Universidade Feevale

Bacharelado em Ciência da Computação

Estrutura de Dados

Matheus Duarte

Trabalho final – Estrutura de Dados

1 Escolha do algoritmo.

O algoritmo escolhido para este trabalho foi o QuickSort. A escolha se deu por conta de sua semelhança aos algoritmos estudados em aula, MergeSort e InsertionSort, sendo em minha percepção quase uma mescla entre os dois algoritmos.

Este foi implementado em linguagem C, em arquivo enviado em conjunto com este documento, com todas as suas partes devidamente comentadas e explicadas.

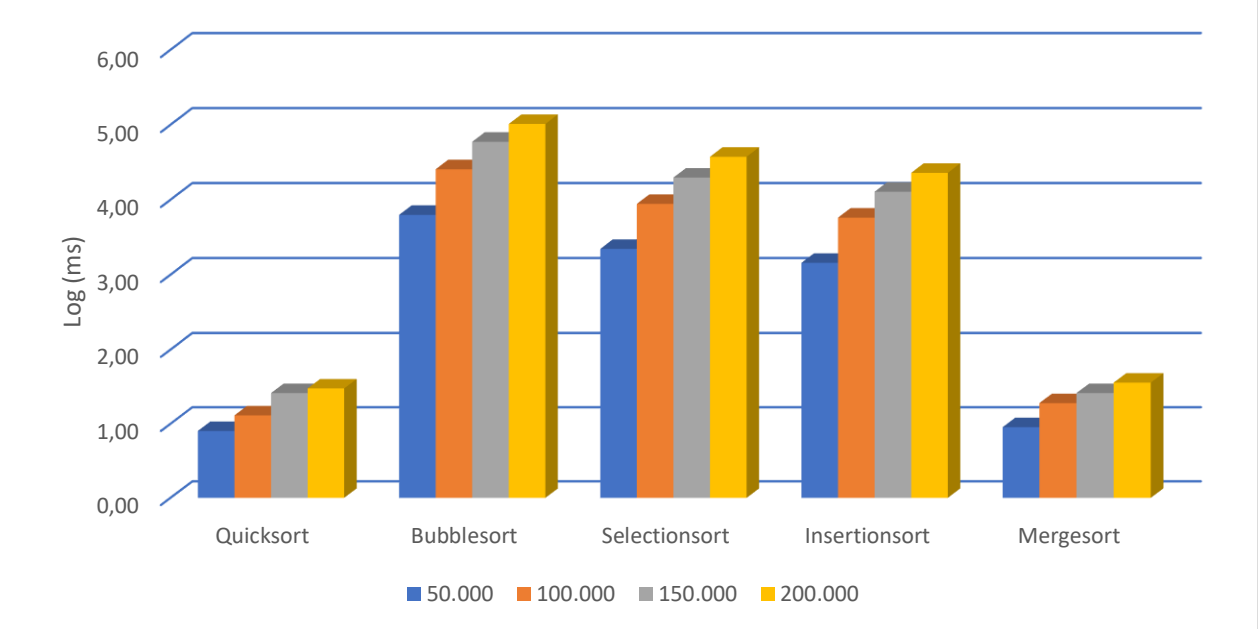
2 Verificação.

Como solicitado foi verificado o tempo de execução do algoritmo em implementado em milissegundos com vetores de 50 mil, 100 mil, 150 mil e 200 mil elementos. Conseguindo os resultados a seguir:

- Vetor de 50 mil: 8 ms.
- Vetor de 100mil: 13 ms.
- Vetor de 150 mil: 26 ms.
- Vetor de 200 mil: 30 ms.

3 Comparação.

Como solicitado foi feita a seguinte comparação gráfica dos tempos de execução entre o algoritmo implementado (QuickSort) com os algoritmos BubbleSort, SelectionSort, InsertionSort e MergeSort, para vetores de 50 mil, 100 mil, 150 mil e 200 mil números inteiros:



Para melhor visualização foi usada escala logarítmica nos tempos de execução em ms.

4 Dados usados

Todos os dados usados foram retirados do mesmo computador na primeira execução de cada algoritmo

- Quicksort:
- 50 mil: 8 ms
- 100 mil: 13 ms
- 150 mil: 26 ms
- 200 mil: 30 ms

- Bubblesort:
- 50 mil: 6277 ms
- 100 mil: 25505 ms
- 150 mil: 59412 ms
- 200 mil: 102585 ms

- Selectionsort:
- 50 mil: 2208 ms
- 100 mil: 8804 ms
- 150 mil: 19746 ms
- 200 mil: 37467 ms

- Insertionsort:
- 50 mil: 1441 ms
- 100 mil: 5776 ms
- 150 mil: 12811 ms
- 200 mil: 22861 ms

- Mergesort:
- 50 mil: 9 ms
- 100 mil: 19 ms
- 150 mil: 26 ms
- 200 mil: 36 ms

5 Explicação didática.

O algoritmo implementado faz o uso de 3 funções:

Função Troca

Função auxiliar da Partição utilizada para trocar de lugar dois valores do vetor.

```
void troca(int *a, int *b) {
   int temp = *a
   *a = *b;
   *b = temp;
}
```

Função Partição

função auxiliar de Quicksort utilizada para fazer a partição do vetor recebido em duas sub partições: sub partição esquerda (com valor menores ou iguais ao pivô) e sub partição direita (com valores maiores que o pivô).

```
int particao(int A[], int primeiro, int ultimo) {
   int pivo = A[ultimo];
   int i = (primeiro - 1);
   for (int j = primeiro; j < ultimo; j++) {
      if (A[j] <= pivo)
         i++;
         troca(&A[i], &A[j]);
      }
   }
   troca(&A[i + 1], &A[ultimo]);
   return (i + 1);
}
```

Função QuickSort

Função principal do algoritmo sendo responsável pela recursividade da função partição.

```
void quick_sort(int A[], int primeiro, int ultimo) {
   if (primeiro < ultimo)
      int pivo = particao(A, primeiro, ultimo);
      quick_sort(A, primeiro, pivo - 1);
      quick_sort(A, pivo + 1, ultimo);
```

```
    }
}
```

Inicialização

- Vetor inicial: {9,8,7,6,5,4,3,2,1,0}
- quick_sort(vetor, 0, 9)

Primeira Partição (vetor completo)

Partição (vetor, 0, 9)

- Pivô: 0
- i = -1
- Iterando j de 0 a 8:
  - Todos os elementos do vetor são maiores que o pivô 0, então não há trocas durante a iteração.
- Troca final: troca(&vetor[0], &vetor[9]) (troca 9 com 0)
- Vetor após a partição: {0,8,7,6,5,4,3,2,1,9}
- Retorna índice do pivô: 0

Chamadas Recursivas após a Primeira Partição

- quick_sort(vetor, 0, -1) (sub vetor vazio, retorna imediatamente)
- quick_sort(vetor, 1, 9)

Segunda Partição (sub vetor {8,7,6,5,4,3,2,1,9})

Partição (vetor, 1, 9)

- Pivô: 9
- i = 0
- Iterando j de 1 a 8:
  - Todos os elementos são menores que o pivô 9, então i é incrementado e trocas são feitas:
    - troca(&vetor[1], &vetor[1]) (8 com 8)
    - troca(&vetor[2], &vetor[2]) (7 com 7)
    - troca(&vetor[3], &vetor[3]) (6 com 6)
    - troca(&vetor[4], &vetor[4]) (5 com 5)
    - troca(&vetor[5], &vetor[5]) (4 com 4)
    - troca(&vetor[6], &vetor[6]) (3 com 3)
    - troca(&vetor[7], &vetor[7]) (2 com 2)
    - troca(&vetor[8], &vetor[8]) (1 com 1)
- Troca final: troca(&vetor[9], &vetor[9]) (não muda nada)
- Vetor após a partição: {0, 1, 7, 6, 5, 4, 3, 2, 8, 9}
- Retorna índice do pivô: 9

Chamadas Recursivas após a Segunda Partição

- quick_sort(vetor, 1, 8)
- quick_sort(vetor, 10, 9) (sub vetor vazio, retorna imediatamente)

Terceira Partição (sub vetor {1, 7, 6, 5, 4, 3, 2, 8})

Partição (vetor, 1, 8)

- Pivô: 8
- i = 0
- Iterando j de 1 a 7:
  - Todos os elementos são menores que o pivô 8, então i é incrementado e trocas são feitas:
    - troca(&vetor[1], &vetor[1]) (1 com 1)
    - troca(&vetor[2], &vetor[2]) (7 com 7)
    - troca(&vetor[3], &vetor[3]) (6 com 6)
    - troca(&vetor[4], &vetor[4]) (5 com 5)
    - troca(&vetor[5], &vetor[5]) (4 com 4)
    - troca(&vetor[6], &vetor[6]) (3 com 3)
    - troca(&vetor[7], &vetor[7]) (2 com 2)
- Troca final: troca(&vetor[8], &vetor[8]) (não muda nada)
- Vetor após a partição: {0, 1, 2, 6, 5, 4, 3, 7, 8, 9}
- Retorna índice do pivô: 8

Chamadas Recursivas após a Terceira Partição

- quick_sort(vetor, 1, 7)
- quick_sort(vetor, 9, 8) (sub vetor vazio, retorna imediatamente)

Quarta Partição (sub vetor {1, 2, 6, 5, 4, 3, 7})

Partição (vetor, 1, 7)

- Pivô: 7
- i = 0
- Iterando j de 1 a 6:
  - Todos os elementos são menores que o pivô 7, então i é incrementado e trocas são feitas:
    - troca(&vetor[1], &vetor[1]) (1 com 1)
    - troca(&vetor[2], &vetor[2]) (2 com 2)
    - troca(&vetor[3], &vetor[3]) (6 com 6)
    - troca(&vetor[4], &vetor[4]) (5 com 5)
    - troca(&vetor[5], &vetor[5]) (4 com 4)
    - troca(&vetor[6], &vetor[6]) (3 com 3)
- Troca final: troca(&vetor[7], &vetor[7]) (não muda nada)
- Vetor após a partição: {0, 1, 2, 3, 5, 4, 6, 7, 8, 9}
- Retorna índice do pivô: 7

Chamadas Recursivas após a Quarta Partição

- quick_sort(vetor, 1, 6)
- quick_sort(vetor, 8, 7) (sub vetor vazio, retorna imediatamente)

Quinta Partição (sub vetor {1, 2, 3, 5, 4, 6})

Partição (vetor, 1, 6)

- Pivô: 6
- i = 0
- Iterando j de 1 a 5:
  - Todos os elementos são menores que o pivô 6, então i é incrementado e trocas são feitas:
    - troca(&vetor[1], &vetor[1]) (1 com 1)
    - troca(&vetor[2], &vetor[2]) (2 com 2)
    - troca(&vetor[3], &vetor[3]) (3 com 3)
    - troca(&vetor[4], &vetor[4]) (5 com 5)
    - troca(&vetor[5], &vetor[5]) (4 com 4)
- Troca final: troca(&vetor[6], &vetor[6]) (não muda nada)
- Vetor após a partição: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
- Retorna índice do pivô: 6

Chamadas Recursivas após a Quinta Partição

- quick_sort(vetor, 1, 5)
- quick_sort(vetor, 7, 6) (sub vetor vazio, retorna imediatamente)

Sexta Partição (sub vetor {1, 2, 3, 4, 5})

Partição (vetor, 1, 5)

- Pivô: 5
- i = 0
- Iterando j de 1 a 4:
  - Todos os elementos são menores que o pivô 5, então i é incrementado e trocas são feitas:
    - troca(&vetor[1], &vetor[1]) (1 com 1)
    - troca(&vetor[2], &vetor[2]) (2 com 2)
    - troca(&vetor[3], &vetor[3]) (3 com 3)
    - troca(&vetor[4], &vetor[4]) (4 com 4)
- Troca final: troca(&vetor[5], &vetor[5]) (não muda nada)
- Vetor após a partição: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
- Retorna índice do pivô: 5

Chamadas Recursivas após a Sexta Partição

- quick_sort(vetor, 1, 4)
- quick_sort(vetor, 6, 5) (sub vetor vazio, retorna imediatamente)

Sétima Partição (sub vetor {1, 2, 3, 4})

Partição (vetor, 1, 4)

- Pivô: 4
- i = 0
- Iterando j de 1 a 3:
  - Todos os elementos são menores que o pivô 4, então i é incrementado e trocas são feitas:
    - troca(&vetor[1], &vetor[1]) (1 com 1)
    - troca(&vetor[2], &vetor[2]) (2 com 2)
    - troca(&vetor[3], &vetor[3]) (3 com 3)
- Troca final: troca(&vetor[4], &vetor[4]) (não muda nada)
- Vetor após a partição: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
- Retorna índice do pivô: 4

Chamadas Recursivas após a Sétima Partição

- quick_sort(vetor, 1, 3)
- quick_sort(vetor, 5, 4) (sub vetor vazio, retorna imediatamente)

Oitava Partição (sub vetor {1, 2, 3})

Partição (vetor, 1, 3)

- Pivô: 3
- i = 0
- Iterando j de 1 a 2:
  - Todos os elementos são menores que o pivô 3, então i é incrementado e trocas são feitas:
    - troca(&vetor[1], &vetor[1]) (1 com 1)
    - troca(&vetor[2], &vetor[2]) (2 com 2)
- Troca final: troca(&vetor[3], &vetor[3]) (não muda nada)
- Vetor após a partição: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
- Retorna índice do pivô: 3

Chamadas Recursivas após a Oitava Partição

- quick_sort(vetor, 1, 2)
- quick_sort(vetor, 4, 3) (sub vetor vazio, retorna imediatamente)

Nona Partição (sub vetor {1, 2})

Partição (vetor, 1, 2)

- Pivô: 2
- i = 0
- Iterando j de 1 a 1:
  - Todos os elementos são menores que o pivô 2, então i é incrementado e trocas são feitas:
    - troca(&vetor[1], &vetor[1]) (1 com 1)

- Troca final: troca(&vetor[2], &vetor[2]) (não muda nada)
- Vetor após a partição: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
- Retorna índice do pivô: 2

Chamadas Recursivas após a Nona Partição

- quick_sort(vetor, 1, 1) (sub vetor com um único elemento, retorna imediatamente)
- quick_sort(vetor, 3, 2) (sub vetor vazio, retorna imediatamente)

Nesse ponto, o vetor está completamente ordenado. O vetor final após todas as partições e chamadas recursivas é:

Vetor Final Ordenado

- {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

Conclusão didática

Após fazer essa parte do trabalho posso concluir que o modo com que implementei o algoritmo QuickSort deve ser considerado um tanto ineficiente quando recebemos um vetor ordenado decrescentemente e temos que ordena-lo crescentemente, isso porque dessa forma meu algoritmo nunca se aproveita da sua estratégia base, dividir para conquistar, sendo que em nenhum momento as duas chamas recursivas são usadas após as partições.