BCC202 - Estrutura de Dados I

Aula 07: Recursividade

Reinaldo Fortes

Universidade Federal de Ouro Preto, UFOP
Departamento de Ciência da Computação, DECOM

Website: **www.decom.ufop.br/reifortes**
Email: **reifortes@iceb.ufop.br**

Material elaborado com base nos slides do Prof. Túlio Toffolo (curso de 2013/01).

2013/02

Conteúdo

Conteúdo

Recursividade

- A recursividade é uma estratégia que pode ser utilizada sempre que uma função f pode ser escrita em função dela própria.
- Exemplo: Cálculo do Fatorial:

$$n! = n * (n-1) * (n-2) * (n-3) * ... * 1$$

Como $(n-1)! = (n-1) * (n-2) * (n-3) * ... * 1$

Então: $n! = n * (n-1)!$

Recursividade

- **Definição**: dentro do corpo de uma função, chamar novamente a própria função.
  - **Recursão direta**: a função A chama a própria função A.
  - **Recursão indireta**: a função A chama uma função B que, por sua vez, chama A.

Condição de Parada

- Nenhum programa nem função pode ser exclusivamente definido por si:
  - Um programa seria um loop infinito.
  - Uma função teria definição circular.
- **Condição de parada**:
  - Permite que o procedimento pare de se executar.
  - Exemplo: $\boldsymbol{f(x) > 0}$ onde $\boldsymbol{x}$ é decrescente.

Consumo de Memória

- Para cada chamada de uma função, recursiva ou não, os parâmetros e as variáveis locais são empilhados na pilha de execução.
- Qual a implicação disto?
- Internamente, quando uma função é chamada, é criado um **Registro de Ativação** na **Pilha de Execução** do programa.
- Este registro armazena os **parâmetros** e **variáveis locais** da função bem como o "**ponto de retorno**" no programa que chamou essa função.
- Ao final da execução dessa função, o registro é desempilhado e a execução volta ao subprograma que chamou a função.

Consumo de Memória

- Para cada chamada de uma função, recursiva ou não, os parâmetros e as variáveis locais são empilhados na pilha de execução.
- Qual a implicação disto? **Maior consumo de memória!**
- Internamente, quando uma função é chamada, é criado um **Registro de Ativação** na **Pilha de Execução** do programa.
- Este registro armazena os **parâmetros** e **variáveis locais** da função bem como o "**ponto de retorno**" no programa que chamou essa função.
- Ao final da execução dessa função, o registro é desempilhado e a execução volta ao subprograma que chamou a função.

Consumo de Memória

- Para cada chamada de uma função, recursiva ou não, os parâmetros e as variáveis locais são empilhados na pilha de execução.
- Qual a implicação disto? **Maior consumo de memória!**
- Internamente, quando uma função é chamada, é criado um **Registro de Ativação** na **Pilha de Execução** do programa.
- Este registro armazena os **parâmetros** e **variáveis locais** da função bem como o "**ponto de retorno**" no programa que chamou essa função.
- Ao final da execução dessa função, o registro é desempilhado e a execução volta ao subprograma que chamou a função.

Consumo de Memória - Exemplo: Fatorial

```
1  int fat1(int n) {
2    int r;
3    if(n <= 0)
4      r = 1;
5    else
6      r = n * fat1(n-1);
7    return r;
8  }
9
10 int fat2(int n) {
11   if(n <= 0)
12     return 1;
13   else
14     return n * fat2(n-1);
15 }
```

```
16 void main() {
17   int f, g;
18   f = fat1(4);
19   g = fat2(4);
20   printf("%d -- %d", f, g);
21 }
```

- Qual a diferença entre `fat1` e `fat2`?
- Qual dos dois você escolheria? Justifique.

Consumo de Memória - Exemplo: Fatorial

```
1  int fat1(int n) {
2    int r;
3    if(n <= 0)
4      r = 1;
5    else
6      r = n * fat1(n-1);
7    return r;
8  }
9
10 int fat2(int n) {
11   if(n <= 0)
12     return 1;
13   else
14     return n * fat2(n-1);
15 }
```

```
16 void main() {
17   int f, g;
18   f = fat1(4);
19   g = fat2(4);
20   printf("%d -- %d", f, g);
21 }
```

- Qual a diferença entre `fat1` e `fat2`?
- Qual dos dois você escolheria? Justifique.

Consumo de Memória - Exemplo: Fatorial

```
1  int fat1(int n) {
2    int r;
3    if(n <= 0)
4      r = 1;
5    else
6      r = n * fat1(n-1);
7    return r;
8  }
9
10 int fat2(int n) {
11   if(n <= 0)
12     return 1;
13   else
14     return n * fat2(n-1);
15 }
```

```
16 void main() {
17   int f, g;
18   f = fat1(4);
19   g = fat2(4);
20   printf("%d -- %d", f, g);
21 }
```

- Qual a diferença entre `fat1` e `fat2`?
- Qual dos dois você escolheria? Justifique.

Consumo de Memória - Exemplo: Fatorial

- A complexidade de tempo do fatorial recursivo é $O(n)$ (veremos como definir isto através de **equação de recorrência**, em breve).
- Mas a complexidade de espaço também é $O(n)$, devido à pilha de execução.
- Já no fatorial não recursivo a complexidade de espaço é $O(1)$.

```
1 int fatIter(int n) {
2   int f = 1;
3   while(n > 0) {
4     f = f * n;
5     n = n - 1;
6   }
7   return f;
8 }
```

Conclusão

- Portanto, podemos concluir que a recursividade nem sempre é a melhor solução, mesmo quando a definição matemática do problema é feita em termos recursivos.

- Além disso, pode-se afirmar que:
  - Todo algoritmo recursivo tem uma versão não recursiva.
  - A questão é: **vale a pena implementar esta versão não-recursiva?**

Exemplo: Série de Fibonacci

- Outro exemplo clássico de recursividade é a **Série de Fibonacci**, definida pela expressão:

$$F(n) = \begin{cases} F(n-1) + F(n-2) & \text{se } x > 2 \\ F(n) = 1 & \text{se } x = 1 \\ F(n) = 0 & \text{se } x = 0 \end{cases}$$

- Originando a série: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Exemplo: Série de Fibonacci – Soluções

Solução Recursiva:

```
1 int fibR(int n) {
2   if(n <= 0)
3     return 0;
4   else if(n == 1)
5     return 1;
6   else
7     return fibR(n-1) +
8            fibR(n-2);
9 }
```

Solução Iterativa:

```
1 int fibI(int n) {
2   int i, k, F;
3   i = 1; F = 0;
4   for(k = 1; k <= n; k++) {
5     F += i;
6     i = F - i;
7   }
8   return F;
9 }
```

Exemplo: Série de Fibonacci – Soluções

Solução Recursiva:

```
1 int fibR(int n) {
2   if(n <= 0)
3     return 0;
4   else if(n == 1)
5     return 1;
6   else
7     return fibR(n-1) +
8            fibR(n-2);
9 }
```

Solução Iterativa:

```
1 int fibI(int n) {
2   int i, k, F;
3   i = 1; F = 0;
4   for(k = 1; k <= n; k++) {
5     F += i;
6     i = F - i;
7   }
8   return F;
9 }
```

- Um mesmo n é computado várias vezes.
- Custo: $O(\phi^n)$:
  - $\phi = 1{,}61803...$ (*Golden Ratio*).
- **Complexidade Exponencial**.

Exemplo: Série de Fibonacci – Soluções

Solução Recursiva:

```
1 int fibR(int n) {
2   if(n <= 0)
3     return 0;
4   else if(n == 1)
5     return 1;
6   else
7     return fibR(n-1) +
8            fibR(n-2);
9 }
```

- Um mesmo n é computado várias vezes.
- Custo: $O(\phi^n)$:
  - $\phi = 1{,}61803...$ (*Golden Ratio*).
- **Complexidade Exponencial**.

Solução Iterativa:

```
1 int fibI(int n) {
2   int i, k, F;
3   i = 1; F = 0;
4   for(k = 1; k <= n; k++) {
5     F += i;
6     i = F - i;
7   }
8   return F;
9 }
```

- Custo: $O(n)$
- **Complexidade Linear**!

Exemplo: Série de Fibonacci – Soluções

Solução Recursiva:

```
1 int fibR(int n) {
2   if(n <= 0)
3     return 0;
4   else if(n == 1)
5     return 1;
6   else
7     return fibR(n-1) +
8            fibR(n-2);
9 }
```

- Um mesmo n é computado várias vezes.
- Custo: $O(\phi^n)$:
  - $\phi = 1{,}61803...$ (*Golden Ratio*).
- **Complexidade Exponencial**.

Solução Iterativa:

```
1 int fibI(int n) {
2   int i, k, F;
3   i = 1; F = 0;
4   for(k = 1; k <= n; k++) {
5     F += i;
6     i = F - i;
7   }
8   return F;
9 }
```

- Custo: $O(n)$
- **Complexidade Linear**!

Conclusão

Não se deve utilizar recursividade cegamente!!!

Exemplo: Série de Fibonacci – Solução Recursiva Melhorada

```
1  int fibRM(int n, int a, int b) {
2    if(n == 0)
3      return a;
4    if(n == 1)
5      return b;
6
7    return fibRM(n-1, b, a + b);
8  }
9
10 int fibM(int n) {
11   return fibRM(n, 0, 1);
12 }
```

Quando vale a pena usar recursividade?

- Recursividade vale a pena para algoritmos complexos, cuja implementação iterativa é complexa e normalmente requer o uso explícito de uma pilha.
- Exemplos:
  - Dividir para Conquistar (Ex. Quicksort).
  - Caminhamento em Árvores (pesquisa, backtracking).

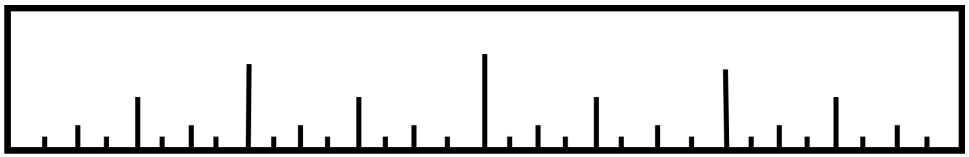
Conteúdo

Definição

- Duas chamadas recursivas.
  - Cada uma resolvendo a metade do problema.
- Muito usado na prática.
  - Solução eficiente de problemas.
  - Decomposição.
- Não se reduz trivialmente como fatorial.
  - Duas chamadas recursivas.
- Não produz recomputação excessiva como fibonacci.
  - Porções diferentes do problema.

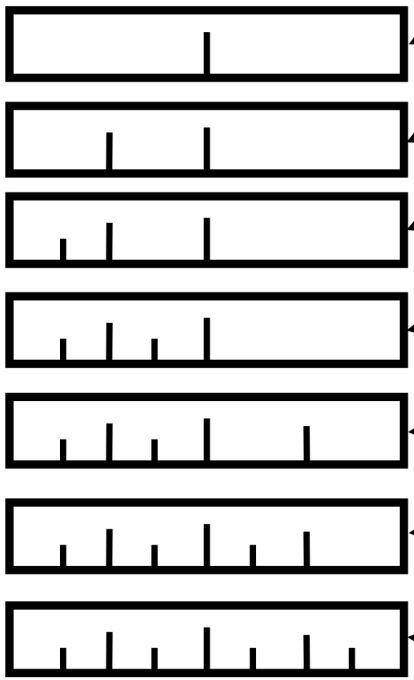
Exemplo: Régua

```
1 int regua(int l, int r, int h) {
2   int m;
3   if(h > 0) {
4     m = (l + r) / 2;
5     marca(m, h);
6     regua(l, m, h - 1);
7     regua(m, r, h - 1);
8   }
9 }
```



Qual é a ordem de impressão das marcas?

Exemplo: Régua – Execução

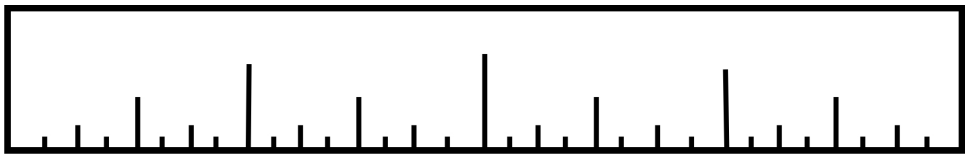


regua(0, 8, 3)
marca(4, 3)
regua(0, 4, 2)
marca(2, 2)
regua(0, 2, 1)
marca(1, 1)
regua(0, 1, 0)
regua(1, 2, 0)
regua(2, 4, 1)
marca(3, 1)
regua(2, 3, 0)
regua(3, 4, 0)
regua(4, 8, 2)
marca(6, 2)
regua(4, 6, 1)
marca(5, 1)
regua(4, 5, 0)
regua(5, 6, 0)
regua(6, 8, 1)
marca(7, 1)
regua(6, 7, 0)
regua(7, 8, 0)

Exemplo: Régua – Outra implementação

```
1  int regua(int l, int r,
2            int h) {
3    int m;
4    if(h > 0) {
5      m = (l + r) / 2;
6      marca(m, h);
7      regua(l, m, h - 1);
8      regua(m, r, h - 1);
9    }
10 }
```

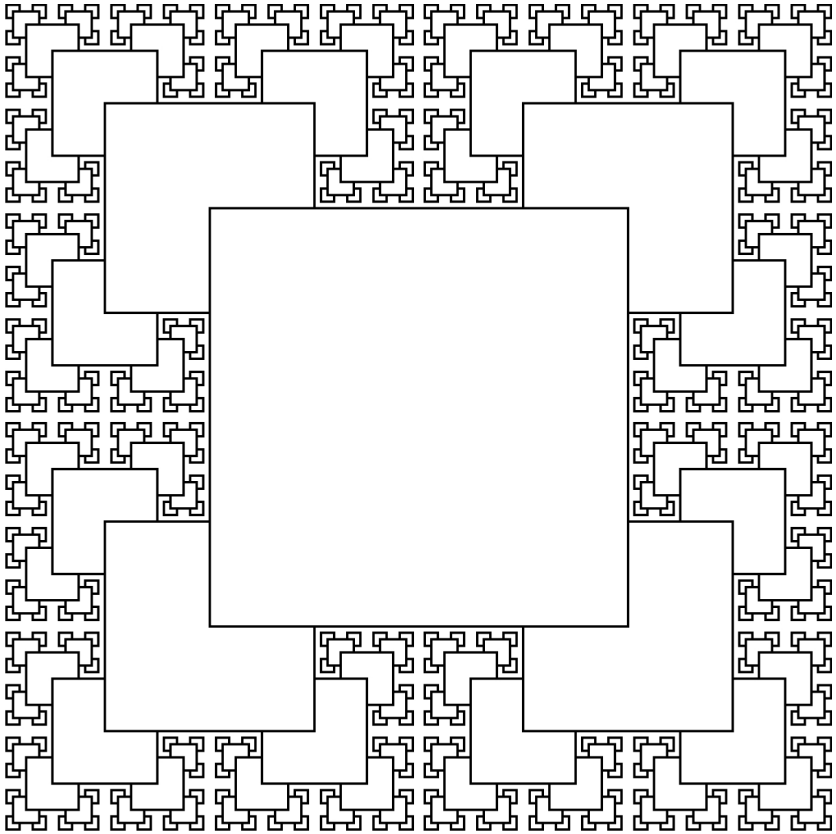
```
1  int regua2(int l, int r,
2             int h) {
3    int m;
4    m = (l + r) / 2;
5    marca(m, h);
6    if(h > 1) {
7      regua2(l, m, h - 1);
8      regua2(m, r, h - 1);
9    }
10 }
```



Exemplo: Estrela

```
1 void estrela1(int x,int y, int r) {
2   if(r > 0) {
3     estrela1(x-r, y+r, r div 2);
4     estrela1(x+r, y+r, r div 2);
5     estrela1(x-r, y-r, r div 2);
6     estrela1(x+r, y-r, r div 2);
7     box(x, y, r);
8   }
9 }
```

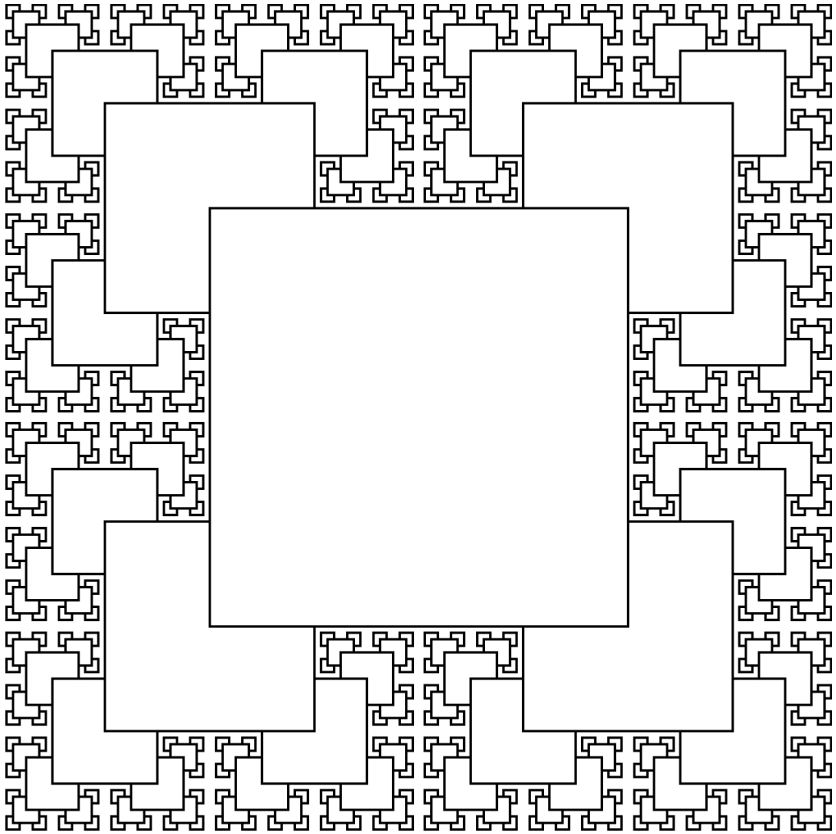
```
1 void estrela2(int x,int y, int r) {
2   if(r > 1) {
3     estrela2(x-r, y+r, r div 2);
4     estrela2(x+r, y+r, r div 2);
5     estrela2(x-r, y-r, r div 2);
6     estrela2(x+r, y-r, r div 2);
7   }
8   box(x, y, r);
9 }
```



Exemplo: Estrela

```
1 void estrela1(int x,int y, int r) {
2   if(r > 0) {
3     estrela1(x-r, y+r, r div 2);
4     estrela1(x+r, y+r, r div 2);
5     estrela1(x-r, y-r, r div 2);
6     estrela1(x+r, y-r, r div 2);
7     box(x, y, r);
8   }
9 }
```

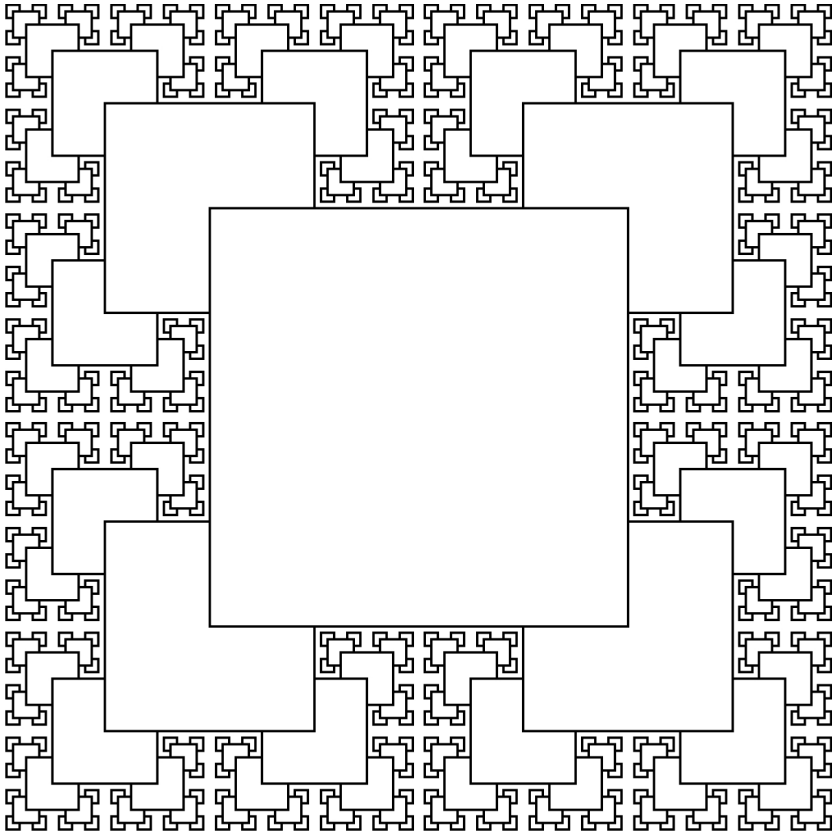
```
1 void estrela2(int x,int y, int r) {
2   if(r > 1) {
3     estrela2(x-r, y+r, r div 2);
4     estrela2(x+r, y+r, r div 2);
5     estrela2(x-r, y-r, r div 2);
6     estrela2(x+r, y-r, r div 2);
7   }
8   box(x, y, r);
9 }
```



Exemplo: Estrela

```
1 void estrela1(int x,int y, int r) {
2   if(r > 0) {
3     estrela1(x-r, y+r, r div 2);
4     estrela1(x+r, y+r, r div 2);
5     estrela1(x-r, y-r, r div 2);
6     estrela1(x+r, y-r, r div 2);
7     box(x, y, r);
8   }
9 }
```

```
1 void estrela2(int x,int y, int r) {
2   if(r > 1) {
3     estrela2(x-r, y+r, r div 2);
4     estrela2(x+r, y+r, r div 2);
5     estrela2(x-r, y-r, r div 2);
6     estrela2(x+r, y-r, r div 2);
7   }
8   box(x, y, r);
9 }
```



Novamente pergunta-se

Qual a melhor implementação? `estrela1` ou `estrela2`?

Conteúdo

Equação de Recorrência

- Define-se uma função de complexidade $f(n)$.
- Identifica-se a equação de recorrência $T(n)$:
  - Especifica-se $T(n)$ como uma função dos termos anteriores.
  - Especifica-se a **condição de parada** (e.g. T(1)).

Exemplo: Função recursiva

```
1 void exemplo(int n) {
2   if(n <= 1)
3     printf("%d", n);
4   else {
5     for(int i = 0;i < n; i++)
6       printf("%d", n);
7     exemplo(n-1);
8   }
9 }
```

Podemos definir a recorrência como:

$$\begin{cases} T(n) = n + T(n-1) \\ T(1) = 1 \end{cases}$$

Exemplo: Função de complexidade

$$\begin{cases} T(n) = n + T(n-1) \\ T(1) = 1 \end{cases}$$

Expandindo:

$$\begin{aligned} T(n) &= n + T(n-1) \\ &= n + (n-1) + T(n-2) \\ &= n + (n-1) + (n-2) + T(n-3) \\ &: \\ &= n + (n-1) + (n-2) + \ldots + 2 + T(1) \\ &= n + (n-1) + (n-2) + \ldots + 2 + 1 \end{aligned}$$

Exemplo: Função de complexidade

$$T(n) = n + (n-1) + (n-2) + \ldots + 2 + 1$$

$$\begin{aligned} 2T(n) &= n + (n-1) + (n-2) + \ldots + \quad 2 \quad + 1 \\ &\quad + 1 + \quad 2 \quad + \quad 3 \quad + \ldots + (n-1) + n \\ &= (n+1) + (n+1) + (n+1) + \ldots + (n+1) \\ &= n\,(n+1) \end{aligned}$$

Logo: $\boldsymbol{T(n) = \frac{n\,(n+1)}{2} = O(n^2)}$

Fatorial

```
1 int fatorial(int n) {
2   if(n == 1)
3     return 1;
4   else{
5     return n * fatorial(n-1);
6   }
7 }
```

Podemos definir a recorrência como:

$$\begin{cases} T(n) = 1 + T(n-1) \\ T(1) = 1 \end{cases}$$

Fatorial: Análise da função

$$\begin{cases} T(n) = 1 + T(n-1) \\ T(1) = 1 \end{cases}$$

Expandindo:

$$\begin{aligned} T(n) &= 1 + T(n-1) \\ &= 1 + 1 + T(n-2) \\ &: \\ &= 1 + 1 + 1 + \ldots + 1 + T(1) \\ &= 1 + 1 + 1 + \ldots + 1 + 1 \\ &= n \end{aligned}$$

Logo: $\boldsymbol{T(n) = n = O(n)}$

Análise de Funções Recursivas

- **Atenção**: lembre-se de que, além da análise de custo de **tempo**, deve-se analisar também o custo de **espaço**.
- Qual a complexidade de espaço da função fatorial (qual o tamanho da pilha de execução)?
  - **Proporcional ao número de chamadas?**

Alguns somatórios úteis

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=0}^{k} 2^k = 2^{k+1} - 1$$

$$\sum_{i=0}^{k} \frac{1}{2^i} = 2 - \frac{1}{2^k}$$

$$\sum_{i=0}^{k} a^i = \frac{a^{k+1} - 1}{a - 1} (a \neq 1)$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

Mais um exemplo: Equação de Recorrência

- Seja a equação de recorrência:

$$\begin{cases} T(n) = n + T(n/3) \\ T(1) = 1 \end{cases}$$

- Resolva por expansão.
- Considere a simplificação de que $\boldsymbol{n}$ seja sempre divisível por 3. Ou seja, $\boldsymbol{n = 3^k}$, $\boldsymbol{k >= 0}$.
- Dica: Somatório de uma PG finita $= \boldsymbol{a_1(1 - q^n)/(1 - q)}$.

Mais um exemplo: Resolvendo a Equação de Recorrência

$$\begin{cases} T(n) = n + T(n/3) \\ T(1) = 1 \end{cases}$$

Resolvendo por expansão:

$$\begin{aligned} T(n) &= n + T(n/3) \\ &= n + n/3 + T(n/3/3) \\ &= n + n/3 + n/3/3 + T(n/3/3/3) \\ &: \\ &= n + n/3 + n/3/3 + \ldots + n/3/3/\ldots/3 + T(n/3/3/\ldots/3) \end{aligned}$$

Mais um exemplo: Resolvendo a Equação de Recorrência

Pela expansão chegamos a:

$$T(n) = n + n/3 + n/3/3 + ... + n/3/3/.../3 + T(n/3/3/.../3)$$

Mas, como $\boldsymbol{n = 3^k}$, então: $\boldsymbol{T(1) = T(n/3^k)}$. Assim, temos:

$$T(n) = \sum_{i=0}^{k-1}(n/3^i) + T(1) = n \sum_{i=0}^{k-1}(1/3^i) + 1$$

Mais um exemplo: Resolvendo a Equação de Recorrência

Até agora temos:

$$T(n) = n \sum_{i=0}^{k-1} (1/3^i) + 1$$

Aplicando o somatório da PG finita, $\boldsymbol{a_1(1-q^n)/(1-q)}$:

$$\begin{aligned} T(n) &= n\,((1-(1/3)^k)/(1-1/3)) + 1 \\ &= n\,(1-1/n)/(1-1/3)) + 1 \\ &= (n-1)/(2/3) + 1 \\ &= 3n/2 - 1/2 \end{aligned}$$

Portanto, $\boldsymbol{T(n) = O(n)}$

Conteúdo

Conclusão

- Nesta aula vimos alguns importantes conceitos de **recursividade** e um poderoso paradigma de programação, denominado "**Dividir para Conquistar**".
- Também aprendemos a calcular a complexidade de funções recursivas através de **equações de recorrência**.
- *Próximas aulas*: Auxílio para Tp1 e Prova 01.
- Dúvidas?

Conteúdo

Exercício 01

- Crie uma função recursiva que calcula a potência de um número.
  - Qual a condição de parada?
  - Qual a complexidade de sua função? Apresente a equação de recorrência e resolva-a.

Exercício 02

- Resolva a sequinte equação de recorrência:

$$\begin{cases} T(n) = T(n/2) + 1 \\ T(1) = 0 \end{cases}$$

Exercício 03

- Descubra o que faz o algoritmo abaixo.

```
1 int algoritmo(int a, int b) {
2   if(b == 0)
3     return a;
4   else
5     return algoritmo(b, a % b);
6 }
```