Contents

1 General

- - Rules of thumb: - $\sim 10^6$ operations is the feasibility bound.

- Use long long and std::setprecision when needed.
- Think of XOR as addition mod 2.
- Don't forget to mod after each iteration.
- Beware when dividing mod something (find the inverse).
- When input is very large use Fast I/O
- Note when updating indices!

1.1 Strategies

- **Complete Search**: traverse the entire search space, use when input is small enough.
- **Greedy**: choose the local optimum at each step. Use when optimal steps result in the optimum.
- **Divide** & **Conquer**: Divide the problem to smaller subproblems, Conquer each subproblem and Combine solutions.
- **Dynamic Programming**: Recursion + Re-use. Use when solution can be constructed efficiently from solutions to subproblems and subproblems overlap.
- **Two Pointers**: Use two pointers when processing elements from both ends or sliding through a sequence efficiently. Common in problems involving searching, merging, or maintaining a window with linear time complexity. Pay attention to how the pointers are advanced.

1.2 Tricks

- **Pattern Recognition**: sometimes there is a structure to the optimal solutions that might enable us to search a smaller subset of the solution space.
- **Change Count Order**: counting from the other side -rather than directly counting what you're asked for, count something easier that leads to the answer.
- **Variable Initialization**: make sure to initialize all variables and use them after they're assigned a value by cin for example.

- If a problem involves pairwise comparisons and asks "how many X before Y", think merge sort trick.
- If you need to process nodes in a graph where "parents before children" matters → think topological sort.
- $\log N$: think about tree, pq, sorting, binary or ternary search.
- DO NOT DIVIDE INTEGERS BEFORE CHECKING MOD!! and check positive if matter.

1.3 All Subsets

:Use binary representation for the sets. To generate all subsets:

```
void search(int k) {
  if (k==n) {
    //process subset
  } else {
    search(k+1);
    subset.push_back(k);
    search(k+1);
    subset.pop_back();
  }
}

for(int b=0; b < (1 << n); b++) {
  vector<int> subset;
  for (int i = 0; i < n; i++) {
    if (b&(1<<i)) subset.push_back(i);
  }
}
```

1.4 All Permutations

:Use next_permutation from the STL. Note that the data structure must be sorted with the same sort function that's passed to next_permutation.

```
vector<int> permutation;
for (int i=0; i < n; i++) {
    permutation.push_back(i);
}
do {
    // process
} while (next_permutation(permutation.begin(),
    permutation.end()));
```

For generating partitions we don't have an STL function, use recursion. It may be useful to represent states as graph nodes and transitions as edges and use DFS.

1.5 Troubleshoot

Pre-submit:
Write a few simple test cases if sample is not enough.
Are time limits close? If so, generate max cases.
Is the memory usage fine?
Could anything overflow?
Make sure to submit the right file.

Wrong answer:
Print your solution! Print debug output, as well.
Are you clearing all data structures between test cases?
Can your algorithm handle the whole range of input?
Read the full problem statement again.
Do you handle all corner cases correctly?
Have you understood the problem correctly?
Any uninitialized variables?
Any overflows?
Confusing N and M, i and j, etc.?
Are you sure your algorithm works?
What special cases have you not thought of?
Are you sure the STL functions you use work as you think?
Add some assertions, maybe resubmit.
Create some testcases to run your algorithm on.
Go through the algorithm for a simple case.
Go through this list again.
Explain your algorithm to a teammate.
Ask the teammate to look at your code.
Go for a small walk, e.g. to the toilet.
Is your output format correct? (including whitespace)
Rewrite your solution from the start or let a teammate do it.

Runtime error:
Have you tested all corner cases locally?
Any uninitialized variables?
Are you reading or writing outside the range of any vector?
Any assertions that might fail?
Any possible division by 0? (mod 0 for example)
Any possible infinite recursion?
Invalidated pointers or iterators?
Are you using too much memory?
Debug with resubmits (e.g. remapped signals, see Various).

Time limit exceeded:
Do you have any possible infinite loops?
What is the complexity of your algorithm?
Are you copying a lot of unnecessary data? (References)
How big is the input and output? (consider scanf)
Avoid vector, map. (use arrays/unordered map)
What do your teammates think about your algorithm?
Memory limit exceeded:
What is the max amount of memory your algorithm should need?
Are you clearing all data structures between test cases?

2 CPP Code Examples

2.1 Template

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

#define rep(i, a, b) for(int i= a; i<(b); ++i)
#define all(x) begin(x), end(x)
#define sz(x) (int)(x).size()

typedef long long ll;
typedef long double ld;
typedef pair<int, int> pii;
typedef vector<int> vi;
typedef vector<vi> Matrix;

void solve(){

}

int main() {
    cin.tie(0)->sync_with_stdio(0);
    cin.exceptions(cin.failbit);

```

```cpp
    int t = 1;
    cin >> t;
    while (t--){
        solve();
    }
}

```

2.2 Read String from cin

```cpp
string s;
getline(cin, s);
```

2.3 Classes

```cpp
class Point
{
    private:
        int x, y;
    public:
        Point() : x(0), y(0) {}
        Point(int x, int y) : x(x), y(y) {}
}
```

2.4 Files

Subscribe stdin and stdout to files

```cpp
freopen("filename.in", "r", stdin);
freopen("filename.out", "w", stdout);
```

2.5 Math Functions:

```cpp
// computes 5 raised to the power 3
cout << pow(5, 3);

// print with precision
cout << fixed << setprecision(6) << value;
```

2.6 Lambdas

- Basic lambda syntax:

```cpp
auto f = [](int x) {
    return x * x;
};
```

- Recursive lambda:

To write a recursive lambda, declare it first using std::function so it can refer to itself:

```cpp
#include <functional>

std::function<void(ll, int, ll)> solve;

solve = [&](ll u, int color, ll p) {
    // Your recursive logic here
    // Example: solve(child, new_color, u);
};
```

2.7 Sorting

```cpp
sort(a, a+n);
sort(a, a+n, [](const type& a, const type& b) {
// some critirion
});

// sort reverse
sort(a.begin(), a.end(), greater<>());
```

Given a sorted range, say an array or vector, and a value x:

- lower_bound(begin, end, x) returns an iterator to the first element $\geq x$.

- upper_bound(begin, end, x) returns an iterator to the first element $> x$.

Both perform binary search internally, so they run in $O(\log n)$ on a random-access sorted sequence.

```cpp
auto it = lower_bound(a, a+n, x);
auto ub = upper_bound(a, a+n, x);
```

2.8 Using Bitmasks

When working with subsets, it's good to have a nice representation of sets. If the i-th (least significant) digit is 1, i is in the set. If the digit is 0, it is not in the set.

- Union of two sets x and y: $x|y$.
- Intersection: $x\&y$.
- Symmetric difference: $x\hat{}\ y$.
- Singleton set $\{i\} : 1 << i$.
- Membership test: $x\&(1 << i)! = 0$.

3 Binary Search and Sorting

3.1 Binary Search

```cpp
// a sorted array is stored as
// a[0], a[1], ..., a[n-1]
int l = -1, r = n;
while (r - l > 1) {
    int m = (l + r) / 2;
    if (k < a[m]) {
        r = m; // a[l] <= k < a[m] <= a[r]
    } else {
        l = m; // a[l] <= a[m] <= k < a[r]
    }
}
```

3.2 Ternary Search

Use to search for a maximum (or minimum) of a convex (or unimodal - has one change at most) function in range $[l, r]$.

```cpp
double ternary_search(double l, double r) {
    double eps = 1e-9;
    while (r - l > eps) {
        double m1 = l + (r - l) / 3;
        double m2 = r - (r - l) / 3;
        double f1 = f(m1);
        double f2 = f(m2);
        if (f1 < f2)
            l = m1;
        else
            r = m2;
    }
    return f(l); // max f(x) in [l, r]
}
```

When $f(x)$ takes integer parameters, the interval $[l, r]$ becomes discrete, so while m_1 and m_2 can still divide it into three roughly equal parts without affecting the algorithm's correctness, the key difference is that the ternary search must stop when $r - l < 3$ and search for the minimum or maximum value in that range.

3.3 Merge Sort

It's an example of recursion, divide and conquer algorithm, and ... sort algorithm.

```
void merge(vl& a, ll left, ll mid, ll right) {
  ll i = left, j = mid+1;
  vl c;

  while (i <= mid && j <= right) {
      if (a[i] <= a[j]) {
        c.push_back(a[i]);
        i++;
      } else {
        c.push_back(a[j]);
        j++;
      }
    }
  while (i <= mid) {
    c.push_back(a[i]);
    i++;
  }
  while (j <= right) {
    c.push_back(a[j]);
    j++;
  }
  for (ll p = 0; p < c.size(); p++) {
    a[left+p] = c[p];
  }
}

void merge_sort(vl& a, ll left, ll right) {
  if (left < right) {
    ll mid = left + (right - left)/2;
    merge_sort(a, left, mid);
    merge_sort(a, mid+1, right);
    merge(a, left, mid, right);
  }
}
```

4 Data Structures

4.1 Monotonic Stack

Usage example: **Previous Smaller Element**

A monotonic stack maintains elements in sorted order (increasing or decreasing) and is useful for problems involving previous/next smaller or greater elements in linear time.

The snippet below finds the 1-based index of the previous smaller element (or 0 if none exists):

```
vl a(n), o(n);
for (ll &x : a) cin >> x;
stack<ll> st;

for (ll i = 0; i < n; i++) {
  while (!st.empty() && a[st.top()] >= a[i])
    st.pop();
  o[i] = st.empty() ? 0 : st.top() + 1;
  st.push(i);
}
for (ll x : o) cout << x << " ";
```

4.2 Priority Queue

TIP: Always check pq.empty() **before** accessing pq.top() to avoid undefined behavior.

TIP: Use 'greater' to reverse the default max-heap behavior:

```
p_q = priority_queue

p_q<int> pq; // max-heap

// min-heap
p_q<int, vector<int>, greater<int>> minpq;
```

Custom Comparator: Overload operator< in the "inverted" way when using with greater:

```
struct State {
    int d;
    bool operator<(const State& o) const {
        return d > o.d; // min-heap
    }
};
p_q<State, vector<State>, greater<State>> pq;
```

Or use a Comparator function:

```
struct CustomLess {
    bool operator()(
        const int& a, const int& b
    ) const {
        return a > b; // min-heap
    }
};

p_q<int, vector<int>, CustomLess>
    pq(data.begin(), data.end());
```

5 DP

Use when we have overlapping subproblems and optimal substructure.

5.1 Steps for DP

1. Define the subproblem (the function meaning and parameters).
2. Find the recursive rule
3. Solve base cases
4. Define the target value (which value do you need to solve the problem)
5. Define the computation order

5.2 DP TIPS

- When each choice affects the allowed next choices, or there's a cost to switching, model DP with a state like $dp[i][choice]$.

- In tree DP start with DFS and fill a parent array to define order. Every subproblem treats a node as a root of some sub-tree.

5.3 Longest Increasing Subsequence (LIS)

Given an array $a[0 \dots n-1]$, we want to compute the length of the longest strictly increasing subsequence.

Quadratic-time DP: A simple dynamic programming solution maintains $d[i]$, the length of the longest increasing subsequence ending at position i. For each i, we look at all previous $j < i$ such that $a[j] < a[i]$ and take the best extension. This runs in $O(n^2)$.

```cpp
// O(n^2) solution
int lis_quadratic(const vector<int>& a) {
    size_t n = a.size();
    if (n == 0) return 0;
    vector<int> d(n, 1); // base: each element alone is length 1

    for (size_t i = 0; i < n; ++i) {
        for (size_t j = 0; j < i; ++j) {
            if (a[j] < a[i]) {
                d[i] = max(d[i], d[j] + 1);
            }
        }
    }

    return *max_element(d.begin(), d.end());
}
```

Optimized $O(n \log n)$ method: We can do better using the patience-like method. Maintain an array `tails` where `tails[len]` is the minimum possible ending value of an increasing subsequence of length `len`. We process each element $x = a[i]$ and find the first position in `tails` where x can extend—using binary search, replacing that value. The length of the longest increasing subsequence is the largest `len` for which `tails[len-1]` is finite.

```cpp
// O(n log n) solution
int lis(const vector<int>& a) {
    vector<int> tails; // tails[len-1] = minimal tail of an increasing subsequence of length len
    for (int x : a) {
        auto it = lower_bound(tails.begin(), tails.end(), x);
        if (it == tails.end()) {
            // extend longest subsequence
            tails.push_back(x);
        } else {
            // improve existing subsequence of this length
            *it = x;
        }
    }
    return static_cast<int>(tails.size());
}
```

5.4 Tree DP

A barn has n sections connected by $n-1$ paths, forming a tree. Each section must be painted with one of three colors, such that no two adjacent sections share the same color. Some sections are already painted and cannot be changed. Determine the number of valid ways to complete the painting.

```cpp
const ll MOD = 1e9 + 7;

int main() {
ll n, k;
    cin >> n >> k;
    vector<vector<ll>> dp (n+1, vector<ll>(4, -1));
    vector<int> colored(n+1,0); // 0 no color or 1,2,3 of given
    vector<vector<ll>> adj(n+1);
    // n-1 edges
    ll a,b;
    for (ll i = 0; i < n-1;i++) {
        cin >>a >>b;
        adj[a].push_back(b);
        adj[b].push_back(a);
    }

    int c;
    for (ll i =0; i < k; i++) {
        cin >> a >> c;
        colored[a] = c;

        for (auto &c1: {1, 2, 3}) {
            if (c1 != c)
                dp[a][c1] = 0;
        }
    }

    function<void(long long, int, long long)> solve;
    solve = [&](ll u, int color, ll p) {
        if (colored[u] && color != colored[u]){
            dp[u][color] = 0;
            return;
        }

        vector<ll> children;
        for (auto &v: adj[u]) {
            if (v == p) continue;
            ll op_v = 0;
            for (auto &c: {1, 2, 3}) {
                if (c == color) continue;
                if (dp[v][c] == -1) {
                    solve(v, c, u);
                }
                op_v = (op_v + dp[v][c]) % MOD;
            }
            children.push_back(op_v);
        }
        dp[u][color] = 1;
        for (auto& child: children) {
            dp[u][color] = (child*dp[u][color]) % MOD;
        }
    };
    ll i;
    for (i= 1; i <= n; i++) {
        if (colored[i] != 0) {
            solve(i, colored[i], 0);
            break;
        }
    }

    cout << dp[i][colored[i]];

    return 0;
}
```

6 math

6.1 Useful Identities

$10^9 + 7$ is prime.

$$\sum_{k=1}^{n} k^2 = \frac{1}{6} n(n+1)(2n+1)$$

$$\sum_{k=1}^{n} k^3 = \left(\sum k\right)^2 = \left(\frac{1}{2} n(n+1)\right)^2$$

Common Denominator Rule:

For $\sum_{i=1}^{n} \frac{n_i}{d_i}$, let: - $D = \prod_{i=1}^{n} d_i$ (common denominator, not necessarily minimal)

Then:

$$\sum_{i=1}^{n} \frac{n_i}{d_i} = \frac{D \cdot \sum_{i=1}^{n} \frac{n_i}{d_i}}{D}$$

Geometric sequence sum: $S_n = a \cdot \frac{1-r^n}{1-r}$

Arithmetic sequence sum: $S_n = \frac{1}{2}(2a + (n-1)d)$

Modular Arithmetic::

$$(a+b) \bmod m = (a \bmod m + b \bmod m) \bmod m$$

$$(a-b) \bmod m = (a \bmod m - b \bmod m) \bmod m$$

$$(a \cdot b) \pmod m = ((a \bmod m) \cdot (b \bmod m)) \bmod m$$

$$a^b \bmod m = (a \bmod m)^b \bmod m$$

```
// return a % b (positive value)
int mod(int a, int b) {
  return ((a%b) + b) % b;
}
```

Fermat's Little Theorem: states that all integers a not divisible by p satisfy $a^{p-1} \equiv 1 \pmod p$. Consequently, $a^{p-2} \cdot a \equiv 1 \pmod p$. Therefore, a^{p-2} is a modular inverse of a modulo p. It can be calculated in $O(\log(n))$ with fast exponentiation.

Inverse mod p with Euclidean Division: To find i^{-1} mod p iteratively for each $1 \leq i \leq p$ when p is prime, use:

$$\text{inv}[1] = 1$$

$$\text{inv}[i] = \left(m - \lfloor \frac{m}{i} \rfloor\right) \cdot \text{inv}[m \mod i] \mod m \quad i = 2, 3, ...$$

6.2 GCD, LCM

$gcd(a,b)$: The largest number that divides both a and b.

```
ll gcd(ll a, ll b) {
  ll a1 = max(a,b);
  ll b1 = min(a,b);
  return b1 == 0 ? a1 : gcd(b1, a1 % b1);
}

// Or use STL gcd
#include <numeric>

int main() {
    cout << std::gcd(6, 10) == 2);
}
```

$extended\_euclid(a,b)$: Given (a,b) find (x,y) such that $ax + by = gcd(a,b)$.

```
// returns d = gcd(a,b)
// finds x,y such that d = ax + by
int extended_euclid(int a, int b,
  int &x, int &y) {
 int xx = y = 0;
 int yy = x = 1;
 while (b) {
 int q = a/b;
 int t = b; b = a%b; a = t;
 t = xx; xx = x-q*xx; x = t;
 t = yy; yy = y-q*yy; y = t;
 }
 return a;
}
```

To solve an equation of type $ax \equiv b \mod m$ we want to find x such that $ax + my = b$.

- If $gcd(a,m)$ does not divide b, there is no solution

- Otherwise, use extended euclid's to find $ax' + my' = gcd(a,m)$.

Multiply by $b/gcd(a,m)$ to obtain b on the rhs.

We get $x = bx'/gcd(a,m)$. Reduce x modulo $m/gcd(a,m)$

```
ll x0, y0;
// finds x0, y0 s.t. ax0 + my0 = gcd(a, m)
extended_euclid(a, m, x0, y0);
ll x = (b / g) * x0;
// reduce mod (m / g)
cout << mod(x, m / g) << "\n";
```

$LCM(a,b)$: Least Common Multiple.

$$lcm(a,b) = \frac{a \cdot b}{gcd(a,b)}$$

6.3 Fast Exponentiation

Fast Binary Exponentiation

```
ll binpow(ll a, ll b) {
  ll res = 1;
  while (b > 0) {
    if (b & 1)
      res *= a;
    a = a * a;
    b >>= 1;
  }
  return res;
}
```

Matrix Multiplication

```
Matrix mat_mult(
    const Matrix &A, const Matrix &B
) {
  ll N = A.size();
  ll M = B.size();
  ll K = B[0].size();

  Matrix C(N, vector<ll>(K, 0));

  for (ll i = 0; i < N; ++i) {
    for (ll k = 0; k < M; ++k) {
      ll a = A[i][k] % MOD;
      for (ll j = 0; j < K; ++j) {
        C[i][j] =
          (C[i][j] + a * (B[k][j] % MOD)) % MOD;
      }
    }
  }

  return C;
}
```

Fast Exponentiation

Goal: compute A^s for possibly very big $s > 2^{10}$.

Idea: use binary representation of s, compute $A^{2^i} = A^{2^{i-1}} \cdot A^{2^{i-1}}$.

Note: can be used to compute probabilities in Markov chains.

```
Matrix matrix_exponent(Matrix A, ll power) {
  int N = A.size();
  Matrix res(N, vector<ll>(N, 0));
  FOR(i, 0, N) { res[i][i] = 1; }

  while (power > 0) {
```

```
    if (power % 2)
      res = mat_mult(res, A);
    A = mat_mult(A, A);
    power /= 2;
  }

  return res;
}
```

Fibonacci Numbers

Fibonacci Recurrence: $F(n) = F(n-1) + F(n-2)$. $F(n)$ can be computed efficiently with Fast Exponentiation.

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F(n+1) & F(n) \\ F(n) & F(n-1) \end{bmatrix}$$

Binomial Coefficients

$\binom{n}{k}$ - number of ways to choose k elements from a set of n elements.

$\binom{n}{k} = \frac{n!}{k!(n-k)!}$

Efficiently compute using $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$, $\binom{n}{0} = \binom{n}{n} = 1$

Prime numbers

```
bool isprime[N];
void sieve(int n) {
    fill(isprime+2, isprime+n, 1);
    for (int i = 2; i*i <= n; i++)
        if (isprime[i])
            for (int j = i*i; j <= n; j += i)
                isprime[j] = 0;
}
```

1-d linear equations

```
// computes x and y such that ax + by = c
// returns whether the solution exists
bool linear_diophantine(
int a, int b, int c, int &x, int &y
) {
  if (!a && !b) {
    if (c) return false;
    x = 0; y = 0;
    return true;
  } if (!a) {
    if (c % b) return false;
    x = 0; y = c / b;
    return true;
  } if (!b) {
    if (c % a) return false;
    x = c / a; y = 0;
    return true;
  }
  int g = gcd(a, b);
  if (c % g) return false;
  x = c / g * mod_inverse(a / g, b / g);
  y = (c - a*x) / b;
  return true;
}
```

Gaussian Elimination

Goal: Given a matrix $A \in \mathbb{R}^{m \times n}$, vector $b \in \mathbb{R}^m$ find $x \in \mathbb{R}^m$ such that $Ax = b$.

There are three types of elementary row operations which may be performed on the rows of a matrix:

1. **swap** the positions of two rows.
2. **multiply** a row by a non-zero scalar.
3. **add** to a new row a scalar multiple of another.

If the matrix is associated with a system of linear equations, then these operations do not change the solution set.

By combining the 3 elementary operations, we can bring a system of equations into its canonical form, then solve it using **back substitution**.

```

// aug = [A | b], sol is a particular solution,
// basis is the nullspace basis
int gauss(vvi& aug, vi& sol, vvi& basis) {
  int n = aug.size(), m = aug[0].size() - 1;
  int row = 0;
  vi where(m, -1);

  for (int col = 0; col < m && row < n; ++col) {
    int sel = -1;
    for (int i = row; i < n; ++i)
      if (aug[i][col]) { sel = i; break; }
    if (sel == -1) continue;

    swap(aug[sel], aug[row]);
    where[col] = row;

    int inv = modinv(aug[row][col]);
    for (int j = col; j <= m; ++j)
      aug[row][j] =
        1LL * aug[row][j] * inv % MOD;

    for (int i = 0; i < n; ++i)
      if (i != row && aug[i][col]) {
        int factor = aug[i][col];
        for (int j = col; j <= m; ++j)
          aug[i][j] =
           (aug[i][j] - 1LL * factor *
             aug[row][j] % MOD + MOD) % MOD;
    }
    ++row;
  }

  for (int i = row; i < n; ++i)
    if (aug[i][m]) return -1;

  sol.assign(m, 0);
  for (int i = 0; i < m; ++i)
    if (where[i] != -1)
      sol[i] = aug[where[i]][m];

  basis.clear();
  for (int i = 0; i < m; ++i)
    if (where[i] == -1) {
      vi vec(m);
      vec[i] = 1;
      for (int j = 0; j < m; ++j)
        if (where[j] != -1)
         vec[j] = (MOD - aug[where[j]][i]) % MOD;
      basis.push_back(vec);
  }

  return (int) basis.size();
}
```

7 Probability

Let X be a discrete random variable with probability $p_X(x)$ of assuming the value x. It will then have an expected value (mean) $\mu = \mathbb{E}(X) = \sum_x x p_X(x)$ and variance

$\sigma^2 = V(X) = \mathbb{E}(X^2) - (\mathbb{E}(X))^2 = \sum_x (x - \mathbb{E}(X))^2 p_X(x)$ where σ is the standard deviation. If X is instead continuous it will have a probability density function $f_X(x)$ and the sums above will instead be integrals with $p_X(x)$ replaced by $f_X(x)$.

Expectation is linear:

$$\mathbb{E}(aX + bY) = a\mathbb{E}(X) + b\mathbb{E}(Y)$$

For independent X and Y,

$$V(aX + bY) = a^2V(X) + b^2V(Y).$$

7.1 Discrete distributions

Binomial distribution

The number of successes in n independent yes/no experiments, each which yields success with probability p is $\mathrm{Bin}(n, p)$, $n = 1, 2, \ldots$, $0 \leq p \leq 1$.

$$p(k) = \binom{n}{k} p^k (1-p)^{n-k}$$

$$\mu = np,\ \sigma^2 = np(1-p)$$

$\mathrm{Bin}(n, p)$ is approximately $\mathrm{Po}(np)$ for small p.

First success distribution

The number of trials needed to get the first success in independent yes/no experiments, each which yields success with probability p is $\mathrm{Fs}(p)$, $0 \leq p \leq 1$.

$$p(k) = p(1-p)^{k-1},\ k = 1, 2, \ldots$$

$$\mu = \frac{1}{p},\ \sigma^2 = \frac{1-p}{p^2}$$

Poisson distribution

The number of events occurring in a fixed period of time t if these events occur with a known average rate κ and independently of the time since the last event is $\mathrm{Po}(\lambda)$, $\lambda = t\kappa$.

$$p(k) = e^{-\lambda}\frac{\lambda^k}{k!},\ k = 0, 1, 2, \ldots$$

$$\mu = \lambda,\ \sigma^2 = \lambda$$

7.2 Continuous distributions

Uniform distribution

If the probability density function is constant between a and b and 0 elsewhere it is $\mathrm{U}(a, b)$, $a < b$.

$$f(x) = \begin{cases} \frac{1}{b-a} & a < x < b \\ 0 & \text{otherwise} \end{cases}$$

$$\mu = \frac{a+b}{2},\ \sigma^2 = \frac{(b-a)^2}{12}$$

Exponential distribution

The time between events in a Poisson process is $\mathrm{Exp}(\lambda)$, $\lambda > 0$.

$$f(x) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

$$\mu = \frac{1}{\lambda},\ \sigma^2 = \frac{1}{\lambda^2}$$

Normal distribution

Most real random values with mean μ and variance σ^2 are well described by $\mathcal{N}(\mu, \sigma^2)$, $\sigma > 0$.

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

If $X_1 \sim \mathcal{N}(\mu_1, \sigma_1^2)$ and $X_2 \sim \mathcal{N}(\mu_2, \sigma_2^2)$ then

$$aX_1 + bX_2 + c \sim \mathcal{N}(\mu_1 + \mu_2 + c, a^2\sigma_1^2 + b^2\sigma_2^2)$$

7.3 Markov chains

A *Markov chain* is a discrete random process with the property that the next state depends only on the current state. Let $X_1, X_2, \ldots$ be a sequence of random variables generated by the Markov process. Then there is a transition matrix $\mathbf{P} = (p_{ij})$, with $p_{ij} = \Pr(X_n = i | X_{n-1} = j)$, and $\mathbf{p}^{(n)} = \mathbf{P}^n \mathbf{p}^{(0)}$ is the probability distribution for X_n (i.e., $p_i^{(n)} = \Pr(X_n = i)$), where $\mathbf{p}^{(0)}$ is the initial distribution.

π is a stationary distribution if $\pi = \pi\mathbf{P}$. If the Markov chain is *irreducible* (it is possible to get to any state from any state), then $\pi_i = \frac{1}{\mathbb{E}(T_i)}$ where $\mathbb{E}(T_i)$ is the expected time between two visits in state i. π_j/π_i is the expected number of visits in state j between two visits in state i.

For a connected, undirected and non-bipartite graph, where the transition probability is uniform among all neighbors, π_i is proportional to node i's degree.

A Markov chain is *ergodic* if the asymptotic distribution is independent of the initial distribution. A finite Markov chain is ergodic iff it is irreducible and *aperiodic* (i.e., the gcd of cycle lengths is 1). $\lim_{k\to\infty} \mathbf{P}^k = \mathbf{1}\pi$.

A Markov chain is an A-chain if the states can be partitioned into two sets $\mathbf{A}$ and $\mathbf{G}$, such that all states in $\mathbf{A}$ are absorbing ($p_{ii} = 1$), and all states in $\mathbf{G}$ leads to an absorbing state in $\mathbf{A}$. The probability for absorption in state $i \in \mathbf{A}$, when the initial state is j, is $a_{ij} = p_{ij} + \sum_{k\in\mathbf{G}} a_{ik}p_{kj}$. The expected time until absorption, when the initial state is i, is $t_i = 1 + \sum_{k\in\mathbf{G}} p_{ki}t_k$.

8 Graphs

Representing Graphs

- Adjacency List - array of lists.

- Adjacency Matrix - 2D array where matrix[i][j] == 1 means there's an edge from node i to node j.

8.1 DFS, BFS

Time Complexity: $\mathcal{O}(|V| + |E|)$.

Use For: cycle detection, spanning trees, traversing a graph.

For BFS - FIFO, use queue. For DFS - LIFO, use stack.

Iterative:

```
void bfs(int start, vvi &graph) {
  int n = graph.size();
  vector<bool> vis(n, false);
  queue<int> q; // for dfs use stack.
  q.push(start);
  vis[start] = true;

  while (!q.empty()) {
    int cur_node = q.front(); // for dfs top.
    q.pop();

    for(int neighbor : graph[cur_node]) {
      if (!vis[neighbor]){
        vis[neighbor] = true;
        q.push(neighbor);
      }
    }
```

```
  }
}
```

Recursive: (Beware of stack overflow)

```
void dfs(
int start, vector<bool> &vis, vvi &graph
) {
  vis[start] = true;

  for(int neighbor : graph[start]) {
    if (!vis[neighbor]) {
      dfs(neighbor, vis, graph);
    }
  }
}
```

8.2 Topological Sort

Idea: An ordering of vertices along a line, such that all edges are pointing in the same direction. For each directed edge (u, v), u appears before v.

Time Complexity: $\mathcal{O}(|V| + |E|)$.

TS exists iff the graph is directed acyclic (DAG).

Kahn's Algorithm:

```
vi topological_sort_bfs(vvi &graph){
  int n = graph.size();
  vi indegree(n, 0);
  vi topo_v;
  for (int i =0; i < n; i++){
    for (int neighbor: graph[i]) {
      indegree[neighbor]++;
    }
  }
  queue<int> q;
  for (int i = 0; i < n; i++){
    if (indegree[i] == 0) {
      q.push(i);
    }
  }

  while (!q.empty()){
    int cur = q.front();
    q.pop();
    if (cur == 0) continue;
    topo_v.push_back(cur);

    for(int neighbor : graph[cur]) {
      indegree[neighbor]--;
      if (indegree[neighbor] == 0) {
        q.push(neighbor);
        }
    }
  }

  return topo_v;
}
```

Recursive:

```
void topological_sort_dfs(
  int node, vector<bool> &vis, stack<int> &s,
  vvi &graph
) {
  vis[node] = true;

  for(int neighbor : graph[node]) {
    if (!vis[neighbor]) {
      topological_sort_dfs(neighbor, vis, s,
      graph);
    }
  }
  s.push(node);
}
```

8.3 Dijkstra

Idea: Finds the shortest path from the source s to every other node in the graph.

Time Complexity: $\mathcal{O}(|E| \log |V|)$.

```
pq = priority_queue

void dijkstra(
    const vector<vpii> &adj, vi &dist, vi &prev,
    ll start
) {
    int n = adj.size();
    prev = vi(n, -1);
    dist = vi(n, INF);
    dist[start] = 0;

    pq<pii, vector<pii>, greater<pii>> queue;
    queue.push({0, start});

    while (!queue.empty()) {
        pii d_u = queue.top(); queue.pop();
        int d = d_u.first;
        int u = d_u.second;

        if (d > dist[u]) continue;

        for (size_t i=0;i < adj[u].size();++i){
            int v = adj[u][i].first;
            int w = adj[u][i].second;
            if (dist[v] > dist[u] + w) {
                dist[v] = dist[u] + w;
                prev[v] = u;
                queue.push({dist[v], v});
            }
        }
    }
}
```

8.4 Bellman-Ford

Idea: Finds shortest path in graph from v to all other vertices, even if there are negative edges.

Time Complexity: $\mathcal{O}(|V||E|)$.

```
vector<ll> bellman_ford(
  int n, int start,
  vector<tuple<int, int, ll>> &edges
) {
  vector<ll> dist(n, LLONG_MAX);
  dist[start] = 0;
  for (int i = 0; i <n-1; i++) {
    for (auto &[u, v, w] : edges) {
      if (dist[u] != LLONG_MAX) {
        dist[v] = min(dist[v], dist[u] + w);
    }}}

  for (auto &[u, v, w] : edges) {
    if (dist[u] != LLONG_MAX &&
        dist[u] + w < dist[v]) {
      cout << "Found a negative-weight cycle";
    }}
  return dist;
}
```

8.5 Floyd-Warshall

Idea: Finds shortest path distance between every pair of nodes. Works when there are negative weights, but no negative cycles!
Time Complexity: $\mathcal{O}(|V|^3)$.

```
vector<vector<ll>> floyd_warshall(
  vector<vector<bool>> &adj,
  vector<vector<ll>> &weights
) {
  int n = adj.size();
  vector<vector<ll>> dist(n, vector<ll>(n));

  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      if (i == j) dist[i][j] = 0;
      else if (adj[i][j])
        dist[i][j] = weights[i][j];
      else dist[i][j] = LLONG_MAX;
    }
  }

  for (int k = 0; k < n; k++) {
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        if (dist[i][k] != LLONG_MAX &&
            dist[k][j] != LLONG_MAX) {
          dist[i][j] =
            min(dist[i][j],
              dist[i][k] + dist[k][j]);
            }
      }
    }
  }
  return dist;
}
```

8.6 SCC

Idea: Finds the strongly connected components of a directed graph and the condensation graph.
Time Complexity: $\mathcal{O}(|V| + |E|)$.
Steps:

1. DFS(G)
2. compute G^T
3. DFS(G^T) by the descending order of visiting vertices in step 1.
4. return each SCC as the vertices of each forest in step 3.

Implementation:

```
void dfs(
    ll v, const vvl &adj, vector<bool> &visited, vl& output
    ) {
  visited[v] = true;
  for (auto u: adj[v]) {
    if (!visited[u]) {
      dfs(u, adj, visited, output);
    }
  }
  output.push_back(v);
}

void scc(
    const vvl &adj, vector<bool> &visited, vvl &components
    ) {
  int n = adj.size();
  components.clear();

  vl order;
  visited.assign(n, false);

  for (ll i = 0; i < n; i++)
    if (!visited[i])
      dfs(i, adj, visited, order);

  vvl adj_rev(n);
  for (ll v = 0; v < n; v++)
    for (auto u : adj[v])
      adj_rev[u].push_back(v);

  visited.assign(n, false);
  reverse(order.begin(), order.end());

  vector<int> roots(n, 0);

  for (auto v : order)
    if (!visited[v]) {
      vl component;
      dfs(v, adj_rev, visited, component);
      components.push_back(component);
      int root = *min_element(begin(component), end(component));
      for (auto u : component)
        roots[u] = root;
    }
}
```

8.7 Solve 2SAT with SCC

Idea: Given m variables and n boolean clauses in 2-CNF (i.e., each clause is a disjunction of two literals), we can reduce the satisfiability problem to graph analysis.

Construct a directed graph with $2m$ vertices: one for each variable x and its negation $\neg x$. For each clause $(a \lor b)$, add two directed edges: $(\neg a \to b)$ and $(\neg b \to a)$. These edges represent the logical implications required to satisfy the clause.

After building the implication graph, compute its strongly connected components (SCCs). The formula is satisfiable if and only if no variable and its negation belong to the same SCC. If each variable and its negation are in different SCCs, a valid assignment exists.

Time Complexity: $\mathcal{O}(m + n)$

```

struct TwoSatSolver {
    int n_vars;
    int n_vertices;
    vector<vi> adj, adj_t;
    vector<bool> used;
    vector<int> order, comp;
    vector<bool> assignment;

    TwoSatSolver(int _n_vars) :
      n_vars(_n_vars), n_vertices(2*_n_vars), adj(n_vertices),
      adj_t(n_vertices), used(n_vertices), order(),
      comp(n_vertices, -1), assignment(n_vars) {
        order.reserve(n_vertices);
    }

    void dfs1(int v) {
        used[v] = true;
        for(int u : adj[v]) {
```

```cpp
            if (!used[u])
                dfs1(u);
        }
        order.push_back(v);
    }

    void dfs2(int v, int cl) {
        comp[v] = cl;
        for (int u : adj_t[v]) {
            if (comp[u] == -1)
                dfs2(u, cl);
        }
    }

    bool solve_2SAT() {
        order.clear();
        used.assign(n_vertices, false);
        for (int i = 0; i < n_vertices; ++i) {
            if (!used[i])
                dfs1(i);
        }

        comp.assign(n_vertices, -1);
        for (int i = 0, j = 0; i < n_vertices; ++i) {
            int v = order[n_vertices - i - 1];
            if (comp[v] == -1)
                dfs2(v, j++);
        }

        assignment.assign(n_vars, false);
        for (int i = 0; i < n_vertices; i += 2) {
            if (comp[i] == comp[i + 1])
                return false;
            assignment[i / 2] = comp[i] < comp[i + 1];
        }
        return true;
    }

    void add_disjunction(int a, bool na, int b, bool nb) {
        // na and nb signify whether a and b are to be negated
        a = 2 * a ^ na;
        b = 2 * b ^ nb;
        int neg_a = a ^ 1;
        int neg_b = b ^ 1;
        adj[neg_a].push_back(b);
        adj[neg_b].push_back(a);
        adj_t[b].push_back(neg_a);
        adj_t[a].push_back(neg_b);
    }
};

int main() {
    int n, m;
    cin >> n >> m;

    TwoSatSolver solver (m);
    char c1, c2;
    int x1, x2;
    for(int i = 0;i < n; i++) {
        cin >> c1 >> x1 >> c2 >> x2;
        bool a1 = (c1 == '+');
        bool a2 = (c2 == '+');
        solver.add_disjunction(x1 - 1, a1, x2 - 1, a2);
    }

    if (solver.solve_2SAT()) {
        for (int i = 0; i < m; i++) {
            cout << (solver.assignment[i]  ? '+' : '-');
            if (i < m - 1) {
                cout << " ";
            }
        }
        cout << "\n";
    } else {
        cout << "IMPOSSIBLE" << "\n";
    }
}
```

8.8 Conclusion Table

Restrictions		SSSP Algorithm	
Graph	Weights	Name	Running Time (
General	Unweighted	BFS	$\|V\| + \|E\|$
DAG	Any	DAG Relaxation	$\|V\| + \|E\|$
General	Non-negative	Dijkstra	$\|V\| \log \|V\| + \|$
General	Any	Bellman-Ford	$\|V\| \cdot \|E\|$

9 Tree Based DAST

9.1 Fenwick (Binary Indexed Tree

A simpler version of segment tree which is good enough in most cases.

```cpp
/*------------------------- Fenwick (Binary Indexed) -----------------------*/
template <class T = long long> struct Fenwick {
    vector<T> bit;
    Fenwick(int n = 0) : bit(n + 1) {}
    void add(int idx, T delta) { // 0-based
        for (++idx; idx < (int)bit.size(); idx += idx & -idx)
            bit[idx] += delta;
    }
    T pref(int idx) const { // sum[0 .. idx]
        T res = 0;
        for (++idx; idx; idx -= idx & -idx)
            res += bit[idx];
        return res;
    }
    T sum(int l, int r) const { // sum[l .. r] (0-based, inclusive)
        return pref(r) - (l ? pref(l - 1) : 0);
    }
};
```

Example usage - Exercise 5 - Problem E

```cpp
    int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n;
    if (!(cin >> n))
        return 0;

```

```
    struct E {
        int x, type, y, y2;
    }; // type: +1 add, 0 query, -1 remove
    vector<E> ev;
    vector<int> ys;

    while (n--) {
        int x1, y1, x2, y2;
        cin >> x1 >> y1 >> x2 >> y2;
        if (y1 == y2) { // horizontal
            if (x1 > x2)
                swap(x1, x2);
            ev.push_back({x1, +1, y1, 0});
            ev.push_back({x2, -1, y1, 0});
            ys.push_back(y1);
        } else { // vertical
            if (y1 > y2)
                swap(y1, y2);
            ev.push_back({x1, 0, y1, y2});
            ys.push_back(y1);
            ys.push_back(y2);
        }
    }

    sort(ys.begin(), ys.end());
    ys.erase(unique(ys.begin(), ys.end()),
    ↪  ys.end());
    auto id = [&](int y) {
        return int(lower_bound(ys.begin(),
        ↪  ys.end(), y) - ys.begin());
    };

    for (auto &e : ev) {
        e.y = id(e.y);
        if (e.type == 0)
            e.y2 = id(e.y2);
    }

    sort(ev.begin(), ev.end(), [](const E &a,
    ↪  const E &b) {
        if (a.x != b.x)
            return a.x < b.x;
        return a.type > b.type; // +1 before 0
        ↪  before -1
    });

    Fenwick bit(ys.size());
    ll ans = 0;
    for (auto &e : ev) {
        if (e.type == +1)
            bit.add(e.y, 1);
        else if (e.type == 0)
            ans += bit.sum(e.y, e.y2);
        else
            bit.add(e.y, -1);
    }
    cout << ans << "\n";
}
```

9.2 Segment Tree

Idea: Balanced binary tree that lets one efficiently

- answer aggregate questions about contiguous intervals, and
- and update those intervals, and
- aggregates should be associative (sum, product, lcm, gcd, etc)

Time Complexity: $O(n)$ to build, $O(\log n)$ to query or update.
Code Implementation of Min Segment Tree:

```
struct SegTree {
  int n;
  vector<int> seg;
  SegTree(int n) : n(n), seg(4 * n, INF) {}

  void update(
    int idx, int val, int node, int l, int r
  ) {
    if (l == r) {
        seg[node] = val;
        return;
    }
    int mid = (l + r) >> 1;
    if (idx <= mid)
        update(idx, val, node<<1, l, mid);
    else
        update(idx, val, node<<1|1, mid+1, r);
    seg[node] =
        min(seg[node<<1], seg[node<<1 | 1]);
  }

  void update(int idx, int val) {
    update(idx, val, 1, 1, n);
  }

  int query(
    int B, int Y, int node, int l, int r
  ) const {
    if (r < B || seg[node] > Y)
        return -1;
    if (l == r)
        return l;
    int mid = (l + r) >> 1;
    int res = query(B, Y, node<<1, l, mid);
    if (res != -1)
        return res;
    return query(B, Y, node <<1|1, mid+1, r);
  }

  int query(int B, int Y) const {
    return query(B, Y, 1, 1, n);
  }
};
```

9.3 Lazy Segment Tree

ADD EXAMPLE!!

9.4 Union Find

Idea: We maintain disjoint sets of elements with support for three operations:

- `make_set(v)` – creates a new set with a single element v.
- `find_set(v)` – returns the representative (leader) of the set containing v.
- `union_sets(a, b)` – merges the sets containing elements a and b.

Two elements a and b are in the same set iff `find_set(a) == find_set(b)`.
Time Complexity: $O(\alpha(n))$ per operation with path compression and union by rank. Disjoint-Set Union

```cpp
/*---------------------------- Disjoint-Set Union ---------------------------*/
struct DSU {
    vector<int> p, sz, val; // parent, component size, optional payload
    DSU(int n) : p(n), sz(n, 1), val(n) { iota(p.begin(), p.end(), 0); }
    int find(int v) { return p[v] == v ? v : p[v] = find(p[v]); }
    int unite(int a, int b) { // returns new root
        a = find(a);
        b = find(b);
        if (a == b)
            return a;
        if (sz[a] > sz[b])
            swap(a, b);
        p[a] = b;
        sz[b] += sz[a];
        return b;
    }
};
```

Example usage - Exercise 5 - Problem C

```cpp
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    const int MAXV = 100000; // label upper-bound per statement
    int T;
    if (!(cin >> T))
        return 0;
    for (int tc = 1; tc <= T; ++tc) {
        int n, q;
        cin >> n >> q;
        vector<int> A(n);
        for (int &x : A)
            cin >> x;

        DSU d(n);
        vector<int> rep(MAXV + 1, -1); // label → DSU root
        // build DSU: merge duplicates so each label has exactly one root
        for (int i = 0; i < n; ++i) {
            d.val[i] = A[i];
            int lbl = A[i];
            if (rep[lbl] == -1) {
                rep[lbl] = i; // first occurrence becomes root
            } else {
                int r = d.unite(rep[lbl], i);
                rep[lbl] = r; // keep root up-to-date
            }
        }

        cout << "Case " << tc << ":\n";
        while (q--) {
            int type;
            cin >> type;
            if (type == 1) { // replace x by y globally
                int x, y;
                cin >> x >> y;
                if (x == y)
                    continue;
                int rx = rep[x];
                if (rx == -1)
                    continue; // no element with label x
                int ry = rep[y];
                if (ry == -1) { // y not present yet: just relabel x-cluster
                    d.val[rx] = y;
                    rep[y] = rx;
                } else {                    // merge the two clusters
                    int r = d.unite(rx, ry); // r is new root
                    d.val[r] = y;
                    rep[y] = r;
                }
                rep[x] = -1; // label x no longer exists
            } else {          // query current label at position idx (1-based)
                int idx;
                cin >> idx;
                int root = d.find(idx - 1);
                cout << d.val[root] << '\n';
            }
        }
    }
    return 0;

```

Also note: A tree-based variant with $O(\log n)$ operations exists and can be more powerful in certain cases.

```cpp
class UnionFind {
private:
    vector<ll> par;
    vector<ll> size;

public:
    explicit UnionFind(ll n) : par(n), size(n,1) {
        for (ll i = 0; i<n; ++i) par[i] = i;
    }

    ll find(ll u) {
        if (par[u] != u)
            par[u] = find(par[u]);
        return par[u];
    }

    // true if merged, false if in same component
    bool merge(ll u, ll v) {
        u = find(u);
        v = find(v);
        if (u == v) return false;

        if (size[u] <= size[v]) {
            size[v] += size[u];
            par[u] = v;
        } else {
            size[u] += size[v];
            par[v] = u;
        }
        return true;
    }
};
```

9.5 Solve 2SAT with UF

We represent each variable x_i and its negation $\neg x_i$ as two separate nodes:

$$\texttt{var\_id}(x, \text{true}) = 2x, \quad \texttt{var\_id}(x, \text{false}) = 2x + 1$$

For each disjunctive clause $(a \vee b)$, add implications using Union-Find:

$$\neg a \Rightarrow b \quad \text{and} \quad \neg b \Rightarrow a$$

This is done by merging:

$$\texttt{merge}(\neg a, b), \quad \texttt{merge}(\neg b, a)$$

Contradiction condition: A variable x and its negation $\neg x$ must not belong to the same component. That is, $\texttt{find}(x_{\text{true}}) \neq \texttt{find}(x_{\text{false}})$.
Assignment rule: Assign `true` to x if the leader of x_{true} is smaller than that of x_{false}:

$$\texttt{assignment}[x] = (\texttt{leader}(x_{\text{true}}) < \texttt{leader}(x_{\text{false}}))$$

```
struct TwoSatSolver {
    int n_vars;
    UnionFind uf;
    vector<vi> adj;

    vector<bool> assignment;

    TwoSatSolver(int _n_vars) : n_vars(_n_vars), uf(2*_n_vars),
                                adj(2*_n_vars), assignment(n_vars) {}

    bool solve_2SAT() {
        for (int i = 0; i < n_vars; ++i) {
            if (uf.find(2*i) == uf.find(2*i+1)) {
                return false;
            }
        }
        return true;
    }

    void add_disjunction(int a, bool na, int b, bool nb) {
        // na and nb signify whether a and b are to be negated
        a = 2 * a ^ na;
        b = 2 * b ^ nb;
        int neg_a = a ^ 1;
        int neg_b = b ^ 1;

        uf.merge(neg_a,b);
        uf.merge(neg_b,a);
    }
};
```

10 Computational Geometry

Tips
- Using complex numbers can be helpful sometimes.
- Sometimes input is small, and brute force works.

10.1 Points and Lines

Points in the $2d$ plane are represented using two coordinates.
Lines Segments Reps:
- Two endpoints (p_1, p_2).
- One endpoint (p_0), direction vector v, and length d.
- One endpoint (p_1), slope α, and length d_x.

Distance between two points $a = (x_1, y_1), b = (x_2, y_2)$:

$$d(a, b) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Dot product in $2d$ defined as:

```
ftype dot(point2d a, point2d b) {
    return a.x * b.x + a.y * b.y;
}
```

Some Properties:

$$\|a\|^2 = a \cdot a$$

$$a \cdot b = \|a\|\|b\| \cos(\theta)$$

$$a \cdot b = 0 \iff a \perp b$$

Sort By Argument:

```
auto cmp = [](const Point &a, const Point &b) {
  return atan2l(a.y, a.x) < atan2l(b.y, b.x);
};
sort(points.begin(), points.end(), cmp);
```

10.2 Cross Product

Determinant of $[\boldsymbol{u}, \boldsymbol{v}]^T$: $\boldsymbol{u} \times \boldsymbol{v} = u_x \cdot v_y - u_y \cdot v_x$.

```
__int128 cross(const Point& a, const Point& b) {
  return (__int128)a.x * b.y - (__int128)a.y * b.x;
}
```

The sign of the cp tells the orientation.
Orientation
Idea: for 3 points a, b, c determine their orientation.

```
x = first, y = second

ll ccw (
  const pll &a, const pll &b, const pll &c
) {
  return (b.x - a.x) * (c.y - a.y) -
    (b.y - a.y) * (c.x - a.x);
}
```

- $\text{ccw}(a, b, c) > 0$: counter-clockwise (CCW) (c on the left).
- $\text{ccw}(a, b, c) < 0$: clockwise (CW). (c on the right).
- $\text{ccw}(a, b, c) = 0$ Points are co-linear.

10.3 Segment-Segment Intersection

Idea: Given 2 segments AB and CD, determine if they intersect.

```
inline int  sgn(ll v)             { return (v > 0) - (v < 0); }

/*  Inside the axis-aligned bounding box that ab is its diagonal.  */
inline bool in_box(P a, P b, P p) {
    return min(a.x, b.x) <= p.x && p.x <= max(a.x, b.x) &&
           min(a.y, b.y) <= p.y && p.y <= max(a.y, b.y);
```

```cpp
}

/*  Checks whether point p lies *on* the (closed) segment ab.  */
inline bool on_seg(P a, P b, P p) {
    return orient(a, b, p) == 0 && in_box(a, b, p);
}

/*  Proper intersection test (segments share an interior point, no endpoints).  */
inline bool seg_proper(P a, P b, P c, P d) {
    int o1 = sgn(orient(a, b, c)), o2 = sgn(orient(a, b, d));
    int o3 = sgn(orient(c, d, a)), o4 = sgn(orient(c, d, b));
    return o1 * o2 < 0 && o3 * o4 < 0;
}


/*  Full intersection test { returns true if the two closed segments intersect
    (including touching at endpoints or overlapping collinear segments).  */
inline bool seg_intersect(P a, P b, P c, P d) {
    if (seg_proper(a, b, c, d)) return true;
    return on_seg(a, b, c) || on_seg(a, b, d) ||
           on_seg(c, d, a) || on_seg(c, d, b);
}
```

10.4 Polygons

Polygon: a closed figure formed by a sequence of segments. Represented as an ordered list of points (clockwise/counterclockwise).

Simple polygon: A polygon that does not intersect itself and contains no holes. Unless stated otherwise, we refer only to simple polygons.

Convex polygon: A polygon is convex if for any two points inside it, the line segment connecting them lies entirely within the polygon.

Internal Angle Sum: for any polygon with n vertices, the sum of internal angles is $180^\circ \cdot (n-2)$.

Area

A simple polygon P has n vertices
$P_1 = (x_1, y_1), (x_2, y_2), \ldots, P_n = (x_n, y_n)$,
where each vertex (x_i, y_i) is adjacent to (x_{i+1}, y_{i+1}) for $i = 1, 2, \ldots, n-1$, and (x_n, y_n) is adjacent to (x_1, y_1).
The area can be computed using:

$$\text{Area}(P) = \frac{1}{2} \sum_{i=1}^{n} P_i \times P_{i+1}$$

```cpp
pair<ll, bool> polygon_area(const vector<Point>& pts) {
    int n = pts.size();
    __int128 acc = 0;
    for (int i = 0; i < n; i++) {
        int j = (i + 1) % n;
        acc += cross(pts[i], pts[j]);
    }
    if (acc < 0) acc = -acc;
    // acc is twice the area.
    ll whole = (ll)(acc / 2);
    bool half = (acc % 2 != 0);
    return {whole, half};
}
```

Convexity Check

You are given a polygon defined by its vertices in clockwise order. A polygon is convex if for any two points inside or on the boundary, the segment connecting them lies entirely inside or on the polygon.

A polygon is convex iff when walking on it clockwise, we make only right turns.

Point Inside Polygon

Given a polygon and a point, determine if the point lies inside, on the boundary, or outside the polygon.

Idea – Ray Casting Method:

1. Imagine shooting a horizontal ray to the right from the query point.

2. Count how many times this ray intersects the polygon's edges.

3. The rules:

- Odd number of intersections → Inside

- Even number of intersections → Outside

4. Boundary check:

- If the point lies exactly on an edge, it's on the boundary. - Check by verifying collinearity and that the point lies within the edge's bounding box.

Special Cases to Handle:

- Point lies exactly on a vertex or edge.

- Ray passes through vertices: count each edge consistently to avoid double counting.

```cpp
inline __int128 orient(const Point& a, const Point& b, const Point& c) {
    // (b - a) x (c - a)
    return cross(Point{b.x - a.x, b.y - a.y}, Point{c.x - a.x, c.y - a.y});
}

bool on_segment(const Point& a, const Point& b, const Point& p) {
    return min(a.x, b.x) <= p.x && p.x <= max(a.x, b.x)
        && min(a.y, b.y) <= p.y && p.y <= max(a.y, b.y);
}

enum class Position { BOUNDARY, INSIDE, OUTSIDE };

Position point_in_polygon(const vector<Point>& poly, const Point& q) {
    int n = sz(poly);
    bool boundary = false;
    int crossings = 0;

    rep(i, 0, n) {
        int j = (i + 1) % n;
        const Point& A = poly[i];
        const Point& B = poly[j];

        // boundary: collinear and within segment
        if (orient(q, A, B) == 0 && on_segment(A, B, q)) {
            boundary = true;
```

```cpp
                break;
        }

        // ray cast to the right: check if edge crosses horizontal line at q.y
        if ((A.y > q.y) != (B.y > q.y)) {
            // compute intersection x-coordinate
            long double x_int = A.x + (long double)(B.x - A.x) * (q.y - A.y) / (long double)(B.y - A.y);
            if (q.x < x_int) crossings++;
        }
    }


    if (boundary) return Position::BOUNDARY;
    if (crossings & 1) return Position::INSIDE;
    return Position::OUTSIDE;
}
```

10.5 Convex Hull

Idea: Given n points on the plane, find the smallest convex polygon that contains all the given points.

```cpp
struct P {
    ll x, y;
    bool operator<(P p) const { return tie(x, y) < tie(p.x, p.y); }
    P operator-(P p) const { return {x - p.x, y - p.y}; }
};
inline ll cross(P a, P b) { return a.x * b.y - a.y * b.x; }
inline ll orientation(P a, P b, P c) { return cross(b - a, c - a); }

vector<P> convex_hull(vector<P> v) { // convex chain, CCW
    sort(v.begin(), v.end());
    vector<P> h;
    for (int k = 0; k < 2; ++k) {
        size_t start = h.size();
        for (P p : v) {
            while (h.size() >= start + 2 &&
                   orientation(h[h.size() - 2], h.back(), p) <= 0)
                h.pop_back();
            h.push_back(p);
        }
        h.pop_back(); // last point repeats
        reverse(v.begin(), v.end());
    }
    return h;
}
```

Angles:

```cpp
const ld PI = acosl(-1.0L);
inline ld deg2rad(ld deg) { return deg * PI / 180.0L;}
inline ld rad2deg(ld rad) { return rad * 180.0L / PI;}

struct P {
    ll x, y;
    bool operator<(P p) const { return tie(x, y) < tie(p.x, p.y); }
    P operator-(P p) const { return {x - p.x, y - p.y}; }
};
inline ll cross(P a, P b) { return a.x * b.y - a.y * b.x; }
inline ld dot(P a, P b) { return (ld)a.x * b.x + (ld)a.y * b.y; }

ld angle(P a, P b, P c) { // at vertex b
    P u = a - b, v = c - b;
    return fabsl(atan2l(cross(u,v), dot(u,v))); // radians
}
```

Exercise 5 – Problem A – Aching Rotation

- **Goal**: Stand at one point, so the head turn angle required to see all others is minimal.
- **Key Observations**:
  - Optimal child = vertex with smallest interior angle on the convex hull.
  - If hull ≤ 2 points (collinear/trivial) $\Rightarrow$ answer $= 0°$.
- **Solution:**
  1. build convex hull (convex chain) $O(n \log n)$
  2. scan vertices, keep min interior angle $O(\text{hull size})$
  3. convert rad $\rightarrow$ deg, print

```cpp
int main() {
    cin.tie(0)->sync_with_stdio(0);
    cin.exceptions(cin.failbit);

    int T;
    cin >> T;
    for (int tc = 1; tc <= T; ++tc) {
        int n;
        cin >> n;
        vector<P> pts(n);
        for (auto &p : pts)
            cin >> p.x >> p.y;

        auto H = convex_hull(pts);
        ld ans = 0; // collinear / n<=2
        if (H.size() >= 3) {
            ans = 1e100;
            int m = H.size();
            for (int i = 0; i < m; ++i) {
                ld ang =
                    angle(H[(i + m - 1) % m], H[i], H[(i + 1) % m]); // radians
                ans = min(ans, ang);
            }
            ans = rad2deg(ans); // → degrees
        }
        cout << "Case " << tc << ": " << fixed << setprecision(6) << (double)ans
             << '\n';
    }
}
```

Exercise 5 – Problem B – Border Line

- **Goal:** Determine if two finite point-sets in the plane be strictly separated by a straight line
- **Key observation**: A single straight line strictly separates two point sets A and B $\iff$ their convex hulls are disjoint, i.e. they share no point, edge or interior region.

- **Solution:**
  1. build the convex hull of each kingdom $O(n+m)$
  2. test whether the two convex polygons intersect or one lies inside the other;
  3. output YES when they are disjoint, NO otherwise.

```cpp
bool separable(vector<P> A, vector<P> B) {
    auto H1 = convex_hull(A), H2 = convex_hull(B);

    /* 1 any edge contact  NO */
    for (size_t i = 0; i < H1.size(); ++i)
        for (size_t j = 0; j < H2.size(); ++j)
            if (seg_intersect(H1[i], H1[(i + 1) % H1.size()], H2[j],
                                  H2[(j + 1) % H2.size()]))
                return false;

    /* 2 one hull strictly surrounds the other  NO */
    if (in_convex(H1, H2[0]) || in_convex(H2, H1[0]))
        return false;

    /* otherwise hulls are disjoint  YES */
    return true;
}

int main() {
    cin.tie(0)->sync_with_stdio(0);
    cin.exceptions(cin.failbit);
    int n, m;
    while (cin >> n >> m, n || m) {
        vector<P> A(n), B(m);
        for (auto &p : A)
            cin >> p.x >> p.y;
        for (auto &p : B)
            cin >> p.x >> p.y;
        cout << (separable(A, B) ? "YES\n" : "NO\n");
    }
    return 0;
}
```

Exercise 5 – Problem D – Darts

- **Goal:** Find the lexicographically-smallest ordering of 7 points that forms a simple polygon whose area A gives

$$p = \left(\frac{A}{4}\right)^3$$

- **Observartions:**
  - We can brute force. Polygon is unchanged by cyclic shifts → fix first vertex (index 0 to get least permuation) and permute the other six → $6! = 720$ candidates.
  - Simplicity: no pair of non-adjacent edges may intersect ($O(7^2)$ checks).
  - Area: **shoelace formula**; accept if $\left|\left(\frac{A}{4}\right)^3 - p\right| \le \varepsilon$
- **Solution:**
  1. Let idx = {0,1,2,3,4,5,6}; iterate next_permutation(idx+1, idx+7).
  2. For each ordering:
     (a) reject if polygon not simple;
     (b) compute A; if area matches, print indices+1 and stop

```cpp
void solve() {
  for (int i = 0; i < N; i++) {
    ld x, y;
    cin >> x >> y;
    P[i] = {x, y};
  }
  double prob_in;
  cin >> prob_in;

  array<int, N> perm;
  iota(perm.begin(), perm.end(), 0);

  array<int, N> res{};

  do {
    if (!is_polygon_simple(perm))
      continue;

    ld area = polygon_area(perm);
    if (abs(pow(area * 0.25, 3) - prob_in) < AREA_EPS) {
      res = perm;
      break;
    }
  } while (next_permutation(perm.begin() + 1,
                             perm.end())); // fix first point permute rest

  for (int i = 0; i < N; i++) {
    cout << res[i] + 1 << (i + 1 == N ? '\n' : ' ');
  }
}

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

Property: Let P be a simple polygon with vertices $p_0, p_1, ..., p_{n-1}$ (either all clockwise or all counter-clockwise).
$\text{Area}(P) = \frac{1}{2}\sum_{i=1}^{n-1} P_i \times P_{i+1}$.
Use to compare areas without sqrts.

Sweep Line

Idea: maintain a line that sweeps through the entire plane and solve the problem locally.

10.6 Comp Geometry Library

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

using ll = long long;
using ld = long double;
using i128 = __int128_t; // 128-bit for overflow-safe cross
const ld PI = acosl(-1.0L);

inline ld deg2rad(ld deg) { return deg * PI / 180.0L; }
```

```cpp
inline ld rad2deg(ld rad) { return rad * 180.0L / PI; }
template <class T> inline int sgn(T v) { return (v > 0) - (v < 0); }

struct P {
    ll x, y;
    bool operator<(P p) const { return tie(x, y) < tie(p.x, p.y); }
    bool operator==(P o) const { return x == o.x && y == o.y; }
    P operator-(P p) const { return {x - p.x, y - p.y}; }
};

/* pretty-print for debugging */
inline ostream &operator<<(ostream &os, const P &p) {
    return os << "(" << p.x << "," << p.y << ")";
}

inline i128 cross(P a, P b) { return (i128)a.x * b.y - (i128)a.y * b.x; }
inline i128 orientation(P a, P b, P c) { return cross(b - a, c - a); }
inline ld dot(P a, P b) { return (ld)a.x * b.x + (ld)a.y * b.y; }

ld angle(P a, P b, P c) { // // abc (0..)
    P u = a - b, v = c - b;
    return fabsl(atan2l(cross(u, v), dot(u, v))); // radians
}

/* ---------- convex hull (Andrew) ----------
   keep_collinear=false → strict hull (drops collinear)
   keep_collinear=true  → keeps boundary collinear pts   */
vector<P> convex_hull(vector<P> v) { // convex chain, CCW
    sort(v.begin(), v.end());
    v.erase(unique(v.begin(), v.end()), v.end());
    if (v.size() <= 2)
        return v; // <-- early-return for 0/1/2 pt cases
    vector<P> h;
    for (int k = 0; k < 2; ++k) {
        size_t start = h.size();
        for (P p : v) {
            // if want keep collinear than strict <
            while (h.size() >= start + 2 &&
                   orientation(h[h.size() - 2], h.back(), p) <= 0)
                h.pop_back();
            h.push_back(p);
        }
        h.pop_back(); // last point repeats
        reverse(v.begin(), v.end());
    }
    return h;
}

/*  Strictly inside the axis-aligned bounding box with diagonal ab.  */
inline bool in_box(P a, P b, P p) {
    return min(a.x, b.x) <= p.x && p.x <= max(a.x, b.x) &&
           min(a.y, b.y) <= p.y && p.y <= max(a.y, b.y);
}
/*  Closed *disk* whose diameter is ab (i.e. all points p s.t.
        apb  90°  iff  (ap)·(bp) <= 0 ).  */
inline bool in_disk(P a, P b, P p) {
    return dot(a - p, b - p) <= 0; // <= keeps endpoints
}
/*  Checks whether point p lies *on* the (closed) segment ab.  */
inline bool on_seg(P a, P b, P p) {
    return orientation(a, b, p) == 0 && in_box(a, b, p);
}

/*  Proper intersection test (segments share an interior point, no endpoints).
 */
inline bool seg_proper(P a, P b, P c, P d) {
    int o1 = sgn(orientation(a, b, c)), o2 = sgn(orientation(a, b, d));
    int o3 = sgn(orientation(c, d, a)), o4 = sgn(orientation(c, d, b));
    return o1 * o2 < 0 && o3 * o4 < 0;
}

/*  Full intersection test { returns true if the two closed segments intersect
    (including touching at endpoints or overlapping collinear segments). */
inline bool seg_intersect(P a, P b, P c, P d) {
    if (seg_proper(a, b, c, d))
        return true;
    return on_seg(a, b, c) || on_seg(a, b, d) || on_seg(c, d, a) ||
           on_seg(c, d, b);
}

/* O(|poly|) test: all turns have same sign  p inside/on boundary */
bool in_convex(const vector<P> &poly, P p) {
    int n = poly.size(), sign = 0;
    if (n == 1)
        return p == poly[0];
    if (n == 2)
        return on_seg(poly[0], poly[1], p);

    for (int i = 0; i < n; ++i) {
        P a = poly[i], b = poly[(i + 1) % n];
        long long v = orientation(a, b, p);

        if (v == 0) { // collinear  inside *only* when
            if (!on_seg(a, b, p))
                return false; // really on the edge
            continue;          // otherwise keep checking
        }
        if (!sign)
            sign = sgn(v); // remember first non-zero side
        else if (sign != sgn(v))
```

```cpp
            return false; // point lies on different sides
    }
    return true; // strictly inside or on boundary
}

bool separable(vector<P> A, vector<P> B) {
    auto H1 = convex_hull(A), H2 = convex_hull(B);

    /* any edge contact  NO */
    for (size_t i = 0; i < H1.size(); ++i)
        for (size_t j = 0; j < H2.size(); ++j)
            if (seg_intersect(H1[i], H1[(i + 1) % H1.size()], H2[j],
                              H2[(j + 1) % H2.size()]))
                return false;

    /* one hull strictly surrounds the other  NO */
    if (in_convex(H1, H2[0]) || in_convex(H2, H1[0]))
        return false;

    /* otherwise hulls are disjoint  YES */
    return true;
}

// Shoelace -- returns **unsigned** area.
ld polygon_area(const vector<P> &poly) {
    i128 twice = 0;
    int n = poly.size();
    for (int i = 0; i < n; i++)
        twice += cross(poly[i], poly[(i + 1) % n]);
    return fabsl((ld)twice) / 2.0L; // |area|
}

// strict simplicity test (no self-intersections)
bool simple_polygon(const vector<P> &v) {
    int n = v.size();
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            P a = v[i], b = v[(i + 1) % n], c = v[j], d = v[(j + 1) % n];
            // skip adjacent edges sharing a vertex
            if (a == c || a == d || b == c || b == d)
                continue;
            if (seg_intersect(a, b, c, d))
                return false;
        }
    }
    return true;
}


struct Pd { ld x,y; };    // floating-point point

/*  Intersection point of lines ab and cd. Assumes they are not parallel. */
Pd line_intersection_ld(P a,P b,P c,P d){
    i128 A1= orientation(c,d,a);   // signed 2*area of cda
    i128 A2= orientation(c,d,b);
    ld t=(ld)A1/(ld)(A1-A2);       // fraction along ab
    return { a.x + t*(b.x-a.x),  a.y + t*(b.y-a.y) };
}


template <class T>
bool interval_intersection(T a, T b, T c, T d, pair<T,T>& out){
    if(a>b) swap(a,b);
    if(c>d) swap(c,d);
    T L = max(a,c), R = min(b,d);
    if(L > R) return false;          // empry set
    out = {L,R};                     // [L,R] (closed)
    return true;
}

/* Convenience overload that just answers \do they overlap?" */
template <class T>
inline bool intervals_overlap(T a,T b,T c,T d){
    if(a>b) swap(a,b);
    if(c>d) swap(c,d);
    return max(a,c) <= min(b,d);
}
```

11 MSTs

A **cut** in a graph is a partition of the vertex set V into two non-empty disjoint subsets S and $V \setminus S$. The set of edges that have one endpoint in S and one in $V \setminus S$ is called the **cut-set** of the cut.

A **Minimum Spanning Tree (MST)** of an undirected weighted graph is a subgraph that

- connects all vertices,
- contains exactly $n - 1$ edges (for n vertices),
- has the minimal possible total edge weight.

Properties:

- The heaviest edge in a cycle will not be in any MST.
- The lightest edge in a cut will be in every MST.
- All MSTs have the same number of edges of each weight.
- If the weights are unique, there is a unique MST.

There is also a corresponding definition of a **Maximum Spanning Tree**, where the goal is to maximize the total weight instead (to find it, use the same algorithms and negate the weight).

11.1 Prim's Algorithm

Idea: Start from an arbitrary node. Repeatedly add the lightest edge that connects a visited node to an unvisited node.

- Use a `min-heap` (priority queue) to efficiently select the next lightest edge.
- Maintain a `visited[]` array to track which nodes are already included in the MST.

```cpp
struct Edge {
  // weight, target, source, edge idx
```

```cpp
  ll w, t, s, id;
  bool operator<(const Edge& o) const {
    return w > o.w;
  } // min heap
};

// { t, w, idx }
using graph = vector<vector<array<ll,3>>>;

vector<Edge> prim(ll n, const graph& g) {
  vector<bool> vis(n+1, false);
  priority_queue<Edge> pq;
  vector<Edge> mst;
  pq.push({0,0,-1,-1}); // start from 0

  while(!pq.empty() && mst.size() < n-1) {
    Edge e = pq.top(); pq.pop();
    if (vis[e.t]) continue;
    vis[e.t] = true;
    // skip fake edge
    if (e.s != -1) mst.push_back(e);
    for (auto [v, w, id] : g[e.t])
      if (!vis[v]) pq.push({w,v,e.t,id});
  }
  return mst;
}
```

11.2 Kruskal's Algorithm

Idea: Sort all edges by weight and process them in increasing order.

- Add an edge to the MST if it connects two different components (i.e., does not create a cycle).
- Use a `Union-Find (Disjoint Set Union)` data structure to detect cycles efficiently.

```cpp
struct Edge {
  // weight, target, source, edge idx
  ll w, t, s, id;
  bool operator<(const Edge& o) const {
    return w < o.w;
  }
};

vector<Edge> kruskal(ll n, vector<Edge>& edges)
 ↪ {
  sort(edges.begin(), edges.end());
  // As defined above!
  UnionFind uf(n);
  vector<Edge> mst;
  mst.reserve(n-1);

  for(Edge& e: edges) {
    if (uf.merge(e.s, e.t)) {
      mst.push_back(e);
      if (mst.size() == n-1)
        break;
    }
  }
  return mst;
}
```

Application: Given a weighted undirected graph, and two nodes s and t, find a path from s to t such that the **maximum edge weight along the path is minimized**.

This is equivalent to finding the path between s and t in a Minimum Spanning Tree (MST) of the graph.

Solution:

- Compute the MST of the graph (e.g., using Kruskal or Prim).

- The unique path between s and t in the MST minimizes the heaviest edge on the path.

12 Flow

Input: A directed graph with capacities on edges, and a designated source s and sink t.

Output: The maximum flow from s to t that satisfies:

- **Capacity constraint:** Flow on any edge $\leq$ capacity.
- **Flow conservation:** For all nodes except s and t, incoming flow = outgoing flow.

Modeling Variants:

- **Vertex capacities:** Simulate by splitting each vertex into two nodes with an edge of given capacity between them.
- **Multiple sources/sinks:** Add a super-source connected to all sources, and a super-sink receiving from all sinks.

Key Theorems and Reductions:

- **Min-Cut = Max-Flow** (Ford-Fulkerson Theorem)
- **Vertex Cover:** A subset of vertices that touches every edge.
- **Max Independent Set** in bipartite graph = complement of min vertex cover.
- **Matching:** A set of edges with no shared endpoints.
- **König's Theorem:** In bipartite graphs, the size of the maximum matching = size of the minimum vertex cover.
- $\Rightarrow$ **Maximum bipartite matching** can be found via max flow.

Algorithm: Edmonds-Karp (implementation of Ford-Fulkerson using BFS)

- Constructs shortest augmenting paths using BFS.
- Runs in $\mathcal{O}(VE^2)$ time.

Implementation idea:

- Build a residual graph using forward and reverse edges.
- In each iteration, find an augmenting path via BFS.
- Augment flow along the path using the bottleneck edge.
- Repeat until no augmenting path exists.

```cpp
struct Edge { // directed edge
    // destination vertex
    int to;
    // index of the reverse edge in g[to]
    int rev;
    // residual capacity on this directed edge
    ll  cap;
};

// residual graph (adjacency list)
vector<vector<Edge>> g;

void addEdge(int u, int v, ll c) {
```

```cpp
    // inserts to the graph the forward
    // edge u -> v with capacity c
    // and the reverse
    // edge v -> u with capacity 0
    Edge fwd{v, (int)g[v].size(), c};
    Edge rev{u, (int)g[u].size(), 0};
    g[u].push_back(fwd);
    g[v].push_back(rev);
}

// build shortest (by edges) augmenting path
bool bfs(int s, int t, vi& parV, vi& parE) {
    parV.assign(g.size(), -1);
    parE.assign(g.size(), -1);
    queue<int> q;
    parV[s] = s;
    q.push(s);

    while (!q.empty() && parV[t] == -1) {
        int u = q.front(); q.pop();
        for (int i = 0;i < (int)g[u].size();i++){
            const Edge& e = g[u][i];
            // if edge can carry flow
            if (e.cap > 0 && parV[e.to] == -1) {
                parV[e.to] = u;
                parE[e.to] = i;
                // if sink reached
                if (e.to == t) return true;
                q.push(e.to);
            }
        }
    }
    return parV[t] != -1;
}

// Compute the bottleneck (min residual capacity)
// on the found path
ll bottleneck(int s, int t, vi& parV, vi& parE) {
    ll delta = INF;
    for (int v = t; v != s; v = parV[v]) {
        Edge& e = g[parV[v]][parE[v]];
        delta = min(delta, e.cap);
    }
    return delta;
}

// Push 'delta' units of flow along the stored path
void push(int s, int t, ll delta, vi& parV, vi& parE) {
    for (int v = t; v != s; v = parV[v]) {
        // forward residual edge
        Edge&  e  = g[parV[v]][parE[v]];
        // corresponding reverse edge
        Edge&  er = g[v][e.rev];
        e.cap  -= delta;
        er.cap += delta;
    }
}

ll maxFlow(int s, int t) {
    ll flow = 0;
    vector<int> parV, parE;

    // Repeat until no augmenting path exists
    while (bfs(s, t, parV, parE)) {
        ll delta = bottleneck(s, t, parV, parE);
        push(s, t, delta, parV, parE);
        flow += delta;
    }
    return flow;
}

```

TIP: Understand the flow algorithm, but in many problems the main challenge is modeling — i.e., how to reduce the problem to a flow network. Once that's done, standard algorithms (like Ford-Fulkerson) can be applied directly.

13 Probability Examples

13.1 Wish I knew how to sort

You are given a binary array, and repeatedly pick a random pair (i, j) with $i < j$, swapping a_i and a_j if $a_i > a_j$, until the array is sorted. The task is to compute the expected number of swaps needed to sort the array under this process. The expected value must be output as a modular fraction $p \cdot q^{-1} \bmod 998244353$, where p/q is the reduced fraction of the expectation.

Solution:

The solution models the process as a sequence of geometric random variables, where each "success" is a swap that moves a 1 from the left of the 0/1 boundary to the right. The expected total steps is the sum of the expected steps for each needed swap, using the linearity of expectation. For each step, the success probability is computed as $\frac{\#(\text{ of 1s left})\times\#(\text{of 0s right})}{\binom{n}{2}}$, and summing the reciprocals of these probabilities yields the expected number of operations.

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

const ll MOD = 998244353;

ll mod_inv(ll v) {
  ll pow = MOD - 2;
  ll res = 1;
  while (pow > 0) {
    if (pow & 1)
      res = (res * v) % MOD;
    v = (v * v) % MOD;
    pow >>= 1;
  }
  return res;
}

int main () {
  ll t, n;
  cin >> t;

  for (ll i = 0; i < t;i++) {
    cin >> n;
    vector<int> arr (n, 0);
    ll ones = 0;
    for (ll j = 0; j < n;j++) {
      cin >> arr[j];
      if (arr[j] == 1){
        ones += 1;
      }
    }

    ll mis_ones = 0;
```

```cpp
    for (ll j = 0; j < (n-ones); j++) {
      if (arr[j] == 1) {
        mis_ones += 1;
      }
    }

    ll common_mult = 1;
    ll j_sum = 0;
    ll num = (n * (n-1)) % MOD;
    for (ll j = 0; j < mis_ones; j++) {
      ll j_squared = (j+1) * (j+1) % MOD;
      common_mult = common_mult *  j_squared % MOD;
      j_sum = (j_sum + mod_inv(j_squared)) % MOD;
    }
    j_sum %= MOD;
    common_mult %= MOD;
    ll den =  2 * common_mult % MOD;

    ll sum_num = j_sum * common_mult % MOD;
    num =(num *  sum_num) % MOD;

    cout <<  num * mod_inv(den) % MOD << "\n";
  }
  return 0;
}
```

13.2 Last Frame

A geologist keeps sampling until exactly k out of the last n samples contain an exotic mineral (each sample has independent probability p of being exotic). The goal is to compute the expected number of samples she needs to process before this stopping condition is met.

Key Observations:

- The decision to stop depends only on the last n samples, which can be modeled as an n-bit binary state (with up to $2^n \leq 64$ states).

- Once a state has exactly k exotic samples (popcount $= k$), it becomes an absorbing state (i.e. you stop).

- We use Markov chain dynamics and linear equations to model expected values $E[\text{state}]$, and solve them using Gaussian elimination.

Solution:

The solution builds a Markov chain with each state representing a binary mask of the last n samples. We set up an equation for each non-absorbing state: $E[c] = 1 + (1-p) \cdot E[\text{next0}(c)] + p \cdot E[\text{next1}(c)]$ Solving this system (with at most 64 variables) via Gaussian elimination gives the expected number of total samples needed.

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

int main() {
  ios::sync_with_stdio(false);
  cin.tie(nullptr);
  int n, k;
  long double p;
  cin >> n >> k >> p;
  const int N = 1 << n;
  const long double EPS = 1e-12;
  vector<vector<long double>> A(N, vector<long double>(N, 0));
  vector<long double> b(N, 1);
  for (int s = 0; s < N; s++) {
    if (__builtin_popcount(s) >= k) {
      A[s][s] = 1;
      b[s] = 0;
      continue;
    }

    A[s][s] = 1;
    int next0 = (s << 1) & (N - 1);
    int next1 = next0 | 1;
    A[s][next0] -= (1 - p);
    A[s][next1] -= p;
  }

  // Gaussian elimination
  for (int col = 0; col < N; col++) {
    int pivot = col;
    for (int row = col + 1; row < N; row++)
      if (fabsl(A[row][col]) > fabsl(A[pivot][col]))
        pivot = row;
    swap(A[col], A[pivot]);
    swap(b[col], b[pivot]);

    long double div = A[col][col];
    if (fabsl(div) < EPS)
      continue;
    for (int j = col; j < N; j++)
        A[col][j] /= div;
    b[col] /= div;

    for (int row = 0; row < N; row++)
      if (row != col) {
        long double factor = A[row][col];
        if (fabsl(factor) < EPS)
          continue;
        for (int j = col; j < N; j++)
          A[row][j] -= factor * A[col][j];
        b[row] -= factor * b[col];
      }
  }
  cout << fixed << setprecision(12) << b[0] << '\n';
  return 0;
}
```