

AED2 - Aula 12

Problemas da seleção e da contagem de inversões

“Podemos fazer melhor?”
- mote do projetista de algoritmos

Uma das ideias centrais em algoritmos é que

- abordagens usadas para resolver um problema
 - podem ser bem sucedidas quando 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
 - então existem k elementos de valor menor que o dele.
- No problema da seleção, dado um vetor v de tamanho n
 - e um inteiro k em $[0, n)$, queremos o valor do elemento de ordem k .

Exemplos:

- 3 2 5 4 1 e $k = 3$
 - Elemento de ordem 3 é 4
- 1 2 3 4 5 e $k = 3$
 - Elemento de ordem 3 é 4
- 5 4 3 2 1 e $k = 3$
 - Elemento de ordem 3 é 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.

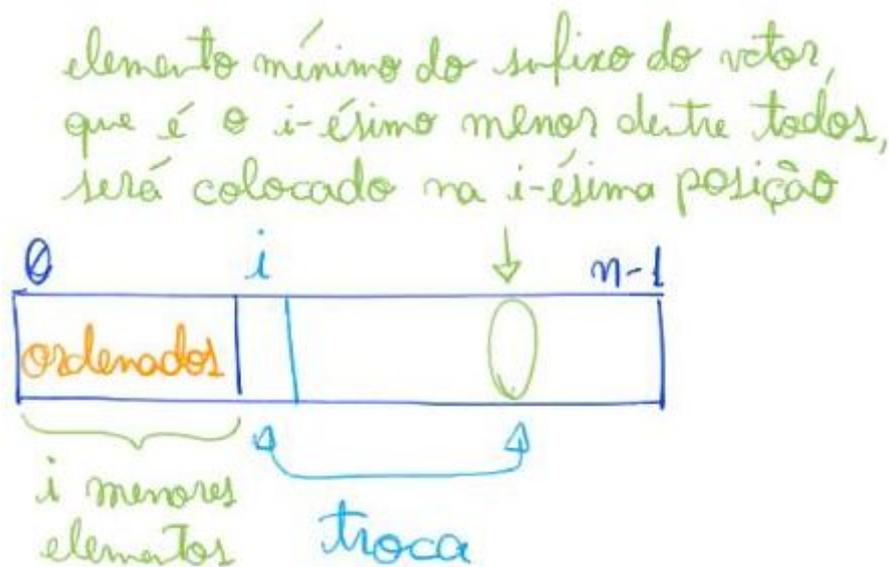
Curiosidades:

- Na permutação ordenada do vetor $v[0 .. n - 1]$
 - o elemento de ordem k ocupa a k -ésima posição.

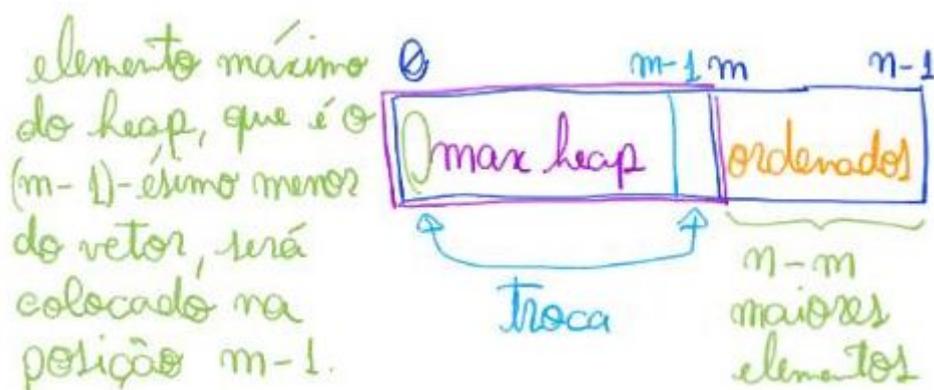
- Note que, podemos definir ordem começando em 0 ou em 1.
 - Escolhi usar ela começando em 0, para combinar com nossos vetores.
 - Assim, o elemento de ordem k ocupa a posição $v[k]$ se v for ordenado.
- Perceba que o problema do mínimo e do máximo
 - são casos particulares do problema da seleção.
 -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

Em AED1

- vimos um algoritmo baseado na ideia do selectionSort,
 - com eficiência $O(n^2)$.



- Também vimos um algoritmo baseado na ideia do heapSort,
 - com eficiência $O(n + (n - k) \log n)$.

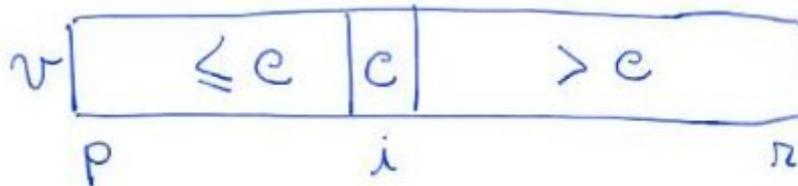


Vamos tentar obter um algoritmo mais eficiente para este problema,

- nos inspirando no quickSort ou,
 - mais especificamente, na rotina de separação.

Relembrando, o problema da separação consiste de,

- dado um vetor v com limites p e r ,
 - i.e., os elementos do vetor estão em $v[p .. r]$,
 - e um pivô c que está em $v[p .. r]$,
- separar os elementos do vetor de modo que
 - o prefixo deste tenha os elementos $\leq c$,
 - e o sufixo tenha os elementos $> c$.



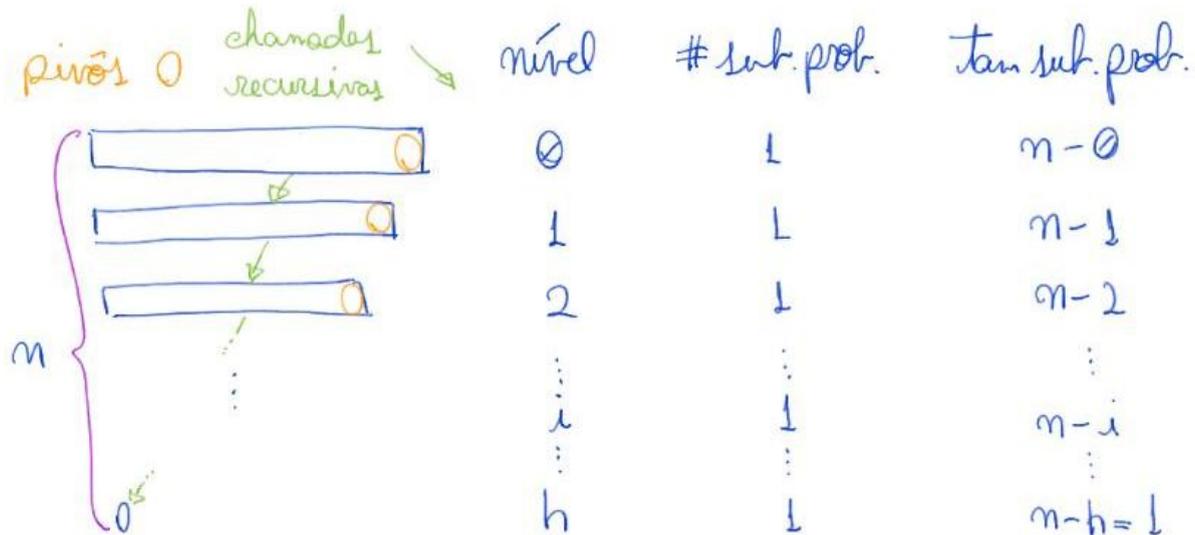
- Isto é, c deve terminar numa posição i tal que
 - $v[p .. i - 1] \leq c = v[i] < v[i + 1 .. r]$.
- Note que, c termina na posição que ele deve ocupar no vetor ordenado,
 - ou seja, ordem de c é i .
- Além disso, todo elemento com ordem menor que i está em $v[p .. i - 1]$
 - e todo elemento com ordem 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 deve 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)
        return v[j];
    if (k < j)
        return selecao3(v, p, j - 1, k);
    // if (k > j)
    return selecao3(v, j + 1, r, k);
}
```

- Note que, não é necessário ajustar a ordem do elemento buscado
 - mesmo quando ele está no segundo subvetor, pois
 - a rotina separa devolve a posição do elemento no vetor original.
- Eficiência de tempo: $O(n^2)$ no pior caso.



- Note que, o trabalho por nível é proporcional
 - ao tamanho do subproblema.
- Por isso, o trabalho total é proporcional a
 - $n + (n - 1) + (n - 2) + \dots + 3 + 2 + 1 = n(n - 1) / 2 \approx n^2 / 2$.
- 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,
 - usaremos aleatoriedade para conseguir bons pivôs em média.

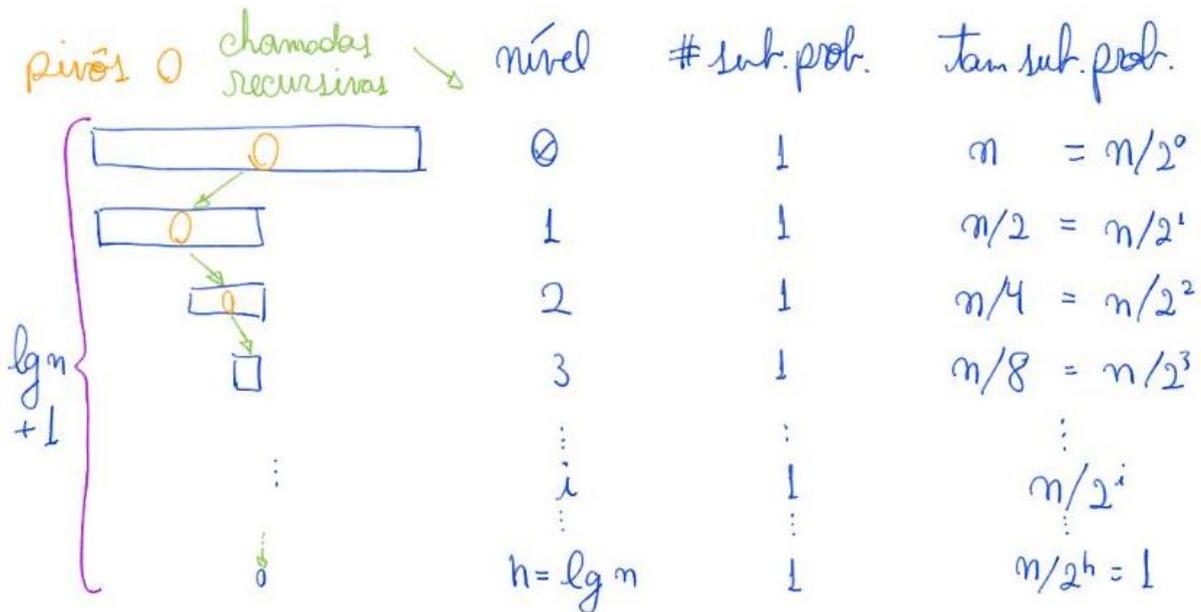
```
// baseado no quickSort aleatorizado
// p indica a primeira posicao e r a ultima
int selecao4(int v[], int p, int r, int k)
{
    int desl, j;
    desl = (int)((double)rand() / (RAND_MAX + 1)) * (double)(r - p
+ 1));
    troca(&v[p + desl], &v[r]);
    j = separa(v, p, r);
    if (k == j)
      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

```

return selecao4(v, p, j - 1, k);
// if (k > j)
return selecao4(v, j + 1, r, k);
}

```

- 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, em média, nem todo pivô dividirá o vetor ao meio,
 - mas em média a cada dois pivôs, um será bom,
 - mantendo o resultado a menos de uma constante.
- 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.

```

// versão iterativa obtida pela remoção da recursão caudal
// p indica a primeira posicao e r a ultima
int selecao5(int v[], int p, int r, int k)
{
    int desl, j;
    while (1)
    {

```

```

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

```

- Eficiência de tempo esperado: $O(n)$
- 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.

// versão iterativa refinada

// p indica a primeira posicao e r a ultima

```
int selecao6(int v[], int n, int k)
```

```
{
```

```
    int desl, j;
```

```
    int p = 0;
```

```
    int r = n - 1;
```

```
    do
```

```
    {
```

```
        desl = (int)((((double)rand() / (RAND_MAX + 1)) * (double)(r
- p + 1)));
```

```
        // printf("p = %d, r = %d, r-p+1 = %d, desl = %d\n", p, r, r
- p + 1, desl);
```

```
        troca(&v[p + desl], &v[r]);
```

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

Quizz:

- 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 corresponde a 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 ,
 - queremos saber quantas inversões existem em v .

Exemplos:

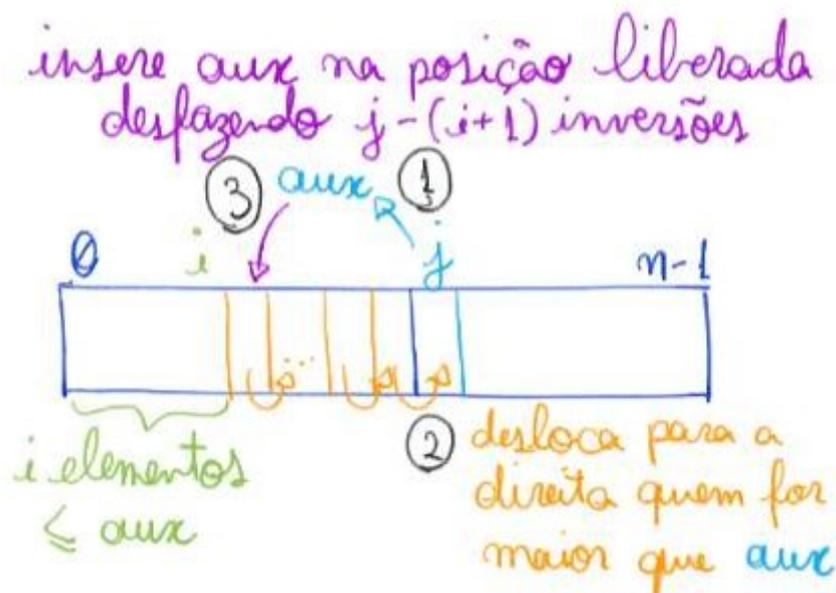
- 3 2 5 4 1
 - 3 está invertido com 2 e 1
 - 2 está invertido com 1
 - 5 está invertido com 4 e 1
 - 4 está invertido com 1
 - Total de inversões = $2 + 1 + 2 + 1 = 6$
- 1 2 3 4 5
 - Total de inversões = 0
- 5 4 3 2 1
 - 5 está invertido com 4, 3, 2 e 1
 - 4 está invertido com 3, 2 e 1
 - 3 está invertido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

Curiosidades:

- Número mínimo de inversões = 0,
 - que ocorre quando o vetor está em ordem crescente.
- Número máximo de inversões = $(n \text{ escolhe } 2) = n(n - 1)/2$.
 - O valor $(n \text{ escolhe } 2)$ corresponde a todo par ser uma inversão,
 - 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

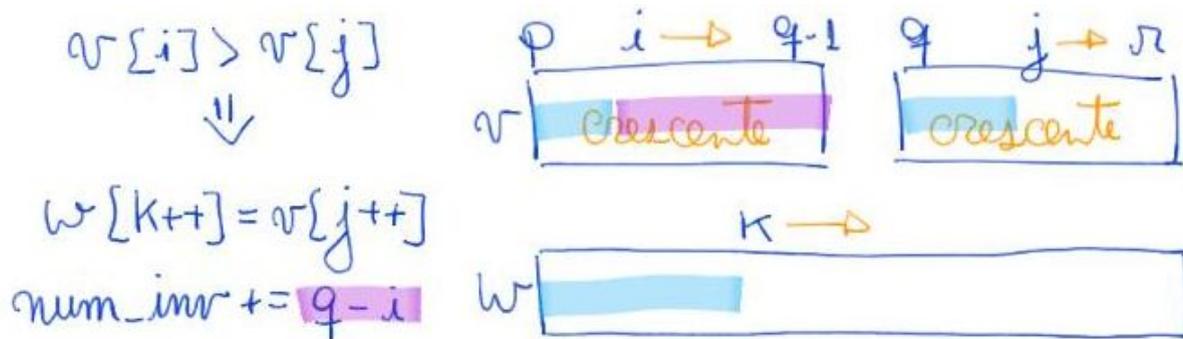
- vimos um algoritmo baseado na ideia do insertionSort,
 - com eficiência $O(n^2)$ no pior caso.



- 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 $v[0 .. i - 1]$ e $v[q .. j - 1]$.
 - num_inv = número de inversões envolvendo
 - elementos de $v[q .. j - 1]$ e elementos de $v[0 .. q - 1]$.
- Eficiência de tempo: $O(r - p)$.
- 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.

```
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.

Quizz:

- Todas as nossas adaptações de algoritmos para contagem de inversões
 - são de algoritmos de ordenação estável. Será coincidência?