AULA 14
ESTRUTURA DE DADOS

Algoritmos de Ordenação

Algoritmos de Ordenação

- Problema:
  - sequência de n itens ou elementos (registros) a chave a[i] está associada ao elemento r[i];
  - se i < j então a[i] precede a[j]

Terminologia

- a ordenação tem por objetivo rearrumar as chaves de forma que obedeçam a uma regra (ex.: ordem numérica ou alfabética)
- a ordenação é **estável** se preserva a ordem relativa das chaves iguais (senão, **instável**)
- os algoritmos podem operar sobre o elemento ou sobre uma tabela de ponteiros

Algoritmos

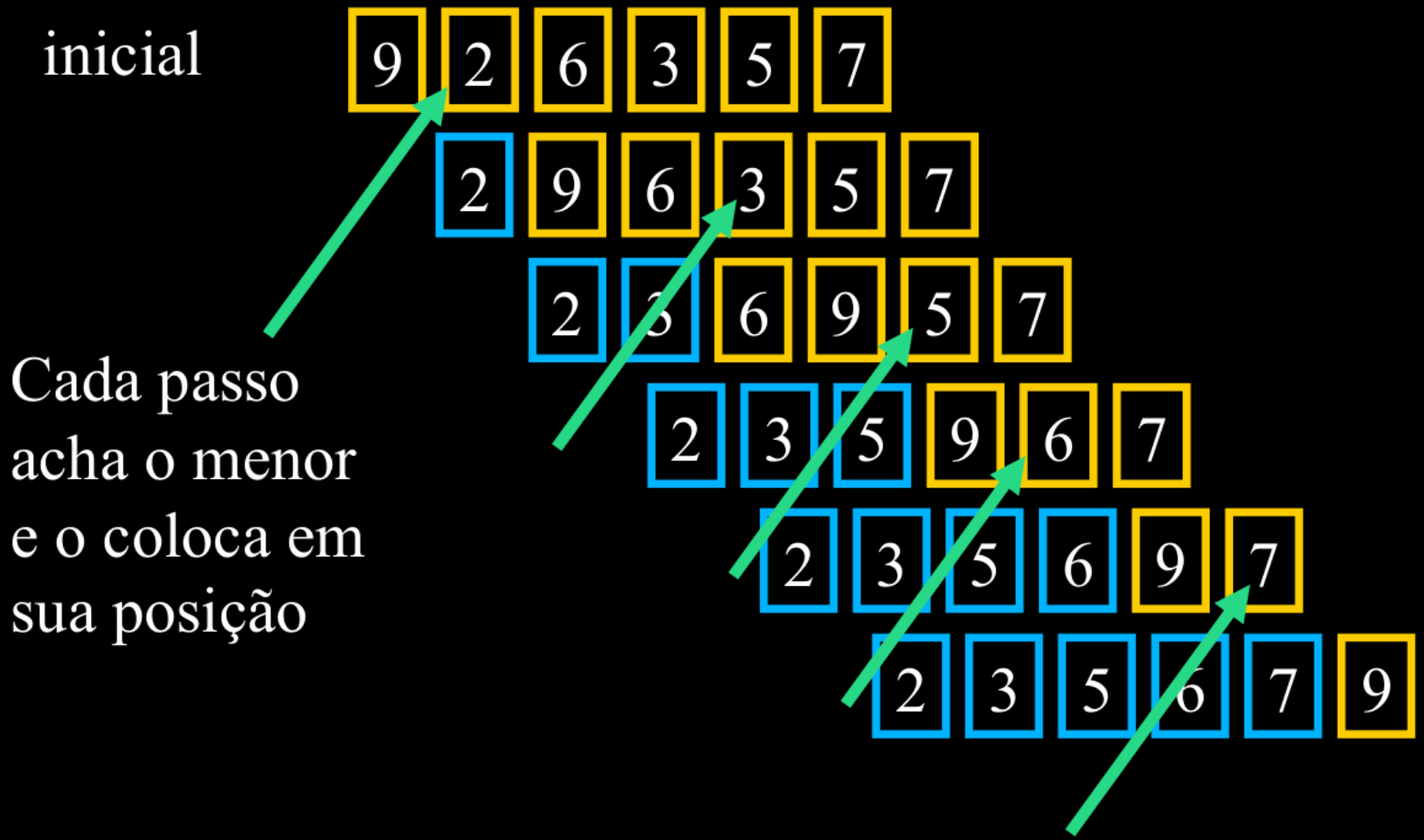
1) Ordenação por seleção;
2) Bubble Sort;
3) Merge Sort;
4) Quick Sort.

Ordenação por Seleção

1) ache primeiro o menor elemento da lista e troque sua posição com o primeiro elemento;
2) ache o segundo menor elemento e troque sua posição com o segundo elemento;
3) continue até que toda lista esteja ordenada !

para cada *i* de *0* a *n-2*, troca o menor elemento da sub-lista que vai de *i* a *N-1* com *a[i]*

Ordenação por Seleção



Ordenação por Seleção

```
void Ord_Seleção(int a[])
{

}
```

Ordenação por Seleção

```
void Ord_Seleção(int a[])
{
    int t, i, j,min;
    for (i=0;i< n-2; i++){
            min = i;
            for (j=i+1; i< n-1; i++)
                if (a[j]<a[min]) min = j;
                    /* troca */
            t = a[min];
            a[min]=a[i];
            a[i] = t;
        }
}
```

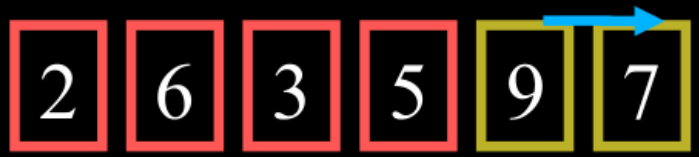
Ordenação por Seleção

```
void Ord_Seleção(int a[])
{
    int t, i, j,min;
    for (i=0;i< n-2; i++){
            min = i;
            for (j=i+1; i< n-1; i++)
                if (a[j]<a[min]) min = j;  //Seleção
                    /* troca */
            t = a[min];
            a[min]=a[i];
            a[i] = t;
        }
}
```

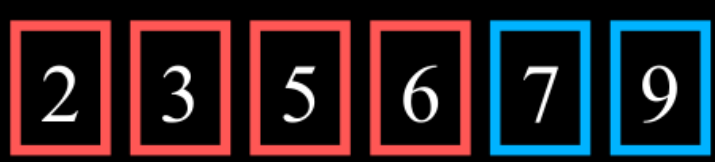
Bubble Sort

1) percorrer a lista trocando elementos adjacentes, se necessário;
2) quando nenhuma troca for efetuada ao percorrer toda a lista " está classificada

Bubble Sort



Bubble Sort



Bubble Sort

Bubble Sort

```
void bubble_sort(int a[])
{
    int i, j, t, trocas;
    for (i = n-1; i>0; i--) {
        trocas = 0;
        for (j = 1; j <= i; j++){
            if (a[j-1] > a[j]){
            /* troca */
            t = a[j-1]; a[j-1] = a[j]; a[j] = t;
            trocas++;
            } // Fim do if
        } // Fim do for
        if (trocas == 0) break;
    } // Fim do for
}
```

Bubble Sort

```
void bubble_sort(int a[])
{

}
```

Bubble Sort

```
void bubble_sort(int a[])
{
    int i, j, t, trocas;
    for (i = n-1; i>0; i--) {
        trocas = 0;  //Contador de trocas
        for (j = 1; j <= i; j++){
            if (a[j-1] > a[j]){
            /* troca */
            t = a[j-1]; a[j-1] = a[j]; a[j] = t;
            trocas++;
            } // Fim do if
        } // Fim do for
        if (trocas == 0) break;
    } // Fim do for
}
```

Bubble Sort

```
void bubble_sort(int a[])
{
    int i, j, t, trocas;
    for (i = n-1; i>0; i--) {
        trocas = 0;
        for (j = 1; j <= i; j++){
            if (a[j-1] > a[j]){
            /* troca */
            t = a[j-1]; a[j-1] = a[j]; a[j] = t;
            trocas++;
            } // Fim do if
        } // Fim do for
        if (trocas == 0) break;
    } // Fim do for
}
```

Bubble Sort

```
void bubble_sort(int a[])
{
    int i, j, t, trocas;
    for (i = n-1; i>0; i--) {
        trocas = 0;
        for (j = 1; j <= i; j++){
            if (a[j-1] > a[j]){
            /* troca */
            t = a[j-1]; a[j-1] = a[j]; a[j] = t;
            trocas++;
            } // Fim do if
        } // Fim do for
        if (trocas == 0) break;  //Saída
    } // Fim do for
}
```

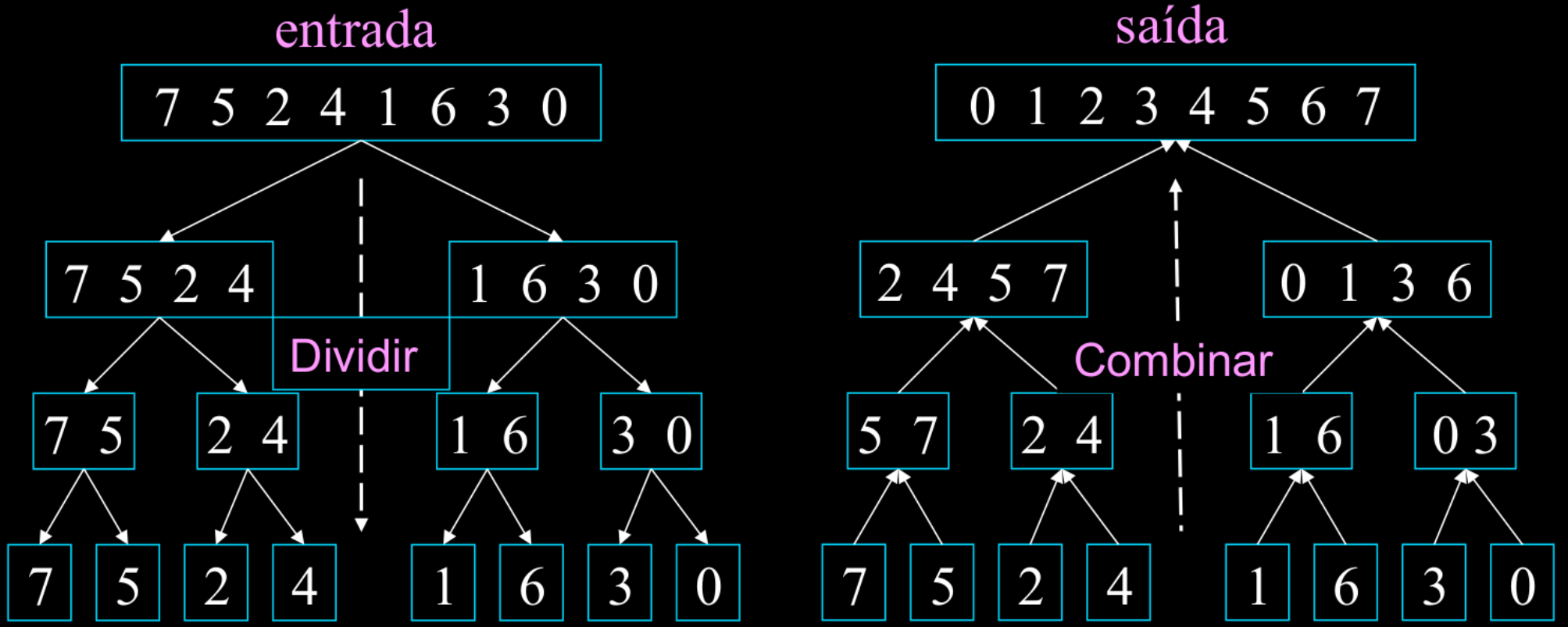
MergeSort

Seja um arranjo A de *n* elementos.
O algoritmo consiste das seguintes fases:

1) Dividir A em 2 sub-arranjos de tamanho ≈ *n /2*;
2) Conquistar: ordenar cada sub-arranjo chamando MergeSort recursivamente;
3) Combinar os sub-arranjos ordenadas formando um único arranjo ordenado

Caso Base: Arranjo com um elemento

MergeSort


entrada
7 5 2 4 1 6 3 0
7 5 2 4
1 6 3 0
Dividir
7 5
2 4
1 6
3 0
7
5
2
4
1
6
3
0
saída
0 1 2 3 4 5 6 7
2 4 5 7
0 1 3 6
Combinar
5 7
2 4
1 6
0 3
7
5
2
4
1
6
3
0

MergeSort

```
void MergeSort (int Vector[],int i,int f)
{
    int meio;
    if (i < f ){
    meio = (i+f)/2;
    MergeSort (Vector,i, meio);
    MergeSort (Vector,meio+1, f);
    Merge (Vector,i, meio, f);
    }
}
```

MergeSort

```
void MergeSort (int Vector[],int i,int f)
{
    int meio;
    if (i < f ){
    meio = (i+f)/2;
    MergeSort (Vector,i, meio);
    MergeSort (Vector,meio+1, f); //Recursão
    Merge (Vector,i, meio, f);
    }
}
```

MergeSort

```
void MergeSort (int Vector[],int i,int f)
{
    int meio
    if (i < f ){
    meio = (i+f)/2;
    MergeSort (Vector,i, meio);
    MergeSort (Vector,meio+1, f);
    Merge (i, meio, f); Combinação
    }
}
```

MergeSort

```
void Merge(int vetor[], int i, int meio, int f) {

????
```

MergeSort

```
void Merge(int vetor[], int i, int meio, int f) {
    int c1 = i, c2 = meio+1, cAux = 0, tam = f-i+1;
    int *vetAux;
    vetAux = (int*)malloc(tam * sizeof(int));

    while(c1 <= meio && c2 <= fim){
          if(vetor[c1] < vetor[c2]) {
           vetAux[cAux] = vetor[c1];
           c1++;
        } else {
           vetAux[cAux] = vetor[c2];
           c2++;
        }
        cAux++;
    }
```

```
    while(c1 <= meio){  //Caso haja elementos na primeira metade
        vetAux[comAux] = vetor[c1];
        cAux++;
        c1++;}

    while(c2 <= fim) {   //Caso haja elementos na segunda metade
        vetAux[cAux] = vetor[c2];
        cAux++;
        c2++; }
for(cAux = i; cAux <= f; cAux++){    //Move os elementos de volta
    para o vetor original
        vetor[cAux] = vetAux[cAux-i];
    }
    free(vetAux);
}
```

QuickSort

Eficiente naturalmente recursivo.
implementação seria “complicada” sem recursão

Algoritmo Básico ! ! ! "dividir para conquistar"

1) particiona a lista de elementos em duas partes e as classifica independentemente
2) a posição exata da partição depende da lista

QuickSort

```
void quickSort (int Vector[],int i,int f)
{
    int p;
    if (i < f ){
    p = partition (Vector,i,f); /* importante * /
    quicksort(i,p-1);
    quicksort(p+1,f);
    }
}
```

QuickSort

A primeira chamada: quicksort(a,1,n)

- A função partition deve:
  - rearrumar a lista de acordo com as três condições:
    - o elemento a[p] está em seu lugar final na lista;
    - todos os elementos em a[i] a a[p-1] são menores que a[p];
    - todos os elementos em a[p+1] a a[f] são maiores ou iguais a a[p].

QuickSort

Partition:

1) escolha arbitrariamente um elemento pivot a[r]
   elemento que estará na sua posição ao final;
2) percorra a lista da esquerda até que um elemento maior que a[r] seja encontrado
3) percorra a lista da direita até um elemento menor que a[r]

esses dois elementos estão fora de posição - **troque-os**

QuickSort

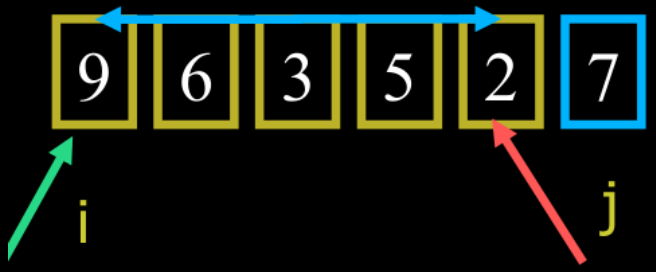
- Parar a varredura toda vez que um elemento for igual ao pivot a[r];
- Dessa forma garante-se que todos os elementos da lista esquerda são menores e os a direita são maiores

QuickSort

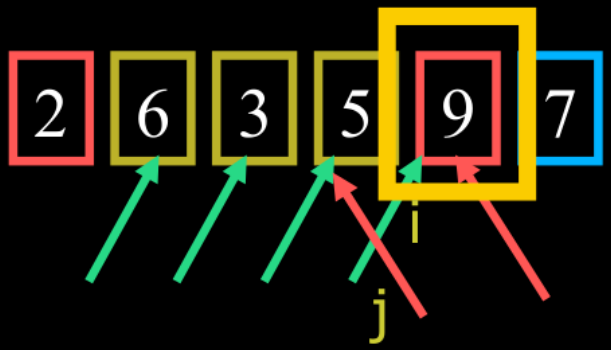
Quando os ponteiros de varredura se cruzam, o processo está quase completo:
Falta trocar a[r] com o elemento mais a esquerda da sub-lista da direita – o elemento a[p] a posição p foi definida

Aplicar quicksort nas sublistas de i a p-1 e p+1 a f;
a[p] é o elemento que já está na sua posição

QuickSort



1ª iteração (do-while externo)



2ª iteração (do-while externo)

- fora do-while externo - trocas
- 7 - em sua posição definitiva
- partition retorna i = 4

QuickSort

```
int partition(int a[],int i, int f) {int v, k, j;
v = a[f]; j = i -1; k = f;
do {
     do{ j = j+1; /* esquerda*/
          } while (a[j] < v) && (j< f) ;
     do{ k = k-1; /* direita*/
     } while (a[k] > v) && (k > 0);
     t=a[j]; a[j]=a[k]; a[k]=t;
} while (j > i)
```

QuickSort

```
int partition(int a[],int i, int f) {int v, k, j;
v = a[f]; j = i -1; k = f;
do {
    do{ j = j+1; /* esquerda*/
        } while (a[j] < v) && (j< f) ;
    do{ k = k-1; /* direita*/
    } while (a[k] > v) && (k > 0);
    t=a[j]; a[j]=a[k]; a[k]=t;
} while (j > i)
```

QuickSort

```
int partition(int a[],int i, int f) {int v, k, j;
v = a[f]; j = i -1; k = f;
do {
    do{ j = j+1; /* esquerda*/
        } while (a[j] < v) && (j< f) ;
    do{ k = k-1; /* direita*/
    } while (a[k] > v) && (k > 0);
    t=a[j]; a[j]=a[k]; a[k]=t;
} while (j > i)
```

QuickSort

```
    /* para desfazer a troca extra realizada quando j<=
       i e saiu do while interno (t já tem o valor de a[j]) */
–
        a[k] = a[j];
        a[j] = a[f];
        a[f] = t;
        return j;
    }
```

QuickSort

O loop interno é muito simples:

- Incrementa um ponteiro e faz uma comparação
- É o que faz o quicksort ser realmente rápido

Pode-se garantir que a[f] nunca será o maior ou o menor elemento?

- o particionamento da lista seria desequilibrado em relação ao número de elementos

QuickSort : Partition

Pegue arbitrariamente 3 elementos de a:

a[f], a[i] e a[$\lfloor (i+f)/2 \rfloor$]

- Compare os 3 elementos e defina o de valor médio
- Troque com a[f]
- Execute o partition

Vantagem: acrescentam-se um número fixo de instruções e não testes que variam com o tamanho do array

QuickSort : Características

- tempo de processamento depende da seleção do pivot
- o algoritmo não é estável
- Cruzamento dos ponteiros na presença de chaves iguais:
  - tanto faz se a varredura de ambos os ponteiros parar, um continuar e outro parar, nenhum parar.
  - grande número dechaves repetidas – Melhorias podem ser feitas

AULA 14
ESTRUTURA DE DADOS

Algoritmos de Ordenação