Melhores momentos

AULA 2

Conceitos discutidos

- um pouco mais de recursão
- um pouco de análise experimental de algoritmos
- um pouco de análise algoritmos

Desempenho de binomialR1

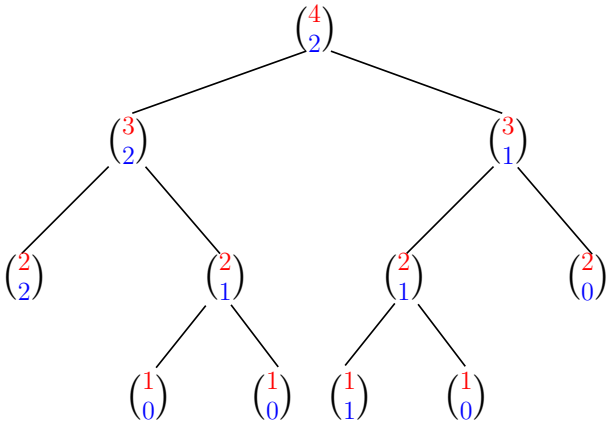
```
 long
 binomialR1(int n, int k)
 {
1   if (n < k) return 0;
2   if (n == k || k == 0) return 1;
3   return binomialR1(n-1, k) +
4          binomialR1(n-1, k-1);
 }
```

Resolve subproblemas muitas vezes

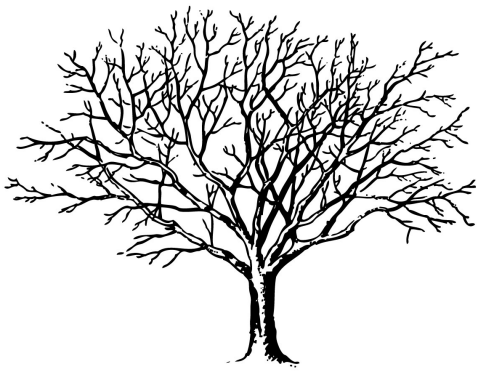
```
binomialR1(6,4)
  binomialR1(5,4)
    binomialR1(4,4)
    binomialR1(4,3)
      binomialR1(3,3)
      binomialR1(3,2)
        binomialR1(2,2)
        binomialR1(2,1)
          binomialR1(1,1)
          binomialR1(1,0)
  binomialR1(5,3)
    binomialR1(4,3)
      binomialR1(3,3)
      binomialR1(3,2)
        binomialR1(2,2)
          binomialR1(2,1)
            binomialR1(1,1)
            binomialR1(1,0)
        binomialR1(2,0)
      binomialR1(3,1)
        binomialR1(2,1)
          binomialR1(1,1)
          binomialR1(1,0)
    binomialR1(4,2)
      binomialR1(3,2)
        binomialR1(2,2)
        binomialR1(2,1)
          binomialR1(1,1)
          binomialR1(1,0)
      binomialR1(3,1)
        binomialR1(2,1)
          binomialR1(1,1)
          binomialR1(1,0)
        binomialR1(2,0)
binom(6,4)=15.
```

Árvore da recursão

`binomialR1` resolve subproblemas muitas vezes.



Árvore



Fonte: http://tfhoa.com/treework

Número de chamadas recursivas

T	0	1	2	3	4	5	6	7	8	...	k
0	0	0	0	0	0	0	0	0	0	...	
1	0	0	0	0	0	0	0	0	0	...	
2	0	2	0	0	0	0	0	0	0	...	
3	0	4	4	0	0	0	0	0	0	...	
4	0	6	10	6	0	0	0	0	0	...	
5	0	8	18	18	8	0	0	0	0	...	
6	0	10	28	38	28	10	0	0	0	...	
7	0	12	40	68	68	40	12	0	0	...	
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱	
n											

Conclusões

Devemos **evitar** resolver o mesmo subproblema várias vezes.

O número de chamadas recursivas feitas por `binomialR1(n,k)` é

$$2 \times \binom{n}{k} - 2 .$$

Mais conclusões

O consumo de tempo da chamada `binomialR1(n,k)` é *proporcional a*

$$2 \times \binom{n}{k} - 2 .$$

Mais conclusões ainda

Quando o valor de k é aproximadamente $n/2$ o consumo de tempo da chamada `binomialR1(n,k)` é *exponencial* pois

$$\binom{n}{k} \geq 2^{\frac{n}{2}}$$

Binomial mais eficiente ainda ...

Supondo $n \geq k \geq 1$ temos que

$$\begin{aligned}
\binom{n}{k} &= \frac{n!}{(n-k)k!} \\
&= \frac{(n-1)!}{(n-k)!(k-1)!} \times \frac{n}{k} \\
&= \frac{(n-1)!}{((n-1)-(k-1))!(k-1)!} \times \frac{n}{k} \\
&= \binom{n-1}{k-1} \times \frac{n}{k} .
\end{aligned}$$

Binomial mais eficiente ainda ...

Logo, supondo $n \geq k \geq 1$, podemos escrever

$$\binom{n}{k} = \begin{cases} n, & \text{quando } k = 1, \\ \binom{n-1}{k-1} \times \frac{n}{k}, & \text{quando } k > 1. \end{cases}$$

```
long
binomialR2(int n, int k)
{
1   if (k == 1) return n;
2   return binomialR2(n-1, k-1) * n / k;
}
```

A função `binomialR2` faz recursão de cauda (*Tail recursion*).

binomialR2(20,10)

```
binomialR2(20,10)
  binomialR2(19,9)
    binomialR2(18,8)
      binomialR2(17,7)
        binomialR2(16,6)
          binomialR2(15,5)
            binomialR2(14,4)
              binomialR2(13,3)
                binomialR2(12,2)
                  binomialR2(11,1)
binom(20,10)=184756.
```

E agora, qual é mais eficiente?

```
meu_prompt> time ./binomialI 30 2
binom(30,2)=435
real                    0m0.002s
user                    0m0.000s
sys                     0m0.000s

meu_prompt> time ./binomialR2 30 2
binom(30,2)=435
real                    0m0.002s
user                    0m0.000s
sys                     0m0.000s
```

E agora, qual é mais eficiente?

```
meu_prompt> time ./binomialI 30 20
binom(30,20)=30045015
real                    0m0.002s
user                    0m0.000s
sys                     0m0.000s

meu_prompt> time ./binomialR2 30 20
binom(30,20)=30045015
real                    0m0.002s
user                    0m0.000s
sys                     0m0.000s
```

Conclusão

O número de chamadas recursivas feitas por binomialR2(n,k) é $k - 1$.

Pausa para nossos comerciais

- Plantão de dúvidas: Vinícius
  Horário: terças das 12h às 13h
  Onde: sala pró-aluno do Biênio
- Treinos para Maratona de Programação:
  Horário: quintas das 14h às 19h
  Onde: sala 6 do CEC, Bloco B, IME-USP
  http://www.ime.usp.br/~maratona/
- XVIII Maratona de Programação: 16 de agosto
  http://www.ime.usp.br/~cef/XVIIImaratona/

AULA 3

Hoje

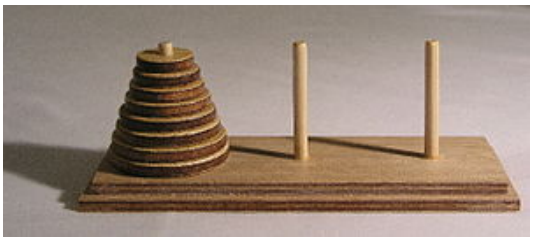
- mais recursão
- mais análise experimental de algoritmos
- um pouco de correção de algoritmos: invariantes
- um pouco de análise algoritmos: "consumo de tempo proporcional a" e notação assintótica

Mais recursão ainda

PF 2.1, 2.2, 2.3 S 5.1

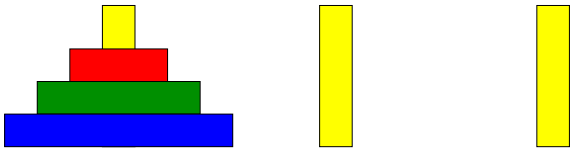
http://www.ime.usp.br/~pf/algoritmos/aulas/recu.html

Torres de Hanoi: epílogo



Fonte: http://en.wikipedia.org/

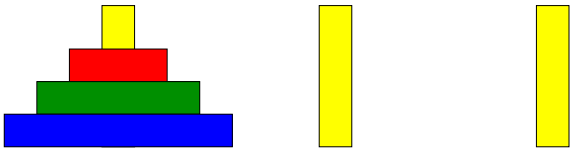
Torres de Hanoi



Desejamos transferir n discos do pino A para o pino C usando o pino B como auxiliar e repeitando as regras:

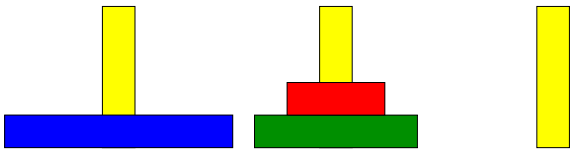
- podemos mover apenas um disco por vez;
- nunca um disco de diâmetro maior poderá ser colocado sobre um disco de diâmetro menor.

Algoritmo recursivo



Para resolver Hanoi(n,A,B,C) basta:

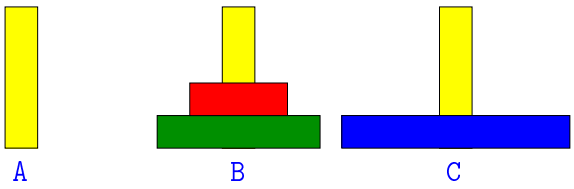
Algoritmo recursivo



Para resolver Hanoi(n,A,B,C) basta:

1. resolver Hanoi(n-1,A,C,B)

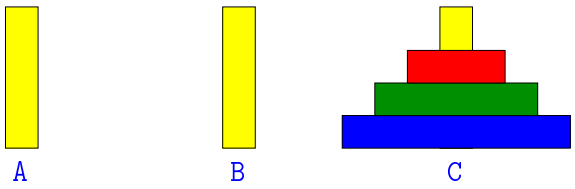
Algoritmo recursivo



Para resolver Hanoi(n,A,B,C) basta:

1. resolver Hanoi(n-1,A,C,B)
2. mover o disco n de A para C

Algoritmo recursivo



Para resolver Hanoi(n,A,B,C) basta:

1. resolver Hanoi(n-1,A,C,B)
2. mover o disco n de A para C
3. resolver Hanoi(n-1,B,A,C)

Base: sabemos revolver Hanoi(**0**,...,...,...)

Número de movimentos

Seja T(n) o número de movimentos feitos pelo algoritmo para resolver o problema das torres de Hanoi com n disco.

Temos que

$$T(0) = 0$$
$$T(n) = 2\,T(n-1) + 1 \quad \text{para } n = 1, 2, 3 \ldots$$

Quanto vale T(n)?

Recorrência

Temos que

$$\begin{aligned} T(n) &= 2\,T(n-1) + 1 \\ &= 2\,(2\,T(n-2) + 1) + 1 \\ &= 2\,(2\,(2\,T(n-3) + 1) + 1) + 1 \\ &= 2\,(2\,(2\,(2\,T(n-4) + 1) + 1) + 1) + 1 \\ &= \cdots \\ &= 2\,(2\,(2\,(2\,(\cdots(2\,T(0) + 1)) + 1) + 1) + 1) + 1 \end{aligned}$$

Recorrência

Logo,

$$\begin{aligned} T(n) &= 2^{n-1} + \cdots + 2^3 + 2^2 + 2 + 1 \\ &= 2^n - 1\,. \end{aligned}$$

n	0	1	2	3	4	5	6	7	8	9
T(n)	0	1	3	7	15	31	63	127	255	511

Conclusões

O número de movimentos feitos pela chamada `hanoi(n, ..., ..., ...)` é

$$2^n - 1\,.$$

Notemos que a função `hanoi` faz o **número mínimo** de movimentos: **não é possível** resolver o quebra-cabeça com menos movimentos.

Enquanto isto . . . os monges . . .

$T(64) = 18.446.744.073.709.551.615 \approx 1{,}84 \times 10^{19}$

Suponha que os monges façam o movimento de 1 disco por segundo(!).

$$\begin{aligned} 18 \times 10^{19}\,\text{seg} &\approx 3{,}07 \times 10^{17}\,\text{min} \\ &\approx 5{,}11 \times 10^{15}\,\text{horas} \\ &\approx 2{,}13 \times 10^{14}\,\text{dias} \\ &\approx 5{,}83 \times 10^{11}\,\text{anos.} \\ &= \mathbf{583}\,\text{bilhões de anos.} \end{aligned}$$

A idade da Terra é **4,54 bilhões de anos**.

The Tower of Hanoi Story

Taken From W.W. Rouse Ball & H.S.M. Coxeter, Mathematical Recreations and Essays, 12th edition. Univ. of Toronto Press, 1974. The De Parville account of the origen from La Nature, Paris, 1884, part I, pp. 285-286.

> *In the great temple at Benares beneath the dome that marks the centre of the world, rests a brass plate in which are fixed three diamond needles, each a cubit high and as thick as the body of a bee. On one of these needles, at the creation, God placed sixty-four discs of pure gold, the largest disk resting on the brass plate, and the others getting smaller and smaller up to the top one. This is the tower of Bramah. Day and night unceasingly the priest transfer the discs from one diamond needle to another according to the fixed and immutable laws of Bramah, which require that the priest on duty must not move more than one disc at a time and that he must place this disc on a needle so that there is no smaller disc below it. When the sixty-four discs shall have been thus transferred from the needle which at creation God placed them, to one of the other needles, tower, temple, and Brahmins alike will crumble into dust and with a thunderclap the world will vanish. The number of separate transfers of single discs which the Brahmins must make to effect the transfer of the tower is two raised to the sixty-fourth power minus 1 or 18,446,744,073,709,551,615 moves. Even if the priests move one disk every second, it would take more than 500 billion years to relocate the initial tower of 64 disks.*

http://www.rci.rutgers.edu/~cfs/472_html/AI_SEARCH/Story_TOH.html

Máximo divisor comum

PF 2.3 S 5.1

http://www.ime.usp.br/~pf/algoritmos/aulas/recu.html

http://www.ime.usp.br/~coelho/mac0122-2012/aulas/mdc/

Divisibilidade

Suponha que m, n e d são números inteiros.

Dizemos que d **divide** m se $m = k\,d$ para algum número inteiro k.
$d \mid m$ é uma abreviatura de "d divide m"

Se d divide m, então dizemos que m é um **multiplo** de d.

Se d divide m e $d > 0$, então dizemos que d é um **divisor** de m

Divisibilidade

Se d divide m e d divide n, então d é um **divisor comum** de m e n.

Exemplos:

os divisores de 30 são: $1, 2, 3, 5, 6, 10, 15$ e 30
os divisores de 24 são: $1, 2, 3, 4, 6, 8, 12$ e 24
os divisores comuns de 30 e 24 são: $1, 2, 3$ e 6

Máximo divisor comum

O **máximo divisor comum** de dois números inteiros m e n, onde pelo menos um é não nulo, é o maior divisor comum de m e n.

O máximo divisor comum de m e n é denotado por $\mathtt{mdc}(m, n)$.

Exemplo:
máximo divisor comum de 30 e 24 é 6
máximo divisor comum de 514229 e 317811 é 1
máximo divisor comum de 3267 e 2893 é 11

Máximo divisor comum

Problema: Dados dois números inteiros não-negativos m e n, determinar mdc(m, n).

Exemplo:
máximo divisor comum de 30 e 24 é 6
máximo divisor comum de 514229 e 317811 é 1
máximo divisor comum de 3267 e 2893 é 11

Solução MAC2166

Recebe números inteiros não-negativos m e n e devolve mdc(m, n). Supõe $m, n > 0$.

```
#define min(m,n) ((m) < (n) ?  (m) :  (n))
int mdc(int m, int n)
{
  int d = min(m,n);
  while (/*1*/ m % d != 0 || n % d != 0)
     /*2*/
     d--;
  /*3*/
  return d;
}
```

Invariantes e correção

Passamos agora a verificar a correção do algoritmo.

Correção da função = a função funciona = a função faz o que promete.

A correção de algoritmos iterativos é comumente baseada na demonstração da validade de **invariantes**.

Estes invariantes são afirmações ou relações envolvendo os objetos mantidos pelo algoritmo.

Invariantes e correção

Eis relações invariantes para a função mdc.

Em /*1*/ vale que

(i0) $1 \le d \le \min(m, n)$, e
(i1) $m \% t \neq 0$ ou $n \% t \neq 0$ para todo $t > d$,

e em /*2*/ vale que

(i2) $m \% d \neq 0$ ou $n \% d \neq 0$.

Invariantes e correção

É evidente que em /*3*/, antes da função retornar d, vale que

$m \% d = 0$ e $n \% d = 0$.

Como (i1) vale em /*1*/, então (i1) também vale em /*3*/. Assim, nenhum número inteiro maior que o valor d retornado divide m e n. Portanto, o valor retornado é de fato o mdc(m,n).

Invariantes são assim mesmo. A validade de alguns torna a correção do algoritmo (muitas vezes) evidente. Os invariantes secundários servem para confirmar a validade dos principais.

Invariantes e correção

Relações invariantes, além de serem uma ferramente útil para demonstrar a correção de algoritmos iterativos, elas nos ajudam a compreender o funcionamento do algoritmo. De certa forma, eles "espelham" a maneira que entendemos o algoritmo.

Consumo de tempo

Quantas iterações do `while` faz a função `mdc`?

Em outras palavras, quantas vezes o comando `"d--"` é executado?

A resposta é `min(m,n)-1` ... no **pior caso**.

Aqui, estamos supondo que $m \geq 0$ e $n \geq 0$.

Por exemplo, para a chamada `mdc(317811,514229)` a função executará 317811-1 iterações, pois $\texttt{mdc}(317811, 514229) = 1$, ou seja, 317811 e 514229 são relativamente primos.

Consumo de tempo

Neste caso, costuma-se dizer que o **consumo de tempo** do algoritmo, no **pior caso**, é *proporcional a* $\min(m, n)$, ou ainda, que o consumo de tempo do algoritmo é da *ordem de* $\min(m, n)$.

A abreviatura de "ordem `blá`" é $O(\texttt{blá})$.

Isto significa que se o valor de $\min(m, n)$ dobra então o tempo gasto pela função pode, no **pior caso** dobrar.

Conclusões

No pior caso, o consumo de tempo da função `mdc` é proporcional a $\min(m, n)$.

O consumo de tempo da função `mdc` é $O(\min(m, n))$.

Se o valor de $\min(m, n)$ dobra, o consumo de tempo pode dobrar.

Argumentos na linha de comando

Argumentos na linha de comando

Quando `main` é chamada, ela recebe dois argumentos:

- `argc` ('c' de *count*) é o número de argumentos que o programa recebeu na linha de comando; e
- `argv[ ]` é um vetor de *strings* contendo cada um dos argumentos.

Por conveção `argv[0]` é o nome do programa que foi chamado. Assim, `argc` é sempre pelo menos 1.

Argumentos na linha de comando

Por exemplo, na chamada

```
meu_prompt> echo Hello World!
```

- `argc` $= 3$
- `argv[0]` $=$ `"echo"`
- `argv[1]` $=$ `"Hello"`
- `argv[2]` $=$ `"World!"`

Argumentos na linha de comando

Na chamada

```
meu_prompt> gcc echo.c -o echo
```

- argc = 4
- argv[0] = "gcc"
- argv[1] = "echo.c"
- argv[2] = "-o"
- argv[3] = "echo"

echo.c

```
#include <stdio.h>
int
main(int argc, char *argv[])
{
   int i;
   for (i = 1; i < argc; i++)
       printf("%s ", argv[i]);
   printf("\n");
   return 0;
}
```

echo.java

```
public class echo {
   public static void main(String argv[])
   {
       for (int i=0; i < argv.length; i++)
           System.out.print(argv[i] + );
       System.out.print("\n");
       System.exit(0);
   }
}
```