AED1 - Aula 23
Problemas da seleção e da contagem de inversões

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 para o problema da ordenação
    - para projetar algoritmos para dois problemas relacionados.

Problema da seleção

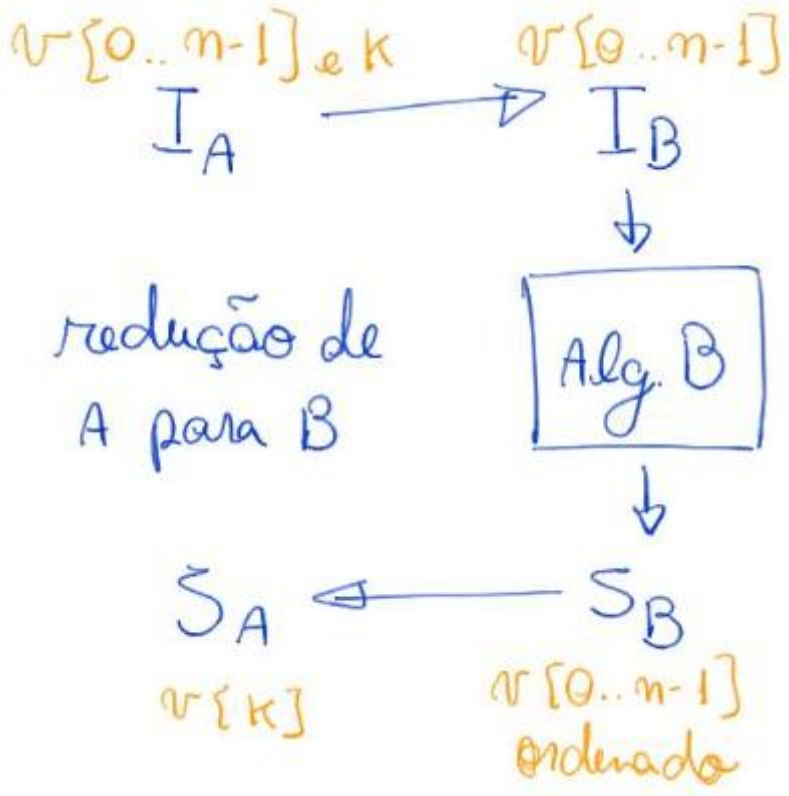
Definições:
- A ordem de um elemento é uma medida da grandeza dele
    - em relação aos seus pares.
- Assim, se a ordem de um elemento é k
    - então existem k elementos de valor menor que o dele.
- Dado um vetor v de tamanho n e um inteiro k em [0, n)
    - no problema da seleção queremos o valor do 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,

        - que também começam na posição 0.
    - Assim, o elemento de ordem k ocupa a posição v[k] se v for ordenado.
- Perceba que os problemas do mínimo e do máximo
    - são casos particulares do problema da seleção.
- Mínimo corresponde ao elemento de ordem 0
    - e máximo corresponde ao elemento de ordem n - 1.
- Observe que o problema da seleção é trivial
    - se v estiver ordenado, ou se ordenarmos ele.
- Esta abordagem é conhecida como redução entre problemas.

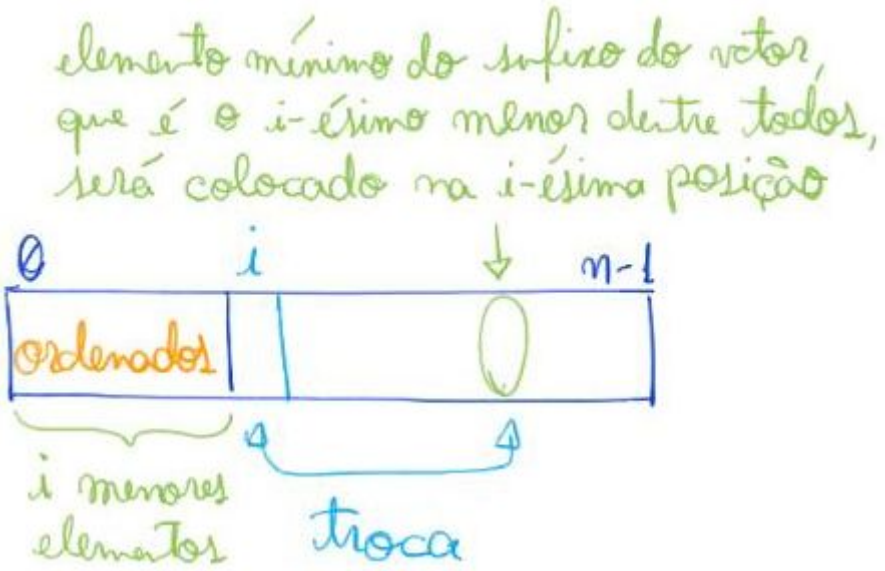


    - Qual é a eficiência da aplicação dessa abordagem neste caso?

Será que conseguimos resolver o problema sem usar esta abordagem?

- Observem que, alguns algoritmos de ordenação que estudamos
    - colocam elementos em suas posições definitivas
        - muito antes do vetor estar totalmente ordenado.
- Talvez possamos explorar essa propriedade.

Algoritmo baseado na ideia do selectionSort:




```c
// devolve o k-ésimo menor elemento
int selecao1(int v[], int n, int k) {
    int i, j, ind_min;
    // em cada iteração encontra o i-ésimo menor
    for (i = 0; i <= k; i++) {
        ind_min = i;
        for (j = i + 1; j < n; j++)
            if (v[j] < v[ind_min])
                ind_min = j;
        troca(&v[i], &v[ind_min]);
    }
    return v[k];
}
```

Invariante e corretude:

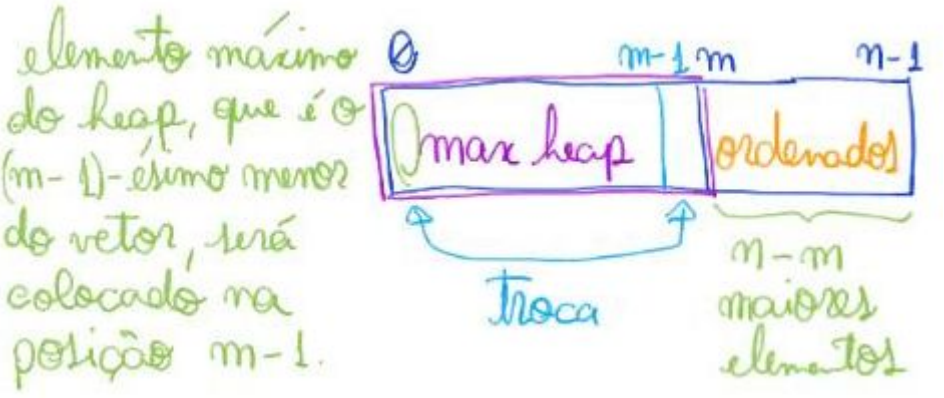
- No início de cada iteração do laço externo
    - v[0 .. i - 1] está ordenado e
    - v[0 .. i - 1] <= v[i .. n - 1].

Eficiência de tempo:

- O(k n), dado que fazemos k buscas lineares pelo mínimo
    - para encontrar o k-ésimo elemento.

Eficiência de espaço: O(1) espaço adicional usado com variáveis auxiliares.

Algoritmo baseado na ideia do heapSort:




```c
int selecao2(int v[], int n, int k) {
    int i, m;
    for (i = n / 2; i >= 0; i--)
        desceHeap(v, n, i);
    // em cada iteração encontra o (m-1)-ésimo menor
    for (m = n; m > k; m--) {
        troca(&v[0], &v[m - 1]);
        desceHeap(v, m - 1, 0);
    }
    return v[k];
}
```

Invariante e corretude:

- No início de cada iteração do segundo laço
  - v[m .. n - 1] está ordenado,
  - v[m .. n - 1] >= v[0 .. m - 1] e
  - v[0 .. m - 1] é um heap de máximo.

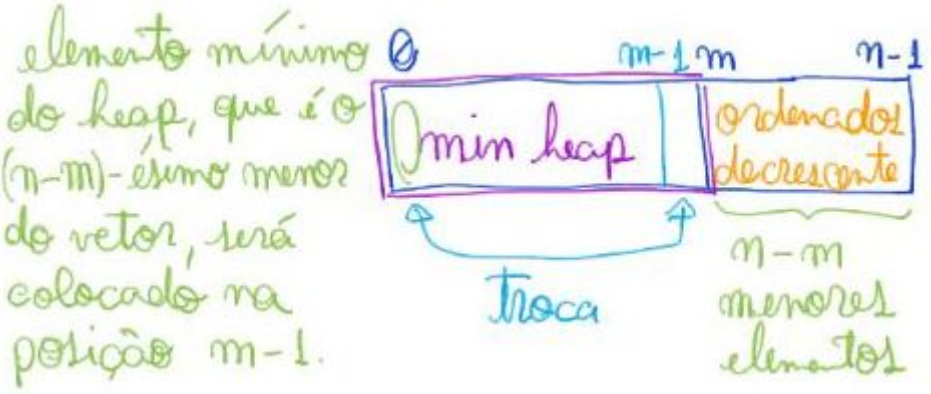
Eficiência de tempo:

- O(n + (n - k) lg n), sendo que
  - o primeiro O(n) é gasto para construir o heap,
  - e (n - k) é o número de remoções do heap de máximo,
    - até encontrar o k-ésimo elemento.
- Note que, se (n - k) <= n / lg n
  - a eficiência do algoritmo é linear em n, i.e., O(n).

Eficiência de espaço: O(1) espaço adicional usado com variáveis auxiliares.

Quizz:

- Como podemos melhorar os algoritmos anteriores,
    - usando como base a comparação de k com n?
- A ideia é que encontrar um elemento de ordem k próxima dos extremos
    - parece ser mais fácil.
- Realizar a melhoria em selecao1 é simples,
    - basta percorrer o vetor do fim para o começo,
        - buscando o máximo a cada iteração.
- Para aplicar a mesma ideia em selecao2
    - temos que resolver algumas complicações envolvendo o heap.



- Em particular, podemos usar um heap de mínimo,
    - ir ordenando o vetor em ordem decrescente
        - e parar quando n - m = k, devolvendo v[m].

Curiosidades:

- Nossas adaptações para o problema da seleção usam algoritmos
    - que posicionam elementos em suas posições definitivas
        - muito antes do vetor estar ordenado.
    - Por exemplo, não seria viável adaptar um que não o faz,
        - como o insertionSort.
    - E o bubbleSort, seria viável?
- Futuramente veremos que
    - usando a técnica de divisão e conquista,
        - aquela que é a base da busca binária,
    - junto de escolhas aleatórias,
        - é possível obter soluções mais eficientes para esse problema.

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,
    - ocorre quando o vetor está em ordem crescente.
- Número máximo de inversões = (n escolhe 2) = n(n - 1)/2.
    - O valor (n escolhe 2) 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.

A seguir temos o pseudocódigo de um primeiro algoritmo para contar inversões:

```
contarInversoes(vetor v[0 .. n - 1]) {
        num_inv = 0
        para i = 0 até n - 1
                para j = i + 1 até n - 1
                        se v[i] > v[j]
                                num_inv += 1
        devolva num_inv
}
```

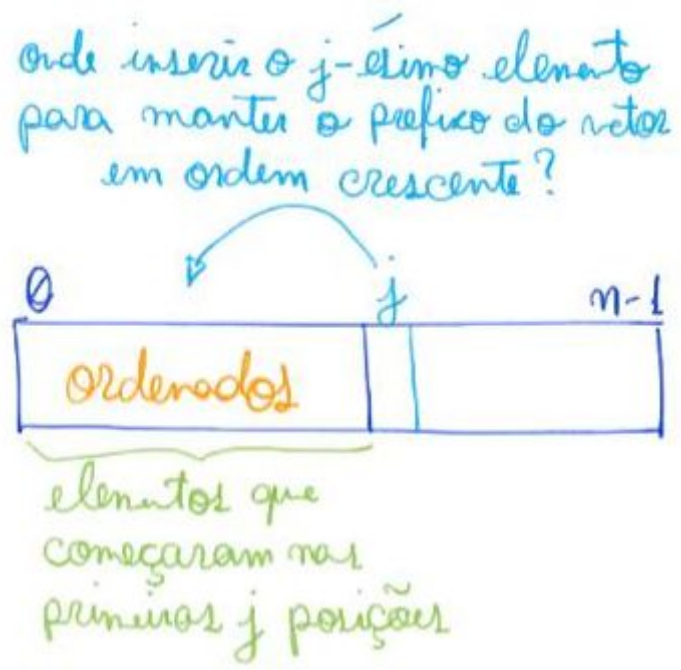
Eficiência de tempo:

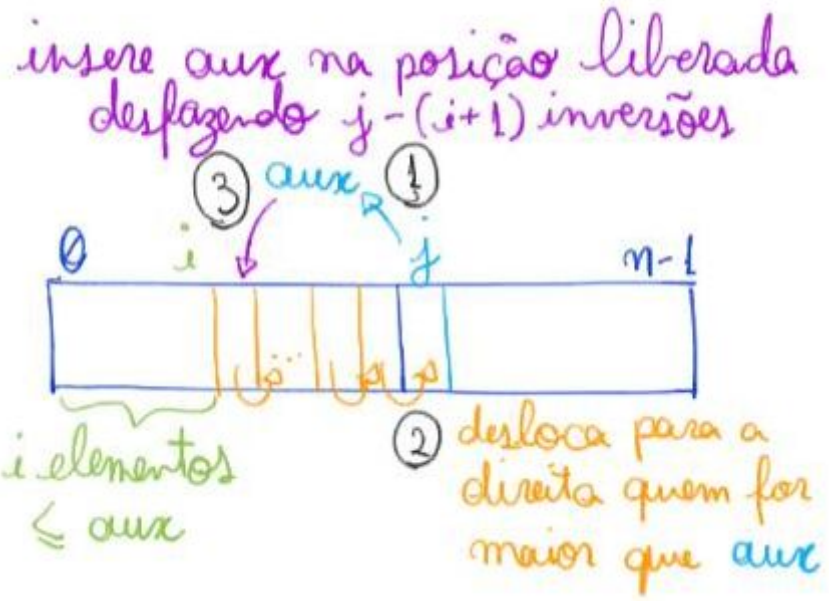
- O(n^2) tanto no pior quanto no melhor caso.

Será que conseguimos fazer melhor

- usando ideias de alguns dos algoritmos de ordenação que estudamos?

Algoritmo baseado na ideia do insertionSort:






```
unsigned long long contarInversoes1(int v[], int n) {
    int i, j, aux;
    unsigned long long num_inv = 0;
    for (j = 1; j < n; j++) {
        aux = v[j];
        for (i = j - 1; i >= 0 && aux < v[i]; i--) {
```

```
            v[i + 1] = v[i]; // desloca à direita os maiores
            // num_inv++;
        }
        num_inv += j - 1 - i;
        v[i + 1] = aux; /* por que i+1? */
    }
    return num_inv;
}
```

Invariante e corretude:

- No início de cada iteração do laço externo
    - v[0 .. j - 1] está ordenado e
    - num_inv = número de inversões envolvendo apenas
        - os elementos do subvetor v[0 .. j - 1].

Eficiência de tempo:

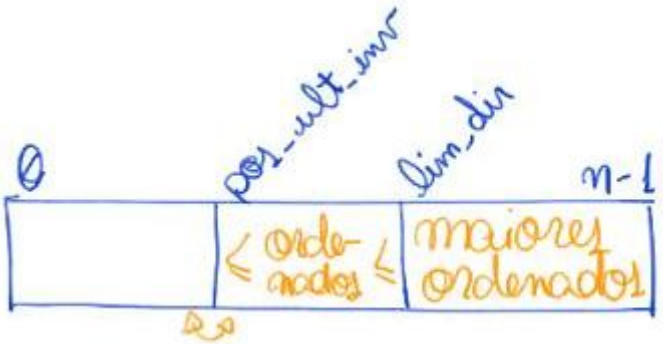
- O(n^2) no pior caso, ex.: vetor em ordem decrescente,
- O(n) no melhor caso, ex.: vetor em ordem crescente.

Eficiência de espaço: O(1) espaço adicional usado com variáveis auxiliares.

Algoritmo baseado na ideia do bubbleSort:

```
unsigned long long contarInversoes2(int v[], int n) {
    int j, i, aux, pos_ult_inv, lim_dir;
    unsigned long long num_inv = 0;
    lim_dir = n;
    for (j = 0; j < n; j++) {
        pos_ult_inv = 0;
        for (i = 1; i < lim_dir; i++)
            if (v[i - 1] > v[i]) {
                aux = v[i - 1];
                v[i - 1] = v[i];
                v[i] = aux;
                pos_ult_inv = i;
                num_inv++;
            }
        lim_dir = pos_ult_inv;
```

```
    }
    return num_inv;
}
```



Invariante e corretude:

- No início de cada iteração do laço externo
    - v[lim_dir .. n - 1] está ordenado,
    - v[lim_dir .. n - 1] >= v[0 .. lim_dir - 1] e
    - num_inv = número de inversões desfeitas até o momento.

Eficiência de tempo:

- O(n^2) no pior caso, ex.: menor elemento na última posição,
    - ainda que neste caso o número de inversões seja n - 1.
- O(n) no melhor caso, ex.: vetor em ordem crescente.

Eficiência de espaço: O(1) espaço adicional usado com variáveis auxiliares.

Quizz1:

- Todas as nossas adaptações de algoritmos para contagem de inversões
    - são de algoritmos de ordenação estável. Será coincidência?
- Perceba que algoritmos não estáveis, como o selectionSort,
    - criam inversões ao mesmo tempo que as eliminam.
- Ainda que eles sempre eliminem mais do que criam, afinal, estão ordenando,
    - isso dificulta a contagem do delta de inversões.

Quizz2:

- Por que devolvemos valor e usamos variáveis unsigned long long int?

Curiosidade:

- Futuramente veremos que, usando a técnica de divisão e conquista
    - é possível obter soluções mais eficientes para esse problema.