Complexidade Assintótica dos Algoritmos

- Os limites assintóticos são usados para **estimar a eficiência dos algoritmos**, avaliando-se a quantidade de memória e de tempo necessário para realizar a tarefa para o qual os algoritmos foram projetados.
- **Complexidade de tempo:** Mede **o número de atribuições** e **comparações** realizadas durante a execução de um programa.

Complexidade Assintótica dos Algoritmos

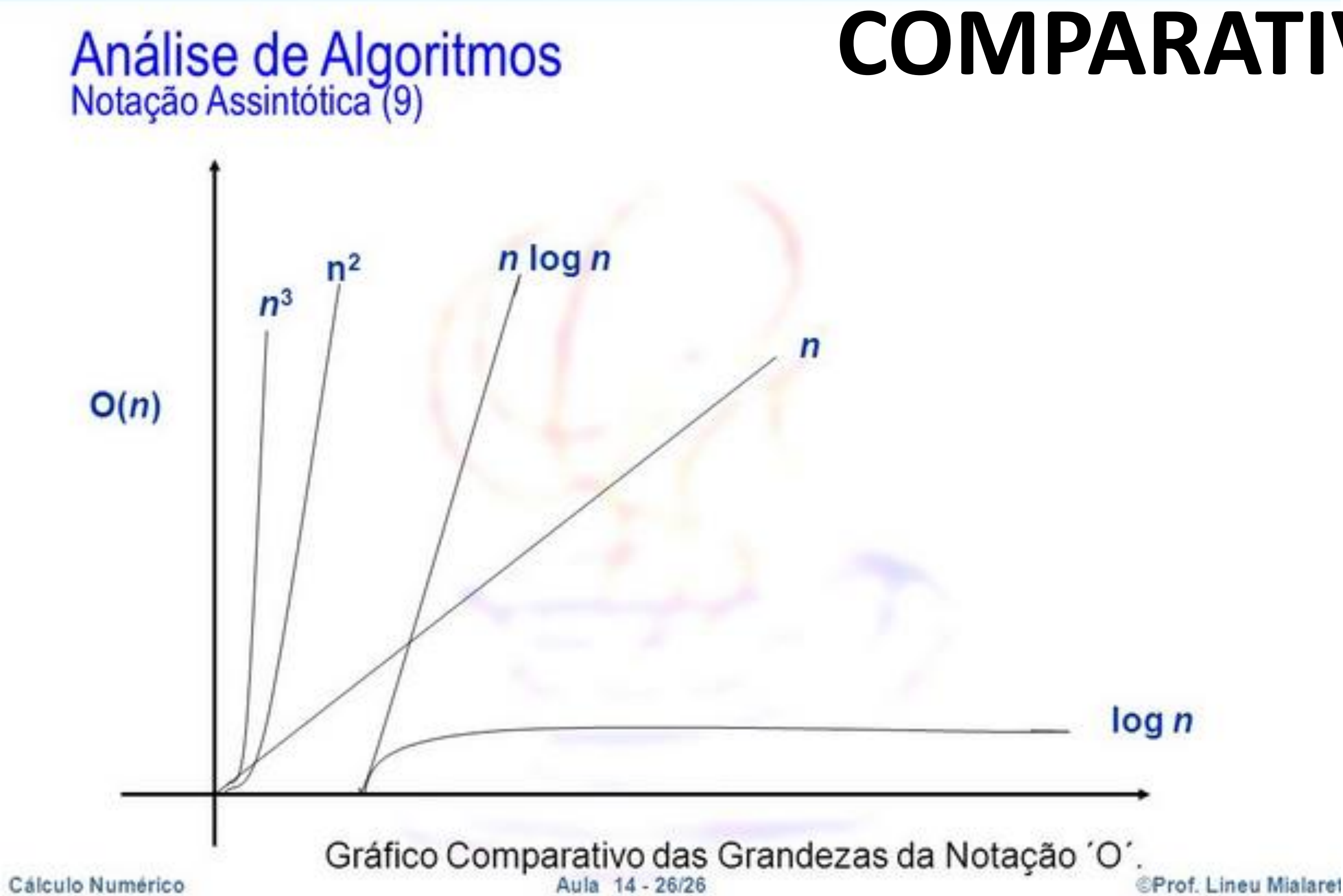
Exemplo 01: Laço simples para calcular a soma dos números em um vetor:

```
soma =0;
for(i=0;i<n;i++)
     soma+=a[i];
```

Primeiro, 2 variáveis são inicializadas, depois o laço itera N vezes. Cada iteração executa 2 atribuições , uma atualiza *"soma"* e a outra atualiza o *"i"*. Logo existem ***2 + 2xN atribuições***.

Sua Complexidade Assintótica é de ***O(n)***.

COMPARATIVO



[SCC-501-Aula_7a_-_Análise_de_algoritmos_-_parte_2.pdf]

[Caso Fibonnaci]

Ordenação pelo Método da Bolha Bubblesort

- Algoritmo:
  - Percorra o vetor inteiro comparando elementos adjacentes (dois a dois)
  - Troque as posições dos elementos se eles estiverem fora de ordem
  - Repita os dois passos acima com os primeiros n-1 itens, depois com os primeiros n-2 itens, até que reste apenas o um item

Exemplo – Bolha Bubblesort

I = 4

4	**3**	**1**	**7**	**2**
j				

3	**4**	**1**	**7**	**2**
	j			

3	**1**	**4**	**7**	**2**
		j		

3	**1**	**4**	**7**	**2**
			j	

3	**1**	**4**	**2**	**7**
				j

I = 3

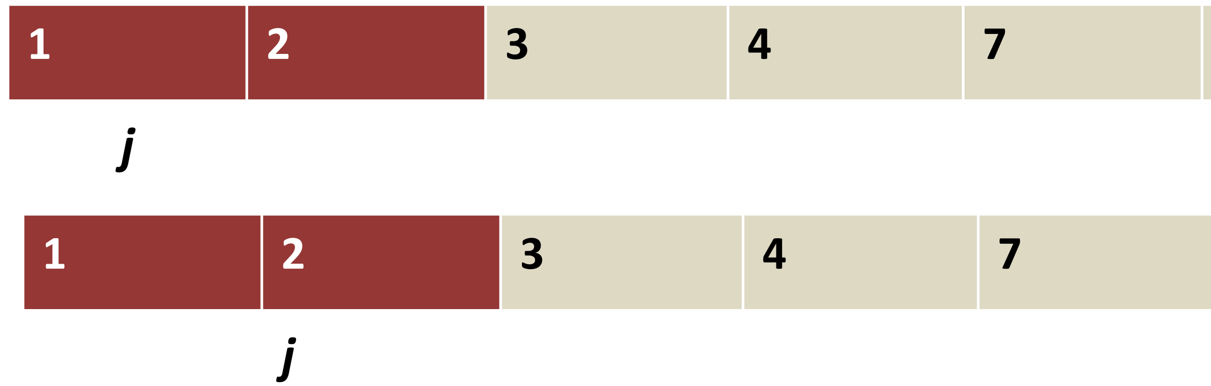
3	1	4	2	7
j				
1	3	4	2	7
	j			
1	3	4	2	7
		j		
1	3	2	4	7
			j	

I = 2

1	3	2	4	7
j				
1	3	2	4	7
	j			
1	2	3	4	7
		j		

- ....

I = 1



```
void bolha(int quant, int* v){
    int i, j;
    for(i=quant-1; i>=1; i--){
        for (j=0; j<i; j++){
                if (v[j] > v[j+1]){
                        int temp = v[j];
                        v[j] = v[j + 1];
                        v[j+1] = temp;
                }
        } } }
```

?

- Na situação:

I = 4

1	**2**	**3**	**7**	**4**
j				

- Quantas comparações vão acontecer?
- Todas são necessárias neste caso?

?

- Número de operações não se altera se vetor já está (parcialmente) ordenado
  - Como melhorar?

```
void bolha(int quant, int* v){
    int i, j;
    int troca;
    for(i=quant-1; i>=1; i--){
        troca = 0;
        for (j=0; j<i; j++){
            if (v[j] > v[j+1]){
                int temp = v[j];
                v[j] = v[j + 1];
                v[j+1] = temp;
                troca = 1;
            }  }
        if (troca==0){
            break;
        } }  }
```

Método da Bolha Melhorado: Termina execução quando nenhuma troca é realizada após uma passada pelo vetor

BUBBLE SORT: CRITÉRIOS DE ANÁLISE

- Sendo ***n*** o número registros no arquivo, as medidas de complexidade relevantes são: Número de comparações ***C(n)*** entre chaves. Número de movimentações ***M(n)*** de registros do arquivo.

- Análise informal com base na operação mais custosa, que é a comparação
  - n etapas no pior caso
  - Na primeira etapa, são feitas n-1 comparações e trocas, na segunda n-2 e assim por diante.
  - Temos então, a soma dos termos de uma progressão aritmética:
    - $(n-1) + (n-2) + \ldots + 1$
    - $Total = \frac{n \cdot (n-1)}{2} \Rightarrow O(n^2)$

- RESUMINDO...

Bubble Sort: o tempo gasto na execução do algoritmo varia em ordem quadrática em relação ao número de elementos a serem ordenados.

- **$T = O(n^2)$** – Notação “Big O”
- Atividades mais custosas:
  - Comparações
  - Trocas de Posição (swap)

- Melhor caso: vetor ordenado
- Pior caso:vetor invertido
- Método muito simples, mas custo alto:
  - Adequado apenas se arquivo pequeno
  - Ruim se registros muito grandes

ORDENAÇÃO POR INSERÇÃO

- Ordena um vetor da esquerda para a direita, por ordem crescente ou decrescente.
- À medida que o vetor vai sendo percorrido ele deixa seus elementos à esquerda ordenados.

ORDENAÇÃO POR INSERÇÃO

- Percorre um vetor, sempre a partir do primeiro índice desordenado (inicialmente o 2º elemento), e em seguida procura inseri-lo na posição correta, comparando-o com o seu(s) anterior(s) e trocando-os de lugar enquanto ele for menor que seu(s) precedente(s).

ORDENAÇÃO POR INSERÇÃO

temp = 5

10	5	6	1	3
j	*i*			

J=-1

V[j+1]=
temp

5	10	6	1	3

temp = 6

5	10	6	1	3
	j	*i*		

J=0

V[j+1]=
temp

5	6	10	1	3
j				

temp = 1

5	6	10	1	3
		j	*i*	

ORDENAÇÃO POR INSERÇÃO

temp = 1

5	6	10	1	3
		j	*i*	

5	6	10	10	3
	j		*i*	

5	6	6	10	3
j			*i*	

J=-1

5	5	6	10	3
			i	

V[j+1]=temp

1	5	6	10	3
			i	

ORDENAÇÃO POR INSERÇÃO

temp=3

1	5	6	10	3
			j	*i*

1	5	6	6	10
		j		*i*

1	5	5	6	10
j				*i*

V[j+1]=temp

1	3	5	6	10
j				*i*

ORDENAÇÃO POR INSERÇÃO

```
int myarray[]={10,5,6,1,3};
void ord_insercao(int v[],int tam){
    int j, i, temp;
*    for (i=1;i<tam;i++)
     {    //guarda o elemento que esta verificando

          temp = v[i];
          //verificando os elementos anteriores a posicao i
          j=i-1;
**        while (v[j]>temp && j>=0)
           {
                v[j+1]=v[j];
                j--;
           }
           //insere o elemento na posicao correta (ordenada) ate i.

           v[j+1] = temp;
        }
}
```

ORDENAÇÃO POR INSERÇÃO

- Melhor caso (itens já ordenados):

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 = O(n)$$

- Pior caso (itens em ordem reversa):
  - $(n\text{-}1) + (n\text{-}2) + \ldots + 1$
  - $Total = \frac{n \cdot (n-1)}{2} \Rightarrow O\left(n^2\right)$

QuickSort

- O algoritmo Quick Sort é um método de ordenação muito rápido e eficiente;

- Idéia: semelhante a um dicionário, ordena-se as palavras, tendo como objetivo reduzir o problema original em subproblemas para assim poder ser resolvido mais fácil e rapidamente.

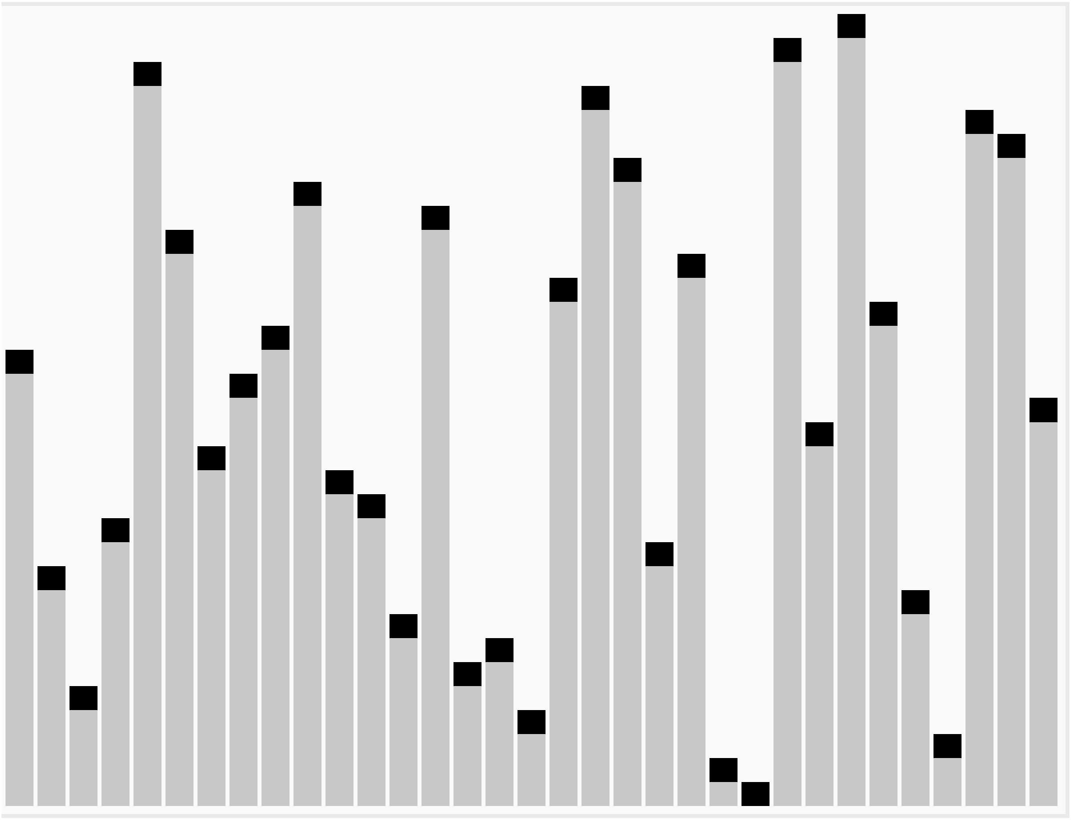
Quicksort

- Este método divide a tabela em duas sub-tabelas, a partir de um elemento chamado **pivô**.

- **Uma das sub-tabelas contém os elementos menores que o pivô** enquanto a **outra contém os maiores. O pivô é colocado entre ambas**, ficando na posição correta.

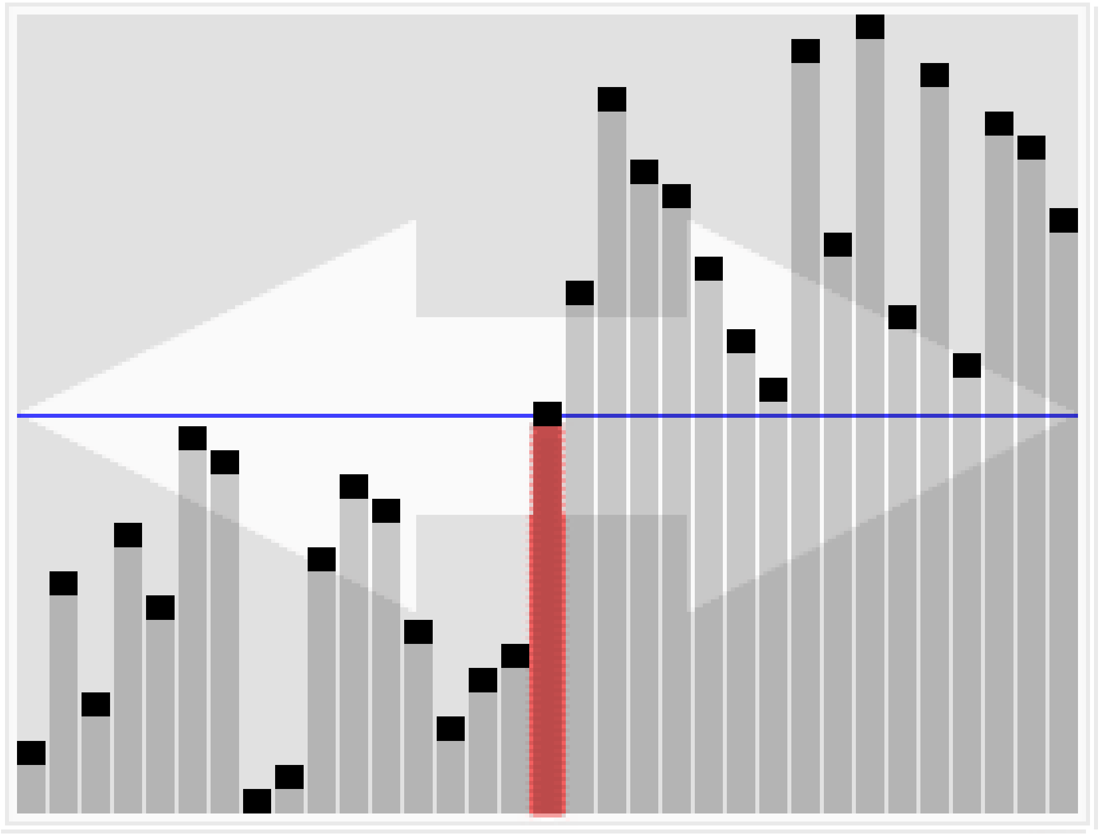
QuickSort

- EXEMPLO: Essas barras de tamanhos diferentes devem ser alinhadas em ordem crescente:



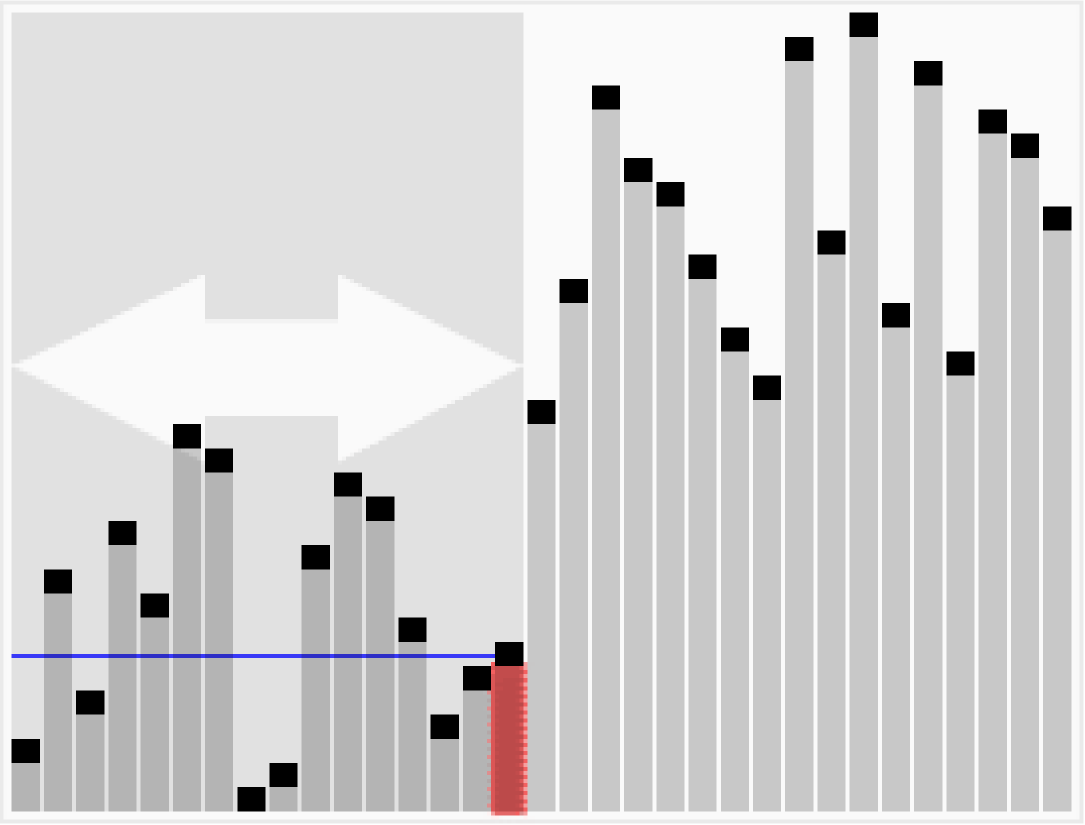
QuickSort

- Os elementos são organizados de maneira que os menores ficam do lado esquerdo do **pivô** e os maiores do lado direito (são as duas sub-tabelas).



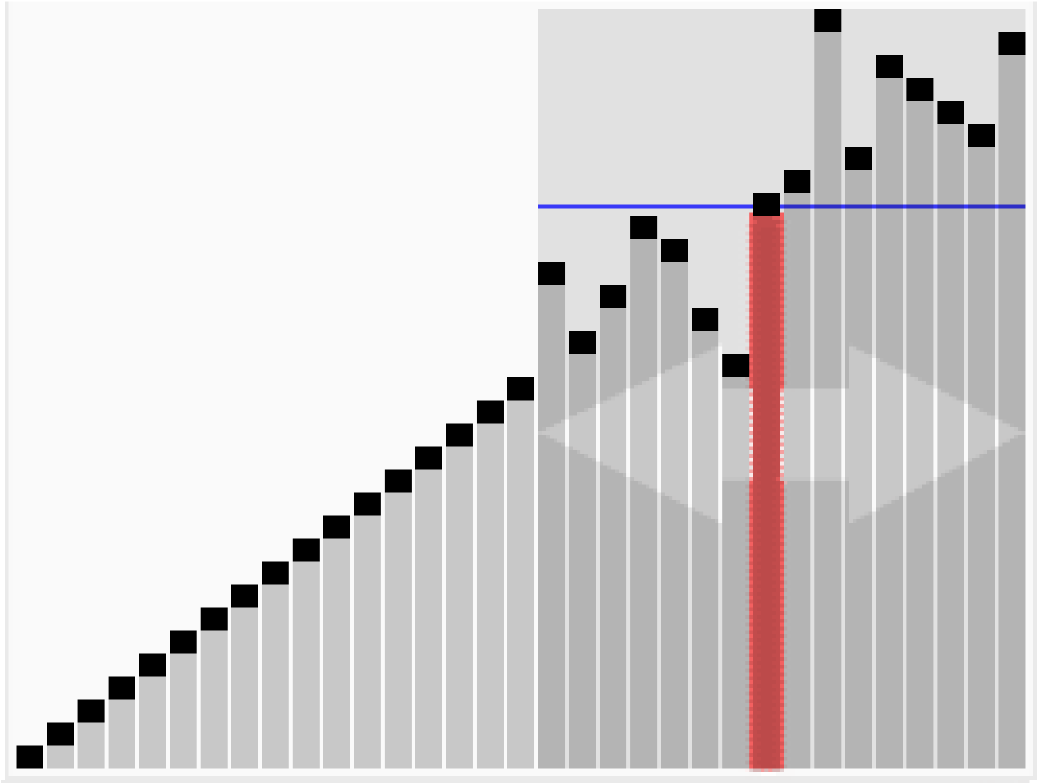
QuickSort

- Depois de encontrar a posição do **pivô** e separar em duas tabelas, ele passa para uma das sub-tabelas e escolhe outro **pivô**.



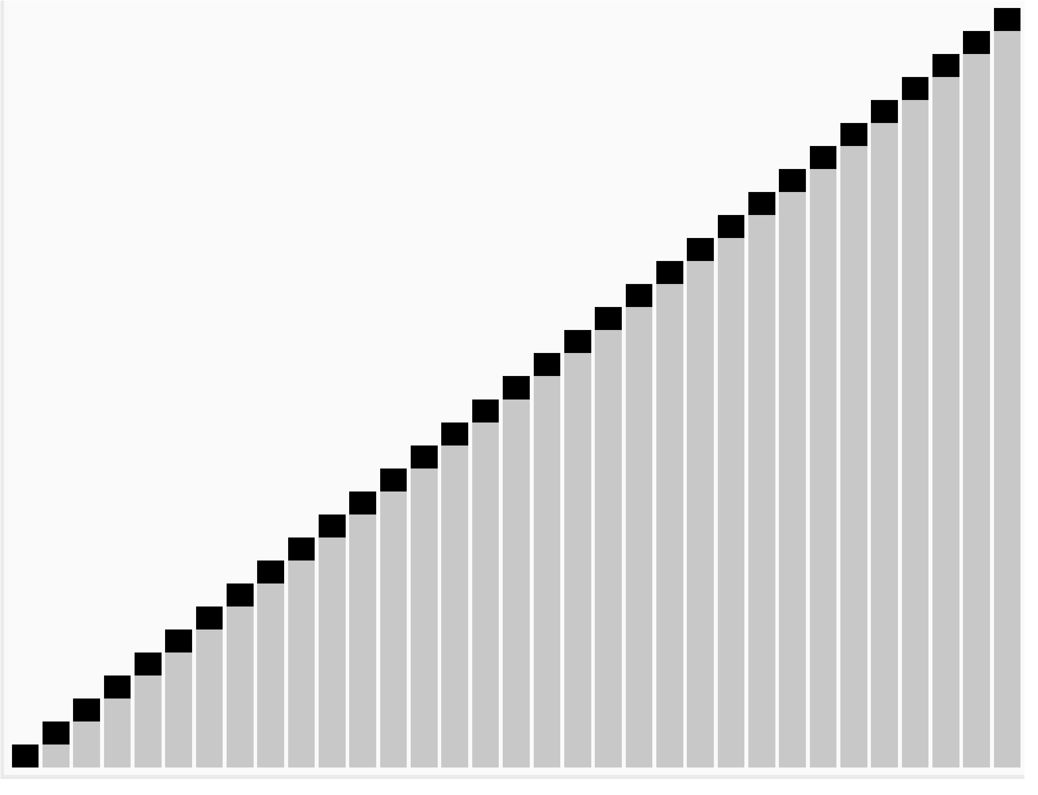
QuickSort

- O pivô do início e todo o lado esquerdo já está ordenado, Agora ele passa para a outra sub-tabela, o lado direito do primeiro **pivô**, escolhe outro **pivô** e ordena.



QuickSort

- E depois de ordenado ele fica assim:



QuickSort

```
#include<stdio.h>
#include<iostream>
using namespace std;
void Quick(int vetor[10], int inicio, int fim);
int main(){
    int vetor[6] = {7, 9, 4, 3,6,1};
    int i;
    Quick(vetor, 0, 5);
    printf("\n2.Vetor ordenado:\n");
    for(i = 0; i <= 5; i++){
        printf("%d ", vetor[i]);
    }
    printf("\n");
}
```

```c
void Quick(int vetor[10], int inicio, int fim){
    int pivo, aux, i, j, meio;
    i = inicio;
    j = fim;
    meio = (int) ((i + j) / 2);
    pivo = vetor[meio];
    do{
        while (vetor[i] < pivo) i = i + 1;
        while (vetor[j] > pivo) j = j - 1;
        if(i <= j){
            aux = vetor[i];
            vetor[i] = vetor[j];
            vetor[j] = aux;
            i = i + 1;
            j = j - 1;
        }
    }while(j > i);
    if(inicio < j) {
     Quick(vetor, inicio, j);
    }
    if(i < fim) {
     Quick(vetor, i, fim);
    }
}
```

Quicksort

7	**9**	**4**	**3**	**6**	**1**
I					j

1	**9**	**4**	**3**	**6**	**7**
	i		j		

1	**3**	**4**	**9**	**6**	**7**
		I ,j			

Quicksort

1	3	4	9	6	7
i		j	i		

1	3	4	9	6	7
	I,j		i		

1	3	4	9	6	7
j		i	i		

1	3	4	9	6	7
	i	i			j

1	3	4	9	6	7
	i		i		j

QuickSort

1	3	4	9	6	7
	i		i		j

1	3	4	7	6	9
	i			l,j	

1	3	4	7	6	9
	i		j		i

1	3	4	7	6	9
	i	i		j	

1	3	4	7	6	9
	i		i	j	

1	3	4	6	7	9
	i		j	i	

QuickSort

1	3	4	6	7	9
	i			i	j

1	3	4	6	7	9
	i			l,j	

1	3	4	6	7	9
			j		i

QuickSort

- É extremamente eficiente para ordenar arquivos de dados. O método necessita apenas de uma pequena pilha como memória auxiliar e requer ***nlogn*** operações, em média, para ordenar ***n*** itens.
- ***C(n) = 2C(n/2)+ n-1***

, onde C(n/2) representa o custo de ordenar uma das metades e n-1 é o número de comparações realizadas.

C(n) ≈ nlogn

Merge Sort

- O princípio básico de funcionamento do Merge Sort é simples e intuitivo: o algoritmo divide a lista a ser ordenada em várias sublistas, sempre em duas partes, até que as listas restantes contenham apenas um elemento.

- Após esta divisão o algoritmo faz o caminho inverso, ou seja, ele começa a "remontar" o vetor, porém ordenando os elementos (que estão em menor número) enquanto os vetores maiores são recriados.

Merge Sort

- Ex. Vetor: 15 99 52 14 50 64 20 77

1. Divisão em duas partes: 15 99 52 14 - 50 64 20 77

2. Divisão da primeira parte em duas outras partes: 15 99 - 52 14

3. Análise do primeiro par: 15- 99
   Como 15 é menor do que 99, a ordem é mantida.

4. Análise do segundo par: 52 -14
   Como 52 é maior do que 14, a ordem é invertida: 14 -52

5. Agora restam dois pares ordenados: 15 99 - 14 52

Merge Sort

- O algoritmo agora compara o menor valor de cada um dos dois vetores: 15 -14
- Como 14 é menor do que 15, logo 14 será o 1º elemento no vetor auxiliar: 14 X X X
- Compara-se 15 com o outro elemento do segundo vetor. Como 15 é menor do que 52: 14 15 X X
- Agora simplesmente comparam-se os dois elementos restantes. 99 é maior do que 52, logo 52 ocupará a terceira posição, e 99 a quarta posição neste vetor de 4 elementos: 14 15 52 99
- O processo é repetido de forma idêntica para a outra primeira metade do vetor inicial: 20 50 64 77
- O próximo e último passo é fazer a fusão destes dois vetores,
- Feito este processo, o resultado final será:

14 15 20 50 52 64 77 99

Merge Sort

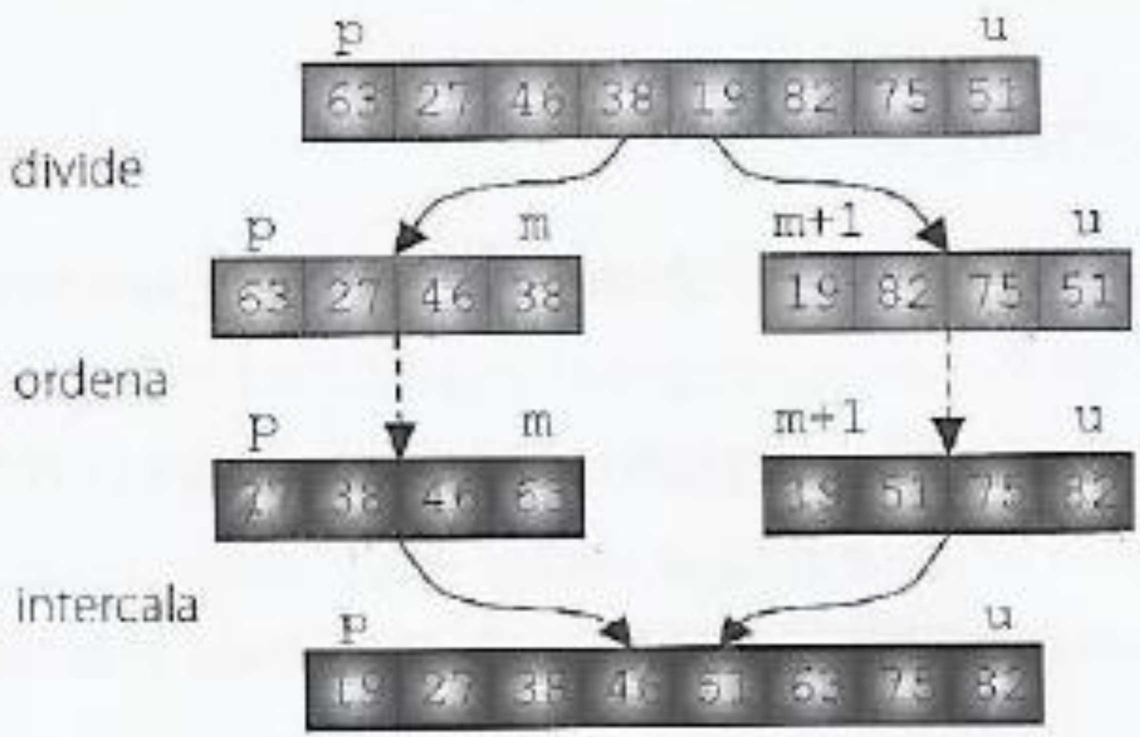


Figura 8.14 | Estratégia recursiva para ordenação por intercalação.

Merge Sort

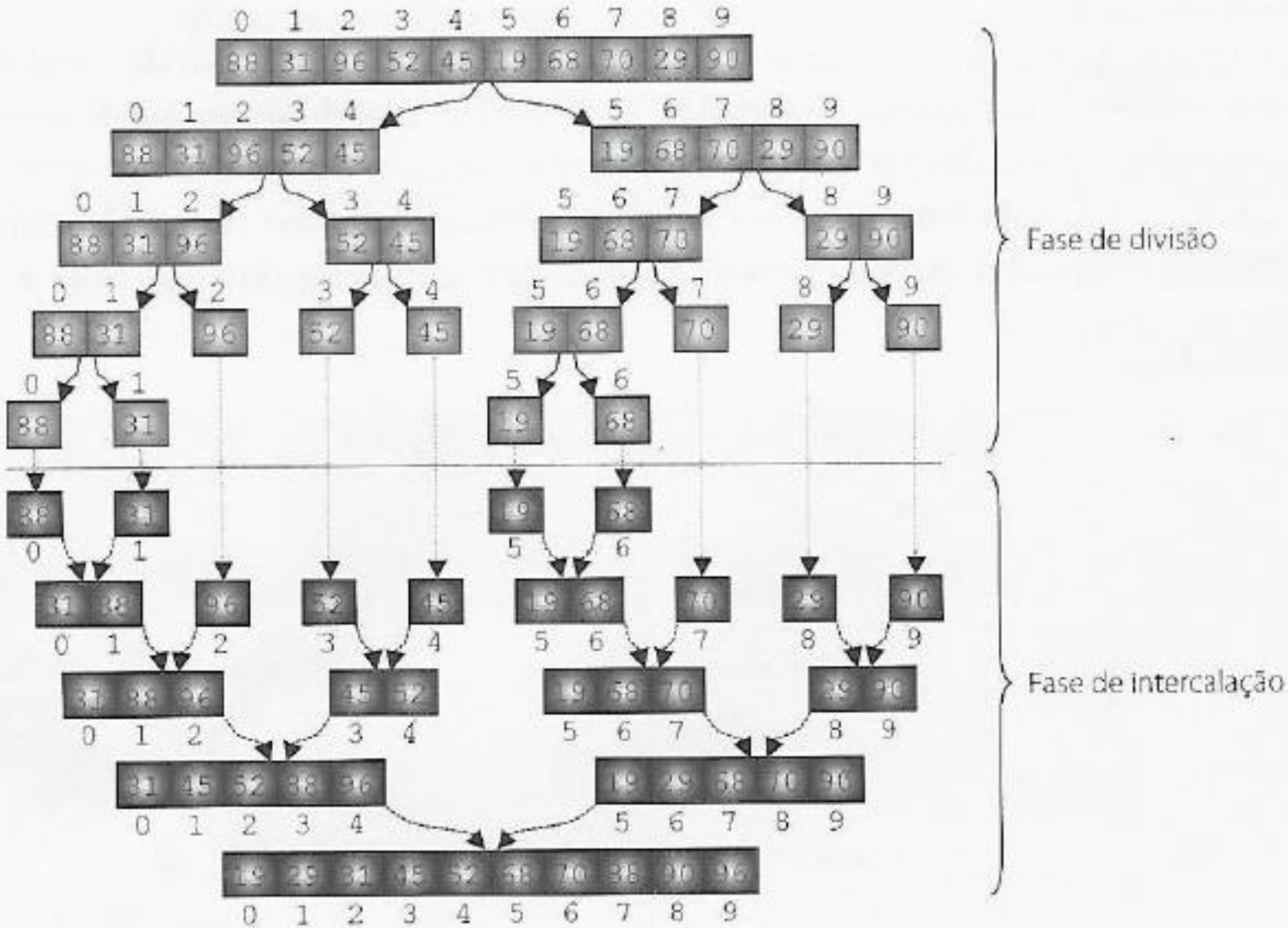


Figura 8.15 | Funcionamento da ordenação por intercalação.

Merge Sort

```
void merge_sort(int v[], int p, int u) {
   if( p == u ) return;
   int m = (p+u)/2;
   merge_sort(v,p,m);
   merge_sort(v,m+1,u);
   intercala(v,p,m,u);
}
```

Merge Sort

```
void intercala(int v[], int p, int m, int u) {
   int *w = malloc((u-p+1)*sizeof(int));
   if( w == NULL ) abort();
   int i = p, j = m+1, k = 0;
   while( i<=m && j<=u )
      if( v[i] < v[j] ) w[k++] = v[i++];
      else w[k++] = v[j++];
   while( i<=m ) w[k++] = v[i++];
   while( j<=u ) w[k++] = v[j++];
   for(k=0; k<=u-p; k++) v[p+k] = w[k];
   free(w);
}
```

Figura 8.13 | Função para intercalação.

Merge Sort

Seja $n = u - p + 1$ o tamanho do vetor. A cada divisão, esse tamanho é reduzido à metade. Então, começando com n itens, após a primeira divisão, teremos subvetores com cerca de $n/2$ itens; após a segunda, $n/2^2$; após a terceira, $n/2^3$; e assim por diante, até que, após a k-ésima divisão, teremos $n/2^k$ itens. Suponha que, após a k-ésima divisão, reste apenas um item em cada subvetor (nesse caso, eles já estarão ordenados). Mas, se $n/2^k = 1$, segue que $2^k = n$, ou seja, $k = \lg n$. Então, o número máximo de níveis na fase de divisão (ou intercalação) é $O(\lg n)$. Como, em cada nível de intercalação são intercalados, no total, n itens, a complexidade de tempo da ordenação por intercalação, no pior caso, é $O(n \lg n)$.

Pesquisa Binária

- entrada: vetor *vet* com *n* elementos, ordenado
  elemento *elem*
- saída: n se o elemento *elem* ocorre em vet[n]
  -1 se o elemento não se encontra no vetor
- procedimento:
  - compare *elem* com o elemento do meio de *vet*
  - se *elem* for menor, pesquise na primeira metade do vetor
  - se *elem* for maior, pesquise na segunda parte do vetor
  - se for igual, retorne a posição
  - continue o procedimento, subdividindo a parte de interesse, até encontrar o elemento ou chegar a uma parte do vetor com tamanho 0

```c
int busca_bin (int n, int* vet, int elem)
{
   /* no início consideramos todo o vetor */
   int ini = 0;
   int fim = n-1;

   /* enquanto a parte restante for maior que zero */
   while (ini <= fim) {
      int meio = (ini + fim) / 2;
      if (elem < vet[meio])
         fim = meio – 1;  /* ajusta posição final */
      else if (elem > vet[meio])
         ini = meio + 1;  /* ajusta posição inicial */
      else
         return meio;     /* elemento encontrado */
   }

   /* não encontrou: restou parte de tamanho zero */
   return -1;
}
```

- **COMPLEXIDADE PESQUISA BINÁRIA**
- Se a chave está no meio da matriz: o laço executa somente uma vez
- Quantas vezes o laço executa no caso em que a chave não está na matriz?
  - **1º:** busca na matriz inteira de tamanho **n**
  - **2º: n/2**
  - **3º:** $\mathbf{n/2^{2}}$
  - **4º.:**$\mathbf{n/2^{3}}$
  - **Última vez:** $\mathbf{n/2^{m}}$
  - , e assim por diante, até que a matriz seja de tamanho **1**.

- $n/2^m = 1 \rightarrow$
- $2^m = n \rightarrow$
- *OBS: Usando a regra logarítmica:*

$$a^{\log_a b} = b$$

- Logo:
- $2^m = n \rightarrow 2^m = 2^{\text{Log } n}$
- Logo ***m = log n***