# Second Round Practical

# Solution Booklet

Swiss Olympiad in Informatics

March 10–13, 2017

# Allotment

There are $N$ flower packets with a known number of seeds per packet. The task was to buy some packets and sow them inside a $w \times h$ garden with area at most $A$. One column of the garden can only contain seeds of one packet and the height of that column must be at least the number of seeds in that packet. For some known subset of packets, $w$ is the number of packets and $h$ is the maximum number of seeds inside the packets and we must have $w \cdot h \leq A$. The goal was to maximize the number of different packets.

The key insight for this problem is: It is always better to take packets with less seeds. Why? The only thing that counts in the end is the number of different packets. We are forced to plant all of the chosen seeds, thus possibly needing a larger column for a packet with a large number of seeds. We can keep the garden small by taking the small packets.

This brings us to an unoptimized, but correct algorithm: Take the smallest packet, calculate the garden size and check if you can put it into an area at most $A$. If yes, take the next smallest packet, and check the area. If you still have space, take the next smallest packet, etc.

To repeatedly take the smallest packet, we just sort the packets by size at the beginning. To compute the maximum size of the chosen packets, we just take the size of the last one taken (all previous packets are smaller). While verifying $w \cdot h \leq A$, we need to be careful for overflows – use `long long` or divide both sides by $w$ or $h$.

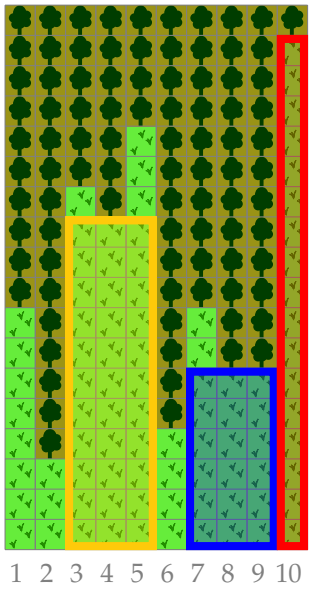
```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

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)
```

# Meadow

The picture to the right should pretty much explain this task. In words: Given some forest boundary, select $k$ rectangles that touch the bottom and don't intersect with each other, such that the enclosed area is as large as possible. Furthermore, each rectangle can have a width of at most $t$.




This task can be solved using dynamic programming.

Let's define a magic function $f$ that solves this task:

$$f(i, r) = \text{max area covered by } r \text{ rectangles up to position } i.$$

What we're trying now is to figure out how $f$ looks like. Some values of $f$ are easy. For example if $i = 0$ there is no area to cover:

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

Also, if $r = 0$ there can't be anything covered:

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

Now assuming we know the solution for smaller $i$ and $r$, we can write down the following recurrence:

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

What does this mean in words? We want to know the best way to put $r$ rectangles in positions from $x = 0$ to $x = i$. We know $f(i', r')$ for all $i' < i$ and $r' < r$.

- If the last rectangle doesn't extend to the right, the answer is just $f(i-1, r)$.
- If the last rectangle does extend to the right, it has started at one of the positions $i, i-1, i-2, \ldots, i-t+1$ (assuming they are all $\geq 0$).

  Let's look at a rectangle from $i-w+1$ to $i$. The width of the area is $w$ (we are not looking at coordinates but at centers of grid tiles). The height is the minimum of all $d_{i-w+1}, d_{i-w+2}, \ldots, d_i$ inside the rectangle.

  To fill up the rest to the left of $i-w+1$ we add $f(i-w, r-1)$ – the best solution for one rectangle less in the remaining space.

What remains is to chose an order for the $i, r$ so that we only access the ones we calculated already. Doing two for-loops starting from 0 is perfectly fine.

```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]))
```

# Easter Eggs

Given a graph and two vertices $a$ and $b$, you need to find a vertex $c$ and two paths $p = a, \ldots, c$ and $q = b, \ldots, c$ which are of equal length ($|p| = |q|$). Select $c$, $p$ and $q$ such that $|p|$ (and thus $|q|$) is as small as possible. In the story, the two vertices $a$ and $b$ are the starting points of Tim and Tom, $c$ is the point in which they meet, and $p/q$ are the lists of hiding places.

Let's first ignore the fact that we have to minimize $|p|$ (or $|q|$) and just find two paths $p$ and $q$ of equal length.

An observation one can make is that if two paths $p = v_1, v_2, \ldots, v_k, c$ and $q = w_1, w_2, \ldots, w_\ell, c$ meet, but are not of equal size, we can make one of them longer and one of them shorter: $p' = v_1, v_2, \ldots, v_{k-1}, v_k, c, w_\ell$ and $q' = w_1, w_2, \ldots, w_{\ell-1}, w_\ell$ still meet at $w_\ell$, but now $|p'| = |p| + 1$ and $|q'| = |q| - 1$.

Can we use this trick to make any pair of paths have equal length? No, but it works if they have the same *parity* (i.e. both are even or both are odd).

Therefore, we need to know the following for each vertex $v$:

- Is vertex $v$ reachable from $a$ by a path of odd length?
- Is vertex $v$ reachable from $a$ by a path of even length?
- Is vertex $v$ reachable from $b$ by a path of odd length?
- Is vertex $v$ reachable from $b$ by a path of even length?

Assuming we know that and we can find a vertex that is reachable by two paths of the same parity from $a$ and $b$, we have found a solution to the problem (we can then just apply the trick from above to make both paths equal).

So what about the minimal solution? We could modify our list of questions to:

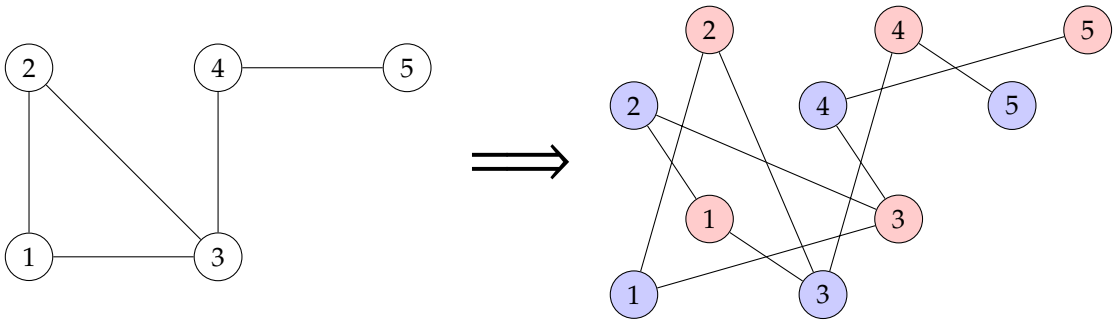
- What is the shortest path of odd length to go from $a$ to $v$?
- What is the shortest path of even length to go from $a$ to $v$?
- What is the shortest path of odd length to go from $b$ to $v$?
- What is the shortest path of even length to go from $b$ to $v$?

Will this find the minimal solution? Well yes, because there must be some $c$ where both meet and that $c$ is reachable from $a$ and $b$ with paths form the same parity. For a minimal solution, clearly both paths should be minimal.

Will we find all solutions with this approach? Assume there is some other solution where we will go from $a$ to $c$, but there exists a shorter path of the same parity. If we would take that one, then it doesn't have the same length as the one from $b$ to $c$. However, we can use the trick from above and find a $c'$ on the second path, such that $a$ to $c'$ is of equal length as $b$ to $c'$. This is a better solution and we would have found it with our approach.

To implement this idea, we could do a BFS with DP (two states, reachable with odd length or even length) and run it once from $a$ and once from $b$.

However, there is a nice trick to avoid DP: We just duplicate the graph. Any node $v_i$ will be split into $e_i$ and $o_i$ (even and odd). An edge from $v_i$ to $v_j$ in the original graph will be converted to two edges: one from $e_i$ to $o_j$ and one from $o_i$ to $e_j$. If we start on one of the even nodes, we have to go to one of the odd nodes next, then back to one of the even nodes, etc. Thus computing the shortest even/odd paths will be just a regular BFS on the modified graph.




There is another trick to simplify our code: We just look for the shortest path of even length from $a$ to $b$. Since the length is divisible by 2, we know that we can split it up into two parts of the same length. This makes us only needing one DFS pass.

```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")
```

# Meteo

For a 3D-room of cubes of $1 \times 1 \times 1$ meters, answer a series of the following request types:

- Query: `M` $x_1, y_1, z_1$ $x_2, y_2, z_2$: Print the average of the values inside the cuboid with corners $(x_1, y_1, z_1)$ and $(x_2, y_2, z_2)$.
- Update: `C` $x, y, z, d$: Set the value of $(x, y, z)$ to $d$

While the full solution is quite complicated, there were some easy subtasks:

**Group 1 (25 points):** The limits were small enough that a brute-force could solve this subtask. Just keep track of all values in an array (`int values[10][10][10]` or `vector<vector<vector<int>>>`). To calculate the average, just sum up all relevant values and divide by the number of elements.

**Group 2 (25 points):** In this subtask there were no updates (requests of type "`C`"). This means that the sum can be calculated using prefix sum.

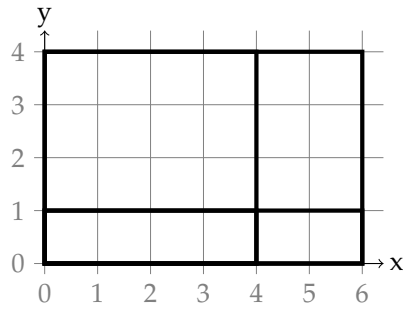
As described on 2H, we can use prefix sum to calculate the sum inside a 2D rectangle: Let $a_{ij}$ be the value at $(i, j)$ and assume we have access to a fast prefix function:

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

Then, the sum between $(x_0, y_0)$ and $(x_1, y_1)$ is

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

You don't need to know this formula, just figure it out yourselves by solving some example: The sum between $(4, 1)$ and $(6, 4)$ looks like $s(6, 4) - s(4, 4) - s(6, 1)$, but we have subtracted the lower left rectangle twice, so we need to add it once and get $s(6, 4) - s(4, 4) - s(6, 1) + s(4, 1)$.




But since are interested in 3D and not 2D, we need to generalize this formula. After a bit of thinking one comes up with this:

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

So how to precompute $s(x, y, z)$ efficiently? We just use the equality $a_{xyz} = \sum_{i=x}^{x} \sum_{j=y}^{y} \sum_{k=z}^{z} a_{ijk}$ to get:

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

or solved for $s(x, y, z)$,

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

```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)))
```

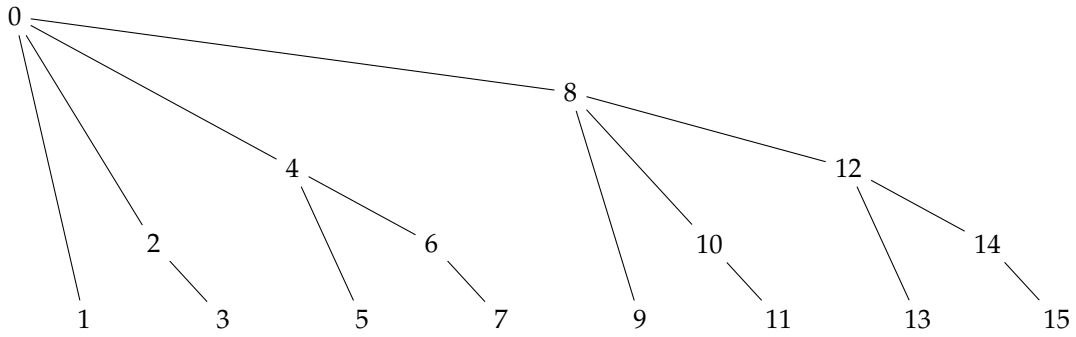
**Group 1 + 2 (50 points):** Clearly both approaches from before can be combined for 50 points in total.

**All groups (100 points):** Let's look at the full solution. The difficult part is to have fast queries *and* updates together (the brute-force solution has fast updates and slow queries, the prefix sum solution fast queries and slow updates).

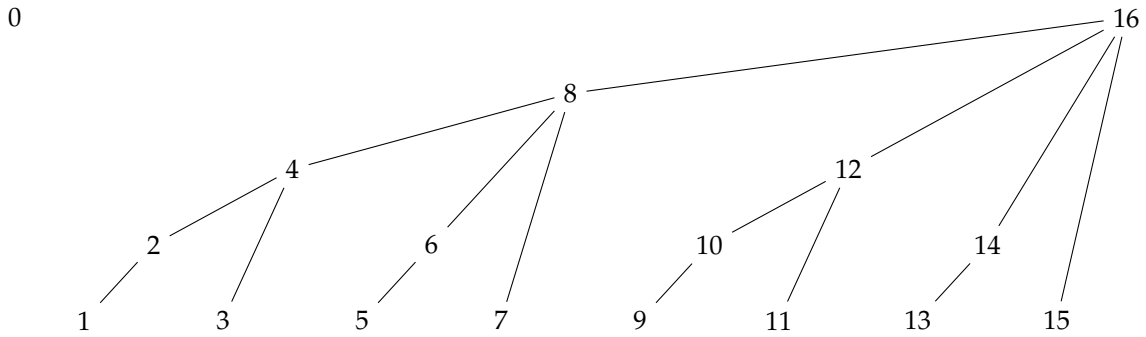
For 1D, a known data structure that solves this and many more problems is a segment trees. A slightly easier data structure that does the same for prefix sums is the fenwick tree (sometimes called BIT for binary indexed tree).

A fenwick tree on $n$ elements is an array of length $n + 1$, where the $i$-th entry stores the sum of all elements from indices $i - 2^k + 1$ to $i$ (including $i$) where $2^k$ is the largest power of two that divides $i$. This means that e.g. the 5-th entry stores only $a_5$, the 6-th entry stores $a_5 + a_6$, the 8-th entry stores $a_1 + a_2 + \ldots + a_8$.

Look at this picture:




The $y$ coordinate represents the range of each vertex (first (lowest) level: divisible by 1, second level: divisible by 2, third level: divisible by 4, etc.). To get the sum of e.g. $a_1 + \ldots + a_{11}$, we have to sum up $a_{11} + (a_9 + a_{10}) + (a_1 + \ldots + a_8)$, which means we add the values of vertices 11, 10 and 8. Those are exactly the parents of $a_{11}$.




This mirrored tree represents the nodes we have to update if we change one value. If we add $x$ to $a_{11}$, we not only have to update vertex 11, but also vertex 12 (which stores $a_{11} + a_{12}$) and vertex 16 (which stores $a_1 + \ldots + a_{16}$.

The reason this tree is so useful is that is is very easy to move from a node to its parent: In the first tree, the parent of $i$ is $i - (i\&(-i))$ (where & is the bitwise and), in the second tree, the parent of $i$ is $i + (i\&(-i))$. (Strictly confirming to the C++ standard, $i - (i\&(-i))$ should be written as $(i\&(i-1))$ and $i + (i\&(-i))$ as $(x|(x-1)) + 1$ because bit operations with negative numbers don't always behave the same – however on machines with two-complement the former bit-tricks usually work and are better to remember because of their symmetry.)

So how to make this fenwick tree work for 2D? We take the tree from before but instead of storing an integer we store another fenwick tree. The $j$-th element of the $i$-th fenwick tree represents $s(i, j)$. The updates and queries will be like before, except on the first layer we add the value to the $j$-th element of the lower segment tree and we query the $j$-th element. For 3D this is pretty much the same.

In case this introduction to Fenwick trees was too fast, there are other tutorials online, including 2D:

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

Below is only the Python code. The C++ implementation is left as an exercise for the reader.

```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)))
```