

Técnicas de projeto de algoritmos: Indução

ACH2002 - Introdução à Ciência da Computação II

Delano M. Beder

Escola de Artes, Ciências e Humanidades (EACH)
 Universidade de São Paulo
 dbeder@usp.br

08/2008

- Forneça soluções recursivas para os problemas abaixo

- cálculo do fatorial de um número.
- cálculo do elemento n da série de Fibonacci.
 - $f_0 = 0, f_1 = 1,$
 - $f_n = f_{n-1} + f_{n-2}$ para $n \geq 2$.
- busca seqüencial.
- busca binária.
- torre de hanói.

Exercícios

- Solução para o algoritmo recursivo para cálculo do fatorial de um número

```
int fatr(int m) {
    if(m == 0) {
        return 1;
    }
    else {
        return (m * fatr(m-1));
    }
}
```

Exercícios

- Solução para o cálculo do elemento n da série de Fibonacci utilizando um algoritmo recursivo

```
int fibonaccir(int n) {
    if(n <= 1) {
        return n;
    } else {
        return (fibonaccir(n-1) + fibonaccir(n-2));
    }
}
```

- Solução para o algoritmo recursivo de busca seqüencial

```
int sequencial(int valor, int[] vetor, int n) {
    if(n == 1) {
        if(v[0] == valor) {
            return 0;
        }
        else {
            return -1;
        }
    } else {
        int index = sequencial(valor, vetor, n-1);
        if(index < 0) {
            if(v[n-1] == valor) {
                index = n-1;
            }
        }
        return index;
    }
}
```

- Solução para o algoritmo recursivo de busca binária

```
int binaria(int valor, int[] vetor, int esq, int dir) {
    int meio = (esq + dir)/2;

    if(esq <= dir) {
        if(valor > vetor[meio]) {
            esq = meio + 1;
            return binaria(valor, vetor, esq, dir);
        } else if(valor < vetor[meio]) {
            dir = meio - 1;
            return binaria(valor, vetor, esq, dir);
        } else {
            return meio;
        }
    } else {
        return -1; // retorna -1 se o valor nao for encontrado
    }
}
```

- Solução para o algoritmo recursivo para a torre de hanói

```
void hanoi(char ori, char dst, char aux, int nro) {
    if(nro == 1) {
        System.out.print("Move de " + ori);
        System.out.println(" para " + dst);
    }
    else {
        move(ori, aux, dst, nro-1);
        move(ori, dst, aux, 1);
        move(aux, dst, ori, nro-1);
    }
}
```

- Como calcular a complexidade temporal no pior caso do algoritmo de busca binária recursiva ?
- Em um método recursivo há uma ou mais chamadas (diretas ou indiretas) para o método dentro do seu corpo.
- Para calcular a complexidade (temporal ou espacial) utiliza-se as *equações de recorrência*.
- Um *equação de recorrência* é uma maneira de definir uma função por uma expressão envolvendo a mesma função.

Exemplo

$$T(n) = \begin{cases} O(1) & \text{se } n = 1 \\ T(n/3) + n & \text{caso contrário} \end{cases}$$

- Quais são os custos da *busca binária recursiva*?
 - Qual é o pior caso?
 - Qual a operação mais relevante?
 - Quando $n = 1$, quanto vale $T(1)$?
 - E quando $n > 1$, quanto vale $T(n)$?
- Baseado nas suas respostas, que equação de recorrência poderíamos escrever?

- Quais são os custos da *busca binária recursiva*?
 - Qual é o pior caso?
 - ocorre quando o valor não está no vetor.
 - Qual a operação que mais ocorre?
 - inspeções do elemento – comparações de `valor` com `v[meio]`.

$$T(n) = \begin{cases} O(2) & \text{se } n = 1 \\ T(n/2) + O(2) & \text{caso contrário} \end{cases}$$

- Como resolver esta equação?

Uma das maneiras de resolver esta equação de recorrência é fazendo a sua expansão

$$\begin{aligned} T(n) &= T(n/2) + O(2) \\ T(n) &= T(n/4) + O(2) + O(2) \\ T(n) &= O(n/8) + O(2) + O(2) + O(2) \\ &\dots \\ T(n) &= T(n/2^i) + (i - 1) * O(2) \end{aligned}$$

onde i é tal que $n/2^i = 1$. Isto é, $i = \log_2 n$

Substituindo $T(n/2^i)$ por $T(1)$, obtém-se
 $T(n) = (i - 1) * O(2) + T(1)$. Mas $T(1) = O(2)$.
 $T(n) = (i - 1 + 1) * O(2)$.

$$\text{Logo, } T(n) = i * O(2).$$

Usando operação $f(n) * O(g(n)) = O(f(n)g(n))$ da notação O , obtém-se

$$T(n) = O(2 * i)$$

que é o mesmo que

$$T(n) = O(i),$$

mas $i = \log_2 n$, portanto:

$$T(n) = O(\log_2 n)$$

Busca seqüencial recursiva

- Quais são os custos da *busca seqüencial recursiva*?

$$T(n) = \begin{cases} O(1) & \text{se } n = 1 \\ T(n-1) + O(1) & \text{caso contrário} \end{cases}$$

$$\begin{aligned} T(n) &= T(n-1) + O(1) \\ T(n) &= T(n-2) + O(1) + O(1) \\ &\dots \\ T(n) &= T(n-i) + i * O(1) \end{aligned}$$

onde i é tal que $n - i = 1$

$T(n) = T(1) + (n-1) * O(1)$. No entanto, $T(1) = O(1)$.

$T(n) = O(1) + (n-1) * O(1)$.

$T(n) = n * O(1)$.

Logo, $T(n) = O(n)$.

Torre de hanói

- Quais são os custos da *torre de hanói*?

$$T(n) = \begin{cases} O(1) & \text{se } n = 1 \\ 2T(n-1) + O(1) & \text{caso contrário} \end{cases}$$

$$\begin{aligned} T(n) &= 2T(n-1) + O(1) \\ T(n) &= 4T(n-2) + 3 * O(1) \\ &\dots \\ T(n) &= 2^i T(n-i) + (2^i - 1) * O(1) \end{aligned}$$

onde i é tal que $n - i = 1$

$T(n) = 2^{n-1} * T(1) + (2^{n-1} - 1) * O(1)$. No entanto, $T(1) = O(1)$.

$T(n) = (2^{n-1+1} - 1) * O(1)$.

$T(n) = (2^n - 1) * O(1)$.

Logo, $T(n) = O(2^n)$.

Fatorial recursivo

- Quais são os custos do *fatorial recursivo*?
 - Qual a operação que mais ocorre?
 - Quando $m = 1$, quanto vale $T(m)$?
 - E quando $m > 1$, quanto vale $T(m)$?
- Baseado nas suas respostas, que equação de recorrência poderíamos escrever?

Fatorial recursivo

- Qual a operação que mais ocorre?
 - a multiplicação da linha (3).
- Quando $m = 1$, quanto vale $T(m)$?
 - Pode-se assumir que ocorre uma multiplicação: `return 1*1;`
 - Portanto, $T(m) = O(1)$.
- E quando $m > 1$, quanto vale $T(m)$?
 - $T(m) = O(1) + T(m-1)$

$$T(m) = \begin{cases} O(1) & \text{se } m = 1 \\ T(m-1) + O(1) & \text{caso contrário} \end{cases}$$

Expandindo a equação de recorrência, obtém-se.

$$\begin{aligned}T(m) &= T(m-1) + O(1) \\T(m) &= T(m-2) + O(1) + O(1) \\T(m) &= T(m-3) + O(1) + O(1) + O(1) \\T(m) &= T(m-4) + O(1) + O(1) + O(1) + O(1) \\&\dots \\T(m) &= T(m-i) + i * O(1)\end{aligned}$$

onde i é tal que $m - i = 1$
 $T(m) = T(1) + (m - 1) * O(1)$. No entanto, $T(1) = O(1)$.
 $T(m) = O(1) + (m - 1) * O(1)$.
 $T(m) = m * O(1)$.

Logo, $T(m) = O(m)$.

- Um probleminha: o que significa m ?
- Mais especificamente: o que significa $T(483)$?
 - Para o número 483, a complexidade assintótica é proporcional a 483.
 - Logo, m não representa a rigor um tamanho da entrada, mas a própria entrada.
- Como escrever a complexidade assintótica em termos do tamanho de entrada?

- Considere-se que m seja um número que é implementado com n bits.
- Ou seja, o tamanho da entrada é n bits.
- Como representar a entrada em termos do tamanho de bits da entrada?
 - Isto é, $T(n) = O(f(n))$

Podemos escrever, m como

$$\begin{aligned}m &= 2^{n-1}d_{n-1} + 2^{n-2}d_{n-2} + \dots + 2^0d_0 \\&\text{onde } d_i \in \{0, 1\} \text{ e } 0 \leq i < n\end{aligned}$$

No pior caso, todos $d_i = 1$, então $m = 2^{n-1} + 2^{n-2} + \dots + 2^0$

Mas $2^n > 2^{n-1} + 2^{n-2} + \dots + 2^0$ (provado em classe por indução)

Logo, podemos escrever que m é $O(2^n)$.

Portanto,

$$T(n) = O(2^n)$$

- Resolver equações de recorrência não é fácil.
- Muitas vezes, a expansão da equação de recorrência não permite resolvê-la.
- O método da substituição pode ser utilizado para estes casos.
- *Rationale* do método de substituição:
 - 1 Pressupor a forma da solução.
 - 2 Usar a indução matemática para encontrar as constantes e mostrar que a solução funciona.

- Método eficiente somente quando que é possível *chutar* uma possível resposta.
- Pode ser utilizado para estabelecer limites superiores e inferiores sobre uma recorrência (Notação O e Notação Ω)

Exemplo

$$T(n) = \begin{cases} O(1) & \text{se } n = 1 \\ 2T(n/2) + n & \text{caso contrário} \end{cases} \tag{1}$$

Supor que $T(n) = O(n \log n)$.

O método consiste em provar que $T(n) \leq c (n \log n)$ para uma constante $c > 0$.

O passo indutivo: assumindo que $T(n/2) \leq c (n/2 \log n/2)$.

$$\begin{aligned} T(n) &\leq 2 (c (n/2 \log n/2)) + n. \\ T(n) &\leq cn (\log n/2) + n \end{aligned}$$

Manipulando somente o lado direito da inequação:

$$\begin{aligned} &= cn \log n - cn \log 2 + n \\ &= cn \log n - cn + n \\ &= cn \log n - n(c-1) \end{aligned}$$

Se escolhermos $c \geq 1$, a parcela $(n(c-1))$ vai somente diminuir a parcela $(cn \log n)$, logo podemos escrever:

$$T(n) \leq cn \log n$$

Está provado o passo indutivo.

- Falta provar ainda o passo base
 - Temos que provar que $T(1) \leq c (1 \log 1)$.
- Mas isto implica que $T(1) \leq c (1 \log 1)$, ou seja, $T(1) \leq 0$.
- Mas, conforme a equação 1, $T(1)=1$.
- Portanto, não conseguimos provar o passo base $T(1) :-$
- Logo, é necessário encontrar um outro passo base!
- A notação assintótica exige que provemos $T(n) \leq c (n \log n)$ para n maior ou igual que uma constante n_0 de nossa escolha.

Escolhendo um novo passo base, com $n_0 = 2$, então $n \geq 2$ e o novo passo base passa ser:

$T(2) \leq c (2 \log 2)$, isto é, $T(2) \leq 2c$. A constante c a ser escolhida tem que valer para o passo indutivo e para o passo base.

Prova:

Pela equação de recorrência 1, $T(1) = 1$ logo $T(2) = T(1) + 2 = 3$.

Escolhendo $c \geq 2$ temos que $T(2) = 3 \leq 2c$, ou seja, o passo base foi provado. (Note-se $c \geq 2$ vale para o passo base e o passo indutivo).

Portanto:

$$T(n) = O(n \log n)$$

Referências utilizadas: [1] (páginas 42-50) e [2] (páginas 35-42).

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein. *Algoritmos - Tradução da 2a. Edição Americana*. Editora Campus, 2002.

[2] N. Ziviani. *Projeto de Algoritmos com implementações em C e Pascal*. Editora Thomson, 2a. Edição, 2004.