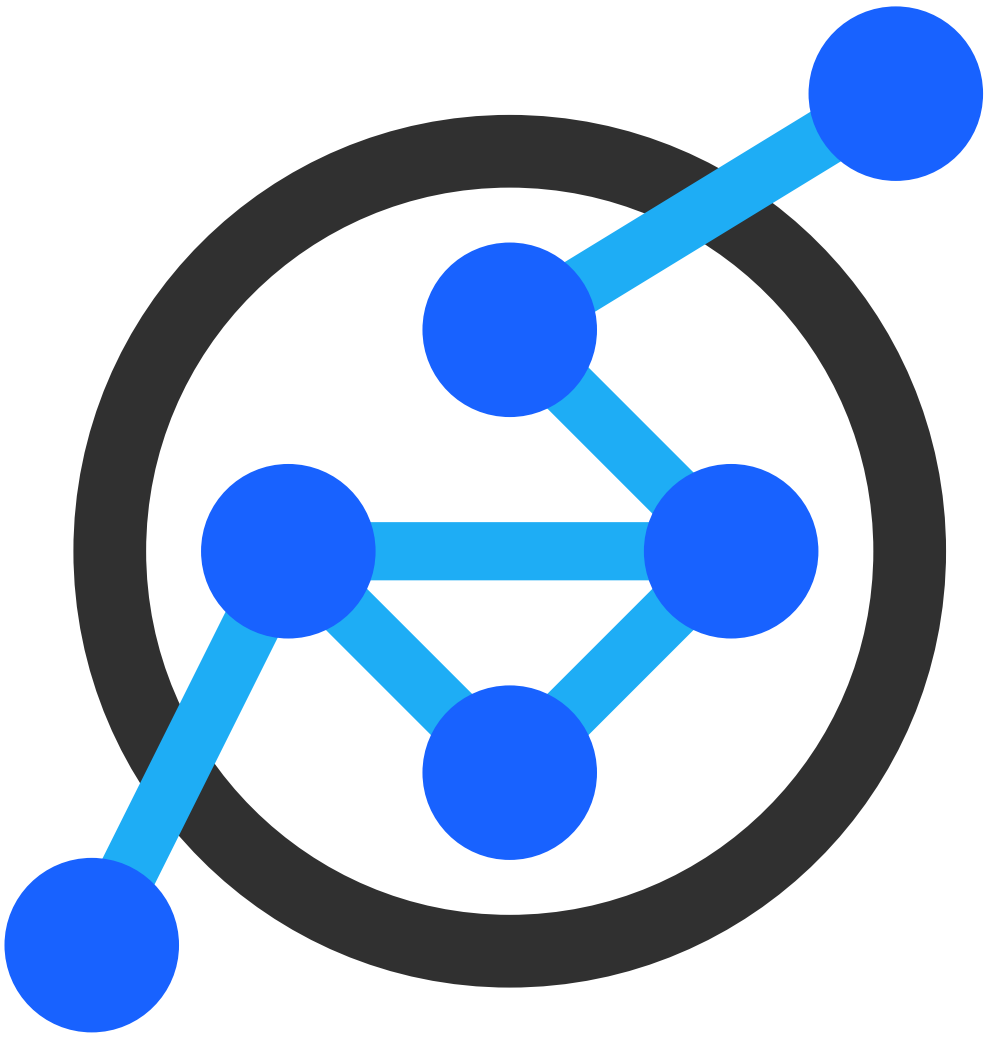
SOI Parmesarnen Contest

Solution Booklet



Swiss Olympiad in Informatics

February 2019

Lecture Seating

Task Idea	Timon Stampfli, Johannes Kapfhammer
Task Preparation	Johannes Kapfhammer
Description English	Timon Stampfli
Description German	Timon Stampfli
Description French	Florian Gatignon
Solution	Timon Stampfli

In Lectureseating you are given the number of rows and columns of a classroom and how many students are given in each row. The students fill a row from one side of the classroom

As the lesson begins, every student moves a row to the front until they are in the front seat or all seats in front of them are occupied. You have to output how many students sit in each row after moving, given the initial seating.

To solve this task, several observations help to find the best solutions.

First, we notice that after the movement, the row with the most students must be the first row. If there was a row with more students than the first row, some students would have empty seats in front of them and would move forward to the front row. The same argument can be made for all other rows: no row contains more students than any row in front of it.

Next, we can ask how many students are going to end up in the first row. To calculate the answer, we can look at every column. If there is at least one mouse in it, some mouse of that column has to end up in the front row. Therefore the number of mice sitting in the front row equals the number of occupied columns. To find the number of occupied columns, we can simply look at the row with the most mice in it. There can't be a column with a mouse in it if there is no mouse in it in the column with the most mice, as mice are filling up the classroom from one side. We then notice that when moving the order of the mice in one column doesn't matter. Therefore, we can simply move all the mice, that are seating in the row with the most mice, that are in a column not occupied in the front row, to the front row. This is the same as swapping the number of mice in the first row and in the row with the most mice

Now after having found the number of mice sitting in the first row, we can ignore it in our coming calculations. The second row becomes our new first row and to find the number of mice ending up in the new first row, we find once again the column with the most mice and swap it. We repeat this until all rows are processed.

The algorithm we executed is selection sort, which repeatedly finds the biggest element and moves it at the end of the already sorted elements. To solve the task any sort algorithm can be used to sort the input and calculate the solution.

The solution to this task is very short:

```cpp
#include <bits/stdc++.h>

using namespace std;

int main() {
  int N, M;
  cin >> N >> M;
  vector<int> v(N);

  for (int i = 0; i < N; ++i) {
    cin >> v[i];
  }

  sort(v.begin(), v.end());
  reverse(v.begin(), v.end());

  for (int i = 0; i < N; ++i) {
    cout << v[i] << " ";
  }
```

```
20   cout << endl;
21 }
```

There are different ways to sort a array descending in C++. I'll show two more way sorting descending using the STL, as familiarity with STL is useful for competitions.

This solution uses reverse iterators. rbegin points to the last element in the vector. rend points to a special element before the first element.

```
1 sort(v.rbegin(), v.rend());
```

You can also pass a function to sort, which is used as comparison function.

```
1 sort(v.begin(), v.end(), greater<>());
```

is a shorthand for

```
1 sort(v.begin(), v.end(), greater<int>());
```

Exquisite Fondue

Task Idea	Timon Stampfli, Johannes Kapfhammer
Task Preparation	Johannes Kapfhammer
Description English	Timon Stampfli
Description German	Timon Stampfli
Description French	Florian Gatignon
Solution	Timon Stampfli

For this task, we have to determine the maximal number of exquisite courses in a series of fondue courses. In every course, one of the n blocks of cheese is added to the pot. To determine whether a course is exquisete it is for every pair of cheeses given whether they harmonize. If a block of cheese is added to the pot that harmonizes with at least one other cheese already in the pot, then the course is exquisite. In the answer not only the number of optimal courses has to be given, but also an order of the cheeses that results in that number of optimal courses.

In the input, we are given for any pair of cheeses, whether they harmonize. This kind of relationship can be represented as a graph, where the nodes are cheeses and two nodes are connected with an edge exactly if they harmonize with each other.

We have to note that the input can contain several components. (A component consists of several nodes connected directly by an edge or indirectly via other nodes that are part of the component. There is no edge between any two nodes of two different components.)

If there are several components, for every component one course has to be non-exquisite, as one of the cheeses of that component has to be the first and it can not harmonize with any other cheese. Once at least one cheese of a component is added, we can continuously add neighbors of that cheese and of its neighbors, that have already been used until no cheese is left in the current component.

Two algorithms that do exactly that are DFS and BFS. For both algorithms, it is trivial to observe that for every node we traverse, except the first one, one of its neighbors has already been traversed. This gives us a way to compute the answer. Simply go through all the components and use DFS or BFS to find an order. The number of exquisite courses is the total number of courses minus the number of components in the graph.

```cpp
#include <bits/stdc++.h>
using namespace std;


int dfs(int v, vector<vector<int>> const& g,
        vector<bool>& vis, vector<int>& order) {
  if (vis[v]) return 0;
  vis[v] = true;
  order.push_back(v);
  int cnt = 1;
  for (auto w : g[v])
    cnt += dfs(w, g, vis, order);
  return cnt;
}

int main() {
  int n; cin >> n;
  vector<vector<int> > g(n);
  for (int i=1; i<n; ++i) {
    for (int j=0; j<i; ++j) {
      int x; cin >> x;
      if (x) {
        g[i].push_back(j);
        g[j].push_back(i);
      }
    }
  }
```

```
28
29    vector<bool> v(n);
30    vector<int> order;
31    int total = 0;
32    for (int i=0; i<n; ++i) {
33      total += max(0, dfs(i, g, v, order) - 1);
34    }
35    cout << total << '\n';
36    for (auto x : order)
37      cout << x << ' ';
38    cout << '\n';
39  }
```

Sledrun

Task Idea	Timon Stampfli, Johannes Kapfhammer
Task Preparation	Johannes Kapfhammer
Description English	Johannes Kapfhammer
Description German	Johannes Kapfhammer
Description French	Florian Gatignon
Solution	Johannes Kapfhammer

This task can be formally described like this. Given an array $h[0..n]$ and an integer k, compute

$$\min_s \sum_{i=0}^{n-1} |h[i] - (s + i \cdot k)|.$$

Here, the value s denotes the target value of the first block of snow. Once you know that, everything all other target values are also fixed; the i-th value should be $s + i \cdot k$.

A brute-force which tries out all possible s (in range $-10000 \leq s \leq 10000$) already scored 20 points.

The first observation in order to make the formula simpler, is that we can "flatten" h to get rid of the slope. Define

$$v[i] = h[i] + i \cdot k$$

Then the above formula gets

$$\min_s \sum_{i=0}^{n-1} |v[i] - s|$$

since the $i \cdot k$ cancel each other out.

In this new formula, the key observation is that the order of the $v[i]$ doesn't matter anymore. So we might as well sort them by value ($v[0] \leq v[1] \leq \cdots \leq v[n-1]$) which allows us to split up the sum into larger and smaller values:

$$\min_s \sum_{i=0}^{n-1} |v[i] - s| = \min_s \left(\sum_{i=0}^{\max\{i|v[i]\leq s\}} (s - v[i]) + \sum_{1+\max\{i|v[i]\leq s\}}^{n-1} (v[i] - s) \right)$$

If s has a value that lies between $v[i]$ and $v[i+1]$ for some i, every time we increase/decrease s by one, the result changes by $k \cdot s$, where k is the difference in sizes between the two sets ($k = n - 2\max\{i \mid v[i] \leq s\}$). So either values $v[i]$ or $v[i+1]$ must be optimal, never some value in between. So we can change the formula to:

$$\min_{k\in\{0,\ldots,n-1\}} \left(\sum_{i=0}^{k-1} (v[k] - v[i]) + \sum_{i=k}^{n-1} (v[i] - v[k]) \right) = \min_{k\in\{0,\ldots,n-1\}} \left((n - 2 \cdot k) \cdot v[k] - \sum_{i=0}^{k-1} v[i] + \sum_{i=k}^{n-1} v[i] \right)$$

If we do prefix sums, we get a linear solution that can be seen as a scanline that runs in $O(n \log n)$ (because of the sorting):

```
// cost(x) = (la*x - lb) + (ub - ua*x)
//           lower (<=x) | upper (>=x)
int64_t la=0, lb=0;
int64_t ua=n, ub=accumulate(v.begin(), v.end(), 0LL);
sort(v.begin(), v.end());
int64_t best = 1e18;
for (auto x : v) {
  ua -= 1;
  ub -= x;
  int64_t cost = (la*x - lb) + (ub - ua*x);
  if (cost < best)
```

```
    best = cost;
  la += 1;
  lb += x;
}
cout << best << '\n';
```

However, with one additional observation, we can simplify our solution further.

Again, look at the formula

$$\min_{k\in\{0,\dots,n-1\}} \left((n-2\cdot k)\cdot v[k] - \sum_{i=0}^{k-1} v[i] + \sum_{i=k}^{n-1} v[i] \right)$$

and think about what happens when $n-2\cdot k > 0$ and we make k larger.

$$\left((n-2\cdot k)\cdot v[k] - \sum_{i=0}^{k-1} v[i] + \sum_{i=k}^{n-1} v[i] \right) - \left((n-2\cdot(k+1))\cdot v[k+1] - \sum_{i=0}^{k} v[i] + \sum_{i=k+1}^{n-1} v[i] \right)$$
$$= \underbrace{(n-2k)}_{>0}\underbrace{(v[k]-v[k+1])}_{\leq 0} + 2\underbrace{(-v[k+1]+[k])}_{\leq 0} \leq 0$$

This shows us that making k larger makes the solution better.

The very same argument can be made for $n-2\cdot k < 0$, where making it smaller makes the solution better (for a proof just use the following formula but with all values negated).

Now, we *could* do a binary search resulting in an $O(n\log n+\log n)+O(n\log n)$ algorithm, but there's no speed-up compared to the scanline algorithm from above.

But those formulas actually show that if we set $k=n/2$ we have an optimal solution. So we can just set our starting value to the median and have a solution.

Computing the median works either by sorting and taking the $\lceil n/2\rceil$-th element (in $O(n\log n)$), or by calling `nth_element` (which runs in $O(n)$). This gives us a solution in $O(n)$):

```
#include <bits/stdc++.h>
using namespace std;

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

  int n, k; cin >> n >> k;
  vector<int64_t> v;
  copy_n(istream_iterator<int>(cin), n, back_inserter(v));

  for (int i=0; i<n; ++i)
    v[i] -= i*(int64_t)k;

  nth_element(v.begin(), v.begin()+n/2, v.end());

  int64_t ans = 0, median = v[n/2];
  for (auto &x: v)
    ans += abs(x - median);
  cout << ans << "\n";
}
```

Dice Cheating

Task Idea	Timon Stampfli, Johannes Kapfhammer
Task Preparation	Johannes Kapfhammer
Description English	Timon Stampfli
Description German	Timon Stampfli
Description French	Florian Gatignon
Solution	Johannes Kapfhammer

You were given a starting sequence on alphabet $\{0, 1, \ldots, k-1\}$ and have to transform it to a target sequence on the same alphabet. The allowed operations were replacing a given character to another one, inserting a new character (anywhere) and deleting a character (anywhere). Each of those operations were associated a cost depending on the characters involved. The goal was to minimize the total cost.

The first subproblem was to find out the actual cost of replacing a single character. The cost to do that was given as a cost matrix, but it may be better to not replace character a directly with character b (cost of x_{ab}) but instead using a sequence of replacements $a \to c \to d \to b$ (cost of $x_{ac} + x_{cd} + x_{db}$).

If we interpret the matrix as graph, where the edge from i to j has cost x_{ij}, this is exactly a shortest path problem. For every pair of characters (i, j), we are interested in the shortest path from i to j. This can be solved in $O(n^3 \log n)$ by starting Dijkstra from every vertex. There's a way to implement Dijkstra for sparse graphs to get rid of the $\log n$ factor, but in this case we can just use the Floyd-Warshall. Floyd-Warshall computes the shortest path of all pairs of vertices in dense graphs in $O(n^3)$.

The code for Floyd-Warshall is listed here:

```
// n: number of vertices
// d[i][j]: distance from i to j
for (int k = 0; k < n; k++)
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
      if (d[i][k] + d[k][j] < d[i][j])
        d[i][j] = d[i][k] + d[k][j];
```

Basically, this is a DP (dynamic programming) that computes the shortest distance from i to j using only edges that go through intermediate vertices at most k. Initially, this is just the edge (because no intermediate vertices are allowed. If we know the shortest path using vertices $< k$, we know that either that's the best we can do, or we can go through vertex k with the path $i \to k \to j$, for which we know the cost `d[i][k] + d[k][j]`.

Doing this was enough to solve the second subtask with $n = m = 1$, because now the solution can be read from `d[i][j]`.

A similar trick has to be done for the insert and delete operations. We can either insert i it directly (initial value) or we can insert first j and then replace j with i. If we take the minimum for all possible j we find the solution:

```
// ins[i]: cost to insert character i
// sub[j][i]: cost to substitute character j with i
for (int i=0; i<k; ++i)
  for (int j=0; j<k; ++j)
    if (ins[j] + sub[j][i] < ins[i])
      ins[i] = ins[j] + sub[j][i];
```

The same can be done for the delete operation, just that we have to first substitute i with j and then delete j.

Now we have a cost matrix where doing an operation directly is always optimal. This was written in the limits for subtask 4 like this: "it is always optimal to rotate a dice directly instead of rotating it first to another side and then rotate to the target side (it is never cheaper to rotate

from side 'a' to side 'b' and then to side 'c' instead of rotating from side 'a' to side 'c'). It is also always cheaper to remove or add a dice directly instead of rotating first". For subtask 4, you only had to do what follows below.

This substitution process can be modeled as DP. Define `DP[i][j]` as the cost to transform the first i characters of the starting sequence to the first j characters of the target sequence.

Any optimal solution consists of an optimal solution of some smaller subproblem, and then a last operation applied. If the solution for the smaller subproblem would not be optimal, we could find a better solution by replacing it with an optimal solution. As such, this problem is amendable by DP.

$$\texttt{DP}[i+1][j+1] = \text{cost to transform s[0..i+1] to t[0..j+1]}$$

$$= \min \begin{cases} \texttt{DP}[i][j+1] + \texttt{DEL}[s[i]] & \text{(delete s[i+1] and transform s[0..i] to t[0..j+1])} \\ \texttt{DP}[i+1][j] + \texttt{INS}[t[j]] & \text{(transform s[0..i+1] to t[0..j] and insert t[j])} \\ \texttt{DP}[i][j] + \texttt{SUB}[s[i]][t[j]] & \text{(transform s[0..i] to t[0..j] and substitute s[i] to s[j])} \end{cases}$$

$$\texttt{DP}[0][0] = 0$$

All combined, the full solution looks like this. We have $O(n^3)$ running time for Floyd-Warshall, and $O(n^2)$ for the substitutions.

```cpp
#include <bits/stdc++.h>
using namespace std;

// helper function: like "a = min(a, b)" but without having to write a twice
template <typename T, typename S>
void xmin(T& a, S&& b){
  if (b < a)
    a = forward<S>(b);
}

void floyd_warshall(int n, vector<vector<int64_t>>& d) {
  for (int k = 0; k < n; k++)
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
        xmin(d[i][j], d[i][k] + d[k][j]);
}

int main() {
  ios::sync_with_stdio(false);

  int n, m, k; cin >> n >> m >> k;
  vector<int64_t> s; copy_n(istream_iterator<int>(cin), n, back_inserter(s));
  vector<int64_t> t; copy_n(istream_iterator<int>(cin), m, back_inserter(t));
  vector<int64_t> ins; copy_n(istream_iterator<int>(cin), k, back_inserter(ins));
  vector<int64_t> del; copy_n(istream_iterator<int>(cin), k, back_inserter(del));
  vector<vector<int64_t>> sub(k);
  for (auto& v : sub)
    copy_n(istream_iterator<int>(cin), k, back_inserter(v));

  floyd_warshall(k, sub);
  for (int i=0; i<k; ++i)
    for (int j=0; j<k; ++j) {
      xmin(ins[i], ins[j] + sub[j][i]);
      xmin(del[i], sub[i][j] + del[j]);
    }

  const int64_t INF = 1e18;

  vector<vector<int64_t> > dp(n+1, vector<int64_t>(m+1, INF));
  dp[0][0] = 0;
  for (int j = 0; j < m; ++j)
    dp[0][j+1] = dp[0][j] + ins[t[j]];
  for (int i = 0; i < n; ++i)
```

```
44      dp[i+1][0] = dp[i][0] + del[s[i]];
45    for (int i = 0; i < n; ++i)
46      for (int j=0; j < m; ++j)
47        dp[i+1][j+1] = min({dp[i][j+1] + del[s[i]],
48                            dp[i+1][j] + ins[t[j]],
49                            dp[i][j] + sub[s[i]][t[j]]});
50    cout << dp[n][m] << '\n';
51  }
```