Ordenação por inserção (insertionSort) e por seleção (selectionSort), embaralhamento de Knuth

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.

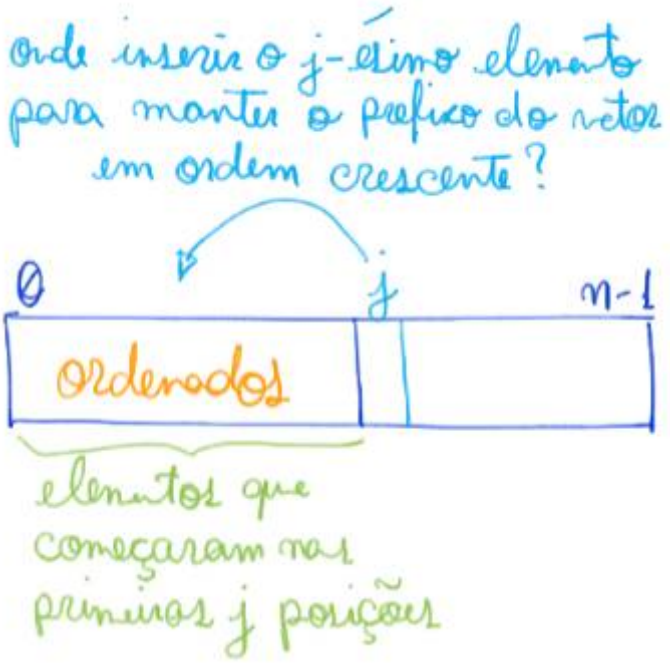
Para dois algoritmos básicos (e três avançados) veremos:

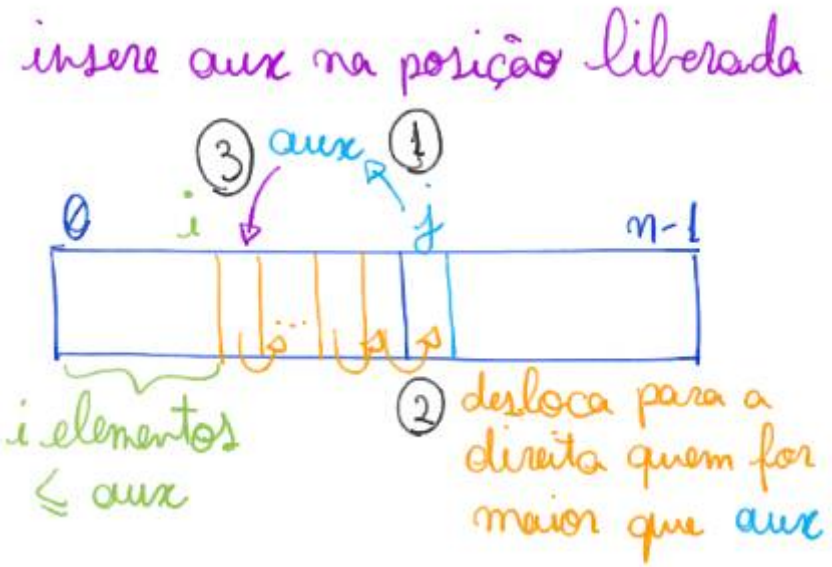
- Ideia e exemplo.
- Código.
- Invariantes e corretude.
- Eficiência de tempo: no melhor e pior casos.
- Estabilidade: algoritmo é estável
  - se não inverte a posição relativa de valores idênticos.
- Eficiência de espaço: algoritmo é in place se não usa estruturas auxiliares
  - (e portanto memória) 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





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;      // Quiz1: 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.
- Quiz2: 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 no melhor e no pior caso?

Estabilidade: Ordenação é estável, i.e., elementos com o mesmo valor

- tem sua ordem relativa preservada. Por que?
- Quiz3: O que acontece se trocarmos "aux < v[i]" por "aux <= v[i]"?
    - Continua ordenando?
    - Continua estável?

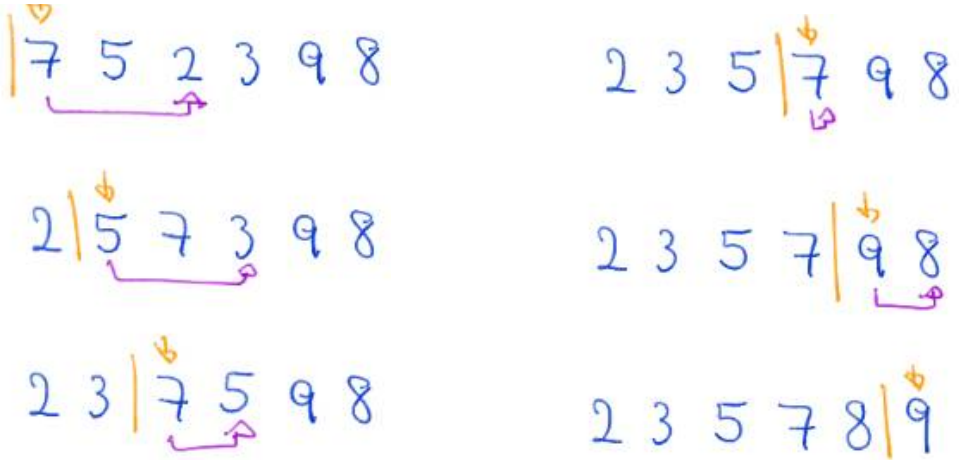
Eficiência de espaço: Ordenação é in place, pois só usa estruturas auxiliares

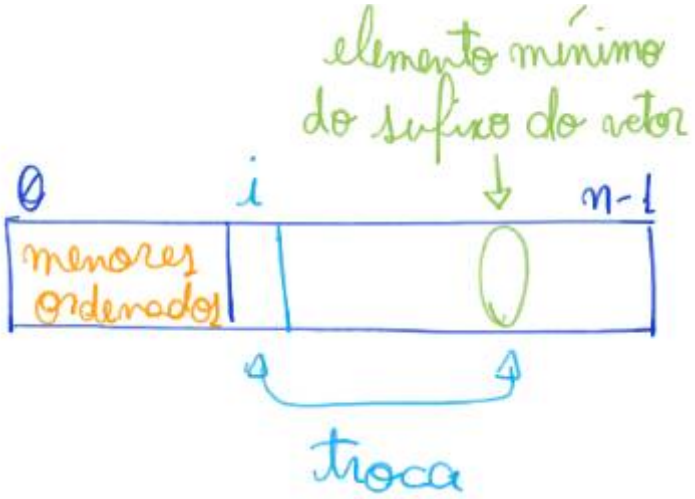
- (e portanto memória) de tamanho constante em relação à entrada.

Ordenação por Seleção (selectionSort)

Ideia e exemplo:

- Varre o vetor do início ao fim e, em cada iteração,
  - busca o mínimo do subvetor restante
    - e o coloca na posição corrente.
- Como exemplo, considere o vetor 7 5 2 3 9 8





Código:

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

- Quiz4: Por que o laço externo só vai até n - 2?
  - Daria problema ir até n - 1?

Invariante e corretude:

- Os invariantes do laço externo, que valem no início de cada iteração são:
  - o vetor é uma permutação do original,
  - v[0 .. i - 1] está ordenado,
  - v[i - 1] <= v[k], para i <= k < n.
- Invariante do laço interno, que vale no início de cada iteração:
  - v[ind_min] <= v[i .. j - 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:

- Em qualquer caso (melhor, médio, pior),
  - em cada iteração do laço externo
    - o laço interno percorre todo o subvetor restante
      - para encontrar o próximo mínimo.
- Por isso, o número total de iterações do laço interno do algoritmo é
  - n-1 + n-2 + n-3 + … + 3 + 2 + 1.
- Assim, o número de operações realizadas pelo algoritmo é da ordem de
  - n(n - 1) / 2 ~= n^2 / 2 = O(n^2).

Estabilidade: Ordenação não é estável.

- Isso porque as trocas do mínimo com a posição corrente
  - podem levar à inversão da ordem relativa entre elementos iguais.
- Como exemplo, considere o vetor [2 2 1 3 4 5 6 7].
  - Observe que a inversão não envolve o mínimo,
    - mas o elemento que está sendo trocado com ele.

Eficiência de espaço: Ordenação é in place, pois só usa estruturas auxiliares

- (e portanto memória) de tamanho constante em relação à entrada, i.e., O(1).

Bônus/Quiz5: Observe que, se soubéssemos encontrar o mínimo
  - do subvetor restante sem precisar percorrê-lo linearmente,
- apenas trocar tal mínimo com o elemento da posição corrente
  - leva tempo constante.
- Essa ideia geraria um algoritmo mais eficiente?
  - Esse algoritmo funciona? Isto é, ele está correto?
- Note também que, podemos propor uma variante deste algoritmo
  - que varre o vetor do fim para o começo e
    - seleciona o máximo do subvetor restante a cada iteração.

Quiz6/Curiosidade: É possível fazer uma versão mais rápida do insertionSort,

- usando uma iteração do selectionSort. O que ganhamos e o que perdemos?

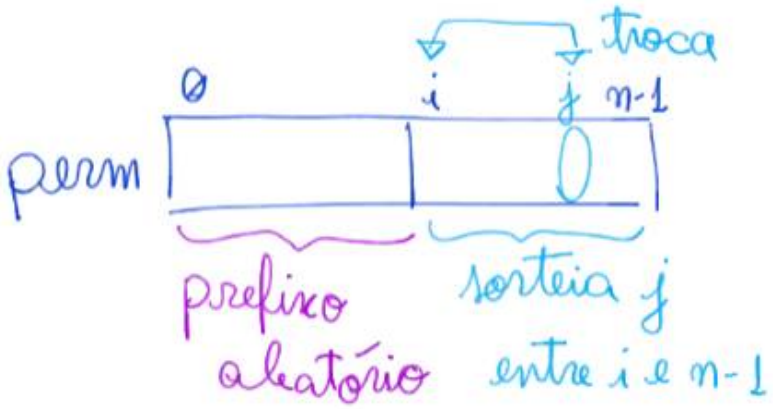
Animações: Visualization and Comparison of Sorting Algorithms - www.youtube.com/watch?v=ZZuD6iUe3Pc

(Bônus) Embaralhamento de Knuth

- Algoritmo do famoso Donald Knuth para produzir uma permutação aleatória.

Ideia: Dada uma permutação em um vetor,

  - percorrer o vetor da esquerda para a direita
- e em cada iteração escolher uniforme e aleatoriamente
  - um elemento do sufixo do vetor
    - para trocar com o elemento da posição corrente.



Código

```
    // ordem aleatória - Knuth shuffle
    for (i = 0; i < n; i++)
        v[i] = i;
    for (i = 0; i < n; i++) {
        // número pseudoaleatório entre 0 e n - i - 1
        desloc = (int)((double)rand()
           / ((double)RAND_MAX + 1) * (n - i));
        aux = v[i + desloc];
        v[i + desloc] = v[i];
        v[i] = aux;
    }
```

Invariante e corretude:

- No início de cada iteração do laço
  - v[0 .. n - 1] é uma permutação do vetor original,
  - v[0 .. i - 1] é um prefixo escolhido com prob. 1 / (n! / (n - i)!).
- Ao final das iterações v[0 .. n - 1] é uma permutação
  - escolhida com probabilidade 1 / n!
- sendo que n! é o número total de permutações com n elementos.

Eficiência de tempo: O(n).

Eficiência de espaço: O(1).

Destaco que, permutações aleatórias são úteis para

- testar empiricamente o comportamento de caso médio dos algoritmos,
  - especialmente porque este costuma ser mais difícil de analisar
    - do que pior e melhor casos.