Analysis and solution of different algorithmic problems

Albert Martínez García
Director: Salvador Roura Ferret

Bachelor Thesis
Bachelor Degree in Informatics Engineering
Specialization: Computing

Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)
April 20, 2017

Abstract

The goal of competitive programming is being able to find abstract solutions for some given algorithmic problems, and and also being able to code those ideas into an efficient and correct computer program. Performing this activity at a high level requires a bit of natural ability, (at least) hundreds of training hours, and a wide range of knowledge, obviously including many algorithms and data structures, some of them not trivial at all.

This project constitutes a compilation of problems from several different relevant topics in competitive programming, with an explanation and analysis of their solution. Most of these problems were solved while training with the UPC programming teams, which have dominated their regional competition for more than one decade.

The author hopes that this collection may eventually increase the interest of some readers towards competitive programming.

Contents

1 Introduction

1.1 Context

Competitive programming can be considered a mind sport where participants are given a set of mathematical and algorithmic problems, with well defined constraints, and they are asked to write computer programs to solve them efficiently. Sometimes, the statements of some problems are presented in such a way to resemble real world problems. Many times, solving this kind of problems requires previous knowledge of a wide range of algorithms and data structures.

Solutions to the problems are usually expected to read some input data, calculate the solution for the given input, and print the answer. Solutions are typically evaluated by automatic judges, which execute the participant's program, and compare the output against the judge's solution to determine if the solution is correct or not. Note that not only the correctness is evaluated, but also the efficiency. Solutions that require too much memory or time to calculate the answer are considered unacceptable. Most problems have a memory limit of 512MiB, and a time limit of 3 seconds on a single threaded CPU.

Competitive programming has a huge community, and there are many programming competitions being held frequently, and also some well-known websites.

For instance, there are two websites that do not run competitions, but do provide a huge list of problems with its online judge to evaluate solutions. UVa Online Judge[14] has many different problems, and ACM-ICPC Live Archive [2] contains a list with most of the problems from past ACM ICPC regionals and World Finals contests.

Also, there are some online communities that hold programming contests regularly. Perhaps the most famous are Codeforces [4], Topcoder [5], and Codechef [3]. These websites usually maintain an ELO-like system for their participants, based on their performance in contests.

Some famous technology companies organize their own annual competitions, mainly to try to recruit bright employees. Indeed, these companies are very interested in the abilities involved in competitive programming. The most popular such competitions are Google Code Jam, from Google [8], HackerCup, by Facebook [7], and Yandex Algorithm Context, by Yandex [15].

One of the oldest competitions is the ACM International Collegiate Programming Contest (ACM-ICPC)[1], where universities from all over the world take part. The UPC runs some competitive programming training activities in order to prepare its teams for the ACM-ICPC contests. The UPC has been taking part in this contest since 2002, qualifying for the World Finals in most occasions.

1.2 Objective

This project could be considered as a part of the training for the ACM ICPC regional contest. The product of this project is a compilation of different problems from competitive programming with an analysis and C++ code of the solution. The problems are structured by topics, covering many important topics in competitive programming. This constitutes a study guide that aims to help any student willing to perform better in programming competitions.

1.3 Actors

In this section we list all entities involved in the project development.

1.3.1 Project development

The project involves solving some problems created by other people from different resources mentioned in the introduction. The analysis will be done by the author of the project. Most of the code for the solutions is written by the author.

1.3.2 Project Directors

The director of the project is professor Salvador Roura, who is also the coach of UPC's teams at ACM ICPC. He is coordinating the project making sure that it meets the goals.

1.3.3 Users

The final product from the project will be a study guide for many different algorithmic problems. It is targeted to any person willing to get into deep in the world of competitive programming, or just wanting to learn more techniques to solve some algorithmic problems.

The development of this project will also be useful for the author as part of his training for the ACM ICPC competitions.

1.4 State of the Art

There are some well-known books that cover a broad range of algorithms and data structures. Notable examples are Introduction to Algorithms [17], and Algorithms [18]. While they are excellent computer science books, they focus on the theoretical part, and using them as the only source to participate in programming competitions can be an overhead.

There are some other books with a more practical approach. Competitive Programming 3[19] is a great example. However, even though it provides a list of problems for each section, it fails to provide the solution code to those problems.

Some of the online communities mentioned before contain some problems with tutorials and explanations, which are great if you know what you are looking for, but someone learning new topics may prefer to have some guide more structured.

The document produced in this project differs from the previous works as it provides a compilation of problems, in a structured form, covering important topics from competitive programming, with both the analysis of why it works and the code of the solution.

1.5 Scope

Since there are so many topics that appear in competitive programming, and there is also such a vast amount of unique problems, we cannot cover them all in a single project and with the time constraints that we have. Therefore, we had to choose a subset of topics, and at least one problem from each topic.

The sections that will be covered are: dynamic programming, graphs, tree structures, sqrt heuristics, and strings. From these sections, the project will try to cover important, recent, as well as interesting issues.

1.6 Methodology and rigor

Part of this project has been developed during the training sessions that are held at the UPC, where the students that take part in ACM ICPC competitions prepare before the contest. Usually, a set of random problems from a past programming contest is chosen from one of the many online resources. Then, the contestant has to fully understand the problem, come up with a solution, perhaps discuss it with his team mates, and write a full code that is both correct and efficient.

For the latest years, these training sessions take place three times per week, five hours per session. The director of this project is involved in the training, as he is the coach of UPC teams for the ACM ICPC. He tries to provide some help for hard problems.

Further development of the project will be supervised on regular meetings with the director of the project, to ensure that it is done properly.

In the documentation, the analysis of the problems will be supervised by the director to make sure that they are correct.

2 Temporal planning

This section is about temporal planning, and it aims to describe the tasks that are going to be executed in order to do the project, giving an action plan that summarizes the actions that have to be taken in order to finish the project in the desired time frame. However, we have to take into account that the planning described in this project is subject to modifications depending on the development of the project.

The project starts on October 2016, and its deadline for the presentation is end of April 2016, therefore we have to develop our schedule given these time constraints.

2.1 Description of tasks

In this section we are going to describe the tasks that we have planned to do in order to make our project.

2.1.1 Discussion of the project

This is the first part, discussing with the director what the project will be about, how to focus the project and how is it going to be done. This happens at the beginning of the project, but further discussion may happen in the future to decide whether some topic should be included or not.

2.1.2 Problem Solving

A good amount of time is dedicated to finding and solving algorithmic problems. Great part of this task is done during the training for the ACM ICPC held at UPC, with the supervision of the director of the project. This creates a pool of problems from which to select the problems to be included in the final document later.

Any problem or section not done during the training will be done with individual work from the author to cover these topics.

During this phase, all the code is written, but most of the analysis will be written in a formal way during the writing of the documentation.

2.1.3 Writing of the documentation

This task includes writing all the documentation for the project. Gathering all the problems and code written, and structure it in the best way.

This part also includes writing all the formal analysis for each problem that has not been done during the problem solving.

2.1.4 Final task

In this task we are going to review all the documentation to make sure is correct, and prepare the final presentation.

2.2 Time table

The table 1 summarizes the time spent in each of the tasks described in the previous section

Table 1: Summary of the time spent in each task.

Task	Time spent (hours)
Discussion of the project	10
Problem Solving	370
Writing of the documentation	100
Final Task	20
Total	500

2.3 Resources used

We can divide the resources that we are going to use in hardware resources and software resources. Here are both lists, with the tasks that each resource is used in included.

2.3.1 Hardware resources

- Laptop (with the following specifications): Intel Core i5 5500U, 8GB RAM.

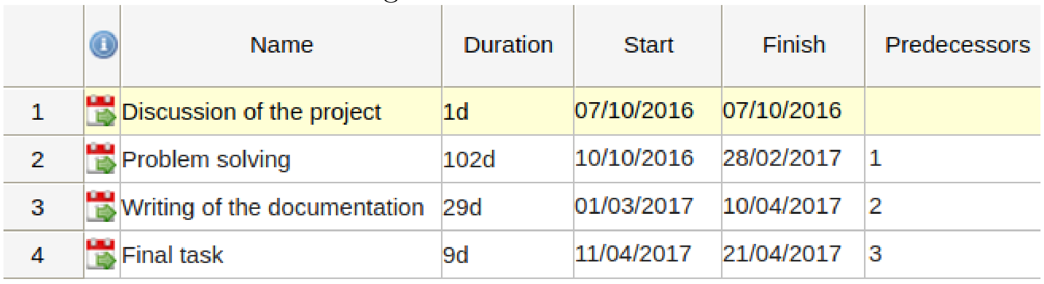
2.3.2 Software resources

- Vim 7.4: editor.
- g++ 5.3.0: compiler.
- LaTeX: used in all documentation.
- Online competitive websites: They are the source to most problems and automatic judging of the solution, as described in the introduction.

2.4 Gantt chart

Our schedule is represented by the Gantt chart shown in the figure 2, along with the dates ranges in figure 1.

Figure 1: Dates interval.



		Name	Duration	Start	Finish	Predecessors
1		Discussion of the project	1d	07/10/2016	07/10/2016	
2		Problem solving	102d	10/10/2016	28/02/2017	1
3		Writing of the documentation	29d	01/03/2017	10/04/2017	2
4		Final task	9d	11/04/2017	21/04/2017	3

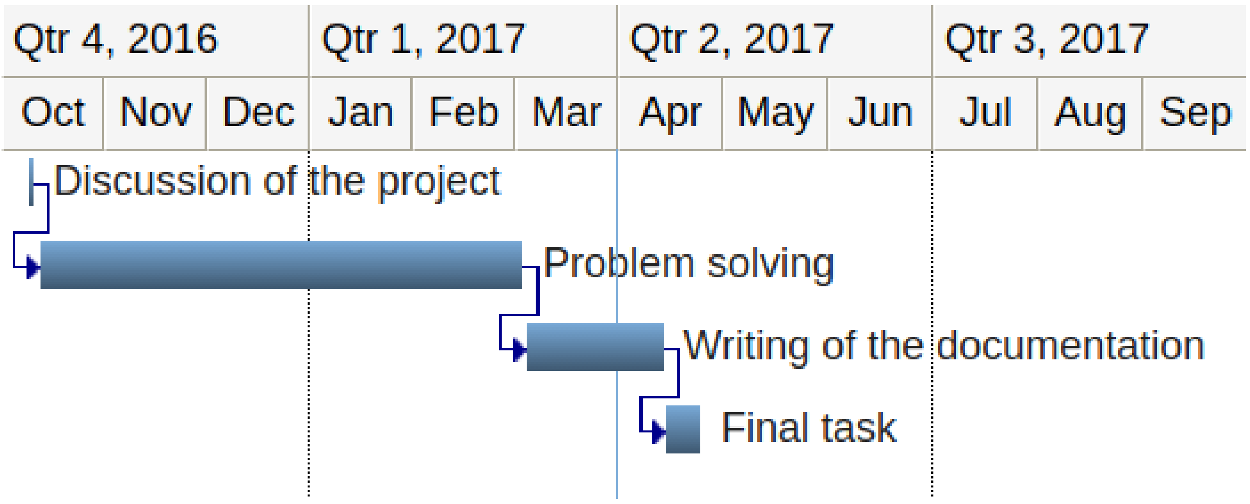


Figure 2: Grantt chart.

2.5 Action plan

In this section we are going to describe how are we going to execute the plan we have created.

Our idea is to work as we have planned, executing the tasks in the order stated in the Gantt chart.

The training for the ACM ICPC at UPC happen three times per week, 5 hours each day. Having a fixed schedule will ensure we are working towards the project as expected. In these training the director of the project is also present, as he is the coach of UPC teams for ACM ICPC, so they will be used to communicate with the director regarding the project to make sure the project succeeds.

After the training, individual work will be done from the author. To coordinate the project from this point, communication from emails and occasional mails with the director will be used.

From the time table it can be deduced that we need a total of 500 hours. Considering that we have approximately 27 weeks to work, that is about 18 hours per week of work, which is doable.

3 Budget and sustainability

This section is about the budget and the sustainability of the project. For this reason, it contains a detailed description of the costs of the project, describing both material and human costs, an analysis of how the different obstacles could affect our budget and an evaluation of the sustainability of the project. Just like in the previous section, the budget described is subject to modifications depending on the development of the project.

3.1 Budget estimation

In this section we are going to do an estimation of the budget needed in order to make our project possible. We are going to divide the budget in three sections, depending on the kind of resources that we are taking into account. They are hardware, software and human resources. At the end of the section, we are going to show the total budget obtained combining the three sections mentioned. To calculate the amortization we are going to take into account two factors, the first one being the useful life and the second one being the fact that our project is going to last for approximately six months.

3.1.1 Hardware resources

The table 2 contains the cost of the hardware that we are going to use in the development of the project.

Product	Price	Useful life	Amortization
Laptop	1500.00 €	4 years	187.00 €
Total	1500.00 €		187.00 €

Table 2: Hardware budget.

3.1.2 Software resources

The table 3 summarizes the costs of the software that we are going to need to develop our project.

3.1.3 Human resources

The table 4 shows the costs of the human resources needed in the development of our project.

Product	Price	Useful life	Amortization
Manjaro Linux	0.00 €	4 years	0.00 €
Vim 7.4	0.00 €	4 years	0.00 €
g++ 5.3.0	0.00 €	4 years	0.00 €
LaTeX	0.00 €	4 years	0.00 €
Total	0.00 €		0.00 €

Table 3: Software budget.

Role	Price per hour	Time	Cost
Project manager	50 €	250 h	12500.00 €
Software engineer	30 €	500 h	15000.00 €
Total		750 h	27500.00 €

Table 4: Human resources budget.

In this case, we have separated the roles of the project manager and software engineer.

The project manager is the director of the project. He supervises all the project. He is helping in the problem solving task, as he coordinates ACM ICPC training at UPC and provides help to solve the problems. The project manager also supervises the documentation written.

The software engineer does all the tasks. During the solving problems he comes up and writes the solution. He also writes the corresponding documentation.

Looking at the Gantt chart in figure 2, it is clear that most of the budget will go into the task of problem solving.

3.1.4 Total budget

Using data shown in tables 2, 3 and 4, we can use table 5 to describe the total cost of the project.

Concept	Cost
Hardware resources	187.00 €
Software resources	0.00 €
Human resources	27500.00 €
Total	27687.00 €

Table 5: Total budget.

3.2 Budget control

As the planning is just an estimation of the total time the project will take, it may happen that the real total time is different from the estimated. Solving a single problem can take from 5 minutes to more than 5 hours, so making an estimation can be hard. This would have a direct impact in the budget, because the time spent in the project is proportional to the budget spent on human resources.

Regarding the software and hardware resources, it is very unlikely that these costs change, as we are not paying for software right now, and the laptop from hardware resources is more than enough to complete the task.

To monitor the expenses during the execution of the project, the total amount of working hours will be counted. This total number of hours is what we are going to pay in the human resources costs.

3.3 Sustainability

In this section we are going to evaluate the sustainability of our project in three different areas: economic area, social area and environmental area.

3.3.1 Economic sustainability

In this document we already can find an assessment of the costs of our project, taking into account both material and human resources.

The cost stated in the budget estimation section could be the only cost in the project. All documentation containing the analysis is done by the author and revised by the director, so it should not contain any errors. If it ends up having errors, some correction should be required which would increase the cost of the project, but errors are very unlikely, so this increase will either not occur, or be very small. Other than this, the project requires no maintenance, as any expansion to this work is likely to be part of a separate project.

Among all costs listed before, the hardware would be the one we could reduce by getting a cheaper laptop, but this laptop was already purchased before this project, so it would be very difficult to create a similar project with lower cost after all.

Overall this project is going to be awarded a 9 in the economical viability area, since the price is mostly adjusted to what is needed.

3.3.2 Social sustainability

This project aims to be a kind of style guide for anyone getting introduced and willing to learn more about competitive programming. Since it aims to help people learn, and it will be open, the project is going to be awarded a 9 in the social sustainability.

3.3.3 Environmental sustainability

The only resource impacting the environment that is used for the development of this project is the computer and the paper used to print the final documentation.

Knowing this, we can estimate the energy spent developing the project. There is an important fact that we are using an ultrabook for this project, with an average power consumption of 30W when working in the project. Calculating for all hours spent in the project, 500h, we get a total of 13.5 kWh, which is the equivalent to 9.5 kg of CO_2. It is a reasonable amount of energy. Here the ultrabook makes a crucial point, since it is very energy efficient. Maybe it is not very powerful, but the power / energy ratio is excellent.

This project is going to be awarded a 10 in the environmental analysis area. We certainly cannot spend less resources, since the only thing that is not environmental friendly is the energy that we need to execute in the project, and this is incredibly low due to the fact that we are using a low energy consumption ultrabook. Moreover, this amount of energy is negligible if we compare to the energy which the average person uses nowadays.

4 Dynamic Programming

Dynamic programming is probably the most common topic in competitive programming. Dynamic programming consists of recursively dividing the problem into subproblems, and to build a solution from that. In dynamic programming, subproblems overlap, and the result is stored so that each subproblem is calculated only once [17, 15, p.359].

Many problems in competitive programming require dynamic programming on top of some other algorithm or data structure. In this section, we only present problems requiring direct dynamic programming solutions, but further sections may contain problems requiring dynamic programming as part of solving it.

4.0.1 2015-2016 XVI Open Cup, Grand Prix of Bashkortostan, SKB Kontur Cup Stage 2 - Judgement

Statement: There is a judging system A in a programming competition consisting of n judges in a jury, each with authority a_i. When there is a submission, each judge will dictate whether it is correct or not, and the submission will be considered correct if the sum of authorities of the judges voting correct is greater than or equal to p.

Afterwards, a new authority system B is created, every judge get a new authority b_i, and a new threshold q instead of p is set to get submissions approved.

You must determine whether the two systems are equivalent, that is, whether any assignment of decisions from the jury will yield to same result on both systems. If not, give an assignment of decisions for which the result differs in both systems. Assume $1 \leq n \leq 100$ and $1 \leq a_i, b_i, p, q \leq 10^6$.

Solution: Obviously, the two systems are not equivalent if we can find an assignment T that outputs correct in A and incorrect in B (or the other way around). With the given constraints, we can model this as a knapsack problem:

We consider authority values a_i to be the item values of a knapsack, and authority values b_i to be the weight of items. We set the capacity to $q - 1$, and we find the maximum value x of the knapsack. If $x \geq p$, then the set of items we pick is the assignment T such that it evaluates to correct in system A (the sum of authority values is greater or equal than p), and incorrect in system B, as the sum of weights is less than q.

We solve two knapsacks, one as described, and another swapping values a_i with b_i, and values p with q, and we check again. If we cannot find such assignment T, then the two systems are equivalent.

The time and space complexity is $\mathcal{O}(n(p+q))$.

Listing 1: Solution for Judgement

```
#include <iostream>
#include <vector>
#include <tuple>
using namespace std;
using VI = vector<int>;
using VB = vector<bool>;
using VVB = vector<VB>;

VI va, vb;
int pa, pb;

void solve() {
  int n = va.size();
  VI v(pb, 0);
  VI tmp(pb, 0);
  VVB mat(n, VB(pb, false));
  for (int i = 0; i < n; ++i) {
    fill(tmp.begin(), tmp.end(), 0);
    int b = vb[i];
    if (b < pb) {
      mat[i][vb[i]] = true;
      tmp[vb[i]] = va[i];
    }
    for (int j = 0; j < pb; ++j) {
      if (v[j] > tmp[j]) {
        mat[i][j] = false;
        tmp[j] = v[j];
      }
      if (j + b < pb && v[j] + va[i] > tmp[j+b]) {
        mat[i][j+b] = true;
        tmp[j+b] = v[j] + va[i];
      }
    }
    swap(v, tmp);
  }
  int ind = 0;
  while (ind < pb && v[ind] < pa) ++ind;
  if (ind >= pb) return;
  cout << "NO" << endl;
  vector<bool> output(n);
  for (int i = n-1; i >= 0; --i) {
    output[i] = mat[i][ind];
    ind -= vb[i] * output[i];
  }
  for (bool b : output) cout << b;
  cout << endl;
  exit(0);
}

int main() {
  ios::sync_with_stdio(false);
  cin.tie(0);
  int n;
  cin >> n;
  va.resize(n);
  vb.resize(n);
  cin >> pa;
  for (int &x : va) cin >> x;
  cin >> pb;
  for (int &x : vb) cin >> x;
  // First case
  solve();
  swap(pa,pb); swap(va,vb);
  solve();
  cout << "YES" << endl;
}
```

4.0.2 2005-2006 ACM-ICPC, NEERC, Southern Subregional Contest - Mars Stomatology

Statement: Martian girl Kate has a toothache. The martian anatomy is very specific. They all have N teeth, each situated on one of K gums. Kate should pay dentist A_i mars euros for the treatment of i-th tooth. Moreover, Kate should pay B_j euros for the anesthesia of the gum j if this gum has at least one tooth cured. What is the maximum number of teeth Kate can cure if her parents gave her P mars euros? Print this value and the list of cured teeth. Assume $1 \leq N \leq 600; 1 \leq K \leq N; 1 \leq P \leq 10^6$.

Solution: We use dynamic programming. We sort all teeth first by gum, then by cost. The state of our dynamic programming is (n, k, gum_paid) where: n is the n-th teeth we are processing from our sorted list, k is the remaining number of teeth we want to cure, and gum_paid is a boolean of whether the cost of curing a teeth in the gum of n-th teeth has already been paid.

At each state, we can either cure the n-th tooth, pay its cost, pay the cost of the gum depending on value of gum_paid, and go to state $(n+1, k-1, new\_gum\_paid)$. The value of new_gum_paid depends on whether the next tooth's gum is same as the gum from n-th tooth.

If we don't cure the n-th tooth, we go to state $(n+1, k, new\_gum\_paid)$. The value of new_gum_paid will be true if the current gum_paid is true, and the next tooth is from the same gum as the current.

The base case is when $k = 0$, because we cured all teeth, so the answer is 0. Otherwise, if $n = N$, then we did not cure enough teeth, so we return an "infinite" cost.

To write the list of teeth that were actually cured, we call the same recursive function with dynamic programming, but now at each state we know the answer from that state. Therefore, if curing the n-th tooth yields to the answer, then we add that tooth to the list, and proceed.

The overall time and space complexity is $\mathcal{O}(N^2)$.

Listing 2: Solution for Mars Stomatology

```cpp
#include <algorithm>
#include <cstring>
#include <cstring>
#include <iostream>
#include <vector>
using namespace std;
using PII = pair<int,int>;
using PPII = pair<int, PII>;
using VPPII = vector<PPII>;
using VI = vector<int>;

const int MAXN = 605;
const int inf = 1e9;

VI gum_cost;

struct Tooth {
  int gum, cost, id;
  Tooth(int gum = 0, int cost = 0, int id = -1)
      : gum(gum), cost(cost), id(id) {}
  bool operator<(const Tooth &other) {
    if (gum != other.gum) return gum < other.gum;
    if (cost != other.cost) return cost < other.cost;
    return id < other.id;
  }
};
vector<Tooth> teeth;

int N, K, P;
int memo[MAXN][MAXN][2];
int f(int n, int k, int gum_paid) {
  if (k == 0) {
    return 0;
  }
  if (n == N) {
    return inf;
  }
  int &ans = memo[n][k][gum_paid];
  if (ans != -1) return ans;

  int same_gum = (n+1 >= N || teeth[n+1].gum == teeth[n].gum);
  // Cure n-th teeth.
  int gum = teeth[n].gum - 1;
  ans = min(inf, (gum_paid ? 0 : gum_cost[gum]) + teeth[n].cost +
                 f(n+1, k-1, same_gum));
  // Do not cure
  ans = min(ans, f(n + 1, k, (same_gum & gum_paid)));
  return ans;
}

bool first_added = false;
// Aux function to print the teeth
int g(int n, int k, int gum_paid) {
  if (k == 0) {
    return 0;
  }
  if (n == N) {
    return inf;
  }
  int &ans = memo[n][k][gum_paid];
  int same_gum = (n+1 >= N || teeth[n+1].gum == teeth[n].gum);
  // Cure n-th teeth.
  int gum = teeth[n].gum - 1;
  int take = (gum_paid ? 0 : gum_cost[gum]) + teeth[n].cost +
             f(n + 1, k - 1, same_gum);
  // If curing this teeth yields to optimal result, is part of the answer.
  if (take == ans) {
    if (!first_added) first_added = true;
    else cout << '␣';
    cout << teeth[n].id;
    return g(n+1, k-1, same_gum);
  }
  return g(n+1, k, (same_gum & gum_paid));
}

int main() {
  ios_base::sync_with_stdio(false);
  cin >> N >> K >> P;
  teeth.resize(N);
  gum_cost.resize(K);

  for (int i = 0; i < K; ++i) {
    cin >> gum_cost[i];
  }
  for (int i = 0; i < N; ++i) {
    cin >> teeth[i].cost >> teeth[i].gum;
    teeth[i].id = i+1;
  }
  sort(teeth.begin(), teeth.end());
  int mx = 0;
  memset(memo, -1, sizeof memo);
  // Find largest number of teeth that can be cured.
  for (int k = 0; k <= N; ++k) {
    int cost = f(0, k, 0);
    if (cost <= P) {
      mx = k;
    }
  }
  cout << mx << endl;
  g(0, mx, 0);
  cout << endl;
}
```

4.0.3 UVA 1252 - Twenty Questions

Statement: There are n objects with m boolean properties each, numbered from 0 to $m-1$. Each object differs from others by at least one property. There is a hidden object (from the set of n objects) which you want to guess. You can ask for one property at a time, and you will get the value of the property for that hidden object. Assume $0 \leq m \leq 11$ and $0 \leq n \leq 128$.

Given the set of n objects, each with its values for each property, minimize the maximum number of questions by which every object is identifiable, and output the result.

Solution: Use dynamic programming. The state is $(bm1, bm2)$, where $bm1$ is the bitmask of questions already made (the value of teh i-th bit represents whether property i was asked or not), and $bm2$ is the bitmask with the answers to the questions made from $bm1$ (if i-th property was asked, i-th bit from $bm2$ is the answer we got from that property).

Our dynamic programming calculates $f(bm1, bm2)$, the minimum number of questions required, with the current information of $bm1$ and $bm2$.

We have the base case where the given answers given match with zero or one objects. Then no more questions are needed. Otherwise, we iterate over all unanswered properties, and visit the corresponding states $(newbm1, newbm2)$, with the proper bits set. For each property, we visit the state where the answer is 0, and the one where the answer is 1, and we keep the highest value, because that is the worst case. Then, from the values of each asked property, we choose the minimum one, because that is the property we have to ask now if we want to minimize the number of questions in the worst case. So the number of questions required from $(newbm1, newbm2)$, plus 1 for the property we are asking now, is the answer to the state $(bm1, bm2)$.

The space complexity is $\mathcal{O}(2^{2m})$, because state is represented by two bitmasks of length m each. Time complexity is $\mathcal{O}((n+m)2^{2m})$, because in each state we iterate over all n objects, and over all m properties.

Listing 3: Solution for Twenty Questions

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;
using VI = vector<int>;
using VVI = vector<VI>;

VI objects;
VVI memo;
int N, M;

inline int set_bit(int bm, int pos, int value) {
  return (bm & (~(1<<pos))) | (value << pos);
}

int num_questions(int bm_asked, int bm_answers) {
  int num = 0;
  for (int object : objects) {
    if ((object & bm_asked) == bm_answers) {
      ++num;
    }
  }
  return num;
}

int f(int bm_asked, int bm_answers) {
  int &ans = memo[bm_asked][bm_answers];
  if (ans != -1) return ans;
  // If one or less objects match, no more questions needed.
  if (num_questions(bm_asked, bm_answers) <= 1) {
    return 0;
  }
  ans = 100;
  for (int i = 0; i < M; ++i) {
    if (bm_asked & (1<<i)) continue;
    int t = max(f(bm_asked | (1<<i), set_bit(bm_answers, i, 0)),
                f(bm_asked | (1<<i), set_bit(bm_answers, i, 1)));
    ans = min(ans, t + 1);
  }
  return ans;
}

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

  while (cin >> M >> N && (M | N)) {
    objects = VI(N);
    memo = VVI(1<<M, VI(1<<M, -1));
    for (int i = 0; i < N; ++i) {
      string bitmask;
      cin >> bitmask;
      objects[i] = stoi(bitmask, 0, 2);
    }
    cout << f(0, 0) << endl;
  }
}
```

5 Graphs

Many real life problems can be modeled as graph problems. This fact makes graphs a common topic in competitive programming. Graph algorithms are very diverse, from calculating the shortest path, to finding the maximum flow in a network, for instance.

5.1 Breadth-first search

Breadth-first search is a simple algorithm to explore a graph, starting from a source and visiting the neighbors first before moving to the next level of neighbors. It can be used to compute the shortest path between two nodes in an unweighted graph. It runs in time $\mathcal{O}(|V| + |E|)$.

5.1.1 Codeforces 101047 - Escape from Ayutthaya

Input: There is an $n \times n$ 2-dimensional map of cells. Each cell is either your position, the exit door, a wall, a fire, or an empty space. Each turn, you can move one cell either vertically or horizontally. Fire propagates in every direction, vertically and horizontally, at the same speed. If you occupy a cell in fire, you die immediately. Decide whether you can escape or not.

Output: The solution is just one BFS from the exit to calculate the shortest distance from any fire to the exit, and from the prisoner to the exit as well. The prisoner can escape if and only if he is strictly closer to the exit than any fire. The overall complexity is linear in the size of the map, that is, $\mathcal{O}(n^2)$.

Listing 4: Solution for Escape from Ayutthaya

```cpp
#include <iostream>
#include <queue>
using namespace std;
using ll = long long;
using PI = pair<int,int>;
const int inf = 1e6;
const int di[4] = {0, 1, 0, -1};
const int dj[4] = {1, 0, -1, 0};

const int maxn = 1005;
char mat[maxn][maxn];
int dist[maxn][maxn];

int main() {
  ios_base::sync_with_stdio(false);
  cin.tie(0);
  int T; cin >> T;
  while (T--) {
    int n, m; cin >> n >> m;
    PI st, nd;
    for (int i = 0; i < n; ++i) {
      for (int j = 0; j < m; ++j) {
        cin >> mat[i][j];
        dist[i][j] = inf;
        if (mat[i][j] == 'E') {
          nd = {i,j};
        }
      }
    }
    queue<PI> q;
    q.push(nd);
    dist[nd.first][nd.second] = 0;
    int firedist = inf;
    int prisonerdist = inf;
    while (!q.empty()) {
      PI u = q.front(); q.pop();
      for (int k = 0; k < 4; ++k) {
        PI npos = {u.first + di[k], u.second + dj[k]};
        if (npos.first < 0 || npos.first >= n ||
            npos.second < 0 || npos.second >= m) {
          continue;
        }
        char t = mat[npos.first][npos.second];
        if (t == '#') continue;
        int ndist = dist[u.first][u.second] + 1;
        if (t == 'F') firedist = min(firedist, ndist);
        if (t == 'S') prisonerdist = min(prisonerdist, ndist);
        if (ndist < dist[npos.first][npos.second]) {
          q.push(npos);
          dist[npos.first][npos.second] = ndist;
        }
      }
    }
    cout << (prisonerdist < firedist ? "Y" : "N") << endl;
  }
}
```

5.2 Dijkstra

Dijkstra's algorithm solves the single-source shortest paths problem on a weighted graph with positive costs. In competitive programming, this is usually implemented with the `std::priority_queue` from the C++ STL for simplicity. The code has a worst-case time complexity of $\mathcal{O}(E \log V)$ [17, 24, p.658-662] [11].

5.2.1 2014 Winter Programming School, Kharkiv, day 1 - A path to knowledge

Statement: Alex studies at the university and goes there everyday by foot. The city where Alex lives can be modeled as a non-oriented graph. Every time that Alex goes to the university or back, he uniformly chooses one of the shortest paths. He wonders how many times per day he goes through each node on the average.

Solution: Run Dijkstra two times. One from Alex's home, and another from the university. On each node we store the minimum distance to both Alex's home and the university. Let the shortest distance from home to the university be d. For each node u, u belongs to a shortest path if the shortest distance from home to u plus the shortest distance from the university to u equals d.

The number of times per day that Alex goes through a node on the average is twice the probability of going through that node. To calculate that probability, we count the number of shortest paths going through that node, and divide by the total number of shortest paths. The paths are counted while running Dijkstra, storing for each node the number of paths, and propagating this information during the algorithm. The number of shortest paths that u belongs to, is the number of shortest paths from home to u, multiplied by the number of shortest paths from the university to node u.

Listing 5: Solution for A path to knowledge

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;
using ll = unsigned long long;
using VI = vector<int>;
using VD = vector<double>;
using VLL = vector<ll>;
using PII = pair<int,int>;
using VPII = vector<PII>;
using VVPII = vector<VPII>;
const int inf = 1e9+10000000;

void dijkstra(int s, int t, const VVPII &G, VI &dist, VD &paths) {
  priority_queue<PII, VPII, greater<PII> >pq;
  paths[s] = 1;
  dist[s] = 0;
  pq.push({0, s});
  while (!pq.empty()) {
    int w = pq.top().first;
    int u = pq.top().second;
    pq.pop();

    if (w > dist[t]) break;
    if (w > dist[u]) continue;
    for (int i = 0; i < int(G[u].size()); ++i) {
      int v = G[u][i].first;
      int d = G[u][i].second + w;
      if (d < dist[v]) {
        dist[v] = d;
        paths[v] = paths[u];
        pq.push({d, v});
      }
      else if (d == dist[v]) {
        paths[v] += paths[u];
      }
    }
  }
}

int main() {
  ios_base::sync_with_stdio(false);
  cout.precision(8);
  cout.setf(ios::fixed);
  int n, m;
  cin >> n >> m;
  VVPII G(n);
  for (int i = 0; i < m; ++i) {
    int a, b, c;
    cin >> a >> b >> c;
    --a; --b;
    G[a].push_back(PII(b,c));
    G[b].push_back(PII(a,c));
  }

  VI dist(n, inf), downdist(n, inf);
  VD paths(n, 0), downpaths(n, 0);

  // Dijkstra from home to university.
  dijkstra(0, n-1, G, dist, paths);
  // Dijkstra from university to home.
  dijkstra(n-1, 0, G, downdist, downpaths);

  double num_paths = paths[n-1];
  int total_dist = dist[n-1];

  for (int i = 0; i < n; ++i) {
    // Check each node if it belongs to a shortest path.
    if (dist[i]+downdist[i] == total_dist) {
      cout << (i?" ":"") << 2.0 * paths[i]*downpaths[i]/num_paths;
    }
    else cout << (i?" ":"") << 0.0;
  }
  cout << endl;
}
```

5.3 Floyd-Warshall

Floyd-Warshall is a dynamic programming algorithm to compute the all-pairs shortest path of a graph in $\mathcal{O}(n^3)$ time, where n is the number of nodes. It works by computing the shortest paths incrementally. It first uses 0 auxiliary nodes, then 1, and so on until using the n vertices. [17, 25.2, p.693-700].

One of the main advantages of this algorithm is that it is very easy and fast to implement, as it consists of just three simple nested loops.

5.3.1 ACM-ICPC Pacific Northwest Region Programming Contest 2014 - Wormhole

Statement: There are p planets, each with 3D coordinates $0 \leq x_i, y_i, z_i \leq 2 \cdot 10^6$. There are also w wormholes. Each wormhole connects two planets, and allows us to go from one planet to another immediately (that is, like with distance 0). There are q queries. Each query asks for the minimum distance from planet a to planet b, allowing the use of any number of wormholes. Assume $1 \leq p \leq 60$.

Solution: We can model this problem as a graph. Each planet is a node. Two nodes a and b are connected with an edge with weight set to the distance between them. If there is a wormhole connecting them, then we set the distance to 0. We then run Floyd-Warshall to precompute all-pairs shortest distance to solve each query in $\mathcal{O}(1)$. Therefore the code runs in $\mathcal{O}(n^3 + q)$.

Listing 6: Solution for Wormhole

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
using LD = long double;
using VD = vector<LD>;
using VVD = vector<VD>;

struct Pt {
  ll x, y, z;
  Pt() {}
};

LD dist(Pt a, Pt b) {
  return sqrt((a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y) + (a.z-b.z)*(a.z-b.z
      ));
}

int main() {
  ios_base::sync_with_stdio(false);
  int T;  cin >> T;
  int ncase = 0;
  while (T--) {
    int n; cin >> n;
    map<string,int> ids;
    vector<Pt> v(n);
    int cur = 0;
    for (int i = 0; i < n; ++i) {
      string name;
      cin >> name;
      ids[name] = cur++;
      cin >> v[i].x >> v[i].y >> v[i].z;
    }
    VVD mat(n, VD(n, 0.0));
    for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) if (i != j) {
        mat[i][j] = dist(v[i], v[j]);
      }
    }

    int m; cin >> m;
    while (m--) {
      string from, to; cin >> from >> to;
      mat[ids[from]][ids[to]] = 0.0;
    }

    for (int k = 0; k < n; ++k) {
      for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
          mat[i][j] = min(mat[i][j], mat[i][k] + mat[k][j]);
        }
      }
    }

    cin >> m;
    cout << "Case␣" << ++ncase << ":" << endl;
    while (m--) {
      string from, to;
      cin >> from >> to;
      cout << "The␣distance␣from␣" << from << "␣to␣" << to << "␣is␣"
           << ll(mat[ids[from]][ids[to]] + 0.5 + 1e-15) << "␣parsecs." <<
               endl;
    }


  }
}
```

5.4 Strongly Connected Components / 2SAT

A strongly connected component of a directed graph (digraph) G is a maximal set of vertices C such that for every pair $u, v \in C$, u can be reached from v in G, and v can be reached from u in G.

All strongly connected components of a digraph G can be found in linear-time for instance with Kosaraju's algorithm [12], which performs two depth first searches, one on G and another on the reversed graph of G.

One of the important applications of strongly connected components is 2-Satisfiability. We can build the implication graph, with a node for each literal. For example, the clause $u \vee -v$ translates to node $-u$ having an arc towards $-v$, and node v having an arc towards u, in the implication graph.

If a 2-SAT formula F translates into graph G, and there is a variable u such that u and $-u$ are in the same strongly connected component in G, then F is unsatisfiable, because u would imply $-u$, and $-u$ would imply u. It is also possible to show that, if there is no such variable u, then the given 2-SAT formula is satisfiable. Therefore, to solve the 2-SAT problem, we build the implication graph, find all strongly connected components, and check that no variable and its negated nodes are in the same component. We can also find a feasible assignment of values to variables, if it exists.

5.4.1 Codeforces 100729 - Piece it together

Statement: There is a grid of white, black or empty cells. Decide whether or not the grid can be constructed from any number of identical pieces, all made of three adjacent cells in an L-shape. The corner of each piece is black, and the two adjacent cells are white. The piece can be rotated in any way, but pieces must not overlap nor get out of the grid.

Solution: We can model this problem as a 2-SAT. Each variable represents a pair of adjacent cells in the grid, whether they are covered by the same piece or not.

Consider a black cell in the grid. We must place the center of the piece there. No matter how we rotate the piece, it will cover exactly one of his horizontal adjacent cells, and exactly one of its vertical adjacent cells. Let us name those pairs as t: center with top, b: center with bottom, l: center with left, and r: center with right. The clauses that express the previous requirement of having exactly one horizontal neighbor covered are $l \vee r$ and $\bar{l} \vee \bar{r}$, and for the vertical part $t \vee b$ and $\bar{t} \vee \bar{b}$.

We also make sure that there are no two pieces that overlap. We must express that no white cell can be part of more than one piece. Using the same notation to describe the pairs of neighbors as before (only that this time we are centered at a white cell), we add the following clauses: for each $u, v \in \{t, b, l, r\}, u \neq v$, we add the clause $\bar{u} \vee \bar{v}$.

We also add clauses to avoid a piece from covering two adjacent cells of the same color. If u is the variable for a pair of adjacent cells with the same color, then we add $\bar{u} \vee u$, so that no piece can be placed there.

The last restriction is to check that all white cells are covered by a piece. Currently, we are covering all black cells, and making sure that those pieces are placed legally, but we could have an isolated white cell never covered and still satisfy the clauses. To solve this, we just check that the number of white cells are two times the number of black cells.

If both the restriction of having as many white cells as twice the number of black cells, and the 2-Sat formula we described are satisfiable, then the grid can be covered with pieces as described.

Listing 7: Solution for Piece it together

```
#include <iostream>
#include <unordered_map>
#include <vector>
using namespace std;
using VC = vector<char>;
```

```cpp
using VVC = vector<VC>;
using ll = unsigned long long;
using VI = vector<int>;
using VVI = vector<VI>;
using VB = vector<bool>;

unordered_map<ll, int> ptoid;
int cur = 0;
int getid(ll a, ll b, VVI &g) {
  ll t = max(a, b);
  a = min(a, b);
  b = t;
  ll x = (a<<31)|b;
  auto i = ptoid.find(x);
  if (i != ptoid.end()) return i->second;
  g.push_back(VI());
  g.push_back(VI());
  ptoid[x] = cur++;
  return cur-1;
}

const int di[4] = {1, -1, 0, 0};
const int dj[4] = {0, 0, 1, -1};

void dfs1(int v, VB& used, VI& order, VVI& g) {
  used[v] = true;
  for (int i = 0; i < int(g[v].size()); ++i) {
      if (not used[g[v][i]]) dfs1(g[v][i], used, order, g);
  }
  order.push_back(v);
}
void dfs2(int v, int cl, VI& comp, VVI& gt) {
  comp[v] = cl;
  for (int i = 0; i < int(gt[v].size()); ++i)
    if (comp[gt[v][i]]==-1)dfs2(gt[v][i],cl, comp, gt);
}
VI scc(VVI &g, VVI &gt) {
  int n = g.size();
  vector<bool> used(n, false);
  vector<int> order;
  for (int i = 0; i < n; ++i)
    if (!used[i]) dfs1(i, used, order, g);
  VI comp(n, -1); int cc = 0;
  for (int i = 0; i < n; ++i) {
    if (comp[order[n-i-1]] == -1) {
      dfs2(order[n-i-1], cc++, comp, gt);
    }
  }
  return comp;
}

int main() {
  ios_base::sync_with_stdio(false);
  cin.tie(0);
  int T; cin >> T;
  while (T--) {
    ptoid.clear();
    VVI g;
    cur = 0;
    int n, m; cin >> n >> m;
    n += 2;
    m += 2;
    VVC mat(n, VC(m, '.'));
    int white = 0, black = 0;
    for (int i = 1; i < n-1; ++i) {
      for (int j = 1; j < m-1; ++j) {
        cin >> mat[i][j];
        white += mat[i][j] == 'W';
        black += mat[i][j] == 'B';
      }
    }
    for (int i = 1; i < n-1; ++i) {
      for (int j = 1; j < m-1; ++j) {
        if (mat[i][j] == '.') continue;
        int pairids[4];
        for (int k = 0; k < 4; ++k) {
          ll t = pairids[k] = getid(i*m+j, (i+di[k])*m + j+dj[k], g);
          if (mat[i + di[k]][j + dj[k]] == '.' ||
              mat[i][j] == mat[i + di[k]][j + dj[k]]) {
            // No piece can touch this cell.
            g[2*t].push_back(2*t+1);
          }
        }
        if (mat[i][j] == 'B') {
          // Restrictions of neighbours of B
          // Only one horizontal neighbour.
          g[2*pairids[0]+1].push_back(2*pairids[1]);
          g[2*pairids[1]+1].push_back(2*pairids[0]);
          g[2*pairids[0]].push_back(2*pairids[1]+1);
          g[2*pairids[1]].push_back(2*pairids[0]+1);
          // Only one vertical neighbour
          g[2*pairids[2]+1].push_back(2*pairids[3]);
          g[2*pairids[3]+1].push_back(2*pairids[2]);
          g[2*pairids[2]].push_back(2*pairids[3]+1);
          g[2*pairids[3]].push_back(2*pairids[2]+1);
        } else if (mat[i][j] == 'W') {
          // No overlap in White cells.
          for (int k = 0; k < 4; ++k) {
            ll t = pairids[k];
            for (int l = 0; l < 4; ++l) if (k != l) {
              ll u = pairids[l];
              g[2*t].push_back(u*2+1);
            }
          }
        }
      }
    }
    bool possible = (black*2) == white;

    // Build inverse.
    VVI gt(g.size());
    for (int i = 0; i < int(g.size()); ++i) {
      for (int u : g[i]) {
        gt[u].push_back(i);
      }
    }

    VI comp = scc(g, gt);
    for (int i = 0; i < int(g.size()); i += 2) {
      if (comp[i] == comp[i+1]) {
        possible = false;
      }
    }
```

```
    cout << (possible ? "YES" : "NO") << endl;
  }
}
```

5.5 Maxflow

Maxflow is the problem of finding a maximum feasible flow through a single-source, single-sink flow network. The minimum cut of a network is a cut that separates the source and the sink vertices, and minimizes the total sum of weights on the edges that connect a vertex from the source side with a vertex from the sink side. The minimum cut problem is equivalent to finding the maximum flow [10].

5.5.1 2014-2015 ACM-ICPC Pacific Northwest Regional Contest - Containment

Statement: There is a three-dimensional grid with good and bad cells. Bad cells must be enclosed by panels which can only be inserted between cells (so each individual panel must be axis-aligned), and each panel separates exactly two cells. Each enclosure must form a closed polytope. Calculate the minimum number of panels to fully enclosure the bad cells, even if that involves covering good cells.

Assume that there are at most 100 bad cells, each at an integer position $(x_i, y_i, z_i); 0 \leq x_i, y_i, z_i \leq 9$.

Solution: This problem can be solved as a minimum cut problem. If we model the grid as a graph, and neighbor cells are connected by edges, any placed panel is a cut in that edge. As we want the bad cells to be fully contained, they need to be disconnected (that is, cut) from the exterior of the grid. In order to use the minimum number of panels, we must cut the minimum number of edges. Therefore, we are looking for a minimum cut.

Listing 8: Solution for Containment

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

const int D = 3;
const int dx[D] = {1, 0, 0};
const int dy[D] = {0, 1, 0};
const int dz[D] = {0, 0, 1};

const int MX = 12;

typedef long long ll;
typedef vector<int> VI;
typedef vector<VI> VVI;
```

```
const ll INF = 1000000000LL;

#define VEI(w,e) ((E[e].u == w) ? E[e].v : E[e].u)
#define CAP(w, e) \
  ((E[e].u == w) ? E[e].cap[0] - E[e].flow : E[e].cap[1] + E[e].flow)
#define ADD(w,e,f) E[e].flow += ((E[e].u == w) ? (f) : (-(f)))

struct Edge {
  int u, v;
  ll cap[2], flow;
};

VI d, act;

bool bfs(int s, int t, VVI& adj, vector<Edge>& E) {
  queue<int> Q;
  d = VI(adj.size(), -1);
  d[t] = 0;
  Q.push(t);
  while (not Q.empty()) {
    int u = Q.front(); Q.pop();
    for (int i = 0; i < int(adj[u].size()); ++i) {
      int e = adj[u][i], v = VEI(u, e);
      if (CAP(v, e) > 0 and d[v] == -1) {
        d[v] = d[u] + 1;
        Q.push(v);
      }
    }
  }
  return d[s] >= 0;
}

ll dfs(int u, int t, ll bot, VVI& adj, vector<Edge>& E) {
  if (u == t) return bot;
  for (; act[u] < int(adj[u].size()); ++act[u]) {
    int e = adj[u][act[u]];
    if (CAP(u, e) > 0 and d[u] == d[VEI(u, e)] + 1) {
      ll inc = dfs(VEI(u, e), t, min(bot, CAP(u, e)), adj, E);
      if (inc) {
        ADD(u, e, inc);
        return inc;
      }
    }
  }
  return 0;
}

ll maxflow(int s, int t, VVI& adj, vector<Edge>& E) {
  for (int i = 0; i < int(E.size()); ++i) E[i].flow = 0;
  ll flow = 0, bot;
  while (bfs(s, t, adj, E)) {
    act = VI(adj.size(), 0);
    while ((bot = dfs(s, t, INF, adj, E))) flow += bot;
  }
  return flow;
}

bool ok(int a) {
  return a >= 0 and a < MX;
}

void add_edge(int u, int v, int c, vector<Edge> &E, VVI &adj) {
  Edge e;
  e.u = u;
  e.v = v;
  e.cap[0] = e.cap[1] = c;
  adj[u].push_back(E.size());
  adj[v].push_back(E.size());
  E.push_back(e);
}


void solve() {
  int n;
  cin >> n;

  VVI adj(MX * MX * MX + 2);
  vector<Edge> E;
  int source = MX * MX * MX, sink = source + 1;

  for (int i = 0; i < n; ++i) {
    int x, y, z;
    cin >> x >> y >> z;
    ++x; ++y; ++z;
    add_edge(MX * MX * x + MX * y + z, sink, INF, E, adj);
  }

  for (int x = 0; x < MX; ++x) {
    for (int y = 0; y < MX; ++y) {
      for (int z = 0; z < MX; ++z) {
        int u = MX * MX * x + MX * y + z;
        if (x == 0 || x == MX - 1 || y == 0 || y == MX - 1 ||
            z == 0 || z == MX - 1) add_edge(source, u, INF, E, adj);
        else add_edge(source, u, 0, E, adj);
        for (int d = 0; d < D; ++d) {
          int nx = x + dx[d], ny = y + dy[d], nz = z + dz[d];
          if (ok(nx) && ok(ny) && ok(nz)) {
            int v = MX * MX * nx + MX * ny + nz;
            add_edge(u, v, 1, E, adj);
          }
        }
      }
    }
  }
  cout << maxflow(source, sink, adj, E) << endl;
}

int main() {
  int t;
  cin >> t;
  while (t--) solve();
}
```

6 Tree Structures

A tree is a connected graph with n nodes and $n-1$ edges. Even though a tree is a graph, we decided to have a separate section for tree structures, as many techniques on trees share a common core, and differ from general graph algorithms. Most subsections here are about using trees as a base for a data structure.

6.1 Segment Tree

A segment tree is a data structure to solve range queries over an array of elements that support a binary operation, such as sum, xor or minimum. This is built on a complete binary tree (a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible), each leaf represents an element of the array, being the leftmost leaf the first element in the array, the second leftmost leaf the second element in the array, and so on. Each intermediate node represents a range that is the union of its children. This can be seen in figure 3. Each node u representing a range $[l, r]$ also stores the result of the query over that range. Imagine a segment tree for calculating the sum of ranges, that is, the sum of all elements from position l to r. Each leaf contains the corresponding element of the array itself. Each intermediate node will contain the sum of its children. If we calculate this approach bottom up, from last node to node 1, we can compute the value of each node in a single iteration over all nodes. We process $\mathcal{O}(n)$ nodes, and each node requires an $\mathcal{O}(1)$ operation (adding two children).

To actually solve a query on a range $[l, r]$, we start from the root. For each node, if it represents the range $[i, j]$, three situations can happen:

1. $[l, r]$ and $[i, j]$ are disjoint. This node does not contain any element in the range. We return the neutral element (0 in the case of sum).

2. $[i, j]$ is entirely contained in $[l, r]$. All elements represented by this node form part of the query. We return all elements (the value at this node).

3. $[l, r]$ and $[i, j]$ intersect. We recursively search on each child of the current node. Once we have the result for each, we calculate the result of this node applying the binary operation on both children's values.

Because we have a complete binary tree, it has $\mathcal{O}(\log n)$ height. Therefore, each query is performed in $\mathcal{O}(\log n)$ time.

To make the coding much simpler, if we have an array of elements v, we can assume its size to be a power of two. Otherwise, we append the neutral

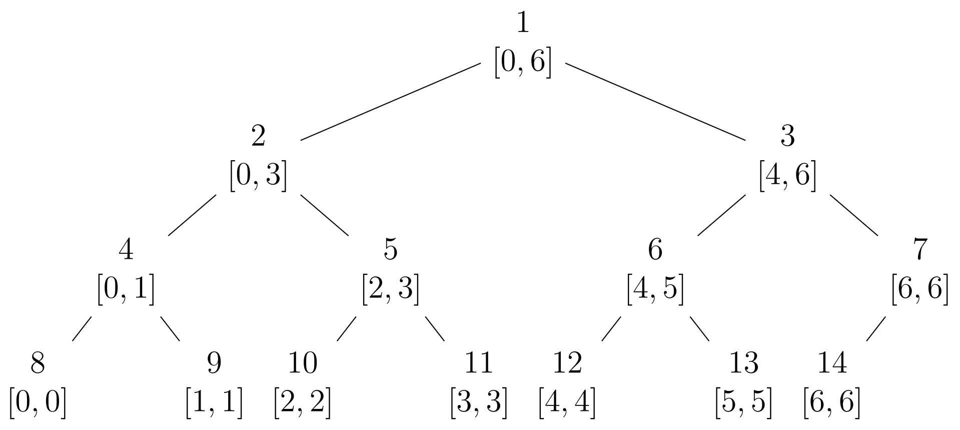


Figure 3: Segment Tree of an array of size 7. Each node contains its id, and the range it represents.

element to it until this holds. Then, let the size of v be $n = 2^k$. We can build a perfect binary tree (a tree with $2^{k+1} - 1$ nodes and height $k+1$) over an array a, where a_i is the node with id i. With this approach, a_i is the parent of nodes a_{2i} and a_{2i+1}. If node a_i represents the range $[l, r)$, then the left child a_{2i} will represent the range $[l, (l + r)/2)$, and the right child a_{2i+1} will represent the range $[(l + r)/2, r)$. For the leaves, a_{n+i} will represent the element v_i. If a leaf has id $n + i$ with $i \leq n$, it does not represent any value of v, so we set it to a neutral value. For instance, if we were solving the sum problem, we would set it to 0.

This method makes writing an interval very fast and easy, ideal for competitive programming.

6.1.1 UVA 1232 - SKYLINE

Statement: There are n axis-aligned rectangles, with integer coordinates and bottom y-coordinate at 0. You are given the left coordinate, the right coordinate, and the height of the rectangle. This rectangle will occupy certain area. For each rectangle, calculate

You want to build a unique skyline for a new city. There are many proposal, each with a sequence of n buildings $\langle b_1, ..., b_n \rangle$, where each building is specified by left, right coordinate, and its height. The buildings are specified from back to front order.

The skyline formed by the first k buildings is the union of rectangles of the first k buildings. The overlap of a building b_i is defined as the total horizontal length of the parts of b_i whose height is greater than or equal to the skyline behind it. This is equivalent to the total horizontal length of parts of the skyline behind b_i which has a height that is less than or equal to h_i, where h_i is the height of building b_i. Initially the skyline has height zero everywhere.

Assume $0 < n < 10^5; 0 < l_i < r_i \leq 10^5; 0 < h_i \leq 10^9$. The total amount of overlap in each case is at most $2 \cdot 10^6$.

Solution: An important observation is that the maximum overlap in each case is at most 2000000. Adding a new building implies updating a range $[l, r]$ with a new height h. This can be done in $\mathcal{O}(\log n)$ if we use lazy propagation. Lazy propagation is done by traversing the tree similar to when doing a query, but whenever we are at a node u with a range entirely included in $[l, r]$, we update this node and all its parents. But we also mark u as updated lazily. Whenever we visit this node again, either for querying or updating, we propagate the lazy value to its children.

For the query part, if the whole interval's height is less or greater than what we are asking, then we stop. Otherwise, we split the query into left and right segments, until condition satisfies. Since the maximum overlap is 2000000 over all cases, that is the maximum number of times this split can happen. Therefore total complexity is $\mathcal{O}(n \log n + 2000000)$. Since $n = 100000$, and $n \log n \simeq 2000000$, we can conclude it runs in $\mathcal{O}(n \log n)$.

Listing 9: Solution for SKYLINE

```
#include <iostream>
#include <vector>
using namespace std;
using VI = vector<int>;

struct ST {
  int n;
  VI mintree;
  VI maxtree;
  ST(int size) {
    n = 1;
    while (n < size) n *= 2;
    mintree = maxtree = VI(2 * n, 0);
  }

  // Returns the number of positions in [i, j) with height >= h.
  int query(int i, int j, int h) { return query(1, i, j, 0, n, h); }

  // Updates interval [i, j) with height h.
  void update(int i, int j, int h) { update(1, i, j, 0, n, h); }

 private:
  int query(int p, int i, int j, int L, int R, int h) {
    if (i >= R or j <= L) return 0;
    if (mintree[p] > h) return 0;
    if (maxtree[p] <= h) return min(j, R) - max(L, i);
    return query(p * 2, i, j, L, (L + R) / 2, h) +
           query(p * 2 + 1, i, j, (L + R) / 2, R, h);
  }

  void update(int p, int i, int j, int L, int R, int h) {
    if (i >= R or j <= L) return;
    maxtree[p] = max(maxtree[p], h);
    if (i <= L and j >= R) mintree[p] = max(mintree[p], h);
    else {
      update(p * 2, i, j, L, (L + R) / 2, h);
      update(p * 2 + 1, i, j, (L + R) / 2, R, h);
      mintree[p] = max(mintree[p], min(mintree[p * 2], mintree[p * 2 + 1]));
    }
  }
};

int main() {
  ios_base::sync_with_stdio(false);
  cin.tie(0);
  int T;
  cin >> T;
  const int maxn = 100001;
  while (T--) {
    int n;
    cin >> n;
    ST tree(maxn);
    int total = 0;
    while (n--) {
      int l, r, h;
      cin >> l >> r >> h;
      total += tree.query(l, r, h);
      tree.update(l, r, h);
    }
    cout << total << endl;
  }
}
```

6.2 Ordered Set

An ordered set is a tree data structure already built-in in C++. It is similar to the `std::set` data structure in STL, which is implemented as a red-black tree, but it also allows to query for the k-th smallest element in the set in $\mathcal{O}(\log n)$, and retrieve the position of an element x in $\mathcal{O}(\log n)$ as well [13].

In listing 10 there is a sample code on how to use this data structure.

Listing 10: Sample code for Ordered set

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
using namespace std;
template <typename T>
using ordered_set = tree<T, null_type, less<T>, rb_tree_tag,
    tree_order_statistics_node_update>;
int main() {
  ordered_set<int> set;
  set.insert(10);  // Insertion key "10" in O(log n)
  set.insert(100);
  set.insert(1000);
  set.order_of_key(10);  // Order of key "10" in O(log n). Returns 0.
  *set.find_by_order(1);  // Key at order "1" in O(log n). Returns 1000.
}
```

6.2.1 NWERC 2011 - Movie collection

Statement: There is a movie collection of n titles organized in a big stack. Each movie is identified by an id from 1 to n. Initially the stack contains the movies with ids $1, 2, ..., n$, in increasing order, where the movie with id 1 is the top-most box. There are m queries. Each query, asks for the position of movie a_i in the stack, that is, how many books does it have above itself. And then, that movie is removed from its position in the stack, and place in the top. Write a program that solves the given queries. Assume $1 \leq n, m \leq 10^5$.

Solution: Initially, all books from 1 to n are placed in the stack in order. If we insert those books in an ordered set with their ids as keys, they will be sorted as intended. If we maintain a counter starting at 0, and every time we put a book at the top of the stack we assign it the value of the counter, and decrease the counter by one, it will be the first element in the ordered set. We do so by creating a map from the original id (the one from 1 to n), to the new value of the key. For each request of a book id, we query its position and erase it in $\mathcal{O}(\log n)$, then insert it again with the new key, as well as updating the map of ids, in $\mathcal{O}(\log n)$. Overall solution runs in $\mathcal{O}(m \log n)$

Listing 11: Solution for Movie collection

```cpp
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
using namespace std;
template <typename T>
using ordered_set = tree<T, null_type, less<T>, rb_tree_tag,
    tree_order_statistics_node_update>;
int main() {
  ios_base::sync_with_stdio(false);
  cin.tie(0);
  int T; cin >> T;
  const int maxn = 100005;
  vector<int> ids(maxn);
  while (T--) {
    int n, m; cin >> n >> m;
    map<int,int> ids;
    ordered_set<int> s;
    int curr = 0;
    for (int i = 1; i <= n; ++i) {
      ids[i] = i;
      s.insert(i);
    }
    for (int i = 0; i < m; ++i) {
      int k; cin >> k;
      int real = ids[k];
      cout << (i?" ":"") << s.order_of_key(real);
      s.erase(real);
      s.insert(curr);
      ids[k] = curr;
      --curr;
    }
    cout << endl;
  }
}
```

6.2.2 Codeforces 100861 - Database Query Engine

Statement: You have to write an index list for a stock market securities. Stock market securities have three attributes:

- `code` - a non-empty empty string of Latin letters and digits, less than 30 characters in length;
- `id` - an integer (auto-incrementing key starting from 0);
- `reliability` - an integer in the range $[-2^{31}, 2^{31})$.

For each new security, the value of id is automatically assigned to the next available id. Initial value of reliability is 0. If a security is removed its id is never reused.

The following requests should be supported: issuing a new security, removing a security from the database, changing the value of reliability for an existing security, and finding the K-th security in the reliability index list.

The requests are of the following form:

- `ISSUE code`: add a new security code or output the security info, if it exists in the database
  - if the security `code` exists, output `EXISTS id reliability`
  - if the security `code` does not exist, output `CREATED id 0`
- `DELETE code`: remove the security `code` from the database
  - if the security code exists, output `OK id reliability`
  - if the security code does not exist, output `BAD REQUEST`
- `RELIABILITY code m`: increase the reliability of the given security by `m`. It is guaranteed that the value of reliability remains in the range $[-2^{31}, 2^{31})$.

  - if the security code exists, output `OK id new_reliability`
  - if the security code does not exist, output `BAD REQUEST`

- `FIND n`: find the n-th security in the list (starting from zero)
  - if the database is not empty, then output the n-th security in the reliability index list; if the list contains less than n items, output the last one in the form OK code id reliability
  - - if the database is empty, output `EMPTY`

Assume number of queries q is less than 10^5.

Solution: Maintain all data in an ordered set, where the key is the pair $\langle -reliability, id\rangle$. This way, keys are stored by non-increasing value of reliability first, and in case of tie, by increasing value of id. Maintain a map from code to such pair. Simulate all operations with the already builtin functions from `ordered_set`. Solution runs in $\mathcal{O}(q \log q)$.

Listing 12: Solution for Database Query Engine

```
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
using namespace std;
typedef long long ll;
typedef pair<ll,int> PII;  // <Rel, id>
template <typename T>
using ordered_set = tree<T, null_type, less<T>,
      rb_tree_tag, tree_order_statistics_node_update>;

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

  ordered_set<PII> s;
  unordered_map<string,PII> code_to_pii;
  vector<string> id_to_code;

  int curr_id = 0;
  int t; cin >> t;
  while (t--) {
    string token; cin >> token;
    if (token == "ISSUE") {
      string code; cin >> code;
      auto it = code_to_pii.find(code);
      if (it == code_to_pii.end()) {
        cout << "CREATED␣" << curr_id << "␣0\n";
        id_to_code.push_back(code);
        code_to_pii[code] = PII(0, curr_id);
        s.insert(PII(0, curr_id));
        ++curr_id;
      } else {
        cout << "EXISTS␣" << it->second.second << "␣" << it->second.first
```

```
             << "\n";
      }

    } else if (token == "DELETE") {
      string code; cin >> code;
      auto it = code_to_pii.find(code);
      if (it == code_to_pii.end()) {
        cout << "BAD␣REQUEST\n";
      } else {
        cout << "OK␣" << it->second.second << "␣" << it->second.first
             << "\n";
        PII key(-it->second.first, it->second.second);
        s.erase(key);
        code_to_pii.erase(it);
      }
    } else if (token == "RELIABILITY") {
      string code; cin >> code;
      ll m; cin >> m;
      auto it = code_to_pii.find(code);
      if (it == code_to_pii.end()) {
        cout << "BAD␣REQUEST\n";
      } else {
        PII key(-it->second.first, it->second.second);
        s.erase(key);
        key.first -= m;
        s.insert(key);
        it->second.first += m;
        cout << "OK␣" << it->second.second << "␣" << it->second.first
             << "\n";
      }

    } else if (token == "FIND") {
      int n; cin >> n;
      if (s.empty()) {
        cout << "EMPTY\n";
      } else {
        PII p = *s.find_by_order(min(int(s.size())-1, n));
        cout << "OK␣" << id_to_code[p.second] << "␣" << p.second << "␣"
             << -p.first << "\n";
      }
    }
  }
}
```

6.3 Treap

Treap (also known as a Cartesian tree) is a binary search tree where each key is given a random priority, and priorities in the treap satisfy the heap condition, that is, the parent will always have a bigger priority than its children. Whenever a new key is inserted, a random priority is assigned to it. These properties cause the treap to have logarithmic height with high probability, which allows many operations to run in $\mathcal{O}(\log n)$ on the average [6].

The treap allows two main operations:

- **Split**: Given a value x, split the treap into two treaps L and R such that all keys in L are not greater than x, and all keys in R are greater than x.

- **Merge**: Given two treaps L and R, where no key in R is greater than any key in L, return a single treap which is the union of L and R.

These operations run in $\mathcal{O}(\log n)$ on the average, and also allow us to insert new nodes on its corresponding position, and answering several queries.

6.3.1 2015-2016 Petrozavodsk Winter Training Camp, Nizhny Novgorod SU Contests - Youngling Tournament

Statement: There are n people standing in a row, i-th person has f_i force value. Then a tournament is executed in the following way: People stand in a row in order of non-increasing f_i. Then, first person fights against all other people united. He wins if his force is greater or equal than the sum of other people force. After that, he is removed and the tournament continues in the same format until there is only one person. He is one of the winners and the tournament finishes.

There are m queries. Each query assigns a new force value to i-th person that will persist until another query changes this value again. After each query, calculate the number of winners of a tournament with current force values.

Assume $1 \leq n \leq 10^5; 1 \leq f_i \leq 10^{12}; 0 \leq m \leq 5 \cdot 10^4$.

Solution: Let's build a Treap, where each node represents a person, and nodes are sorted by force value. In each node u, we will store the following information:

- `sum`: Sum of all forces in the subtree rooted at u.

- `best`: Largest difference between force of node $v \in \mathit{subtree}(u)$, and the sum of forces in nodes with key smaller than $\mathit{key}(v)$. That is, the maximum amount of force that can be placed before person v during the tournament, and still have v as a winner.

Build a Treap with people sorted by force. Each time i-th person is modified, remove that node, and insert it again with the new value of force, updating all attributes of modified nodes. For each query, we will do a traversal counting the number of winners.

In each query, the number of winners is actually small. To maximize the number of winners, the force values would be of the form $1, 1, 2, 4, 8, \ldots$.

Let F be the max value of force. Then there are $\mathcal{O}(\log F)$ winners in each tournament maximum.

When calculating the answer to each query, we traverse the treap from the root. When visiting node u, there were nodes with key smaller than $key(u)$ with total force t. If $best(u) \geq t$, then there is a node $v \in subtree(u)$ that is a winner, because it's force is greater or equal than sum of forces of people before him inside $subtree(u)$, and outside $subtree(u)$. So we keep traversing. Otherwise, there is no winner in $subtree(u)$, and we stop traversing.

There are $\mathcal{O}(\log F)$ winners, and height of treap is $\mathcal{O}(\log n)$ on average, this algorithm has average runtime complexity of $\mathcal{O}(m \log n \log F)$.

Listing 13: Solution for Youngling Tournament

```cpp
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <vector>
using namespace std;
using ll = long long;

struct Node {
  Node *left, *right, *parent;
  ll x, y, sum, best;
  Node (ll x) : x(x), y(rand()) {
    left = right = parent = nullptr;
    sum = best = x;
  }
};

inline ll sum(Node *t) {return (t?t->sum : 0);}
inline ll best(Node *t) {return (t?t->best : -1);}
inline void setp(Node *t, Node *p) { if(t) t->parent = p; }

void update(Node* t) {
  if (!t) return;
  t->sum = t->x + sum(t->left) + sum(t->right);
  t->best = max({t->x - sum(t->left), best(t->left),
               best(t->right) - sum(t->left) - t->x});
  setp(t->left, t);
  setp(t->right, t);
}

Node* merge(Node* t1, Node* t2) {
  if (t1 == nullptr) return t2;
  if (t2 == nullptr) return t1;
  if (t1->y >= t2->y) {
    t1->right = merge(t1->right, t2);
    update(t1);
    return t1;
  } else {
    t2->left = merge(t1, t2->left);
    update(t2);
    return t2;
  }
}
```

```cpp
// leaves on the left all nodes less than x.
pair<Node*, Node*> split(Node* t, ll x) {
  if (t == nullptr) return {nullptr, nullptr};
  if (t->x < x) {
    auto p = split(t->right, x);
    t->right = p.first;
    update(t);
    setp(t, nullptr);
    return {t, p.second};
  } else {
    auto p = split(t->left, x);
    t->left = p.second;
    update(t);
    setp(t, nullptr);
    return {p.first, t};
  }
}

Node* insert(Node* t, Node* n) {
  auto p = split(t, n->x);
  t = merge(p.first, n);
  t = merge(t, p.second);
  return t;
}

int count(Node*t, ll s = 0) {
  if (!t || t->best-s < 0) return 0;
  return (t->x >= sum(t->left) + s) + count(t->left, s) +
         count(t->right, s + sum(t->left) + t->x);
}

Node* update_node(Node* &root, Node* node, ll nx) {
  setp(node->left, nullptr);
  setp(node->right, nullptr);
  auto m = merge(node->left, node->right);
  auto p = node->parent;
  node->left = node->right = node->parent = nullptr;
  node->x = nx;
  update(node);
  if (p) {
    p->left == node ? p->left = m : p->right = m;
    if (m) m->parent = p;
    while (p) {
      update(p);
      p = p->parent;
    }
  } else {
    root = m;
  }
  return insert(root, node);
}

int main() {
  ios_base::sync_with_stdio(false);
  cin.tie(0);
  srand(time(0));
  int n; cin >> n;
  vector<Node*> nodes(n);
  Node* treap = nullptr;
  for (int i = 0; i < n; ++i) {
    ll x; cin >> x;
    Node* n = new Node(x);
    nodes[i] = n;
    treap = insert(treap, n);
  }
  // Update n-th treap with value x
  int m; cin >> m;
  cout << count(treap) << "\n";
  for (int i = 0; i < m; ++i) {
    int k; ll x;
    cin >> k >> x;
    Node* t = nodes[k-1];
    treap = update_node(treap, t, x);
    cout << count(treap) << "\n";
  }
}
```

6.4 Heavy-light decomposition

The heavy-light decomposition of a rooted tree is a method to decompose the vertices of the tree into vertex-disjoint chains, which allows some problems on trees to be solved with certain time complexity.

With heavy-light decomposition, each edge of the tree is either heavy or light. An edge $\langle u, v \rangle$, with u being the parent of v, is said to be heavy if and only if $subtree(v)$ is the biggest (measured with the number of nodes) among the children of u. In case of a tie, only the first children with biggest size is used. Two nodes connected with a heavy edge are said to be in the same chain.

With heavy-light decomposition of a tree with n nodes, the path from any node u to the root t will go through $\mathcal{O}(\log n)$ different chains [9]. As the next problem will show, this property is crucial to get a good efficiency for some operations.

6.4.1 Codeforces 587 - Duff in the Army

Statement: There are n cities with $n-1$ roads that form a tree. There are m people. i-th person has id i and lives in c_i city. There may be more than one person in a city, and cities without people.

There are q queries. Each query you are given integers v, u and a. You should write the a smallest IDs of all people living in cities lying in the path from u to v. If there are less than a, print all IDs. The IDs should be printed in increasing order. Assume $1 \le n, m, q \le 10^5; 1 \le a \le 10$.

Solution: Let's build a heavy-light decomposition on the tree of cities. For each chain, we build a Segment Tree with the a smallest ids in that range of cities.

When we query for cities u, v, we go from each node to $t = LCA(u, v)$, maintaining the list of a smallest ids seen so far. We maintain two pointers l, r, initially set to $u and v$ respectively. While l and r are not in the same chain, let w be the deepest node in the tree between l and r. We update the list of ids with the values in the range $[w, head(w)]$, and set w pointer to $parent(head(w))$. Once l and r are in the same chain, we do the same with the range $[deepest(l, r), highest(l, r)]$. We can keep the list of smallest ids with a priority queue.

For each query, we visit $\mathcal{O}(\log n)$ chains. For each chain, since a is very small, our queries take $\mathcal{O}(\log n)$ time. Overall, the solution runs in $\mathcal{O}(q \log^2 n)$

Listing 14: Solution for Duff in the Army

```
#include <algorithm>
#include <iostream>
#include <queue>
#include <vector>
using namespace std;
using VI = vector<int>;
using VVI = vector<VI>;
using QI = priority_queue<int>;

const int maxa = 10;

QI merge(QI l, QI r) {
  while (!r.empty()) {
    l.push(r.top());
    r.pop();
  }
  while (l.size() > maxa) {
    l.pop();
  }
  return l;
}

struct ST {
  vector<QI> tree;
  int size;
  ST() : size(0) {}
  ST(int n) {
    size = 1;
    while (size < n) {
      size *= 2;
    }
    tree.resize(size * 2);
  }
  QI query(int i, int j, int from, int to, int p = 1) {
    if (i <= from && j >= to) {
      return tree[p];
    }
    if (i >= to || j <= from) {
      return QI();
    }
    return merge(query(i, j, from, (from + to) / 2, p * 2),
                 query(i, j, (from + to) / 2, to, p * 2 + 1));
  }
  void update(int i, QI x) {
    tree[size + i] = x;
    int p = (size + i) / 2;
    while (p >= 1) {
      tree[p] = merge(tree[p * 2], tree[p * 2 + 1]);
      p /= 2;
    }
  }
};

VVI graph;
VI size, depth, parent, head, chainsize, chaindepth;

int dfs(int u, int d = 0, int p = -1) {
  parent[u] = p;
  depth[u] = d;
  size[u] = 1;
  for (int v : graph[u])
    if (v != p) {
      size[u] += dfs(v, d + 1, u);
    }
  return size[u];
}

void hld(int u, int id, int cdepth = 0, int p = -1) {
  head[u] = id;
  chaindepth[u] = cdepth;
  ++chainsize[id];
  int ind = -1;
  for (int v : graph[u])
    if (v != p) {
      if (ind == -1 || size[v] > size[ind]) ind = v;
    }
  if (ind != -1) hld(ind, id, cdepth + 1, u);
  for (int v : graph[u])
    if (v != p && v != ind) {
      hld(v, v, 0, u);
    }
}

int main() {
  ios_base::sync_with_stdio(false);
  int n, m, q;
  cin >> n >> m >> q;
  graph.resize(n);
  for (int i = 0; i < n - 1; ++i) {
    int from, to;
    cin >> from >> to;
    --from;
    --to;
    graph[from].push_back(to);
    graph[to].push_back(from);
  }
  vector<QI> cities(n);
  for (int i = 0; i < m; ++i) {
    int ci;
    cin >> ci;
    --ci;
    if (cities[ci].size() < maxa) {
      cities[ci].push(i + 1);
    }
  }
  // Build the HLD
  size = depth = parent = head = chainsize = chaindepth = VI(n, 0);
  dfs(0);
  hld(0, 0);
  // Build ST for each chain.
  vector<ST> sts(n);
  for (int i = 0; i < n; ++i)
    if (head[i] == i) {
      sts[i] = ST(chainsize[i]);
    }
  // Update elems
  for (int i = 0; i < n; ++i) {
    sts[head[i]].update(chaindepth[i], cities[i]);
  }
  // Now queries
  while (q--) {
    int from, to, k;
    cin >> from >> to >> k;
    --from;
    --to;

    QI acum;
    int ncfrom = head[from];
    int ncto = head[to];
    while (ncfrom != ncto) {
      if (depth[ncfrom] > depth[ncto]) {
        acum = merge(
            acum,
            sts[ncfrom].query(0, chaindepth[from] + 1, 0,
                              sts[ncfrom].size));
        from = parent[head[ncfrom]];
      } else {
        acum = merge(
            acum,
            sts[ncto].query(0, chaindepth[to] + 1, 0, sts[ncto].size));
        to = parent[head[ncto]];
      }
      ncfrom = head[from];
      ncto = head[to];
    }
    acum = merge(
        acum,
        sts[ncfrom].query(
          min(chaindepth[from], chaindepth[to]),
          max(chaindepth[from], chaindepth[to])+1,
          0,
          sts[ncfrom].size));

    while (int(acum.size()) > k) {
      acum.pop();
    }
    cout << acum.size();
    VI v;
    while (!acum.empty()) {
      v.push_back(acum.top());
      acum.pop();
    }
    for (int i = int(v.size() - 1); i >= 0; --i) {
      cout << "␣" << v[i];
    }
    cout << "\n";
  }
}
```

7 Sqrt Heuristics

Sqrt heuristics are methods to reduce the complexity of a problem and ending up with a complexity containing $\sqrt{n}$ multiplier or similar.

We present two important methods. Mo's algorithm, which is based on the rearrangement of queries, and Sqrt decomposition, which organizes the data in two layers.

7.1 Sqrt Decomposition

Sqrt decomposition is a way to build two-layer data structure using two approaches to answer queries to optimize the complexity.

7.1.1 ACM ICPC SEERC 2014 - Most Influential Pumpkin

Statement: You are given an array a with n numbers and q queries. Each query is represented by range $[l, r]$, which means each element of a in the range $[l, r]$ increases by one. After each query, print the value of the median in the array. Assume $1 \leq n, k \leq 60000$, n is odd, and values are between 0 and 10^9.

Solution: A good observation is that after each query, either the previous answer is the same, or increases by one. Each time we process a query, we check if the current median is m by counting the number of elements $e \leq m$. If there are more than $\lfloor m/2 \rfloor$ such elements, the median is still m, otherwise it is $m + 1$.

To count the number of elements $e \leq m$, we divide the array by buckets of size $\sqrt{n}$, and keep each bucket sorted. Whenever we have a query over range $[l, r]$, we update the buckets. Then, for each bucket, we'll be able to find the number of elements $e \leq m$ with binary search.

If the bucket is entirely inside the query range, then all numbers increase by one. The relative order of elements inside the bucket does not change, so we can mark that all elements are increased by one. When we check if m is the median, we actually count elements $e \leq m - 1$.

For buckets that intersect partially with the range of the query, we'll update the sorted elements in the following way: Store pair $\langle value, pos \rangle$ in the sorted buckets, where pos is the original position in the original array. Iterate over the bucket, and increase element e if $l <= pos <= r$. Now we have a sequence that is almost sorted, but there can be positions where it decreases value by 1 from previous element. Using auxiliary space to copy the elements and doing some swaps, this can be ordered in linear time.

Instead of doing binary search, we can store a pointer to the median in each bucket, and update this pointer as needed. This trick gets rid of the $\log n$ complexity from the binary search.

For each query, we update at most $\mathcal{O}(\sqrt{n})$ inside buckets in $\mathcal{O}(1)$, and $\mathcal{O}(1)$ buckets at the sides of the range, in $\mathcal{O}(\sqrt{n})$. To answer the query, we check from the stored pointer in each bucket, which results in $\mathcal{O}(\sqrt{n})$ total.

Overall, the solution has complexity $\mathcal{O}(q\sqrt{n}\log n)$.

Listing 15: Solution for Most Influential Pumpkin

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

const int MAXN = 60050;
const int INF = 1e9+100000000;
const int SQRT = sqrt(MAXN)+1;
using PII = pair<int,int>;
PII a[MAXN];
PII aux[MAXN];
int n, k;
// Bucket properties.
int add[SQRT];
int pmedian[SQRT];

void update_bucket(int l, int r, int b) {
  // Increase only affected elements.
  for (int i = b*SQRT; i < (b+1)*SQRT; ++i) {
    if (a[i].second >= l && a[i].second < r) ++a[i].first;
  }
  // Rearrange.
  int first = 0;
  int lastd = -1;
  for (int i = b*SQRT; i < (b+1)*SQRT; ++i) {
    aux[i] = a[i];
    if (aux[i].first < lastd) {
      swap(aux[first], aux[i]);
      ++first;
    } else if (aux[i].first > lastd) {
      first = i;
    }
    lastd = aux[i].first;
  }
  // Copy back.
  copy(aux + b*SQRT, aux + (b+1)*SQRT, a + b*SQRT);
}

int main() {
  ios_base::sync_with_stdio(false);
  cin.tie(0);
  while (cin >> n >> k) {
    if ((!n) || (!n)) continue;
    for (int i = 0; i < n; ++i) {
      cin >> a[i].first;
      a[i].second = i;
    }
    // Get starting median.
    copy(a, a+n, aux);
    nth_element(aux, aux + n/2, aux+n);
    int median = aux[n/2].first;
    // Fill remaining of last bucket with infinity for convenience.
    for (int i = n; (i%SQRT) != 0; ++i) a[i] = {INF, -1};
    for (int b = 0; b*SQRT < n; ++b) {
      pmedian[b] = b*SQRT;
      add[b] = 0;
      sort(a + b*SQRT, a + (b+1)*SQRT);
    }
    // Process queries.
    while (k--) {
      int l, r;
      cin >> l >> r;
      --l;
      int lb = l / SQRT;
      int rb = r / SQRT;
      // Update buckets affected entirely.
      for (int i = lb+1; i < rb; ++i) ++add[i];

      // Update sides.
      update_bucket(l, r, lb);
      if (lb != rb) { // Right side if needed
        update_bucket(l, r, rb);
      }
      // Check if median changes.
      int qty = 0;
      for (int b = 0; b * SQRT < n; ++b) {
        int x = median - add[b];
        int &p = pmedian[b];
        while (p > b*SQRT && a[p-1].first > x) --p;
        while (p < (b+1)*SQRT && a[p].first <= x) ++p;
        qty += p - b * SQRT;
      }
      if (qty <= n/2) ++median;
      cout << median << "\n";
    }
  }
}
```

7.2 Mo's Algorithm

Mo's algorithm is a method to solve efficiently problems with range queries that satisfy the following properties:

- Problem data falls in a range. E.g. an array.
- All queries must be known before starting. That is an offline algorithm.
- Solution of a query can be constructed incrementally adding or removing one element at a time.

The main idea of mo's algorithm is to reuse overlapping queries to answer them efficiently.

If the range of data has size n, we split the range by buckets of size $\sqrt{n}$, where i-th bucket represents the range $[i\sqrt{n}, (i+1)\sqrt{n})$.

We sort queries before answering them. If the queries spans for a range $[L, R]$, we sort all queries, first by index of bucket of position L. In case of tie, by position R.

During the processing of all queries, we maintain two pointers, l and r, and keep the current answer to the range $[l, r]$. We initially set those pointers to 0, and compute the current answer for range $[0, 0]$. Now, for each query, we move r and l pointers, one position at a time, and recalculating the new answer for range $[l, r]$, until $l = L$, and $r = R$. Once $l = L$ and $r = R$, the current answer is the answer to that query. We repeat this process until we have processed all queries.

Let's assume there are q queries, and each operation of adding/removing an element takes $\mathcal{O}(t)$ time. The way queries are sorted, for each bucket, the pointer r can only move to the right. The only time it moves to the left, is when moving to a new bucket. The total cost of moving the right pointer is $\mathcal{O}(tn\sqrt{n})$.

For the left pointer l, between queries in the same bucket, it will do at most $\mathcal{O}(\sqrt{n})$ steps. And the total sum of steps between queries of different buckets is $\mathcal{O}(n)$, because queries are sorted first by bucket of L.

Overall, the time complexity is $\mathcal{O}(tn\sqrt{n} + tq\sqrt{n}) = \mathcal{O}(t(n + q)\sqrt{n})$.

7.2.1 Codechef - Daddy and Chocolates

Statement: There is an array with N elements, from 0 to 1000000, and you are given D queries. For each query over the range $[l, r]$, you have to compute the following:

For each different value x in $[l, r]$, if element x appears K times in the subarray, add (K^2x) to the answer of this query.

Solution: Use Mo's algorithm. Create an array of size 1000001 to keep track of how many of each element there are in current interval $[l, r]$. Each time we add/remove an element, we update the counter on that element, and update the current cost in $\mathcal{O}(1)$. Overall complexity is $\mathcal{O}((N + D)\sqrt{N})$.

Listing 16: Solution for Daddy and Chocolates

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
using VI = vector<int>;
using ll = long long;

const int MAXN = 200000;
const int MAXC = 1000000;
const int BUCKET_SIZE = 450;
struct Query {
  int from, to;
  int id;
  Query() {};
  bool operator<(const Query &other) {
    int bucket = from/BUCKET_SIZE;
    int other_bucket = other.from/BUCKET_SIZE;
    return bucket != other_bucket ? bucket < other_bucket : to < other.to;
  }
};

int main() {
  ios_base::sync_with_stdio(false);
  cin.tie(0);
  int n, d;
  cin >> n >> d;
  VI costs(n);
  for (auto &c : costs) cin >> c;
  vector<Query> queries(d);
  for (int i = 0; i < d; ++i) {
    cin >> queries[i].from >> queries[i].to;
    --queries[i].from;
    --queries[i].to;
    queries[i].id = i;
  }

  sort(queries.begin(), queries.end());
  VI freq(MAXC+1, 0);
  vector<ll> answers(d);
  ll current_ans = 0;
  int l = 0, r = 0;
  current_ans = costs[0];
  freq[costs[0]] = 1;

  for (auto &q : queries) {
    while (r < q.to) {
      ++r;
      current_ans += (2*freq[costs[r]]+1)*costs[r];
      ++freq[costs[r]];
    }
    while (r > q.to) {
      current_ans += (-2*freq[costs[r]]+1)*costs[r];
      --freq[costs[r]];
      --r;
    }
    while (l < q.from) {
      current_ans += (-2*freq[costs[l]]+1)*costs[l];
      --freq[costs[l]];
      ++l;
    }
    while (l > q.from) {
      --l;
      current_ans += (2*freq[costs[l]]+1)*costs[l];
      ++freq[costs[l]];
    }
    answers[q.id] = current_ans;
  }
  for (auto ans : answers) {
    cout << ans << "\n";
  }
}
```

7.2.2 Codeforces 100962 - Frank Sinatra

Statement: You are given a bidirectional tree T with n vertices. Each edge has a non-negative cost x_i. There are q queries each represented by integers a and b. In j-th query, print the smallest integer not present in the shortest path from node a_j to node b_j. Assume $0 \leq x_i \leq 10^9$, $2 \leq n \leq 10^5$, $1 \leq q \leq 10^5$.

Solution: To use Mo's algorithm, first we have to see how to represent the queries as ranges in the tree. We create an array a of $2n$ elements, and array *pos* of n elements. We arbitrarily root the tree T and do a DFS traversal from the root in the following way:

Everytime we see a node u for first time, we append the tuple $\langle u, p_u \rangle$ to a, where p_u is the cost of the edge from u to its parent, and set pos_u to the position in a where this tuple is inserted. Once we have visited all subtree rooted at u, we append $\langle u, p_u \rangle$ again.

Now if we want to calculate the cost of the shortest path from u to v, assuming $pos_u < pos_v$, we take the subarray $a(pos_u, pos_v]$, and for each $\langle u, p_u \rangle \in a(pos_u, pos_v]$, if u appears exactly once, add p_u to the cost of the path.

The reason behind this is the following: If the node u did not appear in the subarray, it is clearly not in the path. If node u appears twice, because otherwise node v would appear between first and second occurrence.

Now for each query, we can work with intervals on array a using Mo's algorithm. The remaining part is, to keep track of the minimum number not in the interval. We can do this using Sqrt decomposition. Since there are n nodes, there are at most $n-1$ different numbers in the path of a query, so the answer to a query is at most $n-1$.

We maintain an array of occurrences of size n to keep track of number of occurrences of each value. This array is split into buckets of size $\sqrt{n}$. Each bucket will contain how many different values have appeared in our current range. Each element added or removed can be updated in $\mathcal{O}(1)$. When we want to obtain the minimum number not seen of a range, we iterate over all buckets until we find the first bucket containing less than $\sqrt{n}$ different elements. Then, we iterate all positions in this bucket to find the smallest value that is not present, and that is the answer.

For the first iteration, there are $\mathcal{O}(\sqrt{n})$ buckets that we check in $\mathcal{O}(1)$ time each. Inside a bucket, there are $\mathcal{O}(\sqrt{n})$ positions that we check in $\mathcal{O}(1)$

each as well. Overall, we find the smallest integer not present in $\mathcal{O}(\sqrt{n})$. There are $\mathcal{O}(q)$ queries, so the total cost of this part is $\mathcal{O}(q\sqrt{n})$. Since the time complexity of Mo's algorithm is already $\mathcal{O}((n+q)\sqrt{n})$, this doesn't increase time complexity.

Listing 17: Solution for Frank Sinatra

```cpp
#include <algorithm>
#include <iostream>
#include <vector>
#include <tuple>
using namespace std;
using ll = long long;
using VI = vector<int>;
using VVI = vector<VI>;
using PII = pair<int,int>;
using VPII = vector<PII>;
using VVPII = vector<VPII>;

VVPII G;
int act = 0;
VPII guant;
VI pos;
void dfs(int u, int x = 1e6, int p = -1) {
  pos[u] = act;
  guant[act++] = {u, x};
  for (auto e : G[u]) if (e.first != p) {
    dfs(e.first, e.second, u);
  }
  guant[act++] = {u, x};
}

int BUCKET_SIZE = 450;  // SQRT
struct Query {
  int from, to, id;
  Query(int from, int to, int id) : from(from), to(to), id(id) {}
  Query(){}
  bool operator<(const Query &other) {
    int bucket = from/BUCKET_SIZE;
    int other_bucket = other.from/BUCKET_SIZE;
    return bucket != other_bucket ? bucket < other_bucket : to < other.to;
  }
};
using VQ = vector<Query>;
using VVQ = vector<VQ>;

struct Smallest {
  int n;
  int p;
  VVI buckets;
  VI each_bucket;
  VI i_to_bucket;
  vector<bool> seen;
  Smallest(int n, int p) : n(n), p(p) {
    buckets.resize(p, VI(p, 0));
    each_bucket.resize(p, 0);
    i_to_bucket.resize(p*p);
    seen.resize(n, false);
    int ith = 0;
    for (int i = 0; i < p; ++i) {
      for (int j = 0; j < p; ++j) {
        i_to_bucket[ith++] = i;
      }
    }
  }
  void set(int node, int value) {
    if (value >= n) return;
    if (seen[node]) {
      update(value, -1);
    } else {
      update(value, 1);
    }
    seen[node] = !seen[node];
  }
  void update(int value, int dif) {
    if (value >= n) return;
    int b = i_to_bucket[value];
    int c = value%p;
    if (!buckets[b][c]) ++each_bucket[b];
    buckets[b][c] += dif;
    if (!buckets[b][c]) --each_bucket[b];
  }
  int get() {
    int b = 0;
    while (each_bucket[b] == p) ++b;
    int c = 0;
    while (buckets[b][c]) ++c;
    return p*b+c;
  }
};

int main() {
  ios_base::sync_with_stdio(false);
  cin.tie(0);
  int n, q; cin >> n >> q;
  G.resize(n);
  for (int i = 0; i < n-1; ++i) {
    int from, to, k; cin >> from >> to >> k; --from; --to;
    G[from].push_back({to,k});
    G[to].push_back({from,k});
  }
  int m = 2*n;
  // DFS traverse
  guant.resize(m);
  pos.resize(n);
  dfs(0);

  VQ queries(q);
  for (int i = 0; i < q; ++i) {
    int from, to; cin >> from >> to;
    tie(from, to) = minmax(pos[from-1], pos[to-1]);
    queries[i] = {from, to, i};
  }
  sort(queries.begin(), queries.end());
  VI ans(q);
  Smallest smallest(m, BUCKET_SIZE);
  int start = 0;
  int end = 0;
  smallest.set(guant[0].first, guant[0].second);

  for (const Query &q : queries) {
    while (end < q.to) {
      ++end;
      smallest.set(guant[end].first, guant[end].second);
    }
    while (end > q.to) {
      smallest.set(guant[end].first, guant[end].second);
      --end;
    }
    while (start < q.from) {
      ++start; smallest.set(guant[start].first, guant[start].second);
    }
    while (start > q.from) {
      smallest.set(guant[start].first, guant[start].second);
      --start;
    }
    ans[q.id] = smallest.get();
  }
  for (int x : ans) cout << x << "\n";
}
```

8 Strings

8.1 Z-Algorithm

Given a string S of length n, the Z Algorithm produces an array Z where $Z[i]$ is the length of the longest substring starting from $S[i]$ which is also a prefix of S. This array Z can be build in linear time [16].

8.1.1 Codeforces 126 - Password

Statement: Given a string S, find longest substring t that is a prefix of S, a suffix of S, and an interior substring of S (not a prefix nor a suffix).

Solution: For any $1 \leq k \leq Z[i]$, $S[i, i+k)$ is an interior substring of S unless $k = n - i$, in which case it is a suffix of S. For each suffix of S of length l, if there is an interior substring of length $k \leq l$, then $t = S[0, l)$ is a substring that fulfills the conditions, because if $k > l$, then we can remove the $k - l$ trailing characters from the interior substring and it will be equal to the suffix and the prefix.

With this in mind, we just iterate from left to right, and keep track of the longest inside substring we have seen so far from the value of $Z[i]$, and each time we find a suffix (that is, when $Z[i] = n - i$), we update the best answer so far to keep the longest substring t we are looking for.

Listing 18: Solution for Password

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

int main() {
  string s;
  cin >> s;
  int n = s.size();
  vector<int> z(n);
  int L = 0, R = 0;
  for (int i = 1; i < n; i++) {
    if (i > R) {
      L = R = i;
      while (R < n && s[R-L] == s[R]) R++;
      z[i] = R-L; R--;
    } else {
      int k = i-L;
      if (z[k] < R-i+1) z[i] = z[k];
      else {
        L = i;
        while (R < n && s[R-L] == s[R]) R++;
        z[i] = R-L; R--;
      }
    }
  }
  int maxprev = 0;
  int ans = 0;
  for (int i = 1; i < n; ++i) {
    int l = z[i];
    if (l == n-i) {
      if (maxprev >= l) ans = max(ans, min(maxprev, l));
      --l;
    }
    maxprev = max(maxprev, l);
  }
  if (ans > 0) cout << s.substr(0, ans) << endl;
  else cout << "Just a legend" << endl;
}
```

8.2 KMP

The naive algorithm for matching a string pattern P against a string S is to try to match the pattern P from each position of S.

KMP (or Knuth-Morris-Pratt) algorithm is a string matching algorithm that matches a pattern P against a string S in a much more clever way than the naive algorithm. Whenever there is a mismatch, instead of starting again from the beginning, it makes use of the number of previous characters of P matched to know from which position of P to continue. While the naive algorithm runs in quadratic time, KMP runs in linear time with respect to the size of P and S [17, 32, p.285-289].

8.2.1 UVA 445 - Periodic String

Statement: A string has period k if it can be formed by concatenating one or more repetitions of another string of length k. Given a string S, determine its smallest period k.

Solution: Let $T = S + S$. Just match the pattern string S against the string T. Then, the first position $p > 0$ where pattern S matches against T is the value of the minimum k.

The reason for this is the following: Let $n = |S|$. Then $S = S[k, n-1] + S[0, k-1]$, because we are cutting the first period and appending it to the end.

In T, any substring of length n is equivalent to $S[i, n-1] + S[0, i-1]$ for some i. So finding a period means finding a substring of size n in T. Position 0 is an exception, because it would be always equal to S.

Listing 19: Solution for Periodic String

```
#include <iostream>
#include <vector>
using namespace std;
using VI = vector<int>;

string P, T;
VI B;

void precalc() {
  int i = 0, j = -1, m = P.size();
  B[0] = -1;
  while (i < m) {
    while (j >= 0 and P[i] != P[j]) j = B[j];
    ++i; ++j;
    B[i] = j;
  }
}

int match() {
  int i = 0, j = 0, n = T.size(), m = P.size();
  while (i < n) {
    while (j >= 0 and P[j] != T[i]) j = B[j];
    ++i; ++j;
    if (j == m) {
      if (i-m > 0) return i-m;
      j = B[j];
    }
  }
  return -1;
}

int main() {
  int cases; cin >> cases;
  while (cases--) {
    cin >> P;
    T = P+P;
    B = VI(P.size()+1, 0);
    precalc();
    cout << match() << endl;
    if (cases) cout << endl;
  }
}
```

8.2.2 UVA 11022 - String Factoring

Statement: Factoring of a string S consists of taking two or more identical substring A that are placed consecutively, and replacing them with a single occurrence of A between parenthesis, raised to the number of its occurrences. E.g. the string DOODOO can be factored as (DOO)^2 , but also as (D(O)^2)^2. The weight of a factoring is defined as the number of characters it contains excluding parenthesis and numbers. Given a string S, compute the value of the minimum possible weight of a factoring. Assume that S if formed by capital case english letters only, and that it has length up to 80.

Solution: We use dynamic programming with KMP. Let $f(i, j)$ be the minimum weight after factoring $S[i, j]$. Base case is $f(i, i) = 1$, as there is only one letter and we cannot factor it further. Otherwise, either $S[i, j]$ can be expressed as A^k, for $k > 1$, or it is expressed as $A_1^{n_1} + ... + A_k^{n_k}$.

For the first case, we calculate such A, as we have seen in the previous solution for UVA 455, and the answer will be $f(i, i + |A| - 1)$.

For the second case, we try to split the string on each character and calculate the minimum for each part. Therefore,

$$f(i, j) = \min_{i \le k < j} f(i, k) + f(k + 1, j).$$

Listing 20: Solution for String Factoring

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

const int maxn = 81;
int B[maxn];
int memo[maxn][maxn];
string a, b, s;

void precalc() {
  int m = b.size();
  int i = 0, j = -1;
  B[0] = j;
  while (i < m) {
    while (j >= 0 and b[i] != b[j]) j = B[j];
    ++i; ++j;
    B[i] = j;
  }
}

int periodic_length() {
  int res = 0;
  int n = a.size(), m = b.size();
  int i = 0, j = 0;
  while (i < n) {
    while (j >= 0 and a[i] != b[j]) j = B[j];
    ++i; ++j;
    if (j == m) {
      if (i - m > 0) return i-m;
      ++res;
      j = B[j];
    }
  }
  return -1;
}

int f(int, int);

int kmp(int i, int j) {
  b = s.substr(i, j-i+1);
  a = b+b;
  precalc();

  int len = periodic_length();
  if (len == (j-i+1)) {
    return len;
  }
  return f(i, i+len-1);
}

// Minimum weight of factoring S[i, j].
int f(int i, int j) {
  if (i == j) return 1;
  if (memo[i][j] != -1) return memo[i][j];
  int &ans = memo[i][j];

  ans = kmp(i, j);
  for (int k = i; k < j; ++k) {
    ans = min(ans, f(i, k) + f(k+1, j));
  }
  return ans;
}

int main() {
  while (getline(cin,s) and s != "*") {
    memset(memo, -1, sizeof memo);
    cout << f(0, s.size()-1) << endl;
  }
}
```

8.3 Aho-corasick

Aho-corasick is an algorithm to match a set of patters in a string. It works by constructing the Deterministic Finite Automaton (DFA) of the set of patterns to find, and running it against the string.

8.3.1 ACM ICPC SWERC 2016 - Passwords

Statement: Given integers A, B, and a set of blacklisted words W made up of lowercase letters, count the number of different passwords that satisfy the following restrictions:

- The length of the password is between A and B, inclusive.
- The password is made of lowercase letters, uppercase letters and digits.
- It contains at least one lowercase letter, one uppercase letter, and one digit.
- The password does not contain any words from W as a substring.
- Uppercase letters can match its corresponding lowercase letter when checking the previous condition. For instance, “aBcD” is considered to contain the word “bc”.
- In the same way, ’0’ counts as an ’o’, ’1’ counts as an ’i’, ’3’ as an ’e’, ’5’ as an ’s’, and ’7’ as a ’t’.

Since the result can be quite big, output the answer modulo 1000003.

Solution: The solution is to build the automaton with the blacklisted words, and compute the solution with dynamic programming, where the state has five dimensions: $\langle node, pos, lower, upper, digit\rangle$, where:

- node: The urrent node in the DFA.
- pos: How many letters we have added so far.
- lower: Bool indicating if we have already added a lowercase letter.
- upper: Bool indicating if we have already added an uppercase letter.
- digit: Bool indicating if we have already added a digit.

If, at any time, we get into an accepting node, the answer for that state is 0, because it contains a blacklisted word. Otherwise, if $A \le pos \le B$, and all boolean flags are set to true, this state is from a valid password. We increase the answer by one in this state, and proceed adding more characters.

At each state, we try adding each lowercase letter, uppercase letter and digit, and go on traversing to the next node in the automaton.

Listing 21: Solution for Passwords

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

int numnodes = 0;
struct Node {
  int size;
  int ind;
  Node *fail;
  bool accept;
  map<char, Node*> next;
  Node(int size = 0)
      : size(size), ind(numnodes++), fail(nullptr), accept(false) {}
};
using P = pair<Node*, Node*>;
using MCP = map<char, Node*>;
Node* root;

inline void init() { root = new Node(); }

Node* add(string& s, int c=0, Node* p=root) {
  if (p == nullptr) p = new Node(c);
  if (c == int(s.size())) p->accept = true;
  else {
    if (!p->next.count(s[c])) p->next[s[c]] = nullptr;
    p->next[s[c]] = add(s, c + 1, p->next[s[c]]);
  }
  return p;
}

void fill_fail_output() {
  queue<pair<char, P>> Q;
  for (auto &it : root->next) Q.push({it.first, {root, it.second}});
  while (!Q.empty()) {
    Node *pare = Q.front().second.first;
    Node *fill = Q.front().second.second;
    char c = Q.front().first;
    Q.pop();
    while (pare != root && !pare->fail->next.count(c)) pare = pare->fail;
    if (pare == root)
      fill->fail = root;
    else
      fill->fail = pare->fail->next[c];
    fill->accept |= fill->fail->accept;
    for (auto &it : fill->next) Q.push({it.first, {fill, it.second}});
  }
}

const int maxnodes = 50*20 + 10;
const int mod = 1e6+3;
int memo[maxnodes][20][2][2][2];
int A, B;

Node* NextNode(Node *node, char c) {
  if (c == '0') c = 'o';
  if (c == '1') c = 'i';
  if (c == '3') c = 'e';
  if (c == '5') c = 's';
  if (c == '7') c = 't';
  c = tolower(c);
  while (true) {
    if (node->next.count(c)) return node->next[c];
    if (node == root) return root;
    node = node->fail;
  }
}

int calc(Node *node, int pos, bool lower, bool upper, bool digit) {
  if (node->accept || pos > B) return 0;
  int &ans = memo[node->ind][pos][lower][upper][digit];
  if (ans != -1) return ans;
  ans = 0;
  if (pos >= A && lower && upper && digit) ans += 1;
  for (char c='a'; c<='z'; ++c)
    ans += calc(NextNode(node, c), pos+1, true, upper, digit);
  for (char c='A'; c<='Z'; ++c)
    ans += calc(NextNode(node, c), pos+1, lower, true, digit);
  for (char c='0'; c<='9'; ++c)
    ans += calc(NextNode(node, c), pos+1, lower, upper, true);
  return ans %= mod;
}

int main() {
  ios_base::sync_with_stdio(false);
  cin.tie(0);
  init();
  cin >> A >> B;
  int n; cin >> n;
  for (int i = 0; i < n; ++i) {
    string word;
    cin >> word;
    add(word);
  }
  fill_fail_output();
  memset(memo, -1, sizeof memo);
  cout << calc(root, 0, 0, 0, 0) << endl;
}
```

8.4 Suffix Tree

A suffix tree of a string S is a data structure similar to a trie, where edges contain characters, and every suffix of S appear as a path from the root to a leaf. Subpaths containing nodes with only one children are compressed as a single edge representing a substring of S. Ukkonen's algorithm constructs the suffix tree of a string S in linear-time [21].

8.4.1 HackerRank - Build a String

Statement: Greg wants to build a string S of length N. Starting with an empty string, he can perform 2 operations:

- Add a character to the end of S for A dollars.
- Copy any substring of S, and then add it to the end of S for B dollars.

Calculate the minimum amount of money that Greg needs to build S.

Solution: We do dynamic programming, with $f(i)$ = minimum cost to build the string up to position i. We have two options:

- From position i, we can pay A and reach position $i+1$ with cost $f(i)+A$.
- From position i, we can pay B and add the longest substring t in $S[0, i)$ that is also a prefix of $S[0, N)$, and reach position $i + |t|$ with cost $f(i) + B$.

The required answer is $f(N)$.

To efficiently find such substring t when paying for B, we build the suffix tree of S. Then we precalculate, for each node, the maximum depth of all its children from the root. The bigger the depth, the longer the suffix, and therefore, the earlier this suffix starts in S. Then, to find t, we traverse the suffix tree along the edges with the characters we consume from $S[i, N)$, one at a time. We stop when there is no edge to traverse that matches the character, or if the character in the edge doesn't correspond to any position before i.

Listing 22: Solution for Build a String

```
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <iostream>
#include <map>
#include <vector>
using namespace std;
using VI = vector<int>;
const int inf = 1e9;

typedef int ttype;
struct TR { // edge
  int t0, t1; // text on the edge: T[t0..t1]
  int node;   // node that the edge is pointing
  TR() {}
  TR(int _t0, int _t1, int _node) : t0(_t0), t1(_t1), node(_node) {}
};
struct ND { // node
  int link;              // suffix link, internally used by suffixtree()
  map<ttype, TR> edges; // edges to the children
  int dollars;           // generalized: unique dollars in that subtree
  int deepest;           // distance to the deepest dollar in that subtree.
};
const int MAXN = 30050; // maximum text length
int N, M;              // text length, number of nodes
ttype T[MAXN + 17];    // text
ND tree[2*MAXN + 17]; // suffix tree

bool test_and_split(int s, int k, int p, ttype t, int& r) {
  r = s;
  if (p < k) return tree[s].edges.count(t) != 0;
  TR tr = tree[s].edges[T[k]];
  int sp = tr.node, kp = tr.t0, pp = tr.t1;
  if (t == T[kp + p - k + 1]) return true;
  r = M++;
  tree[s].edges[T[kp]] = TR(kp, kp + p - k, r);
  tree[r].edges[T[kp + p - k + 1]] = TR(kp + p - k + 1, pp, sp);
  return false;
}

void canonize(int s, int k, int p, int& ss, int& kk){
  if (p < k) { ss = s; kk = k; return; }
  TR tr = tree[s].edges[T[k]];
  int sp = tr.node, kp = tr.t0, pp = tr.t1;
  while (pp - kp <= p - k) {
    k += pp - kp + 1; s = sp;
    if (k <= p) {
      tr = tree[s].edges[T[k]];
      sp = tr.node; kp = tr.t0; pp = tr.t1;
    }
  }
  ss = s; kk = k;
}

void update(int& s, int& k, int i) {
  int oldr = 0, r;
  while (not test_and_split(s, k, i - 1, T[i], r)) {
    tree[r].edges[T[i]] = TR(i, MAXN, M++);
    if (oldr != 0) tree[oldr].link = r;
    oldr = r;
    canonize(tree[s].link, k, i - 1, s, k);
  }
  if (oldr != 0) tree[oldr].link = s;
}

int sts, stk;
void init() {
  for (int i = 0; i < M; ++i) {
    tree[i].link = 0; tree[i].edges.clear(); }
  M = 2; N = 0; tree[0].link = 1; sts = stk = 0;
}
void add(int c) {
  T[N] = c;
  tree[1].edges[T[N]] = TR(N, N, 0);
  update(sts, stk, N);
  canonize(sts, stk, N++, sts, stk);
}


void tallagespa() { //poda el generalized suffix tree
  vector<int> next(N + 1);
  next[N] = N;
  for (int i = N - 1; i >= 0; --i) {
    if (T[i] < 0) next[i] = i;
    else next[i] = next[i + 1];
  }
  for (int i = 0; i < M; ++i) {
    if (i == 1) continue;
    for (auto it = tree[i].edges.begin(); it != tree[i].edges.end(); ++it) {
      TR &tr = it->second;
      if (next[tr.t0] <= tr.t1) {
        tr.t1 = next[tr.t0];
        tree[tr.node].edges.clear();
      }
    }
  }
}
void compute_depth(int n = 0, int d = 0) {
  tree[n].deepest = -1;
  for (auto& it : tree[n].edges) {
    TR tr = it.second;
    int nd = tr.t1-tr.t0+1;
    compute_depth(tr.node, d + nd);
    tree[n].deepest = max(tree[n].deepest, tree[tr.node].deepest + nd);
  }
}

int A,B;
string S;
int maxmatch(int from) {
  int n = S.size();
  int node = 0;
  int p = from;
  int rem = n-p;
  while (p < n) {
    if (tree[node].edges.count(S[p]) == 0) return p-from;
    TR tr = tree[node].edges[S[p]];
    int deep = tree[tr.node].deepest + (tr.t1 - tr.t0 + 1);
    if (deep <= rem) return p-from;
    int m = min(n - p, min(deep - rem, tr.t1 - tr.t0 + 1));
    for (int i = 0; i < m; ++i)
      if (T[tr.t0+i] != S[p+i]) return (p+i)-from;
    if (m == tr.t1-tr.t0+1) node = tr.node;
    else return (p+m)-from;
    p += m;
  }
  return p-from;
}

int solve() {
  VI memo(S.size()+1, inf);
  memo[0] = 0;
  for (int i = 0; i < int(S.size()); ++i) {
    memo[i+1] = min(memo[i+1], memo[i] + A);
    int l = maxmatch(i);
    if (l) {
      memo[i+l] = min(memo[i+l], memo[i]+B);
    }
  }
  return memo[S.size()];
}

int main() {
  ios::sync_with_stdio(false);
  cin.tie(0);
  int T; cin >> T;
  while (T--) {
    cin >> A >> A >> B >> S;
    init();
    for (char c : S) add(c);
    add(-1);
    tallagespa();
    compute_depth();
    cout << solve() << endl;
  }
}
```

8.5 Eertree

The eertree [20], also known as palindromic tree, is a fairly recent data structure that provides fast access to all substrings of a string that are a palindrome. The eertree is an online data structure that works in both linear time and space.

The data structure maintains two trees, one for palindromes with odd length, and the other for palindromes with even length. Initially, the root for even lengths is the palindrome of length 0. The root for odd lengths is a fictive palindrome of length -1, just for convenience. Then, each edge is marked with a character. An edge with character c from node u to node v means that a palindrome v can be constructed by adding character c to both sides of a palindrome represented by u. The data structure also has suffix links, similar to the ones in a suffix tree. A suffix link from node u goes to a node w such that the palindrome of node w is the largest palindrome that is also a proper suffix of the palindrome of node u.

The construction of the eertree of a string S of length n can be done in $\mathcal{O}(n)$ time and space complexity.

8.5.1 Andrew Stankevich Contest 14 - Palindromes

Statement: Given a string s, count the number of palindromes that are a substring of s.

Solution: This is a trivial problem using an eertree. Just build the eertree of s, and return the number of nodes - 2.

Listing 23: Solution for Palindromes

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

using ll = long long;

struct palindromic_tree {
  struct node {
    node() : len(0), sufflink(1), num(0), cant(0) {
      memset(next, 0, sizeof(next));
    }
    // Edges, length of P, node index of longest palindromic suffix,
    // number of palindrome suffixes.
    int next[26], len, sufflink, num;
    ll cant;
  };
  char s[200005];
  // Length of s, num nodes, longest palindromic suffix.
  int len, num, suff;
  vector<node> tree;
  palindromic_tree() : num(2), suff(2) {}

  void initTree() {
    len = strlen(s);
    tree = vector<node>(len + 3);
    tree[1].len = -1; tree[1].sufflink = 1;
    tree[2].len = 0; tree[2].sufflink = 1;
    for (int i = 0; i < len; i++) addLetter(i);
    for (int i = len + 2; i >= 0; --i)
      tree[tree[i].sufflink].cant += tree[i].cant;
  }

  bool addLetter(int pos) {
    int cur = suff, curlen = 0, let = s[pos] - 'a';
    while (true) {
      curlen = tree[cur].len;
      if (pos - 1 - curlen >= 0 && s[pos - 1 - curlen] == s[pos]) break;
      cur = tree[cur].sufflink;
    }
    if (tree[cur].next[let]) {
      suff = tree[cur].next[let];
      ++tree[suff].cant;
      return false;
    }
    num++;
    suff = num;
    tree[num].len = tree[cur].len + 2;
    tree[num].cant = 1;
    tree[cur].next[let] = num;
    if (tree[num].len == 1) {
      tree[num].sufflink = 2;
      tree[num].num = 1;
      return true;
    }
    while (true) {
      cur = tree[cur].sufflink;
      curlen = tree[cur].len;
      if (pos - 1 - curlen >= 0 && s[pos - 1 - curlen] == s[pos]) {
        tree[num].sufflink = tree[cur].next[let];
        break;
      }
    }
    tree[num].num = 1 + tree[tree[num].sufflink].num;
    return true;
  }
};

int main() {
  freopen("palindromes.in", "r", stdin);
  freopen("palindromes.out", "w", stdout);
  palindromic_tree tree;
  scanf("%s", tree.s);
  tree.initTree();
  cout << tree.num - 2 << endl;
}
```

8.5.2 Timus 2060 - Subpalindrome pairs

Statement: Given a string s of lowercase Latin letters and size n, calculate the number of triplets (i, j, k) such that $i \leq j < k$, $s[i, j]$ is a palindrome, and $s[j + 1, k]$ is a palindrome.

Solution: First, we calculate, for each position p, the values $end(p)$ and $begin(p)$, which are the number of palindromes that end at position p and the number of palindromes that begin at position p, respectively. Then, we can calculate the answer as

$$\sum_{j=0}^{n-2} end(j) \times start(j + 1).$$

To calculate $end(p)$ for each p, we use an eertree. We add the letters one by one to it. At the end of each step, the eertree provides us the longest suffix palindrome, that is, the longest palindrome ending at position p. If we traversed the suffix links from that node, we would visit all the nodes that are a palindrome suffix of the longest suffix palindrome at this point. The value of $end(p)$ is the length of the path from a longest suffix palindrome node to the root using suffix links. We have this number calculated for each node, and after processing the i-th character, we set $end(p)$ to such value from the node of the longest suffix palindrome. Computing $begin(p)$ can be done in the same way with the reversed string. Overall, the code runs in linear time and space complexity.

Listing 24: Solution for Subpalindrome pairs

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

struct palindromic_tree {
  struct node {
    node() : len(0), sufflink(1), num(0) {
    }
    map<int,int> next;
    int len, sufflink, num;
  };
  string s;
  // Num of palindromes ending at position i.
  vector<int> count;
  // Length of s, num nodes, longest palindromic suffix.
  int len, num, suff;
  vector<node> tree;
  palindromic_tree() : num(2), suff(2) {}

  void initTree() {
    len = s.size();
    count.resize(len);
    tree = vector<node>(len + 3);
    tree[1].len = -1; tree[1].sufflink = 1;
    tree[2].len = 0; tree[2].sufflink = 1;
    for (int i = 0; i < len; i++) addLetter(i);
  }

  bool addLetter(int pos) {
    int cur = suff, curlen = 0, let = s[pos] - 'a';
    while (true) {
      curlen = tree[cur].len;
      if (pos - 1 - curlen >= 0 && s[pos - 1 - curlen] == s[pos]) break;
      cur = tree[cur].sufflink;
    }
    if (tree[cur].next.count(let)) {
      suff = tree[cur].next[let];
      count[pos] = tree[suff].num;
      return false;
    }
    num++;
    suff = num;
    tree[num].len = tree[cur].len + 2;
    tree[cur].next[let] = num;
    if (tree[num].len == 1) {
      tree[num].sufflink = 2;
      tree[num].num = 1;
      count[pos] = tree[num].num;
      return true;
    }
    while (true) {
      cur = tree[cur].sufflink;
      curlen = tree[cur].len;
      if (pos - 1 - curlen >= 0 && s[pos - 1 - curlen] == s[pos]) {
        tree[num].sufflink = tree[cur].next[let];
        break;
      }
    }
    tree[num].num = 1 + tree[tree[num].sufflink].num;
    count[pos] = tree[num].num;
    return true;
  }
};

int main() {
  ios_base::sync_with_stdio(false);
  cin.tie(0);
  palindromic_tree tree, rtree;
  cin >> tree.s;
  int n = tree.s.size();
  rtree.s = tree.s;
  reverse(rtree.s.begin(), rtree.s.end());

  tree.initTree();
  rtree.initTree();
  ll ans = 0;
  for (int i = 0; i < n-1; ++i) {
    ans += ll(tree.count[i]) * ll(rtree.count[n-i-2]);
  }
  cout << ans << endl;
}
```

9 Conclusions and future work

Conclusions

The project reached its goals successfully. The result has been a document covering many topics that hopefully some students will be able to use to improve at competitive programming. Even though many problems solved by the author during his training sessions were not included in the final documentation due to lack of time, many important topics from competitive programming were covered, which can be considered a success.

The methodology used to solve problems for this project has been the same as in the ACM ICPC training at UPC. This method has worked well for many years, as UPC has a long history of success at ACM ICPC competitions. The author of this project himself took part with the UPC team at the ACM ICPC World Finals 2016, achieving a 51st position. This is an excellent result taking into account that 40266 students from 2736 universities took part in the previous regional competitions. Also, during the development of the project, the student's team achieved a gold medal int the ACM ICPC Southwestern Regional Contest 2016.

Future work

There are some topics that were not covered in this document and are still relevant in competitive programming. Further work on this document could go into adding topics not covered yet, or expanding the current sections with other problems.

10 Bibliography

References

[1] The acm-icpc international collegiate programming contest. https://icpc.baylor.edu/. Accessed: 2017-02-26.

[2] Acm-icpc live archive. https://icpcarchive.ecs.baylor.edu. Accessed: 2017-04-17.

[3] Codechef. http://codechef.com/. Accessed: 2017-04-18.

[4] Codeforces. http://codeforces.com. Accessed: 2017-04-18.

[5] Competitive programming - topcoder. https://www.topcoder.com/community/competitive-programming/. Accessed: 2017-04-18.

[6] Cornell cs 312 lecture 26: Treaps. https://www.cs.cornell.edu/courses/cs312/2003sp/lectures/lec26.html. Accessed: 2017-04-10.

[7] Facebook hacker cup. https://www.facebook.com/hackercup/. Accessed: 2017-04-18.

[8] Google code jam. https://code.google.com/codejam/. Accessed: 2017-04-18.

[9] Heavy Light Decomposition — anudeep's blog. https://blog.anudeep2011.com/heavy-light-decomposition/. Accessed: 2017-04-03.

[10] Maxflow - notes by james aspnes, phd. http://www.cs.yale.edu/homes/aspnes/pinewiki/MaxFlow.html. Accessed: 2017-04-02.

[11] std::priority_queue - cppreference.com. http://en.cppreference.com/w/cpp/container/priority_queue. Accessed: 2017-04-06.

[12] Strongly Connected Components - lecture by rashid bin muhammad, phd. http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/GraphAlgor/strongComponent.htm. Accessed: 2017-04-01.

[13] Tree-based containers. https://gcc.gnu.org/onlinedocs/libstdc++/ext/pb_ds/tree_based_containers.html. Accessed: 2017-04-06.

[14] Uva online judge. `http://uva.onlinejudge.org/`. Accessed: 2017-04-17.

[15] Yandex algorithm contest. `https://algorithm.contest.yandex.com/`. Accessed: 2017-04-18.

[16] Z-algorithm. `https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/zalg.pdf`. Accessed: 2017-04-08.

[17] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[18] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh Vazirani. *Algorithms*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2008.

[19] S. Halim and F. Halim. *Competitive Programming 3: The New Lower Bound of Programming Contests*. Number v. 3. Lulu.com, 2013.

[20] Mikhail Rubinchik and Arseny M Shur. Eertree: an efficient data structure for processing palindromes in strings. In *International Workshop on Combinatorial Algorithms*, pages 321–333. Springer, 2015.

[21] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.