Solution — Buddies

1 The problem in a nutshell

Given a list of $c \leq 100$ interests for each of $n \leq 400$ buddies, decide if all buddies can be paired so that each pair of buddies shares more than f interests. Every interest is a string with length $\ell \leq 20$.

2 Modeling and subproblems

The problem can be split into three independent consecutive subproblems:

1. Consider a complete graph with the buddies as vertices and the number of shared interests as weights on the edges.
2. Construct the unweighted graph G with edges only between vertices/buddies that share strictly more than f interests.
3. Find a maximum matching in G and check if this matching is perfect, that is, includes all vertices/buddies.

3 Algorithm Design

Subproblem 3: Matching. Let us start from the last subproblem: finding a maximum matching for the general unweighted graph G (*general* means that no additional properties are known, like being bipartite). The algorithm of Edmonds solves the maximum matching problem and has an implementation in BGL:

```
void boost::edmonds_maximum_cardinality_matching(const Graph& G, MateMap mate);
```

The time complexity of this implementation is $\mathcal{O}(mn\alpha(m,n))$ where n is the number of vertices, m is the number of edges in the graph, and $\alpha(\cdot,\cdot)$ is the Inverse Ackermann function which grows super slowly and is at most 4 for any feasible inputs. Note that m can be up to $\binom{n}{2} \in \Theta(n^2)$ here, as there may be an edge between every pair of vertices.

Every edge (u,v) of the maximum matching is included in the resulting mapping as `mate[v] := u` and `mate[u] := v`. In case a vertex u is not matched, `mate[u] == graph_traits::null_vertex()`. What remains to be checked is if all the vertices are matched.

Subproblems 1 and 2: Graph Construction. The goal is to construct a graph in which a perfect matching exists if and only if there is a perfect matching of the buddies such that every pair shares strictly more than f interests. Obviously, the smallest and sufficient number of shared interests is $f+1$. Thus we will construct the graph G with unweighted edges between every pair

of vertices/buddies that share at least $f+1$ common interests. We will consider several different approaches to determine the number of interests for each of the $\mathcal{O}(n^2)$ pairs of vertices.

Naïve: $\mathcal{O}(n^2c^2\ell)$. Each pair among the n buddies is compared as follows. To determine if the vertices u and v share at least $f+1$ interests we count how many of the c interests of u are found in the interests of v by making $\mathcal{O}(c^2)$ string comparisons. Every string comparison takes $\mathcal{O}(\ell)$ time.

Hashing: $\mathcal{O}(nc\ell+n^2c^2)$. The naïve solution can be sped up by intially hashing all the interests. This leads to $\mathcal{O}(c^2)$ interest comparisons each of which takes $\mathcal{O}(1)$ time. In practice, the preprocessing complexity term $\mathcal{O}(nc\ell)$ is dominated by the comparison term $\mathcal{O}(n^2c^2)$ for the specific limitations for n, ℓ and c. This solution shoule gather full points for the task.

Ordered set intersection: $\mathcal{O}(n\ell c\log c+n^2c\ell)$. Another optimization over the naïve solution is to put all interests per buddy into a `std::map` for $\mathcal{O}(n\ell c\log c)$ time. This way the intersection of every two sets is sped up to perform $\mathcal{O}(c)$ string comparisons of $\mathcal{O}(\ell)$ time each. Internally the intersection works by walking with two pointers similar to the sorting solution described below. Note that iterating over a `std::map` takes only time linear in the number of nodes in the binary search tree despite that the height of the tree is logarithmic (this is because the number of edges in the tree are limited to $n-1$ and each of them is traversed twice: once in each direction). In practice, the preprocessing complexity term $\mathcal{O}(n\ell c\log c)$ is dominated by the comparison term $\mathcal{O}(n^2c\ell)$. This solution is slower than the one to be described next (empirically by 30% on our tests) with just sorting despite the equal complexity. The reasons are the complex structures used for implementing sets in STL (binary search trees) and the temporary objects (sets) that are created every time when intersection is computed.

Sorting: $\mathcal{O}(n\ell c\log c+n^2c\ell)$. The same complexity as the set intersection solution can be achieved by sorting the list of interests for every buddy. This way we can intersect every pair of sorted lists in $\mathcal{O}(c)$ time using the following two-pointers technique: One pointer goes through the first list, another through the second list. At every step, we increase the pointer that points to the smaller element. This is very similar to how two sorted lists get merged in the MergeSort algorithm. In practice, the preprocessing complexity term $\mathcal{O}(n\ell c\log c)$ is dominated by the comparison term $\mathcal{O}(n^2c\ell)$. This solution can gather full points for the task. A full implementation of this solution is presented below.

Sorting of hashes: $\mathcal{O}(nc\ell+n^2c\log c)$. Combining the hashing optimization with the two-pointers trick on sorted lists, we can achieve an even faster solution. In practice, the preprocessing complexity term $\mathcal{O}(nc\ell)$ is dominated by the comparison term $\mathcal{O}(n^2c\log c)$.

4 Additional ideas

There may be a desire to avoid the matching subproblem by using some kind of a greedy approach. Unfortunately, without additional assumption about the graph or the solution, greedy approaches lead to finding maxim*al* machings only: they cannot be greedily extended anymore

but do not provide guarantees for being maximum. The easier way to figure out that a specific greedy does not work is to provide a counter-example.

Another line of optimization may be trying to avoid some of the $\Theta(n^2)$ buddy comparisons. Locality-sensitive hashing is a hashing technique such that similar (not only identical) objects get similar hash values and thus can be grouped together. However, it is not expected to provide a speed-up here, especially not in the case of a dense graph, and has the drawback of being probabilistic.

5 A Complete Solution with sorting and general matching in $\mathcal{O}(n\ell c \log c + n^2 c\ell + nm\alpha(m,n))$

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

#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 VertexDescriptor;
const VertexDescriptor NULL_VERTEX = boost::graph_traits<Graph>::null_vertex();
typedef std::vector<std::string> Interests;

// A linear traversal over the two sorted lists of interests.
// Returns true if there are at least 'k' common interests.
bool haveKMatches(const Interests& s1, const Interests& s2, int k) {
  int i1 = 0, i2 = 0;

  // Stop if reaching the end of one of the lists
  // or there are already enough common interests.
  while(i1 < s1.size() && i2 < s2.size() && k > 0) {
    if(s1[i1] == s2[i2])
      --k;

    // Move the pointer to the 'smaller' interest.
    // In case of equality both pointers are moved on after another.
    if(s1[i1] <= s2[i2])
      ++i1;
    else
      ++i2;
  }

  return k <= 0;
}

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 similar enough buddies.
  Graph G(n);
  for(int i = 0; i < n; ++i)
    for(int j = i + 1; j < n; ++j)
      if(haveKMatches(students[i], students[j], f + 1))
        add_edge(i, j, G);

  // Do maximum matching.
  std::vector<VertexDescriptor> mate(n);
  boost::edmonds_maximum_cardinality_matching(G, &mate[0]);

  // Check if all buddies are included in the matching.
  bool success = true;
  for(int i = 0; i < n; ++i)
    if(mate[i] == NULL_VERTEX) {
      success = false;
      break;
    }

  std::cout << (success ? "not␣optimal" : "optimal") << '\n';
}

int main() {
  std::ios_base::sync_with_stdio(false);
  int T;
  std::cin >> T;
  for (; T > 0; --T)
    testcases();
  return 0;
}
```