

Análise da Complexidade VI

Joaquim Madeira

06/04/2021

Sumário

- Recap
- Shell Sort
- Implementação genérica
- Sugestão de leitura

Let's
RECAP

Recapitulação

Selection Sort

- Número fixo de **comparações** :

$$C(n) \approx \frac{n^2}{2}$$

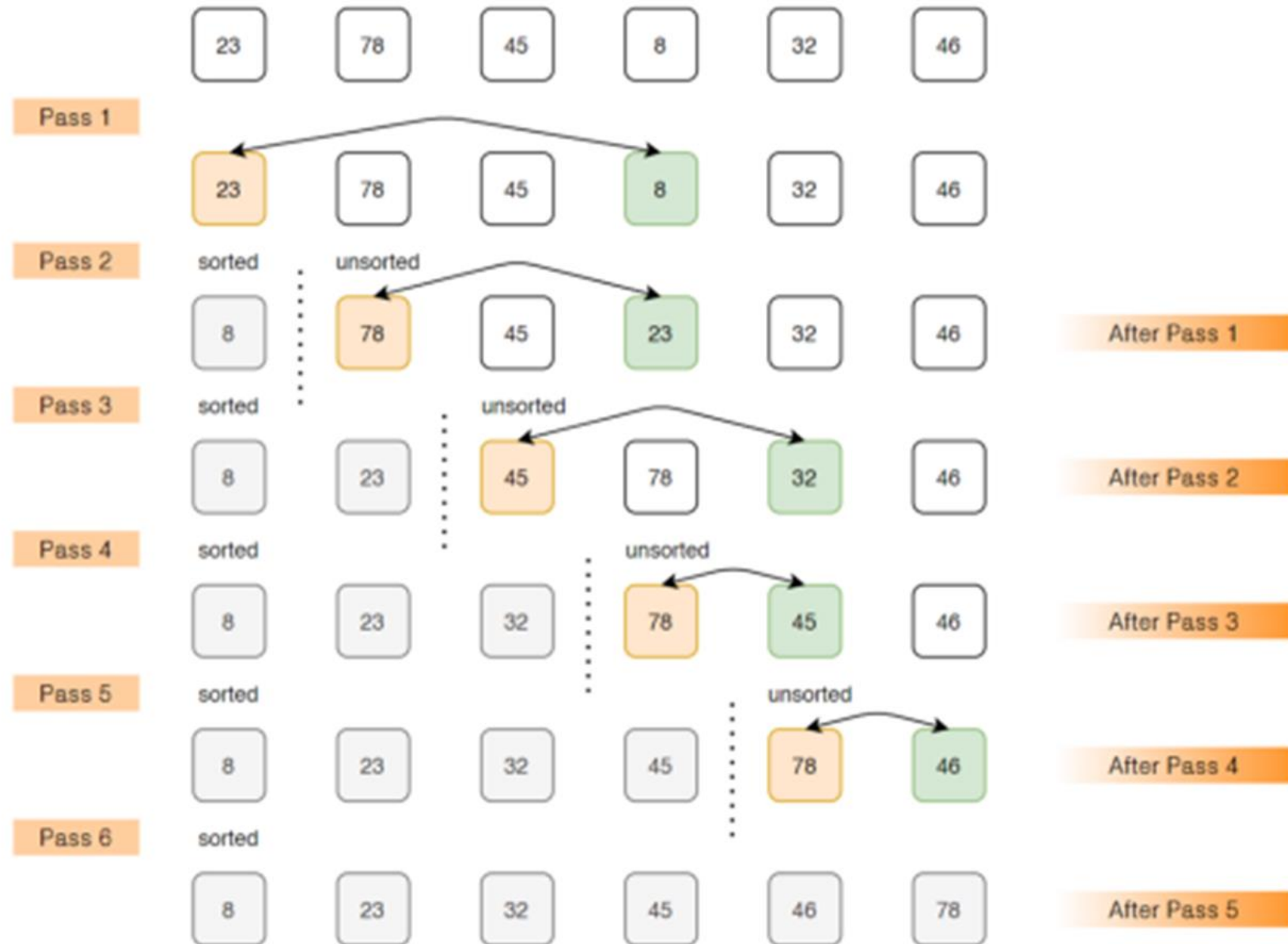
- **Trocas** :

$$W_T(n) = n - 1$$

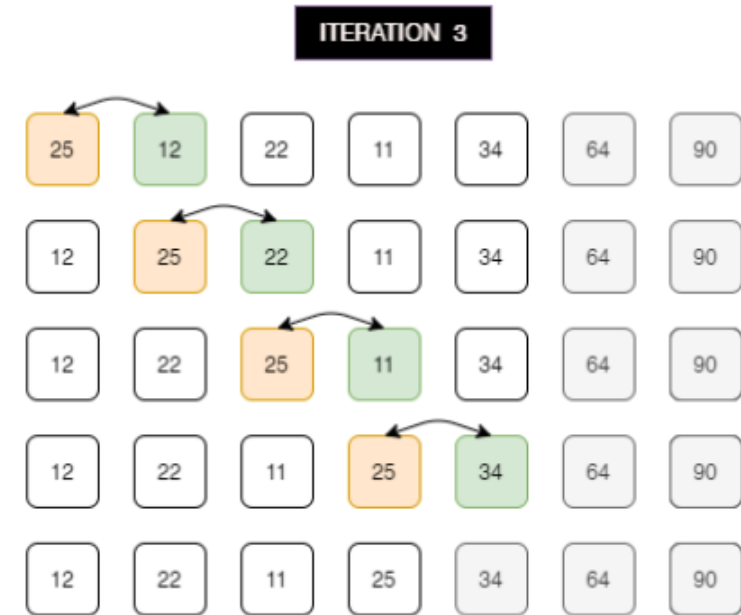
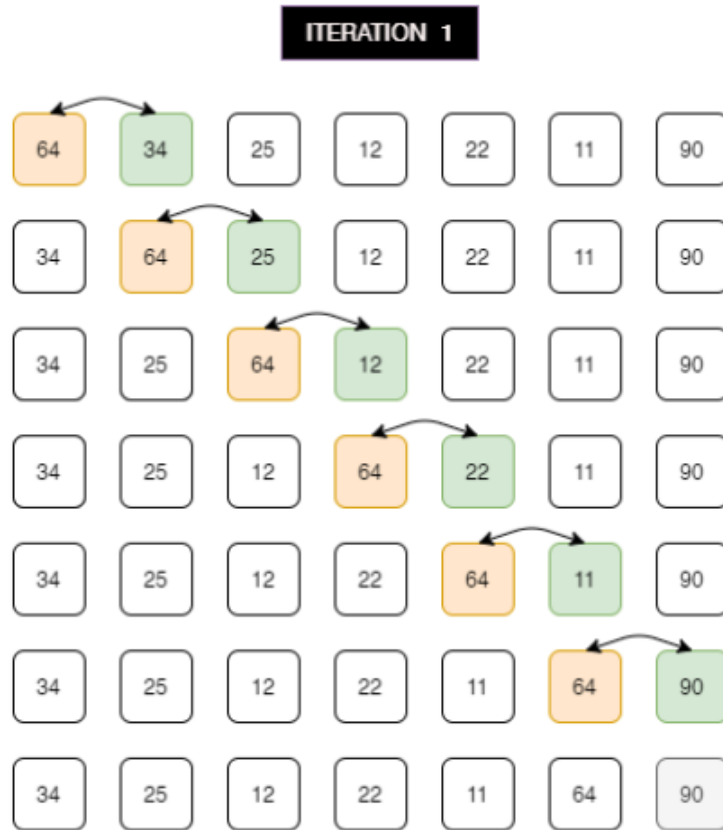
$$A_T(n) \approx n - \ln n$$

$$B_T(n) = 0$$

[Adwiteeya Reyna]



Bubble Sort



[Adwiteeya Reyna]

Bubble Sort

ITERATION 4



ITERATION 7



ITERATION 5



ITERATION 6



- **Comparações :**

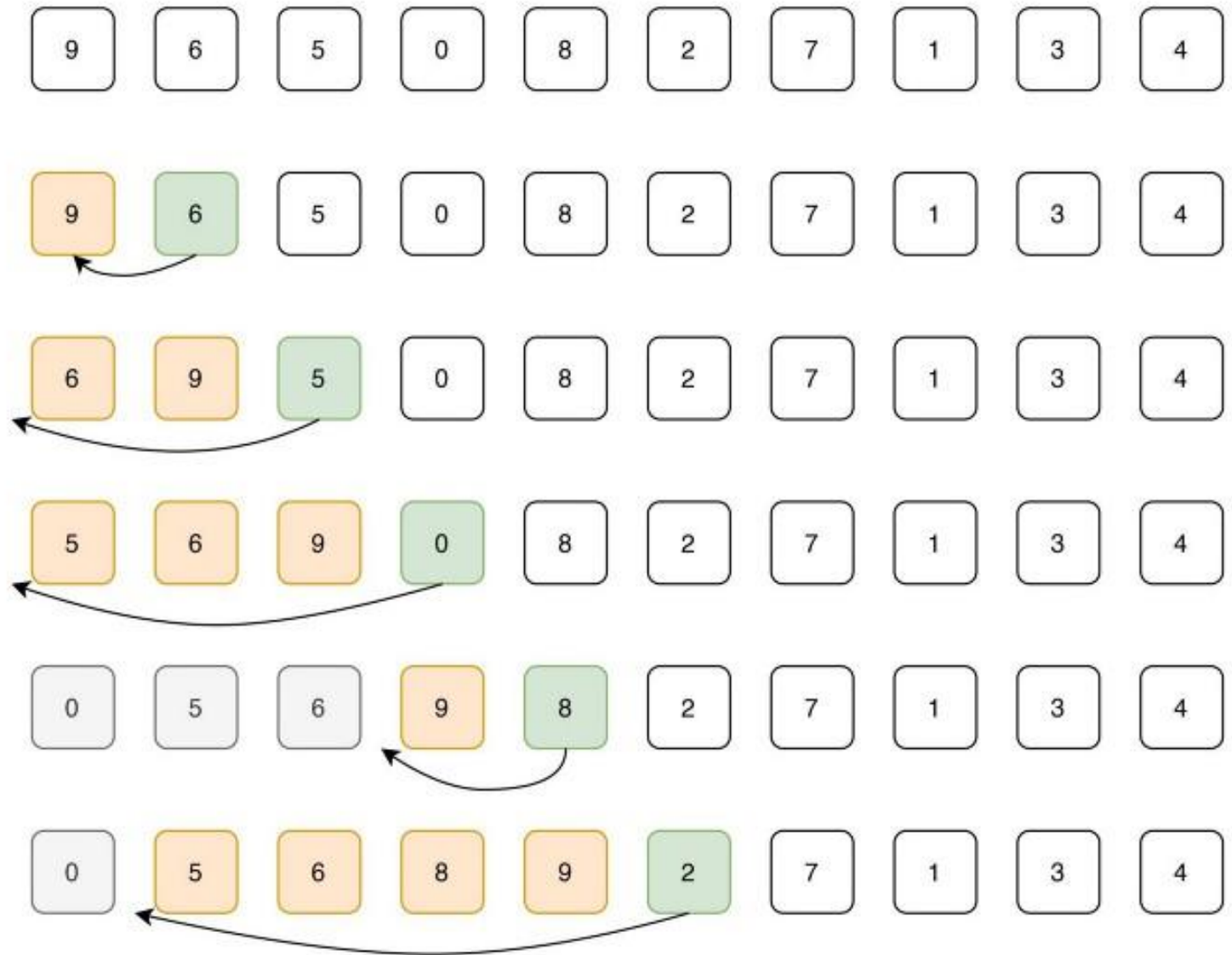
$$W_C(n) \approx \frac{n^2}{2} \quad A_C(n) \approx \frac{n^2}{3} \quad B_C(n) = n - 1$$

- **Trocas :**

$$W_T(n) = W_C(n) \quad A_T(n) \approx \frac{n^2}{6} \quad B_T(n) = 0$$

[Adwiteeya Reyna]

Insertion Sort



[Adwiteeya Reyna]

Insertion Sort

- **Comparações :**

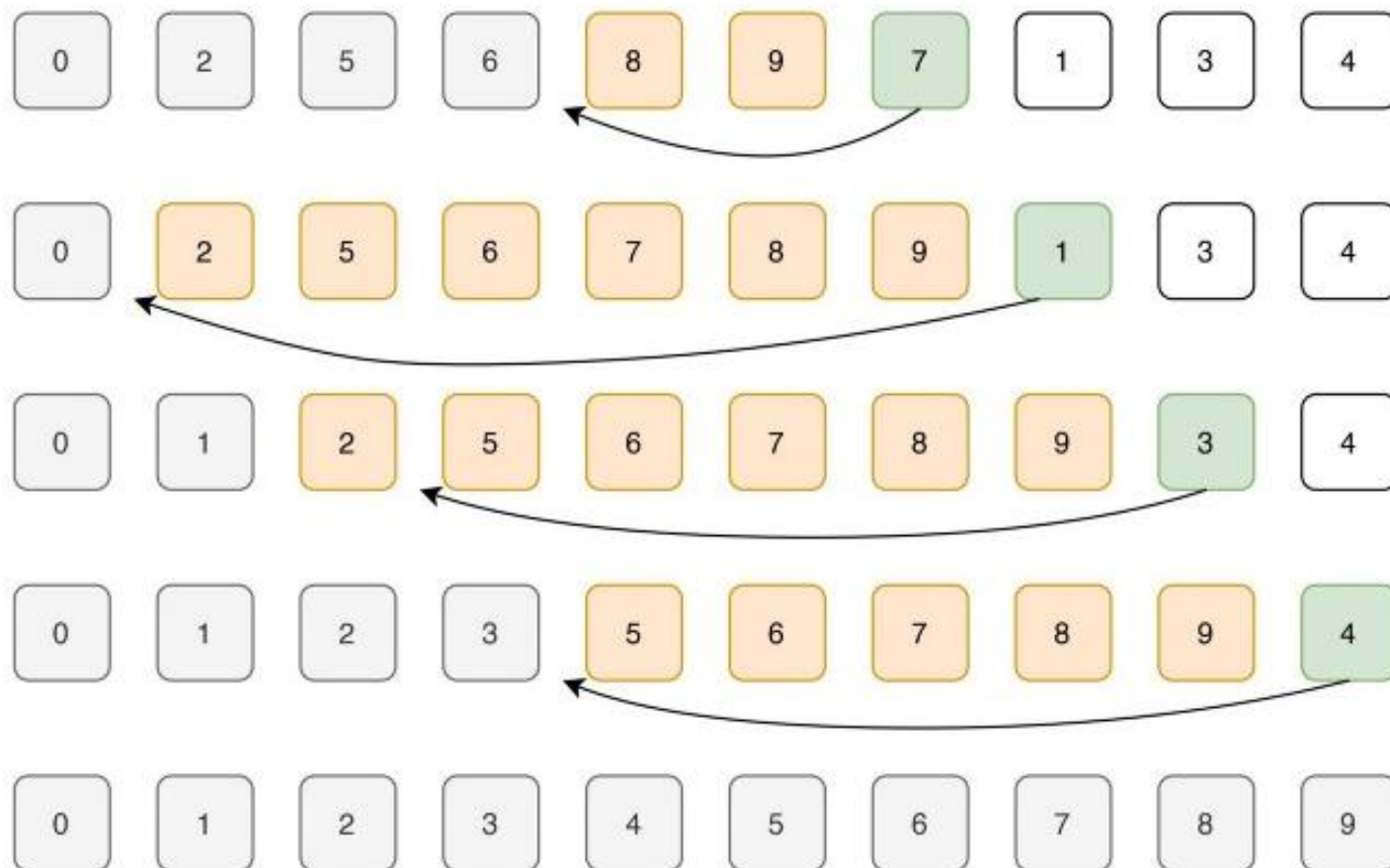
$$W_C(n) \approx \frac{n^2}{2} \quad A_C(n) \approx \frac{n^2}{4}$$

$$B_C(n) = n - 1$$

- **Deslocamentos :**

$$W_D(n) \approx \frac{n^2}{2} \quad A_D(n) \approx \frac{n^2}{8}$$

$$B_D(n) = 0$$



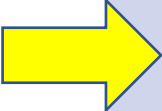
[Adwiteeya Reyna]

Tarefa 1

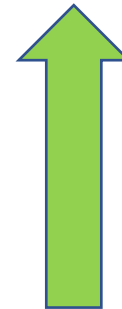
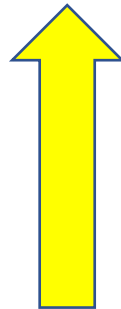
- toptal.com/developers/sorting-algorithms
- Analisar as **animações** disponibilizadas
- Comparar :
 - **Diferentes arrays** para um **mesmo algoritmo**
 - O **mesmo array** para **diferentes algoritmos**




Comparações – Algoritmos Quadráticos



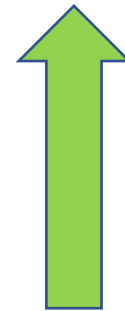
	Pior Caso	Caso Médio	Melhor Caso
Selection Sort	$\approx \frac{n^2}{2}$	$\approx \frac{n^2}{2}$	$\approx \frac{n^2}{2}$
Bubble Sort	$\approx \frac{n^2}{2}$	$\approx \frac{n^2}{3}$	$n - 1$
Insertion Sort	$\approx \frac{n^2}{2}$	$\approx \frac{n^2}{4}$	$n - 1$



Trocas / Deslocamentos



	Pior Caso	Caso Médio	Melhor Caso
Selection Sort	$n - 1$	$\approx n - \ln n$	0
Bubble Sort	$\approx \frac{n^2}{2}$	$\approx \frac{n^2}{6}$	0
Insertion Sort	$\approx \frac{n^2}{2}$	$\approx \frac{n^2}{8}$	0



Bubble Sort – Testes computacionais

- Arrays Ordenados

n	# Comparações	Rácio	# Atribuições
2500	2499		0
5000	4999	2.000	0
10000	9999	2.000	0
20000	19999	2.000	0

Bubble Sort – Testes computacionais

- Arrays por Ordem Inversa

n	# Comparações	Rácio	# Atribuições	Rácio
2500	3123750		9371250	
5000	12497500	4.001	37492500	4.001
10000	49995000	4.000	149985000	4.000
20000	199990000	4.000	599970000	4.000

Bubble Sort – Testes computacionais

- Arrays Aleatórios

n	# Comparações	Rácio	# Atribuições	Rácio
2500	3119285		4536945	
5000	12496939	4.006	18496980	4.077
10000	49993515	4.000	74646285	4.036
20000	199969699	4.000	300555000	4.026

- Valores mais elevados do que os obtidos pela análise formal !!
- Cenário considerado é demasiado simples...

Tarefa 2

- Fazer **testes computacionais** idênticos para os **outros algoritmos**
- Ficheiros com os **dados de teste** estão disponíveis no **Moodle**

Shell Sort

Shell Sort

- Donald Shell, 1959
- **Generalização** do Insertion Sort
- Começar por **ordenar** pares de **elementos distantes entre si**
- Repetir, **reduzindo a distância** entre os elementos comparados
- O primeiro algoritmo a **“quebrar a barreira quadrática”**
- **ATENÇÃO** : o desempenho depende da escolha da **sequência de distâncias** a usar

Ideia

- Considerar **subconjuntos de elementos** do array, mas **distantes** entre si
- Ordenar cada um desses subconjuntos – **Ordenação por Inserção**
- **Repetir**, diminuindo a distância entre os elementos que constituem cada subconjunto
 - Em cada um destes passos, **o array fica mais “próximo” da ordenação final**
- No último passo, considerar todos os elementos do array
- Como **inicializar** e fazer **diminuir a distância** entre elementos de cada subconjunto ?
 - Por **exemplo**, $(n \div 2)$, $(n \div 4)$, \dots , 1 ---- a sequência proposta por D. Shell
 - Há outras sequências “melhores” !
- Algoritmo **in-place** !!

Exemplo

0	1	2	3	4
7	2	6	4	3

Exemplo – 1º passo

0	1	2	3	4
7	2	6	4	3

7
6
3

2
4

1º passo

$h = 5 \text{ div } 2 = 2$

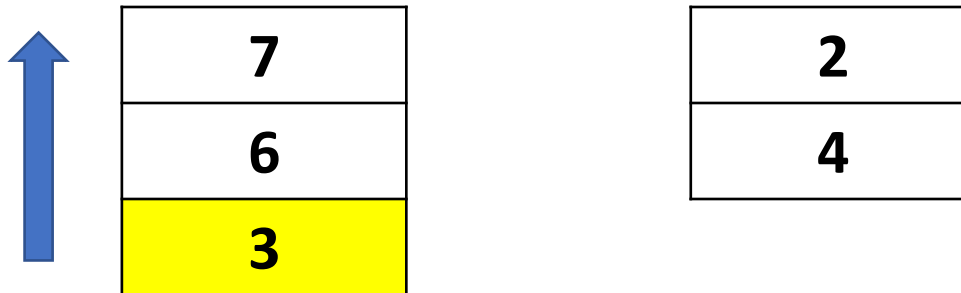
2 subconjuntos

distância = 2

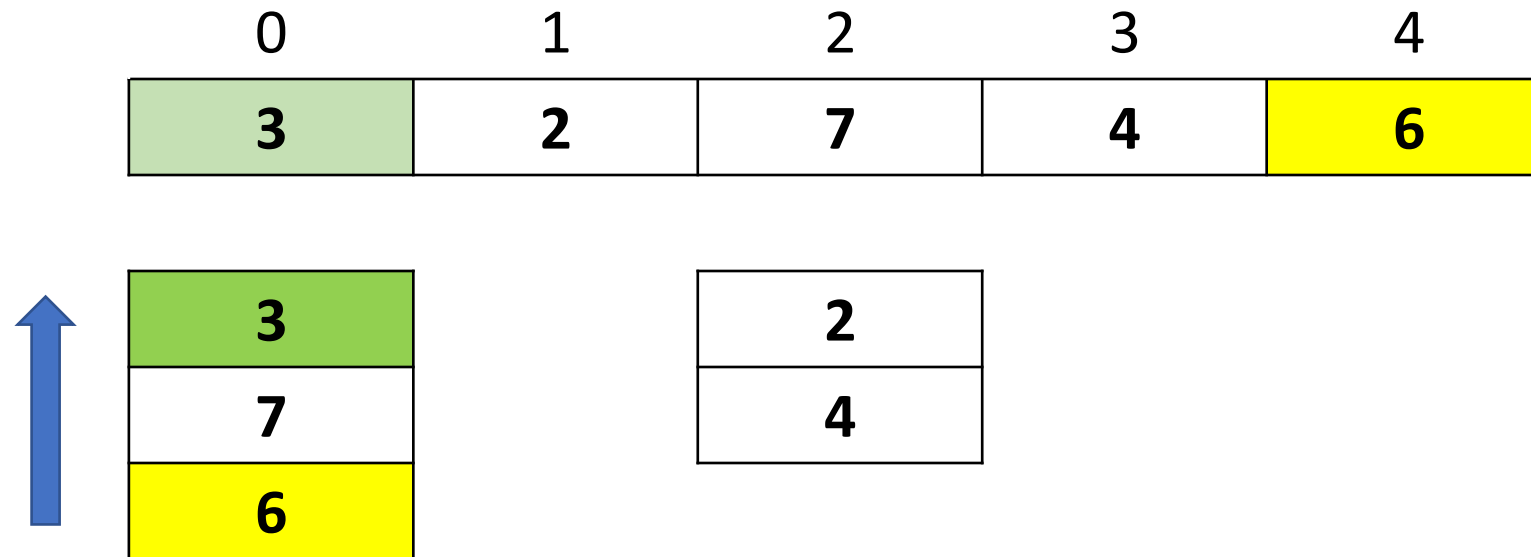
Exemplo – 1º passo

0	1	2	3	4
7	2	6	4	3

1º passo



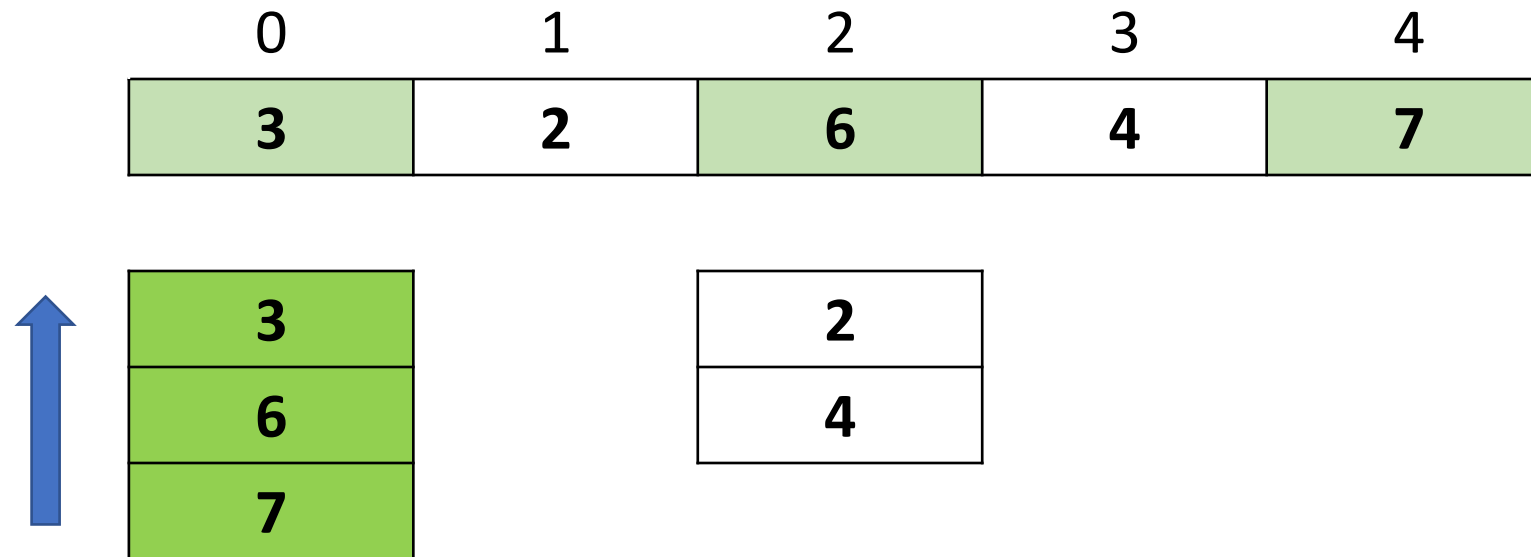
Exemplo – 1º passo



1º passo

- 2 comparações
- 2 deslocamentos

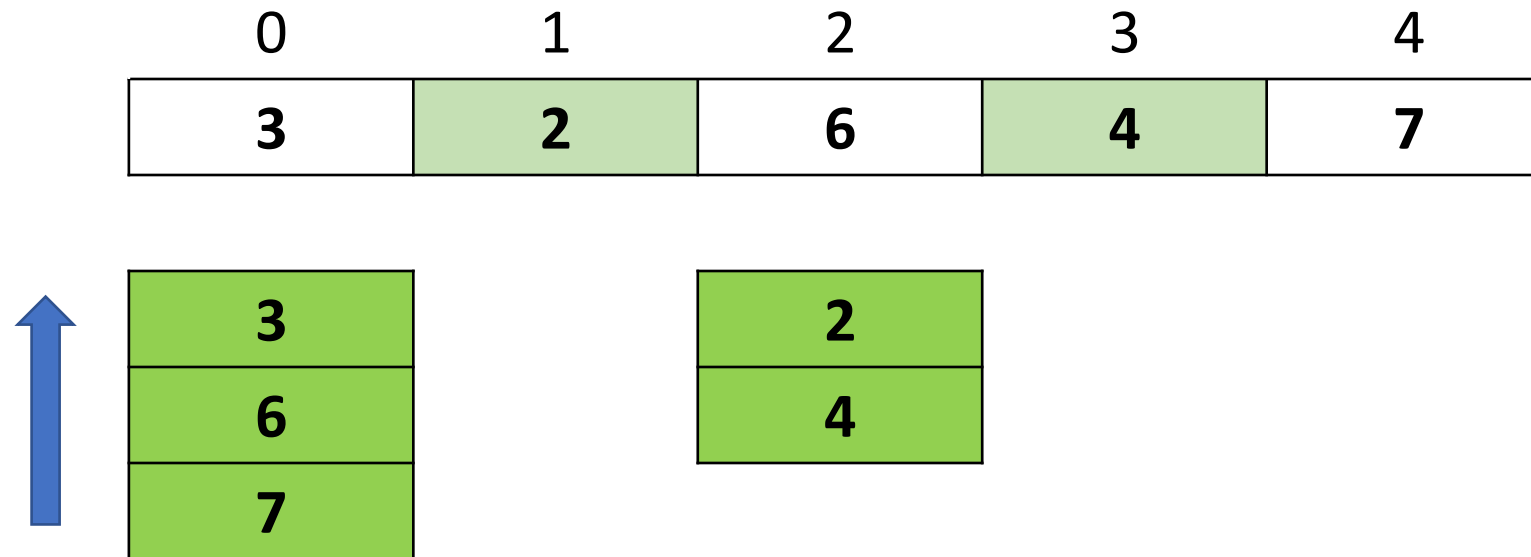
Exemplo – 1º passo



1º passo

- 2 + 1 comparações
- 2 + 1 deslocamentos

Exemplo – 1º passo



1º passo

- $2 + 1 + 1$ comparações
- $2 + 1 + 0$ deslocamentos

Exemplo – 2º passo

0	1	2	3	4
3	2	6	4	7

2º passo

$h = 1$

Alg. Inserção

Exemplo – 2º passo

0	1	2	3	4
3	2	6	4	7
3	2	6	4	7

2º passo

$h = 1$

Alg. Inserção

Exemplo – 2º passo

0	1	2	3	4
3	2	6	4	7

3	2	6	4	7
---	---	---	---	---

2º passo

$h = 1$

Alg. Inserção

Exemplo – 2º passo

0	1	2	3	4
3	2	6	4	7

3	2	6	4	7
2	3	6	4	7

2º passo

$h = 1$

Alg. Inserção

- 1 comparação + 1 deslocamento

Exemplo – 2º passo

0	1	2	3	4
2	3	6	4	7
2	3	6	4	7

2º passo

$h = 1$

Alg. Inserção

Exemplo – 2º passo

0	1	2	3	4
2	3	6	4	7

2	3	6	4	7
2	3	6	4	7

2º passo

$h = 1$

Alg. Inserção

- 1 comparação + 0 deslocamentos

Exemplo – 2º passo

0	1	2	3	4
2	3	6	4	7
2	3	6	4	7

2º passo

$h = 1$

Alg. Inserção

Exemplo – 2º passo

0	1	2	3	4
2	3	6	4	7

2	3	6	4	7
2	3	4	6	7

2º passo

$h = 1$

Alg. Inserção

Exemplo – 2º passo

0	1	2	3	4
2	3	6	4	7

2	3	6	4	7
2	3	4	6	7
2	3	4	6	7

2º passo

$h = 1$

Alg. Inserção

- 2 comparações + 1 deslocamento

Exemplo – 2º passo

0	1	2	3	4
2	3	4	6	7
2	3	4	6	7

2º passo

$h = 1$

Alg. Inserção

Exemplo – 2º passo

0	1	2	3	4
2	3	4	6	7

2	3	4	6	7
2	3	4	6	7

2º passo

$h = 1$

Alg. Inserção

- 1 comparações + 0 deslocamentos

Tarefa 3






- Usar a estratégia apresentada para ordenar o array :

0	1	2	3	4	5	6	7	8	9	10	11	12
81	94	11	96	12	35	17	95	28	58	41	75	15

Tarefa 4

- Organizar **configurações do array** que correspondam:
- Ao **melhor caso** para as **comparações**
- Ao **pior caso** para as **comparações**
- Ao **melhor caso** para os deslocamentos
- Ao **pior caso** para os deslocamentos
- Alguns dos casos anteriores ocorrem em **simultâneo** ?

Shell Sort – Sequência original de distâncias

```
void shellSort( int a[], int n ) {  
    for( int gap = n / 2; gap > 0; gap /= 2 )    // Gap sequence  
        for( int i = gap; i < n; i++ ) {        // Elements to be sorted  
            int tmp = a[ i ];   
            int j = i;   
             for( ; j >= gap && tmp < a[ j - gap ]; j -= gap )  
                a[ j ] = a[ j - gap ];   
            a[ j ] = tmp;   
        }  
    }
```

Tarefa 5

- Usar a o **algoritmo da função shellSort** para ordenar o array :

0	1	2	3	4	5	6	7	8	9	10	11	12
81	94	11	96	12	35	17	95	28	58	41	75	15

Melhor Caso – Array Ordenado

- Considerar casos particulares, para simplificar : $n = 2^k$
- Ciclo externo executa k vezes : $\text{gap} = 2^{k-1}, \dots, 2^0$
- Ciclo intermédio executa $(n - \text{gap})$ vezes
- Ciclo interno nunca executa !! - Porquê ?
- E é efetuada **uma comparação e duas atribuições**

$$B_C(n) = (n - 2^{k-1}) + (n - 2^{k-2}) + \dots + (n - 1)$$

$$B_C(n) = n \log n - (n - 1)$$

Melhor Caso – Array Ordenado

- Considerar casos particulares, para simplificar : $n = 2^k$, $k = \log n$

$$B_c(n) = n \log n - (n - 1) \quad B_c(n) \in O(n \log n)$$

- Esta ordem de complexidade, para o melhor caso, é comum à maioria das sequências de distâncias habitualmente consideradas

Pior Caso

- Considerar casos particulares, para simplificar : $n = 2^k$, $k = \log n$
- Ciclo externo executa k vezes : $\text{gap} = 2^{k-1}, \dots, 2^0$
- Ciclo intermédio executa $(n - \text{gap})$ vezes
- Ciclo interno executa o número total de iterações possíveis !! - Porquê ?
- Repetidas execuções do algoritmo Insertion Sort
- Sempre com o comportamento de Pior Caso
- Para conjuntos de elementos de tamanho sucessivamente maior

Pior Caso

- Considerar casos particulares, para simplificar : $n = 2^k$, $k = \log n$
- Execuções (**Pior Caso**) do algoritmo **Insertion Sort** sobre
 - 2^{k-1} conjuntos de 2 elementos
 - 2^{k-2} conjuntos de 4 elementos
 - ...
 - 1 conjunto de $n = 2^k$ elementos

$$\sum_{i=0}^{k-1} 2^i O\left(\left(\frac{n}{2^i}\right)^2\right) = \sum_{i=0}^{k-1} O\left(\frac{n^2}{2^i}\right) = O(n^2)$$

Pior Caso

- MAS, obteve-se uma ordem de complexidade quadrática ?!?
- Porquê ?
- A sequência de distâncias não é a melhor !
- Shell, 1959 : 1, ..., $n / 4$, $n / 2$ $O(n^2)$
- Hibbard, 1963 : 1, 3, 7, 15, 31, ... $O(n^{3/2})$
- Sedgewick, 1982 : 1, 8, 23, 77, 281, ... $O(n^{4/3})$
- Sedgewick, 1982 : 1, 5, 19, 41, 109, ... $O(n^{4/3})$
- ...



Conclusão importante

- Uma característica única do algoritmo Shell Sort :
- O **algoritmo** é sempre **o mesmo**
- MAS a **escolha** da **sequência de distâncias** a usar tem um **efeito “dramático”** sobre a sua **ordem de complexidade**

Tarefa 6

- Ordenar os arrays dos exemplos anteriores usando as sequências de distâncias de Hibbard e de Sedgewick

Implementação Genérica

Desenvolvimento genérico

- Implementar cada algoritmo de ordenação **uma só vez**
 - Evitar redundância : copiar / colar / modificar
- MAS, **arrays de diferentes tipos** como **argumento de entrada !!**
- **Como fazer ?** -> Tipo genérico: **void ***
- MAS, a operação de **comparação** depende do **tipo de elementos !!**
- **Como fazer ?** -> **Função de comparação** é um **argumento**

Ponteiro para um função

- Em C, o **identificador de uma função** é um **ponteiro** !!

```
Int compare(int x, int y); // protótipo
```

```
// declaração
```

```
// ponteiro para função; dois argumentos inteiros; devolve um inteiro
```

```
Int (*compFunc)(int a, int b);
```

```
compFunc = compare;
```

```
r = compFunc(5, 10); // o mesmo que compare(5, 10)
```

void *

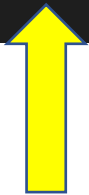
- O tipo **void *** é usado para definir **ponteiros genéricos** em C
 - Semelhante a Object em Java
- Um **ponteiro de qualquer tipo** pode ser **atribuído** a um **ponteiro do tipo void ***
- **MAS**, **perde-se** a informação quanto ao **tipo da variável referenciada**
- Fazer o **casting** para o tipo desejado, **quando necessário**

Tipos auxiliares – typedef

```
// The type for the comparator less function: *p1 < *p2
typedef int (*lessFunc)(void* p1, void* p2);

// The type for the swap function
typedef void (*swapFunc)(void* p1, void* p2);

// The type for the print function
typedef void (*printFunc)(void* p);
```



Imprimir um elemento

```
void printForInts(void* p) {  
    assert(p != NULL);  
    printf("%d", *(int*)p); // Casting to the appropriate pointer type  
}
```




```
void printForPersons(void *p) {  
    assert(p != NULL);  
    struct Person *s = (struct Person *)p; // Casting to the appropriate pointer  
    printf("%s %s", s->firstName, s->lastName);  
}
```




Registo – struct

```
struct Person {  
    char firstName[25];  
    char lastName[25];  
};
```

Trocar dois elementos



```
void swapForInts(void* p1, void* p2) {  
    assert(p1 != NULL && p2 != NULL);  
    int temp = *(int*)p1;  
    *(int*)p1 = *(int*)p2;  
    *(int*)p2 = temp;  
}
```



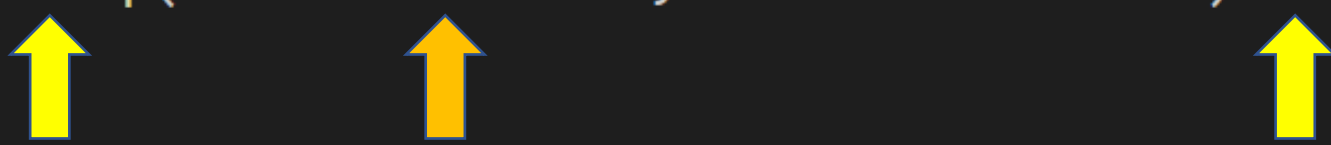
```
void swapForPersons(void *p1, void *p2) {  
    assert(p1 != NULL && p2 != NULL);  
    struct Person temp = *(struct Person *)p1;  
    *(struct Person *)p1 = *(struct Person *)p2;  
    *(struct Person *)p2 = temp;  
}
```

Comparar dois inteiros


```
int lessForInts(void* p1, void* p2) {  
    assert(p1 != NULL && p2 != NULL);  
    int a = *(int*)p1;  
    int b = *(int*)p2;  
    return a < b;  
}
```

Duas funções de comparação

```
int lessForFirstName(void *p1, void *p2) {  
    assert(p1 != NULL && p2 != NULL);  
    struct Person *first = (struct Person *)p1;  
    struct Person *second = (struct Person *)p2;  
    return strcmp(first->firstName, second->firstName) < 0;  
}
```



```
int lessForLastName(void *p1, void *p2) {  
    assert(p1 != NULL && p2 != NULL);  
    struct Person *first = (struct Person *)p1;  
    struct Person *second = (struct Person *)p2;  
    return strcmp(first->lastName, second->lastName) < 0;  
}
```



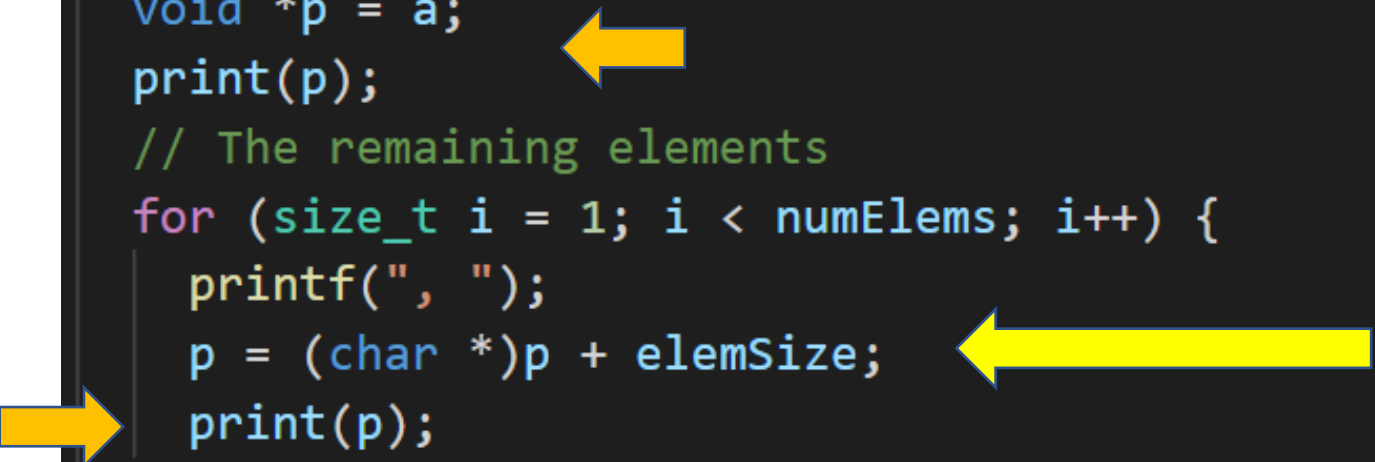
Função Genérica – printArray

```
// printArray
// a : pointer to the array
// numElems : number of elements
// elemSize : number of bytes for each array element
// print : pointer to the element print function
void printArray(void* a, size_t numElems, size_t elemSize, printFunc print);
```



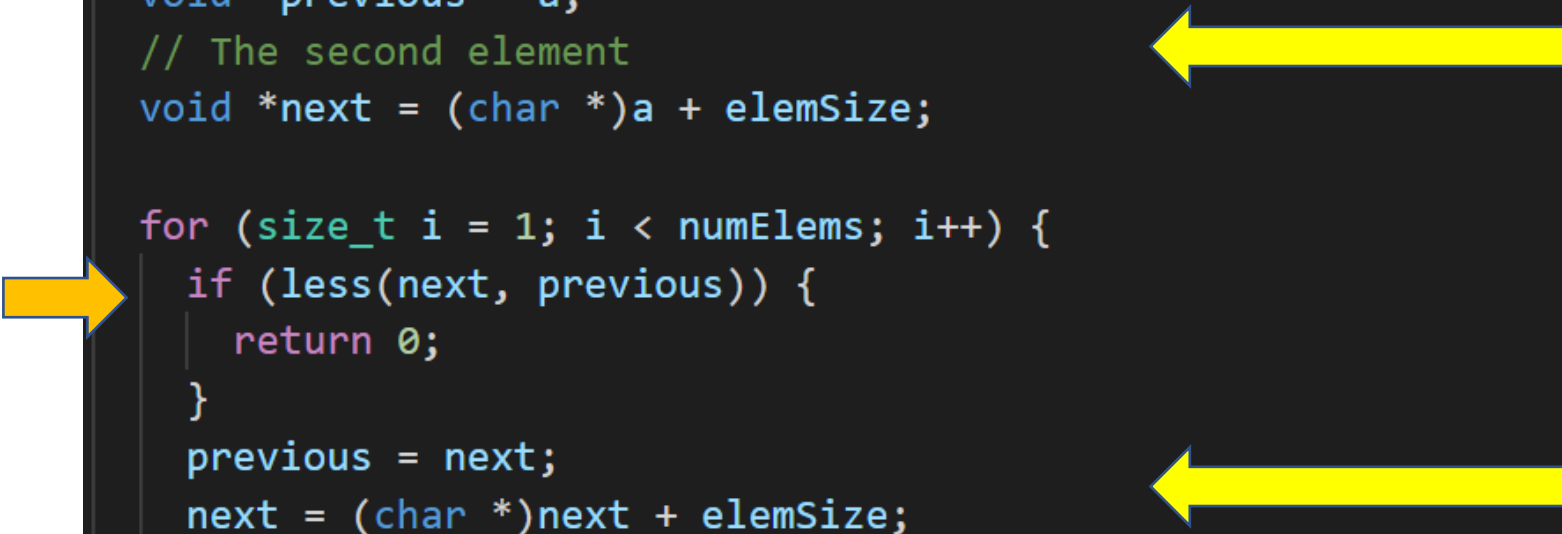
Função Genérica – printArray

```
void printArray(void *a, size_t numElems, size_t elemSize, printFunc print)
{
    assert(a != NULL && numElems > 0 && elemSize > 0 && print != NULL);
    printf("[ ");
    // The first element
    void *p = a;
    print(p);
    // The remaining elements
    for (size_t i = 1; i < numElems; i++) {
        printf(", ");
        p = (char *)p + elemSize;
        print(p);
    }
    printf(" ]\n");
}
```




Função Genérica - isSorted

```
int isSorted(void *a, size_t numElems, size_t elemSize, lessFunc less) {  
    assert(a != NULL && numElems > 0 && elemSize > 0 && less != NULL);  
  
    // Pointer arithmetic is not allowed on void pointers  
    // The first element  
    void *previous = a;  
    // The second element  
    void *next = (char *)a + elemSize;  
  
    for (size_t i = 1; i < numElems; i++) {  
        if (less(next, previous)) {  
            return 0;  
        }  
        previous = next;  
        next = (char *)next + elemSize;  
    }  
    return 1;  
}
```





Função Genérica - bubbleSort

```
void bubbleSort(void *a, size_t numElems, size_t elemSize, lessFunc less,  
               swapFunc swap) {  
    assert(a != NULL && numElems > 0 && elemSize > 0 && less != NULL &&  
           swap != NULL);  
  
    void *previous;  
    void *next;  
  
    size_t k = numElems;  
    int stop = 0;  
  
    while (stop == 0) {  
        stop = 1;  
        k--;  
    }  
}
```



Função Genérica - bubbleSort

```
while (stop == 0) {  
    stop = 1;  
    k--;  
    // Pointer arithmetic is not allowed on void pointers  
    // The first element  
    previous = a;  
    // The second element  
    next = (char *)a + elemSize;  
    for (size_t i = 0; i < k; i++) {  
        if (less(next, previous)) {  
            swap(previous, next);  
            stop = 0;  
        }  
        previous = next;  
        next = (char *)next + elemSize;  
    }  
}
```



Tarefa 7

- Analisar os **exemplos** disponibilizados
 - Ordenar arrays de número inteiros
 - Ordenar arrays de registos
- Implementar **versões genéricas** dos **outros algoritmos de ordenação**
- Testar
- **Desenvolver outros exemplos**, com arrays de diferentes tipos

Sugestão de leitura

Sugestão de leitura

- J. J. McConnell, Analysis of Algorithms, 1st Edition, 2001
 - Capítulo 3: **secção 3.3**