SUMÁRIO

1 MÉTODOS INFERIORES

Neste capítulo, são apresentados quatro algoritmos básicos para o problema da ordenação. Esses algoritmos são chamados de inferiores por serem simples, intuitivos e, principalmente, por apresentarem, no mínimo, complexidade temporal quadrática nos casos médio e pior.

1.1 Bolha

O algoritmo Bolha é um dos algoritmos mais simples para o problema da ordenação. Ele consistem em percorrer várias vezes o vetor, comparando elementos adjacentes e trocando-os de posição caso estejam fora da ordem desejada. Nesse processo, os elementos flutuam para suas posições finais. Em outras palavras, "[...] os elementos da lista são movidos para as posições adequadas de forma contínua, assim como uma bolha move-se num líquido. Se um elemento está inicialmente numa posição i e, para que a lista fique ordenada, ele deve ocupar a posição j, então ele terá que passar por todas as posições entre i e j." (VIANA; CINTRA, 2011, p.8).

```
1  void Bolha(int n, int *v) {
2      char trocou = 1;
3      for (int i = n - 1; i > 0 && trocou; i--) {
4          trocou = 0;
5          for (int j = 0; j < i; j++)
6              if (v[j] > v[j + 1]) {
7                  Troca(v + j, v + j + 1);
8                  trocou = 1;
9              }
10     }
11 }
```

Esta versão do algoritmo Bolha é também conhecida como "Bolha com Flag", por conta do uso da variável *trocou*. Perceba que, sem essa variável, o algoritmo continua funcionando da mesma forma; porém o laço externo executará sempre $n-1$ iterações, mesmo que o vetor fique ordenado antes.

Observe que o algoritmo Bolha oferece uma ordenação estável, pois, quando dois elementos adjacentes são iguais, eles não são trocados.

Corretude

Ao final de uma iteração do laço externo, o maior elemento de $v[0..i]$ estará na posição i. Dessa forma, ao final da primeira iteração, o maior elemento do vetor estará na posição $n-1$; ao final da segunda iteração, o segundo maior estará na posição $n-2$, e assim por diante. Portanto, o algoritmo Bolha é correto.

Desempenho

Em termos de consumo de memória adicional, o algoritmo declara apenas três variáveis escalares, sendo duas delas para controle de laço de repetição. Logo, sua complexidade espacial é $\Theta(1)$. Esta observação será omitida para os algoritmos apresentados a seguir que tenham a mesma complexidade espacial.

Já para a complexidade temporal, o melhor caso ocorre quando a entrada já está ordenada em ordem crescente, pois não é realizada nenhuma troca. Exige-se, assim, apenas uma iteração do laço externo, resultando em complexidade $\Theta(n)$.

Por outro lado, o pior caso ocorre quando o menor elemento está na última posição, sendo necessárias exatamente $n-1$ iterações do laço externo para que ele chegue à primeira posição. Com isso, a quantidade de iterações do laço interno é dada pela seguinte soma:

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \tag{1.1}$$

Isso implica em uma complexidade quadrática $\Theta(n^2)$. Em média, a complexidade temporal do algoritmo Bolha é $\mathcal{O}(n^2)$.

1.2 Coquetel

O Coquetel é um algoritmo de ordenação estável que traz a mesma ideia do Bolha, mas com um laço interno adicional. O primeiro laço, assim como no Bolha, itera da esquerda para a direita, empurrando os maiores elementos para o final do vetor, enquanto o segundo itera no outro sentido, empurrando os menores elementos para o início.

```
1 void Coquetel(int n, int *v) {
2     char trocou = 1;
3     for (int i = 0, j = n - 1; i < j && trocou; i++, j--) {
4         trocou = 0;
5         for (int k = i; k < j; k++)
```

```
6               if (v[k] > v[k + 1])
7                   Troca(v + k, v + k + 1), trocou = 1;
8           if (trocou) trocou = 0;
9           else return;
10          for (int k = j - 2; k >= i; k--)
11              if (v[k] > v[k + 1])
12                  Troca(v + k, v + k + 1), trocou = 1;
13      }
14  }
```

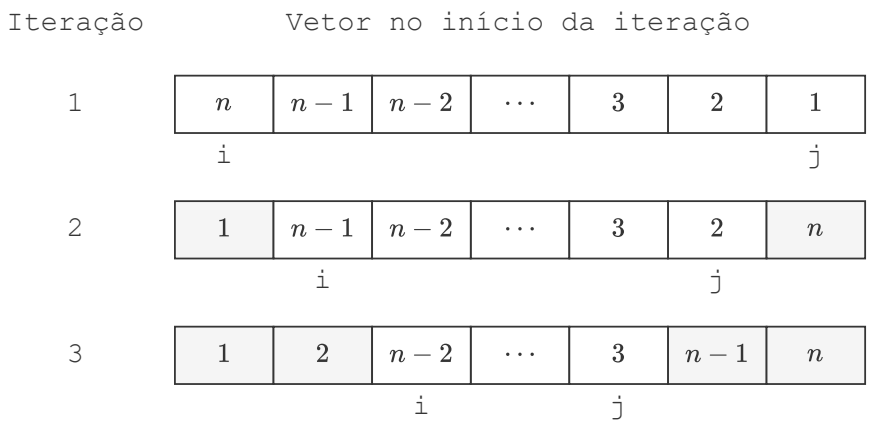
Corretude

Ao final de uma iteração do laço externo, o menor e o maior elemento de $v[i..j]$ estarão nas posições i e j, respectivamente. Dessa forma, ao final da primeira iteração, o menor elemento estará na posição 0 e o maior, na posição $n-1$; na segunda, o segundo menor, na posição 1, e o segundo maior, na posição $n-2$, e assim por diante. Esse processo garante que, quando $i \geq j$, o vetor estará ordenado; portanto, o algoritmo é correto.

Desempenho

Embora as complexidades dos casos melhor, pior e médio continuem sendo, respectivamente, $\Theta(n)$, $\Theta(n^2)$ e $\mathcal{O}(n^2)$, o pior caso do Coquetel é menos frequente, pois ocorre apenas quando a entrada está em ordem decrescente.

Figura 1 – Coquetel aplicado ao vetor $[n, n-1, ..., 2, 1]$.



Fonte: Elaborado pelo autor.

Considere então que $T(n)$ é a quantidade total de iterações dos dois laços internos

no pior caso. A figura 1 ajuda a notar que a seguinte identidade é válida:

$$T(n) = \begin{cases} 0, & n < 2 \\ T(n-2) + 2n - 3, & n \geq 2 \end{cases}$$

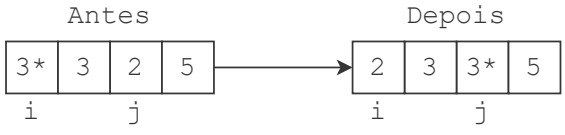
Ao resolver a recorrência mostrada acima, pode-se observar que $T(n)$ é igual a soma 1.1. Portanto, os algoritmos Bolha e Coquetel, em seus piores casos, executam exatamente a mesma quantidade de iterações.

1.3 Seleção

O algoritmo Seleção é um método de ordenação instável, como mostra a figura 2. A ideia é bastante simples: execute n iterações; na i-ésima iteração, selecione o i-ésimo menor elemento e coloque-o na posição $i-1$. Uma propriedade interessante desse algoritmo é que ele realiza o número mínimo possível de trocas para ordenar o vetor.

```
1 void Selecao(int n, int *v) {
2     for (int i = 0; i < n - 1; i++) {
3         int j = i;
4         for (int k = i + 1; k < n; k++)
5             if (v[j] > v[k])
6                 j = k;
7         if (j > i) Troca(v + i, v + j);
8     }
9 }
```

Figura 2 – Seleção aplicado ao vetor $[3,3,2,5]$.



Fonte: Elaborado pelo autor

Corretude

É invariante que, ao final de cada iteração do laço externo, os elementos de $v[0..i]$ já estejam em suas posições definitivas. Portanto, ao final da última iteração, quando i for igual a $n-1$, o vetor $v[0..n-1]$ estará ordenado.

Desempenho

O ponto crítico do algoritmo Seleção é o teste condicional *if* na linha 5. A quantidade de vezes que esse ponto é executado é sempre dada pela soma 1.1, ou seja, não há melhor nem pior caso. Portanto, esse algoritmo sempre requer tempo igual a $\Theta(n^2)$.

1.4 Inserção

O algoritmo Inserção é um método de ordenação estável com o melhor desempenho entre os métodos inferiores. A ideia por trás do algoritmo é simples: para i variando de 1 a $n-1$, nesta ordem, insira $v[i]$ em $v[0..i-1]$, de modo que $v[0..i]$ esteja em ordem.

Em especial, o algoritmo Inserção possui complexidade temporal linear para entradas quase ordenadas. Por conta disso, outros algoritmos fazem uso dele como sub-rotina. O Shellsort, por exemplo, utiliza o Inserção para tornar o vetor h-ordenado.

Definição 1.1 *Para um dado número natural $h > 0$, um vetor v de tamanho n é considerado h-ordenado quando $v[i] \leq v[i+h]$, para todo $0 \leq i < n-h$.*

```
1 void H_Ordena(int n, int h, int *v) {
2     int chave, j;
3     for (int i = h; i < n; i++) {
4         chave = v[i];
5         for (j = i; j >= h && v[j - h] > chave; j -= h)
6             v[j] = v[j - h];
7         v[j] = chave;
8     }
9 }
```

A implementação do Inserção pode ser idêntica à função H_Ordena mostrada acima, bastando fazer $h = 1$, pois, nesse caso em particular, um vetor h-ordenado é simplesmente um vetor ordenado.

```
1 void Insercao(int n, int *v) {
2     H_Ordena(n, 1, v);
3 }
```

Corretude

Ao final de cada iteração do laço externo, o segmento $v[0..i]$ está ordenado. Esse invariante permanece válido durante todas as iterações, inclusive na última. Portanto, ao final do processo, $v[0..n-1]$ estará ordenado.

Desempenho

No melhor caso, a complexidade é linear e ocorre quando o vetor já está ordenado, pois a condição do laço interno será sempre falsa. Já o pior caso acontece quando o vetor está em ordem decrescente; nesse cenário, $v[i]$ é sempre inserido na posição 0. O ponto crítico é a linha 6 e, mais uma vez, a soma 1.1 determina a quantidade de vezes que essa linha será executada. Portanto, a complexidade no pior caso é $\Theta(n^2)$. Em média, o algoritmo Inserção consome tempo proporcional a $\mathcal{O}(n^2)$.

2 MÉTODOS SUPERIORES

2.1 Shellsort

O Shellsort foi desenvolvido por Donald L. Shell e apresentado em 1959 no artigo *A High-Speed Sorting Procedure*. Trata-se de um método de ordenação não estável que se baseia na escolha de uma sequência de incrementos h e, para cada valor h_i, do maior para o menor, transforma o vetor de entrada em um vetor h_i-ordenado.

Definição 2.1 *No contexto do algoritmo Shellsort, uma sequência de incrementos é uma sequência de números inteiros positivos* $h = \{h_1, h_2, ..., h_r\}$*, tal que* $h_1 = 1$ *e* $h_i < h_{i+1}$*.*

A análise de complexidade do Shellsort não é uma tarefa simples e depende da sequência de incrementos escolhida. Diversos pesquisadores propuseram diferentes sequências ao longo dos anos; algumas das mais conhecidas são mostradas na Tabela 1.

Tabela 1 – Sequências de incrementos

Referência	Sequência	Pior caso
(SHELL, 1959)	$h_i = \lfloor n/2^i \rfloor$	$\Theta(n^2)$
(HIBBARD, 1963)	$h_i = 2^i - 1$	$\Theta(n^{3/2})$
(KNUTH, 1973)	$h_i = (3^i - 1)/2$	$\Theta(n^{3/2})$
(SEDGEWICK, 1986)	$h_i = 4^i + 3 \times 2^{i-1} + 1$	$\mathcal{O}(n^{4/3})$
(TOKUDA, 1992)	$h_i = \left\lceil \frac{9(9/4)^i - 4}{5} \right\rceil$	-
(CIURA, 2001)	$1, 4, 10, 23, 57, 132, 301, 701, 1750$	-

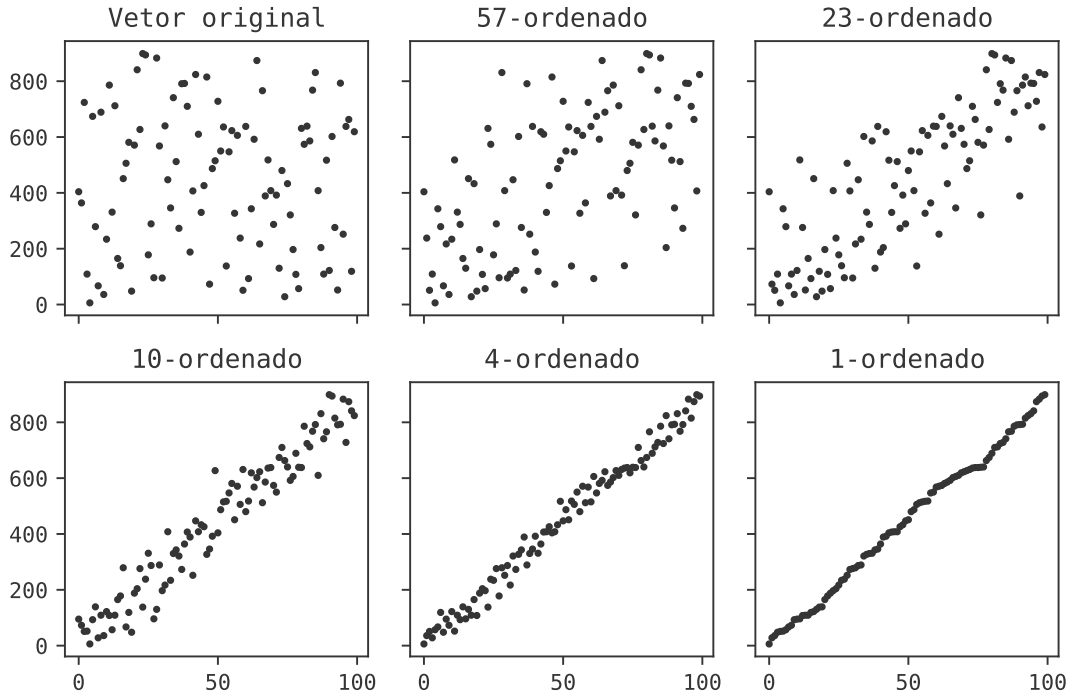
Neste trabalho é utilizada a sequência proposta por (CIURA, 2001). Tal sequência não possui uma análise precisa de complexidade de pior caso, mas demonstrou um bom desempenho experimentalmente, superando todas as outras. Os primeiros oito termos da sequência, escolhidos empiricamente pelo autor, são os mostrados na Tabela 1. Para expandir a sequência, uma abordagem comum é seguir a fórmula:

$$h_i = \lfloor 2{,}25 \times h_{i-1} \rfloor$$

Desta forma, para a implementação do Shellsort mostrada em linguagem C abaixo, considere que o vetor *CiuraSeq* foi criado de forma global e contém os primeiros 24 termos da sequência descrita acima, em ordem decrescente.

```
1 void Shellsort(int n, int *v) {
2     for (int i = 0; i < 24; i++)
3         H_Ordena(n, CiuraSeq[i], v);
4 }
```

Figura 3 – Estado de um vetor durante iterações do Shellsort.



Fonte: Elaborado pelo autor

2.2 Mergesort

O Mergesort é um algoritmo de ordenação estável baseado na técnica de divisão e conquista, proposto por John von Neumann em 1945. Essa técnica consiste em um processo recursivo de três etapas:

1. Divisão: divide-se o problema em instâncias menores do mesmo problema.
2. Conquista: resolve-se cada uma das instâncias definidas na etapa anterior.
3. Combinação: combinam-se as soluções para resolver o problema original.

Enquanto as etapas de divisão e conquista consistem apenas em algumas operações aritméticas e duas chamadas recursivas, a de combinação requer um pouco mais de atenção. Nesse momento, tem-se dois vetores ordenados, e é necessário juntá-los em apenas um vetor também ordenado, em um processo chamado intercalação, que aqui será referido pelo termo em inglês *merge*.

Intercalação de vetores ordenados

A implementação da operação *merge* é mostrada em linguagem C abaixo. Ela assume que $v[l..m-1]$ e $v[m..r-1]$ estão ordenados. O resultado é que $v[l..r-1]$ também ficará em ordem. A função CopiaVetor, na linha 2, copia os $r-l$ elementos de $v[l..r-1]$ para o vetor auxiliar $v\_aux[l..r-1]$. Doravante, essa função será usada de forma similar, sem mais explicações.

```
1 void Merge(int l, int m, int r, int *v, int *v_aux) {
2     CopiaVetor(r - l, v + l, v_aux + l);
3     int i = l, j = m, k = l;
4     while (i < m && j < r) {
5         if (v_aux[i] <= v_aux[j]) v[k++] = v_aux[i++];
6         else v[k++] = v_aux[j++];
7     }
8     while (i < m) v[k++] = v_aux[i++];
9 }
```

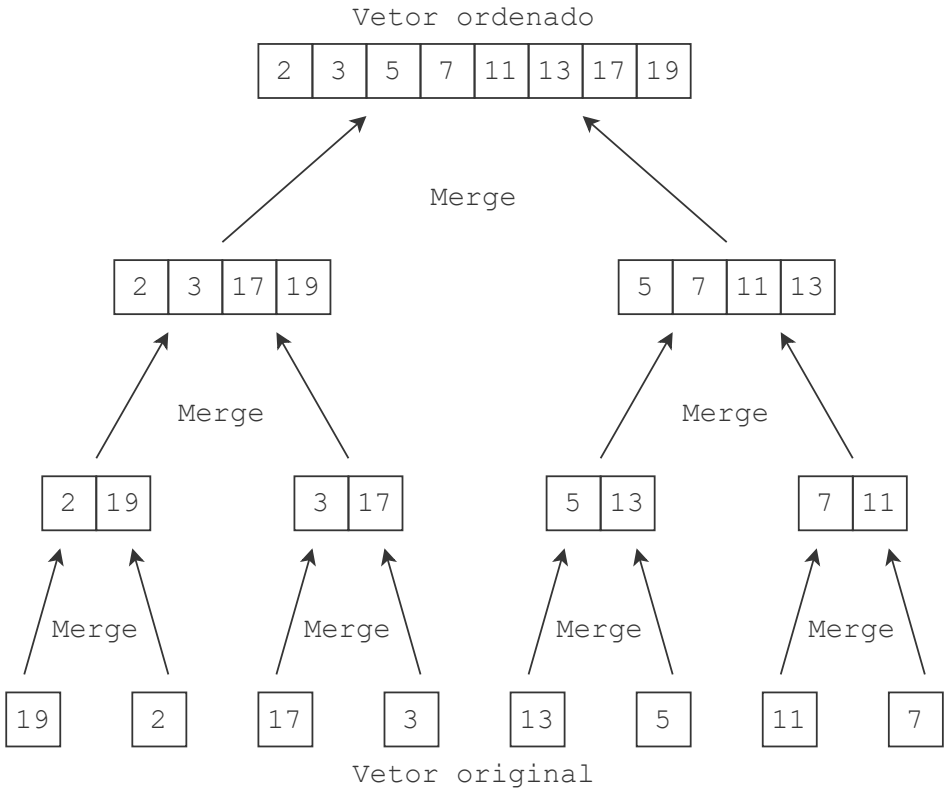
O algoritmo Mergesort

Este trabalho traz uma implementação dividida em dois métodos. O primeiro é apenas uma interface que segue o mesmo padrão de assinatura dos métodos de ordenação mostrados anteriormente, e é responsável por alocar e liberar a memória auxiliar, além de invocar o segundo método. O segundo é uma implementação recursiva do algoritmo que executa as três etapas da técnica já mencionada. A função AlocaVetor usa malloc para alocar um vetor com o tamanho especificado e verifica se a alocação ocorreu com sucesso.

```
1 void Mergesort(int n, int *v) {
2     int *v_aux = AlocaVetor(n);
3     MergesortRec(0, n, v, v_aux);
4     free(v_aux);
```

```
5  }
6  void MergesortRec(int l, int r, int *v, int *v_aux) {
7      if (l < r - 1) {
8          int m = (l + r) / 2;
9          MergesortRec(l, m, v, v_aux);
10         MergesortRec(m, r, v, v_aux);
11         Merge(l, m, r, v, v_aux);
12     }
13 }
```

Figura 4 – Ilustração do Mergesort aplicado ao vetor $[19, 2, 17, 3, 13, 5, 11, 7]$.



Fonte: Elaborado pelo autor

Corretude

Observe que a base da recursão está bem definida, pois um vetor com menos de dois elementos já está ordenado. Note também que, em cada chamada, o vetor é dividido aproximadamente ao meio, e cada metade é ordenada recursivamente, sem deixar lacunas antes, entre ou depois de cada uma. Por fim, veja que a função Merge copia $v[l..r-1]$ para $v\_aux[l..r-1]$ e depois traz os elementos de volta para v, um por um, sempre escolhendo

o menor que ainda não foi copiado. Como cada uma das etapas está correta, o Mergesort também está correto.

Desempenho

Seja $T(n)$ o tempo necessário para ordenar um vetor de n elementos utilizando o Mergesort. Temos então que:

$$T(n) = \begin{cases} \Theta(1), & n = 1 \\ \underbrace{\Theta(1)}_{\text{Divisão}} + \underbrace{2T(n/2)}_{\text{Conquista}} + \underbrace{\Theta(n)}_{\text{Intercalação}}, & n > 1 \end{cases}$$

Desta forma, podemos simplesmente dizer que para $n > 1$ e alguma constante c,

$$T(n) = 2T(n/2) + cn \quad \Rightarrow \quad T(n) = n\log_2 n + cn$$

Portanto, o consumo de tempo do Mergesort é $\Theta(n \log_2 n)$. Já o de memória é $\Theta(n)$, devido ao vetor auxiliar utilizado.

2.3 Heapsort

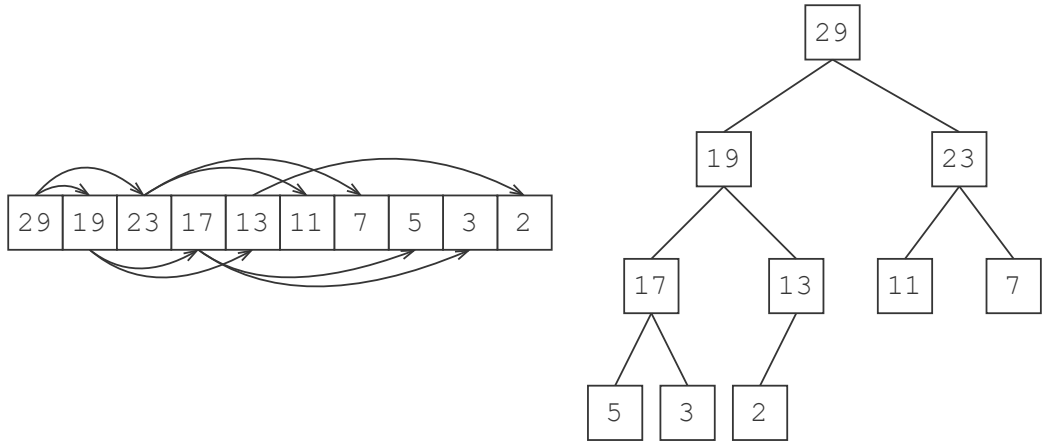
O algoritmo Heapsort foi introduzido por J. W. J. Williams em 1964, no artigo *Algorithm 232: Heapsort*, publicado na revista *Communications of the ACM*. Como o próprio nome sugere, este algoritmo faz uso de uma estrutura de dados chamada *heap*, mais especificamente um *max-heap* binário.

max-heap binário

Um *max-heap* binário – de agora em diante referido apenas como *heap* – é uma árvore binária quase completa que satisfaz a propriedade de que o valor armazenado em um nó pai é maior ou igual ao valor de seus filhos. Uma árvore binária é considerada quase completa quando todos os seus níveis estão completamente preenchidos, exceto possivelmente o último, que deve ser preenchido da esquerda para a direita.

Convenientemente, essa árvore pode ser implementada em um vetor, de modo que cada posição do vetor represente um nó da árvore. Os filhos do nó na posição i estão localizados nas posições $2i+1$ e $2i+2$, desde que essas posições sejam válidas.

Figura 5 – Exemplo de max-heap binário.



Fonte: Elaborado pelo autor.

Função Heapifica

A função Heapifica será usada tanto para construir o *heap* quanto para implementar o Heapsort em si. Ela é responsável por restaurar a propriedade de *heap* em um "quase *heap*". Um "quase *heap*" é uma árvore em que apenas a raiz pode estar violando a propriedade de *heap*. A ideia é simples: começando pela raiz, compara-se o nó atual com seu filho de maior valor; se o filho for maior, os dois são trocados de posição, e o processo é repetido com o novo nó até que a condição seja falsa ou uma folha seja atingida.

```
1  void Heapifica(int i, int n, int *v) {
2      int chave = v[i], j = 2 * i + 1;
3      while (j < n) {
4          if (j < n - 1 && v[j] < v[j + 1]) j++;
5          if (chave >= v[j]) break;
6          v[i] = v[j];
7          i = j;
8          j = 2 * j + 1;
9      }
10     v[i] = chave;
11 }
```

Construção do heap

Considere o problema de transformar um vetor em um *heap*. Para resolvê-lo, basta aplicar a operação Heapifica a todos os nós que não são folhas, da direita para a esquerda.

```
1 void ConstroiHeap(int n, int *v) {
2     for (int i = n / 2 - 1; i >= 0; i--)
3         Heapifica(i, n, v);
4 }
```

O algoritmo Heapsort

O Heapsort começa transformando o vetor de entrada em um *heap*. Após isso, o maior elemento estará na primeira posição. Sabendo disso, $v[0]$ é trocado com $v[n-1]$. Neste momento, o maior elemento está na posição correta, e $v[0..n-2]$ forma um "quase *heap*". Em seguida, a função Heapifica transforma $v[0..n-2]$ em um *heap*, e esse processo é repetido até que o *heap* tenha apenas um elemento.

```
1 void Heapsort(int n, int *v) {
2     ConstroiHeap(n, v);
3     for (int i = n - 1; i > 0; i--) {
4         Troca(v, v + i);
5         Heapifica(0, i, v);
6     }
7 }
```

Desempenho

A quantidade de operações que a função Heapifica executa quando invocada em um nó de altura h é proporcional a h. Além disso, existem exatamente $2^M/2^h$ nós com altura $h > 1$, onde $M = \lfloor \log_2 n \rfloor + 1$. Logo, a quantidade total de operações na construção do *heap* é dada por:

$$\sum_{h=2}^{M} h\left(\frac{2^M}{2^h}\right) \;=\; 2^M \cdot \sum_{h=2}^{M} h\left(\frac{1}{2}\right)^h \;<\; 2^M \cdot \sum_{h=1}^{\infty} h\left(\frac{1}{2}\right)^h \;=\; 2^{M+1} \;\approx\; 2n$$

Portanto, o *heap* é construído em tempo linear, $\mathcal{O}(n)$. Já a segunda parte do algoritmo, a partir da linha 3, executa aproximadamente

$$\sum_{i=1}^{n-1} \log_2 i \;=\; \log_2 (n-1)! \;\leq\; n \log_2 n$$

operações e, portanto, todo o algoritmo consome tempo proporcional a $\mathcal{O}(n \log_2 n)$.

2.4 Quicksort

O Quicksort é um algoritmo de ordenação que provavelmente é o mais amplamente utilizado. A versão básica foi desenvolvida em 1960 por C. A. R. Hoare e, desde então, tem sido estudado por muitas pessoas. O Quicksort é popular porque não é difícil de implementar, é um bom algoritmo de ordenação de uso geral e, em muitos casos, consome menos recursos do que qualquer outro método de ordenação (SEDGEWICK, 1990, p.115, tradução nossa).

O algoritmo Quicksort é um método de ordenação não estável que funciona da seguinte maneira: particiona-se o vetor em dois segmentos, de modo que os menores elementos fiquem no primeiro e os maiores no segundo. Em seguida, o processo é repetido recursivamente em cada um dos segmentos, enquanto eles tiverem tamanho maior que um.

A escolha de um elemento pivô define quais elementos são considerados menores e quais são considerados maiores. As estratégias para escolha do pivô, bem como o processo de particionamento, são apresentados a seguir.

Particionamento

Existem versões básicas do Quicksort que definem como pivô um elemento localizado em um índice fixo do intervalo, como, por exemplo, sempre escolher um dos extremos. Essa estratégia pode levar à complexidade $\mathcal{O}(n^2)$ para entradas que já estejam ordenadas, além de empilhar cerca de n chamadas recursivas, o que pode causar um estouro de pilha. Contudo, a versão do Quicksort apresentada neste trabalho já incorpora algumas otimizações. A função Particiona, apresentada a seguir, escolhe o pivô de forma pseudoaleatória.

```
1  int Particiona(int l, int r, int *v) {
2      Troca(v + r, v + GeraNumeroAleatorioNoIntervalo(l, r));
3      int pivo = v[r], j = l;
4      for (int i = l; i < r; i++)
5          if (v[i] <= pivo) {
6              Troca(v + j, v + i);
7              j++;
8          }
9      v[r] = v[j]; v[j] = pivo;
10     return j;
11 }
```

Na implementação da função Particiona mostrada acima, a função GeraNumeroAleatorioNoIntervalo utiliza a função *rand*, presente na biblioteca padrão da linguagem C. O retorno da função Particiona é um índice j tal que

$$l \leq j \leq r \quad \text{e} \quad \begin{cases} v[j] = piv\hat{o} \\ v[k] \leq piv\hat{o}, \quad l \leq k < j \\ v[k] > piv\hat{o}, \quad j < k \leq r \end{cases}$$

O algoritmo Quicksort

A implementação do algoritmo Quicksort apresentada a seguir é idêntica àquela encontrada em (FEOFILOFF, 2009, p.90). Ela traz duas melhorias em relação à implementação básica. A primeira consiste em sempre tratar primeiro o menor dos segmentos gerados pela função Particiona, garantindo que a altura da pilha de recursão seja, no máximo, $\log_2 n$. A segunda é a eliminação de uma das chamadas recursivas, que é transformada em uma iteração, convertendo o algoritmo em uma recursão de cauda.

```
1  void Quicksort(int n, int *v) {
2      QuicksortRec(0, n - 1, v);
3  }
4  void QuicksortRec(int l, int r, int *v) {
5      while (l < r) {
6          int i = Particiona(l, r, v);
7          if (i - l < r - i) {
8              QuicksortRec(l, i - 1, v);
9              l = i + 1;
10         } else {
11             QuicksortRec(i + 1, r, v);
12             r = i - 1;
13         }
14     }
15 }
```

Desempenho

O pior caso do Quicksort tem complexidade $\mathcal{O}(n^2)$ e ocorre quando a função Particiona retorna sempre, ou quase sempre, um índice j muito próximo de l ou r. Isso

acontece, por exemplo, quando todos os elementos do vetor são iguais. Já o melhor caso tem complexidade $\mathcal{O}(n \log_2 n)$ e ocorre quando j está próximo do ponto médio entre l e r em todas as chamadas recursivas.

Em média, a complexidade desta versão aleatorizada do Quicksort é proporcional a $\mathcal{O}(n \log_2 n)$. Esse resultado pode ser demonstrado por meio do cálculo do valor esperado do número de comparações realizadas na linha 5 da função Particiona, conforme apresentado em (CORMEN *et al.*, 2009, p.180–184). Para isso, observe que a probabilidade de os elementos $v[i]$ e $v[j]$ serem comparados durante a execução do algoritmo é dada por:

$$P_{ij} = \frac{2}{j - i + 1}$$

Seja X a variável aleatória que representa a quantidade total de comparações feitas. Assim, sendo $k = j - i$, o valor esperado de X é:

$$\begin{aligned}
E[X] &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} \frac{2}{j - i + 1} \\
&= \sum_{i=0}^{n-2} \sum_{k=1}^{n-i-1} \frac{2}{k+1} \\
&< \sum_{i=0}^{n-2} \sum_{k=1}^{n} \frac{2}{k} \\
&< 2(n-1) \ln n \\
&= \left(\frac{2(n-1)}{\log_2 \mathrm{e}} \right) \log_2 n
\end{aligned}$$

2.5 Métodos híbridos

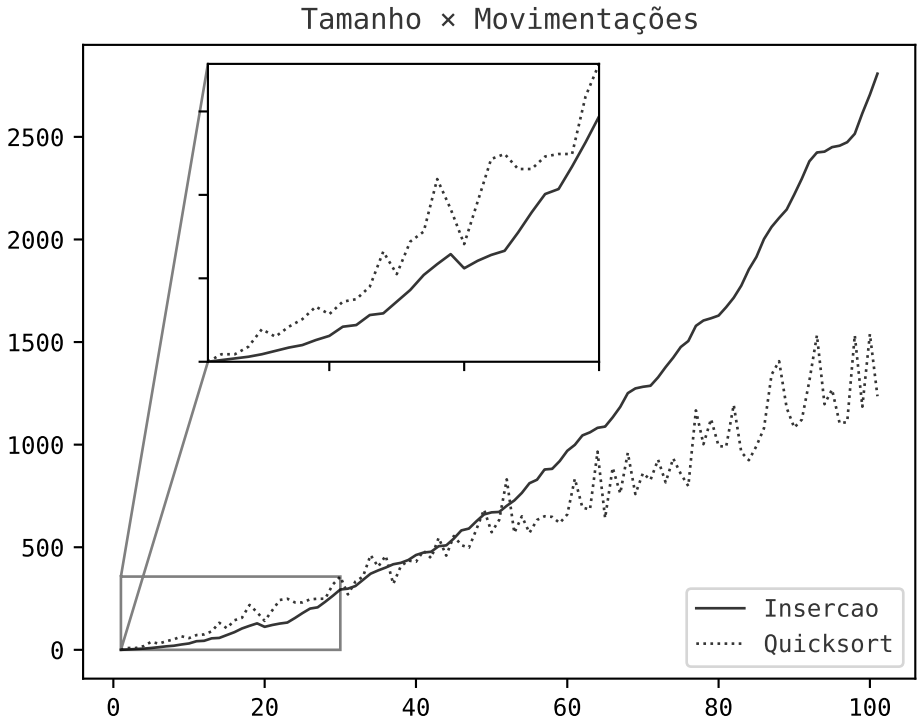
Esta seção apresenta dois algoritmos de ordenação híbridos, assim chamados por combinarem dois ou mais dos algoritmos discutidos anteriormente.

2.5.1 Quicksort com Inserção

Este algoritmo combina a eficiência do Quicksort para vetores grandes com a do Insercao para vetores pequenos e quase ordenados, como ilustra a Figura 6. Ele consiste em duas etapas: a primeira é um Quicksort que ignora subvetores com tamanho menor ou igual a 16; a segunda é a aplicação do Insercao a todo o vetor. Observe que, ao final da

primeira etapa, cada elemento estará, no máximo, a 16 posições de sua posição correta, de modo que o Insercao executará em tempo linear para finalizar a ordenação.

Figura 6 – Número médio de movimentações (10 execuções por tamanho) realizadas pelos algoritmos Quicksort e Insercao em vetores gerados de forma pseudoaleatória.



Fonte: Elaborado pelo autor

```
1  void QuicksortI(int n, int *v) {
2      QuicksortIRec(0, n - 1, v);
3      Insercao(n, v);
4  }
5  void QuicksortIRec(int l, int r, int *v) {
6      while (r - l > 16) {
7          int j = Particiona(l, r, v);
8          if (j - l < r - j) {
9              QuicksortIRec(l, j, v);
10             l = j + 1;
11         } else {
12             QuicksortIRec(j, r, v);
13             r = j - 1;
14         }
15     }
16 }
```

2.5.2 Introsort

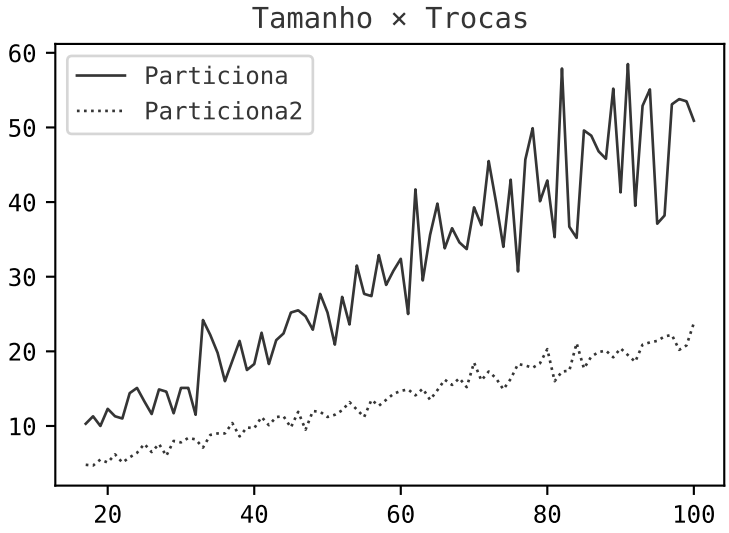
O algoritmo Introsort foi introduzido em (MUSSER, 1997) e é atualmente utilizado na implementação do método *sort* da biblioteca padrão da linguagem C++.

Outro método de particionamento

O Introsort controla a altura da pilha de recursão de maneira diferente daquela utilizada no Quicksort, o que possibilita o uso de outro método de particionamento que, como mostra a Figura 7, realiza menos trocas. Além disso, o elemento pivô é escolhido de forma distinta. A função MoveMedianaFim posiciona, na última posição do vetor, a mediana entre os elementos $v[l]$, $v[(l+r)/2]$ e $v[r]$. Essa estratégia de escolha do pivô é conhecida como "mediana de três".

```
1  int Particiona2(int l, int r, int *v) {
2      MoveMedianaFim(l, r, v);
3      int pivot = v[r], i = l - 1, j = r + 1;
4      while (1) {
5          do --j; while (v[j] > pivot);
6          do ++i; while (v[i] < pivot);
7          if (i >= j) break;
8          Troca(v + i, v + j);
9      }
10     return j;
11 }
```

Figura 7 – Número médio de trocas (10 execuções por tamanho) realizadas pelos métodos Particiona e Particiona2 em vetores gerados de forma pseudoaleatória.



Fonte: Elaborado pelo autor

O algoritmo Introsort

A ideia consiste em evitar o pior caso do Quicksort utilizando o Heapsort quando o tamanho da pilha de recursão atinge o valor $\lfloor 2\log_2 n \rfloor$. Dessa forma, como o Heapsort possui complexidade no pior caso linearítmica, o pior caso do Introsort é $\mathcal{O}(n \log_2 n)$. Além disso, assim como no algoritmo anterior, o Insercao é utilizado para concluir a ordenação.

```
1  void Introsort(int n, int *v) {
2      IntrosortRec(0, n - 1, log2(n) * 2, v);
3      Insercao(n, v);
4  }
5  void IntrosortRec(int l, int r, int d, int *v) {
6      while (r - l > 16) {
7          if (d == 0) {
8              Heapsort(r - l + 1, v + l);
9              return;
10         }
11         --d;
12         int j = Particiona2(l, r, v);
13         IntrosortRec(l, j, d, v);
14         l = j + 1;
15     }
16 }
```

3 ORDENAÇÃO EM TEMPO LINEAR

Este capítulo apresenta três métodos de ordenação que não se baseiam diretamente em comparações e que, sob determinados critérios, realizam a ordenação em tempo linear.

3.1 Contagem

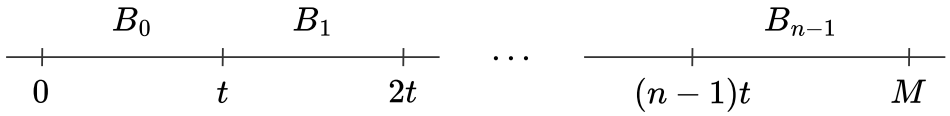
O algoritmo Contagem baseia-se na contagem do número de ocorrências de cada elemento na entrada para determinar suas posições na saída. Foi introduzido por Harold H. Seward em 1954 e é particularmente eficiente quando os valores dos elementos a serem ordenados pertencem a um intervalo pequeno e conhecido. A implementação apresentada a seguir assume que o vetor de entrada não contém elementos negativos e que o maior elemento é suficientemente pequeno para permitir a criação de um vetor com esse tamanho.

```
1  void Contagem(int n, int *v) {
2      int *contagem = AlocaVetorLimpo(M + 1);
3      int *v_aux = AlocaVetor(n);
4      for (int i = 0; i < n; i++) contagem[v[i]]++;
5      for (int i = 1; i <= M; i++) contagem[i] += contagem[i - 1];
6      for (int i = n - 1; i >= 0; i--) {
7          v_aux[contagem[v[i]] - 1] = v[i];
8          contagem[v[i]]--;
9      }
10     CopiaVetor(n, v_aux, v);
11     free(contagem);
12     free(v_aux);
13 }
```

3.2 Balde

O Balde é um algoritmo de ordenação descrito por John W. Tukey em 1944. Ele se baseia na distribuição dos elementos em diferentes grupos, chamados “baldes”, cujos conteúdos são ordenados individualmente. Por fim, os baldes são concatenados, resultando em um vetor ordenado.

Figura 8 – Ilustração do intervalo correspondente a cada balde, em que B_i representa o i-ésimo balde e M é o maior elemento que pode existir no vetor.



$$t = \left\lceil \frac{M}{n} \right\rceil$$

$$x \in B_i \iff x \in [it,\ \ (i+1)t)$$

Fonte: Elaborado pelo autor.

Caso os elementos do vetor de entrada estejam distribuídos de forma uniforme no intervalo $[0, M]$, é esperado que cada balde contenha uma quantidade constante de elementos, de modo que o algoritmo Balde execute em tempo linear. No entanto, não se sabe previamente quantos elementos cada balde armazenará durante a execução. Por isso, cada balde foi implementado como uma estrutura de crescimento dinâmico, utilizando listas ligadas. Além disso, o método InsereOrdenado foi desenvolvido de forma semelhante ao algoritmo Insercao.

```
1  void Balde(int n, int *v) {
2      Celula **baldes = (Celula **) calloc(n, sizeof(Celula*));
3      double t = ceil(M / (double) n);
4      for (int i = n - 1; i >= 0; i--) {
5          int j = (int) floor(v[i] / t);
6          InsereOrdenado(v[i], &baldes[j]);
7      }
8      for (int i = 0, k = 0; i < n; i++) {
9          Celula *atual = baldes[i];
10         while (atual != NULL) {
11             v[k++] = atual->valor;
12             Celula *temp = atual;
13             atual = atual->prox;
14             free(temp);
15         }
16     }
17     free(baldes);
18 }
```

3.3 Radixsort

O algoritmo Radixsort é um método de ordenação estável para números naturais, que funciona da seguinte maneira: dado um vetor de n números naturais com d dígitos cada, utiliza-se um algoritmo de ordenação estável para ordená-los pelo dígito menos significativo, depois pelo segundo menos significativo e assim por diante. O Radixsort depende da estabilidade do algoritmo auxiliar para garantir a correção do processo. Se existirem k valores possíveis para cada dígito, e o algoritmo auxiliar ordenar em tempo proporcional a $\mathcal{O}(n+k)$, então o Radixsort terá complexidade $\mathcal{O}(d(n+k))$, ou seja, complexidade linear.

A seguir, são apresentadas as duas versões do Radixsort que foram implementadas: uma utilizando uma adaptação do algoritmo Contagem e outra, uma adaptação do Balde. Como o objetivo deste trabalho é simplesmente ordenar números naturais em base decimal, a função Digito(x, p), utilizada em ambas as versões, assume que $p = 10^r$ e retorna $\lfloor x/p \rfloor \bmod 10$, ou seja, o dígito de x que está na posição $r+1$, da direita para a esquerda.

Radixsort com Contagem

A principal diferença em relação ao algoritmo Contagem apresentado anteriormente é que o vetor de *contagem* possui apenas 10 posições e armazena as ocorrências dos dígitos de 0 a 9, extraídos das posições indicadas por p nos elementos do vetor v.

```
1  void RadixsortC(int n, int *v) {
2      for (int p = 1; M / p > 0; p *= 10)
3          ContagemDigital(p, n, v);
4  }
5  void ContagemDigital(int p, int n,  int *v) {
6      int *contagem = AlocaVetorLimpo(10);
7      int *v_aux = AlocaVetor(n);
8      for (int i = 0; i < n; i++) contagem[Digito(v[i], p)]++;
9      for (int i = 1; i < 10; i++) contagem[i] += contagem[i - 1];
10     for (int i = n - 1; i >= 0; i--) {
11         v_aux[contagem[Digito(v[i], p)] - 1] = v[i];
12         contagem[Digito(v[i], p)]--;
13     }
14     CopiaVetor(n, v_aux, v);
15     free(contagem);
16     free(v_aux);
17 }
```

Radixsort com Balde

Em comparação com o algoritmo Balde, a lista de baldes contém apenas 10 posições, uma para cada valor possível de dígito. Por exemplo, se o dígito indicado por p em $v[i]$ for igual a 5, então $v[i]$ é colocado no balde 5. Observe que é suficiente inserir cada elemento diretamente no início de cada balde, ou seja, como primeiro elemento da lista ligada, sem a necessidade de manter a lista ordenada.

```
1  void RadixsortB(int n, int *v) {
2      for (int p = 1; M / p > 0; p *= 10)
3          BaldeDigital(p, n, v);
4  }
5  void BaldeDigital(int p, int n, int *v) {
6      Celula **baldes = (Celula**) calloc(10, sizeof(Celula*));
7      for (int i = n - 1; i >= 0; i--) {
8          Celula *celula = CriaCelula(v[i]);
9          celula->prox = baldes[Digito(v[i], p)];
10         baldes[Digito(v[i], p)] = celula;
11     }
12     for (int i = 0, k = 0; i < 10; i++) {
13         Celula *atual = baldes[i];
14         while (atual != NULL) {
15             v[k++] = atual->valor;
16             Celula *temp = atual;
17             atual = atual->prox;
18             free(temp);
19         }
20     }
21     free(baldes);
22 }
```

REFERÊNCIAS

CIURA, M. Best increments for the average case of shellsort. In: SPRINGER. **International Symposium on Fundamentals of Computation Theory**. [S.l.], 2001. p. 106–117.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Introduction to Algorithms**. 3rd. ed. Cambridge, MA: MIT Press, 2009. ISBN 978-0-262-03384-8.

FEOFILOFF, P. **Algoritmos em linguagem C**. [S.l.]: Elsevier/Campus, 2009.

HIBBARD, T. N. An empirical study of minimal storage sorting. **Communications of the ACM**, ACM New York, NY, USA, v. 6, n. 5, p. 206–213, 1963.

KNUTH, D. E. The art of computer programming, volume 3: Searching and sorting. **Addison-Westley Publishing Company: Reading, MA**, 1973.

MUSSER, D. R. Introspective sorting and selection algorithms. **Software: Practice and Experience**, Wiley Online Library, v. 27, n. 8, p. 983–993, 1997.

SEDGEWICK, R. A new upper bound for shellsort. **Journal of Algorithms**, Elsevier, v. 7, n. 2, p. 159–173, 1986.

SEDGEWICK, R. **Algorithms in C**. 1. ed. [S.l.]: Addison-Wesley Professional, 1990. (Computer Science Series). ISBN 0201514257,9780201514254.

SHELL, D. L. A high-speed sorting procedure. **Communications of the ACM**, ACM New York, NY, USA, v. 2, n. 7, p. 30–32, 1959.

TOKUDA, N. An improved shellsort. In: **Proceedings of the IFIP 12th World Computer Congress on Algorithms, Software, Architecture - Information Processing '92, Volume 1 - Volume I**. NLD: North-Holland Publishing Co., 1992. p. 449–457. ISBN 044489747X.

VIANA, G. V. R.; CINTRA, G. F. Pesquisa e ordenação de dados. **Fortaleza: Publicação do Sistema UAB/UECE**, 2011.