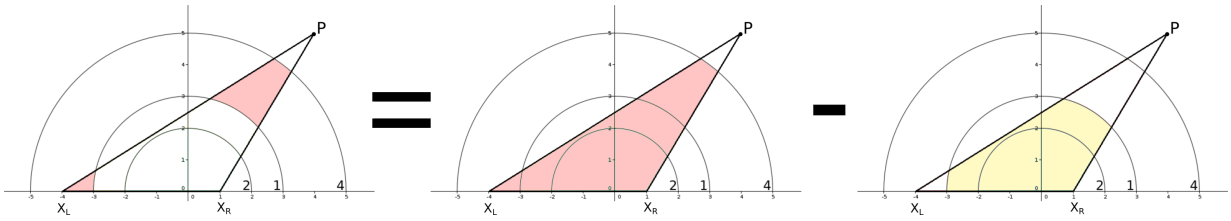
PPCI 2024 - Maratona de Programação

19 de outubro de 2024

A: Arremesso de Triângulo

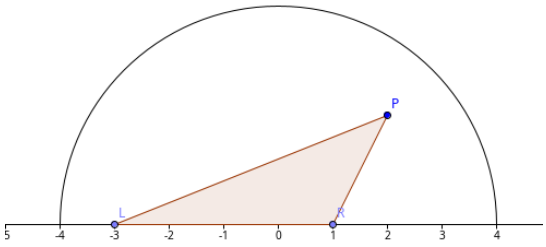
Seja S_i a área de intersecção do triângulo com a região de pontuação da i-ésima semicircunferência (de forma que a resposta é $\sum_{i=1}^{N} v_i \times S_i$).

Primeiramente, note que S_i é igual à área de intersecção do triângulo com o $i-$ésimo semicírculo, subtraído da área de intersecção do triângulo com o $(i-1)-$ésimo semicírculo:

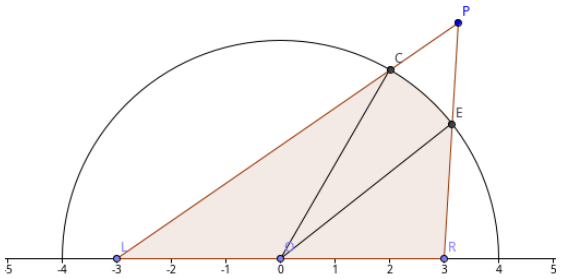


Desta forma, o problema é reduzido a determinar a área A de intersecção de um triângulo com um semicírculo.

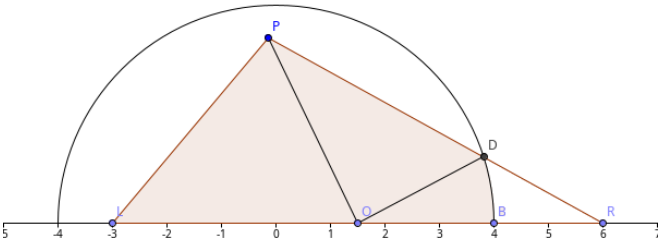
Tal área pode ser dada pelas áreas de alguns triângulos e de alguns setores circulares, de acordo com os seguintes casos (sejam $L = (X_L, 0)$, $R = (X_R, 0)$ e $O = (0, 0)$):



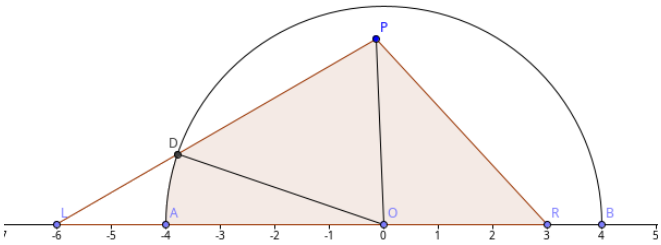
Caso 1: P, L e R dentro do semicírculo.
$A = \Delta LRP$



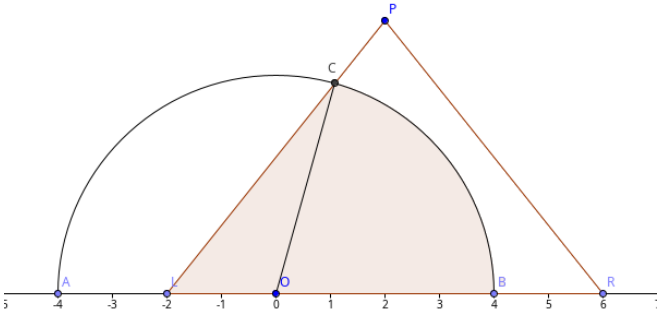
Caso 2: P fora; L e R dentro do semicírculo.
$A = \Delta LCO + \measuredangle OCE + \Delta ERO$



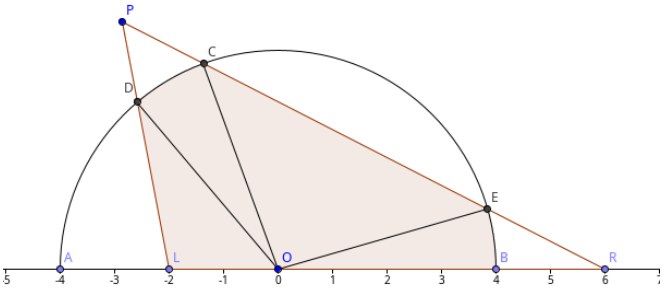
Caso 3: R fora; L e P dentro do semicírculo.
$A = \Delta LPO + \Delta PDO + \measuredangle ODB$



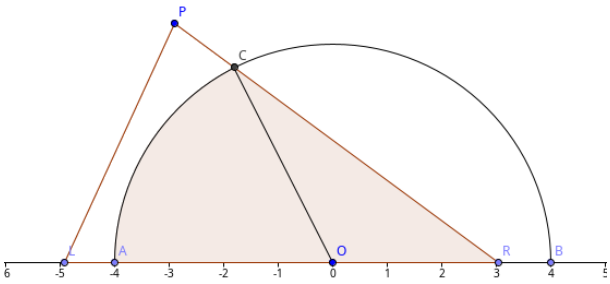
Caso 4: L fora; R e P dentro do semicírculo.
$A = \measuredangle OAD + \Delta PDO + \Delta PRO$



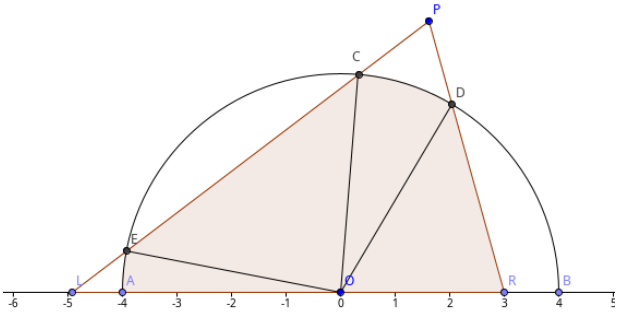
Caso 5.1: R e P fora; L dentro do semicírculo; $\overline{PR}$ não intersecta a semicircunferência.
$A = \Delta CLO + \measuredangle OCB$



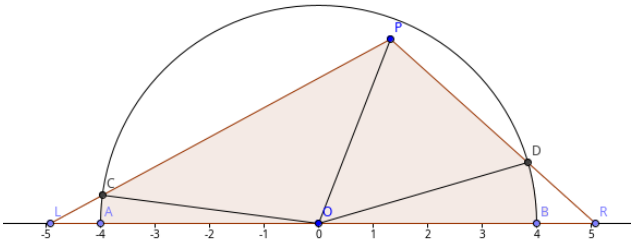
Caso 5.2: R e P fora; L dentro do semicírculo; $\overline{PR}$ intersecta a semicircunferência.
$A = \Delta DLO + \measuredangle ODC + \Delta CEO + \measuredangle OEB$



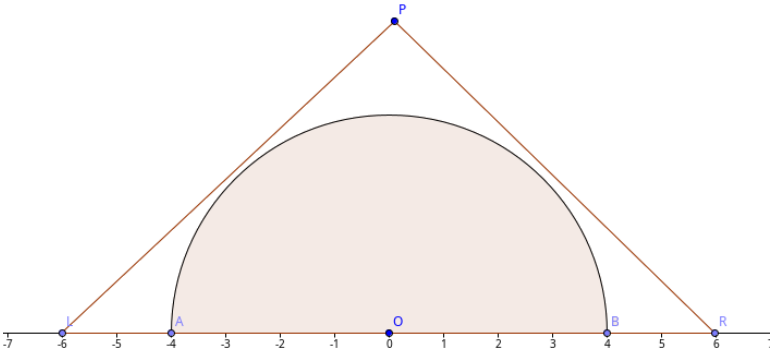
Caso 6.1: L e P fora; R dentro do semicírculo; $\overline{PL}$ não intersecta a semicircunferência.
$A = \measuredangle OAC + \Delta CRO$



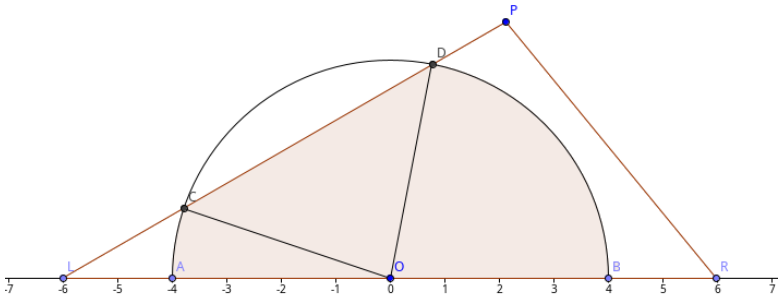
Caso 6.2: L e P fora; R dentro do semicírculo; $\overline{PL}$ intersecta a semicircunferência.
$A = \measuredangle OEA + \Delta ECO + \measuredangle OCD + \Delta ODR$



Caso 7: L e R fora; P dentro do semicírculo.
$A = \measuredangle OAC + \Delta PCO + \Delta PDO + \measuredangle ODB$

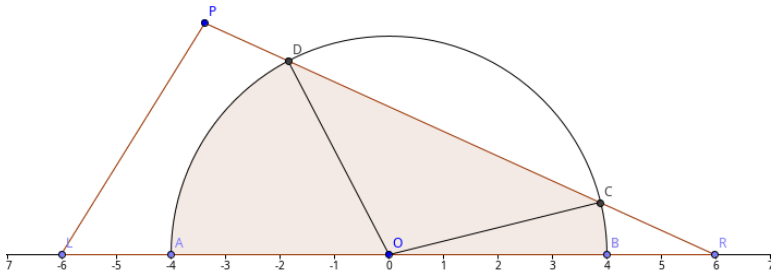


Caso 8.1: P, L e R fora do semicírculo; $\overline{PL}$ e $\overline{PR}$ não intersectam a semicircunferência.
$$A = \measuredangle OAB = \frac{\pi r^2}{2}$$



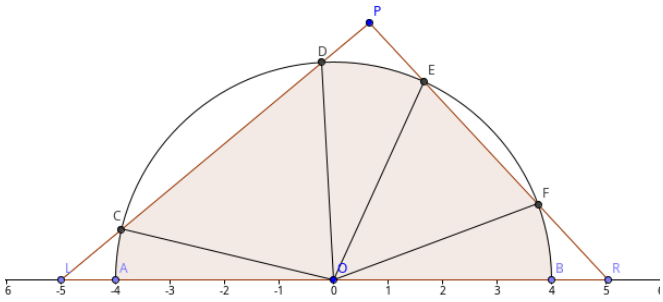
Caso 8.2: P, L e R fora do semicírculo; $\overline{PL}$ intersecta; $\overline{PR}$ não intersecta a semicircunferência.

$$A = \measuredangle OAC + \Delta CDO + \measuredangle ODB$$



Caso 8.3: P, L e R fora do semicírculo; $\overline{PR}$ intersecta; $\overline{PL}$ não intersecta a semicircunferência.

$$A = \measuredangle OAD + \Delta CDO + \measuredangle OCB$$



Caso 8.4: P, L e R fora do semicírculo; $\overline{PL}$ e $\overline{PR}$ intersectam a semicircunferência.

$$A = \measuredangle OAC + \Delta CDO + \measuredangle ODE + \Delta EFO + \measuredangle OFB$$

Note que, para determinar os triângulos e setores circulares, se faz necessário conhecer diversas operações de geometria, como: intersecção de reta com círculo; área de triângulo dados três vértices; ângulo entre dois segmentos de reta; área de setor circular; etc.

Recomenda-se fortemente ter uma biblioteca de Geometria 2D em suas anotações contendo uma grande coleção de operações. Suas implementações ficam como exercício para o leitor.

Complexidade total: $O(N)$.

B: Bloons

Para resolver o problema, pode-se utilizar programação dinâmica, considerando cada balão de forma independente. Assim, para cada balão, queremos encontrar a quantidade mínima de dardos para destruí-lo, usando os dados disponíveis. Vamos definir:

$dp[x]$: o número mínimo de dardos necessários para destruir um balão com resistência x.

$$dp[0] = 0,$$

já que para um balão com 0 camadas de resistência, não precisamos de nenhum dardo.

Para valores maiores de resistência, definimos $dp[x] = \infty$, representando que ainda não sabemos como destruir aquele balão.

O estado de transição será: Para cada camada x e cada macaco distinto com dano d_j, se for possível usar o macaco ($x \geq d_j$), então podemos atualizar o número mínimo de dardos:

$$dp[x] = \min(dp[x], dp[x - d_j] + 1)$$

Essa transição verifica se podemos destruir o balão com resistência x usando um dardo que destrói d_j camadas e o número mínimo de dardos necessários para destruir um balão de resistência $x - d_j$.

Assim, para cada dardo d_j disponível, atualizamos o array dp para cada resistência de balão possível $dp[x]$. Se, ao final do processo, $dp[h_i] = \infty$, isso significa que não é possível destruir o balão de resistência h_i com os dardos disponíveis, então retornamos -1. Caso contrário, somamos o total de dardos mínimos necessários para destruir todos os balões.

Código

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

int main() {
    ll N, K;
    ll MaximoDeCamadas = 1000;

    cin >> N >> K;

    vector<ll> baloes(N);
    vector<ll> macacos(K);

    for (ll i = 0; i < N; ++i) {
        cin >> baloes[i];
    }

    for (ll i = 0; i < K; ++i) {
        cin >> macacos[i];
    }

    vector<ll> dp(MaximoDeCamadas + 1, INF);
    dp[0] = 0;

    for (ll h = 1; h <= MaximoDeCamadas; ++h) {
        for (ll d : macacos) {
            if (h >= d && dp[h - d] != INF) {
                dp[h] = min(dp[h], dp[h - d] + 1);
            }
        }
    }

    ll total_macacos = 0;
    for (ll h : baloes) {
        if (dp[h] == INF) {
            cout << -1 << endl;
            return 0;
        }
        total_macacos += dp[h];
    }

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

Complexidade O(N+K)

D: Double Casting

A ideia principal deste problema é encontrar o número de duplicatas dentro de um intervalo de magias em que a compra seja possível, para isto vamos utilizar "two pointers" (dois ponteiros) no qual um marca o começo do intervalo e outro marca o final do intervalo, vamos chamá-los P_l e P_r.

P_l e P_r iniciam no i_0, iremos avançar P_r até i_{n-1}. Caso o intervalo se torne maior do que K iremos diminuir o intervalo avançando P_l. Devemos atualizar o número de duplicatas ao fazer qualquer avanço em P_r ou P_l.

Podemos utilizar da estrutura map para guardar a quantidade de vezes que um número apareceu dentro de nosso intervalo e assim determinar o número de pares em cada avanço.

double-casting.cpp

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

int a[200005];
int main(){
    int n, k;
    cin >> n >> k;
    map<int,int> num;
    for(int i=0; i<n; i++){
        cin >> a[i];
    }
    int r=0, l=0, soma_temp=0, pares=0, resp=0;
    // O(r+l) = O(n)
    while(r<n){
        soma_temp += a[r];
        num[a[r]]++;
        if(num[a[r]]%2==0)  pares++;
        r++;
        while(soma_temp>k){
            soma_temp -= a[l];
            num[a[l]]--;
            if(num[a[l]]%2==1)  pares--;
            l++;
        }
        resp = max(resp, pares);
    }
    cout << resp << '\n';

    return 0;
}
```

E: Estresse

É necessário utilizar um algoritmo para busca de um padrão em um texto. Um destes algoritmos é o KMP (Knuth-Morris-Pratt), que tem complexidade de pior caso $\mathcal{O}(|S| + |P|)$.

O algoritmo deve ser executado duas vezes sobre S: uma vez procurado por P, e outra vez procurando pelo reverso de P, aqui denotado por P'. Então, basta contar a quantidade de vezes que P e P' pode ser visto, e imprimir o resultado.

Um *corner case*[1] é se o padrão P procurado é um palíndromo (ou seja, se $P = P'$). A exemplo, se é buscado o palíndromo $P =$ "reviver", no texto $S =$ "xxreviverxx", em um mesmo momento (a partir do terceiro caractere de S) a papagaia poderia ter falado tanto P quanto P'. No entanto, note que se trata do **mesmo** momento, e não de dois momentos distintos contabilizados separadamente.

Dessa forma, caso P seja palíndromo, deve-se fazer apenas uma execução do KMP, ou dividir o resultado final por 2. Esse caso é retratado no Exemplo 3, onde $S =$ "*arararara*" e $P =$ "*arara*", cuja saída deve ser 3.

estresse.cpp

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

using namespace std;

int lsp[1000001]; // lsp[i] = Longest Suffix Preffix de p[0..i], ou Maior Sufixo Que É
    Prefixo Também

int kmp(string &s, string &p) {
    int saida = 0;

    // Pré-processamento
    lsp[0] = 0;
    for (int i = 1, j = 0; i<p.size();) {
        if (p[i] == p[j]) lsp[i++] = ++j;
        else if (j) j = lsp[--j];
        else lsp[i++] = 0;
    }

    // Busca pelas ocorrências do padrão
    for (int i = 0, j = 0; i<s.size();) {
        if (s[i] == p[j]) j++, i++;
        else if (j) j = lsp[--j];
        else i++;

        if (j == p.size()) {
            ++saida;
            j = lsp[--j];
        }
    }

    return saida;
}
```

[1] Caso especial onde a regra geral não se aplica; caso limite.

```
char saux[1000001], paux[1000001];

int main() {
    scanf("%s %s", saux, paux);

    string s(saux), p(paux);

    int saida;

    saida = kmp(s, p);

    string p_rev = p;
    reverse(p_rev.begin(), p_rev.end());

    if (p != p_rev)
        saida += kmp(s, p_rev);

    printf("%d\n", saida);

    return 0;
}
```

I: Ímpar e Impares

O primeiro passo da resolução é lembrar que a soma entre um número ímpar com um número par resultará em um número ímpar. O próximo passo é sempre utilizar o maior valor de ímpar possível para realizar a operação e perceper que a soma se tornará o maior ímpar do vetor.

O menor número de operações será pelo menos igual ao número de pares, portanto podemos realizar as operações nos números pares em ordem crescente e com isso garantir que o valor ímpar da operação é o maior possível.

Caso o ímpar da operação seja menor do que o par podemos substiuir o par fazendo 2 operações com estes números. Ex: 4 e 3 ficariam 11 e 7 realizando 2 operações.

Porém nesses casos podemos aproveitar e aumentar o valor do ímpar de modo que todas as próximas operações ocorram 1 única vez ao realizar a operação entre maior ímpar atual e o maior número par do vetor.

Logo podemos concluir que o número de operações será o número de pares + 1 (apenas se algum par requerir 2 operações).

impar-e-impares.cpp

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

int main(){
    int n;
    cin >> n;
    vector<int> a(n);
    int num_impar=0, maior_impar=0, maior_par=0;
    for(int i=0; i<n; ++i){
        cin >> a[i];
        if(a[i]%2==1){
            num_impar++;
            maior_impar = max(maior_impar, a[i]);
        }
        else{
            maior_par = max(maior_par, a[i]);
        }
    }
    sort(a.begin(), a.end());
    if(num_impar==0)    cout << "-1\n";
    else{
        int resp = 0;
        for(int i=0; i<n; ++i){
            if(a[i]%2==0 and a[i]<maior_impar){
                resp++;
                maior_impar += a[i];
            }
            if(a[i]%2==0 and a[i]>maior_impar){
                resp+=2;
                maior_impar += maior_par;
            }
        }
        cout << resp << '\n';
    }
```

```
return 0;
}
```

K: Kleber e a Convenção

Dado alguma coleção precisamos realizar o xor entre todos os números presentes nela, vamos denominar o número resultante "x".

Em relação aos números do estande é importante escolhermos guarda-los em uma estrutura que possibilite busca logaritmica devido a complexidade, algumas opções seriam set ou map.

Para determinar o número de insígnias vamos busca o complemento binário de **x** dentre os números do estande, a cada doação seja recebendo ou realiando iremos realizar o $x \oplus e_i$ e posteriormente fazer a busca pelo complemento binário de **x**.

Complexidade esperada: $\mathcal{O}(N \cdot \log n \cdot \log x)$. **x** é o xor resultante.

kleber-e-a-convencao.cpp

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


int main(){
    int n, e, q;
    cin >> n >> e;
    map<int,int> m;
    int x=0, y;
    for(int i=0; i<n; i++){
        cin >> y;
        x ^= y;
    }
    for(int i=0; i<e; i++){
        cin >> y;
        m[y]++;
    }
    int resp=0;
    int j=0, xx;
    while(1<<j<=x) j++;
    xx = ((1<<j)-1)^x;
    if(m[xx]) resp++;
    cin >> q;
    // O(n*(logn+log(ai))) == O(n*2logn)
    for(int i=0; i<q; i++){
        char c;
        int y, xx, j=0;
        cin >> c >> y;
        x ^= y;
        while(1<<j<=x) j++;
        xx = ((1<<j)-1)^x;
        if(m[xx]) resp++;
    }
    cout << resp << '\n';

    return 0;
}
```