Melhores momentos

AULA 3

Conceitos discutidos

- mais recursão: binomial e curvas de Hilbert
- um pouco de análise experimental de algoritmos
- análise de algoritmos:
  quando usar ou não usar recursão

AULA 4

Hoje

- mais recursão: algoritmo de Euclides
- mais análise (experimental) de algoritmos
- argumentos na linha de comando
- structs

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 $\mathtt{m}, \mathtt{n}$ e $\mathtt{d}$ são números inteiros.

Dizemos que $\mathtt{d}$ **divide** $\mathtt{m}$
se $\mathtt{m} = \mathtt{k}\,\mathtt{d}$ para algum número inteiro $\mathtt{k}$.

$\mathtt{d} \mid \mathtt{m}$ é uma abreviatura para "$\mathtt{d}$ divide $\mathtt{m}$"

Divisibilidade

Suponha que $\mathtt{m}, \mathtt{n}$ e $\mathtt{d}$ são números inteiros.

Dizemos que $\mathtt{d}$ **divide** $\mathtt{m}$
se $\mathtt{m} = \mathtt{k}\,\mathtt{d}$ para algum número inteiro $\mathtt{k}$.

$\mathtt{d} \mid \mathtt{m}$ é uma abreviatura para "$\mathtt{d}$ divide $\mathtt{m}$"

Se $\mathtt{d}$ divide $\mathtt{m}$,
então dizemos que $\mathtt{m}$ é um **múltiplo** de $\mathtt{d}$.

Se $\mathtt{d}$ divide $\mathtt{m}$ e $\mathtt{d} > 0$,
então dizemos que $\mathtt{d}$ é um **divisor** de $\mathtt{m}$.

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

Quem são os divisores de 0?

Máximo divisor comum

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

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

Máximo divisor comum

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

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

Problema: Dados dois números inteiros não-negativos $\mathtt{m}$ e $\mathtt{n}$, determinar $\mathtt{mdc}(\mathtt{m}, \mathtt{n})$.

Exemplos:

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 $\texttt{mdc}(\texttt{m}, \texttt{n})$. Supõe $\texttt{m}, \texttt{n} > 0$.

```
#define min(m,n) ((m) < (n) ? (m) : (n))
```

Solução MAC2166

Recebe números inteiros não-negativos `m` e `n` e devolve $\mathtt{mdc}(\mathtt{m}, \mathtt{n})$. Supõe $\mathtt{m}, \mathtt{n} > 0$.

```c
#define min(m,n) ((m) < (n) ? (m) : (n))

int mdc(int m, int n) {
   int d = min(m,n);

   while (m % d != 0 || n % d != 0)
      d--;

   return d;
}
```

Consumo de tempo

```
int mdc(int m, int n) {
   int d = min(m,n);
   while (m % d != 0 || n % d != 0)
      d--;
   return d;
}
```

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

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

Consumo de tempo

```
int mdc(int m, int n) {
   int d = min(m,n);
   while (m % d != 0 || n % d != 0)
      d--;
   return d;
}
```

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**.

(Lembre-se que estamos supondo que `m` > 0 e `n` > 0.)

Consumo de tempo

```
int mdc(int m, int n) {
   int d = min(m,n);
   while (m % d != 0 || n % d != 0)
      d--;
   return d;
}
```

São `min(m,n)-1` iterações no **pior caso**.

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

Consumo de tempo

```
int mdc(int m, int n) {
   int d = min(m,n);
   while (m % d != 0 || n % d != 0)
      d--;
   return d;
}
```

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

Consumo de tempo

```
int mdc(int m, int n) {
   int d = min(m,n);
   while (m % d != 0 || n % d != 0)
      d--;
   return d;
}
```

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* $\min(m, n)$.

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

Consumo de tempo

```
int mdc(int m, int n) {
   int d = min(m,n);
   while (m % d != 0 || n % d != 0)
      d--;
   return d;
}
```

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

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

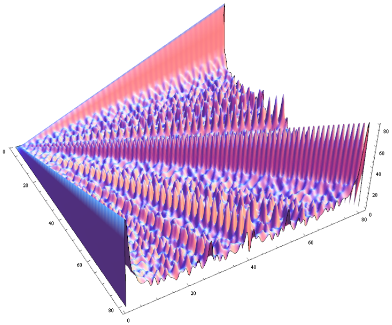
Conclusões

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

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

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

Algoritmo de Euclides



Fonte: http://math.stackexchange.com/

PF 2 (Exercícios) S 5.1

http://www.ime.usp.br/~pf/algoritmos/aulas/recu.html
http://www.ime.usp.br/~coelho/mac0122-2014/aulas/mdc/

Algoritmo de Euclides

O máximo divisor comum pode ser determinado através de um algoritmo de 2300 anos (cerca de 300 A.C.), o **algoritmo de Euclides**.

Algoritmo de Euclides

O máximo divisor comum pode ser determinado através de um algoritmo de 2300 anos (cerca de 300 A.C.), o **algoritmo de Euclides**.

Para calcular o $\texttt{mdc}(\texttt{m}, \texttt{n})$
o algoritmo de Euclides usa a recorrência:

$$\texttt{mdc}(\texttt{m}, 0) = \texttt{m};$$
$$\texttt{mdc}(\texttt{m}, \texttt{n}) = \texttt{mdc}(\texttt{n}, \texttt{m}\%\texttt{n}), \text{ para } \texttt{n} > 0.$$

Algoritmo de Euclides

O máximo divisor comum pode ser determinado através de um algoritmo de 2300 anos (cerca de 300 A.C.), o **algoritmo de Euclides**.

Para calcular o $\texttt{mdc}(\texttt{m}, \texttt{n})$
o algoritmo de Euclides usa a recorrência:

$$\begin{aligned} \texttt{mdc}(\texttt{m}, 0) &= \texttt{m}; \\ \texttt{mdc}(\texttt{m}, \texttt{n}) &= \texttt{mdc}(\texttt{n}, \texttt{m}\%\texttt{n}), \text{ para } \texttt{n} > 0. \end{aligned}$$

Assim, por exemplo,

$$\texttt{mdc}(12, 18) = \texttt{mdc}(18, 12) = \texttt{mdc}(12, 6) = \texttt{mdc}(6, 0) = 6.$$

Correção

A correção da recorrência proposta por Euclides é baseada no seguinte fato.

Se $\mathtt{m}$, $\mathtt{n}$ e $\mathtt{d}$ são números inteiros, $\mathtt{m} \geq 0$, $\mathtt{n}, \mathtt{d} > 0$, então

$$\mathtt{d} \text{ divide } \mathtt{m} \text{ e } \mathtt{n} \quad \Leftrightarrow \quad \mathtt{d} \text{ divide } \mathtt{n} \text{ e } \mathtt{m}\%\mathtt{n}.$$

Euclides recursivo

O algoritmo de Euclides usa a recorrência:

$$\begin{aligned} \mathtt{mdc}(\mathtt{m}, 0) &= \mathtt{m}; \\ \mathtt{mdc}(\mathtt{m}, \mathtt{n}) &= \mathtt{mdc}(\mathtt{n}, \mathtt{m}\%\mathtt{n}), \text{ para } \mathtt{n} > 0. \end{aligned}$$

```
int euclidesR(int m, int n) {
   if (n == 0) return m;
   return euclidesR(n, m % n);
}
```

euclidesR(317811,514229)

```
euclidesR(317811,514229)
  euclidesR(514229,317811)
    euclidesR(317811,196418)
      euclidesR(196418,121393)
        euclidesR(121393,75025)
          euclidesR(75025,46368)
            euclidesR(46368,28657)
              euclidesR(28657,17711)
                euclidesR(17711,10946)
                  euclidesR(10946,6765)
                    euclidesR(6765,4181)
                      euclidesR(4181,2584)
                        euclidesR(2584,1597)
                          euclidesR(1597,987)
                            euclidesR(987,610)
                              euclidesR(610,377)
                                euclidesR(377,233)
                                  euclidesR(233,144)
                                    euclidesR(144,89)
                                      euclidesR(89,55)
                                        euclidesR(55,34)
                                          euclidesR(34,21)
                                            euclidesR(21,13)
                                              euclidesR(13,8)
                                                euclidesR(8,5)
                                                  euclidesR(5,3)
                                                    euclidesR(3,2)
                                                      euclidesR(2,1)
                                                        euclidesR(1,0)
mdc(317811,514229) = 1.
```

Qual é mais eficiente?

```
meu_prompt>time ./mdc 317811 514229
mdc(317811,514229)=1
real  0m0.004s
user  0m0.002s
sys   0m0.001s

meu_prompt>time ./euclidesR 317811 514229
mdc(317811,514229)=1
real  0m0.003s
user  0m0.001s
sys   0m0.001s
```

Qual é mais eficiente?

```
meu_prompt>time ./mdc 2147483647 2147483646
mdc(2147483647,2147483646)=1
real   0m5.487s
user   0m5.481s
sys    0m0.003s

meu_prompt>time ./euclidesR 2147483647 2147483646
mdc(2147483647,2147483646)=1
real   0m0.003s
user   0m0.001s
sys    0m0.001s
```

Consumo de tempo

```
int euclidesR(int m, int n) {
   if (n == 0) return m;
   return euclidesR(n, m % n);
}
```

O consumo de tempo da função euclidesR é proporcional ao número de chamadas recursivas.

Consumo de tempo

```
int euclidesR(int m, int n) {
   if (n == 0) return m;
   return euclidesR(n, m % n);
}
```

O consumo de tempo da função euclidesR é proporcional ao número de chamadas recursivas.

Suponha que euclidesR faz k chamadas recursivas e que, no início da 1a. chamada, tem-se $0 < n \leq m$.

Consumo de tempo

```
int euclidesR(int m, int n) {
   if (n == 0) return m;
   return euclidesR(n, m % n);
}
```

O consumo de tempo da função euclidesR é proporcional ao número de chamadas recursivas.

Suponha que euclidesR faz k chamadas recursivas e que, no início da 1a. chamada, tem-se $0 < n \leq m$.

Sejam

$$(m, n) = (m_0, n_0), (m_1, n_1), \ldots, (m_k, n_k) = (\mathrm{mdc}(m, n), 0),$$

os valores dos parâmetros no início de cada chamada.

Número de chamadas recursivas

Por exemplo, para $\texttt{m} = 514229$ e $\texttt{n} = 317811$, tem-se

$$\begin{aligned}
(\texttt{m}_0, \texttt{n}_0) &= (514229, 317811), \\
(\texttt{m}_1, \texttt{n}_1) &= (317811, 196418), \\
(\texttt{m}_2, \texttt{n}_2) &= (196418, 121393), \\
\cdots &= \cdots \\
(\texttt{m}_{27}, \texttt{n}_{27}) &= (1, 0).
\end{aligned}$$

Número de chamadas recursivas

```
int euclidesR(int m, int n) {
   if (n == 0) return m;
   return euclidesR(n, m % n);
}
```

Suponha que euclidesR faz k chamadas recursivas.

Estimaremos o valor de k em função de $n = \min(m, n)$.

Note que

$m_{i+1} = n_i$ e $n_{i+1} = m_i \% n_i$ para i=0,1,...,k-1.

Número de chamadas recursivas

```
int euclidesR(int m, int n) {
   if (n == 0) return m;
   return euclidesR(n, m % n);
}
```

Suponha que euclidesR faz k chamadas recursivas.

Estimaremos o valor de k em função de $n = \min(m, n)$.

Note que
$m_{i+1} = n_i$ e $n_{i+1} = m_i \% n_i$ para i=0,1,...,k-1.

E que, para inteiros a e b, com $0 < b \leq a$, vale que

$$a\%b < \frac{a}{2} \quad \text{(verifique!)}.$$

Número de chamadas recursivas

Como $\mathtt{m}_{\mathtt{i}+1} = \mathtt{n}_{\mathtt{i}}$ e $\mathtt{n}_{\mathtt{i}+1} = \mathtt{m}_{\mathtt{i}} \% \mathtt{n}_{\mathtt{i}}$ para `i=0,1,...,k-1`
e $\mathsf{a} \% \mathsf{b} < \frac{\mathsf{a}}{2}$ para inteiros a e b com $0 < \mathsf{b} \leq \mathsf{a}$,
tem-se que

$$
\begin{array}{lcccccccccccc}
\mathtt{n}_2 & = & \mathtt{m}_1 \% \mathtt{n}_1 & = & \mathtt{n}_0 \% \mathtt{n}_1 & < & \mathtt{n}_0/2 & = & \mathtt{n}/2 & = & \mathtt{n}/2^1 \\
\mathtt{n}_4 & = & \mathtt{m}_3 \% \mathtt{n}_3 & = & \mathtt{n}_2 \% \mathtt{n}_3 & < & \mathtt{n}_2/2 & < & \mathtt{n}/4 & = & \mathtt{n}/2^2 \\
\mathtt{n}_6 & = & \mathtt{m}_5 \% \mathtt{n}_5 & = & \mathtt{n}_4 \% \mathtt{n}_5 & < & \mathtt{n}_4/2 & < & \mathtt{n}/8 & = & \mathtt{n}/2^3 \\
\mathtt{n}_8 & = & \mathtt{m}_7 \% \mathtt{n}_7 & = & \mathtt{n}_6 \% \mathtt{n}_7 & < & \mathtt{n}_6/2 & < & \mathtt{n}/16 & = & \mathtt{n}/2^4 \\
\mathtt{n}_{10} & = & \mathtt{m}_9 \% \mathtt{n}_9 & = & \mathtt{n}_8 \% \mathtt{n}_9 & < & \mathtt{n}_8/2 & < & \mathtt{n}/32 & = & \mathtt{n}/2^5
\end{array}
$$

...

...

Número de chamadas recursivas

Como $\mathtt{m}_{\mathtt{i}+1} = \mathtt{n}_{\mathtt{i}}$ e $\mathtt{n}_{\mathtt{i}+1} = \mathtt{m}_{\mathtt{i}} \% \mathtt{n}_{\mathtt{i}}$ para $\mathtt{i}\mathtt{=0,1,\ldots,k-1}$ e $\mathtt{a}\%\mathtt{b} < \frac{\mathtt{a}}{2}$ para inteiros $\mathtt{a}$ e $\mathtt{b}$ com $0 < \mathtt{b} \leq \mathtt{a}$, tem-se que

$$
\begin{array}{lllllllllllll}
\mathtt{n}_2 & = & \mathtt{m}_1\%\mathtt{n}_1 & = & \mathtt{n}_0\%\mathtt{n}_1 & < & \mathtt{n}_0/2 & = & \mathtt{n}/2 & = & \mathtt{n}/2^1 \\
\mathtt{n}_4 & = & \mathtt{m}_3\%\mathtt{n}_3 & = & \mathtt{n}_2\%\mathtt{n}_3 & < & \mathtt{n}_2/2 & < & \mathtt{n}/4 & = & \mathtt{n}/2^2 \\
\mathtt{n}_6 & = & \mathtt{m}_5\%\mathtt{n}_5 & = & \mathtt{n}_4\%\mathtt{n}_5 & < & \mathtt{n}_4/2 & < & \mathtt{n}/8 & = & \mathtt{n}/2^3 \\
\mathtt{n}_8 & = & \mathtt{m}_7\%\mathtt{n}_7 & = & \mathtt{n}_6\%\mathtt{n}_7 & < & \mathtt{n}_6/2 & < & \mathtt{n}/16 & = & \mathtt{n}/2^4 \\
\mathtt{n}_{10} & = & \mathtt{m}_9\%\mathtt{n}_9 & = & \mathtt{n}_8\%\mathtt{n}_9 & < & \mathtt{n}_8/2 & < & \mathtt{n}/32 & = & \mathtt{n}/2^5 \\
\ldots \\
\ldots
\end{array}
$$

Depois de cada 2 chamadas recursivas, o valor do segundo parâmetro é reduzido a menos da sua metade.

Número de chamadas recursivas

Seja $\mathtt{t}$ o número inteiro tal que $2^{\mathtt{t}} \leq \mathtt{n} < 2^{\mathtt{t}+1}$.

Número de chamadas recursivas

Seja $\mathtt{t}$ o número inteiro tal que $2^{\mathtt{t}} \leq \mathtt{n} < 2^{\mathtt{t}+1}$.

Da primeira desigualdade temos que $\mathtt{t} \leq \lg \mathtt{n}$, onde $\lg \mathtt{n}$ denota o logaritmo de $\mathtt{n}$ na base 2.

Número de chamadas recursivas

Seja t o número inteiro tal que $2^t \leq n < 2^{t+1}$.

Da primeira desigualdade temos que $t \leq \lg n$, onde $\lg n$ denota o logaritmo de n na base 2.

Logo, da desigualdade estrita, o número k de chamadas recursivas é tal que

$$k \leq 2(t+1) - 1 = 2t + 1 \leq 2 \lg n + 1.$$

Número de chamadas recursivas

Seja $\mathtt{t}$ o número inteiro tal que $2^{\mathtt{t}} \le \mathtt{n} < 2^{\mathtt{t}+1}$.

Da primeira desigualdade temos que $\mathtt{t} \le \lg \mathtt{n}$, onde $\lg \mathtt{n}$ denota o logaritmo de $\mathtt{n}$ na base 2.

Logo, da desigualdade estrita, o número $\mathtt{k}$ de chamadas recursivas é tal que

$$\mathtt{k} \le 2(\mathtt{t}+1) - 1 = 2\mathtt{t} + 1 \le 2 \lg \mathtt{n} + 1.$$

Para o exemplo onde $\mathtt{m}$=514229 e $\mathtt{n}$=317811, temos que

$$2 \lg \mathtt{n} + 1 = 2 \lg(317811) + 1 < 2 \times 18{,}3 + 1 < 37{,}56$$

e o número de chamadas recursivas feitas por `euclidesR`(514229,317811) foram 27.

Consumo de tempo

Resumindo, a quantidade de tempo consumida pelo algoritmo de Euclides é, no pior caso, proporcional a $\lg \mathtt{n}$.

Este desempenho é significativamente melhor que o desempenho do algoritmo *café com leite*, já que a função $f(\mathtt{n}) = \lg \mathtt{n}$ cresce muito mais lentamente que a função $g(\mathtt{n}) = \mathtt{n}$.

Consumo de tempo

$\mathtt{n}$	(int) $\lg \mathtt{n}$
4	2
5	2
6	2
10	3
64	6
100	6
128	7
1000	9
1024	10
1000000	19
1000000000	29

Conclusões

Suponha que $\mathtt{m} > \mathtt{n}$.

O número de chamadas recursivas da função `euclidesR` é $\leq 2(\lg \mathtt{n}) - 1$.

No pior caso, o consumo de tempo da função `euclidesR` é proporcional a $\lg \mathtt{n}$.

Conclusões

Suponha que $\mathtt{m} > \mathtt{n}$.

$$\boxed{\text{O consumo de tempo da função } \texttt{euclidesR} \text{ é } O(\lg \mathtt{n}).}$$

Para que o consumo de tempo da função `euclidesR` dobre, é necessário que o valor de $\mathtt{n}$ seja elevado ao quadrado.

Euclides e Fibonacci

Demonstre por indução em $\texttt{k}$ que:

Se $\texttt{m} > \texttt{n} \geq 0$ e se a chamada $\texttt{euclidesR(m,n)}$ faz $\texttt{k} \geq 0$ chamadas recursivas, então

$$\texttt{m} \geq \texttt{fibonacci}(\texttt{k}+1) \;\text{ e }\; \texttt{n} \geq \texttt{fibonacci}(\texttt{k}).$$

Recursão de cauda

```
int euclidesR(int m, int n) {
   if (n == 0) return m;
   return euclidesR(n, m % n);
}
```

Euclides iterativo

```
/* Pre-condicao:  a funcao supoe n > 0 */
int euclidesI(int m, int n) {
   int r;
   do {
      r = m % n;
      m = n;
      n = r;
   } while (r != 0);
   return m;
}
```

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.

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 convenção, `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;
}
```

Registros e Structs

PF Apêndice E

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

Registros e Structs

Fonte: http://colorblindprogramming.com/geek-joke

Registros e structs

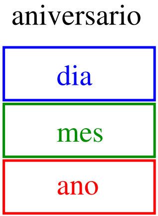
Um **registro** (= *record*) é uma coleção de diversas variáveis, possivelmente de tipos diferentes.

Registros e structs

Um **registro** (= *record*) é uma coleção de diversas variáveis, possivelmente de tipos diferentes.

Na linguagem C, registros são conhecidos como **structs**.

```
struct {
   int dia;
   int mes;
   int ano;
} aniversario;
```



Nomes de estruturas

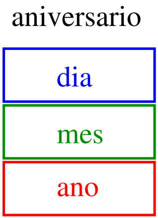
É uma boa idéia dar um nome,
digamos `data`, à estrutura.

Nomes de estruturas

É uma boa idéia dar um nome,
digamos data, à estrutura.

Nosso exemplo ficaria melhor assim:

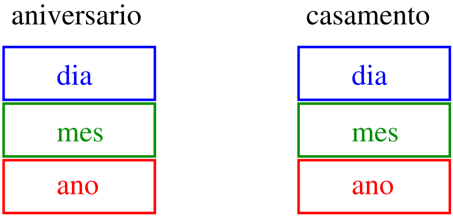
```
struct data {
   int dia;
   int mes;
   int ano;
};

struct data aniversario;
```



Estruturas e tipos

Uma declaração de `struct` define um tipo.

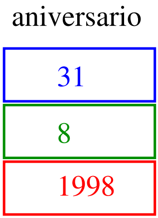
```
struct data aniversario;
struct data casamento;
```



Campos de uma estrutura

É fácil atribuirmos valores aos campos de uma estrutura:

```
aniversario.dia = 31;
aniversario.mes = 8;
aniversario.ano = 1998;
```



Estruturas e typedef

Para não repetir "`struct data`" o tempo todo, podemos definir uma abreviatura via `typedef`:

```
struct data {
    int dia;
    int mes;
    int ano;
};

typedef struct data Data;

Data aniversario;
Data casamento;
```

Estruturas e typedef

Um modo ainda mais compacto de fazer isso:

```
typedef struct {
    int dia, mes, ano;
} Data;

Data aniversario;
Data casamento;
```

Aula que vem

Endereços e ponteiros

Alocação dinâmica de memória