Universidade do Estado do Amazonas
Escola Superior de Tecnologia
Curso de Sistemas de Informação
Projeto e Análise de Algoritmos

Trabalho Prático 1
Avaliação empírica de desempenho

Discente:
Victor Emanuel Barros de Lima

Docente:
Marcio Palheta Piedade

Manaus - AM
2024

Sumário

1. Introdução

O objetivo desta pesquisa é realizar uma análise comparativa do desempenho dos três algoritmos de ordenação clássicos: Heap Sort, Merge Sort e Quick Sort.

Os algoritmos escolhidos foram executados em C e cada algoritmo foi testado em vetores de tamanhos diferentes, que vão de 10.000 a 100.000.000 elementos que foram gerados aleatoriamente dentro do intervalo de 0 a 65.535.

Foi realizado um experimento cronometrando o tempo de ordenação de cada algoritmo para cada tamanho de entrada para avaliar seu desempenho. Os resultados foram consolidados em tempos médios de execução e desvios padrão obtidos ao executar 50 iterações para cada configuração de entrada. Para facilitar a comparação entre os algoritmos em termos de eficiência, esses dados foram então apresentados em forma de gráfico.

Com todos os arquivos desse trabalho é fornecido os códigos fontes dos algoritmos que foram implementados, um manual para compilar e executar com apenas a linha de comando e os relatórios de análise de desempenho, que fornecem uma análise detalhada dos experimentos realizados.

2. Manual

O código foi executado e desenvolvido no subsistema Linux do Windows (Ubuntu) por meio do WSL.

2.1 Estrutura

- O código para verificação de desempenho dos algoritmos foi desenvolvido de maneira que seja preciso compilar apenas um arquivo principal (main.c).

- Os algoritmos foram separados em arquivos cabeçalhos: heap sort.h, merge sort.h e quick sort.h.

- Foi criado um arquivo auxiliar chamado vetor.h que foi usado para auxiliar nas operações relacionadas ao vetor, mais especificamente as ações de gerar um vetor aleatório com números entre 0 e 65535, e salvar esse vetor em um arquivo de texto.

2.2 Execução

Como solicitado nas especificações do trabalho, a execução do código é realizada por linha de comando. Após estar na pasta correta a compilação e a execução acontecem da seguinte forma:

- A compilação do código e por meio do compilador gcc com o comando:
    - **gcc main.c -o main**

- A execução se dá pelo comando:
    - **./main**

Após ser executado, o código irá exigir do usuário certas informações para executar o mesmo de acordo com os parâmetros do usuário. As informações a serem preenchidas pelo usuário são:

- **O tamanho do vetor**, sendo ele entre 0 e 100.000.000.
  Se o tamanho inserido pelo usuário ultrapassar esse valor, o programa irá retornar ao usuário:
    - "O tamanho inserido é muito grande. Insira um valor entre 0 e 100000000."

  Se o tamanho inserido pelo usuário for menor ou igual a 100.000.000, o programa segue para a sua próxima etapa.

- **O método de ordenação,** sendo especificado para o usuário escolher com números **1** para **Heap Sort**, **2** para **Merge Sort** e **3** para **Quick Sort.** Se algum número diferente for inserido o programa irá retornar para o usuário:
    - "Método Inválido."
    - Obs: O programa irá executar mais uma exigência antes de encerrar a execução por causa de uma inserção incorreta.

- **A visualização do vetor,** sendo exigido que o usuário escolha se deseja ver o vetor ordenado escolhendo **1 (TRUE)** se quiser visualizar ou **0 (FALSE)** se não desejar ver o vetor.
    - Obs: Se o usuário inserir um valor maior que 1, o programa irá assumir como **TRUE** e irá salvar o vetor em um arquivo de texto e dará prosseguimento na execução.

Após o usuário fazer todas as inserções exigidas, o programa finaliza e retorna para o usuário:

- O método que ordenou o vetor.
- O tempo de ordenação.
- E, se o usuário optar por sim, o arquivo em que o vetor for salvo.

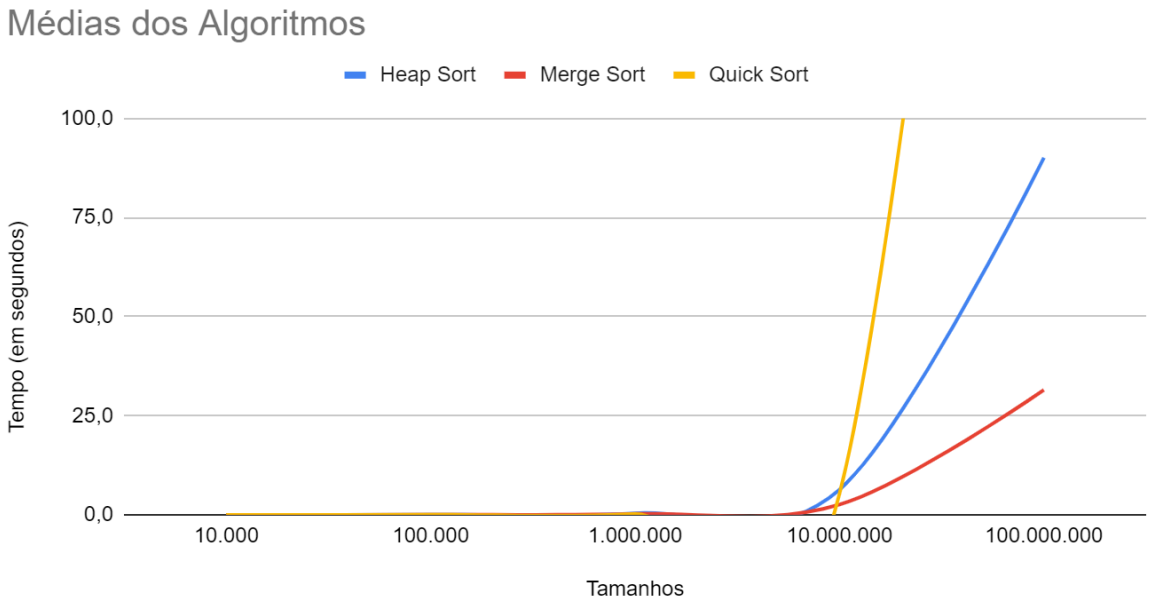
3. Desempenho

O desempenho dos algoritmos foi determinado pelo tempo de **ordenação** de cada algoritmo. Para essa análise, é gerado um vetor aleatório com números entre 0 e 65.535 e logo em seguida esse vetor é repassado para o algoritmo especificado

para ser feita a ordenação. Esse processo acontece 50 vezes e é calculada a média dos tempos e o seu desvio padrão para haver a comparação com os três algoritmos.

3.1 Médias

Médias					
	10.000	**100.000**	**1.000.000**	**10.000.000**	**100.000.000**
Heap Sort	0,003264	0,034013	0,392927	6,328661	90,072954
Merge Sort	0,003765	0,029035	0,232670	2,567733	31,490981
Quick Sort	0,002008	0,023752	0,198461	5,287809	398,584139



Como foi ilustrado acima, as diferenças entres os tempos dos vetores pôde ser de fato notada a partir dos vetores de tamanho 10.000.000. Pelo fato do custo do Heapsort e do Mergesort ser O($n \log n$); o seu crescimento é próximo e relativamente parecido possuindo uma diferença aceitável que se deve pela aleatoriedade durante a criação dos vetores. Diferentemente, o Quicksort inicialmente aparentava apresentar as mesmas características do Heap e do Merge por possuir, em certos casos, custo O($n \log n$) igual aos outros dois algoritmos em seu melhor e médio caso; porém, em seu pior caso, seu custo equivale a O(n^2), o que ocasionou sua perceptível diferença em relação aos outros algoritmos também devido a grande massa de dados dos tamanhos e a maior chance de uma aleatoriedade “ruim” de ordenar para o Quicksort.

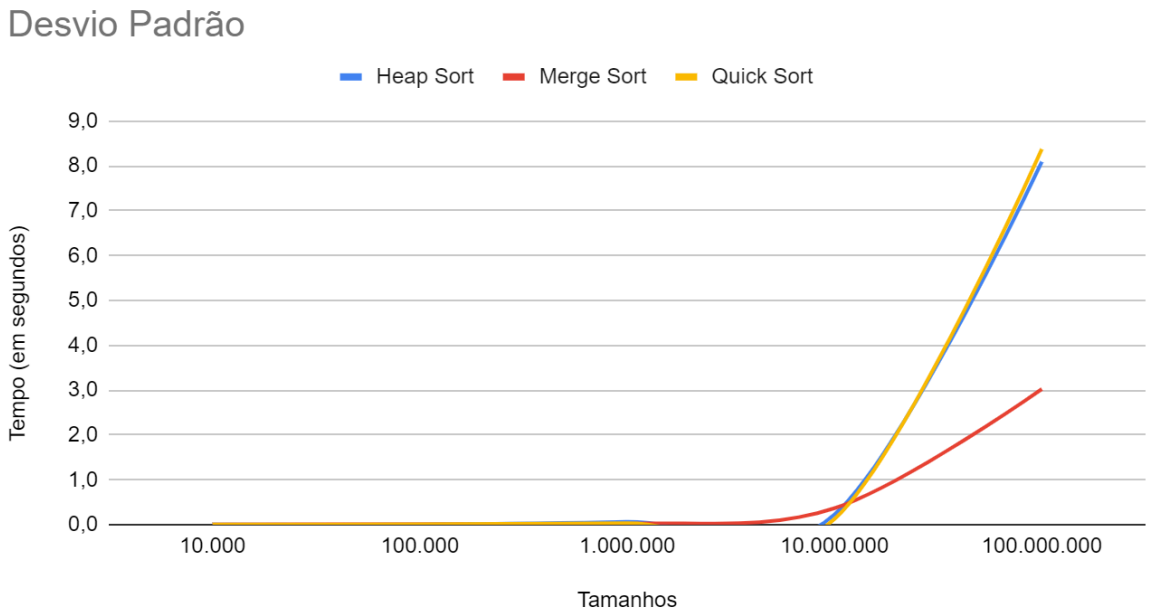
Vale ressaltar que a dificuldade que foi encontrada em executar o vetor maior de 100.000.000 com o Quicksort foi relativamente mais trabalhosa e desgastante em relação aos outros algoritmos. Isso se deve porque a maneira que acontecia a coleta de dados era feita por meio do loop demonstrado a seguir:

```
for (int i=0; i<50; i++){
    vetor_alt(&vetor, tam);
    start = clock();

    switch (alg) {
        case 1:
            nome_alg = "Heap Sort";
            heap_sort(vetor, tam);
            printf("Vetor ordenado com o Heap Sort\n");
            break;
        case 2:
            nome_alg = "Merge Sort";
            merge_sort(vetor, 0, tam - 1);
            printf("Vetor ordenado com o Merge Sort\n");
            break;
        case 3:
            nome_alg = "Quick Sort";
            quick_sort(vetor, 0, tam - 1);
            printf("Vetor ordenado com o Quick Sort\n");
            break;
        default:
            printf("Metodo Inválido\n");
            exit(1);
    }

    end = clock();
    tempo = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Tempo de ordenação: %f segundos\n", tempo);
};
```

Para os vetores menores e até mesmo o de 10.000.000, esse método de coleta foi efetivo e trouxe os resultados esperados porém, o vetor de 100.000.000 com os algoritmos do Mergesort e do Heapsort mas não com o Quicksort pois a execução acabava por ficar muito pesada para o processador e por diversas vezes matava a execução do código por não suportar a grande massa de dados. Sendo assim, na busca por otimizar o código e facilitar a coleta de dados, a duração do loop foi diminuída para 10 e foi executada 5 vezes para enfim poder calcular a média dos tempos do Quicksort. Em relação aos outros tamanhos e algoritmos não foi encontrada uma grande dificuldade na coleta de dados e definição das médias.

3.2 Desvio Padrão

Desvio Padrão					
	10.000	**100.000**	**1.000.000**	**10.000.000**	**100.000.000**
Heap Sort	0,000644	0,002868	0,064300	0,225872	8,092479
Merge Sort	0,001515	0,010393	0,018512	0,365853	3,022477
Quick Sort	0,000612	0,008869	0,035499	0,123310	8,372401



Em relação ao desvio padrão dos tempos dos algoritmos, foi relativamente simples a sua coleta pois os dados necessários para esse cálculo já estavam todos em mãos, tanto as médias quanto os tempos individuais de cada ordenação feita. Com isso, para facilitar o cálculo e otimizar a execução; foi criado um código em C que calcula o desvio padrão desde que eu fornecesse os tempos individuais e a media da seguinte maneira:

```c
int main() {
    double tempos[] = {
    0.311486, 0.278581, 0.296978, 0.243161, 0.240744, 0.263961, 0.247233, 0.234447,
    0.236258, 0.237208, 0.256106, 0.238958, 0.236734, 0.234109, 0.238669, 0.263012,
    0.239280, 0.239510, 0.237067, 0.233102, 0.231044, 0.234342, 0.248578, 0.285808,
    0.242774, 0.260262, 0.268133, 0.243097, 0.237782, 0.242946, 0.238061, 0.239322,
    0.237573, 0.254220, 0.281228, 0.261298, 0.245006, 0.272520, 0.290742, 0.238126,
    0.244837, 0.239386, 0.238584, 0.248000, 0.269672, 0.263996, 0.242161, 0.246910,
    0.246036, 0.242370
    };

    int n = sizeof(tempos) / sizeof(tempos[0]);
    double soma = 0.0, variancia = 0.0, desvio_padrao;
    double media = 0,2508;

    // Cálculo da variância
    for (int i = 0; i < n; i++) {
        variancia += pow(tempos[i] - media, 2);
    }
    variancia /= n;

    // Cálculo do desvio padrão
    desvio_padrao = sqrt(variancia);

    // Exibir resultados
    printf("Média do tempo de ordenação: %f segundos\n", media);
    printf("Desvio padrão do tempo de ordenação: %f segundos\n", desvio_padrao);

    return 0;
}
```

Com esse código foi possível obter o desvio padrão dos algoritmos de forma simples e eficiente.

3.3 Verificação

Para fazer a verificação e determinar se os tempos estavam corretos foram feitos os seguintes levantamentos:

3.3.1 Heap Sort

- Complexidade: $O(n \log n)$
- Constante ajustada: $C \approx 2.52 \times 10^{-8}$

n	Tempo Empírico (s)	Tempo Esperado (s)
10.000	0,003264	0,003344
100.000	0,034013	0,041811
1.000.000	0,392927	0,501737

10.000.000	6,328661	5,853603
100.000.000	90,072954	66,898315

3.3.2 Merge Sort

- Complexidade: $O(n\ log\ n)$
- Constante ajustada: $C \approx 1.61 \times 10^{-8}$

n	Tempo Empírico (s)	Tempo Esperado (s)
10.000	0,003765	0,002136
100.000	0,029035	0,026702
1.000.000	0,23267	0,320423
10.000.000	2,567733	3,738265
100.000.000	31,490981	42,723033

3.3.3 Quick Sort

Melhor /Médio Caso:

- Complexidade: $O(n\ log\ n)$
- Constante ajustada: $C \approx 4.24 \times 10^{-8}$

n	Tempo Empírico (s)	Tempo Esperado (s)
10.000	0,002008	0,005636
100.000	0,023752	0,070455
1.000.000	0,198461	0,845461
10.000.000	5,287809	9,863716
100.000.000	398,584139	112,728182

Pior Caso:

- Complexidade: $O(n^2)$
- Constante ajustada: $C \approx 4.55 \times 10^{-8}$

n	Tempo Empírico (s)	Tempo Esperado (s)
10.000	0,002008	0,000455
100.000	0,023752	0,045493

1.000.000	0,198461	4,549279
10.000.000	5,287809	454,92795
100.000.000	398,584139	45492,795

Conclusão

Os resultados obtidos mostraram que cada algoritmo possui suas próprias características de desempenho, que são variáveis conforme o tamanho da entrada. O Quicksort, apesar de sua complexidade média $O(n\ log\ n)$, apresentou um desempenho parecido com os outros algoritmos. No entanto, o Quicksort pode sofrer com pior desempenho em casos de entradas já ordenadas ou quase ordenadas, onde seu comportamento degrada para $O(n^2)$.

O Heapsort e o Mergesort, com complexidade $O(n\ log\ n)$, mostrou-se eficiente em termos de utilização de memória e ordenação dos vetores de diferentes tamanhos.

Pode-se concluir que a escolha do algoritmo de ordenação mais adequado depende não apenas do tamanho dos dados a serem ordenados, mas também de outras considerações como a necessidade de estabilidade da ordenação, a quantidade de memória disponível e a natureza dos dados de entrada.