Pontif´ıcia Universidade Católica de Minas Gerais
Curso de Ciência da Computação
Disciplina: Algoritmos e Estruturas de Dados II

Laboratório 06 - Quicksort e seu pivô

Lucas Carneiro Nassau Malta

Relatório Sobre a Implementação do Quicksort com Variação na Escolha do Pivô

Introdução

Contexto: Ordenar um vetor de inteiros é um problema clássico e comum na computação, tendo em vista sua grande aplicação em diversos contextos nessa área do conhecimento. Um dos algoritmos mais conhecidos para resolver esse problema é o QuickSort, proposto em, 1961 pelo cientista da computação britânico Charles Antony Richard Hoare. Essa solução é especialmente famosa por sua alta eficiência, sendo considerado o algoritmo ótimo para a classe de problemas de ordenação.

Objetivo: O objetivo deste relatório consiste em analisar a influência da estratégia de escolha do pivô no desempenho geral do QuickSort. Pretende-se implementar esse algoritmo de 4 formas diferentes e comparar os resultados obtidos por cada uma delas.

Metodologia: O QuickSort será implementado com quatro estratégias de escolha do pivô: início do subvetor, fim do subvetor, posição aleatória e mediana de três elementos (início, meio e fim). Cada versão foi testada com arranjos ordenados, quase ordenados e aleatórios, de tamanhos 100, 1000 e 10000 elementos. Para aumentar a precisão da análise, cada combinação de tipo de pivô, tipo de arranjo e quantidade de elementos foi testada 30 vezes.

Especificação da metodologia

Estratégias de escolha de pivô: As estratégias de escolha de pivô para o Quicksort foram implementadas na linguagem Java.

Pivô no início: Aqui, o pivô é o primeiro elemento do subvetor, representado pela variável “esq”:

```
public static void quicksortFirstPivot (int array [], int esq, int
dir)
  {
    int i = esq, j = dir;
    int pivo = array[esq];
    while (i <= j)
    {
      while (array[i] < pivo) {  i++; comparacoes++;  }
      while (array[j] > pivo) {  j--; comparacoes++;  }
      if (i <= j)
      {
        swap(array, i, j);
        movimentacoes++;
        i++;
        j--;
      }
    }
      if (esq < j)  quicksortFirstPivot (array, esq, j);
      if (i < dir)  quicksortFirstPivot (array, i, dir);
  }
```

Pivô no fim: Nesta variação, o pivô é o último elemento, representado por “dir”:

```
  public static void quicksortLastPivot (int array [], int esq, int
dir)
  {
    int i = esq, j = dir;
    int pivo = array[dir];
    while (i <= j)
    {
      while (array[i] < pivo) {  i++; comparacoes++;  }
      while (array[j] > pivo) {  j--; comparacoes++;  }
      if (i <= j)
      {
        swap(array, i, j);
        movimentacoes++;
        i++;
        j--;
      }
    }
      if (esq < j)  quicksortLastPivot (array, esq, j);
      if (i < dir)  quicksortLastPivot (array, i, dir);
  }
```

Pivô aleatório: Aqui, um pivô é escolhido aleatoriamente dentro do subvetor:

```
  public static Random random = new Random();

  public static int gerarAleatorio ()
  {
    return Math.abs(random.nextInt());
  }
```

```
  public static void quicksortRandomPivot (int array [], int esq,
int dir)
  {
    int i = esq, j = dir;
    int pivo = array[esq + gerarAleatorio() % (dir - esq + 1)];
    while (i <= j)
    {
      while (array[i] < pivo) {  i++; comparacoes++;  }
      while (array[j] > pivo) {  j--; comparacoes++;  }
      if (i <= j)
      {
        swap(array, i, j);
        movimentacoes++;
        i++;
        j--;
      }
    }
      if (esq < j)  quicksortRandomPivot (array, esq, j);
      if (i < dir)  quicksortRandomPivot (array, i, dir);
  }
```

Mediana de três: A última implementação escolhe o pivô como a mediana de três elementos (início, meio e fim):

```
  public static int median (int array[], int esq, int dir)
  {
  //Definir dados locais
    int mid;
    int median = 0;
  //Testar tamanho
    if (array.length == 1)
      mid = array[0];
    else if (array.length == 2)
      mid = (array[0] + array[1]) / 2;
    else
    {
    //Definir dados
      mid = esq + (dir - esq) / 2;
      int a = array[esq], b = array[mid], c = array[dir];
    //Calcular a posicao do meio entre tres numeros
      if ( (b < a && a < c) || (c < a && a < b) )
        median = esq;
```

```
      else if ( (a < b && b < c) || (c < b && b < a) )
        median = mid;
      else
        median = dir;
    }
  //Retornar
    return median;
  }
```

```
public static void quicksortMedianOfThree (int array [], int esq,
int dir)
  {
    int i = esq, j = dir;
    int pivo = array[ median(array, esq, dir) ];
    while (i <= j)
    {
      while (array[i] < pivo) {  i++; comparacoes++;  }
      while (array[j] > pivo) {  j--; comparacoes++;  }
      if (i <= j)
      {
        swap(array, i, j);
        movimentacoes++;
        i++;
        j--;
      }
    }
      if (esq < j)  quicksortMedianOfThree (array, esq, j);
      if (i < dir)  quicksortMedianOfThree (array, i, dir);
  }
```

Estratégias de construção dos arranjos: As estratégias de construção de arranjos foram implementadas também na linguagem Java.

Arranjo em ordem crescente: O arranjo foi preenchido com valores de 1 até ‘n’, sendo ‘n’ a quantidade de elementos:

```
  public static int [] arrayCrescente (int n)
  {
  //Definir dados locais
    int array [] = new int[n];
  //Preencher array
    for (int i = 0; i < n; i++)
      array[i] = i + 1;
  //Retornar
    return array;
  }
```

Arranjo em ordem decresente: O arranjo foi preenchido com valores de 'n' até 1, sendo 'n' a quantidade de elementos:

```
public static int [] arrayDecrescente (int n)
{
//Definir dados locais
  int array [] = new int[n];
//Preencher array
  for (int i = 0; i < n; i++)
    array[i] = n - i;
//Retornar
  return array;
}
```

Arranjo quase ordenado: O arranjo foi primeiramente gerado em ordem crescente e posteriormente desordenado de acordo com a variável 'heap', que indica a distância entre posições que terão seus elementos trocados. Para todos os testes realizados, a variável 'heap' foi inicializada com seu valor igual a 4:

```
public static int [] arrayQuaseOrdenado (int n, int heap)
{
//Definir dados locais
  int array [] = arrayCrescente(n);
//Bagunçar arranjo
  for (int i = 0; i < heap; i += 2)
    swap ( array, i * heap, (i + 1) * heap );
//Retornar
  return array;
}
```

Arranjo aleatório: O arranjo foi populado com valores aleatórios limitados de 0 até 'n – 1', sendo 'n' a quantidade total de elementos:

```
public static int [] arrayRandom (int n)
{
//Definir dados locais
  int array [] = new int[n];
//Preencher arranjo com valores aleatorios
  for (int i = 0; i < n; i++)
    array[i] = gerarAleatorio() % n;
//Retornar
  return array;
}
```

Pontif´ıcia Universidade Católica de Minas Gerais
Curso de Ciência da Computação
Disciplina: Algoritmos e Estruturas de Dados II

Especificação da coleta dos dados

Obtenção dos dados: Cada variação do QuickSort foi modificada para contar a quantidade de comparações e movimentações realizadas. Com essa alteração, cada estratégia foi testada para arranjos em ordem crescente, decrescente, quase ordenados e aleatórios. Além disso, todas essas combinaçõs foram testadas para 100, 1000 e 10000 elementos. O resultado da contagem de cada execução foi salvo arquivos CSV para posterior tratamento e visualização.

Tratamento dos dados: Os dados obtidos foram subdivididos em arquivos individuais com as colunas “comparações” e “movimentações”, e com 30 linhas cada, representando o resultado de todos os testes.

Visualização dos dados: Todos esses arquivos CSV gerados foram convertidos em gráficos por meio de um script em Python com as bibliotecas “panda” e “matplotlib”. Segue o código implementado:

```
import pandas as pd
import glob
import os
import seaborn as sns
import matplotlib.pyplot as plt

# Diretório principal onde estão as pastas
main_dir = 'path/to/your/directory'  # Substitua pelo caminho para
o diretório principal

# Inicializar uma lista para armazenar os DataFrames
dataframes = []

# Carregar todos os arquivos CSV
for pivot in ['FirstPivot', 'LastPivot', 'MedianPivot',
'RandomPivot']:
    for order in ['Crescente', 'Decrescente', 'Quase', 'Random']:
        # Caminho para os arquivos CSV
        folder_path = os.path.join(main_dir, pivot, order)
        csv_files = glob.glob(os.path.join(folder_path, '*.csv'))

        for csv_file in csv_files:
            df = pd.read_csv(csv_file)
            # Adicionar informações sobre o tipo de pivô e tipo de
arranjo
            df['tipo_de_pivo'] = pivot
            df['tipo_de_arranjo'] = order
            dataframes.append(df)
```

```
# Concatenar todos os DataFrames em um único DataFrame
full_data = pd.concat(dataframes, ignore_index=True)

# Gráficos de linhas para cada combinação de pivô e tamanho
tamanhos = [100, 1000, 10000]

for tamanho in tamanhos:
    for pivot in ['FirstPivot', 'LastPivot', 'MedianPivot',
'RandomPivot']:
        df_tamanho = full_data[(full_data['tamanho'] == tamanho) &
(full_data['tipo_de_pivo'] == pivot)]
        plt.figure(figsize=(12, 6))
        sns.lineplot(data=df_tamanho, x='tipo_de_arranjo',
y='comparacoes', marker='o', label='Comparações')
        sns.lineplot(data=df_tamanho, x='tipo_de_arranjo',
y='movimentacoes', marker='o', label='Movimentações')
        plt.title(f'Resultados do Quicksort com {pivot} para
{tamanho} elementos')
        plt.ylabel('Número')
        plt.xlabel('Tipo de Arranjo')
        plt.legend()
        plt.grid()
        plt.savefig(f'resultados_{pivot}_{tamanho}_elementos.png')
        plt.show()

# Gráfico de colunas para as médias
summary = full_data.groupby(['tamanho',
'tipo_de_pivo']).agg({'comparacoes': 'mean', 'movimentacoes':
'mean'}).reset_index()

plt.figure(figsize=(12, 6))
summary_melted = summary.melt(id_vars=['tamanho', 'tipo_de_pivo'],
value_vars=['comparacoes', 'movimentacoes'], var_name='Tipo',
value_name='Valor')

sns.barplot(data=summary_melted, x='tamanho', y='Valor',
hue='Tipo', ci=None)
plt.title('Média de Comparações e Movimentações por Tamanho')
plt.ylabel('Média')
plt.xlabel('Tamanho do Arranjo')
plt.legend(title='Tipo')
plt.savefig('media_comparacoes_movimentacoes.png')
plt.show()
```

Resultados obtidos

Métrica: Cada combinação de tipo de pivô e tipo de arranjo foi testada 30 vezes. A métrica utilizada será a média desses testes.

Tipos de pivô: Cada tipo de pivô foi testado e os resultados foram apresentados em tabelas:

Pivô no início:

Arranjo crescente:

Elementos	Comparações	Movimentações
100	4.950	99
1.000	495.000	999
10.000	49.500.000	9.999

Arranjo decrescente:

Elementos	Comparações	Movimentações
100	4.050	99
1.000	490.000	999
10.000	49.000.000	9.999

Arranjo quase ordenado:

Elementos	Comparações	Movimentações
100	4.402	99
1.000	493.552	999
10.000	49.935.052	9.999

Arranjo aleatório:

Elementos	Comparações	Movimentações
100	488	192
1.000	8.705	2.673
10.000	123.963	34.365

Pivô no final:

Arranjo crescente:

Elementos	Comparações	Movimentações
100	4.950	99
1.000	499.500	999
10.000	49.995.000	9.999

Arranjo decrescente:

Elementos	Comparações	Movimentações
100	4.900	99
1.000	499.000	999
10.000	49.990.000	9.999

Arranjo quase ordenado:

Elementos	Comparações	Movimentações
100	4.924	99
1.000	499.474	999
10.000	49.994.974	9.999

Arranjo aleatório:

Elementos	Comparações	Movimentações
100	486	192
1.000	8.727	2.678
10.000	124.150	34.431

Pontif´ıcia Universidade Católica de Minas Gerais
Curso de Ciência da Computação
Disciplina: Algoritmos e Estruturas de Dados II

Pivô mediana de três:

Arranjo crescente:

Elementos	Comparações	Movimentações
100	480	63
1.000	7.987	551
10.000	113.631	5.904

Arranjo decrescente:

Elementos	Comparações	Movimentações
100	386	112
1.000	6.996	1.010
10.000	103.644	10.904

Arranjo quase ordenado:

Elementos	Comparações	Movimentações
100	476	65
1.000	7.986	513
10.000	113.632	5.906

Arranjo aleatório:

Elementos	Comparações	Movimentações
100	369	193
1.000	6.198	2.723
10.000	89.115	34.979

Pontif´ıcia Universidade Católica de Minas Gerais
Curso de Ciência da Computação
Disciplina: Algoritmos e Estruturas de Dados II

Pivô aleatório:

Arranjo crescente:

Elementos	Comparações	Movimentações
100	661	66
1.000	11.202	668
10.000	156.185	6.667

Arranjo decrescente:

Elementos	Comparações	Movimentações
100	575	113
1.000	10.064	1.162
10.000	144.926	11.663

Arranjo quase ordenado:

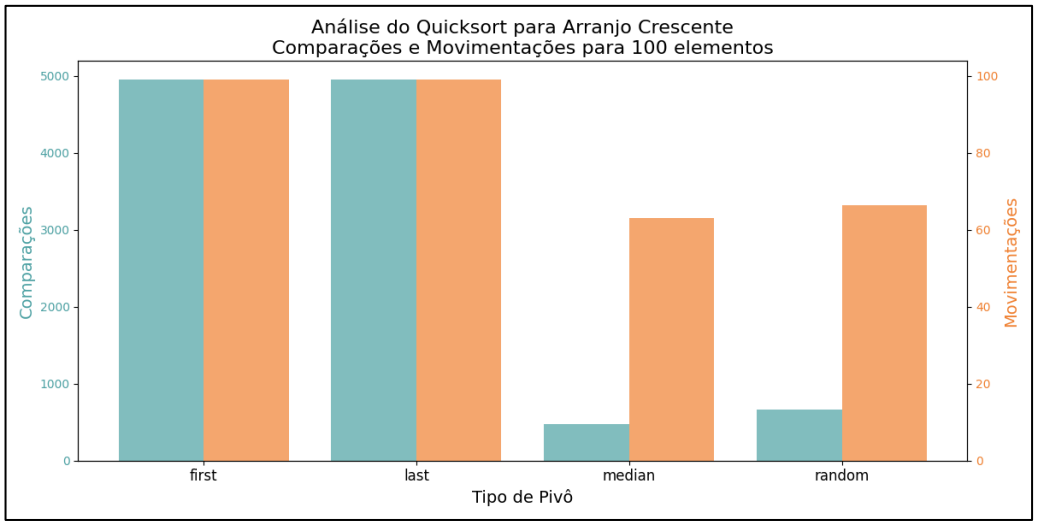
Elementos	Comparações	Movimentações
100	651	67
1.000	11.016	670
10.000	157.559	6.666

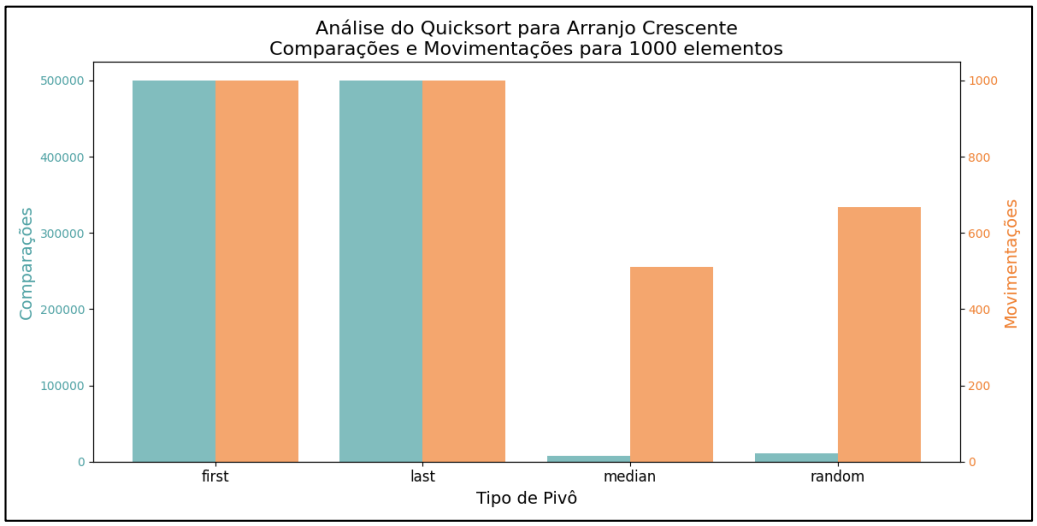
Arranjo aleatório:

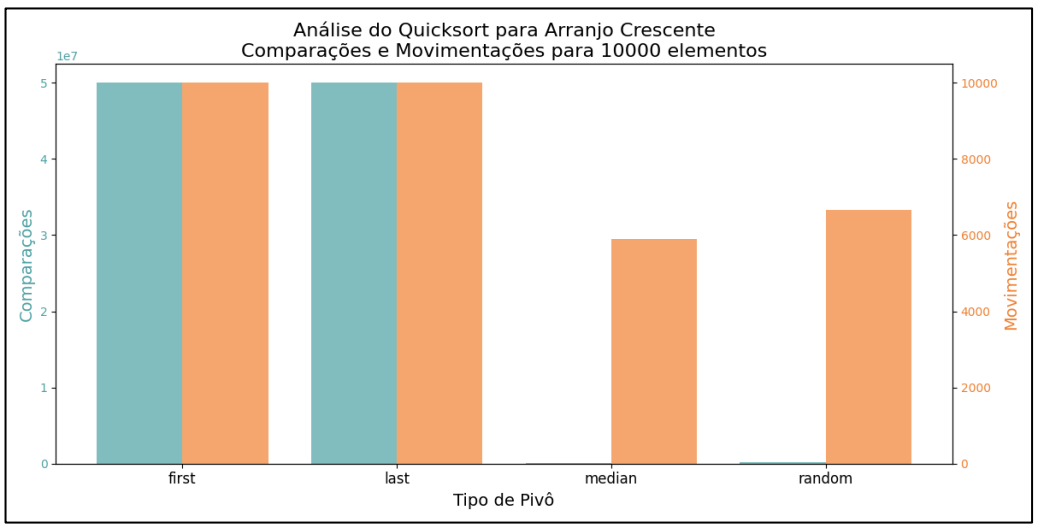
Elementos	Comparações	Movimentações
100	424	192
1.000	7.552	2.699
10.000	105.653	34.659

Tipos de arranjo: Cada tipo de arranjo foi testado com os quatro tipos de pivôs e os resultados das comparações e movimentações foram representados em gráficos:

Arranjo crescente:

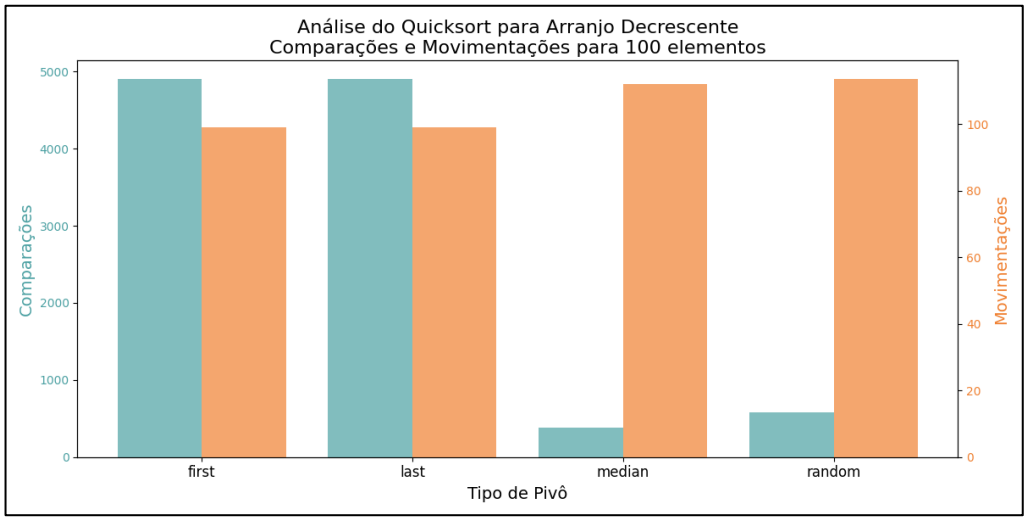


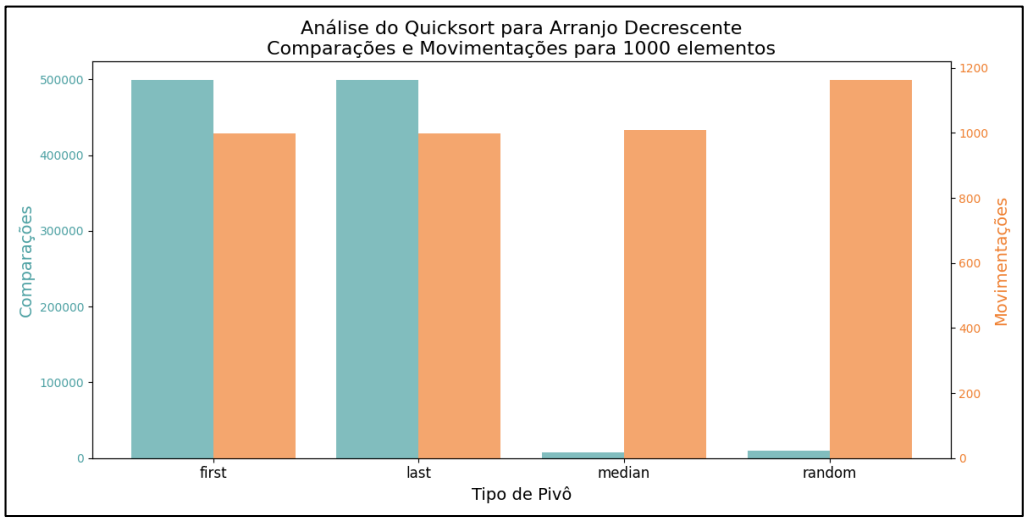


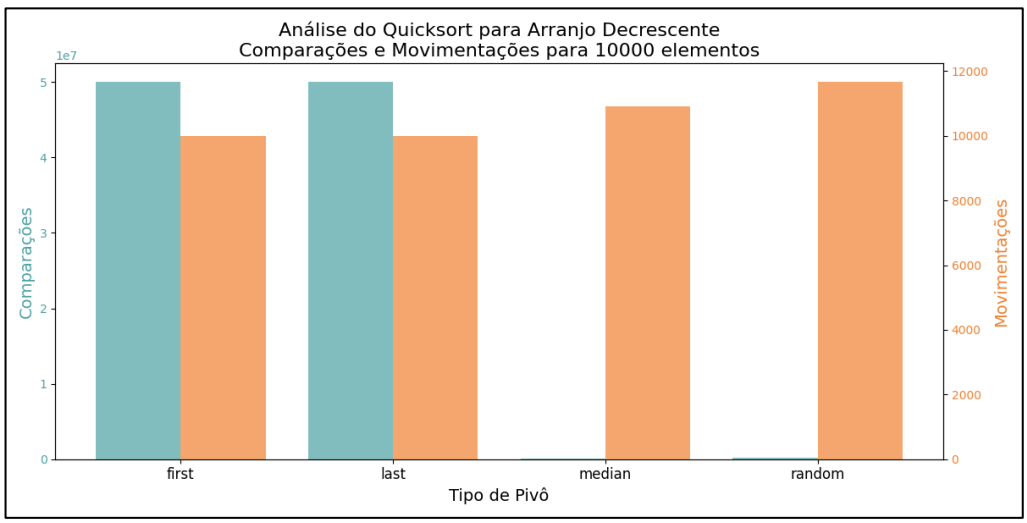


Pontif´ıcia Universidade Católica de Minas Gerais
Curso de Ciência da Computação
Disciplina: Algoritmos e Estruturas de Dados II

Arranjo decrescente:

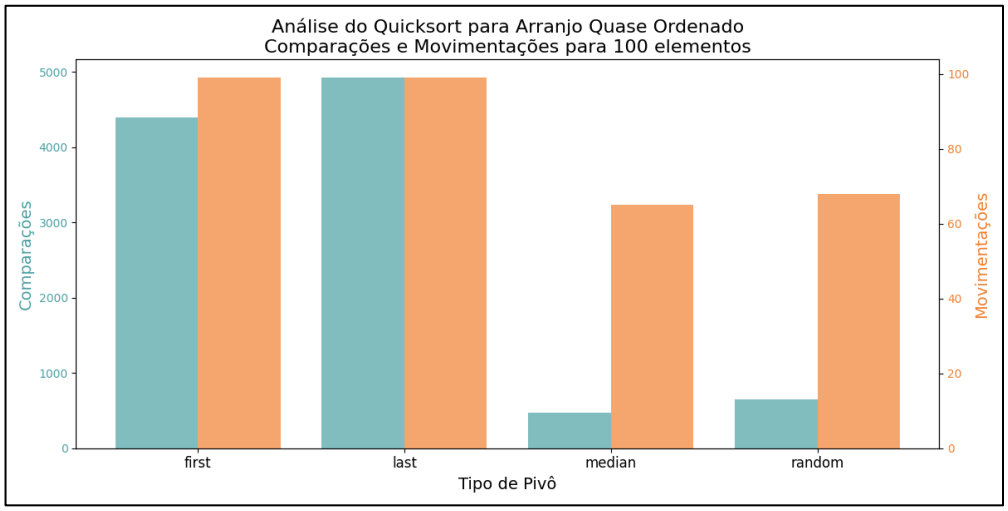


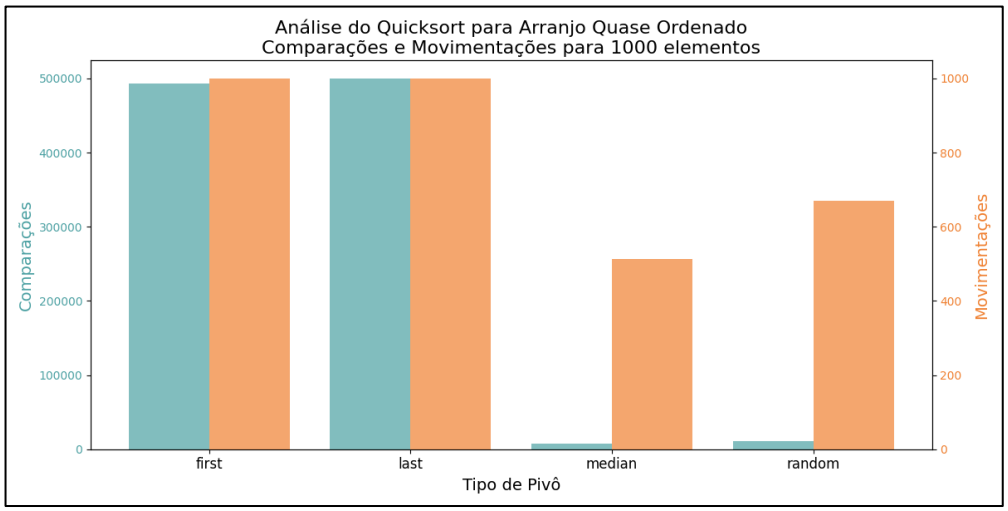


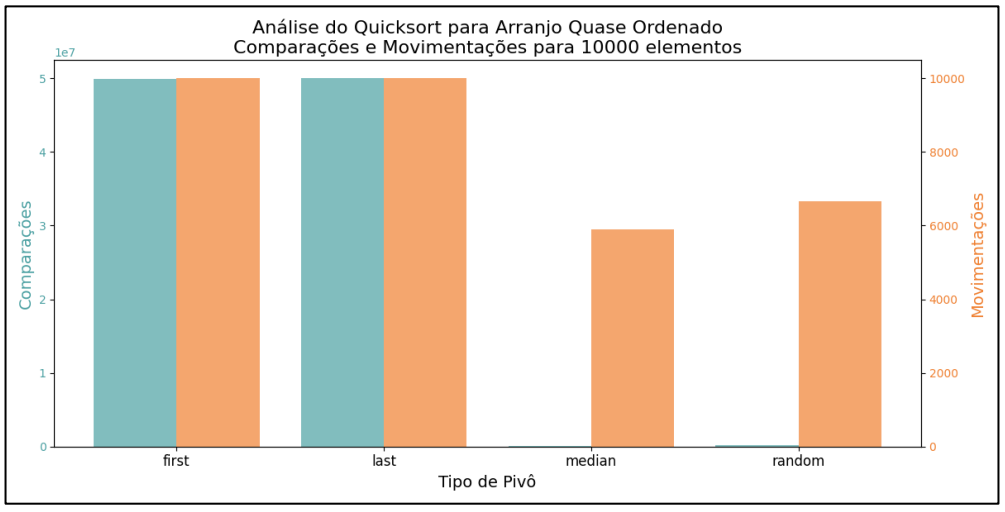


Pontif´ıcia Universidade Católica de Minas Gerais
Curso de Ciência da Computação
Disciplina: Algoritmos e Estruturas de Dados II

Arranjo quase ordenado:

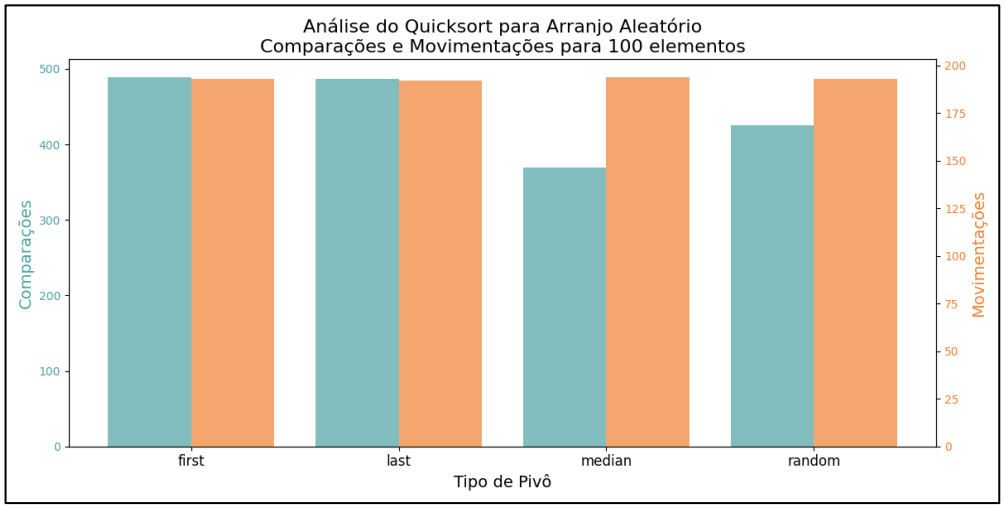


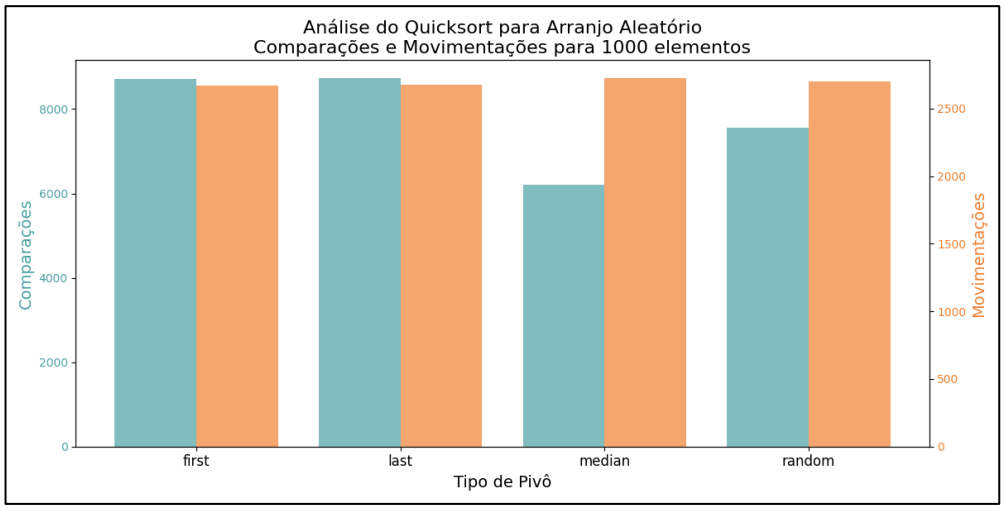


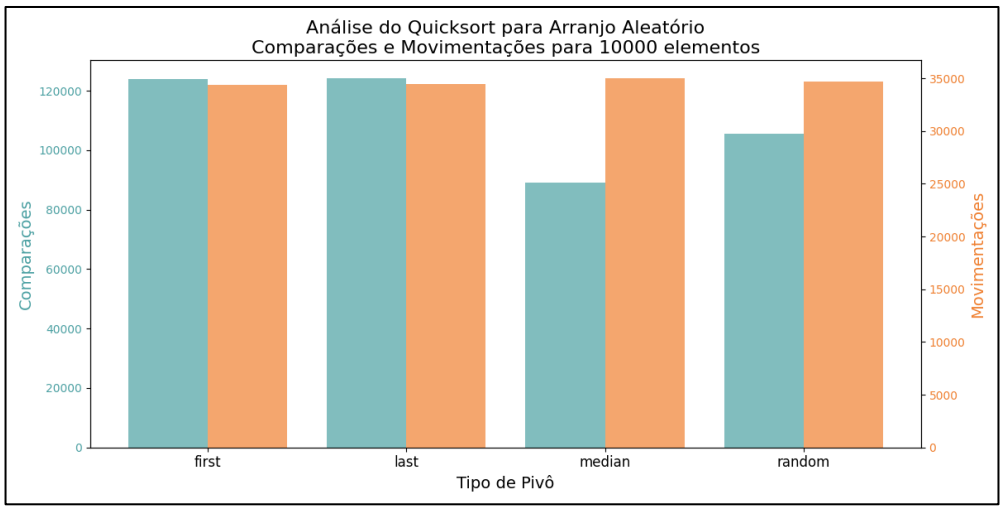


Pontif´ıcia Universidade Católica de Minas Gerais
Curso de Ciência da Computação
Disciplina: Algoritmos e Estruturas de Dados II

Arranjo aleatório:







Conclusão

Com base nos resultados obtidos, é possivel identificar em quais tipos de arranjo as diferentes estratégias de escolha de pivô do QuickSort apresentam melhor desempenho. Além disso, podemos indicar as melhores escolhas para cada tipo de arranjo.

Pivô no início: Os melhores desempenhos foram registrados nos arranjos decrescentes e aleatórios. Os arranjos aleatórios se destacaram pelo número de comparações signifivativamente menor em relação aos demais, apresentando aproximadamente 8,3 vezes menos para 100 elementos (488 contra 4.050), aproximadamente 56,2 vezes menos para 1.000 elementos (8.705 contra 490.000) e aproximadamente 395,3 menos para 10.000 elementos (123.963 contra 49.000.000).

Por outro lado, o arranjo decrescente teve um número menor de movimentações – cerca de 1,9 vezes menos para 100 elementos (99 contra 192), cerca de 2,7 vezes menos para 1.000 elementos (999 contra 2.673) e cerca de 3,4 vezes menos para 10.000 elementos (9.999 contra 34.365). É importante destacar que todos os tipos de arranjos, exceto o aleatório, apresentaram 9.999 movimentações; porém, o arranjo descrescente apresentou o menor número de comparações, sendo assim considerado o melhor caso dentre eles e, por isso, foi comparado com os arranjos aleatórios.

Dessa forma, o QuickSort com pivô no início desempenha melhor em arranjos aleatórios em termos de comparações e os outros arranjos empataram em termos de movimentações. O arranjo aleatório se destaca especialmente quando o número de registros é grande, pois apresenta uma redução relevante no número de comparações (395,3 vezes menos para 10.000 elementos), o que pode tornar o aumento nas movimentações irrelevante (3,4 vezes mais para 10.000 elementos), dependendo do contexto.

Pivô no final: As melhores performances foram semelhantes às do pivô no início: em arranjos decrescentes e aleatórios. O arranjo aleatório também se destacou pelo número menor de comparações, apresentando cerca de 10 vezes menos para 100 elementos (486 contra 4.900), cerca de 57,2 vezes menos para 1.000 elementos (8.727 contra 499.000) e cerca de 402 vezes menos para 10.000 elementos (124.150 contra 49.900.000).

Em contrapartida, o arranjo decrescente teve menos movimentações – aproximadamente duas vezes menos para 100 elementos (99 contra 192), aproximadamente 2,7 vezes menos para 1.000 elementos (999 contra 2.678) e aproximadamente 3,4 vezes menos para 10.000 elementos (9.999 contra 34.431). É relevante pontuar que, assim como para o pivô no início, todos os tipos de arranjo, exceto o aleatório, apresentaram 9.999 movimentações; entretanto, o arranjo decrescente performou o menor número de comparações, sendo então o melhor caso entre eles e, por isso, foi comparado com os arranjos aleatórios.

Desse modo, a escolha do pivô no final possui um melhor desempenho em arranjos aleatórios para comparações e os outros arranjos empataram para movimentações. O arranjo aleatório novamente se destaca pelo significativamente inferior número de comparações, principalmente para grandes conjuntos de elementos.

Pivô mediana de três: Os melhores desempenhos para essa estratégia foram mais equilibrados, diferentemente das anteriores.

Para 100 elementos, os arranjos aleatórios e crescentes performaram melhor para comparações e movimentações, respectivamente. Os aleatórios apresentaram 369 comparações contra 480, 386 e 476 dos demais, enquanto o crescente apresentou 63 movimentações contra 112, 65 e 193.

Para 1.000 elementos, os arranjos aleatórios ainda desempenharam melhor para comparações, mas os arranjos quase ordenado desempenharam melhor para movimentações. Os aleatórios tiveram 6.198 comparações contra 7.987, 6.996 e 7.986, enquanto os quase ordenados tiveram 513 movimentações contra 551, 1.010 e 2.723.

Para 10.000 elementos, os arranjos aleatórios se mantiveram o com menos comparações e os arranjos crescentes voltaram a ser os com menos movimentações. Os aleatórios fizeram 89.115 contra 113.631, 103.644 e 113.632, enquanto os crescentes fizeram 5.904 movimentações contra 10.904, 5.906 e 34.979.

Dessa maneira, a estratégia do pivô mediana de três performou melhor em arranjos aleatórios em termos de comparações e em arranjos crescentes em termos de movimentações. Vale ressaltar, contudo, que a diferença no número de comparações entre os arranjos não se mostrou tão significativa quanto a do número de movimentações. Os arranjos crescentes e decrescentes (que tiveram desempenhos semelhantes) apresentaram cerca de duas vezes menos movimentações que os arranjos quase ordenados e cerca de seis vezes menos que os arranjos aleatórios. Portanto, mesmo que os arranjos aleatórios tenham menos comparações, o seu grande número de movimentações torna esse conjunto de elementos o menos ideal. O pivô mediana de três performa melhor, então, para arranjos crescentes e decrescentes, com pouca diferença entre eles.

Pivô aleatório: Nessa técnica, percebe-se que dois tipos de arranjo se sobressaem em relação aos demais: os crescentes e os quase ordenados.

Para 100 elementos, o arranjo aleatório foi o que realizou menos comparações – 424 - contra 661 do crescente, 575 do decrescente e 651 do quase ordenado. Enquanto isso, o arranjo crescente realizou menos movimentações – 66 - contra 113 do decrescente, 67 (uma a mais) do quase ordenado e 192 do aleatório.

Para 1.000 elementos, os arranjos aleatórios ainda foram os que apresentaram menos comparações – 7.552 - contra 11.202 dos crescentes, 10.064 dos decrescentes e 11.016 dos quase ordenados. Para movimentações, os arranjos crescentes se mantiveram os com menos números – 668 – contra 1.162 dos decrescentes, 670 dos quase ordenados e 2.699 dos aleatórios.

Para 10.000 elementos, os arranjos aleatórios se mantiveram os melhores em termos de comparação com 105.653 ocorrências contra 156.185 dos crescentes, 144.926 dos decrescentes e 157.559 dos quase ordenados. Já em termos de movimentações, os melhores foram os arranjos quase ordenados, com 6.666, contra 6.667 (um a mais) dos crescentes, 11.663 dos decrescentes e 34.659 dos quases ordenados.

Desse modo, a escolha de um pivô aleatório teve um melhor desempenho, em comparações, para os arranjos aleatórios e, em movimentações para os crescentes e quase ordenados (com diferenças quase insignificantes entre eles). Percebe-se, todavia, que os arranjos aleatórios, apesar de possuírem menos comparações, realizam significativamente mais movimentações (192 para 100 elementos, 2.699 para 1.000 e 34.659 para 10.000), o que torna essa alternativa pouco eficiente para registros maiores. Assim, essa estratégia é mais indicada para arranjos crescentes e quase ordenados.

Arranjo crescente: Para os arranjos crescentes, o menor número de comparações e movimentações ocorreu para o QuickSort com pivô mediana de três e pivô aleatório. Em especial, destaca-se a significativa redução do número de comparações para essas duas estratégias à medida que o tamanho do registro aumenta, chegando a uma quantidade aproximadamente 440 vezes menor para 10.000 elementos (pivô mediana de três contra pivô no final).

Dessa maneira, essas estratégias se mostram as melhores, com a escolha do pivô mediana de três apresentando um desempenho superior em termos de comparações (103.644 contra 156.185 para 10.000 elementos) e movimentações (5.906 contra 6.667 para 10.000 elementos). Assim, esse tipo de pivô é a melhor escolha para arranjos crescentes, principalmente pelo equilíbrio entre comparações e movimentações e um desempenho mais consistente.

Arranjo decrescente: Para os arranjos decrescentes, percebe-se um comportamento semelhante ao dos arranjos crescentes, visto que o menor número de comparações e movimentações também foi observado nos pivôs mediana de três e aleatório. Esse comportamento é evidenciado sobretudo para grandes conjuntos de elementos, chegando a um número de comparações 439 vezes menor (pivô mediana de três contra pivô no final) e um número de movimentações 1,6 menor (pivô mediana de três contra pivô no início ou pivô no final).

Dessa maneira, as duas estratégias são as melhores, mas com uma superioridade do pivô mediana de três, assim como para os arranjos crescentes.

Arranjo quase ordenado: Para os arranjos quase ordenados, nota-se um resultado semelhante ao dos arranjos anteriores. Os pivôs mediana de três e aleatórios se sobressaem com facilidade em relação aos pivôs no início e no final, contudo com uma superioridade do pivô mediana de três tanto em comparações quanto em movimentações.

Arranjo aleatório: Para os arranjos aleatórios, destacam-se os pivôs mediana de três e no início, mas com uma diferença menor para as outras estratégias. O primeiro lidera no número de comparações (89.115 para 10.000 elementos, contra 123.963 do pivô no início, 124.150 do pivô no final e 105.653 do pivô aleatório), enquanto o segundo lidera por pouco no número de movimentações (34.365 para 10.000 elementos contra 34.431 do pivô no final, 34.979 do pivô mediana de três e 34.659 do pivô aleatório).

Logo, o pivô mediana de três é o ideal para esse tipo de registro, tendo em vista um equilíbrio entre comparações e movimentações.