First Round SOI 2015

Solution Booklet

Swiss Olympiad in Informatics

January 24, 2015

1 Cheese Congress

This task is about distributing experts of two different categories from different regions into rooms with two beds each. From each region, there participate the same number of experts per category. Each room should become fully occupied, but there should be at most one expert from each category and region in one room.

Subtask 1: Two Regions (10 points)

For this subtask, there are only two regions. The task is to determine whether distributing the experts according to the rules is possible. Since all experts of the first category from the first region need to be paired up with experts of the second category from the second region and vice-versa, distributing them is possible if and only if there participate the same number of experts of each category.

```
#include <iostream>
using namespace std;

int main(){
        int T;
        cin>>T;
        for(int t=1;t<=T;t++){
                int n1,n2;
                cin>>n1>>n2;
                cout<<"Case #"<<t<<"%d: "<<
                        (n1==n2?"POSSIBLE\n":"IMPOSSIBLE\n");
        }
}
```

Subtask 2: More Regions (20 points)

For this subtask, the restriction on the number of regions is removed. If more than half of all experts are from the same region, there will be more experts from one region than there are different rooms. Therefore, in this case the rooms cannot be distributed according to the rules. In fact, this is also a sufficient condition, as there is an explicit construction for distributing experts into rooms if at most half of all experts are from the same region. The explicit construction will be outlined below.

```
#include <iostream>
using namespace std;

int main(){
        int T;
        cin>>T;
        for(int t=1;t<=T;t++){
                int m,n,mn=0;
                long long s=0;
                cin>>m;
                for(int i=0;i<m;i++){
```

```
12                      cin>>n;
13                      if(n>mn) mn=n;
14                      s+=n;
15              }
16              cout<<"Case #"<<t<<": "<<
17                      (2*mn>s?"IMPOSSIBLE\n":"POSSIBLE\n");
18      }
19 }
```

Subtask 3: Explicit Scheme (40 points)

For this subtask, we should also explicitly construct a distribution of experts into rooms, if this is possible. As explained above, it suffices to consider the case where at most half of all experts are from the same region. In this case, we can first determine the number `mn` of experts per category of the region with most experts. By assumption, this number is at most half the total number of experts per category. We can then arrange all experts from the same region in pairs, one expert of each category in each pair. Those pairs can then be arranged in a circle, such that all experts from one category occupy a contiguous section of the circle. Now the experts of the second category can move `mn` positions further in the circle (all in the same direction). We now claim that the pairs thus formed constitute a valid distribution of experts into rooms: All experts of the second category move at least as many positions further than there are experts of the first category, hence even the first expert of a region moves sufficiently far to pass all positions occupied by experts of the first category from his own region. Furthermore, no expert moves farther than half the total number of positions around the circle, therefore no expert moves far enough to make it back to the section occupied by experts of category one from his own region. Therefore, every expert of the second category finds a partner of the first category who is not from his own region by this process.

```
1  #include <iostream>
2  using namespace std;
3
4  int n[200];
5  int a[40000];
6  int main(){
7          int T;
8          cin>>T;
9          for(int t=1;t<=T;t++){
10                 int m,mn=0,s=0;
11                 cin>>m;
12                 for(int i=0;i<m;i++){
13                         cin>>n[i];
14                         if(n[i]>mn) mn=n[i];
15                         s+=n[i];
16                 }
17                 cout<<"Case #"<<t<<":";
18                 if(2*mn>s){ cout<<" IMPOSSIBLE\n"; continue; }
19                 cout<<"\n";
20                 int k=0;
21                 for(int i=0;i<m;i++)
22                         for(int j=0;j<n[i];j++,k++)
23                                 a[k]=i+1;
24                 for(int i=0;i<s;i++)
25                         cout<<a[i]<<" "<<a[(i+mn)%s]<<"\n";
```

```
        }
}
```

Subtask 4: More Categories (20 points)

For this subtask, there are no longer only two categories of experts: there are now K categories. The rooms are able to host K experts accordingly. All other rules remain the same. We can generalize our previous observations in a similarly straightforward way: If more than one expert in K is from the same region, there will be more experts from one category than there are different rooms, so in this case distributing is impossible. Otherwise, we can analogously arrange the experts from each region in groups of K, such that in each group there is one expert of each category. After arranging those groups around a circle. The experts of category `j` (counted from one) should move `(j-1)*mn` positions further down the circle. This way, every expert moves far enough to avoid being paired up with experts from a smaller category. `(K-1)*mn` will be an upper bound on the total distance travelled by any expert, hence no expert moves far enough to meet other experts from his own region again.

```
#include <iostream>
using namespace std;

int n[200];
int a[40000];
int main(){
        int T;
        cin>>T;
        for(int tt=1;tt<=T;tt++){
                int m,k,mn=0,s=0;
                cin>>m>>k;
                for(int i=0;i<m;i++){
                        cin>>n[i];
                        if(n[i]>mn) mn=n[i];
                        s+=n[i];
                }
                cout<<"Case #"<<tt<<":";
                if(k*mn>s){ cout<<" IMPOSSIBLE\n"; continue; }
                cout<<"\n";
                int l=0;
                for(int i=0;i<m;i++)
                        for(int j=0;j<n[i];j++,l++)
                                a[l]=i+1;
                for(int i=0;i<s;i++){
                        for(int j=0;j<k;j++)
                                cout<<(j?" ":"")<<a[(i+j*mn)%s];
                        cout<<"\n";
                }
        }
}
```

2 Wordchain

This task is about words that occur inside other words. For example "bach" occurs in "**b**el**a**us**ch**en" but not in "hab".

Subtask 1: Two Words (20 points)

In this subtask you had to check for two given words if one occurred in the other one or not.

A good start for a solution is to write a function that checks if word `a` occurs in word `b`. Then you can either check which word is longer and then call the function with word `word1, word2` or `word2, word1` respectively. Or you can simply call the function once with `word1, word2` and once with `word2, word1` and check if one call returned `true`.

One solution for this function is: It iterates over the longer word (`b`) and it has an index to the shorter word (`a_it`) that points to the first character at the beginning. Whenever the current character of the longer word is equal to the character of the shorter one, it increases the index by one. So it searches for the next character in the shorter word. When the index of the shorter word has reached the end, word `a` occurs in word `b`. If the iteration finished without that being the case, it does not.

Since the algorithm iterates once through every word, the runtime is $O(n)$ (n is the length of the words)

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

using namespace std;

//returns true if a occurs in b.
bool a_in_b_enth(string a, string b)
{
    for(int a_it = 0, b_it = 0; b_it < b.length(); b_it++)
    {
        if(a[a_it] == b[b_it])
        {
            a_it++;
            if(a_it == a.length())
                return true;
        }
    }
    return false;
}

int main()
{
    int t;
    cin >> t;
    for(int i = 0; i < t; i++)
    {
        string word1, str2;
        cin >> word1 >> word2;
```

```
29
30          if(a_in_b_enth(word1, word2) || a_in_b_enth(word2, word1))
31              cout << "Case #" << i+1 << ": YES" << endl;
32          else
33              cout << "Case #" << i+1 << ": NO" << endl;
34      }
35  }
```

Subtask 2: Short Poems (30 points)

In this subtask you had to check if there was any word in a list of n words that occurs in another one.

One possible solution is to iterate through all words and for every word iterate through all words with a bigger index. So you ensure that you compare every word once with every other word. As soon as a match has been found, you can stop comparing and print "YES".

The comparison is the same as the one from subtask 1.

This algorithm iterates nestedly twice through all words: It has a runtime of $O(n^2m)$ where n is the number of words and m is the length of the words.

```
1  #include <vector>
2
3  ...
4
5  int main()
6  {
7      int t;
8      cin >> t;
9      for(int i = 0; i < t; i++)
10     {
11         int n;
12         cin >> n;
13         vector<string> words(n);
14         for(int j = 0; j < n; j++)
15             cin >> words[j];
16
17         bool possible = false;
18         for(int j = 0; j < n && !possible; j++)
19             for(int k = j+1; k < n; k++)
20             {
21                 if(a_in_b_enth(words[k], words[j]) || a_in_b_enth(words[j], words[k]))
22                 {
23                     cout << "Case #" << i+1 << ": YES" << endl;
24                     possible = true;
25                     break;
26                 }
27             }
28
29         if(!possible)
30             cout << "Case #" << i+1 << ": NO" << endl;
31     }
32 }
```

Subtask 3: Long Poems (50 points)

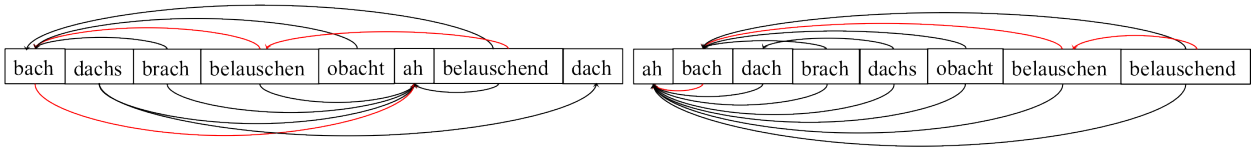
In this subtask you had to find the longest poem out of a given list of words. A poem is a list of words where the first word is contained in the second one and this one in the third word and so on.

The words can be represented as a directed graph, where every word is a node that points to every word that occurs in it. You then have to find the longest path ("chain of arrows") in this graph.

You can either build this graph and perform a DFS from every node. To make this algorithm fast, you have to save the longest chain for every node. When you then arrive at a node for which the value is already known, you don't have to repeat the DFS for this node.

Another algorithm (that we implemented) works only with the list of words: You first have to do a topological sorting, which means, that all arrows have to point in the same direction:

Unsorted (left) and topologically sorted (right). The longest chain is marked in red:



The topological sorting is not unique, there can be several valid orders.

You may have noticed that an arrow can only point from a shorter word to a longer one, so a topological sorting can simply be achieved by sorting the list according to the length of the words. This is exactly what we do in our implementation.

The next step is to make an additional list (`chainLength`) in which we write for every word how long the longest chain for this word is (for "brach" it would be 3: "brach" - "bach" - "ah"). The list is initialized with 1 since the minimal chain length is 1.

We now calculate the longest chain for every word, starting from the smallest index. For every word we do the following: Of all words with smaller index that occur in this word, we take the one with the longest chain. To this value we add one and store it in to its entry in the list.

The last thing to do is to find the biggest entry in this list which is the value we wanted to determine.

This algorithm has a runtime of $O(n^2m)$ since it has two nested iterations and sorting is only $O(n \log(n))$.

```
#include <algorithm>

...

bool cmpStrLen(string str1, string str2)
{
    return str1.length() < str2.length();
}

int main()
{
    ...
    for(int i = 0; i < t; i++)
```

```
14      {
15          ...
16
17          sort(words.begin(), words.end(), cmpStrLen);
18
19          vector<int> chainLength(n, 1);
20
21          int longestChain = 1;
22          for(int j = 0; j < n; j++)
23          {
24              for(int k = 0; k < j; k++)
25                  if(a_in_b_enth(words[k], words[j]))
26                     chainLength[j]  = max(chainLength[j], chainLength[k]+1);
27              longestChain = max(longestChain, chainLength[j]);
28          }
29
30          cout << "Case #" << i+1 << ": " << longestChain << endl;
31      }
32  }
```

3 Audio

This task is about shortening audio files that consist of waves with integer periods.

Subtask 1: Single Interference (20 points)

With all periods being played at the same time, the shortest common wavelength is just the least common multiple of all periods.

The least common multiple $\text{lcm}(a_1, a_2, \ldots, a_n)$ is defined as the smallest integer that is divisible by all its arguments $a_1, a_2, \ldots, a_n$. If both a_1 and a_2 divide $\text{lcm}(a_1, a_2, \ldots, a_n)$, then $\text{lcm}(a_1, a_2)$ must divide it as well and we can conclude that $\text{lcm}(a_1, a_2, \ldots, a_n) = \text{lcm}(\text{lcm}(a_1, a_2), a_3, \ldots, a_n) = \ldots = \text{lcm}(\text{lcm}(\ldots(\text{lcm}(a_1, a_2), a_3)\ldots), a_n)$.

For two numbers, the least common multiple is the product divided by the greatest common divisor (gcd): $\text{lcm}(a, b) = \frac{a \cdot b}{\text{gcd}(a,b)}$. To calculate the gcd, one can use Euclid's algorithm.

One has to be careful by calculating the lcm with 64-bit integers since $(a \cdot b)/\text{gcd}(a, b)$ might overflow (consider $a = b = 2^{63}$). This can be avoided by dividing first: $(a/\text{gcd}(a, b)) \cdot b = a \cdot (b/\text{gcd}(a, b)$, both work just fine.

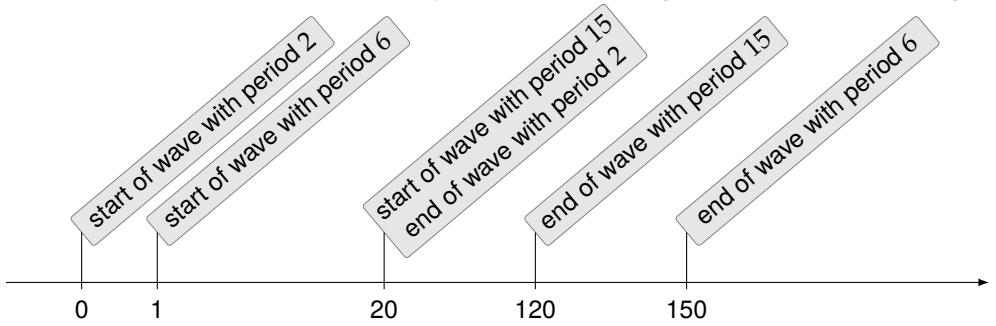
```cpp
#include <iostream>
#include <cstdint>
int64_t gcd(int64_t a, int64_t b) {
    return b==0 ? a : gcd(b, a%b);
}
int64_t lcm(int64_t a, int64_t b) {
    return a/gcd(a,b)*b;
}
void solve_testcase(int t) {
    int n;
    std::cin >> n;
    int64_t resulting_period=1;
    while (n--) {
        int64_t period;
        std::cin >> period;
        resulting_period = lcm(resulting_period, period);
    }
    std::cout << "Case #" << t << ": " << resulting_period << '\n';
}
int main() {
    int t;
    std::cin >> t;
    for (int i=0; i<t; ++i)
        solve_testcase(i+1);
}
```

This code uses "`int64_t`" from <cstdint> instead of "`unsigned long long int`" to save some typing.

Time complexity for the gcd, and thus each testcase, is $O(\log n)$.

Subtask 2: Multiple Interferences (40 points)

In this subtask, the waves start and end at different times. One way to look at it is to imagine a time line with some events. An event may either be starting a new wave or ending an old wave.



The solution is then to sweep through this line and only stop at the events. During that, keep track of the active waves and update in each event accordingly: when encountering an insert event, the period is added to this list, when encountering a delete event, the period is removed.

The general technique is called “sweep line” and the actions “events”. However, the subtask could be solved without the knowledge of it.

```
...
#include <tuple>
#include <algorithm>
#include <unordered_set>
#include <vector>
...
struct event {
    unsigned time;
    bool start;
    uint64_t period;
};
bool operator<(const event& a, const event& b) {
    return std::tie(a.time, a.start) < std::tie(b.time, b.start);
}
void solve_testcase(int t) {
    int n; std::cin >> n;
    std::vector<event> events;
    while (n--) {
        uint64_t period;
        int start, end;
        std::cin >> period >> start >> end;
        events.push_back({start, true, period});
        events.push_back({end, false, period});
    }
    std::sort(events.begin(), events.end());

    std::unordered_multiset<uint64_t> active_periods;
    std::vector<std::pair<uint64_t, int> > sections;
    unsigned last_start = events.front().time;
    for (auto& event : events) {
        if (last_start != event.time && !active_periods.empty()) {
            uint64_t period = 1;
            for (auto& p : active_periods)
```

```
34                 period = lcm(period, p);
35             unsigned time = event.time - last_start;
36             sections.push_back({period, time <= period ? time : period + time%period});
37         }
38         last_start = event.time;
39         if (event.start)
40             active_periods.insert(event.period);
41         else
42             active_periods.erase(active_periods.find(event.period));
43     }
44     std::cout << "Case #" << t << ":\n"
45               << sections.size() << ' ' << events.front().time << '\n';
46     for (auto& s : sections)
47         std::cout << s.first << ' ' << s.second << '\n';
48 }
49 ...
```

This code uses an `std::unordered_multiset` to store the active periods, but there's no difference here to using a `std::multiset`. The same could be done with a `std::map` or `std::unordered_map`, if one keeps track of the count of each period.

Alternatively, the active waves need not to be stored, in each event, one could loop through all waves and only calculate the lcm with the ones that are currently active.

Time complexity is $O(n^2)$, as there are $O(n)$ active periods in the worst case and for each of the $O(n)$ one iterates through all of them.

Subtask 3: Big Files and Big Periods (40 points)

Looking at the limits, in particular $n = 10^7$, our previous submission with $O(n^2)$ time complexity would be too slow. This is thus not a subtask for coding or using big integers.

The main difficulty is to calculate everything modulo $10^9 + 7$ while still being able to know how to calculate the lcm or gcd, because $\gcd(a, b) \bmod m \neq \gcd(a \bmod m, b \bmod m) \bmod m$ (as an example, take $10^9 + 8$ and 2; their gcd is 2, but $10^9 + 8 \bmod 10^9 + 7 = 1$ and $\gcd(1, 2) = 1$).

Instead, don't calculate the gcd at all – use prime factorization! The active periods of our sweep line can be replaced by active period prime factors (or rather primes and their exponent). Inserting a wave is just adding new prime factors and deleting is just removing some prime factors. The lcm modulo m is then just the product of the primes to their biggest exponent.

As last step, we need a good way of updating the product. Just looping through all primes is still $O(n)$ and we would not have gained anything. However, there are two solutions to this problem:

- Calculating modular inverses, so we can "divide" numbers modulo m in $O(1)$.
- Storing those factors in a segment tree, so that each update only takes $O(\log n)$ operations.

I will only explain the first method in this solution booklet, the second method is left as an exercise for the reader.

Fermat's little theorem tells us that $a^p \equiv a \pmod{p}$, so $a^p \equiv a \pmod{p}$ for p prime, so as long as $a \not\equiv 0$, we can use it to calculate the inverse of a: $1 = a^{p-1} = a \cdot a^{p-2}$, so $a^{-1} = a^{p-2}$. The $p - 2$-th power can be calculated using binary exponentiation.

Another way to find modular multiplicative inverses is to use the extended euclidean algorithm.

The code below is slightly more general than it needed to be, as it was guaranteed that $P_e \leq 2^{20}$, so no period could be a multiple of m, however the example in the task description did contain such a case.

```
...
const uint64_t mod=1e9+7;
uint64_t powmod(uint64_t b, uint64_t e) {
    uint64_t res = 1;
    for (b %= mod; e>0; e/=2, b=(b*b)%mod)
        if (e%2)
            res = (res*b)%mod;
    return res;
}
uint64_t modinv(uint64_t x) {
    return powmod(x, mod-2);
}
void solve_testcase(int t) {
    int n; std::cin >> n;
    std::vector<event> events;
    while (n--) {
        uint64_t period;
        int start, end;
        std::cin >> period >> start >> end;
        events.push_back({start, true, period});
        events.push_back({end, false, period});
    }
    std::sort(events.begin(), events.end());

    int mod_powers = 0;
    std::unordered_map<uint64_t, std::multiset<int> > prime_powers;
    std::vector<uint64_t> sections;
    uint64_t online_lcm = 1;
    int last_start = events.front().time;
    for (auto& event : events) {
        if (last_start != event.time && (!prime_powers.empty() || mod_powers))
            sections.push_back(mod_powers ? 0 : online_lcm);
        last_start = event.time;
        for (uint64_t d=2; event.period > 1; ++d) {
            if (d*d > event.period)
                d = event.period;
            int power = 0;
            for (; event.period%d == 0; event.period /= d)
                ++power;
            if (!power)
                continue;
            if (d == mod) {
                mod_powers += (event.start ? 1 : -1)*power;
            } else if (event.start) {
                int old_max_power = prime_powers[d].empty() ? 0 : *prime_powers[d].rbegin();
                prime_powers[d].insert(power);
                for (int i=old_max_power; i<power; ++i)
                    online_lcm = (online_lcm*d)%mod;
            } else {
                prime_powers[d].erase(prime_powers[d].find(power));
                if (prime_powers[d].empty())
                    prime_powers.erase(d);
```

```
                int new_max_power = !prime_powers.count(d) ? 0 : *prime_powers[d].rbegin();
                uint64_t d_inverse = modinv(d);
                for (int i=new_max_power; i<power; ++i)
                    online_lcm = (online_lcm*d_inverse)%mod;
            }
        }
    }
    std::cout << "Case #" << t << ":\n"
              << sections.size() << ' ' << events.front().time << '\n';
    for (auto& s : sections)
        std::cout << s << '\n';
}
...
```

This was a hard subtask, so partial points were awarded for:

- $O(n^2)$ algorithm with prime factorization (8 points)
- Using a segment tree in $O(n^2)$ (16 points)

Just submitting the same idea as in subtask 2 just with big integers did not score any points, as the time complexity would be $O(n^3)$.

4 Yurt

This task is about sorting a set of yurts into the correct order using as few swaps as possible.

For each position, we are given the position the yurt on it should be brought to. We will make use of the following observations:

- We say that yurt B is the successor of yurt A, if yurt A should be brought to the position that is currently occupied by yurt B.
- We say that A is the predecessor of B if yurt B is the successor of yurt A.
- Note that successors and predecessors are unique, since every position is occupied by exactly one yurt, and every yurt should stand at exactly one position in the end. Hence, by following successors, the yurts are partitioned into independent groups, appropriately called *cycles*. A set of such cycles is called a *cycle partition*.
- We will now investigate the effect of one swap on the cycle partition:
  - If we swap two yurts from different cycles, this has the effect of merging the two cycles.
  - If we swap two yurts from the same cycle, this has the effect of splitting the cycle into two cycles.

  Hence every swap can change the number of cycles in the partition by at most one.

Subtask 1: Practical Part with Few Yurts (20 points)

See next subtask.

Subtask 2: Practical Part with Many Yurts (20 points)

For this subtask, we should write a program that prints a sequence of swaps bringing the yurts into the correct order. One way to ensure this is to increase the number of cycles by one in each step, since the target arrangement has the maximal number of N cycles in its partition. This increase can be implemented by swapping a yurt with its successor. This particular strategy furthermore has the effect that every swap places one more yurt in its final spot. Such yurts will not be moved again later.

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

int main(){
        int T;
        cin>>T;
        for(int t=1;t<=T;t++){
                int N;
                cin>>N;
                vector<int> y(N+1);
```

```
13                  for(int i=1;i<=N;i++) cin>>y[i];
14                  for(int i=1;i<=N;i++){
15                          while(i!=y[i]){
16                                  cout<<i<<" "<<y[i]<<"\n";
17                                  swap(y[i],y[y[i]]);
18                          }
19                  }
20          }
21  }
```

This program runs in time and space $O(N)$.

Subtask 3: Theoretical Part: At Most N-1 Swaps (30 points)

For this subtask, we have to argue that the algorithm we have implemented performs at most $N-1$ swaps. After $N-1$ swaps, $N-1$ yurts have been placed at the right positions. Therefore the remaining yurt also has to be in the right place after at most $N-1$ swaps. Hence the algorithm never uses more than $N-1$ swaps. Alternatively, the bound follows from the general expression for the number of swaps discussed for the next subtask.

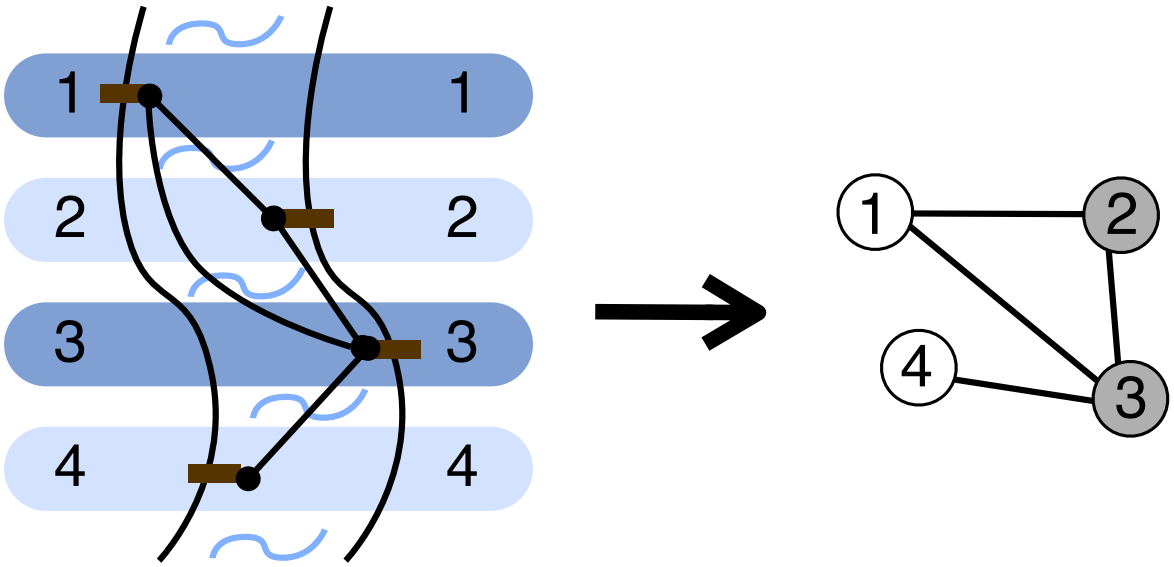
Subtask 4: Optimal Number of Swaps (30 points)

For this subtask, we have to argue that our algorithm always uses the optimal number of swaps, i.e. no other algorithm can use fewer swaps to order the yurts correctly. We can see this by considering the cycle partition: If the initial arrangement of the yurts has $C \geq 1$ cycles, at least $N-C$ swaps are needed to change this to the N cycles in the final arrangement, since every swap can change the number of cycles by at most one. Since our algorithm increases the number of cycles by one with each swap, it uses exactly $N-C$ swaps, matching the lower bound. Therefore the algorithm always uses the optimal number of swaps.

5 River

We are given a list of cities along a river and a list of ferry connections between pairs of these cities. Our task was to assign each city either to the left or to the right side of the river, so that as many of the connections as possible are crossing the river.

This task can nicely be solved using the concepts of graph theory. There we look at networks of nodes and edges. The cities in our task correspond to a set of nodes and the ferry connections correspond to edges between pairs of these nodes. The fact that the cities are ordered along the river is not important. The figure below shows how you can get from the picture of the river to such a network of nodes and edges.



Subtask 1: Only River-Crossing Ferry Connections (total 50 points)

In this first subtask, we want to assign the cities to the two sides of the river so that ALL connections cross the river. Translated to the world of graph theory, this corresponds to the following coloring problem: We want to color all the nodes using two colors, let's say red and blue, so that all edges are incident to two nodes of different color. This is called a proper two-coloring of the graph.

Implementation (20 points)

For this subtask, we were given the condition that every pair of cities is connected by at most one path of ferry connections. In other words: There is no cycle in the network of ferry connections. Such a graph without cycles is called a forest, because each connected component is called a tree. As we will argue in the next subtask, a valid two-coloring of such a graph can be found using an algorithm called breadth first search. Breadth first search is a standard graph traversal algorithm,

i.e. it is a systematic way of visiting each node of the graph exactly once. It performs the following simple steps:

- Take any vertex of the graph that has not been visited yet and start your search from there.
- Keep a queue of vertices that we still have to visit and add the starting vertex to it.
- Whenever you visit a vertex, you add all its yet-unvisited neighbors to this queue.
- Whenever the queue is empty, you add the next unvisited vertex to it.

For the first vertex, we pick an arbitrary color. Whenever we visit a vertex, we will color all its neighbors using the opposite color and enqueue all of these vertices for the traversal, so that next we will color the neighbors of the neighbors again with the opposite color. Below you can see an implementation of this idea in C++.

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

void testcase() {
    int N; cin >> N;
    int M; cin >> M;
    // To store all the ferry connections, we will keep in
    // G[x] a list of all the neighbors of city x.
    vector<vector<int> > G(N);
    for (int i = 0; i <= M; i++) {
        // Read the two endpoints of the ferry connection.
        int a, b; cin >> a >> b; a--; b--;
        G[a].push_back(b); G[b].push_back(a);
    }
    // Keep track of which vertices we have already seen and
    // how we colored them.
    vector<int> seen(N);
    vector<int> color(N);
    // The queue of vertices we still have to visit in our BFS.
    queue<int> Q;
    for (int start = 0; start < N; start++) {
        // Start a new breadth first search in every component.
        if (seen[start]) continue;
        Q.push(start);
        seen[start] = true;
        while (Q.size()) {
            int v = Q.front(); Q.pop();
            // Loop through all the neighbors of node v.
            for (int e = 0; e < G[v].size(); e++) {
                int w = G[v][e];
                if (seen[w]) continue;
                seen[w] = true;
                // Mark w with the color different from v.
                color[w] = !color[v];
                Q.push(w);
            }
        }
    }
    for (int i = 0; i < N; i++) {
```

```
42          if(color[i]) cout << "L";
43          else cout << "R";
44      }
45  }
46
47  int main() {
48      // Loop through all the testcases.
49      int T; cin >> T;
50      for (int t = 0; t<T; t++) {
51          cout << "Case #" << t+1 << ": ";
52          testcase();
53          cout << endl;
54      }
55  }
```

Justification (15 points)

But why is this approach correct? We call this strategy a greedy startegy, because it starts coloring somewhere and then just goes from there, coloring one neighboring vertex after the other. Why does such a greedy strategy work? Why can we never get stuck and run out of options? Why are we sure that no conflict, i.e. an edge with the same color on both sides, appears?

Let us look at how we process the edges of our graph: Every edge is considered exactly twice. The first time occurs when we visit the first of its vertices, let's call this vertex v and the other vertex w. At this point vertex v already has a color and is marked as seen. Vertex w on the other side has never been seen before.

Why is this? Assume it would have been seen before when we visited some vertex v', so there has to be an edge from v' to w. If we now trace back the way we got to v to the starting vertex s and figure out the path of how we got from s to v' in our traversal, these two paths will have at least the starting vertex in common. Now this means that from s there are two different paths to w. We can either take our path to v and then take the edge to w or follow the path from the start to v' and get to w from there. This is a contradiction to the assumption that was given for this subtask. So we can conclude that such a vertex v' can not exist and we really see w for the first time.

Now as we see w for the first time, we can simply color it with the color different from v and know for sure that the edge from v to w will not be monochromatic. The second time we look at this edge is when we visit w, but at that point v will be marked as seen and not recolored. As this argument applies for any edge in the graph, we conclude that our greedy algorithm correctly colors all the nodes.

The runtime of this approach is linear in the number of nodes n and edges m, as we visit each node only once and look at each edge only twice, so $O(n+m)$ runtime. Our memory consumption is also linear in the number of nodes and edges, so also $O(n+m)$. One can argue that in this subtask, where there can not be any cycles in the graph, the number of edges can not exceed the number of nodes and therefore everything is just in $O(n)$.

We awarded 9 points for the explanation of why a greedy strategy works in this subtask and why we can never get stuck while using it and 6 points for the complexity analysis.

Relaxation of the condition (15 points)

In this subtask we were asked to find a looser condition on when it is possible to let all connections cross the river. So when exactly is it possible to color a graph using only two colors? The restriction

of not allowing cycles that we had before was too strict, as you saw in the example given in the task description. If you look at our argument for the greedy algorithm above, we did not really need that two different paths from the start vertex s to vertex w are not possible, it suffices if their lengths have the same parity. If we start at s and color along these two paths in alternating colors we will end up with the same color for w if and only if both paths have even or both have odd length. As long as this condition holds for every pair of paths between two vertices our greedy procedure from before will not get into trouble. One can also equivalently state this condition as: All cycles in the graph must be of even length.

So this condition is necessary (if it holds, we find a coloring) but why is it also sufficient (if it does not hold, there is no valid coloring)? Assume the condition does not hold, which means that there is a cycle of odd length in our graph. Any cycle of odd length can not be properly colored using two colors. If you start coloring the vertices red, blue, red, blue, ... then you will end up with the same color for the first and last vertex.

So our breadth first search algorithm from before works equally well for any two-colorable graph with the same runtime and memory complexities. Also approaches based on depth first search work equally well in both subtasks. We awarded 6 points for the correct condition, 6 points for a proper argument and 3 points for the algorithm and its complexity.

Subtask 2: At Least Half of It River-Crossing (total 50 points)

For the second half of this task we dropped all the conditions on the graph and therefore can no longer guarantee that it is two-colorable. So the task here was to just find a coloring that has at least half of its edges colored in two different colors.

Implementation (20 points)

There are two interesting solutions for this task, that we call the *incremental* and the *inductive* solution. You can find implementations of these here and explanations and analysis below.

Incremental Solution

```
#include <iostream>
#include <vector>
#include <queue>
#include <cstdlib>

using namespace std;

void testcase() {
    int N; cin >> N;
    int M; cin >> M;
    vector<vector<int> > G(N);
    for (int i = 0; i < M; i++) {
        int a, b; cin >> a >> b; a--; b--;
        G[a].push_back(b); G[b].push_back(a);
    }
    // Initialize with a random coloring.
    vector<int> color(N);
    for (int i = 0; i < N; i++) {
```

```
        color[i] = rand() % 2;
    }
    // Repeat until we can no longer find a vertex to flip.
    bool found = true;
    while (found) {
        found = false;
        for (int i = 0; i < N; i++) {
            // Count the number of edges to vertices of the same color.
            int count = 0;
            for (int e = 0; e < G[i].size(); e++) {
                int j = G[i][e];
                count += (color[i] != color[j]);
            }
            if (2 * count < G[i].size()) {
                // We found a vertex with fewer neighbors of different color than
                // neighbors of the same color so we will put it to the other side.
                found = true;
                color[i] = !color[i];
                break;
            }
        }
    }
    for (int i = 0; i < N; i++) {
        if(color[i]) cout << "L";
        else cout << "R";
    }
}

int main() {
    // Loop through all the testcases.
    int T; cin >> T;
    for (int t = 0; t < T; t++) {
        cout << "Case #" << t+1 << ": ";
        testcase();
        cout << endl;
    }
}
```

Inductive Solution

```
#include <iostream>
#include <vector>
#include <queue>
#include <cstdlib>

using namespace std;

void testcase() {
    int N; cin >> N;
    int M; cin >> M;
    vector<vector<int> > G(N);
    for (int i = 0; i < M; i++) {
        int a, b; cin >> a >> b; a--; b--;
        G[a].push_back(b); G[b].push_back(a);
    }
```

```
    // Initialize with an empty coloring. We use
    // 1: Left, 0: No color assigned, -1: Right
    vector<int> color(N, 0);
    // Inductively place the vertices left or right without ever changing them again.
    for (int v = 0; v < N; v++) {
        // Count whether there are more edges to the left or right.
        int sum = 0;
        for (int e = 0; e < G[v].size(); e++) {
            sum += color[G[v][e]];
        }
        color[v] = (sum > 0) ? -1 : 1;
    }
    for (int i = 0; i < N; i++) {
        if(color[i] == 1) cout << "L";
        else cout << "R";
    }
}

int main() {
    // Loop through all the testcases.
    int T; cin >> T;
    for (int t = 0; t < T; t++) {
        cout << "Case #" << t + 1 << ": ";
        testcase();
        cout << endl;
    }
}
```

Justification (30 points)

The central idea for solving this task is this: If less than half of all edges cross the river, than there has to be a vertex that has more than half of its neighbors on the same side (Proof: Assume the contrary, i.e. all vertices have at least half of their neighbors on the other side, then at least half of all edges cross the river - a contradiction to our assumption). So whenever our current coloring does not color half of the edges with two colors, there has to be a vertex with more neighbors on the same than on the other side. We can easily find such a vertex just by checking all of them. Once we have found such a vertex, we can increase the number of river-crossing edges by putting this vertex to the other side.

If we repeat this process until we can no longer find such a vertex, we get our incremental solution. Since each step strictly increases the number of river-crossing edges, this process has to stop after at most $O(m)$ many steps. Each step, as implementated above, takes $O(n + m)$ time and so the total runtime can not exceed $O(m \cdot (n + m))$. As the number of edges can be as large as $O(n^2)$, this approach can be rather slow and in fact there is a faster solution.

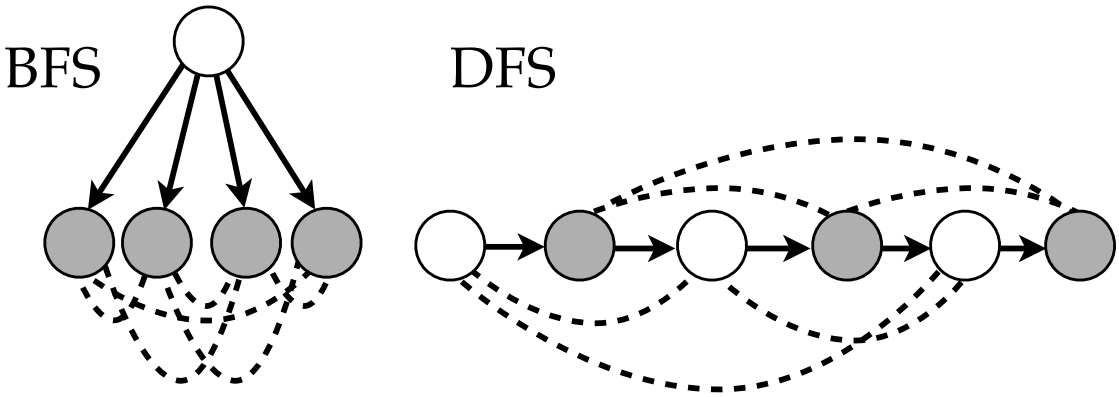
In this next, inductive approach we will take n steps. In step i we only look at all the edges among the first i vertices. So in the first step, we only look at the first vertex v_1 and no edges at all. We can color it with any color. In the second step, we take the second vertex v_2 and see whether there is an edge between v_1 and v_2. If this is the case, we color v_2 with the color different from v_1. So far, we can easily achieve that at least half of all the edges cross the river and we will now argue that we can keep up this invariant throughout all the steps. In the general step i, we consider the new vertex v_i and all edges between v_i and $v_1, v_2, \ldots, v_{i-1}$. For all the edges that were considered before step i, we know inductively that at least half of them cross the river. For all the edges newly

considered in step i, we can achieve this as well by coloring the new vertex v_i in the minority color among his neighbors in $v_1, v_2, \ldots, v_{i-1}$. So if more neighbors of v_i are colored blue, we color v_i red and vice versa. As we can achieve our bound for both the old and new edges, it also still holds overall. After performing all n steps, it holds for the entire graph, as we looked at all the edges.

In each step, we look at all the neighbors of the new vertex, so in total we look at each vertex once and at each edge twice. The inductive solution therefore only takes $O(n + m)$ time. Both solutions use $O(n + m)$ memory, simply to store the graph and the coloring.

We awarded 15 points for a correct idea, 6 points for a description of how the idea can be turned into an algorithm and 6 points for the analysis of the algorithm. This way 27 points can be earned for the iterative approach. The 3 remaining points where awarded for the faster, inductive solution.

For this subtask we also saw a lot of wrong submissions that claimed that the algorithms from the first part still work in this subtask. A lot of you submitted breadth first or depth first search solutions and claimed that they also guarantee that half of the edges will cross the river. Unfortunately, this is not true. While you might have been lucky and it worked on the practical test data, it is not hard to find counterexamples against specific implementations by hand. In the two pictures below you can find a counterexample to BFS and DFS. The black edges are the ones taken by the algorithm and the dashed edges are the remaining ones, that all happen to be monochromatic. You can see that less than half of the edges have two differently colored endpoints in both examples:



Final note: Of course, some of these solutions are much more extensive than what we expected from you. If you have any questions about these solutions or about the feedback for your submissions, please let us know: info@soi.ch

6 Bazaar

In this task, you had to write a bot to compete in an auction of diamonds.

Subtask 1: Grading scheme

You could get 30 points for a correct bot and at most 70 points for how well your bot played compared to the other participants.

To award the 30 points, every bot started with full points and for mistakes points were subtracted. Small mistakes like a missing newline or an off-by-one error (e.g. you used r-2 instead of r-1) resulted in at most 10 points of reduction, depending on the mistake. Bigger mistakes which we could fix (e.g. most "divided by zero" errors or Integer overflow) were awarded with 5 points. If your bot didn't compile or the mistake was in the idea and not in the implementation no points were awarded.

To award the 70 points all bots participated in a big tournament (with several thousand games so we could eliminate luck). Depending on the rank in this tournament more or less points were awarded. The first few bots received 70 points, the last few 10 points and in between the points were scaled linearly.