Resumo do vector:

push_back(valor)
Insere um elemento no final.

pop_back()
Remove o elemento do final (não retorna o valor).

size()
Retorna o número de elementos.

empty()
Retorna true se o vetor está vazio.

clear()
Remove todos os elementos.

front() / back()
Retorna referência ao primeiro/último elemento.

Acesso por índice:
Permite acessar/modificar qualquer elemento com vector[i].

at(i)
Acessa o elemento na posição i com verificação de limites.

insert(it, valor)
Insere elemento na posição indicada pelo iterador.

erase(it)
Remove elemento na posição indicada pelo iterador.

resize(n)
Altera o tamanho do vetor.

emplace_back(args...)
Constrói e insere elemento diretamente no final.

Resumo da stack:

push(valor)
Insere um elemento no topo da pilha.

pop()
Remove o elemento do topo (não retorna o valor).

top()
Retorna uma referência ao elemento do topo, mas sem remover

size()
Retorna o número de elementos.

empty()
Retorna true se a pilha está vazia.

emplace(args...)
Constrói e insere o elemento diretamente no topo.

Observações
Estrutura LIFO (Last In, First Out).
Não permite acesso direto aos outros elementos, só ao topo.
Não permite remoção de elementos arbitrários, só do topo.
Operações de inserção, remoção e acesso ao topo são O(1).

Resumo da queue:

push(valor)
Insere um elemento no final da fila.

pop()
Remove o elemento do início da fila (não retorna o valor).

front()
Retorna uma referência ao elemento do início da fila.

back()
Retorna uma referência ao elemento do final da fila.

size()
Retorna o número de elementos.

empty()
Retorna true se a fila está vazia.

emplace(args...)
Constrói e insere o elemento diretamente no final da fila.

Observações
Estrutura FIFO (First In, First Out).
Não permite acesso direto aos outros elementos, só ao início (front()) e ao final (back()).
Não permite remoção de elementos arbitrários, só do início.
Operações de inserção, remoção e acesso ao início/final são O(1).

Resumo do deque:

push_front(valor)
Insere um elemento no início.

push_back(valor)
Insere um elemento no final.

pop_front()
Remove o elemento do início (não retorna o valor).

pop_back()
Remove o elemento do final (não retorna o valor).

front()
Retorna uma referência ao elemento do início.

back()
Retorna uma referência ao elemento do final.

size()
Retorna o número de elementos.

empty()
Retorna true se o deque está vazio.

clear()
Remove todos os elementos.

emplace_front(args...) / emplace_back(args...)
Constrói e insere o elemento diretamente no início ou final.

Acesso por índice:
Permite acessar qualquer elemento com deque[i].

Observações
Permite inserção e remoção eficiente tanto no início quanto no final (O(1)).
Permite acesso por índice como um vetor.
Útil para implementar filas com acesso nas duas extremidades.

```cpp
#include <iostream>
#include <vector>
#include <stack>
#include <utility>
#include <iomanip>
#include <algorithm>
using namespace std;

template <typename T>
class MonoStack
{
public:
    MonoStack(int start_idx = 0, int inc = 1) : pos(start_idx), invalid(start_idx - inc), inc(inc) {}
    int push(const T &x)
    {
        while (not st.empty() and st.top().second <= x) // Pilha não-crescente
            st.pop();

        auto i = st.empty() ? invalid : st.top().first;
        st.emplace(pos, x);
        pos += inc;

        return i;
    }

    void pop() { st.top(); };
    auto top() const { return st.top(); }
    bool empty() const { return st.empty(); }

private:
    stack<pair<int, T>> st;
    int pos, invalid, inc;
};

// pge = previous greater element
auto pge(const vector<int> &xs)
{
    MonoStack<int> ms;
    vector<int> ans;

    for (auto x : xs)
        ans.emplace_back(ms.push(x));

    return ans;
}

// nge = next greater element
// Faz o mesmo que o nge, mas comeca da direita para a esquerda
auto nge(const vector<int> &xs)
{
    MonoStack<int> ms(6,-1);
    vector<int> ans;
    auto temp(xs);
    reverse(temp.begin(), temp.end());

    for (auto x : temp)
        ans.insert(ans.begin(), ms.push(x));

 return ans;
}
```

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

using ll = long long;

// pge usando programacao dinamica
auto pge(const vector<ll> &as)
{
   auto N = (int)as.size();
   vector<ll> dp(N, -1);

   for (int i = 1; i < N; ++i)
   {
      ll j = i - 1;

      while (j >= 0 and as[j] <= as[i]) // Se o elemento anterior for menor que o atual
         j = dp[j];                     // Pulamos para o maior a esqueda dele

      dp[i] = j; // Armazena ou o maior ou uma posicao invalida
   }

   return dp;
}

// ngee(next greater or equal element) usando programacao dinamica
auto ngee(const vector<ll> &as)
{
   auto N = (int)as.size();
   vector<ll> dp(N, N);

   for (int i = N - 2; i >= 0; --i)
   {
      ll j = i + 1;

      while (j < N and as[j] < as[i]) // Se o elemento posterior for menor que o atual
         j = dp[j];                   // Pulamos para o maior a direita dele

      dp[i] = j; // Armazena ou o maior ou uma posicao invalida
   }
   return dp;
}

auto sum_of_subarray_maximums(const vector<ll> &as)
{
   auto N = (int)as.size();
   auto L = pge(as), R = ngee(as);
   ll sum = 0;

   for (int i = 0; i < N; ++i)
      sum += as[i] * (i - L[i]) * (R[i] - i);
   return sum;
}
```

Monoqueue

```
// Menores elementos em uma janela movel de tamanho k
auto mesw(int n, int k, const vector<int> &xs)
{
    deque<int> q;

    // Cria uma fila nao-descrescente para a primeira janela
    for (int i = 0; i < k; ++i)
    {
        while (not q.empty() and xs[q.back()] >= xs[i])
            q.pop_back();

        q.emplace_back(i);
    }

    vector<int> ms{xs[q.front()]};

    for (int L = 0, R = k; R < n; ++L, ++R)
    {
        if (q.front() == L)
            q.pop_front();

        while (not q.empty() and xs[q.back()] >= xs[R])
            q.pop_back();

        q.emplace_back(R);
        ms.emplace_back(xs[q.front()]);
    }
    return ms;
}
```

```
// Implementacao usando ponteiro
template <typename T>

class BinaryTree
{
private:
    struct Node
    {
        T info;
        Node *left, *right;
    };

    Node *root;

public:
    BinaryTree() : root(nullptr) {}
};

// Implementacao usando vetor
#include <vector>

template <typename T>
class BinaryTree
{
private:
    std::vector<T> nodes;

    int left(int p) { return 2 * p; }
    int right(int p) { return 2 * p + 1; }
    int parent(int i) { return i / 2; }

public:
    // O indice zero nao´e utilizado, mas deve estar alocado
    BinaryTree() : nodes(1) {}
};
```

Resumo do map:

insert({chave, valor}) - Insere um par chave-valor. Não insere se a chave já existe.

operator[] (chave)
Acessa ou insere o valor associado à chave. Se a chave não existe, cria com valor padrão.

```
m[1] = "A";        // Insere chave 1 com valor "A"
cout << m[1];      // Imprime "A"
```

erase(chave) / erase(it) - Remove o elemento pela chave ou pelo iterador.

find(chave) - Retorna um iterador para o elemento com a chave (ou end() se não existir).

count(chave) - Retorna 1 se a chave existe, 0 se não.

size() - Retorna o número de elementos. O(1)

empty() - Retorna true se o map está vazio. O(1)

clear() - Remove todos os elementos.

begin() / end() - Iteradores para o início e o fim do map.

lower_bound(chave) - Iterador para o primeiro elemento ≥ chave.

upper_bound(chave) - Iterador para o primeiro elemento > chave.

emplace(chave, valor) - Insere o par construindo direto no map (mais eficiente para tipos complexos).

Observações:
Elementos são sempre ordenados pela chave.
Não permite chaves repetidas.
Operações de busca, inserção e remoção são O(log n).
Chave e valor podem ser de tipos diferentes (paramétricos).

Dica:
Para acessar o menor chave: map.begin()->first
Para acessar o maior chave: map.rbegin()->first
Para acessar o valor: it->second (onde it é um iterador)

Resumo de multimap:

insert({chave, valor}) - Insere um par chave-valor. Permite chaves repetidas.

erase(chave) / erase(it) - Remove todos os pares com a chave ou apenas o elemento apontado pelo iterador.

find(chave) - Retorna um iterador para o primeiro elemento com a chave (ou end() se não existir).

count(chave)- Retorna quantos elementos têm a chave.

size()- Retorna o número de elementos.

empty()- Retorna true se o multimap está vazio.

clear()- Remove todos os elementos.

begin() / end() - Iteradores para o início e o fim do multimap.

lower_bound(chave) - Iterador para o primeiro elemento ≥ chave.

upper_bound(chave) - Iterador para o primeiro elemento > chave.

equal_range(chave) - Retorna um par de iteradores para o intervalo de todos os elementos com a chave.

emplace(chave, valor) - Insere o par construindo direto no multimap.

Observações
Elementos são ordenados pela chave.
Permite múltiplos valores para a mesma chave.
Não suporta acesso por índice (mm[chave] não funciona).
Operações de busca, inserção e remoção são O(log n).

Dica:
Para percorrer todos os valores de uma chave:

```
auto [it1, it2] = mm.equal_range(chave);
for (auto it = it1; it != it2; ++it)
    cout << it->second << " ";
```

Resumo do Set:

insert(valor) - Insere um elemento. Não permite duplicados.

erase(valor) / erase(it) - Remove um elemento pelo valor ou pelo iterador.

find(valor) - Retorna um iterador para o elemento igual a valor (ou end() se não existir).

count(valor) - Retorna 1 se o elemento existe, 0 se não.

size() - Retorna o número de elementos. O(1)

empty() - Retorna true se o set está vazio. O(1)

clear() -Remove todos os elementos.

begin() / end() - Iteradores para o início e o fim do set.

lower_bound(valor)- Iterador para o primeiro elemento ≥ valor.

upper_bound(valor)- Iterador para o primeiro elemento > valor.

emplace(args...)- Insere elemento construindo direto no set (mais eficiente para tipos complexos).

Observações
Elementos são sempre ordenados.
Não permite elementos repetidos.
Operações de busca, inserção e remoção são O(log n).

Dica:
Para acessar o menor elemento: *s.begin()
Para acessar o maior elemento: *prev(s.end()) ou *--s.end()

Resumo do multiset:

insert(valor) - Insere um elemento. Permite elementos repetidos.

erase(valor) / erase(it) - Remove todos os elementos iguais a valor ou apenas o elemento apontado pelo iterador.

find(valor) - Retorna um iterador para um elemento igual ao valor (ou end() se não existir).

count(valor) - Retorna quantas vezes valor aparece no multiset.

size() - Retorna o número de elementos. O(1)

empty() - Retorna true se o multiset está vazio. O(1)

clear() - Remove todos os elementos.

begin() / end() - Iteradores para o início e o fim do multiset.

lower_bound(valor) - Iterador para o primeiro elemento ≥ valor.

upper_bound(valor) - Iterador para o primeiro elemento > valor.

equal_range(valor) - Retorna um par de iteradores para o intervalo de todos os elementos iguais a valor.

emplace(args...) - Insere elemento construindo direto no multiset (mais eficiente para tipos complexos).

Observações
Elementos são sempre ordenados.
Permite elementos repetidos.
Operações de busca, inserção e remoção são O(log n).

Dica:
Para acessar o menor elemento: *ms.begin()
Para acessar o maior elemento: *prev(ms.end()) ou *--ms.end()

```
#include <bits/stdc++.h>
```

Set com estatisticas

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using namespace std;
using namespace __gnu_pbds;

typedef tree<
    int,
    null_type,
    less<int>,
    rb_tree_tag,
    tree_order_statistics_node_update>
    ordered_set;

int main()
{
    ordered_set s;

    s.insert(2);
    s.insert(3);
    s.insert(5);
    s.insert(7);
    s.insert(11);
    s.insert(13);

    cout << "Segundo primo: " << *s.find_by_order(1) << '\n';
    cout << 11 << " ´e o " << s.order_of_key(11) + 1 << "ºprimo\n";

    return 0;
}
```

Multiset com estatisticas

```
#include <bits/stdc++.h>

#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using namespace std;
using namespace __gnu_pbds;

typedef tree<
    int,
    null_type,
    less_equal<int>,
    rb_tree_tag,
    tree_order_statistics_node_update>
    ordered_multiset;

int main()
{
    ordered_multiset s;

    s.insert(1);
    s.insert(2);
    s.insert(3);
    s.insert(3);
    s.insert(3);

    // s = 1 2 2 3 3 3
    for (auto x : s)
        cout << x << ' ';
    cout << '\n';

    s.erase(s.find_by_order(s.order_of_key(3)));

    // s = 1 2 2 3 3
    for (auto x : s)
        cout << x << ' ';
    cout << '\n';

    return 0;
}
```

```
// Tinha que construir a arvore e rodar as travessias pos,pre e in
#include <iostream> using namespace std;

struct BST
{
    struct Node
    {
        int info; Node *left, *right;
    };
    Node *root;
    BST() : root(nullptr) {}
    void inorder(const Node *node) const
    {
        if (node)
        {
            inorder(node->left);
            cout << ' '<< node->info;
            inorder(node->right);
        }
    }
    void preorder(const Node *node) const
    {
        if (node)
        {
            cout << ' '<< node->info;
            preorder(node->left);
            preorder(node->right);
        }
    }

    void postorder(const Node *node) const
    {
        if (node)
        {
            postorder(node->left);
            postorder(node->right);
            cout << ' '<< node->info;
        }
    }
    void insert(int info)
    {
        Node **node = &root;

        while (*node)
        {
            if ((*node)->info == info)
                return;
            else if (info < (*node)->info)
                node = &(*node)->left;
            else
                node = &(*node)->right;
        }

        *node = new Node{info, nullptr, nullptr};
    }
};
```

```
// Recebe a arvore no percurso em ordem e preordem, monta ela
// e depois imprime no posordem

void insert(char info, const int rank[])
    {
        Node **node = &root;

        while (*node)
        {
            if ((*node) -> info == info)
                return;
            else if (rank[info - 'A'] < rank[(*node)->info - 'A'])
                node = &(*node)->left;
            else
                node = &(*node)->right;
        }

        *node = new Node{info, nullptr, nullptr};
    }
};

int main()
{
    string preorder, inorder;

    while (cin >> preorder >> inorder)
    {
        int rank[30], nxt = 1;

        for (const auto &c : inorder)
            rank[c - 'A'] = nxt++;

        BST tree;

        for (const auto &c : preorder)
            tree.insert(c, rank);

        tree.postorder(tree.root);
        cout << "\n";
    }
```

```
// Questao de a partir de N chegar em M atraves
// De operacoes especificas
#include <bits/stdc++.h>

using namespace std;
using ii = pair<int, int>;

int solve(int N, int M)
{
    queue<ii> ns;
    set<int> found;

    ns.push(make_pair(N, 0));
    found.insert(N);

    while (not ns.empty())
    {
        auto [n, ops] = ns.front();
        ns.pop();

        if (n == M)
            return ops;
        vector<int> xs{n * 2, n * 3, n / 2, n / 3, n + 7, n - 7};

        for (auto x : xs)
        {
            if (found.count(x) == 0)
            {
                ns.push(ii(x, ops + 1));
                found.insert(x);
            }
        }
    }

    return -1;
}

int main()
{
    int N, M;
    cin >> N >> M;

    auto ans = solve(N, M);

    cout << ans << endl;
    return 0;}
```

```
// Questao do cavaleiros
#include <bits/stdc++.h>
using namespace std;
struct Knight
{
    int p, c, idx;

    bool operator<(const Knight &k) const
    {
        return p < k.p;
    }
};
vector<long long> solve(vector<Knight> &ks, size_t K)
{
    vector<long long> ans(ks.size());
    priority_queue<int> coins;
    long long sum = 0;
    sort(ks.begin(), ks.end());
    for (auto &knight : ks)
    {
        ans[knight.idx] = (knight.c + sum);
        coins.push(-knight.c);
        sum += knight.c;

        if (coins.size() > K)
        {
            auto coin = coins.top();
            coins.pop();

            sum += coin;
        }
    }
    return ans;
}
int main()
{
    ios::sync_with_stdio(false);
    int n, k;   cin >> n >> k; vector<int> ps(n), cs(n);
    for (int i = 0; i < n; ++i)
        cin >> ps[i];

    for (int i = 0; i < n; ++i)
        cin >> cs[i];
    vector<Knight> ks(n);
    for (int i = 0; i < n; ++i)
        ks[i] = Knight{ps[i], cs[i], i};
    auto ans = solve(ks, k);
    for (int i = 0; i < n; ++i)
        cout << ans[i] << (i + 1 == n ? "\n" : " ");
    return 0;}
```

```
#include <bits/stdc++.h>
using namespace std;
bool matrioska(const vector<int>& xs)
{
   stack<int> s;

   for (auto x : xs)
   {
      if (x < 0) // Todo abre eh empilhado
      {
         s.push(x);
      }
      else // Fecha
      {
         // Soma os que ja foram fechados dentro do atual
         int stacked = 0;
         while (not s.empty() and s.top() > 0) {
            stacked += s.top();
            s.pop();
         }

         // Se a soma ultrapassar o elemento atual
         // Ou se nao tiver nada na pilha
         // Ou se o top nao corresponder ao de abrir
         // Retorna falso
         if (stacked >= x or s.empty() or s.top() != -x)
            return false;

         // Quando fecha o brinquedo atual, coloco ele postivo na pilha
         // Assim o proximo que for fechar vai somar ele e
         // Verificar a soma no laco de cima
         s.pop();
         s.push(x);
      }
   }
   while (not s.empty() and s.top() > 0)
      s.pop();
   return s.empty();
}
int main()
{
   ios::sync_with_stdio(false);
   string line;
   while (getline(cin, line))
   {
      istringstream is(line);

      vector<int> xs;

      // lê todos os inteiros de is e os coloca no vetor xs
      copy(istream_iterator<int>(is), istream_iterator<int>(), back_inserter(xs));

      cout << ":-" << (matrioska(xs) ? ") Matrioshka!" : "( Try again.") << '\n';
   }
   return 0;}
```

```
// Questao da quantidade de intercessoes entre as listas
#include <bits/stdc++.h>

using namespace std;

int solve(multiset<int> &s, const multiset<int> &r)
{
    vector<int> xs;
    set_intersection(s.begin(), s.end(), r.begin(), r.end(), back_inserter(xs));

    return s.size() + r.size() - 2 * xs.size();
}

int main()
{
    int T;
    cin >> T;

    while (T--)
    {
        int N, M, x;
        cin >> N >> M;
        multiset<int> s, r;

        while (N--)
        {
            cin >> x;
            s.insert(x);
        }

        while (M--)
        {
            cin >> x;
            r.insert(x);
        }

        cout << solve(s, r) << '\n';
    }

    return 0;      }
```

Resumo da priority_queue:

push(valor) - Insere um elemento na fila de prioridade.

pop() - Remove o elemento do topo (maior ou menor, dependendo do comparador).

top()- Retorna o elemento do topo (maior ou menor).

size() - Retorna o número de elementos.

empty() - Retorna true se a fila está vazia.

Construtores

priority_queue<T>: heap de máximo (maior elemento no topo).
priority_queue<T, vector<T>, greater<T>>: heap de mínimo (menor elemento no topo).

Observações
Não permite acesso direto aos outros elementos, só ao topo.
Não permite remoção de elementos arbitrários, só do topo.
Operações de inserção e remoção são O(log n).
Útil para algoritmos que precisam sempre do maior (ou menor) elemento rapidamente.