Universidade Federal do Espirito Santo

Departamento de Engenharia

Engenharia de Computação

Relatório

Aluno(a): Arthur Coelho Estevão e Milena da Silva Mantovanelli

Professora: Veruska Carretta Zamborlini

Vitória

2022

1 Introdução

Este trabalho tem como objetivo estudar diferentes tipos de algoritmos de ordenação, tais como: heapsort, quicksort, shellsort, ordenação por seleção e ordenação por inserção.

O estudo organizou-se em quatro partes, na primeira realizou-se uma breve pesquisa a cerca dos algoritmos, em seguida, foi feita a implementação de seus respectivos códigos na linguagem C. Na terceira parte, efetuou-se a testagem e a coleta dos dados gerados, e por fim, foi realizada uma análise do desempenho dos algoritmos.

2 Teoria

2.1 O que são Algoritmos

No mundo da computação, algoritmos são processos computacionais bem definidos que podem receber uma entrada, processá-la e produzir um ou mais valores como saída.
De forma geral, podemos pensar em algoritmos como uma ferramenta para resolver um problema bem definido. A definição de um problema se baseia em um conjunto de dados que se deseja processar, com suas especificidades, e o resultado que é desejado alcançar. Neste trabalho, elaboramos um código contendo um conjunto de algoritmo capaz de, receber uma base de dados de n números inteiros positivos e ordenar de forma parcial em memória principal de modo decrescente. Os métodos de ordenação serão explicados posteriormente.

2.2 Eficiência dos algoritmos

Existem atualmente diversas ferramentas que analisam a performance de programas, porém, não são muito eficientes em questão de algoritmos. A complexidade de algoritmos analisa um algoritmo ignorando qualquer implementação de linguagens específicas ou hardware.

2.3 Complexidade

Apenas comparar algoritmos levando em consideração o tempo que eles gastam para executar um arquivo, não é parâmetro para dizer que um algoritmo é melhor que outro, logo, precisamos de outras ferramentas para isso, a complexidade do algoritmo. Resumidamente, complexidade de algoritmos é uma ferramenta útil para escolher o desenvolvimento do melhor algoritmo a ser utilizado para resolver determinado problema.

2.4 Selection Sort

Nesse algoritmo, destaca-se como custo relevante, o número de elementos do vetor a ser ordenado, e considera-se o número de trocas como a operação relevante na determinação do custo. (BIGONHA)

```
1  void Selection(int* a, int n){
2      for (i = 1; i <= n-1; i++) {
3          min = i;
4          for (j = i+1; j <= n; j++)
5              if (A[j] < A[min]) min = j;
6          x=A[min]; A[min]=A[i]; A[i]=x;
7      }
8  }
```

O custo do algoritmo de ordenação por seleção, no pior caso, quando considerado somente o custo da comparação de chaves de pesquisa com o elemento do arranjo, é: (BIGONHA)

$$f(n) = \sum_{i=1}^{n-1} \cdot (n-i) = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} \tag{1}$$

Como o número de trocas é considerado como operação relevante, temos, tanto no melhor como no pior dos casos, n-1 trocas:

$$f(n) = \sum_{i=1}^{n-1} \cdot 1 = n-1 \tag{2}$$

E, temos a complexidade do algoritmo igual a $O(n^2)$, para ambos os casos.

2.5 Insertion Sort

Nesse algoritmo, a instrução destacada como relevante é a de comparação de chaves **x**<**a[j]**, que ocorre na linha (5), e o tamanho do problema é n, o número de elementos no arranjo. (BIGONHA)

```
1   void Insertion(int* a, int n){
2       int i, j, x;
3       for (int i = 2; i < n; i++) {
4           x = a[i]; j = i-1; a[0] = x;
5           while (x < a[j])
6               a[j+1] = a[j]; j--;
8           a[j+1] = x;
8       }
10  }
```

No comando **WHILE** da linha (5), a instrução de comparação de chaves **x**<**a[j]**, no pior caso, é feita uma vez para cada j no intervalo [0, i-1], portanto, o custo dessa comparação de chaves é i, para i [2, n-1]. (BIGONHA)

O corpo do comando **FOR** da linha (3) tem o custo da única operação relevante nele contida, i.e., seu custo é i, pois o custo das demais operações não precisa ser considerado. Assim, o custo do comando **FOR** é $\sum_{i=2}^{n-1} \cdot i$.

$Dadoque \sum_{i=1}^{n} \cdot i = \frac{n[n+1]}{2}$, tem-se que:

$$\sum_{i=2}^{n-1} \cdot i = \sum_{k=1}^{n+2} \cdot (k+1) = \sum_{k=1}^{n+2} \cdot k + \sum_{k=1}^{n-2} \cdot 1 \tag{3}$$

ou

$$\sum_{i=2}^{n-1} \cdot i = \frac{(n-2)[(n-2)+1]}{2} + (n-2) \tag{4}$$

Portanto,

$$\sum_{i=2}^{n-1} \cdot i = \frac{n^2}{2} - \frac{n}{2} - 1 \tag{5}$$

Assim, o custo no pior caso do algoritmo de ordenação de n inteiros pelo método da inserção é:

$$f(n) = \frac{n^2}{2} - \frac{n}{2} - 1 \tag{6}$$

E o melhor caso desse algoritmo, ocorre quando o arranjo dado já estiver ordenado, pois, a comparação **x**<**a[j]** será feita somente uma vez para cada valor de i, produzindo assim o custo: (BIGONHA)

$$f(n) = \sum_{i=2}^{n-1} \cdot i = n - 2 \tag{7}$$

E, temos a complexidade do algoritmo igual a $O(n^2)$ para o pior caso, e $O(n)$ para o melhor, que ocorre na linha (9).

2.6 Shell Sort Donald Shell

Nesse algoritmo, a instrução destacada como relevante é a de comparação **a[j−h]**>**x**, tendo um h no intervalo de [n/2, 1].
Formando a sequência de:

$$\frac{n}{2}, \frac{n}{2^2}, \frac{n}{2^3}, \ldots, \frac{n}{2^m}$$

Sendo m a quantidade de passos, na última sequência temos a sequência equivalente a 1, logo m = $\log$ n.

```
1  void shellsort(int *a, int n) {
2      int j, x;
3      int h = n;
4      while(h > 1) {
5          h = h/2;
6          for(int i = h; i < n; i++) {
7              x = a[i];
8              j = i;
9              while(a[j-h] > x) {
10                 a[j] = a[j-h];
11                 j -= h;
12                 if(j < h) break;
13             }
14             a[j] = x;
15         }
16     }
17 }
```

Quando é considerado o melhor caso, onde os elementos do vetor já estão ordenados, a complexidade do algoritmo é O(n). Já no pior caso, quando os elementos do vetor estão na ordem reversa, a melhor complexidade conhecida é O(n $\log^2$ n).

2.7 Quick Sort

Nesse algoritmo, o ponto chave é a rotina de partição, pois escolhe-se o elemento pivô e o coloca em sua posição correta.

```
1   int particao(int *a,int p,int r){
2       x = a[r];    //pivo
3           i = p - 1;
4           for(int j = p; r - 1; j++){
5               if(a[j] <= x){
6                   i++;
7                   int aux = a[i];
8                   a[i] = a[j];
9                   a[j] = aux;
10              }
11          }
12          int aux = a[i + j];
13          a[i + j] = a[r];
14          a[r] = aux;
15      return i+1;
16  }
17
18  void quicksort(int *a, int p, int r){
19      if(p < r ){
20          q = particao(a, p ,r);
21          quicksort(a, p, q-1);
22          quicksort(a, q+1, r);
23      }
24  }
```

Após particionar o vetor em dois, os segmentos são ordenados por recursão da esquerda pra direita. Quando considerado o melhor caso, quando as partições possuem sempre o mesmo tamanho, a complexidade é O(nlogn). Já no pior caso, quando as chamadas recursivas geram partições com 0 e n-1 elementos, a complexidade é O(n^2);

2.8 Heap Sort

Nesse algoritmo, a estrutura Heap pode ser vista como uma árvore binária completa. Ela deve satisfazer uma das seguintes condições:

- Heap Mínimo: todo nó deve ter valor menor ou igual que seus filhos. O menor elemento é armazenado na raiz.
- Heap Máximo: todo nó deve ter valor maior ou igual que seus filhos. O maior elemento é armazenado na raiz.

```
1  void constroiHeap(int *a, int n, int i){
2      int menor = i;
3      int l = 2*i +1;
4      int r = 2*i +2;
5      if(l < n && a[l] < a[menor])
6          menor = l;
7      if(r < n && a[r] < a[menor])
8          menor = r;
9      if(menor != i){
10     int aux = a[r];
11     a[r] = a[menor];
12     a[menor] = aux;
13     constroiHeap(a, n, menor);
14     }
15 }
16 
17 void heapsort(int *a, int n){
18     for(int i = n-1/2; i>=0; i--)
19         constroiHeap(a, n, i);
20     for(int i = n - 1; i>=0; i--){
21         int aux = a[0];
22         a[0] = a[i];
23         a[i] = aux;
24         constroiHeap(a, i ,0);
25     }
26 }
```

A Heap recebe um vetor que será representado por uma árvore binária da seguinte forma:

- Raiz da árvore: primeira posição do vetor.
- Filhos do nó na posição i: posições 2*i e 2*i +1.
- Pai do nó na posição i: posição[i/2].

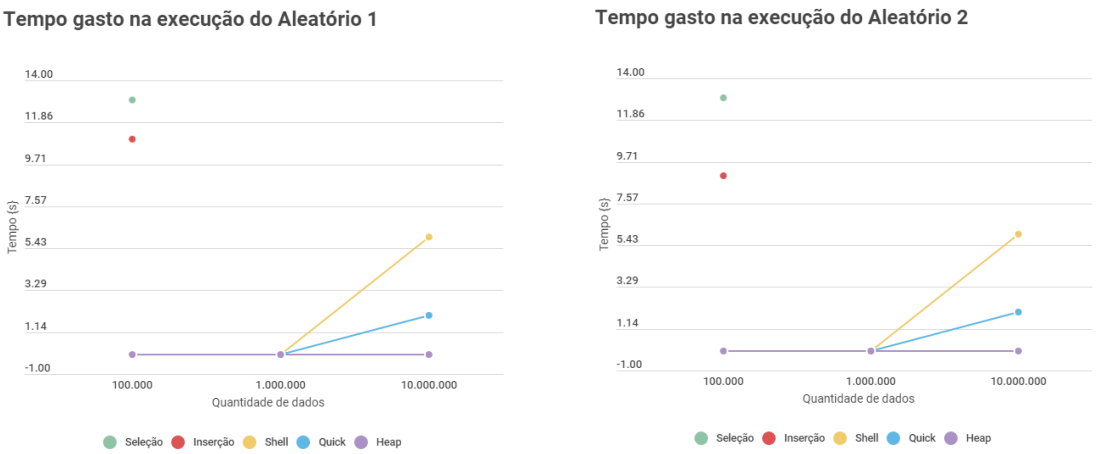
A altura da Heap de n elementos é $\log n$. A complexidade de construção da Heap é O(n), pois o número de comparações está sempre dentro do intervalo (n, 2n), já a complexidade para ajustar o elemento é de O($\log n$). Portanto a complexidade em qualquer caso do algoritmo Heap é O($n \log n$).

Os códigos acima são meramente ilustrativos e foram implementados de outra forma na execução do trabalho, podendo ser encontrados no diretório do github: `https://github.com/arthurcoelho442/Trabalho---ED-II`

3 Resultados

Para a execução do trabalho, foi elaborado um código em c contemplando os algoritmos citados anteriormente. A base de dados utilizada para teste, são arquivos com 100.000, 1.000.000 e 10.000.000, ordenados de 4 formas diferentes, sendo eles: Aleatórias, Invertidos (para analisar o pior dos casos), Ordenados (para analisar o melhor dos casos) e Quase-Ordenados (Cenário Médio). Dados, estes disponibilizados pela professora, foi elaborada uma planilha com os dados de saída, que pode ser acessada em `https://docs.google.com/spreadsheets/d/1uq3GMIXgIW6N8SkzWbBeEjcYrBjtBvVfYSyJKtV0eNA/edit?usp=sharing`.

Na Fig.1, podemos observar que tando os algoritmos de inserção e seleção não são eficientes para uma base de dados tão grande, só conseguindo executar os 100.000 primeiros dados. Já em outra análise feita, é que os algoritmos HEAP, QUICK e SHELL, conseguem executar todas os dados aleatórios, mantendo os tempos de execução aproximados até n = 1.000.000, mas no último caso de teste n = 10.000.000, o algoritmo de **HEAP** obteve o menor tempo de execução entre os três, se tornando o mais eficaz para essa base de dados.



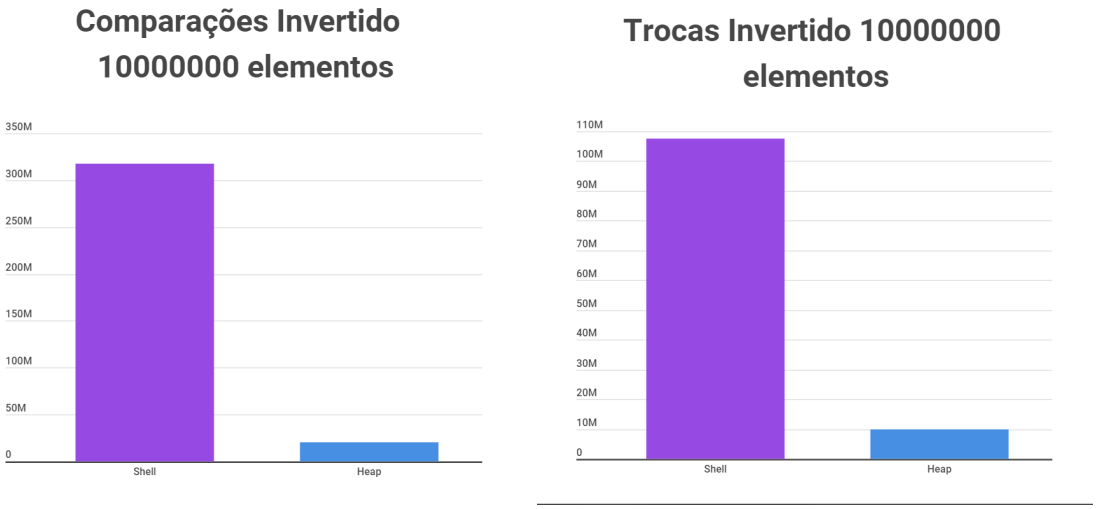
(a) Dados Aleatórios pasta 1.

(b) Dados Aleatórios pasta 2.

Figura 1: Comparação do tempo gasto referente a base de dados Aleatória

Na Fig.2, podemos analisar o desempenho dos algoritmos para uma base de dados com ordenação invertida, podendo assim descrever o pior caso de cada algoritmo utilizado. Notamos que, além do algoritmo de Inserção e Seleção, o algoritmo de Quick não consegue ordenar para dados com n>100.000, com os três tendo uma complexidade do pior caso de O(n^2). Já os outros dois algoritmos SHELL e HEAP, ambos possuem complexidade no pior caso de O($n \log n$), porém o **HEAP** obtém um melhor desempenho para n=10.000.000, executando um número menor de comparações e trocas em relação ao SHELL como podemos ver na Fig3.

Figura 2: Comparação do tempo gasto referente a base de dados Invertidos



(a) Comparações SHELL e HEAP.

(b) Trocas SHELL e HEAP.

Figura 3: Comparação e Trocas referente a base de dados invertida de 10.000.000 elementos

Na Fig.4, podemos analisar o desempenho dos algoritmos para uma base de dados ordenada, nela temos o número de trocas sempre 0 independente do algoritmo possibilitando analisar o melhor dos casos em todos eles. O tempo gasto em cada um deles dependerá exclusivamente da quantidade de comparações que eles fazem. Mesmo não aparecendo claramente no gráfico temos os resultados de tempo do HEAP e do INSERTION muito parecidos, olhando na planilhas conseguimos observar que o algoritmo de **HEAP** se torna milésimos de segundo mais eficiente do que o de inserção, mesmo executando a mesma quantidade de comparações.

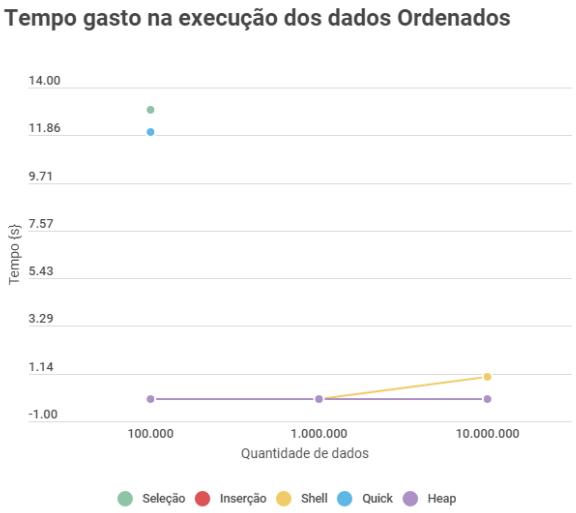


Figura 4: Comparação do tempo gasto referente a base de dados Ordenados

Para os arquivos quase ordenados, obteve-se resultados parecidos com os ordenados, mas destaca ainda mais a eficiência do algoritmo **HEAP**, pois, ele executa um número muito inferior de comparações e trocas referentes aos outros algoritmos. Essa solução se assemelha bastante como os dados do HEAP referentes aos arquivos Invertidos.

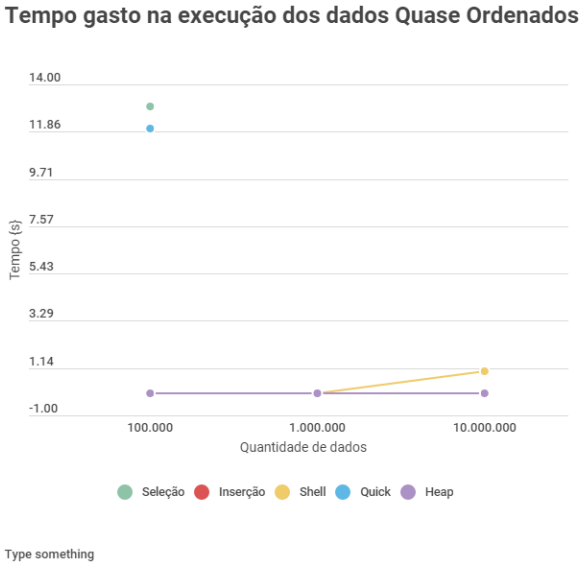


Figura 5: Comparação do tempo gasto referente a base de dados Quase-Ordenados

4 Conclusão

Conseguimos concluir que cada algoritmo possui sua particularidade e que a eficiência está insintricamente ligado a forma em que ele será utilizado. Para análise de ordenação parcial em memória principal foi possível observar dentre os algoritmos utilizados que o Heapsort, apesar de não ser um algoritmo de ordenação estável, obteve os melhores resultados tanto na execução de piores casos (Invertidos) quanto na execução dos melhores (Ordenados). Possuindo um desempenho em tempo de execução muito bom em conjuntos aleatoriamente ordenados com um uso de memória bem comportado e o seu desempenho em pior cenário (Invertidos) é praticamente igual ao desempenho em cenário médio (Quase-Ordenado).

5 Referências

BIGONHA, Roberto S. Complexidade de Algoritmos.