ALGORITMOS E ESTRUTURAS DE DADOS I

Prof. Caio César de Freitas Dantas

Complexidade - Algoritmos Ordenação

BubbleSort

Ele é um algoritmo simples, apesar de não muito eficiente.

fase	*i*	*j*	*v*[1]	*v*[2]	*v*[3]	*v*[4]	*v*[5]
1º	1	1	**46**	**39**	55	14	27
		2	39	**46**	**55**	14	27
		3	39	46	**55**	**14**	27
		4	39	46	14	**55**	**27**
			39	46	14	27	**55**
2º	2	1	**39**	**46**	14	27	**55**
		2	39	**46**	**14**	27	**55**
		3	39	14	**46**	**27**	**55**
			39	14	27	**46**	**55**
3º	3	1	**39**	**14**	27	**46**	**55**
		2	14	**39**	**27**	**46**	**55**
			14	27	**39**	**46**	**55**
4º	4	1	**14**	**27**	**39**	**46**	**55**
			14	**27**	**39**	**46**	**55**

Complexidade - Algoritmos Ordenação

Vamos analisar o funcionamento do algoritmo.

Suponha um vetor de inteiros com n elementos que queremos ordenar em ordem crescente.

1.Tomamos o primeiro elemento e analisamos o seu sucessor no array. Caso o sucessor seja menor que o antecessor, trocamos os dois de lugar.

2.Repetimos o passo 1 para todos os pares até o final do array. Ao final desse processo o elemento de maior valor ficará na última posição do array.

3.Repetimos a operação n vezes, uma para cada elemento. Ao final do processo o vetor estará ordenado.

Complexidade - Algoritmos Ordenação

Este algoritmo vai funcionar para qualquer vetor de inteiros.

- O algoritmo precisa percorrer o vetor inteiro várias vezes em sua execução (exatamente n^2 vezes).

- O número de operações a serem executadas é sempre o mesmo, indepentente de como os valores estão no vetor original, isto é, o tempo de "ordenar" um vetor já ordenado e de ordenar um vetor completamente não-ordenado, neste algoritmo, é o mesmo.

```
void bubbleSort(int v[], int n){
    int i, j;
    // passo 3
    for(j = 0; j < n; j++){
        // passo 2
        for(i = 0; i < n-1; i++){
            // passo 1
            if(v[i] > v[i+1]){
                int aux = v[i];
                v[i] = v[i+1];
                v[i+1] = aux;
```

Complexidade - Algoritmos Ordenação

Vamos analisar o funcionamento do algoritmo.

Podemos melhorar o algoritmo. A ideia básica continua sendo a mesma, mas a eficiência aumenta, pois melhoramos a performance dele para alguns casos mais comuns, adicionando as seguintes mudanças.

1.Adicionamos uma flag "ok" para indicar se uma passada pelo vetor não produziu nenhuma mudança. Isto é, se percorremos o vetor inteiro e nenhuma mudança foi necessária, podemos parar.

2.Sabemos que a cada execução do passo 2 um elemento vai para a sua posição final. Podemos tirar vantagem desse fato fazendo com que menos pares sejam comparados por iteração do loop mais interno.

Complexidade - Algoritmos Ordenação

A mudança pode parecer pequena, mas aumenta bastante a eficiência do algoritmo.

Para int v[] = {-10, 2, 0, 4, 6, 2, -5, 20, 7, 9};

P primeiro BubbleSort, possui 283 passos, enquanto o segundo só possui 190, uma melhora de quase 33%.

```
void betterBubbleSort(int v[], int n){
    int i, j, ok = 0;
    // passo 3
    for(j = 0; j < n && ok == 0; j++){
        // passo 2
        ok = 1;
        for(i = 0; i < n-j-1; i++){
            // passo 1
            if(v[i] > v[i+1]){
                ok = 0;
                int aux = v[i];
                v[i] = v[i+1];
                v[i+1] = aux;
```

Complexidade - Algoritmos Ordenação

- A primeira linha é executada somente 1 vez.
- O laço do passo 3 é executado n vezes.
- O laço do passo 2 é executado n vezes para cada execução do passo 3, portanto n^2 vezes.
- A condição if do passo 1 é executada sempre, portanto n^2 vezes.

- Portanto a estimativa seria da ordem de $1+ 2n^2 + \alpha$, onde α é a quantidade de vezes que as operações internas ao if do passo 1 são executadas.
  Removendo as constantes e variáveis de menor ordem temos que a complexidade do BubbleSort, assintóticamente, é **$O(n^2)$.**

```
void bubbleSort(int v[], int n){
    int i, j; // executado 1 vez
    // passo 3 - executado n vezes
    for(j = 0; j < n; j++){
        // passo 2 - executado n vezes
        for(i = 0; i < n-1; i++){
            // passo 1
            if(v[i] > v[i+1]){  // executado toda vez
                int aux = v[i];
                v[i] = v[i+1];
                v[i+1] = aux;
```

Complexidade - Algoritmos Ordenação

InsertionSort

É o método que percorre um vetor de elementos da esquerda para a direita e à medida que avança vai ordenando os elementos à esquerda.

O funcionamento do algoritmo é bem simples: consiste em cada passo a partir do segundo elemento selecionar o próximo item da sequência e colocá-lo no local apropriado de acordo com o critério de ordenação.

Complexidade - Algoritmos Ordenação

InsertionSort

i	v_1	v_2	v_3	v_4	v_5	v_6
1	34	17	68	29	50	47
2	17	34	68	29	50	47
3	17	34	68	29	50	47
4	17	29	34	68	50	47
5	17	29	34	50	68	47
	17	29	34	47	50	68

Complexidade - Algoritmos Ordenação

InsertionSort

INSERTION-SORT(A, n)	Tempo
1 **para** $j \leftarrow 2$ **até** n **faça**	$\Theta(n)$
2 $\quad chave \leftarrow A[j]$	$\Theta(n)$
3 $\quad \triangleright$ Insere $A[j]$ em $A[1 \ldots j-1]$	
4 $\quad i \leftarrow j-1$	$\Theta(n)$
5 $\quad$ **enquanto** $i \geq 1$ **e** $A[i] > chave$ **faça**	$nO(n) = O(n^2)$
6 $\quad\quad A[i+1] \leftarrow A[i]$	$nO(n) = O(n^2)$
7 $\quad\quad i \leftarrow i-1$	$nO(n) = O(n^2)$
8 $\quad A[i+1] \leftarrow chave$	$O(n)$

Consumo de tempo: $O(n^2)$

Complexidade - Algoritmos Ordenação

InsertionSort

```
void insercao (int vet, int tam){
int i, j, x;
for (i=2; i<=tam; i++){              O(n)
    x = vet[i];                      O(n)
    j=i-1;                           O(n)
    vet[0] = x;                      O(n)
    while (x < vet[j]){              nO(n) = n^2
        vet[j+1] = vet[j];           nO(n) = n^2
        j--;                         nO(n) = n^2
    }
    vet[j+1] = x;                    O(n)
}
```

Complexidade - Algoritmos Ordenação

Complexidade de tempo no pior caso: Θ(n2)
Vetor em ordem decrescente
Comparações: Θ(n2)
Trocas: Θ(n2)

Complexidade de tempo no melhor caso: Θ(n)
Vetor em ordem crescente
Comparações: Θ(n)
Trocas: zero

Bom método a ser usado quando a sequência esta quase ordenada, ou quando se deseja adicionar poucos itens a uma sequência já ordenada.

Complexidade - Algoritmos Ordenação

Selection Sort

A ordenação por seleção ou selection sort consiste em selecionar o menor item e colocar na primeira posição, selecionar o segundo menor item e colocar na segunda posição, segue estes passos até que reste um único elemento.

Complexidade - Algoritmos Ordenação

Selection Sort

Fase	*i*	*k*	a_1	a_2	a_3	a_4	a_5	a_6
1º	1	4	46	55	59	14	38	27
2º	2	6	14	55	59	46	38	27
3º	3	5	14	27	59	46	38	55
4º	4	4	14	27	38	46	59	55
5º	5	6	14	27	38	46	59	55
			14	27	38	46	55	59

Complexidade - Algoritmos Ordenação

Selection Sort

Como funciona:

1.Encontre o menor elemento no array e troque-o de lugar com o primeiro elemento.

2.Encontre o segundo menor elemento e troque com o segundo elemento no array.

3.Encontre o terceiro menor elemento e troque com o terceiro elemento no array.

4.Repita o processo de encontrar o próximo menor elemento e trocá-lo na posição correta até que todo o array esteja ordenado.

Complexidade - Algoritmos Ordenação

Selection Sort

SELECTION-SORT(A, n)		Tempo
1	**para** $i \leftarrow 1$ **até** $n-1$ **faça**	$\Theta(n)$
2	$min \leftarrow i$	$\Theta(n)$
3	**para** $j \leftarrow i+1$ **até** n **faça**	$\Theta(n^2)$
4	**se** $A[j] < A[min]$ **então** $min \leftarrow j$	$\Theta(n^2)$
5	$A[i] \leftrightarrow A[min]$	$\Theta(n)$

Consumo de tempo no pior caso: $\Theta(n^2)$

Complexidade - Algoritmos Ordenação

Selection Sort

```
void selecao (int vet, int tam){
    int i, j, min, x;
    for (i=1; i<=n-1; i++){                 n
        min = i;                            n
    for (j=i+1; j<=n; j++){                 n*n = n^2
            if (vet[j] < vet[min])          n^2
            min = j;                        n^2
    }
    x = vet[min];                           n
    vet[min] = vet[i];                      n
    vet[i] = x;                             n
    }
}
```

Complexidade - Algoritmos Ordenação

Selection Sort

Complexidade de tempo no pior caso: Θ(n2)
Comparações: Θ(n2)
Trocas: Θ(n)

Complexidade de tempo no melhor caso: Θ(n2)

Complexidade - Algoritmos Ordenação

Selection Sort

- Uma vantagem do Selection Sort é que entre os algoritmos de ordenação ele apresenta uma das menores quantidades de movimentos entre os elementos, assim pode haver algum ganho quando se necessita ordenar estruturas complexas.

- Uma desvantagem é que o número de comparações é igual para o melhor caso, caso médio e o pior caso. Assim, mesmo que o vetor esteja ordenado o custo continua quadrático (n 2).

Complexidade - Algoritmos Ordenação

QuickSort

É um algoritmo de comparação que emprega a estratégia de “divisão e conquista”. A ideia básica é dividir o problema de ordenar um conjunto com n itens em dois problemas menores. Os problemas menores são ordenados independentemente e os resultados são combinados para produzir a solução final.

Basicamente a operação do algoritmo pode ser resumida na seguinte estratégia: divide sua lista de entrada em duas sub-listas a partir de um pivô, para em seguida realizar o mesmo procedimento nas duas listas menores até uma lista unitária.

Complexidade - Algoritmos Ordenação

QuickSort

Funcionamento do algoritmo:

- Escolhe um elemento da lista chamado pivô.
- Reorganiza a lista de forma que os elementos menores que o pivô fiquem de um lado, e os maiores fiquem de outro. Esta operação é chamada de “particionamento”.
- Recursivamente ordena a sub-lista abaixo e acima do pivô.

Complexidade - Algoritmos Ordenação

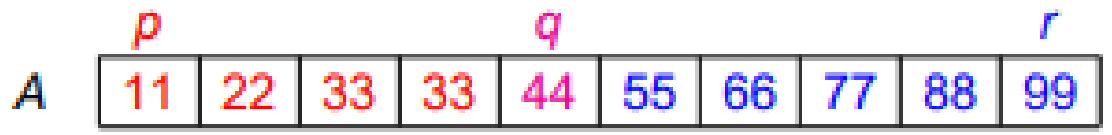
QuickSort

```
PARTICIONE(A, p, r)                          Tempo
1  x ← A[r]   ▷ x é o "pivô"                 Θ(1)
2  i ← p−1                                   Θ(1)
3  para j ← p até r − 1 faça                 Θ(n)
4      se A[j] ≤ x                           Θ(n)
5          então i ← i + 1                   O(n)
6                A[i] ↔ A[j]                 O(n)
7  A[i+1] ↔ A[r]                             Θ(1)
8  devolva i + 1                             Θ(1)
```

$T(n) = \Theta(2n + 4) + O(2n) = \Theta(n)$

Conclusão:
A complexidade de PARTICIONE é $\Theta(n)$.

```
QUICKSORT(A, p, r)
1  se p < r
2      então q ← PARTICIONE(A, p, r)
3            QUICKSORT(A, p, q − 1)
4            QUICKSORT(A, q + 1, r)
```



Complexidade - Algoritmos Ordenação

QuickSort

```
void swap(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
}
int partition (int arr[], int low, int high){
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high- 1; j++){
        if (arr[j] <= pivot){
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
```

```
void quickSort(int arr[], int low, int high){
    if (low < high) {
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
```

Complexidade - Algoritmos Ordenação

QuickSort

A complexidade de tempo do Quick Sort é aproximadamente O(nlog(n)) quando a seleção do pivô divide o array original em dois sub arrays de tamanhos quase iguais.

Por outro lado, se o algoritmo, que seleciona o elemento pivô dos arrays de entrada, produz consistentemente 2 sub-arrays com uma grande diferença em termos de tamanho, o algoritmo de ordenação rápida pode atingir a complexidade temporal de pior caso de O(n^2).

Complexidade - Algoritmos Ordenação

QuickSort

A complexidade de tempo do QUICKSORT no pior caso é Θ(n2).

A complexidade de tempo do QUICKSORT é O(n2).

Complexidade - Algoritmos Ordenação

MergeSort

- Esse algoritmo divide o problema em pedaços menores, resolve cada pedaço e depois junta (merge) os resultados.

- O vetor será dividido em duas partes iguais, que serão cada uma divididas em duas partes, e assim até ficar um ou dois elementos cuja ordenação é trivial.

- Para juntar as partes ordenadas os dois elementos de cada parte são separados e o menor deles é selecionado e retirado de sua parte. Em seguida os menores entre os restantes são comparados e assim se prossegue até juntar as partes.

Complexidade - Algoritmos Ordenação

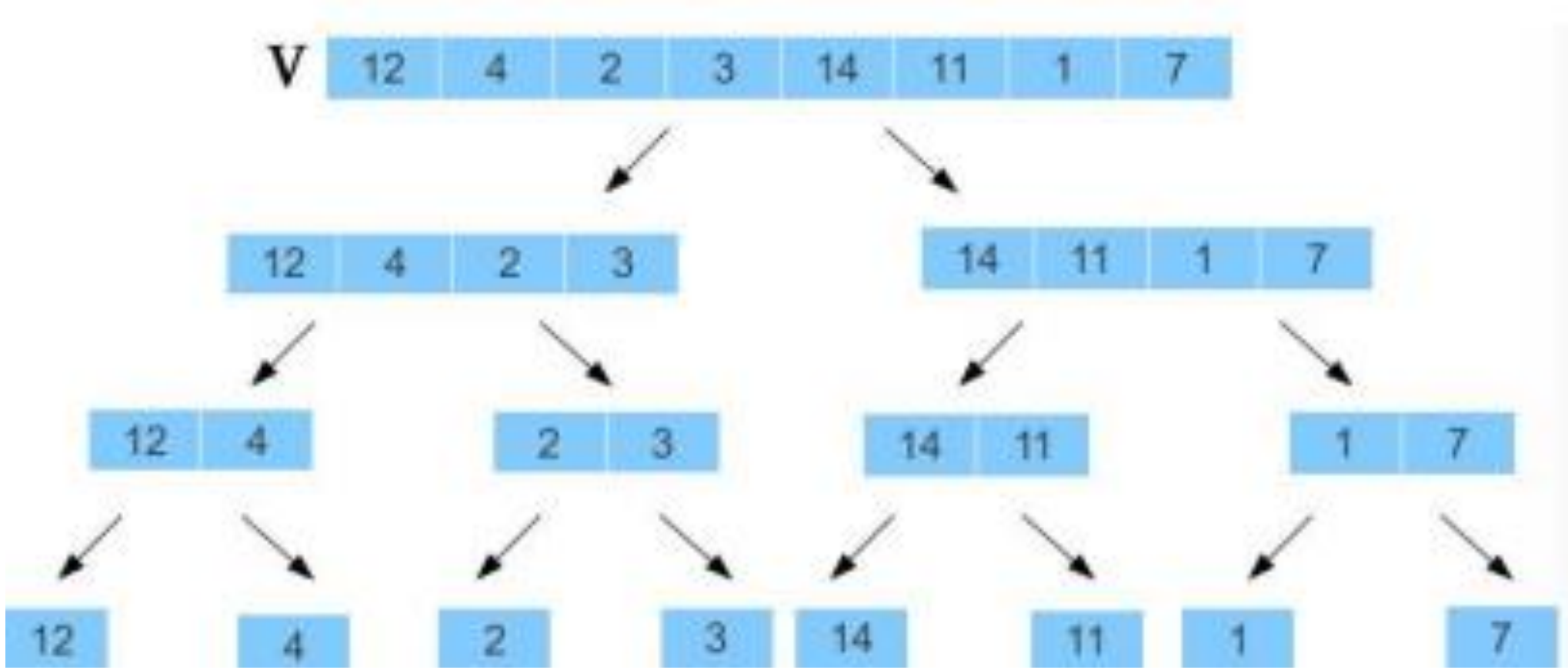
MergeSort

- Dividir e Conquistar;
- Divide, recursivamente, o conjunto de dados até que o subconjunto possua 1 elemento
- Combina 2 subconjuntos de forma a obter 1 conjunto maior e ordenado
- Esse processo se repete até que exista apenas 1 conjunto.

- Divida o vetor em 2 subvetores (ao meio) recursivamente até ele ter o tamanho 1. Intercale pares de elementos adjacentes.
- Repita esse processo até restar apenas um arquivo de tamanho n.

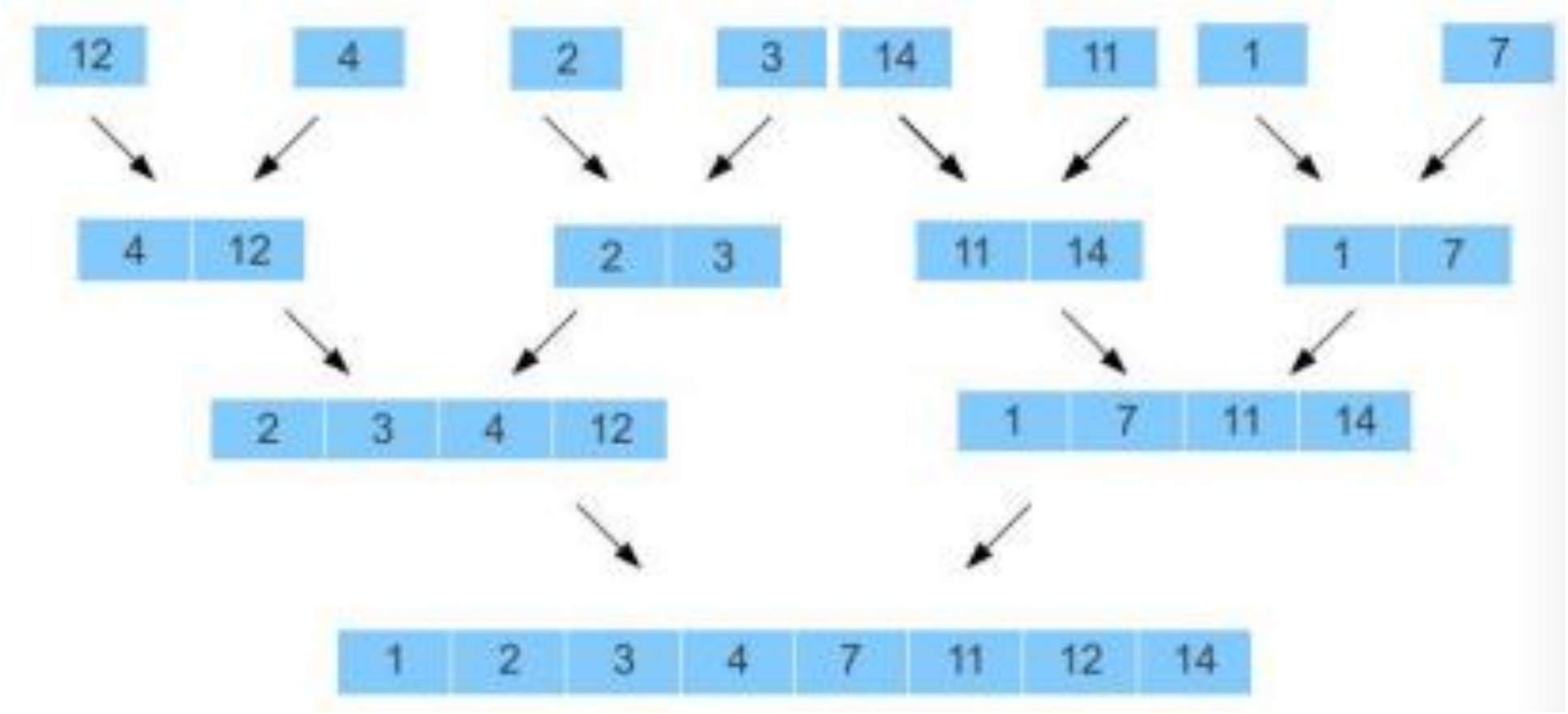
Complexidade - Algoritmos Ordenação

MergeSort



Complexidade - Algoritmos Ordenação

MergeSort



Complexidade - Algoritmos Ordenação

MergeSort

```
função mergesort (vetor a)
     se (n == 1) retornar a

     vetor l1 = a[0] ... a[n/2]
     vetor l2 = a[n/2 + 1] ... a[n]

     l1 = mergesort(l1)
     l2 = mergesort(l2)

     retornar mesclar(l1, l2)
fim da função mergesort
```

```
função mesclar (vetor a, vetor b)
     vetor c

     enquanto (a e b têm elementos)
          if (a[0] > b[0])
               adicionar b[0] ao final de c
               remover b[0] de b
          senão
               adicionar a[0] ao final de c
               remover a[0] de a
     enquanto (a tem elementos)
          adicionar a[0] ao final de c
          remover a[0] de a
     enquanto (b tem elementos)
          adicionar b[0] ao final de c
          remover b[0] de b
     retornar c
fim da função mesclar
```

Complexidade - Algoritmos Ordenação

MergeSort

```
void mergeSort(int vetor[], int comeco, int fim){
    if (comeco < fim) {
        int meio = (fim+comeco)/2;

        mergeSort(vetor, comeco, meio);
        mergeSort(vetor, meio+1, fim);
        merge(vetor, comeco, meio, fim);
    }
}
```

Complexidade - Algoritmos Ordenação

MergeSort

```
void merge(int vetor[], int inicio, int meio, int fim) {
int com1 = inicio, com2 = meio+1, comAux = 0, vetAux[fim-inicio+1];
while(com1<=meio && com2<=fim){
if(vetor[com1] <= vetor[com2]){
vetAux[comAux] = vetor[com1];
com1++;
}else{
vetAux[comAux] = vetor[com2];
com2++; }
comAux++; }
while(com1<=meio){ //Caso ainda haja elementos na primeira metade
vetAux[comAux] = vetor[com1];
comAux++;com1++; }
while(com2<=fim){ //Caso ainda haja elementos na segunda metade
vetAux[comAux] = vetor[com2];
comAux++;com2++; }
for(comAux=inicio;comAux<=fim;comAux++){ //Move os elementos de volt
//para o vetor original
vetor[comAux] = vetAux[comAux-inicio];
}
```

Complexidade - Algoritmos Ordenação

MergeSort

Complexidade de tempo: $\Theta(n \lg n)$
Comparações: $\Theta(n \lg n)$
Trocas: $\Theta(n \lg n)$

O pior caso e o melhor caso tem a mesma complexidade.

Utiliza mais memória para poder ordenar (vetor auxiliar).

Complexidade - Algoritmos Ordenação

Comparação

- Em comparação a outros algoritmos de divisão e conquista, como o Quicksort, o Merge apresenta uma complexidade semelhante.

- Já em comparação a algoritmos mais básicos de ordenação por comparação e troca (bubble, insertion e selection sort), o Merge é mais rápido e eficiente quando é utilizado sobre uma grande quantidade de dados.

- Para entradas pequenas os algoritmos de ordenação por comparação mais básicos são pró-eficientes.

Complexidade - Algoritmos Ordenação

Comparação

Algoritmo	Tempo		
	Melhor	Médio	Pior
Merge sort	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$
Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$

FIM!