Cursos de Verão 2010 - IME/USP
Tópicos de Programação
Heapsort
Prof. Alexandre da Silva Freire

1 Heapsort: uma versão "felina" do Selectionsort

O primeiro algoritmo de ordenação que vimos foi o Selectionsort (ordenação por seleção). A estratégia usada no Selectionsort era determinar um elemento mínimo de um intervalo para depois colocá-lo em sua respectiva posição correta. O algoritmo Heapsort utiliza a mesma estratégia, porém, com duas diferenças. A primeira diferença é que o Selectionsort ordena o vetor "da esquerda para a direita" através da seleção de elementos mínimos, já o Heapsort ordena o vetor "da direita para a esquerda" através da seleção de elementos máximos. Isso não faz diferença alguma. Inclusive, podemos facilmente adaptar o Insertionsort para que ele ordene o vetor "da direita para a esquerda" através da seleção de elementos máximos, ou vice-versa. A segunda diferença, esta sim faz toda a diferença, é que o Heapsort determina um elemento máximo em um intervalo através do uso de uma estrutura de dados denominada "max-heap", ou simplesmente "heap".

Um max-heap é um vetor $v[1\,..\,m]$ tal que $v[\lfloor \frac{i}{2} \rfloor] \geq v[i]$, para $i = 2, 3, \ldots, m$ (o elemento $v[0]$ não está no heap). Dizemos que $v[1]$ é a *raiz* do heap. O *pai* de um elemento $v[i]$ é $v[\lfloor \frac{i}{2} \rfloor]$ (por definição, a raiz do heap não tem pai). O *filho esquerdo* de um elemento $v[i]$ é $v[2i]$ e o filho direito é $v[2i+1]$. Se $2i > m$, então $v[i]$ não tem filho esquerdo. Se $2i+1 > m$, então $v[i]$ não tem filho direito. Um elemento que não possui filhos é chamado de *folha*. Utilizando esta terminologia, podemos dizer que um vetor é um max-heap se o valor de todo pai é maior ou igual que o valor de qualquer de seus dois filhos.

A seguir apresentamos algumas funções auxiliares que implementam as definições acima:

```
int esq(int i) {
   return 2*i;
}

int dir(int i) {
   return 2*i+1;
}

int pai(int i) {
   return i/2;
}
```

Além das funções acima, utilizaremos a seguinte função auxiliar para fazer trocas entre elementos de um vetor:

```
    void troca(int v[], int i, int j) {
       int aux = v[i];
       v[i] = v[j];
       v[j] = aux;
    }
```

Considere o seguinte problema: dado um "subvetor" $v[i\ ..\ m]$ tal que os "subvetores" cujos índices das raízes são filhos de i já são max-heaps, rearranjar $v[i\ ..\ m]$ de forma que $v[i\ ..\ m]$ seja um max-heap. Eis um algoritmo para resolver o problema:

```
    void heapify(int v[], int i, int m) {
        int k = i;
        if(dir(i) <= m && v[dir(i)] > v[k]) {
           k = dir(i);
        }
        if(esq(i) <= m && v[esq(i)] > v[k]) {
           k = esq(j);
        }
        if(k != i){
            troca(v, i, k);
(8)         heapify(v, k, m);
        }
    }
```

Lema 1.1. *O algoritmo* `heapfy` *está correto.*

Prova. Por indução em n, onde n é o tamanho do heap. Suponha que $n = 1$. Neste caso o elemento $v[i]$ é folha e, portanto, é um max-heap. Nenhuma das condições dos três if's é satisfeita e o algoritmo pára sem alterar o vetor. Agora suponha que $n > 1$. Neste caso o que o algoritmo faz é guardar na variável k um índice tal que $v[k] \geq v[i]$, $v[k] \geq v[dir(i)]$ e $v[k] \geq v[esq(i)]$. Se $k = i$, então o algoritmo não faz nenhuma troca, pois $v[i\ ..\ m]$ já é um max-heap. Caso contrário, o algoritmo faz a troca entre os elementos $a = v[k]$ e $b = v[i]$. Como a é um elemento máximo de $v[i\ ..\ m]$, então ele está em sua posição definitiva. Por hipótese de indução, *a chamada recursiva feita na linha (8) faz com que* $v[k\ ..\ m]$ *seja um max-heap.* Ou seja, caso o algoritmo troque o elemento $v[i]$ com algum de seus filhos, a chamada recursiva feita na linha (8) faz com que a subárvore correspondente seja um max-heap. Portanto, o algoritmo está correto. ∎

Seja $T(n)$ o consumo de tempo do algoritmo `heapfy`, onde n é o tamanho do heap. Para n da forma $2^k - 1$, definimos $T(n)$ através da seguinte recor-

rência:

$$T(n) = \begin{cases} 1, & \text{se } n = 1 \\ T(\frac{n+1}{2} - 1) + 1, & \text{se } n > 1 \end{cases}$$

Lema 1.2. *Para n da forma $2^k - 1$, temos que $T(n) \leq \lg(n+1)$.*

Prova. Provaremos por indução em n. Suponha que $n = 1$. Neste caso temos que $T(n) = 1 = \lg(n+1)$. Agora suponha que $n > 1$. Neste caso temos que $T(n) = T(\frac{n+1}{2} - 1) + 1$. Por hipótese de indução vale que $T(\frac{n+1}{2} - 1) \leq \lg(\frac{n+1}{2} - 1 + 1) = \lg(\frac{n+1}{2})$. Portanto, $T(n) \leq \lg(\frac{n+1}{2}) + 1 = \lg(\frac{n+1}{2}) + \lg 2 = \lg(n+1)$. ∎

Corolário 1.1. $T(n) = O(\lg n)$.

Prova. Note que para todo número inteiro n existe um inteiro k tal que $2^k - 1 \leq n \leq 2^{k+1} - 1$. Portanto, podemos aplicar o lema 1.2 para concluir que $T(n) \leq \lg(2n+1) \Rightarrow T(n) = O(\lg n)$. ∎

O próximo passo é utilizar a função `heapify` para construir um max-heap. Eis um algoritmo para rearranjar o vetor $v[1\,..\,n]$ de forma que ele seja um max-heap:

```
    void buildheap(int v[], int n) {
       int i;
(1)    for (i = n/2; i >= 1; i--) {
(2)       heapify(v, i, n);
       }
    }
```

Lema 1.3. *O algoritmo* `buildheap` *está correto.*

Prova. Considere o seguinte invariante: (i0) *imediatamente antes de qualquer execução da linha (1), vale que j é a raiz de um max-heap, para $j = i+1, \ldots, n$.* Mostraremos que (i0) é válido ao longo de toda a execução do algoritmo.

Inicialização: todos os elementos de $v[\lfloor \frac{n}{2} \rfloor + 1\,..\,n]$ são folhas e, portanto, são max-heaps.

Manutenção: pelo lema 1.1, a chamada feita na linha (2) faz com que i seja a raiz de um max-heap (preservando os max-heaps já existentes) e, portanto, (i0) vale imediatamente antes da próxima iteração.

Término: o laço da linha (1) pára quando $i = 0$. Como o invariante (i0) é válido após a última iteração, temos que $v[1\,..\,n]$ é um max-heap. Portanto,

o algoritmo está correto. ∎

Lema 1.4. *O algoritmo* `buildheap` *consome tempo* $\Theta(n)$.

Prova. Suponha que n é da forma $2^k - 1$. Pelo lema 1.2, uma chamada à função `heapify` consome tempo no máximo $\lg(m+1)$, para um heap de tamanho m. Logo, o consumo de tempo do algoritmo `buildheap` é no máximo:

$$\sum_{i=2}^{\lg(n+1)} \left(\frac{n+1}{2^i}\right) i = (n+1) \sum_{i=2}^{\lg(n+1)} \left(\frac{1}{2^i}\right) i < (n+1) \sum_{i=0}^{\infty} \left(\frac{1}{2^i}\right) i.$$

Obs.: para chegar na fórmula acima basta observar quantas vezes a função `heapify` é chamada para cada tamanho de heap (exemplo: uma vez para um heap de tamanho n, duas vezes para heaps de tamanho $\frac{n+1}{2}$, quatro vezes para heaps de tamanho $\frac{n+1}{4}$, e assim por diante). Além disso, é necessário observar que o consumo de tempo do `heapify`, para um heap de tamanho $\frac{n+1}{2^i}$, é i.

Sabe-se que $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$, para $|x| < 1$ (veja o apêndice do *Introduction to Algorithms* sobre somatórios). Fazendo $k = \frac{1}{2}$, temos que

$$(n+1) \sum_{i=0}^{\infty} \left(\frac{1}{2^i}\right) i = (n+1) \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = (n+1)2 = 2n+2.$$

Seja $T(n)$ o consumo de tempo do algoritmo `buildheap`, para um vetor de tamanho n. Vimos que se n é da forma $2^k - 1$, então $T(n) < 2n + 2$. Note que para todo número inteiro n existe um inteiro k tal que $2^k - 1 \leq n \leq 2^{k+1} - 1$. Portanto, temos que $T(n) < 4n + 2$, o que implica em $T(n) = O(n)$. Como o laço da linha (1) é executado $\frac{n}{2} = \Theta(n)$ vezes, temos que o algoritmo `buildheap` é $\Omega(n)$. Como $T(n) = \Omega(n)$ e $T(n) = O(n)$, então vale que $T(n) = \Theta(n)$. ∎

Por fim, o algoritmo Heapsort:

Lema 1.5. *O algoritmo* `heapsort` *está correto.*

Prova. Considere os seguintes invariantes que valem imediatamente antes de qualquer execução da linha (2):

(i0) *o vetor* $v[1 \,..\, n]$ *é uma permutação do vetor original*;

(i1) *o vetor* $v[1 \,..\, m]$ *é um max-heap*;

(i2) $v[1 \,..\, m] \leq v[m+1 \,..\, n]$;

```
   /* ordena o intervalo v[1 .. n] */
    void heapsort(int v[], int n) {
       int i, m;
(1)    buildheap(v, n);
(2)    for (m = n; m >= 2; m--) {
(3)       troca(v, 1, m);
(4)       heapify(v, 1, m-1);
       }
    }
```

(i3) *o vetor* $v[m+1\,..\,n]$ *está em ordem crescente.*

É fácil ver que o invariante (i0) é válido ao longo de toda a execução do algoritmo. Mostraremos que os invariantes (i1), (i2) e (i3) são válidos ao longo de toda a execução do algoritmo.

Inicialização: como na inicialização o intervalo $v[m+1\,..\,n]$ é vazio, os invariantes (i2) e (i3) são válidos. Pelo lema 1.3, após a execução da linha (1) o vetor $v[1\,..\,n=m]$ é um max-heap. Logo, o invariante (i1) é válido na inicialização.

Manutenção: após a troca entre os elementos $v[1]$ e $v[m]$, a chamada à função `heapify` feita na linha (4) garante que o invariante (i1) seja válido antes da próxima iteração (isto porque somente os elementos $v[1]$ e $v[m]$ foram alterados e, portanto, os filhos de $v[1]$ são raizes de max-heaps). Por (i1), após a troca feita na linha (3), vale que $v[m]$ é um elemento máximo do intervalo $v[1\,..\,.m]$. Logo, o invariante (i2) é válido antes da próxima iteração. Por (i2), temos que $v[m] \leq v[m+1]$. Logo, (i3) é válido antes da próxima iteração.

Término: o laço da linha (2) pára quando $m = 1$. Por (i2), temos que $v[1] \leq v[2\,..\,n]$. Por (i3), temos que $v[2\,..\,n]$ está em ordem crescente. Portanto, após o término do laço da linha (2) vale que $v[1\,..\,n]$ está em ordem crescente e, consequentemente, o algoritmo está correto. ∎

Lema 1.6. *O algoritmo* `heapsort` *consome tempo* $O(n \lg n)$.

Prova. A linha (1) consome tempo $O(n)$. O laço da linha (2) é executado $n-1$ vezes, sendo que cada uma delas consome tempo $O(\lg n)$. Logo, o consumo total de tempo do algoritmo `heapsort` é $O(n) + (n-1) \cdot O(\lg n) = O(n \lg n)$. ∎