

Paradigmas de Resolução de Problemas

Algoritmos Gulosos: *Two pointers*

Prof. Edson Alves – UnB/FGA

1. *Two pointers*
2. Exemplos de aplicação de *two pointers*

Two pointers

Definição

- *Two pointers* é uma técnica gulosa aplicada em problemas em vetores
- Nela são utilizados dois ponteiros unidirecionais L e R
- Assim, a cada iteração do algoritmo, estes ponteiros só podem avançar na direção pré-definida, sem recuar
- Isto faz com que cada ponteiro observe cada elemento do vetor uma única vez
- Para tal, é preciso considerar quais valores ainda podem fazer parte da solução, e quais podem, e devem, ser descartados
- Desde modo, esta é uma técnica gulosa, uma vez que, fixado um dos ponteiros, o segundo se move o máximo possível, e em seguida os ponteiros são reposicionados para as melhores posições possíveis

Implementação

- Em geral, o ponteiro L (*left*) aponta para o início do vetor e é incrementado a cada passo do algoritmo
- O ponteiro R (*right*) geralmente parte de L (ou $L + 1$) e avança enquanto o intervalo $[L, R)$ constituir uma subsolução válida do problema
- Em alguns problemas, o ponteiro L pode saltar diretamente para R , caso R não possa mais avançar
- Também há problemas onde R inicia no último elemento do vetor, e caminha em direção ao início do mesmo
- Na maioria dos casos, o uso desta técnica leva a algoritmos $O(N)$, onde N é o tamanho do vetor

Exemplos de aplicação de *two pointers*

Maior substring em ordem lexicográfica

- Seja S uma string com N caracteres
- O problema consiste em determinar o tamanho M da maior substring b de S tal que os caracteres de b estão em ordem lexicográfica
- Em outras palavras, o problema é determinar a maior substring $b = b[1..M]$ de S tal que $b_{i-1} \leq b_i, \forall i \in [2, M]$
- A string S tem $O(N^2)$ substrings, e a verificação se uma substring está em ordem lexicográfica ou não é feita em $O(N)$
- Assim um algoritmo de busca completa tem complexidade $O(N^3)$

Maior substring em ordem lexicográfica

- Porém, é possível utilizar a técnica *two pointers* neste problema
- Inicie L no primeiro caractere de S
- Para cada valor de L , faça $R = L + 1$
- A substring $S[L..(R - 1)]$ tem, inicialmente, um único caractere, de modo que está em ordem lexicográfica
- Enquanto $R \leq N$ e $S[R - 1] \leq S[R]$, incremente R
- Ao final do processo, a substring $S[L..(R - 1)]$ terá $R - L$ caracteres e estará em ordem lexicográfica
- Atualize L para R e prossiga enquanto $L \leq N$
- Como tanto L quanto R observam cada caractere de S uma única vez, esta solução tem complexidade $O(N)$ e memória $O(1)$

Tamanho da maior substring ordenada

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 auto max_ordered_substring(const string& s)
6 {
7     auto N = s.size(), L = 0ul, ans = 0ul;
8
9     while (L < N) {
10         auto R = L + 1;
11
12         while (R < N and s[R - 1] <= s[R])
13             ++R;
14
15         ans = max(ans, R - L);
16         L = R;
17     }
18
19     return ans;
20 }
```

Tamanho da maior substring ordenada

```
22 int main()
23 {
24     string s1 { "abcde" }, s2 { "cba" }, s3 { "teste" };
25
26     cout << max_ordered_substring(s1) << '\n';
27     cout << max_ordered_substring(s2) << '\n';
28     cout << max_ordered_substring(s3) << '\n';
29
30     return 0;
31 }
```

Maior subvetor com, no máximo, K números ímpares

- Seja \vec{v} um vetor com N números inteiros
- O problema consiste em terminar o maior subvetor $\vec{x} = \vec{v}[i..j]$ de \vec{v} que contenha, no máximo, K números ímpares
- Novamente, \vec{v} em $O(N^2)$ subvetores, de modo que uma solução de busca completa que verifique cada um deles terá complexidade $O(N^3)$
- A ideia central da solução usando *two pointers* é identificar intervalos $[L, R)$ tais que os subvetores $\vec{v}[L..(R-1)]$ tenham, no máximo, K números ímpares
- Observe que $|\vec{v}[L..(R-1)]| = R - L$
- O ponteiro L observará, um a um, os elementos de \vec{v} , do primeiro para o último
- O ponteiro R iniciará apontando para o primeiro elemento

Maior subvetor com, no máximo, K números ímpares

- Um contador c , inicialmente igual a zero, manterá o registro do número de elementos ímpares dentre os elementos de \vec{v} cujos índices estão no intervalo $[L, R)$
- Para cada valor de L , o ponteiro R apontará para L ou manterá o seu valor, o que estiver mais distante do início do vetor
- Enquanto R apontar para um elemento par, ou apontar para um ímpar com o contador $c < K$, o contador é atualizado com o valor de $\vec{v}[R]$ e R é incrementado
- Ao final deste processo, a resposta é atualizada em relação ao tamanho do subvetor válido $(R - L)$
- Por fim, o contador é ajustado, caso o valor de $\vec{v}[L]$ tenha sido considerado, e L é incrementado
- A complexidade da solução é $O(N)$

Visualização do algoritmo do maior subvetor com, no máximo, $K = 1$ números ímpares

$ans = 0$

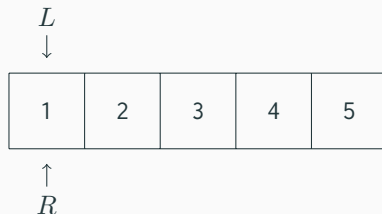
$c = 0$

1	2	3	4	5
---	---	---	---	---

Visualização do algoritmo do maior subvetor com, no máximo, $K = 1$ números ímpares

$ans = 0$

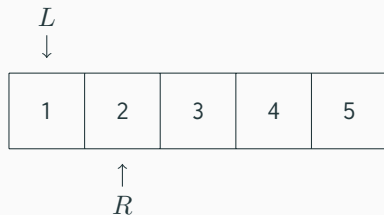
$c = 1$



Visualização do algoritmo do maior subvetor com, no máximo, $K = 1$ números ímpares

$ans = 0$

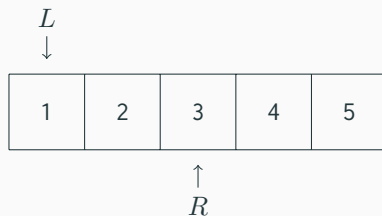
$c = 1$



Visualização do algoritmo do maior subvetor com, no máximo, $K = 1$ números ímpares

$$ans = 0$$

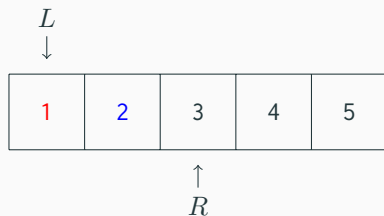
$$c = 1$$



Visualização do algoritmo do maior subvetor com, no máximo, $K = 1$ números ímpares

$ans = 2$

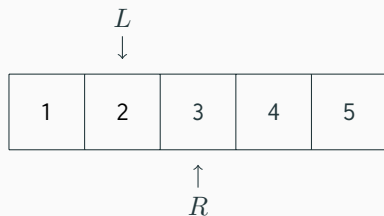
$c = 1$



Visualização do algoritmo do maior subvetor com, no máximo, $K = 1$ números ímpares

$$ans = 2$$

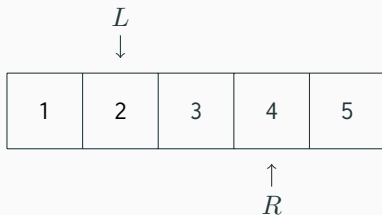
$$c = 0$$



Visualização do algoritmo do maior subvetor com, no máximo, $K = 1$ números ímpares

$ans = 2$

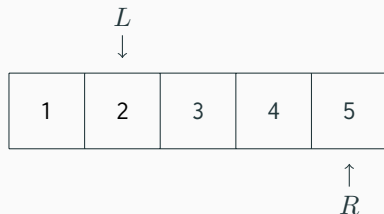
$c = 1$



Visualização do algoritmo do maior subvetor com, no máximo, $K = 1$ números ímpares

$$ans = 2$$

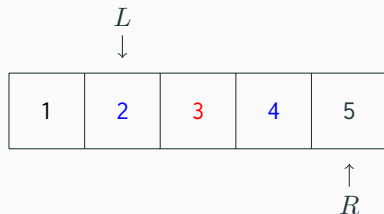
$$c = 1$$



Visualização do algoritmo do maior subvetor com, no máximo, $K = 1$ números ímpares

$ans = 3$

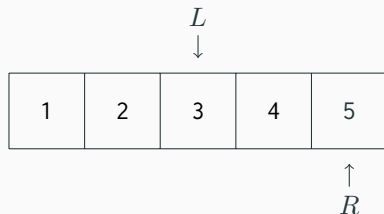
$c = 1$



Visualização do algoritmo do maior subvetor com, no máximo, $K = 1$ números ímpares

$$ans = 3$$

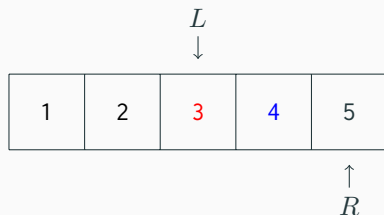
$$c = 1$$



Visualização do algoritmo do maior subvetor com, no máximo, $K = 1$ números ímpares

$ans = 3$

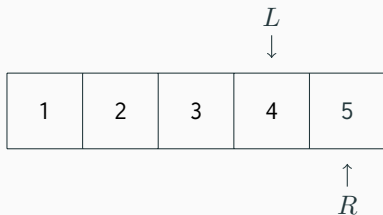
$c = 1$



Visualização do algoritmo do maior subvetor com, no máximo, $K = 1$ números ímpares

$ans = 3$

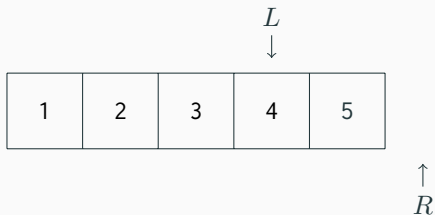
$c = 0$



Visualização do algoritmo do maior subvetor com, no máximo, $K = 1$ números ímpares

$ans = 3$

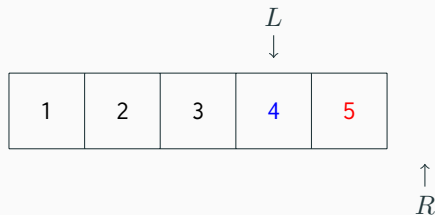
$c = 1$



Visualização do algoritmo do maior subvetor com, no máximo, $K = 1$ números ímpares

$$ans = 3$$

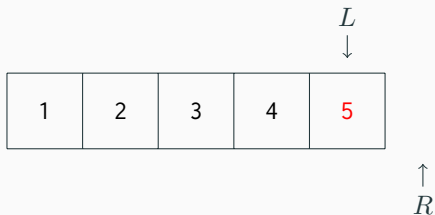
$$c = 1$$



Visualização do algoritmo do maior subvetor com, no máximo, $K = 1$ números ímpares

$$ans = 3$$

$$c = 1$$



Implementação do maior subvetor com, no máximo, K ímpares

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 size_t max_subarray(const vector<int>& xs, size_t K) {
6     auto N = xs.size(), L = 0ul, R = 0ul, odds = 0ul, ans = 0ul;
7
8     while (L < N) {
9         R = max(L, R);
10
11         while (R < N and (xs[R] % 2 == 0 or odds < K))
12             odds += (xs[R++] % 2);
13
14         ans = max(ans, R - L);
15         odds = max(odds - (xs[L] % 2), 0ul);
16         ++L;
17     }
18
19     return ans;
20 }
```

Implementação do maior subvetor com, no máximo, K ímpares

```
22 int main()
23 {
24     vector<int> xs { 1, 3, 5, 4, 5 }, ys { 1, 3, 5 };
25
26     cout << max_subarray(xs, 1) << '\n';
27     cout << max_subarray(xs, 2) << '\n';
28     cout << max_subarray(xs, 3) << '\n';
29     cout << max_subarray(ys, 0) << '\n';
30
31     return 0;
32 }
```

Menor elemento em um subvetor de tamanho K

- Quando o intervalo delimitador por $[L, R)$ tem um tamanho K fixo, a técnica dos dois ponteiros também é conhecida como *sliding window*
- Seja \vec{v} um vetor com N inteiros
- O problema de se determinar o menor elemento de cada um dos $N - K + 1$ subintervalos de tamanho K pode ser resolvido com dois ponteiros e duas pilhas P_{in} e P_{out}
- As pilhas armazenarão pares de valores (x, m) , onde x é o elemento do vetor a ser inserido, e m é o menor dentre os valores contidos na pilha e o próprio x

Menor elemento em um subvetor de tamanho K

- Observe que m será igual ao próprio x , caso a pilha esteja vazia
- Nos demais casos, $m = \min\{x, M\}$, onde M é o segundo elemento do par que está no topo da pilha
- Inicialmente, insira os primeiros K elementos de \vec{v} na pilha P_{in}
- O segundo elemento do par do topo de P_{in} será a resposta para o primeiro intervalo
- Faça $L = 0$ e $R = K$
- Caso a pilha P_{out} esteja vazia mova, um a um, os elementos de P_{in} para P_{out} , atualizando corretamente os valores de m

Menor elemento em um subvetor de tamanho K

- Exclua o elemento do topo de P_{out} : isto removerá o elemento $\vec{v}[L]$ das pilhas
- Em seguida, incremente L
- Agora, insira $\vec{v}[R]$ na pilha P_{in} e incremente R
- Após estes ajustes, as pilhas conterão os elementos do intervalo de tamanho K adjacente ao anterior (isto é, a janela se moveu de $[L, R)$ para $[L + 1, R + 1)$)
- A resposta para L será o menor entre os valores m dos topos das pilhas
- Como cada ponteiros observa cada elemento de \vec{v} no máximo uma vez, e cada elemento passa por, no máximo, duas pilhas, a complexidade do algoritmo é $O(N)$

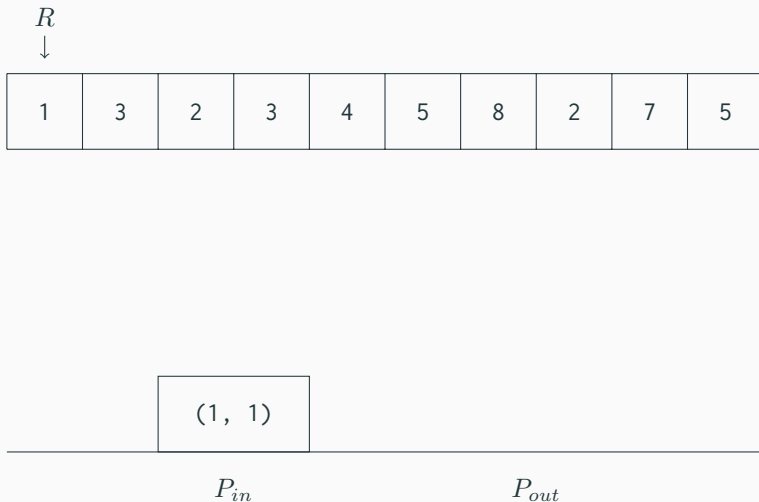
Visualização do menor elemento em cada um dos subvetores de tamanho $K = 3$

1	3	2	3	4	5	8	2	7	5
---	---	---	---	---	---	---	---	---	---

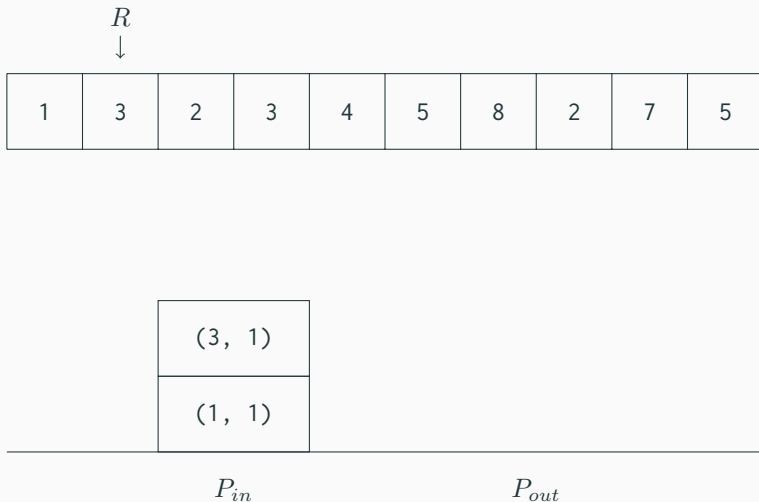
P_{in}

P_{out}

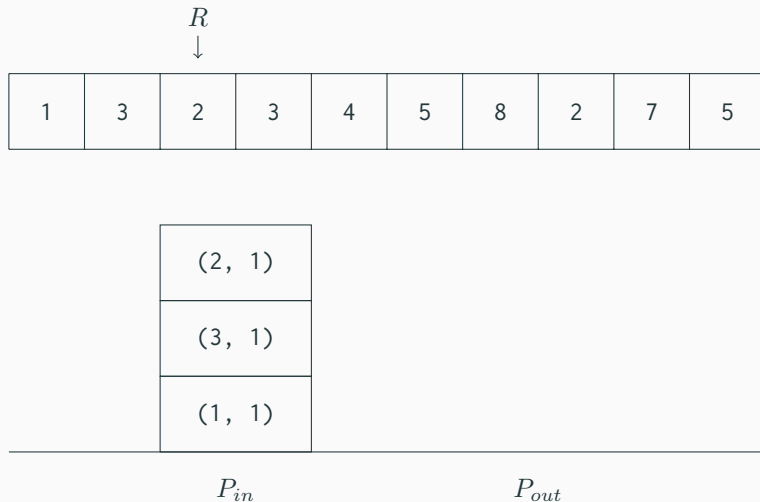
Visualização do menor elemento em cada um dos subvetores de tamanho $K = 3$



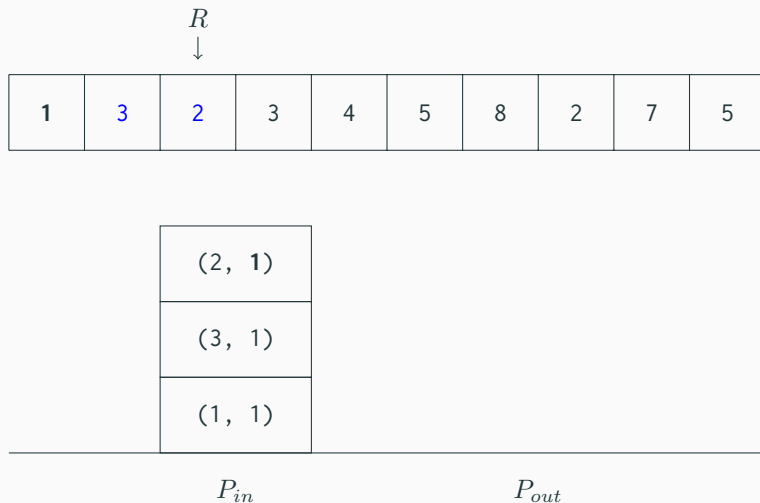
Visualização do menor elemento em cada um dos subvetores de tamanho $K = 3$



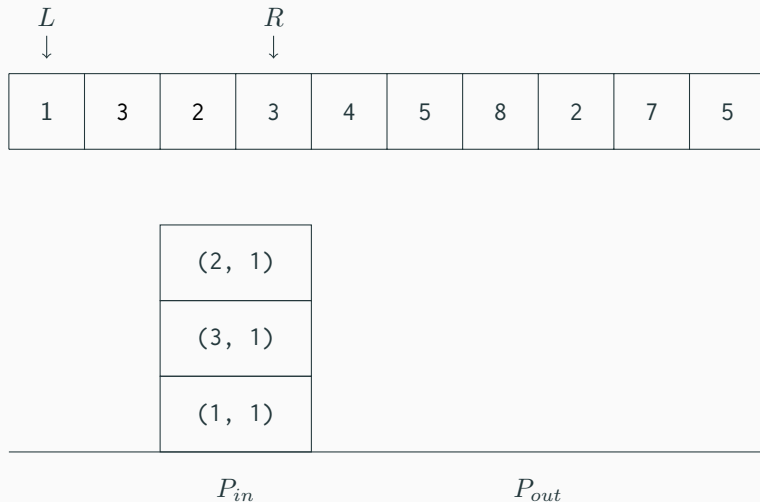
Visualização do menor elemento em cada um dos subvetores de tamanho $K = 3$



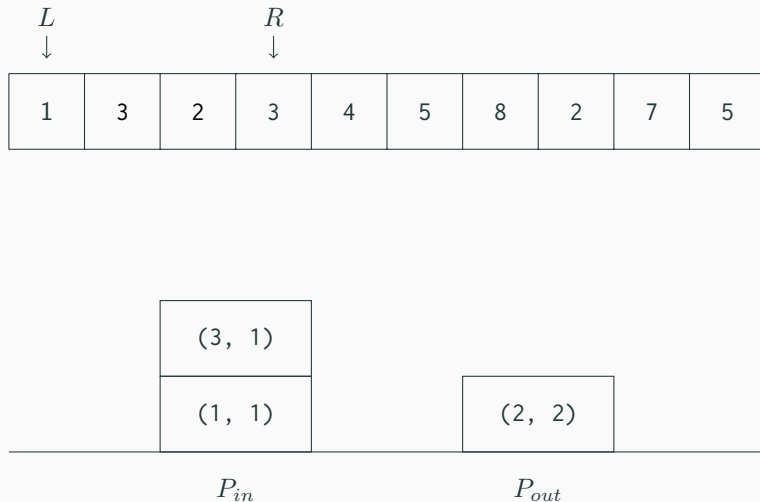
Visualização do menor elemento em cada um dos subvetores de tamanho $K = 3$



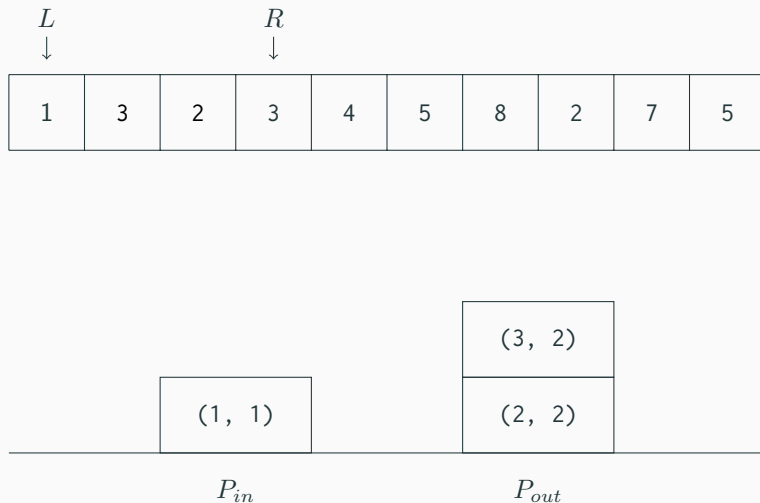
Visualização do menor elemento em cada um dos subvetores de tamanho $K = 3$



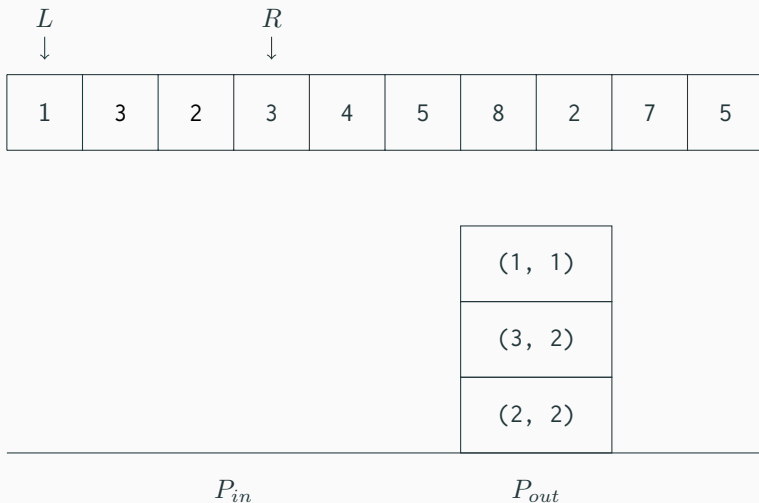
Visualização do menor elemento em cada um dos subvetores de tamanho $K = 3$



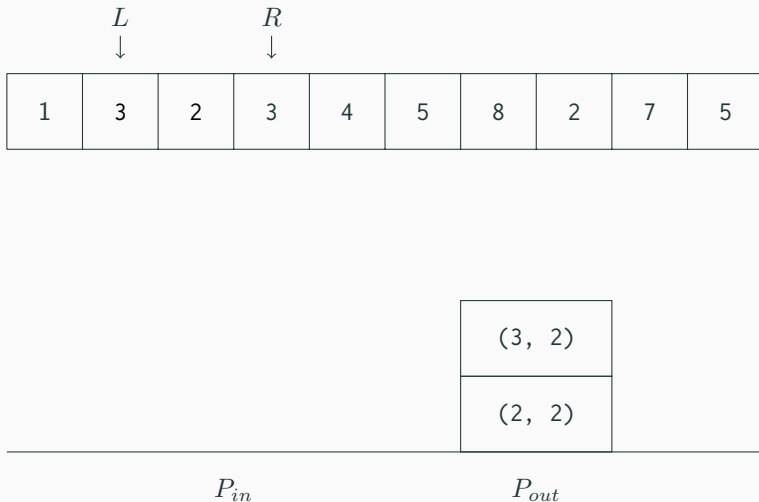
Visualização do menor elemento em cada um dos subvetores de tamanho $K = 3$



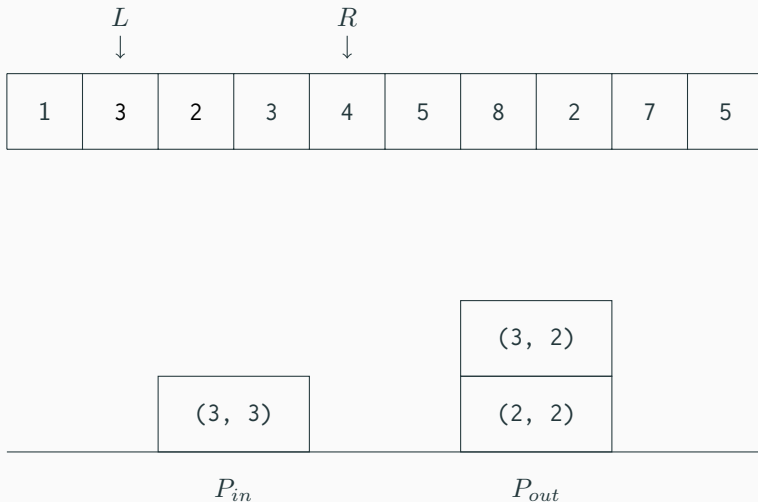
Visualização do menor elemento em cada um dos subvetores de tamanho $K = 3$



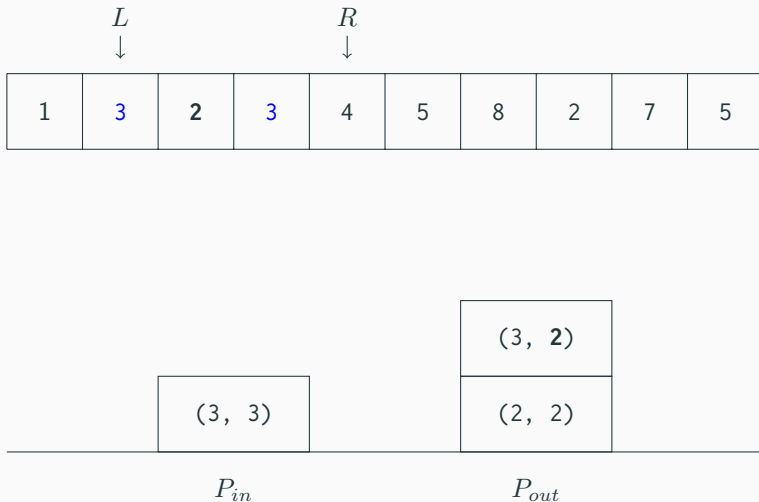
Visualização do menor elemento em cada um dos subvetores de tamanho $K = 3$



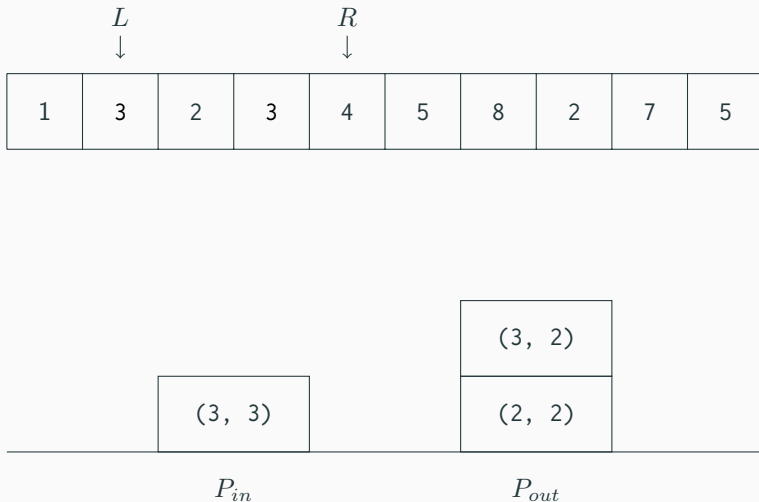
Visualização do menor elemento em cada um dos subvetores de tamanho $K = 3$



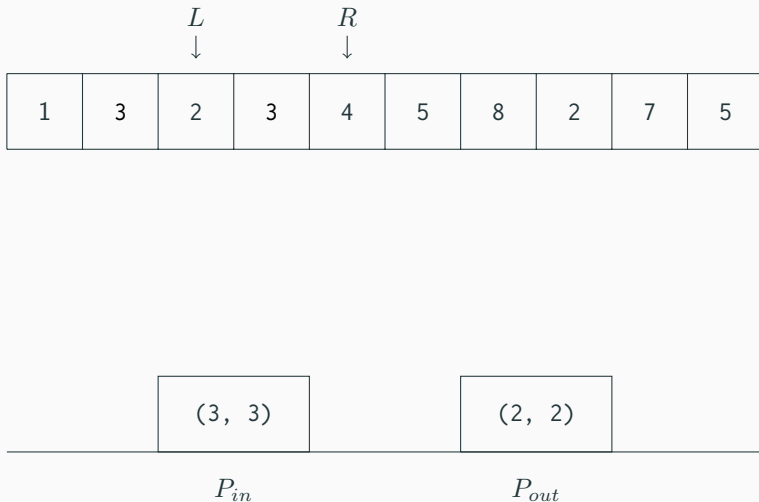
Visualização do menor elemento em cada um dos subvetores de tamanho $K = 3$



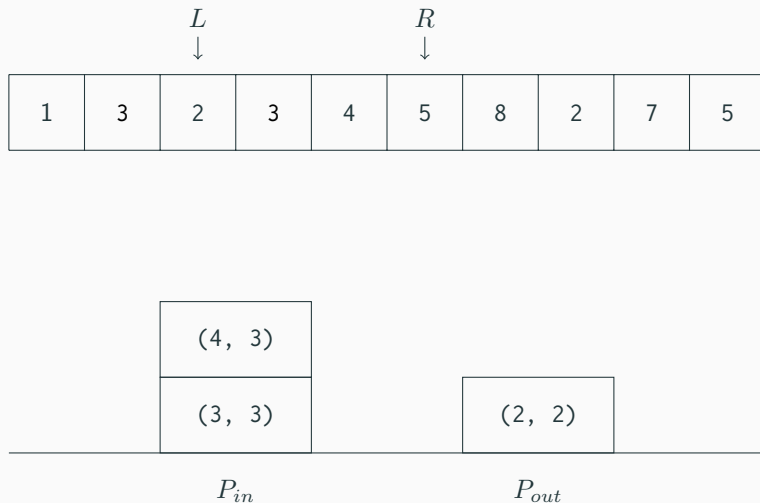
Visualização do menor elemento em cada um dos subvetores de tamanho $K = 3$



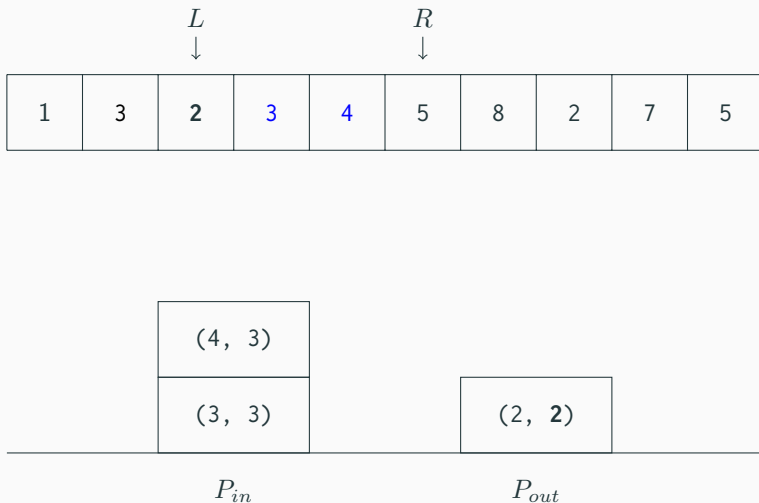
Visualização do menor elemento em cada um dos subvetores de tamanho $K = 3$



Visualização do menor elemento em cada um dos subvetores de tamanho $K = 3$



Visualização do menor elemento em cada um dos subvetores de tamanho $K = 3$



Implementação

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4 using ii = pair<int, int>;
5 const int oo { 1000000010 };
6
7 void insert(stack<ii>& s, int x)
8 {
9     int m = s.empty() ? x : min(s.top().second, x);
10    s.push(ii(x, m));
11 }
12
13 void move(stack<ii>& out, stack<ii>& in)
14 {
15     while (not in.empty()) {
16         auto x = in.top().first;
17         in.pop();
18         insert(out, x);
19     }
20 }
```

Implementação

```
22 vector<int> minimum(int N, int K, const vector<int>& xs)
23 {
24     stack<ii> in, out;
25
26     int L = 0, R;
27
28     for (R = 0; R < K; ++R)
29         insert(in, xs[R]);
30
31     move(out, in);
32
33     vector<int> ans(N - K + 1, -1);
34
35     ans[L] = out.top().second;
36
37     while (R < N)
38     {
39         if (out.empty())
40             move(out, in);
```

Implementação

```
42     insert(in, xs[R]);
43     out.pop();
44
45     ++L;
46     ++R;
47
48     auto a = in.empty() ? oo : in.top().second;
49     auto b = out.empty() ? oo : out.top().second;
50
51     ans[L] = min(a, b);
52 }
53
54 return ans;
55 }
```

Implementação

```
57 int main()
58 {
59     vector<int> xs { 1, 3, 2, 3, 4, 5, 8, 2, 7, 5, 3, 10, 6, 9 };
60     int K = 3;
61
62     auto ans = minimum((int) xs.size(), K, xs);
63
64     for (size_t i = 0; i < ans.size(); ++i)
65     {
66         cout << "[";
67
68         for (int j = 0; j < K; ++j)
69             cout << xs[i + j] << (j + 1 == K ? "]" : ", ");
70
71         cout << " -> " << ans[i] << '\n';
72     }
73
74     return 0;
75 }
```

- O 2SUM é um problema bastante conhecido: dado um vetor \vec{v} com N inteiros, determine se, para um dado S , existem dois elementos v_i e v_j tais que $v_i + v_j = S$
- Há $O(N^2)$ pares de elementos de \vec{v} , de modo que uma solução que verificasse cada um destes pares teria complexidade $O(N^2)$
- É possível usar a técnica dos dois ponteiros para reduzir complexidade
- Se \vec{v} não estiver ordenado, ordene-o em ordem não-decrescente
- Inicie os ponteiros $L = 0$ e $R = N - 1$
- Enquanto $R > L$ (ou $R \geq L$, se um elemento puder se utilizado duas vezes) e $\vec{v}[L] + \vec{v}[R] > S$, decemente R

- Caso $\vec{v}[L] + \vec{v}[R] = S$, a solução foi encontrada e o algoritmo termina
- Caso contrário, incremente L e repita o processo
- Se, em algum momento, $R < L$, o algoritmo finalizará e o problema não tem solução
- Este problema é um exemplo onde os ponteiros avançam em direções opostas
- Como cada elemento é observado, no máximo, uma única vez por cada ponteiros, a complexidade do algoritmo é $O(N)$
- Caso o vetor não esteja inicialmente ordenado, a complexidade passa a ser $O(N \log N)$, devido ao algoritmo de ordenação

Implementação do 2SUM

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 bool _2sum(int N, int S, vector<int>& xs)
6 {
7     sort(xs.begin(), xs.end());
8
9     int L = 0, R = N - 1;
10
11     // A solução exige dois elementos distintos de xs
12     while (L < R) {
13         while (R > L and xs[L] + xs[R] > S)
14             --R;
15
16         if (R <= L)
17             break;
18
19         if (xs[L] + xs[R] == S)
20             return true;
```

Implementação do 2SUM

```
22         ++L;  
23     }  
24  
25     return false;  
26 }  
27  
28 int main()  
29 {  
30     vector<int> xs { 1, -2, 5, 8, -3, 7, -5 };  
31     int N = (int) xs.size();  
32  
33     cout << _2sum(N, 0, xs) << endl;  
34     cout << _2sum(N, 1, xs) << endl;  
35     cout << _2sum(N, 4, xs) << endl;  
36     cout << _2sum(N, 14, xs) << endl;  
37  
38     return 0;  
39 }
```


1. **LAARKSONEN**, Antti. *Competitive Programmer's Handbook*, 2017.
2. **HALIM**, Steve; **HALIM**, Felix. *Competitive Programming 3*, Lulu, 2013.