

AED1 - Aula 21

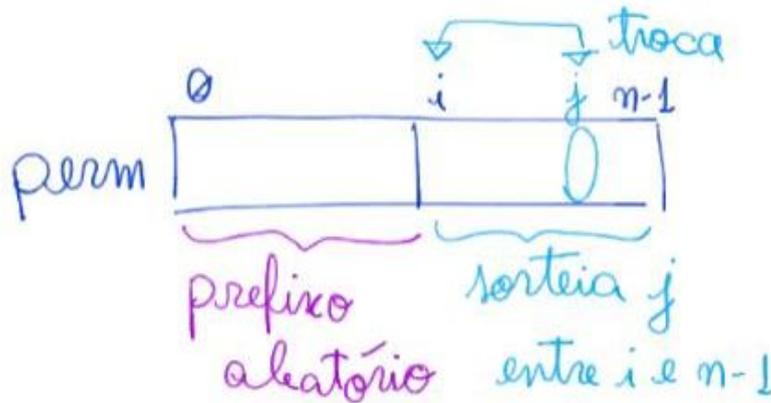
Embaralhamento de Knuth, melhoria no insertionSort, Ordenação por seleção (selectionSort)

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
 - $v[0 \dots n - 1]$ é uma permutação do vetor original,
 - $v[0 \dots i - 1]$ é um prefixo escolhido com prob. $1 / (n! / (n - i)!)$.
- Ao final das iterações $v[0 \dots n - 1]$ é uma permutação
 - escolhida com probabilidade $1 / n!$

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
 - o que pior e melhor caso.

Melhoria no insertionSort

- Inspirado no livro Algorithms in C++, Parts 1-4 de R. Sedgwick.

Relembrando o algoritmo de ordenação por inserção

```
void insertionSort1(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?
    }
}
```

- Observe que, na condição de parada do laço interno
 - temos dois testes, sendo um apenas para evitar sair fora do vetor.
- Existe uma maneira interessante de evitar o teste “ $i \geq 0$ ”.
 - Dado que o outro teste é “ $aux < v[i]$ ”,
 - basta colocar o menor elemento do vetor na primeira posição.
- Para tanto, podemos usar a ideia do laço interno do algoritmo bubbleSort,
 - mas indo do fim para o começo do vetor.
- Essa é a ideia implementada no seguinte algoritmo.

```

void insertionSort2(int v[], int n) {
    int i, j, aux;
    for (i = n - 1; i > 0; i--)
        if (v[i - 1] > v[i])
        {
            aux = v[i];
            v[i] = v[i - 1];
            v[i - 1] = aux;
        }
    for (j = 2; j < n; j++)
    {
        aux = v[j];
        for (i = j - 1; aux < v[i]; i--)
            v[i + 1] = v[i]; // desloca à direita os maiores
        v[i + 1] = aux;      // por que i+1?
    }
}

```

- Dado que fazemos uma “iteração do laço interno do bubbleSort”,
 - observe que a posição em que este realiza a última troca
 - delimita um prefixo ordenado que contém
 - os menores elementos do vetor.
- Assim, podemos começar o laço típico do insertionSort
 - a partir desta posição.
- Essa é a ideia implementada no seguinte algoritmo.

```

void insertionSort3(int v[], int n) {
    int i, j, aux;
    j = n - 1;
    for (i = n - 1; i > 0; i--)
        if (v[i - 1] > v[i])
        {
            aux = v[i];
            v[i] = v[i - 1];
            v[i - 1] = aux;
            j = i; // guarda a posição da troca
        }
    for (j++; j < n; j++)

```

```

{
    aux = v[j];
    for (i = j - 1; aux < v[i]; i--)
        v[i + 1] = v[i]; // desloca à direita os maiores
    v[i + 1] = aux;      // por que i+1?
}
}

```

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

↓
 | 7 5 2 3 9 8
 └──────────┘

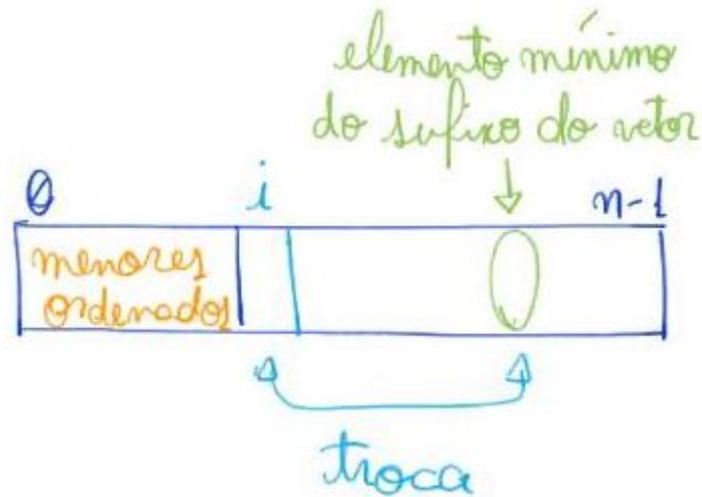
↓
 2 | 5 7 3 9 8
 └────────┘

↓
 2 3 | 7 5 9 8
 └───┘

↓
 2 3 5 | 7 9 8
 └──┘

↓
 2 3 5 7 | 9 8
 └──┘

↓
 2 3 5 7 8 | 9



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;
    }
}
```

Invariante e corretude:

- Os invariantes do laço externo, que valem no início de cada iteração são:
 - o vetor é uma permutação do original,
 - $v[0 \dots i - 1]$ está ordenado,
 - $v[i - 1] \leq v[k]$, para $i \leq k < n$.
- Invariante do laço interno, que vale no início de cada iteração:
 - $v[ind_min] \leq v[i \dots j - 1]$.
- 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
 - $n-1 + n-2 + n-3 + \dots + 3 + 2 + 1$.
- Assim, o número de operações realizadas pelo algoritmo é da ordem de
 - $n(n-1)/2 \approx n^2/2 = O(n^2)$.

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.

Bônus/quiz:

- 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.

Animações:

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