Competitive Programming Library

Bernardo Flores Salmeron

1. Template

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

using namespace std;

#define INF 1ll << 62
#define pb push_back
#define ii pair<int,int>
#define OK cerr <<"OK"<< endl
#define debug(x) cerr << #x " = " << (x) << endl
#define ff first
#define ss second
#define int long long

signed main () {

  ios_base::sync_with_stdio(false);
  cin.tie(NULL);

}
```

2. Funcoes Interessantes

2.1. GCD

```cpp
int _gcd(int a, int b){
  if(a == 0 || b == 0) return 0;
  else return abs(__gcd(a,b));
}
```

2.2. HIPOTENUSA

```cpp
cout << hypot(3,4); // output: 5
```

2.3. SCANF DE UMA STRING

```cpp
char sentence[]="Rudolph is 12 years old";
char str [20]; int i;
sscanf (sentence,"%s %*s %d",str,&i);
printf ("%s -> %d\n",str,i);
// Output: Rudolph -> 12
```

2.4. PRINTF PARA UMA STRING

```cpp
char buffer [50];
int n, a=5, b=3;
n=sprintf (buffer, "%d plus %d is %d", a, b, a+b);
printf ("[%s] is a string %d chars long\n",buffer,n);
// Output:
// [5 plus 3 is 8] is a string 13 chars long
```

2.5. CLIMITS

```cpp
LONG_MIN -> (-2^31+1) ::  LONG_MAX -> (2^31-1)
ULONG_MAX -> (2^32-1) -> UNSIGNED
LLONG_MIN, LLONG_MAX, ULLONG_MAX
```

2.6. ROTATE (LEFT)

```cpp
Passado o inicio o meio e o fim ele rotaciona de forma que o meio seja o
    novo inicio.
vector<int> arr(n); // 1 2 3 4 5 6 7 8 9
rotate(arr.begin(),arr.begin()+3,arr.end()); //4 5 6 7 8 9 1 2 3
```

2.7. ROTATE (RIGHT)

```cpp
vector<int> arr(n); // 1 2 3 4 5 6 7 8 9
rotate(arr.begin(),arr.begin()+3,arr.end()); //7 8 9 1 2 3 4 5 6
```

2.8. WIDTH

```cpp
cout << width(13);
cout << 100 << endl; // "     100      "
cout.fill('x');
cout.width(13);
cout << 100 << endl; // "xxxxx100xxxxx"
cout << right << 100 << endl; "xxxxxxx100"
```

2.9. INT TO STRING (C++11)

```cpp
int a; string b;
b = to_string(a);
```

2.10. PERMUTAÇÃO

```cpp
int v[] = {1,2,3};
sort(v, v+3);
do {
  cout << v[0] << ' ' << v[1] ' ' << v[2];
} while(next_permutation(v, v+3));
```

2.11. MAIOR E MENOR ELEMENTO NUM VETOR

```cpp
int maior = *max_element(arr.begin(), arr.end());
int menor = *min_element(arr.begin(), arr.end());
// OBS: Retorna iterador
```

2.12. CHECAGEM E TRANSFORMAÇÃO DE CARACTERE

```cpp
#include <cctype>
isdigit(str[i]);//checa se str[i] é número
isalpha(str[i]);//checa se é uma letra
islower(str[i]);//checa minúsculo
isupper(str[i]);//checa maiúsculo
isalnum(str[i]);//checa letra ou número
tolower(str[i]);//converte para minusculo
toupper(str[i]);//converte para maiusculo
```

2.13. SUBSTRING

```cpp
string s = "abcdef";
s.substr(posição inicial, qtd de char(opcional));
string s2 = s.substr(3,2); // s2 = "de"
string s3 = s.substr(2); // s3 = "cdef"
```

2.14. REMOVE REPETIÇÕES CONTÍNUAS NUM VETOR

```
// arr = {10,20,20,20,30,20,20,10}
it = unique(arr.begin(), arr.end());
// arr = {10,20,30,20,10, iterator aponta pra aqui, ...}
arr.resize(distance(arr.begin(), it));
// arr = {10,20,30,20,10}
```

2.15. CHECAGEM DE BITS

```
// OBS: SO FUNCIONA PARA INT (NAO FUNCIONA COM LONG LONG)
__builtin_popcount(int) -> Número de bits ativos;
__builtin_ctz(int) -> Número de zeros à direita
__builtin_clz(int) -> Número de zeros à esquerda
__builtin_parity(int) -> Retorna se a quantidade de uns é ímpar(1) ou par(0)
```

2.16. INT TO BINARY STRING

```
string s = bitset<qtdDeBits>(intVar).to_string();
Ex: x = 10, qtdDeBits = 32;
s = bitset<32>(x).to_string(); // s = 00...0001010
```

2.17. BINARY STRING TO INT

```
int y = bitset<qtdDeBits>(stringVar).to_ulong();
Ex: x = 1010, qtdDeBits = 32;
y = bitset<32>(x).to_ulong(); // y = 10
```

2.18. STRING TO LONG LONG

```
string s = "0xFFFF"; int base = 16;
string::size_type sz = 0;
int ll = stoll(s,&sz,base); //  ll = 65535, sz = 6;
OBS: Não precisa colocar o sz, pode colocar 0; // stoll(s,0,base);
```

2.19. SPLIT FUNCTION

```
// SEPARA STRING POR UM DELIMITADOR
// EX: str=A-B-C split -> x = {A,B,C}
vector<string> split(const string &s, char delim) {
    stringstream ss(s);
    string item;
    vector<string> tokens;
    while (getline(ss, item, delim)) {
        tokens.push_back(item);
    }
    return tokens;
}
int main () {
  vector<string> x = split("cap-one-best-opinion-language", '-');
// x = {cap,one,best,opinion,language};
}
```

2.20. LEITURA NUM ARQUIVO

```
ifstream cin("input.txt");
```

2.21. ESCRITA EM ARQUIVO

```
ofstream cout("output.txt");
```

2.22. CHECAGEM BRUTE FORCE COM SOLUCAO

```
$ g++ -std=c++11 gen.cpp && ./a.out > gen.out && g++ -std=c++11 brute.cpp &&
    (./a.out < gen.in) > brute.out && g++ -std=c++11 sol.cpp && (./a.out <
    gen.in) > sol.out && diff brute.out sol.out
```

3. DP

3.1. PROBLEMA DO TROCO

```
// função que recebe o valor de troco N, o número de moedas disponíveis M,
// e um vetor com as moedas disponíveis arr
// essa função deve retornar o número mínimo de moedas,
// de acordo com a solução com Programação Dinamica.
int num_moedas(int N, int M, int arr[]) {
  int dp[N+1];
  // caso base
  dp[0] = 0;
  // sub-problemas
  for(int i=1; i<=N; i++) {
  // é comum atribuir um valor alto, que concerteza
  // é maior que qualquer uma das próximas possibilidades,
  // sendo assim substituido
  dp[i] = 1000000;
  for(int j=0; j<M; j++) {
    if(i-arr[j] >= 0) {
      dp[i] = min(dp[i], dp[i-arr[j]]+1);
    }
  }
  }
  // solução
  return dp[N];
}
```

3.2. PROBLEMA DA MOCHILA

```
int dp[2001][2001];
int moc(int q,int p,vector<ii> vec) {
  for(int i = 1; i <= q; i++)
  {
    for(int j = 1; j <= p;j++) {
      if(j >= vec[i-1].ff)
        dp[i][j] = max(dp[i-1][j],vec[i-1].ss + dp[i-1][j-vec[i-1].ff]);
      else
        dp[i][j] = dp[i-1][j];
    }
  }
  return dp[q][p];
}
int main(int argc, char *argv[])
{
  int p,q;
  vector<ii> vec;
  cin >> p >> q;
  int x,y;
  for(int i = 0; i < q; i++) {
       cin >> x >> y;
       vec.push_back(make_pair(x,y));
  }
  for(int i = 0; i <= p; i++)
    dp[0][i] = 0;
  for(int i = 1; i <= q; i++)
    dp[i][0] = 0;
  sort(vec.begin(),vec.end());
  cout << moc(q,p,vec) << endl;
}
```

3.3. CATALAN (1, 1, 2, 5, 14, 42, 132, 429) - $O(n)$

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\,n!} = \prod_{k=2}^{n}\frac{n+k}{k} \qquad \text{para } n \geq 0.$$

```
// The first few Catalan numbers for n = 0, 1, 2, 3, ...
// are 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, ...
// Formula Recursiva:
// cat(0) = 0
// cat(n+1) = somatorio(i from 0 to n)(cat(i)*cat(n-i))
//
// Using Binomial Coefficient
// We can also use the below formula to find nth catalan number in O(n) time.
// Formula acima

// Returns value of Binomial Coefficient C(n, k)

int binomialCoeff(int n, int k) {
  int res = 1;

  // Since C(n, k) = C(n, n-k)
  if (k > n - k)
    k = n - k;

  // Calculate value of [n*(n-1)*---*(n-k+1)] / [k*(k-1)*---*1]
  for (int i = 0; i < k; ++i) {
      res *= (n - i);
      res /= (i + 1);
  }

    return res;
}
// A Binomial coefficient based function to find nth catalan
// number in O(n) time
int catalan(int n) {
    // Calculate value of 2nCn
    int c = binomialCoeff(2*n, n);

    // return 2nCn/(n+1)
    return c/(n+1);
}
```

3.4. LONGEST COMMON SUBSEQUENCE - $O(n^2)$

```
void lcs(char *X, char *Y, int m, int n){
   int L[m+1][n+1];
   /* Following steps build L[m+1][n+1] in bottom up fashion. Note that
    L[i][j] contains length of LCS of X[0..i-1] and Y[0..j-1] */
   for (int i=0; i<=m; i++){
     for (int j=0; j<=n; j++){
       if (i == 0 || j == 0)
         L[i][j] = 0;
       else if (X[i-1] == Y[j-1])
         L[i][j] = L[i-1][j-1] + 1;
       else
         L[i][j]=max(L[i-1][j],L[i][j-1]);
     }
   }
   // Following code is used to print LCS
   int index = L[m][n];// L[m][n] contem o numero de caracteres

```

```
//Create char array to save the lcs string
   char lcs[index+1];
   lcs[index] = '\0'; // Set the terminating character
   // Start from the right-most-bottom-most corner and
   // one by one store characters in lcs[]
   int i = m, j = n;
   while (i > 0 && j > 0) {
      // If current character in X[] and Y are same, then
      // current character is part of LCS
      if (X[i-1] == Y[j-1]){
          lcs[index-1] = X[i-1]; // Put current character in result
          i--; j--; index--;     // reduce values of i, j and index
      }
// If not same then find the larger of two and go in the direction of larger
    value
      else if (L[i-1][j] > L[i][j-1])
         i--;
      else
         j--;
   }
   // Print the lcs
 cout<<"LCSof"<<X<<"and"<<Y<<" is " <<lcs;
}
int main()
{
  char X[] = "AGGTAB";
  char Y[] = "GXTXAYB";

  int m = strlen(X);
  int n = strlen(Y);
  lcs(X, Y, m, n);
  return 0;

}
```

3.5. LONGEST COMMON SUBSTRING - $O(n^2)$

```
int LCSubStr(char *X, char *Y, int m, int n) {
  // Create a table to store lengths of longest common suffixes of
  // substrings.   Notethat LCSuff[i][j] contains length of longest
  // common suffix of X[0..i-1] and Y[0..j-1]. The first row and
  // first column entries have no logical meaning, they are used only
  // for simplicity of program
  int LCSuff[m+1][n+1];
  int result = 0;  // To store length of the longest common substring

  /* Following steps build LCSuff[m+1][n+1] in bottom up fashion. */
  for (int i=0; i<=m; i++) {
    for (int j=0; j<=n; j++) {
      if (i == 0 || j == 0)
        LCSuff[i][j] = 0;

      else if (X[i-1] == Y[j-1]) {
        LCSuff[i][j] = LCSuff[i-1][j-1] + 1;
        result = max(result, LCSuff[i][j]);
      }
      else LCSuff[i][j] = 0;
    }
  }
  return result;
}
```

3.6. LONGEST INCREASING SUBSEQUENCE - $O(nlog(n))$

```
int lis(vector<int> &arr){
  int n = arr.size();
  vector<int> lis;
  for(int i = 0; i < n; i++){
    int l = 0, r = (int)lis.size() - 1;
    int ansj = -1;
    while(l <= r){
      int mid = (l+r)/2;
      // OBS: PARA >= TROCAR SINAL EMBAIXO POR <=
      if(arr[i] < lis[mid]){
        r = mid - 1;
        ansj = mid;
      }
      else l = mid + 1;
    }
    if(ansj == -1){
      // se arr[i] e maior que todos
      lis.push_back(arr[i]);
    }
    else {
      lis[ansj] = arr[i];
    }
  }

  return lis.size();
}
```

3.7. LONGEST INCREASING SUBSEQUENCE 2D (NOT SORTED) - $O(nlog(n))$

```
set<ii> s[(int)2e6];
bool check(ii par, int ind) {

  auto it = s[ind].lower_bound(ii(par.ff, -INF));
  if(it == s[ind].begin())
    return false;

  it--;

  if(it->ss < par.ss)
    return true;
  return false;
}

int lis2d(vector<ii> &arr) {

  int n = arr.size();
  s[1].insert(arr[0]);

  int maior = 1;
  for(int i = 1; i < n; i++) {

    ii x = arr[i];

    int l = 1, r = maior;
    int ansbb = 0;
    while(l <= r) {
      int mid = (l+r)/2;
      if(check(x, mid)) {
        l = mid + 1;
        ansbb = mid;
      } else {
        r = mid - 1;
```

```
      }
    }

    // inserting in list
    auto it = s[ansbb+1].lower_bound(ii(x.ff, -INF));
    while(it != s[ansbb+1].end() && it->ss >= x.ss)
      it = s[ansbb+1].erase(it);

    it = s[ansbb+1].lower_bound(ii(x.ff, -INF));
    if(s[ansbb+1].size() > 0 && it != s[ansbb+1].end() && it->ff == x.ff &&
    it->ss <= x.ss)
      continue;
    s[ansbb+1].insert(arr[i]);

    maior = max(maior, ansbb + 1);
  }

  return maior;

}
```

3.8. LONGEST INCREASING SUBSEQUENCE 2D (SORTED) - $O(nlog(n))$

```
set<ii> s[(int)2e6];
bool check(ii par, int ind) {

  auto it = s[ind].lower_bound(ii(par.ff, -INF));
  if(it == s[ind].begin())
    return false;

  it--;

  if(it->ss < par.ss)
    return true;
  return false;
}

int lis2d(vector<ii> &arr) {

  int n = arr.size();
  s[1].insert(arr[0]);

  int maior = 1;
  for(int i = 1; i < n; i++) {

    ii x = arr[i];

    int l = 1, r = maior;
    int ansbb = 0;
    while(l <= r) {
      int mid = (l+r)/2;
      if(check(x, mid)) {
        l = mid + 1;
        ansbb = mid;
      } else {
        r = mid - 1;
      }
    }

    // inserting in list
    auto it = s[ansbb+1].lower_bound(ii(x.ff, -INF));
    while(it != s[ansbb+1].end() && it->ss >= x.ss)
      it = s[ansbb+1].erase(it);

    it = s[ansbb+1].lower_bound(ii(x.ff, -INF));
    if(s[ansbb+1].size() > 0 && it != s[ansbb+1].end() && it->ff == x.ff &&
    it->ss <= x.ss)
      continue;
    s[ansbb+1].insert(arr[i]);

    maior = max(maior, ansbb + 1);
  }

  return maior;

}
```

3.9. ACHAR MAIOR PALÍNDROMO

```
Fazer LCS da string com o reverso
```

3.10. SUBSET SUM COM BITSET - $O((maxSum)*n/32)$

```
bitset<312345> bit;
int arr[112345];
void subsetSum(int n) {
  bit.reset();
  bit.set(0);
  for(int i = 0; i < n; i++) {
    bit |= (bit << arr[i]);
  }
}
```

3.11. DIGIT DP

```
/// How many numbers x are there in the range a to b, where the digit d
    occurs exactly k times in x?
vector<int> num;
int a, b, d, k;
int DP[12][12][2];
/// DP[p][c][f] = Number of valid numbers <= b from this state
/// p = current position from left side (zero based)
/// c = number of times we have placed the digit d so far
/// f = the number we are building has already become smaller than b? [0 =
    no, 1 = yes]

int call(int pos, int cnt, int f){
  if(cnt > k) return 0;

  if(pos == num.size()){
    if(cnt == k) return 1;
    return 0;
  }

  if(DP[pos][cnt][f] != -1) return DP[pos][cnt][f];
  int res = 0;
  int lim = (f ? 9 : num[pos]);

  /// Try to place all the valid digits such that the number doesn't exceed b
  for(int dgt = 0; dgt<=LMT; dgt++){
    int nf = f;
    int ncnt = cnt;
    if(f == 0 && dgt < LMT) nf = 1; /// The number is getting smaller at
    this position
    if(dgt == d) ncnt++;
    if(ncnt <= k) res += call(pos+1, ncnt, nf);
```

```
  }

  return DP[pos][cnt][f] = res;
}

int solve(int b){
  num.clear();
  while(b>0){
    num.push_back(b%10);
    b/=10;
  }
  reverse(num.begin(), num.end());
  /// Stored all the digits of b in num for simplicity

  memset(DP, -1, sizeof(DP));
  int res = call(0, 0, 0);
  return res;
}

int main () {

  cin >> a >> b >> d >> k;
  int res = solve(b) - solve(a-1);
  cout << res << endl;

  return 0;
}
```

4. Math

4.1. INCLUSÃO-EXCLUSÃO

$$\left|\bigcup_{i=1}^{n} A_i\right| = \sum_{k=1}^{n} (-1)^{k+1} \left(\sum_{1 \le i_1 < \cdots < i_k \le n} |A_{i_1} \cap \cdots \cap A_{i_k}| \right)$$

```
int f(vector<int> arr, int LIMIT) {

  int n = arr.size();
  int c = 0;

  for(int mask = 1; mask < (1<<n); mask++) {
    int lcm = 1;
    for(int i = 0; i < n; i++)
      if(mask&(1<<i))
        lcm=lcm*arr[i]/__gcd(lcm,arr[i]);
    // se o numero de conjutos a unir for impar entao soma
    if(__builtin_popcount(mask) %2 == 1)
      c += LIMIT/lcm;
    else // se nao subtrai
      c -= LIMIT/lcm;
  }

  return LIMIT-c;

}
```

4.2. TODOS OS DIVISORES DE N - $O(sqrt(n))$

```
// OBS: ATÉ cuberoot(N) DIVISORES DE N
vector<int> divisors(int n) {
  int sq = sqrt(n);
  vector<int> ans;
  for (int i=1; i<=sq+1; i++) {
    if (n%i==0) {
      // If divisors are equal, print only one
      if (n/i == i)
        ans.pb(i);

      else // Otherwise print both
        ans.pb(i), ans.pb(n/i);
    }
  }
  return ans;
}
```

4.3. ALGORITMO DE EUCLIDES ESTENDIDO - $O(log(min(a,b)))$

```
int gcd,x,y;

//Ax + By = gcd(A,B)

void extendedEuclidian(int a,int b){

  if(b==0){
    gcd=a;
```

```
    x=1;
    y=0;
  } else{
    extendedEuclidian(b, a%b);

    int temp = x;
    x=y;
    y = temp - (a/b)*y;
  }
}
```

4.4. TEOREMA CHINES DO RESTO

```
/*
    Ex: x Congruo 2 (mod 3)
        x Congruo 3 (mod 5)
        x Congruo 2 (mod 7)
 */

int inv(int a, int m) {
  int m0 = m, t, q;
  int x0 = 0, x1 = 1;

  if (m == 1)
    return 0;


// Apply extended Euclid Algorithm
  while (a > 1) {
// q is quotient
    q = a / m;
    t = m;
    // m is remainder now, process same as euclid's algo
    m = a % m, a = t;
    t = x0;
    x0 = x1 - q * x0;
    x1 = t;
  }

  // Make x1 positive
  if (x1 < 0)
    x1 += m0;

  return x1;
}
// k is size of num[] and rem[].  Returns the smallest
// number x such that:
//  x % num[0] = rem[0],
//  x % num[1] = rem[1],
//  ..................
//  x % num[k-2] = rem[k-1]
// Assumption: Numbers in num[] are pairwise coprime
// (gcd for every pair is 1)
int findMinX(int num[], int rem[], int k){
  // Compute product of all numbers
  int prod = 1;
  for (int i = 0; i < k; i++)
    prod *= num[i];

  // Initialize result
  int result = 0;

  // Apply above formula
  for (int i = 0; i < k; i++){
    int pp = prod / num[i];
    result+=rem[i]*inv(pp,num[i])*pp;
  }

  return result % prod;
}
```

4.5. BELL NUMBERS (NUMBER OF WAYS TO PARTITION A SET) - $O(n^2)$

```
int bellNumber(int n) {
  int bell[n+1][n+1];
  bell[0][0] = 1;
  for (int i=1; i<=n; i++) {
    // Explicitly fill for j = 0
    bell[i][0] = bell[i-1][i-1];

    // Fill for remaining values of j
    for (int j=1; j<=i; j++)
      bell[i][j] = bell[i-1][j-1] + bell[i][j-1];
  }
  return bell[n][0];
}
```

4.6. FATORAÇÃO SIMPLES - $O(sqrt(n))$

```
set<int> primeFactors(int n) {
  set<int> ret;
  while (n%2 == 0) {
    ret.insert(2);
    n = n/2;
  }

  int sq = sqrt(n);
  for (int i = 3; i <= sq+2; i = i+2) {
    while (n%i == 0) {
      ret.insert(i);
      n = n/i;
    }
    /* OBS1
  IF(N < 1E7)
    FATORE COM SPF
    */
  }

  if (n > 2)
    ret.insert(n);

  return ret;
}
```

4.7. FATORAÇÃO PRA MULTIPLAS QUERIES - $O(n(log(log(n))))$

```
//stor smallest prime factor for every num
int spf[MAXN];
// Calculating SPF (Smallest Prime Factor) for every number till MAXN.
// Time Complexity : O(nloglogn)
void sieve() {
  spf[1] = 1;
  for (int i=2; i<MAXN; i++)
    // marking smallest prime factor for every number to be itself.
    spf[i] = i;

```

```
  // separatelyMarking spf for every even
  // number as 2
  for (int i=4; i<MAXN; i+=2)
    spf[i] = 2;

  for (int i=3; i*i<MAXN; i++) {
    // checking if i is prime
    if (spf[i] == i){
      // marking SPF for all numbers divisible by i
      for (int j=i*i; j<MAXN; j+=i)
        // marking spf[j] if it is not previously marked
        if (spf[j]==j)
          spf[j] = i;
    }
  }
}
// A O(log n) function returning primefactorization
// by dividing by smallest prime factor at every step
vector<int> getFactorization(int x) {
  vector<int> ret;
  while (x != 1) {
    ret.push_back(spf[x]);
    x = x / spf[x];
  }
  return ret;
}
```

4.8. CRIVO COMUM E SEGMENTADO COM BITSET

```
bitset<(int)1e5+3> primesInSeg;
bitset<(int)1e6+3> isPrime;
int spf[(int)1e6+3];
vector<int> primes;
vector<int> segPrimes;

void sieve(int n = (int)1e6) {

  isPrime.set();
  for(int i = 2; i*i <= n; i++) {
    if(!isPrime[i])
      continue;

    for(int j = i*i; j <= n; j+=i) {
      isPrime[j] = false;
      spf[j] = min(i, spf[j]);
    }
    primes.pb(i);
  }
}

vector<int> getFactorization(int x) {
  vector<int> ret;
  while (x != 1) {
    ret.push_back(spf[x]);
    x = x / spf[x];
  }
  return ret;
}

void segSieve(int l, int r) {
  // primos de l..r
  // transladados de 0..(l-r)
  segPrimes.clear();
  primesInSeg.set();
  int sq = sqrt(r);

  for(int p: primes) {
    if(p > sq)
      break;

    for(int i = l - l%p; i <= r; i += p) {
      if(i - l < 0)
        continue;

      // se i for menor q 1e6 checa na array do crivo
      if(i >= (int)1e6 || !isPrime[i])
        primesInSeg[i-l] = false;
    }
  }

  for(int i = 0; i < r-l+1; i++) {
    if(primesInSeg[i])
      segPrimes.pb(i+l);
  }
}
```

4.9. CHECAR SE UM NUMERO É PRIMO - $O(sqrt(n))$

```
bool isPrime(int n) {
  if (n <= 1)  return false;
  if (n <= 3)  return true;
  // This is checked so that we can skip
  // middle five numbers in below loop
  if (n%2 == 0 || n%3 == 0)
    return false;
  for (int i=5; i*i<=n; i += 6)
    if (n%i == 0 || n%(i+2) == 0)
      return false;
  return true;
}
```

4.10. EXPONENCIAÇÃO BINÁRIA - $O(log(n))$

```
int power(int x, int p, int MOD) {
  if(p == 0)
    return 1%MOD;
  if(p == 1)
    return x%MOD;
  int res = power(x, p/2, MOD);
  res = (long long)res*res%MOD;
  if(p&1)
    res = (long long)res*x%MOD;
  return res;
}
```

4.11. PRECOMPUTAR COMBINAÇÃO nCr - $O(n^2)$

```
int C[1123][1123];

int mod(int n) {return n%((int)1e9+7);}

int nCr(int n, int k) {
  for(int i = 0; i <= n; i++) {
    for(int j = 0; j <= min(i,k); j++) {
      if(j == 0 || j == i) {
        C[i][j] = 1;
```

```
      } else {
        C[i][j]=mod(C[i-1][j-1]+C[i-1][j]);
      }
    }
  }
}
```

4.12. COMBINAÇÃO nCr - $O(n)$

```
// Returns value of Binomial Coefficient C(n, k)
int binomialCoeff(int n, int k) {
  int res = 1;
  // Since C(n, k) = C(n, n-k)
  if (k > n - k)
    k = n - k;
  // Calculate value of [n*(n-1)*---*(n-k+1)] / [k*(k-1)*---*1]
  for (int i = 0; i < k; ++i) {
    res *= (n - i);
    res /= (i + 1);
  }
  return res;
}
```

4.13. POLLARD RHO (FIND A DIVISOR FOR N) - $O(n^{(1/4)})$

```
/* Function to calculate (base^exponent)%modulus */
int modular_pow(int base, int exponent,
            int modulus) {
  /* initialize result */
  int result = 1;
  while (exponent > 0) {
    /* if y is odd, multiply base with result */
    if (exponent & 1)
      result = (result * base) % modulus;
    /* exponent = exponent/2 */
    exponent = exponent >> 1;
    /* base = base * base */
    base = (base * base) % modulus;
  }
  return result;
}

/* method to return prime divisor for n */
int PollardRho(int n) {
  /* initialize random seed */
  srand (time(NULL));

  /* no prime divisor for 1 */
  if (n==1) return n;

  /* even number means one of the divisors is 2 */
  if (n % 2 == 0) return 2;

  /* we will pick from the range [2, N) */
  int x = (rand()%(n-2))+2;
  int y = x;

  /* the constant in f(x).
  * Algorithm can be re-run with a different c
  * if it throws failure for a composite. */
  int c = (rand()%(n-1))+1;

  /* Initialize candidate divisor (or result) */
  int d = 1;

  /* until the prime factor isn't obtained.
  If n is prime, return n */
  while (d==1) {
    /* Tortoise Move: x(i+1) = f(x(i)) */
    x = (modular_pow(x, 2, n) + c + n)%n;

    /* Hare Move: y(i+1) = f(f(y(i))) */
    y = (modular_pow(y, 2, n) + c + n)%n;
    y = (modular_pow(y, 2, n) + c + n)%n;

    /* check gcd of |x-y| and n */
    d = __gcd(abs(x-y), n);

    /* retry if the algorithm fails to find prime factor
    * with chosen x and c */
    if (d==n) return PollardRho(n);
  }

  return d;
}

/* driver function */
signed main() {
  int n = 12;
  printf("One of the divisors for %lld is %lld.",
    n, PollardRho(n));
  return 0;
}
```

4.14. EQUAÇÃO DIOFANTINA (ACHAR UMA SOLUÇÃO)

```
int n;
vector<vector<int>> adj;
vector<char> color;
vector<int> parent;
int cycle_start, cycle_end;

bool dfs(int v) {
  color[v] = 1;
  for (int u : adj[v]) {
    if (color[u] == 0) {
      parent[u] = v;
      if (dfs(u))
        return true;
    } else if (color[u] == 1) {
      cycle_end = v;
      cycle_start = u;
      return true;
    }
  }
  color[v] = 2;
  return false;
}

void find_cycle() {
  color.assign(n, 0);
  parent.assign(n, -1);
  cycle_start = -1;

  for (int v = 0; v < n; v++) {
    if (dfs(v))
      break;
```

```
  }

  if (cycle_start == -1) {
    cout << "Acyclic" << endl;
  } else {
    vector<int> cycle;
    cycle.push_back(cycle_start);
    for (int v = cycle_end; v != cycle_start; v = parent[v])
      cycle.push_back(v);
    cycle.push_back(cycle_start);
    reverse(cycle.begin(), cycle.end());

    cout << "Cycle found: ";
    for (int v : cycle)
      cout << v << " ";
    cout << endl;
  }
}
```

4.15. EULER'S TOTIENT FUNCTION

```
int phi(int n) {
  int result = n;
  for (int i = 2; i * i <= n; i++) {
    if(n % i == 0) {
      while(n % i == 0)
          n /= i;
        result -= result / i;
    }
  }

  if(n > 1)
    result -= result / n;
  return result;
}
```

5. Geometry

5.1. PRODUTO VETORIAL

```
// Outra forma de produto vetorial
// reta ab,ac se for zero e colinear
// se for < 0 entao antiHorario, > 0 horario
bool ehcol(pto a,pto b,pto c) {
  return ((b.y-a.y)*(c.x-a.x) - (b.x-a.x)*(c.y-a.y));
}
---------------------------------------
//Produto vetorial AB x AC, se for zero e colinear
int cross(pto A, pto B, pto C){
  pto AB, AC;
  AB.x = B.x-A.x;
  AB.y = B.y-A.y;
  AC.x = C.x-A.x;
  AC.y = C.y-A.y;
  int cross = AB.x*AC.y-AB.y * AC.x;
  return cross;
}

// OBS: DEFINE ÁREA DE QUADRILÁTERO FORMADO PELAS RETAS, A ÁREA DO TRIÂNGULO
    É A METADE
```

5.2. DISTANCIA PONTO RETA

```
double ptoReta(double x1, double y1, double x2,double y2,double pointX,
    double pointY, double *ptox,double *ptoy){
  double diffX = x2 - x1;
  float diffY = y2 - y1;
  if ((diffX == 0) && (diffY == 0)) {
    diffX = pointX - x1;
    diffY = pointY - y1;
    //se os dois sao pontos
    return hypot(pointX - x1,pointY - y1);
  }
  float t = ((pointX - x1) * diffX + (pointY - y1) * diffY) /
                  (diffX * diffX + diffY * diffY);
  if (t < 0) {
    //point is nearest to the first point i.e x1 and y1
    // Ex:     _____________     .
    // cord do pto na reta = pto inicial(x1,y1);
    *ptox = x1, *ptoy = y1;
    diffX = pointX - x1;
    diffY = pointY - y1;
  } else if (t > 1) {
    //point is nearest to the end point i.e x2 and y2
    // Ex : .    _______________
    // cord do pto na reta = pto final(x2,y2);
    *ptox = x2, *ptoy = y2;
    diffX = pointX - x2;
    diffY = pointY - y2;
  } else  {
      //if perpendicular line intersect the line segment.
      // pto nao esta mais proximo de uma das bordas do segmento
      // Ex:              .
      //                  |
      //                  |(Ângulo Reto)
      //__________________________________________
      // cord x do pto na reta = (x1 + t * diffX)
      // cord y do pto na reta = (y1 + t * diffY)
    *ptox = (x1 + t * diffX), *ptoy = (y1 + t * diffY);
    diffX = pointX - (x1 + t * diffX);
```

```
    diffY = pointY - (y1 + t * diffY);
  }
  //returning shortest distance
  return sqrt(diffX * diffX + diffY * diffY);
}
```

5.3. ÁREA DE POLÍGONOS

```

double polygonArea(vector<pto> &arr, int n) {
  int area = 0;
  // N = quantidade de pontos no polígono e armazenados em p;
  // OBS: VALE PARA CONVEXO E NÃO CONVEXO
  for(int i = 0; i<n; i++){
    area += cross(arr[i], arr[(i+1)%n]);
  }
  return (double)abs(area/2.0);
}
```

5.4. PONTOS DENTRO DE UM POLIGONO

```

/* Traça-se uma reta do ponto até um outro ponto qualquer fora do triangulo
    e checa o número de interseção com a borda do polígono se este for ímpar
    então está dentro se não está fora */

// Define Infinite (Using INT_MAX caused overflow problems)
#define INF 10000

struct pto {
    int x, y;
    pto() {}
    pto(int x, int y) : x(x), y(y) {}
};

// Given three colinear ptos p, q, r, the function checks if
// pto q lies on line segment 'pr'
bool onSegment(pto p, pto q, pto r) {
  if (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) &&
      q.y <= max(p.y, r.y) && q.y >= min(p.y, r.y))
    return true;
  return false;
}

// To find orientation of ordered triplet (p, q, r).
// The function returns following values
// 0 --> p, q and r are colinear
// 1 --> Clockwise
// 2 --> Counterclockwise
int orientation(pto p, pto q, pto r) {
  int val = (q.y - p.y) * (r.x - q.x) -
            (q.x - p.x) * (r.y - q.y);

  if (val == 0) return 0;  // colinear
  return (val > 0)? 1: 2; // clock or counterclock wise
}

// The function that returns true if line segment 'p1q1'
// and 'p2q2' intersect.
bool doIntersect(pto p1, pto q1, pto p2, pto q2) {
  // Find the four orientations needed for general and
  // special cases
  int o1 = orientation(p1, q1, p2);
  int o2 = orientation(p1, q1, q2);
  int o3 = orientation(p2, q2, p1);
  int o4 = orientation(p2, q2, q1);

  // General case
  if (o1 != o2 && o3 != o4)
    return true;

  // Special Cases
  // p1, q1 and p2 are colinear and p2 lies on segment p1q1
  if (o1 == 0 && onSegment(p1, p2, q1)) return true;

  // p1, q1 and p2 are colinear and q2 lies on segment p1q1
  if (o2 == 0 && onSegment(p1, q2, q1)) return true;

  // p2, q2 and p1 are colinear and p1 lies on segment p2q2
  if (o3 == 0 && onSegment(p2, p1, q2)) return true;

   // p2, q2 and q1 are colinear and q1 lies on segment p2q2
  if (o4 == 0 && onSegment(p2, q1, q2)) return true;

  return false; // Doesn't fall in any of the above cases
}

// Returns true if the pto p lies inside the polygon[] with n vertices
bool isInside(pto polygon[], int n, pto p) {
  // There must be at least 3 vertices in polygon[]
  if (n < 3)  return false;

  // Create a pto for line segment from p to infinite
  pto extreme = pto(INF, p.y);

  // Count intersections of the above line with sides of polygon
  int count = 0, i = 0;
  do {
    int next = (i+1)%n;

    // Check if the line segment from 'p' to 'extreme' intersects
    // with the line segment from 'polygon[i]' to 'polygon[next]'
    if (doIntersect(polygon[i], polygon[next], p, extreme)) {
      // If the pto 'p' is colinear with line segment 'i-next',
      // then check if it lies on segment. If it lies, return true,
      // otherwise false
      if (orientation(polygon[i], p, polygon[next]) == 0)
        return onSegment(polygon[i], p, polygon[next]);

      count++;
    }
    i = next;
  } while (i != 0);

  // Return true if count is odd, false otherwise
  return count&1;  // Same as (count%2 == 1)
}
```

5.5. PONTOS DENTRO E NA BORDA DE UM POLIGONO

```
int cross(pto a, pto b) {
  return a.x * b.y - b.x * a.y;
}

int boundaryCount(pto a, pto b) {
  if(a.x == b.x)
    return abs(a.y-b.y)-1;
```

```
  if(a.y == b.y)
    return abs(a.x-b.x)-1;
  return _gcd(abs(a.x-b.x), abs(a.y-b.y))-1;
}

int totalBoundaryPolygon(vector<pto> &arr, int n) {

  int boundPoint = n;
  for(int i = 0; i < n; i++) {
    boundPoint += boundaryCount(arr[i], arr[(i+1)%n]);
  }
  return boundPoint;
}

int polygonArea2(vector<pto> &arr, int n) {
  int area = 0;
  // N = quantidade de pontos no polígono e armazenados em p;
  // OBS: VALE PARA CONVEXO E NÃO CONVEXO
  for(int i = 0; i<n; i++){
    area += cross(arr[i], arr[(i+1)%n]);
  }
  return abs(area);
}

int internalCount(vector<pto> &arr, int n) {

  int area_2 = polygonArea2(arr, n);
  int boundPoints = totalBoundaryPolygon(arr,n);
  return (area_2 - boundPoints + 2)/2;
}
```

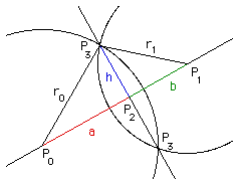
5.6. INTERSECÇÃO DE RETAS

```
// Intersecção de retas Ax + By = C    dados pontos (x1,y1) e (x2,y2)
A = y2-y1
B = x1-x2
C = A*x1+B*y1
//Retas definidas pelas equações:
A1x + B1y = C1
A2x + B2y = C2
//Encontrar x e y resolvendo o sistema
double det = A1*B2 - A2*B1;
if(det == 0){
  //Lines are parallel
}else{
  double x = (B2*C1 - B1*C2)/det;
  double y = (A1*C2 - A2*C1)/det;
}
```

5.7. INTERSECÇÃO DE SEGMENTOS

```
// Given three colinear points p, q, r, the function checks if
// point q lies on line segment 'pr'
int onSegment(Point p, Point q, Point r) {
  if (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) && q.y <= max(p.y, r.y)
    && q.y >= min(p.y, r.y))
    return true;
  return false;
}
/* PODE SER RETIRADO
int onSegmentNotBorda(Point p, Point q, Point r) {
    if (q.x < max(p.x, r.x) && q.x > min(p.x, r.x) && q.y <= max(p.y, r.y)
    && q.y >= min(p.y, r.y))
        return true;
    if (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) && q.y < max(p.y, r.y)
    && q.y > min(p.y, r.y))
        return true;
    return false;
}
*/
// To find orientation of ordered triplet (p, q, r).
// The function returns following values
// 0 --> p, q and r are colinear
// 1 --> Clockwise
// 2 --> Counterclockwise
int orientation(Point p, Point q, Point r) {
  int val = (q.y - p.y) * (r.x - q.x) -
            (q.x - p.x) * (r.y - q.y);
  if (val == 0) return 0;  // colinear
  return (val > 0)? 1: 2; // clock or counterclock wise
}
// The main function that returns true if line segment 'p1p2'
// and 'q1q2' intersect.
int doIntersect(Point p1, Point p2, Point q1, Point q2) {
  // Find the four orientations needed for general and
  // special cases
  int o1 = orientation(p1, p2, q1);
  int o2 = orientation(p1, p2, q2);
  int o3 = orientation(q1, q2, p1);
  int o4 = orientation(q1, q2, p2);

  // General case
  if (o1 != o2 && o3 != o4) return 2;

/* PODE SER RETIRADO
  if(o1 == o2 && o2 == o3 && o3 == o4 && o4 == 0) {
    //INTERCEPTAM EM RETA
    if(onSegmentNotBorda(p1,q1,p2) || onSegmentNotBorda(p1,q2,p2)) return 1;
    if(onSegmentNotBorda(q1,p1,q2) || onSegmentNotBorda(q1,p2,q2)) return 1;
  }
*/
  // Special Cases (INTERCEPTAM EM PONTO)
  // p1, p2 and q1 are colinear and q1 lies on segment p1p2
  if (o1 == 0 && onSegment(p1, q1, p2)) return 2;
  // p1, p2 and q1 are colinear and q2 lies on segment p1p2
  if (o2 == 0 && onSegment(p1, q2, p2)) return 2;
  // q1, q2 and p1 are colinear and p1 lies on segment q1q2
  if (o3 == 0 && onSegment(q1, p1, q2)) return 2;
   // q1, q2 and p2 are colinear and p2 lies on segment q1q2
  if (o4 == 0 && onSegment(q1, p2, q2)) return 2;
  return false; // Doesn't fall in any of the above cases
}
// OBS: SE (C2/A2 == C1/A1) SÃO COLINEARES
```

5.8. INTERSECÇÃO DE CIRCULOS (2PTOS)



```
void circle_circle_intersection(/*...*/){
  double a, d, h, rx, ry,x2,y2;
  d = hypot(x1 - x0,y1 - y0);
  /* Determine the distance from point 0 to point 2. */
  a = ((r0*r0) - (r1*r1)+(d*d))/(2.0* d) ;
  /*Determine the coordinates of point 2*/
  x2 = x0 + (dx * a/d); y2 = y0 + (dy * a/d);
  /* Determine the distance from point 2 to either of the intersection
     points.*/
  h = sqrt((r0*r0) - (a*a));
  /* Now determine the offsets of the intersection points from
   * point 2.*/
  rx = -dy * (h/d); ry = dx * (h/d);
  /* Determine the absolute intersection points. */
  *xi = x2 + rx;
  *xi_prime = x2 - rx;
  *yi = y2 + ry;
  *yi_prime = y2 - ry;
}
```

5.9. PONTO DENTRO DE UM POLIGONO

```

/* Traça-se uma reta do ponto até um outro ponto qualquer fora do triangulo
    e checa o número de interseção com a borda do polígono se este for ímpar
    então está dentro se não está fora */

// Define Infinite (Using INT_MAX caused overflow problems)
#define INF 10000

struct pto {
    int x, y;
    pto() {}
    pto(int x, int y) : x(x), y(y) {}
};

// Given three colinear ptos p, q, r, the function checks if
// pto q lies on line segment 'pr'
bool onSegment(pto p, pto q, pto r) {
  if (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) &&
      q.y <= max(p.y, r.y) && q.y >= min(p.y, r.y))
    return true;
  return false;
}

// To find orientation of ordered triplet (p, q, r).
// The function returns following values
// 0 --> p, q and r are colinear
// 1 --> Clockwise
// 2 --> Counterclockwise
int orientation(pto p, pto q, pto r) {
  int val = (q.y - p.y) * (r.x - q.x) -
            (q.x - p.x) * (r.y - q.y);

  if (val == 0) return 0;  // colinear
  return (val > 0)? 1: 2; // clock or counterclock wise
}

// The function that returns true if line segment 'p1q1'
// and 'p2q2' intersect.
bool doIntersect(pto p1, pto q1, pto p2, pto q2) {
  // Find the four orientations needed for general and
  // special cases
  int o1 = orientation(p1, q1, p2);
  int o2 = orientation(p1, q1, q2);
  int o3 = orientation(p2, q2, p1);
  int o4 = orientation(p2, q2, q1);

  // General case
  if (o1 != o2 && o3 != o4)
    return true;

  // Special Cases
  // p1, q1 and p2 are colinear and p2 lies on segment p1q1
  if (o1 == 0 && onSegment(p1, p2, q1)) return true;

  // p1, q1 and p2 are colinear and q2 lies on segment p1q1
  if (o2 == 0 && onSegment(p1, q2, q1)) return true;

  // p2, q2 and p1 are colinear and p1 lies on segment p2q2
  if (o3 == 0 && onSegment(p2, p1, q2)) return true;

   // p2, q2 and q1 are colinear and q1 lies on segment p2q2
  if (o4 == 0 && onSegment(p2, q1, q2)) return true;

  return false; // Doesn't fall in any of the above cases
}

// Returns true if the pto p lies inside the polygon[] with n vertices
bool isInside(pto polygon[], int n, pto p) {
  // There must be at least 3 vertices in polygon[]
  if (n < 3)  return false;

  // Create a pto for line segment from p to infinite
  pto extreme = pto(INF, p.y);

  // Count intersections of the above line with sides of polygon
  int count = 0, i = 0;
  do {
    int next = (i+1)%n;

    // Check if the line segment from 'p' to 'extreme' intersects
    // with the line segment from 'polygon[i]' to 'polygon[next]'
    if (doIntersect(polygon[i], polygon[next], p, extreme)) {
      // If the pto 'p' is colinear with line segment 'i-next',
      // then check if it lies on segment. If it lies, return true,
      // otherwise false
      if (orientation(polygon[i], p, polygon[next]) == 0)
        return onSegment(polygon[i], p, polygon[next]);

      count++;
    }
    i = next;
  } while (i != 0);

  // Return true if count is odd, false otherwise
  return count&1;  // Same as (count%2 == 1)
}
```

5.10. CLOSEST PAIR OF POINTS

```
struct Point {
  int x, y;
};
int compareX(const void *a,const void *b){
  Point *p1 = (Point *)a,  *p2 = (Point *)b;
  return (p1->x - p2->x);
```

```
}
int compareY(const void *a,const void *b) {
  Point *p1 = (Point *)a,*p2 =(Point *)b;
  return (p1->y - p2->y);
}
float dist(Point p1, Point p2) {
  return sqrt((p1.x- p2.x)*(p1.x- p2.x) +(p1.y - p2.y)*(p1.y - p2.y));
}
float bruteForce(Point P[], int n){
  float min = FLT_MAX;
  for (int i = 0; i < n; ++i)
    for (int j = i+1; j < n; ++j)
      if (dist(P[i], P[j]) < min)
        min = dist(P[i], P[j]);
  return min;
}
float min(float x, float y) {
  return (x < y)? x : y;
}
float stripClosest(Point strip[], int size, float d) {
  float min = d;
  for (int i = 0; i < size; ++i)
    for (int j = i+1; j < size && (strip[j].y - strip[i].y) < min; ++j)
      if (dist(strip[i],strip[j]) < min)
        min = dist(strip[i], strip[j]);
  return min;
}
float closestUtil(Point Px[], Point Py[], int n){
  if (n <= 3)
    return bruteForce(Px, n);
  int mid = n/2;
  Point midPoint = Px[mid];
  Point Pyl[mid+1];
  Point Pyr[n-mid-1];
  int li = 0, ri = 0;
  for (int i = 0; i < n; i++)
    if (Py[i].x <= midPoint.x)
      Pyl[li++] = Py[i];
    else
      Pyr[ri++] = Py[i];

  float dl = closestUtil(Px, Pyl, mid);
  float dr = closestUtil(Px + mid, Pyr, n-mid);
  float d = min(dl, dr);
  Point strip[n];
  int j = 0;
  for (int i = 0; i < n; i++)
    if (abs(Py[i].x - midPoint.x) < d)
      strip[j] = Py[i], j++;
  return min(d, stripClosest(strip, j, d));
}

float closest(Point P[], int n) {
  Point Px[n];
  Point Py[n];
  for (int i = 0; i < n; i++) {
    Px[i] = P[i];
    Py[i] = P[i];
  }
  qsort(Px, n, sizeof(Point), compareX);
  qsort(Py, n, sizeof(Point), compareY);
  return closestUtil(Px, Py, n);
}
```

5.11. CENTRO DE MASSA DE UM POLÍGONO

```
double area = 0;
pto c;

c.x = c.y = 0;
for(int i = 0; i < n ; i++) {
  double aux = (arr[i].x * arr[i+1].y) - (arr[i].y * arr[i+1].x); // shoelace
  area += aux;
  c.x += aux*(arr[i].x + arr[i+1].x);
  c.y += aux*(arr[i].y + arr[i+1].y);
}

c.x /= (3.0*area);
c.y /= (3.0*area);

cout  << c.x <<  ' ' << c.y << endl;
```

5.12. POINT AND LINE STRUCT

```
int sgn(double x) {
    if(abs(x) < 1e-8)   return 0;
    return x > 0 ? 1 : -1;
}
inline double sqr(double x) {   return x * x;   }

struct Point {
    double x, y, z;
    Point() {};
    Point(double a, double b): x(a), y(b) {};
    Point (double x, double y, double z): x(x), y(y), z(z) {}

    void input() {  scanf(" %lf %lf", &x, &y);  };
    friend Point operator+(const Point &a, const Point &b) {
        return Point(a.x + b.x, a.y + b.y);
    }
    friend Point operator-(const Point &a, const Point &b) {
        return Point(a.x - b.x, a.y - b.y);
    }

    bool operator !=(const Point& a) const {
        return (x != a.x || y != a.y);
    }

    bool operator <(const Point &a) const{
      if(x == a.x)
        return y < a.y;
      return x < a.x;
    }

    double norm() {
        return sqrt(sqr(x) + sqr(y));
    }
};
double det(const Point &a, const Point &b) {
    return a.x * b.y - a.y * b.x;
}
double dot(const Point &a, const Point &b) {
    return a.x * b.x + a.y * b.y;
}
double dist(const Point &a, const Point &b) {
    return (a-b).norm();
}

```

```

struct Line {
    Point a, b;
    Line() {}
    Line(Point x, Point y): a(x), b(y) {};
};

double dis_point_segment(const Point p, const Point s, const Point t) {
    if(sgn(dot(p-s, t-s)) < 0)
      return (p-s).norm();
    if(sgn(dot(p-t, s-t)) < 0)
      return (p-t).norm();
    return abs(det(s-p, t-p) / dist(s, t));
}
```

5.13. CONVEX HULL - $O(nlog(n))$

```
// Asymptotic complexity: O(n log n).
struct pto {
  double x, y;
  bool operator <(const pto &p) const {
    return x < p.x || (x == p.x && y < p.y);
    /* a impressao será em prioridade por mais a esquerda, mais
       abaixo, e antihorário pelo cross abaixo */
  }
};

double cross(const pto &O, const pto &A, const pto &B) {
  return (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
}

vector<pto> convex_hull(vector<pto> P) {
  int n = P.size(), k = 0;
  vector<pto> H(2 * n);
  // Sort points lexicographically
  sort(P.begin(), P.end());
  // Build lower hull
  for (int i = 0; i < n; ++i) {
    // esse <= 0 representa sentido anti-horario, caso deseje mudar
    // trocar por >= 0
    while (k >= 2 && cross(H[k - 2], H[k - 1], P[i]) <= 0)
      k--;
    H[k++] = P[i];
  }
  // Build upper hull
  for (int i = n - 2, t = k + 1; i >= 0; i--) {
    // esse <= 0 representa sentido anti-horario, caso deseje mudar
    // trocar por >= 0
    while (k >= t && cross(H[k - 2], H[k - 1], P[i]) <= 0)
      k--;
    H[k++] = P[i];
  }
  H.resize(k);
  /* o último ponto do vetor é igual ao primeiro, atente para isso
  as vezes é necessário mudar */
  return H;
}
```

5.14. UPPER AND LOWER HULL - $O(nlog(n))$

```
struct pto {
  double x, y;
  bool operator <(const pto &p) const {
    return x < p.x || (x == p.x && y < p.y);
    /* a impressao será em prioridade por mais a esquerda, mais
       abaixo, e antihorário pelo cross abaixo */
  }
};
double cross(const pto &O, const pto &A, const pto &B) {
  return (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
}

vector<pto> lower, upper;

vector<pto> convex_hull(vector<pto> &P) {
  int n = P.size(), k = 0;
  vector<pto> H(2 * n);
  // Sort points lexicographically
  sort(P.begin(), P.end());
  // Build lower hull
  for (int i = 0; i < n; ++i) {
    // esse <= 0 representa sentido anti-horario, caso deseje mudar
    // trocar por >= 0
    while (k >= 2 && cross(H[k - 2], H[k - 1], P[i]) <= 0)
      k--;
    H[k++] = P[i];
  }
  // Build upper hull
  for (int i = n - 2, t = k + 1; i >= 0; i--) {
    // esse <= 0 representa sentido anti-horario, caso deseje mudar
    // trocar por >= 0
    while (k >= t && cross(H[k - 2], H[k - 1], P[i]) <= 0)
      k--;
    H[k++] = P[i];
  }
  H.resize(k);
  /* o último ponto do vetor é igual ao primeiro, atente para isso
  as vezes é necessário mudar */

  int j = 1;
  lower.pb(H.front());
  while(H[j].x >= H[j-1].x) {
    lower.pb(H[j++]);
  }

  int l = H.size()-1;
  while(l >= j) {
    upper.pb(H[l--]);
  }
  upper.pb(H[l--]);

  return H;
}
```

5.15. CONDIÇÃO DE EXISTÊNCIA DE UM TRIÂNGULO

```

    | b - c | < a < b + c
    | a - c | < b < a + c
    | a - b | < c < a + b

Para a < b < c, basta checar
    a + b > c

OBS: Para um conjunto n >= 100 sempre exite um retângulo válido, pois a
    sequência de rettângulos não válidos seguem a sequência de Fibonacci e
    Fib(100) > 2^64
```

5.16. ÁREA POLÍGONO 3D - $O(n)$

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

using namespace std;

struct point{
  double x,y,z;
  void operator=(const point & b){
    x = b.x;
    y = b.y;
    z = b.z;
  }
};

point cross(point a, point b){
  point ret;
  ret.x = a.y*b.z - b.y*a.z;
  ret.y = a.z*b.x - a.x*b.z;
  ret.z = a.x*b.y - a.y*b.x;
  return ret;
}

int main(){
  int num;
  cin >> num;
  point v[num];
  for(int i=0; i<num; i++) cin >> v[i].x >> v[i].y >> v[i].z;

  point cur;
  cur.x = 0, cur.y = 0, cur.z = 0;

  for(int i=0; i<num; i++){
    point res = cross(v[i], v[(i+1)%num]);
    cur.x += res.x;
    cur.y += res.y;
    cur.z += res.z;
  }

  double ans = sqrt(cur.x*cur.x + cur.y*cur.y + cur.z*cur.z);

  double area = abs(ans);

  cout << fixed << setprecision(9) << area/2. << endl;
}
```

6. Strings

6.1. KMP ALGORITMO - $O(n+m)$

```cpp
// (achar substring numa string)
// Complexidade: O(n)

// TXT = "ABABABABA", PAT = "AB"
// Fills lps[] for given patttern pat[0..M-1]
void compute(string &pat, int m, int lps[]) {
  // length of the previous longest prefix suffix
  int len = 0;
  lps[0] = 0; // lps[0] is always 0

  // the loop calculates lps[i] for i = 1 to m-1
  int i = 1;
  while (i < m) {
    if (pat[i] == pat[len]) {
      len++, i++;
      lps[i] = len;
    } else {// (pat[i] != pat[len])
      if (len != 0)
        len = lps[len-1];
      else
        lps[i++] = 0;
    }
  }
}

// Driver program to test above function
// Prints occurrences of txt[] in pat[]
int KMPSearch(string pat, string txt) {
  int n = txt.size(), m = pat.size();
  // create lps[] that will hold the longest prefix suffix values for pattern
  int lps[m], c = 0;

  // Preprocess the pattern (calculate lps[] array)
  compute(pat, m, lps);

  int i = 0;  // index for txt[]
  int j = 0;  // index for pat[]
  while (i < n) {
    if (pat[j] == txt[i])
      j++, i++;

    if (j == m) {
      printf("pattern at %d \n", i-j);
      j = lps[j-1];
      c++;
      // Múltiplos matches
    }

    // mismatch after j matches
    else if (i < n && pat[j] != txt[i]){
      // Do not match lps[0..lps[j-1]] characters, they will match anyway
      if (j != 0)
        j = lps[j-1];
      else
        i = i+1;
    }
  }
  // return the number of occurrences
  return c;
}
int main() {
```

```
  string txt = "ABABDABACDABABCABAB";
  string pat = "AB";
  KMPSearch(pat, txt);
}
Output:
pattern at 0,2,5,10,12,15,17
```

6.2. TRIE

```
struct No {
  map<char,No*> trans;
  bool word = false;
};

No* head;

No* makeNo() {
  return new No();
}

void insert(string x) {

  No* aux = head;

  for(char c: x) {
    if(aux->trans[c]) {
      aux = aux->trans[c];
    } else {
      aux = aux->trans[c] = makeNo();
    }
  }
  aux->word = true;
}

bool search(string s){
  no* aux = head;
  for(char c : s){
    if(aux->trans[c])
      aux = aux->trans[c];
    else
      return false;
  }
  return aux->word;
}

int main(){
  head = makeNo();
}
```

6.3. TRIE (CONTANDO TRANSICOES)

```
// Contém também deleção de números e conversão
// NUMERO TA EM LONG LONG
struct No {
  map<char,pair<No*,int> > trans;
  bool word = false;
};

No *head;

No *makeNo() {
  return new No();
}

char comp(char x) {
  return (x == '0' ? '1':'0');
}

string toBin(int x) {
  string r;
  for(int i = 32; i >= 0; i--)
    r += ((bool)(x & (1ll<<i)) + '0');
  return r;
}

string toBinQuery(int x) {
  string r = "";
  for(int i = 32; i >= 0; i--) {
    if((bool)(x & (1ll<<i)))
      r += '0';
     else
      r += '1';

  }
  return r;
}

int toInt(string x) {
  int c = 0;
  for(int i = 0; i < 33; i++) {
    if(x[i] == '1')
      c += (1ll<<(32-i));
  }
  return c;
}

void insert(string x) {
  No *aux = head;
  for(char c: x) {

    if(aux->trans[c].ff) {
      aux->trans[c].ss += 1;
    } else {
      aux->trans[c].ff = makeNo();
      aux->trans[c].ss = 1;
    }
    aux = aux->trans[c].ff;
  }
  aux->word = true;
}

void del(string x) {
  No *aux = head;
  for(char c: x) {
    if((--(aux->trans[c].ss)) == 0) {
      aux->trans[c].ff = NULL;
      return;
    }
    aux = aux->trans[c].ff;
  }
}

string query(string x) {
  No *aux = head;
  string r = "";
  for(char c: x) {
    if(aux->trans[c].ff) {
      r += c;
```

```
      aux = aux->trans[c].ff;
    } else {
      r += comp(c);
      aux = aux->trans[comp(c)].ff;
    }
  }
  return r;
}

int main() {
  No *head = makeNo();
}
```

6.4. TRIE (MAXIMUM XOR BETWEEN TWO ELEMENTS)

```
1. Dada uma trie de números binários e um numero X, tente achar o número
    máximo que resultante da operação XOR

Ex: Para o número 10(=(1010)2), o número que resulta no xor máximo é (0101)2
    , tente acha-lo na trie.
```

6.5. TRIE (MAXIMUM XOR SUM)

```
// XOR(L,R) = XOR(1,L-1) ^ XOR(1,R)
ans= pre = 0
Trie.insert(0)
for i=1 to N:
    pre = pre XOR a[i]
    Trie.insert(pre)
    ans=max(ans, Trie.query(pre))
print ans

// a funcao query é a mesma da maximum xor between two elements
```

6.6. Z-FUNCTION - $O(n)$

```
// What is Z Array?
// For a string str[0..n-1], Z array is of same length as string. An element
    Z[i] of Z array stores length of the longest substring starting from
    str[i] which is also a prefix of str[0..n-1]. The first entry of Z array
    is meaning less as complete string is always prefix of itself.
// Example:
// Index
// 0   1   2   3   4   5   6   7   8   9  10  11
// Text
// a   a   b   c   a   a   b   x   a   a   a   z
// Z values
// X   1   0   0   3   1   0   0   2   2   1   0
// More Examples:
// str  = "aaaaaa"
// Z[]  = {x, 5, 4, 3, 2, 1}

// str = "aabaacd"
// Z[] = {x, 1, 0, 2, 1, 0, 0}

// str = "abababab"
// Z[] = {x, 0, 6, 0, 4, 0, 2, 0}

vector<int> z_function(string s) {
  int n = (int) s.length();
  vector<int> z(n);
  for (int i = 1, l = 0, r = 0; i < n; ++i) {
    if (i <= r)
      z[i] = min (r-i+1, z[i - l]);
    while (i + z[i] < n && s[z[i]] == s[i + z[i]])
      ++z[i];
    if (i + z[i] - 1 > r)
      l = i, r = i + z[i] - 1;
  }
  return z;
}
```

7. Data Structures

7.1. RMQ MIN-MAX + LAZY PROPAGATION

```
struct node {

  int menor, maior, lazy;
  node() {}
  node(int menor, int maior, int lazy) : menor(menor), maior(maior),
    lazy(lazy) {}

  node operator^(const node &x) const {
    return node(min(menor, x.menor), max(maior, x.maior), 0);
  }

};

int arr[1000003];
node tree[4123456];

void propagate(int l, int r, int pos) {

  if(tree[pos].lazy != 0) {
    tree[pos].menor += tree[pos].lazy;
    tree[pos].maior += tree[pos].lazy;
    if(l != r) {
      tree[2*pos+1].lazy += tree[pos].lazy;
      tree[2*pos+2].lazy += tree[pos].lazy;
    }
    tree[pos].lazy = 0;
  }

}

node build(int l, int r, int pos) {
  if(l == r)
    return tree[pos] = node(arr[l], arr[l], 0);

  int mid = (l+r) >> 1;
  return tree[pos] = build(l, mid, 2*pos+1)^build(mid + 1, r, 2*pos+2);
}

node query(int l, int r, int i, int j, int pos) {

  propagate(l, r, pos);

  if(l > r || l > j || r < i) {
    return node(INF, -INF, 0);
  }

  if(i <= l && r <= j) {
    return tree[pos];
  }

  int mid = (l+r)>>1;
  return query(l, mid, i, j, 2*pos+1)^query(mid+1, r, i, j, 2*pos+2);
}

// range sum update
node update(int l, int r, int i, int j, int v, int pos) {

  propagate(l, r, pos);

  if(l > r || l > j || r < i) {
    return tree[pos];
  }

  if(i <= l && r <= j) {
    tree[pos].lazy += v;
    propagate(l, r, pos);
    return tree[pos] = node(tree[pos].menor, tree[pos].maior, 0);
  }

  int mid = (l+r)>>1;
  return tree[pos] = update(l, mid, i, j, v, 2*pos+1)^update(mid+1, r, i, j,
    v, 2*pos+2);

}
```

7.2. ARVORE BINARIA

```
// C program to demonstrate delete operation in binary search tree
#include<stdio.h>
#include<stdlib.h>

struct node  {
  int key;
  struct node *left, *right;
};

// A utility function to create a new BST node
struct node *newNode(int item) {
  struct node *temp =  (struct node *)malloc(sizeof(struct node));
  temp->key = item;
  temp->left = temp->right = NULL;
  return temp;
}

// A utility function to do inorder traversal of BST
void inorder(struct node *root) {
  if (root != NULL) {
    inorder(root->left);
    printf("%d ", root->key);
    inorder(root->right);
  }
}

/* A utility function to insert a new node with given key in BST */
struct node* insert(struct node* node, int key) {
  /* If the tree is empty, return a new node */
  if (node == NULL) return newNode(key);

  /* Otherwise, recur down the tree */
  if (key < node->key)
    node->left  = insert(node->left, key);
  else
    node->right = insert(node->right, key);

  /* return the (unchanged) node pointer */
  return node;
}

/* Given a non-empty binary search tree, return the node with minimum
  key value found in that tree. Note that the entire tree does not
  need to be searched. */
struct node * minValueNode(struct node* node) {
  struct node* current = node;

  /* loop down to find the leftmost leaf */
```

```
  while (current->left != NULL)
    current = current->left;

  return current;
}

/* Given a binary search tree and a key, this function deletes the key
  and returns the new root */
struct node* deleteNode(struct node* root, int key) {
  // base case
  if (root == NULL) return root;

  // If the key to be deleted is smaller than the root's key,
  // then it lies in left subtree
  if (key < root->key)
    root->left = deleteNode(root->left, key);

  // If the key to be deleted is greater than the root's key,
  // then it lies in right subtree
  else if (key > root->key)
    root->right = deleteNode(root->right, key);

  // if key is same as root's key, then This is the node
  // to be deleted
  else {
    // node with only one child or no child
    if (root->left == NULL) {
      struct node *temp = root->right;
      free(root);
      return temp;
    } else if (root->right == NULL) {
      struct node *temp = root->left;
      free(root);
      return temp;
    }

    // node with two children: Get the inorder successor (smallest
    // in the right subtree)
    struct node* temp = minValueNode(root->right);

    // Copy the inorder successor's content to this node
    root->key = temp->key;

    // Delete the inorder successor
    root->right = deleteNode(root->right, temp->key);
  }
  return root;
}

// Driver Program to test above functions
int main() {
  /* Let us create following BST
             50
          /     \
         30      70
        /  \    /  \
      20   40  60   80 */
  struct node *root = NULL;
  root = insert(root, 50);
  root = insert(root, 30);
  root = insert(root, 20);
  root = insert(root, 40);
  root = insert(root, 70);
  root = insert(root, 60);
  root = insert(root, 80);

  printf("Inorder traversal of the given tree \n");
  inorder(root);

  printf("\nDelete 20\n");
  root = deleteNode(root, 20);
  printf("Inorder traversal of the modified tree \n");
  inorder(root);

  printf("\nDelete 30\n");
  root = deleteNode(root, 30);
  printf("Inorder traversal of the modified tree \n");
  inorder(root);

  printf("\nDelete 50\n");
  root = deleteNode(root, 50);
  printf("Inorder traversal of the modified tree \n");
  inorder(root);

  return 0;
}
```

7.3. SQRT DECOMPOSITION

```
// Problem: Sum from l to r
// Ver MO'S ALGORITHM
// -------------------------------------
int getId(int indx,int blockSZ) {
    return indx/blockSZ;
}
void init(int sz) {
    for(int i=0; i<=sz; i++)
  BLOCK[i]=inf;
}
int query(int left, int right) {
int startBlockIndex=left/sqrt;
int endIBlockIndex = right / sqrt;
int sum = 0;
for (int i = startBlockIndex + 1; i < endIBlockIndex; i++) {
        sum += blockSums[i];
    }
for(i=left...(startBlockIndex*BLOCK_SIZE-1))
  sum += a[i];
for(j = endIBlockIndex*BLOCK_SIZE ... right)
  sum += a[i];
}
```

7.4. MO'S ALGORITHM (MOST FREQUENT VALUE IN INTERVALS) - $O((m+n)*\sqrt{n})$

```
struct Tree {
  int l, r, ind;
};
Tree query[311111];
int arr[311111];
int freq[1111111];
int ans[311111];
int block = sqrt(n), cont = 0;

bool cmp(Tree a, Tree b) {
  if(a.l/block == b.l/block)
    return a.r < b.r;
  return a.l/block < b.l/block;
```

```
}

void add(int pos) {
  freq[arr[pos]]++;
  if(freq[arr[pos]] == 1) {
    cont++;
  }
}
void del(int pos) {
  freq[arr[pos]]--;
  if(freq[arr[pos]] == 0)
    cont--;
}
int main () {
  int n; cin >> n;
  block = sqrt(n);

  for(int i = 0; i < n; i++) {
    cin >> arr[i];
    freq[arr[i]] = 0;
  }

  int m;  cin >> m;

  for(int i = 0; i < m; i++) {
    cin >> query[i].l >> query[i].r;
    query[i].l--, query[i].r--;
    query[i].ind = i;
  }
  sort(query, query + m, cmp);

  int s,e;
  s = e = query[0].l;
  add(s);
  for(int i = 0; i < m; i++) {
    while(s > query[i].l)
      add(--s);
    while(s < query[i].l)
      del(s++);
    while(e < query[i].r)
      add(++e);
    while(e > query[i].r)
      del(e--);
    ans[query[i].ind] = cont;

  }
  for(int i = 0; i < m; i++)
    cout << ans[i] << endl;
}
```

7.5. MERGE SORT TREE (K-ESIMO MAIOR ELEMENTO NUM INTERVALO, VALORES MAIORES QUE K NUM INTERVALO, ...)

```
// retornar a qtd de números maiores q um numero k numa array de i...j
struct Tree {
  vector<int> vet;
};
Tree tree[4*(int)3e4];
int arr[(int)5e4];

int query(int l,int r, int i, int j, int k, int pos) {
  if(l > j || r < i)
    return 0;

  if(i <= l && r <= j) {
    auto it = upper_bound(tree[pos].vet.begin(), tree[pos].vet.end(), k);
    return tree[pos].vet.end()-it;
  }

  int mid = (l+r)>>1;
  return query(l, mid, i, j, k, 2*pos+1) + query(mid+1,r,i,j,k,2*pos+2);
}

void build(int l, int r, int pos) {

  if(l == r) {
    tree[pos].vet.pb(arr[l]);
    return;
  }

  int mid = (l+r)>>1;
  build(l, mid, 2*pos+1);
  build(mid + 1, r, 2*pos+2);

  merge(tree[2*pos+1].vet.begin(), tree[2*pos+1].vet.end(),
    tree[2*pos+2].vet.begin(), tree[2*pos+2].vet.end(),
    back_inserter(tree[pos].vet));
}
```

7.6. ORDENAÇÃO DE ESTRUTURAS(PQ,ETC)

```
struct cmp {
  bool operator(ii a, ii b) {
  //ordena primeiro pelo first(decrescente), dps pelo second(crescente)
    if(a.first == b.first)
      return a.second < b.second;
    return a.first > b.first;
  }
}
Ex: pq<ii,vector<ii>,cmp> fila;
```

7.7. POLICY BASED DATA STRUCTURES - ORDERED SET

```
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/trie_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;

ordered_set X;
X.insert(1); X.insert(2);
X.insert(4); X.insert(8);
X.insert(16);
// 1, 2, 4, 8, 16
// retorna o k-ésimo maior elemento a partir de 0
cout<<*X.find_by_order(1)<<endl; // 2
cout<<*X.find_by_order(2)<<endl; // 4
cout<<*X.find_by_order(4)<<endl; // 16
```

```
cout<<(end(X)==X.find_by_order(6))<<endl; // true

// retorna o número de itens estritamente menores que o número
cout<<X.order_of_key(-5)<<endl;  // 0
cout<<X.order_of_key(1)<<endl;   // 0
cout<<X.order_of_key(3)<<endl;   // 2
cout<<X.order_of_key(4)<<endl;   // 2
cout<<X.order_of_key(400)<<endl; // 5
```

7.8. CONTANDO INVERSÕES

```
int count_inversion_merge(int array[], int first, int last) {
  int mid = (first+last)/2;
  int ai = first, bi = mid+1;
  int final[last-first+1], finali=0;
  int inversion = 0, i;

  while (ai <= mid && bi <= last) {
    if (array[ai] <= array[bi]) {
      final[finali++] = array[ai++];
    } else {
      final[finali++] = array[bi++];
      inversion += mid - ai + 1;
    }
  }

  while (ai <= mid)
    final[finali++] = array[ai++];

  while (bi <= last)
    final[finali++] = array[bi++];

  for (i=0 ; i<last-first+1 ; i++)
    array[i+first] = final[i];

  return inversion;
}

int count_inversion(int array[], int a, int b) {
  // a começa com 0 e b com n-1
  int x, y, z, mid;
  if (a >= b) return 0;

  mid = (a+b)/2;

  x = count_inversion(array, a, mid);
  y = count_inversion(array, mid+1, b);
  z = count_inversion_merge(array, a, b);

  return x+y+z;
}
```

8. GRAPHS

8.1. CICLO GRAFO - $O(V+E)$

```
int n;
vector<vector<int>> adj;
vector<char> color;
vector<int> parent;
int cycle_start, cycle_end;

bool dfs(int v) {
  color[v] = 1;
  for (int u : adj[v]) {
    if (color[u] == 0) {
      parent[u] = v;
      if (dfs(u))
        return true;
    } else if (color[u] == 1) {
      cycle_end = v;
      cycle_start = u;
      return true;
    }
  }
  color[v] = 2;
  return false;
}

void find_cycle() {
  color.assign(n, 0);
  parent.assign(n, -1);
  cycle_start = -1;

  for (int v = 0; v < n; v++) {
    if (dfs(v))
      break;
  }

  if (cycle_start == -1) {
    cout << "Acyclic" << endl;
  } else {
    vector<int> cycle;
    cycle.push_back(cycle_start);
    for (int v = cycle_end; v != cycle_start; v = parent[v])
      cycle.push_back(v);
    cycle.push_back(cycle_start);
    reverse(cycle.begin(), cycle.end());

    cout << "Cycle found: ";
    for (int v : cycle)
      cout << v << " ";
    cout << endl;
  }
}
```

8.2. CHECA GRAFO BIPARTIDO - $O(V+E)$

```
bool isBipartite(int src){

  int colorArr[V];
  for (int i = 0; i < V; ++i)
    colorArr[i] = -1;
  colorArr[src] = 1;

  queue <int> q; q.push(src);
```

```
9
10  while (!q.empty()) {
11    int u = q.front(); q.pop();
12
13    // Find all non-colored adjacent vertices
14    for (auto it = adj[u].begin(); it != adj[u].end(); it++) {
15    //Return false if there is a self-loop
16        if (u == *it)
17            return false;
18      // An edge from u to v exists and destination v is not colored
19
20      if (colorArr[*it] == -1) {
21        // Assign alternate color to this adjacent v of u
22        colorArr[*it] = 1 - colorArr[u];
23        q.push(*it);
24      }
25      // An edge from u to v exists and destination v is colored with same
        color as u
26      else if (colorArr[*it] == colorArr[u])
27        return false;
28    }
29  }
30  // If we reach here, then all adjacent vertices can be colored with
31  // alternate color
32  return true;
33 }
```

8.3. BELLMAN FORD (MENOR CAMINHO ARESTAS NEGATIVAS) - $O(V*E)$

```
1  void BellmanFord(vector<edge> edges,int src, int V, int E)  {
2    // V = qtd of vertices, E = qtd de arestas
3    int dist[V];
4
5    // Step 1: Initialize distances from src to all other vertices
6    // as INFINITE
7    for (int i = 0; i < V; i++)
8      dist[i] = INF;
9    dist[src] = 0;
10   // Step 2: Relax all edges |V| - 1 times. A simple shortest path from src
       to any other vertex can have at-most |V| - 1 edges
11   for (int i = 1; i <= V-1; i++) {
12     for (int j = 0; j < E; j++) {
13       int u = edges[j].src;
14       int v = edges[j].dest;
15       int weight = edges[j].weight;
16       if (dist[u] != INF && dist[u] + weight < dist[v])
17         dist[v] = dist[u] + weight;
18     }
19   }
20   // Step 3: check for NEGATIVE-WEIGHT CYCLES.  The above step guarantees
       shortest distances if graph doesn't contain negative weight cycle.  If
       we get a shorter path, then there is a cycle.
21   for (int i = 0; i < E; i++) {
22     int u = edges[i].src;
23     int v = edges[i].dest;
24     int weight = edges[i].weight;
25     if (dist[u] != INF && dist[u] + weight < dist[v])
26       printf("Graph contains negative weight cycle");
27   }
28   printArr(dist, V);
29 }
```

8.4. DIAMETRO EM ARVORE (MAIOR CAMINHO ENTRE DOIS VERTICES)

```
1  Calcula qual o vértice a mais distante de um qualquer vértice X e do vértice
       A calcula-se o vértice B mais distante dele.
```

8.5. PONTES NUM GRAFO - $O(V+E)$

```
1  //SE TIRA-LAS O GRAFO FICA DESCONEXO
2  // OBS: PRESTAR ATENCAO EM SELF-LOOPS, é MELHOR NãO ADICIONA-LOS
3  // SO FUNCIONA EM GRAFO NãO DIRECIONADO
4  int t=1;
5  vector<int> T((int)2e6,0); //Tempo necessário para chegar naquele vértice na
       dfs
6  vector<int> adj[(int)2e6];
7  vector<int> Low((int)2e6); // Tempo "mínimo" para chegar naquele vértice na
       dfs
8  vector<int> ciclo((int)2e6, false);
9  vector<ii> bridges;
10 void dfs(int u, int p){
11   Low[u] = T[u] = t;
12   t++;
13   for(auto v : adj[u]){
14     if(v==p){
15       //checa arestas paralelas
16       p=-1;
17       continue;
18     }
19     //se ele ainda não foi visited
20     else if(T[v]==0){
21       dfs(v,u);
22       Low[u]=min(Low[u], Low[v]);
23       if(Low[v]>T[u]) {
24         bridges.pb(ii(min(u,v), (max(u,v))));
25       // ponte de u para v
26       }
27     }
28     else
29       Low[u]=min(Low[u], T[v]);
30     ciclo[u] |= (T[u]>=Low[v]);
31     //checa se o vértice u faz parte de um ciclo
32   }
33 }
34
35 void clear() {
36
37   for(int i = 0; i <= n; i++) {
38     T[i] = 0, Low[i] = 0, adj[i].clear(), ciclo[i] = false;
39   }
40   bridges.clear();
41
42 }
43
44 signed main () {
45
46   for(int i = 0; i < n; i++)
47     if(T[i] == 0)
48       dfs(i, -1);
49
50   sort(bridges.begin(), bridges.end());
51
52   cout << (int)bridges.size() << endl;
53   for(int i = 0; i < bridges.size(); i++) {
54     cout << bridges[i].ff << " - " << bridges[i].ss << endl;
55   }
56   cout << endl;
```

```
  clear();

}
```

8.6. PONTOS DE ARTICULAÇÃO NUM GRAFO (se retirar esses vértices o grafo fica desconexo) - $O(V+E)$

```
// SE TIRAR TAIS VERTICES O GRAFO FICA DESCONEXO

vector<bool> ap(100000,false);
vector<int> low(100000,0), T(100000,0);
int tempo = 1;
list<int> adj[100000];

void artPoint(int u, int p) {

  low[u] = T[u] = tempo++;
  int children = 0;

  for(int v: adj[u]) {

  // cuidado com arestas paralelas
  // se tiver nao podemos fazer assim

    if(T[v] == 0) {

      children++;
      artPoint(v,u);
      low[u] = min(low[v],low[u]);

    if(p == -1 && children > 1) {
      ap[u] = true;
    }

    if(p != -1 && low[v] > T[u])
      ap[u] = true;
    } else if(v != p)
      low[u] = min(low[u], T[v]);

  }
}

int main() {

  for(int i = 0; i < n; i++)
    if(T[i] == 0)
      artPoint(i,-1);
}
```

8.7. LCA (shortest path)

```
// pra achar so o lca é so tirar o q ta comentado com (// short path)
int anc[105000][(int)log2(150000)+2];

vector<ii> adj[105000];
vector<int> level(105000), dist(105000), vis(105000,false);

void dfs(int u, int p, int d = 1) {

  level[u] = d;
  vis[u] = true;

  for(ii x: adj[u]) {
    if(vis[x.ff])
      continue;
    dist[x.ff] = dist[u] + x.ss; // short path
    dfs(x.ff,u,d+1);
  }
}

void build(int n) {

  for(int i = 0; i <= n; i++) {
    for(int j = 0; (1<<j) < n; j++) {
      anc[i][j] = -1;
    }
  }

  // ler a aresta i -> x com peso c
  for(int i = 1; i < n; i++) {
    int x,c;
    cin >> x >> c;
    adj[x].pb(ii(i,c));
    adj[i].pb(ii(x,c));
    anc[i][0] = x;
  }


  dist[0] = 0; // short path
  dfs(0,-1);

  for(int j = 1; (1<<j) < n; j++) {
    for(int i = 0; i < n; i++) {
      if(anc[i][j-1] != -1)
        anc[i][j] = anc[anc[i][j-1]][j-1];
    }
  }

}
int lca(int a, int b) {

  int ac = a, bc = b;
  // a esta mais embaixo na arvore
  if(level[b] > level[a])
    swap(a,b);

  int lg = log2(level[a]);

  // colocando a e b no msm nivel
  for(int i = lg; i >= 0; i--) {
    if(level[a]-(1<<i) >= level[b]) {
      a = anc[a][i];
    }
  }
  // return a; // to find lca
  if(a == b)
    return dist[ac] + dist[bc] - 2*dist[a]; // short path

  // achando o filho do lca
  for(int i = lg; i >= 0; i--) {
    if(anc[a][i] != -1 && anc[a][i] != anc[b][i]) {
      a = anc[a][i];
      b = anc[b][i];
    }
  }
  // anc[a][0] é o LCA
  // return anc[a][0]; retorna o lca
```

```
  //cout << "LCA  = " << anc[a][0] << endl;
  return dist[ac] + dist[bc] - 2*dist[anc[a][0]];

}
/*
Codigo para voltar k posicoes a partir de x
for(int i = 0; i < 32; i++) {
  if(k & (1<<i))
    x = anc[x][i];
}
return x;
*/
```

8.8. FORD FULKERSSON (MAXIMUM FLOW) - $O(V*(E^2))$

```
int rGraph[2000][2000];
int graph[2000][2000];

int V;
bool bfs(int s, int t, int parent[]) {
  bool visited[V];
  memset(visited, 0, sizeof(visited));

  // Create a queue, enqueue source vertex and mark source vertex
  // as visited
  queue <int> q;
  q.push(s);
  visited[s] = true;
  parent[s] = -1;

  // Standard BFS Loop
  while (!q.empty()) {
    int u = q.front();
    q.pop();

    for (int v=0; v<V; v++) {
      if (visited[v]==false && rGraph[u][v] > 0) {
        q.push(v);
        parent[v] = u;
        visited[v] = true;
      }
    }
  }
  // If we reached sink in BFS starting from source, then return true, else
    false
  return (visited[t] == true);
}

// Returns the maximum flow from s to t in the given graph
int fordFulkerson(int s, int t) {
  int u, v;
  // Create a residual graph and fill the residual graph with given
    capacities in the original graph as residual capacities in residual
    graph residual capacity of edge from i to j (if there is an edge. If
    rGraph[i][j] is 0, then there is not)
  for (u = 0; u < V; u++)
    for (v = 0; v < V; v++)
      rGraph[u][v] = graph[u][v];

  int parent[V];  // This array is filled by BFS and to store path

  int max_flow = 0;// There is no flow initially

  // Augment the flow while tere is path from source to sink
  while (bfs(s, t, parent)) {
    // Find minimum residual capacity of the edges along the path filled by
    BFS. Or we can say find the maximum flow through the path found.
    int path_flow = INT_MAX;
    for (v=t; v!=s; v=parent[v]) {
      u = parent[v];
      path_flow = min(path_flow, rGraph[u][v]);
    }

    // update residual capacities of the edges and reverse edges
    // along the path
    for (v=t; v != s; v=parent[v]) {
      u = parent[v];
      rGraph[u][v] -= path_flow;
      rGraph[v][u] += path_flow;
    }

    // Add path flow to overall flow
    max_flow += path_flow;
  }

  // Return the overall flow
  return max_flow;
}

// PRINT THE FLOW AFTER RUNNING THE ALGORITHM
void print(int n) {
  for(int i = 1; i <= m; i++) {
    for(int j = m+1; j <= m*2; j++) {
      cout << "flow from i(left) to j(right) is " << graph[i][j] -
    rGraph[i][j] << endl;
    }
  }
}

void addEdge(int l, int r, int n, int x) {
  graph[l][r+n] = x;
}

void addEdgeSource(int l, int x) {
  graph[0][l] = x;
}

void addEdgeSink(int r, int n, int x) {
  graph[r+n][V-1] = x;
}
```

8.9. DINIC (MAXIMUM FLOW) -> $O(E+(V^2))$

```
struct Dinic {

  struct FlowEdge{
    int v, rev, c, cap;
    FlowEdge() {}
    FlowEdge(int v, int c, int cap, int rev) : v(v), c(c), cap(cap),
    rev(rev) {}
  };

  vector< vector<FlowEdge> >  adj;
  vector<int> level, used;
  int src, snk, V;
  int sz;
  int max_flow;
  Dinic(){}
```

```
  Dinic(int n){
    src = 0;
    snk = n+1;
    adj.resize(n+2, vector< FlowEdge >());
    level.resize(n+2);
    used.resize(n+2);
    sz = n+2;
    V = n+2;
    max_flow = 0;
  }

  void add_edge(int u, int v, int c){
    int id1 = adj[u].size();
    int id2 = adj[v].size();
    adj[u].pb(FlowEdge(v, c, c, id2));
    adj[v].pb(FlowEdge(u, 0, 0, id1));
  }

  void add_to_src(int v, int c){
    adj[src].pb(FlowEdge(v, c, c, -1));
  }

  void add_to_snk(int u, int c){
    adj[u].pb(FlowEdge(snk, c, c, -1));
  }

  bool bfs(){
    for(int i=0; i<sz; i++){
      level[i] = -1;
    }

    level[src] = 0;
    queue<int> q; q.push(src);

    while(!q.empty()){
      int cur = q.front();
      q.pop();
      for(FlowEdge e : adj[cur]){
        if(level[e.v] == -1 && e.c > 0){
          level[e.v] = level[cur]+1;
          q.push(e.v);
        }
      }
    }

    return (level[snk] == -1 ? false : true);
  }

  int send_flow(int u, int flow){
    if(u == snk) return flow;

    for(int &i = used[u]; i<adj[u].size(); i++){
      FlowEdge &e = adj[u][i];

      if(level[u]+1 != level[e.v] || e.c <= 0) continue;

      int new_flow = min(flow, e.c);
      int adjusted_flow = send_flow(e.v, new_flow);

      if(adjusted_flow > 0){
        e.c -= adjusted_flow;
        if(e.rev != -1) adj[e.v][e.rev].c += adjusted_flow;
        return adjusted_flow;
      }
    }

    return 0;
  }

  void calculate(){
    if(src == snk){max_flow = -1; return;} //not sure if needed

    max_flow = 0;

    while(bfs()){
      for(int i=0; i<sz; i++) used[i] = 0;
      while(int inc = send_flow(src, INF)) max_flow += inc;
    }

  }

  vector< ii > mincut(){
    bool vis[sz];
    for(int i=0; i<sz; i++) vis[i] = false;
    queue<int> q;
    q.push(src);
    vis[src] = true;
    while(!q.empty()){
      int cur = q.front();
      q.pop();
      for(FlowEdge e : adj[cur]){
        if(e.c > 0 && !vis[e.v]){
          q.push(e.v);
          vis[e.v] = true;
        }
      }
    }
    vector< ii > cut;
    for(int i=1; i<=sz-2; i++){
      if(!vis[i]) continue;
      for(FlowEdge e : adj[i]){
        if(1 <= e.v && e.v <= sz-2 && !vis[e.v] && e.cap > 0 && e.c == 0)
    cut.pb(ii(i, e.v));
      }
    }
    return cut;
  }

  vector< ii > min_edge_cover(){
    bool covered[sz];
    for(int i=0; i<sz; i++) covered[i] = false;
    vector< ii > edge_cover;
    for(int i=1; i<sz-1; i++){
      for(FlowEdge e : adj[i]){
        if(e.cap == 0 || e.v > sz-2) continue;
        if(e.c == 0){
          edge_cover.pb(ii(i, e.v));
          covered[i] = true;
          covered[e.v] = true;
          break;
        }
      }
    }
    for(int i=1; i<sz-1; i++){
      for(FlowEdge e : adj[i]){
        if(e.cap == 0 || e.v > sz-2) continue;
        if(e.c == 0) continue;
        if(!covered[i] || !covered[e.v]){
          edge_cover.pb(ii(i, e.v));
          covered[i] = true;
```

```
          covered[e.v] = true;
        }
      }
    }
    return edge_cover;
  }

  vector<vector<int>> allFlow() {
    vector<int> row(V, 0);
    vector<vector<int>> ret(V, row);

    for(int i = 0; i < V; i++) {
      for(FlowEdge x: adj[i]) {
        // flow from vertex i to x.v
        ret[i][x.v] = x.cap - x.c;
      }
    }

    // for(int i = 0; i < V; i++) {
    //   for(int j  = 0; j < V; j++) {
    //     cout << ret[i][j] << ' ';
    //   }
    //   cout << endl;
    // }

    return ret;
  }

};
```

8.10. CAMINHO EULERIANO (Caminho para visitar todas as arestas) - $O(V+E)$

```
void DFSUtil(int v, bool vis[]) {
    // Mark the current node as vis and print it
    vis[v] = true;

    // Recur for all the vertices adjacent to this vertex
    vector<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
      if (!vis[*i])
        DFSUtil(*i, vis);
}

// Method to check if all non-zero degree vertices are connected.
// It mainly does DFS traversal starting from
bool isConnected() {
    // Mark all the vertices as not vis
    bool vis[V];
    int i;
    for (i = 0; i < V; i++)
      vis[i] = false;

    // Find a vertex with non-zero degree
    for (i = 0; i < V; i++)
      if (adj[i].size() != 0)
        break;

    // If there are no edges in the graph, return true
    if (i == V)
      return true;

    // Start DFS traversal from a vertex with non-zero degree
    DFSUtil(i, vis);

    // Check if all non-zero degree vertices are vis
    for (i = 0; i < V; i++)
      if (vis[i] == false && adj[i].size() > 0)
        return false;

    return true;
}

/* The function returns one of the following values
   0 --> If graph is not Eulerian
   1 --> If graph has an Euler path (Semi-Eulerian)
   2 --> If graph has an Euler Circuit (Eulerian)  */
int isEulerian() {
  // Check if all non-zero degree vertices are connected
  if (isConnected() == false)
    return 0;

  // Count vertices with odd degree
  int odd = 0;
  for (int i = 0; i < V; i++)
    if (adj[i].size() & 1)
      odd++;

  // If count is more than 2, then graph is not Eulerian
  if (odd > 2)
    return 0;

  // If odd count is 2, then semi-eulerian. If odd count is 0, then eulerian
  // Note that odd count can never be 1 for undirected graph
  return (odd) ? 1 : 2;
}
```

8.11. DIJKSTRA COM PRIORITY QUEUE - $O(E*(logV)))$

```
int Dijkstra(int src, int dest, int n) {

  priority_queue<ii,vector <ii>,greater<ii>> pq;
  vector<int> dist(n+1, INF);
  // vector<vector<int> > parent(n+1);
  // for(int i = 0; i <= n; i++)
  //   parent[i].pb(i);
  pq.push(make_pair(0, src));
  dist[src] = 0;

  while (!pq.empty()) {
    int u = pq.top().ss;
    pq.pop();

    for (ii x: adj[u]) {
      int v = x.ff;
      int w = x.ss;

      if (dist[u] + w < dist[v]) {
        // parent[v].clear();
        // parent[v].pb(u);
        dist[v] = dist[u] + w;
        pq.push(ii(dist[v], v));
      }
      // else if(dist[u] + w == dist[v]) {
      //   parent[v].pb(u);
      // }
    }
  }
```

```
30
31    return dist[dest];
32  }
```

8.12. ORDENAÇÃO TOPOLOGICA (FILA) - $O(V+E)$

```
1   void topologicalSort(int n) {
2     vector<int> in_degree(n, 0);
3
4     for (int u=0; u<n; u++){
5       vector<int>::iterator itr;
6       for (itr = adj[u].begin(); itr != adj[u].end(); itr++)
7         in_degree[*itr]++;
8     }
9
10    queue<int> q;
11    for (int i = 0; i < n; i++)
12      if (in_degree[i] == 0)
13        q.push(i);
14
15    int cnt = 0;
16    vector <int> top_order;
17    while (!q.empty()) {
18      int u = q.front();  q.pop();
19      top_order.push_back(u);
20      vector<int>::iterator itr;
21      for (itr = adj[u].begin(); itr != adj[u].end(); itr++) {
22        if (--in_degree[*itr] == 0)
23          q.push(*itr);
24      cnt++;
25    }
26
27    if (cnt != n) {
28      cout << "There exists a cycle in the graph\n";
29      return;
30    }
31
32    for (int i=0; i<top_order.size(); i++)
33      cout << top_order[i] << " ";
34    cout << endl;
35  }
```

8.13. KRUSKAL (UNION FIND - $O(log(n))$) - $O(E*log(V))$

```
1   struct edge {
2     int u, v, w;
3     edge() {}
4     edge(int u, int v, int w) : u(u), v(v), w(w) {}
5   }
6
7   vector<edge> edges((int)2e6);
8   vector<int> root((int)2e6), sz((int)2e6);
9
10  void init(int n) {
11
12    iota(root.begin(), root.begin() + n + 1, 0);
13    fill(sz.begin(), sz.begin()+ n + 1, 1);
14
15  }
16
17  int Find(int x) {
18    if(root[x] == x)
19      return x;
20    return root[x] = Find(root[x]);
21  }
22
23  bool Union(int p, int q) {
24    p = Find(p), q = Find(q);
25    if(p == q)
26      return false;
27
28    if(sz[p] > sz[q]) {
29      root[q] = p;
30      sz[p] += sz[q];
31    } else {
32      root[p] = q;
33      sz[q] += sz[p];
34    }
35    return true;
36  }
37
38
39  int kruskal(int n, int m) {
40
41    init(n);
42
43    int c = 0;
44    for(int i = 0; i < (int)edges.size(); i++) {
45      if(Union(edges[i].u, edges[i].v)) {
46        c += edges[i].w;
47      }
48    }
49
50    // returns weight of mst
51    return c;
52  }
```

8.14. FLOYD WARSHALL - $O(V^3)$

```
1   // OBS: ZERAR adj[i][i] sempre
2   for(int i = 0; i < n; i++)
3     adj[i][i] = 0;
4
5   for(int k = 0; k < n; k++) {
6     for(int i = 0; i < n; i++) {
7       for(int j = 0; j < n; j++) {
8         adj[i][j] = min(adj[i][j],adj[i][k] + adj[k][j]);
9       }
10    }
11  }
```

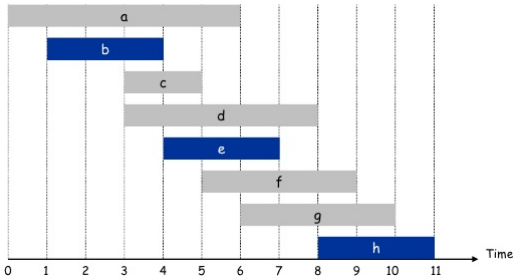
8.15. SCC (Kosaraju) - $O(V+E)$

```
1   int comp[312345];
2   vector<int> adj[312345];
3   // grafo reverso de adj
4   vector<int> trans[312345];
5   vector<int> scc[312345];
6   int V;
7
8   void dfsTrans(int u, int id, int comp[]) {
9     comp[u] = id;
10    scc[id].push_back(u);
11
12    for (int v: trans[u])
13      if (!comp[v])
```

```
      dfsTrans(v, id, comp);
}

void getTranspose() {
  for (int u = 0; u < V; u++)
    for(int v: adj[u])
      trans[v].push_back(u);
}

void dfsFillOrder(int u, int comp[], stack<int> &Stack) {
  comp[u] = true;

  for(int v: adj[u])
    if(!comp[v])
      dfsFillOrder(v, comp, Stack);

  Stack.push(u);
}

// The main function that finds and prints all strongly connected
// components
void computeSCC() {

  stack<int> Stack;
  // Fill vertices in stack according to their finishing times
  for(int i = 0; i < V; i++)
    if(comp[i] == false)
      dfsFillOrder(i, comp, Stack);

  // Create a reversed graph
  getTranspose();

  memset(comp, 0, sizeof(comp));

  // Now process all vertices in order defined by Stack
  int id = 1;
  while(Stack.empty() == false) {
    int v = Stack.top();
    Stack.pop();

    if(comp[v] == false)
      dfsTrans(v, id++, comp);
  }
}
```

9. Diverse

9.1. INTERVAL SCHEDULING




```
1 -> Ordena pelo final do evento, depois pelo inicio.
2 -> Vai iterando pelos eventos, se eles não tiverem horário em comum então
     adiciona o evento à lista.
```

9.2. PROBLEMA DAS OITO RAINHAS

```
#define N 4
bool isSafe(int mat[N][N],int row,int col) {
  for(int i = row - 1; i >= 0; i--)
    if(mat[i][col])
      return false;
  for(int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--,j--)
    if(mat[i][j])
      return false;
  for(int i = row - 1, j = col + 1; i >= 0 && j < N; i--,j++)
    if(mat[i][j])
      return false;
  return true;
}
// inicialmente a matriz esta zerada
int queen(int mat[N][N], int row  = 0) {
  if(row >= N) {
    for(int i = 0; i < N; i++) {
      for(int j = 0; j < N; j++) {
        cout << mat[i][j] << ' ';
      }
      cout << endl;
    }
    cout << endl << endl;
    return false;
  }
  for(int i = 0; i < N; i++) {
    if(isSafe(mat,row,i)) {
      mat[row][i] = 1;
      if(queen(mat,row+1))
        return true;
      mat[row][i] = 0;
    }
  }
  return false;
}
```

9.3. 3SUM PROBLEM (a[x]+a[y]+a[z] = valor) - $O(n^2)$

```
// vetor arr e valor x, a soma de três valores desse vetor deve ser igual a x

bool sum3(int arr[], int x, int n) {
  sort(arr,arr + n);
  for(int i = 0; i < n-2; i++) {
    int l = i+1, r = n-1;
  /* 2SUM problem -> ponteiro que aponta para o primeiro e ultimo da
     sequencia e caso a soma for menor do que x adianta em uma casa o
     ponteiro da esquerda caso seja maior diminui em uma casa o ponteiro da
     direita */
    while(l < r) {
      if(arr[i] + arr[l] + arr[r] == x) {
        return true;
      } else if(arr[i] + arr[l] + arr[r] < x)
        l++;
      else
        r--;
    }
  }
  return false;
}
```

9.4. INFIX TO PREFIX

```
int main() {
map<char,int> prec;
stack<char> op;

string postfix;
string infix;
cin >> infix;

prec['+'] = prec['-'] = 1;
prec['*'] = prec['/'] = 2;
prec['^'] = 3;
for(int i = 0; i < infix.length(); i++) {
  char x = infix[i];
  if('0' <= x && x <= '9') {
    for(i;i < infix.length() && ('0' <= infix[i] && infix[i] <= '9');i++)
      postfix += infix[i];
    i--;
  } else if(('a' <= x && x <= 'z') || ('A' <= x && x <= 'Z')) {
    postfix += x;
  } else if (x == '(')
    op.push('(');
  else if(x == ')') {
    while(!op.empty() && op.top() != '(') {
      postfix += op.top();
      op.pop();
    }
    op.pop();
  } else {
    while(!op.empty() && prec[op.top()] >= prec[x]) {
    postfix += op.top();
    op.pop();
  }
  op.push(x);
  }
}
while(!op.empty()) {
postfix += op.top();
op.pop();
}
cout << postfix << endl;
}
```

9.5. KADANE (MAIOR SOMA NUM VETOR) - $O(n)$

```
int kadane(int arr[], int l) {

  int soma, total;
  soma = total = arr[0];

  for(int i = 1; i < l; i++) {
    soma = max(arr[i], arr[i] +soma);
    if(soma > total)
      total = soma;
  }
  return total;

}
```

9.6. KADANE 2D - $O(n^3)$

```

// Program to find maximum sum subarray in a given 2D array
#include <stdio.h>
#include <string.h>
#include <limits.h>
int mat[1001][1001]
int ROW = 1000, COL = 1000;

// Implementation of Kadane's algorithm for 1D array. The function
// returns the maximum sum and stores starting and ending indexes of the
// maximum sum subarray at addresses pointed by start and finish pointers
// respectively.
int kadane(int* arr, int* start, int* finish, int n) {
    // initialize sum, maxSum and
    int sum = 0, maxSum = INT_MIN, i;

    // Just some initial value to check for all negative values case
    *finish = -1;

    // local variable
    int local_start = 0;

    for (i = 0; i < n; ++i) {
        sum += arr[i];
        if (sum < 0) {
            sum = 0;
            local_start = i+1;
        }
        else if (sum > maxSum){
            maxSum = sum;
            *start = local_start;
            *finish = i;
        }
    }

     // There is at-least one non-negative number
    if (*finish != -1)
        return maxSum;

    // Special Case: When all numbers in arr[] are negative
    maxSum = arr[0];
```

```
    *start = *finish = 0;

    // Find the maximum element in array
    for (i = 1; i < n; i++) {
        if (arr[i] > maxSum) {
            maxSum = arr[i];
            *start = *finish = i;
        }
    }
    return maxSum;
}

// The main function that finds maximum sum rectangle in mat[][]
int findMaxSum() {
    // Variables to store the final output
    int maxSum = INT_MIN, finalLeft, finalRight, finalTop, finalBottom;

    int left, right, i;
    int temp[ROW], sum, start, finish;

    // Set the left column
    for (left = 0; left < COL; ++left) {
        // Initialize all elements of temp as 0
        for(int i = 0; i < ROW; i++)
            temp[i] = 0;

        // Set the right column for the left column set by outer loop
        for (right = left; right < COL; ++right) {
           // Calculate sum between current left and right for every row 'i'
            for (i = 0; i < ROW; ++i)
                temp[i] += mat[i][right];

            // Find the maximum sum subarray in temp[]. The kadane()
            // function also sets values of start and finish.  So 'sum' is
            // sum of rectangle between (start, left) and (finish, right)
            //  which is the maximum sum with boundary columns strictly as
            //  left and right.
            sum = kadane(temp, &start, &finish, ROW);

            // Compare sum with maximum sum so far. If sum is more, then
            // update maxSum and other output values
            if (sum > maxSum) {
                maxSum = sum;
                finalLeft = left;
                finalRight = right;
                finalTop = start;
                finalBottom = finish;
            }
        }
    }

    return maxSum;
    // Print final values
    printf("(Top, Left) (%d, %d)\n", finalTop, finalLeft);
    printf("(Bottom, Right) (%d, %d)\n", finalBottom, finalRight);
    printf("Max sum is: %d\n", maxSum);
}
```

9.7. KADANE (SEGMENT TREE) - (QUERY IN RANGE $O(logn)$)

```
struct node {
  int pref, suf, tot, best;
  node () {}
  node(int pref, int suf, int tot, int best) : pref(pref), suf(suf),
    tot(tot), best(best) {}
};

node tree[500000];
int arr[100000];

node query(int l, int r, int i, int j, int pos) {

  node x;

  if(l > r || l > j || r < i) {
  return node(-INF, -INF, -INF, -INF);
  }

  if(i <= l && r <= j) {
  return node(tree[pos].pref, tree[pos].suf, tree[pos].tot, tree[pos].best);
  }

  int mid = (l + r)/2;
  node left = query(l,mid,i,j,2*pos+1), right = query(mid+1,r,i,j,2*pos+2);
  x.pref = max({left.pref, left.tot, left.tot + right.pref});
  x.suf = max({right.suf, right.tot, right.tot + left.suf});
  x.tot = left.tot + right.tot;
  x.best = max({left.best,right.best, left.suf + right.pref});
  // imprimir.best
  return x;
}

void build(int l, int r, int pos) {

  if(l == r) {
  tree[pos] = node(arr[l], arr[l], arr[l], arr[l]);
  return;
  }

  int mid = (l + r)/2;
  build(l,mid,2*pos+1); build(mid+1,r,2*pos+2);
  l = 2*pos+1, r = 2*pos+2;
  tree[pos].pref = max({tree[l].pref, tree[l].tot, tree[l].tot +
    tree[r].pref});
  tree[pos].suf = max({tree[r].suf, tree[r].tot, tree[r].tot + tree[l].suf});
  tree[pos].tot = tree[l].tot + tree[r].tot;
  tree[pos].best = max({tree[l].best,tree[r].best, tree[l].suf +
    tree[r].pref});

}
```

9.8. COMPRESSAO DE PONTOS

```
map<int, int> rev;

vector<int> compress(vector<int> &arr) {

  for(int x : arr) {
    s1.insert(x);

  vector<int> aux;
  for(int x : s1) aux.pb(x);
  for(int i=0; i<n; i++){
    int id = lower_bound(aux.begin(), aux.end(), arr[i]) - aux.begin();
    // rev[id] = arr[i];
    arr[i] = id;
  }
```

```
}
```

9.9. TORRE DE HANOI - $O(2^{n-1})$

```
#include <stdio.h>

// C recursive function to solve tower of hanoi puzzle
void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod) {
  if (n == 1) {
    printf("\n Move disk 1 from rod %c to rod %c", from_rod, to_rod);
    return;
  }
  towerOfHanoi(n-1, from_rod, aux_rod, to_rod);
  printf("\n Move disk %d from rod %c to rod %c", n, from_rod, to_rod);
  towerOfHanoi(n-1, aux_rod, to_rod, from_rod);
}

int main() {
  int n = 4; // Number of disks
  towerOfHanoi(n, 'A', 'C', 'B'); // A, B and C are names of rods
  return 0;
}
```

9.10. FIBONACCI MATRIX EXPONENTIATION - $O(logn)$

```
int fib (int n) {
  long long fib[2][2]= {{1,1},{1,0}};
  int ret[2][2]= {{1,0},{0,1}};
  int tmp[2][2]= {{0,0},{0,0}};
  int i,j,k;
  while(n) {
    if(n&1) {
      memset(tmp,0,sizeof tmp);
      for(i=0; i<2; i++)
        for(j=0; j<2; j++)
          for(k=0; k<2; k++)
            tmp[i][j]=(tmp[i][j]+ret[i][k]*fib[k][j]);
      for(i=0; i<2; i++)
        for(j=0; j<2; j++)
          ret[i][j]=tmp[i][j];
    }
    memset(tmp,0,sizeof tmp);
    for(i=0; i<2; i++)
      for(j=0; j<2; j++)
        for(k=0; k<2; k++)
          tmp[i][j]=(tmp[i][j]+fib[i][k]*fib[k][j]);
    for(i=0; i<2; i++)
      for(j=0; j<2; j++)
        fib[i][j]=tmp[i][j];
    n/=2;
  }
  return (ret[0][1]);
}
```

9.11. 2-SAT PROBLEM - $O(V+E)$

```
int comp[312345];
vector<int> adj[312345];
// grafo reverso de adj
vector<int> trans[312345];
vector<int> scc[312345];
int V;
// Nao esquecer de preencher o V com o tamanho dos vertices

void dfsTrans(int u, int id, int comp[]) {
  comp[u] = id;
  scc[id].push_back(u);

  for (int v: trans[u])
    if (!comp[v])
      dfsTrans(v, id, comp);
}

void getTranspose() {
  for (int u = 0ll; u < V; u++)
    for(int v: adj[u])
      trans[v].push_back(u);
}

void dfsFillOrder(int u, int comp[], stack<int> &Stack) {
  comp[u] = true;

  for(int v: adj[u])
    if(!comp[v])
      dfsFillOrder(v, comp, Stack);

  Stack.push(u);
}

// The main function that finds and prints all strongly connected
// components
void computeSCC() {

  stack<int> Stack;
  // Fill vertices in stack according to their finishing times
  for(int i = 0ll; i < V; i++)
    if(comp[i] == false)
      dfsFillOrder(i, comp, Stack);

  // Create a reversed graph
  getTranspose();

  memset(comp, 0ll, sizeof(comp));

  // Now process all vertices in order defined by Stack
  int id = 1ll;
  while(Stack.empty() == false) {
    int v = Stack.top();
    Stack.pop();

    if(comp[v] == false)
      dfsTrans(v, id++, comp);
  }
}

//  (X v Y)  = (X -> ~Y) and (~X -> Y)
void orEdge(int u, int v, int idxu, int idxv, int n) {
  // idx represents the index of the atoms
  // Example there are atoms X, ~X, Y, ~Y
  // Then for Clause (X v ~Y) idxu = 0ll and idxv = 3ll
  idxv ^= 1ll;
  adj[u + idxu*n].pb(v + idxv*n);
  idxu ^= 1ll, idxv ^= 1ll;
  adj[v + idxv*n].pb(u + idxu*n);
}

```

```
// (X xor Y) = (X v Y) and (~X v ~Y)
// for this function the result is always 0 1 or 1 0
void xorEdge(int u, int v, int idxu, int idxv, int n) {
  orEdge(u, v, idxu, idxv, n);
  orEdge(u, v, idxu^1ll, idxv^1ll, n);
}

bool check(int n) {

  for(int i = 0ll; i < V; i += 2ll*n) {
    for(int j = i, k = 0ll; k < n; k++, j++) {
      if(comp[j] == comp[j+n] && comp[j] != 0) {
        return false;
      }
    }
  }
  return true;
}


signed main () {

  computeSCC();

  if(check(n)) {
    cout << "YES" << endl;
  } else
    cout << "NO" << endl;
}
```