The CENTRE for EDUCATION in MATHEMATICS and COMPUTING

cemc.uwaterloo.ca

2022 CCC Junior Problem Commentary

Creation of this commentary is a new initiative for the Canadian Computing Competition. Our goal is to give a brief outline of what is required to solve each problem and what challenges may be involved. The notes below can be used to guide anyone interested in trying to solve these problems, but are 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.

J1 Cupcake Party

Solving this problem involves correctly reading the input and performing some arithmetic. The provided samples illustrate how to compute the number of cupcakes. Then, 28 must be subtracted from this value to determine the number of cupcakes left over.

J2 Fergusonball Ratings

Loops and if statements are required for this problem. A clean approach is to loop N times and read two lines of input per loop. This allows you to process one player at a time. A calculation can be performed to determine the player's star rating, and we can also maintain a count of the number of players with a star rating greater than 40. Care may be needed to produce only one line of output exactly as specified.

J3 Harp Tuning

There is a lot to consider when attempting to solve this problem. The biggest challenge is to break the input into pieces. This is known as *tokenizing* the input and the pieces are typically called *tokens*.

Specifically, the first subtask requires breaking a single instruction into a sequence of letters, the character `+` or `-`, and the number of turns. The focus of the second subtask is to break the input into different instructions, but each instruction has a simpler form. The third subtask combines these two objectives. The final subtask adds an extra layer of difficulty by allowing the number of turns to be a multi-digit number.

One elegant way to tokenize the input is to iterate through the input character by character while also remembering the last character read. This allows you to recognize when and where one token begins and another token ends. An alternative is to keep track of the current "state" which is an indication of which of the three types of token is currently under consideration.

Outputting the translation of each instruction as it is tokenized, rather than storing up all the translations until the end, removes the need for any extra data structures.

J4 Group Work

The fourth problem in this year's competition is the first that requires a collection of input to be stored together in a data structure. Because the assignment of students to groups is not given until the end of the input, the constraints cannot be fully considered as they are encountered, and so must be stored in memory. It is helpful to also store the student groups in memory.

Once both types of input are in memory, a solution can be obtained by considering each constraint individually and maintaining a count of how many are violated.

Likely pitfalls include incorrectly assuming something about the order in which student names appear, and erroneously thinking that a group can only violate one constraint.

Care must be taken with the Boolean logic required to test if a given constraint is violated. This will require determining if a given student is in a given group. Doing this inefficiently by looping through all the groups or looping through all the students should have allowed a submission to earn 14 of the available 15 marks. To earn the final mark, this look-up must be made more efficient. This can be accomplished using a data structure built-in to the language designed for this very purpose (e.g. a dictionary in Python, a `HashMap` in Java, or an `unordered_map` in C++).

J5 Square Pool

Any participant who was able to make any progress on this problem (or the previous problem, J4) is strongly encouraged to attempt the Senior Contest next year if they remain eligible. Earning more than 8 marks on J5 was especially difficult and intended to provide a significant challenge.

The first subtask can be solved by recognizing that when there is only one tree, a largest square pool must sit at one of the "corners" of the tree. An example of this is drawn in the explanation for the first sample input where a largest pool can sit at the "bottom-left corner" of the tree. As a result, the first three marks can be earned by considering the four possible cases and taking the maximum value.

Once there is more than one tree, a lot more work must be performed. For the second subtask, an approach that uses *brute-force* will earn full marks. There are different ways to accomplish this but they all amount to essentially testing all possible locations and sizes of a square pool and for each possibility, determining whether or not it contains a tree. Doing this successfully is done most naturally using several nested loops.

Allowing yards to be as large as 500 000-by-500 000 prevents a brute-force approach from finishing within the time limit. However, an important observation for the third subtask is that the number of trees is still guaranteed to be quite small. This is a clue that we might want a solution which depends only on the number of trees and not on the size of the yard. Recursion can be used to do just this. The key idea of the first subtask can be reused here. By inspecting one tree, we can limit our further search for an optimal placement of the pool to above, below, to the left, or to the right of this tree. This reduces the problem to a smaller (now possibly rectangular) yard.

Recursion will fail once the number of trees climbs much higher than 10 because the runtime will grow exponentially as the number of trees increases. Therefore, we require a solution which still depends only on the number of trees, but to a lesser extent. A very clever idea allows us to accomplish this. A square pool that is as large as possible can be moved up until it hits a tree or the upper boundary of the yard. Similarly, a square pool that is as large as possible can be moved left until it hits a tree or the left boundary of the yard. If we imagine adding coordinates to the yard, this means that each tree and the upper boundary provide $T+1$ candidates for the topmost position of an optimally placed pool. Similarly, there are $T+1$ candidates for the leftmost position of an optimally placed pool. By considering all pairs consisting of a candidate for the leftmost position and a candidate for the topmost position, we considerably limit the number of possible locations for an optimally placed pool. Moreover, we can figure out the size of the largest possible pool at one of these possible locations by only considering the remaining trees and boundaries. This approach yields a solution with a runtime that computer scientists characterize as *cubic* in T.

Le CENTRE d'ÉDUCATION en MATHÉMATICS et en INFORMATIQUE

cemc.uwaterloo.ca

Commentaires sur le CCI de niveau junior de 2022

La création de ce commentaire est une nouvelle initiative pour le Concours canadien d'informatique. Son objectif est de donner un bref aperçu de ce qu'il faut faire pour résoudre chaque problème et des défis à relever. Les notes ci-dessous peuvent être utilisées pour guider toute personne intéressée à essayer 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.

J1 Petits gâteaux festifs

Pour résoudre ce problème, il faut que les données d'entrée soient bien lues et il faut effectuer quelques opérations arithmétiques. Les exemples fournis démontrent comment calculer le nombre de petits gâteaux. Il faut ensuite soustraire 28 de cette valeur pour déterminer le nombre de petits gâteaux restants.

J2 Classements de Fergusonball

Il faut utiliser des boucles et des instructions conditionnelles pour résoudre ce problème. Une bonne approche serait d'effectuer N boucles et de lire deux lignes de données d'entrée par boucle. Cela permet ainsi de traiter un joueur à la fois. Un calcul peut être effectué pour déterminer le classement du joueur et on peut également garder le compte du nombre de joueurs dont le classement est supérieur à 40. Il faudra s'assurer de ne produire qu'une seule ligne de données de sortie de la manière précisée.

J3 Accorder une harpe

Il y a beaucoup de choses à prendre en compte lorsque l'on tente de résoudre ce problème. Le plus grand défi consiste à décomposer les données d'entrée en morceaux. C'est ce qu'on appelle la *tokenisation* de l'entrée et les morceaux sont souvent appelés *tokens*.

Plus précisément, la première sous-tâche consiste à décomposer une instruction unique en une séquence de lettres, le caractère `+` ou `-`, et le nombre de tours. La deuxième sous-tâche consiste à décomposer l'entrée en différentes instructions, mais chaque instruction a une forme plus simple. La troisième sous-tâche combine ces deux objectifs. La dernière sous-tâche ajoute une couche supplémentaire de difficulté en faisant de sorte que le nombre de tours puisse être un nombre à plusieurs chiffres.

Une façon élégante de tokeniser l'entrée est d'enclencher un processus d'itération à travers l'entrée caractère par caractère tout en se souvenant du dernier caractère lu. Cela vous permet de voir quand et où commence un token et où se termine un autre token. Une autre solution consiste à garder la trace de « l'état » actuel, ce qui indique lequel des trois types de tokens est en cours d'examen.

Le fait d'émettre la traduction de chaque instruction au fur et à mesure de sa tokénisation, plutôt que de stocker toutes les traductions jusqu'à la fin, élimine le besoin de structures de données supplémentaires.

J4 Travail de groupe

Le quatrième problème du concours de cette année est le premier qui exige qu'une collection d'entrées soit stockée ensemble dans une structure de données. Comme les groupes d'élèves ne sont donnés qu'à la fin de l'entrée, les contraintes ne peuvent pas être entièrement évaluées au fur et à mesure qu'elles sont rencontrées, et doivent donc être stockées en mémoire. Il est utile de stocker également les groupes d'élèves en mémoire.

Une fois que les deux types d'entrée sont en mémoire, une solution peut être obtenue en considérant chaque contrainte individuellement et en gardant le compte du nombre de contraintes violées.

Quelques pièges possibles sont le fait de faire une supposition erronée quant à l'ordre dans lequel paraissent les noms des élèves ou de supposer à tort qu'un groupe ne peut violer qu'une seule contrainte.

Il faudra veiller à bien développer la logique booléenne nécessaire pour tester si une contrainte donnée est violée. Pour le faire, il faudra déterminer si un élève donné se trouve dans un groupe donné. Une manière inefficace de le faire serait de parcourir en boucle tous les groupes ou tous les élèves. Une soumission employant cette technique ne se verrait attribuer que 14 des 15 points disponibles. Pour obtenir le dernier point, cette recherche doit être plus efficace. Pour ce faire, on peut utiliser une structure de données intégrée au langage et conçue à cet effet (par exemple, un dictionnaire en Python, une `HashMap` en Java ou un `unordered_map` en C++).

J5 Piscine carrée

On encourage fortement tout participant qui a pu progresser dans la résolution de ce problème (ou le problème précédent, J4) à tenter le concours sénior l'année prochaine s'il reste éligible. Le problème J5 a été conçu de manière à poser un véritable défi ; l'obtention de plus de 8 points sur ce problème était donc une tâche très difficile !

La première sous-tâche peut être résolue en remarquant que lorsqu'il n'y a qu'un seul arbre, la plus grande piscine carrée doit être située à l'un des « coins » de l'arbre. Un tel exemple parait dans la justification des données de sortie du 1er exemple. On voit dans cet exemple que la plus grande piscine carrée est située au « coin inférieur gauche » de l'arbre. Par conséquent, les trois premiers points peuvent être obtenus en considérant les quatre cas possibles et en prenant la valeur maximale.

Dès qu'il y a plus d'un arbre, le montant de travail à effectuer augmente considérablement. Pour la deuxième sous-tâche, une approche utilisant une *recherche par force brute* obtiendra un maximum de points. Il existe différentes façons d'accomplir cette tâche, mais elles reviennent toutes à tester tous les emplacements et toutes les tailles possibles d'une piscine carrée et, pour chaque possibilité, à déterminer si elle contient ou non un arbre. Pour y parvenir, on peut employer plusieurs boucles imbriquées.

Le fait qu'une cour peut être de dimensions 500×500 fait de sorte qu'une recherche par force brute ne peut se terminer dans le temps alloué. Cependant, une observation importante pour la troisième sous-tâche est que le nombre d'arbres sera toujours un nombre assez petit. Cela est un indice comme quoi notre solution devrait dépendre du nombre d'arbres et non de la taille de la cour. À cette fin, on peut employer une approche récursive. On peut également réutiliser l'idée clé de la première sous-tâche. En examinant un arbre, on peut limiter notre recherche ultérieure pour un placement optimal de la piscine au-dessus, au-dessous, à gauche ou à droite de cet arbre. Cela réduit le problème à une cour plus petite (qui pourrait également être rectangulaire).

Cette approche récursive échouera dès que le nombre d'arbres dépassera 10, car le temps d'exécution augmentera de façon exponentielle avec le nombre d'arbres. Par conséquent, on a besoin d'une solution

qui dépend toujours du nombre d'arbres, mais dans une moindre mesure. Une idée très astucieuse nous permet d'y parvenir. La plus grande piscine carrée possible peut être déplacée vers le haut jusqu'à ce qu'elle touche soit un arbre, soit le bord supérieur de la cour. De même, la plus grande piscine carrée possible peut être déplacée vers la gauche jusqu'à ce qu'elle atteigne soit un arbre, soit le bord gauche de la cour. Imaginons que l'on ajoute des coordonnées à la cour. Dans ce cas, à partir des arbres et du bord supérieur, on a $T + 1$ emplacements possibles pour la position la plus haute d'une piscine placée de manière optimale. De même, il y a $T + 1$ emplacements possibles pour la position la plus à gauche d'une piscine placée de manière optimale. En considérant tous les couples possibles constitués d'un emplacement pour la position la plus à gauche et d'un emplacement pour la position la plus haute, on limite considérablement le nombre d'emplacements possibles pour une piscine placée de manière optimale. De plus, on peut déterminer la taille de la plus grande piscine possible à l'un de ces emplacements possibles en ne considérant que les bords et les arbres restants. Cette approche permet d'obtenir une solution avec un temps d'exécution que les informaticiens caractérisent comme *cubique* en T.