Universidade Federal de Mato Grosso do Sul
Facom - Faculdade de Computação
Disciplina: Algoritmos e Programação II
Professora: Liana Duenha
Alunos:
Pietro Dal Moro
Lucas Grijó
2014

Relatório de Experimentação

Trabalho Prático 1

Comparação de Métodos de Ordenação

1 Introdução

Este relatório tem o propósito de comparar a complexidade de tempo dos cinco métodos (algoritmos) básicos de ordenação, são eles: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort e Quick Sort. Cada um deles utiliza uma filosofia de ordenação diferente, portanto seu tempo de execução é distinto mesmo para uma entrada igual. Além disso devemos levar a idéia em conta o pior caso e o melhor caso de cada algoritmo, que criam situações onde o algoritmo X pode ser mais eficiente que o Y para uma entrada assim como o Y pode ser melhor do que o X em uma outra entrada.

Tais peculiaridades tornam impossível a seleção de um "melhor" algoritmo. E é por essa ideia indefinida que realizamos experimentos com os cinco métodos, testando seu tempo de execução e também adicionando uma simples variável em cada algoritmo para contar a quantidade de comparações. Entendemos por comparação toda vez que o algoritmo verifica se um valor em algum espaço do vetor é maior, igual ou menor que outro valor em outra posição desse vetor. Ao final deste relatório esperamos ter uma aproximação da realidade, através de diversos casos de teste, por trás da complexidade desses algoritmos tão usados na programação nos dias atuais.

2 Algoritmos Utilizados

- Bubble Sort

  O **bubble sort**, ou também chamado de 'ordenação por flutuação', é um dos mais simples algoritmos de ordenação.Por percorrer o vetor diversas vezes e a cada passagem fazer flutuar para o topo o maior elemento da sequência, seu movimento assemelha-se a forma como as bolhas em um tanque de água procuram seu próprio nível, sendo conhecido assim, literalmente como método 'por bolha'.

  - **Complexidade Pior Caso:** $O(n^2)$
  - **Complexidade Caso Médio:** $O(n^2)$
  - **Complexidade Melhor Caso:** $O(n)$

```
1  int bubble_sort(int n, int v[MAX]){
2
3      int i, j, temp;
4      int contador_comp = 0;
5      for(i=n-1; i>0; i--){
6
7          for(j=0;j<i;j++){
8          contador_comp++;
9              if(v[j] > v[j+1]){
10             temp = v[j+1];
11             v[j+1] = v[j];
12             v[j] = temp;
13             }
14         }
15     }
16 return contador_comp;
17 }
```

Figura 1: Função Bubble Sort com Contador

- Selection Sort

  O **selection sort**, do inglês 'ordenação por seleção', se baseia em passar sempre o menor valor do vetor para a primeira posição (ou o maior, dependendo da ordem requerida), depois o segundo menor valor para a segunda posição, realizando essa operação sucessivamente para os (n-1) elementos restantes, até os últimos dois elementos.

  - **Complexidade Pior Caso:** $O(n^2)$
  - **Complexidade Caso Médio:** $O(n^2)$
  - **Complexidade Melhor Caso:** $O(n^2)$

```
1  int selection_sort(int n, int v[MAX]){
2
3      int i, j, min, temp;
4      int contador_comp = 0;
5      for(i=0;i<n-1; i++){
6          min = i;
7          for(j = i+1;j<n; j++){
8              contador_comp++;
9                  if(v[j] < v[min]){
10                     min = j;
11                 }
12             }
13         temp = v[min];
14         v[min] = v[i];
15         v[i] = temp;
16         }
17     return contador_comp;
18 }
```

Figura 2: Função Selection Sort com Contador

- Insertion Sort

  O **insertion sort**, ou ordenação por inserção, é um simples algoritmo de ordenação bastante eficiente quando aplicado a um pequeno número de elementos. Ele percorre um vetor de elementos da esquerda para direita e à medida que avança vai deixando os elementos mais à esquerda ordenados, podendo assim ser comparado como o modo que muitas pessoas ordenam cartas como em um jogo de baralho.

  - **Complexidade Pior Caso:** $O(n^2)$
  - **Complexidade Caso Médio:** $O(n^2)$
  - **Complexidade Melhor Caso:** $O(n)$

```
1  int insertion_sort(int n, int v[MAX]){
2
3      int i, j, x, z;
4      int contador_comp = 0;
5      for(i=1;i<n;i++){
6          x = v[i];
7          for(z=i-1;z>=0;z--, contador_comp++){}
8          for(j=i-1;j>=0 && v[j] > x; j--){
9              v[j+1] = v[j];
10         }
11         v[j+1] = x;
12     }
13 return contador_comp;
14 }
```

Figura 3: Função Insertion Sort com Contador

- Merge Sort

  O **merge sort**, ou ordenação por inserção, é um simples algoritmo de ordenação bastante eficiente quando aplicado a um pequeno número de elementos. Ele percorre um vetor de elementos da esquerda para direita e à medida que avança vai deixando os elementos mais à esquerda ordenados, podendo assim ser comparado como o modo que muitas pessoas ordenam cartas como em um jogo de baralho.

  - **Complexidade Pior Caso:** $O(n \log n)$
  - **Complexidade Caso Médio:** $O(n \log n)$
  - **Complexidade Melhor Caso:** $O(n \log n)$

```
1  void intercala(int p, int q, int r, int v[MAX]){
2      int i, j, k, w[MAX];
3      i = p;
4      j = q;
5      k = 0;
6      while (i<q && j<r){
7          contador_comp++;
8          if(v[i] < v[j]){
9              w[k] = v[i];
10             i++;
11         }
12         else{
13             w[k] = v[j];
14             j++;
15         }
16         k++;
17     }
18     while(i < q){
19         w[k] = v[i];
20         i++;
21         k++;
22     }
23     while(j < r){
24         w[k] = v[j];
25         j++;
26         k++;
27     }
28     for(i=p; i<r; i++){
29         v[i] = w[i-p];
30     }
31 }
```

Figura 4: Função Intercala (Merge Sort) com Contador

```
void mergesort(int p, int r, int v[MAX]){
    int q;
    if(p < r - 1){
        q = (p + r) /2;
        mergesort(p, q, v);
        mergesort(q, r, v);
        intercala(p, q, r, v);
    }
}
```

Figura 5: Função Merge Sort

- Quick Sort

  O **quick sort** é um método de ordenação muito rápido e eficiente. É frequentemente usado na prática para ordenação já que é rápido "na média"e apenas para algumas entradas especiais o método é lento como os métodos elementares de ordenação.

  - **Complexidade Pior Caso:** $O(n^2)$
  - **Complexidade Caso Médio:** $O(n \log n)$
  - **Complexidade Melhor Caso:** $O(n \log n)$

```
int separa(int p, int r, int v[MAX]){

    int x, i, j;
    x = v[p];
    i = p - 1;
    j = r + 1;
    while (1) {
        do {
            j--;
        } while(v[j] > x);
        do {
            i++;
        } while(v[i] < x);
        if(i<j){
            contador_comp++;
            troca(&v[i], &v[j]);
        }
        else{
            contador_comp++;
            return j;
        }
    }
}
```

Figura 6: Função Separa (Quick Sort) com Contador

```
1 void troca(int *x, int *y){
2     int temp;
3     temp = *y;
4     *y = *x;
5     *x = temp;
6 }
```

Figura 7: Função Troca (Quick Sort)

```
1 void quicksort(int p, int r, int v[MAX]){
2
3     int q;
4     if(p < r){
5         q = separa(p,r,v);
6         quicksort(p, q, v);
7         quicksort(q+1, r, v);
8     }
9 }
```

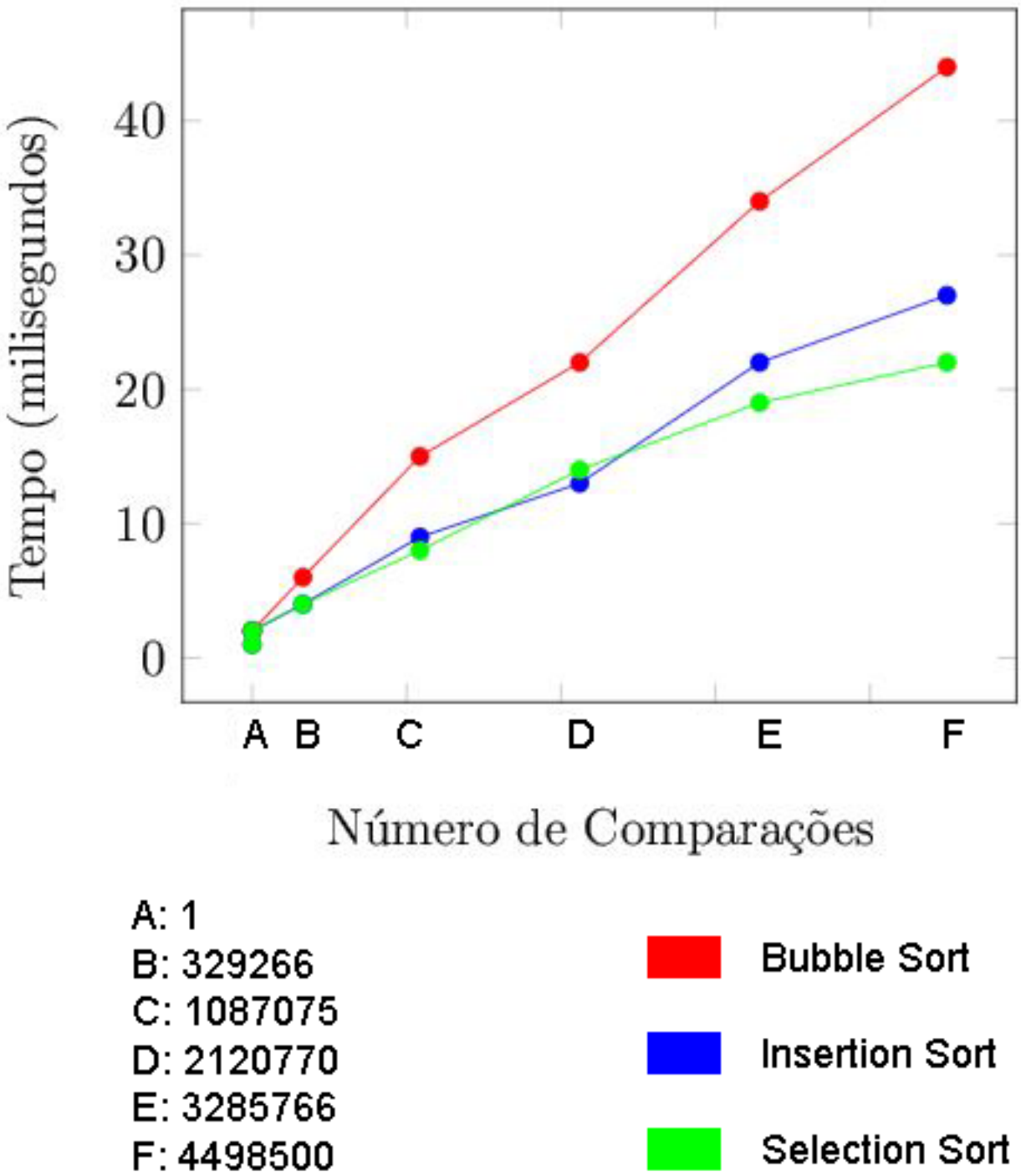
Figura 8: Função Troca (Quick Sort)

3 Experimentação

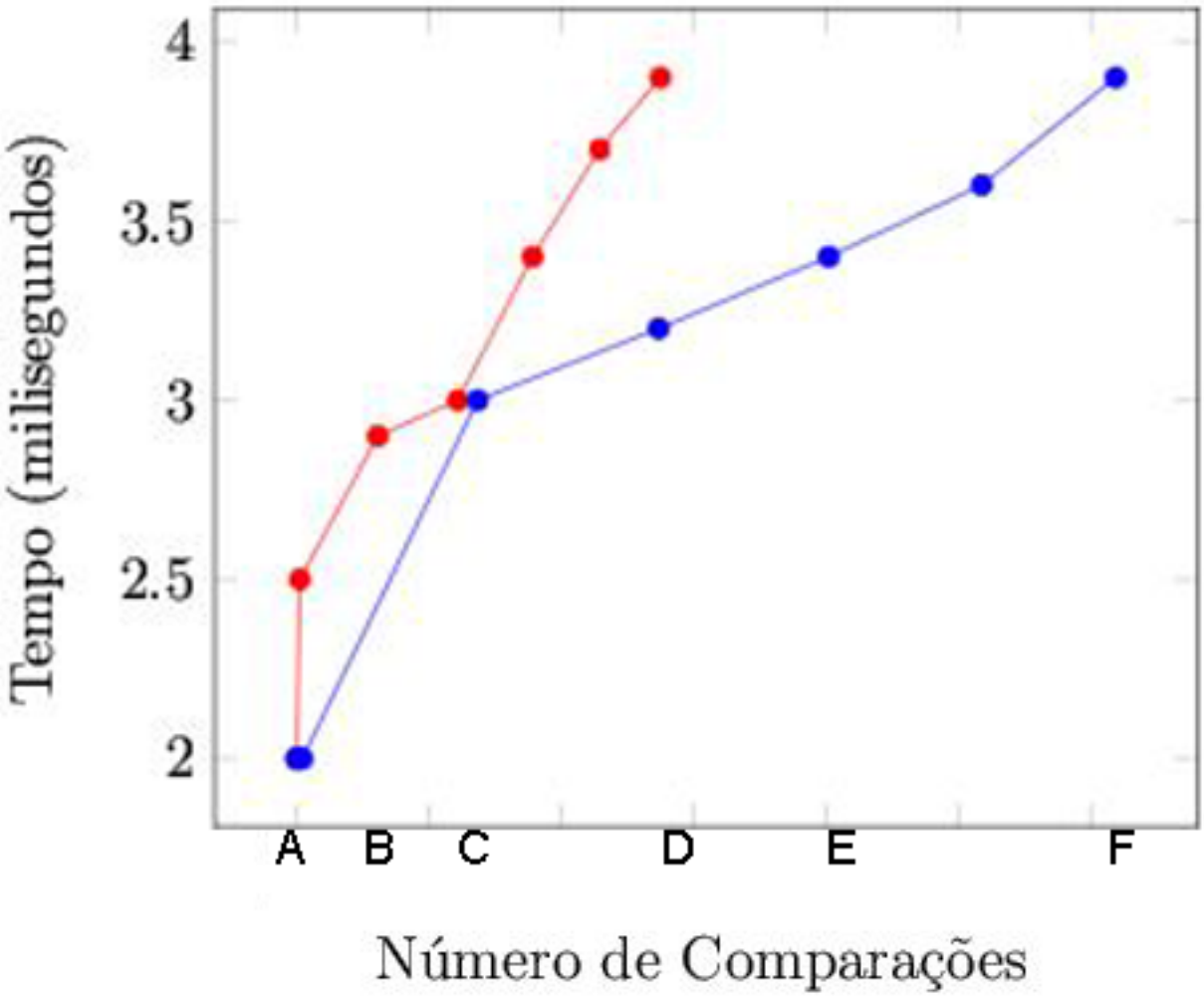
Para gerar os gráficos fizemos uma bateria de testes usando vetores cujo tamanho varia de 0 até 3000 elementos. E usamos 7 testes diferentes cada um com um vetor nos cinco algoritmos de ordenação. Os dados especificicos de cada teste podem ser vistos no arquivo "Gráficos.txt"que está no mesmo diretório deste relatório.

O método que utilizamos para verificar o tempo de execução dos algoritmos é similar ao modo de uso do arquivo gerador.c. Nós criamos 7 arquivos de entrada, e executamos os algoritmos com o seguinte comando no terminal (time ./algoritmo ¡ teste) que nos provia com o tempo de execução total do algoritmo. Depois de ter todos os dados necessários para gerar o gráfico utilizamos os próprios comandos do LaTeX para gerar os gráficos. Escolhemos separar os cinco algoritmos em 2 gráficos diferentes: um para o bubble, insertion e selection sort e outro para o quick e merge sort. Devido a similaridade do numero de comparações nos 3 primeiros algoritmos citados.

- Gráfico 1


Tempo (milisegundos)
0
10
20
30
40
A
B
C
D
E
F
Número de Comparações
A: 1
B: 329266
C: 1087075
D: 2120770
E: 3285766
F: 4498500
Bubble Sort
Insertion Sort
Selection Sort

- Gráfico 2



A: 1
B: 3070
C: 6840
D: 13740
E: 20103
F: 30936

Quick Sort

Merge Sort