Estrutura de Dados – 1º semestre de 2020

Professor Mestre Fabio Pereira da Silva

Divisão e Conquista

- Construção incremental
- Consiste em, inicialmente, resolver o problema para um subconjunto dos elementos da entrada e, então adicionar os demais elementos um a um.
- Em muitos casos, se os elementos forem adicionados em uma ordem ruim, o algoritmo não será eficiente.
- Ex: Calcule n!, recursivamente

Divisão e Conquista

- Dividir o problema em determinado número de subpro-blemas.
- Conquistar os subproblemas, resolvendo os recursivamente.
- Se o tamanho do subproblema for pequeno o bastante, então a solução é direta.
- Combinar as soluções fornecidas pelos subproblemas, a fim de produzir a solução para o problema original.

Quick Sort

- O Quick Sort usa do mesmo princípio de divisão que o Merge Sort, entretanto, **o mesmo não utiliza a intercalação**, uma vez que não subdivide a dada estrutura em muitas menores.
- Esse algoritmo simplesmente **faz uso de um dos elementos da estrutura linear** (determinada pelo programador) como parâmetro inicial, denominado pivô.

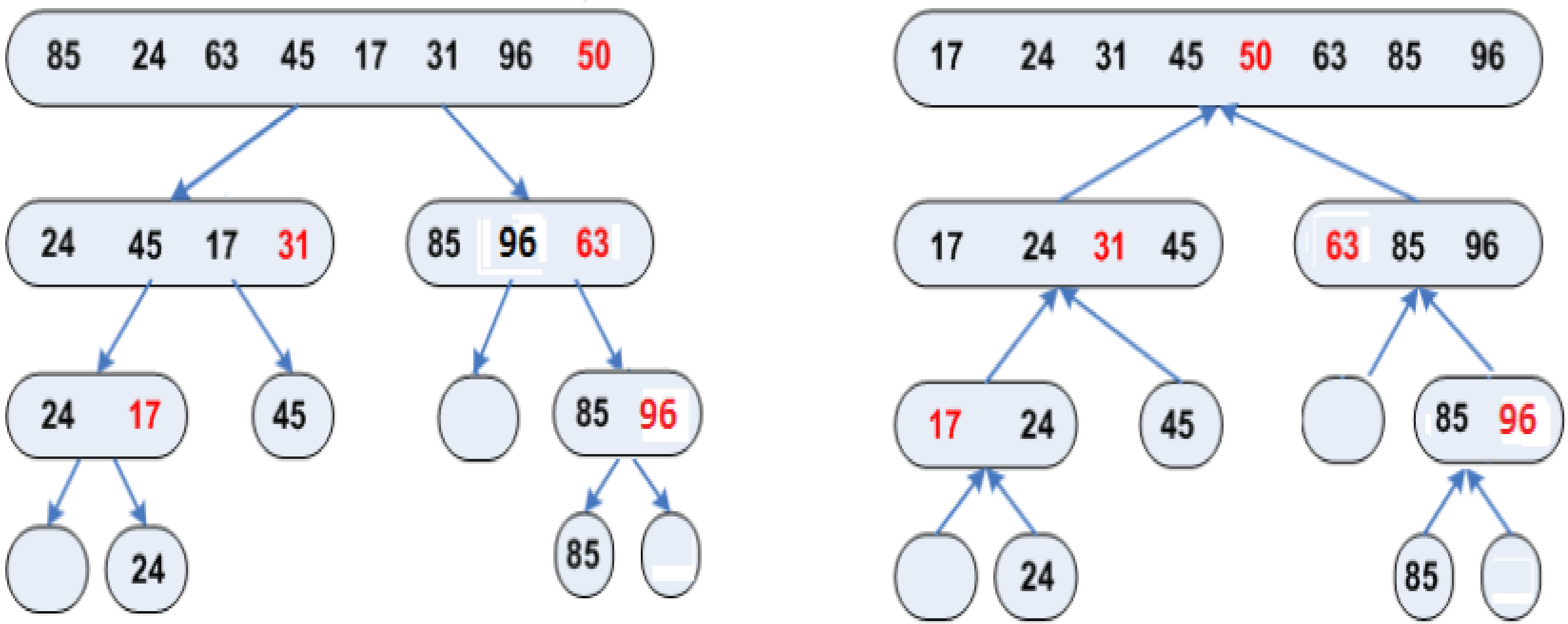
Quick Sort

- Com o pivô definido, o algoritmo irá dividir a estrutura inicial em duas, a primeira, à esquerda, contendo todos os elementos de valores menores que o pivô, e, à direita, todos os elementos com valores maiores.
- Em seguida, o mesmo procedimento é realizado com o a primeira lista (valores menores<pivô<valores maiores). **O mesmo processo se repete até que todos os elementos estejam ordenados**
- **Classificação por troca**

Quick Sort

- Quick Sort é um algoritmo aleatório baseado no paradigma de divisão e conquista
- Divisão: pegue um elemento x aleatório (chamado pivô) e particione S em
  - L elementos menor que x
  - E elementos igual a x
  - G elementos maiores que x
- Recursão: ordene L e G
- Conquista: junte L, E e G

Quick Sort



(a) Fase de Divisão

(b) Fase de Conquista

Algorítmo Quick Sort

Dados: $v[0], v[1], \ldots, v[n-1]$

qsort(l, u) : // ordenar $v[l \ldots u]$

se $l \geq u$ então termina imediatamente;

// caso contrário:

$m \leftarrow$ partition(l, u) // partir usando pivo

qsort$(l, m-1)$ // ordenar $v[l \ldots m-1]$

qsort$(m+1, u)$ // ordenar $v[m+1 \ldots u]$

Particionamento

- Mecanismo principal dentro do algoritmo do Quick Sort
- Para particionar um determinado conjunto de dados, separamos de um lado todos os itens cuja as chaves sejam maiores que um determinado valor, e do outro lado, colocamos todos os itens cuja as chaves sejam menores que um determinado valor
  - Exemplo: Dividir as fichas de empregados entre quem mora a menos de 15km de distância da empresa e quem mora a uma distância acima de 15km

Particionamento

- Apesar de termos dois grupos de valores, não quer dizer que os valores estejam ordenados nestes grupos.
- Porém, só o fato de estarem separados pelo pivô numa classificação de maior/menor que o pivô, já facilita o trabalho de ordenação.
- A cada passo que um novo pivô é escolhido os grupos ficam mais ordenados do que antes.

Particionamento

- O algoritmo trabalha começando com **2 "ponteiros", um em cada ponta do array**
- O "ponteiro" da esquerda **leftPtr** move-se para a direita e o "ponteiro" da direita **rightPtr** movese para a esquerda
- **LeftPtr** é inicializado com o índice zero e será incrementado e **rightPtr** é inicializado com índice do último elemento do vetor e será decrementado

Estratégias de escolha do Pivô

- Primeiro elemento
- Último elemento
- Elemento do meio
- Elemento aleatório
- Mediana de 3 (primeiro, meio e último)
- Mediana de 3 (aleatório)

Estratégias de escolha do Pivô

- Primeiro elemento.
- Pior caso: quando os elementos estão em ordem crescente ou decrescente.
  - Exemplo: | 0 | 1 | 3 | 4 | 5 | 7 | 9 |
- Último elemento.
- Pior caso: quando os elementos estão em ordem crescente ou decrescente.
  - Exemplo: | 9 | 7 | 5 | 4 | 3 | 1 | 0 |
- Elemento do meio.
- Pior caso: quando os elementos formam uma espécie de triângulo.
  - Exemplo: | 1 | 2 | 3 | 4 | 3 | 2 | 1 |
- Elemento aleatório.
- Pior caso: depende da escolha dos índices (índices: 3, 0, 2, 6, 5, 1, 4).
  - Exemplo: | 3 | 8 | 4 | 0 | 9 | 7 | 5 |

Quick Sort em Listas Ligadas

- Nesse caso é interessante tratar o problema da partição como sendo a partição em 3 Listas:
- L1 contendo chaves menores que o pivô.
- L2 contendo chaves maiores que o pivô. Lv contendo chaves iguais ao pivô.
- A ordenação é realizada apenas em L1 e L2 e não em Lv . A concatenação é realizada na forma: S1, Lv , L2.

```
    | 5 | 7 | 5 | 0 | 6 | 5 | 5 |
L1  | 0 |
L2  | 7 | 6 |
Lv  | 5 | 5 | 5 | 5 |
```

Quick Sort em Listas Ligadas

.

```
   | 5 | 7 | 5 | 0 | 6 | 5 | 5 |
L1 | 0 |
L2 | 7 | 6 |
Lv | 5 | 5 | 5 | 5 |
```

Desempenho

- Quick Sort é considerado rápido para realizar ordenação in-place, ou seja, que utiliza apenas movimentações dentro do próprio arranjo, **sem uso de memória auxiliar**.
- É importante prestar atenção à implementação para evitar casos de execução quadrática. Mesmo alguns livros fornecem algoritmos que podem ser lentos em alguns casos.

Desempenho

- Se o particionamento é balanceado, o algoritmo é executado assintoticamente tão rápido quanto o MergeSort.
- Isto é, O(n log n).
- Vantagem adicional em relação ao MergeSort: é in place, isto é, não utiliza um vetor auxiliar. Note-se que basta ser balanceado, não precisa ser o particionamento mais uniforme!
- Contudo, se o particionamento não é balanceado, ele pode ser executado tão lentamente quanto o BubbleSort.

Processo de Ordenação

- Começamos com a sequência:

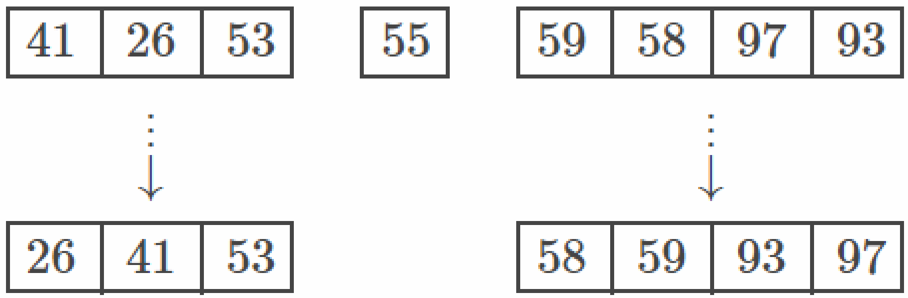
55	41	59	26	53	58	97	93

- Escolhemos o primeiro valor como pivô e reorganizamos os valores:

$$\underbrace{\boxed{41 \mid 26 \mid 53}}_{<55}\;\boxed{\mathbf{55}}\;\underbrace{\boxed{59 \mid 58 \mid 97 \mid 93}}_{>55}$$

Processo de Ordenação

- Recursivamente ordenamos as duas subsequências repetindo este método:



- Sequência final ordenada:

26	41	53	55	58	93	97

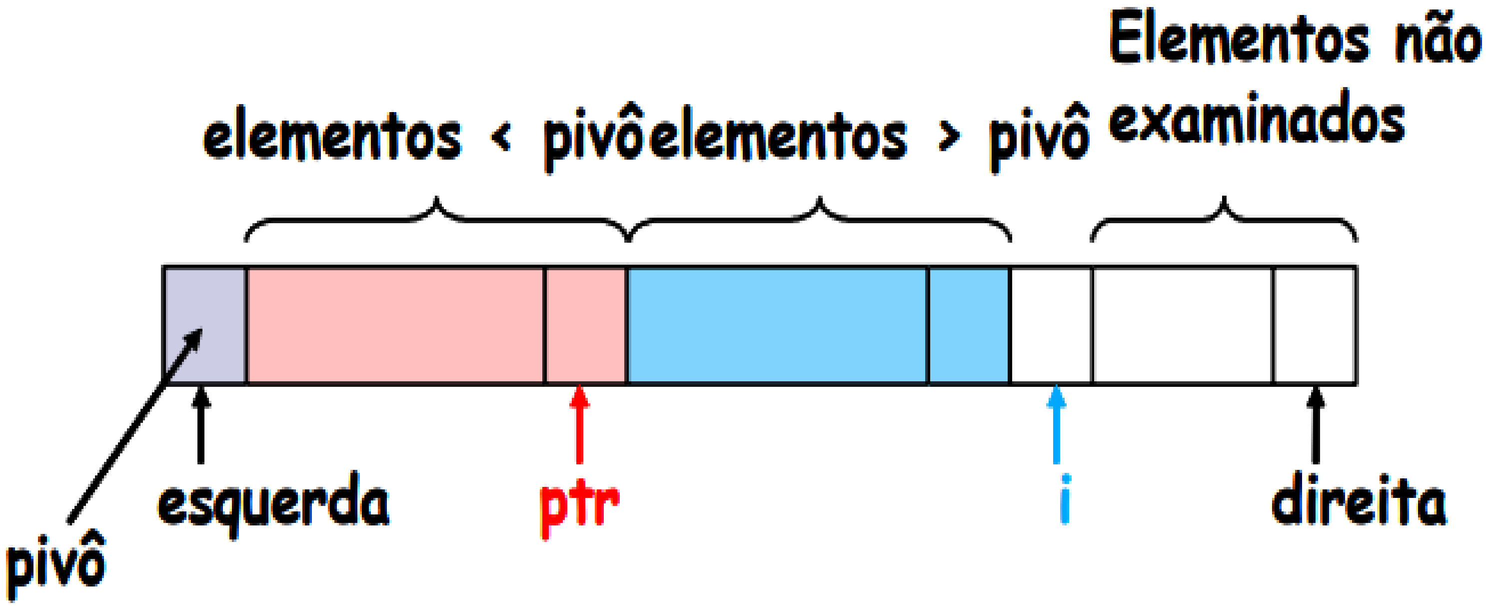
Estratégias de escolha do pivô

- Particionamento do array ou subarray em um grupo de chaves menores (lado esquerdo) e um grupo de chaves maiores (lado direito)
- Chamada recursiva para ordenar/particionar o lado esquerdo
- Chamada recursiva para ordenar/particionar o lado direito

Estratégias de escolha do pivô

- O pivô deve ser algum dos valores que compõem o array O pivô pode ser escolhido aleatoriamente. Para simplificar, como pivô será usado o elemento que está na extrema direita de todo subarray que será particionado
- Após o particionamento, se o pivô é inserido no limite entre os dois subarrays particionados, ele já estará automaticamente em sua posição correta na ordenação

Estratégias de escolha do pivô



Pivô aleatório

Exemplo

Particionar a seguinte lista com pivô = 15:

2	23	89	7	14	35
LP					RP

Pivô aleatório

Exemplo

pivô = 15:

2	23	89	7	14	35

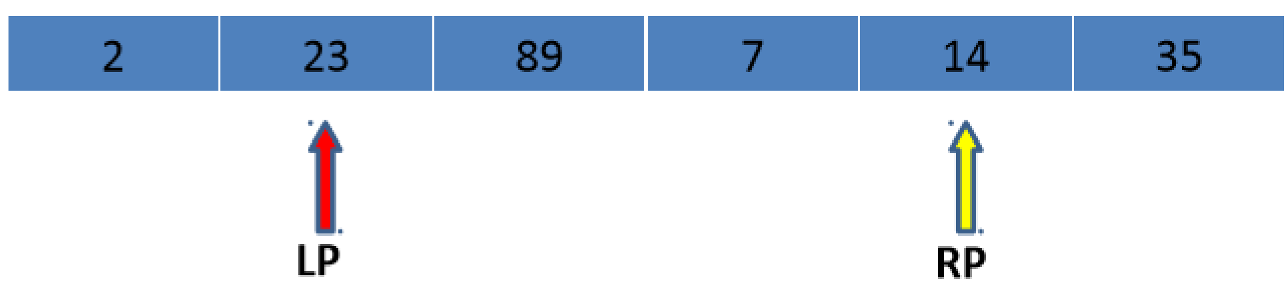
LP RP

23 > 15 logo LP pára e RP começa a se mover!

Pivô aleatório

Exemplo

pivô = 15:



14 < 15 logo RP pára! Logo é necessário fazer a troca dos elementos: ***swap*(1,4)**

Pivô aleatório

Exemplo

pivô = 15:

2	14	89	7	23	35

LP RP

O **LP** volta a caminhar no vetor!

Pivô aleatório

Exemplo

pivô = 15:

2	14	89	7	23	35

89 > 15 logo LP pára e RP volta a se mover!

Pivô aleatório

Exemplo

pivô = 15:

2	14	89	7	23	35

7 < 15 logo RP pára! Logo é necessário fazer a troca dos elementos: ***swap*(2,3)**

Pivô aleatório

Exemplo

pivô = 15:

2	14	7	89	23	35

LP RP

O **LP** volta a caminhar no vetor!

Pivô aleatório

Exemplo

pivô = 15:

2	14	7	89	23	35

LP RP

89 > 15 logo LP pára e RP volta a se mover!

Exemplo

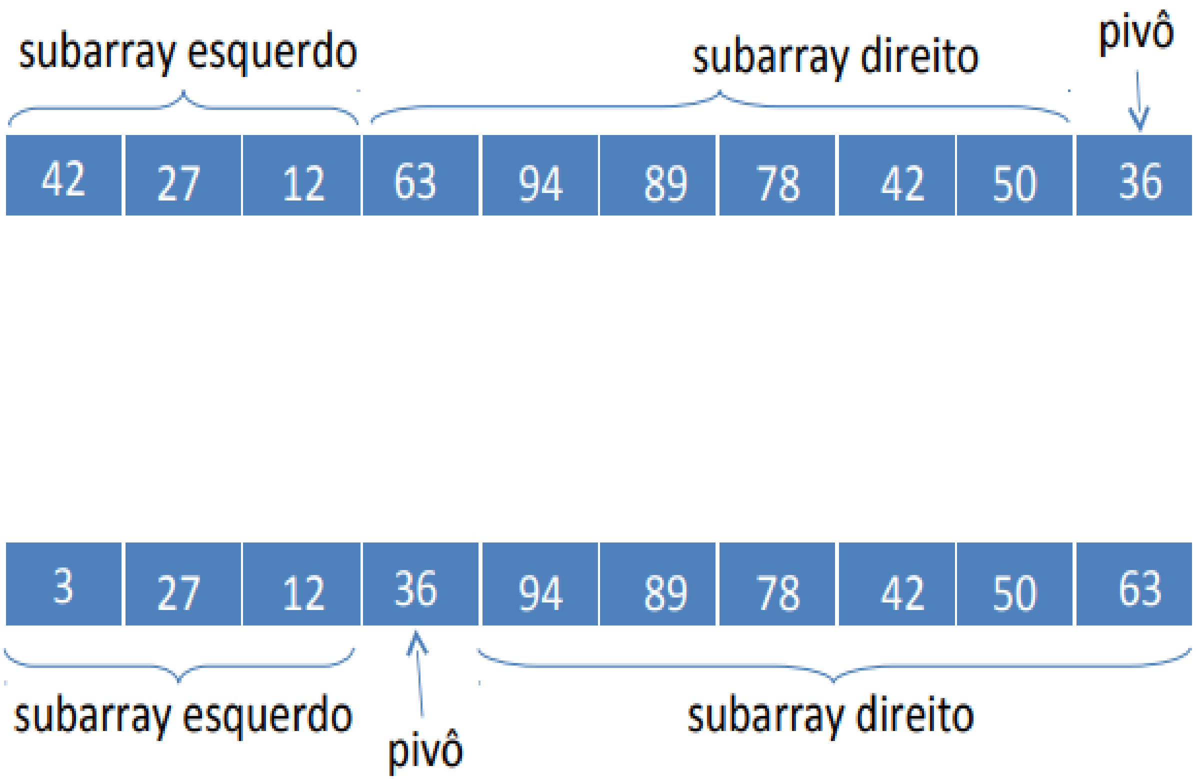
pivô = 15:

2	14	7	89	23	35

RP LP

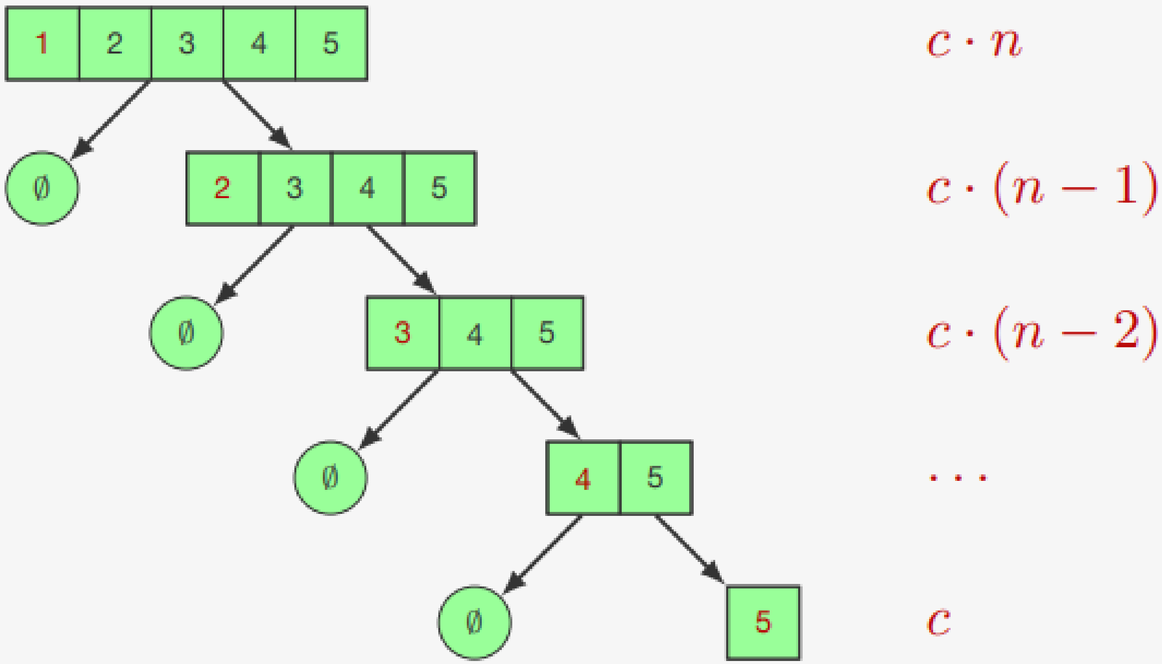
7 < 15 logo RP pára! A condição **LP** >= **RP** é satisfeita e o particionamento termina!

Pivô à direita



Pivô = 36
Após o particionamento troca-se o 36 pelo 63 que é o primeiro elemento do grupo dos valores maiores que o pivô

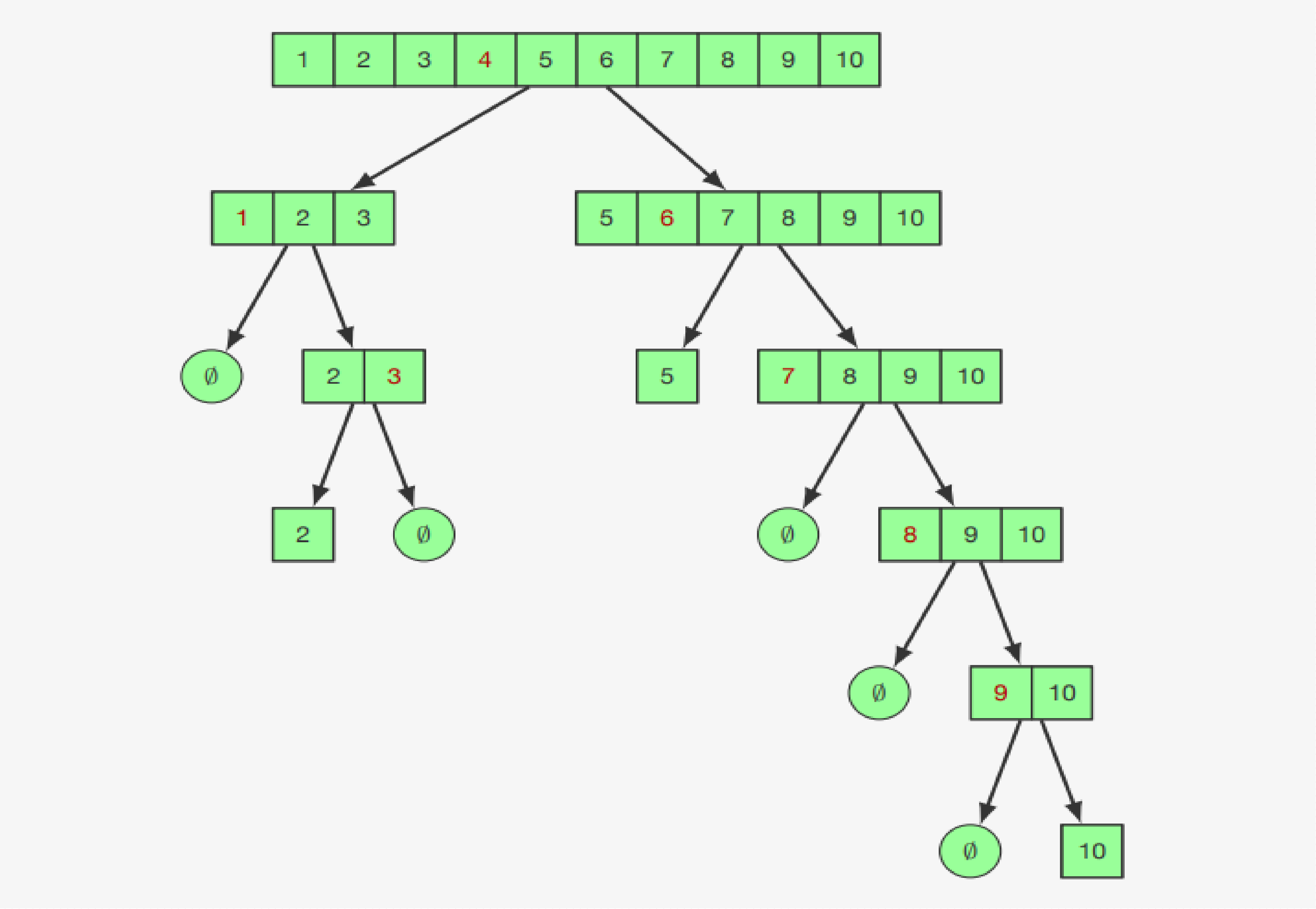
Pivô à esquerda



O tempo de execução do Quicksort é, no pior caso:

$$c \cdot n + c \cdot (n-1) + \cdots + c = c \sum_{i=0}^{n-1} (n-i) = c \sum_{j=1}^{n} j = c\frac{n(n+1)}{2} = \mathrm{O}(n^2)$$

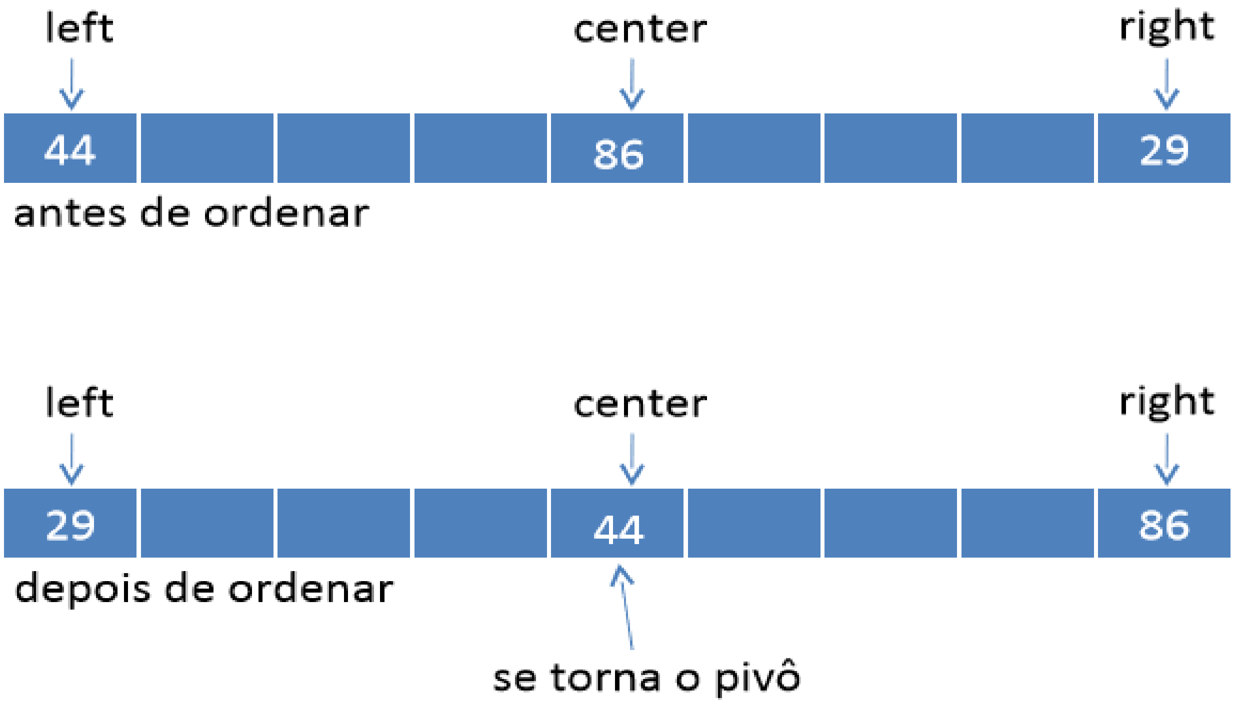
Pivô dentro do conjunto de dados



Pivô na média

- Uma solução simples e atraente é obter o valor mediano entre três elementos do array:
  - 1º elemento
  - Elemento no meio do array
  - Último elemento
- Processo chamado “média-dos-três”
  - Agilidade no processo e possui altas taxas de sucesso
  - Ganho de desempenho no algoritmo

Pivô na média



Detalhamento do algoritmo

- Algoritmo usa 2 funções
  - quickSort : divide os dados em arrays cada vez menores
  - particiona: calcula o pivô e rearranja os dados

```
void quickSort(int *V, int inicio, int fim) {
    int pivo;
    if(fim > inicio){
        pivo = particiona(V, inicio, fim);
        quickSort(V, inicio, pivo-1);
        quickSort(V, pivo+1, fim);
    }
}
```

Chama a função para as 2 metades

Separa os dados em 2 partições

Detalhamento do algoritmo

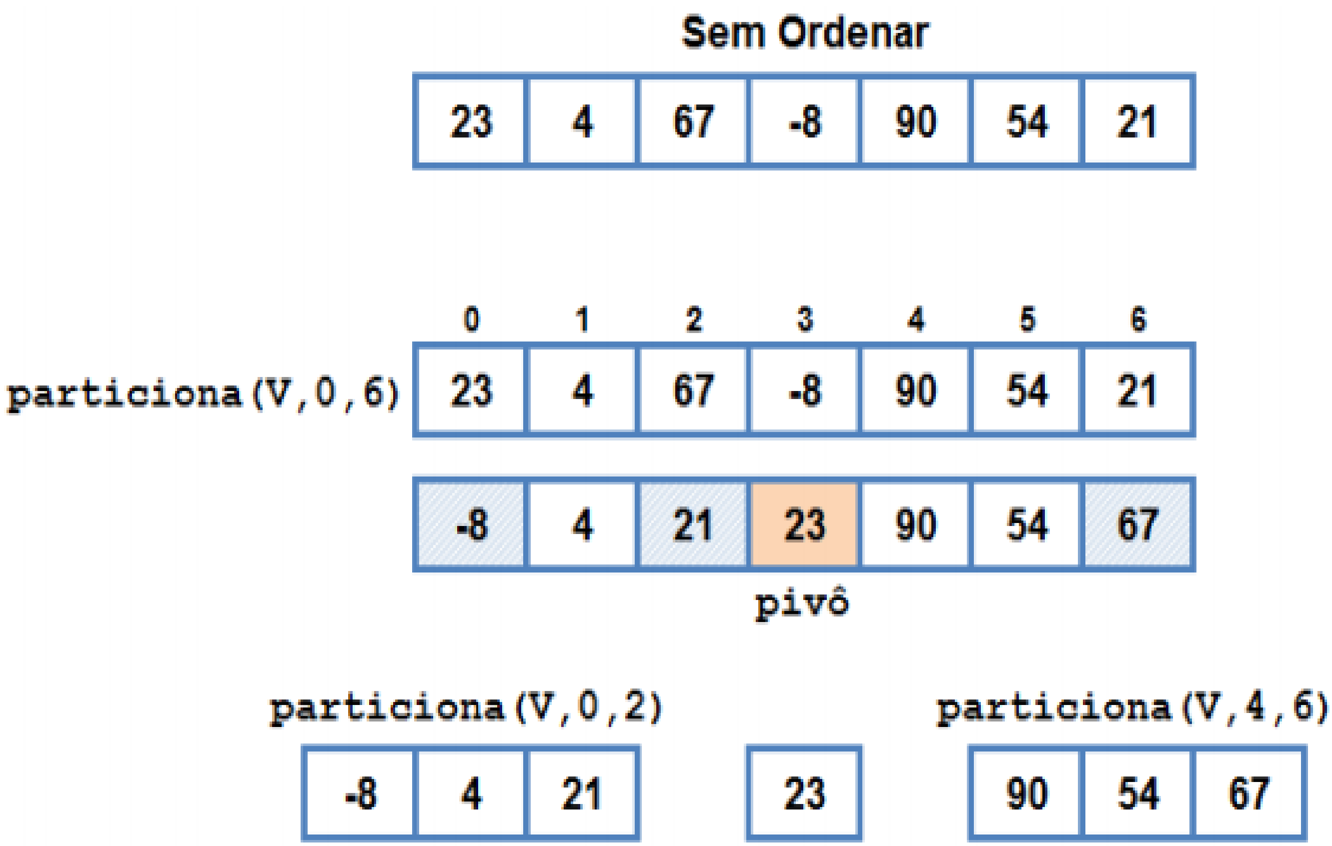
```c
int particiona(int *V, int inicio, int final ){
    int esq, dir, pivo, aux;
    esq = inicio;
    dir = final;
    pivo = V[inicio];
    while(esq < dir){
        while(esq <= final && V[esq] <= pivo)
            esq++;

        while(dir >= 0 && V[dir] > pivo)
            dir--;

        if(esq < dir){
            aux = V[esq];
            V[esq] = V[dir];
            V[dir] = aux;
        }
    }
    V[inicio] = V[dir];
    V[dir] = pivo;
    return dir;
}
```

Avança posição da esquerda

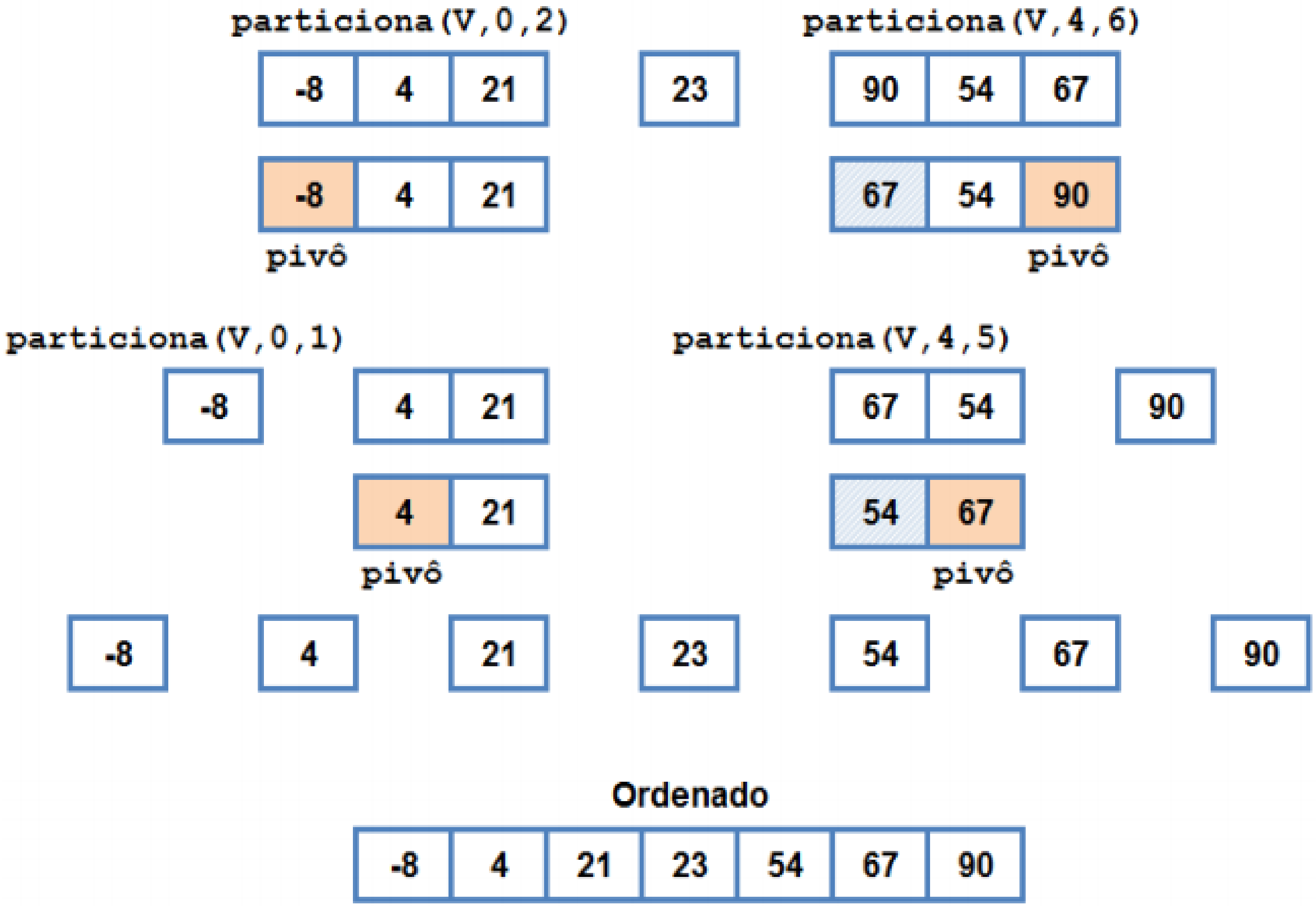
Recua posição da direita

Trocar esq e dir

Pivô central



Pivô central



Particionamento com pivô à direita

```
int particao (int[] A, int ini, int fim) {
 int i, j, temp;
 int x = A[fim]; // pivô
 i = ini;
 j = fim - 1;
 while (i <= j) {
   if(A[i] <= x) {
     i++;
   } else if (A[j] > x) {
     j--;
   } else { // trocar A[i] e A[j]
     temp = A[i];
     A[i] = A[j];
     A[j] = temp;
     i++; j--;
   }
 }
 A[fim] = A[i];  // reposicionar o pivô
 A[i] = x;
 return i;
}
```

Particionamento com pivô aleatório

```
int particaoAleatoria (int[] A, int ini, int fim) {
  int i, temp;
  double f;
  // Escolhe um número aleatório entre ini e fim
  f = java.lang.Math.random();
  // retorna um real f tal que 0 <= f < 1
  i = (int) (ini + (fim - ini) * f);
  // i é tal que ini <= i < fim
  // Troca de posicao A[i] e A[fim]
  temp = A[fim];
  A[fim] = A[i];
  A[i] = temp;
  return particao(A, ini, fim);
}
```

Implementação da ordenação

```
void quickSortAleatorio(int[] A, int ini, int fim) {
  if (ini < fim) {
    int q = particaoAleatoria(A, ini, fim);
    quickSortAleatorio(A, ini, q - 1);
    quickSortAleatorio(A, q + 1, fim);
  }
}
```

Vantagens

- Apesar de seu pior caso ser quadrático, costuma ser a melhor opção prática para ordenação de **grandes conjuntos de dados**
- Flexibilidade para escolha do elemento que será utilizado como parâmetro de comparação
- Possuí várias formas de implementação, que podem ser utilizadas em sistemas de larga escala

Desvantagens

- **Não é um algoritmo estável**
- **Como escolher o pivô?**
- **Existem várias abordagens diferentes**
- No pior caso o pivô divide o array de N em dois: uma partição com N-1 elementos e outra com 0 elementos
- **Particionamento não é balanceado**
- Quando isso acontece a cada nível da recursão, temos o tempo de execução de O(N2)

Análise do algoritmo

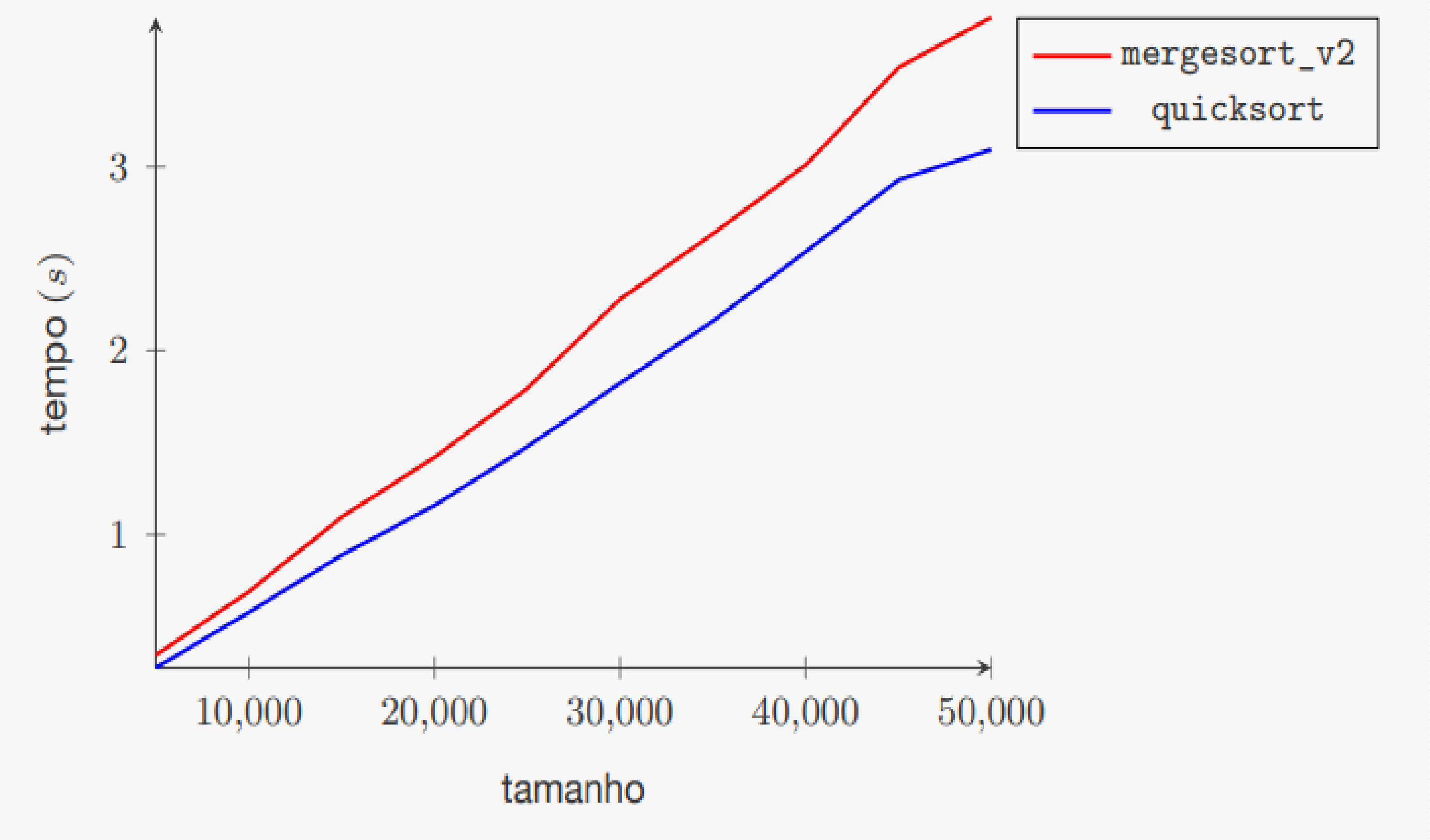
- Quick Sort pode ser mais rápido que o Merge Sort na prática
- Leva tempo O(n lg n) (em média) para o processo de ordenação
- Sua versão aleatorizada é O(n lg n) em média

Desempenho dos algoritmos de Ordenação

	QuickSort	*HeapSort*	*MergeSort*
Pior caso	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Caso médio	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Melhor caso	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

- Classificando por melhor desempenho médio:
  1º) *MergeSort* (algoritmo mais simples)
  2º) *QuickSort*
  3º) *HeapSort*

Comparação com o Merge Sort



Desempenho dos algoritmos de Ordenação

VETOR [10.000]									
Lista	Ordem Crescente			Ordem Decrescente			Ordem Aleatória		
Algoritmo	Tempo (s)	Comp.	Trocas	Tempo (s)	Comp.	Trocas	Tempo (s)	Comp.	Trocas
Bubble Sort	0,4269048	49995000	0	0,9847921	49995000	49995000	0,7649256	49995000	25084128,1
Insertion Sort	0,0003026	9999	0	0,4580984	9999	49995000	0,225615	9999	24963151
Selection Sort	0,3637704	49995000	0	0,3827789	49995000	5000	0,360824	49995000	9988
Merge Sort	0,0058387	135423	250848	0,0056613	74911	254944	0,006185	132011,1	252879
Quick Sort	0,4415975	49995000	0	1,192945	49995000	49995000	0,1867259	158055	25098217,7
Shell Sort	0,001431	75243	0	0,0019362	75243	161374	0,0034228	75243	161374

Contatos

- Email: fabio.silva321@fatec.sp.gov.br
- Linkedin: https://br.linkedin.com/in/b41a5269
- Facebook: https://www.facebook.com/fabio.silva.56211