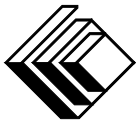

The CENTRE for EDUCATION in MATHEMATICS and COMPUTING

cemc.uwaterloo.ca

2024 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 Hat Circle

Subtask 1

We note that there can only be either 2 or 4 people at the table. This can be solved with conditional statements. In the case that there are only 2 people, they are seated across from each other, so we just compare the values of both hats. With 4 people, we can compare the equality of the hats at seat 1 and 3 and seat 2 and 4.

Subtask 2

All people have the same hat number 1. This means we can simply count the number of people at the table.

Subtask 3

Since people in even-numbered seats have hat number 1 and people in odd-numbered seats have hat number 0, they will only see matching hats if the number of people is divisible by 4.

Subtask 4

There was no specific intended approach for this subtask. We observe that in a circular arrangement, the person in seat i is directly opposite to the person in seat $i + \frac{N}{2}$.

One possible solution would be to store for each hat number, a list of the seat numbers of the people wearing that hat number. We can then iterate over the lists for each hat number and check if for each seat i in a list, if both i and $i + \frac{N}{2}$ exist in the same list.

Subtask 5

We notice that we can do the check directly on the input as a list. For each person at seat i, we check if the opposite person at seat $i + \frac{N}{2}$ has the same hat number. Since each seat is indexed from 1 to N, if $i > \frac{N}{2}$ then $i + \frac{N}{2}$ will be greater than N. To ensure that we stay within bounds, there are a couple approaches we can use:

- loop from 1 to N and compare i and $(i + \frac{N}{2})\%N$ to count person at seat i;
- loop from 1 to N and subtract N from $i + \frac{N}{2}$ if it is greater than N to count person at seat i;
- loop from 1 to $\frac{N}{2}$ and count for person at seat i and seat $i + \frac{N}{2}$.

S2 Heavy-Light Composition

Subtask 1

Hard-code every possible string containing "a" and "b" of length between 2 and 4 (there are 30 such strings) and output whether it alternates between heavy "a" and light "b" or heavy "b" and light "a".

Subtask 2

There was no intended solution for this subtask. It was intended to allow for possibly slower full solutions.

Subtask 3

We consider two cases for if it starts with "a" or not:

- Check that every second letter is "a" and that the rest are not "a".
- Check that every second letter is not "a" and that the rest are "a".

If either case is true, output `T`; otherwise, output `F`.

Subtask 4

First, create an array that will store the frequency of each letter in the string. Each element represents the count of a particular letter. Then, iterate over all letters in the string and increment the frequency of it in the array.

We realize we can extend the logic from subtask 3 with this array. If a letter has a frequency greater than 1 in the array, it is heavy; if not, it is light.

We consider two cases for if it starts with a heavy letter or not:

- Check that every second letter has a frequency greater than 1 and that the rest are light.
- Check that every second letter does not have a frequency greater than 1 and that the rest are heavy.

If either case is true, output *T*; otherwise, output *F*.

S3 Swipe

Subtask 1

Since $N = 2$, we can use casework on all possible cases. Alternatively, note that the only moves that can be made are to do nothing, swipe left on $[0, 1]$, or swipe right on $[0, 1]$. After 1 swipe, additional swipes will not change the array A, as both elements of A will be the same. Thus, we can try all possible moves, and check if array A becomes B. Otherwise, the answer will be `NO`.

Subtask 2

The intended solution for this subtask is to iteratively brute force all possible arrays by trying all possible swipes at each step. This can be implemented similarly to BFS, with a visited map, to ensure we don't visit the same array A twice. If we are able to reach array B, then we print out the sequence of swipes and return. As $N \leq 8$, with careful implementation, this should pass comfortably in time.

Subtask 3 and 4

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

Let B' be the "compressed" version of B, where all adjacent equal elements of B are removed. Then, the answer is `YES` if and only if B' is a sub-sequence of A.

To see why this is the case, notice that intersecting a necessary left swipe with a necessary right swipe will overwrite valuable elements. Take $A = [1, 2]$ and $B = [2, 1]$ as an example. In order to make the first element a 2, we must swipe left and we get $A = [2, 2]$. However, we cannot make the second element a 1 now, as the 1 has been overwritten. A similar argument follows if we first try to swipe right. Thus, it is impossible for array A to turn into B. When B' is not a sub-sequence of A, there will inevitably be a case where a necessary left swipe intersects a necessary right swipe, which makes it impossible.

To construct the solution, we employ a 2-pointers approach. Iterate i through each element of A, and increment j while A_i is equal to B_j. This will capture the range of elements that we need to swipe across. We push all left swipes into a list of swipes $left$, and right swipes into a list of swipes $right$. Notice that left swipes should be performed in increasing order of right endpoints, while right swipes should be performed in decreasing order of left endpoints, so that valuable elements will not get overwritten. Thus, we can push the reversed $right$ into $left$ to get the correct order of swipes.

This solution runs in $\mathcal{O}(N)$ time, which is sufficient for full marks. Subtask 3 was meant for suboptimal implementations of the full solution.

S4 Painting Roads

Subtask 1

We can model the intersections and roads as an undirected graph. In this subtask, the roads connecting intersection i with intersection $i + 1$ for all $1 \leq i < N$ ensure that the graph is connected and has a Hamiltonian path $1 \leftrightarrow 2 \leftrightarrow \cdots \leftrightarrow N$.

First, observe that Alanna must paint at least $N - 1$ roads. If not, then the coloured roads will not form a connected subgraph. Because the original graph is connected, there must be a grey road connecting two different components of coloured roads, and there is no coloured path connecting the endpoints of this grey road.

Because we're guaranteed a Hamiltonian path $1 \leftrightarrow 2 \leftrightarrow \cdots \leftrightarrow N$, let's consider choosing these $N - 1$ roads to paint. Thinking about what constraints on the colours are necessary to satisfy the condition in the problem, we can realize that it suffices to colour the roads alternating between red and blue along the path $1 \leftrightarrow 2 \leftrightarrow \cdots \leftrightarrow N$.

Subtask 2

In this subtask, the graph is connected and has an equal number of edges and nodes. In other words, the graph forms a pseudotree. In particular, there is exactly one simple cycle in the graph.

Just like in subtask 1, it can be proven that Alanna must paint at least $N - 1$ roads. To achieve this bound, we can colour the edges of the cycle by alternating between red and blue, leaving exactly one cycle edge grey. The other edges should be coloured red or blue arbitrarily.

This cycle may be found by a carefully implemented graph-traversal algorithm like BFS or DFS.

Subtask 3

In this subtask, the graph is no longer guaranteed to be connected. The condition of no road belonging to two or more simple cycles can be rephrased as each connected component of the graph being a cactus graph.

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

By generalizing the ideas from subtasks 1 and 2, the coloured edges should form a spanning forest of the graph. That would ensure that the coloured edges form the same connected components as the original graph while colouring the minimum number of edges to meet this requirement.

Let's consider choosing an arbitrary spanning forest of the graph to colour with red or blue. For each grey edge $u \leftrightarrow v$ not in the spanning forest, we can colour the path (in the corresponding tree) connecting u and v by alternating between red and blue. Because the component is a cactus, these paths do not share any edges, so we don't have to worry about these edges being involved in conflicting constraints.

An arbitrary spanning forest can be found with algorithms like BFS or DFS. To determine the correct colours, we cannot afford to spend $O(N)$ time per path because there could be as many as $\frac{N-1}{2}$ grey edges. Instead, we need to identify the corresponding path in $O(\text{length of path})$ time. One way to do this is by first precomputing parents and depths of each node within a tree. Then, for each grey edge $u \leftrightarrow v$, trace out the path by repeatedly moving the deeper node (u or v) up to its parent until u and v coincide.

Subtask 4

The solution from subtask 3 no longer works because the paths induced by each grey edge are no longer disjoint, so it is possible that they cause some conflicting constraints on the colours. It is no longer enough to take an arbitrary spanning forest.

Guided by these intuitions, let's think about what conditions on the spanning forest would be nice to have. Let's focus on a single connected component, where the coloured edges form a tree, and let's root the tree arbitrarily. Consider a grey edge $u \leftrightarrow v$, and let the LCA (lowest common ancestor) of u and v in the tree be l. The constraints on the colours that this grey edge places are that (1) all edges between u and l must be different from its parent edge, (2) all edges between v and l must be different from its parent edge, and (3) if $u \neq l$ and $v \neq l$, then two child edges of l are different.

Intuitively, most constraints are from types (1) and (2). It is actually possible to satisfy all type (1) and (2) constraints by colouring each edge by the parity of its depth in the tree, but this fails to satisfy any type (3) constraints. This motivates us to consider: Is there a spanning forest where we end up with no type (3) constraints?

In other words, we want a spanning forest with no *cross edges*: edges that do not connect an ancestor with a descendent. For undirected graphs, the DFS tree (tree formed by a depth-first traversal) has exactly this property we're looking for. This results in a very short solution to the full problem: For each connected component, take a DFS tree and colour the tree edges red or blue based on the parity of its depth.

S5 Chocolate Bar Partition

The first section will go over the full solution and the last paragraph will touch upon the intended solutions for some subtasks.

Let the mean of the entire array be μ. Notice that the mean of each component of a valid partition must be μ. Now consider the transformation of $A_{i,j} = T_{i,j} - \mu$. This reduces the problem to splitting array A into the maximum number of parts such that each part has a sum of 0.

Now let $P_{j,i}$ be the prefix sum array for A_j. That is,

$$P_{j,i} = \sum_{c=1}^{i} A_{j,c}$$

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

We will solve the new problem with dynamic programming. Let $dp[k][i]$ represent the maximum number of parts we can split the first i columns of the array such that the sum of each part is 0 and there is a (possibly empty) component sticking out of the k-th row (row 1 or row 2). Thus, for the base case we have that $dp[k][0] = 0$.

We can incrementally update $dp[k][i]$ starting from the smallest i. First set $dp[k][i]$ to be the maximum of

1. $dp[k][i-1]$ ($A_{k,i}$ is sticking out)
2. $dp[3-k][i-1]$ ($A_{k,i}$ and $A_{3-k,i}$ are in one component)
3. $dp[k][j]+1$ such that $P_{3-k,j} = P_{3-k,i}$ and $j < i$ (Add a "line component" from $A_{3-k,j+1}$ to $A_{3-k,i}$)
4. $dp[3-k][j]+1$ such that $P_{k,i} + P_{3-k,j} = 0$ (Finish the component by cutting a chunk from the first i cells on row $3-k$ and the first j cells on row k)

Then if $P_{1,i} + P_{2,i} = 0$, simultaneously update $dp[1][i]$ and $dp[2][i]$ as $\max(dp[0][i], dp[1][i]) + 1$ to signify that we split between column i and $i+1$ (or we reach the end of $i = N$).

The final answer is $dp[1][N]$. To help understand how this is sufficient, it is helpful to draw out the transitions with multiple cases and notice that the sum of the component that is sticking out for $dp[k][i]$ is $P_{k,i}$.

This results in an $\mathcal{O}(N^2)$ algorithm as seen in cases 3 and 4. We can make it run in sub-quadratic time by maintaining maps of $P_{k,i} \to \max dp[k][i]$ and $P_{k,i} \to \max dp[3-k][i]$ as we compute $dp[k][i]$.

For the first 2 subtasks, we can generate all the possible partitions by hand for Subtask 1 or by running a clever brute force for Subtask 2. The other subtasks are used to reward suboptimal dynamic programming solutions such as using the sum of the component as a state, using the prefix of the first row and the second row simultaneously as a state or with suboptimal transitions.

The CENTRE for EDUCATION in MATHEMATICS and COMPUTING

cemc.uwaterloo.ca

Commentaires sur le CCI de niveau sénior de 2024

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 Cercle de chapeaux

Sous-tâche 1

On remarque qu'il ne peut y avoir que 2 ou 4 personnes assises autour de la table. On peut donc aborder cette situation en employant des instructions conditionnelles. Si on a 2 personnes, elles se trouvent face à face ; dans ce cas, il suffit de comparer directement les numéros de leurs chapeaux. S'il y a 4 personnes, on peut comparer l'égalité des chapeaux aux places 1 et 3 et aux places 2 et 4.

Sous-tâche 2

Toutes les personnes ont le même numéro de chapeau, soit le numéro 1. Cela signifie que l'on peut tout simplement compter le nombre de personnes assises autour de la table.

Sous-tâche 3

Puisque les personnes occupant les places paires ont le chapeau numéro 1 et que celles occupant les places impaires ont le chapeau numéro 0, elles ne verront que des chapeaux portant le même numéro que le leur si le nombre de personnes est divisible par 4.

Sous-tâche 4

Aucune méthode spécifique n'était préconisée pour cette sous-tâche. Cependant, on remarque qu'en disposition circulaire, la personne assise au siège i se trouve directement en face de la personne assise au siège $i + \frac{N}{2}$.

Une approche envisageable consiste à stocker pour chaque numéro de chapeau une liste des numéros de sièges des personnes portant ce numéro de chapeau. On peut ensuite itérer sur les listes pour chaque numéro de chapeau et vérifier, pour chaque siège i dans une liste, si les sièges i et $i + \frac{N}{2}$ se trouvent dans la même liste.

Sous-tâche 5

On remarque que l'on peut effectuer la vérification directement sur l'entrée sous forme de liste. Pour chaque personne au siège i, on vérifie si la personne assise en face d'elle au siège $i + \frac{N}{2}$ a le même numéro de chapeau. Puisque chaque siège est numéroté de 1 à N, si $i > \frac{N}{2}$ alors $i + \frac{N}{2}$ sera supérieur à N. Pour respecter les bornes, plusieurs approches sont possibles :

- parcourir en boucle les sièges de 1 à N et comparer i et $(i + \frac{N}{2})\%N$ pour prendre en compte la personne au siège i ;
- parcourir en boucle les sièges de 1 à N et soustraire N de $i + \frac{N}{2}$, si cette dernière est supérieure à N, pour prendre en compte la personne au siège i ;
- parcourir en boucle les sièges de 1 à $\frac{N}{2}$ et prendre en compte les personnes aux sièges i et $i + \frac{N}{2}$.

S2 Composition lourde-légère

Sous-tâche 1

Il faudra coder en dur toutes les chaînes possibles contenant les lettres « a » et « b » dont les longueurs varient entre 2 et 4 lettres (il y a 30 telles chaînes). Ensuite, les données de sortie devraient indiquer si les lettres de chaque chaîne alternent entre un « a » lourd et un « b » léger ou entre un « b » lourd et un « a » léger.

Sous-tâche 2

Cette sous-tâche n'avait pas de solution définie au préalable. Son objectif était d'accepter des solutions qui pourraient être plus lentes dans leur exécution.

Sous-tâche 3

On considère deux cas, selon que la chaîne commence par « a » ou non :
- Vérifier qu'une lettre sur deux est « a » et que les autres lettres ne sont pas « a ».
- Vérifier qu'une lettre sur deux n'est pas « a » et que les autres lettres sont « a ».

Si l'un des cas est vrai, afficher `T` ; sinon, afficher `F`.

Sous-tâche 4

La première étape consiste à créer un tableau qui stockera la fréquence de chaque lettre dans la chaîne. Chaque élément représente le nombre de fois qu'une lettre particulière parait. Ensuite, il faut itérer sur toutes les lettres de la chaine et incrémenter les fréquences respectives des lettres dans le tableau.

On se rend compte qu'il est possible d'appliquer la logique de la sous-tâche 3 en se servant de ce tableau. Si une lettre a une fréquence supérieure à 1 dans le tableau, elle est considérée comme lourde ; dans le cas contraire, elle est légère.

On considère deux cas, selon que la chaîne commence par une lettre lourde ou non :
- Vérifier qu'une lettre sur deux a une fréquence supérieure à 1 et que les autres lettres sont légères.
- Vérifier qu'une lettre sur deux n'a pas une fréquence supérieure à 1 et que les autres lettres sont lourdes.

Si l'un des cas est vrai, afficher T; sinon, afficher F.

S3 Swipe

Sous-tâche 1

Étant donné que $N = 2$, il est possible d'effectuer une analyse exhaustive de tous les cas envisageables. Une autre approche consiste à remarquer que les seules opérations que l'on peut effectuer sont de ne rien faire, d'effectuer un glissement vers la gauche sur $[0, 1]$ ou un glissement vers la droite sur $[0, 1]$. Après un seul glissement, le fait d'effectuer des glissements supplémentaires ne modifiera pas le tableau A, puisque ses deux éléments seront identiques. Donc, on peut tester tous les glissements possibles et vérifier si cela transforme le tableau A en B. Si ce n'est pas le cas, la réponse sera `NO`.

Sous-tâche 2

Pour cette sous-tâche, la stratégie envisagée est d'adopter une approche de force brute de manière itérative sur tous les tableaux possibles en testant tous les glissements possibles à chaque étape. Cette méthode peut être employée de façon similaire à un parcours en largeur (BFS), accompagnée d'une structure de données pour suivre les tableaux déjà visités, afin d'éviter de traiter plusieurs fois le même tableau A. Si l'on peut

obtenir le tableau B, alors la séquence des glissements effectués est affichée avant de conclure le processus. Puisque $N \leq 8$, avec une programmation soignée, cette solution devrait être exécutable dans la limite de temps imparti.

Sous-tâches 3 et 4

Soit B' la version « compressée » de B, obtenue en supprimant tous les éléments adjacents identiques dans B. La réponse est `YES` si et seulement si B' est une sous-séquence de A.

Pour comprendre pourquoi ceci est vrai, il est important de remarquer qu'une intersection entre un glissement nécessaire vers la gauche et un glissement nécessaire vers la droite entraînerait la suppression d'éléments essentiels. Considérons l'exemple où $A = [1, 2]$ et $B = [2, 1]$. Pour transformer le premier élément en 2, on doit effectuer un glissement vers la gauche pour obtenir $A = [2, 2]$. Cependant, il devient alors impossible de transformer le second élément en 1, car le 1 initial a été écrasé. Un raisonnement semblable s'applique si l'on tente d'abord un glissement vers la droite. Donc, il est impossible pour que le tableau A devienne le tableau B. Lorsque B' n'est pas une sous-séquence de A, il y aura toujours un cas où il y aura une intersection entre un glissement nécessaire vers la gauche et un glissement nécessaire vers la droite, ce qui rendra la transformation impossible.

Pour élaborer la solution, on utilise une approche à deux pointeurs. On itère i sur chaque élément de A et on augmente j tant que A_i équivaut à B_j. Cette méthode permet d'identifier la plage d'éléments à glisser. On enregistre tous les glissements vers la gauche dans une liste *gauche* et ceux vers la droite dans une liste *droite*. Pour éviter que des éléments essentiels soient écrasés, il est important d'effectuer les glissements vers la gauche dans l'ordre croissant de leurs extrémités droites et les glissements vers la droite dans l'ordre décroissant de leurs extrémités gauches. Donc, en ajoutant *droite* inversé à *gauche*, on obtient la séquence correcte d'opérations de glissements.

Cette méthode s'exécute en temps $\mathcal{O}(N)$, ce qui est suffisant pour obtenir la note maximale. La sous-tâche 3 était destinée aux implémentations sous-optimales de la solution complète.

S4 Peindre les routes

Sous-tâche 1

Il est possible de représenter les intersections et les routes par un graphe non-orienté. Dans cette sous-tâche, l'existence de routes reliant chaque intersection i à l'intersection $i + 1$ pour $1 \leq i < N$ assure que le graphe est connexe et qu'il contient un chemin hamiltonien de la forme $1 \leftrightarrow 2 \leftrightarrow \cdots \leftrightarrow N$.

Il est évident qu'Alanna doit peindre au moins $N - 1$ routes. Sinon, les routes peintes ne formeraient pas un sous-graphe connexe. Puisque le graphe initial est connexe, il existe forcément une route grise reliant deux composantes différentes de routes peintes, sans qu'aucun chemin peint ne relie les extrémités de cette route grise.

Sachant qu'il existe un chemin hamiltonien $1 \leftrightarrow 2 \leftrightarrow \cdots \leftrightarrow N$, considérons le fait de peindre ces $N-1$ routes. En analysant les exigences de coloration requises pour satisfaire les conditions du problème, on remarque qu'il suffit de peindre les routes en alternant entre rouge et bleu le long du chemin $1 \leftrightarrow 2 \leftrightarrow \cdots \leftrightarrow N$.

Sous-tâche 2

Dans cette sous-tâche, le graphe est toujours connexe mais présente une particularité : il compte autant d'arêtes que de noeuds, formant ainsi un pseudo-arbre. Cela signifie qu'il existe exactement un cycle simple dans le graphe.

Comme pour la sous-tâche 1, on peut démontrer qu'Alanna doit peindre au moins $N-1$ routes. Pour atteindre cette limite, on peut colorer les arêtes du cycle en alternant entre rouge et bleu, laissant exactement une arête du cycle grise. Les autres arêtes doivent être colorées en rouge ou en bleu de manière arbitraire.

La détection de ce cycle peut être effectuée à l'aide d'un algorithme de parcours de graphe, tel que BFS (parcours en largeur) ou DFS (parcours en profondeur), à condition qu'il soit correctement mis en oeuvre.

Sous-tâche 3

Dans cette sous-tâche, il n'est plus assuré que le graphe soit connexe. La condition stipulant qu'aucune route ne doit appartenir à deux cycles simples ou plus peut être interprétée de la manière suivante : chaque composante connexe du graphe doit être un graphe cactus.

En extrapolant les idées des deux premières sous-tâches, les arêtes colorées devraient constituer une forêt couvrante du graphe. Cela assurerait que les arêtes colorées forment les mêmes composantes connexes que le graphe initial, tout en colorant le nombre minimal d'arêtes nécessaire.

Considérons le fait de choisir une forêt couvrante arbitraire du graphe pour la colorier en rouge ou en bleu. Pour chaque arête grise $u \leftrightarrow v$ qui n'est pas incluse dans la forêt couvrante, on peut colorer le chemin (dans l'arbre correspondant) qui relie u à v en alternant entre rouge et bleu. Étant donné que la composante est un cactus, ces chemins ne partagent aucune arête, éliminant ainsi le risque de contraintes contradictoires entre les arêtes.

Pour trouver une forêt couvrante arbitraire, on peut utiliser des algorithmes tels que le BFS ou le DFS. Pour déterminer les couleurs correctes, il n'est pas envisageable de consacrer $O(N)$ temps par chemin, car il pourrait y avoir jusqu'à $\frac{N-1}{2}$ arêtes grises. Il faudrait plutôt identifier le chemin correspondant en un temps O(longueur du chemin). Pour ce faire, une méthode consiste d'abord à précalculer les parents et les profondeurs de chaque noeud de l'arbre. Puis, pour chaque arête grise $u \leftrightarrow v$, on trace le chemin en remontant progressivement le noeud le plus profond (u ou v) vers son parent jusqu'à ce que u et v se rejoignent.

Sous-tâche 4

La solution de la sous-tâche 3 n'est plus applicable parce que les chemins induits par les arêtes grises ne sont désormais plus disjointes, ce qui pourrait créer des contraintes conflictuelles concernant les couleurs. Il n'est donc plus suffisant de choisir une forêt couvrante de manière arbitraire.

Suivant cette intuition, examinons quelles pourraient être les conditions idéales pour la forêt couvrante. Concentrons-nous sur une seule composante connexe où les arêtes colorées forment un arbre et enracinons cet arbre de façon arbitraire. Considérons une arête grise $u \leftrightarrow v$ et soit l le plus petit ancêtre commun de u et v dans l'arbre. Les contraintes imposées par cette arête grise sur les couleurs sont telles que (1) toutes les arêtes entre u et l doivent être différentes de leur arête parente, (2) il en va de même pour les arêtes entre v et l et (3) si $u \neq l$ et $v \neq l$, alors deux arêtes enfants de l sont différentes.

La majorité des contraintes proviennent des types (1) et (2). Il est possible de répondre à toutes les contraintes de types (1) et (2) en attribuant une couleur à chaque arête selon la parité de sa profondeur dans l'arbre, mais cela ne permet pas de répondre aux contraintes de type (3). Ce constat nous amène à nous demander s'il existe une forêt couvrante sans contraintes de type (3).

Autrement dit, on recherche une forêt couvrante sans *arêtes transversales*, c'est-à-dire des arêtes qui ne relient pas un ancêtre à un descendant. Dans le cas des graphes non-orientés, l'arbre DFS (arbre formé par un parcours en profondeur) possède précisément cette propriété que l'on recherche. Ceci nous conduit à une solution très concise pour résoudre le problème : pour chaque composante connexe, on doit choisir un arbre DFS et colorier les arêtes de cet arbre en rouge ou en bleu en fonction de la parité de leur profondeur.

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

S5 Diviser une barre de chocolat

La première partie de ce texte détaillera la solution complète, tandis que le dernier paragraphe s'intéressera aux solutions envisagées pour certaines sous-tâches.

Soit μ la moyenne de l'ensemble du tableau. Remarquons que la moyenne de chaque composant d'une partition valide doit être égale à μ. Ensuite, considérons la transformation de $A_{i,j} = T_{i,j} - \mu$. Ce procédé simplifie le problème, qui consiste désormais à séparer le tableau A en le nombre maximal de parties de sorte que chaque partie ait une somme de 0.

Soit $P_{j,i}$ le tableau de sommes cumulées de A_j :

$$P_{j,i} = \sum_{c=1}^{i} A_{j,c}$$

Pour résoudre ce nouveau problème, on va utiliser la programmation dynamique. Soit $dp[k][i]$ le nombre maximal de parties en lesquelles on peut diviser les i premières colonnes du tableau de manière que la somme de chaque partie soit 0, tout en permettant l'existence d'une composante saillante (potentiellement vide) de la k-ème rangée (la rangée 1 ou la rangée 2). Donc, pour le cas initial, on a $dp[k][0] = 0$.

On peut actualiser $dp[k][i]$ de manière incrémentielle en commençant par le plus petit i. Tout d'abord, on définit $dp[k][i]$ comme étant le maximum de

1. $dp[k][i-1]$ (avec $A_{k,i}$ comme composante saillante)
2. $dp[3-k][i-1]$ ($A_{k,i}$ et $A_{3-k,i}$ sont dans une composante unique)
3. $dp[k][j] + 1$ tel que $P_{3-k,j} = P_{3-k,i}$ et $j < i$ (on ajoute une « composante linéaire » de $A_{3-k,j+1}$ à $A_{3-k,i}$)
4. $dp[3-k][j] + 1$ tel que $P_{k,i} + P_{3-k,j} = 0$ (on finit la composante en sectionnant un morceau des i premières cellules de la rangée $3-k$ et des j premières cellules de la rangée k)

Si $P_{1,i} + P_{2,i} = 0$, on actualise simultanément $dp[1][i]$ et $dp[2][i]$ par $\max(dp[0][i], dp[1][i]) + 1$, indiquant ainsi une division entre la colonne i et la colonne $i+1$ (ou on atteint la fin de $i = N$).

La réponse finale est $dp[1][N]$. Pour comprendre pourquoi cela suffit, il est utile de visualiser les transitions dans divers cas et de noter que la somme de la composante saillante de $dp[k][i]$ correspond à $P_{k,i}$.

Cette approche conduit à un algorithme de complexité $\mathcal{O}(N^2)$, comme dans les cas 3 et 4. Pour accélérer le processus et atteindre un temps d'exécution sous-quadratique, on peut maintenir des mappages de $P_{k,i} \to \max dp[k][i]$ et de $P_{k,i} \to \max dp[3-k][i]$ à mesure que $dp[k][i]$ est calculé.

Pour les deux premières sous-tâches, il est possible de générer toutes les partitions possibles manuellement pour la sous-tâche 1 ou en employant une approche de force brute pour la sous-tâche 2. Les autres sous-tâches visent à récompenser des solutions de programmation dynamique moins optimales, telles que l'utilisation de la somme de la composante comme un état ou l'usage simultané des sommes cumulées de la première et de la seconde rangée comme un état ou encore l'emploi de transitions sous-optimales.

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