Aula 9
Ordenação
Algoritmos Elementares

Algoritmos e Estruturas de Dados

Ordenação

Algoritmos elementares

Ordenação

- Objectivo
  - Dado um array de elementos
  - Organizar o array usando uma determinada ordem

  - Ordenar qualquer tipo de dados

Ordenação e *arrays*

- Algoritmos de ordenação são muito eficientes a trabalhar com arrays

- Pq?

Ordenação e *arrays*

- Algoritmos de ordenação são muito eficientes a trabalhar com *arrays*
- Na maior parte dos casos, não precisam de aumentar o tamanho do *array*
- Trabalham principalmente com trocas de elementos dentro do *array*
  - *Trocas de elementos 2 a 2 (não precisamos de fazer shift)*

Algoritmos de Ordenação Genéricos

- Utilização de Interface Comparable<T>

```
public interface Comparable<Item>
{
    public int compareTo(Item that);
}
```

Ordem total

- Uma relação de ordem total é uma relação binária ≤ que satisfaz:
  - Antisimetria: se a ≤ b e b ≤ a, então a = b
  - Transitividade: se a ≤ b, e b ≤ c, então a ≤ c
  - Totalidade: a ≤ b ou b ≤ a ou ambas são verdade

- Ex:
  - ordem númerica para números naturais e reais
  - ordem cronológica para datas e tempo
  - ordem alfabética para *strings*

Classe Sort

- Métodos de ordenação podem ser implementados através de 2 operadores principais
  - Comparação de 2 elementos
  - Troca de 2 elementos

```
public class Sort {
    protected static boolean less(Comparable v, Comparable w)
    {
        return v.compareTo(w) < 0;
    }
    protected static void exchange(Comparable[] a, int i, int j)
    {
        Comparable t = a[i];
        a[i] = a[j];
        a[j] = t;
    }
    public static boolean isSorted(Comparable[] a)
    {
        for (int i = 1; i < a.length; i++)
        {
            if (less(a[i],a[i-1])) return false;
        }
        return true;
    }
```

usado em testes para verificar a ordenação

Classe Sort

- A classe Sort apresentada é aquilo a que em Java se chama uma utility class ou library class.

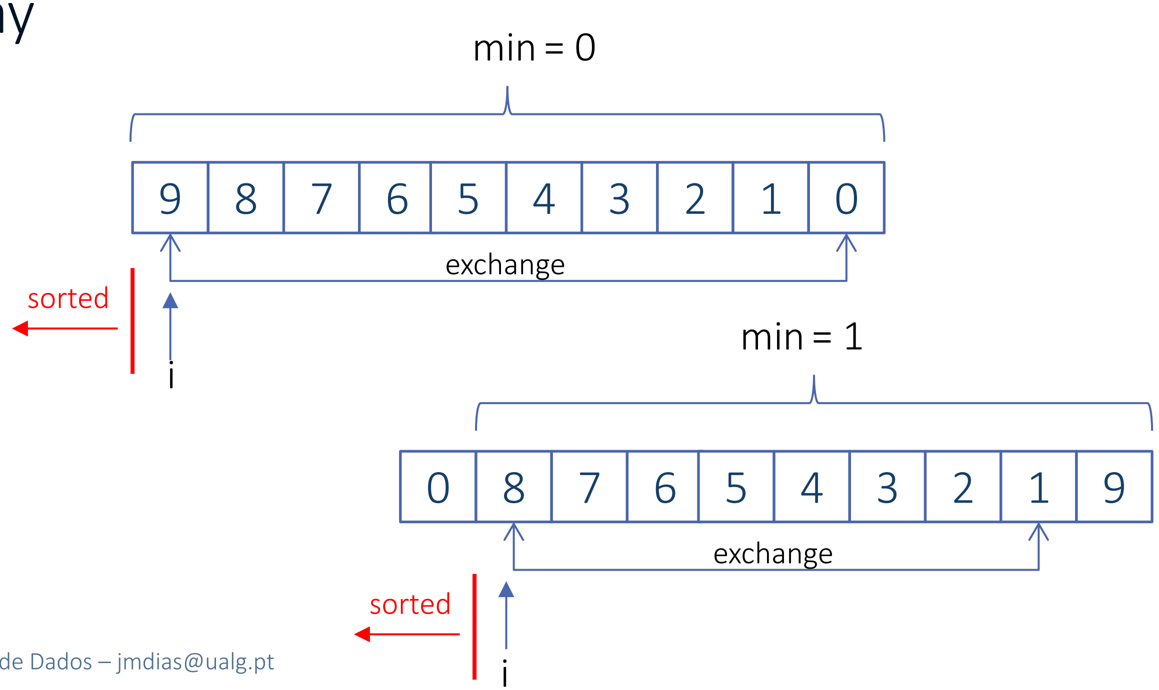
- Classe que corresponde a um conjunto de métodos estáticos que estão relacionados

- Não vou criar instâncias (new) desta classe, simplesmente vou usá-la para aceder aos métodos pretendidos

Selection Sort

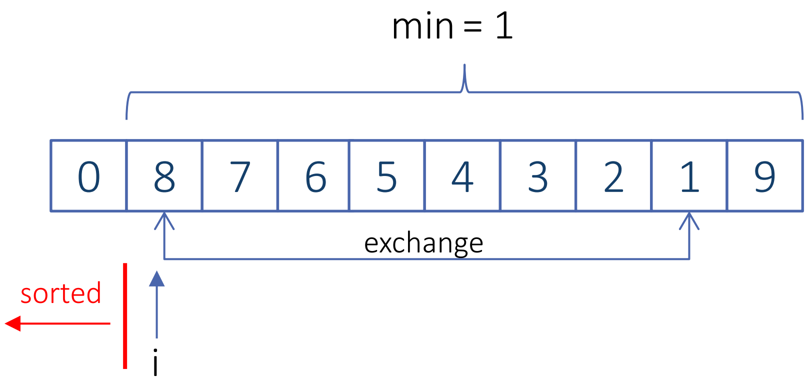
Selection Sort

- Ideia muito simples e ingénua
  - Colocar na 1.ª posição do array o menor elemento
  - Colocar na 2.ª posição do array o menor elemento do resto do array
  - ...



Selection Sort

```
public class SelectionSort extends Sort
{
    public static void sort(Comparable[] a)
    {
        int n = a.length;
        int minIndex;
        for(int i = 0; i < n; i++)
        {
            minIndex = i;
            for(int j = i+1; j < n; j++)
            {
                if (less(a[j], a[minIndex]))
                {
                    minIndex = j;
                }
            }
            exchange(a, i, minIndex);
        }
    }
}
```



Complexidade temporal

- Análise por modelos matemáticos (notação tilde)
- Vamos analisar frequência das operações *compare* e *exchange*
- *Interior 2.º for é executado*

  (n – 1) + (n – 2) + ... + 2 + 1 + 0

Progressão aritmética

$a_n = a_{n-1} + r$

Soma dos n primeiros termos

$$S_n = \frac{n}{2}(a_1 + a_n)$$

```
for(int i = 0; i < n; i++)
{
    minIndex = i;
    for(int j = i+1; j < n; j++)
    {
        if (less(a[j], a[minIndex]))
        {
            minIndex = j;
        }
    }
    exchange(a, i, minIndex);
}
```

Complexidade temporal

- Análise por modelos matemáticos (notação tilde)
- Vamos analisar frequência das operações *compare* e *exchange*
- *Interior 2.º for é executado*

  $(n-1) + (n-2) + \ldots + 2 + 1 + 0$

Progressão aritmética

$a_n = a_{n-1} + r$

Soma dos n primeiros termos

$$S_n = \frac{n}{2}(a_1 + a_n)$$

$$= \frac{n}{2}((n-1)+0) = \frac{n^2}{2} - \frac{n}{2} \sim \frac{n^2}{2}$$

```
for(int i = 0; i < n; i++)
{
    minIndex = i;
    for(int j = i+1; j < n; j++)
    {
        if (less(a[j], a[minIndex]))
        {
            minIndex = j;
        }
    }
    exchange(a, i, minIndex);
}
```

Complexidade temporal

- *Interior 2.º for é executado*

$(n-1) + (n-2) + \ldots + 2 + 1 + 0$

$$= \frac{n}{2}((n-1)+0) = \frac{n^2}{2} - \frac{n}{2} \sim \frac{n^2}{2}$$

```
for(int i = 0; i < n; i++)
{
    minIndex = i;
    for(int j = i+1; j < n; j++)
    {
        if (less(a[j], a[minIndex]))
        {
            minIndex = j;
        }
    }
    exchange(a, i, minIndex);
}
```

SelectionSort	freq.	O
less/compare	$\sim n^2/2$	$O(n^2)$
exchange	$\sim n$	

Análise usando notação tilde

Análise assintótica

Insertion Sort

Insertion Sort

- Ordenar o array como se estivessemos a ordenar a nossa mão num jogo de cartas
  - Mas só podemos mover uma carta uma posição de cada vez

mão ordenada

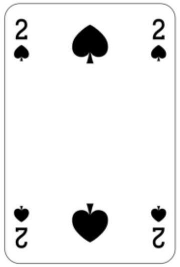
resto do array, não ordenado

Insertion Sort

- 1) Começar com 2 cartas na mão
- 2) Ordenar a mão
  - Da direita para esquerda

    se houver duas cartas adjacentes fora de ordem, trocá-las

    avançar para a próxima carta à esquerda
- 3) Aumentar tamanho da mão
  - Adicionar próxima carta do lado direito da mão
- 4) Repetir a partir de 2) até não existirem mais cartas a ordenar

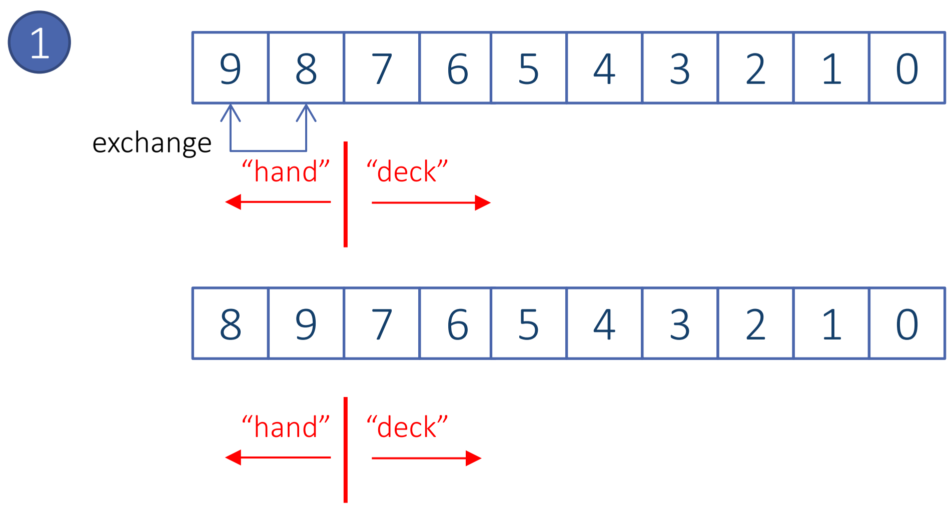

resto do array, não ordenado

mão ordenada

nova carta

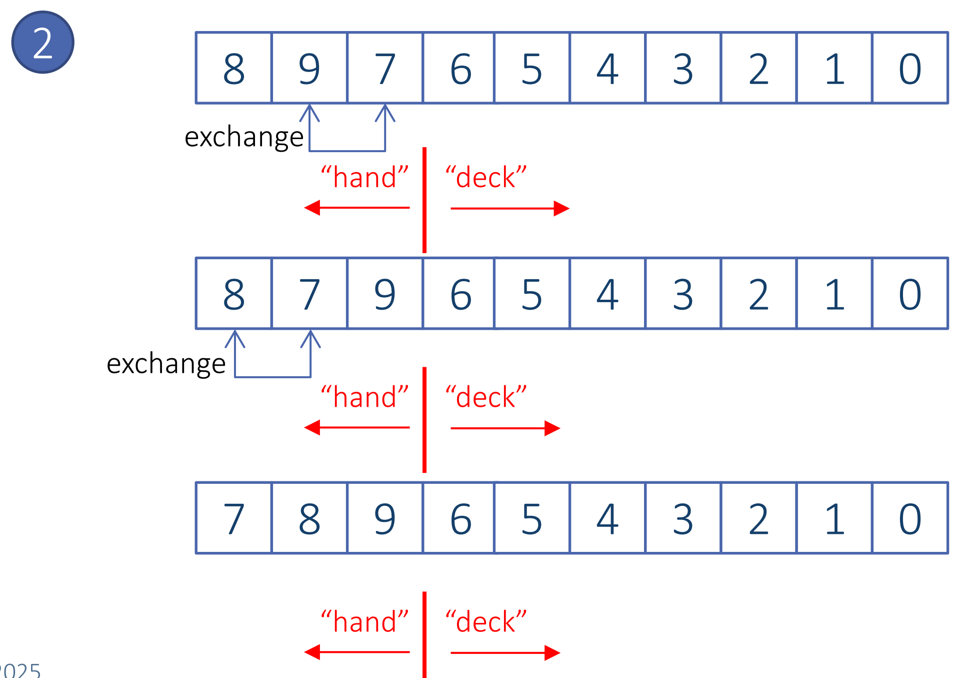
Insertion Sort

- Ideia, ordenar os elementos como se estivessemos a ordenar a nossa mão num jogo de cartas
  - Ordenar a mão (ordenando da direita para esquerda, “carta” a “carta”)
  - Aumentar o tamanho da mão, e repetir



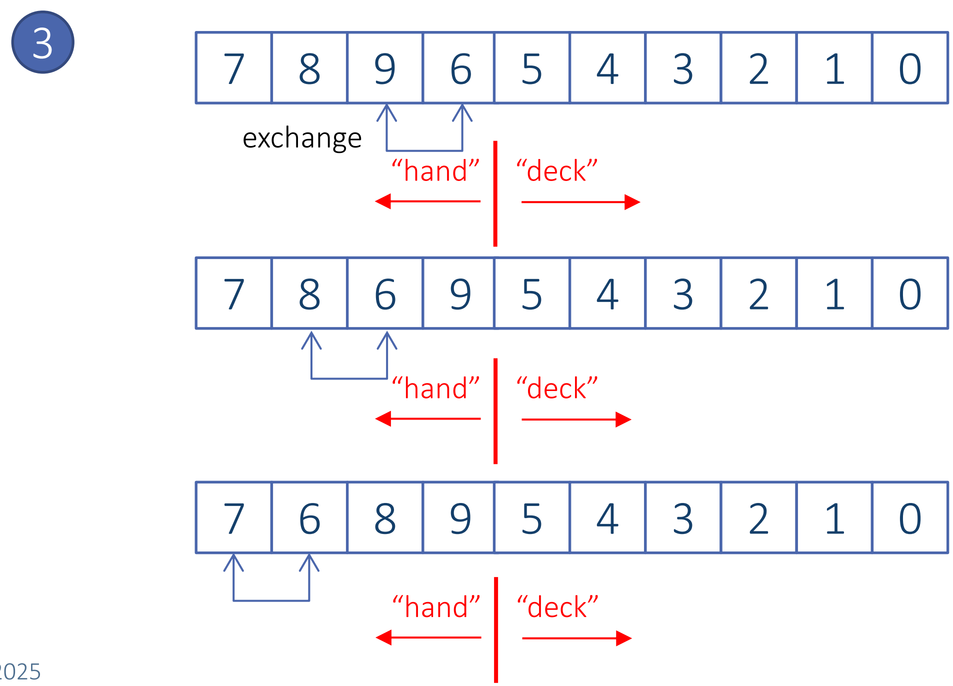
Insertion Sort

- Ideia, ordenar os elementos como se estivessemos a ordenar a nossa mão num jogo de cartas
  - Ordenar a mão (ordenando da direita para esquerda, “carta” a “carta”)
  - Aumentar o tamanho da mão, e repetir



Insertion Sort

- Ideia, ordenar os elementos como se estivessemos a ordenar a nossa mão num jogo de cartas
  - Ordenar a mão (ordenando da direita para esquerda, “carta” a “carta”)
  - Aumentar o tamanho da mão, e repetir



Insertion Sort

```
public static void sort(Comparable[] a)
{
    int n = a.length;
    for(int i = 1; i < n; i++)
    {
        for(int j = i; j > 0 ; j--)
        {
            if(less(a[j],a[j-1]))
            {
                exchange(a, j,j-1);
            }
            else break;
        }
    }
}
```

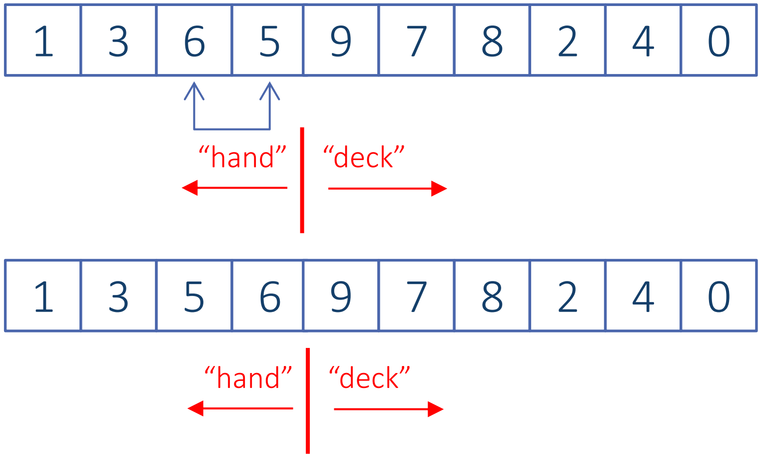
Começamos por i=1 pois queremos ter pelo menos 2 cartas inicialmente

“aumentar a mão”

“ordenar a mão”, da direita para a esquerda

podemos sair mais cedo, caso não exista troca

pois isto quer dizer que a mão já está ordenada



Complexidade Temporal

- A complexidade do insertion sort varia muito de acordo com o array dado
  - Melhor caso

    array já se encontra ordenado
  - Pior caso

    array encontra-se ordenado pela ordem contrária
  - Caso médio

    array está baralhado de forma aleatória

Complexidade Temporal

- *No pior caso*
  - *teste less retorna sempre true*
  - *2.º for*

$$\overbrace{1 + 2 + \ldots + (n-2) + (n-1)}^{\text{n-1}}$$

```java
public static void sort(Comparable[] a)
{
    int n = a.length;
    for(int i = 1; i < n; i++)
    {
        for(int j = i; j > 0 ; j--)
        {
            if(less(a[j],a[j-1]))
            {
                exchange(a, j,j-1);
            }
            else break;
        }
    }
}
```

Complexidade Temporal

- *No pior caso*
  - *2.º for*

$$\overbrace{1 + 2 + \ldots + (n-2) + (n-1)}^{\text{n-1}}$$

$$= \frac{n-1}{2}(1 + n - 1)$$

$$= \frac{n^2-n}{2}$$

$$\sim \frac{n^2}{2}$$

Progressão aritmética

$a_n = a_{n-1} + r$

Soma dos n primeiros termos

$$S_n = \frac{n}{2}(a_1 + a_n)$$

```java
public static void sort(Comparable[] a)
{
    int n = a.length;
    for(int i = 1; i < n; i++)
    {
        for(int j = i; j > 0 ; j--)
        {
            if(less(a[j],a[j-1]))
            {
                exchange(a, j,j-1);
            }
            else break;
        }
    }
}
```

Complexidade Temporal

- *No melhor caso (array já ordenado)*
  - *teste less falha sempre*

- *2.º for*

$$\overbrace{1 + 1 + \ldots + 1 + 1}^{n-1}$$

$$= n-1$$

$$\sim n$$

```
public static void sort(Comparable[] a)
{
    int n = a.length;
    for(int i = 1; i < n; i++)
    {
        for(int j = i; j > 0 ; j--)
        {
            if(less(a[j],a[j-1]))
            {
                exchange(a, j,j-1);
            }
            else break;
        }
    }
}
```

não precisamos de fazer trocas, e executamos sempre o break.

implica que o for interno irá executar sempre apenas uma iteração

Complexidade Temporal

- *Array aleatório*
  - *teste less falha depois de termos testado, em média, metade dos elementos*
  - *2.º for*

$$\frac{1}{2}+\frac{2}{2}+\cdots+\frac{n-2}{2}+\frac{n-1}{2}$$
$$=\frac{n}{2}\left(\frac{1}{2}+\frac{n-1}{2}\right)=\frac{n}{2}\left(\frac{n}{2}\right)$$
$$\sim\frac{n^2}{4}$$

```
public static void sort(Comparable[] a)
{
    int n = a.length;
    for(int i = 1; i < n; i++)
    {
        for(int j = i; j > 0 ; j--)
        {
            if(less(a[j],a[j-1]))
            {
                exchange(a, j,j-1);
            }
            else break;
        }
    }
}
```

Complexidade Temporal

```java
public static void sort(Comparable[] a)
{
    int n = a.length;
    for(int i = 1; i < n; i++)
    {
        for(int j = i; j > 0 ; j--)
        {
            if(less(a[j],a[j-1]))
            {
                exchange(a, j,j-1);
            }
            else break;
        }
    }
}
```

InsertionSort	Best case	Worst Case	Aleatório	O
less/compare	~n	~$n^2/2$	~$n^2/4$	$O(n^2)$
exchange	*0*	~$n^2/2$	~$n^2/4$	

Complexidade temporal

- Embora no pior caso $T(n) = O(n^2)$
- Insertion sort porta-se muito bem **~n** quando:
  - Arrays parcialmente ordenado, com poucos elementos fora do sítio
  - Um array pequeno adicionado a um grande array já ordenado
  - Array onde cada elemento não está muito longe de onde deveria estar
- Estas propriedades verificam-se com alguma frequência