

Projeto e Análise de Algoritmos (2025-1)

Flávio L. C. de Moura*

28 de maio de 2025

Heapsort

Exercício:

Ordene os n elementos de um vetor $A[1..n]$ da seguinte forma: encontre o maior elemento de A e troque este elemento com o elemento $A[n]$. Em seguida, encontre o maior elemento de $A[1..n-1]$ e troque este elemento com $A[n-1]$, e assim por diante até que o vetor A esteja ordenado.

Escreva o pseudocódigo do seu algoritmo.

Solução força-bruta:

Algorithm 1: selection-sort(A)

```
1 for  $i = n$  downto 2 do
2    $max \leftarrow i$ ;
3   for  $j = i - 1$  downto 1 do
4     if  $A[j] > A[max]$  then
5        $max \leftarrow j$ ;
6     end
7   end
8   swap  $A[i]$  and  $A[max]$ ;
9 end
```

- Análise assintótica:

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^n 1 = \sum_{i=1}^{n-1} (n-i) = \sum_{i=1}^{n-1} i = \frac{(n-1) \cdot n}{2} = \Theta(n^2).$$

- A correção pode ser provada a partir da seguinte invariante:

Antes de cada iteração do laço **for** indexado por i (linhas 1-8), o subvetor $A[i+1..n]$ está ordenado e contém os $(n-i)$ maiores elementos do vetor A .

*flaviomoura@unb.br

O algoritmo Heapsort

Estudaremos um novo algoritmo de ordenação baseado em comparação de chaves, mas bem diferente dos algoritmos (*insertion sort* e *mergesort*) vistos anteriormente. Este novo algoritmo, conhecido como *heapsort*, possui tempo de execução $O(n \cdot \log n)$, como *mergesort*, e o processo de ordenação é feito *in place* (como em *insertion sort*). Portanto *heapsort* combina as vantagens de *insertion sort* e *mergesort*. A estrutura de dados utilizada por este algoritmo é conhecida como *heap*:

DEF. Um *heap* (binário) T é uma estrutura de dados que corresponde a uma árvore binária com chaves associadas aos nós, sendo uma chave por nó, que satisfaz às seguintes condições:

1. T é uma árvore binária completa em todos os níveis, exceto possivelmente o último nível;
2. Todos os caminhos para uma folha do último nível estão à esquerda de todos os caminhos para uma folha do penúltimo nível;
3. A chave de cada nó é maior ou igual do que a chave dos seus filhos.

Os itens 1 e 2 da definição acima caracterizam a chamada de **propriedade do corpo do /heap**. O item 3 corresponde a **propriedade de heap** (de máximo).

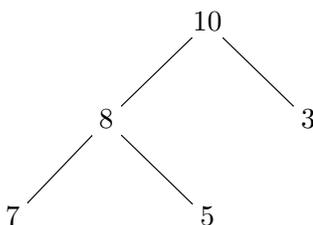
Segundo a propriedade 2, uma enumeração dos nós de um *heap* deve começar de cima para baixo, *i.e.* a partir da raiz do *heap*, e da esquerda para a direita.

Assim, em um *heap* o nó mais à direita pode ter apenas um filho à esquerda, mas não pode ter somente um filho à direita.

Todos os outros nós internos possuem dois filhos.

- Exemplo 1

A figura abaixo é um *heap* de máximo:

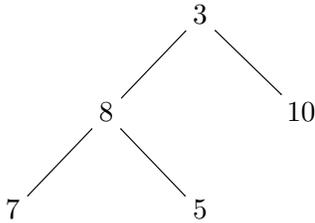


DEF. Um *heap* (binário) T é uma estrutura de dados que corresponde a uma árvore binária com chaves associadas aos nós, sendo uma chave por nó, que satisfaz às seguintes condições:

1. T é uma árvore binária completa em todos os níveis, exceto possivelmente o último nível;
2. Todos os caminhos para uma folha do último nível estão à esquerda de todos os caminhos para uma folha do penúltimo nível;
3. A chave de cada nó é maior ou igual do que a chave dos seus filhos.

• Exemplo 2

A figura abaixo não é um *heap de máximo* porque não satisfaz a propriedade 3:

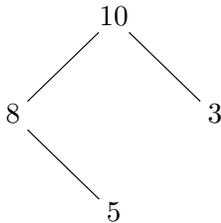


DEF. Um *heap* (binário) T é uma estrutura de dados que corresponde a uma árvore binária com chaves associadas aos nós, sendo uma chave por nó, que satisfaz às seguintes condições:

1. T é uma árvore binária completa em todos os níveis, exceto possivelmente o último nível;
2. Todos os caminhos para uma folha do último nível estão à esquerda de todos os caminhos para uma folha do penúltimo nível;
3. A chave de cada nó é maior ou igual do que a chave dos seus filhos.

• Exemplo 3

A figura abaixo não é um *heap* porque não satisfaz a propriedade 1:



DEF. Um *heap* (binário) T é uma estrutura de dados que corresponde a uma árvore binária com chaves associadas aos nós, sendo uma chave por nó, que satisfaz às seguintes condições:

1. T é uma árvore binária completa em todos os níveis, exceto possivelmente o último nível;

2. Todos os caminhos para uma folha do último nível estão à esquerda de todos os caminhos para uma folha do penúltimo nível;
3. A chave de cada nó é maior ou igual do que a chave dos seus filhos.

- Propriedades

A grande vantagem da estrutura de *heap* é que ela permite a implementação das operações de inserção de um novo elemento (ou uma nova chave), e extração do maior elemento (maior chave) em tempo logarítmico.

Note que em um vetor (ou em uma lista), a inserção pode ser feita em tempo constante, mas a extração do maior elemento vai exigir, no pior caso, uma busca em todo o vetor (ou lista), o que tem custo linear.

A estrutura de *heap* suporta simultaneamente as duas operações, e como veremos, de forma assintoticamente mais eficiente do que em listas ou vetores.

- Implementação em vetores

Um *heap* binário pode ser implementado como um subvetor de um vetor A , onde somente os elementos em $A[1..A.heap-size]$ ($0 \leq A.heap-size \leq A.length$) são elementos válidos do *heap*. A raiz do *heap* é $A[1]$, e dado o índice i de um nó, o índice do filho à esquerda (resp. direita) é $2i$ (resp. $2i + 1$), enquanto que o índice do nó correspondente ao pai do nó de índice i é igual a $\lfloor i/2 \rfloor$.

Alguns autores chamam a estrutura definida acima de *heap* de máximo (ou *max-heap*), isto é, um *heap* onde todo nó i diferente da raiz é tal que $A[\lfloor i/2 \rfloor] \geq A[i]$. Analogamente, podemos definir um *heap* de mínimo (ou *min-heap*) considerando que todo nó i diferente da raiz é tal que $A[\lfloor i/2 \rfloor] \leq A[i]$.

Desta forma, o maior (resp. menor) elemento de um *max-heap* (resp. *min-heap*) é armazenado na raiz, e a subárvore com raiz em um determinado nó contém apenas valores que são menores ou iguais (resp. que são maiores ou iguais) ao valor deste nó.

- O algoritmo Build-Max-Heap

O algoritmo Heapsort utiliza *max-heaps*, e portanto o primeiro passo do algoritmo será transformar o vetor A em um *max-heap*. Este trabalho é feito pelo algoritmo a seguir:

Algorithm 2: Build-Max-Heap(A)

```
1 A.heap-size  $\leftarrow$  A.length for  $i = \lfloor A.length/2 \rfloor$  downto 1 do
2   | Max-Heapify( $A, i$ );
3 end
4
```

1. Mostre que na representação vetorial de um *heap* com n elementos, as folhas são os elementos do vetor com índices $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n$.

- O algoritmo Max-Heapify

O algoritmo Build-Max-Heap constrói o *max-heap* de baixo para cima a partir do primeiro vértice que não é uma folha, e o algoritmo Max-Heapify(A, i) reconstrói um *max-heap* a partir de uma árvore cuja raiz $A[i]$ seja o único elemento que precise ser reposicionado, ou seja, as subárvores com raiz $A[2i]$ e $A[2i + 1]$ já são *max-heaps*:

Algorithm 3: Max-Heapify(A, i)

```
1  $l \leftarrow 2i$ ;
2  $r \leftarrow 2i + 1$ ;
3 if  $l \leq A.heap-size$  and  $A[l] > A[i]$  then
4   |  $largest \leftarrow l$ ;
5 end
6 else
7   |  $largest \leftarrow i$ ;
8 end
9 if  $r \leq A.heap-size$  and  $A[r] > A[largest]$  then
10  |  $largest \leftarrow r$ ;
11 end
12 if  $largest \neq i$  then
13   | exchange  $A[i]$  with  $A[largest]$ ;
14   | Max-Heapify( $A, largest$ );
15 end
```

– Complexidade de Max-Heapify

1. Mostre que, em um *heap* com n elementos e raiz $A[i]$, cada uma das subárvores com raiz em $2i$ e $2i + 1$ têm, no máximo, $2n/3$ elementos.

Considerando o fato estabelecido no exercício anterior, temos que o tempo de execução de Max-Heapify é dado pela recorrência

$$T(n) \leq T(2n/3) + \Theta(1) \tag{1}$$

que, pelo teorema mestre, tem solução $O(\lg n)$.

Teorema Mestre:

Seja $T(n)$ uma função eventualmente não-decrescente que satisfaz a recorrência

$$T(n) = a.T(n/b) + f(n), \quad \text{para } n = b^k, k = 1, 2, 3, \dots \quad \text{onde } a \geq 1, b \geq 2 \text{ e } c \geq 0. \text{ Se}$$

$$T(1) = c$$

$f(n) = \Theta(n^d)$, onde $d \geq 0$, então

$$T(n) = \begin{cases} \Theta(n^{\log_b a}), & \text{se } a > b^d \\ \Theta(n^d \cdot \lg n), & \text{se } a = b^d \\ \Theta(n^d), & \text{se } a < b^d \end{cases}$$

– Complexidade de Build-Max-Heap

Algorithm 4: Build-Max-Heap(A)

```

1 A.heap-size = A.length;
2 for i = ⌊A.length/2⌋ downto 1 do
3   | Max-Heapify(A,i);
4 end

```

Qual o tempo de execução do procedimento Build-Max-Heap(A)? Temos a seguinte

cota superior, considerando um *heap* com n elementos: $\sum_{i=1}^{n/2} O(\lg n) = O(\lg n \cdot \sum_{i=1}^{n/2} 1) =$

$O(\lg n \cdot (n/2)) = O(n \cdot \lg n)$.

* Uma cota mais precisa para Build-Max-Heap

A cota $O(n \cdot \lg n)$, apesar de correta, não é a mais precisa que podemos encontrar. De fato, se observarmos que:

1. A altura de um *heap* contendo n elementos é igual a $\lceil \lg n \rceil$.
2. Um *heap* com n elementos possui, no máximo, $\lceil n/2^{h+1} \rceil$ nós com altura h .

Então, observando que Max-Heapify tem complexidade $O(h)$ quando executado em um nó de altura h , concluímos que o tempo de execução de Build-Max-Heap(A),

assumindo que A possui n elementos, tem a seguinte cota superior: $\sum_{h=0}^{\lceil \lg n \rceil} \lceil n/2^{h+1} \rceil \cdot$

$$O(h) = O(n \cdot \sum_{h=0}^{\lceil \lg n \rceil} \lceil h/2^h \rceil) = O(n), \text{ pois } \sum_{h=0}^{\lceil \lg n \rceil} h/2^h \leq \sum_{h=0}^{\infty} h/2^h, \text{ que por sua vez converge.}$$

Assim, um *heap* pode ser construído em tempo linear.

– A correção de Build-Max-Heap

Prove a seguinte invariante de laço, e conclua que o algoritmo Build-Max-Heap é correto:

No início de cada iteração do laço **for** (linhas 2-4),
cada nó nas posições $i + 1, i + 2 \dots n$ é a raiz de um *max-heap*.

- O algoritmo principal

O algoritmo *heapsort* recebe como argumento um vetor A qualquer contendo $n > 0$ elementos, e inicialmente o transforma em um *max-heap*. Neste momento, sabemos que a raiz do *heap* contém o maior elemento do vetor A , que pode então ser movido para sua posição correta. Em seguida, decrementamos o tamanho do *heap* em uma unidade, e repetimos o processo:

Algorithm 5: Heapsort(A)

```
1 Build-Max-Heap( $A$ );
2 for  $i = A.length$  downto 2 do
3   | exchange  $A[1]$  with  $A[i]$ ;
4   |  $A.heap-size = A.heap-size - 1$ ;
5   | Max-Heapify( $A, 1$ );
6 end
```

- A complexidade de Heapsort

Algorithm 6: Heapsort(A)

```
1 Build-Max-Heap( $A$ );
2 for  $i = A.length$  downto 2 do
3   | exchange  $A[1]$  with  $A[i]$ ;
4   |  $A.heap-size = A.heap-size - 1$ ;
5   | Max-Heapify( $A, 1$ );
6 end
```

A complexidade de Heapsort(A) no pior caso, se A é um vetor com $n > 0$ elementos, é
 $O(n) + \sum_{i=2}^n O(\lg n) = O(n) + O(\lg n) \cdot \sum_{i=2}^n 1 = O(n) + O((n - 1) \cdot \lg n) = O(n \cdot \lg n)$.

- A correção de Heapsort

Algorithm 7: Heapsort(A)

```
1 Build-Max-Heap( $A$ );
2 for  $i = A.length$  downto 2 do
3   | exchange  $A[1]$  with  $A[i]$ ;
4   |  $A.heap-size = A.heap-size - 1$ ;
5   | Max-Heapify( $A, 1$ );
6 end
```

Prove a correção do algoritmo *Heapsort* utilizando a seguinte invariante:

No início de cada iteração do laço **for** (linhas 2-6), o subvetor $A[1..i]$ é um *max-heap* que contém os i menores elementos do vetor $A[1..n]$, e o subvetor $A[i + 1..n]$ está ordenado e contém os $(n - i)$ maiores elementos do vetor $A[1..n]$.

– Leituras recomendadas

1. Capítulo 6 do livro do Cormen (Introduction to Algorithms);
2. Capítulo 6 do livro do Levitin (Introduction to the Design and Analysis of Algorithms).

Filas de prioridade

Fila de prioridade é uma estrutura de dados que armazena elementos com base na sua prioridade, garantindo que elementos com maior prioridade sejam atendidos primeiro.

Assim, cada elemento aparece associado a um valor que é a sua prioridade.

As operações básicas de uma fila de prioridade são:

1. **remove**: remove o elemento com maior prioridade da fila;
2. **insert**: insere um novo elemento na fila.

- Filas de prioridade em listas

Podemos construir uma fila de prioridades utilizando listas:

Em uma lista (qualquer) contendo n elementos,

- a inserção (na primeira posição) tem complexidade constante: $\Theta(1)$.
- a remoção de um elemento arbitrário tem complexidade linear já que pode ser necessário percorrer toda a lista: $\Theta(n)$.

Em uma lista ordenada contendo n elementos,

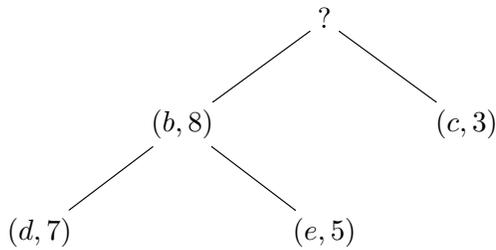
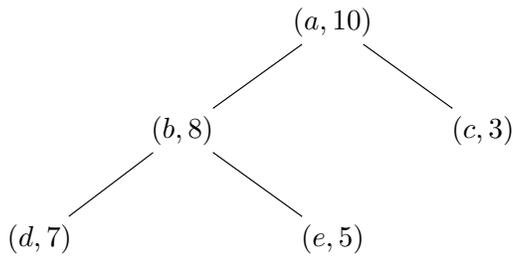
- a inserção tem complexidade linear: $\Theta(n)$.
- a remoção de um elemento arbitrário tem complexidade constante: $\Theta(1)$.

- Filas de prioridade em *heaps*

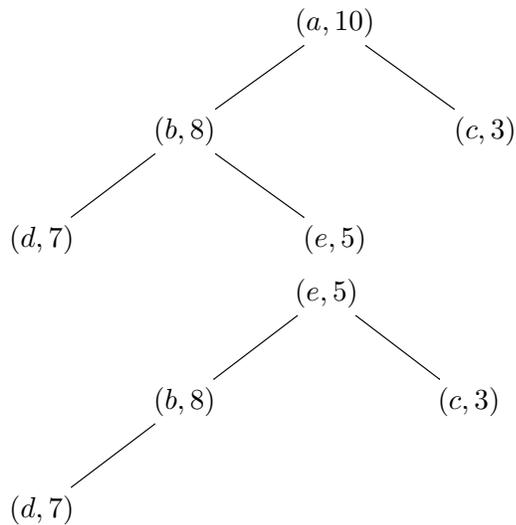
Uma das principais aplicações da estrutura de *heap* consiste na implementação de filas de prioridade, já que tanto a inserção quanto a remoção podem ser implementados em tempo logarítmico.

- Remoção

A remoção do elemento com maior prioridade é feita em tempo constante, mas em seguida o *heap* precisa ser reconstruído via o algoritmo Max-heapify que tem complexidade $\Theta(\lg(n))$.



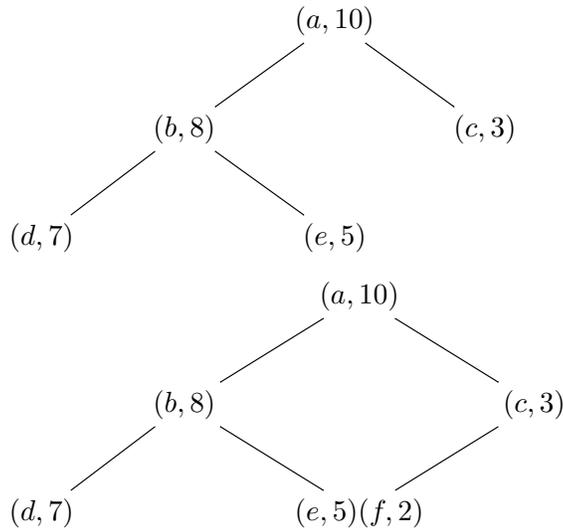
* remove(A)



Algorithm 8: remove($A[1..n]$)

```
1  $max \leftarrow A[1]$ ;
2 exchange  $A[1]$  with  $A[n]$ ;
3  $A.heap\text{-}size \leftarrow (n - 1)$ ;
4 Max-Heapify( $A, 1$ );
5 return  $max$ ;
```

– Inserção



Algorithm 9: insert($(x, y), A[1..n]$)

```
1  $A.size \leftarrow (n + 1)$ ;
2  $k \leftarrow (n + 1)$ ;
3  $A[k] \leftarrow (x, y)$ ;
4 while  $k > 1$  and  $A[\lfloor k/2 \rfloor] < A[k]$  do
5   | exchange  $A[\lfloor k/2 \rfloor]$  with  $A[k]$ ;
6   |  $k \leftarrow \lfloor k/2 \rfloor$ ;
7 end
```

* Exercício

Prove que o algoritmo insert é correto.

Algorithm 10: insert($(x, y), A[1..n]$)

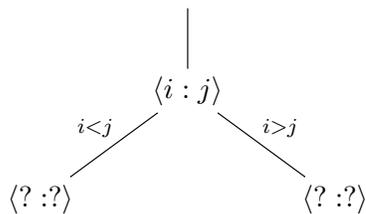
```
1  $A.size \leftarrow (n + 1)$ ;
2  $k \leftarrow (n + 1)$ ;
3  $A[k] \leftarrow (x, y)$ ;
4 while  $k > 1$  and  $A[\lfloor k/2 \rfloor] < A[k]$  do
5   | exchange  $A[\lfloor k/2 \rfloor]$  with  $A[k]$ ;
6   |  $k \leftarrow \lfloor k/2 \rfloor$ ;
7 end
```

Cota inferior para algoritmos de ordenação baseados na comparação de chaves

Queremos estabelecer uma cota inferior para algoritmos baseados na comparação de chaves. Para facilitar a análise, assumiremos que os elementos a serem ordenados são **distintos**. A execução de um algoritmo qualquer de ordenação pode ser simulada por meio de uma árvore binária, chamada de **árvore de decisão**.

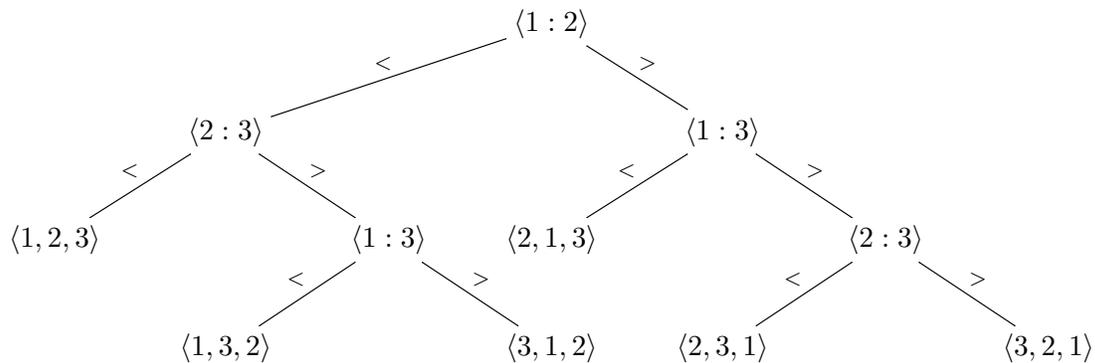
Considere o vetor $[x_1, x_2, \dots, x_n]$ que queremos ordenar. Cada nó da árvore de decisão tem duas formas possíveis:

1. Uma folha da forma $\langle i_1, i_2, \dots, i_n \rangle$ indicando que a ordenação foi finalizada, e neste caso, o vetor ordenado é $[x_{i_1}, x_{i_2}, \dots, x_{i_n}]$ que corresponde a uma permutação do vetor original $[x_1, x_2, \dots, x_n]$
2. Um nó interno $\langle i : j \rangle$ que corresponde à operação que compara x_i com x_j . Neste caso, este nó tem dois filhos:



Exemplo

Por exemplo, a árvore de decisão para o algoritmo *insertion sort* em um vetor com 3 elementos tem a seguinte forma:



Conclusões

O primeiro ponto a ser observado é que a árvore de decisão de um algoritmo correto deve conter pelo menos $n!$ folhas, uma para cada possível permutação da entrada. De fato, se alguma permutação não puder ser gerada, então o algoritmo pode falhar para esta permutação.

O número de nós internos em um caminho da raiz até uma folha da árvore de decisão corresponde ao número de comparações realizadas pelo algoritmo. Assim, o caminho mais longo da raiz até uma folha (altura da árvore) nos fornece o comportamento do algoritmo no pior caso.

Sabemos que uma árvore binária de altura h possui, no máximo, 2^h folhas (árvore binária completa).

Então, o número de folhas da árvore de decisão, digamos l , tem que ser pelo menos $n!$, e no máximo, 2^h .

$$n! \leq l \leq 2^h \implies h \geq \lg(n!) = \Theta(n \cdot \lg(n))$$

Exercícios:

1. Mostre que $\lg(n!) = O(n \cdot \lg(n))$
2. Mostre que $\lg(n!) = \Omega(n \cdot \lg(n))$
3. Como você ordenaria um vetor finito contendo apenas 0s e 1s? Qual a complexidade da sua solução?

Ordenação em tempo linear

Nesta seção veremos uma outra forma de ordenação baseada na ideia de contagem. O que precisamos fazer é contar, para cada elemento a ser ordenado, o número total de elementos que são menores do que ele e guardar este resultado em uma tabela. Os valores da tabela indicarão a posição dos elementos na lista ordenada. Por exemplo, se existem 6 elementos menores do que o elemento x , então x deve ser colocado na sétima posição do vetor ordenado. Desta forma podemos ordenar um vetor simplesmente deslocando os elementos para a posição correta. O pseudocódigo a seguir implementa esta ideia:

Qual é a complexidade desta abordagem? Que vantagens/desvantagens você pode apresentar?

Algorithm 11: comparison-counting-sort($A[0..n - 1]$)

```
1 let  $C[0..n - 1]$  be a new array;
2 for  $i = 0$  to  $n - 1$  do
3   |  $C[i] \leftarrow 0$ ;
4 end
5 for  $i = 0$  to  $n - 2$  do
6   | for  $j = i + 1$  to  $n - 1$  do
7     | if  $A[i] < A[j]$  then
8       | |  $C[j] \leftarrow C[j] + 1$ ;
9       | end
10      | else
11      | |  $C[i] \leftarrow C[i] + 1$ ;
12      | end
13   | end
14 end
15 for  $i = 0$  to  $n - 1$  do
16   |  $B[C[i]] \leftarrow A[i]$ ;
17 end
18 return  $B$ ;
```

Intervalo conhecido

No entanto, podemos utilizar a ideia da contagem de forma mais eficiente se os elementos a serem ordenados forem "conhecidos".

Por exemplo, se o vetor a ser ordenado contém apenas 0s e 1s então podemos utilizar esta informação para fazer a ordenação sem a necessidade de fazer comparações porque com uma única passagem sobre o vetor podemos obter o número k de 0s, e assim retornar o vetor contendo 0s da posição 0 até $k - 1$, e 1s da posição k em diante.

De uma forma mais geral, se os elementos a serem ordenados são inteiros entre l e h então podemos computar a frequência de cada um destes elementos, e armazená-las em um vetor, digamos $C[0..h - l]$, de forma que as primeiras $C[0]$ posições do vetor ordenado serão preenchidas com l , as $C[1]$ posições seguintes com $l + 1$, e assim por diante.

Observe que se os elementos do vetor original não puderem ser sobrescritos então precisaremos de um novo vetor (espaço adicional) para escrever o vetor ordenado.

Counting sort

O pseudocódigo a seguir, assume que cada um dos n inteiros a serem ordenados estão no intervalo entre l e h .

Algorithm 12: counting-sort($A[0..n-1], l, h$)

```
1 let  $C[0..h-l]$  be a new array;
2 for  $i = 0$  to  $h-l$  do
3   |  $C[i] \leftarrow 0$ ;
4 end
5 for  $i = 0$  to  $n-1$  do
6   |  $C[A[i]-l] \leftarrow C[A[i]-l] + 1$ ;
7 end
8 for  $j = 1$  to  $h-l$  do
9   |  $C[j] \leftarrow C[j] + C[j-1]$ ;
10 end
11 for  $i = n-1$  downto  $0$  do
12   |  $j \leftarrow A[i]-l$ ;
13   |  $B[C[j]-1] \leftarrow A[i]$ ;
14   |  $C[j] \leftarrow C[j]-1$ ;
15 end
16 return  $B$ ;
```

Qual é a complexidade deste algoritmo?

Por simplicidade denotaremos $k = h - l$, ou seja, k denota o tamanho do intervalo que contém os elementos a serem ordenado. Assim, o laço **for** das linhas 2-4 é executado em tempo $\Theta(k)$, o laço das linhas 5-7 em tempo $\Theta(n)$, o laço das linhas 8-10 em tempo $\Theta(k)$, e por fim o laço das linhas 11-14 em tempo $\Theta(n)$, o que perfaz um total de $\Theta(n + k)$. Assumindo que $k = O(n)$, temos que o tempo de execução de counting-sort é $\Theta(n)$.

Radix Sort

O algoritmo radix sort ordena uma sequência de inteiros com d dígitos cada, em tempo linear. Ele utiliza um algoritmo auxiliar, que precisa ser estável, para ordenar a sequência de inteiros do dígito menos significativo para o mais significativo. Podemos utilizar counting-sort, por exemplo, como algoritmo auxiliar.

Algorithm 13: radix-sort(A, d)

```
1 for  $i = 1$  to  $d$  do
2   | use a stable sort algorithm to sort array  $A$  on digit  $i$ ;
3 end
```

Teorema

Dados n números com d dígitos, que por sua vez podem assumir até k valores, radix-sort ordena corretamente estes números em tempo $\Theta(d \cdot (n + k))$, se o algoritmo auxiliar estável tem complexidade de tempo $\Theta(n + k)$.

Definição

Um algoritmo de ordenação é dito *estável* se não altera a posição relativa dos elementos que têm o mesmo valor.

Exercício

Prove que o algoritmo counting-sort é estável.

Exercício

Prove que o algoritmo *mergesort* é estável.

Exercício

Prove que o algoritmo *insertion sort* é estável.

Exercício

Mostre como podemos ordenar n inteiros contidos no intervalo de 0 a $n^2 - 1$ em tempo linear, ou seja, em $O(n)$.