AED1 - Aula 20
Ordenação por inserção (insertionSort)
e por transposição (bubbleSort)

Problema da ordenação

Considere um vetor v de inteiros com n elementos.

Dizemos que ele está em ordem crescente se
    v[0] <= v[1] <= v[2] <= ... <= v[n-2] <= v[n-1].

Um algoritmo para o problema da ordenação deve
- rearranjar (permutar) os elementos de v de modo a torná-lo crescente.

Podemos definir o problema complementar para ordenação decrescente.

Também podemos definir o problema para quaisquer elementos
- cujos objetos são comparáveis e possuem uma relação de ordem total.

Para três algoritmos básicos (e um mais avançado) veremos:
- Ideia e exemplo.
- Código.
- Invariantes e corretude.
- Eficiência de tempo: no melhor e pior casos.
- Estabilidade: algoritmo é estável
  - se não inverte a posição relativa de valores idênticos.
- Eficiência de espaço: algoritmo é in place se não usa estruturas auxiliares
  - (e portanto memória) com tamanho proporcional à entrada.

Ordenação por inserção (insertionSort)

Ideia e exemplo:
- Varre o vetor do início ao fim e, a cada novo elemento encontrado,
  - o coloca na posição correta no subvetor já visitado.
- Como exemplo, considere o vetor 7 5 2 3 9 8

7 | 5 2 3 9 8

5 7 | 2 3 9 8

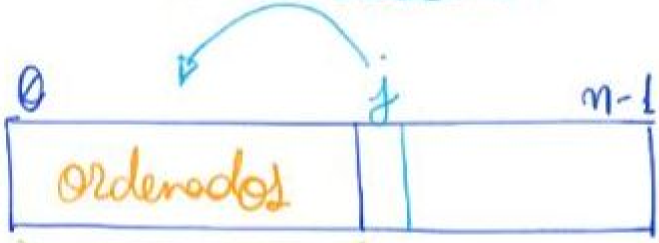
2 5 7 | 3 9 8

2 3 5 7 | 9 8

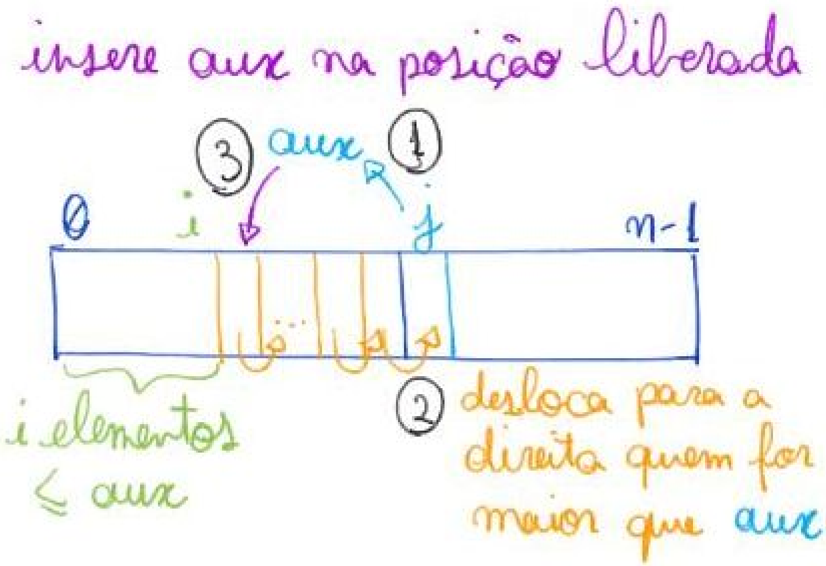
2 3 5 7 9 | 8

2 3 5 7 8 9 |

onde inserir o j-ésimo elemento para manter o prefixo do vetor em ordem crescente?



elementos que começaram nas primeiras j posições



Código:

```
void insertionSort(int v[], int n)
{
    int i, j, aux;
    for (j = 1; /*1*/ j < n; j++)
    {
        aux = v[j];
        for (i = j - 1; /*2*/ i >= 0 && aux < v[i]; i--)
            v[i + 1] = v[i]; // desloca à direita os maiores
        v[i + 1] = aux;      // por que i+1?
    }
}
```

Invariante e corretude:

- Os invariantes do laço externo,
  - que valem no início `/*1*/` de cada iteração são
    - o vetor é uma permutação do original,
    - v[0 .. j - 1] está ordenado.
- Os invariantes do laço interno,
  - que valem no início `/*2*/` de cada iteração são
    - v[0 .. i] e v[i+2 .. j] são crescentes,
    - v[0 .. i] ≤ v[i+2 .. j],
    - v[i+2 .. j] > aux.
- Demonstrar que esses invariantes estão corretos,
  - verificando que eles valem logo antes da primeira iteração

            - e que seguem valendo de uma iteração para outra.
- Verificar que, no final do laço,
    - os invariantes implicam a corretude do algoritmo.

Eficiência de tempo:

- No melhor caso é da ordem de n, i.e., O(n).
    - Ex.: vetor está ordenado ou tem apenas adjacentes fora de ordem.
- O pior caso ocorre quando, em todas as n - 1 iterações do laço externo,
    - o laço interno realiza o número máximo de iterações,
        - isto é, j.
    - Como j começa em 1 e cresce de 1 a cada iteração do laço externo,
        - o total de iterações do laço interno é a soma da PA
            - 1 + 2 + 3 + … + n - 1 = n (n - 1) / 2 ~= n^2 / 2 = O(n^2).
    - Ex.: vetor está em ordem decrescente.
- Bônus:
    - Já que o prefixo v[0.. j - 1] do vetor sempre está ordenado,
        - por que não usamos busca binária para encontrar
            - a posição de inserção do j-ésimo elemento?
    - Essa variante do algoritmo funciona?
        - Isto é, ela está correta?
    - Qual sua eficiência?

Estabilidade:

- Ordenação é estável.
    - Por que?
- O que acontece se trocarmos "aux < v[i]" por "aux <= v[i]"?
    - Continua ordenando?
    - Continua estável?

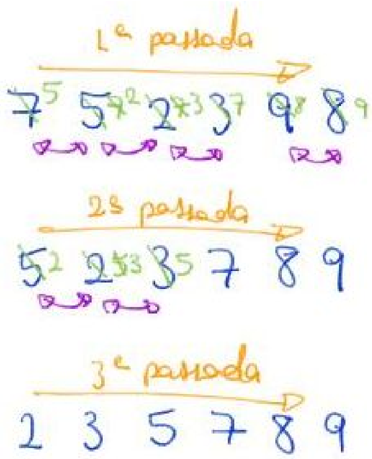
Eficiência de espaço:

- Ordenação é in place, pois só usa estruturas auxiliares (e portanto memória)
    - de tamanho constante em relação à entrada.

Ordenação por transposição (bubbleSort)

Ideia e exemplo:

- Varre o vetor diversas vezes, invertendo pares adjacentes fora de ordem.
- Como exemplo, considere o vetor 7 5 2 3 9 8



Códigos:

```c
void bubbleSort1(int v[], int n)
{
    int i, aux, mudou;
    do {
        mudou = 0;
        for (i = 1; i < n; i++)
            if (v[i - 1] > v[i])
            {
                aux = v[i - 1];
                v[i - 1] = v[i];
                v[i] = aux;
                mudou = 1;
            }
    } while (mudou == 1);
}
```

- Por que esse algoritmo para?
    - Porque um vetor em que todo par adjacente respeita
        - v[i - 1] <= v[i] está ordenado por definição.
- Mas qual será a eficiência de tempo de pior caso dele?
    - Voltaremos para isso depois.

```c
void bubbleSort2(int v[], int n)
{
    int j, i, aux;
    for (j = 0; j < n; j++)
        for (i = 1; i < n; i++)
```

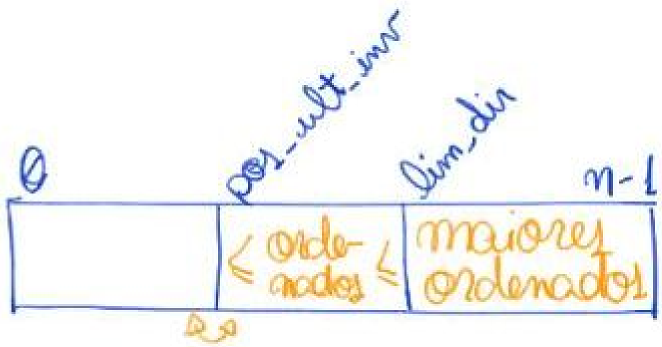
```
            if (v[i - 1] > v[i])
            {
                aux = v[i - 1];
                v[i - 1] = v[i];
                v[i] = aux;
            }
}
```

- Qual a eficiência de tempo de pior caso desse algoritmo?
    - E de melhor caso?
- Quando esse algoritmo para, o vetor está ordenado?
    - Sim, pois existem no máximo (n escolhe 2) = n(n - 1)/2 pares invertidos
        - e depois de n passagens desinvertendo pares adjacentes
            - todos estão em ordem.
    - Mais precisamente, note que uma passagem leva o maior elemento
        - para a última posição do vetor,
    - a segunda passagem leva o segundo maior para a penúltima posição,
        - e assim por diante.
    - Portanto, depois de n passagens todos os elementos estão ordenados.
        - Por motivo semelhante, depois de O(n^2) operações
            - a primeira versão do bubbleSort termina.
- Da observação sobre o maior elemento
    - decorre a melhoria do próximo algoritmo,
        - já que o laço interno não precisa passar pelas últimas posições
            - que contém os maiores elementos em ordem crescente.

```
void bubbleSort3(int v[], int n)
{
    int j, i, aux;
    for (j = 0; j < n; j++)
        for (i = 1; i < n - j; i++)
            if (v[i - 1] > v[i])
            {
                aux = v[i - 1];
                v[i - 1] = v[i];
                v[i] = aux;
            }
}
```

- Qual a eficiência de tempo de pior caso desse algoritmo?
    - E de melhor caso?

        - Elas só melhoraram por um fator constante.
- Será que podemos fazer melhor?
    - Observe que todos os elementos
        - depois da posição da última inversão já estão ordenados,
            - caso contrário teria ocorrido alguma inversão entre eles.
    - Além disso, eles correspondem aos maiores elementos do vetor,
        - como observado anteriormente.
    - Portanto, a passagem subsequente do laço interno não precisa passar
        - pelas posições após a última inversão da iteração anterior.



- Essa melhoria é implementada no próximo algoritmo.

```c
void bubbleSort4(int v[], int n)
{
    int j, i, aux, pos_ult_inv, lim_dir;
    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;
            }
        lim_dir = pos_ult_inv;
    }
}
```

Invariante e corretude:

- Os principais invariantes que valem
  - no início de cada iteração do laço externo são
    - o vetor é uma permutação do original,
    - v[lim_dir .. n - 1] está ordenado,
    - v[0 .. lim_dir - 1] <= v[lim_dir .. n - 1].
- Demonstrar que esses invariantes estão corretos,
  - verificando que eles valem antes da primeira iteração
    - e que seguem valendo de uma iteração para outra.
- Verificar que, no final do laço,
  - os invariantes implicam a corretude do algoritmo.

Eficiência de tempo:
- No melhor caso é O(n).
  - Ex.: vetor está ordenado ou tem apenas adjacentes fora de ordem.
- No pior caso é O(n^2).
  - Ex.: vetor está ordenado exceto pelo menor estar na última posição.

Estabilidade:
- Ordenação é estável.
  - Por que?
- O que acontece se trocarmos "v[i-1] > v[i]" por "v[i-1] >= v[i]"?
  - Continua ordenando?
  - Continua estável?
  - Sempre termina?

Eficiência de espaço:
- Ordenação é in place, pois só usa estruturas auxiliares
  - (e portanto memória) de tamanho constante em relação à entrada.

Quiz: Embora a melhor versão do bubbleSort tenha eficiência
- tanto no melhor quanto no pior caso assintoticamente iguais ao insertionSort,
  - este segundo algoritmo costuma ser mais rápido na prática.
    - Por que?

Curiosidade:
- É possível fazer uma versão ainda mais rápida do insertionSort,
  - usando uma iteração do bubbleSort.

Animação:
- Visualization and Comparison of Sorting Algorithms - www.youtube.com/watch?v=ZZuD6iUe3Pc