AED1 - Aula 05
Recursão, máximo divisor comum, Fibonacci

"Talvez o mais importante princípio do bom projetista de algoritmos seja se recusar a estar satisfeito" - Aho, Hopcroft e Ullman, the design and analysis of computer algorithms, 1974.

Máximo divisor comum

Definição dos conceitos de divisor e múltiplo:
- Considerando números inteiros d, m e k,
  - podemos escrever m = k * d + m%d.
- Dizemos que "d divide m" se existe k tal que
  - m = k * d, ou seja, m%d = 0.
- A notação matemática para "d divide m" é d | m.
- Se d | m então dizemos que m é múltiplo de d.
- Se d | m e d > 0 então dizemos que d é um divisor de m.

Divisores comuns:
- Se d | m e d | n então d é um divisor comum de m e n.
- Exemplo:
  - Divisores de 20 são: 1, 2, 4, 5, 10 e 20.
  - Divisores de 12 são: 1, 2, 3, 4, 6 e 12.
  - Portanto, divisores comuns de 20 e 12 são 1, 2 e 4.

Máximo divisor comum:
- Denotado por mdc(m, n),
  - corresponde ao maior divisor comum de m e n.
- Exemplos:
  - mdc(20, 12) = 4.
  - mdc(514229, 317811) = 1.
  - mdc(85, 34) = 17.

Problema:
- Dados dois números inteiros não-negativos m e n,
  - encontrar o máximo divisor comum deles,
    - i.e., mdc(m, n).

Ideia básica:
- Como o divisor de um número deve ser menor que este número,

        - podemos testar todos os números entre 1 e min(m, n).
- Como queremos o maior dentre todos os divisores,
    - podemos começar em min(m, n) e ir diminuindo
        - até encontrar o primeiro divisor comum.

Algoritmo iterativo simples:

```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;
}
```

Corretude e invariante:

- No início de cada iteração, para todo valor t > d,
    - temos m % t != 0 ou n % t != 0,
        - ou seja, t não é divisor comum de m e n.
- Demonstração
    - O invariante vale no início, pois antes da primeira iteração
        - d = min(m, n) e um divisor de um número
            - é sempre menor ou igual ao número.
    - Supondo que o invariante valha no início de uma iteração qualquer,
        - podemos verificar que ele vale no início da próxima.
    - Pelo invariante, t não é divisor comum de m e n para t > d.
        - Como o algoritmo entrou na iteração atual,
            - sabemos que d não é divisor comum de m e n.
        - Nesta iteração d é decrementado, ou seja, d' = d - 1.
    - Portanto, no início da iteração seguinte temos que
        - todo t > d' (que agora inclui o antigo d)
            - não é divisor comum de m e n.
- Note que, quando o laço termina d é divisor comum de m e n,
    - já que essa é a única condição que permite sair do laço.
- Pelo invariante, todo t > d
    - não é divisor comum de m e n.
- Portanto, d é o mdc(m, n).

Eficiência de tempo:

- O algoritmo itera no máximo min(m, n) - 1 vezes,
    - e em cada iteração realiza um número constante de operações.
- Portanto, seu consumo de tempo é proporcional a min(m, n) no pior caso,
    - i.e., O(min(m, n)).

Bônus:
- Podemos melhorar o algoritmo anterior
    - fazendo d = min(m, n) / i a cada iteração,
        - com i variando de 1 até min(m, n)^(1/2).
- Qual a eficiência desse novo algoritmo?
- Qual(is) invariante(s) de laço podemos usar para provar que ele está correto?

Agora veremos o algoritmo mais antigo deste curso,
- que já era conhecido a mais de 2 mil anos (datado de 300 a.C.).
    - Trata-se do algoritmo de Euclides.

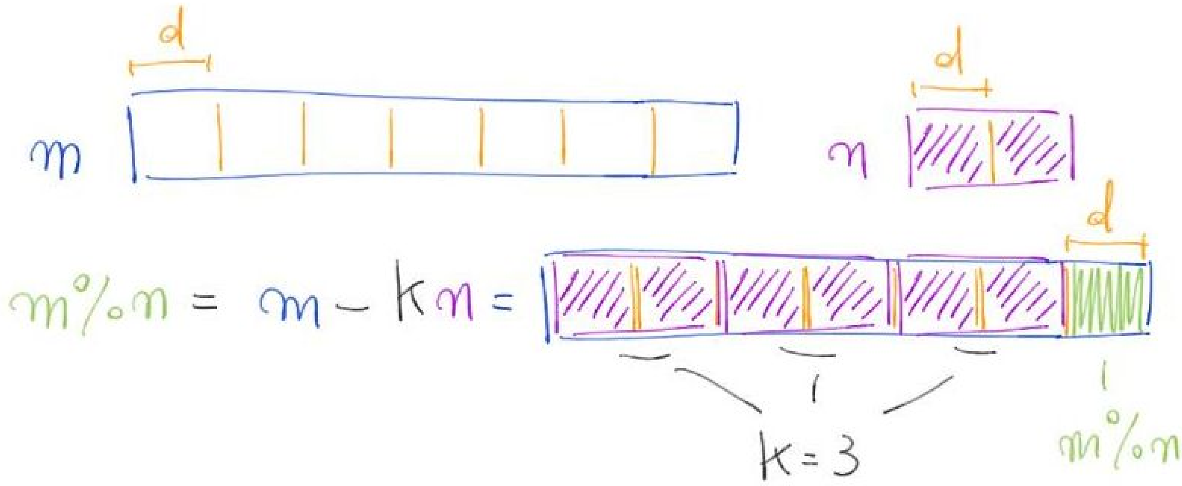
Recorrência que motiva o algoritmo de Euclides:

    mdc(m, n) = { mdc(n, m%n), se n > 0,
                  m, se n = 0. }

Exemplo:
- mdc(12, 18) = mdc(18, 12) = mdc(12, 6) = mdc(6, 0) = 6.
- Note que, se m < n então a primeira aplicação da recorrência
    - realiza a inversão dos valores, pois m % n = m se n > m.
- Observe que, a base da recorrência vale,
    - pois qualquer inteiro positivo divide 0.

Recorrência deriva da propriedade:
- d divide m e n se e somente se d divide n e m%n.
    - Ou seja, o conjunto de divisores comuns a m e n
        - é igual ao conjunto de divisores comuns a n e m%n.
- Demonstração (opcional):
    - Primeiro vamos mostrar a volta da implicação dupla.
        - Suponha que d divide n e m%n, ou seja,
            - n = d * p,
            - m%n = d * q.
        - Queremos mostrar que d divide m.
        - Note que, m = k * n + m%n para algum k >= 0.
        - Substituindo temos,
            - m = k * (d * p) + (d * q) = d * (k * q + h).
        - Ou seja, d divide m.
    - A prova ida é semelhante.

        - Suponha que d divide m e n, ou seja,
            - m = d * p,
            - n = d * q.
        - Queremos mostrar que d divide m%n.
        - Note que, m = k * n + m%n para algum k >= 0.
        - Substituindo temos,
            - (d * p) = k * (d * q) + m%n.
        - Logo,
            - m%n = d * (p) - d * (k * q) = d * (p - k * q)
        - Ou seja, d divide m%n.
- Interpretação:
    - Como m = k * n + m%n temos m%n = m - k * n.
    - Sendo m e n múltiplos de d, o resultado de m - k*n
        - corresponde à remoção de um número inteiro de “d”s.
    - Assim, o que sobra em m%n também é um número inteiro de “d”s,
        - como mostra a figura a seguir.



Da relação de recorrência obtemos o algoritmo recursivo de Euclides:

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

Como a recursão é caudal, podemos transformá-lo

- no algoritmo iterativo de Euclides:

```
int euclidesI1(int m, int n)
{
```

```c
    int r;
    while (1)
    {
        if (n == 0)
            return m;
        r = m % n;
        m = n;
        n = r;
    }
}

int euclidesI2(int m, int n)
{
    int r;
    while (1)
    {
        if (n == 0)
            break;
        r = m % n;
        m = n;
        n = r;
    }
    return m;
}

int euclidesI3(int m, int n)
{
    int r;
    while (n != 0)
    {
        r = m % n;
        m = n;
        n = r;
    }
    return m;
```

```
}
```

Análise de eficiência experimental:

- testar os diferentes algoritmos com $m = 2147483647$ e $n = 2147483646$.

Eficiência de tempo:

- Duas observações centrais na análise de euclidesR(m, n):
    - a eficiência é proporcional ao número de chamadas recursivas.
    - o número de chamadas depende de quão rápido m e n diminuem.
- Propriedade: para $a >= b > 0$ temos $a\%b < a / 2$.
    - Note que, $a = k * b + a\%b \rightarrow a\%b = a - k * b$
    - Intuitivamente,
        - se $b > a/2$ então tirando b de a sobrará menos da metade de a,
        - se $b = a/2$ então o resto $a\%b$ será zero,
        - se $b < a/2$ então tirando vários b de a sobrará algo menor que b.
    - Formalmente,
        - supondo, por contradição, que $a\%b >= a/2$ temos
            - $a\%b = a - k * b >= a/2 \rightarrow 2a - 2 k * b >= a \rightarrow a >= 2 k * b$
        - Absurdo, já que k é o maior inteiro tal que $k * b <= a$.
- Vamos usar o índice i nos parâmetros m_i e n_i para representar
    - seus valores na i-ésima chamada recursiva do algoritmo.
- Assim:
    - $n_{i+1} = m_i \% n_i$ e $m_{i+1} = n_i$,
        - o que implica $n_{i+2} = n_i \% n_{i+1}$.
- Portanto, $n_{i+2} = n_i \% n_{i+1} < n_i / 2$, ou seja,
    - a cada duas chamadas o tamanho de n cai por pelo menos metade.
- Exemplo:
    - $n_2 = n_0\%n_1 < n_0 / 2 = n / 2 = n / 2^1$
    - $n_4 = n_2\%n_3 < n_2 / 2 < n / 4 = n / 2^2$
    - $n_6 = n_4\%n_5 < n_4 / 2 < n / 8 = n / 2^3$
    - $n_8 = n_6\%n_7 < n_6 / 2 < n / 16 = n / 2^4$
    - ...
- Generalizando:
    - $n_t < n / 2^t$, para a 2t-ésima chamada recursiva.
- Lembre que, depois de dividir um número por 2 (arredondando para baixo)
    - mais de log n vezes, ele se torna zero.
- Disso derivamos que o algoritmo faz
    - no máximo 2 lg n + 1 chamadas recursivas.
- Assim, ele leva tempo O(log min(m,n)).
- Note que, a mesma análise se aplica ao algoritmo iterativo,
    - pois a atualização dos valores de m e n a cada iteração
        - é idêntica à atualização à cada chamada recursiva.

Fibonacci

Números de Fibonacci:

- F0 = 0
- F1 = 1
- Fn = Fn-1 + Fn-2

Sequência:

- n 0 1 2 3 4 5 6 7 8 9
- Fn 0 1 1 2 3 5 8 13 21 34

Algoritmo recursivo para Fn

```
long long int fibonacciR(int n)
{
   if (n == 0)
       return 0;
   if (n == 1)
       return 1;
   return fibonacciR(n - 1) + fibonacciR(n - 2);
}
```

Algoritmo iterativo para Fn

```
long long int fibonacciI(int n)
{
   int i;
   long long int proximo, anterior, atual;
   if (n == 0)
       return 0;
   if (n == 1)
       return 1;
   anterior = 0; // Fi-1
   atual = 1;    // Fi
   for (i = 1; i < n; i++)
   {
       proximo = anterior + atual;
       anterior = atual;
```

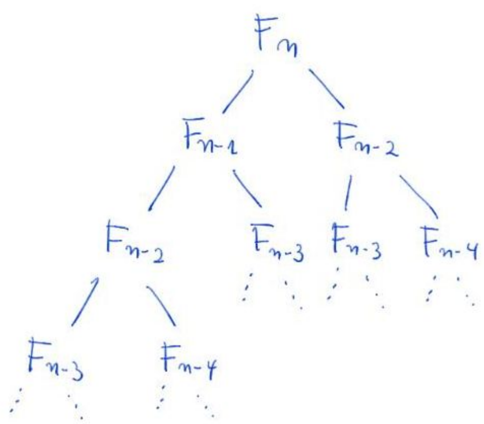
```
        atual = proximo;
    }
    return atual;
}
```

Corretude:

- Como de costume, a corretude do algoritmo recursivo deriva
  - diretamente da corretude da relação de recorrência que o inspirou.
- Sobre o algoritmo iterativo, qual o invariante principal
  - para demonstrar que ele obtem o resultado correto?
- Ou seja, que propriedade/relação se mantém verdadeira
  - ao longo de todas as suas iterações?
- Resp: no início de cada iteração i as variáveis atual e anterior
  - possuem, respectivamente, os valores Fi e Fi-1.

Eficiência:

- Como de costume, a eficiência do algoritmo iterativo
  - deriva do número de vezes que o laço é executado,
    - e neste caso é da ordem de n.
- Já o algoritmo recursivo depende do número total de chamadas recursivas.
  - Vamos obter intuição deste número usando uma árvore de recorrência.



- Note que, ela lembra a árvore de uma exponencial base 2,
  - mas desbalanceada para um lado.
- Observe também o grande número de subproblemas recalculados,
  - sugerindo ineficiência.
- Para obter a ordem do número de chamadas recursivas,
  - vamos analisar a seguinte recorrência, que descreve tal número?
    - T(n) = T(n - 1) + T(n - 2) + 2 para n > 1, T(0) = T(1) = 0.
  - Um limitante inferior para T(n)
    - T(n) = T(n - 1) + T(n - 2) + 2 >= 2 T(n - 2) + 2.

- Assim,

  $T(n) = 2\ T(n - 2) + 2$
  $T(n - 2) = 2\ T(n - 4) + 2$
  $T(n - 4) = 2\ T(n - 6) + 2$
  $T(n - 6) = 2\ T(n - 8) + 2$

- Logo,

  $T(n) = 2\ T(n - 2) + 2$
  $= 2\ (2\ T(n - 4) + 2) + 2 = 4\ T(n - 4) + 6$
  $= 4\ (2\ T(n - 6) + 2) + 6 = 8\ T(n - 6) + 14$
  $= 8\ (2\ T(n - 6) + 2) + 14 = 16\ T(n - 8) + 30$

- Observando o padrão,

  $T(n) = 2^1\ T(n - 2) + 2^2 - 2$
  $= 2^2\ T(n - 4) + 2^3 - 2$
  $= 2^3\ T(n - 6) + 2^4 - 2$
  $= 2^4\ T(n - 8) + 2^5 - 2$

- Chegamos a,

  $T(n) = 2^i\ T(n - 2i) + 2^{(i + 1)} - 2$

- Escolhendo $i = n / 2$,

  $T(n) = 2^{(n / 2)}\ T(n - n) + 2^{((n / 2)+1)} - 2$
  $= 2^{(n / 2 + 1)} - 2$

- Portanto, o número de chamadas recursivas cresce
  - pelo menos como uma exponencial de base 2 e expoente n/2.