AED1 - Aula 02
Recursão, fatorial e torres de Hanoi

"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

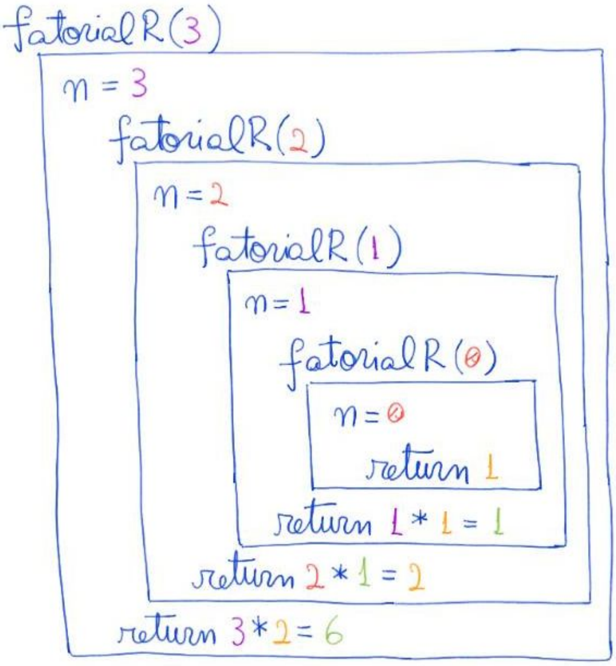
Definição recursiva de fatorial:

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:

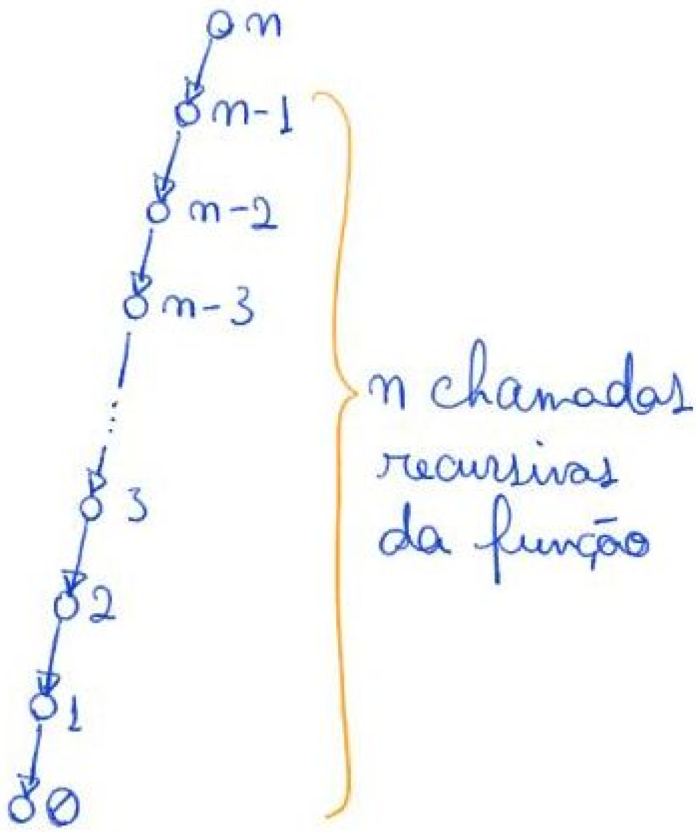


Corretude:

- Deriva diretamente da definição de fatorial.

Eficiência de tempo:

- Qual o número de chamadas recursivas 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) = 0, 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
            - não é realizada nenhuma chamada recursiva.
- Resolvendo a recorrência por substituição
    - 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
    - $n - i = 0 \Rightarrow i = n$
- Portanto, $T(n) = T(n - n) + n = T(0) + n = n$.
    - Ou seja, o número de chamadas recursivas é n.
- 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.

Código para fatorial iterativo:

```
unsigned long long int fatorialI(unsigned long long int n)
{
    unsigned long long int i, fat = 1;
    for (i = 1; i <= n; i++)
        fat *= i; // fat = fat * i;
    return fat;
}
```

Invariantes:

- São propriedades sobre as variáveis de um algoritmo iterativo,
    - que se mantém verdadeiras ao longo das iterações do laço.
- São úteis tanto para demonstrar que um algoritmo está correto,
    - quanto para compreendermos/verificarmos seu comportamento.
- Isso porque, eles descrevem o comportamento global do algoritmo,
    - que pode ser bastante sofisticado e nada óbvio,
        - em um ou mais comportamentos básicos e locais,
            - que podem ser verificados a cada iteração.

Invariante e corretude:

- O invariante principal da função fatorial iterativo é que
    - no início de cada iteração temos fat = (i - 1)!
- Observe que este invariante vale
    - no início da primeira iteração,
    - e se mantém válido de uma iteração para outra.
- Observe também que quando o laço termina
    - ele garante o resultado desejado.
    - Isso porque, no final do laço i = n + 1 e
        - fat = (i - 1)! = (n + 1 - 1)! = n!

Eficiência de tempo:

- Qual o número de iterações em função de n?
    - É igual a n, pois o laço começa com i = 1 e vai até i = n.
- Como o número de operações realizadas em cada iteração é constante,
    - o número total de operações é proporcional a n, i.e., O(n).

Eficiência de espaço:

- Qual a quantidade de memória auxiliar usada?
    - O(1), pois a função possui um pequeno número de variáveis simples.

Estrutura geral de um programa recursivo

se a instância em questão é pequena,
    resolva-a diretamente;
senão
    reduza-a a uma instância menor do mesmo problema,
    aplique o método à instância menor
    e use a solução desta para resolver a instância original.

Torres de Hanoi

Lenda:

- Num templo Hindu, situado no centro do universo, Brahma criou uma torre com 64 discos de ouro e mais duas estacas equilibradas sobre uma plataforma. Então, ordenou aos monges do templo que movessem todos os discos de uma estaca para outra respeitando as seguintes regras: apenas um disco poderia ser movido por vez e nunca um disco maior deveria ficar por cima de um disco menor. Segundo a lenda, quando todos os discos fossem transferidos de uma estaca para a outra, o templo iria desmoronar e o mundo desapareceria.

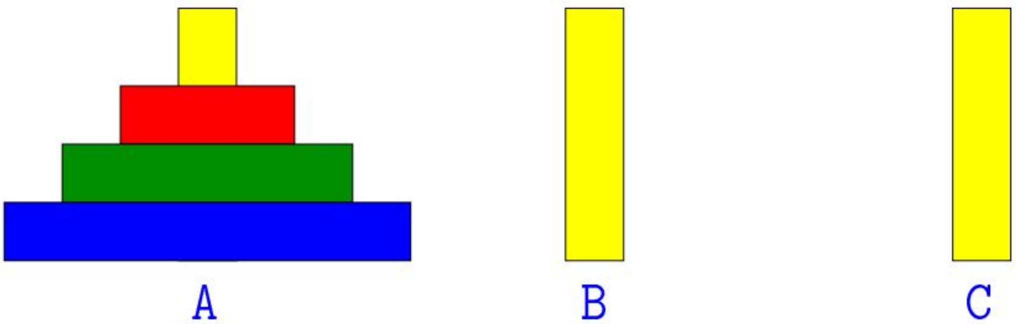
Supondo que a lenda seja verdadeira, será que devemos ficar preocupados com o iminente fim do mundo?

Definição do problema:

- Temos n discos, cada um com um diâmetro diferente,
    - empilhados em ordem decrescente de diâmetro na coluna origem (A).
- Queremos mover todos os discos de A até a coluna destino (C)
    - usando a coluna B como auxiliar.
- Mas devemos respeitar as regras:
    - podemos mover apenas um disco por vez,
    - um disco de diâmetro maior não pode ficar
        - sobre um disco de diâmetro menor.

Estratégia para atacar o problema:

- Embora não seja óbvio qual é o primeiro movimento,
    - o movimento do meio nós conseguimos deduzir qual é.
- Por exemplo, na seguinte instância do problema
    - o movimento do meio é mover o disco azul de A para C,
        - pois, sendo o maior disco ele deve necessariamente
            - estar na base da torre de discos no destino.



Para conseguirmos mover o disco azul,

- primeiro precisamos liberá-lo removendo os demais discos de cima dele,
    - ou seja, tudo que está sobre ele deve ser movido para B.
- Então, o azul deve ser movido para C.
- Depois, tudo que está em B deve ser movido para C.

Para descrever o problema (e nossa solução) de modo mais claro e preciso,

- vamos chamar de Hanoi(n, A, B, C) o problema de mover:
    - os n menores discos
    - da torre A para a torre C

- usando a torre B como auxiliar.

Assim, nossa solução para Hanoi(n, A, B, C) pode ser descrita como:
- Hanoi(n - 1, A, C, B)
- mover disco restante (n-ésimo) de A para C
- Hanoi(n - 1, B, A, C)

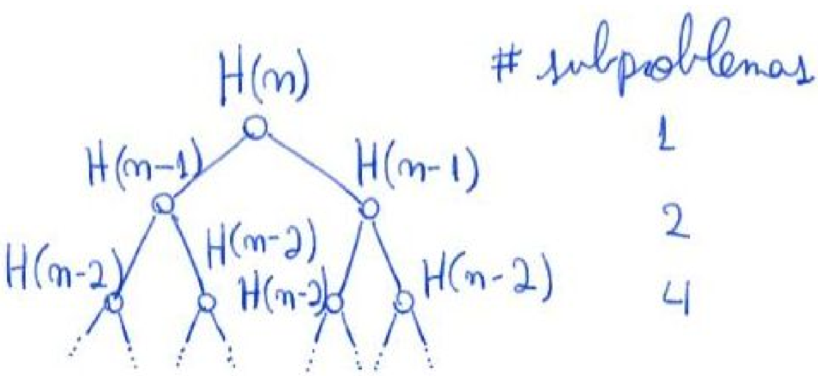
Note que, estamos reduzindo o problema de mover n discos
- para 2 problemas de mover n - 1 discos.

Um adendo importante é que,
- quando soubermos resolver o problema diretamente
  - paramos de reduzir.
- Neste problema, isso acontece quando n = 0,
  - pois mover 0 discos é trivial.

Código recursivo para Hanoi:
```
void Hanoi(int n, char origem, char auxiliar, char destino)
{
   if (n == 0)
       return;
   Hanoi(n - 1, origem, destino, auxiliar);
   printf("mova o disco %d de %c para %c.\n", n, origem, destino);
   Hanoi(n - 1, auxiliar, origem, destino);
}
```

Corretude:
- Deriva diretamente da definição do problema.

Eficiência de tempo:



H(n)=Hanoi (n, origem, auxiliar, destino)

- Qual o número de movimentos realizados em função de n?
  - Para descobrirmos a resposta, seja T(n) este número.
- Temos que, $T(n) = 2\ T(n - 1) + 1$ e $T(0) = 0$, i.e.,
  - o número de movimentos para mover n discos para o destino é igual
    - ao número de movimentos para mover n - 1 discos para uma torre auxiliar,
    - mais um movimento para mover o n-ésimo disco para o destino,
    - mais o número de movimentos para mover n - 1 discos da torre auxiliar para o destino.
  - Além disso, o número de movimentos para mover 0 discos é 0.
- Resolvendo a recorrência por substituição
  - $T(n) = 2\ T(n - 1) + 1$
  - $T(n - 1) = 2\ T(n - 2) + 1$
  - $T(n - 2) = 2\ T(n - 3) + 1$
  - $T(n - 3) = 2\ T(n - 4) + 1$
  - ...
- Substituindo
  - $T(n) = 2\ T(n - 1) + 1$
  - $T(n) = 2\ (2\ T(n - 2) + 1) + 1 = 4\ T(n - 2) + 3$
  - $T(n) = 4\ (2\ T(n - 3) + 1) + 3 = 8\ T(n - 3) + 7$
  - $T(n) = 8\ (2\ T(n - 4) + 1) + 7 = 16\ T(n - 4) + 15$
  - ...
- Observe que
  - $T(n) = 2\ T(n - 1) + 1 = 2^1\ T(n - 1) + 2^1 - 1$
  - $T(n) = 4\ T(n - 2) + 3 = 2^2\ T(n - 2) + 2^2 - 1$
  - $T(n) = 8\ T(n - 3) + 7 = 2^3\ T(n - 3) + 2^3 - 1$
  - $T(n) = 16\ T(n - 4) + 15 = 2^4\ T(n - 4) + 2^4 - 1$
  - …
- Generalizando
  - $T(n) = 2^k\ T(n - k) + 2^k - 1$
- No final (caso base da recursão) temos
  - $n - k = 0 \Rightarrow k = n$
- Portanto, $T(n) = 2^n\ T(n - n) + 2^n - 1 = 2^n\ T(0) + 2^n - 1 = 2^n - 1$,
  - pois $T(0) = 0$.
- Ou seja, o número de movimentos cresce exponencialmente.
  - Mais especificamente, como uma exponencial de base 2
    - em função do número de discos n.
- Vale observar que, nossa solução não realiza movimentos desnecessários.

    - Por isso, não é possível resolver este problema com menos movimentos.
- Assim, voltemos à nossa preocupação inicial
    - com os monges e o fim do mundo.
- Quantos movimentos eles tem que fazer pra mover 64 discos?
    - 2^64 - 1 ~= 1,84 * 10^19
- Supondo que levem um segundo para realizar cada movimento,
    - eles precisarão de aproximadamente:
        - 3,07 * 10^17 minutos,
        - 5,11 * 10^15 horas,
        - 2,13 * 10^14 dias,
        - 5,83 * 10^11 anos ~= 583 bilhões de anos.
    - Podemos dormir tranquilos!
- E qual o número de operações realizadas por nossa solução?
    - Note que, o número de chamadas recursivas
        - é proporcional ao número de movimentos.
    - Além disso, o número de operações locais
        - realizadas em cada chamada recursiva é constante.
    - Por isso, o número total de operações é proporcional a 2^n,
        - i.e., O(2^n).

Eficiência de espaço:
- Qual a quantidade de memória auxiliar utilizada pelo algoritmo?
    - Da ordem de n, i.e., O(n).
- Isso porque, cada chamada recursiva reduz n em 1.
- Assim, teremos no máximo n chamadas recursivas encadeadas
    - em qualquer momento da execução do algoritmo.
- E cada chamada da função
    - tem um número constante de variáveis locais simples.