**Documentație Problema** [**XOR Maximă**](https://www.infoarena.ro/problema/xormax)

**Procesul Gândirii:**

Problema: Găsirea secvenței de numere într-un șir care produce valoarea maximă a XOR-ului.

Observație: XOR-ul a două numere poate crește dacă bitul corespunzător este diferit în cele două numere.

**Ideea Principală:**   
Construirea unui trie care să rețină prefixele XOR ale șirului pentru a găsi eficient secvența cu XOR maxim.

**Motivația Structurii de Date:**

Alegerea Arborelui Trie:

* Permite stocarea eficientă a prefixelor XOR în funcție de biti.
* Oferă căutare rapidă pentru a găsi numărul care maximizează XOR-ul când este combinat cu un prefix dat.
* Reduce complexitatea temporală comparativ cu abordările brute-force.

**Complexitate:**

* Complexitatea Temporală:
* Construcția Trie-ului: O(N \* B) unde N este numărul de elemente din șir și B este numărul de biți considerat (în cazul nostru, 21).
* Căutarea XOR-ului Maxim: O(N \* B) deoarece fiecare element necesită traversarea trie-ului pentru a găsi XOR maxim.
* Complexitatea Totală: O(N \* B), eficientă pentru valorile mari ale lui N.
* Complexitatea Spațială: Depinde de numărul de noduri în trie, care este O(N \* B).

**Observații:**

Această abordare este mult mai rapidă comparativ cu metodele brute force, în special pentru șiruri mari.

**#include <iostream>**

**#include <vector>**

**using namespace std;**

**const int BITS = 21;**

**struct TrieNode {**

**TrieNode \*children[2];**

**int index;**

**TrieNode() {**

**children[0] = children[1] = nullptr;**

**index = -1;**

**}**

**};**

**void insert(TrieNode \*root, int num, int index) {**

**TrieNode \*node = root;**

**for (int i = BITS; i >= 0; i--) {**

**int bit = (num >> i) & 1;**

**if (!node->children[bit]) {**

**node->children[bit] = new TrieNode();**

**}**

**node = node->children[bit];**

**}**

**node->index = index;**

**}**

**pair<int, int> maxXorWithIndex(TrieNode \*root, int num) {**

**TrieNode \*node = root;**

**int maxXor = 0;**

**for (int i = BITS; i >= 0; i--) {**

**int bit = (num >> i) & 1;**

**int oppositeBit = bit ^ 1;**

**if (node->children[oppositeBit]) {**

**maxXor |= (1 << i);**

**node = node->children[oppositeBit];**

**} else {**

**node = node->children[bit];**

**}**

**}**

**return make\_pair(maxXor, node->index);**

**}**

**int main() {**

**// ifstream cin("xormax.in");**

**// ofstream cout("xormax.out");**

**int n;**

**cin >> n;**

**vector<int> arr(n);**

**for (int i = 0; i < n; i++) {**

**cin >> arr[i];**

**}**

**TrieNode \*root = new TrieNode();**

**insert(root, 0, -1);**

**int maxXor = 0, maxXorStart = 0, maxXorEnd = 0, prefixXor = 0;**

**for (int i = 0; i < n; i++) {**

**prefixXor ^= arr[i];**

**pair<int, int> result = maxXorWithIndex(root, prefixXor);**

**if (result.first > maxXor) {**

**maxXor = result.first;**

**maxXorStart = result.second + 1;**

**maxXorEnd = i;**

**}**

**insert(root, prefixXor, i);**

**}**

**cout << maxXor << " " << maxXorStart + 1 << " " << maxXorEnd + 1 << endl;**

**return 0;**

**}**

**Documentație Problema** [**NrPits**](https://www.infoarena.ro/problema/nrpits)

**Proces de gandire**

Scop: Trebuie să numărăm „gropi” într-o matrice, definită ca secvențe în care toate elementele interne sunt mai mici decât punctele finale.

**Abordare**

Structura datelor: O stivă este utilizată pentru capacitatea sa de a urmări eficient punctele de început și de sfârșit potențiale ale „gropi”.

**Metodă**:   
Iterațam prin matrice, folosind stiva pentru a identifica și număra „gropi” valide.

**Observații cheie**

Stiva menține elementele într-o secvență descrescătoare.

Un nou „grup” este identificat atunci când elementul curent este mai mare decât vârful stivei.

**Complexitate**

Timp: O(N) - fiecare element este împins și explodat cel mult o dată.

Spațiu: O(N) - în cel mai rău caz, stiva ar putea conține toate elementele.

**Motivul pentru Stack**

**Complexitatea timpului:**   
O(N), unde N este dimensiunea matricei. Fiecare element este împins și explodat cel mult o dată.

**Complexitatea spațiului:**   
O(N) în cel mai rău caz, când stiva ar putea conține toate elementele dacă matricea este strict în scădere

#include <fstream>

#include <vector>

#include <iostream>

#include <stack>

using namespace std;

int countPits(const vector<int>& arr) {

int n = arr.size();

stack<int> s;

int count = 0;

for (int i = 0; i < n; i++) {

while (!s.empty() && arr[i] > arr[s.top()]) {

int top = s.top();

s.pop();

if (!s.empty()) {

count++;

}

}

s.push(i);

}

return count;

}

int main() {

ifstream cin("nrpits.in");

ofstream cout("nrpits.out");

int N;

cin >> N;

vector<int> arr(N);

for (int i = 0; i < N; i++) {

cin >> arr[i];

}

cout << countPits(arr) << endl;

return 0;

}

**Documentație Problema** [**D. Event Dates**](https://codeforces.com/contest/45/problem/D)

**Proces de gândire**

Scop: Trebuie să atribuim date pentru n evenimente istorice, având cunostință de un interval de zile [li, ri] în care fiecare eveniment ar fi putut avea loc. Trebuie să garantăm că fiecare eveniment are loc într-o zi unică, în cadrul intervalului său permis.

**Abordare**

Structura datelor: Folosim un vector pentru a stoca evenimentele și un set pentru a ține evidența zilelor deja utilizate.

**Metodă:**

1- Sortăm evenimentele în funcție de data de sfârșit (ri) a intervalului permis.

2- Pentru fiecare eveniment, alegem cea mai mică dată disponibilă care este în interiorul intervalului său și care nu a fost încă folosită.

3- Alocăm această dată evenimentului și marcăm ziua ca fiind utilizată.

**Observații cheie**

Sortarea evenimentelor după ri permite abordarea evenimentelor cu cele mai restrictive intervale mai întâi.

Utilizarea unui set pentru a ține evidența zilelor deja folosite asigură că fiecare eveniment are loc într-o zi unică.

Dacă pentru un eveniment nu găsim o dată disponibilă în intervalul său, atunci soluția nu este posibilă (dar enunțul problemei garantează existența unei soluții).

Complexitate

Timp: O(n log n) - majoritatea timpului este petrecută sortând evenimentele, iar sortarea are complexitatea O(n log n). Iterarea prin evenimente și alocarea datelor are complexitatea O(n).

Spațiu: O(n) - folosim spațiu suplimentar pentru stocarea evenimentelor și a setului de date utilizate.

Code:

| #include <iostream> #include <vector> #include <algorithm> #include <set> using namespace std;  bool comp(const pair<pair<int, int>, int>& a, const pair<pair<int, int>, int>& b) {  return a.first.second < b.first.second; }  int main() {  int n;  cin >> n;   vector<pair<pair<int, int>, int> > events(n);  for (int i = 0; i < n; ++i) {  cin >> events[i].first.first >> events[i].first.second;  events[i].second = i;  }    sort(events.begin(), events.end(), comp);   vector<int> result(n);  set<int> usedDates;   for (auto& event : events) {  int date;  for (date = event.first.first; date <= event.first.second; ++date) {  if (usedDates.find(date) == usedDates.end()) {  break;  }  }  if (date > event.first.second) {  cout << "No solution possible" << endl;  return 0;  }  result[event.second] = date;  usedDates.insert(date);  }   for (int date : result) {  cout << date << " ";  }  cout << endl;   return 0; } |
| --- |