Recursão, fatorial, problema do máximo, máximo divisor comum

"Ao tentar resolver o problema, encontrei obstáculos dentro de obstáculos. Por isso, adotei uma solução recursiva" - citação extraída do livro do Prof. Paulo Feofiloff.

Recursão é uma técnica de projeto de algoritmos que nos permite resolver um problema a partir da solução de instâncias menores do mesmo problema.

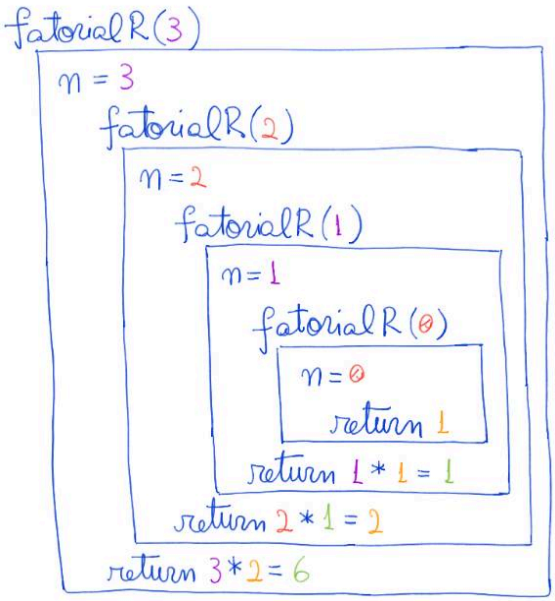
Fatorial

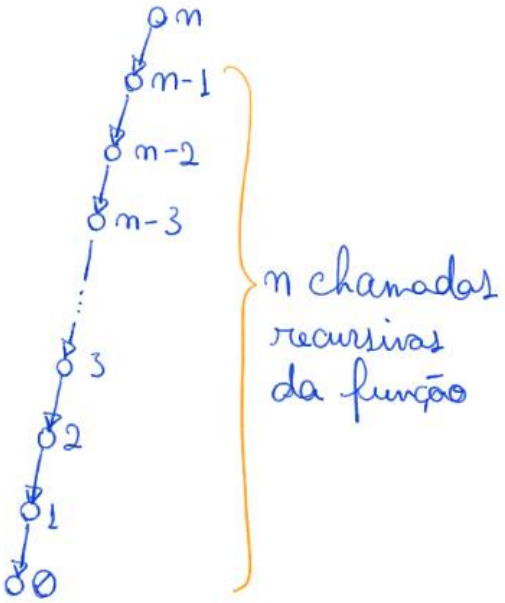
Definição recursiva de fatorial:
n! = { 1, se n = 0,
n * (n - 1)!, se n > 0. }

Código para fatorial recursivo:

```
long long int fatorialR(long long int n)
{
   if (n == 0)
       return 1;
   return n * fatorialR(n - 1);
}
```

Diagrama de execução do fatorial recursivo:





Corretude:

- Deriva diretamente da definição de fatorial.

Eficiência de tempo:

- Quantas chamadas da função recursiva são realizadas?
    - Da ordem de n, i.e., O(n).

- Para verificar isso, seja T(n) o número que desejamos descobrir.
    - Temos, T(n) = T(n - 1) + 1 e T(0) = 1, i.e.,
        - o número de chamadas recursivas para calcular fatorial de n
            - é igual a 1 mais o número de chamadas recursivas
                - para calcular fatorial de n - 1.
        - Além disso, para calcular fatorial de 0
            - se usa apenas uma chamada da função recursiva.

Resolvendo a recorrência por substituição

- Expandindo
    - T(n) = T(n - 1) + 1
    - T(n - 1) = T(n - 2) + 1
    - T(n - 2) = T(n - 3) + 1
    - T(n - 3) = T(n - 4) + 1
    - ...
- Substituindo
    - T(n) = T(n - 1) + 1
    - T(n) = (T(n - 2) + 1) + 1 = T(n - 2) + 2
    - T(n) = (T(n - 3) + 1) + 2 = T(n - 3) + 3
    - T(n) = (T(n - 4) + 1) + 3 = T(n - 4) + 4
    - ...
- Generalizando
    - T(n) = T(n - i) + i
- No final (caso base da recursão) temos
    - n - i = 0 ⇒ i = n
- Portanto, T(n) = T(n - n) + n = T(0) + n = 1 + n.
    - Ou seja, o número de chamadas da função recursiva é n + 1.
- Note que, o número de operações locais
    - realizadas em cada chamada recursiva é constante.
- Por isso, o número total de operações é proporcional a n, i.e., O(n).

Eficiência de espaço: Qual a quantidade de memória auxiliar utilizada?

- Nesse caso é igual à eficiência de tempo, i.e., O(n).
- Isso acontece porque cada nova chamada recursiva
    - utiliza algumas variáveis auxiliares locais,
        - que só começam a ser liberadas
            - depois que a última chamada é resolvida.

Estrutura geral de um programa recursivo

se a instância em questão é pequena,
    resolva-a diretamente;
senão
    reduza-a a uma ou mais instâncias menores do mesmo problema,
    aplique o método recursivamente às instâncias menores
    e use as soluções destas para resolver a instância original.

Problema do máximo

Definição: Dado um vetor v de inteiros com tamanho n,
- devolva o valor do maior elemento deste vetor.

Ideia de uma abordagem recursiva:
- Tome um elemento arbitrário do vetor.
- Encontre recursivamente o máximo do subproblema
  - que contém os demais elementos.
- Compare o elemento tomado com o máximo do subproblema
  - e devolva o maior deles.
- Observe que se o subproblema tem apenas um elemento,
  - então ele pode ser resolvido diretamente.

Algoritmo recursivo que reduz o vetor pelo fim.

```
int maximoRend(int v[], int n) {
   int x; // auxiliar que guarda o máximo do subproblema
   if (n == 1)
       return v[0];
   x = maximoRend(v, n - 1);
   if (x > v[n - 1])
       return x;
   return v[n - 1];
}
```

Algoritmo recursivo que reduz o vetor pelo início.

```
int maximoRbegin(int v[], int inicio, int n) {
   int x; // auxiliar que guarda o máximo do subproblema
   if (inicio == n - 1)
       return v[inicio];
   x = maximoRbegin(v, inicio + 1, n);
   if (x > v[inicio])
       return x;
   return v[inicio];
}

int maximo(int v[], int n)
{
   return maximoRbegin(v, 0, n);
}
```

Eficiência de tempo:

- Qual a ordem do número de operações dos algoritmos recursivos?
    - Da ordem de n, i.e., O(n), pois cada chamada da função
        - realiza um número constante de operações locais
    - e desencadeia no máximo uma chamada recursiva
        - na qual o tamanho do subvetor é reduzido em uma unidade.
- Quiz1: Resolver a recorrência T(n) = 1 + T(n-1) com T(1) = 1.

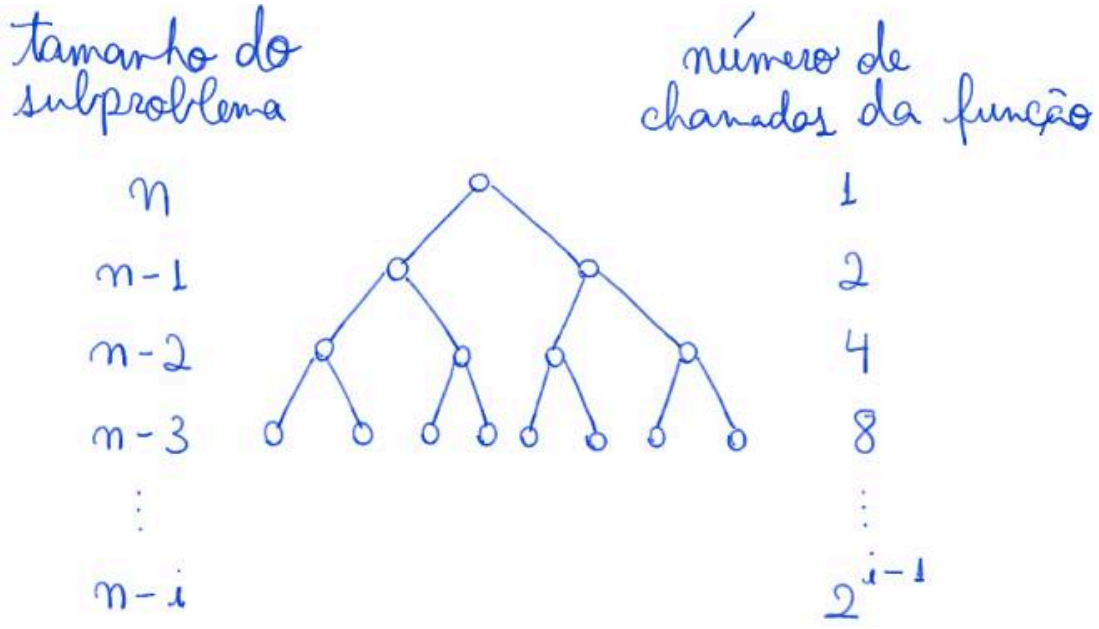
Eficiência de espaço:

- Qual a quantidade de memória auxiliar utilizada?
    - Nesse caso é igual à eficiência de tempo, i.e., O(n).
- Isso acontece porque cada nova chamada recursiva
    - utiliza algumas variáveis auxiliares locais,
- que são colocadas na pilha de execução e só começam a ser liberadas
    - depois que a última chamada é resolvida.

Quiz3: qual a eficiência de pior caso do seguinte algoritmo recursivo?

```
int maximoR(int v[], int n)
{
   int x;
   if (n == 1)
       return v[0];
   if (maximoR(v, n - 1) > v[n - 1])
       return maximoR(v, n - 1);
   return v[n - 1];
}
```

- No pior caso ele leva tempo O(2^n), pois cada chamada da função
    - pode dar origem a duas chamadas recursivas.



- Isso torna esse algoritmo muito ineficiente, embora
    - ele esteja correto e, conceitualmente, seja muito parecido
        - com o primeiro algoritmo recursivo que estudamos.

- Para além da intuição da figura anterior,
    - esse resultado deriva da resolução da seguinte recorrência
        - $T(n) = 2\,T(n-1) + 1$ com $T(1) = 1$

Resolvendo a Recorrência: $T(n) = 2T(n-1) + 1$ e $T(1) = 1$

Expandindo:

- $T(n) = 2T(n-1) + 1$
- $T(n-1) = 2T(n-2) + 1$
- $T(n-2) = 2T(n-3) + 1$
- $T(n-3) = 2T(n-4) + 1$
- …

Substituindo:

- $T(n) = 2T(n-1) + 1$
- $T(n) = 2(2T(n-2) + 1) + 1 = 4T(n-2) + 3$
- $T(n) = 4(2T(n-3) + 1) + 3 = 8T(n-3) + 7$
- $T(n) = 8(2T(n-4) + 1) + 7 = 16T(n-4) + 15$
- …

Arrumando:

- $T(n) = 2^1\,T(n-1) + 2^1 - 1$
- $T(n) = 2^2\,T(n-2) + 2^2 - 1$
- $T(n) = 2^3\,T(n-3) + 2^3 - 1$
- $T(n) = 2^4\,T(n-4) + 2^4 - 1$
- …

Generalizando:

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

No final:

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

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).

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:

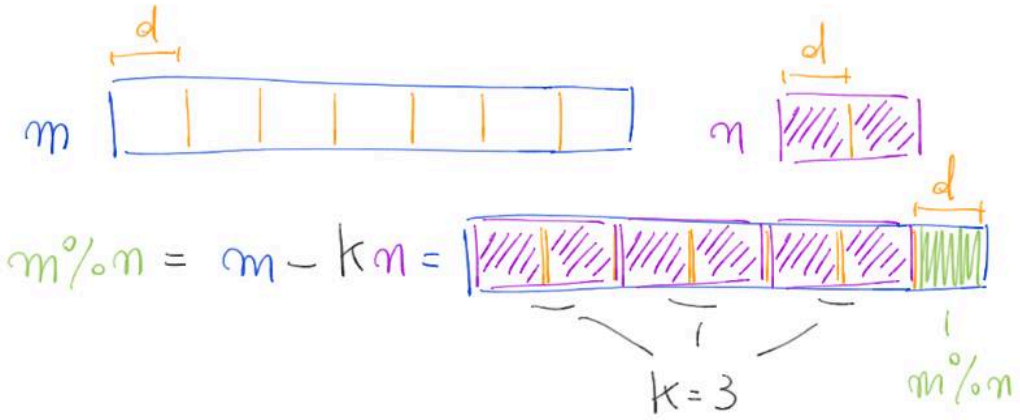
mdc(m, n) = { mdc(n, m%n), se n > 0,
m, se n = 0. }

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

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 → 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 → 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/2), 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 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.