Universidade Federal de Viçosa

Campus **Rio Paranaíba**

"Thiago Matheus De Oliveira Costa" - "8101"

"Analise de Algoritmos de Ordenação"

Rio Paranaíba - MG

2023

Universidade Federal de Viçosa

Campus **Rio Paranaíba**

"Thiago Matheus de Oliveira Costa" - "8101"

"Análise De Algoritmos de Ordenação "

Trabalho apresentado para obtenção de créditos na disciplina SIN213 - Projeto de Algoritmo da Universidade Federal de Viçosa - Campus de Rio Paranaíba, ministrada pelo Professor Pedro Moisés de Souza.

Rio Paranaíba - MG

2023

1 RESUMO

Os algoritmos de ordenação são de grande importância na ciência da computação e em diversar outras áreas da tecnologia, pois realizam um grande papel no mundo moderno, como em aplicações rotineiras e práticas do cotidiano, feitas em aparelhos celulares como exemplo, que utilizam desses algoritmos. Diante dessa grande importância ao longo dos anos surgem alguns algoritmos de ordenação na literatura, onde são analisados casos em que cada algoritmo se enquadra. Sendo assim se torna necessário um estudo para analisar o comportamento desses algoritmos, bem como suas complexidades para indicar qual é o mais adequado para cada tipo de problema.

Nesse estudo utilizaremos os seguintes algoritmos: Insertion Sort, Selection Sort, Shell Sort, Bubble Sort, Quick Sort, Merge Sort, Heapsort e suas variações. Nas situações randomica, crescente ou decrescente, com entradas de tamanho 10, 100, 1000, 10000, 100000, 1000000.

Com o objetivo de dar um certo entendimento do funcionamento de cada um dos algoritmos, seus desempenhos e quais são os melhores para utilizar em cada situação.

Sumário

2 INTRODUÇÃO

Os algoritmos de ordenação são ferramentas amplamente utilizadas no mundo moderno em diversas áreas da tecnologia, em que desempenha um papel crucial na vida cotidiana, muitas vezes de forma invisivel á nós humanos. Foram projetados para organizar um conjunto de dados em uma ordem específica definida pelo usuario/programador, facilitando como exemplo a busca, recuperação de dados, tomada de decisões, entre outros, e também para aplicações na vida real como em filas de prioridade.

Portanto imagine como seria diferente coisas simples na nossa vida, como uma lista telefônica eletronica que contém diversos nomes e números de telefone. Caso não forem ordenados de forma eficiente, consequentemente localizar um número específico seria uma tarefa tediosa e demorada.

Portanto Diante dessa importância, surgiu-se alguns algoritmos de ordenação na literatura, com a sua primeira origem registrada pelo RadixSort[Kadam and Kadam(2014)], Fabricada por Hermann Hollerith, com no decorrer dos anos melhorando cada vez mais esses algoritmos, chegando até algoritmos atuais como o Bubble Sort.

Segundo Dijkstra, um algoritmo corresponde a uma descrição de um padrão de comportamento, expresso em termos de um conjunto finito de ações.

Em razão disso nesse estudo estará sendo realizado a analise de diversos algoritmos de ordenação e suas complixidades, e ao final uma comparação entre os resultados de cada algoritmo.

3 ALGORITMOS

3.1 Insertion Sort

O Algoritmo Insertion sort é um algoritmo de ordenação, simples e eficiente principalmente para listas de tamanhos pequenos, com seus numeros em sua maioria já ordenados. Criado inicialmente em 1945, pelo engenheiro alemão **Konrad Zuse**. Para listas com tamanhos maiores e com seus valores pouco ordenados percebe-se que seu tempo de processamento aumenta exponencialmente de acordo com seu tamanho assim tornando-se uma opção menos viavel nesse caso.

Nesse estudo será possivel visualizar o desempenho do algoritmo de acordo com o tamanho da entrada (10, 100, 1000, 10000, 100000, 1000000) e com ordem crescente, decrescente e randomica.

Uma de suas principais vantagens é de que sua implementação é simples e não há a necessidade de memorial adicional significativa.

O algoritmo funciona da seguinte maneira: Começa considerando de que o primeiro elemento da lista já está ordenado e examina o proximo elemento não classificado e inserindo na sua posição correta na parte mais a esquerda da lista e deslocando os elementos maiores á direita, que nem demonstrado na figura abaixo:

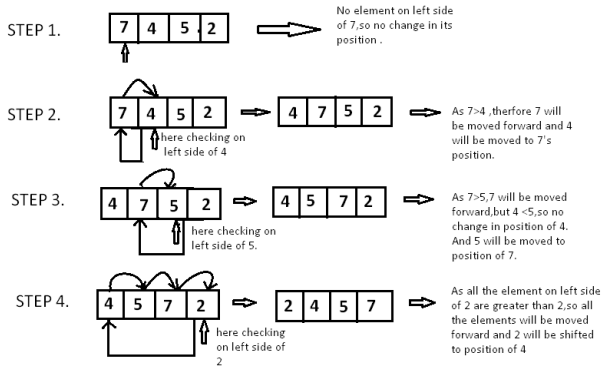


Figura 1: Enter Caption

E na proxima imagem o codigo escrito do algoritmo:

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Figura 2: Enter Caption

3.2 Bubble Sort

O algoritmo de ordenação Bubble Sort é um método simples de ordenação e intuitivo que pertence à categoria de algoritmos de ordenação por comparação, que percorre repetidamente uma lista, compara elementos adjacentes e os troca de posição se estiverem fora de ordem. O processo continua até que nenhum par de elementos precise ser trocado indicando que a lista está ordenada.

A principal desvantagem do Bubble Sort é a sua ineficiência. Ele tem uma complexidade de tempo de pior caso de O(ne2), onde "n"é o número de elementos na lista. Significando que o tempo de execução aumenta quadraticamente com o tamanho da lista. Portanto percebe-se de que o Bubble Sort não é adequado para ordenar grandes conjuntos de dados.

Seu funcionamento é da seguinte maneira, percorre uma lista de elementos, comparando pares de elementos adjacentes. Se um par estiver fora de ordem, eles são trocados de posição. Esse processo é repetido até que todos os elementos estejam em suas posições corretas na lista, conforme demonstrado na imagem 1 abaixo e na imagem 2 seu codigo:

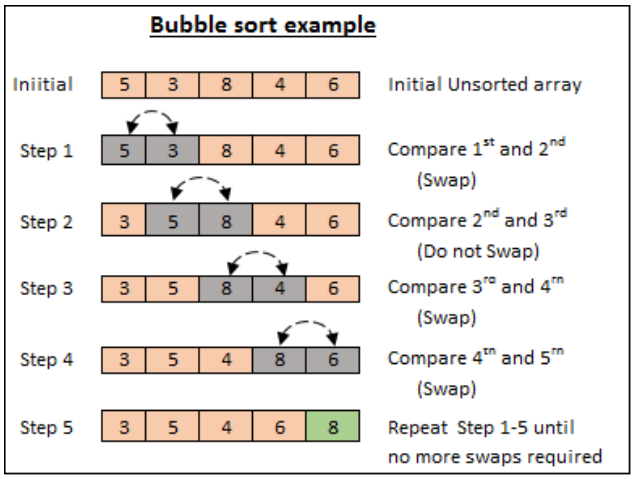


Figura 3: Funcionamento Bubble Sort

```
void BubbleSort (int *v, int TamVet)
{
    int i, j, aux;

    for (i=1; i<TamVet; i++)
    {
       for (j= TamVet-1; j>=i; j--)
       {
          if (v[j-1] > v[j])
          {
               aux = v[j-1];
               v[j-1] = v[j];
               v[j] = aux;
          }
       }
    }
    return;
}
```

Figura 4: Codigo Bubble Sort

3.3 Shell Sort

O algoritmo shell Sort é um algoritmo de ordenação que melhora a eficiência do algoritmo de ordenação por inserção Insertion Sort ao reduzir o número de trocas de elementos. Desenvolvido por Donald Shell em 1959, conhecido por ser um dos primeiros algoritmos de ordenação eficiente para grandes conjuntos de dados.

Sua principal vantagem em relação ao Insertion Sort é de que ele realiza várias ordenações locais, tornando os elementos mais próximos de suas posições finais, reduzindo o número de trocas de elementos necessárias durante a ordenação final.

Sua Eficiencia depende da escolha dos intervalos. A complexidade de tempo média do Shell Sort é dificil de determinar com precisão, pois ela varia com base nos intervalos escolhidos, porém geralmente fica entre O(n) e O(ne2). Em média o Shell Sort é mais eficiente do que o Insertion Sort.

Seu funcionamento ao invés de comparar e mover elementos um por um, ele compara e move elementos a uma distância fixa("intervalo"), seu funcionamento é da seguinte maneira:

1 - Escolhe um conjunto de intervalos que serão usados para dividir a lista em subgrupos. Os intervalos são selecionados de forma a começar com um valor relativamente grande e diminuir progressivamente. 2 - Divide a lista em vários subgrupos. Em seguida, aplica o algoritmo de ordenação Insertion Sort a cada um dos subgrupos. 3 - Reduz o intervalo e repete o processos anteriores. 4 - Realiza a ordenação final com o intervalo igual a 1.

Que nem demonstrada na imagem 1(funcionamento) e na imagem 2(codigo) abaixo:

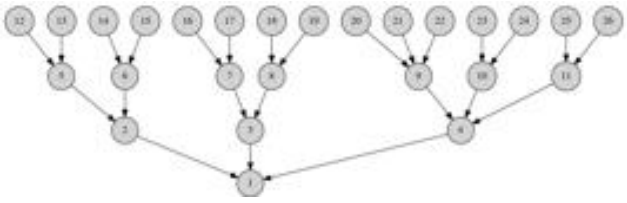
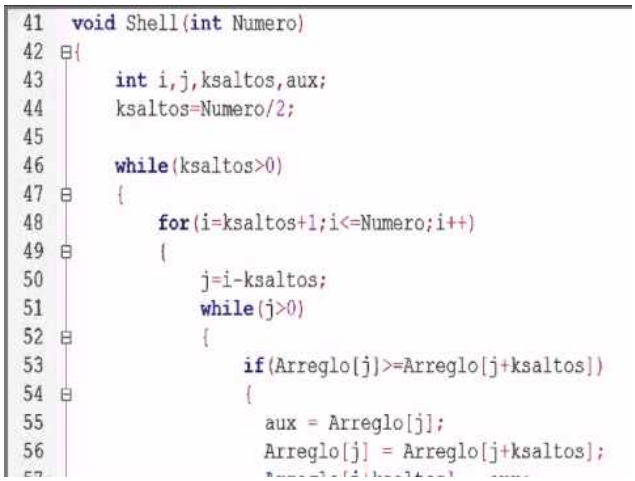


Figura 5: Funcionamento Shell Sort



```
41  void Shell(int Numero)
42  {
43      int i,j,ksaltos,aux;
44      ksaltos=Numero/2;
45
46      while(ksaltos>0)
47      {
48          for(i=ksaltos+1;i<=Numero;i++)
49          {
50              j=i-ksaltos;
51              while(j>0)
52              {
53                  if(Arreglo[j]>=Arreglo[j+ksaltos])
54                  {
55                    aux = Arreglo[j];
56                    Arreglo[j] = Arreglo[j+ksaltos];
```

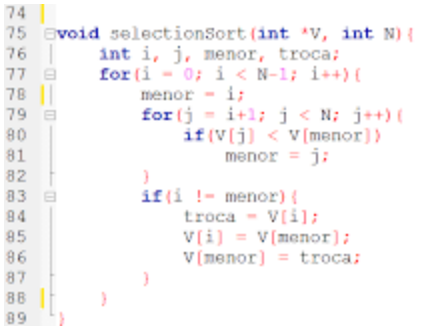
Figura 6: Codigo Shell Sort

3.4 Selection Sort

O algoritmo Selection Sort é um dos mais simples para ordenar uma lista de elementos. Ele funciona selecionando repetidamente o menor ou maior (dependendo da ordem desejada), elemento da lista não ordenada e movendo-o para a parte ordenada da lista.

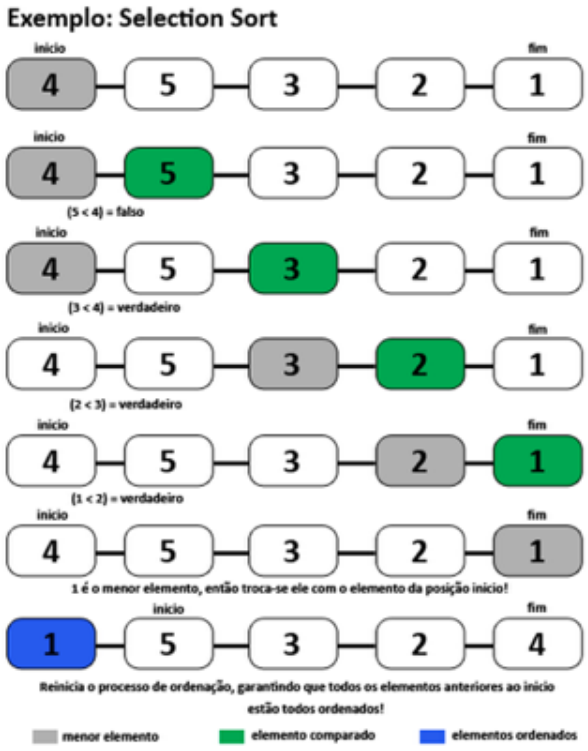
O Selection Sort é chamado assim em razão da "chamada"que ele realiza do menor ou maior elemento a cada interação e o move para a parte ordenada. No entanto nota-se que independentemente da ordem original dos elementos, ele realizará o mesmo numero d comparações e movimentações. Significando que o seu desempenho é consistentemente ineficiente para listas de qualquer tamanho, por ter uma complexidade de tempo de pior caso de O(ne2), portanto não é recomendado para grandes conjuntos de dados.

Seu funcionamento e seu codigo são demonstrados nas imagens 1 e 2 abaixo:



```
74
75  void selectionSort(int *V, int N){
76      int i, j, menor, troca;
77      for(i = 0; i < N-1; i++){
78          menor = i;
79          for(j = i+1; j < N; j++){
80              if(V[j] < V[menor])
81                  menor = j;
82          }
83          if(i != menor){
84              troca = V[i];
85              V[i] = V[menor];
86              V[menor] = troca;
87          }
88      }
89  }
```

Figura 7: Codigo Selection Sort



3.5 Quick Sort

O algoritmo QuickSort foi proposto por Hoare em 1960, sendo publicado em 1962. É o algoritmo de ordenação mais rápido desenvolvido, para uma ampla variedade de situações, sendo amplamente utilizado. A ideia básica do algoritmo é o de dividir um problema de ordenação de um conjunto com n itens em dois problemas menores, com esses problemas menores sendo ordenados independentemente, em seguida os resultados são combinados para produzir a solução final, mais conhecido como os metodos de divisão e conquista.

Uma da caracteristica do Quicksort é a sua ineficiência para arquivos já ordenados, quando a escolha do pivô é inadequada, por exemplo, a escolha sistemática dos extremos de um arquivo já ordenado leva ao seu pior caso. Nesse caso, as partições serão extremamentes desiguais, e o metodo Quicksort será chamada recursivamente n vezes, eliminando apenas um item em cada chamada.

O algoritmo quicksort possui 4 versões, em que a escolha do pivo pode ser: o primeiro elemento, por meio da média, mediana de três elementos, randomicamente, em que a principal diferença entre eles é da forma como é feita a escolha do pivô, assim alterandon o metodo

particiona, em que dependendo da situação um metodo pode ser mais eficiente que o outro.

Sua parte mais complexa está relacionada ao método de partição, em que o vetor v[esq...dir] é rearranjado por meio da escolha arbitraria de um pivô, após a escolha o vetor é dividido em duas partes:

-A esquerda com chaves menores ou iguais a x. -A direita com chaves maiores ou iguais a x.

O algoritmo particiona funciona da seguinte maneira:

1°- Escolha arbitraria do pivo.

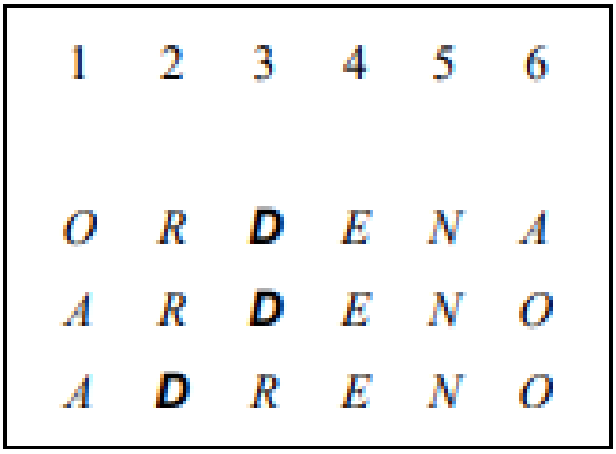
2°- Percorre o vetor a partir da esquerda até que v[j] seja maior ou igual a x.

3°- Percorre o vetor a partir da direita até que v[j] seja menor ou igual a x.

4°- Troca v[i] com v[j].

5°- Realiza esse processo até que i e j se cruzem.

Que nem representado na imagem a baixo:



A partição na imagem acima é realizado da seguinte maneira:

O pixo x é escolhido como sendo v[(i+j)/ 2], como inicialmente i = 1 e j = 6, então x = v[3] = D. Ao final do processo de partição i e j se cruzam em i = 3 e j = 2.

O codigo do metodo Particiona e do QuickSort está presente nas imagens abaixo:

```
private static class LimiteParticoes { int i; int j;
private static LimiteParticoes particao
                          (Item v[], int esq, int dir)
  LimiteParticoes p = new LimiteParticoes ();
  p.i = esq;  p.j = dir;
  Item x = v[(p.i + p.j) / 2]; // obtém o pivo x
  do {
    while (x.compara (v[p.i]) > 0) p.i++;
    while (x.compara (v[p.j]) < 0) p.j--;
    if (p.i <= p.j) {
      Item w = v[p.i]; v[p.i] = v[p.j]; v[p.j] = w;
      p.i++; p.j--;
    }
  } while (p.i <= p.j);
  return p;
}
```

```
private static void ordena (Item v[], int esq, int dir) {
  LimiteParticoes p = particao (v, esq, dir);
  if (esq < p.j) ordena (v, esq, p.j);
  if (p.i < dir) ordena (v, p.i, dir);
}

public static void quicksort (Item v[], int n) {
        ordena (v, 1, n);
        }
```

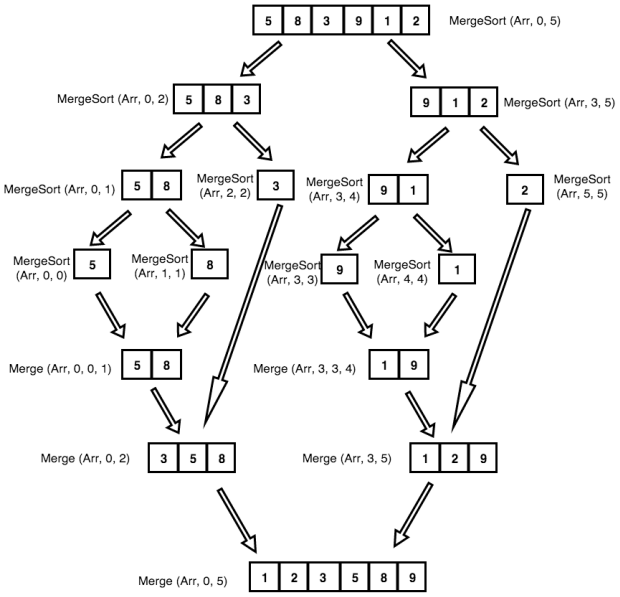
3.6 Merge Sort

O algoritmo Merge Sort é um método estavel para a ordenação de listas. Seu funcionamento é por meio de uma abordagem de divisão e conquista, onde a lista é dividida repetidamente em duas metades até que cada sublista contenha apenas um elemento, em seguida, as sublistas são mescladas em pares, ordenando a cada mesclação das suas sublistas, isso que é realizado repetidamente até que a lista esteja completamente ordenada. Abaixo está o codigo do MergeSort:

```
MERGE(A, p, q, r)
1   n1 = q − p + 1
2   n2 = r − q
3   sejam L[1..n1 + 1] e R[1.. n2 + 1] novos arranjos
4   for i = 1 to n1
5       L[i] = A[p + i − 1]
6   for j = 1 to n2
7       R[j] = A[q + j]
8   L[n1 + 1] = ∞
9   R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          then A[k] = L[i]
15               i = i + 1
16          else A[k] = R[j]
17               j = j + 1
```

Em detalhe, o procedimento Merge funciona da seguinte maneira. A linha 1 calcula o comprimento n1 do subarranjo A[p .. q] e a linha 2 calcula o comprimento n2 do subarranjo A[q + 1.. r]. Criamos os arranjos esquerda e direita, respectivamente, na linha 3 a posição extra em cada arranjo conterá a sentinela. O laço for das linhas 4 e 5 copia o subarranjo A[p .. q] em L[1 ..n1], e o laço for das linhas 6 e 7 copia o subarranjo A[q + 1 .. r] em R[1 .. n2]. As linhas 8 e 9 colocam as sentinelas nas extremidades dos arranjos L e R. As linhas 10 a 17, ilustradas na figura a seguir, executam os r p + 1 passos básicos:

Na imagem abaixo se consegue ver de uma forma mais clara o funcionamento do algoritmo mergesort em que é relizado sucessivas divisões no vetor, até que se encontre somente um elemento, em seguido, mesclando os vetores divididos e ordenando-os.



3.7 Heap Sort

O algoritmo Heap Sort é um algoritmo de ordenação baseado em comparações, utilizando a estrutura de dados heap. Proposto por J.W.J. Williams em 1964 e desenvolvido por Robert W. Floyd em 1969. Heap é uma árvore binária praticamente completa em que cada nó pai é maior ou menor dependendo da ordem desejada que seus filhos(esquerdo e direito). Na imagem abaixo está um exemplo de uma árvore binaria:

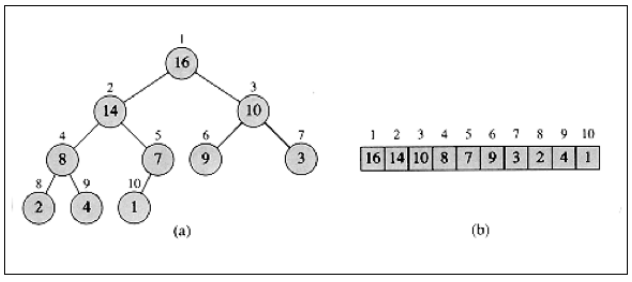


Figura 8: Árvore Binaria

A estrutura de dados heap possui diversar aplicações na vida real, em que a sua principal são em filas de prioridade, seja maxima ou minima, em que depende da situação em que serão utilizadas.

Existe dois tipos de heaps binarios, o heapmaximo e o heapminimo. Um heap é dito maximo quando para todo o heap os nós "pai"são maiores que seus nós "filhos", portanto

com seu maior elemento armazenado na "raiz". E o heap é dito minimo, quando para todo os nós "pais"são menores que seus nós "filhos", portanto com seus menor elemento na "raiz". Nesse estudo será realizado o heapminimo.

A tabela abaixo apresenta o algoritmo da função heapsort:

Passo	**Código**	**Descrição**
1	`Build_min_heap(A)`	Construir um heap mínimo a partir de A.
2	`for i <-- comprimento(A) down to 2`	Loop de `comprimento(A)` até 2 (decrementando).
3	`swap(A[1] <-> A[i])`	Trocar os elementos `A[1]` e `A[i]`.
4	`comprimento(A) = comprimento(A) - 1`	Reduzir o `comprimento(A)` em 1.
5	`min_heapify(A, 1)`	Garantir que a propriedade de heap mínimo seja mantida em `A[1]`.

O algoritmo faz uso de duas funções para seu funcionamento, minheapify que é uma função para verificar se o nó pai é menor que seus respectivos nós filhos, que nem demonstrado na tabela abaixo:

height**Linha**	**Código**	**Descrição**
1	`L <-- 2 * i`	Atribui o dobro do índice i à variável L.
2	`R <-- 2 * i + 1`	Atribui o dobro do índice i mais um à variável R.
3	`if i <= comprimento(A) e A[L] < A[R]`	Condição: se o índice i é válido e o elemento em $A[L]$ é menor que $A[R]$.
4	`then menor <-- L`	Atribui o valor de L à variável `menor`.
5	`else menor <-- i`	Caso contrário, atribui o valor de i à variável `menor`.
6	`if R <= comprimento(A) e A[R] < A[maior]`	Condição: se o índice R é válido e o elemento em $A[R]$ é menor que A[maior].
7	`then maior <-- R`	Atribui o valor de R à variável `maior`.
8	`if maior != i`	Condição: se `maior` não é igual a i.
9	`then swap(A[i] <-> A[maior])`	Troca os elementos $A[i]$ e A[maior].
10	`min_heapify(A, maior)`	Chama recursivamente a função `min_heapify` com o novo valor de `maior`.

E a função BuildMinHeap para a construção do heap que nem demonstrado na tabela abaixo:

Linha	Código
1	`tamanho_do_heap(A) <- comprimento(A)`
2	`for i <- comprimento[A]/2 down to 1`
3	`do minHeapify(A, i)`

Nesse estudo foi utilizado outras funções derivadas dessas, são elas: HeapMinimum, HeapExtractMin, HeapIncreaseKey, MaxHeapInsert.

A função HeapExtractMin retira o menor valor da arvore, que se encontra na raiz, seu funcionamento é da seguinte forma, o elemento mínimo é removido do heap. Após a remoção, é necessário reorganizar a estrutura, geralmente por meio da função "heapify". O elemento mínimo originalmente removido é retornado como resultado da operação. Seu codigo é apresentado na tabela abaixo:

height**Número**	**Código**	**Ação**
1	`if comprimento(A) < 1`	Verifica se o comprimento de A é < 1
2	`then error "Heap under low"`	Emite um erro "Heap under low"
3	`minimum = A[1]`	Atribui o valor de `A[1]` a `minimum`
4	`A[1] = A[comprimento(A)]`	Atribui o valor de `A[comprimento(A)]` a `A[1]`
5	`comprimento(A) = comprimento(A) - 1`	Reduz o comprimento de A em 1
6	`minHeapfy(A, 1)`	Chama a função `minHeapfy` com os parâmetros A e 1
7	`return minimum`	Retorna o valor de `minimum`

A função HeapMinimum, encontra e retornar o menor valor na arvoré, seu codigo é demonstrado na tabela abaixo:

height**Número da Linha**	**Código**	**Comentário**
1	`minimum <-- A[1]`	Inicializa `minimum` com o valor no índice 1 de A.
2	`for i <-- comprimento(A) down to 1`	Loop de i igual ao comprimento de A até 1 (decrescente).
3	`L <-- 2 * i`	Calcula o índice à esquerda (L) usando a fórmula $2 \times i$.
4	`R <-- 2 * i + 1`	Calcula o índice à direita (R) usando a fórmula $2 \times i + 1$.
5	`if i <= comprimento(A) e A[L] < minimum`	Verifica se i é válido e se $A[L]$ é menor que `minimum`.
6	`then minimum = A[L]`	Atualiza `minimum` com o valor em $A[L]$.
7	`if i <= comprimento(A) e A[R] < minimum`	Verifica se i é válido e se $A[R]$ é menor que `minimum`.
8	`then minimum = A[R]`	Atualiza `minimum` com o valor em $A[R]$.
9	`return minimum`	Retorna o valor mínimo encontrado durante o loop.

A função Heapincreasekey, tem como objetivo aumentar a chave de um nó específico na árvore de heap e, em seguida, ajustar a estrutura de heap para manter as propriedades do heap. Na tabela abaixo é aprensentado o codigo da função:

Passo	**Código**	**Ação**
1	`if chave < A[1]`	"Error"
2	`then "Error"`	
3	`A[i] <-- chave`	
4	`while i > 1 e A[parent[i]] > A[i]`	
5	`do swap(A[i] <-> A[parent[i]])`	Realiza a troca de elementos
6	`i <-- parent[i]`	Atualiza o índice i

E por ultimo a função MaxHeapInsert, O objetivo da função é inserir um novo elemento no max heap, mantendo a propriedade de max heap. A função realiza os seguintes passos: adiciona o novo elemento ao final da árvore, como uma folha. Em seguida, ajusta a posição do novo elemento para cima na árvore, para um índice menor até que a propriedade de max heap seja restaurada. E na tabela abaixo é apresentado o codigo da função:

height**Número**	**Código**	**Descrição**
1	`comprimento(A) <-- comprimento(A) + 1`	Atualiza o comprimento do array A, adicionando 1.
2	`A[comprimento(A)] <-- -rand()`	Atribui um valor negativo aleatório à posição `comprimento(A)` do array A.
3	`heapIncreaseKey(A, comprimento(A), chave)`	Chama a função `heapIncreaseKey` no array A, com o índice `comprimento(A)` e a chave fornecida como parâmetro.

4 ANÁLISE DE COMPLEXIDADE

4.1 Insertion Sort

Para o calculo da complexidade do algoritmo Insertion Sort, nota-se o pseudo-código abaixo, para axuliar o entendimento, em forma de tabela na imagem abaixo:

Insertion Sort(A,n)	Custo	Vezes
1 for j ← 2 to comprimento	C_1	n
2 — do chave ← A[j] C2 n-1	C_2	n-1
3 — //inserir A[j] na sequência ordenada A(1...N)	$C_3 = 0$	n-1
4 — i ← j-1	C_4	n-1
5 — while i > 0 e A[i] > chave	C_5	$\sum_2^n tj$
6 —— do A[i+1] ← A[i]	C_6	$\sum_2^n (tj-1)$
7 —— i ← i-1	C_7	$\sum_2^n (tj-1)$
8 — A[i+1] ← chave	C_8	n-1

Com a tabela acima[Ziviani et al.(2004)], calcula-se a complexidade do algoritmo a partir de certos cálculos, demonstrados abaixo:

$$t(n) = C_1 n + C_2(n-1) + C_4(n-1) + C5[\sum_2^n tj] + C_6[\sum_2^n (tj-1)] + C_7[\sum_2^n (tj-1)] + C_8(n-1) \tag{1}$$

Acima, pode-se observar o cálculo do **Custo** × **Vezes**, que é a quantidade de vezes que a linha do código foi executada. Agora, partinfo adiante, contemplamos mais cálculos para os **Melhor, Médio e Pior casos**.

4.1.1 Melhor Caso

No Melhor Caso, é o caso em que os valores da entrada estão ordenadas, na ordem crescente. Com isso, a linha 5 do nosso pseudo-código será **falsa**, já que a verificação só

ocorrerá apenas uma unica vez. Com isso, tem-se o seguinte cálculo a fazer:

$$t(n) = [C_1 n] + [C_2(n-1)] + [C_4(n-1)] + [C_5(n-1)] + [C_8(n-1)] \tag{2}$$

Concluído a primeira parte, segue-se a realização da próxima operação para melhor entendimento, que é colocar o n em envidência:

$$t(n) = n[C_1 + C_2 + C_4 + C_5] + [-C_2 - C_4 - C_5 - C_8] \tag{3}$$

Agora, com uma simples noção de conhecimento, assimila-se que o cálculo acima pode ser representado, também, da seguinte forma:

$$t(n) = A^{'} n + B \tag{4}$$

4.1.2 Pior Caso

No **Pior Caso**, a ordem será decrescente, ou seja, assim o algoritmo terá que ordenar todos os valores da entrada. Aqui, observando a tabela do Insertion Sort, serão utilizados todos os valores de **Vezes**. Então, fica-se com uma **Soma de Progressão Aritmética**. Para sua representação, nota-se abaixo:

$$SPA = \frac{(a_1 + a_n)n}{2} \rightarrow \frac{(n-1)(2+n)}{2} \tag{5}$$

Resolvendo essa equação, o resultado é:

$$\frac{n + n^2}{2} - 1, \tag{6}$$

que também pode ser escrito da seguinte forma:

$$\frac{n(n-1)}{2} - 1 \tag{7}$$

Com isso, os valores necessários para a conclusão do cálculo foram encontrados. É possível,

agora, seguir adiante para encontrar a **fórmula geral** do **Pior Caso**:

$$t(n) = C_1 n + C_2(n-1) + C_4(n-1) + C_5[\sum_{2}^{n}(j-1)] + \tag{8}$$

$$C_6[\sum_{2}^{n}(j-1)] + C_7[\sum_{2}^{n}(j-1)] + C_8(n-1) \tag{9}$$

Concluindo alguns cálculos, a primera parte é mostrada abaixo:

$$t(n) = C_1 n + C_2 n - C_2 + C_4 n - C_4 + C_5\frac{n^2+n}{2} - 1 + \tag{10}$$

$$C_6\frac{n^2+n}{2} - 1 + C_7\frac{n^2+n}{2} - 1 + C_8 n - C_8 \tag{11}$$

E a segunda parte, com alguns cálculos a mais:

$$t(n) = C_1 n + C_2 n - C_2 + C_4 n - C_4 + \frac{C_5 n}{2} + \frac{C_5 n^2}{2} - C_5 + \tag{12}$$

$$\frac{C_6 n^2}{2} - \frac{C_6 n}{2} + \frac{C_7 n^2}{2} - \frac{C_7 n}{2} + C_8 n - C_8 \tag{13}$$

A partir da equação acima, é possível ver que colocar os valores em evidência é plausível, da seguinte forma:

$$t(n) = n^2(\frac{C_5 + C_6 + C_7}{2}) + n(C_1 + C_2 + C_4 + \frac{C_5 - C_6 - C_7}{2} + C_8) + (-C_2 - C_4 - C_5 - C_8) \tag{14}$$

Com isso, fazendo as devidas trocas que já foram feitas para representar os pedaços da equação, nota-se uma representação abaixo:

$$t(n) = A^{''}n^2 + B^{''}n + C \tag{15}$$

A conclusão que pode ser feita é que o tempo de execução do **Pior Caso** de um algoritmo estabelece um limite superior para o tempo de qualquer entrada. Conhecê-lo permite uma garantia de que o algotimo nunca demorará mais do que esse tempo, já que foi delimitado.

4.1.3 Médio Caso

No **Médio Caso**, será em que as entradas não estão nem na ordem crescente, nem na decrescente, assim considerando que estão na ordem aleatoria. Com isso, o seguinte cálculo para executar será:

$$t(n) = C_1 n + C_2(n-1) + C_4(n-1) + C_5[\sum_{2}^{n} j] + \tag{16}$$

$$C_6[\sum_{2}^{n}(j-1)] + C_7[\sum_{2}^{n}(j-1)] + C_8(n-1) \tag{17}$$

A partir dele, será feito os seguintes cômputos:

$$t(n) = C_1 n + C_2 n + C_4 n + C_5(\frac{n^2}{2} - 1) + C_6(\frac{n(n-1)}{2}) + C_7(\frac{n(n-1)}{2}) + C_8 n + C_8 \tag{18}$$

Lembrando que: $\sum_2^n j\frac{1}{2}$ é o mesmo que que $\frac{1}{2}\sum_2^n j$. Como ainda falta a multiplicação da fração com seus respectivos valores, as partes estão em amostra abaixo:

- $C_5 = \frac{1}{2}(\frac{n^2+n}{2} - 1) \rightarrow \frac{n^2+n-2}{4}$
- $C_6 e C_7 = \frac{1}{2}(\frac{n(n-1)}{2}) - (n-1) \rightarrow \frac{n^2-n}{4} - (n+1) \rightarrow \frac{n^2-5n+4}{4}$

Com isso, é possível continuar com os cálculos:

$$t(n) = C_1 n + C_2 n - C_2 + C_4 n - C_4 + \tag{19}$$

$$\frac{C_5 n^2 + C_5 - 2C_5}{4} + \frac{C_6 n^2 - 5C_6 n - 4C_6}{4} + \frac{C_7 n^2 - 5C_7 n + 4C_6}{4} + C8n - C8 \tag{20}$$

$$t(n) = n^2(\frac{C_5 + C_6 + C_7}{4} + n(C_1 + C_2 + C_4 + \frac{C_5 - 5C_5 - 5C_7}{4} + C_8) - \tag{21}$$

$$(C_2 + C_4 + \frac{2C_5 - 4C_6 - 4C_7}{4} + C_8) \tag{22}$$

Fazendo sua representação:

$$t(n) = A^{''}n^2 + B^{''}n + C, \tag{23}$$

que é a mesma do **Pior Caso**.

4.2 Bubble Sort

Para o cálculo de complexidade do algoritmo de ordenação ***Bubble Sort***, pode-se ver melhor o pseudo-código a tabela abaixo:

Bubble Sort(A,n)	Custo	Vezes
1 for i=1 até comprimento [A]-1	C1	n
2 — for j=1 até comprimento [A]-i	C2	$\sum_1^{n-1} tj$
3 —— Se (A[j] > A[j+1]) então	C3	$\sum_1^{n-1}(tj-1)$
4 ——— troca A[j] $\leftarrow\rightarrow$ A[j+1]	C4	$\sum_1^{n-1}(tj-1)$

A partir da tabela-base[Ziviani et al.(2004)], é possível fazer o cáclculo da complexidade do algoritmo de ordenação. O primeiro contato é com a **equação geral**:

$$t(n) = C1n + C2[\sum_1^{n-1} tj] + C3[\sum_1^{n-1}(tj-1)] + C4[\sum_1^{n-1}(tj-1)] \tag{24}$$

Portanto, pode-se partir para os próximos cálculos. Primeiramente, resolver as **equações de somatórios** é essencial, antes de divergir para os **Melhor, Médio e Pior casos**:

- $\sum_1^{n-1} tj = (1, 2, 3, 4, ..., n-1)$
- $\sum_1^{n-1}(tj-1)$

O primeiro somatório, como é possível notar, assemelha-se à uma **Progressão Aritmética**. Dito isso, aplicando a fórmula:

$$SPA = \frac{(a_1 + a_n)n}{2} \rightarrow \frac{(1+n-1)(n-1)}{2} \rightarrow \frac{n^2-n}{2} \tag{25}$$

A segunda, entretanto, é possível concluir separando em:

$$\sum_{1}^{n-1}(tj-1) \rightarrow \sum_{1}^{n-1} tj - \sum_{1}^{n-1} 1 \tag{26}$$

Como já foi descoberto o resultado anterior, podemos apenas trocar o segundo somatório por $n-1$:

$$\sum_{1}^{n-1}(tj-1) = \frac{n^2-n}{2} - n - 1 \rightarrow \frac{n^2-n-2n+n}{2} \rightarrow \frac{n^2-3n}{2} + 1 \tag{27}$$

Com os valores e os cálculos todos expostos, os **Melhor, Médio e Pior casos** podem ser feitos.

4.2.1 Melhor Caso

No **Melhor Caso**, **a entrada já está ordenada na ordem crescente**, ou seja, a *linha 4* do pseudo-código **não irá ocorrer**. Como já foi mostrado acima, na figura **??**, existem outras formas de deixar o código do **Bubble Sort** mais dinâmico para utilização. Porém será demonstrado apenas **Bubble Sort normal**:

$$t(n) = C1n + C2\sum_{1}^{n-1} tj + C3(\sum_{1}^{n-1}(tj-1)) \tag{28}$$

Com a equação geral modificada acima demonstrada, já é possível ver todo o cálculo até a **equação geral** do **Melhor Caso**:

$$t(n) = C1n + C2(\tfrac{n^2-n}{2} + C3(\tfrac{n^2-3n+2}{2})$$
$$t(n) = C1n + (\tfrac{C2n^2-C2}{2}) + (\tfrac{C3n^2-3nC3+C3}{2})$$
$$t(n) = n^2[\tfrac{C2+C3}{2}] + n[\tfrac{-C2-3C3}{2}] + C3$$
$$t(n) = An^2 + Bn + C$$

4.2.2 Pior Caso

Para o **Pior Caso**, tem-se que a entrada estará na ordem decrescente. A imagem abaixo explica exatamente esse exemplo:

Com a imagem mostrada acima, é viável ir em direção aos cáculos, que nesse caso, **terá a *linha 4* da tabela**, ja que terá de trocar de posições $n-1$ vezes.

Alguns valores já foram mostrados anteriormente, no **Melhor Caso**, então será iniciado o contato direto com o **termo geral do Pior Caso**:

$$t(n) = C_1 n + C_2(\tfrac{n^2-n}{2} + C_3(\tfrac{n^2-3n+2}{2}) + C_4(\tfrac{n^2-3n+2}{2})$$
$$t(n) = C_1 n + (\tfrac{C_2 n^2 - C_2}{2}) + (\tfrac{C_3 n^2 - 3nC_3 + C_3}{2}) + (\tfrac{C_4 n^2 - 3nC_4 + C_4}{2})$$
$$t(n) = n^2[\tfrac{C_2+C_3+C_4}{2}] + n[\tfrac{-C_2-3C_3-3C_4}{2}] + C_3 + C_4$$
$$t(n) = An^2 + Bn + C$$

4.2.3 Médio Caso

Partindo para o último caso, no **Médio Caso**, usaremos $j = \frac{1}{2}$ para a feitura dos cálculos de todos os somatórios:

$$\sum_{1}^{n-1}(\frac{tj}{2}) \tag{29}$$

O valor de tj já foi calculado acima, como mostrado na 24. A partir disso, é necessário apenas adaptar o novo valor, acrescentando o $\frac{1}{2}$ nos cálculos:

$$\sum_{1}^{n-1}(\frac{tj}{2}) = \frac{n^2-n}{2} \times \frac{1}{2} = \frac{n^2-n}{4} \tag{30}$$

Com o primeiro somatório modificado, é necessário partir para o próximo: $\sum_1^{n-1}(\frac{tj}{2} - 1)$

$$\sum_{1}^{n-1}(\frac{tj-1}{2}) = \sum_{1}^{n-1}\frac{tj}{2} - \sum_{1}^{n-1}\frac{1}{2} \tag{31}$$

Abstraindo os valores já existentes:

$$\sum_{1}^{n-1}(\frac{tj-1}{2}) = \frac{n^2-n}{2} \times \frac{1}{2} - \frac{n-1}{2} \tag{32}$$

Resolvendo essa equação, fica-se com:

$$\sum_{1}^{n-1}(\frac{tj-1}{2}) = \frac{n^2-n}{4} - \frac{n-1}{2}$$
$$\sum_{1}^{n-1}(\frac{tj-1}{2}) = \frac{n^2-n-(2n-2)}{4}$$
$$\sum_{1}^{n-1}(\frac{tj-1}{2}) = \frac{n^2-n-2n+2}{4}$$

Como resultado final, é obtido:

$$\sum_{1}^{n-1}(\frac{tj-1}{2}) = \frac{n^2-3n+2}{4} \tag{33}$$

A partir do resultado obtido acima, é possível, agora, avançar nos cálculos posteriores:

$$t(n) = C_1 n + \frac{C_2 n^2 - C_2 n}{4} + \frac{C_3 n^2 - 3C_3 n}{4} + C_3 + \frac{C_4 n^2 - 3C_4 n}{4} + C_4$$
$$t(n) = n^2(\frac{C_2+C_3+C_4}{4}) + n(C_1 - \frac{-C_2-3C_3-3C_4}{4}) + C_3 + C_4$$

Como resultado final, têm-se a mesma equação geral que os outros casos acimas já mostrados:

$$t(n) = An^2 + Bn + C$$

Com isso, pode-se concluir que a complexidade *Big O* do algoritmo de ordenação ***Bubble Sort*** é $O(n^2)$. Isso se deve aos seus dois *loops*, onde o primeiro *loop* executa n **iterações**, que são as quantidade de elementos presentes na lista, e o segundo *loop*, que também executa n iterações, porém essa é apenas na primera, e a partir da segunda iteração, o algoritmo executa $n-1$, na terceira iteração, $n-2$, e assim por diante, até o fim dos elementos do *Array*.

4.3 Shell Sort

Para o algoritmo shell sort não se tem ainda uma formula para a analise da complexidade. Isso é devido a seus problemas matemáticos muito complicados e dificeis.

Porém, falaremos sobre o melhor, pior e medio caso abaixo:

4.3.1 Melhor Caso

No melhor caso, a complexidade de tempo do Shell Sort depende da sequência de lacunas utilizada. Se uma sequência de lacunas é escolhida de forma que a lista esteja quase ordenada desde o inicio, o número de comparações e trocas será menor. Em um cenário ideal, onde a lista já está ordenada, a complexidade de tempo pode ser melhor do que O(ne2). No entanto, mesmo no melhor caso, a complexidade geralmente não é melhor do que O(n log n).

4.3.2 Pior Caso

No pior caso do Shell Sort, a escolha da sequência de gaps é tal que o algoritmo faz um grande número de comparações e trocas. Se a sequência de lacunas for O(ne2), a complexidade de tempo no pior caso é O(ne2).

4.3.3 Médio Caso

No caso médio, a complexidade do Shell Sort depende da sequência de lacunas escolhida. Para muitas sequências de gaps, a complexidade de tempo é aproximadamente O(ne1,5). Isso ocorre porque o Shell Sort melhora substancialmente a lista em cada iteração, mas ainda realiza algumas comparações amplas.

4.4 Selection Sort

Para o cálculo da complexidade do algoritmo Selection Sort, pode-se ver melhor o pseudo-codigo na tabela abaixo:

[htbp]

Passo	Descrição
1	Para i de 0 até o comprimento da lista - 1:
2	Encontrar o índice do menor elemento na lista não ordenada
3	Trocar o menor elemento com o primeiro elemento não ordenado
4	Fim Para

Portanto a formula de analise da complexidade é:

C(n) = (n.[n-1])/2

4.4.1 Melhor caso

No melhor caso, o algoritmo Selection Sort também executa n iterações do loop externo, mas no loop interno, ele só precisa fazer uma comparação para cada iteração. O motivo é que, no melhor caso, a lista já está ordenada, então o menor elemento está sempre no início da lista não ordenada.

Portanto, a fórmula para o número de comparações no melhor caso do algoritmo Selection Sort é:

C(n) = n

onde C(n) representa o número de comparações no melhor caso, e n é o número de elementos na lista. Neste cenário, o algoritmo faz n comparações, uma para cada elemento na lista.

Essa fórmula representa o melhor caso de complexidade de tempo do Selection Sort, que é linear no número de elementos na lista.

4.4.2 Pior caso

No pior caso, o algoritmo Selection Sort realiza n iterações do loop externo, como no melhor caso. No entanto, no loop interno, ele faz n1 comparações na primeira iteração, n2 comparações na segunda iteração e assim por diante, até fazer apenas uma comparação na última iteração. Isso é porque na segunda iteração ele já encontrou o segundo menor elemento, na terceira iteração o terceiro menor, e assim por diante.

Portanto, a fórmula para o número de comparações no pior caso do algoritmo Selection Sort é a soma dos primeiros n1 números naturais, que é dada pela fórmula da soma aritmética:

C(n) = n.(n-1) / 2

Nesta fórmula, C(n) representa o número de comparações no pior caso, e n é o número de elementos na lista. Esta fórmula representa o pior caso de complexidade de tempo do Selection Sort, que é quadrática no número de elementos na lista (O(ne2)).

4.4.3 Medio Caso

No caso médio, a análise de complexidade do Selection Sort é a mesma que no pior caso, pois o número de comparações e trocas permanece o mesmo, independentemente da disposição dos elementos na lista. Em cada iteração do loop externo, o algoritmo faz n1 comparações no loop interno, resultando em uma complexidade de tempo de O(ne2) para o caso médio. Isso se deve ao fato de que, em média, o algoritmo precisará fazer um número quadrático de operações para ordenar a lista.

Portanto, no caso médio, a fórmula para o número de comparações é a mesma que a do pior caso:

C(n) = n.(n-1) / 2

onde C(n) representa o número médio de comparações no caso médio, e n é o número de elementos na lista.

4.5 Quick Sort

Abaixo será realizado a analise de complexidade do algoritmo QuickSort para os 3 casos: Melhor Caso, Pior Caso, Medio Caso, que depende de o particionamento ser balanceado ou não balanceado.

4.5.1 Melhor Caso

Na divisão mais equitativa possivel, o metodo particiona produz dois subproblemas, cada um de tamanho não maior que n/2, já que um é de tamanho n/2 e o outro é de tamanho n/2 - 1. Nesse caso, a execução do quicksort é consideravelmente mais rápida. Então, a recorrência para o tempo de execução é:

$$T(n) = 2T(n/2) + \Theta(n)$$

Com a resolução dessa recorrencia sendo T(n) = O(n log n). Balanceando igualmente os dois lados da partição em todo nivel da recursão.

4.5.2 Medio Caso

O tempo de execução do caso medio do quicksort se assemelha muito com o do melhor caso. POr causa de que o equilibrio do particionamento é refletido na recorrência que descreve o tempo de execução. Portanto no caso medio o algoritmo ainda terá uma solução de T(n) = O(n log n).

4.5.3 Pior Caso

O comportamento do pior caso para o quicksort ocorre quando a rotina de particionamento produz um subproblema com n - 1 elementos e um com 0 elementos. Considerando de que o particionamento não balanceado surja em cada chamada recursiva. O particionamento custa T(n). Visto que a chamada recursiva para um arranjo de tamanho 0 apenas retorna, T(0) = Q(1) e a recorrencia para o tempo de execução é:

Provando a recorreência abaixo se obtem a solução T(n) = O(n^2).

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n). \end{aligned}$$

4.6 Merge Sort

4.6.1 Melhor, Medio e Pior Caso

A analise do algoritmo MergeSort é geralmente expressa em termos da complexidade temporal, e da quantidade de memoria adicional que será necessaria para a ordenação e em razão disso sua complexidade será a mesma para todos os casos.

O Merge Sort tem uma complexidade temporal de O(n log n), onde n é o numero de elementos na lista a ser ordenada. Assim tornando um algoritmo muito eficiente, especialmente para grandes conjuntos de dados. Esse eficiencia que se dá em razão da abordagem de divisão e conquista, que garante com que todas as comparações e operações de mesclagem sejam realizadas eficientimente. E a complexidade espacial que é O(n), o que significa que ele utiliza memoria proporcional ao numero de elementos nas lista, isso em razão de que durante a etapa da mesclagem, são criadas sublistas temporárias para o armazenamento temporario das divisões feitas apartir da lista original.

Portanto conclui-se de que o Merge sort tem uma complexidade de O(n log n) para todos os casos e de que é um algoritmo eficiente em termos de tempo, especialmente para listas com grandes quantidades de dados, porém consome espaço adicional visando garantir a estabilidade do algoritmo.

4.7 Heap Sort

4.7.1 Melhor, medio e pior Caso

A seguir será realizado a analise de complexidade do algoritmo Heapsort e todas suas funções derivadas, são elas o minHeapfy, buildMinHeap, heapMinimum, heapExtractMin, heapIncreaseKey e maxHeapInsert.

Primeiramente faremos a analise das funções buildMinHeap e minHeapfy, já que todas as outras funções fazem uso delas para a construção do heap.

Portanto na tabela abaixo é apresentado o algoritmo da função minHeapfy:

height**Linha**	**Código**	**Descrição**
1	`L <-- 2 * i`	Atribui o dobro do índice i à variável L.
2	`R <-- 2 * i + 1`	Atribui o dobro do índice i mais um à variável R.
3	`if i <= comprimento(A) e A[L] < A[R]`	Condição: se o índice i é válido e o elemento em $A[L]$ é menor que $A[R]$.
4	`then menor <-- L`	Atribui o valor de L à variável `menor`.
5	`else menor <-- i`	Caso contrário, atribui o valor de i à variável `menor`.
6	`if R <= comprimento(A) e A[R] < A[maior]`	Condição: se o índice R é válido e o elemento em $A[R]$ é menor que A[maior].
7	`then maior <-- R`	Atribui o valor de R à variável `maior`.
8	`if maior != i`	Condição: se `maior` não é igual a i.
9	`then swap(A[i] <-> A[maior])`	Troca os elementos $A[i]$ e A[maior].
10	`min_heapify(A, maior)`	Chama recursivamente a função `min_heapify` com o novo valor de `maior`.

Seja T(n) o tempo de execução para um heap de tamanho n, a etapa dividir ocorre com a determinação do maior entre A[i], A[2i] e A[2i+1], assim O(n) = O(1), pois são feitas duas comparações. A etapa de combinação não ocorre pois a cada passo o elemento selecionado é colocado em sua posição correta, portanto conclui-se de que a complexidade do verifica é O(n).

Para a analise da função buildMinHeap será apresentado o codigo da função abaixo:

Linha	Código
1	`tamanho_do_heap(A) <- comprimento(A)`
2	`for i <- comprimento[A]/2 down to 1`
3	`   do minHeapify(A, i)`

Devido a linha 2 do algoritmo, há O(n) chamadas ao procedimento da linha 3, quem tem tempo O(log n), portanto o tempo de execução é O(n log n).

E a seguir o algoritmo heapSort que possui o seguinte codigo apresentado na tabela abaixo:

Passo	**Código**	**Descrição**
1	`Build_min_heap(A)`	Construir um heap mínimo a partir de A.
2	`for i <-- comprimento(A) down to 2`	Loop de `comprimento(A)` até 2 (decrementando).
3	`swap(A[1] <-> A[i])`	Trocar os elementos `A[1]` e `A[i]`.
4	`comprimento(A) = comprimento(A) - 1`	Reduzir o `comprimento(A)` em 1.
5	`min_heapify(A, 1)`	Garantir que a propriedade de heap mínimo seja mantida em `A[1]`.

Nesse algoritmo devido a linha 2, há O(n) chamadas ao procedimento da linha 5, que possui O(log n). Portanto o tempo de execução do heapsort tambem é O(n log n). A principio o Heapsort não parece eficiente, devido as varias movimentações das chaves entretanto, a função minHeapfy gasta O(log n) operações no pior caso. Portanto o heapSort gasta O(n log n) no pior caso, medio caso e melhor caso.

Calculo de custo:

T(n) = C1(n) + C2(n - 1) + C3(n - 1) + C5(log n)

T(n) = C1n + C2n - C2 + C3n - C3 + C5log n

T(n) = n(C1 + C2 + C3) - (C2 + C3) + log n(C5)

Algumas observações sobre o heapSort é de que ele não é recomendado para arquivos com poucas entradas, em razão do tempo necessario para a construção de um heap. No pior caso, quando o ultimo nivel do heap tem a metade completa dos nós, temos aproximadamente 2/3

dos nós, sendo quando a última linha da arvore não está exatamente cheia, assim o tempo para a conquista é t(2/3n).

A seguir será feita a analise das funções HeapExtracMin, heapMinimum, heapIncreaseKey e maxHeapInsert. na tabela abaixo é apresentado o codigo da função heapExtractMin:

height**Número**	**Código**	**Ação**
1	`if comprimento(A) < 1`	Verifica se o comprimento de A é < 1
2	`then error "Heap under low"`	Emite um erro "Heap under low"
3	`minimum = A[1]`	Atribui o valor de `A[1]` a `minimum`
4	`A[1] = A[comprimento(A)]`	Atribui o valor de `A[comprimento(A)]` a `A[1]`
5	`comprimento(A) = comprimento(A) - 1`	Reduz o comprimento de A em 1
6	`minHeapfy(A, 1)`	Chama a função `minHeapfy` com os parâmetros A e 1
7	`return minimum`	Retorna o valor de `minimum`

Na função heapExtractMin o tempo de execução é O(log n) pois o algoritmo executa apenas uma porção constantte do trabalho sobre o O(log n) para o verifica.

Na tabela abaixo é demonstrado o codigo da função heapMinimum:

height**Número da Linha**	**Código**	**Comentário**
1	`minimum <-- A[1]`	Inicializa `minimum` com o valor no índice 1 de A.
2	`for i <-- comprimento(A) down to 1`	Loop de i igual ao comprimento de A até 1 (decrescente).
3	`L <-- 2 * i`	Calcula o índice à esquerda (L) usando a fórmula $2 \times i$.
4	`R <-- 2 * i + 1`	Calcula o índice à direita (R) usando a fórmula $2 \times i + 1$.
5	`if i <= comprimento(A) e A[L] < minimum`	Verifica se i é válido e se $A[L]$ é menor que `minimum`.
6	`then minimum = A[L]`	Atualiza `minimum` com o valor em $A[L]$.
7	`if i <= comprimento(A) e A[R] < minimum`	Verifica se i é válido e se $A[R]$ é menor que `minimum`.
8	`then minimum = A[R]`	Atualiza `minimum` com o valor em $A[R]$.
9	`return minimum`	Retorna o valor mínimo encontrado durante o loop.

Na função heapMinimum em razão do menor elemento da árvore se encontrar na raiz, a sua complexidade de tempo constante é O(1).

E a seguir na tabela abaixo é mostrado o codigo da função HeapIncreaseKey:

Passo	**Código**	**Ação**
1	`if chave < A[1]`	"Error"
2	`then "Error"`	
3	`A[i] <-- chave`	
4	`while i > 1 e A[parent[i]] > A[i]`	
5	`do swap(A[i] <-> A[parent[i]])`	Realiza a troca de elementos
6	`i <-- parent[i]`	Atualiza o índice i

Para essa função o tempo de execução sobre um heap de n elementos é O(log n), em razão de que o caminho traçado desde o nó raiz até a folha tem comprimento log n(altura da árvore). E por ultimo na tabela abaixo é apresentado o codigo da função MaxHeapInsert:

height**Número**	**Código**	**Descrição**
1	`comprimento(A) <-- comprimento(A) + 1`	Atualiza o comprimento do array A, adicionando 1.
2	`A[comprimento(A)] <-- -rand()`	Atribui um valor negativo aleatório à posição `comprimento(A)` do array A.
3	`heapIncreaseKey(A, comprimento(A), chave)`	Chama a função `heapIncreaseKey` no array A, com o índice `comprimento(A)` e a chave fornecida como parâmetro.

Devido a linha 3, com a chamada da função heapIncreaseKey a função MaxHeapIncrease possui um tempo de execução de execução sobre um heap de n elementos é O(log n).

Portanto conclui-se que, um heap pode admitir qualquer operação de uma fila de prioridade em um tempo de O(log n). E de que o algoritmo heapSort para todos os três casos(melhor, medio e pior caso) o algoritmo possui tempo de complexidade O(n log n).

5 TABELA E GRÁFICO

5.1 Insertion Sort

	10	100	1000	10000	100000	1.000.000
Crescente	0,0	0,0	0,0	0,0	0,0010	0,0050
Decrescente	0	0	0,002	0,124	0,118	1.204,25
Random	0	0	0,001	0,06	5,877	597,29

Figura 9: Tabela de tempo por segundo do algoritmo Insertion Sort

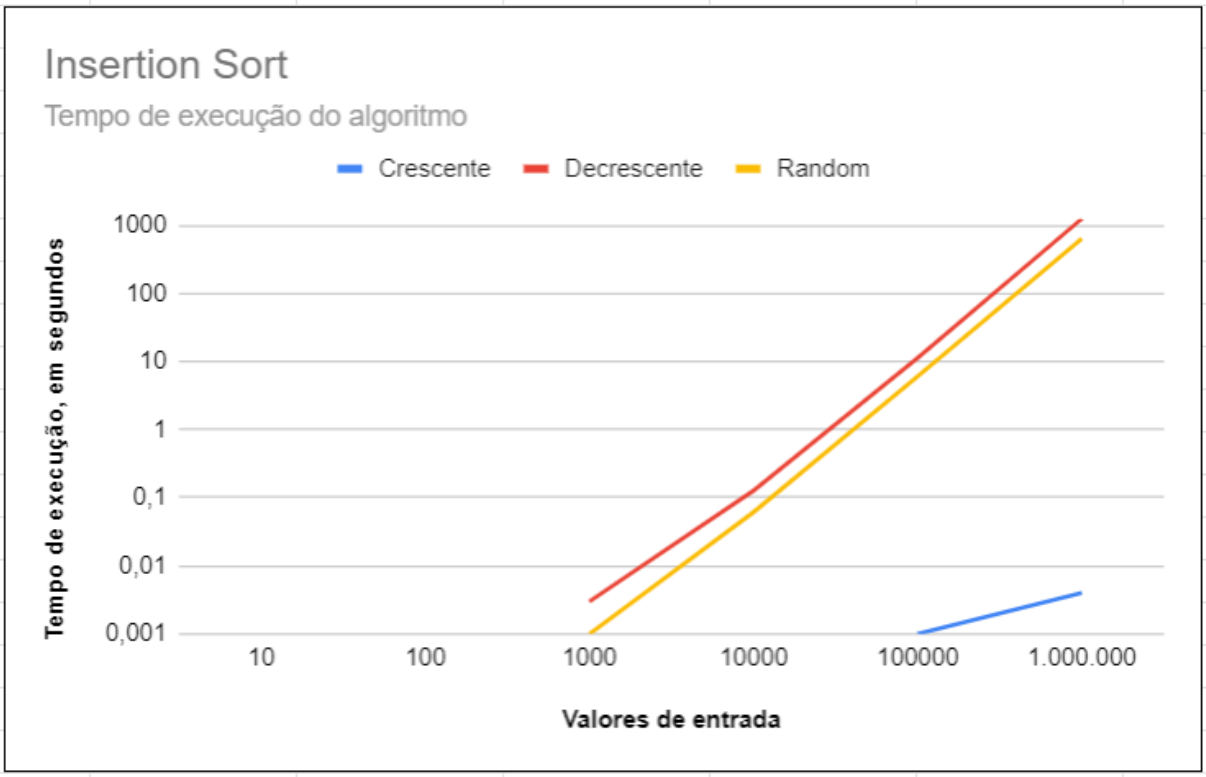


Figura 10: Gráfico de tempo por segundo do algoritmo Insertion Sort

5.2 Bubble Sort

	10	100	1000	10000	100000	1.000.000
Crescente	0	0	0,001	10,63	0,0010	1.072,11
Decrescente	0	0	0,006	0,48	11,190	47,981
Random	0	0	0,003	0,444	54,383	4.620,27

Figura 11: Tabela de tempo por segundo do algoritmo Bubble Sort

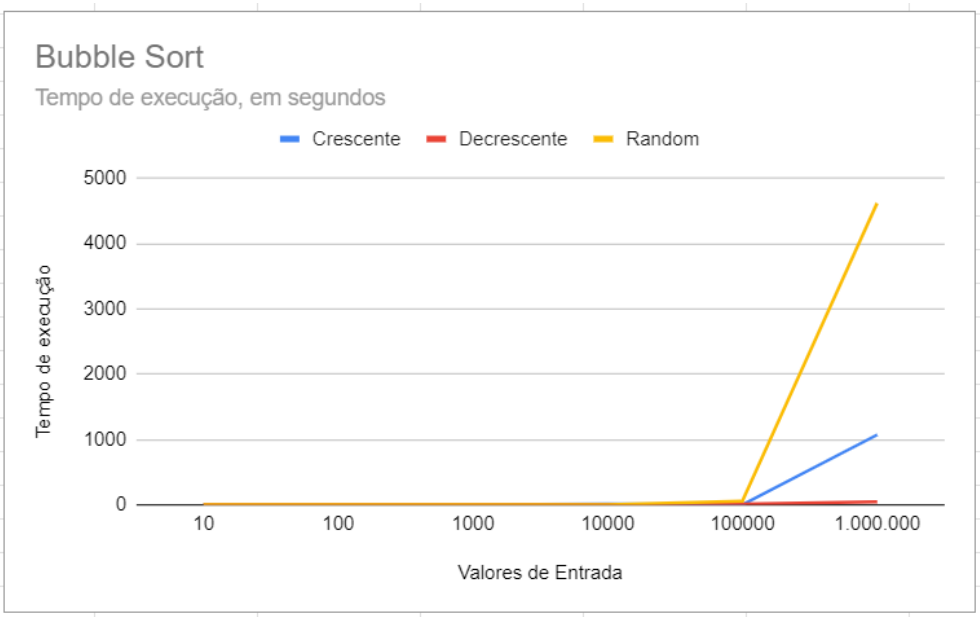


Figura 12: Gráfico de tempo por segundo do algoritmo Bubble Sort

5.3 Shell Sort

	10	100	1000	10000	100000	1000000
Crescente	1.602	1.858	2.403	2.681	3.042	3.230
Decrescente	3.000	3.103	3.668	4.645	4.745	5.371
Randomico	2.695	2.805	3.580	4.708	5.169	5.370

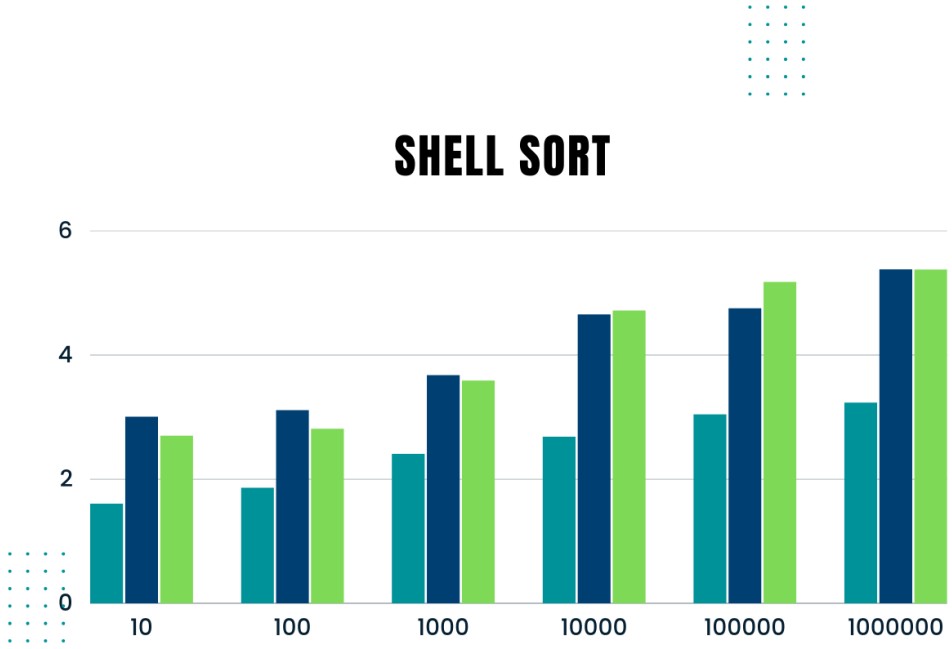


Figura 13: Grafico do Shell Sort

5.4 Selection Sort

	10	100	1000	10000	100000	1000000
Crescente	2.120	2.560	3.334	3.647	14.714	1155.8
Decrescente	2.430	2.833	4.435	4.525	15.217	1252.3
Randomico	8.881	9.010	9.510	9.812	17.521	1424.1

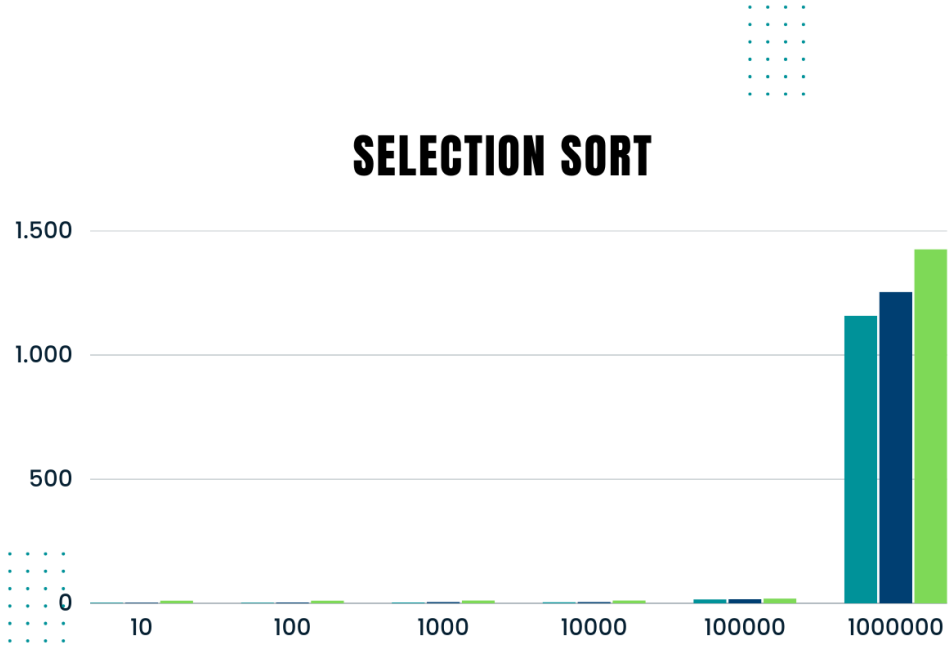


Figura 14: Grafico Sellection Sort

5.5 Quick Sort Randon

	10	100	1000	10000	100000	1000000
Decrescente	0	0	0	157	14573	2034278
Crescente	0	0	6	91	9145	1210033
Randomico	0	0	0	0	31	235

5.6 Quick Sort Primeiro Elemento

	10	100	1000	10000	100000	1000000
Decrescente	0	0	0	0	15	297
Crescente	0	0	0	16	15	251
Randomico	0	0	0	0	31	295

5.7 Quick Sort Media

5.8 Quick Sort Mediana

	10	100	1000	10000	100000	1000000
Decrescente	0	0	0	12	15	218
Crescente	0	0	0	0	15	140
Randomico	0	0	0	0	14	313

5.9 Merge Sort

	10	100	1000	10000	100000	1000000
Decrescente	0	0	0	0	15	203
Crescente	0	0	0	15	15	187
Randomico	0	0	0	0	15	310

5.10 Heap Sort

Heap Sort	10	100	1000	10000	100000	1000000
Randomico	0	0	3,5	5	40	250
Crescente	0	0	4	7	32	225
Decrescente	0	0	6	6	37	282

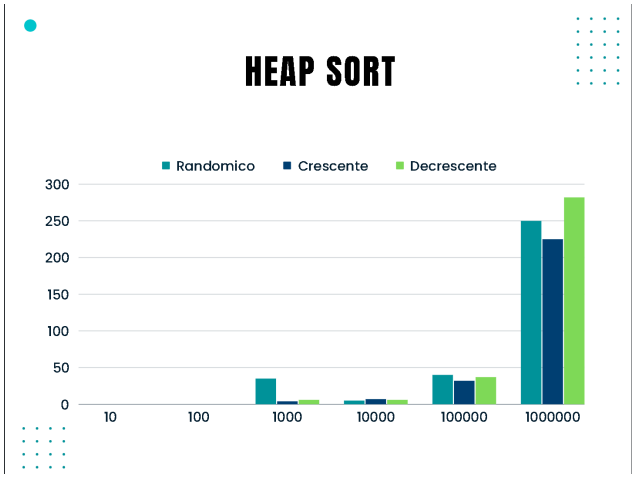


Figura 15: Heap Sort

Obs: Todos os graficos e tabelas estão em medida de segundos.

5.11 Todos Algorimos

No grafico a seguir terá a comparação de todos os algoritmos de ordenação para que possa demonstrar de forma mais visivel a diferença na performance de todos os algoritmos:

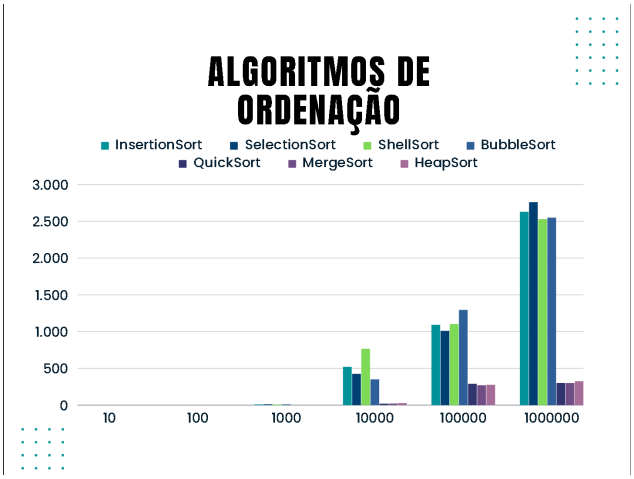


Figura 16: Algoritmos de ordenação

6 CONCLUSÃO

A partir dos Dados mostrados nas tabelas e graficos dos algoritmos Insertion Sort, Bubble Sort, Shell Sort e Insertion Sort, Quick Sort em suas 4 versões, Merge Sort e o HeapSort e suas funções derivadas. Nas ordens Randomica, Crescente e Decrescente, com entradas de 10, 100, 1000, 10000, 100000, 1000000 e com os estudos explicativos.

Percebe-se de que o algoritmo que tem melhor desempenho nas diversas entradas apresentadas é o Algoritmo Quick Sort, que por meio da utilização do metodo de divisão e conquista, e com a escolha do pivo otimizada para a situação em que está sendo utilizado, mesmo com entradas maiores seu tempo de processamento é baixo e satisfatorio, além disso também tem um bom desempenho para entradas menores. Porém outros algoritmos também conseguem apresentar resultados satifatorios dependendo da situação em que são utilizados, como exemplo o algoritmo heapSort que é recomendado para programas com entradas maiores, mas que em contra-partida não é muito recomendado para entradas pequenas.

Conclui-se de que para a ordenação de entradas, se tem diversos algoritmos que podem ser utilizados, com algoritmos como o QuickSort, MergeSort e Heapsort apresentando os melhores resultados. Porém há outros fatores que determinam a qual utilizar. Estão entre eles: sua aplicação, como exemplo o HeapSort que pode ser utilizado para filas de prioridade, da situação em que serão usados, como exemplo o tamanho da entrada, que caso se um programa de entradas pequenas o HeapSort já não é a melhor escolha, tendo outros que

apresentão performances melhores nessa situação, entre outros.

7 REFERÊNCIAS BIBLIOGRÁFICAS

Referências

[Ziviani et al.(2004)] Nivio Ziviani et al. *Projeto de algoritmos: com implementações em Pascal e C*, volume 2. Thomson Luton, 2004.