Instituto de Matemática e Estatística da Universidade de São Paulo (IME-USP)
Patrícia da Silva Rodrigues

Análise de Algoritmos

São Paulo
2020

Os algoritmos de ordenação foram criados a fim de colocar os elementos em uma dada sequência, como, por exemplo, ordem crescente ou decrescente. A ordenação pode ser feita em matrizes até a n-ésima ordem e de dimensões diversas. Pode-se ordenar números, caracteres, etc. Os algoritmos de ordenação analisados serão, respectivamente: Bubble Sort, Insertion Sort e Quick Sort. Nesse trabalho utilizaremos números pseudoaleatórios gerados pela função rand() e será verificado a quantidade de comparações feitas por cada algoritmo para efetuar a ordenação dos números por ordem crescente. Os números gerados pelos algoritmos pertencem ao intervalo [0,100] ∈ N. As observações buscarão analisar quantas comparações serão feitas por cada um para ordenar sequencias cada vez maiores de números pseudoaleatórios. A quantidade de número se iniciará em 250 dobrará de tamanho até o limite que o sistema processou. A quantidade de comparações feitas por cada algoritmo em cada nova iteração com diferentes quantidades de números pseudoaleatórios estará em uma tabela e no gráfico, para o reconhecimento de um possível padrão e análise posterior. O código usado para gerar cada uma das tabelas e gráficos estão abaixo:

Bubble-Sort

Código:

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

void main()
{

    int i = 0;
    int cont = 0;
    int iteracoes = 0;
    printf("Quantas iterações deseja fazer?: ");
    scanf("%d", &iteracoes);
    int resp = 0;
    int *comp;
    comp = malloc(iteracoes * sizeof(int));
    int tamanhoVetor = 0;
```

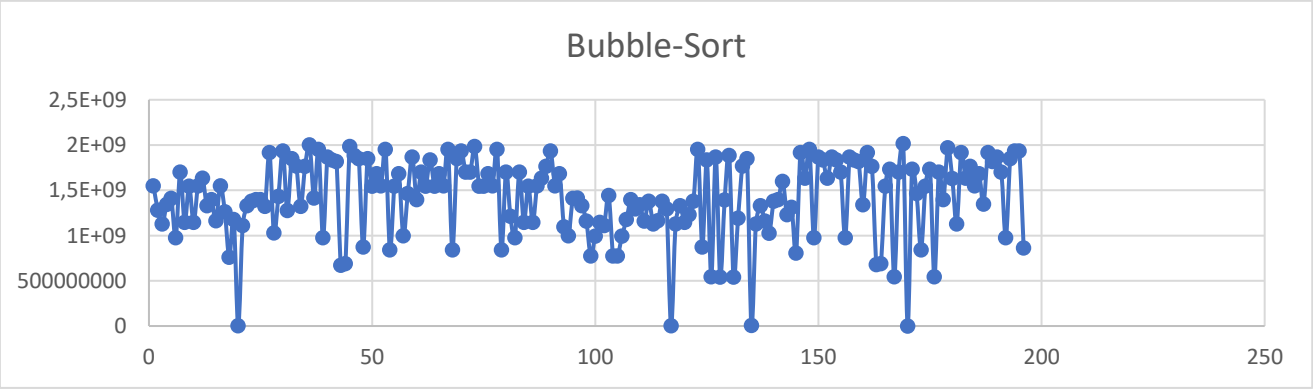
```
        for(i = 0; i < iteracoes; i++)
        {

            tamanhoVetor = tamanhoVetor + 250;
            int *vetor;

            vetor = malloc(tamanhoVetor * (sizeof(int))); //cria um vetor de
tamanho tamanhoVetor

                for(i = 0; i < tamanhoVetor; i++)
                {
                        vetor[i] = rand() % 100 + 1; // preenche o vetor vetor
com numeros aleatórios entre 0 e 100
                }




                resp = bubble_sort(vetor, tamanhoVetor, comp);

                comp[i] = resp;

                free(vetor);
        }

        for(i = 0; i < iteracoes; i++)
        {
                printf("%d\n", comp[i]);
        }
}
```

```
int bubble_sort (int vetor[], int n, int comp)
{
        int comparando = comp;
   int k, j, aux;



   for (k = 1; k < n; k++) {
        comp++;
      printf("\n[%d] ", k);

      for (j = 0; j < n - 1; j++) {
        comp++;
         printf("%d, ", j);

         if (vetor[j] > vetor[j + 1]) {
               comp++;
            aux          = vetor[j];
            vetor[j]     = vetor[j + 1];
            vetor[j + 1] = aux;
         }
      }
   }
   return comparando;
}
```

Tabela de números Pseudoaleatórios:

1551069797	1279607900	1702130553	1551132271	1230261829	1835102823	1128097908
1280066884	1850694732	1546793837	1684957527	1380272454	1635017028	1917869114
1129270272	1769096293	1835360855	1098086255	1953384765	1869762560	1634887535
1346456641	1325426038	1547322171	997421168	875981925	1835102823	1766203501
1413563472	1767325011	1684957527	1413566464	1835091488	1701603654	1551066476
977485121	2003788910	1551071087	1415071048	544828521	977485171	1684957527
1702057308	1414422387	1953724755	1329802813	1867325494	1869762652	1349744495
1146909554	1952534528	842231141	1160657741	543974756	1835102823	1919252335
1548504133	977485160	1852397404	775636312	1394620726	1818838560	1818585171
1148219457	1869762652	1937207140	995377474	1886414196	1342206821	1867340908
1549890657	1835102823	1702326096	1145914158	543649385	1919381362	1701606756
1633906508	1818838560	1701335922	1112944187	1193290801	1766223201	977484659
1330380908	673215333	1985768556	1445870419	1769303653	678651244	1852397404
1397641031	691419256	1546661425	775636290	1850303854	691419256	1937207140
1163285061	1986348124	1547322171	775639882	7103860	1547322173	1937339228
1549548882	1886405421	1684957527	994399050	1129271888	1735357008	862807412
1263748420	1852394844	1551071087	1179866926	1330860869	544039282	
760237908	875976519	1953724755	1398222395	1162633042	1701603654	
1177767238	1852400220	842231141	1294875464	1028408662	2015895667	
4737620	1547322171	1701859164	1342194515	1380974646	2700856	
1112364366	1684957527	1213420398	1162039122	1397048143	1735357008	
1331647045	1551071087	977484636	1380930387	1599229779	1466786162	
1380998982	1953724787	1702057308	1129464159	1230390610	842216502	
1397048143	842231141	1146909554	1163151688	1313818963	1547322173	
1397903187	1547322171	1548504133	1381323843	808793149	1735357008	
1325413437	1684957527	1148219457	1296121157	1917845553	544039282	
1917085038	997422959	1549890657	3421764	1634887535	1701603654	
1030059625	1465662019	1633906508	1129271888	1952531565	1397751923	
1432107587	1868852841	1766677612	1330860869	977485153	1969516365	
1936876915	1398567799	1936683619	1145659218	1869762652	1632658796	

Gráfico:



Análise:

A quantidade de comparações feitas no algoritmo Bubble Sort se mantém muito pouco próximo da média de 1421356948, com um grande desvio padrão, o que implica a conclusão de que a quantidade de comparações feitas pelo algoritmo não está relacionada proporcionalmente com a quantidade de elementos que ele manipula, caso esses elementos números pseudoaleatórios. Em matéria de processamento, isso pode significar que ele não afetaria necessariamente o desempenho de um sistema de ordenação pela quantidade de elementos, caso ela seja relativamente grande. A quantidade de elementos manipulados pelo algoritmo não está relacionada necessariamente com a performasse dele, caso esses elementos sejam números pseudoaleatórios. Isso é algo que pode ser deduzido pela sua natureza. Fazemos isso quando analisamos o melhor e o pior caso.

O melhor caso do algoritmo Bubble sort é encontrar uma sequência de elementos já ordenados, pois o algoritmo irá percorrer a lista apenas uma única vez, e, uma vez que o tempo de processamento é um fator de qualidade, ele terá o melhor desempenho possível. Sua analise assintótica é O(n^2)

O pior caso será quando os menores elementos se encontrarem no final da sequencia analisada, visto que o algoritmo irá necessariamente ter que realizar muitas iterações para que a sequência seja ordenada.

No caso, os números pseudoaleatórios não estão organizados nem em ordem crescente, nem decrescentes e é por esse motivo que podemos observar uma grande variação por todo o gráfico, visto que os números também variam de tamanho por toda a tabela.

Insertion Sort

Código:

```
#include <stdio.h>

void main()

{

   int iteracoes, i, temp;

   int *vetor;

   int tamanhoVetor = 0;

   int comp = 0;

            int *compara;

   printf("Digite a quantidade de iterações: ");

   scanf("%d", &iteracoes);


            compara = malloc(iteracoes * sizeof(int));
```

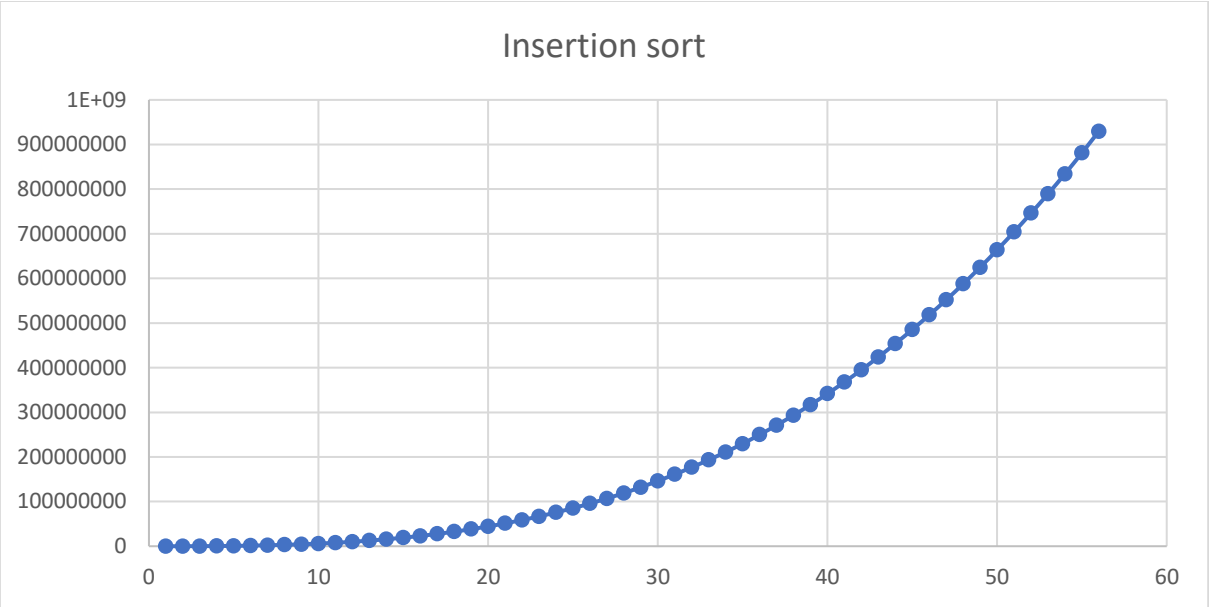
```
int k = 0;
for(k = 0; k < iteracoes; k++)
{
	tamanhoVetor = tamanhoVetor + 250;
	vetor = malloc(tamanhoVetor * sizeof(int)); //cria um vetor de tamanho tamanhoVetor
		int j = 0;
		for(j = 0; j < tamanhoVetor; j++)
		{
			vetor[j] = rand() % 100 + 1;
			 // preenche o vetor vetor com numeros aleatórios entre 0 e 100
		}

	j = 0;

   for (i = 1 ; i <= tamanhoVetor - 1; i++)
   {
	comp++;
	   j = i;

	 while ( j > 0 && vetor[j-1] > vetor[j])
	 {
				comp++;
	    temp     = vetor[j];
	    vetor[j]   = vetor[j-1];
	    vetor[j-1] = temp;
	    j--;

	 }
   }
```

```
            compara[i] = comp;

            for (i = 0; i <= tamanhoVetor - 1; i++)
            {

            }

            printf("%d\n", compara[i]);
        }

}
```

Tabela de números pseudosaleatórios:

15472	1,61E+08
78705	1,77E+08
208339	1,93E+08
458203	2,11E+08
845148	2,3E+08
1393185	2,5E+08
2169690	2,71E+08
3153165	2,94E+08
4410593	3,17E+08
5946560	3,42E+08
7782894	3,68E+08
9998954	3,95E+08
12665507	4,24E+08
15688228	4,54E+08
19186793	4,85E+08
23128273	5,18E+08
27619896	5,52E+08
32641792	5,88E+08
38275641	6,25E+08
44433796	6,64E+08
51251948	7,04E+08
58776745	7,46E+08
66853508	7,89E+08
75812620	8,34E+08
85352280	8,81E+08
95809110	9,3E+08
107037101	
119179101	
132177494	
146288823	

Gráfico:



Análise:

Como podemos ver, o algoritmo de inserção é estável. A quantidade de números pseudoaleatórios afetou a quantidade de comparações feitas pelo algoritmo, mas o fato de serem números pseudoaleatórios não afetou, como no caso do Bubble sort. Isso se deu pelo fato de que ele é afetado pela quantidade de elementos a serem ordenados e não pela sua ordem.

O algoritmo de inserção é considerado um algoritmo com uma performasse ruim para grande quantidade de elementos a serem manipulados, pois ele precisa percorrer a lista para cada elemento. Sua análise assintótica será ($O(n^2)$)).

Seu uso é adequado para quando novos números são inseridos na sequencia já ordenada, pois ele percorrerá a lista apenas uma vez para descobrir a posição de o elemento deve ficar.

Quick Sort

Código:

```
#include<stdio.h>

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

int particao (int vetor[], int comeco, int fim)
{
                int comp = 0;
    int pivo = vetor[fim];    //pivo
    int i = (comeco - 1);  // indice do menor elemento
                int j = 0;
    for( j = comeco; j <= fim- 1; j++)
    {
                comp++;
        // se o elemento atual for menor q o pivo
        if (vetor[j] < pivo)
        {
        comp++;
            i++;
            troca(&vetor[i], &vetor[j]);
        }
    }
    troca(&vetor[i + 1], &vetor[fim]);
    return ((i + 1), comp);
```

```
}

int quickSort(int vetor[], int comeco, int fim)
{
		int comp = 0;
   if (comeco < fim)
   {
		int aux;
		comp++;
      int pi;
      pi, aux = particao(vetor, comeco, fim);
		 comp = comp + aux;
      //classifica os elementos de antes da particao e de depois
      quickSort(vetor, comeco, pi - 1);
      quickSort(vetor, pi + 1, fim);
   }
   return comp;
}


void main()
{
		int i = 0;
		 int cont = 0;
		 int iteracoes = 0;
		 printf("Quantas iterações deseja fazer?: ");
		 scanf("%d", &iteracoes);
		 int resp = 0;
		 int *comp;
		 comp = malloc(iteracoes * sizeof(int));
```

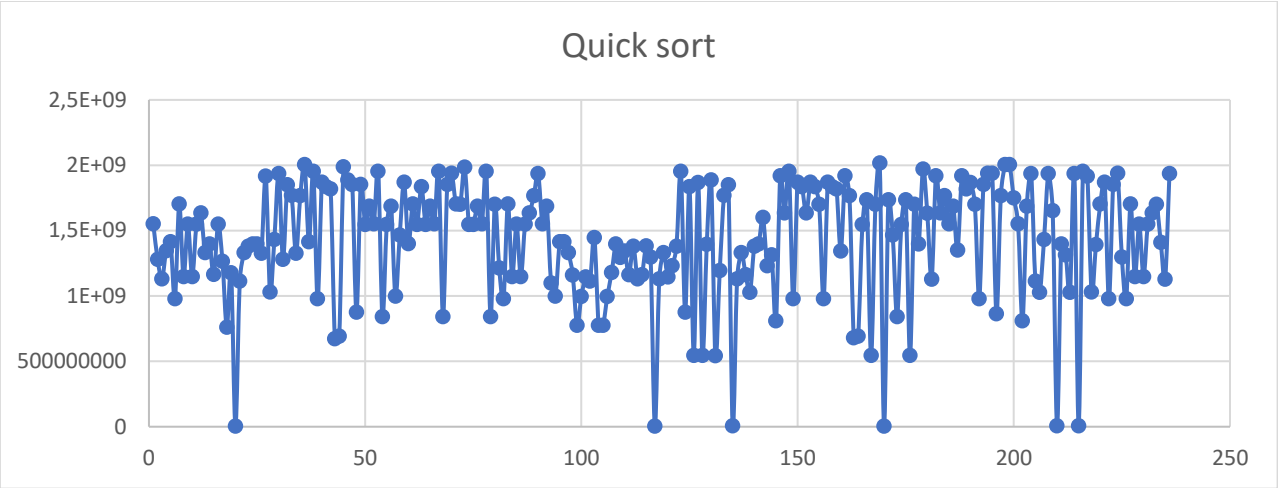
```
        int tamanhoVetor = 0;
        int compara = 0;

            for(i = 0; i < iteracoes; i++)
            {
            tamanhoVetor = tamanhoVetor + 250;
          int *vetor;

            vetor = malloc(tamanhoVetor * (sizeof(int))); //cria um vetor de
tamanho tamanhoVetor
                    for(i = 0; i < tamanhoVetor; i++)
                    {
                        vetor[i] = rand() % 100 + 1; // preenche o vetor vetor
com numeros aleatórios entre 0 e 100
                    }


                    int n = sizeof(vetor)/sizeof(vetor[0]);
                    compara = quickSort(vetor, 0, n-1);

                    comp[i] = compara;

                    free(vetor);
            }


            for(i = 0; i < iteracoes; i++)
            {
                    printf("%d\n", comp[i]);
            }
    }
```

Tabela com números pseudoaleatórios:

1551069797	1,28E+09	1,7E+09	1,55E+09	1,23E+09	1,84E+09	1,13E+09	1,4E+09
1280066884	1,85E+09	1,55E+09	1,68E+09	1,38E+09	1,64E+09	1,92E+09	1,31E+09
1129270272	1,77E+09	1,84E+09	1,1E+09	1,95E+09	1,87E+09	1,63E+09	1,03E+09
1346456641	1,33E+09	1,55E+09	9,97E+08	8,76E+08	1,84E+09	1,77E+09	1,94E+09
1413563472	1,77E+09	1,68E+09	1,41E+09	1,84E+09	1,7E+09	1,55E+09	6646895
977485121	2E+09	1,55E+09	1,42E+09	5,45E+08	9,77E+08	1,68E+09	1,95E+09
1702057308	1,41E+09	1,95E+09	1,33E+09	1,87E+09	1,87E+09	1,35E+09	1,92E+09
1146909554	1,95E+09	8,42E+08	1,16E+09	5,44E+08	1,84E+09	1,92E+09	1,03E+09
1548504133	9,77E+08	1,85E+09	7,76E+08	1,39E+09	1,82E+09	1,82E+09	1,39E+09
1148219457	1,87E+09	1,94E+09	9,95E+08	1,89E+09	1,34E+09	1,87E+09	1,7E+09
1549890657	1,84E+09	1,7E+09	1,15E+09	5,44E+08	1,92E+09	1,7E+09	1,87E+09
1633906508	1,82E+09	1,7E+09	1,11E+09	1,19E+09	1,77E+09	9,77E+08	9,77E+08
1330380908	6,73E+08	1,99E+09	1,45E+09	1,77E+09	6,79E+08	1,85E+09	1,85E+09
1397641031	6,91E+08	1,55E+09	7,76E+08	1,85E+09	6,91E+08	1,94E+09	1,94E+09
1163285061	1,99E+09	1,55E+09	7,76E+08	7103860	1,55E+09	1,94E+09	1,3E+09
1549548882	1,89E+09	1,68E+09	9,94E+08	1,13E+09	1,74E+09	8,63E+08	9,77E+08
1263748420	1,85E+09	1,55E+09	1,18E+09	1,33E+09	5,44E+08	1,77E+09	1,7E+09
760237908	8,76E+08	1,95E+09	1,4E+09	1,16E+09	1,7E+09	2E+09	1,15E+09
1177767238	1,85E+09	8,42E+08	1,29E+09	1,03E+09	2,02E+09	2E+09	1,55E+09
4737620	1,55E+09	1,7E+09	1,34E+09	1,38E+09	2700856	1,75E+09	1,15E+09
1112364366	1,68E+09	1,21E+09	1,16E+09	1,4E+09	1,74E+09	1,55E+09	1,55E+09
1331647045	1,55E+09	9,77E+08	1,38E+09	1,6E+09	1,47E+09	8,08E+08	1,63E+09
1380998982	1,95E+09	1,7E+09	1,13E+09	1,23E+09	8,42E+08	1,69E+09	1,7E+09

1397048143	8,42E+08	1,15E+09	1,16E+09	1,31E+09	1,55E+09	1,94E+09	1,41E+09
1397903187	1,55E+09	1,55E+09	1,38E+09	8,09E+08	1,74E+09	1,11E+09	1,13E+09
1325413437	1,68E+09	1,15E+09	1,3E+09	1,92E+09	5,44E+08	1,03E+09	1,93E+09
1917085038	9,97E+08	1,55E+09	3421764	1,63E+09	1,7E+09	1,43E+09	
1030059625	1,47E+09	1,63E+09	1,13E+09	1,95E+09	1,4E+09	1,94E+09	
1432107587	1,87E+09	1,77E+09	1,33E+09	9,77E+08	1,97E+09	1,65E+09	
1936876915	1,4E+09	1,94E+09	1,15E+09	1,87E+09	1,63E+09	6515052	

Gráfico:



Análise:

Podemos ver que o Quick sort não é estável, assim como no bubble sort, aqui a ordem dos elementos não afeta necessariamente a quantidade de comparações que serão feitas. Podemos notar que existe a quantidade de comparações se mantém pouco próxima da média, o que caracteriza sua instabilidade, o desvio padrão é grande.

O pior caso do quick sort é quando o pivô divide a lista de forma desbalanceada, uma com tamanho zero e outra com tamanho n – 1. Isso ocorre quando o pivô é o maior ou o menor elemento da lista, quando a lista já está ordenada oou inversamente ordenada. Sua complexidade no tempo é O(nlogn)O(nlog⁡n) no melhor caso e no caso médio e O(n2)O(n2) no pior caso.

Merge Sort

O merge sort é um algoritmo de intercalação recursivo e estável na maioria das implementações.

Opera da seguinte forma: divide o arranjo em duas sequências de n/2 elementos, classifica as duas sequências recursivamente, utilizando a própria ordenação por intercalação e faz a intercalação das duas sequências ordenadas, de modo a produzir a resposta ordenada. Esse método é conhecido como dividir e ordenar. Seu tempo de execução é proporcional a função f(n) = n, algo que permite a dedução de que ele é constante, independetemente do tamanho da sequência.

A vantagem é que ele é mais eficiente que os algoritmos bubble sort e insertion sort, mas a desvantagem é que ele tem gasto extra de memória, visto que ele cria uma cópia da lista para cada nível da chamada recursiva, tendo um adicional de memória de O(n log n)