(1) 9ª Aula (8ª Aula: 1ª Avaliação)

Projeto de algoritmos por indução

→ Usaremos indução para projetar algoritmos que resolvem certos problemas

→ A formulação do algoritmo será análoga a uma demonstração por indução

→ Assim, para resolver um problema P:

* mostramos como resolver instâncias pequenas de P (base) e
* mostramos como obter uma solução de uma instância de P a partir das soluções de instâncias menores de P.

(2)

O processo indutivo resulta em algoritmos recursivos, tais que:

→ a base da indução corresponde à solução de casos base da recursão

→ a aplicação da hipótese de indução corresponde a uma ou mais chamadas recursivas

→ o passo da indução corresponde ao processo de obtenção da resposta para o problema original a partir das respostas devolvidas pelas chamadas recursivas

③

- Um benefício imediato é que o uso (correto) da técnica dá uma prova de corretude do algoritmo
- A complexidade do algoritmo é expressa por uma recorrência
- Muitas vezes é imediato a conversão do algoritmo recursivo em um iterativo
- Frequentemente o algoritmo é eficiente, embora existam exemplos simples em que isso não aconteça

④ Exemplo: Cálculo de polinômios

Problema:

Entrada: sequência de números reais $a_n, a_{n-1}, \dots, a_1, a_0$ e $x \in \mathbb{R}$

Saída: $P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$

- Problema simples
- Estamos interessados em projetar um algoritmo que efetue o menor número de operações aritméticas

5 Solução 1

Hipótese de indução: (1ª tentativa)

Dada uma sequência de números reais $a_{n-1}, \dots, a_1, a_0$ e $x \in \mathbb{R}$, sabemos calcular $P_{n-1}(x) = P_{n-1}x^{n-1} + \dots + a_1 x + a_0$.

Base: Para $n = 0$ a solução é a_0.

Passo: Para calcular $P_n(x)$, basta calcular x^n, multiplicar o resultado por a_n e somar com $P_{n-1}(x)$.

6 Solução 1

```
Calculo-Polinomio(A, n, x)
if n == 0 then
    P ← a0
else
    A' ← an-1, ..., a1, a0
    P' ← Calculo-Polinomio(A', n-1, x)
    xn ← x
    for i = 0 to n do
        xn ← xn * x
    P ← P' + an * xn
return P
```

⑦ Solução 1

Seja $T(n)$ o número de operações aritméticas realizadas pelo algoritmo.

$$T(n) = \begin{cases} 0 & , \text{ se } n=0 \\ T(n-1) + n \text{ multiplicações} + 1 \text{ adição} & , \text{ se } n>0 \end{cases}$$

Não é difícil perceber que:

$$T(n) = \sum_{i=1}^{n} (i \text{ multiplicações} + 1 \text{ adição}) =$$

$$= \frac{n(n+1)}{2} \text{ multiplicações} + n \text{ adições}$$

Note que a solução desperdiça muito tempo recalculando x^n!

⑧ Solução 2

Ideia: Eliminar a computação redundante movendo o cálculo de x^{n-1} para dentro da hipótese de indução.

Hipótese de indução: (2ª tentativa)

Sabemos calcular $P_{n-1}(x) = a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$ e também o valor x^{n-1}.

Base: Para $n=0$ a solução é $(a_0, 1)$

Passo: Primeiro calculamos x^n multiplicando x por x^{n-1}, depois calculamos $P_n(x)$ multiplicando x^n por a_n e somando o valor obtido com $P_{n-1}(x)$.

9 Solução 2

```
Calcula-Polinomio(A, n, x)
if n = 0 then
    P ← a_0
    x_n ← 1
else
    A' ← a_{n-1}, ..., a_1, a_0
    (P', x') ← Calcula-Polinomio(A', n-1, x)
    x_n ← x * x'
    P ← P' + a_n * x_n
return (P, x_n)
```

10 Solução 2

$T(n)$: número de operações aritméticas

$$T(n) = \begin{cases} 0 & \text{, se } n = 0 \\ T(n-1) + 2 \text{ multiplicações} + 1 \text{ adição} & \text{, se } n > 0 \end{cases}$$

A solução da recorrência é:

$$T(n) = \sum_{i=1}^{n} (2 \text{ multiplicações} + 1 \text{ adição}) =$$

$$= 2n \text{ multiplicações} + n \text{ adições}$$

(11) Solução 3

- Considerar o polinômio $P_{n-1}(x)$ na hipótese de indução não é a única escolha possível
- A hipótese de indução ainda pode ser reforçada para termos ganho de complexidade

Hipótese de indução: (3ª tentativa)

Sabemos calcular o polinômio $P'_{n-1}(x) = a_n x^{n-1} + a_{n-1} x^{n-2} + \ldots + a_2 x + a_1$.

Importante: Note que $P_n(x) = x P'_{n-1}(x) + a_0$, ou seja, com apenas 1 multiplicação e 1 adição calculamos $P_n(x)$ a partir de $P'_{n-1}(x)$.

(12) Solução 3

```
Calculo-Polinomio(A, n, x)
1. if n = 0 then
2.     P ← a0
3. else
4.     A' ← an, an-1, ..., a1
5.     P' ← Calcula-Polinomio(A', n-1, x)
6.     P ← x * P' + a0
7. return P
```

Note que a base da indução é trivial, pois para $n = 0$ a solução é a_0.

13 Solução 3

$T(n)$: número de operações aritméticas

$$T(n) = \begin{cases} 0 & , \text{ se } n = 0 \\ T(n-1) + 1 \text{ multiplicação} + 1 \text{ adição}, & \text{ se } n > 0 \end{cases}$$

$$T(n) = \sum_{i=1}^{n} (1 \text{ multiplicação} + 1 \text{ adição}) =$$

$$= n \text{ multiplicações} + n \text{ adições}$$

Esta última maneira de calcular $P_n(x)$ é chamada de regra de Horner.

14 Atividade

Algoritmo de busca binária por indução:

Determinar se um elemento x pertence a $A[i..f]$ (ordenado), retornando o índice do vetor com o elemento ou 0 (zero) caso contrário.

Hipótese de indução:

Se o vetor A tem tamanho n, então sabemos aplicar a busca binária nas metades com $\lceil n/2 \rceil$ e $\lfloor n/2 \rfloor$ elementos.

(15) Busca-Binaria (A, i, f, x)

```
se (i > f) ✓
    retorna 0
senão
    se (i = f) ✓
        se (x = A[i])
            retorna i
        senão
            retorna 0
    senão
        m ← ⌊(i+f)/2⌋ ✓
        se (x = A[m]) ✓
            retorna m
        senão
            se (x < A[m]) ✓
                retorna Busca-Binaria (A, i, m-1, x)
            senão
                retorna Busca-Binaria (A, m+1, f, x)
```

(16) Complexidade de pior caso da busca binária:

$$T(n) = \max\{T(\lfloor n/2 \rfloor), T(\lceil n/2 \rceil)\} + 5$$

Verifique que $T(n) = O(\lg n)$.

(17) "Tarefa" para casa (feriado)

Considere as funções f1 e f2 que realizam o mesmo cálculo. Qual delas é mais rápida/eficiente?

Dica: Teste as funções com valores x, y menores que 50, entre 50 e 100, maiores que 100, maiores que 1000, e assim por diante.

```
int f1 (int x, int y){
    if (x == 1 || y == 1)
        return 0;
    else
        return f1(x-1, y) + f1(x, y-1) + x*y;
}
```

(18)

```
int f2 (int x, int y){
    int i, j; t[100][100]

    for (i= 1; i<= x; i++)
        t[i][1] = 0;
    for (j= 2; j <= y; j++)
        t[1][j] = 0;

    for (i=2; i <= x; i++)
        for (j = 2; j<=y; j++)
            t[i][j] = t[i-1][j] + t[i][j-1] + i*j;
    return t[x][y];
}
```