Notas de aula 20/05

ANÁLISE DA COMPLEXIDADE DE ALGORITMOS RECURSIVOS

Cálculo do fatorial:

O código abaixo apresenta uma função recursiva para cálculo de fatorial:

```
int fatorial_rec(int n) {
  if(n == 1)
      return 1;
  else
      return n * fatorial_rec(n - 1);
}
```

Complexidade de tempo: a comparação feita no if terá custo 1. A subtração feita em fatorial_rec(n - 1) terá custo 1. A multiplicação n * fatorial_rec(n - 1) também terá custo 1. Não sabemos, entretanto, qual será o custo para executar a função fatorial_rec(n - 1). Podemos dizer então, que a complexidade de tempo desta função pode ser descrita pela seguinte **relação de recorrência**:

T(n) = 3 + T(n - 1)

T(1) = 1 (quando *n* é igual a 1, teremos apenas o custo da comparação feita no if)

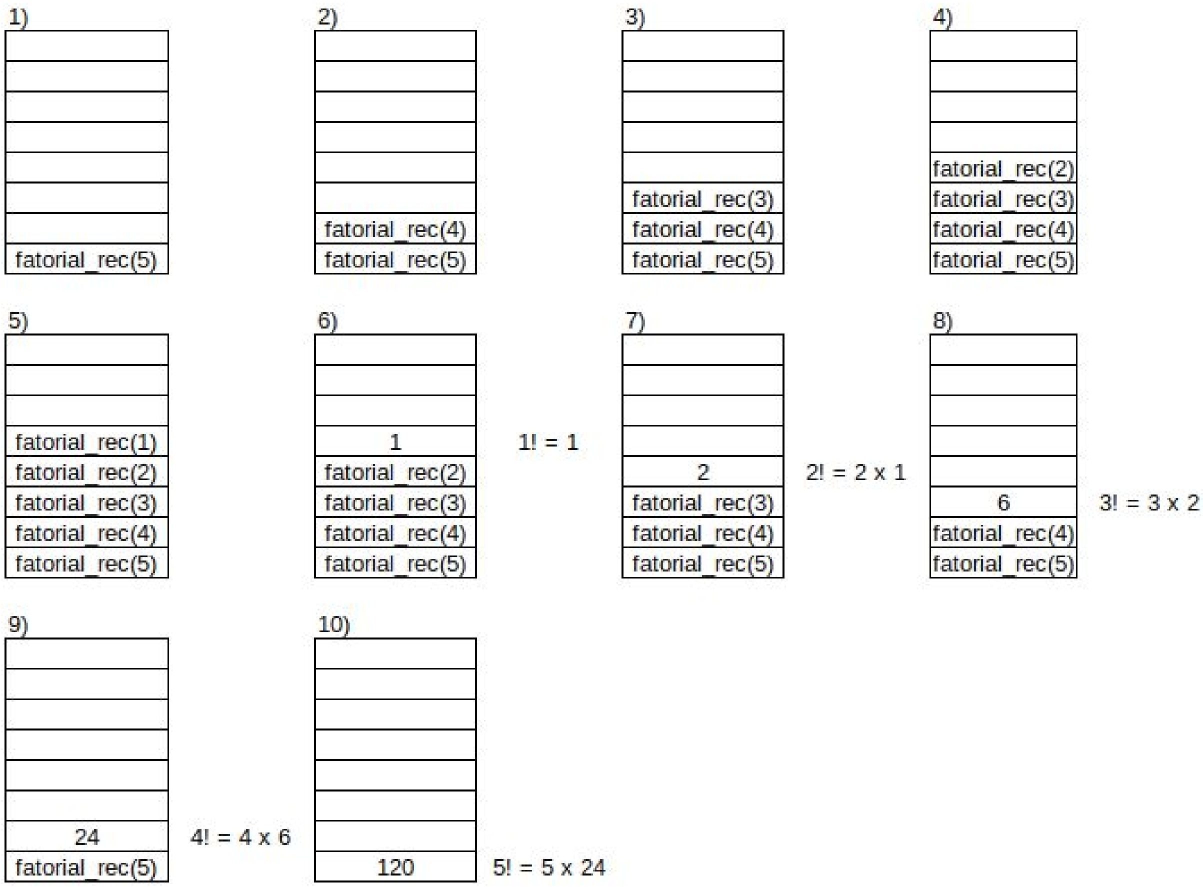
Resolvemos a relação de recorrência até chegar ao caso base, T(1):

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

Reorganizando:

T(n) = 3 + ~~T(n - 1)~~
~~T(n - 1)~~ = 3 + ~~T(n - 2)~~
~~T(n - 2)~~ = 3 + ~~T(n - 3)~~
~~T(n - 3)~~ = 3 + ~~T(n - 4)~~
…
~~T(n - (n - 1))~~ = 1
T(n) = 3 + 3 + 3 + 3 + … + 3 + 1
T(n) = n * 3 + 1
T(n) = 3n + 1 → **O(n)**

Complexidade de espaço: vamos considerar as chamadas de função que são feitas para n = 5:



Sendo *n* = 5, é possível ver no passo 5, que até *n* chamadas da função ocupam a pilha de execução do programa. Cada chamada precisa guardar informações como endereço de retorno, parâmetros e variáveis locais da função. Para guardar essas informações o custo é constante. O que vai definir a complexidade de espaço da função é a quantidade máxima de chamadas ocupando a pilha, que é **O(n)**.

Soma dos N primeiros números:

A função abaixo calcula a soma dos n primeiros números de forma recursiva:

```
int soma_n_rec(int n){
  if(n == 1)
      return 1;
  else
      return n + soma_n_rec(n - 1);
}
```

<u>Complexidade de tempo</u>: o código é muito parecido com o da função fatorial recursiva, exceto que temos a soma:

n + soma_n_rec(n - 1)

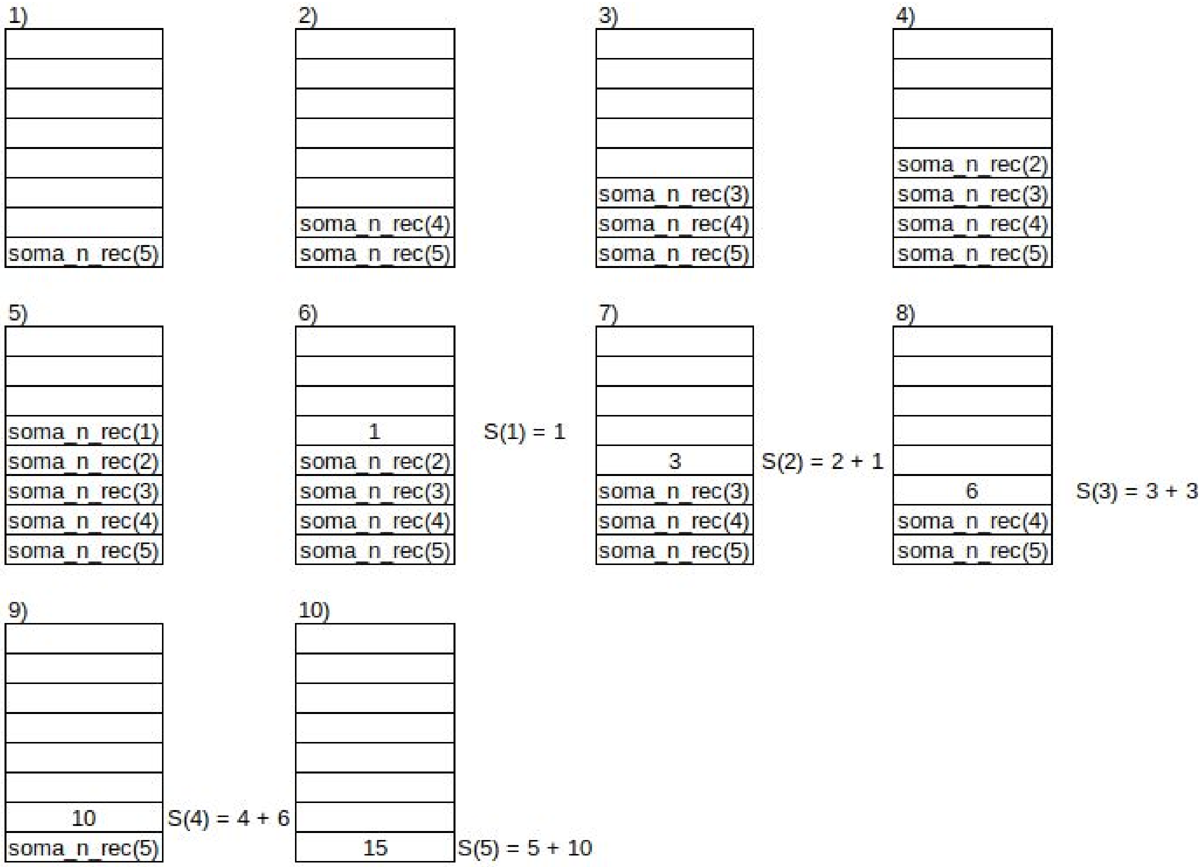
em vez da multiplicação:

n * fatorial_rec(n - 1).

Isso quer dizer que vamos ter a mesma relação de recorrência:

T(n) = 3 + T(n - 1)
T(1) = 1
ou seja, a complexidade será **O(n)**.

Complexidade de espaço: vamos considerar as chamadas de função que são feitas para n = 5:



no passo 5 temos *n* chamadas para a função, ocupando *n* unidades na pilha de execução, o que leva a uma complexidade **O(n)**.

Busca do maior elemento do vetor:

O código abaixo apresenta uma função que busca o maior elemento de um vetor de forma recursiva:

```c
int maior_rec(int n, int v[]){
  if(n == 1)
      return v[0];
  else {
      int maior;
      maior = maior_rec(n - 1, v);
      if(maior > v[n - 1])
          return maior;
      else
          return v[n - 1];
```

Complexidade de tempo:

- o custo da comparação na linha 2 é 1.
- o acesso a um elemento do vetor via índice, na linha 3 tem custo 1.
- o custo da declaração na linha 5 é 1.
- o custo da atribuição na linha 6 tem custo 1 (considerando apenas o custo da atribuição, visto que ainda não é sabido o custo de chamar a função). Nesta mesma linha existe uma subtração que tem custo 1. Assim, a linha toda tem custo 2.
- a comparação da linha 7 tem custo 1. Nesta mesma linha existe uma subtração que tem custo 1. Também é feito o acesso a um elemento do vetor via índice, também com custo 1. Assim, a linha toda tem custo 3.
- na última linha, a operação de subtração tem custo 1, enquanto o acesso a um elemento do vetor via índice também tem custo 1, totalizando custo 2 para a linha toda.

ou seja:
T(n) = 10 + T(n - 1)
T(1) = 1

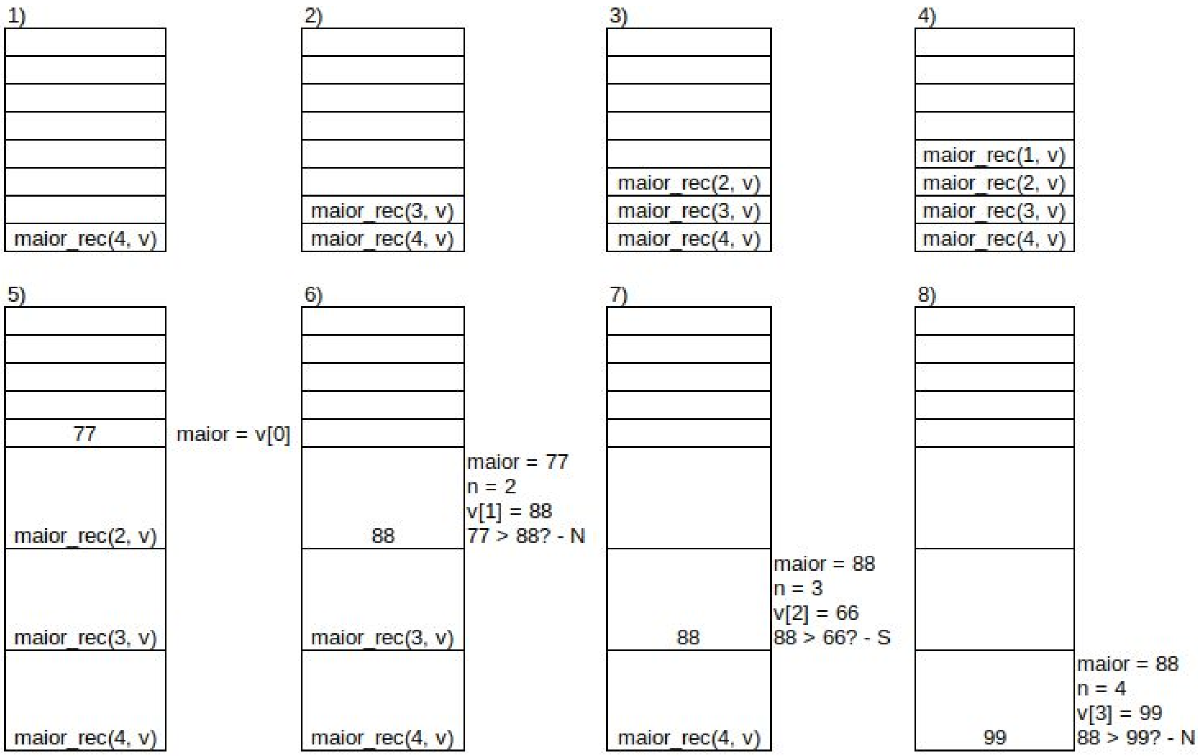
Resolvendo a relação de recorrência:

T(n) = 10 + T(n - 1)
T(n - 1) = 10 + T(n - 2)
T(n - 2) = 10 + T(n - 3)
T(n - 3) = 10 + T(n - 4)
…
T(n - (n - 1)) = 1 (caso base)

Reorganizando:

T(n) = 10 + ~~T(n - 1)~~
~~T(n - 1)~~ = 10 + ~~T(n - 2)~~
~~T(n - 2)~~ = 10 + ~~T(n - 3)~~
~~T(n - 3)~~ = 10 + ~~T(n - 4)~~
…
~~T(n - (n - 1))~~ = 1
T(n) = 10 + 10 + 10 + 10 + … + 1
T(n) = n * 10 + 1
T(n) = 10n + 1 → **O(n)**

Complexidade de espaço: analisando a pilha de chamadas para n = 4:



novamente a complexidade será **O(n)**, conforme o uso da pilha no passo 4.

Cálculo do n-ésimo termo da série Fibonacci:

Uma função recursiva para calcular o n-ésimo termo da série de Fibonacci apresentada abaixo:

```
int fib_rec(int x){
      if(x == 0)
          return 0;
      if(x == 1)
          return 1;
      else
          return fib_rec(x - 1) + fib_rec(x - 2);
}
```

<u>Complexidade de tempo</u>: o primeiro if tem uma comparação com custo 1. O segundo if também tem uma comparação com custo 1. A instrução de soma dentro do else tem custo 1. A subtração em fib_rec(x - 1) tem custo 1, assim como a subtração em fib_rec(x - 2). Estas operações simples totalizam um custo 5. Precisamos ainda saber qual o custo de chamar as funções fib_rec(x - 1) e fib_rec(x - 1). Podemos então escrever a seguinte relação de recorrência:

$T(n) = 5 + T(n - 1) + T(n - 2)$

$T(1) = T(0) = 1$

vamos simplificar esta relação de recorrência considerando que T(n - 2) será no máximo tão custosa quanto T(n - 1). Como 5 é uma constante vamos reescrever como c. Temos então:

T(n) = c + 2T(n - 1)
T(1) = T(0) = 1

Resolvendo a relação de recorrência:

T(n) = c + 2T(n - 1)
2T(n - 1) = 2(c + 2T(n - 1 - 1)) = 2c + 4T(n - 2)
4T(n - 2) = 4(c + 2T(n - 2 - 1)) = 4c + 8T(n - 3)
8T(n - 3) = 8(c + 2T(n - 3 - 1)) = 8c + 16T(n - 4)
...

É possível perceber um padrão nas fórmulas acima: todos os termos são precedidos por potências de 2:

$2^0 T$ (n - 0) = 2^0 c + 2^1 T(n - 1)
$2^1 T$ (n - 1) = 2^1 c + 2^2 T(n - 2)
$2^2 T$ (n - 2) = 2^2 c + 2^3 T(n - 3)
$2^3 T$ (n - 3) = 2^3 c + 2^4 T(n - 4)
…
$2^{n-1} T$ (n - (n - 1)) = 2^{n-1} * 1 (caso base)

Somando todos os termos:

T(n) = $2^{n-1} + 2^0$ c + 2^1 c + 2^2 c + 2^3 c + ... + $2^{n-2} c$

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

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

Precisamos resolver o somatório. Para facilitar vamos considerar que *m = n - 2,* assim o somatório pode ser reescrito como:

$$\sum_{i=o}^{m} 2^i$$

Vamos aplicar a técnica da perturbação para resolver o somatório. Esta técnica consiste em:

- “Perturbar” o somatório adicionando um termo a mais;
- Reescrever o somatório como:
    - Primeiro termo + somatório dos próximos termos;
    - Último termo + somatório dos termos anteriores.

Ou seja:

$$\sum_{i=o}^{m+1} 2^i = 2^0 + \sum_{i=o}^{m} 2^{i+1}$$

$$\sum_{i=o}^{m+1} 2^i = \sum_{i=o}^{m} 2^i + 2^{m+1}$$

Sendo assim, é possível dizer que:

$$2^0 + \sum_{i=o}^{m} 2^{i+1} = \sum_{i=o}^{m} 2^i + 2^{m+1}$$

Logo:

$$1 + \sum_{i=o}^{m} 2^{i+1} = \sum_{i=o}^{m} 2^i + 2^{m+1}$$

$$1 + \sum_{i=o}^{m} 2^i 2 = \sum_{i=o}^{m} 2^i + 2^{m+1}$$

$1 + 2\sum_{i=o}^{m} 2^i = \sum_{i=o}^{m} 2^i + 2^{m+1}$

$\sum_{i=o}^{m} 2^i = 2^{m+1} - 1$

Como $m = n - 2$:

$\sum_{i=o}^{n-2} 2^i = 2^{n-1} - 1$

Substituindo na fórmula original:

$2^{n-1} + c\,(2^{n-1} - 1)$

$2^{n-1} + 2^{n-1}c - c$, logo, a complexidade é **O(** 2^n **)**

<u>Complexidade de espaço</u>: analisando a pilha de chamadas para n = 5:

1)	2)	3)	4)	5)	6)
				fib_rec(1)	1
			fib_rec(2)	fib_rec(2)	fib_rec(2)
		fib_rec(3)	fib_rec(3)	fib_rec(3)	fib_rec(3)
	fib_rec(4)	fib_rec(4)	fib_rec(4)	fib_rec(4)	fib_rec(4)
fib_rec(5)	fib_rec(5)	fib_rec(5)	fib_rec(5)	fib_rec(5)	fib_rec(5)

7)	8)	9)	10)	11)	12)
fib_rec(0)	0				
1	1		fib_rec(1)	1	
fib_rec(2)	fib_rec(2)	1	1	1	
fib_rec(3)	fib_rec(3)	fib_rec(3)	fib_rec(3)	fib_rec(3)	2
fib_rec(4)	fib_rec(4)	fib_rec(4)	fib_rec(4)	fib_rec(4)	fib_rec(4)
fib_rec(5)	fib_rec(5)	fib_rec(5)	fib_rec(5)	fib_rec(5)	fib_rec(5)

13)	14)	15)	16)	17)	18)
			fib_rec(0)	0	
	fib_rec(1)	1	1	1	
fib_rec(2)	fib_rec(2)	fib_rec(2)	fib_rec(2)	fib_rec(2)	1
2	2	2	2	2	2
fib_rec(4)	fib_rec(4)	fib_rec(4)	fib_rec(4)	fib_rec(4)	fib_rec(4)
fib_rec(5)	fib_rec(5)	fib_rec(5)	fib_rec(5)	fib_rec(5)	fib_rec(5)

19)	20)	21)	22)	23)	24)
					fib_rec(0)
			fib_rec(1)	1	1
		fib_rec(2)	fib_rec(2)	fib_rec(2)	fib_rec(2)
	fib_rec(3)	fib_rec(3)	fib_rec(3)	fib_rec(3)	fib_rec(3)
3	3	3	3	3	3
fib_rec(5)	fib_rec(5)	fib_rec(5)	fib_rec(5)	fib_rec(5)	fib_rec(5)

25)	26)	27)	28)	29)	30)
0					
1		fib_rec(1)	1		
fib_rec(2)	1	1	1		
fib_rec(3)	fib_rec(3)	fib_rec(3)	fib_rec(3)	2	
3	3	3	3	3	
fib_rec(5)	fib_rec(5)	fib_rec(5)	fib_rec(5)	fib_rec(5)	5

existem situações onde até n chamadas ocupam a pilha, portanto, pode-se dizer que a complexidade de espaço é **O(n)**.

Atividade:

Calcular a complexidade de tempo da função recursiva que apresenta um número decimal em sua forma binária (exercício da aula passada):

Solução:

Abaixo uma possível implementação da função:

```
void imprime_bin_rec(int x) {
  if (x == 0)
      printf("0");
  else if(x == 1)
      printf("1");
  else {
      imprime_bin_rec(x / 2);
      printf("%d", x % 2);
  }
}
```

O primeiro if terá custo 2 (comparação + printf). O segundo if também terá custo 2 (comparação + printf). O else terá custo 3 (divisão + resto + printf) + o custo de chamar a função. Assim a fórmula de recorrência da função é dada por:

$T(n) = 7 + T(n/2)$

$T(1) = T(0) = 2$

Resolvendo:

$T(n) = 7 + T(n/2)$

$T(n/2) = 7 + T(n/4)$

$T(n/4) = 7 + T(n/8)$

T(n/8) = 7 + T(n/16)

…

É possível perceber um padrão:

T(n/ 2^0) = 7 + T(n/ 2^1)

T(n/ 2^1) = 7 + T(n/ 2^2)

T(n/ 2^2) = 7 + T(n/ 2^3)

T(n/ 2^3) = 7 + T(n/ 2^4)

…

T(n/ 2^k) = 2 (caso base, quando 2^k = n)

Somando os termos, teremos 2 + 7 + 7 + 7 + … + 7. O número sete é somado *k* vezes. Para descobrir o valor de *k* podemos usar a seguinte propriedade:

2^k = n $\Leftrightarrow$ k = $log_2 n$

Logo:

T(n) = 2 + $log_2 n \, * \, 7$ → **O(n)**