Capítulo 6

Heapsort

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

Definição 6.0.1. *Um heap (binário) T é uma estrutura de dados que corresponde a uma árvore binária de altura h que satisfaz às seguintes condições:*

1. *T é uma árvore completa até a altura $(h-1)$;*
2. *Todas as folhas de T têm profundidade h ou $(h-1)$;*
3. *Todos os caminhos para uma folha com profundidade h são à esquerda de todos os caminhos para uma folha de altura $(h-1)$.*

Em um *heap*, o nó mais à direita de profundidade $(h-1)$ pode ter apenas um filho à esquerda, mas não pode ter somente um filho à direita. Todos os outros nós internos possuem dois filhos. Um outro nome para um *heap* binário é *árvore binária completa à esquerda*.

Um *heap* binário pode ser implementado como um subvetor de um vetor A, onde somente os elementos em $A[1..A.\text{heap-size}]$ $(0 \leq A.\text{heap-size} \leq A.length)$ 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 dado por $2i$ (resp. $2i+1$). Por fim, o índice do nó correspondente ao pai do nó de índice i é dado por $\lfloor i/2 \rfloor$.

Existem dois tipos de *heaps* binários: *max-heap* e *min-heap*. Em um *max-heap* (resp. *min-heap*) todo nó i diferente da raiz é tal que $A[\lfloor i/2 \rfloor] \geq A[i]$ (resp. $A[\lfloor i/2 \rfloor] \leq A[i]$). 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 *heapsort* utiliza *max-heaps*, e portanto o primeiro passo do algoritmo será transformar o vetor A de entrada em um *max-heap*. Este trabalho é feito pelo procedimento a seguir:

Algorithm 5: 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
```

onde 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 6: Max-Heapify(A,i)

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

Observe que cada uma das subárvores com raiz em $2i$ e $2i+1$ têm, no máximo, $2n/3$ elementos, e portanto o tempo de execução de Max-Heapify é dado pela recorrência

$$T(n) \leq T(2n/3) + \Theta(1) \tag{6.1}$$

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

Qual o tempo de execução do procedimento Build-Max-Heap(A)? Para responder esta pergunta considere os seguintes fatos:

1. A altura do n-ésimo elemento de um *heap* é igual a $\lfloor \lg n \rfloor$. (Exercício!)
2. Um *heap* com n elementos possui, no máximo, $\lceil n/2^{h+1} \rceil$ nós com altura h. (Exercício!)

Portanto, o tempo de execução de Build-Max-Heap(A), assumindo que A possui n elementos, é dado por $\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil n/2^{h+1} \rceil \cdot O(h) = O(n. \sum_{h=0}^{\lfloor \lg n \rfloor} \lceil h/2^{h+1} \rceil) = O(n)$, pois $\sum_{h=0}^{\lfloor \lg n \rfloor} h/2^{h+1} \leq \sum_{h=0}^{\infty} h/2^{h+1}$, que por sua vez converge. Assim, um *heap* pode ser construído em tempo linear.

O algoritmo *heapsort* receberá como argumento um vetor A qualquer contendo $n > 0$ elementos, e o transformará 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 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
```

O tempo de execução de Heapsort(A) no pior caso, se A é um vetor com n elementos, é $O(n) + n.O(\lg n) = O(n.\lg n)$.

Exercício 6.0.2. *Mostre que na representação vetorial de um* heap *com n elementos, as folhas são os elementos do vetor com índices $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$.*

Exercício 6.0.3. *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.*

Exercício 6.0.4. *Mostre que a complexidade de tempo de Max-Heapify no pior caso é $\Omega(\lg n)$.*

Exercício 6.0.5. *Prove a correção do algoritmo* Heapsort.

Exercício 6.0.6. *Mostre que a complexidade de tempo de* Heapsort *no pior caso é $\Omega(n \lg n)$.*