brisant les égalités en fonction du coût et en les ajoutant au graphe si elles améliorent le chemin le plus court entre ses extrémités. Cette solution présente également l'avantage de ne pas avoir à se soucier séparément des arêtes de longueur 0 ou des multi-arêtes.

S5 Le filtre

Sous-tâche 1

Les réponses pour $N = 3$ sont $0, 1, 2, 3$.

Supposons que les réponses pour $N = 3^k$ soient $a_1, a_2, \ldots, a_M$. Les réponses pour le premier tiers de $N = 3^{k+1}$ sont

$$a_1, a_2, \ldots, a_M$$

L'ensemble de Cantor a des copies de la même structure. Plus précisément, le premier tiers de l'ensemble de Cantor est identique au dernier tiers. Les réponses restantes pour $N = 3^{k+1}$ sont

$$2 \times 3^k + a_1, 2 \times 3^k + a_2, \ldots, 2 \times 3^k + a_M$$

Ces observations sont suffisantes pour générer la réponse pour toute puissance de 3.

Sous-tâche 2

Remarquons que $\frac{34676}{51169}$ est couvert par le 49^{e} filtre, mais pas par un filtre antérieur. Ceci est l'enregistrement maximal pour $N \leq 10^5$. Toute approche naïve risque d'échouer lorsque $N = 51169$.

Cette sous-tâche nécessite des observations plus approfondies sur les filtres d'Alice.

Propriété 1 (Les filtres sont symétriques). Si r est couvert par le $k^{\text{ième}}$ filtre, alors $1 - r$ est couvert par le $k^{\text{ième}}$ filtre. De même, si le $k^{\text{ième}}$ filtre ne couvre pas r, alors $1 - r$ n'est pas couvert par le $k^{\text{ième}}$ filtre.

Propriété 2 (Le $k^{\text{ième}}$ filtre et le $(k+1)^{\text{ième}}$ filtre sont liés). Soit k un entier strictement positif et soit $0 \leq r \leq \frac{1}{3}$. Si r est couvert par le $(k+1)^{\text{ième}}$ filtre, alors $3r$ est couvert par le $k^{\text{ième}}$ filtre. De même, si r n'est pas couvert par le $(k+1)^{\text{ième}}$ filtre, alors $3r$ n'est pas couvert par le $k^{\text{ième}}$ filtre.

On vous laisse le soin de démontrer ces 2 propriétés. A partir de ces propriétés, on peut définir la fonction $f(r) = 3 \times \min(r, 1 - r)$. Voici quelques théorèmes sur f.

Théorème 1. Si r n'est couvert par aucun filtre, alors $f(r)$ n'est couvert par aucun filtre.
Démonstration. Appliquer la Propriété 1 et la Propriété 2.

Théorème 2. Supposons que r est couvert par le $k^{\text{ième}}$ filtre. Alors soit $k = 1$, soit $f(r)$ est couvert par le $(k+1)^{\text{ième}}$ filtre.
Démonstration. Appliquer la Propriété 1 et la Propriété 2.

Voici l'approche pour résoudre la sous-tâche 2:

- Soit $r = \frac{x}{N}$. Construire cette suite: $r, f(r), f(f(r)), f(f(f(r))), \ldots$
- Si r est dans l'ensemble de Cantor, alors d'après le théorème 1, chaque terme de la suite fait partie de l'ensemble Cantor. Comme chaque terme est un multiple de $\frac{1}{N}$, alors la suite entrera dans un cycle.
- Si r n'est pas dans l'ensemble de Cantor, alors r est couvert par un filtre. D'après le théorème 2, un terme dans la suite sera couvert par le premier filtre.

Il y a 2 résultats possibles. Soit la suite entrera dans un cycle, soit un terme sera couvert par le premier filtre. L'algorithme doit gérer les deux résultats. La complexité temporelle prévue est de l'ordre de $O(N)$ ou légèrement plus lent.

English version appears before the French version.

Alternativement, il suffit d'ignorer la détection de cycle et de construire les 49 premiers termes. Dans le pire des cas (c'est-à-dire $r = \frac{34676}{51169}$), le 49^{e} terme est couvert par le premier filtre.

Sous-tâche 3

Remarquons que $\frac{222534799}{867579680}$ est couvert par le 130^{e} filtre, mais pas par les filtres précédents. On est d'avis que ceci est l'enregistrement maximum pour $N \leq 10^9$.

Tout d'abord, on utilise les 18 premiers filtres pour éliminer la plupart des nombres. Après cette étape, il reste moins de 10^6 nombres.

Ensuite, on applique l'algorithme de la sous-tâche 2 à chacun des nombres restants. Une bonne idée pour améliorer la complexité temporelle serait d'utiliser la mémoïsation. Il existe d'autres astuces pour améliorer la performance.

La complexité temporelle prévue est de l'ordre de $O(N^{\log_3 2} \log N)$ car il y a $O(N^{\log_3 2})$ nombres à considérer et chaque nombre prend environ $O(\log N)$ à considérer.