Zweite Runde Praxis

Lösungsbüchlein

Swiss Olympiad in Informatics

10.–13. März 2017

Schrebergarten

Bekannt sind N Packungen Blumensamen, wobei jede Packung eine andere Blumensorte beinhaltet und die Anzahl Samen bekannt ist. Die Aufgabe ist, einige dieser Packungen zu kaufen und in einem Garten der Grösse $w \times h$ anzupflanzen. Der Garten darf dabei höchstens A gross sein. Eine Spalte des Gartens enthält ausschliesslich Samen aus einer Packung und muss mindestens so hoch sein wie die Anzahl Samen dieser Packung.

Für eine bekannte Liste von Samen ist deshalb w die Anzahl Packungen und h die maximale Anzahl Samen in einer dieser Packungen. Es muss dabei gelten, dass $w \cdot h \leq A$. Das Ziel der Aufgabe ist es, die Anzahl unterschiedlicher Packungen zu maximieren.

Es ist immer besser, Packungen mit weniger Samen zu nehmen. Dies, weil es am Ende nur darauf ankommt, wie viele verschiedene Packungen genommen wurden. Mit einer grösseren Packung wäre man möglicherweise gezwungen, den Garten höher zu machen, was seine Fläche vergrössert.

Damit führt zur ersten Version eines Algorithmus: Nimm die kleinste Packung, berechne die Grösse des Gartens und überprüfe, ob seine Fläche höchstens A ist. Falls er noch klein genug ist, nimm die nächste Packung, und überprüfe die Grösse des Gartens. Falls er immer noch klein genug ist, nimm die nächste Packung, usw.

Wie implementiert man das am besten? Um immer die kleinste übrige Packung zu nehmen reicht es, die Packungen am Anfang nach Grösse zu sortieren. Um die maximale Grösse der bisherigen Packungen zu bestimmen kann man einfach die Grösse der letzten Packung nehmen (da alle vorherigen Packungen kleiner waren). Bei der Überprüfung $w \cdot h \leq A$ sollte man auf einen Overflow aufpassen – und entweder einen "`long long`" benutzen oder beide Seiten durch w oder h dividieren.

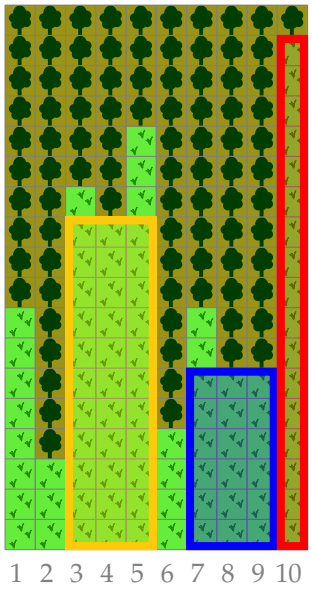
```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

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)
```

Weidland

Durch das Bild rechts ist die Aufgabe fast selbserklärend. Gegeben eine Waldgrenze, wähle k Rechtecke aus, welche bis ganz nach unten gehen und sich nicht überlappen, so dass die von ihnen umschlossene Fläche maximal ist. Alle Rechtecke dürfen höchstens t breit sein.



Diese Aufgabe kann mit dem Konxept der dynamischen Programmierung gelöst werden.

Dazu definieren wir uns eine "magische" Funktion, die die Aufgabe löst:

$$f(i, r) = \text{max. Fläche von } r \text{ Rechtecken bis zu Position } i.$$

Nun ist das Ziel, f herauszufinden. Einige Werte von f sind einfach. Für $i = 0$ zum Beispiel ist die Fläche immer leer:

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

Bei $r = 0$ kann ebenfalls keine Fläche überdeckt sein:

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

Angenommen, die Lösung für kleinere i und r ist bekannt. Dann gilt folgende rekursive Formel:

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

Was bedeutet das? Wir möchten die beste Lösung wissen um r Rechtecke in den Positionen von $x = 0$ bis $x = i$ zu platzieren. Dabei ist $f(i', r')$ für $i' < i$ und $r' < r$ bekannt.

- Falls das letzte Rechteck gar nicht bis ganz nach rechts geht, dann ist die Lösung einfach $f(i-1, r)$.
- Falls das letzte Rechteck bis ganz nach rechts geht, dann ist sein linkes Ende auf einer der Positionen $i, i-1, i-2, \ldots, i-t+1$ (sofern alle ≥ 0).

  Betrachten wir eines dieser Rechtecke mit beliebiger Breite w. Es geht von $i-w+1$ bis i. Seine Höhe kann höchstens so gross sein wie das Minimum aus $d_{i-w+1}, d_{i-w+2}, \ldots, d_i$, den Höhen der Grasstreifen innerhalb des Rechtecks.

  Um den Rest links vom letzten Rechteck aufzufüllen, also alles links von $i-w+1$, können wir auf f zurückgreifen: Wir addieren $f(i-w, r-1)$, die beste Lösung mit einem Rechteck weniger im verbleibenden Platz.

Mit dieser Rekursion ist die Aufgabe schon fast gelöst. Wir müssen nur noch eine Reihenfolge von i, r so auswählen, dass wir nur auf bereits berechnete Werte von f zugreifen. Zwei aufsteigende for-Schleifen machen genau das.

```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';
}
```

```python
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]))
```

Ostereier

Gegeben einen ist Graph und zwei Knoten a und b. Gesucht ist ein Knoten c und zwei Pfade $p = a, \ldots, c$ und $q = b, \ldots, c$ gleicher Länge ($|p| = |q|$). Wähle c, p und q so, dass $|p|$ (und damit auch $|q|$) so klein wie möglich sind. In der Aufgabenbeschreibung waren a und b die Startpositionen von Tim und Tom, c der Treffpunkt der beiden und p/q die Listen der Verstecke.

Versuchen wir zuerst, die Aufgabe ohne die Bedingung zu lösen, dass $|p|$ (und $|q|$) minimiert werden müssen und suchen einfach nur zwei Pfade p und q mit gleicher Länge.

Wenn wir zwei Pfade $p = v_1, v_2, \ldots, v_k, c$ und $q = w_1, w_2, \ldots, w_\ell, c$ haben, diese aber nicht die gleiche Länge besitzen, dann können den kürzeren von beiden verlängern und den längeren der beiden verkürzen, in dem wir den Treffpunkt verschieben: $p' = v_1, v_2, \ldots, v_{k-1}, v_k, c, w_\ell$ und $q' = w_1, w_2, \ldots, w_{\ell-1}, w_\ell$ treffen sich neu in w_ℓ, mit den Längen $|p'| = |p| + 1$ und $|q'| = |q| - 1$.

Dieser Trick funktioniert leider nur für Pfade, welche die gleiche Parität (gerade/ungerade) besitzen.

Also müssen wir für einen Knoten v folgende Informationen kennen:

- Ist Knoten v von a aus erreichbar mit einem Pfad gerader Länge?
- Ist Knoten v von a aus erreichbar mit einem Pfad ungerader Länge?
- Ist Knoten v von b aus erreichbar mit einem Pfad gerader Länge?
- Ist Knoten v von b aus erreichbar mit einem Pfad ungerader Länge?

Sind die Antworten zu diesen Fragen bekannt, dann können wir eine Lösung rekonstruieren. Wir nehmen einen beliebigen Knoten v, welcher zweie Pfade gleicher Parität von a und b aus hat. Dann machen wir den kürzeren beider Pfade länger und den längeren kürzer bis beide gleich lang sind.

Wenden wir uns wieder dem vollen Problem zu, bei dem eine minimale Lösung gefragt ist. Wir können unsere Fragen modifizieren zu:

- Wie lange ist der kürzeste Pfad gerade Länge von a to v?
- Wie lange ist der kürzeste Pfad ungerade Länge von a to v?
- Wie lange ist der kürzeste Pfad gerade Länge von b to v?
- Wie lange ist der kürzeste Pfad ungerade Länge von b to v?

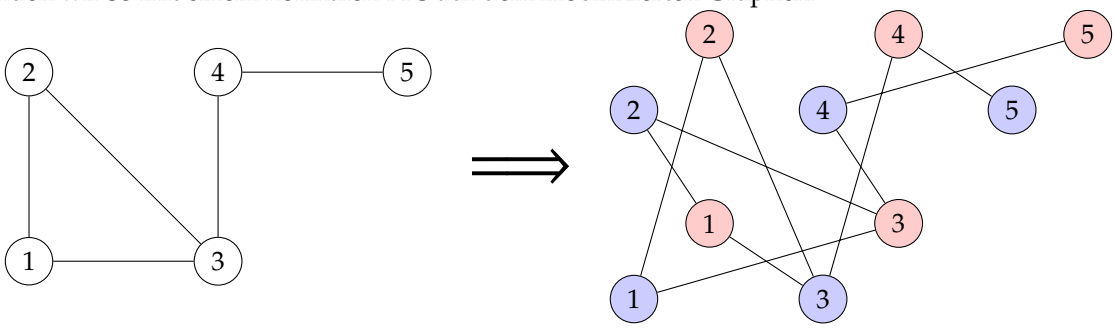
Dadurch können wir die minimale Lösung finden. Falls eine Lösung existiert, gibt, muss es auch ein c geben, welches von a und b aus mit Pfaden gleicher Parität erreichbar ist. Für eine minimale Lösung sollten beide Pfade so kurz wie möglich sein.

Finden wir so alle Lösungen? Falls es eine andere Lösung gäbe, bei dem wir von a nach c gehen, es aber einen kürzeren Pfad gleicher Parität gäbe. Falls wir diesen nehmen würden, hätte er aber nicht die gleiche Länge wie der von b nach c. Dann könnten wir aber den Trick oben anwenden und ein c' auf dem zweiten Pfad auswählen, so dass a bis c' gleich lang ist wie b bis c'. Damit hätten wir eine besser Lösung gefunden.

Eine Möglichkeit, diese Idee zu implementieren ist eine Breitensuche (BFS) mit dynamischer Programmierung (DP): Pro Knoten müssen wir die Antwort auf zwei Ja/Nein-Fragen wissen: Ist er erreichbar mit einem Pfad, der gerade/ungerade ist? Diesen BFS lassen wir einmal von a aus und einmal von b aus laufen.

Es gibt aber eine schönere Lösung ohne DP, nämlich mit Graphduplikation. Wir teilen den Knoten v_i in zwei Hälften g_i und u_i (gerade/ungerade) auf. Eine Kante von v_i nach v_j im

ursprünglichen Graph ersetzen wir mit zwei Kanten, einer von g_i nach u_j und eine von u_i nach g_j. Falls wir auf einem geraden Knoten starten, können wir nur zu einem ungeraden gehen; und müssen dann wieder zurück auf einen geraden gehen. Den kürzesten graden/ungeraden Pfad finden wir so mit einem normalen BFS auf dem modifizierten Graphen.



Es gibt noch eine weitere Idee, mit der wir unseren Code vereinfachen können: Es reicht, den kürzesten Pfad gerader Länge von a nach b zu kennen. Weil seine Länge durch 2 teilbar ist, können wir ihn in zwei Teile gerader Länge aufteilen. Dadurch benötigen wir nur noch einen BFS-Durchgang.

```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";
}
```

```
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")
```

Meteo

Für einen 3D-Raum, aufgeteilt in $1 \times 1 \times 1$-Würfel mit Zahlen darin, soll eine Reihe folgender Fragen beantwortet werden:

- Query: `M` x_1, y_1, z_1 x_2, y_2, z_2: Bestimme den Mittelwert der Zahlen innerhalb des Quaders mit Ecken (x_1, y_1, z_1) und (x_2, y_2, z_2).
- Update: `C` x, y, z, d: Setze den Wert in (x, y, z) auf d

Die vollständige Lösung ist zwar relativ schwierig, es gab aber Punkte in zwei einfachen Teilaufgaben zu holen:

Testgruppe 1 (25 points): Die Limits waren klein genug, um diese Teilaufgabe mit Brute-Force zu lösen. Einfach alle Werte in einem dreidimensionalem Array (`int values[10][10][10]` bzw. `vector<vector<vector<int>>>`) abspeichern. Um den Durchschnitt zu berechnen bestimme die Summe aller relevanten Werte und dividiere durch die Anzahl Elemente darin.

Testgruppe 2 (25 points): In dieser Teilaufgabe gab es keine Updates (Abfragen vom Typ "`C`"). Das bedeutet, dass die Summe mittels Prefix-Sum bestimmt werden kann.

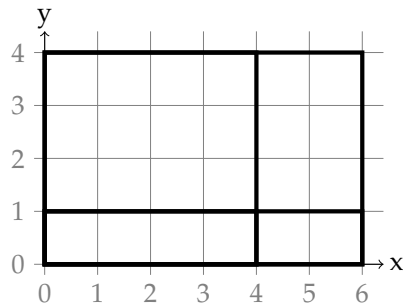
Wie während 2H erklärt können wir Präfix-Summe benutzen um die Summe eines 2D-Rechtecks zu berechnen: Sei a_{ij} der Wert in (i, j) und nehme an, alle Präfix-Summen sind bereits vorberechnet:

$$s(x, y) = \sum_{i=0}^{x} \sum_{j=0}^{y} a_{ij}$$

Dann ist die Summe von (x_0, y_0) bis (x_1, y_1):

$$\sum_{i=x_0}^{x_1} \sum_{j=y_0}^{y_1} a_{ij} = s(x_1, y_1) - s(x_0 - 1, y_1) - s(x_1, y_0 - 1) + s(x_0, y_0).$$

Diese Formel kann man sich leicht selber herleiten in dem man ein Beispiel durchrechnet. Die Summe zwischen $(4, 1)$ und $(6, 4)$ sieht aus wie $s(6, 4) - s(4, 4) - s(6, 1)$, aber dann haben wir das untere rechte Rechteck doppelt abgezogen, also müssen wir es wieder einmal hinzufügen. Am Ende erhalten wir $s(6, 4) - s(4, 4) - s(6, 1) + s(4, 1)$.



Aber da wir in 3D und nicht 2D interessiert sind, müssen wir diese Formel verallgemeinern. Nach einiger Überlegung kommt man auf folgende Formel:

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

Wie können wir $s(x,y,z)$ effizient vorberechnen? Wenn wir $a_{xyz} = \sum_{i=x}^{x}\sum_{j=y}^{y}\sum_{k=z}^{z} a_{ijk}$ benutzen, erhalten wir:

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

oder, aufgelöst auf $s(x,y,z)$,

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

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

```
from sys import stdin

l, w, h = map(int, stdin.readline().split())
prefixsum = [[[0]*(h+1) for _ in range(w+1)] for _ in range(l+1)]

def query(x0, y0, z0, x1, y1, z1):
    return sum((s0*s1*s2*prefixsum[x][y][z])
                for s0, x in ((-1, x0), (1, x1))
                for s1, y in ((-1, y0), (1, y1))
                for s2, z in ((-1, z0), (1, z1)))

for i in range(h):
    for j in range(w):
        for k, x in enumerate(map(int, stdin.readline().split())):
            prefixsum[k+1][j+1][i+1] = x - query(k, j, i, k+1, j+1, i+1)

for _ in range(int(stdin.readline())):
    q = tuple(stdin.readline().split())
    if q[0]=='C':
        continue
    else:
        x1, y1, z1, x2, y2, z2 = map(int, q[1:])
        print(query(x1, y1, z1, x2, y2, z2)/((x2-x1)*(y2-y1)*(z2-z1)))
```

Testgruppen 1 + 2 (50 points): Beide Ansätze können kombiniert werden und führen zu einer 50-Punkte-Lösung.

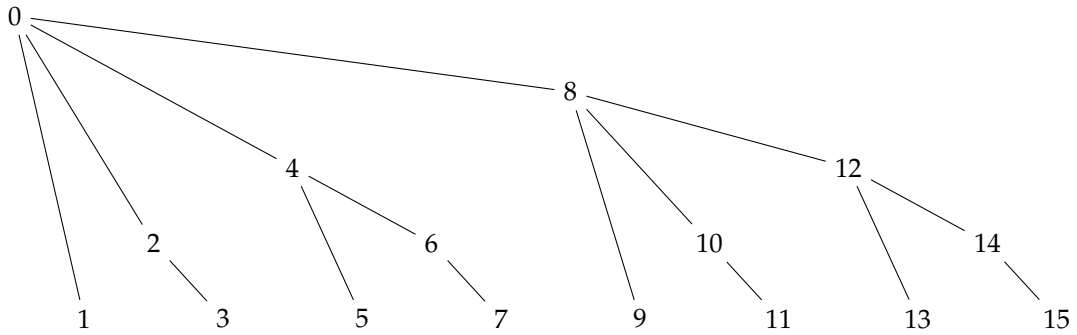
All groups (100 points): Die Hauptschwierigkeit der vollen Lösung ist es, sowohl schnelle Queries als auch schnelle Updates zu haben (mit der Brute-Force-Lösung gibt es schnelle Updates aber langsame Queries, mit der Prefix-Summe schnelle Queries aber langsame Updates).

Im eindimensionalen Fall löst ein Segment-Baum dieses Problem. Segmentbäume sind im Davos-Camp vorgestellt worden. Eine leicht einfachere Datenstruktur, welche für Präfixsummen funktioniert, ist der Fenwick-Tree (manchmal auch BIT für "Binary Indexed Tree" genannt).

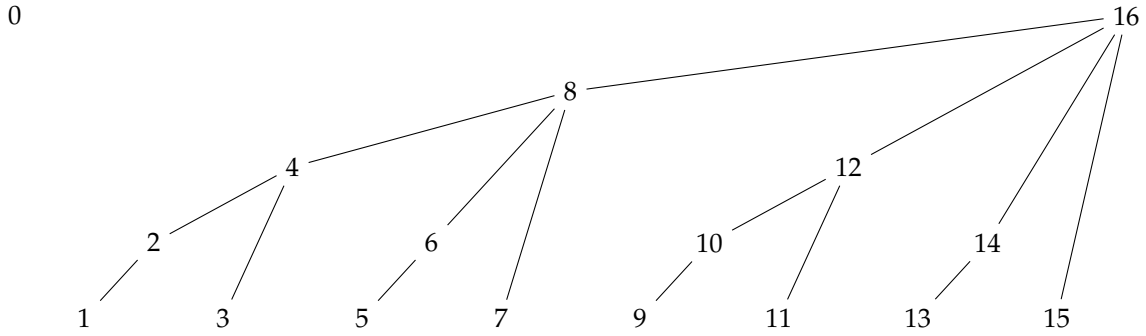
Ein Fenwick-Tree auf n Elementen ist ein Array der Länge $n + 1$, wobei das i-te Element die Summe aller Elemente von Indizes $i - 2^k + 1$ bis i (inklusive i) speichert. 2^k ist die grösste Zweierpotenz, welche i teilt. Das bedeutet z.B., dass das the 5-te Element a_5, das 6-te Element $a_5 + a_6$ und das 8-te Element $a_1 + a_2 + \ldots + a_8$ abspeichert.

Das folgende Bild hilft, das zu verstehen. Die y-Koordinate steht für den Bereich jedes Elements (erste (unterste) Ebene: nur durch 1 teilbar, zweite Ebene: durch 2 teilbar, dritte Ebene:

durch 4 teilbar, usw.). Um die Summe of von z.B. $a_1 + \ldots + a_{11}$ zu bestimmen, müssen wir $a_{11} + (a_9 + a_{10}) + (a_1 + \ldots + a_8)$, berechnen, also können wir die Werte der Elemente 11, 10 und 8 aufaddieren. Das sind genau die Vorgänger von a_{11}.



Ein gespiegelter Baum beschreibt die Knoten, die wir ändern müssen, wenn wir einen Wert verändern. Falls wir x zu a_{11} addieren müssen wir das nicht nur bei Element 11 tun, sondern auch bei 12 (welcher $a_{11} + a_{12}$ beinhaltet) und Element 16 (welches $a_1 + \ldots + a_{16}$ beinhaltet).



Dieser Baum sieht zwar unpraktisch aus, ist aber genau das Gegenteil. Es ist nämich extrem einfach, von einem Element zu seinem Vater zu springen. Im ersten Baum ist der Vater von i auf Index $i - (i\&(-i))$ und im zweiten Baum ist der Vater auf Index $i + (i\&(-i))$. (Streng nach C++-Standard sollte $i - (i\&(-i))$ als $(i\&(i-1))$ und $i + (i\&(-i))$ als $(x|(x-1)) + 1$ umgeschrieben werden, da bitweise Operationen auf negativen Ganzzahlen sich nicht überall gleich verhalten. Auf Maschinen mit Zweierkomplement sollten die Bit-Tricks aber funktionieren und sind dank ihrer Symmetrie einfacher zu merken.)

Wie können wir diesen Fenwick-Tree für 2D zum Laufen bringen? Wir nehmen den Baum von vorher, aber anstatt eine Zahl zu speichern speichern wir einen anderen Fenwick-Tree darin ab. Das j-te Element des i-ten Fenwick-Tree repräsentiert $s(i, j)$ (bzw. die Abfragen auf das j-te Elements des i-ten Fenwick-Tres). Die Updates und Queries funktionieren analog, nur dass wir im obersten Fenwick-Tree nicht Zahlen addieren, sondern die Werte der Abfrage auf das j-te Element. For 3D geht es analog.

Falls diese Einführung in Fenwick-Trees zu schnell war, gibt es gute Tutorials online, welche auch den 2D-Fall genau erklären:

`https://www.topcoder.com/community/data-science/data-science-tutorials/binary-indexed-trees/`

Unten ist Python-Code. Die Implementierung in C++ überlassen wir dem Leser als Übung.

```python
from sys import stdin

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)))
```