ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Algorithms Lab HS21
Department of Computer Science
Prof. Dr. A. Steger, Prof. Dr. E. Welzl
cadmo.ethz.ch/education/lectures/HS21/algolab

Solution — Buddy Selection

1 The problem in a nutshell

Given a list of $c \leqslant 100$ interests for each of $n \leqslant 400$ buddies, decide if all buddies can be paired so that each pair of buddies share strictly more than f interests. Every interest is a string of length at most $\ell = 20$.

2 Modeling and subproblems

We model the buddies as vertices of an unweighted graph G with an edge between a pair of buddies if they share strictly more than f interests. In this graph, we need to find a maximum matching and check if this matching is perfect, i.e., if it includes all vertices/buddies.

3 Algorithm Design

To construct the graph G, we need to determine the number of common interests for every pair of buddies. We present two approaches to constructing the graph G. Then we show how to find a maximum matching in G and check its size.

Naïve: $\mathcal{O}(n^2c^2\ell)$. For every pair u, v of buddies, we count how many of u's interests are found among v's interests by making at most $\mathcal{O}(c^2)$ string comparisons. Here, every string comparison takes $\mathcal{O}(\ell)$ time. There are $\Theta(n^2)$ pairs of buddies, the overall running time to construct the graph is thus $\mathcal{O}(n^2c^2\ell)$.

Sorting: $\mathcal{O}(n\ell c \log c + n^2c\ell)$. Instead of comparing each interest of u with each interest of v, we can first sort the lists of u's and v's interests and then apply the following two-pointers technique: one pointer goes through the list of u's interests, the other goes through the list of v's interests. At every step, we compare the strings pointed by the two pointers and increase the pointer that points to the smaller string (or increase both pointers if the two strings are equal). This is very similar to how two sorted lists are merged in the merge sort algorithm.

Sorting the lists of interests for every buddy takes $\mathcal{O}(n\ell c \log c)$ time, because comparing two interests takes $\mathcal{O}(\ell)$ time. Then for each pair of buddies, we perform $\mathcal{O}(c)$ string comparions (because at least one pointer is advanced at every step and each pointer can advance $\mathcal{O}(c)$ times). Because a string comparison takes $\mathcal{O}(\ell)$ time, the running time of this phase is in $\mathcal{O}(n^2c\ell)$. Hence, the overall running time is in $\mathcal{O}(n\ell c \log c + n^2c\ell)$. An implementation of this solution is presented below.

Matching: $\mathcal{O}(mn \cdot \alpha(m,n))$. A maximum matching in a general[1] unweighted graph G can be found using Edmond's algorithm implemented in BGL:

```
std::vector<vertex_desc> mate(n);
boost::edmonds_maximum_cardinality_matching(G, boost::make_iterator_property_map(
  mate.begin(), boost::get(boost::vertex_index, G)));
```

The time complexity of this algorithm is in $\mathcal{O}(mn \cdot \alpha(m,n))$, where n is the number of vertices, m is the number of edges, and $\alpha(m,n)$ is the Inverse Ackermann function which grows very slowly and is at most 4 for any feasible input.

Afterwards, the size of the maximum matching can be found as follows:

```
int matching_size = boost::matching_size(G, boost::make_iterator_property_map(
  mate.begin(), boost::get(boost::vertex_index, G)));
```

4 Remark

It might be tempting to use some kind of greedy approach to compute the maximum matching. However, greedy approaches only lead to a maximal matching, i.e., one that cannot be further extended, but does not necessarily contain the maximum possible number of edges in the matching.

5 Full Solution with Running Time in $\mathcal{O}(n\ell c \log c + n^2 c\ell + mn \cdot \alpha(m,n))$

```
#include <iostream>
#include <string>
#include <vector>

#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/max_cardinality_matching.hpp>

typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::undirectedS> graph;
typedef boost::graph_traits<graph>::vertex_descriptor vertex_desc;
typedef std::vector<std::string> interests;

// A linear traversal over the two sorted lists of interests.
// Returns the number of common interests.
int common_interests(const interests &s1, const interests &s2) {
  int i1 = 0, i2 = 0;
  int common = 0;

  // Stop if reaching the end of one of the lists.
  while (i1 < s1.size() && i2 < s2.size()) {
    // Advance the pointer to the 'smaller' interest.
    // In case of equality advance both pointers.
    if (s1[i1] == s2[i2]) {
        ++common;
        ++i1;
        ++i2;
    } else if (s1[i1] < s2[i2]) {
      ++i1;
    } else {
      ++i2;
```

[1] in particular, not necessarily bipartite

```
    }
  }

  return common;
}

void testcases() {
  int n, c, f;
  std::cin >> n >> c >> f;

  // Construct a list of sorted interests for every buddy.
  std::vector<interests> students(n);
  for (int i = 0; i < n; ++i) {
    students[i].resize(c);
    for (int j = 0; j < c; ++j) std::cin >> students[i][j];
    std::sort(students[i].begin(), students[i].end());
  }

  // Construct a graph with edges between buddies sharing more than ‘f‘ interests.
  graph G(n);
  for (int i = 0; i < n; ++i) {
    for (int j = i + 1; j < n; ++j) {
      if (common_interests(students[i], students[j]) > f) {
        boost::add_edge(i, j, G);
      }
    }
  }

  // Compute maximum matching.
  std::vector<vertex_desc> mate(n);
  boost::edmonds_maximum_cardinality_matching(G, boost::make_iterator_property_map(
    mate.begin(), boost::get(boost::vertex_index, G)));

  // Compute the size of the maximum matching.
  int matching_size = boost::matching_size(G, boost::make_iterator_property_map(
    mate.begin(), boost::get(boost::vertex_index, G)));

  std::cout << (2 * matching_size == n ? "not optimal" : "optimal") << "\n";
}

int main() {
  std::ios_base::sync_with_stdio(false);
  int T;

  std::cin >> T;
  while (T--) testcases();

  return 0;
}
```