UNIVERSIDADE DO ESTADO DO AMAZONAS
ESCOLA SUPERIOR DE TECNOLOGIA

BEATRIZ DAMASCENO FERNANDES - 2115310003
MATEUS BASTOS MAGALHÃES MAR- 2215310063
MONIKE FREITAS DE SOUSA - 2115310040

AVALIAÇÃO DE DESEMPENHO DE ALGORITMOS ORDENADOS

MANAUS/AM
2023

BEATRIZ DAMASCENO FERNANDES
MATEUS BASTOS MAGALHÃES MAR
MONIKE FREITAS DE SOUSA

AVALIAÇÃO DE DESEMPENHO DE ALGORITMOS ORDENADOS

Relatório de Algoritmos e Estruturas de Dados II apresentado na Universidade do Estado do Amazonas a fim de obtenção de nota parcial no curso de Sistemas de Informação

MANAUS/AM
2023

Sumário

Introdução

Neste relatório será feita a comparação de performance dentre algoritmos de ordenação, visando compreender quais algoritmos mais se adequam a determinadas situações, fazendo uso de vetores de tamanhos díspares para melhor discernimento. Os algoritmos a serem aqui avaliados serão os seguintes:

a) Bubble Sort: Ele é um algoritmo que percorre a lista, compra elementos que estão um ao lado do outro e os inverte, caso estejam em ordem incorreta, Faz-se a repetição deste diversas vezes, até que toda a fila esteja ordenada.
b) Select Sort: divide a lista em duas partes: uma parte ordenada e uma parte não ordenada. Em cada iteração, encontra o menor elemento da parte não ordenada e o troca com o primeiro elemento da parte não ordenada. Isso continua até que toda a lista esteja ordenada.
c) Insert Sort: Ele cria uma lista ordenando um item de cada vez, sendo bem mais eficiente em listas com poucos dados. Se ele verifica que um elemento-chave for menor que seu predecessor, ele irá comparar com seus elementos anteriores, mover os maiores uma posição acima e liberar espaço para o elemento a ser trocado.
d) Shell Sort: É um algoritmo indicado para quando a entrada já está parcialmente ordenada. Ele passa varias vezes pela lista dividindo um grupo maior em grupos menores, onde será aplicado o método de organização por inserção.
e) Quick Sort: usa uma estratégia de divisão e conquista para ordenar elementos. Ele escolhe um elemento como "pivô" e reorganiza os elementos à esquerda do pivô para serem menores que ele, enquanto os elementos à direita são maiores. Esse processo é aplicado recursivamente a cada uma das metades resultantes, até que toda a lista esteja ordenada.
f) Heap Sort: O Heap Sort transforma a lista em uma estrutura de heap, que é uma árvore binária completa onde cada nó pai é maior (ou menor, dependendo do tipo de heap) do que seus filhos. Em seguida, o algoritmo extrai repetidamente o elemento máximo (ou mínimo) da heap, reestruturando a heap e repetindo esse processo até que a lista esteja ordenada.
g) Merge Sort: O Merge Sort também utiliza a estratégia de divisão e conquista. Ele divide repetidamente a lista pela metade até que cada sublista tenha um único elemento. Em seguida, mescla essas sublistas em pares ordenados, continuando o processo de mesclagem até que toda a lista esteja ordenada.

Implementação

No programa em C houve a implementação das funções de ordenação Bubble Sort, Selection Sort, Insertion Sort, Shell Sort, Quick Sort, Heap Sort e Merge Sort.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "vetores_aleatórios.h" // Importa os vetores em ordem aleatória
#include "vetores_crescentes.h" // Importa os vetores em ordem crescente

long unsigned int comparacoes = 0; // Contador de comparações
long unsigned int movimentacoes = 0; // Contador de movimentações
int operacao; // Operação escolhida pelo usuário no menu principal
int *pont_vetor; // Ponteiro para o vetor escolhido pelo usuário

void imprime_vetor(int *vetor, int n); // Função que imprime um vetor
int* copiar_vetor(int vetor_global[], int tam); // Função que copia o vetor global para um vetor local

void bubble_sort(int vetor[], int n); // Função para ordenar um vetor usando bubble sort
void selection_sort(int vetor[], int n); // Função para ordenar um vetor usando selection sort
void insertion_sort(int vetor[], int n); // Função para ordenar um vetor usando insertion sort
void shell_sort(int vetor[], int n); // Função para ordenar um vetor usando shell sort
void quick_sort(int vetor[], int inicio, int fim); // Função para ordenar um vetor usando quick sort
void heap_sort(int vetor[], int n); // Função para ordenar um vetor usando heap sort
void merge_sort(int vetor[], int inicio, int fim); // Função para ordenar um vetor usando merge sort

```

Ele inicia definindo as bibliotecas a serem utilizadas, assim como importando os vetores a serem utilizados ao longo da ordenação - vetores em ordem aleatória e em ordem crescente. Em seguida ocorre a definição de contadores, variáveis globais para fins de uso no código posteriormente e as funções que serão chamadas para ordenação.

```c
// Função para retornar o tipo de ordem desejado pelo usuário
int tipo_ordem(){
    int tipo;

    printf("\n \n");
    printf("************************************************** \n");
    printf("Escolha a ordem original do vetor: \n\n");
    printf("1 - Aleatório \n");
    printf("2 - Ordenado \n");
    printf("************************************************** \n \n");
    scanf("%d", &tipo);

    return tipo;
}
```

Após isso, é criada uma função que permitirá ao usuário escolher entre ordenar um vetor Aleatório ou um vetor em ordem crescente.

```c
// Função que obtêm a referência do vetor escolhido pelo usuário e retorna o seu tamanho
int get_vetor(int tipo_ordem){
    int num_vetor, tamanho;

    if (tipo_ordem == 1) {

        do {
            printf("\n \n");
            printf("************************************************* \n");
            printf("Escolha o tamanho do vetor: \n\n");
            printf("1 - 100 \n");
            printf("2 - 1000 \n");
            printf("3 - 10000 \n");
            printf("4 - 100000 \n");
            printf("************************************************* \n \n");
            scanf("%d", &num_vetor);
        } while(num_vetor != 1 && num_vetor != 2 && num_vetor != 3 && num_vetor != 4);

        switch (num_vetor) {

            case 1: {
                pont_vetor = copiar_vetor(aleat_100, 100);
                tamanho = 100;
            }
                break;

            case 2: {
                pont_vetor = copiar_vetor(aleat_1000, 1000);
                tamanho = 1000;
            }
                break;

            case 3: {
                pont_vetor = copiar_vetor(aleat_10000, 10000);
                tamanho = 10000;
            }
                break;

            case 4: {
                pont_vetor = copiar_vetor(aleat_100000, 100000);
                tamanho = 100000;
            }
                break;
        }
    }
```

```
    else if (tipo_ordem == 2) {
        do {
            printf("\n \n");
            printf("************************************************ \n");
            printf("Escolha o tamanho do vetor: \n\n");
            printf("1 - 100 \n");
            printf("2 - 1000 \n");
            printf("3 - 10000 \n");
            printf("4 - 100000 \n");
            printf("************************************************ \n \n");
            scanf("%d", &num_vetor);
            } while(num_vetor != 1 && num_vetor != 2 && num_vetor != 3 && num_vetor != 4);

        switch (num_vetor) {

            case 1: {
                pont_vetor = copiar_vetor(orden_100, 100);
                tamanho = 100;
            }
                break;

            case 2: {
                pont_vetor = copiar_vetor(orden_1000, 1000);
                tamanho = 1000;
            }
                break;

            case 3: {
                pont_vetor = copiar_vetor(orden_10000, 10000);
                tamanho = 10000;
            }
                break;

            case 4: {
                pont_vetor = copiar_vetor(orden_100000, 100000);
                tamanho = 100000;
            }
                break;
        }
    }

    return tamanho;
}

```

Nas linhas seguintes, há uma função para que o usuário informe o tamanho do vetor que deseja utilizar, podendo escolher entre os valores, 100, 1000, 10000 e 100000.

```c
// Função para executar um dos algoritmos de ordenação de acordo com a escolha do usuário
void executar_sort(int operacao, int tamanho){
    clock_t inicio, fim;

    double tempo;

    // Inicia a contagem do tempo
    inicio = clock();

    // Executa um dos algoritmos de ordenação a seguir de acordo com a escolha do usuário
    switch (operacao) {

        case 1: {
            // Ordena o vetor usando bubble sort
            bubble_sort(pont_vetor, tamanho);
        }
            break;

        case 2: {
            // Ordena o vetor usando selection sort
            selection_sort(pont_vetor, tamanho);
        }
            break;

        case 3: {
            // Ordena o vetor usando insertion sort
            insertion_sort(pont_vetor, tamanho);
        }
            break;

        case 4: {
            // Ordena o vetor usando shell sort
            shell_sort(pont_vetor, tamanho);
        }
            break;

        case 5: {
            // Ordena o vetor usando quick sort
            quick_sort(pont_vetor, 0, tamanho - 1);
        }
            break;

        case 6: {
            // Ordena o vetor usando heap sort
            heap_sort(pont_vetor, tamanho);
        }
            break;
```

```c
        case 7: {
            // Ordena o vetor usando merge sort
            merge_sort(pont_vetor, 0, tamanho - 1);
        }
            break;
    }

    // Finaliza a contagem do tempo
    fim = clock();

    // Calcula o tempo em segundos
    tempo = (double)(fim - inicio) / CLOCKS_PER_SEC;

    // Imprime os resultados
    printf("Tempo de execução: %f segundos\n", tempo);
    printf("Número de comparações: %d\n", comparacoes);
    printf("Número de movimentações: %d\n", movimentacoes);
}
```

Por último, o usuário seleciona qual algoritmo de ordenação será utilizado, tendo como opções Bubble Sort, Selection Sort, Insertion Sort, Shell Sort, Quick Sort, Heap Sort e

Merge Sort. Um clock é definido para fazer computar o tempo levado durante cada uma das ordenações, a fim de comparar ao final qual é mais eficiente.

```c
void menu_principal(){
    //operacao
    printf("\n \n");
    printf("************************************************ \n");
    printf("Escolha o operacao de ordenação: \n\n");
    printf("1 - BubbleSort \n");
    printf("2 - SelectSort \n");
    printf("3 - InsertSort \n");
    printf("4 - ShellSort \n");
    printf("5 - QuickSort \n");
    printf("6 - HeapSort  \n");
    printf("7 - MergeSort  \n");
    printf("************************************************ \n \n");
    scanf("%d", &operacao);

    switch(operacao)
    {
        case 1:
            {
                printf("Execucao do BubbleSort \n\n");

                int ordem = tipo_ordem();
                int tamanho = get_vetor(ordem);

                int imprimir;


                do{
                    printf("Deseja imprimir o vetor antes da operação? \n\n");
                    printf("1 - Sim \n");
                    printf("0 - Não \n\n");

                    scanf("%d", &imprimir);

                    if (imprimir != 0 && imprimir != 1) {
                        printf("Opção inválida \n");
                    }
                } while (imprimir != 0 && imprimir != 1);


                if (imprimir == 1){
                    imprime_vetor(pont_vetor, tamanho);
                }

                executar_sort(operacao, tamanho);

```

```c
            do{
                printf("Deseja imprimir o vetor depois da operação? \n\n");
                printf("1 - Sim \n");
                printf("0 - Não \n\n");

                scanf("%d", &imprimir);

                if (imprimir != 0 && imprimir != 1) {
                    printf("Opção inválida \n");
                }
            } while (imprimir != 0 && imprimir != 1);


            if (imprimir == 1){
                imprime_vetor(pont_vetor, tamanho);
            }

            comparacoes = 0;
            movimentacoes = 0;

            menu_principal();
        }
        break;

    case 2:
        {
            printf("Execucao do SelectSort \n\n");

            int ordem = tipo_ordem();
            int tamanho = get_vetor(ordem);

            int imprimir;


            do{
                printf("Deseja imprimir o vetor antes da operação? \n\n");
                printf("1 - Sim \n");
                printf("0 - Não \n\n");

                scanf("%d", &imprimir);

                if (imprimir != 0 && imprimir != 1) {
                    printf("Opção inválida \n");
                }
            } while (imprimir != 0 && imprimir != 1);
```

```c
            if (imprimir == 1){
                imprime_vetor(pont_vetor, tamanho);
            }

            executar_sort(operacao, tamanho);

            do{
                printf("Deseja imprimir o vetor depois da operação? \n\n");
                printf("1 - Sim \n");
                printf("0 - Não \n\n");

                scanf("%d", &imprimir);

                if (imprimir != 0 && imprimir != 1) {
                    printf("Opção inválida \n");
                }
            } while (imprimir != 0 && imprimir != 1);


            if (imprimir == 1){
                imprime_vetor(pont_vetor, tamanho);
            }

            comparacoes = 0;
            movimentacoes = 0;

            menu_principal();
        }

        break;

    case 3:
        {
            printf("Execucao do InsertSort \n\n");

            int ordem = tipo_ordem();
            int tamanho = get_vetor(ordem);

            int imprimir;


            do{
                printf("Deseja imprimir o vetor antes da operação? \n\n");
                printf("1 - Sim \n");
                printf("0 - Não \n\n");

                scanf("%d", &imprimir);
```

```
348                 if (imprimir != 0 && imprimir != 1) {
349                     printf("Opção inválida \n");
350                 }
351             } while (imprimir != 0 && imprimir != 1);
352 
353 
354             if (imprimir == 1){
355                 imprime_vetor(pont_vetor, tamanho);
356             }
357 
358             executar_sort(operacao, tamanho);
359 
360             do{
361                 printf("Deseja imprimir o vetor depois da operação? \n\n");
362                 printf("1 - Sim \n");
363                 printf("0 - Não \n\n");
364 
365                 scanf("%d", &imprimir);
366 
367                 if (imprimir != 0 && imprimir != 1) {
368                     printf("Opção inválida \n");
369                 }
370             } while (imprimir != 0 && imprimir != 1);
371 
372 
373             if (imprimir == 1){
374                 imprime_vetor(pont_vetor, tamanho);
375             }
376 
377             comparacoes = 0;
378             movimentacoes = 0;
379 
380             menu_principal();
381         }
382         break;
383 
384     case 4:
385         {
386             printf("Execucao do ShellSort \n\n");
387 
388             int ordem = tipo_ordem();
389             int tamanho = get_vetor(ordem);
390 
391             int imprimir;
392 
393 
394             do{
```

```
395                 printf("Deseja imprimir o vetor antes da operação? \n\n");
396                 printf("1 - Sim \n");
397                 printf("0 - Não \n\n");
398 
399                 scanf("%d", &imprimir);
400 
401                 if (imprimir != 0 && imprimir != 1) {
402                     printf("Opção inválida \n");
403                 }
404             } while (imprimir != 0 && imprimir != 1);
405 
406 
407             if (imprimir == 1){
408                 imprime_vetor(pont_vetor, tamanho);
409             }
410 
411             executar_sort(operacao, tamanho);
412 
413             do{
414                 printf("Deseja imprimir o vetor depois da operação? \n\n");
415                 printf("1 - Sim \n");
416                 printf("0 - Não \n\n");
417 
418                 scanf("%d", &imprimir);
419 
420                 if (imprimir != 0 && imprimir != 1) {
421                     printf("Opção inválida \n");
422                 }
423             } while (imprimir != 0 && imprimir != 1);
424 
425 
426             if (imprimir == 1){
427                 imprime_vetor(pont_vetor, tamanho);
428             }
429 
430             comparacoes = 0;
431             movimentacoes = 0;
432 
433             menu_principal();
434         }
435         break;
436 
437     case 5:
438         {
439             printf("Execucao do QuickSort \n\n");
440 
441             int ordem = tipo_ordem();
442             int tamanho = get_vetor(ordem);
```

```c
            int imprimir;


            do{
                printf("Deseja imprimir o vetor antes da operação? \n\n");
                printf("1 - Sim \n");
                printf("0 - Não \n\n");

                scanf("%d", &imprimir);

                if (imprimir != 0 && imprimir != 1) {
                    printf("Opção inválida \n");
                }
            } while (imprimir != 0 && imprimir != 1);


            if (imprimir == 1){
                imprime_vetor(pont_vetor, tamanho);
            }

            executar_sort(operacao, tamanho);

            do{
                printf("Deseja imprimir o vetor depois da operação? \n\n");
                printf("1 - Sim \n");
                printf("0 - Não \n\n");

                scanf("%d", &imprimir);

                if (imprimir != 0 && imprimir != 1) {
                    printf("Opção inválida \n");
                }
            } while (imprimir != 0 && imprimir != 1);


            if (imprimir == 1){
                imprime_vetor(pont_vetor, tamanho);
            }

            comparacoes = 0;
            movimentacoes = 0;

            menu_principal();
        }
        break;

```

```c
case 6:
    {
        printf("Execucao do HeapSort  \n\n");

        int ordem = tipo_ordem();
        int tamanho = get_vetor(ordem);

        int imprimir;


        do{
            printf("Deseja imprimir o vetor antes da operação? \n\n");
            printf("1 - Sim \n");
            printf("0 - Não \n\n");

            scanf("%d", &imprimir);

            if (imprimir != 0 && imprimir != 1) {
                printf("Opção inválida \n");
            }
        } while (imprimir != 0 && imprimir != 1);


        if (imprimir == 1){
            imprime_vetor(pont_vetor, tamanho);
        }

        executar_sort(operacao, tamanho);

        do{
            printf("Deseja imprimir o vetor depois da operação? \n\n");
            printf("1 - Sim \n");
            printf("0 - Não \n\n");

            scanf("%d", &imprimir);

            if (imprimir != 0 && imprimir != 1) {
                printf("Opção inválida \n");
            }
        } while (imprimir != 0 && imprimir != 1);


        if (imprimir == 1){
            imprime_vetor(pont_vetor, tamanho);
        }

```

```
536                 comparacoes = 0;
537                 movimentacoes = 0;
538 
539                 menu_principal();
540             }
541             break;
542 
543         case 7:
544             {
545                 printf("Execucao do MergeSort  \n\n");
546 
547                 int ordem = tipo_ordem();
548                 int tamanho = get_vetor(ordem);
549 
550                 int imprimir;
551 
552 
553                 do{
554                     printf("Deseja imprimir o vetor antes da operação? \n\n");
555                     printf("1 - Sim \n");
556                     printf("0 - Não \n\n");
557 
558                     scanf("%d", &imprimir);
559 
560                     if (imprimir != 0 && imprimir != 1) {
561                         printf("Opção inválida \n");
562                     }
563                 } while (imprimir != 0 && imprimir != 1);
564 
565 
566                 if (imprimir == 1){
567                     imprime_vetor(pont_vetor, tamanho);
568                 }
569 
570                 executar_sort(operacao, tamanho);
571 
572                 do{
573                     printf("Deseja imprimir o vetor depois da operação? \n\n");
574                     printf("1 - Sim \n");
575                     printf("0 - Não \n\n");
576 
577                     scanf("%d", &imprimir);
578 
579                     if (imprimir != 0 && imprimir != 1) {
580                         printf("Opção inválida \n");
581                     }
```

```
571
572                 do{
573                     printf("Deseja imprimir o vetor depois da operação? \n\n");
574                     printf("1 - Sim \n");
575                     printf("0 - Não \n\n");
576
577                     scanf("%d", &imprimir);
578
579                     if (imprimir != 0 && imprimir != 1) {
580                         printf("Opção inválida \n");
581                     }
582                 } while (imprimir != 0 && imprimir != 1);
583
584
585                 if (imprimir == 1){
586                     imprime_vetor(pont_vetor, tamanho);
587                 }
588
589                 comparacoes = 0;
590                 movimentacoes = 0;
591
592                 menu_principal();
593             }
594             break;
595
596         default:
597             {
598                 printf("Opcao invalida");
599             }
600             break;
601     }
602
603
604 }
605
```

Na função acima, a partir da escolha do usuário acerca de um método de ordenação, o programa encarrega-se de chamar o método de verificar qual dos dois tipos de ordem será utilizado, assim como o método que verifica a opção de tamanho de vetor, ambas anteriormente descritas, para logo em seguida iniciar a ordenação, chamando a função do tipo escolhido.

```c
// Função para ordenar um vetor usando bubble sort
void bubble_sort(int vetor[], int n) {
  int i, j, aux;
  // Percorre o vetor n - 1 vezes
  for (i = 0; i < n - 1; i++) {
    // Compara cada elemento com o seu sucessor
    for (j = i + 1; j < n; j++) {
      // Incrementa o número de comparações
      comparacoes++;
      // Se o elemento atual for maior que o próximo, troca de posição
      if (vetor[i] > vetor[j]) {
        aux = vetor[i];
        vetor[i] = vetor[j];
        vetor[j] = aux;
        // Incrementa o número de movimentações
        movimentacoes++;
      }
    }
  }
}

```

O bubble sort, a primeira opção de ordenação, percorre o vetor em seu tamanho menos uma posição (n-1), comparando cada elemento com seu sucessor. Visto que uma comparação foi realizada, um contador será incrementado de 1 (comparacoes++), seguindo para uma verificação sobre se o próximo elemento é menor que o elemento atual. Caso positivo, ocorrerá a troca de posição e o contador de trocas será acrescido de uma unidade (movimentacoes++).

```c
// Função para ordenar um vetor usando selection sort
void selection_sort(int vetor[], int n) {
  int i, j, min, aux;
  // Percorre o vetor da primeira até a penúltima posição
  for (i = 0; i < n - 1; i++) {
    // Assume que o elemento atual é o menor
    min = i;
    // Procura o menor elemento nas posições restantes
    for (j = i + 1; j < n; j++) {
      // Incrementa o número de comparações
      comparacoes++;
      // Se encontrar um elemento menor que o atual, atualiza o índice do menor
      if (vetor[j] < vetor[min]) {
        min = j;
      }
    }
    // Se o índice do menor for diferente do atual, troca de posição
    if (min != i) {
      aux = vetor[min];
      vetor[min] = vetor[i];
      vetor[i] = aux;
      // Incrementa o número de movimentações
      movimentacoes++;
    }
  }
}
```

O segundo método é o Selection Sort, o qual vai percorrer o vetor até sua penúltima posição (n-1) e assume que o elemento atual é o menor, procurando nos próximos elementos um que seja menor, iniciando, então, um laço de comparação no qual um contador será acrescido a cada comparação e, caso encontre um valor menor com índice maior, faz a troca, incrementando, assim, o contador de movimentações.

```
655  // Função para ordenar um vetor usando insertion sort
656  void insertion_sort(int vetor[], int n) {
657    int i, j, k, aux;
658    // Percorre o vetor da segunda posição até a última
659    for (i = 1; i < n; i++) {
660      aux = vetor[i]; // Guarda o elemento atual
661      // Procura a posição correta para inserir o elemento atual
662      for (j = 0; j < i; j++) {
663        // Incrementa o número de comparações
664        comparacoes++;
665        // Se o elemento atual for menor que o elemento da posição j, interrompe o loop
666        if (aux < vetor[j]) {
667          break;
668        }
669      }
670      // Move os elementos maiores que o elemento atual para a direita
671      for (k = i; k > j; k--) {
672        vetor[k] = vetor[k - 1];
673        // Incrementa o número de movimentações
674        movimentacoes++;
675      }
676      // Insere o elemento atual na posição correta
677      vetor[j] = aux;
678    }
679  }
680
```

O método três, o Insertion Sort, percorre até a última posição do vetor com a ajuda de um auxiliar, que vai guardando o elemento atual e comparando, buscando a posição correta para inseri-lo (comparacao++). Encontrando um elemento que seja maior, o loop é interrompido, movendo os elementos maiores que o atual para a direita (movimentacoes++) e atualizando a posição do atual.

```
680
681 // Função para ordenar um vetor usando shell sort
682 void shell_sort(int vetor[], int n) {
683   int i, j, k, h, aux;
684   // Define o intervalo inicial como metade do tamanho do vetor
685   h = n / 2;
686   // Repete o processo até que o intervalo seja menor que 1
687   while (h > 0) {
688     // Percorre o vetor da posição h até a última
689     for (i = h; i < n; i++) {
690       aux = vetor[i]; // Guarda o elemento atual
691       // Compara o elemento atual com os elementos do intervalo
692       for (j = i; j >= h && aux < vetor[j - h]; j -= h) {
693         // Incrementa o número de comparações
694         comparacoes++;
695         // Move o elemento maior para a direita
696         vetor[j] = vetor[j - h];
697         // Incrementa o número de movimentações
698         movimentacoes++;
699       }
700       // Coloca o elemento atual na posição correta
701       vetor[j] = aux;
702     }
703     // Reduz o intervalo pela metade
704     h = h / 2;
705   }
706 }
707
```

O quarto método é o Shell Sort. Ele utiliza um intervalo inicial de metade do tamanho do vetor e, a cada iteração, reduz esse intervalo pela metade até que ele seja menor que 1. Dentro do laço principal, o algoritmo percorre o vetor e, para cada elemento, compara-o com os elementos separados pelo intervalo. Se um elemento for menor que o outro dentro desse intervalo, são feitas trocas para posicionar o elemento atual na posição correta dentro desse grupo. Isso continua até que todo o vetor seja percorrido com o intervalo estabelecido. A ideia é criar sequências parcialmente ordenadas no vetor, o que auxilia na redução do número de movimentações necessárias para a ordenação completa.

```c
// Função para escolher um pivô aleatório
int escolhe_pivo(int inicio, int fim) {
  // Retorna um número entre fim (inclusive) e inicio (inclusive)
  return (rand() % (fim - inicio + 1)) + inicio;
}

// Função para ordenar um vetor usando quick sort
void quick_sort(int vetor[], int inicio, int fim) {
  if (inicio < fim) {
    // Escolhe um pivô aleatório
    int pivo_indice = escolhe_pivo(inicio, fim);
    // Coloca o pivô no fim do vetor
    troca(&vetor[pivo_indice], &vetor[fim]);
    // Define o pivô como o último elemento
    int pivo = vetor[fim];
    // Define o índice da posição inicial
    int i = inicio;
    // Percorre o vetor da posição inicial até a penúltima
    for (int j = inicio; j < fim; j++) {
      // Incrementa o número de comparações
      comparacoes++;
      // Se o elemento atual for menor ou igual ao pivô, troca de posição com o elemento
      if (vetor[j] <= pivo) {
        troca(&vetor[i], &vetor[j]);
        // Incrementa o número de movimentações
        movimentacoes++;
        // Incrementa o índice da posição i
        i++;
      }
    }
    // Coloca o pivô na posição correta
    troca(&vetor[i], &vetor[fim]);
    // Incrementa o número de movimentações
    movimentacoes++;
    // Ordena as partes esquerda e direita do pivô
    quick_sort(vetor, inicio, i - 1);
    quick_sort(vetor, i + 1, fim);
  }
}
```

O código acima implementa o algoritmo de ordenação conhecido como Quick Sort. Ele seleciona um pivô aleatório dentro do intervalo especificado e rearranja os elementos do vetor de forma que os elementos menores que o pivô fiquem à esquerda dele e os maiores à direita.O pivô é escolhido aleatoriamente para evitar casos em que o algoritmo tenha um desempenho ruim com certos tipos de entradas ordenadas ou pré-ordenadas. O Quick Sort então divide o vetor em partições menores com base na posição correta do pivô, classificando recursivamente as partições restantes até que o vetor esteja totalmente ordenado.

```c
// Função para ajustar o heap com raiz em 'i'
void ajusta_heap(int vetor[], int n, int i) {
    int maior = i;
    int esquerda = 2 * i + 1;
    int direita = 2 * i + 2;

    // Comparação com o filho da esquerda
    if (esquerda < n && vetor[esquerda] > vetor[maior]) {
        comparacoes++;
        maior = esquerda;
    }

    // Comparação com o filho da direita
    if (direita < n && vetor[direita] > vetor[maior]) {
        comparacoes++;
        maior = direita;
    }

    // Se o maior não é a raiz
    if (maior != i) {
        movimentacoes++;
        // Troca a raiz com o maior elemento
        troca(&vetor[i], &vetor[maior]);

        // Recursivamente ajusta o subárvore afetado
        ajusta_heap(vetor, n, maior);
    }
}

// Função principal que realiza o heap sort
void heap_sort(int vetor[], int n) {
    // Constrói o heap (reorganiza o array)
    for (int i = n / 2 - 1; i >= 0; i--)
        ajusta_heap(vetor, n, i);

    // Extrai elementos um por um do heap
    for (int i = n - 1; i > 0; i--) {
        movimentacoes++;
        // Move a raiz atual para o final
        troca(&vetor[0], &vetor[i]);

        // Chama ajusta_heap no heap reduzido
        ajusta_heap(vetor, i, 0);
    }
}
```

Esse código implementa o algoritmo Heap Sort. Ele organiza os elementos em um vetor, transformando-o em uma estrutura de dados chamada de heap, onde o maior elemento está sempre na raiz. O algoritmo possui duas etapas principais: Construção do Heap (max-heap): O vetor inicial é organizado como um heap, começando das folhas e subindo até a raiz, garantindo que o maior elemento esteja na posição 0. E a Extração do Heap: O maior elemento (na raiz) é movido para a última posição do vetor, e em seguida, o heap é ajustado para garantir que o próximo maior elemento vá para a raiz. Esse processo é repetido até que todo o vetor esteja ordenado. O método ajusta_heap é usado para garantir que a estrutura de heap seja mantida durante a construção e extração, trocando elementos conforme necessário para reorganizar o heap corretamente.

```c
void merge(int vetor[], int inicio, int meio, int fim) {
  int i = inicio; // Índice do primeiro elemento da primeira metade
  int j = meio + 1; // Índice do primeiro elemento da segunda metade
  int k = 0; // Índice do elemento a ser inserido no vetor auxiliar
  int *aux = malloc(sizeof(int) * (fim - inicio + 1)); // Aloca um vetor auxiliar
  // Percorre as duas metades do vetor, comparando e copiando os elementos em ordem
  while (i <= meio && j <= fim) {
    // Incrementa o número de comparações
    comparacoes++;
    // Se o elemento da primeira metade for menor ou igual ao da segunda, copia ele par
    if (vetor[i] <= vetor[j]) {
      aux[k] = vetor[i];
      i++;
    }
    // Senão, copia o elemento da segunda metade para o vetor auxiliar
    else {
      aux[k] = vetor[j];
      j++;
    }
    // Incrementa o número de movimentações
    movimentacoes++;
    // Incrementa o índice do vetor auxiliar
    k++;
  }
  // Copia os elementos restantes da primeira metade, se houver
  while (i <= meio) {
    aux[k] = vetor[i];
    i++;
    k++;
    // Incrementa o número de movimentações
    movimentacoes++;
  }
  // Copia os elementos restantes da segunda metade, se houver
  while (j <= fim) {
    aux[k] = vetor[j];
    j++;
    k++;
    // Incrementa o número de movimentações
    movimentacoes++;
  }
  // Copia os elementos do vetor auxiliar de volta para o vetor original
  for (i = inicio; i <= fim; i++) {
    vetor[i] = aux[i - inicio];
    // Incrementa o número de movimentações
    movimentacoes++;
  }
  free(aux); // Libera a memória do vetor auxiliar
}

// Função para ordenar um vetor usando merge sort
void merge_sort(int vetor[], int inicio, int fim) {
  // Se o vetor tiver mais de um elemento
  if (inicio < fim) {
    // Calcula o índice do meio do vetor
    int meio = (inicio + fim) / 2;
    // Ordena a primeira metade do vetor
    merge_sort(vetor, inicio, meio);
    // Ordena a segunda metade do vetor
    merge_sort(vetor, meio + 1, fim);
    // Combina as duas metades ordenadas em um único vetor ordenado
    merge(vetor, inicio, meio, fim);
  }
}
```

Esse código implementa a função merge, a última dentre as funções de ordenação, parte essencial do algoritmo Merge Sort. Ela recebe um vetor, um índice de início, um índice do ponto médio e um índice do fim do vetor. Ela combina duas partes ordenadas do vetor (da posição inicio até meio e de meio + 1 até fim) para formar uma única parte ordenada. O algoritmo funciona da seguinte forma: Cria um vetor auxiliar do tamanho apropriado, copia elementos do vetor original para o vetor auxiliar, mantendo a ordem crescente; compara os elementos das duas partes do vetor e os intercala em ordem crescente no vetor auxiliar; e copia os elementos ordenados do vetor auxiliar de volta para o vetor original. No final, o vetor estará ordenado.

```
879  // Função que copia o vetor global para um vetor local
880  int* copiar_vetor(int vetor_global[], int tam) {
881    int* vetor_local = (int*) malloc(tam * sizeof(int)); //vetor local
882    int i; //variável de controle do laço
883
884    //copiando cada elemento do vetor global para o vetor local
885    for (i = 0; i < tam; i++) {
886      vetor_local[i] = vetor_global[i];
887    }
888
889      //retorna a referência do vetor local
890      return vetor_local;
891  }
892
893  int main() {
894      // Executa o menu principal
895      menu_principal();
896
897      return 0;
```

Por último, temos uma função que recebe um vetor global e seu tamanho como argumentos e cria um vetor local alocado dinamicamente na memória, copiando o conteúdo do vetor global para esse vetor local. Por fim, temos a função main, que apenas chama uma função menu_principal(), visando iniciar a execução de um menu para o usuário.

Resultados

Para realizar a comparação de performance, os algoritmos foram implementados em C e executados em um computador com processador Intel Core i5-8265U 1.60 GHz e 8GB de memória RAM DDR4. Os tamanhos dos vetores usados foram de 100, 1000, 10000 e 100000 elementos. Para cada tamanho, foram usados vetores de entrada com ordem crescente e gerados aleatoriamente.

Para os vetores em ordem aleatória, os resultados da comparação de performance são apresentados na tabela abaixo:

Tempo (segundos)							
Tamanho do vetor/Algoritmo	BubbleSort	SelectSort	InsertSort	ShellSort	QuickSort	HeapSort	MergeSort
100	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
1000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
10000	0.125000	0.031000	0.078000	0.000000	0.000000	0.000000	0.000000
100000	19,781000	7,27	7,26	0.016000	0.000000	0.000000	0.015000

Comparações							
Tamanho do vetor/Algoritmo	BubbleSort	SelectSort	InsertSort	ShellSort	QuickSort	HeapSort	MergeSort
100	4950	4950	2557	378	632	673	545
1000	499500	499500	257499	7473	11601	11788	8698
10000	49995000	49995000	24926078	149361	158099	166390	120417
100000	704982704	704982704	? (Estouro)	2923738	2002234	2163519	1536507

Movimentações							
Tamanho do vetor/Algoritmo	BubbleSort	SelectSort	InsertSort	ShellSort	QuickSort	HeapSort	MergeSort
100	2486	94	2486	378	382	576	1344
1000	242995	995	242995	7473	7282	9109	19952
10000	25078911	9991	25078911	149361	92438	124221	267232

100000	? (Estouro)	99986	? (Estouro)	2923738	110546	157486 0	3337856

Para os vetores ordenados, os resultados da comparação de performance são apresentados na tabela abaixo:

Tempo (segundos)							
Tamanho do vetor/Algoritmo	Bubble Sort	SelectSort	InsertSort	ShellSort	QuickSort	HeapSort	MergeSort
100	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
1000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
10000	0.093000	0.078000	0.047000	0.000000	0.000000	0.000000	0.000000
100000	5,50	7,50	7,62	0.000000	0.000000	0.016000	0.000000

Comparações							
Tamanho do vetor/Algoritmo	BubbleSort	SelectSort	InsertSort	ShellSort	QuickSort	HeapSort	MergeSort
100	4950	4950	4950	0	593	795	356
1000	499500	499500	499500	0	10349	12963	5044
10000	49995000	49995000	49995000	0	159359	180584	69008
100000	704982704	704982704	704982704	0	2066600	2303177	853904

Movimentações							
Tamanho do vetor/Algoritmo	Bubble Sort	SelectSort	InsertSort	ShellSort	QuickSort	HeapSort	MergeSort
100	0	0	0	0	369	640	1344
1000	0	0	0	0	5926	9708	19952
10000	0	0	0	0	87780	131956	267232
100000	0	0	0	0	1162199	1650854	3337856

Conclusão

A comparação de performance dos algoritmos de ordenação apresentados neste relatório mostrou que o Quick Sort é o algoritmo mais eficiente, seguido do Heap Sort e do Merge Sort. Os algoritmos Bubble Sort, Select Sort e Insert Sort são os menos eficientes, com desempenho crescente.

Entre as principais dificuldades encontradas na implementação do trabalho, foi a de gerar um programa que possa ser executado continuamente (ser reutilizado depois da execução de uma configuração definida pelo usuário em momento de execução), que inclui como ordenar os vetores sem perder a ordem original (para serem reutilizados). Outra dificuldade encontrada foi o estouro da variável do número de comparações para o vetor aleatório de 100.000 no insert sort e da variável do número de movimentações para o vetor 100.000 para o bubble e o insert, apesar de ter sido utilizado os modificadores "long” e “unsigned" para ambas as variáveis.