Engenharia de Software

Estrutura de Dados II

Aula 4: Ordenação - QuickSort

Professor: M.e. Henrique Valle de Lima
henrivalle@gmail.com

Introdução

- A noção de um conjunto de dados ordenados é de considerável importância na nossa vida cotidiana e por conseguinte também em computação.

- Exemplos:
  - Listas telefônicas
  - Listas de clientes de uma empresa
  - Listas de peças em um catálogo

- Algoritmos eficientes para ordenar grandes quantidades de dados são de extrema importância.

Conceitos básicos

- Arquivo:
  - um arquivo r de tamanho n é uma seqüência de n itens r[0], r[1], …, r[n-1];
  - cada item em um arquivo é chamado registro.
- Um arquivo é classificado por chave:
  - se para cada registro r[i] existe uma parte de r[i], k[i] chamada chave de classificação;
  - se $i<j$ implicar que k[i] precede k[j] de acordo com algum critério qualquer predefinido.

Conceitos básicos

- Exemplo:
  - Em um catálogo telefônico o arquivo é o próprio catálogo;
  - Um registro é uma entrada com nome, número e endereço;
  - O nome é a chave.

Localização de execução

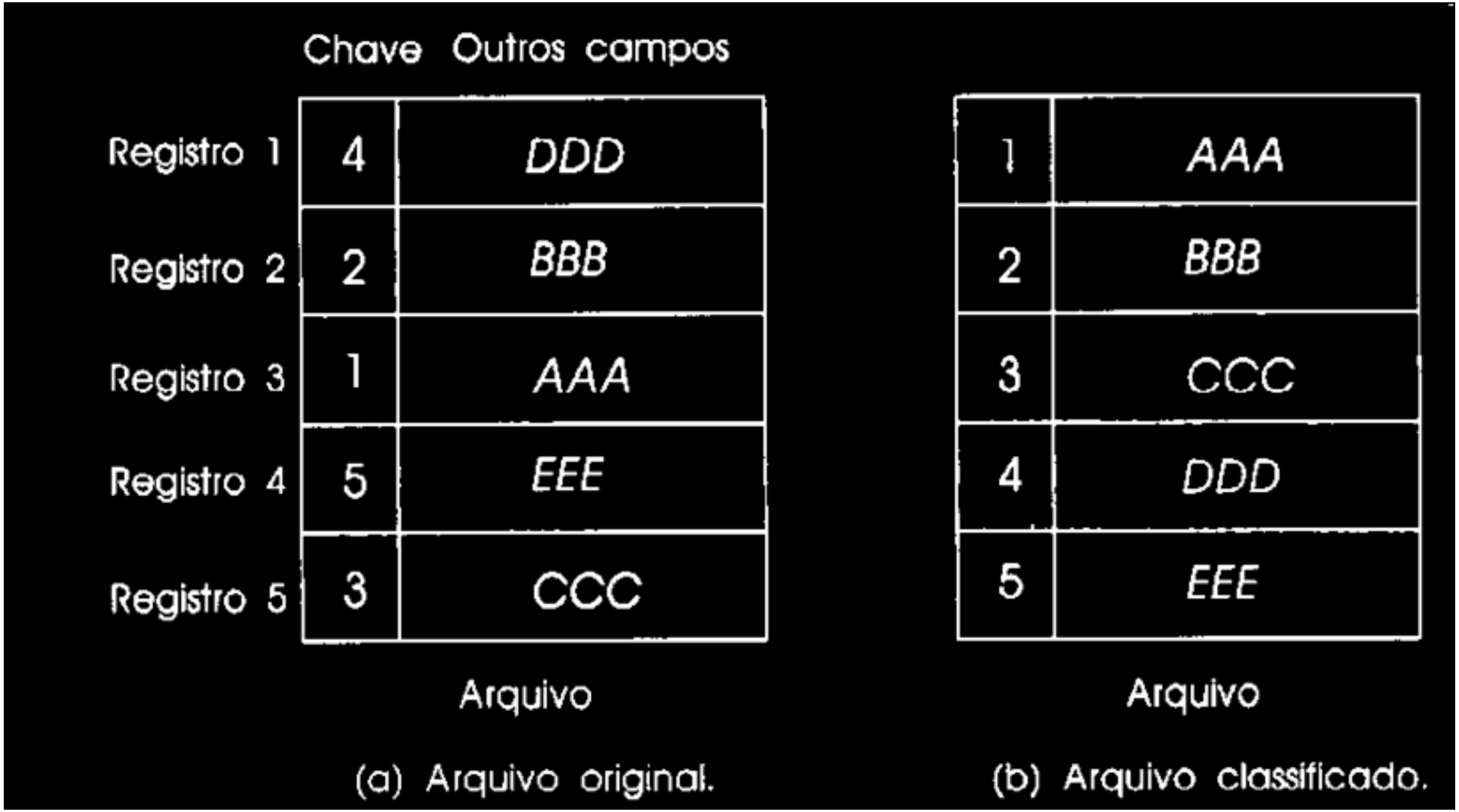
- Ordenação Interna é aquela realizada na memória principal do computador;

- Ordenação Externa é aquela onde os registros podem estar em uma memória auxiliar (arquivo em disco).

Classificações

- **Uma ordenação pode ocorrer sobre os próprios registros ou sobre uma tabela auxiliar de ponteiros:**
  - classificação direta é uma ordenação onde um arquivo é ordenado diretamente, sendo os registros movidos de uma posição física para outra;
  - classificação por endereços é uma ordenação onde somente uma tabela de ponteiros é ordenada; os registros em si não são tocados.

Classificação direta



	Chave	Outros campos
Registro 1	4	DDD
Registro 2	2	BBB
Registro 3	1	AAA
Registro 4	5	EEE
Registro 5	3	CCC

Arquivo

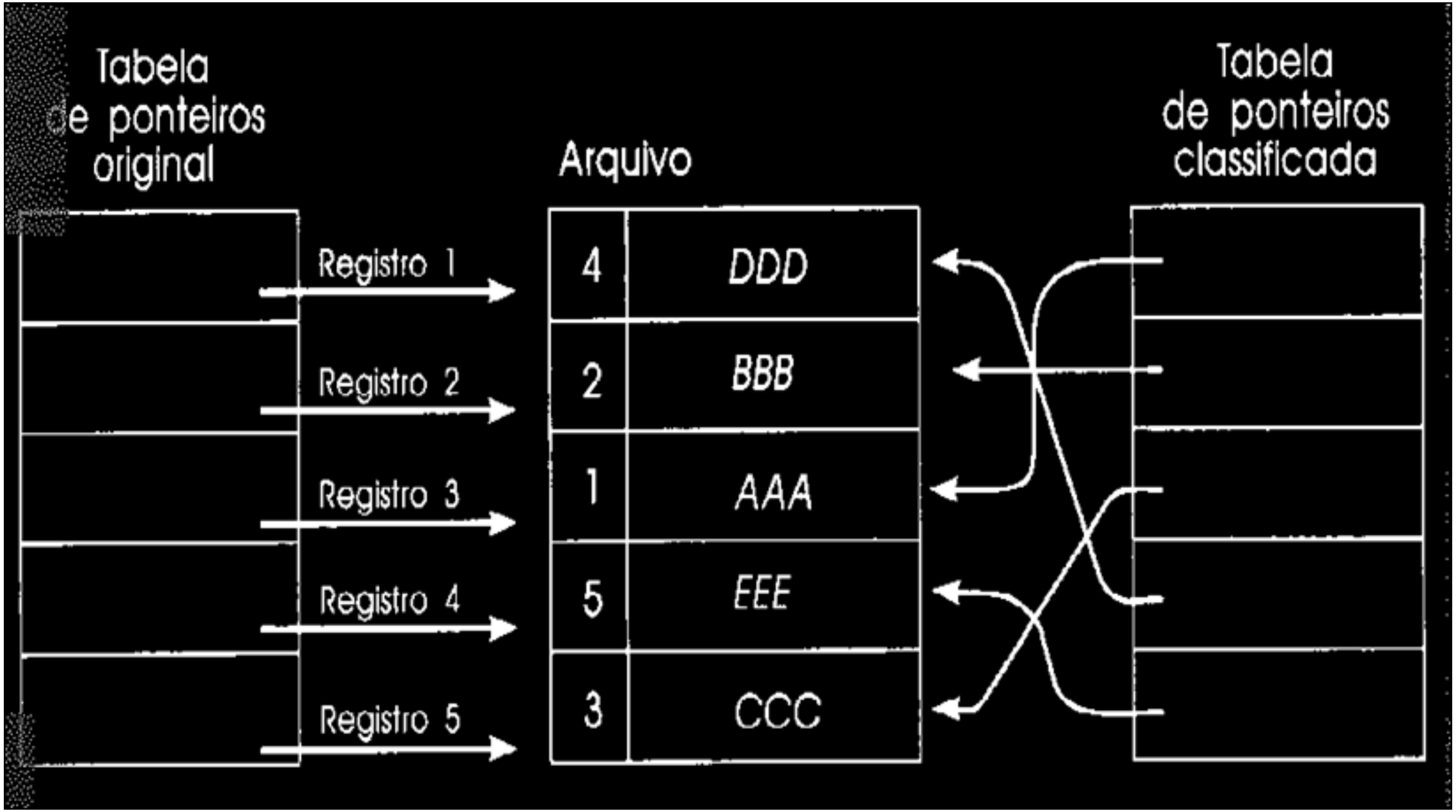
(a) Arquivo original.

1	AAA
2	BBB
3	CCC
4	DDD
5	EEE

Arquivo

(b) Arquivo classificado.

Classificação por endereços



Critérios para seleção de ordenação

- Eficiência no tempo:
  - Se um arquivo é pequeno (ex.: 20 registros) técnicas sofisticadas podem ser piores:
    - algoritmos complexos, módulos grandes;
  - Número de vezes que a ordenação será executada:
    - Se vamos classificar um conjunto de dados só uma vez, pode ser mais interessante implementar um algoritmo simples;
  - Um programa que executa ordenações repetidamente deve ser eficiente;
  - Esforço de programação não deve ser uma desculpa para a utilização de um algoritmo inadequado. Um sistema ineficiente não vai vender.

Critérios para seleção de ordenação

- **Aspectos da eficiência de tempo:**
  - O critério básico para a determinação da eficiência de tempo de um algoritmo não é somente sua complexidade assintótica;
  - Operações críticas:
    - Operação crítica é uma operação básica no processo de ordenação.
    - Ex.: a comparação de duas chaves ou o deslocamento de um registro de uma posição de memória para outra;
    - Um algoritmo que executa todas as operações críticas na memória será geralmente mais rápido do que outro que executa algumas em memória secundária (disco).

Critérios para seleção de ordenação

- **Eficiência de espaço:**
    - A técnica que escolhi é compatível com as características de hardware do ambiente para o qual a estou destinando?

Quicksort

- A ordenação por troca de partição ou quicksort é provavelmente o algoritmo de ordenação mais utilizado;
- *Divide and Conquer*;
- Complexidade média O(n log n).

Quicksort

- Vantagens:
  - simples de implementar;
  - muito bem compreendido;
  - extensas análises matemáticas de seu comportamento já foram feitas.

- Desvantagens:
  - complexidade pode ser $O(n^2)$ no pior caso, em que o arquivo está ordenado ou ordenado em ordem inversa.

- Existem versões melhoradas.

Quicksort - Algoritmo básico

- Quicksort trabalha particionando um arquivo em duas partes e então as ordenando separadamente;

- pode ser definido recursivamente.

```
quicksort(tInfo: a[], inteiro: limInf, limSup)
inteiro i;
início
      SE(limSup > limInf) ENTÃO
            i <- particione(a, limInf, limSup);
            quicksort(a, limInf, i-1);
            quicksort(a, i+1, limSup);
      FIM SE
fim
```

Algoritmo básico

- Os parâmetros dir e esq delimitam os subarquivos dentro do arquivo original, dentro dos quais a ordenação ocorre;
- A chamada inicial pode ser feita com quicksort(a, 1, N);
- O ponto crucial é o algoritmo de partição.

Particionamento em Quicksort

- O procedimento de particionamento deve rearranjar o arquivo de maneira que as seguintes condições valham:
  - O elemento a[i] está em seu lugar final no arquivo para um i dado;
  - Nenhum dos elementos em a[esq],...,a[i-1] são maiores do que a[i];
  - Nenhum dos elementos em a[i+1],...,a[dir] são menores do que a[i].

Exemplo

Ordenação do vetor inicial 25 57 48 37 12 92 86 33

- Se o primeiro elemento (25) for colocado na sua posição correta, teremos:
  12 25 57 48 37 92 86 33

  Neste ponto:

  - Todos os elementos abaixo de 25 serão menores e
  - Todos os elementos acima de 25 serão maiores.

Exemplo

- Como 25 está na sua posição final, o problema foi decomposto na ordenação dos subvetores: (12) e (57 48 37 92 86 33);
- O subvetor (12) já está classificado;
- Agora o vetor pode ser visualizado: 12 25 (57 48 37 92 86 33);
- Repetir o processo para a[2]...a[7] resulta em: 12 25 (48 37 33) 57 (92 86)

Exemplo

- Se continuarmos particionando 12 25 (48 37 33) 57 (92 86), teremos:

12 25 (37 33) 48 57 (92 86)

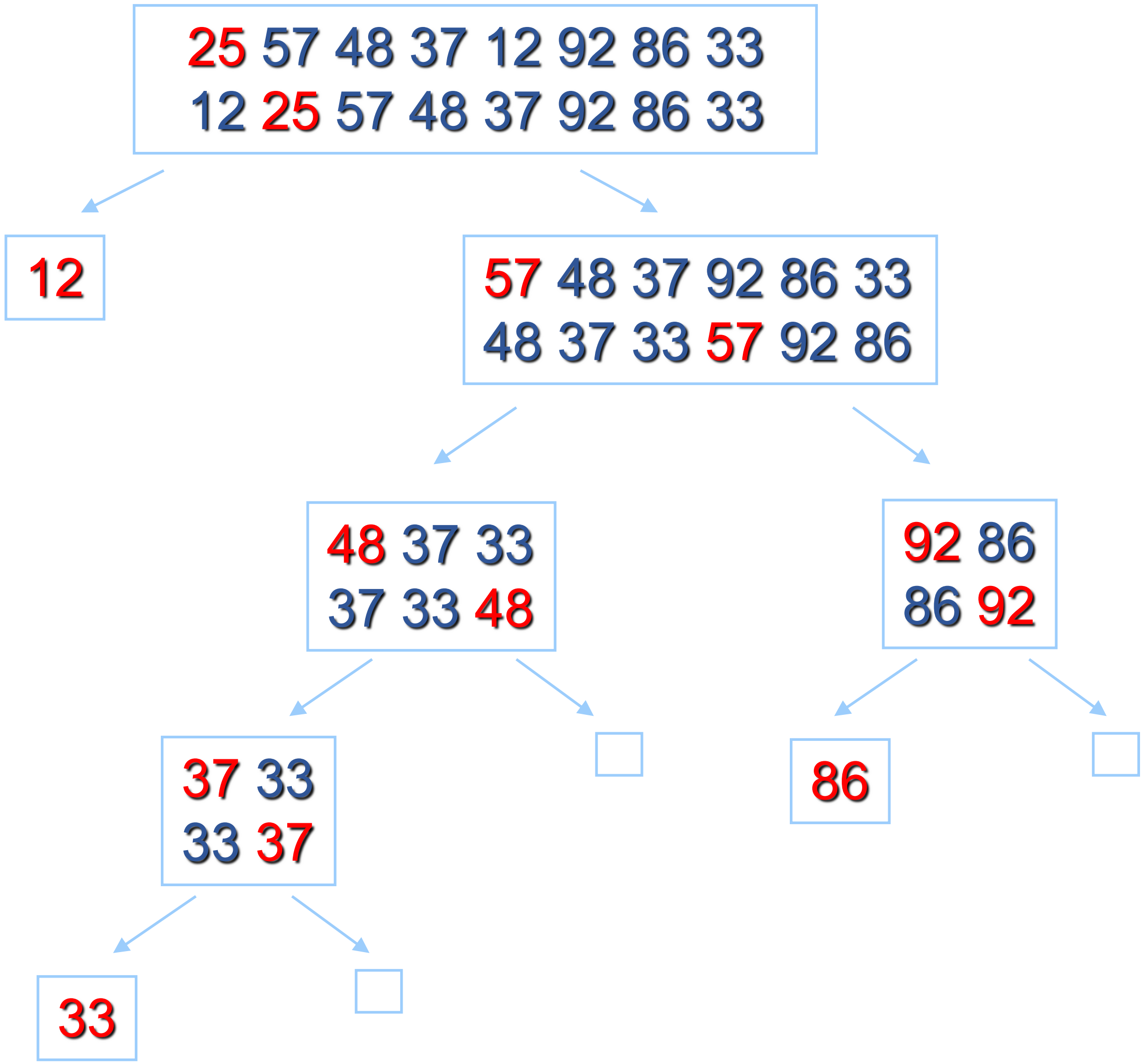
12 25 (33) 37 48 57 (92 86)

12 25 33 37 48 57 (92 86)

12 25 33 37 48 57 (86) 92

12 25 33 37 48 57 86 92

Visão da Recursividade do Quicksort como Árvore



Método para o particionamento

- Considere pivô = a[limInf] como o elemento cuja posição final é a procurada:
  - usar sempre o primeiro é só um artifício para facilitar a implementação.

- Dois ponteiros alto e baixo são inicializados como os limites máximo e mínimo do subvetor que vamos analisar;

- em qualquer ponto da execução, todo elemento acima de alto é maior do que x e todo elemento abaixo de baixo é menor do que x.

Método para o particionamento

- Os dois ponteiros alto e baixo são movidos um em direção ao outro da seguinte forma:
  - 1. incremente baixo em uma posição até que a[baixo] >= pivô;
  - 2. decremente alto em uma posição até que a[alto] < pivô;
  - 3. se alto > baixo, troque a[baixo] por a[alto].
- O processo é repetido até que a condição descrita em 3 falhe (quando alto <= baixo). Neste ponto a[alto] será trocado por a[limInf], cuja posição final era procurada, e alto é retornado em i.

Simulação de Quicksort

limInf limSup

a											
23	66	53	75	56	21	11	33	6	11	79	43

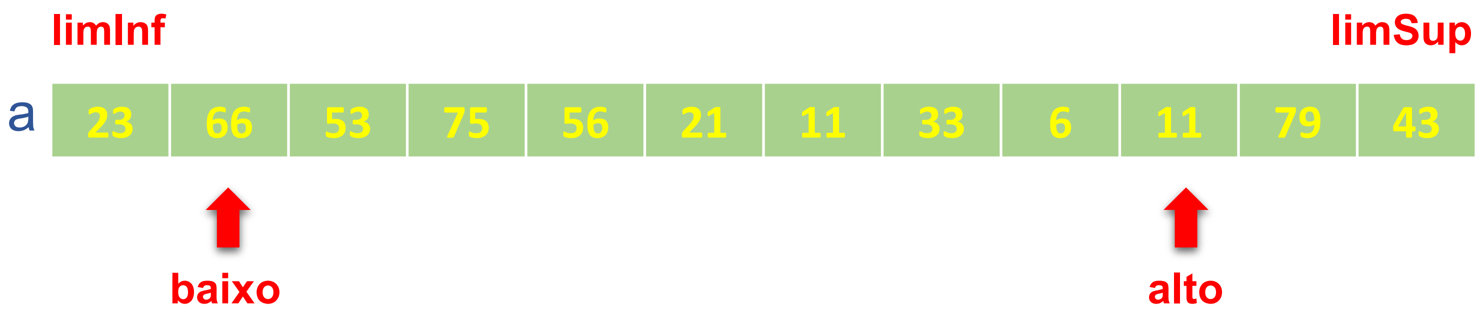
baixo alto

- Ordenamos um vetor de 12 posições de limInf até limSup. Escolhemos para pivô a[limInf]. Para ordená-lo, vamos dividí-lo em duas partes: a parte da esquerda possuirá os elementos menores que o pivô, a parte direita os maiores.

Simulação de Quicksort

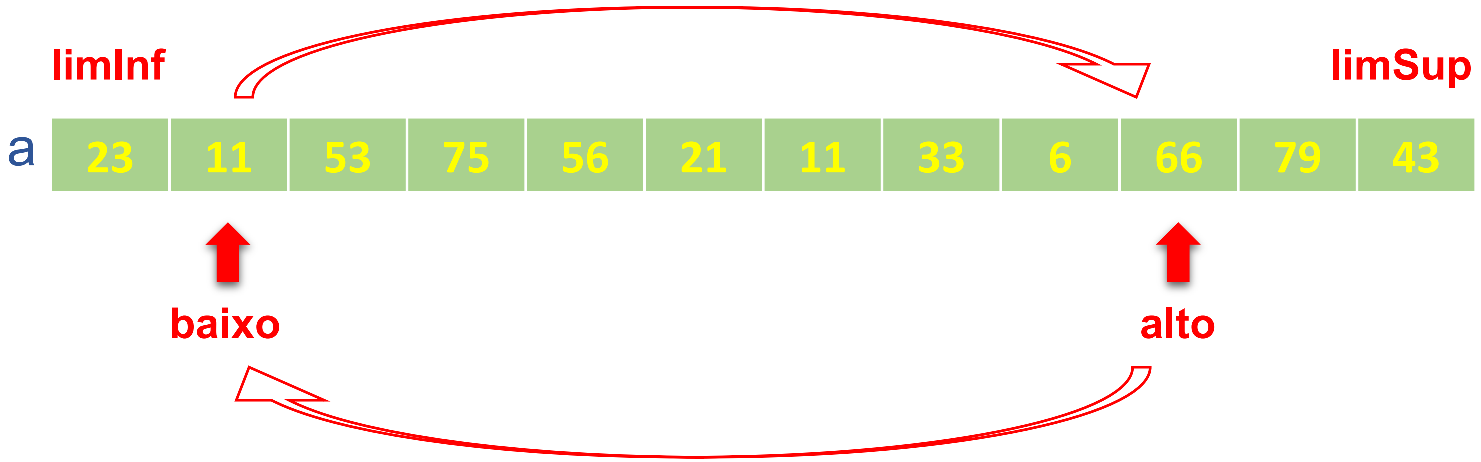
- Incrementamos baixo até encontrar um elemento maior que o pivô (23). Nesse caso achamos 66.

Simulação de Quicksort



- Decrementamos alto até encontrar um elemento menor que o pivô (23). Nesse caso achamos 11.

Simulação de Quicksort



- Trocamos a[alto] e a[baixo] de posição.

Simulação de Quicksort

limInf limSup

a											
23	11	53	75	56	21	11	33	6	66	79	43

baixo alto

- Troca realizada.

Simulação de Quicksort

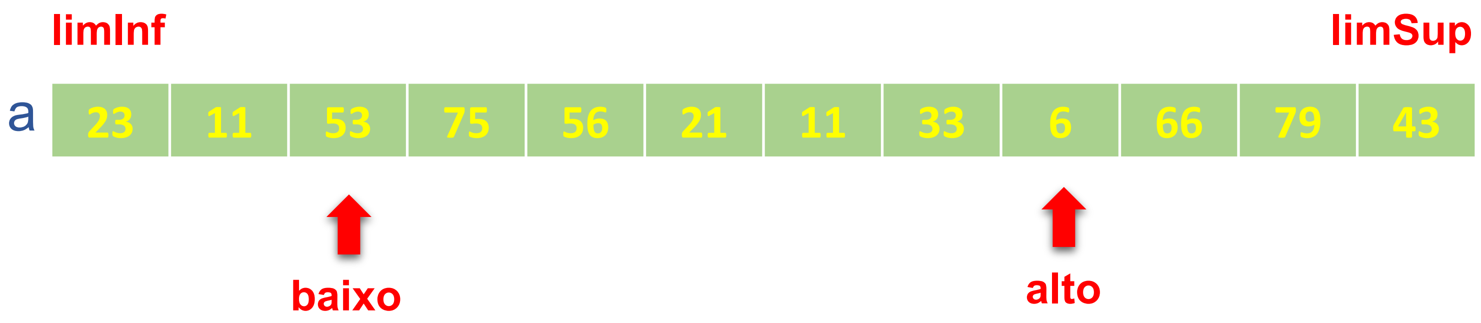
limInf limSup

a												
	23	11	53	75	56	21	11	33	6	66	79	43
			↑ baixo							↑ alto		

- Continuamos a incrementar baixo até encontrar um elemento maior que o pivô (23). Nesse caso achamos 53.

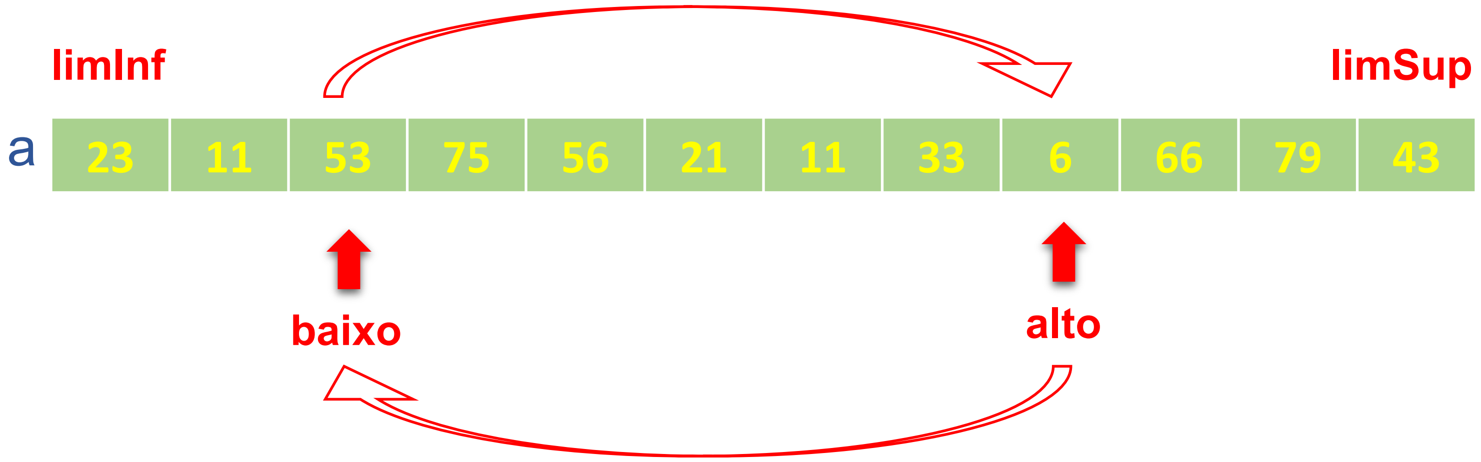
UniEVANGÉLICA
UNIVERSIDADE EVANGÉLICA DE GOIÁS

Simulação de Quicksort



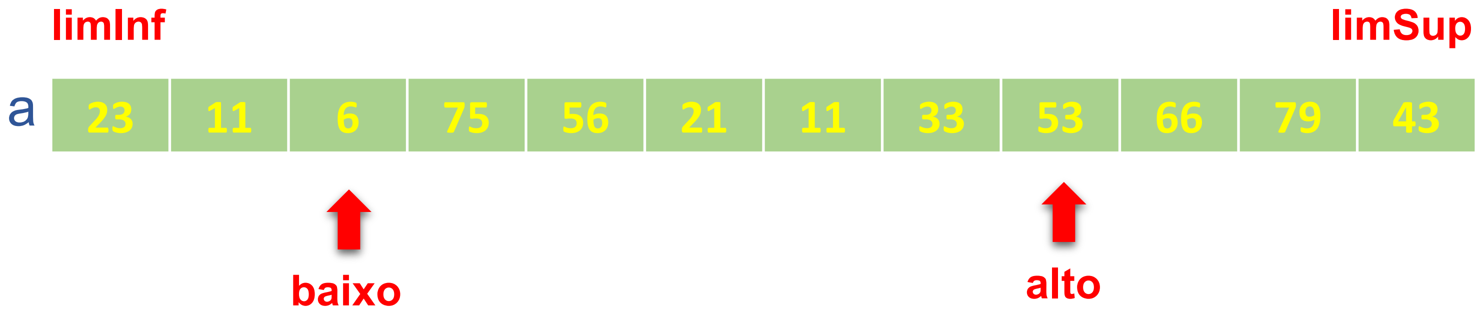
- Continuamos a decrementar alto até encontrar um elemento menor que o pivô (23). Nesse caso achamos 6.

Simulação de Quicksort



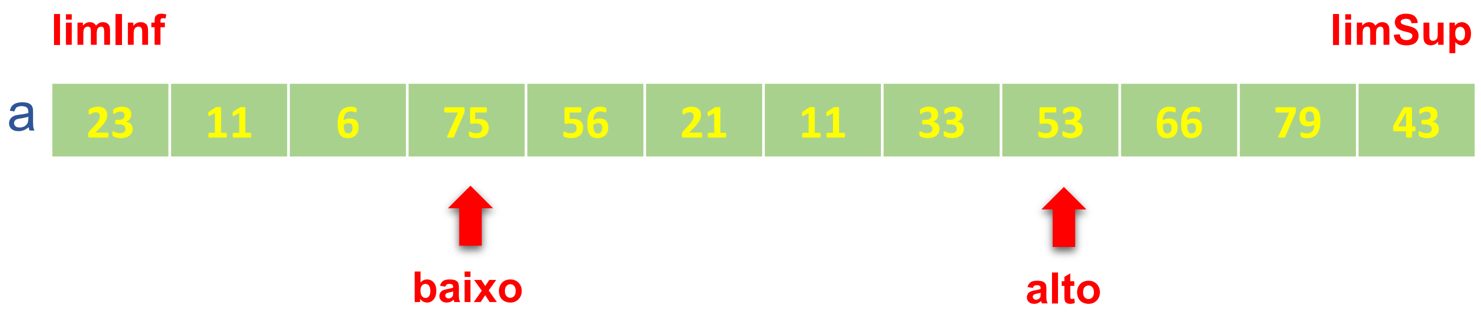
- Trocamos a[alto] e a[baixo].

Simulação de Quicksort



- Troca realizada

Simulação de Quicksort



- Continuamos a incrementar baixo até encontrar um elemento maior ou igual ao pivô (23). Nesse caso achamos 75.

Simulação de Quicksort

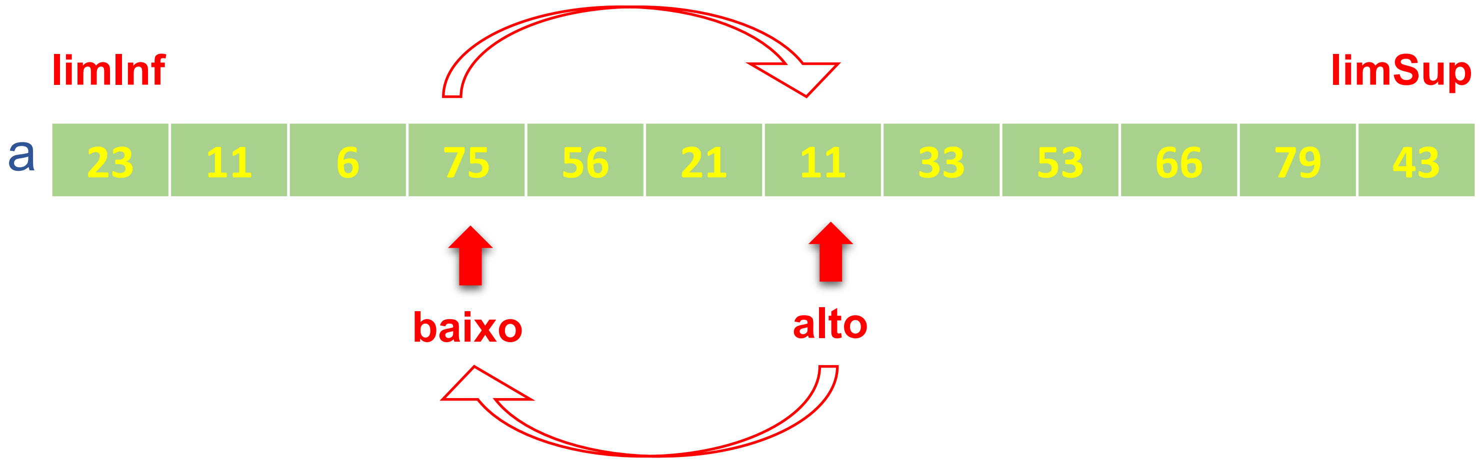
limInf limSup

a	23	11	6	75	56	21	11	33	53	66	79	43

baixo alto

- Continuamos a decrementar alto até encontrar um elemento menor que o pivô (23). Nesse caso achamos 11.

Simulação de Quicksort



- Trocamos a[alto] e a[baixo].

UniEVANGÉLICA
UNIVERSIDADE EVANGÉLICA DE GOIÁS

Simulação de Quicksort

limInf limSup

a											
23	11	6	11	56	21	75	33	53	66	79	43

baixo alto

- Troca realizada

Simulação de Quicksort

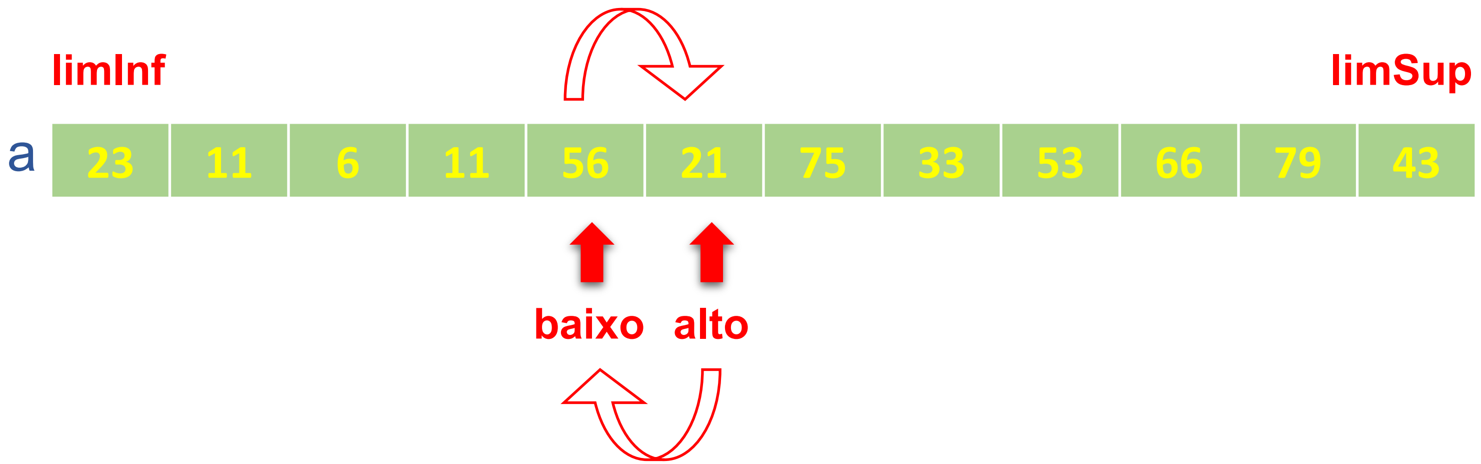
	limInf											limSup
a	23	11	6	11	56	21	75	33	53	66	79	43
					baixo		alto					

- Continuamos a incrementar baixo até encontrar um elemento maior ou igual ao pivô (23). Nesse caso achamos 56.

Simulação de Quicksort

- Continuamos a decrementar alto até encontrar um elemento menor que o pivô (23). Nesse caso achamos 21.

Simulação de Quicksort



- Trocamos a[alto] e a[baixo].

Simulação de Quicksort

limInf limSup

a	23	11	6	11	21	56	75	33	53	66	79	43

baixo alto

- Troca realizada

Simulação de Quicksort

limInf limSup

a												
	23	11	6	11	21	56	75	33	53	66	79	43

alto

baixo

- Continuamos a incrementar baixo até encontrar um elemento maior ou igual ao pivô (23). Nesse caso achamos novamente o 56.

UniEVANGÉLICA
UNIVERSIDADE EVANGÉLICA DE GOIÁS

Simulação de Quicksort

limInf limSup

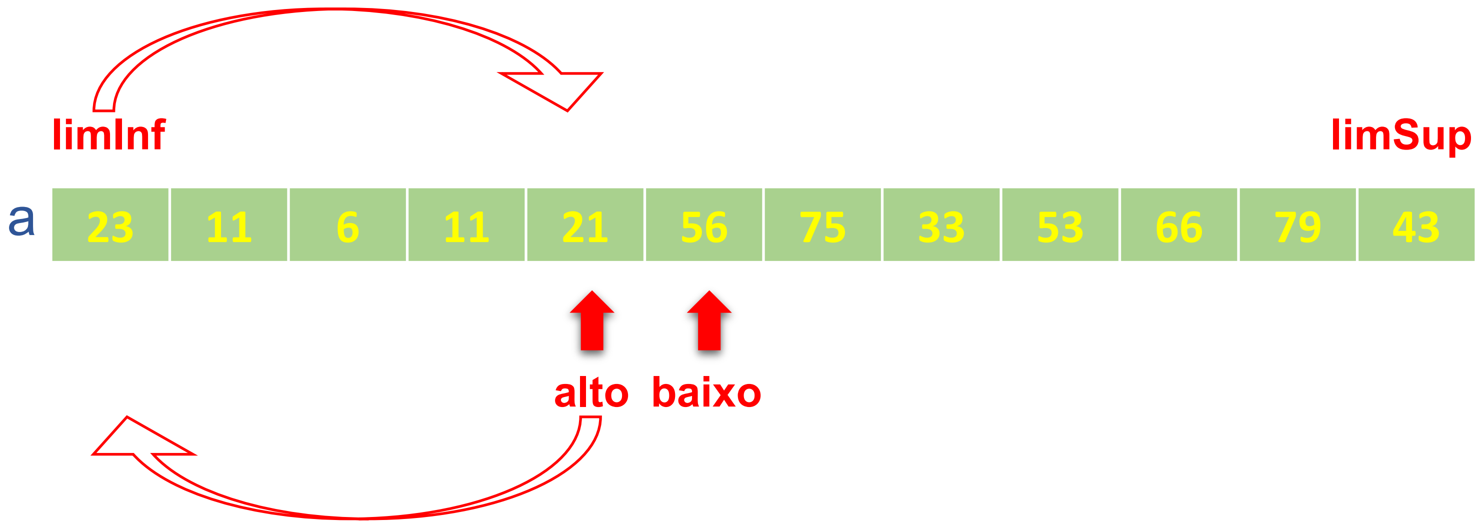
a

23	11	6	11	21	56	75	33	53	66	79	43

alto baixo

- Continuamos a decrementar alto até encontrar um elemento menor que o pivô (23). Nesse caso achamos novamente o 21, mas alto é menor que baixo, então atingimos a condição de parada!

Simulação de Quicksort



- Agora trocamos o pivô a[limInf] com a[alto] e a divisão do vetor está completa

Simulação de Quicksort

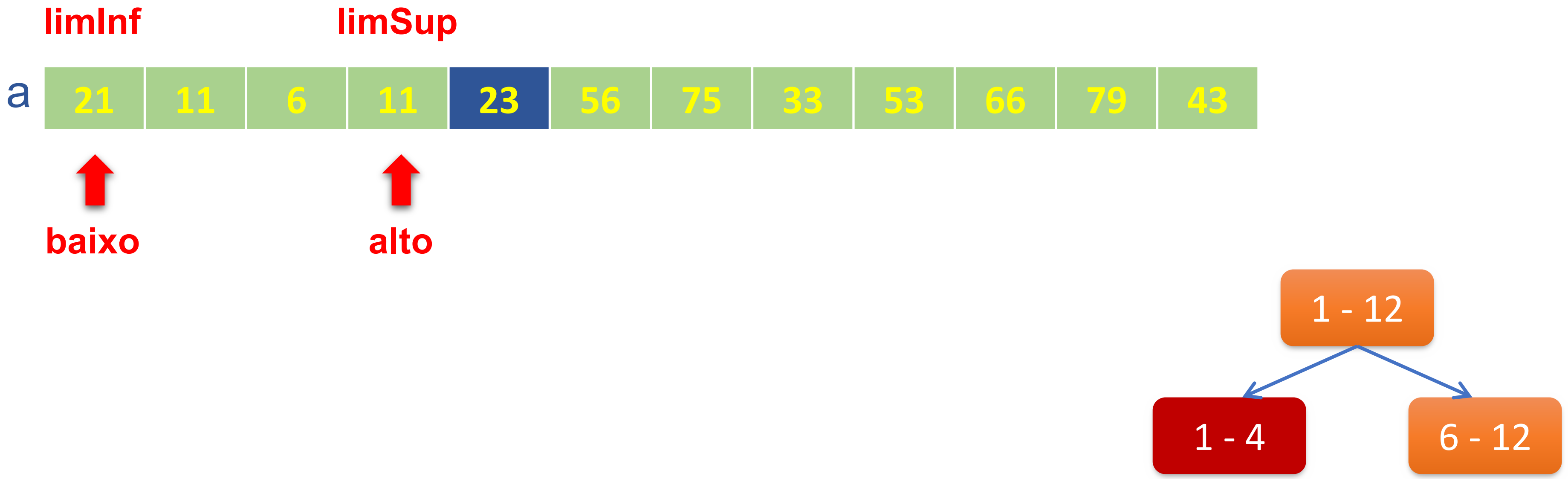
limInf limSup

a											
21	11	6	11	23	56	75	33	53	66	79	43

alto baixo

- Troca realizada

Simulação de Quicksort



- Agora chamamos a partição com limSup = alto – 1 = 4 e particionamos o subvetor esquerdo.

Simulação de Quicksort

limInf limSup

a											
21	11	6	11	23	56	75	33	53	66	79	43

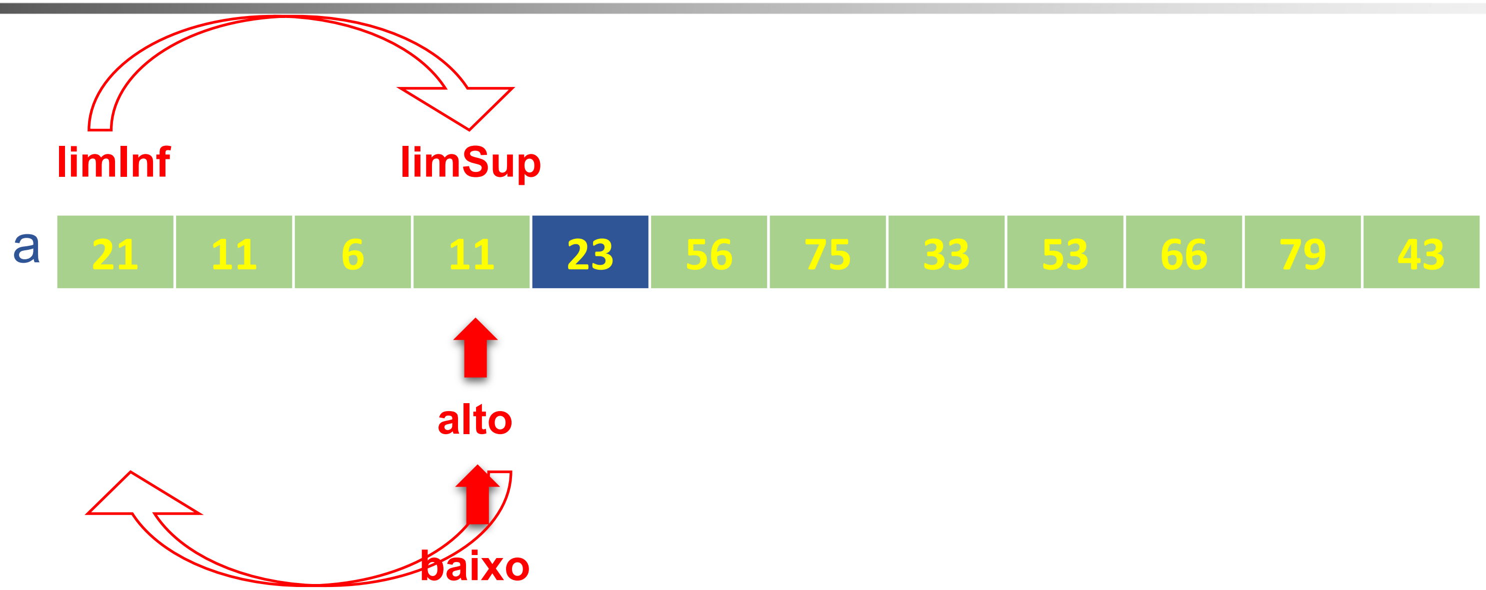
alto

baixo

- Incrementamos baixo até encontrar um elemento maior ou igual ao pivô (21) ou baixo deixar de ser menor que alto. Nesse caso encontramos alto.

- Decrementamos alto até encontrar um elemento menor que o pivô (21) ou alto deixar de ser maior que baixo. Nesse caso já nos encontramos em baixo e não fazemos nada.

Simulação de Quicksort



- Agora trocamos o pivô a[limInf] com a[alto] e a divisão do vetor está completa!

Simulação de Quicksort

limInf limSup

a												
	11	11	6	21	23	56	75	33	53	66	79	43

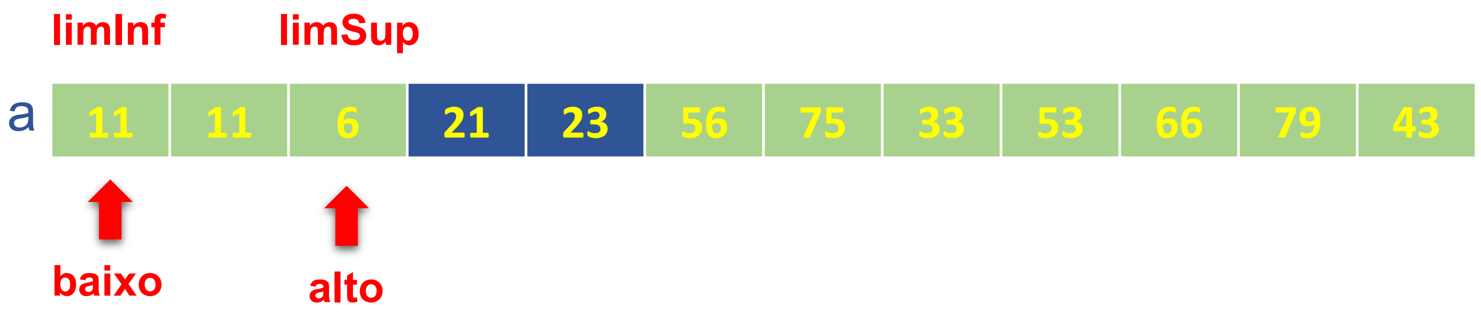
alto

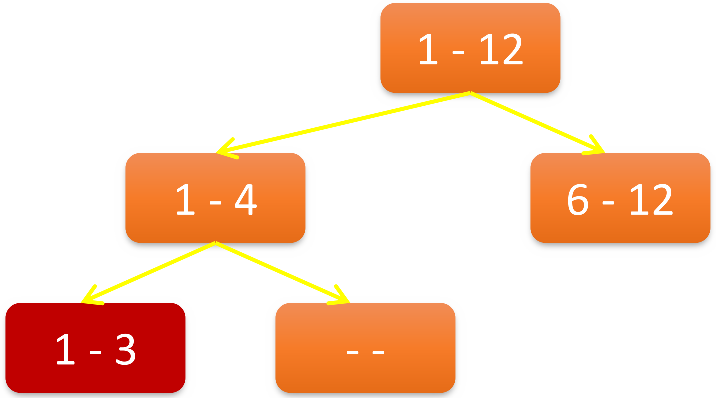
baixo

- Troca realizada

UniEVANGÉLICA
UNIVERSIDADE EVANGÉLICA DE GOIÁS

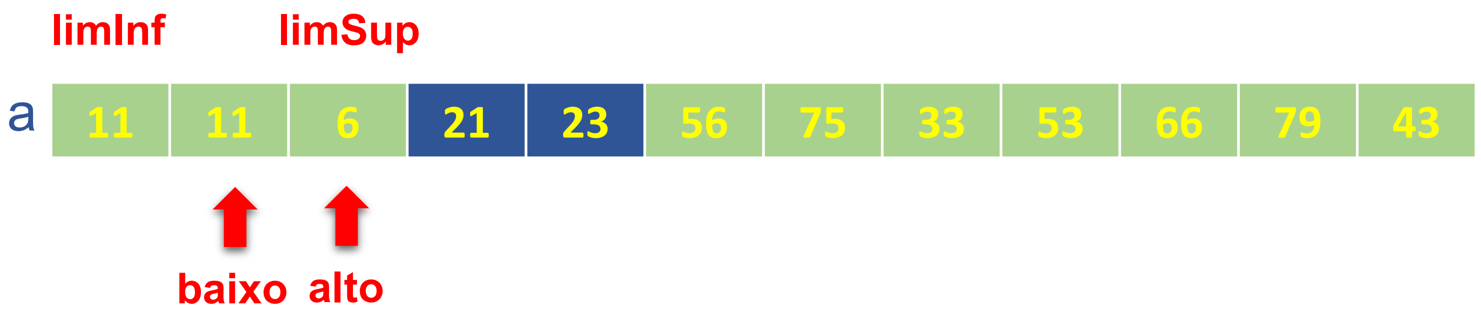
Simulação de Quicksort





- Agora chamamos a partição com limSup = alto - 1 = 3 e particionamos o subvetor esquerdo do subvetor esquerdo

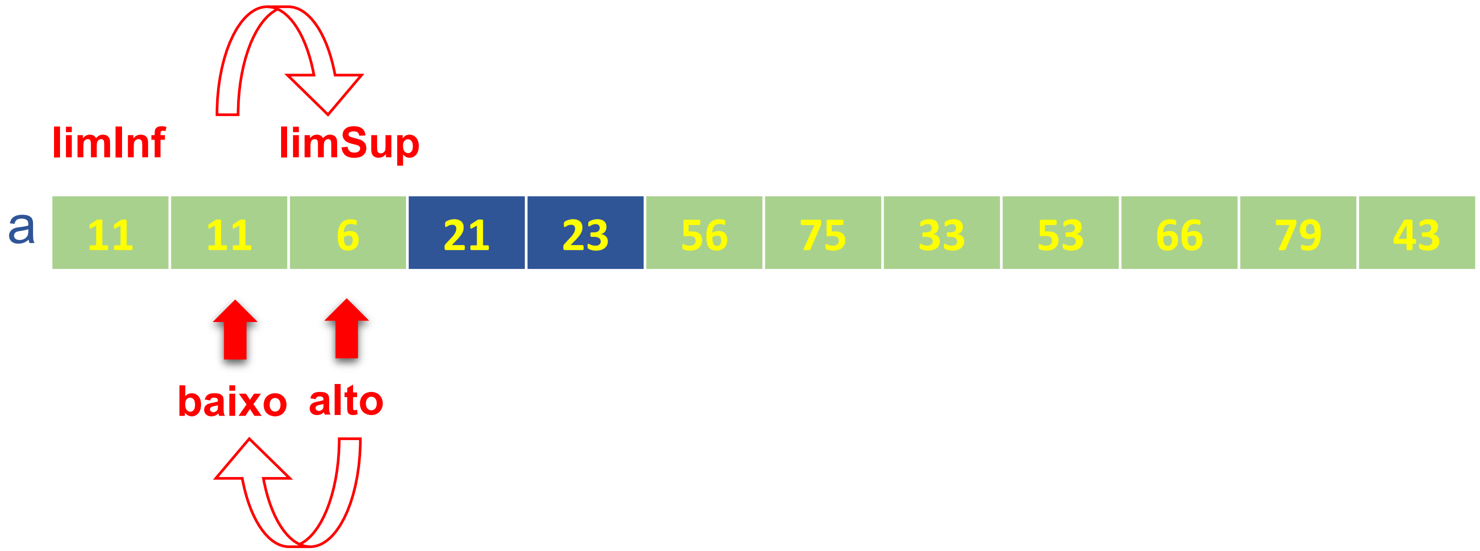
Simulação de Quicksort



- Incrementamos baixo até encontrar um elemento maior ou igual que o pivô (11) ou baixo deixar de ser menor que alto. Nesse caso encontramos 11.

- Decrementamos alto até encontrar um elemento menor que o pivô (11) ou alto deixar de ser maior que baixo. Nesse caso já nos encontramos em 6 e não fazemos nada.

Simulação de Quicksort



- Trocamos a[alto] e a[baixo]

Simulação de Quicksort

limInf limSup

a

11	6	11	21	23	56	75	33	53	66	79	43

baixo alto

- Troca realizada

Simulação de Quicksort

limInf limSup

a											
11	6	11	21	23	56	75	33	53	66	79	43

alto
baixo

- Continuamos a incrementar baixo até encontrar um elemento maior ou igual ao pivô (11). Nesse caso achamos 11.

Simulação de Quicksort

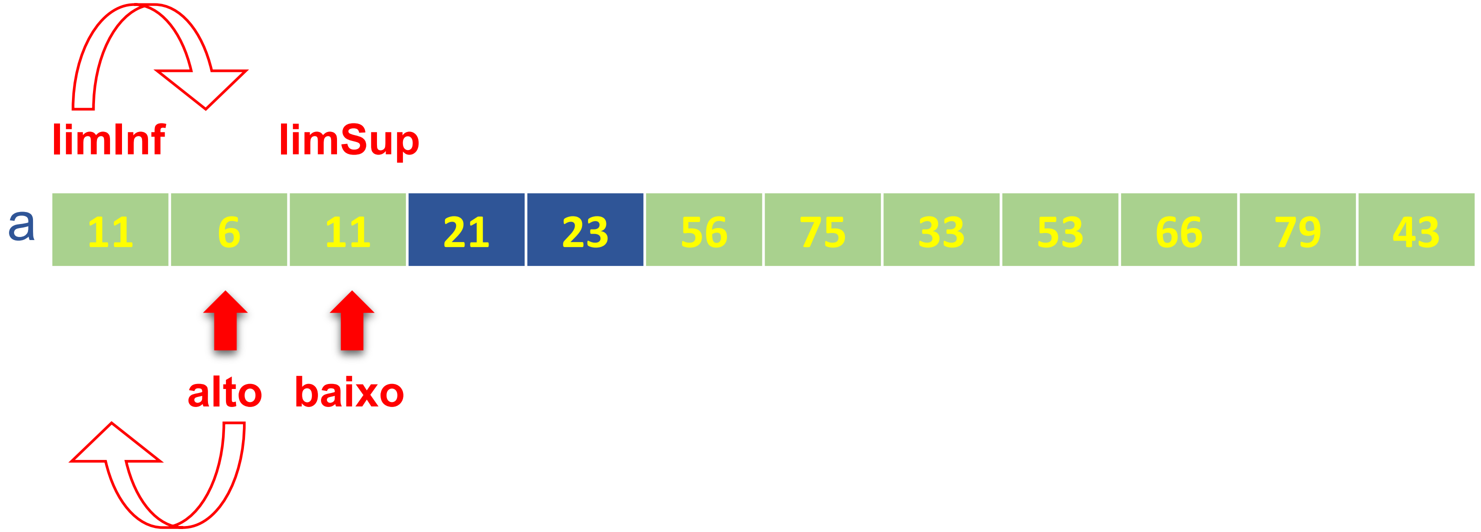
limInf limSup

a												
	11	6	11	21	23	56	75	33	53	66	79	43

alto baixo

- Continuamos a decrementar alto até encontrar um elemento menor que o pivô (11). Nesse caso achamos 6 e alto é menor que baixo, então atingimos a condição de parada!

Simulação de Quicksort



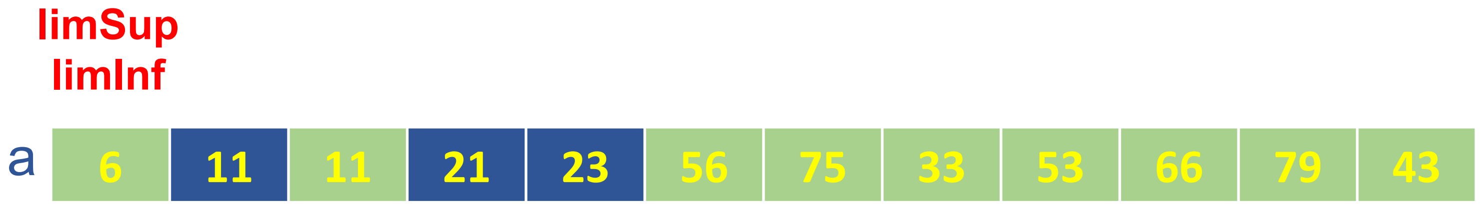
- Agora trocamos o pivô a[limInf] com a[alto] e a divisão do vetor está completa!

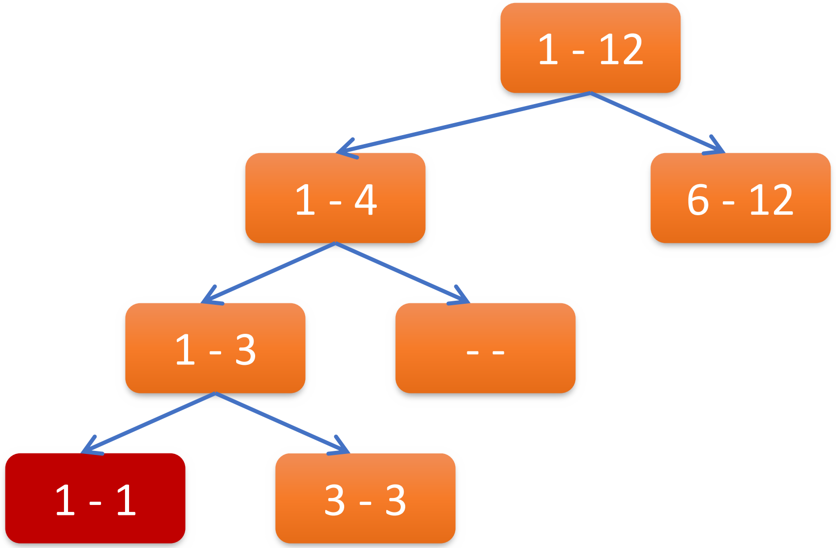
Simulação de Quicksort

	limInf		limSup									
a	6	11	11	21	23	56	75	33	53	66	79	43
		alto	baixo									

- Troca realizada

Simulação de Quicksort





- Agora chamamos a partição com limSup = alto - 1 = 1 e particionamos o subvetor esquerdo.

- Como `limSup > limInf` é falso, paramos.

Algoritmo de particionamento

```
inteiro partição(tInfo: a[], inteiro: limInf, limSup)
   variáveis
         tInfo: pivo, temp;
         inteiro: baixo, alto;
   início
         pivô <- a[limInf];
         alto <- limSup;
         baixo <- limInf + 1;
         enquanto (baixo < alto) faça
                    enquanto (a[baixo] < pivô E baixo < limSup) faça
                               incremente baixo;  // Sobe no arquivo.
                    enquanto (a[alto] => pivô) faça
                               decremente alto;  // Desce no arquivo.
                    se (baixo < alto) então  // Troca.
                               temp <- a[baixo];
                               a[baixo] <- a[alto];
                               a[alto] <- temp;
                    fim se
         fim enquanto
         a[limInf] <- a[alto];
         a[alto] <- pivô;
         retorne alto;
   fim
```

Comentários

- Para evitar o pior caso e casos ruins onde elementos estão em grupos ordenados pode-se utilizar uma estratégia probabilística:
  - Selecione para pivô, ao invés do primeiro, um elemento aleatório. Critérios:
    - seleção totalmente randômica: selecione qualquer elemento do subvetor usando um gerador de números aleatórios;
      - desvantagem: tempo de processamento extra para o gerador.
    - seleção pseudorandômica: selecione um elemento do subvetor com base em um cálculo qualquer baseado em valores que você tem à mão (alto, baixo, chave do primeiro);
    - seleção média: pegue ao invés do elemento inicial, sempre o do meio.
- Todos estes métodos melhoram a performance média.

UniEVANGÉLICA
UNIVERSIDADE EVANGÉLICA DE GOIÁS

Dúvidas?