Universidade Federal de Juiz de Fora

Instituto de Ciências Exatas

Departamento de Ciência da Computação

DCC001
ANÁLISE E PROJETO DE ALGORITMOS
Trabalho Prático

Rafael Terra de Souza
Mateus Coutinho Marim
Aleksander Yacovenco
Mattheus Soares Santos

Professor - Stênio Soares

Juiz de Fora - MG
24 de abril de 2017

Sumário

Lista de Figuras

Lista de Programas

Lista de Tabelas

1 Introdução

O objetivo deste trabalho é a implementação e análise de algoritmos de ordenação, dentre os quais foram utilizados: BubbleSort, SelectionSort, InsertionSort, MergeSort, QuickSort e HeapSort. Foram analisados o número de comparações realizadas, o número de atribuições realizadas e o tempo de execução para cada um dos algoritmos citados anteriormente.

1.1 Considerações iniciais

- Ambiente de desenvolvimento do código fonte: CLion, Atom+terminal e Gedit+terminal.
- Linguagem utilizada: Linguagem C/C++.
- Ambiente de desenvolvimento da documentação: TexStudio, editor de latex para Texlive.

1.2 Especifição do problema

A partir de vetores de dados passados, utilizar os algoritmos BubbleSort, SelectionSort, InsertionSort, MergeSort, QuickSort e HeapSort para ordenar os dados dos vetores, calcular o tempo de execução de cada um desses algoritmos, assim como quantas atribuições e quantas comparações são feitas por cada um dos algoritmos citados anteriormente. Após tais dados coletados, comparar os algoritmos não eficientes (BubbleSort, InsertionSort e SelectionSort) e eficientes (QuickSort, HeapSort e MergeSort), apontando a análise de complexidade de cada um dos algoritmos e quais foram os testes realizados com os vetores passados.

2 Algoritmo e estruturas de dados

Estrutura de dados utilizada:

As instâncias utilizadas são armazenadas em uma struct contendo uma chave to tipo inteiro e uma string que armazena o nome de cada elemento.

2.1 Estruturas gerais

```
typedef long long int lli;

extern lli comp;
extern lli atrib;

struct Node{
  int key;
  std::string info;
};
```

Algoritmo 1: Struct

Função swap, utilizada para trocar realizar a troca de dois elementos do vetor.

```
void swap(Node v[], int a, int b){
  Node aux = v[a];
  v[a] = v[b];
  v[b] = aux;
  atrib += 3;
}
```

Algoritmo 2: Swap

2.2 Algoritmos de ordenação

```
void bubble_sort(Node v[], int n){
  int i, j, swaps;

  for(i = 0; ; ++i){
    swaps = 0;
    for(j = 0; j < n-1; ++j){
      comp++;
      if(v[j].key > v[j + 1].key){
        swap(v, j + 1, j);
        swaps++;
      }
    }
    if(!swaps) return;
  }
}
```

Algoritmo 3: BubbleSort

```
void insertion_sort(Node v[], int n){
  int i, j;
  Node temp;
  for(i = 1;i <= n-1;i++){
        temp = v[i];
        j = i-1;
  comp++;
        while((j >= 0) && (temp.key < v[j].key)){
            v[j+1] = v[j];
            j = j-1;
            comp++;
            atrib += 2;
        }
        v[j+1] = temp;
        atrib += 3;
    }
}
```

Algoritmo 4: InsertionSort

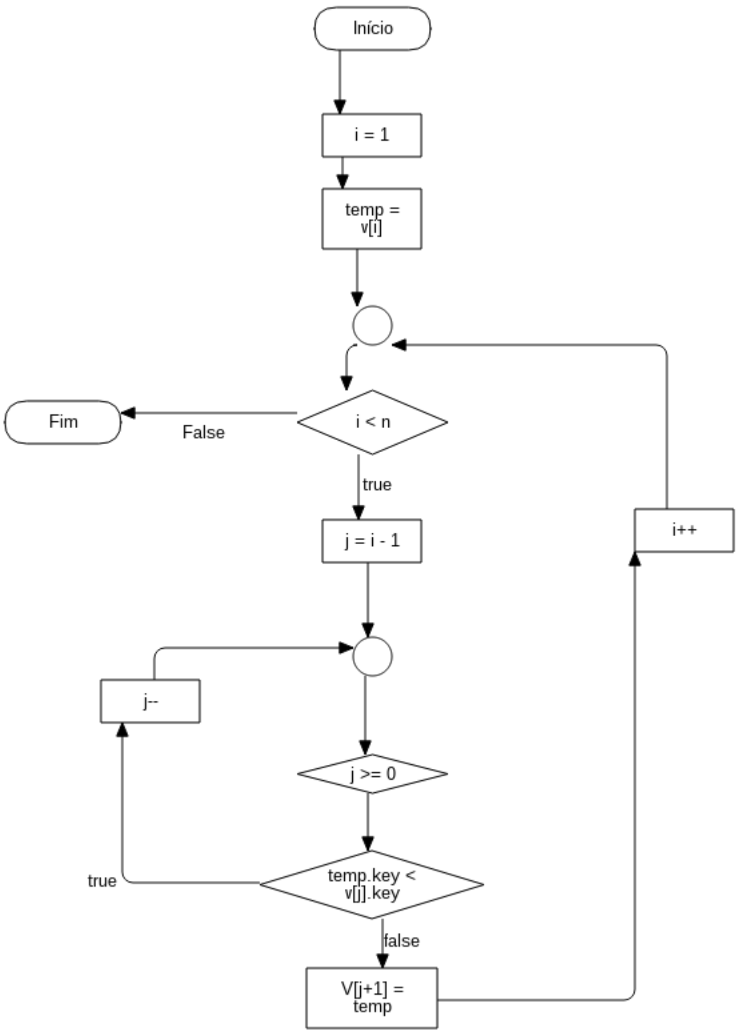

Figura 1: InsertionSort

```
void quick_sort(Node v[], int low, int high)
{
  int pivot, aux;
  int i, j;
  pivot = v[(low + high) / 2].key;
  i = low;
  j = high;
  atrib += 3;

  while(i <= j)
  {
    while(v[i].key < pivot)
    {
      comp++;
      i++;
    }
    while(v[j].key > pivot)
    {
      comp++;
      j--;
    }
```

```
    if(i <= j)
    {
      comp++;
      swap(v, i, j);
      i++;
      j--;
    }
  }

  if(j > low)
    quick_sort(v, low, j);

  if(i < high)
    quick_sort(v, j + 1, high);
}
```

Algoritmo 5: QuickSort

```
void merge_sort(Node v[], int n)
{
  mergePart(v, 0, n / 2 - 1);
  mergePart(v, n / 2, n - 1);
  merge(v, 0, n - 1);
}

void mergePart(Node v[], int a, int b)
{
  comp++;
  if (b - a > 1)
  {
    mergePart(v, a, (a + b) / 2);
    mergePart(v, (a + b) / 2 + 1, b);
    merge(v, a, b);
  }
  else if (v[a].key > v[b].key)
  {
    comp++;
    swap(v, a, b);
  }
}
void merge(Node v[], int a, int b)
{
  int tam = b - a + 1;
  int m = (a + b) / 2;
  int j = a;
  int k = m + 1;
  int vetAux[tam];
  atrib += 5;
  for (int i = 0; i < tam; i++)
  {
    if (v[j].key < v[k].key && j <= m)
      vetAux[i] = v[j++].key;
    else
      vetAux[i] = v[k++].key;
    atrib++;
    comp += 3;
  }
  for (int i = 0; i < tam; i++)
```

```
    {
      v[a+i].key = vetAux[i];
      comp++;
      atrib++;
    }
}
```

Algoritmo 6: MergeSort

```
void max_heapify(Node a[], int i, int n)
{
      int largest = i;
      int l = 2*i + 1;
      int r = 2*i + 2;
      if (l < n && a[l].key > a[largest].key){
          largest = l;
        atrib++;
      }
      if (r < n && a[r].key > a[largest].key){
          largest = r;
        atrib++;
      }
      comp += 2;
      if (largest != i)
      {
        comp++;
          swap(a, i, largest);
          max_heapify(a, largest, n);
      }
      return;
}
void heap_sort(Node a[], int n)
{
      for (int i = n / 2 - 1; i >= 0; i--){
          max_heapify(a, i, n);
      }

      for (int i=n-1; i>=0; i--)
      {
          swap(a, 0, i);

          max_heapify(a, 0, i);
      }
}
```

Algoritmo 7: HeapSort

3 Análise de complexidade dos algoritmos

3.1 BubbleSort

O bubble sort faz múltiplas passadas em uma lista, em cada passada ele verifica se um par de elementos adjacentes estão em ordem, caso não estejam, a posição deles é trocada de forma que o maior deles fique após o menor, isso é repetido até que não sejam mais necessárias trocas.

O melhor caso do bubble sort é quando a sua entrada é uma lista ordenada, neste caso os elementos já estão em ordem nenhuma troca é efetuada e o algoritmo termina na primeira passada, logo sua complexidade é na ordem de $O(n)$.

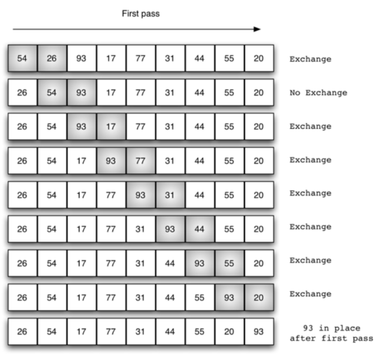

Figura 2: Uma passada do bubble sort.

No pior caso podemos levar em consideração quando os elementos estão ordenados em ordem decrescente, neste caso na iteração 0 o block roda $n-1-0$ vezes, na iteração 1 ele roda $n-1-(n-1)=0$ vezes. Então no total, o bloco roda a quantidade de vezes expressa na Equação 7.

$$O(n) = \sum_{i=0}^{n-1} n-i-1 = n^2 - \sum_{i=0}^{n-1} i - n = n^2 - n*(n-1)/2 = n^2/2 - n/2 \qquad (1)$$

Dando um pior caso na ordem de $O(n^2)$.

3.2 InsertionSort

Figura 3: Insertion sort funciona como um jogo de cartas.

O Insertion sort funciona usando a mesma ideia de quando organizamos cartas nas nossas mãos, para cada carta em nossas mãos verificamos se ela obedece a propriedade de lista ordenada de que a próxima carta tem um valor maior do que a carta atual, caso isso não aconteça, voltamos carta por carta até encontrar a posição que a carta pertence. O melhor caso ocorre quando a lista já está ordenada e todos

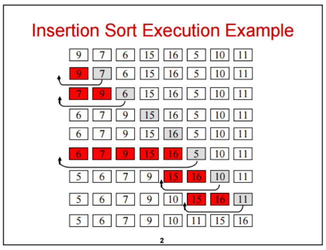

Figura 4: Exemplo de execução.

elementos estão em ordem, neste caso o algoritmo tem uma complexidade na ordem de $O(n)$.

O pior caso ocorre quando a lista está ordenada de forma decrescente, vamos tentar achar uma fórmula para a quantidade de movimentos necessários.
Iteração 0: 0
Iteração 1: 1
Iteração 2: $1 + 1 = 2$
Iteração 3: $2 + 1 = 3$
.
.
.
Iteração n-1: $\sum_0^{n-1} i = n * (n-1)/2 = n^2/2 - n/2$
Logo, a ordem de complexidade no pior caso é $O(n^2)$.

3.3 Selection Sort

O selection sort ordena a lista por repetidamente pegar o menor elemento da sublista restante quando se está na posição i e trocar a posição do menor elemento com o da posição da iteração i. O custo para pegar o elemento mínimo em cada iteração i é $n-i$ mesmo que o vetor esteja ordenado, pois não há uma verificação se os elementos já estão em ordem, então vamos ter um custo total dado pela equação 7.

$$O(n) = \sum_{i=0}^{n-1} n - i - 1 = n^2 - \sum_{i=0}^{n-1} i - n = n^2 - n * (n-1)/2 = n^2/2 - n/2 \tag{2}$$

Logo, a complexidade do algoritmo tanto no melhor, pior e caso médio é de $O(n^2)$.

3.4 MergeSort

O merge sort funciona utilizando o paradigma da divisão em conquista, dividindo a lista em dois recursivamente até que ela esteja ordenada, que é o caso em que o tamanho da lista é 1 e depois faz o merge das listas mantendo elas ordenadas.

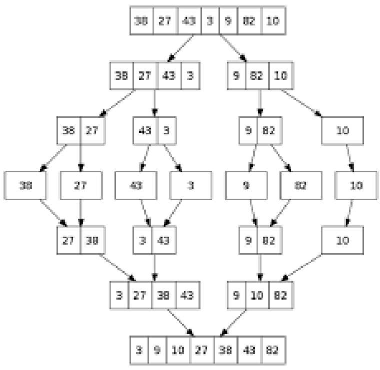

Figura 5: Árvore de execução do merge sort.

A equação de recorrência do merge sort é dada por 3.

$$T(n) = 2T(n/2) + n = c * n * log(n) \tag{3}$$

Logo, por o comportamento do merge sort não variar de acordo com a natureza da lista, a sua ordem de complexidade em todos os casos é $O(n * log(n))$.

3.5 HeapSort

O algoritmo Heap Sort insere todos os elementos (de um vetor não ordenado) em um heap então troca seu primeiro elemento (máximo) com o último (mínimo) e reduz o tamanho da heap por 1 por seu último elemento já está na posição final no vetor ordenado. Depois usamos o procedimento Heapify pois nesse processo podemos ter quebrado a propriedade de heap máxima. Continuamos esse processo até que o tamanho final da heap seja 1.

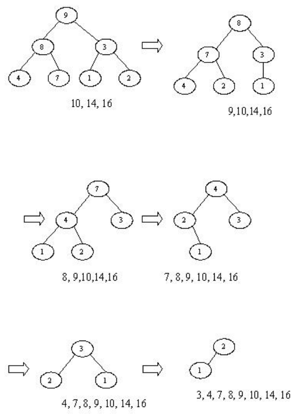

Figura 6: Exemplo do heap sort.

Primeiro nós construímos o max-heap que é um processo de ordem n e depois executamos o algoritmo Heapify que tem custo $log(n)$ por n vezes, logo a equação da complexidade do Heap sort é dada por 4.

$$T(n) = n + n * log(n) \tag{4}$$

Então o pior, melhor e o caso médio do Heap sort é da ordem de $O(n * log(n))$.

3.6 QuickSort

O algoritmo quicksort possui a seguinte equação de recorrência:

$$T(n) = T(k) + T(n - k) + c \times n \tag{5}$$

Onde o algoritmo particiona a lista recursivamente em duas partes escolhendo um pivô e as partes são k e n - k. No pior caso o pivô é o pior possível, então:

(6)

A equação resultante da análise de complexidade pode ser vista na Equação 7.

$$O(n) = \sum_{i=1}^{n} i^2 + 1 \tag{7}$$

4 Testes

Os testes foram realizados com os quatro algoritmos de ordenação. Como os algoritmos BubbleSort, InsertionSort e SelectionSort são considerados ineficientes em relção aos outros eles foram separados e comparados entre si. Para realizar os testes utilizamos quatro tipos de instâncias: lista ordenada em ordem crescente, lista ordenada em ordem decrescente, lista quase aleatória(primeiro e ultimo elementos trocados) e lista aleatória. Cada tipo possui instâncias de 10, 100, 1000, 10000, 100000 e 1000000 elementos.

4.1 Lista ordenada em ordem crescente

4.1.1 Algoritmos ineficientes

4.1.2 Algoritmos eficientes

Tabela 1: Comparações - lista ordenada crescentemente

Quantidade de comparações			
Tamanho da lista	Algoritmo de ordenação utilizado		
	Merge	Quick	Heap
10	114	53	93
100	2526	866	1923
1000	37022	11896	29127
10000	513502	154739	395871
100000	6531070	1868358	4952565
1000000	77048574	21880232	59363379

Tabela 2: Atribuições - lista ordenada crescentemente

Quantidade de atribuições			
Tamanho da lista	Algoritmo de ordenação utilizado		
	Merge	Quick	Heap
10	77	42	121
100	1515	456	2718
1000	20555	4506	42090
10000	280363	47706	576455
100000	3527675	465528	7255742
1000000	40621435	4524282	87453863

Tabela 3: Tempo gasto - lista ordenada crescentemente

Tempo gasto em segundos			
Tamanho da lista	Algoritmo de ordenação utilizado		
	Merge	Quick	Heap
10	1,00E-05	1,10E-05	1,80E-05
100	3,60E-05	3,70E-05	0,000171
1000	9,20E-05	7,50E-05	0,000687
10000	0,001205	0,000783	0,009227
100000	0,014039	0,007828	0,093136
1000000	0,153944	0,083055	1,19613

4.2 Lista ordenada em ordem decrescente

4.2.1 Algoritmos ineficientes

4.2.2 Algoritmos eficientes

Tabela 4: Comparações - Lista ordenada decrescentemente

Quantidade de comparações			
Tamanho da lista	Algoritmo de ordenação utilizado		
	Merge	Quick	Heap
10	118	50	66
100	2562	870	1551
1000	37510	11894	24951
10000	517598	154736	350091
100000	6565534	1868362	4492305
1000000	77524286	21880230	55000227

Tabela 5: Atribuições - lista ordenada decrescentemente

Quantidade de atribuições			
Tamanho da lista	Algoritmo de ordenação utilizado		
	Merge	Quick	Heap
10	89	54	81
100	1623	609	2112
1000	22019	6003	35291
10000	292651	62703	503710
100000	3631067	615531	6517115
1000000	42048571	6024279	80233851

Tabela 6: Tempo gasto - lista ordenada decrescentemente

Tempo gasto em segundos			
Tamanho da lista	Algoritmo de ordenação utilizado		
	Merge	Quick	Heap
10	3,00E-06	3,00E-06	4,00E-06
100	1,20E-05	1,30E-05	3,60E-05
1000	0,000103	0,000115	0,000498
10000	0,001216	0,000915	0,006804
100000	0,01467	0,009953	0,087639
1000000	0,179583	0,110791	1,12486

4.3 Lista quase ordenada

4.3.1 Algoritmos ineficientes

4.3.2 Algoritmos eficientes

Tabela 7: Comparações - lista quase ordenada

Quantidade de comparações			
Tamanho da lista	Algoritmo de ordenação utilizado		
	Merge	Quick	Heap
10	116	53	84
100	2527	866	1914
1000	37023	11896	29088
10000	513504	154739	395796
100000	6531071	1868358	4952538
1000000	77048575	21880232	59366004

Tabela 8: Atribuições - lista quase ordenada

Quantidade de atribuições			
Tamanho da lista	Algoritmo de ordenação utilizado		
	Merge	Quick	Heap
10	83	45	109
100	1518	459	2707
1000	20558	4509	42036
10000	280369	47709	576361
100000	3527678	465531	7253529
1000000	40621438	4524285	87457639

Tabela 9: Tempo gasto - lista quase ordenada

Tempo gasto em segundos			
Tamanho da lista	Algoritmo de ordenação utilizado		
	Merge	Quick	Heap
10	3,00E-06	3,00E-06	4,00E-06
100	9,00E-06	1,00E-05	4,20E-05
1000	0,000105	6,10E-05	0,000559
10000	0,000998	0,00064	0,007651
100000	0,012542	0,007002	0,09462
1000000	0,153052	0,086916	1,18886

4.4 Lista Aleatória

4.4.1 Algoritmos ineficientes

4.4.2 Algoritmos eficientes

Tabela 10: Comparações - lista aleatória

Quantidade de comparações			
Tamanho da lista	Algoritmo de ordenação utilizado		
	Merge	Quick	Heap
10	117	62	84
100	2546	989	1737
1000	37279	14712	27102
10000	515561	189538	372300
100000	6548267	2448530	4726323
1000000	77286078	29393733	57147021

Tabela 11: Atribuições - lista aleatória

Quantidade de atribuições			
Tamanho da lista	Algoritmo de ordenação utilizado		
	Merge	Quick	Heap
10	86	63	107
100	1575	852	2403
1000	21326	10674	38742
10000	286540	130830	538751
100000	3579266	1528554	6890311
1000000	41333947	17559420	83768004

Tabela 12: Tempo gasto - lista aleatória

Tempo gasto em segundos			
Tamanho da lista	Algoritmo de ordenação utilizado		
	Merge	Quick	Heap
10	3,00E-06	1,80E-05	4,00E-06
100	1,30E-05	2,10E-05	4,00E-05
1000	0,000134	0,000248	0,000592
10000	0,001644	0,003233	0,00768
100000	0,01941	0,039624	0,106496
1000000	0,239133	0,470272	1,91489

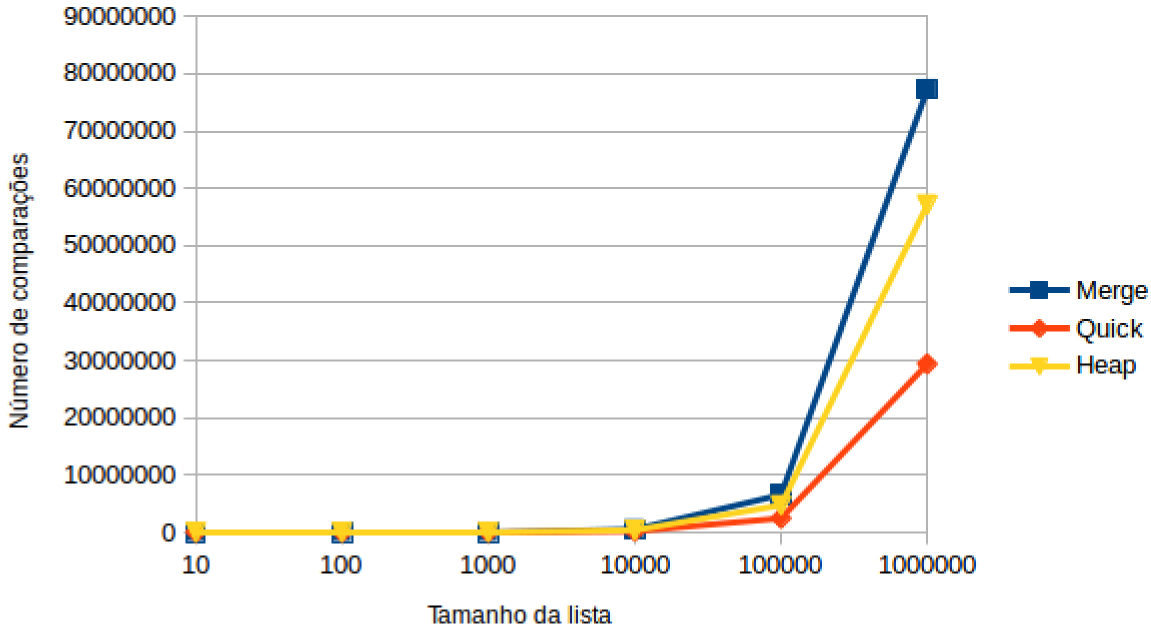

Figura 7: Comparações - Lista aleatória

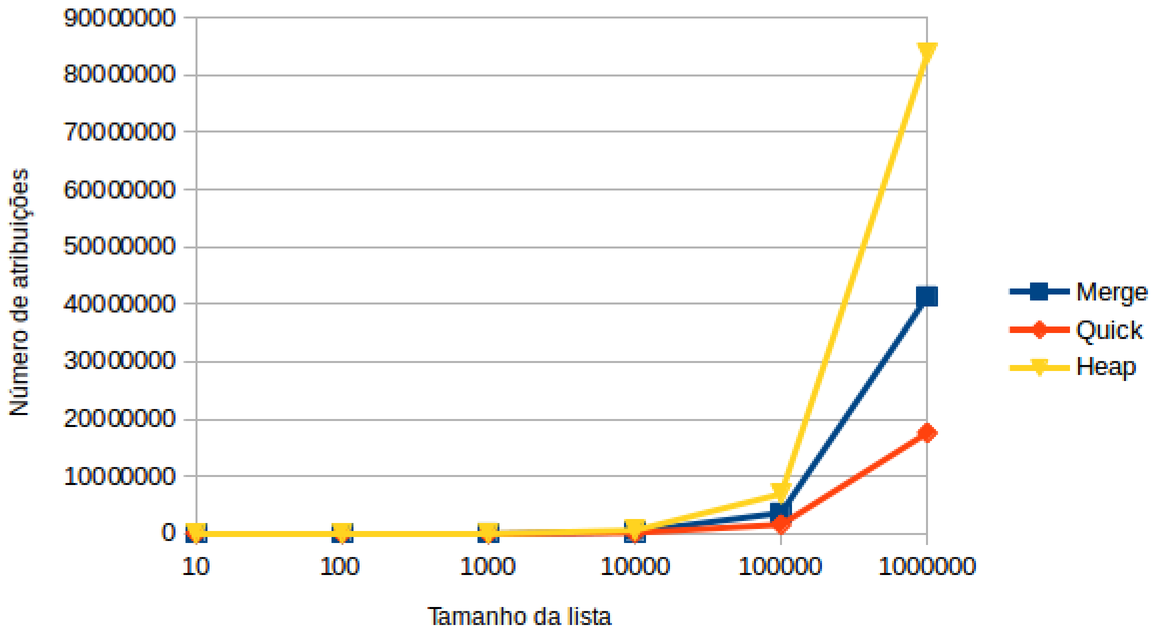

Figura 8: Atribuições - Lista aleatória

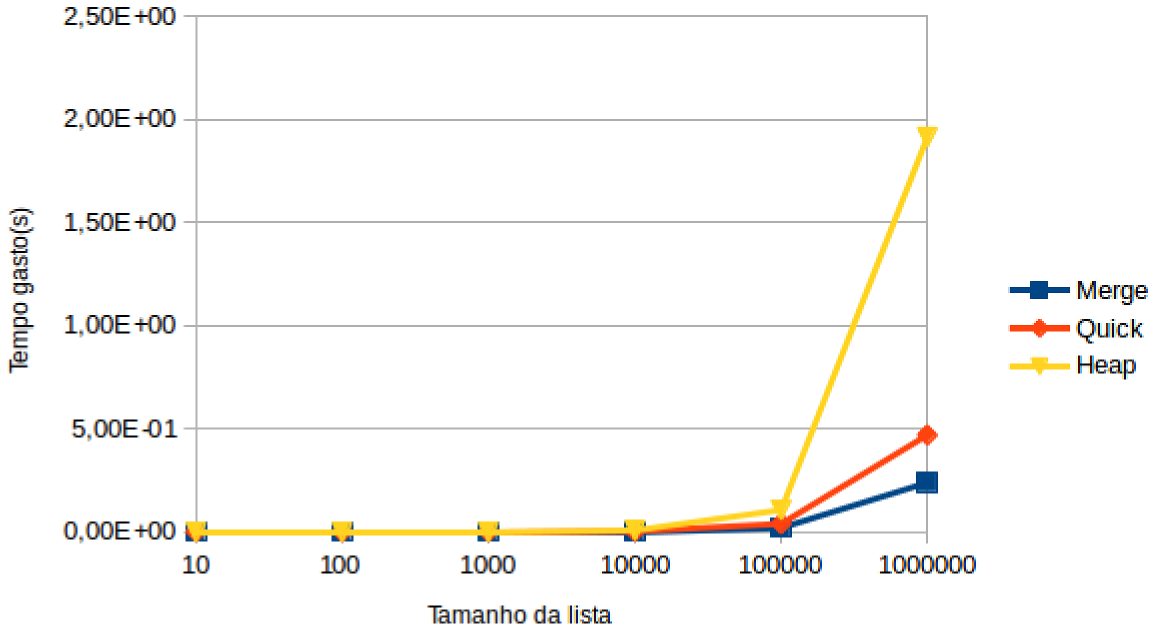

Figura 9: Tempo gasto - Lista aleatória

5 Conclusão

Escrever conclusão