Deuxième tour pratique

Livret de solutions

Swiss Olympiad in Informatics

10–13 mars 2017

Parcelle

Il y a N paquets de fleurs avec un nombre connu de graines par paquet. La tâche était d'acheter certains paquets et de les planter dans un jardin de taille $w \times h$ avec une aire maximale de A. Une colonne du jardin ne peut contenir que des graines d'un même paquet et la hauteur de cette colonne doit être au moins le nombre de graines du paquet. Pour un sous-ensemble connu de paquets, w est le nombre de paquets et h et le nombre maximal de graines par paquet et nous devons avoir $w \cdot h \leq A$. Le but était de maximiser le nombre de paquets différents.

L'idée clé pour ce problème était la suivante : Il est toujours mieux de prendre des paquets avec moins de graines. Pourquoi ? La seule chose qui compte à la fin est le nombre de paquets différents. Nous sommes forcés de planter toutes les graines choisies, donc on a possiblement besoin d'une plus grande colonne pour un paquet avec plus de graines. Nous pouvons donc garder un jardin plus petit en prenant les plus petits paquets.

Cela nous mène à un algorithme non optimisé, mais correct : Prends le paquet le plus petit, calcule la taille du jardin et vérifie si tu peux le mettre dans une aire d'au plus A. Si oui, prends le paquet le plus petit suivant, et vérifie l'aire. Si tu as encore de l'espace, prends le paquet suivant, etc.

Pour prendre plusieurs fois le plus petit paquet, nous devons juste trier les paquets selon leur taille au début. Pour calculer la taille maximale des paquets choisis, nous prenons simplement la taille du dernier choisi (tous les précédents sont plus petits). En vérifiant $w \cdot h \leq A$, nous devons faire attention aux overflows – utilise `long long` ou divise les deux côtés par w ou h.

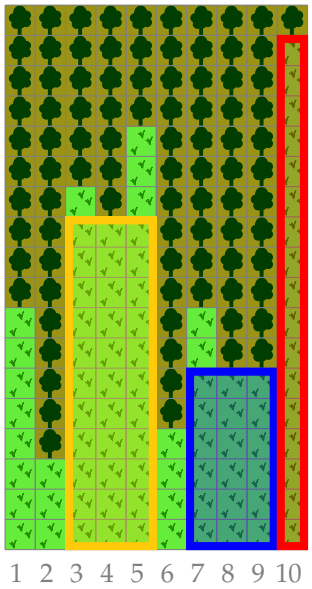
```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
  int a, n;
  cin >> a >> n;
  vector<int> s(n);
  for (int i = 0; i < n; i++)
    cin >> s[i];
  sort(s.begin(), s.end());
  int answer = 0;
  for (int i = 0; i < n; i++)
    if ((i+1) * (long long)s[i] <= a)
      answer = i+1;
  cout << answer << '\n';
}
```

```python
from sys import stdin
a, n = map(int, stdin.readline().split())
answer = 0
for w, h in enumerate(sorted(map(int, stdin.readline().split())), 1):
    if w*h <= a:
        answer = w
print(answer)
```

Pré

L'image à droite devrait bien expliquer cette tâche. Pour une certaine limite de forêt, chosis k rectangles qui touchent le bas du grand rectangle et ne s'intersectent pas entre eux tels que l'aire couverte est la plus grande possible. de plus chaque rectangle peut avoir une largeur maximale de t.



Cette tâche peut être résolue grâce à de la programmation dynamique.

Définissons une fonction magique f qui résout cette tâche :

$f(i, r)$ = aire max couverte par $r$rectangles jusqu'à la position i.

Ce que nous essayons actuellement de faire c'est voir à quoi f ressemble. Certaines valeurs de f sont faciles. Par exemple si $i = 0$ il n'y a pas d'aire à couvrir :

$$f(0, r) = 0$$

De même, si $r = 0$ il ne peut rien y avoir de couvert :

$$f(i, 0) = 0$$

Maintenant, en supposant que nous connaissons la solution pour des i et r plus petits, nous pouvons établir la récurrence suivante :

$$f(i, r) = \max\left(f(i-1, r), \max_{w \in \{1,2,\ldots,\min(t,i)\}} \left\{f(i-w, r-1) + w \cdot \min\{d_{i-w+1}, d_{i-w+2}, \ldots, d_i\}\right\}\right)$$

Qu'est-ce que cela signifie ? Nous voulons connaître la meilleur manière de mettre r rectangles dans des positions de $x = 0$ à $x = i$. Nous connaissons $f(i', r')$ pour tout $i' < i$ et $r' < r$.

— Si le dernier rectangle ne s'étend pas vers la droite, la réponse est juste $f(i-1, r)$.

— S'il ne s'étend pas vers la droite, il a commencé à une des positions $i, i-1, i-2, \ldots, i-t+1$ (supposant qu'elles sont ≥ 0).
Observons un rectange de $i - w + 1$ à i. La largeur de l'aire est w (nous ne regardons pas les coordonnées mais les centres des cases). La hauteur est le minimum de tous les $d_{i-w+1}, d_{i-w+2}, \ldots, d_i$ à l'intérieur du rectangle.
Pour remplir le reste à gauche de $i - w + 1$, nous ajoutons $f(i - w, r - 1)$ – la meilleur solution pour un rectangle de moins dans l'espace restant.

Ce qu'il reste à faire est de choisir un ordre pour les i, r tels que nous accédons seulement aux valeurs déjà calculées auparavant. Faire deux boucles for en partant de 0 est parfaitement adéquat.

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main() {
  int n, k, t;
  cin >> n >> k >> t;
```

```
  vector<int> d(n);
  for (int i=0; i<n; ++i)
    cin >> d[i];
  vector<vector<int> > DP(n, vector<int>(k+1));
  for (int r=1; r<=k; ++r) {
    for (int i=0; i<n; ++i) {
      int value = i ? DP[i-1][r] : 0;
      int h = d[i];
      for (int w=1; w<=t && i-w>=0; ++w) {
        h = min(h, d[i-w+1]);
        value = max(value, DP[i-w][r-1] + h*w);
      }
      DP[i][r] = value;
    }
  }
  cout << DP[n-1][k] << '\n';
}
```

```
from sys import stdin
n, k, t = map(int, stdin.readline().split())
d = [int(stdin.readline()) for _ in range(n)]
DP = [[0]*(k+1)]
for i in range(0, n):
    v = DP[i].copy()
    for r in range(1, k+1):
        x = v[r]
        m = d[i]
        for j in range(0, min(i+1, t)):
            m = min(m, d[i-j])
            x = max(x, DP[i-j][r-1] + m*(j+1))
        v[r] = x
    DP.append(v)
print(max(DP[-1]))
```

Oeufs de Pâques

Pour un graphe et deux sommets a et b, tu dois trouver un sommet c et deux chemins $p = a, \ldots, c$ et $q = b, \ldots, c$ de longueur égale ($|p| = |q|$). Sélectionne c, p et q de telle sorte que $|p|$ (et donc $|q|$) est aussi petit que possible. Dans l'histoire, les sommets a et b sont les points de départ de Tim et Tom, c et le point où ils se rencontrent, et p/q sont les listes de cachettes.

Commençons par ignorer le fait que nous devons minimiser $|p|$ (ou $|q|$) et simplement trouver deux chemins p et q de même longueur.

Une observation qu'on peut faire est que si deux chemins $p = v_1, v_2, \ldots, v_k, c$ et $q = w_1, w_2, \ldots, w_\ell, c$ se rencontrent, mais ne sont pas de même longueur, ne pouvons en allonger un et racourcir l'autre : $p' = v_1, v_2, \ldots, v_{k-1}, v_k, c, w_\ell$ et $q' = w_1, w_2, \ldots, w_{\ell-1}, w_\ell$ se rencontrent toujours en w_ℓ, mais maintenant $|p'| = |p| + 1$ et $|q'| = |q| - 1$.

Pouvons-nous utiliser cette astuce pour faire en sorte que n'importe quelle paire de chemins ait la même longueur ? Non, mais elle fonctionne s'ils ont la même *parité* (i.e. les deux sont pairs ou les deux sont impairs).

Nous devons donc savoir ce qui suit pour tout sommet v :

— Le sommet v est-il atteignable depuis a par un chemin de longueur impaire ?
— Le sommet v est-il atteignable depuis a par un chemin de longueur paire ?
— Le sommet v est-il atteignable depuis b par un chemin de longueur impaire ?
— Le sommet v est-il atteignable depuis a par un chemin de longueur paire ?

En supposant que nous savons cela et que nous pouvons trouver un sommet qu'on peut atteindre par deux chemin de même parité edpuis a et b, nous avons trouvé une solution au problème (nous pouvons juste appliquer l'astuce ci-dessus pour égaliser la longueur des deux chemins).

Et quant à la solution optimale ? Nous pourrions modifier notre liste de questions :

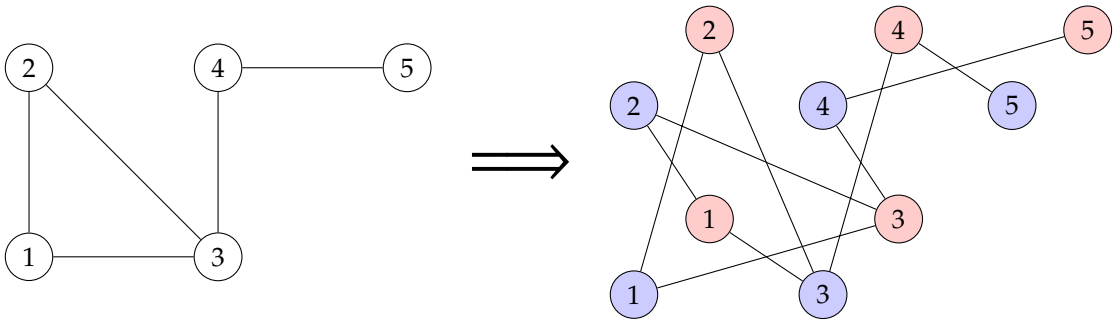
— Quel est le plus court chemin de longueur impaire de a à v ?
— Quel est le plus court chemin de longueur paire de a à v ?
— Quel est le plus court chemin de longueur impaire de b à v ?
— Quel est le plus court chemin de longueur paire de b à v ?

Cela va-t-il trouver la solution minimale ? Oui, car il doit exister un c où les deux se rencontrent et tel que c est atteignable depuis a et b avec des chemins de même parité. Pour une solution minimale, les deux chemins devraient évidemment être de longueur minimale.

Trouverons-nous toutes les solutions avec cette méthode ? Suppose qu'il y a une autre solution qui va de a à c, mais qu'il existe un plus court chemin de même parité. Si nous prenions celui-ci, alors il n'aura pas la même longueur que celui de b à c. Cependant, nous pouvons utiliser l'astuce ci-dessus et trouver un c' sur le second chemin tel que le chemin de a à c' est de même longueur que de b à c'. C'est une meilleure solution et nous l'aurions trouvée avevc notre approche,

Pour implémenter cette idée,nous pourrions faire une BFS avec DP (deux états, atteignables avec une longueur paire ou impaire) et l'éxécuter une fois depuis a et une fois depuis b.

Cependant il y a une bonne astuce pour éviter la DP : Nous dupliquons simplement le graphe. Chaque sommet v_i est divisé en e_i et o_i (pair et impair). Un arc de v_i à v_j sera converti en deux arcs : un de e_i à o_j et un de o_i à e_j. Si nous commençons sur un sommet pair, nous devons ensuite aller sur un sommet impair puis de nouveau vers un sommet pair, etc. onc calculer les chemins les plus courts pairs et impairs sera une simple BFS sur le graphe modifié.



Il y a une autre astuce pour simplifier notre code : Nous cherchons juste pour le chemin le plus court de longueur paire de a à b. Comme la longueur est divisible par 2, nous savons que nous pouvons le séparer en deux parties de même longueur. Cela ne nécessite qu'une DFS.

```cpp
#include <iostream>
#include <algorithm>
#include <queue>
using namespace std;

int main() {
  int n, m, s, t;
  cin >> n >> m >> s >> t;
  vector<vector<int> > g(2*n);
  for (int i=0; i<m; ++i) {
    int a, b;
    cin >> a >> b;
    --a;
    --b;
    g[2*a].push_back(2*b+1);
    g[2*b].push_back(2*a+1);
    g[2*a+1].push_back(2*b);
    g[2*b+1].push_back(2*a);
  }
  vector<int> d(2*n, -1);
  queue<int> q;
  q.push(2*(s - 1));
  d[q.front()] = 0;
  while (!q.empty()) {
    int v = q.front();
    q.pop();
    if (v == 2*(t - 1)) {
      cout << d[v]/2 << '\n';
      return 0;
    }
    for (int w : g[v]) {
      if (d[w] == -1) {
        d[w] = d[v] + 1;
        q.push(w);
      }
    }
  }
  cout << "IMPOSSIBLE\n";
}
```

```python
from sys import stdin
from collections import deque
read = lambda: map(int, stdin.readline().split())
```

```
n, m, s, t=read()
g=[[] for _ in range(2*n)]
for _ in range(m):
    a, b=map(lambda x: x-1, read())
    for i in range(2):
        g[2*a+i].append(2*b+(not i))
        g[2*b+i].append(2*a+(not i))
q=deque()
d=[-1]*(2*n)
q.append(2*(s-1))
d[q[0]]=0
while q:
    v = q.popleft()
    if v == 2*(t-1):
        print(d[v]//2)
        exit(0)
    for w in g[v]:
        if d[w] == -1:
            d[w] = d[v] + 1
            q.append(w)
print("IMPOSSIBLE")
```

Météo

Pour un espace en trois dimensions composés de cubes de $1 \times 1 \times 1$ mètres, réponds à une série de telles requêtes :

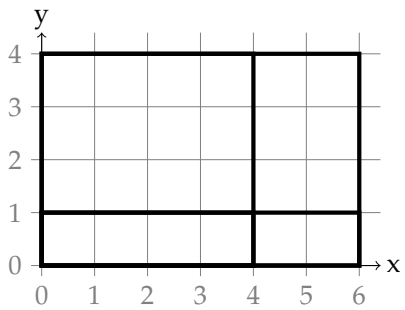
— Query : `M` x_1, y_1, z_1 x_2, y_2, z_2 : Imprime la moyenne des valeurs dans le parallélépipède de sommets (x_1, y_1, z_1) et (x_2, y_2, z_2).
— Mise à jour : `C` x, y, z, d : Assigne la valeur d à (x, y, z)

Bien que la solution complète est plutôt compliquée, il y avait des sous-tâches faciles :

Groupe 1 (25 points) : Les limites étaient assez petites pour qu'une solution bruteforce puisse résoudre cette sous-tâche. Retiens juste toutes les valeurs dans un tableau (`int values[10][10][10]` ou `vector<vector<vector<int>>>`). Pour calculer la moyenne, additionne juste toutes les valeurs nécessaires et divise par le nombre d'éléments.

Groupe 2 (25 points) : Dans cette sous-tâche il n'y avait pas de mises à jour (demande du type "`C`"). Cela signifie que la somme peut être calculée par une somme de préfixes.

Comme décrit durant 2H, nous pouvons utiliser une somme de préfixes pour calculer une aire de rectangle 2D (voir 2H).



Mais comme nous sommes intéressés par la 3D et pas par la 2D, nous trouvons ceci après un peu de réflexion :

$$\begin{aligned}\sum_{i=x_0}^{x_1}\sum_{j=y_0}^{y_1}\sum_{k=z_0}^{z_1} a_{ijk} = \big(&s(x_1, y_1, z_1)\\ &- s(x_0-1, y_1, z_1) - s(x_1, y_0-1, z_1) - s(x_1, y_1, z_0-1)\\ &+ s(x_0-1, y_0-1, z_1) + s(x_0-1, y_1, z_0-1) + s(x_1, y_0-1, z_0-1)\\ &- s(x_0-1, y_0-1, z_0-1)\big).\end{aligned}$$

onc comment précalculer $s(x, y, z)$ efficacement ? Nous utiisons juste l'égalité $a_{xyz} = \sum_{i=x}^{x}\sum_{j=y}^{y}\sum_{k=z}^{z} a_{ijk}$ pour obtenir :

$$\sum_{i=x}^{x}\sum_{j=y}^{y}\sum_{k=z}^{z} a_{ijk} = \big(s(x,y,z) \\ - s(x-1,y,z) - s(x,y-1,z) - s(x,y,z-1) \\ + s(x-1,y-1,z) + s(x-1,y,z-1) + s(x,y-1,z-1) \\ - s(x-1,y-1,z-1)\big),$$

ou, résolu pour $s(x,y,z)$,

$$s(x,y,z) = a_{ijk} \\ + s(x-1,y,z) - s(x,y-1,z) - s(x,y,z-1) \\ - s(x-1,y-1,z) + s(x-1,y,z-1) + s(x,y-1,z-1) \\ + s(x-1,y-1,z-1)\big).$$

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main() {
  cin.tie(0);
  ios::sync_with_stdio(false);

  int l, w, h; cin >> l >> w >> h;
  vector<vector<vector<long long> > >
    prefixsum(l+1, vector<vector<long long> >(w+1, vector<long long>(h+1)));

  auto query = [&](int x0, int y0, int z0,
                   int x1, int y1, int z1) {
    return (- prefixsum[x0][y0][z0]
            + prefixsum[x1][y0][z0] + prefixsum[x0][y1][z0] + prefixsum[x0][y0][z1]
            - prefixsum[x0][y1][z1] - prefixsum[x1][y0][z1] - prefixsum[x1][y1][z0]
            + prefixsum[x1][y1][z1]);
  };

  for (int i=0; i<h; ++i)
    for (int j=0; j<w; ++j)
      for (int k=0; k<l; ++k) {
        int x;
        cin >> x;
        prefixsum[k+1][j+1][i+1] = x - query(k, j, i, k+1, j+1, i+1);
      }

  int q; cin >> q;
  while (q--) {
    char t;
    cin >> t;
    if (t == 'C') {
      int x, y, z, d;
      cin >> x >> y >> z >> d;
      continue;
    } else {
      int x1, y1, z1, x2, y2, z2;
      cin >> x1 >> y1 >> z1 >> x2 >> y2 >> z2;
```

```
40        cout << query(x1, y1, z1, x2, y2, z2)/(double)((x2-x1)*(y2-y1)*(z2-z1)) << '\n';
41      }
42    }
43  }
```

```
1  from sys import stdin
2
3  l, w, h = map(int, stdin.readline().split())
4  prefixsum = [[[0]*(h+1) for _ in range(w+1)] for _ in range(l+1)]
5
6  def query(x0, y0, z0, x1, y1, z1):
7      return sum((s0*s1*s2*prefixsum[x][y][z])
8                 for s0, x in ((-1, x0), (1, x1))
9                 for s1, y in ((-1, y0), (1, y1))
10                for s2, z in ((-1, z0), (1, z1)))
11
12 for i in range(h):
13     for j in range(w):
14         for k, x in enumerate(map(int, stdin.readline().split())):
15             prefixsum[k+1][j+1][i+1] = x - query(k, j, i, k+1, j+1, i+1)
16
17 for _ in range(int(stdin.readline())):
18     q = tuple(stdin.readline().split())
19     if q[0]=='C':
20         continue
21     else:
22         x1, y1, z1, x2, y2, z2 = map(int, q[1:])
23         print(query(x1, y1, z1, x2, y2, z2)/((x2-x1)*(y2-y1)*(z2-z1)))
```

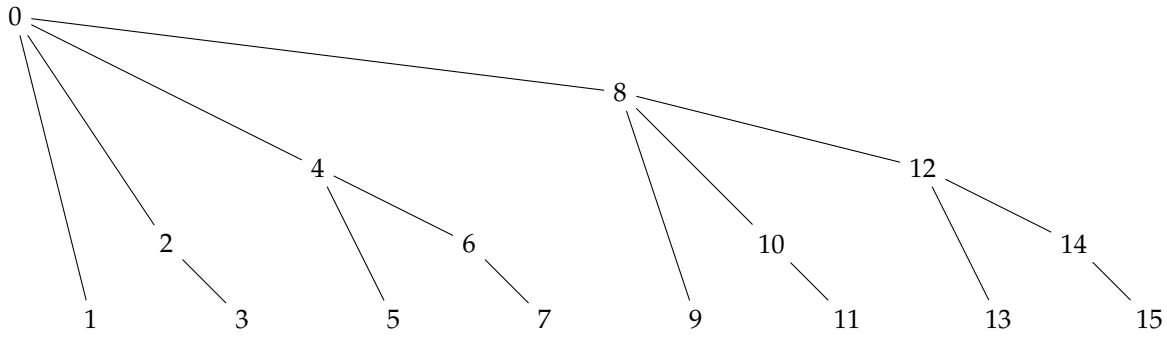
Groupes 1 + 2 (50 points) : Les deux approches peuvent clairement être combinées pour un total de 50 points.

Tous les groupes (100 points) : Regardons la solution complète. La partie difficile est de répondre rapidement aux queries *et* aux mises à jour en même temps (la solution brute-force a des mises à jour rapides et des queries lentes, la somme de préfixe des queries rapides et des mises à jour lentes).

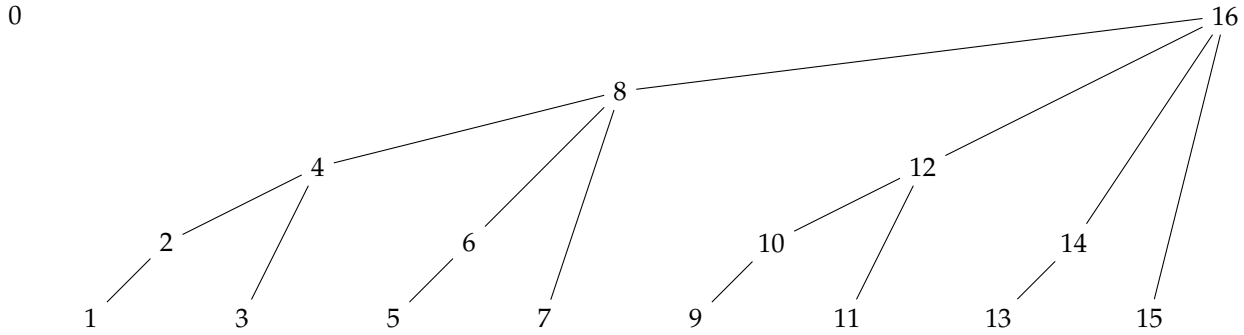
Pour une dimension, une structure de données connue qui résoud cela et ien d'autres problèmes est un segment tree. Une structure de données plus simple qui fait la même chose pour des sommes de préfixes est le fenwick tree (parfois appelé BIT pour binary indexed tree, arbre binaire indexé).

Un fenwick tree de $n$éléments est un tableau de longueur $n + 1$, où la i-ème entrée stocke la somme des éléments des indices $i - 2^k + 1$ à i (i inclus) où 2^k est la plus grande puissance de 2 qui divise i. Cela signifie que p. ex. la cinquième entrée stocke seulement a_5, la sixième $a_5 + a_6$, et la huitième $a_1 + a_2 + \ldots + a_8$.

Regarde cette image :



La coordonnée y représente l'intervalle de chaque sommet (premier niveau (le plus bas) : divisible par 1, deuxième niveau : divisible par 2, troisième level : divisible par 4, etc.). Pour obtenir la somme de par exemple $a_1 + \ldots + a_{11}$, nous devons additionner $a_{11} + (a_9 + a_{10}) + (a_1 + \ldots + a_8)$, ce qui signifie que nous additionnons les valeurs des sommets 11, 9 et 8. Ce sont exactement les parents de a_{11}.



Cet arbre miroir représente les sommets que nous devons mettre à jour si nous changeons une valeur. Si nous ajoutons x à a_{11}, nous ne devons pas seulement mettre à jour le sommet 11, mais aussi le 12 (qui stocke $a_{11} + a_{12}$) et le 16 (qui stocke $a_1 + \ldots + a_{16}$.

La raison pour laquelle cet arbre est si utile est qu'il est très facile de se déplacer d'un sommet à son parent : ans le premier arbre, le parent de i est $i - (i\&(-i))$ (où & est le ET binaire), dans le deuxième arbre, le parent de i est $i + (i\&(-i))$. (Pour se conformer strictement au standard C++ $i - (i\&(-i))$ devrait être écrit $(i\&(i-1))$ et $i + (i\&(-i))$ comme $(x|(x-1)) + 1$ parce que les opérations sur les bits avec des nombres négatifs ont un comportement indéfini – cependant sur les machines avec complément à 2 les bit-tricks précédants fonctionnent en général et sont plus simples à retenir grâce à leur symétrie.)

onc comment faire marcher le fenwick tree pour la 2D? Nous prenons l'abre précédant mais au lieu de stocker un entier nous stockons un autre fenwick tree. Le j-ème élément du i-ème fenwick tree représente $s(i, j)$. Les mises à jour et les queries seront comme avant, sauf que sur la première couche nous ajoutons la valeur au j-ème élément du segment tree du bas et nous faisons une query sur le j-ème élément. Pour la 3D c'est à peu près la même chose.

Si cette introduction aux Fenwick trees était trop rapide, il y a d'autre tutoriaux en ligne, incluant la 2D : `https://www.topcoder.com/community/data-science/data-science-tutorials/binary-indexed-trees/`

Ci-dessous n'est présenté qu'une solution en Python. L'implémentation en C++ est laissée en exercice au lecteur.

```
1 from sys import stdin
2
```

```python
class FenwickTree3D:
    def __init__(self, x, y, z):
        self.x, self.y, self.z = x+1, y+1, z+1
        self.d = [[[0]*self.z for _ in range(self.y)] for _ in range(self.x)]

    def query(self, x, y, z):
        s = 0
        i = x
        while i:
            j = y
            while j:
                k = z
                while k:
                    s += self.d[i][j][k]
                    k -= k&-k
                j -= j&-j
            i -= i&-i
        return s

    def query_rect(self, x1, x2, y1, y2, z1, z2):
        get = self.query
        return (get(x2, y2, z2)
                - get(x1, y2, z2) - get(x2, y1, z2) - get(x2, y2, z1)
                + get(x1, y1, z2) + get(x1, y2, z1) + get(x2, y1, z1)
                - get(x1, y1, z1))

    def add(self, x, y, z, v):
        i = x+1
        while i < self.x:
            di = self.d[i]
            j = y+1
            while j < self.y:
                dij = di[j]
                k = z+1
                while k < self.z:
                    dij[k] += v
                    k += k&-k
                j += j&-j
            i += i&-i

    def set(self, x, y, z, v):
        self.add(x, y, z, v - self.query_rect(x, x+1, y, y+1, z, z+1))


l, w, h = map(int, stdin.readline().split())
ft = FenwickTree3D(l, w, h)
for k in range(h):
    for j in range(w):
        for i, x in enumerate(map(int, stdin.readline().split())):
            ft.add(i, j, k, x)
for _ in range(int(stdin.readline())):
    q=tuple(stdin.readline().split())
    if q[0]=='C':
        x, y, z, v = map(int, q[1:])
        ft.set(x, y, z, v)
    else:
        x1, y1, z1, x2, y2, z2 = map(int, q[1:])
        print(ft.query_rect(x1, x2, y1, y2, z1, z2)/((x2-x1)*(y2-y1)*(z2-z1)))
```