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.
  - d | m é nossa notação matemática para "d divide 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, 1, 2 e 4 são divisores comuns de 20 e 12.

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 mdc(m, n).

Algoritmo iterativo simples:

```
#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 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 final da mesma:
    - 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, 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. Pelo invariante todo t > d não o é. Portanto, d é o mdc(m, n).

Eficiência:

- o algoritmo itera por no máximo min(m, n) - 1 vezes
- e em cada iteração realiza um número constante de operações.
- portanto seu tempo é proporcional a min(m, n) no pior caso.
  - outra maneira de dizer "proporcional a bla" é dizer "da ordem de bla",
  - e uma notação matemática para isso é O(bla).

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) ao longo das iterações.
- 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 nos (datado de 300 a.C.). Trata-se do algoritmo de Euclides.

Recorrência que motiva o algoritmo de Euclides:

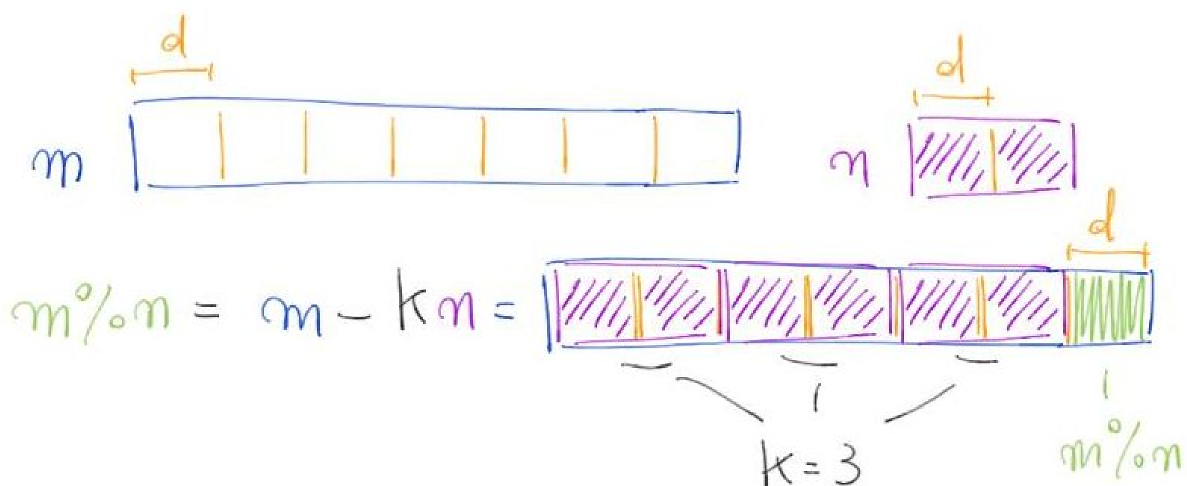
mdc(m,n) = { mdc(n, m%n), se n > 0,
m, se n = 0. }

Exemplo: mdc(12, 18) = mdc(18, 12) = mdc(12, 6) = mdc(6, 0) = 6

- note que, se m < n, 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
        - 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 $k >= 0$
        - portanto, $m = k * (d * p) + (d * q) = d * (k * q + h)$
            - ou seja, d divide m
    - a 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 $k >= 0$
        - portanto, $(d * p) = k * (d * q) + m\%n$
            - 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 $m - k*n$ é a remoção de um número inteiro de “d”s
        - sobrando assim também um número inteiro de “d”s em m%n, 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 euclidesI(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.

Análise de eficiência:

- 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 → 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 - k * b >= a/2 → 2a - 2 k * b >= a → 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 t-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 log n + 1 chamadas recursivas.
- assim, ele leva tempo O(lg 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:

- $F_0 = 0$
- $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$

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 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 fibonacciI(int n)
{
   int i, proximo, anterior, atual;
```

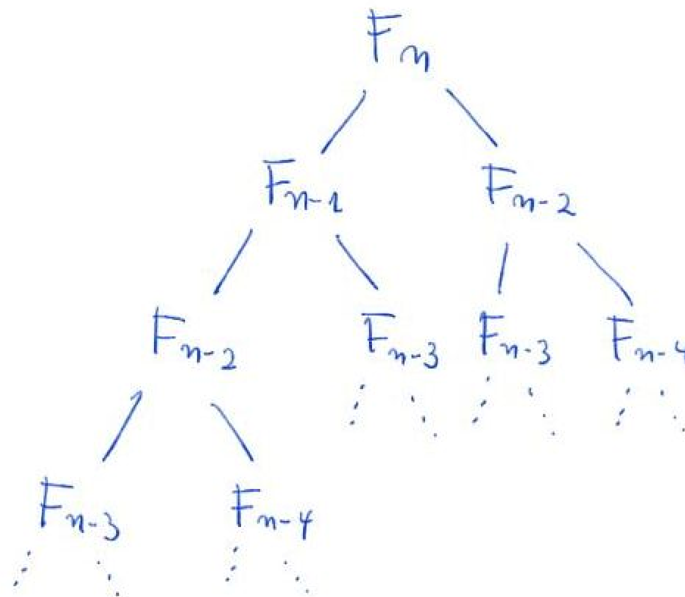
```
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;
    anterior = 0;
    atual = 1;
    for (i = 1; i < n; i++)
    {
        proximo = anterior + atual;
        anterior = atual;
        atual = proximo;
    }
    return atual;
}
```

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 ao 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 $n = 2i$,

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

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