The CENTRE for EDUCATION in MATHEMATICS and COMPUTING

cemc.uwaterloo.ca

2024 CCC Junior 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 it 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.

J1 Conveyor Belt Sushi

The first problem on this year's contest involves printing out the result of an arithmetic expression. The expression consists of three input values representing the number of plates of each colour, and constant values which are the cost of each colour of plate.

J2 Dusa And The Yobis

This year, a twist was introduced at this early point of the contest. We are told Dusa's starting size and the size of the Yobis, but we are not told how many Yobis there are. This means that a `while` loop, or the equivalent, is needed instead of a `for` loop which might be more familiar. That is, we can continue reading input and adding the size of the current Yobi to the size of the Dusa *while* the size of the Dusa is less than the size of the current Yobi. Put another way, the loop continues *until* a Yobi is encountered that is the same size as the Dusa or larger.

J3 Bronze Count

There are several different ways of solving this problem.

One approach is to maintain six variables storing values for the gold, silver and bronze score as well as a count of the number of participants with each of these scores. Taking care to initialize these variables properly, they can then be updated as needed given each new score. This approach can be used to pass all three subtasks.

Another approach is to first store a collection of all the scores in a data structure (e.g. in a list in Python or an `ArrayList` in Java). Then the maximum score (gold score) can be calculated and all occurrences of this score removed from this collection. Then the new maximum score (silver score) can be calculated and all occurrences of this score removed from the updated smaller collection. The maximum of the scores remaining in this even smaller new collection is the bronze score and the number of times it occurs can be computed and printed. This approach will pass the first two subtasks. Whether or not this approach passes the last subtask depends on how computing and removing occurrences of the maximum score is implemented. As long as this is done by passing over the collection a fixed/constant number of times (i.e. increasing the size of the collection does not affect how often a pass is made over the collection), this approach should also pass the third subtask.

Sorting can also be used here. After sorting the scores, the bronze score can be identified quickly for the first subtask where the scores are distinct. Otherwise, a loop is needed to scan from highest to lowest score effectively finding the "boundaries" between the gold and silver scores and between the silver and bronze scores. Any built-in sorting function/method should be fast enough to pass the last subtask. It is also possible but challenging to implement a sorting algorithm which will be fast enough to earn full marks with this strategy.

J4 Troublesome Keys

It can take some time to wrap your head around the behaviour of Alex's strange keyboard. To help us discuss it here, we will refer to the following five values.

`pressed` – the first line of input representing the keys pressed by Alex

`displayed` – the second line of input representing the letters displayed on the screen

`silly` – the letter corresponding to the silly key

`wrong` – the letter displayed when the silly key is pressed

`quiet` – the letter corresponding to the quiet key

For the first subtask, the quiet key is not pressed which means that the letter in `pressed` that is not in `displayed` must be `silly`. Also, `wrong` must be the letter that is in `displayed` but not in `pressed`. Finding a character that appears in one string but not the other can be done with a loop and search. The search can be accomplished with a built-in function or nested second loop. (In general, we can check whether or not the quiet key is pressed by comparing the lengths of `pressed` and `displayed`.)

For the other subtasks, we can still begin by determining `wrong` as above. Similarly, we can determine a pair of letters `x` and `y` that correspond to `silly` and `quiet`, but we need to work a bit harder to determine whether `x` corresponds to `silly` and `y` corresponds to `quiet`, or the other way around. For the second subtask, we can use the fact that the first troublesome key pressed is the silly key. Otherwise, one way to do this is to simulate Alex's keyboard by replacing all occurrences of `x` in `pressed` with `wrong` and removing `y` from `pressed`. It the result equals `displayed` then we know that `x` corresponds to `silly` and `y` corresponds to `quiet`. Otherwise, we know that it is the other way around. To solve the final subtask where a large number of keys are pressed, the number of times we pass over the input strings cannot depend on `N`, the bound on their lengths. This can be achieved by storing the letters in `pressed` and `displayed` in a data structure that allows for efficient searches (e.g. a dictionary in Python or `HashMap` in Java.)

This problem can also be solved without predetermining `wrong` and candidates for `silly` and `quiet`. We can scan `pressed` and `displayed` left to right looking for the first place where they differ. For the second subtask, we then immediately know that `silly` must be the mismatched character in `pressed` and `wrong` must be the mismatched character in `displayed`. To determine quiet (if it was pressed), we can continue scanning until we find a mismatch that does not involve `silly`. We omit the details here, but for the third and fourth subtasks, we can extend this approach by considering cases and simulating the behaviour of Alex's keyboard once we determine `silly` or `quiet`.

J5 Harvest Waterloo

For the final problem in this year's competition, the fundamental task is to determine all the locations that the farmer can reach. By "visiting" each of these locations and summing the values of the pumpkins at these locations, we can produce the total value harvested by the farmer. This requires us to store the pumpkin patch in memory which must ultimately be done with a two-dimensional structure (e.g. a list of strings or two-dimensional array).

For the first subtask, the farmer can reach every location. Visiting every location can be done with a nested loop.

For the second subtask, we can begin at the farmer's starting location and move in one direction until blocked by hay or the edge of the patch. Since we know that the locations the farmer can reach form a

rectangle, doing this up, down, left and right, allows us to determine the dimensions of the rectangle and the location of a corner of the rectangle. This allows us to use a nested loop as in the first subtask to visit all the locations that the farmer can reach.

A different strategy is required for the final two subtasks. We can maintain a collection of locations to visit. Initially this collection will consist only of the starting location of the farmer. Then we can repeatedly remove a location from the collection and if it has not yet been accounted for, add the value of the pumpkin at this location to a running total. Also, since the farmer can only move in four possible directions, every location we visit presents up to four potential new locations to add to the collection. Locations should not be added if they are not reachable (contain hay or do not exist because we are at the edge of the patch) or if they have already been accounted for. One clever way to record that a location has been accounted for is to change the `S`, `M` or `L` at that location to a `*`. This general technique is an example of what is called a *flood fill algorithm.*

Notice that the collection of locations to visit can theoretically grow very large and contain many duplicates. This is because, for example, visiting the location at row 4 and column 7 could result in us adding the location at row 3 and column 7 to the collection, but this same location might be added when visiting row 3 and column 8 (or row 2 and column 7, or row 3 and column 6). Avoiding this is needed to earn 15 marks.

Depending on how the collection is implemented, the full solution described here is called *breadth-first search* (BFS) or *depth-first search* (DFS). DFS can also be implemented using a *recursive* function or method which is a function or method that calls itself. For recursive functions, the collection is implicit and stored in a portion of memory usually referred to as the *call stack*. Different languages and machines can put different limits on how large the call stack can be. Python tends to be especially restrictive and so using recursion and Python for this problem likely requires you to instruct the system to make enough room by importing the `sys` module and issuing a command such as `sys.setrecursionlimit(100000)`.

The CENTRE for EDUCATION in MATHEMATICS and COMPUTING

cemc.uwaterloo.ca

Commentaires sur le CCI de niveau junior 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.

J1 Sushi sur tapis roulant

Le premier défi du concours de cette année est de calculer et afficher le résultat d'une expression arithmétique. L'expression comprend trois valeurs d'entrée représentant le nombre d'assiettes de chaque couleur et des valeurs constantes qui représentent le coût de chaque couleur d'assiette.

J2 Dusa et les Yobis

Cette année, une nouveauté a été introduite au début du concours. On a la taille initiale de Dusa ainsi que celle des Yobis, mais le nombre de Yobis n'est pas précisé. Cela exige donc l'emploi d'une boucle `while`, ou son équivalent, au lieu d'une boucle `for`. Autrement dit, on peut continuer à lire les entrées et à ajouter la taille du Yobi à celle de Dusa *tant que* la taille de Dusa est inférieure à celle du Yobi confronté. C'est-à-dire que la boucle continue *jusqu'à ce que* Dusa confronte un Yobi de la même taille que lui ou plus grand.

J3 Compte de bronze

Il existe plusieurs façons de résoudre ce problème.

L'une d'entre elles consiste à maintenir six variables stockant les valeurs des scores d'or, d'argent et de bronze, ainsi que le nombre de participants ayant obtenu chacun de ces scores. En prenant soin d'initialiser correctement ces variables, elles peuvent ensuite être mises à jour en fonction de chaque nouveau score. Cette approche peut être utilisée pour mener à bien les trois sous-tâches.

Une autre approche consiste à stocker d'abord une collection de tous les scores dans une structure de données (par exemple dans une liste en Python ou un `ArrayList` en Java). Ensuite, le score maximum (le niveau or) est calculé puis toutes les occurrences de ce score sont supprimées de cette collection. Ensuite, le nouveau score maximum (le niveau argent) est calculé puis toutes les occurrences de ce score sont également supprimées. Finalement, le maximum des scores restant dans cette nouvelle collection est le score du niveau bronze et il est possible de calculer et d'afficher le nombre de fois que ce score parait dans la collection. Cette approche peut être utilisée pour mener à bien les deux premières sous-tâches. L'efficacité de cette approche pour la dernière sous-tâche dépend de la façon dont sont effectués le calcul et la suppression des occurrences du score le plus élevé sont effectués. Si ces opérations sont exécutées en parcourant la collection un nombre fixe de fois (c'est-à-dire que le fait d'augmenter la taille de la collection n'influence pas le nombre de fois que la collection est parcourue), cette approche devrait aussi permettre de mener à bien la troisième sous-tâche.

On peut également utiliser le tri pour résoudre ce problème. Après avoir trié les scores, le score du niveau bronze peut être identifié rapidement pour la première sous-tâche où les scores sont distincts. Sinon, il faut utiliser une boucle pour examiner les scores du plus élevé au plus bas afin de trouver les « frontières » entre les scores des niveaux or et argent et entre les scores des niveaux argent et bronze. Toute fonction/méthode

de tri intégrée devrait être suffisamment rapide pour réussir la dernière sous-tâche. Il est également possible, quoique plus complexe, de développer un algorithme de tri suffisamment rapide qui permette d'obtenir le nombre maximum de points.

J4 Touches problématiques

Comprendre le fonctionnement du clavier atypique d'Alex peut demander un certain temps. Pour en discuter plus facilement, on emploie les cinq valeurs suivantes.

`frappée` – la première ligne des données d'entrée, soit les touches qu'Alex frappe sur le clavier

`affichée` – la seconde ligne des données d'entrée, soit les lettres qui s'affichent à l'écran

`absurde` – la lettre correspondant à la touche absurde

`erronée` – la lettre affichée lorsque Alex frappe la touche absurde

`silencieuse` – la lettre correspondant à la touche silencieuse

Pour la première sous-tâche, la touche silencieuse n'est pas frappée, ce qui signifie que la lettre dans `frappée` qui n'est pas dans `affichée` doit être `absurde`. De plus, la lettre qui parait dans `affichée` mais qui ne parait pas dans `frappée` doit être `erronée`. La recherche d'un caractère qui parait dans une chaîne mais pas dans l'autre peut être effectuée à l'aide d'une boucle et d'une recherche. La recherche peut être effectuée à l'aide d'une fonction intégrée ou d'une deuxième boucle imbriquée. (De façon générale, on peut vérifier si la touche silencieuse a été frappée ou non en comparant les longueurs de `frappée` et `affichée`).

Pour aborder les autres sous-tâches, on peut toujours commencer en déterminant d'abord `erronée` de la manière décrite précédemment. De même, on peut déterminer un couple de lettres `x` et `y` qui représentent `absurde` et `silencieuse`. Cependant, il nous faut une analyse plus approfondie pour déterminer si `x` correspond à `absurde` et si `y` correspond à `silencieuse`, ou l'inverse. Pour la deuxième sous-tâche, on peut utiliser le fait que la première touche problématique frappée est la touche absurde. Sinon, une façon de procéder consiste à simuler le clavier d'Alex en remplaçant toutes les occurrences de `x` dans `frappée` par `erronée` et en supprimant `y` de `frappée`. Si le résultat correspond à `affichée`, alors on sait que `x` correspond à `absurde` et que `y` correspond à `silencieuse`. Dans le cas contraire, la situation est inversée. Pour résoudre la dernière sous-tâche, où un grand nombre de touches sont frappées, le nombre de fois que l'on parcourt les chaînes d'entrée ne peut dépendre de `N`, soit la limite de leur longueur. Cela peut être accompli en stockant les lettres de `frappée` et `affichée` dans une structure de données favorisant des recherches rapides (comme un dictionnaire en Python ou un `HashMap` en Java).

Il est également possible de résoudre ce problème sans avoir à identifier préalablement `erronée` et les lettres pouvant correspondre à `absurde` et à `silencieuse`. On peut parcourir `frappée` et `affichée` de gauche à droite en cherchant le premier point où elles diffèrent. Pour la deuxième sous-tâche, on peut immédiatement conclure que `absurde` est le caractère non concordant dans `frappée` et `erronée` est le caractère non concordant dans `affichée`. Pour déterminer `silencieuse` (si elle a été frappée), il suffit de continuer à parcourir les chaînes de caractères jusqu'à ce qu'on trouve un point de non-concordance où `absurde` n'est pas impliqué. Cette approche peut être adaptée et employée pour les troisième et quatrième sous-tâches en effectuant une analyse minutieuse des différents cas possibles et en simulant le fonctionnement du clavier d'Alex après avoir identifié `absurde` ou `silencieuse`.

J5 Récolte Waterloo

Pour la compétition de cette année, le défi final consiste à identifier tous les emplacements accessibles par le fermier. En « visitant » chacun de ces emplacements et en additionnant les valeurs des citrouilles à ces emplacements, on peut calculer la valeur totale des citrouilles que le fermier a récoltées. Cette opération nécessite que l'on stock en mémoire le champ de citrouilles, ce que l'on peut faire à l'aide d'une structure bidimensionnelle, telle qu'une liste de chaînes ou un tableau bidimensionnel.

Pour la première sous-tâche, le fermier peut atteindre chaque emplacement. On peut visiter chaque emplacement à l'aide d'une boucle imbriquée.

Pour la deuxième sous-tâche, on peut commencer à l'emplacement de départ du fermier et on se déplace dans une direction jusqu'à ce qu'on soit bloqué par une botte de foin ou le bord du champ. Étant donné que les emplacements accessibles par le fermier forment un rectangle, le fait d'explorer dans les quatre directions (haut, bas, gauche, droite) permet de définir les dimensions de ce rectangle et de localiser un de ses coins. Cela nous permet d'utiliser une boucle imbriquée comme dans la première sous-tâche pour visiter tous les emplacements que le fermier peut atteindre.

Il faut employer une approche différente pour les deux dernières sous-tâches. On peut maintenir une collection d'emplacements à visiter. Initialement, cette collection ne comprendra que l'emplacement de départ du fermier. Ensuite, on peut retirer progressivement les emplacements de cette collection et, si elles n'ont pas encore été prises en compte, additionner les valeurs des citrouilles à ces emplacements au total cumulé. De plus, puisque le fermier ne peut se déplacer que dans quatre directions possibles, chaque emplacement visité présente jusqu'à quatre nouveaux emplacements potentiels à ajouter à la collection. Les emplacements ne doivent pas être ajoutés s'ils ne sont pas accessibles (contiennent du foin ou n'existent pas parce que nous sommes en dehors des limites du champ) ou s'ils ont déjà été pris en compte. Une manière élégante d'enregistrer qu'un emplacement a été pris en compte est de remplacer le `S`, `M` ou `L` à cet emplacement par un `*`. Cette technique est un exemple de ce qu'on appelle un *algorithme de remplissage par diffusion*.

Remarquons que la liste des emplacements à visiter peut rapidement s'agrandir et contenir des doublons car la visite d'un emplacement peut entraîner l'ajout multiple d'un même emplacement depuis des points adjacents. Par exemple, le fait de visiter l'emplacement à la rangée 4 et à la colonne 7 pourrait nous amener à ajouter l'emplacement à la rangée 3 et à la colonne 7 à la collection. Or, ce même emplacement aurait pu être ajouté lors de la visite de la rangée 3 et de la colonne 8 (ou de la rangée 2 et de la colonne 7 ou de la rangée 3 et de la colonne 6). Il faut éviter cela afin de gagner les 15 points.

Selon la manière dont la collection est mise en oeuvre, la solution complète décrite est appelée *le parcours en largeur* (BFS) ou *le parcours en profondeur* (DFS). Le DFS peut également être mis en oeuvre en utilisant une fonction ou méthode *récursive*, soit une fonction ou méthode qui s'appelle elle-même. Pour les fonctions récursives, la collection est implicite et stockée dans une partie de la mémoire appelée la *pile d'appels*. Les limites imposées sur la taille de la pile d'appels varient selon les langages et les systèmes. Python, en particulier, se montre assez restrictif à cet égard. Ainsi, le fait de recourir à la récursion en Python pour aborder ce problème implique souvent la nécessité d'augmenter manuellement la limite de récursivité. Cela se fait en important le module `sys` et en ajustant la limite à travers une commande, telle que `sys.setrecursionlimit(100000)`, pour allouer l'espace nécessaire au bon fonctionnement du programme.