



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

2026 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 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.

Attempts to solve past problems can be submitted on the [CCC Grader](#). Also, after a contest is complete, official test data is available by clicking the corresponding “Download” button on our [Past Contests](#) webpage.

J1 Concert Tickets

Determining whether or not Besa can buy her desired number of tickets involves using an `if` statement. There are different ways to structure this. One way is to use the condition $B + P > T$ and print `N` when the condition is true and print `Y` followed by the value of $T - P - B$ when the condition is false.

J2 Olympic Scores

There are always exactly six input values for the second problem. It can be solved by storing the last value, the difficulty factor, in a variable and storing the five scores in five different variables or in a sequential data structure (e.g. a list, an array, or a vector). However, it is interesting to note that storing these values is not absolutely necessary. As we read in each score, we can (i) add it to a running total of the scores, (ii) maintain a “current lowest score” L which is initialized to the value of 10, and (iii) maintain a “current highest score” H which is initialized to 0. Then the athlete’s overall score to be printed is $(S - H - L) \times D$ where S is the sum of the five scores and D is the difficulty factor. For the sample input, this would be $(35 - 0 - 10) \times 3 = 75$. Notice how this equals $(7 + 8 + 10) \times 3$ as given in the explanation of the output for the sample input.

J3 Creative Candy Consumption

A full solution to the third problem on the contest requires repeatedly solving its first subtask where Ngoc and Minh each have only one candy. An `if` statement can handle the small fixed number of possibilities in this case. To handle additional candies, the `if` statement can be placed inside a `while` loop with exactly one or two candies eaten per iteration of the loop.

The body of the loop can be structured to simulate eating one or two candies. That is, each sequence of candies (letters) can be represented by a string. When a candy is eaten, the first character of the string can be removed. Depending on which programming language or string type we use, this might mean replacing the string by a new string that is a substring or slice of the larger string. This approach will solve the second and third subtasks, but a less complex loop condition is needed for the second subtask.

Now, like your dentist, we recommend that you take a very long time if you decide to consume 50 candies. However, a computer program can simulate this in microseconds. That is, efficiency is not a concern until the fourth subtask where at least one of Ngoc and Minh will eat an absurd amount of candy. To ensure we do not exceed the time limit in this case, we must ensure each iteration of the loop is quick. It is possible to do this with the simulation approach discussed above, and a carefully chosen data structure and an appropriate removal function/method. However, a cleaner solution is to leave the original string unchanged. We can accomplish this by maintaining the index of the next candy at the front of each line. That is, we will consider

the candies to the left of each index to be removed even though they remain part of the string. This way, each iteration of the loop involves using a solution to the first subtask to determine whether to increment one or both of the indices by one.

J4 Snail Path

In order to count the number of times the snail enters a slimy square, we can keep track of which squares are slimy as the snail crawls from square to square. The different subtasks correspond to different ways we might represent a set of slimy squares.

For the first two subtasks, we can represent every square that the snail might enter. This two-dimensional grid of squares can be represented as a sequence of sequences (e.g. list of lists, array of arrays, or a vector of vectors). Alternatively, the two-dimensional grid can be represented as one long sequence: the first row, followed by the second row, followed by the third row, etcetera. For the first subtask, the snail can crawl up to $20 \times 20 = 400$ squares south and also up to 400 squares east. Therefore, the snail starts at the corner of a 401×401 grid. The second subtask is more challenging because it requires us to realize that it involves the snail starting in the middle of a 801×801 grid.

For the third subtask, the snail enters up to 20×1200 squares in each direction so it is difficult to fit the grid within the memory limits of the CCC grader. Instead, we can store only the set of slimy squares with the understanding that if a square is not slimy, then it is not in our set. Then for every snail movement, we can check to see if the square the snail is entering is in our set. The set can be represented using a sequential data structure and even if we search through the entire structure for every movement the snail makes, we should finish within the time limit.

In the final subtask, the snail can move extremely far from its initial position so running time is a bigger concern. To solve this subtask, we need to be able to very quickly determine whether a given square is in our set of slimy squares. A data structure that supports “fast look-up” is required (e.g. a dictionary in Python, a `HashSet` in Java, or an `unordered_set` in C++.)

An alternative solution is to build a sequence of every square that the snail enters including duplicates. The correct answer will be the length of this sequence minus the number of distinct squares in the sequence. This is because if a square the snail enters appears x times, it will be slimy exactly $x - 1$ of those times. If we can count the number of duplicates in the sequence efficiently (e.g. perhaps by sorting the list), this can lead to a solution that meets the time limit and passes the final subtask.

J5 Beams of Light

The last challenge of the 2026 CCC Junior Competition requires answering Q questions about N parking spots illuminated by L lights.

For the first subtask, $L \leq 1$. If there isn’t a light, the answer to every question is N. Otherwise, we can determine if parking spot i is illuminated by checking if $P - S \leq i \leq P + S$ where P is the parking spot directly below the light and S is the *spread* of the light (how many adjacent parking spots it illuminates on either side).

For the other three subtasks, upper bounds on the values of Q , N and L can tell us which algorithms will allow us to answer the questions within the time limit.

For the second subtask, L and Q are small so for every question, we can consider each light to determine whether or not a parking space is illuminated.

For the last two subtasks, L and Q are both very large so we can no longer consider each light to answer

each question. Instead, we need to do some work before starting to answer the questions. Specifically, we can construct a sequence A (for “answers”) where position i in the sequence records whether or not parking spot i is illuminated. As for the previous problems, this sequence could be a list, array, or vector with care taken to account for the fact that a sequential structure probably starts indexing at 0.

When N is small, we can build A by looping through each light and updating A for each parking spot it illuminates. Alternatively, we can loop through each parking spot and for each light, check if it illuminates the parking spot. Either approach will solve the third subtask.

For the final subtask, N is very large so we need to be more clever. Below is the outline of two possible approaches that consider the intervals illuminated by each light. That is, for a light above spot P and with spread S , the associated interval is $[P - S, P + S]$.

- “Merge” the intervals. For example, the intervals in the sample input are $[1, 2]$, $[2, 6]$ and $[8, 8]$ and they can be merged into the intervals $[1, 6]$ and $[8, 8]$. There are several ways to do this. It is tricky, but the key is to ensure we don’t loop through the lights or parking spots too often (i.e. only a small fixed number of times like once or twice).
- Start with an initial value of $v(i) = 0$ associated with each parking spot i . Then for each interval $[a, b]$ add one to $v(a)$ and subtract one from $v(b + 1)$. When done, $v(1) + v(2) + \dots + v(i)$ will be the total number of lights that illuminate parking spot i . It is possible to compute these sums in one loop and then use them to quickly answer each question.

Above, each value $P - S$, $P + S$, i , a and b represents a parking spot and we are ignoring the requirement that this means it must lie inside the interval $[1, N]$. This cannot be ignored when implementing the ideas.



The CENTRE for EDUCATION in MATHEMATICS and COMPUTING

cemc.uwaterloo.ca

Commentaires sur le CCI Intermédiaire de 2026

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.

Les tentatives de résolution des problèmes des concours précédents peuvent être soumises sur le [CCC Grader](#). De plus, une fois un concours terminé, les données de test officielles sont disponibles en cliquant sur le bouton « Télécharger » sur notre page [Concours précédents](#).

J1 Billets de concert

Déterminer si Besa est en mesure ou non d'acheter le nombre de billets désirés nécessite l'utilisation d'une instruction `if`. Il existe différentes façons de structurer cela. Une façon consiste à utiliser la condition $B + P > T$ et à afficher `N` lorsque la condition est vraie, et à afficher `Y` suivi de la valeur $T - P - B$ lorsque la condition est fausse.

J2 Notes olympiques

Il y a toujours exactement six valeurs d'entrée pour le deuxième problème. On peut procéder en affectant la dernière valeur, qui correspond au facteur de difficulté, à une variable et en affectant les cinq notes à cinq variables différentes ou à une structure de données séquentielle (par exemple, une liste, un tableau ou un vecteur). Cependant, il est intéressant de noter que l'affectation de ces valeurs n'est pas obligatoire. À mesure qu'on lit chaque note, on peut : (i) ajouter chaque note à un total cumulatif, (ii) maintenir *la note la plus basse actuelle* L , initialisée à 10, et (iii) maintenir *la note la plus haute actuelle* H , initialisée à 0. La note finale à afficher est alors $(S - H - L) \times D$, où D est le facteur de difficulté et S est la somme de cinq notes. Pour l'exemple fourni, cela donne $(35 - 0 - 10) \times 3 = 75$, ce qui correspond également à $(7 + 8 + 10) \times 3$, comme indiqué dans la section « Précisions par rapport aux données de sortie ».

J3 Consommation créative de confiseries

Une solution complète au troisième problème du concours nécessite d'appliquer à plusieurs reprises la solution de la première sous-tâche, où Ngoc et Minh ont chacun un bonbon. À l'aide d'une instruction `if`, on peut gérer un petit nombre de cas déterminés à l'avance. Pour traiter les bonbons supplémentaires, l'instruction `if` peut être placée dans une boucle `while`, avec exactement un ou deux bonbons mangés à chaque itération de la boucle.

Le corps de la boucle peut être structuré de manière à simuler la consommation d'un ou de deux bonbons. Autrement dit, chaque séquence de bonbons (lettres) peut être représentée par une chaîne de caractères. Lorsqu'un bonbon est mangé, le premier caractère de la chaîne peut être retiré. Selon le langage de programmation ou le type de chaîne utilisé, cela peut signifier remplacer la chaîne par une nouvelle chaîne qui est une sous-chaîne ou une tranche de la chaîne originale. Cette approche permet de résoudre les deuxième et troisième sous-tâches. Cependant, pour la deuxième sous-tâche, la condition de boucle peut être plus simple.

Maintenant, comme votre dentiste, nous vous recommandons de prendre beaucoup de temps si vous décidez de manger 50 bonbons. Cependant, un programme informatique peut simuler ceci en quelques microsecondes.

Autrement dit, l'efficacité n'est pas un problème avant la quatrième sous-tâche, où au moins l'un des deux, de Ngoc ou Minh mangera énormément de bonbons. Pour garantir que la limite de temps n'est pas dépassée dans ce cas, il faut s'assurer que chaque itération de la boucle soit rapide. Il est possible de faire cela avec l'approche de simulation discutée ci-dessus, en choisissant soigneusement une structure de données et une fonction ou méthode de suppression appropriée. Cependant, une solution plus claire consiste à laisser la chaîne originale inchangée. Nous pouvons accomplir cela en utilisant un index du prochain bonbon au début de chaque ligne. Autrement dit, nous considérons les bonbons à gauche de chaque index comme retirés, même s'ils font toujours partie de la chaîne. De cette façon, chaque itération de la boucle utilise la solution de la première sous-tâche pour déterminer si seulement un indice ou les deux doivent être augmentés d'un.

J4 Le trajet de l'escargot

Afin de compter le nombre de fois que l'escargot entre dans un carré gluant, nous pouvons suivre quels carrés sont gluants lorsque l'escargot rampe d'un carré à l'autre. Les différentes sous-tâches correspondent aux différentes manières dont on pourrait représenter l'ensemble des carrés gluants.

Pour les deux premières sous-tâches, on peut représenter chaque carré dans lequel l'escargot pourrait entrer. Cette grille bidimensionnelle de carrés peut se représenter comme une séquence de séquences (par exemple, liste de listes, tableau de tableaux ou vecteur de vecteurs). De façon alternative, la grille bidimensionnelle peut se représenter comme une longue séquence : la première rangée, suivie par la deuxième rangée, puis la troisième, et ainsi de suite. Pour la première sous-tâche, l'escargot peut ramper jusqu'à $20 \times 20 = 400$ carrés vers le sud et également jusqu'à 400 carrés vers l'est. En conséquence, l'escargot commence dans le coin d'une grille de 401×401 . La deuxième sous-tâche est plus exigeante, car elle nécessite de comprendre que l'escargot commence au milieu d'une grille de 801×801 .

Pour la troisième sous-tâche, l'escargot entre jusqu'à 20×1200 carrés dans chaque direction, il est donc difficile de conserver la grille sans qu'elle dépasse la mémoire allouée du correcteur CCC. À la place, on peut conserver uniquement l'ensemble des carrés gluants, en considérant que si un carré n'est pas gluant, alors il ne se trouve pas dans notre ensemble. Ensuite, pour chaque mouvement de l'escargot, on peut vérifier si le carré sur lequel il rampe se trouve dans notre ensemble. L'ensemble peut être représenté par une structure de données séquentielle et, même si on parcourt la totalité de la structure pour chaque mouvement de l'escargot, nous resterons dans la limite de temps.

Dans la sous-tâche finale, l'escargot peut se déplacer très loin de sa position initiale, donc le temps d'exécution devient un enjeu important. Pour résoudre cette sous-tâche, il faut pouvoir déterminer très rapidement si un carré donné se trouve dans notre ensemble de carrés gluants. Une structure de données permettant des recherches rapides est nécessaire ici (par exemple, un dictionnaire en Python, un `HashSet` en Java, ou un `unordered_set` en C++).

Une solution alternative consiste à construire une séquence de tous les carrés dans lesquels l'escargot entre, en incluant les doublons. La réponse correcte est alors la différence entre la longueur de cette séquence et le nombre de carrés distincts. Si un carré visité par l'escargot apparaît x fois dans cette séquence, alors il sera gluant exactement $x - 1$ fois. Si l'on compte le nombre de doublons de façon efficace (par exemple, en triant la liste), on peut obtenir une solution qui respecte la contrainte de temps d'exécution et permet de résoudre la dernière sous-tâche.

J5 Faisceaux lumineux

Le dernier défi du concours CCI Intermédiaire 2026 consiste à répondre à Q questions portant sur N places de stationnement éclairées par L lumières.

Pour la première sous-tâche, $L \leq 1$. S'il n'y a aucune lumière, la réponse à chaque question est N. Sinon, on

peut déterminer si la place de stationnement i est éclairée en vérifiant si $P - S \leq i \leq P + S$, où P est la place de stationnement directement sous la lumière et S est la *portée* de la lumière, c'est-à-dire le nombre de places de stationnement adjacentes qu'elle éclaire de chaque côté.

Pour les trois autres sous-tâches, les limites supérieures des valeurs de Q , N et L indiquent quels algorithmes permettent d'y répondre dans la limite de temps.

Pour la deuxième sous-tâche, L et Q ont de petites valeurs ; ainsi, pour chaque question, on peut considérer chaque lumière individuellement afin de déterminer si une place de stationnement est éclairée ou non.

Pour les deux dernières sous-tâches, L et Q ont des valeurs très grandes, ce qui nous empêche de considérer chaque lumière pour répondre à chaque question. À la place, il faut effectuer certaines préparations avant de répondre aux questions. Concrètement, on peut construire une séquence R (pour « réponses ») où la position i indique si la place de stationnement i est éclairée ou non. Comme pour les problèmes précédents, cette séquence peut être une liste, un tableau ou un vecteur. Il faudra tenir compte du fait qu'une structure séquentielle commence probablement à l'index 0.

Pour de petites valeurs de N , on peut construire R en parcourant chaque lumière et en actualisant chaque place de stationnement qu'elle éclaire. Sinon, on peut parcourir chaque place de stationnement et, pour chaque lumière, vérifier si elle éclaire cette place. L'une ou l'autre approche résout la troisième sous-tâche.

Pour la sous-tâche finale, N est très grand, donc une approche ingénieuse est nécessaire. Par la suite, on décrit deux approches possibles, chacune considérant les intervalles éclairés par chaque lumière. Plus précisément, pour la lumière située au-dessus de la place de stationnement P et ayant une portée S , on considère l'intervalle $[P - S, P + S]$.

- « Fusionner » les intervalles. Par exemple, les intervalles dans l'exemple de données d'entrée sont $[1, 2]$, $[2, 6]$ et $[8, 8]$. Ils peuvent être fusionnés pour former les intervalles $[1, 6]$ et $[8, 8]$. Il existe plusieurs approches pour effectuer cette opération. Le principal défi consiste à garantir que les lumières ou les places de stationnement ne sont parcourues qu'un nombre constant et limité de fois (par exemple, une ou deux).
- Commencer avec une valeur initiale $v(i) = 0$ associée à chaque place de stationnement i . Ensuite, pour chaque intervalle $[a, b]$, ajouter 1 à $v(a)$ et soustraire 1 à $v(b+1)$. À la fin, la somme $v(1) + v(2) + \dots + v(i)$ donne le nombre total de lumières qui éclairent la place de stationnement i . Ces sommes peuvent être calculées en une seule boucle, puis utilisées pour répondre rapidement à chaque question.

Ci-dessus, chaque valeur $P - S$, $P + S$, i , a et b représente une place de stationnement. La contrainte selon laquelle ces valeurs doivent se situer dans l'intervalle $[1, N]$ ne peut pas être ignorée lors de l'implémentation.