GBC034 - Ordenação interna

Marcelo K. Albertini

12 de Novembro de 2013

Aula de hoje

Nesta aula veremos:

- Ordenação interna
- Complexidade

Revisão

Conceitos

- Vetor: **int vetor[] = {5, 1, 7, 3, 0};**

Revisão

Conceitos

- Vetor: **int vetor[] = {5, 1, 7, 3, 0};**
- Variável índice: posição para acesso de elemento

Revisão

Conceitos

- Vetor: **int vetor[] = {5, 1, 7, 3, 0};**
- Variável índice: posição para acesso de elemento
- Variável auxiliar: armazenamento temporário

Revisão

Conceitos

- Vetor: **int vetor[] = {5, 1, 7, 3, 0};**
- Variável índice: posição para acesso de elemento
- Variável auxiliar: armazenamento temporário
  - útil para troca de posição de elementos do vetor

Ordenação interna

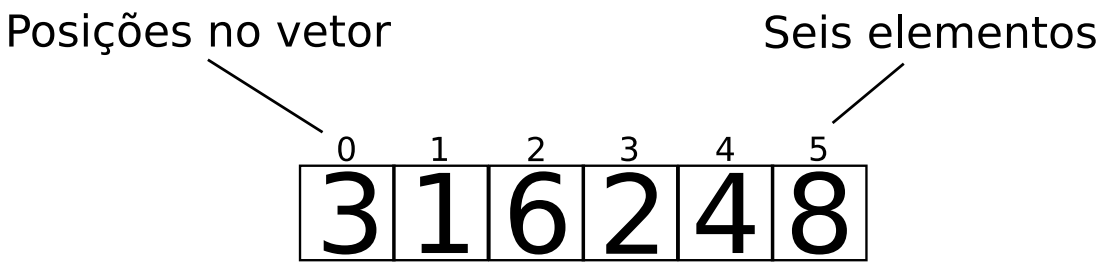
ordenar em memória

- Pré-condições: vetor em memória principal, inicializado com elementos

Ordenação interna

ordenar em memória

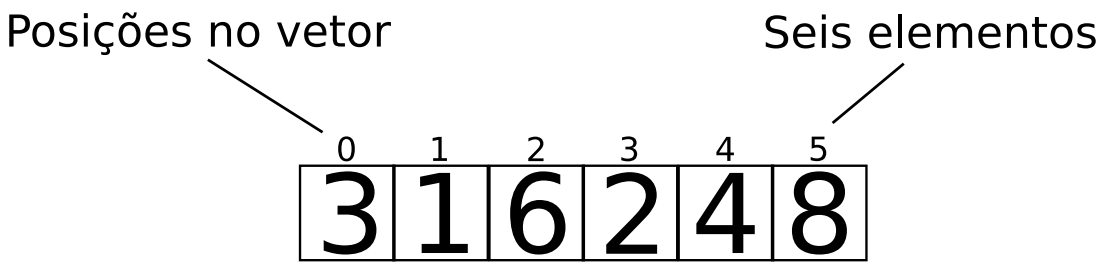
- Pré-condições: vetor em memória principal, inicializado com elementos
- Pós-condições: vetor com elementos em ordem crescente (ou decrescente)



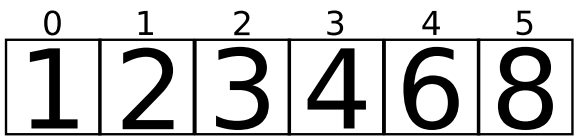
Ordenação interna

ordenar em memória

- Pré-condições: vetor em memória principal, inicializado com elementos
- Pós-condições: vetor com elementos em ordem crescente (ou decrescente)



Como fica ordenado?



Vetores e Ordenação: exemplos

Exemplos:

- Números inteiros ou ponto flutuante

Vetores e Ordenação: exemplos

Exemplos:

- Números inteiros ou ponto flutuante
- Vetor de strings

Vetores e Ordenação: exemplos

Exemplos:

- Números inteiros ou ponto flutuante
- Vetor de strings
- Tipos compostos: necessário definir função para comparar

Vetores e Ordenação: exemplos

Exemplos:

- Números inteiros ou ponto flutuante
- Vetor de strings
- Tipos compostos: necessário definir função para comparar
  - **int compare(ITEM item1, ITEM item2);**

Exemplo

```
1  int compare(Aluno a) {
2      if (this.media > a.media) {
3          return 1;
4      }
5      else if (this.media == a.media) {
6          return (-1);
7      } else {
8          return(0);
9      }
10 }
```

Algoritmo de ordenação

Definição de ordenação

Sequência de comparações e trocas de posição entre elementos para obter vetor ordenado.

Algoritmo de ordenação

Definição de ordenação

Sequência de comparações e trocas de posição entre elementos para obter vetor ordenado.

Complexidade

Quantas trocas, comparações (complexidade de tempo) e variáveis auxiliares (de espaço) são necessárias?

Bubblesort - Ordenação em "bolhas"

Como programar um algoritmo de ordenação simples?

Ideia

Comparar pares consecutivos de elementos e trocá-los de posição caso o primeiro seja maior que o segundo.

```
1 if (vetor[i] > vetor[i+1]) {
2     aux = vetor[i];
3     vetor[i] = vetor[i+1];
4     vetor[i+1] = aux;
5 }
```

Bubblesort - Ordenação em "bolhas"

Como programar um algoritmo de ordenação simples?

Ideia

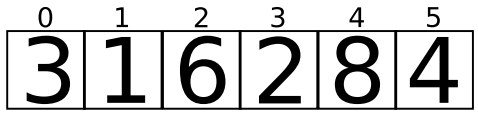
Comparar pares consecutivos de elementos e trocá-los de posição caso o primeiro seja maior que o segundo.

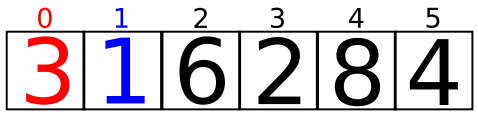
```
1 if (vetor[i] > vetor[i+1]) {
2     aux = vetor[i];
3     vetor[i] = vetor[i+1];
4     vetor[i+1] = aux;
5 }
```

Variável auxiliar aux é essencial.

Primeira iteração

Início da iteração

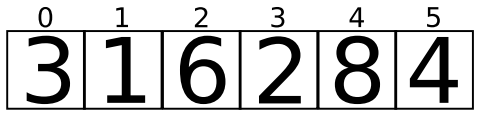




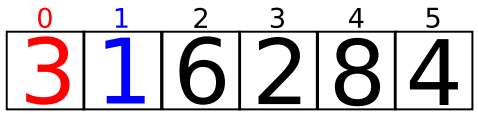
vetor[i]	vetor[i+1]	troca?
3	1	sim

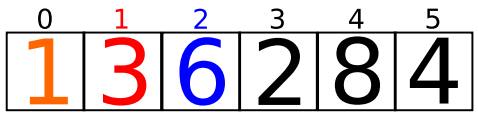
Primeira iteração

Início da iteração



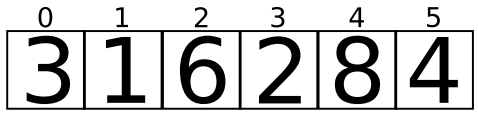
vetor[i]	vetor[i+1]	troca?
3	1	sim
3	6	não

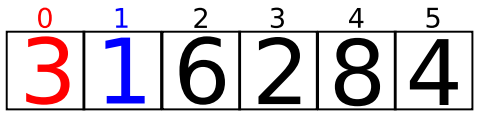
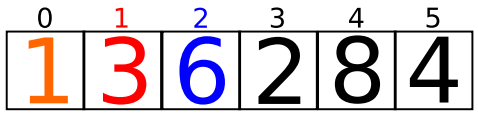
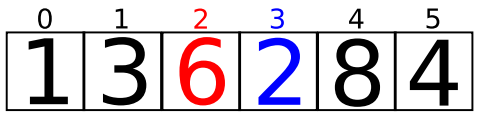




Primeira iteração

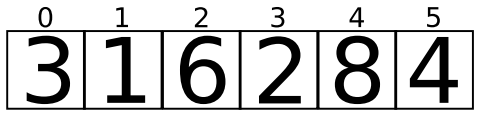
Início da iteração

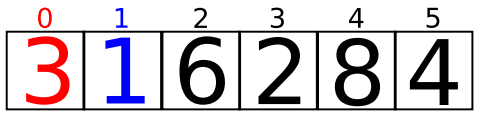
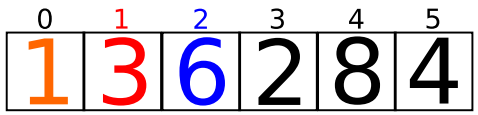
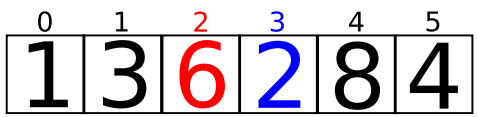
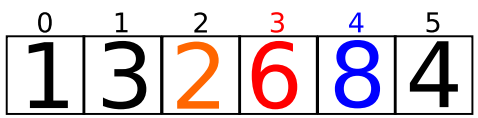


	vetor[i]	vetor[i+1]	troca?
	3	1	sim
	3	6	não
	6	2	sim

Primeira iteração

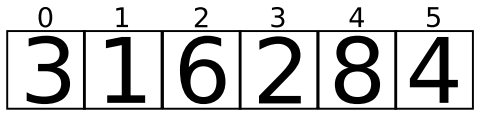
Início da iteração

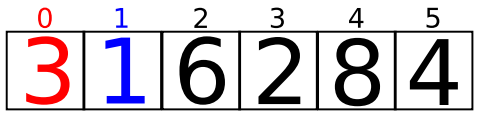
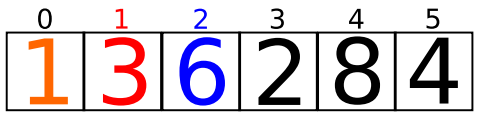
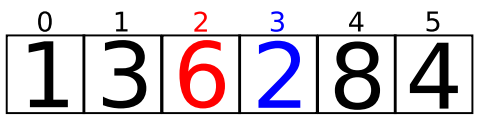
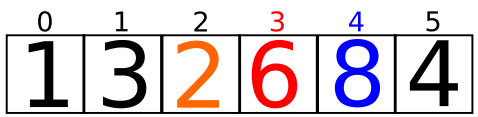
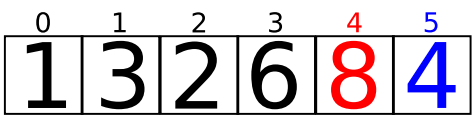


	vetor[i]	vetor[i+1]	troca?
	3	1	sim
	3	6	não
	6	2	sim
	6	8	não

Primeira iteração

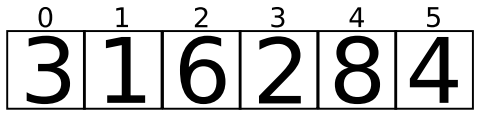
Início da iteração

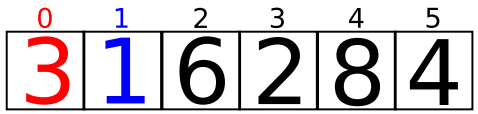
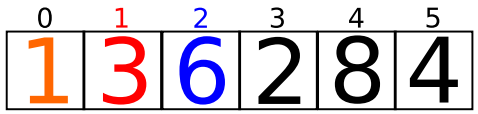
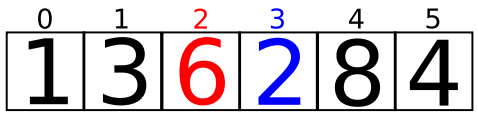
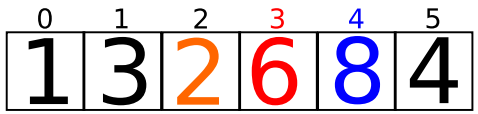
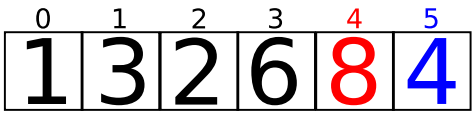


	vetor[i]	vetor[i+1]	troca?
	3	1	sim
	3	6	não
	6	2	sim
	6	8	não
	8	4	sim

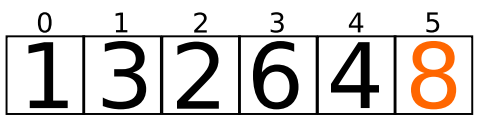
Primeira iteração

Início da iteração



	vetor[i]	vetor[i+1]	troca?
	3	1	sim
	3	6	não
	6	2	sim
	6	8	não
	8	4	sim

Fim da iteração



Primeira iteração

Maior elemento sempre está na sua posição ordenada na primeira iteração.

Algoritmo Bubblesort: versão simplificada

```
1 bubblesort (int vetor[], int nelem) {
```

Algoritmo Bubblesort: versão simplificada

```
1 bubblesort (int vetor[], int nelem) {
2     int i, iteracao, aux;
```

Algoritmo Bubblesort: versão simplificada

```
1 bubblesort (int vetor[], int nelem) {
2     int i, iteracao, aux;
3
4     /*controle do número de iterações (n - 1)*/
5     for (iteracao = 0; iteracao < nelem-1; iteracao++){
```

Algoritmo Bubblesort: versão simplificada

```
1 bubblesort (int vetor[], int nelem) {
2    int i, iteracao, aux;
3
4    /*controle do número de iterações (n - 1)*/
5    for (iteracao = 0; iteracao < nelem-1; iteracao++){
6       /*repeticao interna, percorre vetor (n - 1)*/
7       for (i = 0; i < nelem - 1; i++){
```

Algoritmo Bubblesort: versão simplificada

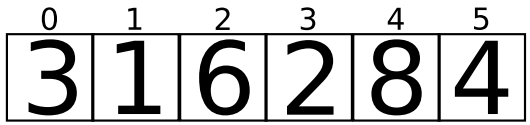
```
1 bubblesort (int vetor[], int nelem) {
2    int i, iteracao, aux;
3
4    /*controle do número de iterações (n - 1)*/
5    for (iteracao = 0; iteracao < nelem-1; iteracao++){
6       /*repeticao interna, percorre vetor (n - 1)*/
7       for (i = 0; i < nelem - 1; i++){
8          if (vetor[i] > vetor[i+1]){
```

Algoritmo Bubblesort: versão simplificada

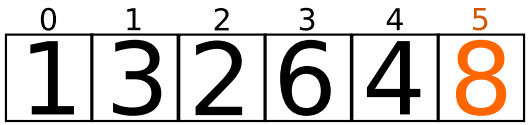
```
1  bubblesort (int vetor[], int nelem) {
2     int i, iteracao, aux;
3
4     /*controle do número de iterações (n - 1)*/
5     for (iteracao = 0; iteracao < nelem-1; iteracao++){
6        /*repeticao interna, percorre vetor (n - 1)*/
7        for (i = 0; i < nelem - 1; i++){
8           if (vetor[i] > vetor[i+1]){
9              /*é necessária uma troca*/
10             aux = vetor[i];
11             vetor[i] = vetor[i+1];
12             vetor[i+1] = aux;
13          }
14       }
15    }
16 }
```

Outras iterações

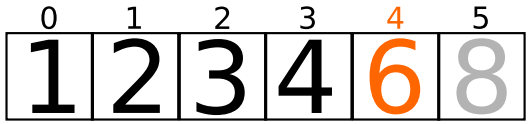
Início



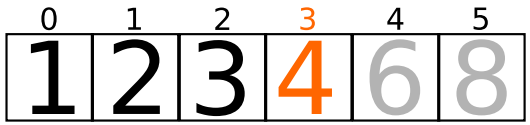
1a iteração



2a iteração



3a iteração



Limite assintótico de complexidade

Limite assintótico superior $O(g(n))$

Objetivo: encontrar função limitante superior $g(n)$ para representar o "teto" do custo do algoritmo.

Limite assintótico de complexidade

Limite assintótico superior $O(g(n))$

Objetivo: encontrar função limitante superior $g(n)$ para representar o "teto" do custo do algoritmo.

Bubblesort simplificado

Para n elementos, faz-se $n-1$ iterações e $n-1$ comparações em cada iteração: $g(n) = (n-1)^2$. Pior caso de número de trocas: $g(n) = n \times (n-1)/2$. Então a complexidade de tempo é $O(n^2)$.

Bucket sort

Ideia: ordenação de números inteiros

Cada elemento é representado por uma posição em um vetor (um balde). Conta-se as repetições de cada número.

Bucket sort

Ideia: ordenação de números inteiros

Cada elemento é representado por uma posição em um vetor (um balde). Conta-se as repetições de cada número.

```
1 static int[] bucketSort(int[] vetor, int max) {
2
3     long[] baldes = new long[max+1];
```

Bucket sort

Ideia: ordenação de números inteiros

Cada elemento é representado por uma posição em um vetor (um balde). Conta-se as repetições de cada número.

```
1 static int[] bucketSort(int[] vetor, int max) {
2
3     long[] baldes = new long[max+1];
4     for (int i = 0; i < vetor.length; i++) {
5         baldes[vetor[i]]++;
6     }
```

Bucket sort

Ideia: ordenação de números inteiros

Cada elemento é representado por uma posição em um vetor (um balde). Conta-se as repetições de cada número.

```
1 static int[] bucketSort(int[] vetor, int max) {
2
3     long[] baldes = new long[max+1];
4     for (int i = 0; i < vetor.length; i++) {
5         baldes[vetor[i]]++;
6     }
7     // olhar cada balde em ordem e tirar os numeros
8     int i = 0;
9     for (int j = 0; j < baldes.length; j++) {
```

Bucket sort

Ideia: ordenação de números inteiros

Cada elemento é representado por uma posição em um vetor (um balde). Conta-se as repetições de cada número.

```
1  static int[] bucketSort(int[] vetor, int max) {
2
3      long[] baldes = new long[max+1];
4      for (int i = 0; i < vetor.length; i++) {
5          baldes[vetor[i]]++;
6      }
7      // olhar cada balde em ordem e tirar os numeros
8      int i = 0;
9      for (int j = 0; j < baldes.length; j++) {
10         while (baldes[j] > 0 ) {
11             baldes[j] = baldes[j] - 1;
```

Bucket sort

Ideia: ordenação de números inteiros

Cada elemento é representado por uma posição em um vetor (um balde). Conta-se as repetições de cada número.

```
1  static int[] bucketSort(int[] vetor, int max) {
2
3      long[] baldes = new long[max+1];
4      for (int i = 0; i < vetor.length; i++) {
5          baldes[vetor[i]]++;
6      }
7      // olhar cada balde em ordem e tirar os numeros
8      int i = 0;
9      for (int j = 0; j < baldes.length; j++) {
10         while (baldes[j] > 0 ) {
11             baldes[j] = baldes[j] - 1;
12             vetor[i] = j;
```

Bucket sort

Ideia: ordenação de números inteiros

Cada elemento é representado por uma posição em um vetor (um balde). Conta-se as repetições de cada número.

```
1  static int[] bucketSort(int[] vetor, int max) {
2
3      long[] baldes = new long[max+1];
4      for (int i = 0; i < vetor.length; i++) {
5          baldes[vetor[i]]++;
6      }
7      // olhar cada balde em ordem e tirar os numeros
8      int i = 0;
9      for (int j = 0; j < baldes.length; j++) {
10         while (baldes[j] > 0 ) {
11             baldes[j] = baldes[j] - 1;
12             vetor[i] = j;
13             i = i + 1;
```

Bucket sort

Ideia: ordenação de números inteiros

Cada elemento é representado por uma posição em um vetor (um balde). Conta-se as repetições de cada número.

```
1  static int[] bucketSort(int[] vetor, int max) {
2
3      long[] baldes = new long[max+1];
4      for (int i = 0; i < vetor.length; i++) {
5          baldes[vetor[i]]++;
6      }
7      // olhar cada balde em ordem e tirar os numeros
8      int i = 0;
9      for (int j = 0; j < baldes.length; j++) {
10         while (baldes[j] > 0 ) {
11             baldes[j] = baldes[j] - 1;
12             vetor[i] = j;
13             i = i + 1;
14         }
15     }
16
```

Bucket sort

Ideia: ordenação de números inteiros

Cada elemento é representado por uma posição em um vetor (um balde). Conta-se as repetições de cada número.

```
1  static int[] bucketSort(int[] vetor, int max) {
2
3      long[] baldes = new long[max+1];
4      for (int i = 0; i < vetor.length; i++) {
5          baldes[vetor[i]]++;
6      }
7      // olhar cada balde em ordem e tirar os numeros
8      int i = 0;
9      for (int j = 0; j < baldes.length; j++) {
10         while (baldes[j] > 0 ) {
11             baldes[j] = baldes[j] - 1;
12             vetor[i] = j;
13             i = i + 1;
14         }
15     }
16
17     return(vetor);
18 }
```

Complexidades

Complexidade de tempo

Como cada número é avaliado apenas uma vez, o Bucket sort tem complexidade de tempo $O(n)$.

Complexidades

Complexidade de tempo

Como cada número é avaliado apenas uma vez, o Bucket sort tem complexidade de tempo $O(n)$.

Complexidade de espaço

Somente viável com inteiros ou poucas casas decimais. Cresce com a faixa de valores consideradas, ou seja, $O(10^{|w|})$, com $|w|$ sendo o tamanho do número.

Complexidade de espaço do Bucket sort

Problema do banco: ordenação de 40 milhões de números

Se 1% das transações forem de mais de 1 milhão de reais, então é possível usar 2 algoritmos de ordenação:

Complexidade de espaço do Bucket sort

Problema do banco: ordenação de 40 milhões de números

Se 1% das transações forem de mais de 1 milhão de reais, então é possível usar 2 algoritmos de ordenação:

- Para transações de até 1 milhão, usa-se o Bucket sort: 4 segundos.
- Para transações de mais de 1 milhão, pode-se usar o Bubble sort: 2.5 minutos.

Complexidade de espaço do Bucket sort

Problema do banco: ordenação de 40 milhões de números

Se 1% das transações forem de mais de 1 milhão de reais, então é possível usar 2 algoritmos de ordenação:

- Para transações de até 1 milhão, usa-se o Bucket sort: 4 segundos.
- Para transações de mais de 1 milhão, pode-se usar o Bubble sort: 2.5 minutos.

Balanceamento de complexidades

Bucket sort: complexidade de tempo baixa e complexidade de espaço alta
Bubble sort: complexidade de tempo alta e complexidade de espaço baixa

Ordenação por seleção

Funcionamento

1. seleciona menor elemento de região não ordenada
2. troca o primeiro elemento da região pelo menor elemento
3. diminui tamanho da região não ordenada

Algoritmo: ordenação por seleção

```
1 public static int[] selectionSort(int[] vetor){
2   for (int i = 0; i < vetor.length-1; i++) {
```

Algoritmo: ordenação por seleção

```
1 public static int[] selectionSort(int[] vetor){
2   for (int i = 0; i < vetor.length-1; i++) {
3     int menor = vetor[i];
```

Algoritmo: ordenação por seleção

```
1 public static int[] selectionSort(int[] vetor){
2   for (int i = 0; i < vetor.length-1; i++) {
3     int menor = vetor[i];
4     int menorI = i;
```

Algoritmo: ordenação por seleção

```
1 public static int[] selectionSort(int[] vetor){
2   for (int i = 0; i < vetor.length-1; i++) {
3     int menor = vetor[i];
4     int menorI = i;
5
6     for (int j = i+1; j < vetor.length; j++) {
```

Algoritmo: ordenação por seleção

```
1 public static int[] selectionSort(int[] vetor){
2   for (int i = 0; i < vetor.length-1; i++) {
3     int menor = vetor[i];
4     int menorI = i;
5
6     for (int j = i+1; j < vetor.length; j++) {
7       if (vetor[j] < menor) {
```

Algoritmo: ordenação por seleção

```
1 public static int[] selectionSort(int[] vetor){
2   for (int i = 0; i < vetor.length-1; i++) {
3     int menor = vetor[i];
4     int menorI = i;
5
6     for (int j = i+1; j < vetor.length; j++) {
7       if (vetor[j] < menor) {
8         menorI = j;
9         menor = vetor[j];
```

Algoritmo: ordenação por seleção

```
1  public static int[] selectionSort(int[] vetor){
2    for (int i = 0; i < vetor.length-1; i++) {
3      int menor = vetor[i];
4      int menorI = i;
5
6      for (int j = i+1; j < vetor.length; j++) {
7        if (vetor[j] < menor) {
8          menorI = j;
9          menor = vetor[j];
10       }
11     }
```

Algoritmo: ordenação por seleção

```
1  public static int[] selectionSort(int[] vetor){
2    for (int i = 0; i < vetor.length-1; i++) {
3      int menor = vetor[i];
4      int menorI = i;
5
6      for (int j = i+1; j < vetor.length; j++) {
7        if (vetor[j] < menor) {
8          menorI = j;
9          menor = vetor[j];
10       }
11     }
12
13     int aux = vetor[i];
14     vetor[i] = menor;
15     vetor[menorI] = aux;
16   }
```

Algoritmo: ordenação por seleção

```
1  public static int[] selectionSort(int[] vetor){
2    for (int i = 0; i < vetor.length-1; i++) {
3      int menor = vetor[i];
4      int menorI = i;
5
6      for (int j = i+1; j < vetor.length; j++) {
7        if (vetor[j] < menor) {
8          menorI = j;
9          menor = vetor[j];
10       }
11     }
12
13     int aux = vetor[i];
14     vetor[i] = menor;
15     vetor[menorI] = aux;
16   }
17   return(vetor);
18 }
```

Custo: ordenação por seleção

Exercício 1

Qual é a função de custo considerando apenas comparações?

Exercício 2

Qual é a função de custo considerando apenas trocas?

Exercício 3: Qual é a complexidade?

Notação O, Notação Ω, Notação Θ

Algoritmo: ordenação por inserção

```
1 public static int[] insertionSort(int[] vetor) {
2
3   int n = vetor.length;
```

Algoritmo: ordenação por inserção

```
1 public static int[] insertionSort(int[] vetor) {
2
3   int n = vetor.length;
4
5   for (int j = 1; j < n; j++) {
```

Algoritmo: ordenação por inserção

```
1 public static int[] insertionSort(int[] vetor) {
2
3   int n = vetor.length;
4
5   for (int j = 1; j < n; j++) {
6     int chave = vetor[j];
```

Algoritmo: ordenação por inserção

```
1 public static int[] insertionSort(int[] vetor) {
2
3   int n = vetor.length;
4
5   for (int j = 1; j < n; j++) {
6     int chave = vetor[j];
7     int i = j - 1;
```

Algoritmo: ordenação por inserção

```
1  public static int[] insertionSort(int[] vetor) {
2
3    int n = vetor.length;
4
5    for (int j = 1; j < n; j++) {
6      int chave = vetor[j];
7      int i = j - 1;
8
9       // procura lugar de insercao e desloca numeros
10     while (i >= 0 && vetor[i] > chave) {
11         vetor[i+1] = vetor[i];
12         i = i - 1;
13     }
```

Algoritmo: ordenação por inserção

```
1  public static int[] insertionSort(int[] vetor) {
2
3    int n = vetor.length;
4
5    for (int j = 1; j < n; j++) {
6      int chave = vetor[j];
7      int i = j - 1;
8
9       // procura lugar de insercao e desloca numeros
10     while (i >= 0 && vetor[i] > chave) {
11         vetor[i+1] = vetor[i];
12         i = i - 1;
13     }
14     vetor[i+1] = chave;
15   }
16
17   return(vetor);
18 }
```

Algoritmo: ordenação por inserção

```
1 insertionSort(int[] vetor){
2
3 int n = vetor.length;
```

Algoritmo: ordenação por inserção

```
1 insertionSort(int[] vetor){
2
3 int n = vetor.length;
4 for (int j = 1; j<n; j++){
```

Algoritmo: ordenação por inserção

```
1 insertionSort(int[] vetor){
2
3 int n = vetor.length;
4 for (int j = 1; j<n; j++){
5   int chave = vetor[j];
```

Algoritmo: ordenação por inserção

```
1 insertionSort(int[] vetor){
2
3 int n = vetor.length;
4 for (int j = 1; j<n; j++){
5   int chave = vetor[j];
6   int i = j - 1;
```

Algoritmo: ordenação por inserção

```
1  insertionSort(int[] vetor){
2
3  int n = vetor.length;
4  for (int j = 1; j<n; j++){
5    int chave = vetor[j];
6    int i = j - 1;
7
8    // desloca numeros
9    while (i >= 0 &&
10           vetor[i] > chave){
11
12     vetor[i+1] = vetor[i];
13     i = i - 1;
14   }
```

Algoritmo: ordenação por inserção

```
1  insertionSort(int[] vetor){
2
3  int n = vetor.length;
4  for (int j = 1; j<n; j++){
5    int chave = vetor[j];
6    int i = j - 1;
7
8    // desloca numeros
9    while (i >= 0 &&
10           vetor[i] > chave){
11
12     vetor[i+1] = vetor[i];
13     i = i - 1;
14   }
15 //insere chave
16     vetor[i+1] = chave;
17   }
18
19 return(vetor);
20 }
```

chave = 5

6	5	2	1	9	6

Algoritmo: ordenação por inserção

```
1  insertionSort(int[] vetor){
2
3  int n = vetor.length;
4  for (int j = 1; j<n; j++){
5    int chave = vetor[j];
6    int i = j - 1;
7
8    // desloca numeros
9    while (i >= 0 &&
10           vetor[i] > chave){
11
12     vetor[i+1] = vetor[i];
13     i = i - 1;
14   }
15 //insere chave
16     vetor[i+1] = chave;
17   }
18
19 return(vetor);
20 }
```

chave = 5

6	5	2	1	9	6
6	5	2	1	9	6

Algoritmo: ordenação por inserção

```
1  insertionSort(int[] vetor){
2
3  int n = vetor.length;
4  for (int j = 1; j<n; j++){
5    int chave = vetor[j];
6    int i = j - 1;
7
8    // desloca numeros
9    while (i >= 0 &&
10           vetor[i] > chave){
11
12     vetor[i+1] = vetor[i];
13     i = i - 1;
14   }
15 //insere chave
16     vetor[i+1] = chave;
17   }
18
19 return(vetor);
20 }
```

chave = 5

6	5	2	1	9	6
6	5	2	1	9	6
6	6	2	1	9	6

Algoritmo: ordenação por inserção

```
1  insertionSort(int[] vetor){
2
3  int n = vetor.length;
4  for (int j = 1; j<n; j++){
5    int chave = vetor[j];
6    int i = j - 1;
7
8    // desloca numeros
9    while (i >= 0 &&
10           vetor[i] > chave){
11
12     vetor[i+1] = vetor[i];
13     i = i - 1;
14   }
15 //insere chave
16     vetor[i+1] = chave;
17   }
18
19 return(vetor);
20 }
```

chave = 5

6	5	2	1	9	6
6	5	2	1	9	6
6	6	2	1	9	6
5	6	2	1	9	6

Algoritmo: ordenação por inserção

```
1  insertionSort(int[] vetor){
2
3  int n = vetor.length;
4  for (int j = 1; j<n; j++){
5    int chave = vetor[j];
6    int i = j - 1;
7
8    // desloca numeros
9    while (i >= 0 &&
10           vetor[i] > chave){
11
12     vetor[i+1] = vetor[i];
13     i = i - 1;
14   }
15 //insere chave
16     vetor[i+1] = chave;
17   }
18
19 return(vetor);
20 }
```

chave = 5

6	5	2	1	9	6
6	5	2	1	9	6
6	6	2	1	9	6
5	6	2	1	9	6

chave = 6

5	6	2	1	9	6

Algoritmo: ordenação por inserção

```
1  insertionSort(int[] vetor){
2
3  int n = vetor.length;
4  for (int j = 1; j<n; j++){
5    int chave = vetor[j];
6    int i = j - 1;
7
8    // desloca numeros
9    while (i >= 0 &&
10           vetor[i] > chave){
11
12     vetor[i+1] = vetor[i];
13     i = i - 1;
14   }
15 //insere chave
16     vetor[i+1] = chave;
17   }
18
19 return(vetor);
20 }
```

chave = 5

6	5	2	1	9	6
6	5	2	1	9	6
6	6	2	1	9	6
5	6	2	1	9	6

chave = 6

5	6	2	1	9	6

chave = 2

5	6	2	1	9	6

Algoritmo: ordenação por inserção

```
1  insertionSort(int[] vetor){
2
3  int n = vetor.length;
4  for (int j = 1; j<n; j++){
5    int chave = vetor[j];
6    int i = j - 1;
7
8    // desloca numeros
9    while (i >= 0 &&
10           vetor[i] > chave){
11
12     vetor[i+1] = vetor[i];
13     i = i - 1;
14   }
15 //insere chave
16     vetor[i+1] = chave;
17   }
18
19 return(vetor);
20 }
```

chave = 5

6	5	2	1	9	6
6	5	2	1	9	6
6	6	2	1	9	6
5	6	2	1	9	6

chave = 6

5	6	2	1	9	6

chave = 2

5	6	2	1	9	6
5	6	6	1	9	6

Algoritmo: ordenação por inserção

```
1  insertionSort(int[] vetor){
2
3  int n = vetor.length;
4  for (int j = 1; j<n; j++){
5    int chave = vetor[j];
6    int i = j - 1;
7
8    // desloca numeros
9    while (i >= 0 &&
10           vetor[i] > chave){
11
12     vetor[i+1] = vetor[i];
13     i = i - 1;
14   }
15 //insere chave
16     vetor[i+1] = chave;
17   }
18
19 return(vetor);
20 }
```

chave = 5

6	5	2	1	9	6
6	5	2	1	9	6
6	6	2	1	9	6
5	6	2	1	9	6

chave = 6

5	6	2	1	9	6

chave = 2

5	6	2	1	9	6
5	6	6	1	9	6
5	5	6	1	9	6

Algoritmo: ordenação por inserção

```
1  insertionSort(int[] vetor){
2
3  int n = vetor.length;
4  for (int j = 1; j<n; j++){
5    int chave = vetor[j];
6    int i = j - 1;
7
8    // desloca numeros
9    while (i >= 0 &&
10           vetor[i] > chave){
11
12     vetor[i+1] = vetor[i];
13     i = i - 1;
14   }
15 //insere chave
16     vetor[i+1] = chave;
17   }
18
19 return(vetor);
20 }
```

chave = 5

6	5	2	1	9	6
6	5	2	1	9	6
6	6	2	1	9	6
5	6	2	1	9	6

chave = 6

5	6	2	1	9	6

chave = 2

5	6	2	1	9	6
5	6	6	1	9	6
5	5	6	1	9	6
2	5	6	1	9	6

Algoritmo: ordenação por inserção

```
1  insertionSort(int[] vetor){
2
3  int n = vetor.length;
4  for (int j = 1; j<n; j++){
5    int chave = vetor[j];
6    int i = j - 1;
7
8    // desloca numeros
9    while (i >= 0 &&
10           vetor[i] > chave){
11
12     vetor[i+1] = vetor[i];
13     i = i - 1;
14   }
15 //insere chave
16     vetor[i+1] = chave;
17   }
18
19 return(vetor);
20 }
```

chave = 5

6	5	2	1	9	6
6	5	2	1	9	6
6	6	2	1	9	6
5	6	2	1	9	6

chave = 6

5	6	2	1	9	6

chave = 2

5	6	2	1	9	6
5	6	6	1	9	6
5	5	6	1	9	6
2	5	6	1	9	6

chave = 1

2	5	6	1	9	6

Algoritmo: ordenação por inserção

```
1  insertionSort(int[] vetor){
2
3  int n = vetor.length;
4  for (int j = 1; j<n; j++){
5    int chave = vetor[j];
6    int i = j - 1;
7
8    // desloca numeros
9    while (i >= 0 &&
10           vetor[i] > chave){
11
12     vetor[i+1] = vetor[i];
13     i = i - 1;
14   }
15 //insere chave
16     vetor[i+1] = chave;
17   }
18
19 return(vetor);
20 }
```

chave = 5

6	5	2	1	9	6
6	5	2	1	9	6
6	6	2	1	9	6
5	6	2	1	9	6

chave = 6

5	6	2	1	9	6

chave = 2

5	6	2	1	9	6
5	6	6	1	9	6
5	5	6	1	9	6
2	5	6	1	9	6

chave = 1

2	5	6	1	9	6
2	5	6	6	9	6

Algoritmo: ordenação por inserção

```
1  insertionSort(int[] vetor){
2
3  int n = vetor.length;
4  for (int j = 1; j<n; j++){
5    int chave = vetor[j];
6    int i = j - 1;
7
8    // desloca numeros
9    while (i >= 0 &&
10           vetor[i] > chave){
11
12     vetor[i+1] = vetor[i];
13     i = i - 1;
14   }
15 //insere chave
16     vetor[i+1] = chave;
17   }
18
19 return(vetor);
20 }
```

chave = 5

6	5	2	1	9	6
6	5	2	1	9	6
6	6	2	1	9	6
5	6	2	1	9	6

chave = 6

5	6	2	1	9	6

chave = 2

5	6	2	1	9	6
5	6	6	1	9	6
5	5	6	1	9	6
2	5	6	1	9	6

chave = 1

2	5	6	1	9	6
2	5	6	6	9	6
2	5	5	6	9	6

Algoritmo: ordenação por inserção

```
1  insertionSort(int[] vetor){
2
3  int n = vetor.length;
4  for (int j = 1; j<n; j++){
5    int chave = vetor[j];
6    int i = j - 1;
7
8    // desloca numeros
9    while (i >= 0 &&
10           vetor[i] > chave){
11
12     vetor[i+1] = vetor[i];
13     i = i - 1;
14   }
15 //insere chave
16     vetor[i+1] = chave;
17   }
18
19 return(vetor);
20 }
```

chave = 5

6	5	2	1	9	6
6	5	2	1	9	6
6	6	2	1	9	6
5	6	2	1	9	6

chave = 6

5	6	2	1	9	6

chave = 2

5	6	2	1	9	6
5	6	6	1	9	6
5	5	6	1	9	6
2	5	6	1	9	6

chave = 1

2	5	6	1	9	6
2	5	6	6	9	6
2	5	5	6	9	6
2	2	5	6	9	6

Algoritmo: ordenação por inserção

```
1  insertionSort(int[] vetor){
2
3  int n = vetor.length;
4  for (int j = 1; j<n; j++){
5    int chave = vetor[j];
6    int i = j - 1;
7
8    // desloca numeros
9    while (i >= 0 &&
10           vetor[i] > chave){
11
12     vetor[i+1] = vetor[i];
13     i = i - 1;
14   }
15 //insere chave
16     vetor[i+1] = chave;
17   }
18
19 return(vetor);
20 }
```

chave = 5

6	5	2	1	9	6
6	5	2	1	9	6
6	6	2	1	9	6
5	6	2	1	9	6

chave = 6

5	6	2	1	9	6

chave = 2

5	6	2	1	9	6
5	6	6	1	9	6
5	5	6	1	9	6
2	5	6	1	9	6

chave = 1

2	5	6	1	9	6
2	5	6	6	9	6
2	5	5	6	9	6
2	2	5	6	9	6
1	2	5	6	9	6

Algoritmo: ordenação por inserção

```
1  insertionSort(int[] vetor){
2
3  int n = vetor.length;
4  for (int j = 1; j<n; j++){
5    int chave = vetor[j];
6    int i = j - 1;
7
8    // desloca numeros
9    while (i >= 0 &&
10           vetor[i] > chave){
11
12     vetor[i+1] = vetor[i];
13     i = i - 1;
14   }
15 //insere chave
16     vetor[i+1] = chave;
17   }
18
19 return(vetor);
20 }
```

chave = 5

6	5	2	1	9	6
6	5	2	1	9	6
6	6	2	1	9	6
5	6	2	1	9	6

chave = 6

5	6	2	1	9	6

chave = 2

5	6	2	1	9	6
5	6	6	1	9	6
5	5	6	1	9	6
2	5	6	1	9	6

chave = 1

2	5	6	1	9	6
2	5	6	6	9	6
2	5	5	6	9	6
2	2	5	6	9	6
1	2	5	6	9	6

chave = 9

1	2	5	6	9	6

Algoritmo: ordenação por inserção

```
1  insertionSort(int[] vetor){
2
3  int n = vetor.length;
4  for (int j = 1; j<n; j++){
5    int chave = vetor[j];
6    int i = j - 1;
7
8    // desloca numeros
9    while (i >= 0 &&
10           vetor[i] > chave){
11
12     vetor[i+1] = vetor[i];
13     i = i - 1;
14   }
15 //insere chave
16     vetor[i+1] = chave;
17   }
18
19 return(vetor);
20 }
```

chave = 5

6	5	2	1	9	6
6	5	2	1	9	6
6	6	2	1	9	6
5	6	2	1	9	6

chave = 6

5	6	2	1	9	6

chave = 2

5	6	2	1	9	6
5	6	6	1	9	6
5	5	6	1	9	6
2	5	6	1	9	6

chave = 1

2	5	6	1	9	6
2	5	6	6	9	6
2	5	5	6	9	6
2	2	5	6	9	6
1	2	5	6	9	6

chave = 9

1	2	5	6	9	6

chave = 6

1	2	5	6	9	6

Algoritmo: ordenação por inserção

```
1  insertionSort(int[] vetor){
2
3  int n = vetor.length;
4  for (int j = 1; j<n; j++){
5    int chave = vetor[j];
6    int i = j - 1;
7
8    // desloca numeros
9    while (i >= 0 &&
10           vetor[i] > chave){
11
12     vetor[i+1] = vetor[i];
13     i = i - 1;
14   }
15 //insere chave
16     vetor[i+1] = chave;
17   }
18
19 return(vetor);
20 }
```

chave = 5

6	5	2	1	9	6
6	5	2	1	9	6
6	6	2	1	9	6
5	6	2	1	9	6

chave = 6

5	6	2	1	9	6

chave = 2

5	6	2	1	9	6
5	6	6	1	9	6
5	5	6	1	9	6
2	5	6	1	9	6

chave = 1

2	5	6	1	9	6
2	5	6	6	9	6
2	5	5	6	9	6
2	2	5	6	9	6
1	2	5	6	9	6

chave = 9

1	2	5	6	9	6

chave = 6

1	2	5	6	9	6
1	2	5	6	9	9

Algoritmo: ordenação por inserção

```
1  insertionSort(int[] vetor){
2
3  int n = vetor.length;
4  for (int j = 1; j<n; j++){
5    int chave = vetor[j];
6    int i = j - 1;
7
8    // desloca numeros
9    while (i >= 0 &&
10           vetor[i] > chave){
11
12     vetor[i+1] = vetor[i];
13     i = i - 1;
14   }
15 //insere chave
16     vetor[i+1] = chave;
17   }
18
19 return(vetor);
20 }
```

chave = 5

6	5	2	1	9	6
6	5	2	1	9	6
6	6	2	1	9	6
5	6	2	1	9	6

chave = 6

5	6	2	1	9	6

chave = 2

5	6	2	1	9	6
5	6	6	1	9	6
5	5	6	1	9	6
2	5	6	1	9	6

chave = 1

2	5	6	1	9	6
2	5	6	6	9	6
2	5	5	6	9	6
2	2	5	6	9	6
1	2	5	6	9	6

chave = 9

1	2	5	6	9	6

chave = 6

1	2	5	6	9	6
1	2	5	6	9	9
1	2	5	6	6	9

Custo: ordenação por inserção

Custo e complexidade

- Melhor caso, notação Ω
- Pior Caso, notação O

Shell Sort

- Inventado por Donald Shell em 1959
- Primeiro a baixar a complexidade de ordenação

Ideia

Similar ao Insert Sort, mas compara elementos distantes em vez de elementos consecutivos

Shell Sort

- Inventado por Donald Shell em 1959
- Primeiro a baixar a complexidade de ordenação

Ideia

Similar ao Insert Sort, mas compara elementos distantes em vez de elementos consecutivos

Complexidade empírica

Entre $O(n^{1.2})$ e $O(1.6n^{1.25})$

Quicksort

Quicksort

- Tony Hoare, Moscou, União Soviética, 1960
- Aplicação original: ordenar dicionário russo-inglês

Pseudo-código: quicksort simplificado

```
1 int[] quicksort(vetor) {
2   if (tamanho(vetor) <= 1) {
3     return(vetor) // vetor já ordenado
4   }
```

Pseudo-código: quicksort simplificado

```
1 int[] quicksort(vetor) {
2   if (tamanho(vetor) <= 1) {
3     return(vetor) // vetor já ordenado
4   }
5 // pivot é elemento de referencia para ordenar
6   pivot = escolher_E_remover(vetor)
```

Pseudo-código: quicksort simplificado

```
1 int[] quicksort(vetor) {
2   if (tamanho(vetor) <= 1) {
3     return(vetor) // vetor já ordenado
4   }
5 // pivot é elemento de referencia para ordenar
6   pivot = escolher_E_remover(vetor)
7 // criar vetores de elementos menores e maiores
8   vetorMenores = new int[tamanho(vetor)]
```

Pseudo-código: quicksort simplificado

```
1 int[] quicksort(vetor) {
2   if (tamanho(vetor) <= 1) {
3     return(vetor) // vetor já ordenado
4   }
5 // pivot é elemento de referencia para ordenar
6   pivot = escolher_E_remover(vetor)
7 // criar vetores de elementos menores e maiores
8   vetorMenores = new int[tamanho(vetor)]
9   vetorMaiores = new int[tamanho(vetor)]
```

Pseudo-código: quicksort simplificado

```
1  int[] quicksort(vetor) {
2    if (tamanho(vetor) <= 1) {
3      return(vetor) // vetor já ordenado
4    }
5  // pivot é elemento de referencia para ordenar
6    pivot = escolher_E_remover(vetor)
7  // criar vetores de elementos menores e maiores
8    vetorMenores = new int[tamanho(vetor)]
9    vetorMaiores = new int[tamanho(vetor)]
10
11   for (x in vetor) {
```

Pseudo-código: quicksort simplificado

```
1  int[] quicksort(vetor) {
2    if (tamanho(vetor) <= 1) {
3      return(vetor) // vetor já ordenado
4    }
5  // pivot é elemento de referencia para ordenar
6    pivot = escolher_E_remover(vetor)
7  // criar vetores de elementos menores e maiores
8    vetorMenores = new int[tamanho(vetor)]
9    vetorMaiores = new int[tamanho(vetor)]
10
11   for (x in vetor) {
12     if (x <= pivot)
```

Pseudo-código: quicksort simplificado

```
1  int[] quicksort(vetor) {
2    if (tamanho(vetor) <= 1) {
3      return(vetor) // vetor já ordenado
4    }
5  // pivot é elemento de referencia para ordenar
6    pivot = escolher_E_remover(vetor)
7  // criar vetores de elementos menores e maiores
8    vetorMenores = new int[tamanho(vetor)]
9    vetorMaiores = new int[tamanho(vetor)]
10
11   for (x in vetor) {
12     if (x <= pivot)
13       insereNoFim(x, vetorMenores)
```

Pseudo-código: quicksort simplificado

```
1  int[] quicksort(vetor) {
2    if (tamanho(vetor) <= 1) {
3      return(vetor) // vetor já ordenado
4    }
5  // pivot é elemento de referencia para ordenar
6    pivot = escolher_E_remover(vetor)
7  // criar vetores de elementos menores e maiores
8    vetorMenores = new int[tamanho(vetor)]
9    vetorMaiores = new int[tamanho(vetor)]
10
11   for (x in vetor) {
12     if (x <= pivot)
13       insereNoFim(x, vetorMenores)
14     else
```

Pseudo-código: quicksort simplificado

```
1  int[] quicksort(vetor) {
2    if (tamanho(vetor) <= 1) {
3      return(vetor) // vetor já ordenado
4    }
5  // pivot é elemento de referencia para ordenar
6    pivot = escolher_E_remover(vetor)
7  // criar vetores de elementos menores e maiores
8    vetorMenores = new int[tamanho(vetor)]
9    vetorMaiores = new int[tamanho(vetor)]
10
11   for (x in vetor) {
12       if (x <= pivot)
13           insereNoFim(x, vetorMenores)
14       else
15           insereNoFim(x, vetorMaiores)
```

Pseudo-código: quicksort simplificado

```
1  int[] quicksort(vetor) {
2    if (tamanho(vetor) <= 1) {
3      return(vetor) // vetor já ordenado
4    }
5  // pivot é elemento de referencia para ordenar
6    pivot = escolher_E_remover(vetor)
7  // criar vetores de elementos menores e maiores
8    vetorMenores = new int[tamanho(vetor)]
9    vetorMaiores = new int[tamanho(vetor)]
10
11   for (x in vetor) {
12       if (x <= pivot)
13           insereNoFim(x, vetorMenores)
14       else
15           insereNoFim(x, vetorMaiores)
16   }
```

Pseudo-código: quicksort simplificado

```
1  int[] quicksort(vetor) {
2    if (tamanho(vetor) <= 1) {
3      return(vetor) // vetor já ordenado
4    }
5  // pivot é elemento de referencia para ordenar
6    pivot = escolher_E_remover(vetor)
7  // criar vetores de elementos menores e maiores
8    vetorMenores = new int[tamanho(vetor)]
9    vetorMaiores = new int[tamanho(vetor)]
10
11   for (x in vetor) {
12       if (x <= pivot)
13           insereNoFim(x, vetorMenores)
14       else
15           insereNoFim(x, vetorMaiores)
16   }
17   quicksort(vetorMenores) // ordenar os menores
18
```

Pseudo-código: quicksort simplificado

```
1  int[] quicksort(vetor) {
2    if (tamanho(vetor) <= 1) {
3      return(vetor) // vetor já ordenado
4    }
5  // pivot é elemento de referencia para ordenar
6    pivot = escolher_E_remover(vetor)
7  // criar vetores de elementos menores e maiores
8    vetorMenores = new int[tamanho(vetor)]
9    vetorMaiores = new int[tamanho(vetor)]
10
11   for (x in vetor) {
12       if (x <= pivot)
13           insereNoFim(x, vetorMenores)
14       else
15           insereNoFim(x, vetorMaiores)
16   }
17   quicksort(vetorMenores) // ordenar os menores
18   quicksort(vetorMaiores) // ordenar os maiores
19
```

Pseudo-código: quicksort simplificado

```
1  int[] quicksort(vetor) {
2    if (tamanho(vetor) <= 1) {
3      return(vetor) // vetor já ordenado
4    }
5  // pivot é elemento de referencia para ordenar
6    pivot = escolher_E_remover(vetor)
7  // criar vetores de elementos menores e maiores
8    vetorMenores = new int[tamanho(vetor)]
9    vetorMaiores = new int[tamanho(vetor)]
10
11   for (x in vetor) {
12       if (x <= pivot)
13           insereNoFim(x, vetorMenores)
14       else
15           insereNoFim(x, vetorMaiores)
16   }
17   quicksort(vetorMenores) // ordenar os menores
18   quicksort(vetorMaiores) // ordenar os maiores
19   return(juntar(vetorMenores, pivot, vetorMaiores));
20 }
```

Escolha do pivot

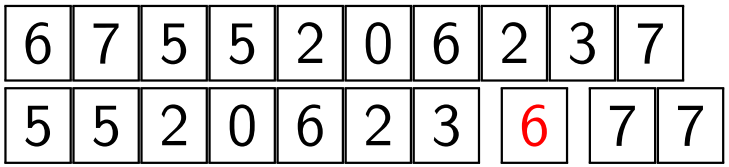
```
1  pivot = escolher_E_remover(vetor)
```

- custo do algoritmo depende da escolha do pivot
- possibilidades
  - o primeiro/último elemento
  - aleatório
  - o elemento do meio
  - a mediana entre o primeiro, meio e último

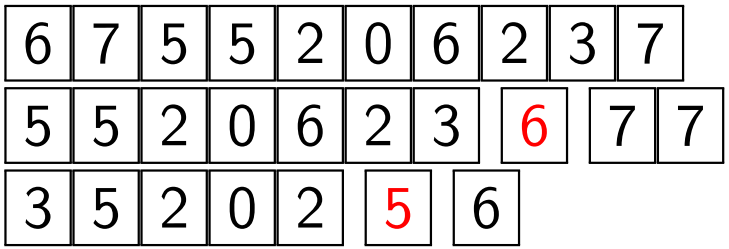
Simulação: pivot – o primeiro elemento

6	7	5	5	2	0	6	2	3	7

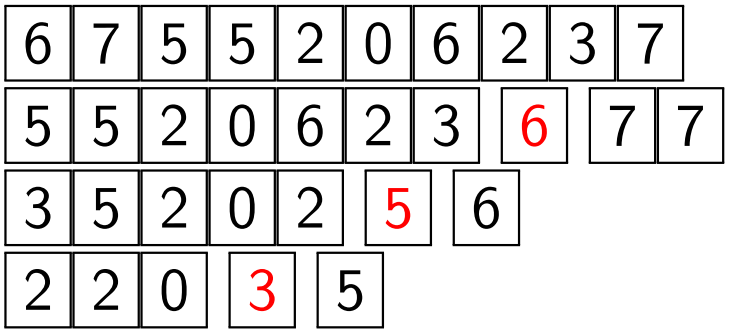
Simulação: pivot – o primeiro elemento



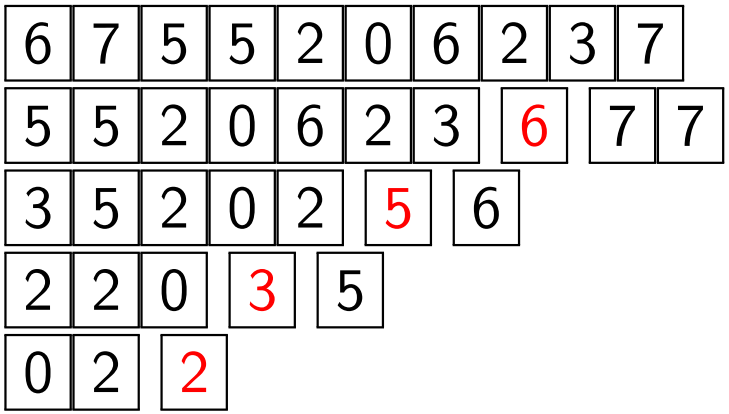
Simulação: pivot – o primeiro elemento



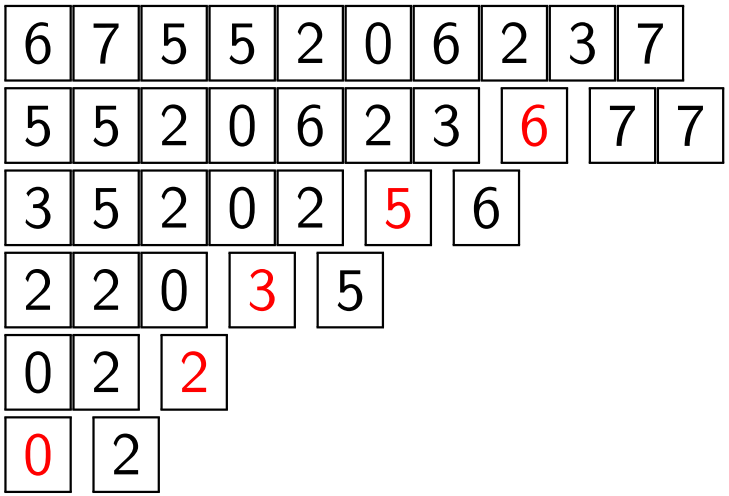
Simulação: pivot – o primeiro elemento



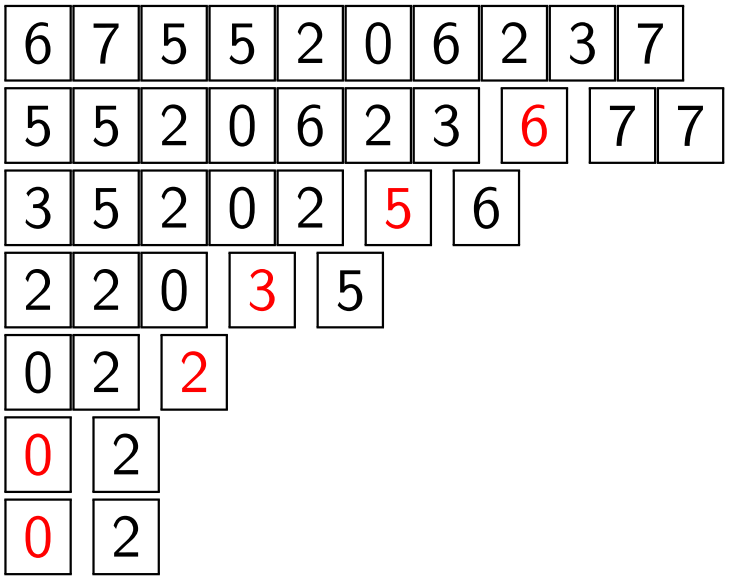
Simulação: pivot – o primeiro elemento



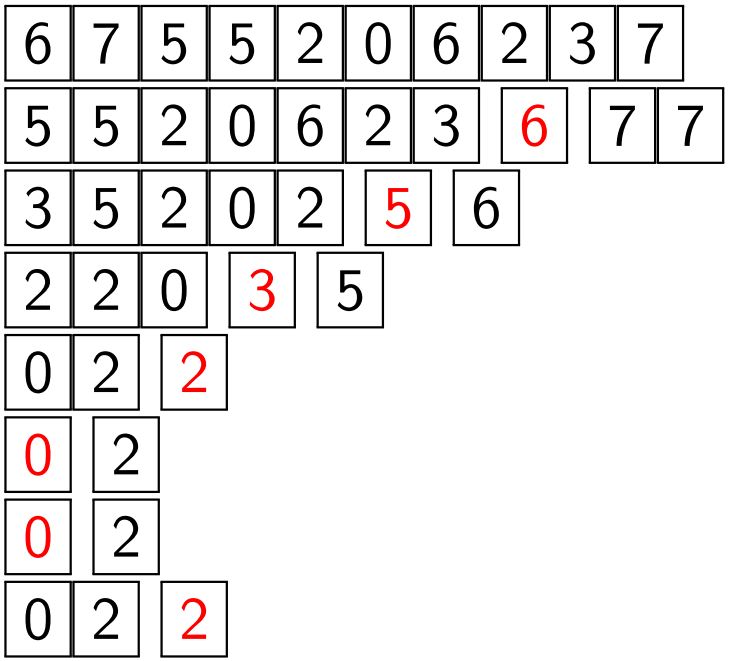
Simulação: pivot – o primeiro elemento



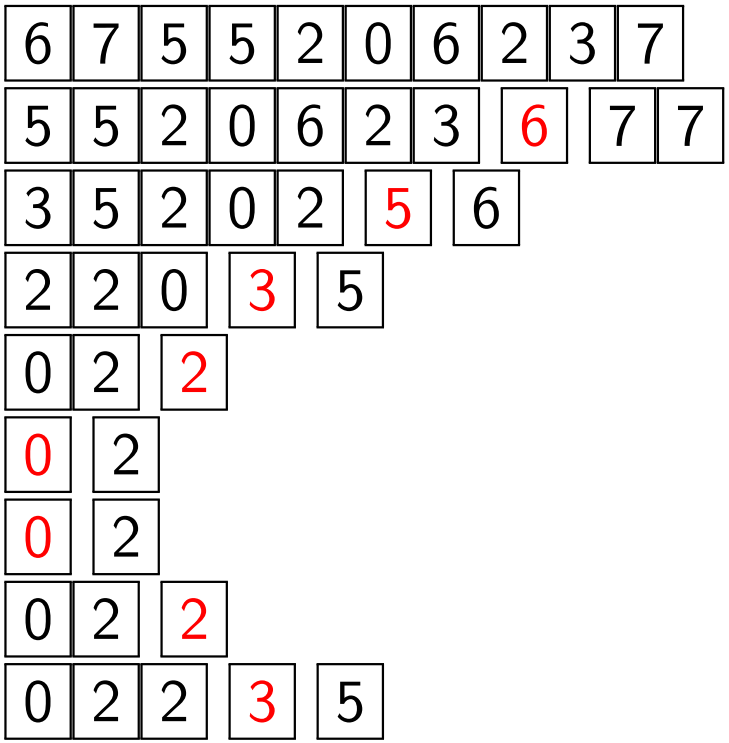
Simulação: pivot – o primeiro elemento



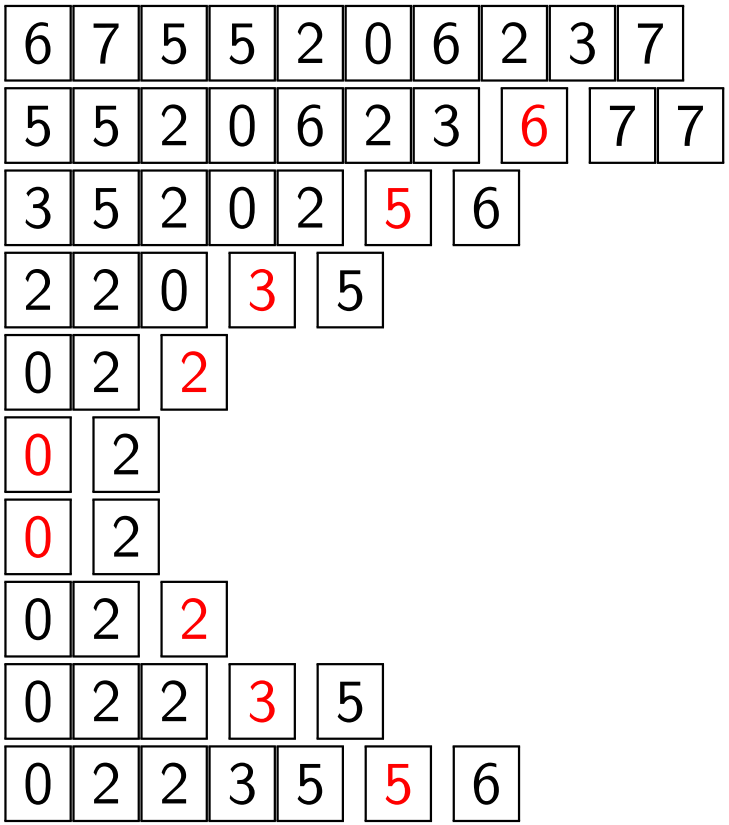
Simulação: pivot – o primeiro elemento



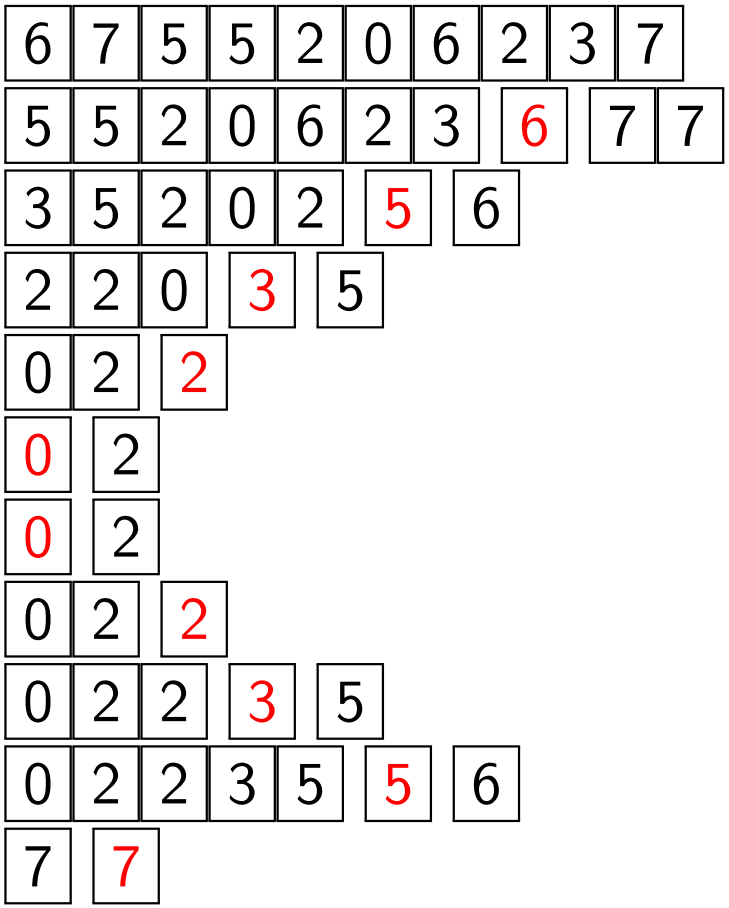
Simulação: pivot – o primeiro elemento



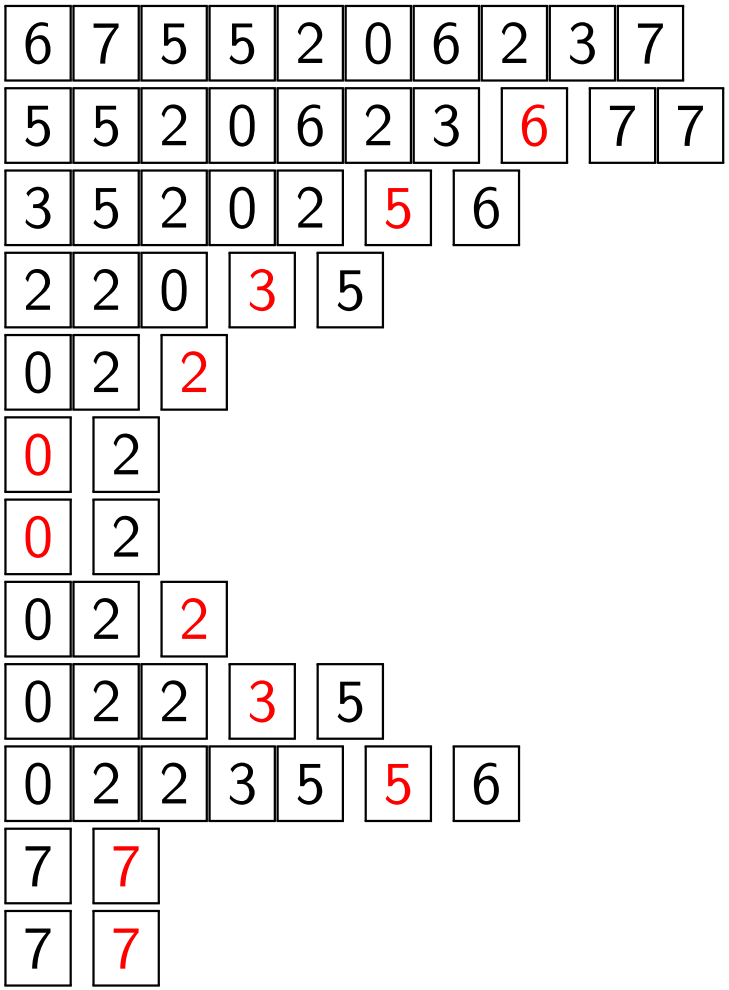
Simulação: pivot – o primeiro elemento



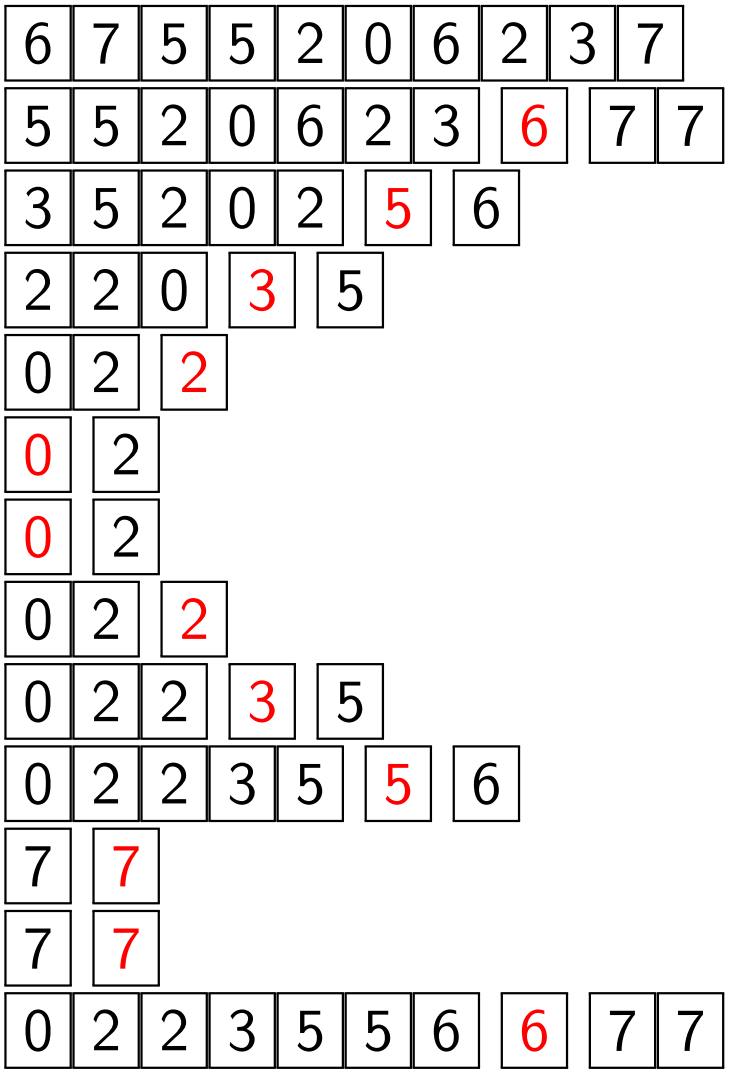
Simulação: pivot – o primeiro elemento



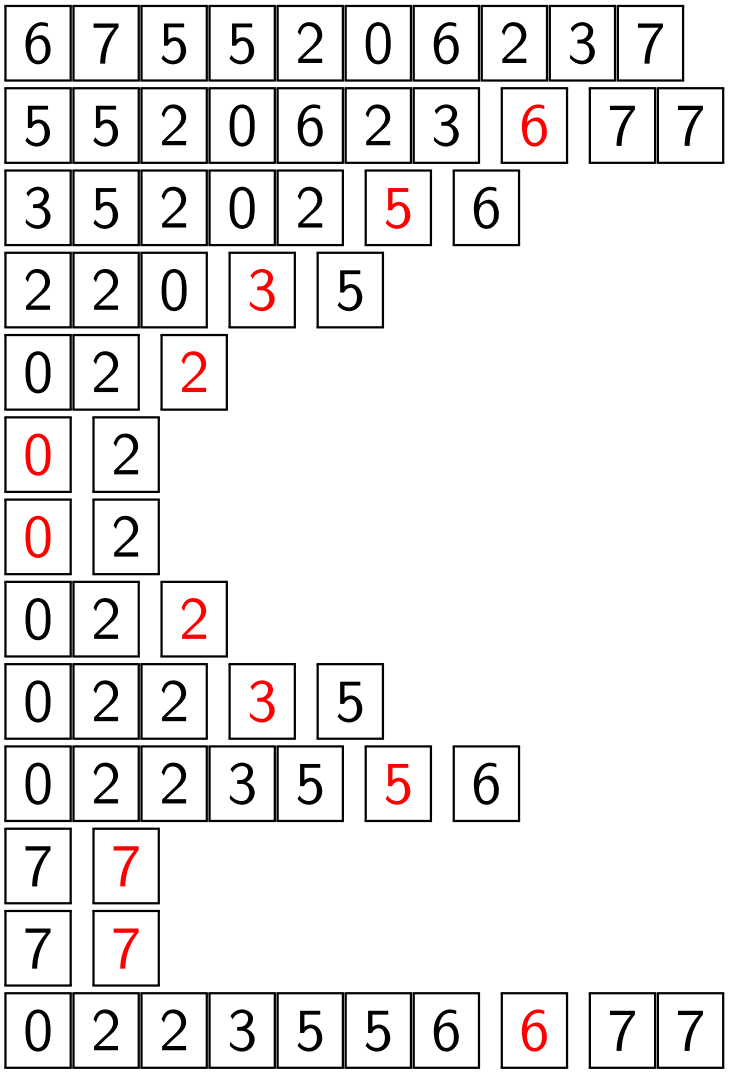
Simulação: pivot – o primeiro elemento



Simulação: pivot – o primeiro elemento



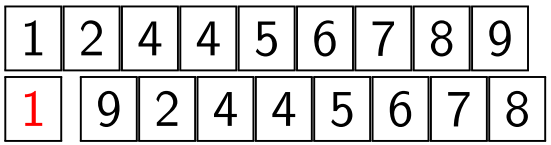
Simulação: pivot – o primeiro elemento



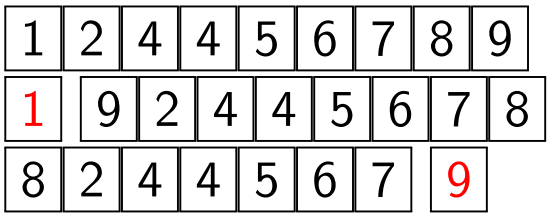
Simulação: pivot – o primeiro elemento – degeneração

1	2	4	4	5	6	7	8	9

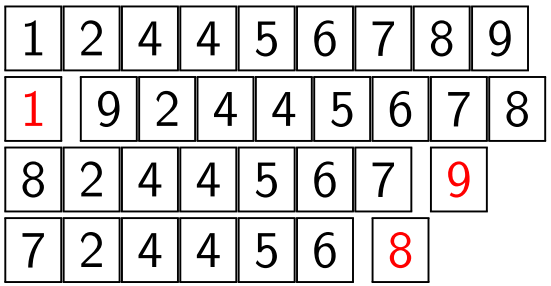
Simulação: pivot – o primeiro elemento – degeneração



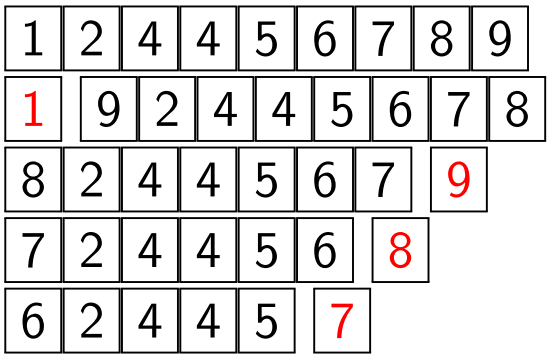
Simulação: pivot – o primeiro elemento – degeneração



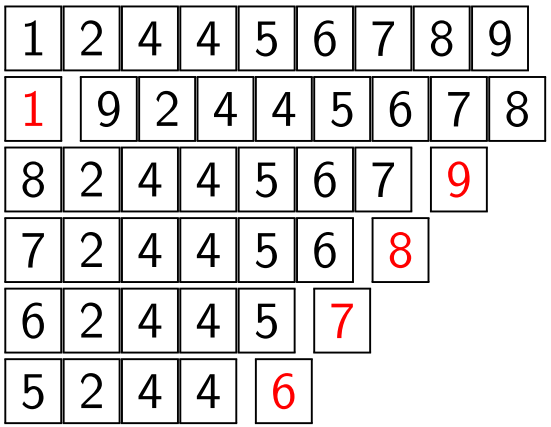
Simulação: pivot – o primeiro elemento – degeneração



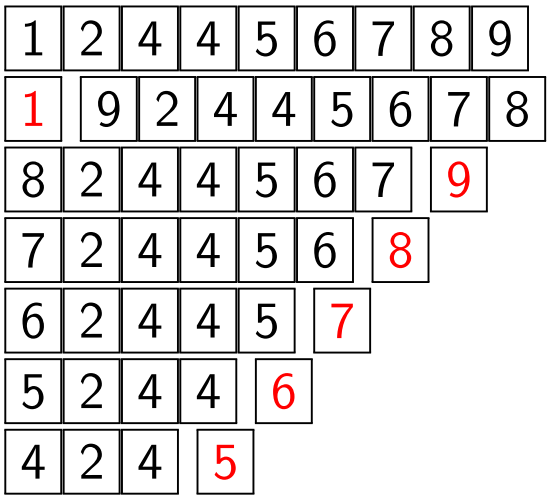
Simulação: pivot – o primeiro elemento – degeneração



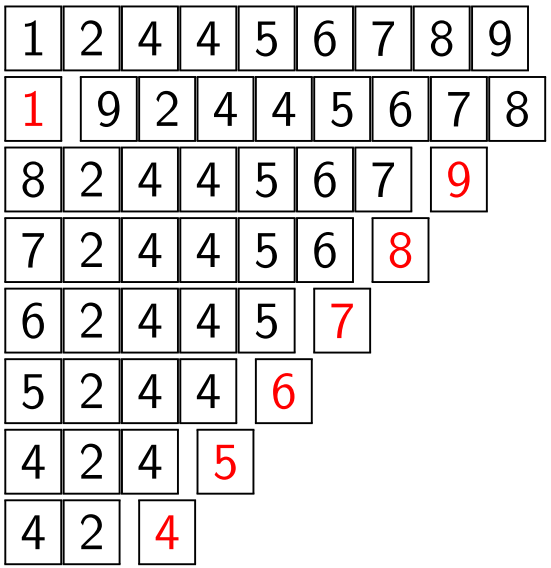
Simulação: pivot – o primeiro elemento – degeneração



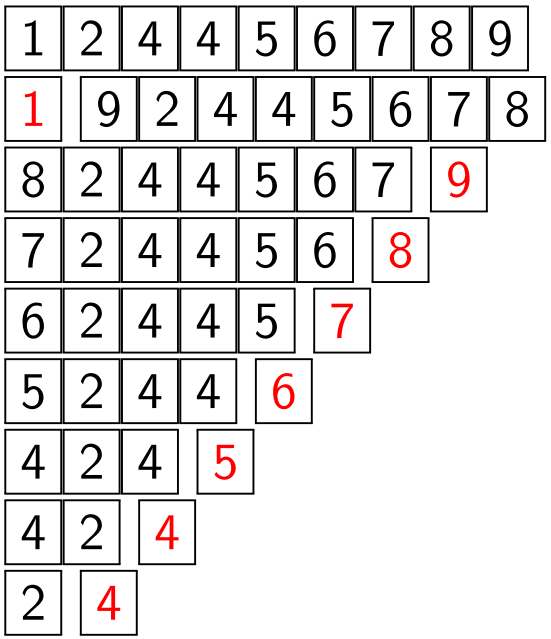
Simulação: pivot – o primeiro elemento – degeneração



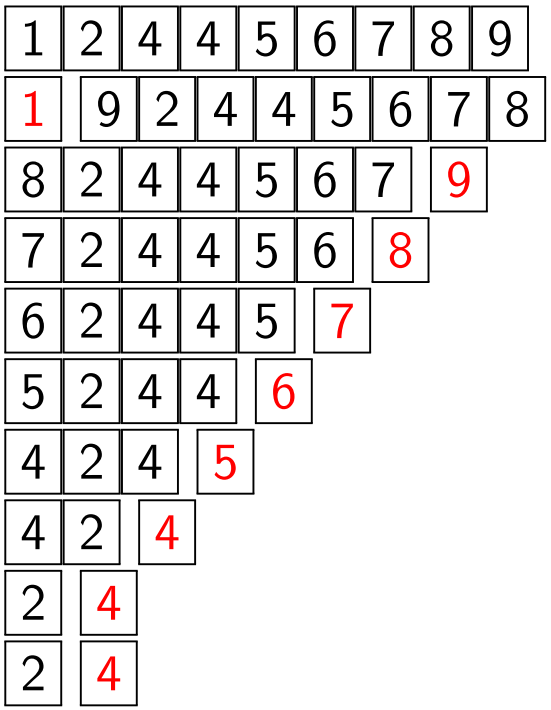
Simulação: pivot – o primeiro elemento – degeneração



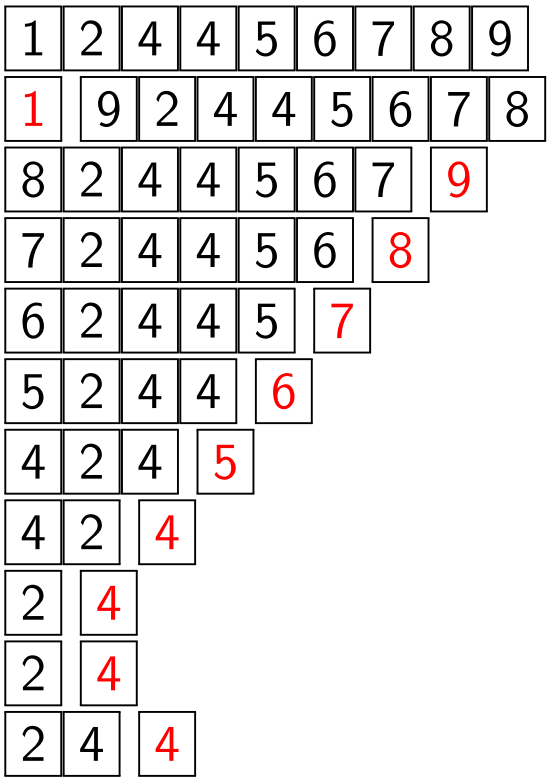
Simulação: pivot – o primeiro elemento – degeneração



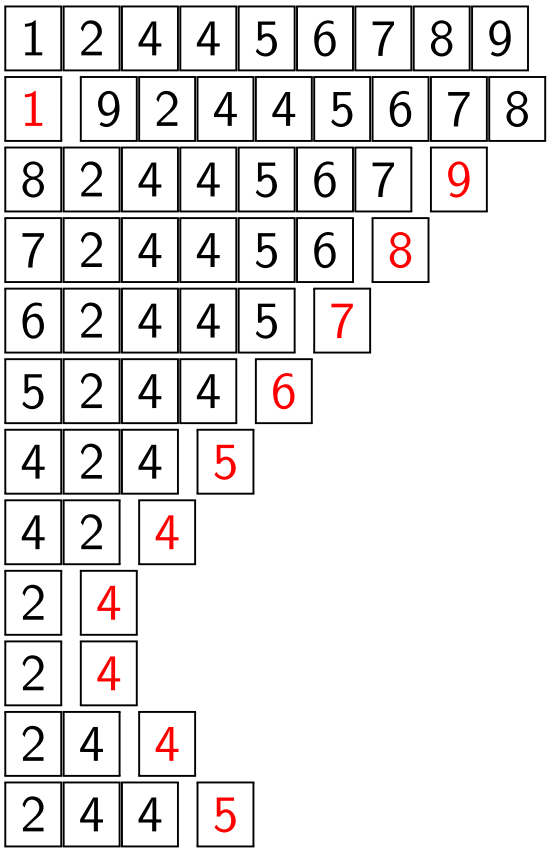
Simulação: pivot – o primeiro elemento – degeneração



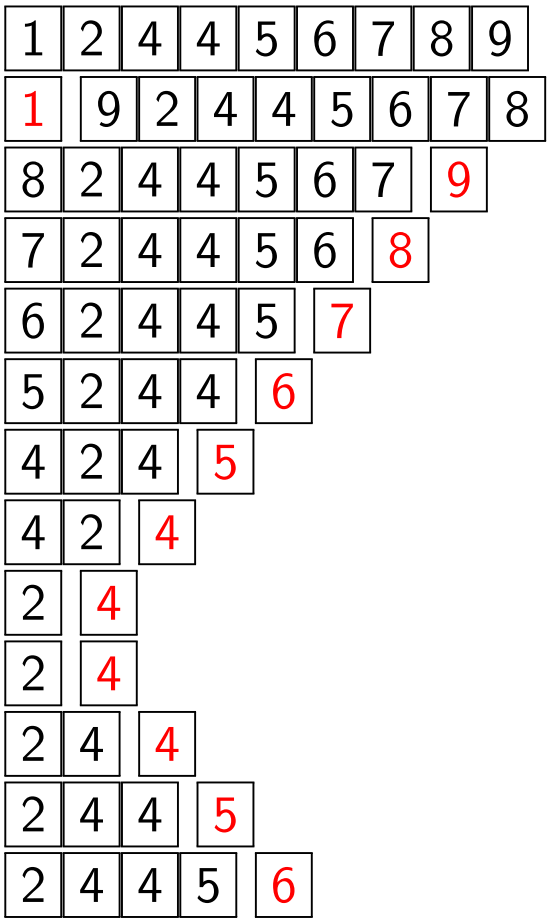
Simulação: pivot – o primeiro elemento – degeneração



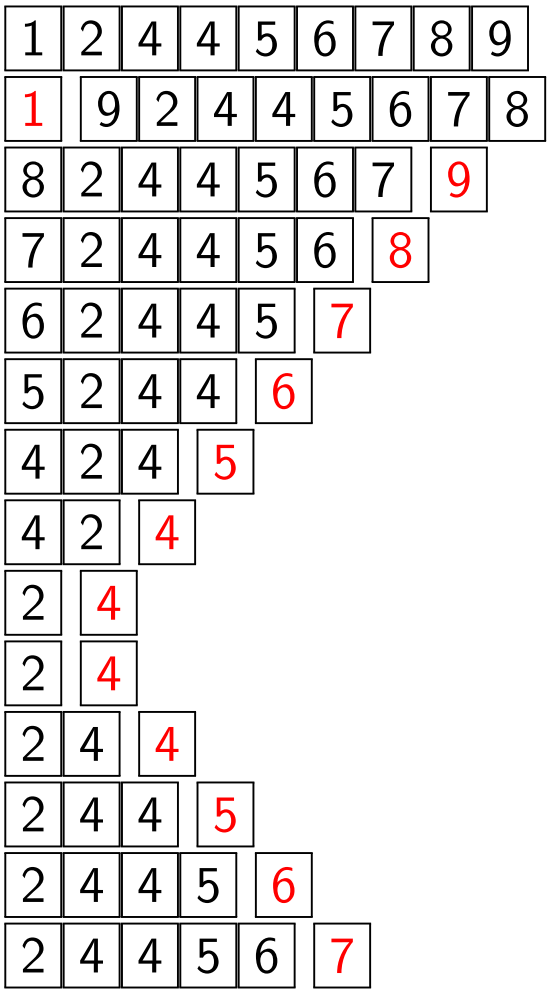
Simulação: pivot – o primeiro elemento – degeneração



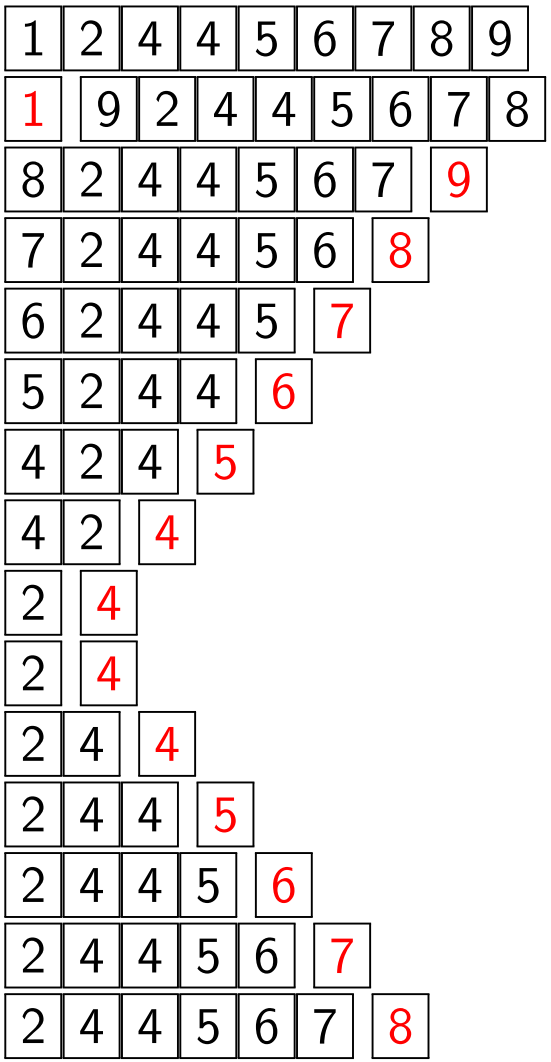
Simulação: pivot – o primeiro elemento – degeneração



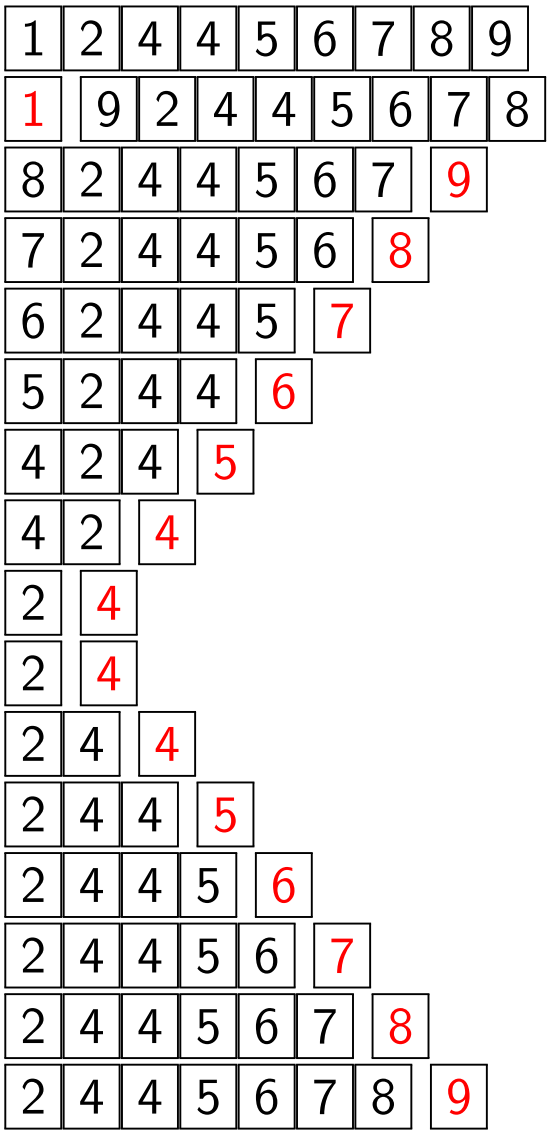
Simulação: pivot – o primeiro elemento – degeneração



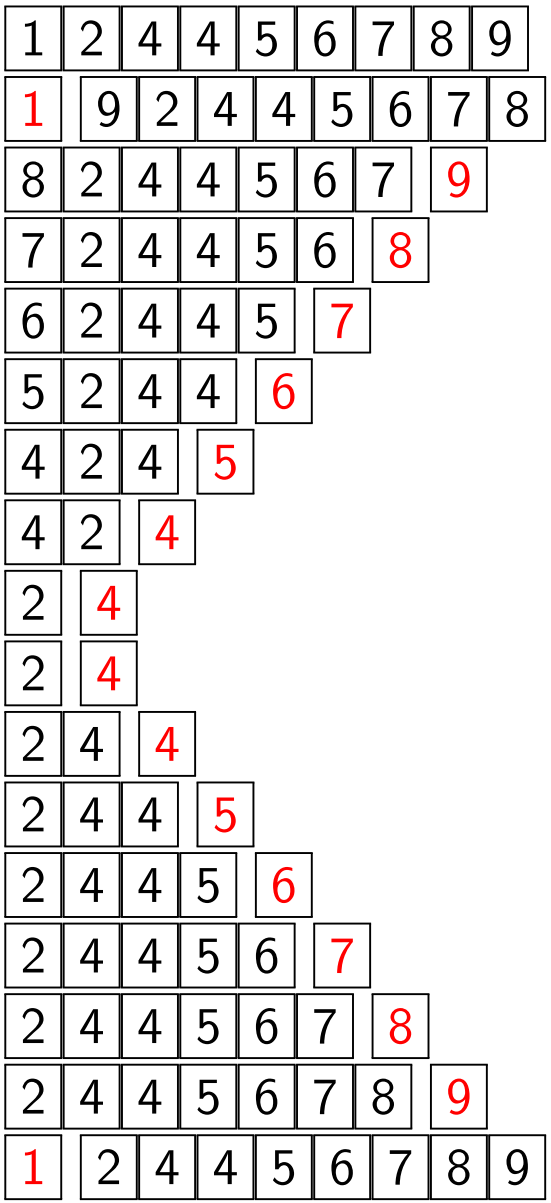
Simulação: pivot – o primeiro elemento – degeneração



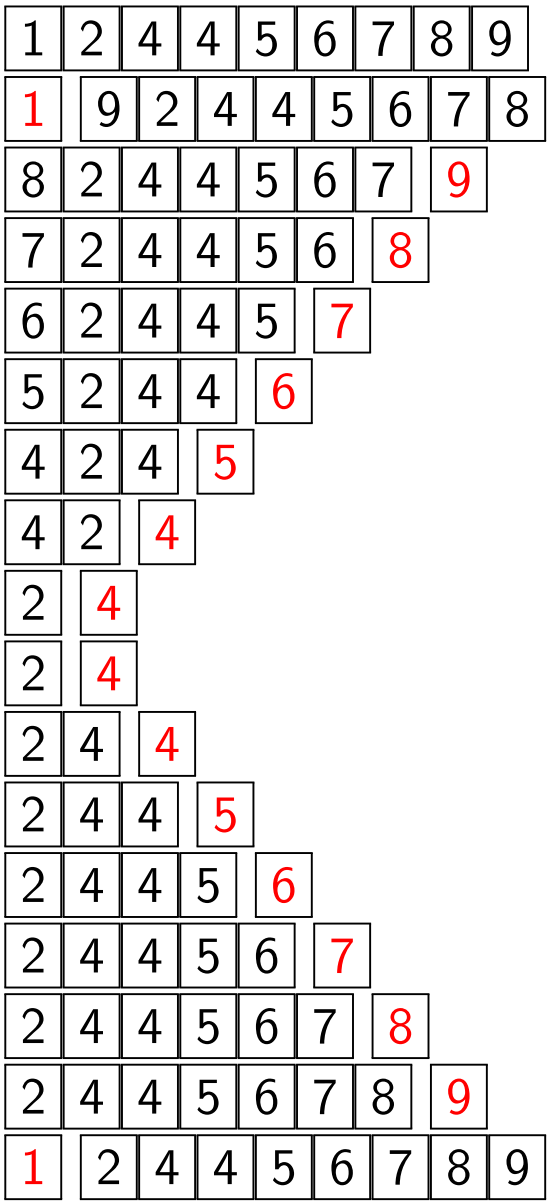
Simulação: pivot – o primeiro elemento – degeneração



Simulação: pivot – o primeiro elemento – degeneração



Simulação: pivot – o primeiro elemento – degeneração



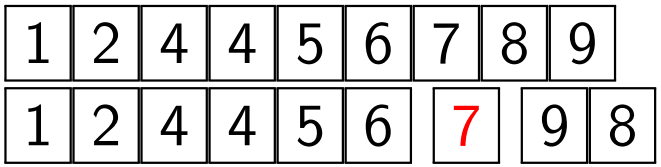
Pivot aleatório

```
1 public static int escolherPivot(int[] vetor) {
2
3     Random sorteio = new Random();
4
5     return(sorteio.nextInt(vetor.length));
6 }
```

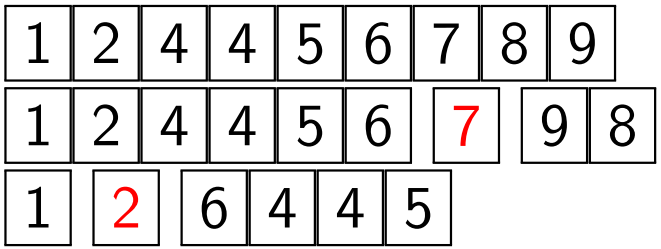
Simulação: pivot aleatório evita degeneração

1	2	4	4	5	6	7	8	9

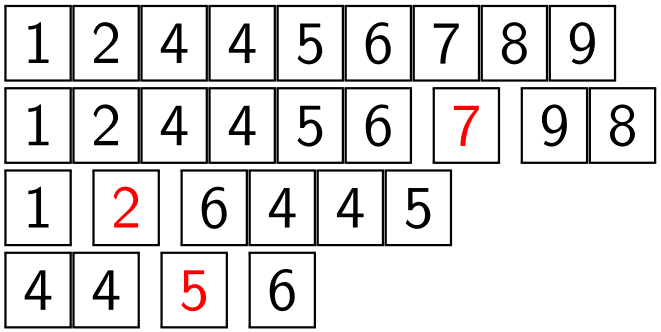
Simulação: pivot aleatório evita degeneração



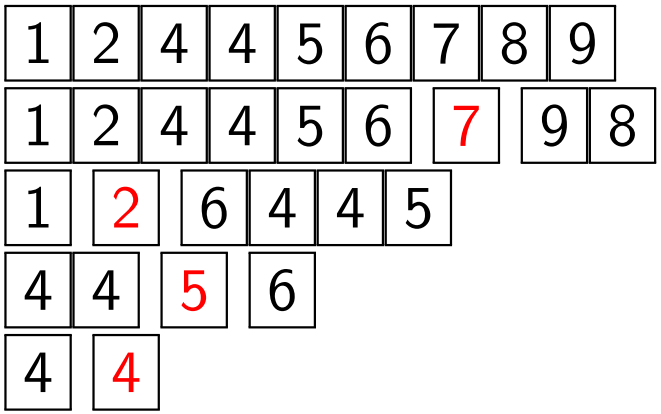
Simulação: pivot aleatório evita degeneração



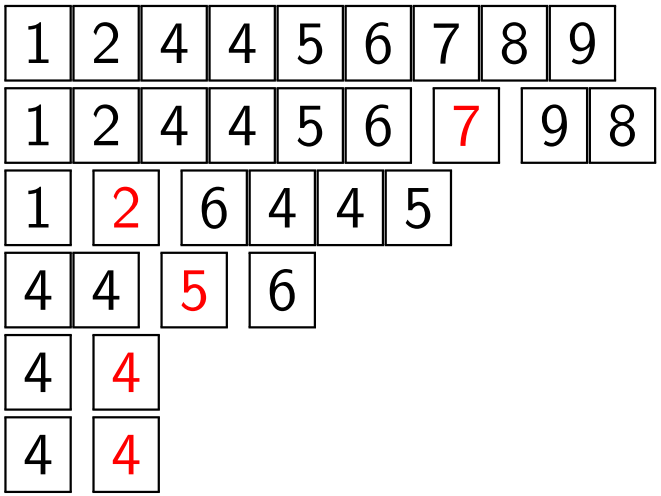
Simulação: pivot aleatório evita degeneração



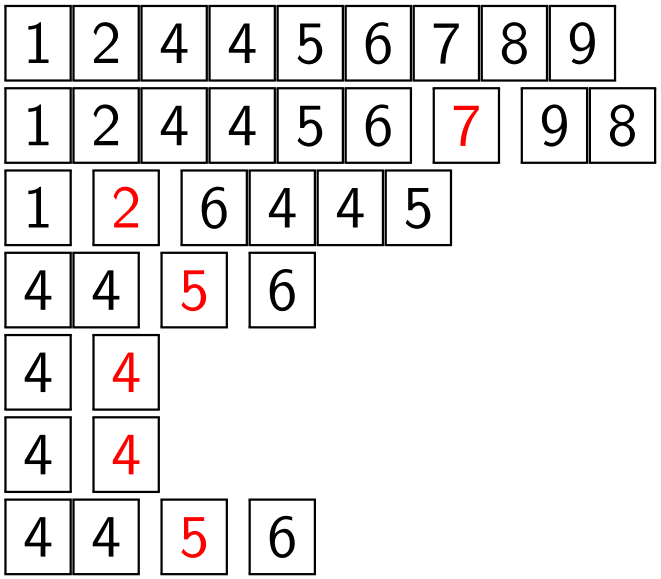
Simulação: pivot aleatório evita degeneração



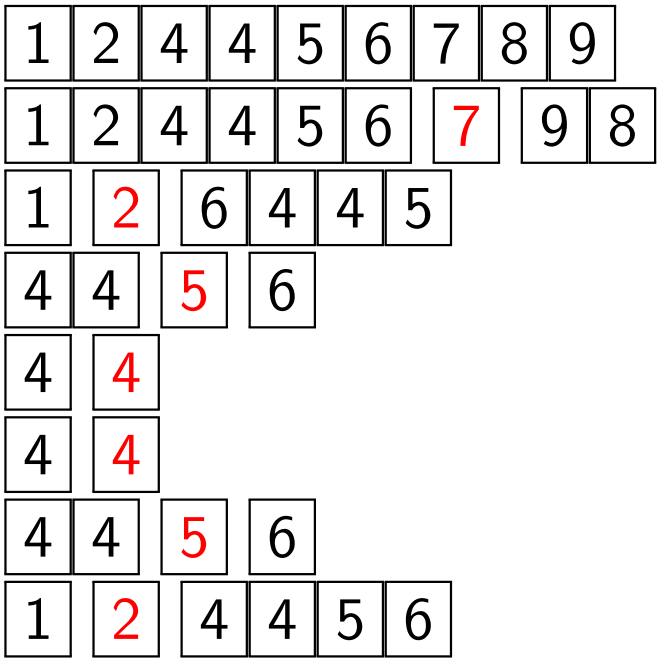
Simulação: pivot aleatório evita degeneração



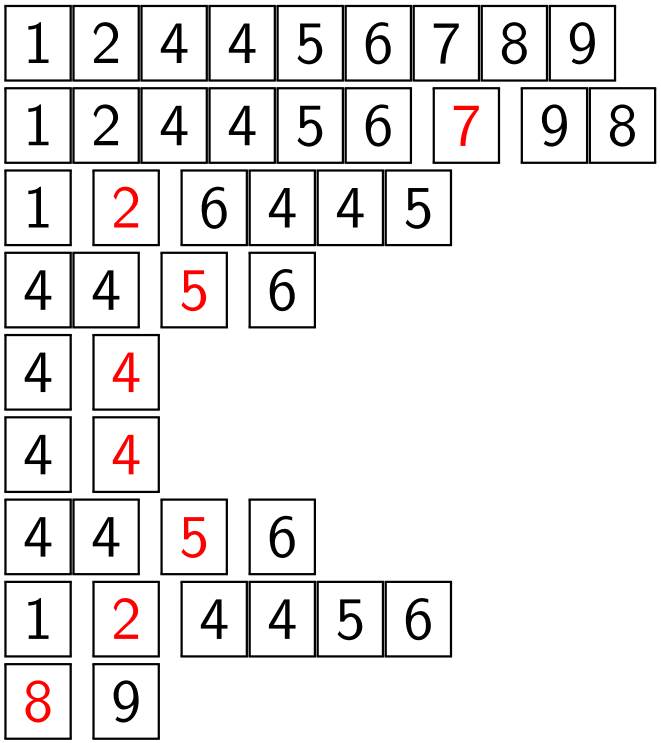
Simulação: pivot aleatório evita degeneração



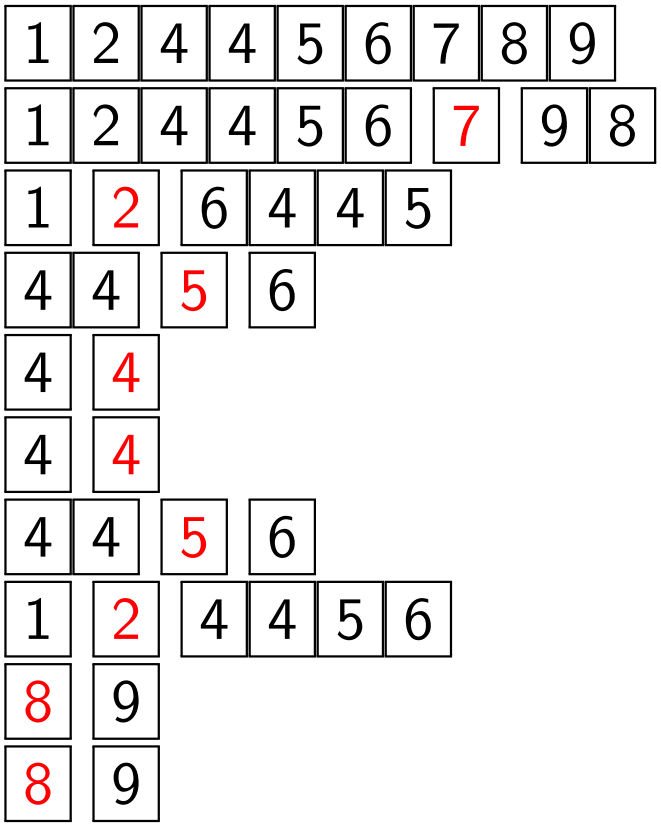
Simulação: pivot aleatório evita degeneração



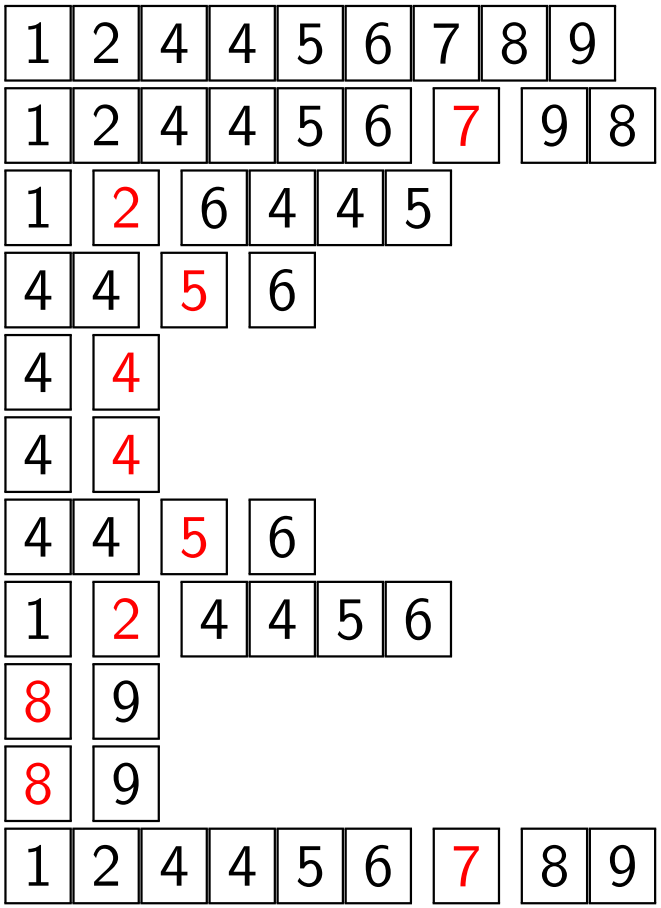
Simulação: pivot aleatório evita degeneração



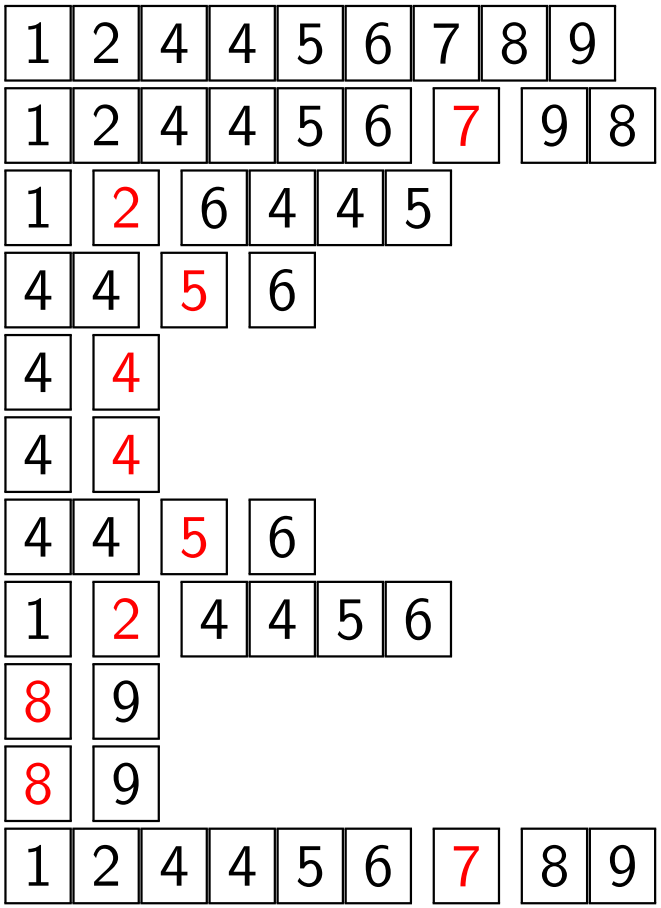
Simulação: pivot aleatório evita degeneração



Simulação: pivot aleatório evita degeneração



Simulação: pivot aleatório evita degeneração



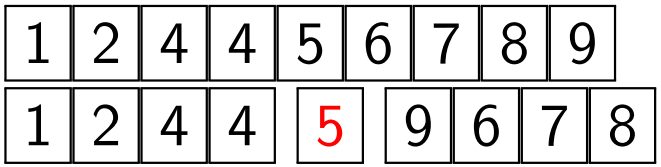
Pivot mediana

```
1  public static int escolherPivotMediana(int[] vetor) {
2
3   int fim = vetor.length-1;
4   int meio = (int) vetor.length/2;
5   int comeco = 0;
6
7   if ( vetor[fim] > vetor[meio] ) {
8     if (vetor[meio] > vetor[comeco]) {
9         return(meio);
10    } else {
11        return(comeco);
12    }
13  } else {
14    if (vetor[fim] > vetor[comeco]) {
15      return(fim);
16    } else {
17      return(comeco);
18    }
19  }
20 }
```

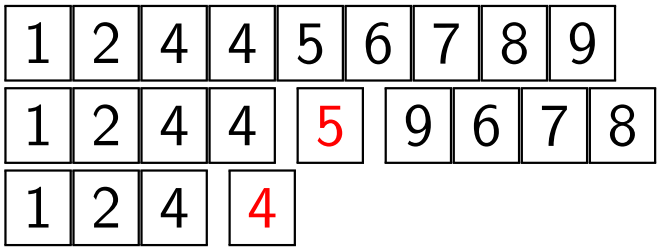
Simulação: pivot mediana

1	2	4	4	5	6	7	8	9

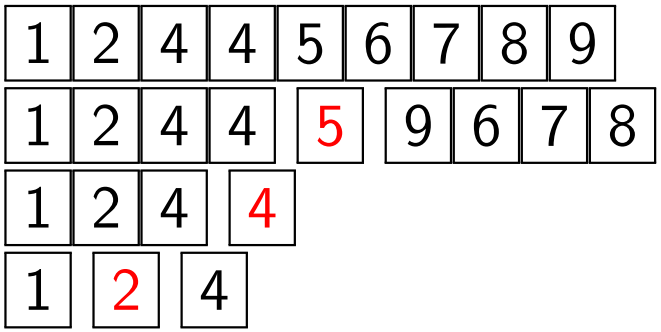
Simulação: pivot mediana



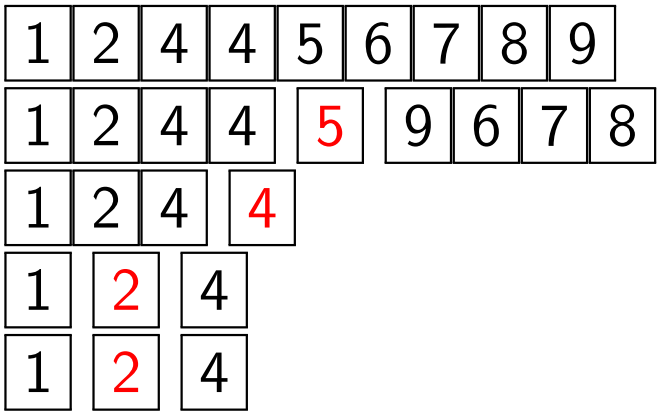
Simulação: pivot mediana



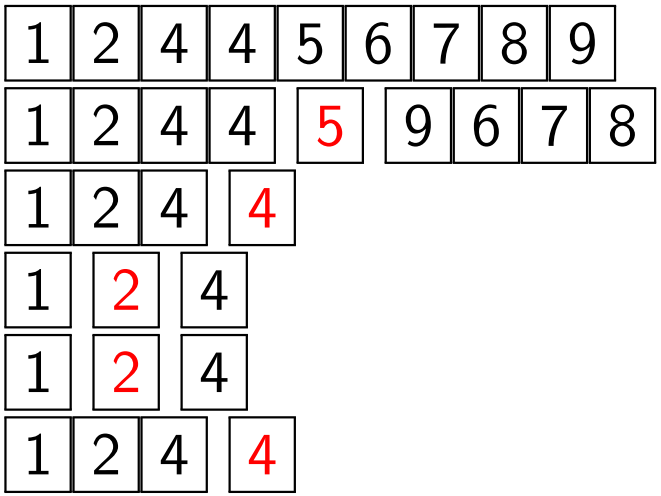
Simulação: pivot mediana



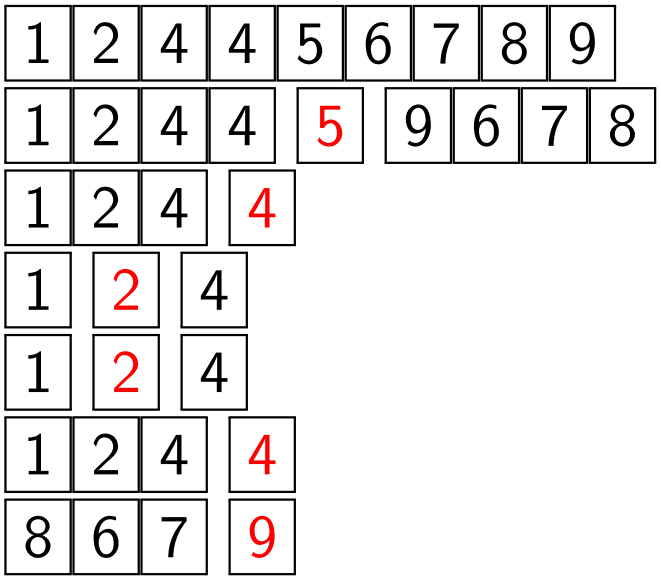
Simulação: pivot mediana



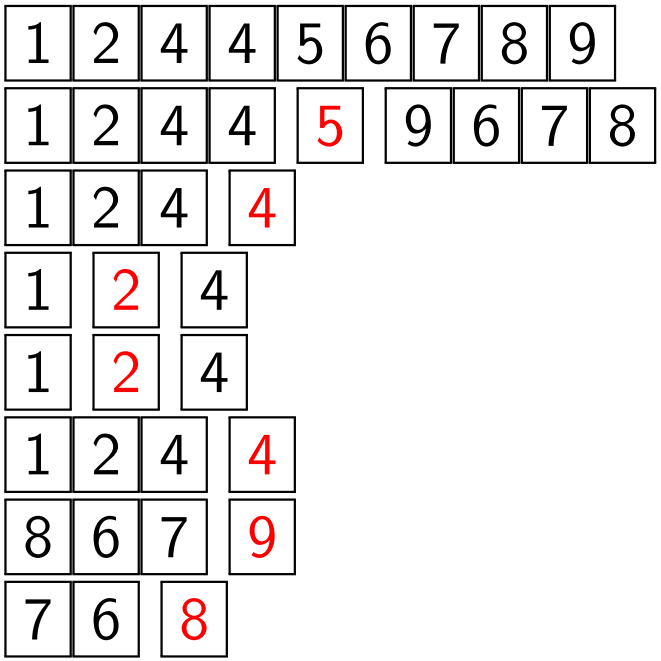
Simulação: pivot mediana



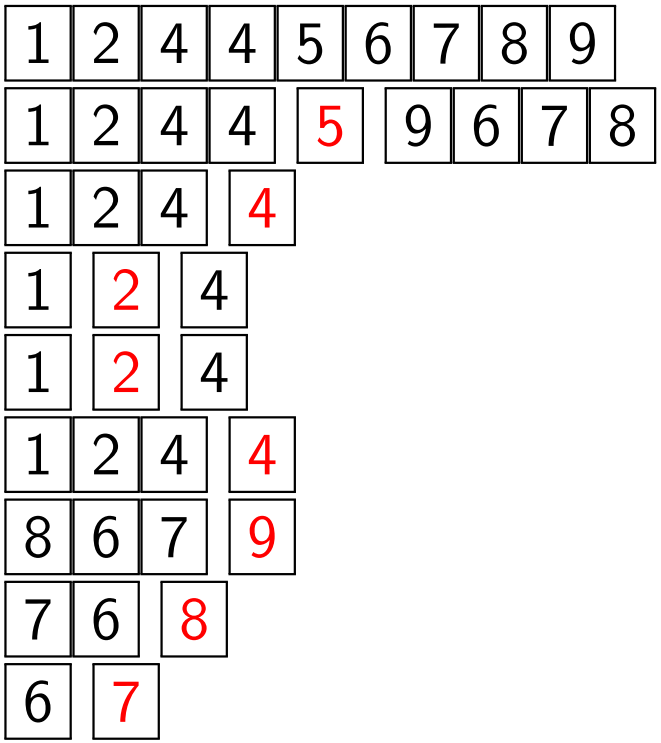
Simulação: pivot mediana



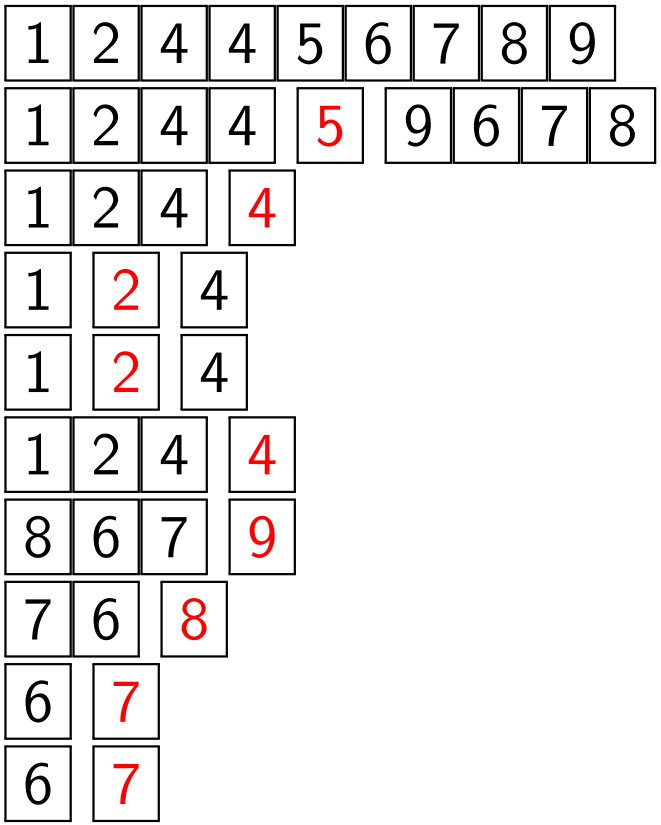
Simulação: pivot mediana



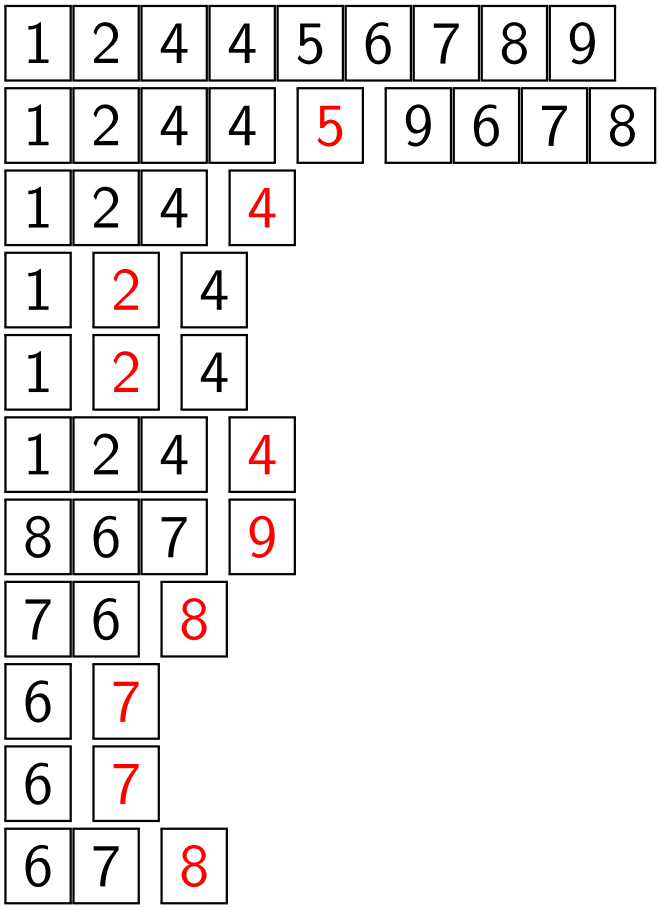
Simulação: pivot mediana



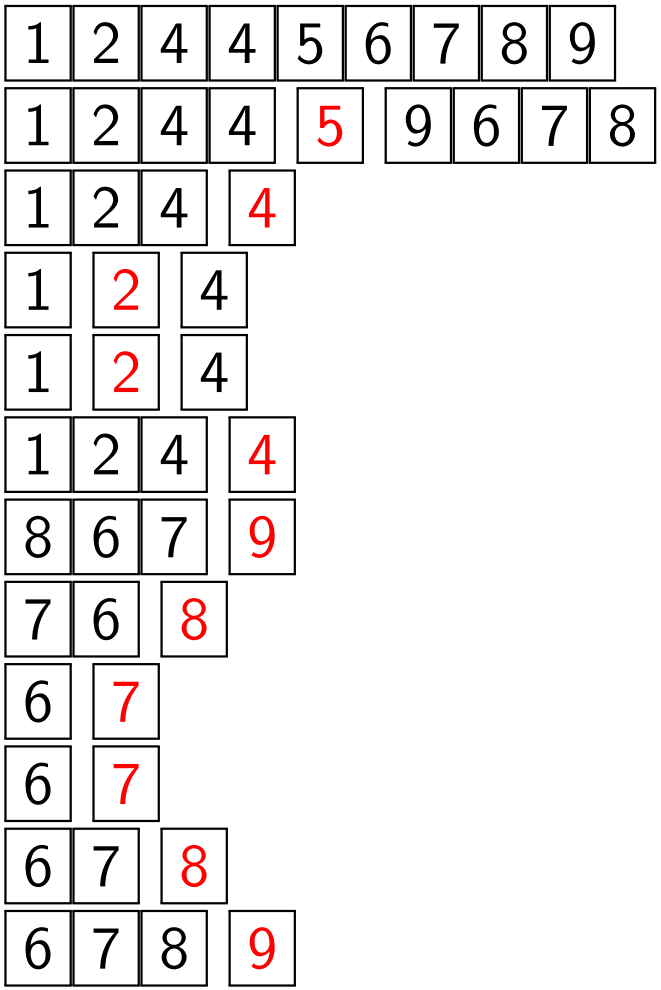
Simulação: pivot mediana



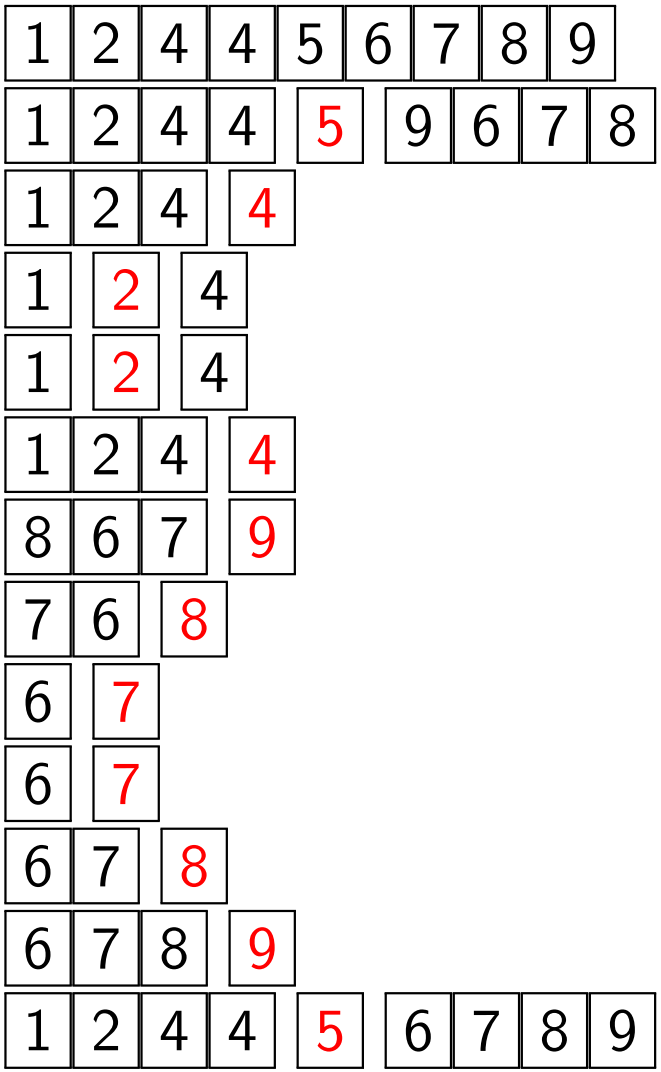
Simulação: pivot mediana



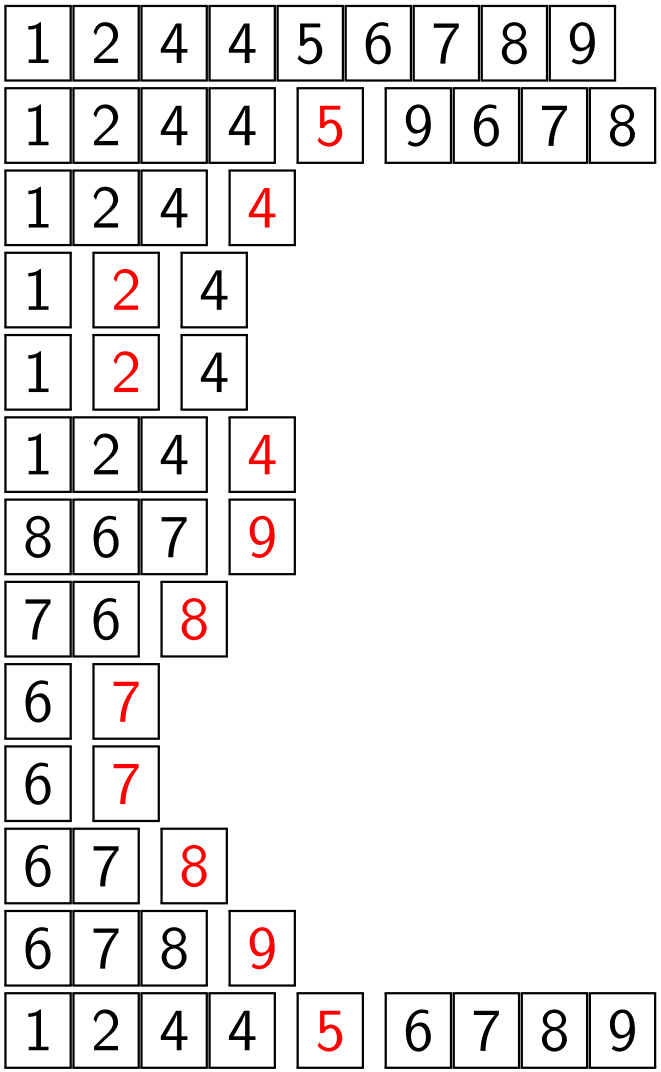
Simulação: pivot mediana



Simulação: pivot mediana



Simulação: pivot mediana



Pivot mediana: propriedades

- “Implementing Quicksort programs”, Robert Sedgewick

Pivot mediana: propriedades

- “Implementing Quicksort programs”, Robert Sedgewick
- Estudos práticos sobre o Quicksort

Pivot mediana: propriedades

- “Implementing Quicksort programs”, Robert Sedgewick
- Estudos práticos sobre o Quicksort
- Recomenda o uso de pivot mediana

Pivot mediana: propriedades

- “Implementing Quicksort programs”, Robert Sedgewick
- Estudos práticos sobre o Quicksort
- Recomenda o uso de pivot mediana
- Estatisticamente, o custo do Quicksort tende a ser menor para vetores maiores

Quicksort: versões

- Existem muitas versões do Quicksort
- Em geral eles atuam no vetor original para fazer as partições
  - economia de uso de espaço
- Existem muitas otimizações
  - "A origem de todo o mal é a otimização precoce", Donald Knuth

Custo: Quicksort simplificado

Custo e complexidade

- Melhor caso, notação Ω
  - altura da árvore de recursão
- Pior Caso, notação O
  - degeneração