

Problemas da seleção e da contagem de inversões

Uma das ideias centrais em algoritmos é que

- **abordagens** usadas para resolver um **problema**
 - podem ser **aplicadas/adaptadas** para problemas **diferentes**.

Nesta aula veremos como usar o que aprendemos

- ao estudar algoritmos eficientes para o problema da **ordenação**
 - para projetar algoritmos para dois problemas relacionados.

Problema da seleção

Definições:

- A **ordem** de um elemento é uma medida da **grandeza** dele
 - em **relação** aos seus **pares**.
- Assim, se a **ordem** de um elemento é $k \in \mathbb{Z}^+$
 - então existem k elementos de valor **menor** que o dele. $\Leftarrow$
- No problema da seleção, dado um vetor v de tamanho n
 - e um inteiro $k \in [0, n)$, queremos o **valor** do elemento de ordem k

Exemplos:

- $v = 3\ 2\ 5\ 4\ 1$ e $k = 3$. Elemento de ordem 3 é 4
- $v = 1\ 2\ 3\ 4\ 5$ e $k = 3$. Elemento de ordem 3 é 4
- $v = 5\ 4\ 3\ 2\ 1$ e $k = 3$. Elemento de ordem 3 é 4
- $v = 50\ 4\ 30\ 20\ 10$ e $k = 0$. *Rep.: 4*

A resposta não muda nas diferentes permutações,

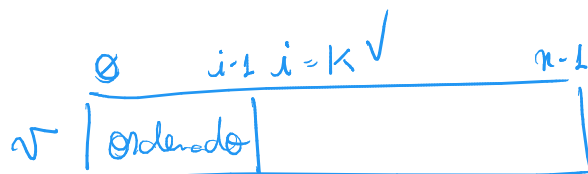
- pois a ordem de um elemento depende da comparação
 - de seu valor com os demais elementos,
- e não de sua posição no vetor.

Curiosidades:

- Na permutação **ordenada** do vetor $v[0 \dots n-1]$
 - o elemento de ordem k ocupa a k -ésima posição.
- Note que, podemos definir ordem começando em 0 ou em 1
 - Escolhi usar começar em 0 , para combinar com nossos vetores.
- Assim, se v estiver for ordenado
 - o elemento de ordem k ocupa a posição $v[k]$
- Perceba que o problema do **mínimo** e do **máximo**
 - são casos particulares do problema da seleção.
 - **Mínimo** corresponde ao elemento de **ordem** 0
 - **Máximo** corresponde ao elemento de **ordem** $n-1$
- O problema da seleção é trivial se v estiver ordenado,
 - ou se **ordenarmos** ele.
 - Qual seria a eficiência dessa abordagem? $O(n \lg n)$
- Será que conseguimos resolver o problema sem usar esta abordagem?

Em AED1, vimos um algoritmo baseado na ideia do selectionSort,

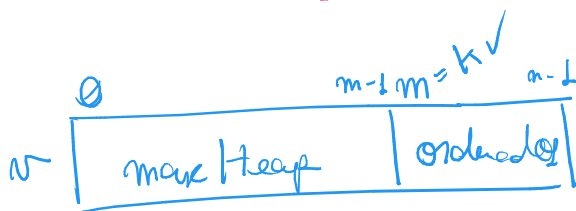
- com eficiência $O(n^2)$



iterar k vezes no laço do selection Sort

Também vimos um algoritmo baseado na ideia do heapSort,

- com eficiência $O((n-k) \lg n + n)$



m = n começo
m = k parada

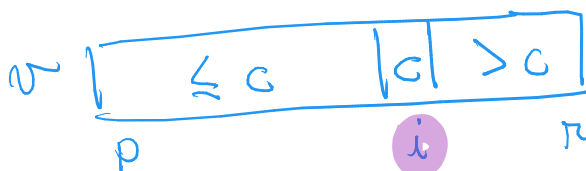
iterar por (n-k) vezes

Vamos tentar obter um algoritmo mais eficiente para este problema,

- nos inspirando no quickSort ou, mais especificamente,
 - no algoritmo para o problema da separação.

Relembrando, no problema da separação temos vetor v com limites p e r

- i.e., os elementos estão no subvetor $v[p..r]$
- e recebemos um pivô c que pertence a $v[p..r]$
- O objetivo é separar os elementos do vetor de modo que
 - o prefixo deste tenha os elementos $\leq c$
 - e o sufixo tenha os elementos $> c$



- Isto é, c deve terminar na posição i tal que $v[p..i-1] \leq c = v[i] < v[i+1..r]$
- Note que, c termina na posição que ele deve ocupar no vetor ordenado,
 - ou seja, a ordem de c é i
- Além disso, todo elemento com ordem menor que i está em $v[p..i-1]$
 - e todo elemento com ordem maior que i está em $v[i+1..r]$

Por isso, podemos projetar um algoritmo para o problema da seleção,

- que começa **separando** o **vetor** ao redor de um **pivô**,
 - e então decide em qual subvetor continuar a busca pelo k-ésimo.

// baseado no quickSort determinístico

// p indica a primeira posicao e r a ultima

```
int selecao3(int v[], int p, int r, int k) {
```

```
    int j;
```

```
    j = separa(v, p, r);
```

```
    if (k == j) — achou
```

```
        return v[j];
```

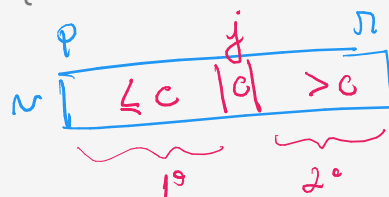
```
    if (k < j)
```

```
        return selecao3(v, p, j - 1, k); — busca no 1° subv.
```

```
    // if (k > j)
```

```
    return selecao3(v, j + 1, r, k); — busca no 2° subv.
```

```
}
```



Note que, não é necessário ajustar a ordem do elemento buscado

- mesmo quando ele está no **segundo subvetor**, pois
 - a rotina separa considera a posição do elemento no **vetor original**.

Eficiência de tempo: $O(n^2)$ no **pior caso**.

pivôs 0 **chamadas recursivas** **nível** **# sub. prob.** **tam sub. prob.**



0

1

n



1

1

n-1



2

1

n-2

⋮

⋮

⋮

⋮



h

1

1 = n - h

⇓

h = n - 1

- Note que, o trabalho por nível é proporcional
 - ao tamanho do subproblema.
- Por isso, o trabalho total é proporcional a

$$n + (n-1) + (n-2) + \dots + 2 + 1 = \frac{(n+1) \cdot n}{2} \approx \frac{n^2}{2}$$

Eficiência de espaço: $O(n)$ espaço adicional,

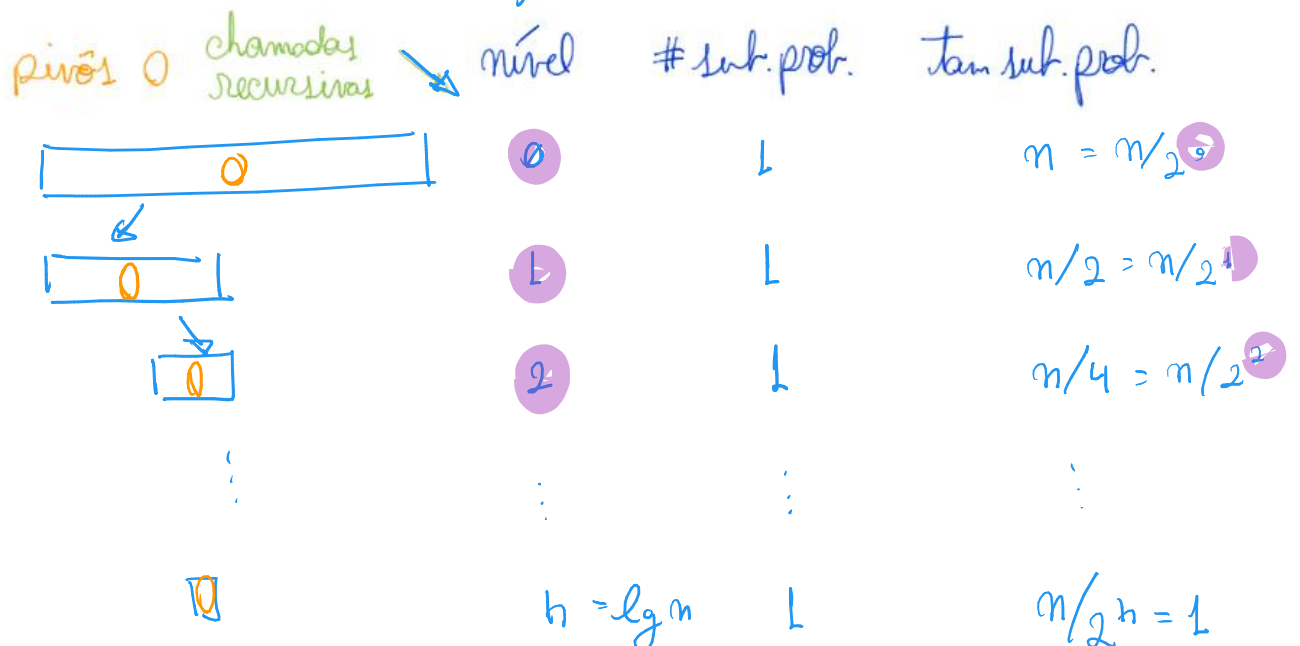
- por conta da altura da pilha de recursão.

Para evitar esse pior caso, precisamos de bons pivôs.

- Assim como no caso do quickSort,
 - usamos aleatoriedade para conseguir bons pivôs, em média.

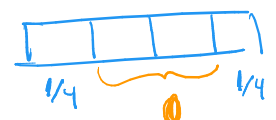
```
// baseado no quickSort aleatorizado
// p indica a primeira posicao e r a ultima
int selecao4(int v[], int p, int r, int k) {
    int desl, j;
    desl = (int)((double)rand() / (RAND_MAX + 1)) * (double)(r - p) + 1));
    troca(&v[p + desl], &v[r]);
    j = separa(v, p, r); —  $O(n-p)$ 
    if (k == j)
        return v[j]; — achou
    if (k < j)
        return selecao4(v, p, j - 1, k); — 1°
    // if (k > j)
    return selecao4(v, j + 1, r, k); — 2°
}
```

Eficiência de tempo esperado: $O(n)$



- Note que, o trabalho por nível é proporcional
 - ao tamanho do subproblema.
- Por isso, o trabalho total é proporcional a $n + n/2 + n/4 + \dots \leq 2n$
- Claro que nem todo pivô dividirá o vetor exatamente ao meio,
 - mas, em média, a cada dois pivôs, um será bom,
 - mantendo o resultado a menos de uma constante.

$$2^h = n \Rightarrow h = \lg n$$



Eficiência de espaço esperado: $O(\lg n)$ espaço adicional,

- por conta da altura da pilha de recursão.

Observe que, o algoritmo recursivo anterior apresenta recursão caudal,

- o que nos permite transformá-lo em um algoritmo iterativo.

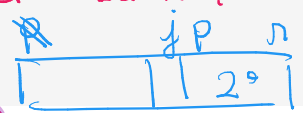
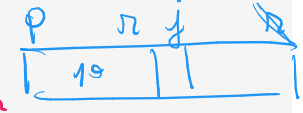
```
int selecao5(int v[], int p, int r, int k) {
    while (1) {
        int desl = (int)(((double)rand() / (RAND_MAX + 1)) *
        (double)(r - p + 1));
        troca(&v[p + desl], &v[r]);
        int j = separa(v, p, r);
        if (k == j) return v[j];
        if (k < j) r = j - 1;
        else /* (k > j) */ p = j + 1;
    }
}
```

laco

encontrei

busca 1º subvetor

busca 2º subvetor

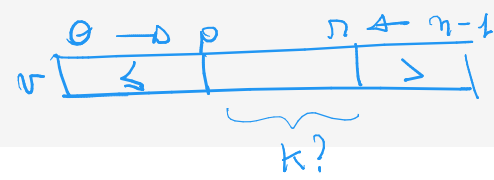


- Eficiência de tempo esperado: $O(n-p) = O(n)$
- Eficiência de espaço: $O(1)$ espaço adicional,
 - já que o número de variáveis auxiliares independe da entrada.

Podemos refinar esta versão iterativa,

- colocando a condição de parada no do {...} while(...)
 - e removendo os limites do vetor dos parâmetros.

```
int selecao6(int v[], int n, int k) {
    int p = 0;
    int r = n - 1;
    do {
        int desl = (int)(((double)rand() / (RAND_MAX + 1)) *
        (double)(r - p + 1));
        troca(&v[p + desl], &v[r]);
        int j = separa(v, p, r);
        if (k < j) r = j - 1;
        else /* if (k > j) */ p = j + 1;
    } while (k != j);
    return v[j];
}
```



- Invariantes e corretude:
 - $v[0..p-1] \leq v[p..r] < v[r+1..n-1]$
 - Depois do separa(), $v[j]$ corresponde ao j -ésimo menor elemento.
- Eficiência de tempo esperado: $O(n)$
- Eficiência de espaço: $O(1)$ espaço adicional,
 - já que o número de variáveis auxiliares independe da entrada.

Quiz1: Todas as nossas adaptações de algoritmos para seleção são de algoritmos

- que colocam alguns elementos em suas posições definitivas,
 - muito antes do vetor todo estar ordenado. Será coincidência?

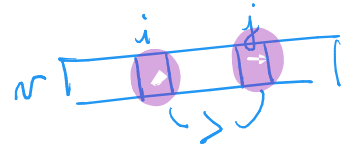
Problema da Contagem de Inversões

Definição:

- Uma **inversão** é um par de elementos $v[i]$ e $v[j]$ tal que $i < j$ e $v[i] > v[j]$
- Dado um vetor v de tamanho n , quantas inversões existem em v ?

Exemplos:

- $v = 3\ 2\ 5\ 4\ 1$
 - 3 está invertido com 2 e 1
 - 2 está invertido com 1
 - 5 está invertido com 4 e 1
 - 4 está invertido com 1
 - Total de inversões = $2 + 1 + 2 + 1 = 6$
- $v = 1\ 2\ 3\ 4\ 5$
 - Total de inversões = 0
- $v = 5\ 4\ 3\ 2\ 1$
 - 5 está invertido com 4, 3, 2 e 1
 - 4 está invertido com 3, 2 e 1
 - 3 está invertido com 2 e 1
 - 2 está invertido com 1
 - Total de inversões = $4 + 3 + 2 + 1 = 10$

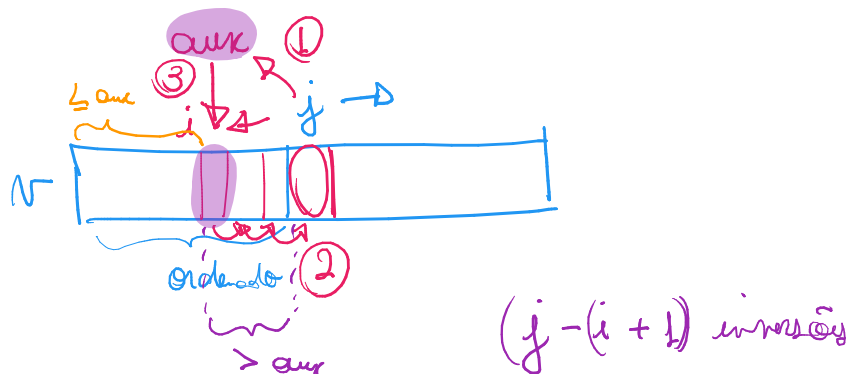


Curiosidades: Número mínimo de inversões = 0

- que ocorre quando o vetor está em **ordem crescente**.
- Número máximo de inversões = $\binom{n}{2} = C_2^n = \frac{n \cdot (n-1)}{2}$
 - O valor $\binom{n}{2} = C_2^n$ corresponde a todo par ser uma inversão,
 - e ocorre quando o vetor está em **ordem decrescente**.
- Assim, podemos pensar no número de inversões
 - como uma medida da **desordem** dos elementos de um vetor.

Em AED1, vimos um algoritmo baseado na ideia do insertionSort,

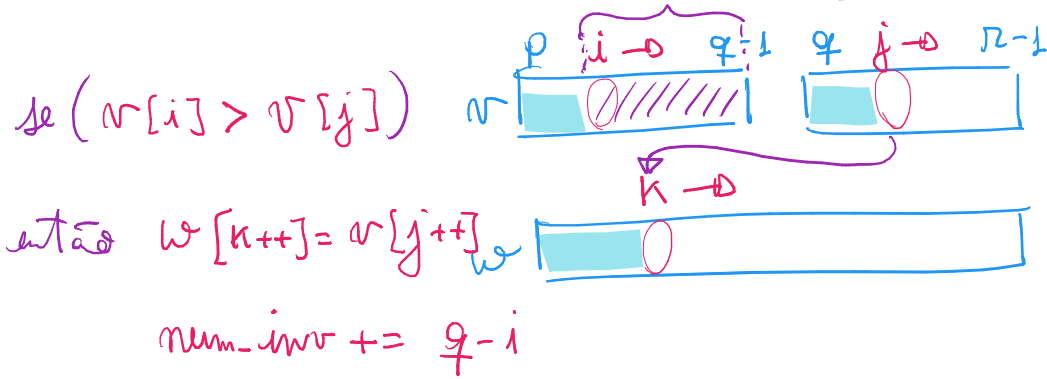
- com eficiência $O(n^2)$ no pior caso.



- Também vimos um algoritmo baseado na ideia do bubbleSort,
 - que conta +1 cada vez que desfaz uma inversão.

Vamos tentar obter um algoritmo mais eficiente para este problema,

- nos inspirando no mergeSort e, em particular, na rotina de intercalação.



// primeiro subvetor entre p e $q-1$, segundo subvetor entre q e $r-1$

```
unsigned long long intercalaContando(int v[], int p, int q, int r) {
    int i, j, k, tam;
    unsigned long long num_inv = 0;

    i = p;
    j = q;
    k = 0;
    tam = r - p;
    int *w = malloc(tam * sizeof(int));

    while (i < q && j < r) {
        if (v[i] <= v[j])
            w[k++] = v[i++];
        else { // v[i] > v[j]
            w[k++] = v[j++];
            num_inv += q - i;
        }
    }

    while (i < q) w[k++] = v[i++];
    while (j < r) w[k++] = v[j++];

    for (k = 0; k < tam; k++)
        v[p + k] = w[k];

    free(w);
    return num_inv;
}
```

Invariante e corretude:

- $w[0..k-1]$ ordenado e possui os elementos de $v[p..i-1]$ e $v[q..j-1]$
- num_inv = número de inversões envolvendo
 - elementos de $v[q..j-1]$ e elementos de $v[p..q-1]$

Eficiência de tempo: $O(r-p) = O(tam)$

Eficiência de espaço: $O(r-p)$ espaço adicional, por conta do vetor auxiliar.

Usando essa variante da rotina de intercalação

- que devolve o número de inversões que ela desfez ao intercalar,
 - podemos projetar uma variante do mergeSort,
 - que conta o número de inversões enquanto ordena o vetor.

```
// baseado no mergeSort
```

```
// p indica a primeira posicao e r-1 a ultima
```

```
unsigned long long contarInversoesR(int v[], int p, int r) {
```

```
    int m;
```

```
    unsigned long long num_inv = 0;
```

```
    if (r - p > 1) {
```

```
        m = (p + r) / 2; // m = p + (r - p) / 2;
```

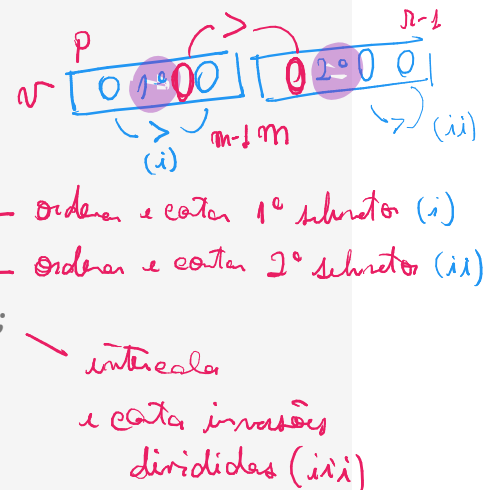
```
        num_inv += contarInversoesR(v, p, m);
```

```
        num_inv += contarInversoesR(v, m, r);
```

```
        num_inv += intercalaContando(v, p, m, r);
```

```
    return num_inv;
```

```
}
```



- Eficiência de tempo: $O((r-p) \lg(r-p))$
- Eficiência de espaço: $O(r-p)$ espaço adicional.

Quiz2: Por que, ao analisar o espaço adicional necessário,

- não estou preocupado com o tamanho da pilha de recursão?

```
unsigned long long contarInversoes3(int v[], int n) {
```

```
    return contarInversoesR(v, 0, n);
```

```
}
```

- Eficiência de tempo: $O(n \lg n)$
- Eficiência de espaço: $O(n)$ espaço adicional.

Quiz3: Todas as nossas adaptações de algoritmos para contagem de inversões

- são de algoritmos de ordenação estável. Será coincidência?