

Recursão

Estrutura de Dados — QXD0010



UNIVERSIDADE
FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Prof. Atílio Gomes Luiz
gomes.atilio@ufc.br

Universidade Federal do Ceará

2º semestre/2020



Recursão

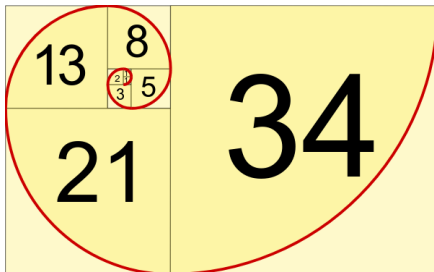
- Uma **entidade** ou **conceito** é dito ser **recursivo** quando é constituída(o) por instâncias semelhantes mais simples ou menores.

Recursão

- Uma **entidade** ou **conceito** é dito ser **recursivo** quando é constituída(o) por instâncias semelhantes mais simples ou menores.
- Existem inúmeros exemplos na **natureza**, nas **artes** e na **matemática** nos quais podemos observar essa propriedade.

Recursão

- Uma **entidade** ou **conceito** é dito ser **recursivo** quando é constituída(o) por instâncias semelhantes mais simples ou menores.
- Existem inúmeros exemplos na **natureza**, nas **artes** e na **matemática** nos quais podemos observar essa propriedade.



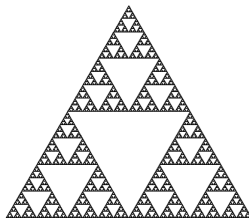
Concha de Nautilus

Recursão



Bonecas russas Matrioska

Recursão

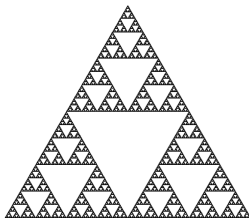


Bonecas russas Matrioska Fractal: Triângulo de Sierpiński

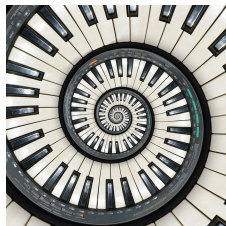
Recursão



Bonecas russas Matrioska

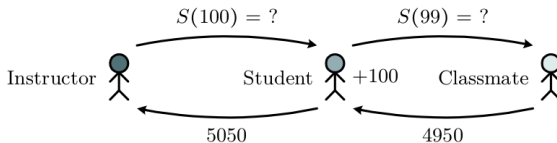


Fractal: Triângulo de Sierpiński



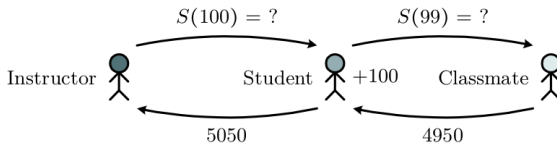
Efeito Droste

Problema: soma dos n primeiros inteiros positivos

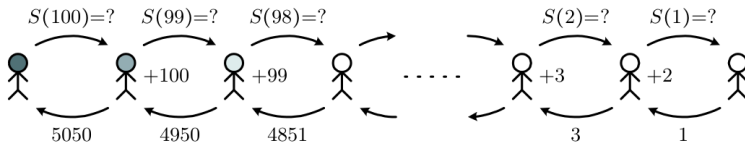


Experimento em uma sala onde o instrutor pede um estudante para adicionar os primeiros 100 inteiros positivos. O valor $S(n)$ representa a soma dos n primeiros inteiros positivos.

Problema: soma dos n primeiros inteiros positivos



Experimento em uma sala onde o instrutor pede um estudante para adicionar os primeiros 100 inteiros positivos. O valor $S(n)$ representa a soma dos n primeiros inteiros positivos.



Recursão

- O conceito de **recursão** é de fundamental importância em **computação**.

Recursão

- O conceito de **recursão** é de fundamental importância em **computação**.
 - É um **princípio poderoso** que permite que **problemas sejam definidos em função de instâncias menores** do mesmo problema.

Recursão

- O conceito de **recursão** é de fundamental importância em **computação**.
 - É um **princípio poderoso** que permite que **problemas sejam definidos em função de instâncias menores** do mesmo problema.
 - Dizemos que esses problemas têm **estrutura recursiva**.

Recursão

- O conceito de **recursão** é de fundamental importância em **computação**.
 - É um **princípio poderoso** que permite que **problemas sejam definidos em função de instâncias menores** do mesmo problema.
 - Dizemos que esses problemas têm **estrutura recursiva**.

Para resolver tal problema é natural aplicar o seguinte método:

Recursão

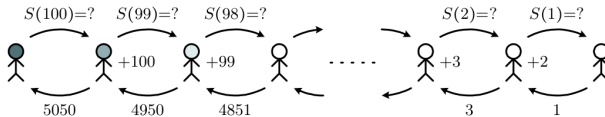
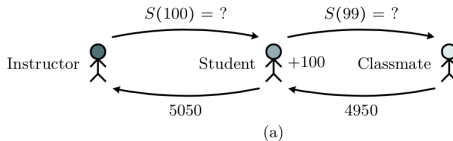
- O conceito de **recursão** é de fundamental importância em **computação**.
 - É um **princípio poderoso** que permite que **problemas sejam definidos em função de instâncias menores** do mesmo problema.
 - Dizemos que esses problemas têm **estrutura recursiva**.

Para resolver tal problema é natural aplicar o seguinte método:

Se a instância em questão é pequena,
 resolva-a diretamente (use força bruta se necessário)
Senão,
 reduza-a a uma instância menor do mesmo problema,
 aplique o método à instância menor
 e volte à instância original.

A aplicação deste método produz um **algoritmo recursivo**.

Problema: soma dos n primeiros inteiros positivos



Vamos produzir um algoritmo recursivo usando o método:

Se a instância em questão é pequena,
resolva-a diretamente (use força bruta se necessário)

Senão,
reduza-a a uma instância menor do mesmo problema,
aplique o método à instância menor
e volte à instância original.

Exercício

- Escreva um algoritmo recursivo que calcule a soma dos n primeiros inteiros positivos, onde o inteiro positivo n é dado como entrada.

Exercício

- Escreva um algoritmo recursivo que calcule a soma dos n primeiros inteiros positivos, onde o inteiro positivo n é dado como entrada.

Exercício

- Escreva um algoritmo recursivo que calcule a soma dos n primeiros inteiros positivos, onde o inteiro positivo n é dado como entrada.

Muitas **funções matemáticas** são definidas recursivamente. A seguir, veremos algumas delas.

Recursão — Fatorial de n

- O fatorial de um inteiro não-negativo n é definido como:

$$n! = (n)(n-1)(n-2) \cdots (1)$$

Convenção: $0! = 1$.

Recursão — Fatorial de n

- O fatorial de um inteiro não-negativo n é definido como:

$$n! = (n)(n-1)(n-2) \cdots (1)$$

Convenção: $0! = 1$.

- Como calcular $n!$ **recursivamente**?

Recursão — Fatorial de n

- O fatorial de um inteiro não-negativo n é definido como:

$$n! = (n)(n-1)(n-2) \cdots (1)$$

Convenção: $0! = 1$.

- Como calcular $n!$ **recursivamente**?
 - Devemos inicialmente identificar um caso base e um caso geral.

Recursão — Fatorial de n

- O fatorial de um inteiro não-negativo n é definido como:

$$n! = (n)(n-1)(n-2) \cdots (1)$$

Convenção: $0! = 1$.

- Como calcular $n!$ **recursivamente**?
 - Devemos inicialmente identificar um caso base e um caso geral.
- Como $(n-1)! = (n-1)(n-2) \cdots 1$, temos que $n! = (n)(n-1)!$

Recursão — Fatorial de n

- O fatorial de um inteiro não-negativo n é definido como:

$$n! = (n)(n-1)(n-2) \cdots (1)$$

Convenção: $0! = 1$.

- Como calcular $n!$ **recursivamente**?
 - Devemos inicialmente identificar um caso base e um caso geral.
- Como $(n-1)! = (n-1)(n-2) \cdots 1$, temos que $n! = (n)(n-1)!$
- Note que a forma de resolver $(n-1)!$ é a mesma para resolver $n!$

Recursão — Fatorial de n

Formalmente, podemos definir a função fatorial de n de forma recursiva como a seguir:

$$fat(n) = \begin{cases} 1, & \text{se } n = 0; \\ n \cdot fat(n - 1), & \text{se } n \geq 1. \end{cases}$$

Recursão — Fatorial de n

Formalmente, podemos definir a função fatorial de n de forma recursiva como a seguir:

$$fat(n) = \begin{cases} 1, & \text{se } n = 0; \\ n \cdot fat(n - 1), & \text{se } n \geq 1. \end{cases}$$

- Esta definição estabelece um processo recursivo para calcular o fatorial de um inteiro não-negativo n .

Recursão — Fatorial de n

Formalmente, podemos definir a função fatorial de n de forma recursiva como a seguir:

$$fat(n) = \begin{cases} 1, & \text{se } n = 0; \\ n \cdot fat(n - 1), & \text{se } n \geq 1. \end{cases}$$

- Esta definição estabelece um processo recursivo para calcular o fatorial de um inteiro não-negativo n .
- **Caso trivial:** $n = 0$. Neste caso, temos $n! = 0! = 1$.

Recursão — Fatorial de n

Formalmente, podemos definir a função fatorial de n de forma recursiva como a seguir:

$$fat(n) = \begin{cases} 1, & \text{se } n = 0; \\ n \cdot fat(n - 1), & \text{se } n \geq 1. \end{cases}$$

- Esta definição estabelece um processo recursivo para calcular o fatorial de um inteiro não-negativo n .
- **Caso trivial:** $n = 0$. Neste caso, temos $n! = 0! = 1$.
- **Caso geral:** para $n \geq 1$, temos que $n! = n * (n - 1)!$

Fatorial de n

```
1 int fat(int n) {
```

Fatorial de n

```
1 int fat(int n) {
```

Fatorial de n

```
1 int fat(int n) {  
2     if (n == 0) /* caso base */  
3         return 1;
```

Fatorial de n

```
1 int fat(int n) {  
2     if (n == 0) /* caso base */  
3         return 1;  
4     else /* caso geral */  
5         return n * fat(n-1); /* instancia menor */  
6 }
```

Elemento máximo de um vetor

- Gostaríamos de determinar o valor de um elemento máximo de um vetor $v[0 \dots n - 1]$.

Elemento máximo de um vetor

- Gostaríamos de determinar o valor de um elemento máximo de um vetor $v[0 \dots n - 1]$.
- Para preparar o terreno, examinemos a seguinte solução iterativa para o problema:

Elemento máximo de um vetor

- Gostaríamos de determinar o valor de um elemento máximo de um vetor $v[0 \dots n - 1]$.
- Para preparar o terreno, examinemos a seguinte solução iterativa para o problema:

```
1 int maximo (int n, int v[]) {  
2     int max = v[0];  
3  
4     for (int i = 1; i < n; i++)  
5         if (max < v[i])  
6             max = v[i];  
7  
8     return max;  
9 }
```

Elemento máximo de um vetor

- Gostaríamos de determinar o valor de um elemento máximo de um vetor $v[0 \dots n - 1]$.
- Para preparar o terreno, examinemos a seguinte solução iterativa para o problema:

```
1 int maximo (int n, int v[]) {  
2     int max = v[0];  
3  
4     for (int i = 1; i < n; i++)  
5         if (max < v[i])  
6             max = v[i];  
7  
8     return max;  
9 }
```

Como seria uma solução recursiva?

Elemento máximo de um vetor

Solução recursiva para o problema:

```
1 // Ao receber v e n >= 1, esta funcao devolve o valor de
2 // um elemento maximo do vetor v[0..n-1]
3 int maximoR (int n, int v[]) {
4     if (n == 1)
5         return v[0];
6     else {
7         int max;
8         max = maximoR (n-1, v); // max eh o maximo de v[0..n-2]
9         if (max > v[n-1])
10             return max;
11         else
12             return v[n-1];
13     }
14 }
```

Elemento máximo de um vetor

Solução recursiva para o problema:

```
1 // Ao receber v e n >= 1, esta funcao devolve o valor de
2 // um elemento maximo do vetor v[0..n-1]
3 int maximoR (int n, int v[]) {
4     if (n == 1)
5         return v[0];
6     else {
7         int max;
8         max = maximoR (n-1, v); // max eh o maximo de v[0..n-2]
9         if (max > v[n-1])
10             return max;
11         else
12             return v[n-1];
13     }
14 }
```

Como convencer-nos e/ou convenceremos outros de que nosso algoritmo recursivo está correto?

Análise de corretude de funções recursivas

- **Passo 1:** Escreva o que a função deve fazer.

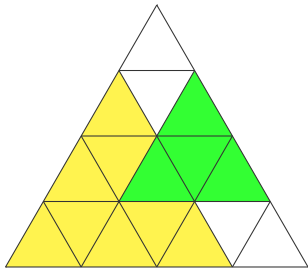
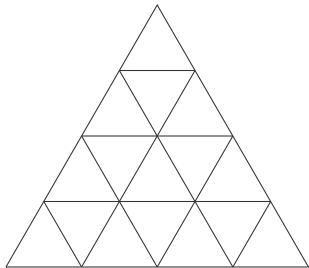
Análise de corretude de funções recursivas

- **Passo 1:** Escreva o que a função deve fazer.
- **Passo 2:** Verifique se a função faz o que deveria quando n é pequeno ($n = 1$ no nosso exemplo do vetor).

Análise de corretude de funções recursivas

- **Passo 1:** Escreva o que a função deve fazer.
- **Passo 2:** Verifique se a função faz o que deveria quando n é pequeno ($n = 1$ no nosso exemplo do vetor).
- **Passo 3:** Imagine que n é grande ($n > 1$ no nosso exemplo) e suponha que a função fará a coisa certa se no lugar de n tivermos algo menor que n .
Sob esta hipótese, verifique que a função faz o que dela se espera.

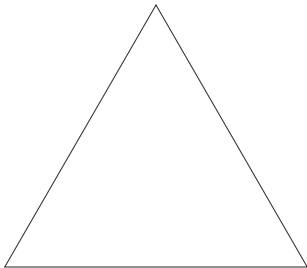
Problema dos Triângulos



Quantos triângulos em pé (tamanhos variados) podemos encontrar em uma grade de triângulos com altura n ?

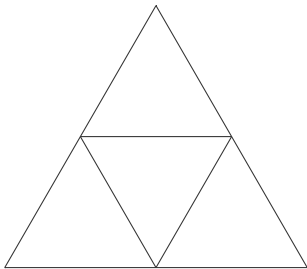
- No exemplo, a grade tem altura 4

Triângulos



Para uma grade de altura $n = 1$, temos $t(1) = 1$ triângulo

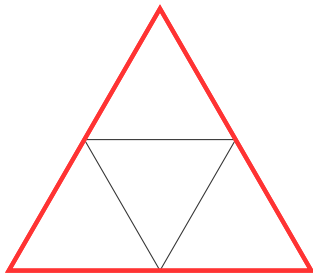
Triângulos



Para altura $n = 2$, temos $t(2) = 4$ triângulos:

- 2 com o vértice superior
- 2 outros triângulos

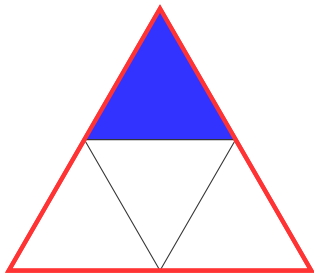
Triângulos



Para altura $n = 2$, temos $t(2) = 4$ triângulos:

- 2 com o vértice superior
- 2 outros triângulos

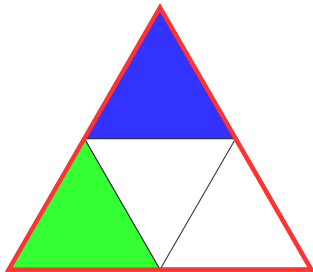
Triângulos



Para altura $n = 2$, temos $t(2) = 4$ triângulos:

- 2 com o vértice superior
- 2 outros triângulos

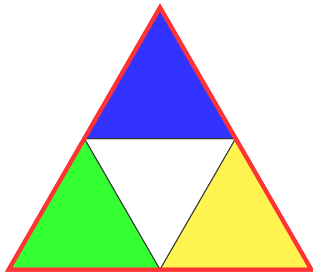
Triângulos



Para altura $n = 2$, temos $t(2) = 4$ triângulos:

- 2 com o vértice superior
- 2 outros triângulos

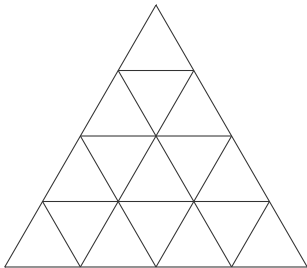
Triângulos



Para altura $n = 2$, temos $t(2) = 4$ triângulos:

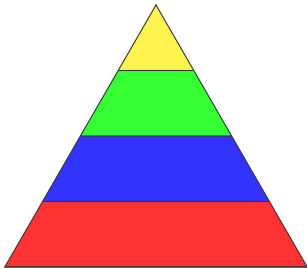
- 2 com o vértice superior
- 2 outros triângulos

Triângulos



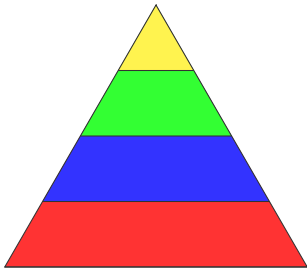
Podemos encontrar algum padrão para $n = 4$?

Triângulos



4 triângulos têm o vértice superior coincidente com o vértice superior do triângulo maior

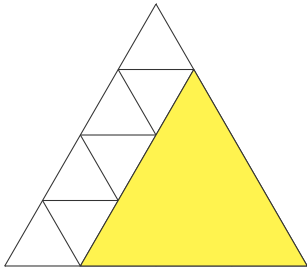
Triângulos



4 triângulos têm o vértice superior coincidente com o vértice superior do triângulo maior

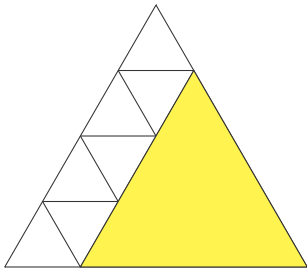
Além desses, quantos outros triângulos faltam?

Triângulos



Faltam os triângulos do lado direito e os do lado esquerdo

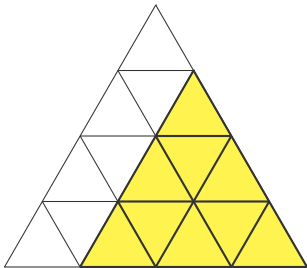
Triângulos



Faltam os triângulos do lado direito e os do lado esquerdo

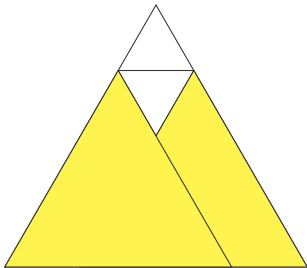
Mas como calcular o número de triângulos de um lado?

Triângulos



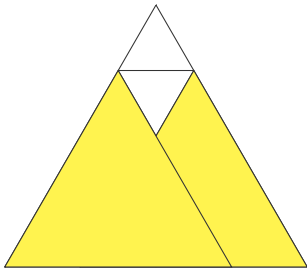
Recaímos no mesmo problema anterior mas agora para $n = 3$

Triângulos



Suponha que já sabemos: $t(1) = 1, t(2) = 4, t(3) = 10$

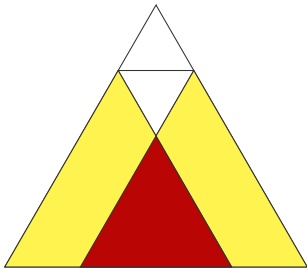
Triângulos



Suponha que já sabemos: $t(1) = 1, t(2) = 4, t(3) = 10$

Calculamos $t(4)$ somando os triângulos superiores aos os triângulos da esquerda e da direita

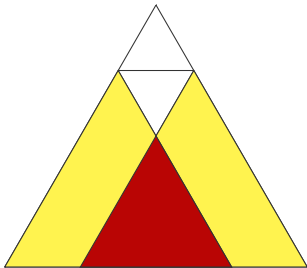
Triângulos



Suponha que já sabemos: $t(1) = 1, t(2) = 4, t(3) = 10$

Calculamos $t(4)$ somando os triângulos superiores aos os triângulos da esquerda e da direita e subtraindo a interseção

Triângulos

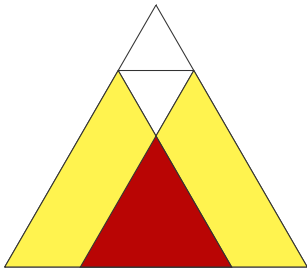


Suponha que já sabemos: $t(1) = 1, t(2) = 4, t(3) = 10$

Calculamos $t(4)$ somando os triângulos superiores aos os triângulos da esquerda e da direita e subtraindo a interseção

$$t(4) =$$

Triângulos

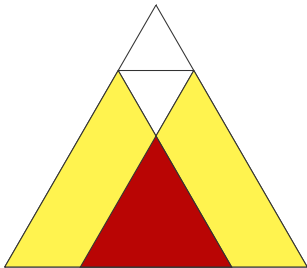


Suponha que já sabemos: $t(1) = 1, t(2) = 4, t(3) = 10$

Calculamos $t(4)$ somando os triângulos superiores aos os triângulos da esquerda e da direita e subtraindo a interseção

$$t(4) = 4 + t(3) + t(3) - t(2)$$

Triângulos

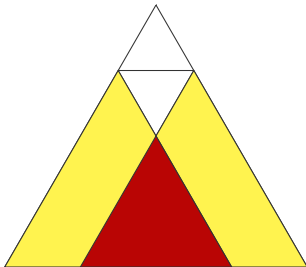


Suponha que já sabemos: $t(1) = 1, t(2) = 4, t(3) = 10$

Calculamos $t(4)$ somando os triângulos superiores aos os triângulos da esquerda e da direita e subtraindo a interseção

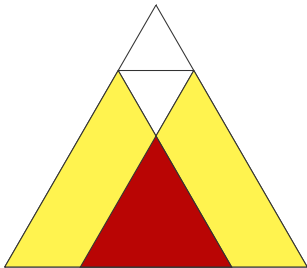
$$t(4) = 4 + t(3) + t(3) - t(2) = 20$$

Triângulos



E para calcular $t(n)$ para um n qualquer?

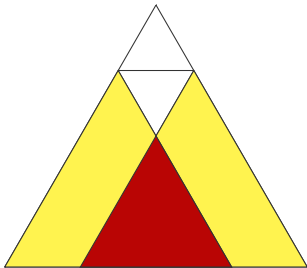
Triângulos



E para calcular $t(n)$ para um n qualquer?

- Se $n = 0$, então $t(n) = 0$

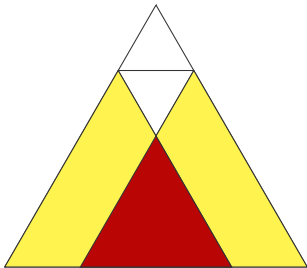
Triângulos



E para calcular $t(n)$ para um n qualquer?

- Se $n = 0$, então $t(n) = 0$
- Se $n = 1$, então $t(n) = 1$

Triângulos



E para calcular $t(n)$ para um n qualquer?

- Se $n = 0$, então $t(n) = 0$
- Se $n = 1$, então $t(n) = 1$
- Do contrário, $t(n) = n + 2 \cdot t(n - 1) - t(n - 2)$

Triângulos - código

Escreva uma função que calcule o número de triângulos em pé de uma grade de tamanho n

Triângulos - código

Escreva uma função que calcule o número de triângulos em pé de uma grade de tamanho n

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

Definições recursivas

Algumas operações matemáticas ou objetos matemáticos têm uma definição recursiva

- Ex: fatorial, sequência de Fibonacci, palíndromos, etc...

Definições recursivas

Algumas operações matemáticas ou objetos matemáticos têm uma definição recursiva

- Ex: fatorial, sequência de Fibonacci, palíndromos, etc...
- ou podem ser vistos do ponto de vista da recursão
 - multiplicação, divisão, exponenciação, etc...

Definições recursivas

Algumas operações matemáticas ou objetos matemáticos têm uma definição recursiva

- Ex: fatorial, sequência de Fibonacci, palíndromos, etc...
- ou podem ser vistos do ponto de vista da recursão
 - multiplicação, divisão, exponenciação, etc...

Isso nos permite projetar algoritmos para lidar com essas operações/objetos

Definições recursivas

Algumas operações matemáticas ou objetos matemáticos têm uma definição recursiva

- Ex: fatorial, sequência de Fibonacci, palíndromos, etc...
- ou podem ser vistos do ponto de vista da recursão
 - multiplicação, divisão, exponenciação, etc...

Isso nos permite projetar algoritmos para lidar com essas operações/objetos

Ex: Exponenciação

Seja a é um número real e b é um número inteiro não-negativo

- Se $b = 0$, então $a^b = 1$
- Se $b > 0$, então $a^b = a \cdot a^{b-1}$

Definições recursivas

Algumas operações matemáticas ou objetos matemáticos têm uma definição recursiva

- Ex: fatorial, sequência de Fibonacci, palíndromos, etc...
- ou podem ser vistos do ponto de vista da recursão
 - multiplicação, divisão, exponenciação, etc...

Isso nos permite projetar algoritmos para lidar com essas operações/objetos

Ex: Exponenciação

Seja a é um número real e b é um número inteiro não-negativo

- Se $b = 0$, então $a^b = 1$
- Se $b > 0$, então $a^b = a \cdot a^{b-1}$

```
1 double potencia(double a, int b) {  
2     if (b == 0)  
3         return 1;  
4     else  
5         return a * potencia(a, b-1);  
6 }
```

Palíndromos

Uma palavra é um **palíndromo** se ela é igual ao seu reverso

Palíndromos

Uma palavra é um **palíndromo** se ela é igual ao seu reverso

- Ex: ana, ovo, osso, radar

Palíndromos

Uma palavra é um **palíndromo** se ela é igual ao seu reverso

- Ex: ana, ovo, osso, radar

Matematicamente, uma palavra é palíndromo se:

Palíndromos

Uma palavra é um **palíndromo** se ela é igual ao seu reverso

- Ex: ana, ovo, osso, radar

Matematicamente, uma palavra é palíndromo se:

- ou tem zero letras (palavra vazia)

Palíndromos

Uma palavra é um **palíndromo** se ela é igual ao seu reverso

- Ex: ana, ovo, osso, radar

Matematicamente, uma palavra é palíndromo se:

- ou tem zero letras (palavra vazia)
- ou tem uma letra

Palíndromos

Uma palavra é um **palíndromo** se ela é igual ao seu reverso

- Ex: ana, ovo, osso, radar

Matematicamente, uma palavra é palíndromo se:

- ou tem zero letras (palavra vazia)
- ou tem uma letra
- ou é da forma *$\alpha p \alpha$* onde

Palíndromos

Uma palavra é um **palíndromo** se ela é igual ao seu reverso

- Ex: ana, ovo, osso, radar

Matematicamente, uma palavra é palíndromo se:

- ou tem zero letras (palavra vazia)
- ou tem uma letra
- ou é da forma $\alpha p \alpha$ onde
 - α é uma letra

Palíndromos

Uma palavra é um **palíndromo** se ela é igual ao seu reverso

- Ex: ana, ovo, osso, radar

Matematicamente, uma palavra é palíndromo se:

- ou tem zero letras (palavra vazia)
- ou tem uma letra
- ou é da forma $\alpha p \alpha$ onde
 - α é uma letra
 - p é um palíndromo

Palíndromos

Uma palavra é um **palíndromo** se ela é igual ao seu reverso

- Ex: ana, ovo, osso, radar

Matematicamente, uma palavra é palíndromo se:

- ou tem zero letras (palavra vazia)
- ou tem uma letra
- ou é da forma $\alpha p \alpha$ onde
 - α é uma letra
 - p é um palíndromo

```
1 bool eh_palindromo(char palavra[], int ini, int fim) {  
2     if (ini >= fim)  
3         return true;
```

Palíndromos

Uma palavra é um **palíndromo** se ela é igual ao seu reverso

- Ex: ana, ovo, osso, radar

Matematicamente, uma palavra é palíndromo se:

- ou tem zero letras (palavra vazia)
- ou tem uma letra
- ou é da forma $\alpha p \alpha$ onde
 - α é uma letra
 - p é um palíndromo

```
1 bool eh_palindromo(char palavra[], int ini, int fim) {  
2     if (ini >= fim)  
3         return true;
```


Palíndromos

Uma palavra é um **palíndromo** se ela é igual ao seu reverso

- Ex: ana, ovo, osso, radar

Matematicamente, uma palavra é palíndromo se:

- ou tem zero letras (palavra vazia)
- ou tem uma letra
- ou é da forma $\alpha p \alpha$ onde
 - α é uma letra
 - p é um palíndromo

```
1 bool eh_palindromo(char palavra[], int ini, int fim) {
2     if (ini >= fim)
3         return true;
4     return (palavra[ini] == palavra[fim]) &&
5         eh_palindromo(palavra, ini+1, fim-1);
6 }
```

Palíndromos

Uma palavra é um **palíndromo** se ela é igual ao seu reverso

- Ex: ana, ovo, osso, radar

Matematicamente, uma palavra é palíndromo se:

- ou tem zero letras (palavra vazia)
- ou tem uma letra
- ou é da forma $\alpha p \alpha$ onde
 - α é uma letra
 - p é um palíndromo

```
1 bool eh_palindromo(char palavra[], int ini, int fim) {
2     if (ini >= fim)
3         return true;
4     return (palavra[ini] == palavra[fim]) &&
5             eh_palindromo(palavra, ini+1, fim-1);
6 }
7
8 eh_palindromo(palavra, 0, strlen(palavra)-1);
```

Exemplo: Inversão dos elementos de um vetor

1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---



9	8	7	6	5	4	3	2	1
---	---	---	---	---	---	---	---	---

Exemplo: Inversão dos elementos de um vetor

1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---



9	8	7	6	5	4	3	2	1
---	---	---	---	---	---	---	---	---

```
1 void inverter (int v[], int l, int r) {
```

Exemplo: Inversão dos elementos de um vetor

1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---



9	8	7	6	5	4	3	2	1
---	---	---	---	---	---	---	---	---

```
1 void inverter (int v[], int l, int r) {  
2     if (l < r) { // caso geral  
3         int aux = v[l];  
4         v[l] = v[r];  
5         v[r] = aux;  
6         inverter(v, l+1, r-1);  
7     }  
8 }
```

Comparando recursão e algoritmos iterativos

Normalmente algoritmos recursivos são:

- mais simples de entender
- menores e mais fáceis de programar
- mais “elegantes”

Comparando recursão e algoritmos iterativos

Normalmente algoritmos recursivos são:

- mais simples de entender
- menores e mais fáceis de programar
- mais “elegantes”

Mas algumas vezes podem ser

- **muito** ineficientes (quando comparados a algoritmos iterativos para o mesmo problema)

Comparando recursão e algoritmos iterativos

Normalmente algoritmos recursivos são:

- mais simples de entender
- menores e mais fáceis de programar
- mais “elegantes”

Mas algumas vezes podem ser

- **muito** ineficientes (quando comparados a algoritmos iterativos para o mesmo problema)

Estratégia ideal:

1. encontrar algoritmo recursivo para o problema
2. reescrevê-lo como um algoritmo iterativo

Sequência de Fibonacci

- Sequência: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

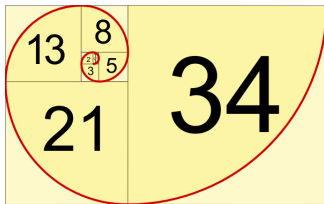
$$F(n) = \begin{cases} n, & \text{se } 0 \leq n \leq 1; \\ F(n-1) + F(n-2), & \text{se } n > 2. \end{cases}$$

Sequência de Fibonacci

- Sequência: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

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

- Curiosidade:** Ao transformar esses números em quadrados e dispô-los de maneira geométrica, é possível traçar uma espiral perfeita, que também aparece em diversos organismos vivos.



Fibonacci: recursivo vs. iterativo

Sequência de Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, ...

Fibonacci: recursivo vs. iterativo

Sequência de Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, ...

```
1 int fib_rec(int n) {  
2     if (n < 2)  
3         return n;  
4     else  
5         return fib_rec(n-2)+  
           fib_rec(n-1);  
6 }
```

Fibonacci: recursivo vs. iterativo

Sequência de Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, ...

```
1 int fib_rec(int n) {  
2     if (n < 2)  
3         return n;  
4     else  
5         return fib_rec(n-2)+  
6             fib_rec(n-1);  
6 }
```

```
1 int fib_iterativo(int n) {  
2     int ant, atual, prox, i;  
3     ant = 0;  
4     atual = 1;  
5     for (i = 2; i <= n; i++) {  
6         prox = ant + atual;  
7         ant = atual;  
8         atual = prox;  
9     }  
10    return (n > 0) ? atual : ant;  
11 }
```

Fibonacci: recursivo vs. iterativo

Sequência de Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, ...

```
1 int fib_rec(int n) {
2     if (n < 2)
3         return n;
4     else
5         return fib_rec(n-2)+
6             fib_rec(n-1);
7 }

1 int fib_iterativo(int n) {
2     int ant, atual, prox, i;
3     ant = 0;
4     atual = 1;
5     for (i = 2; i <= n; i++) {
6         prox = ant + atual;
7         ant = atual;
8         atual = prox;
9     }
10    return (n > 0) ? atual : ant;
11 }
```

Número de operações:

Fibonacci: recursivo vs. iterativo

Sequência de Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, ...

```

1 int fib_rec(int n) {
2     if (n < 2)
3         return n;
4     else
5         return fib_rec(n-2)+
6             fib_rec(n-1);
7 }

1 int fib_iterativo(int n) {
2     int ant, atual, prox, i;
3     ant = 0;
4     atual = 1;
5     for (i = 2; i <= n; i++) {
6         prox = ant + atual;
7         ant = atual;
8         atual = prox;
9     }
10    return (n > 0) ? atual : ant;
11 }

```

Número de operações:

- iterativo: $\approx n$
- recursivo: aproximadamente 1.6^n

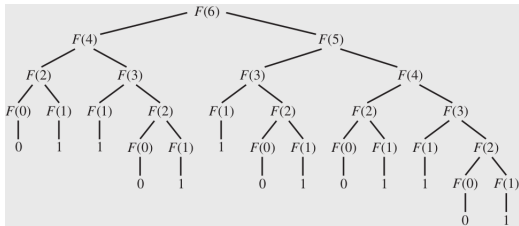
n	10	20	30	40	50	60
1.6^n	110	12089	1329227	146150163	16069380442	$1,766847065 \times 10^{12}$

1 operação $\approx 1/10^6$ segundos

15 dias $\approx 1,29 \cdot 10^6$ segundos $\approx 1,29 \cdot 10^{12}$ operações.

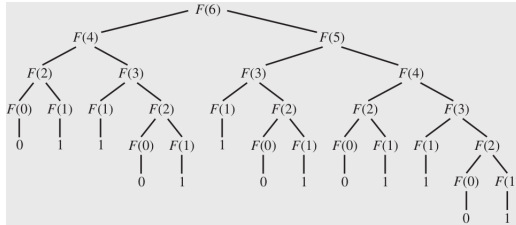
Fibonacci: recursivo vs. iterativo

Árvore de recursão para calcular $F(6)$. Note a quantidade de cálculos repetidos:



Fibonacci: recursivo vs. iterativo

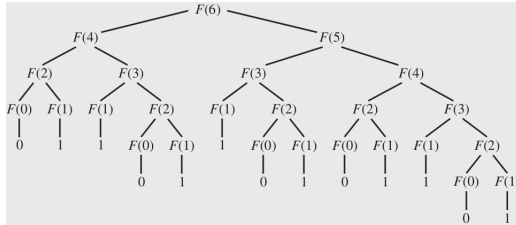
Árvore de recursão para calcular $F(6)$. Note a quantidade de cálculos repetidos:



- O número de chamadas à função recursiva $F(n)$ cresce rapidamente mesmo para número bem pequenos da série.

Fibonacci: recursivo vs. iterativo

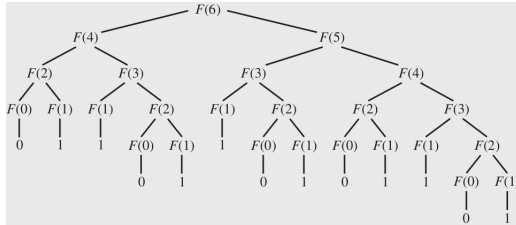
Árvore de recursão para calcular $F(6)$. Note a quantidade de cálculos repetidos:



- O número de chamadas à função recursiva $F(n)$ cresce rapidamente mesmo para número bem pequenos da série.
- Por exemplo, o fibonacci de 30 exige aproximadamente 2.692.537 chamadas à função.

Fibonacci: recursivo vs. iterativo

Árvore de recursão para calcular $F(6)$. Note a quantidade de cálculos repetidos:

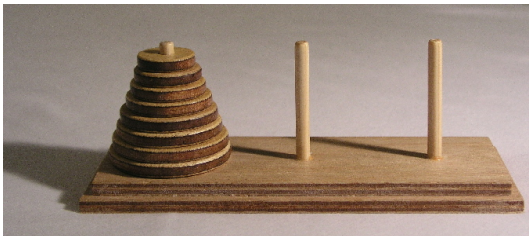


- O número de chamadas à função recursiva $F(n)$ cresce rapidamente mesmo para número bem pequenos da série.
- Por exemplo, o fibonacci de 30 exige aproximadamente 2.692.537 chamadas à função.
- Logo, é interessante evitar programas recursivos no estilo de fibonacci que resultam em uma ‘explosão’ exponencial de chamadas.

Torres de Hanoi

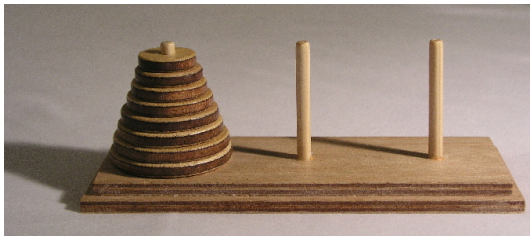


Torres de Hanói



A torre de Hanói é um brinquedo com três estacas *A*, *B* e *C* e discos de tamanhos diferentes.

Torres de Hanói

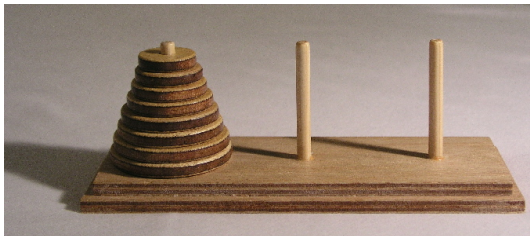


A torre de Hanói é um brinquedo com três estacas *A*, *B* e *C* e discos de tamanhos diferentes.

Objetivo:

- mover todos os discos da estaca *A* para a estaca *C*

Torres de Hanói



A torre de Hanói é um brinquedo com três estacas *A*, *B* e *C* e discos de tamanhos diferentes.

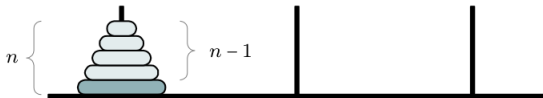
Objetivo:

- mover todos os discos da estaca *A* para a estaca *C*

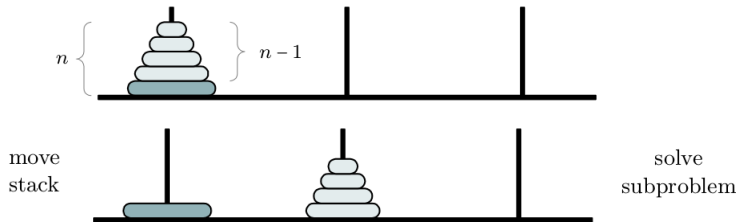
Regras:

- Apenas um disco pode ser movido de cada vez
- Um disco maior não pode ser colocado sobre um menor

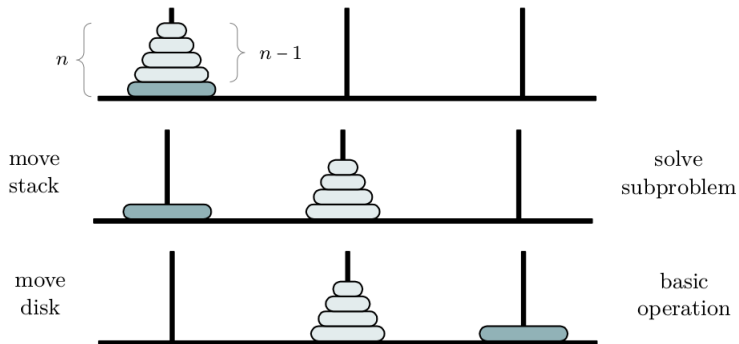
Torres de Hanói



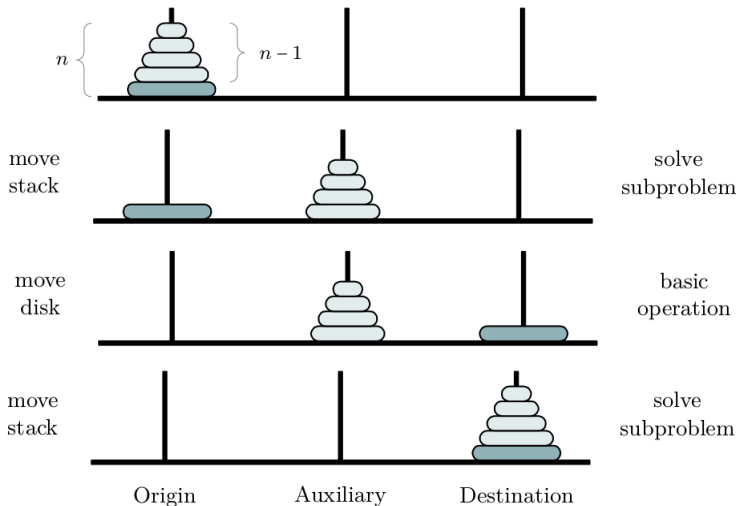
Torres de Hanói



Torres de Hanói



Torres de Hanói



Torres de Hanói

```
1 void hanoi(int n, char orig, char dest, char aux) {  
2     /* caso base: n == 0 - nao faz nada */  
3     if (n > 0) { /* caso geral */  
4         hanoi(n-1, orig, aux, dest);  
5         cout << "move " << orig << "para" << dest << "\n";  
6         hanoi(n-1, aux, dest, orig);  
7     }  
8 }
```

Chamada da função: `hanoi(n, 'a', 'c', 'b');`

Exercícios



Exercício 1 — Soma dos positivos

- Escreva uma função recursiva que calcule a soma dos elementos positivos do vetor de inteiros $A[0..n - 1]$.
- O problema faz sentido quando $n = 0$? Quanto deve valer a soma neste caso?

Exercício 2 — Soma de dígitos

- Escreva uma função recursiva que calcule a soma dos dígitos decimais de um inteiro positivo.
- Por exemplo, a soma dos dígitos de 132 é 6.

Exercício 3 - Max-Min

- Escreva uma função recursiva que calcule a diferença entre o valor de um elemento máximo e o valor de um elemento mínimo de um vetor A com $n \geq 1$ elementos.

Exercício 4 - Triângulo das Somas

Dado um vetor de inteiros A , imprima um triângulo de números tal que:

- na base do triângulo estejam todos os elementos do vetor original;
- o número de elementos em cada nível acima da base é um a menos que no nível inferior;
- e cada elemento no i -ésimo nível é a soma de dois elementos consecutivos do nível inferior.

Exemplo:

Input: $A = \{1, 2, 3, 4, 5\}$

Output:

48

20, 28

8, 12, 16

3, 5, 7, 9

1, 2, 3, 4, 5

Exercício 5 - Coeficientes Binomiais

- O **coeficiente binomial** é uma relação estabelecida entre dois números naturais n e k , $n \geq k \geq 0$, indicada por:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

- Escreva uma função recursiva que calcule o coeficiente binomial de dois números inteiros não negativos n e k , $n \geq k$.
- Dica:** Use a **relação de Stifel**:

$$\binom{n-1}{k-1} + \binom{n-1}{k} = \binom{n}{k}$$

Exercício 6 - Algoritmo de Euclides

A seguinte função, conhecida como algoritmo de Euclides, calcula o maior divisor comum dos inteiros positivos m e n .

```
euclides(int m, int n){  
    int r;  
    do {  
        r = m % n;  
        m = n;  
        n = r;  
    } while (r != 0);  
    return m;  
}
```

Escreva uma função recursiva equivalente.

Exercício 7 - Calculando $\lfloor \log_2 n \rfloor$

O **piso** de um número x é o único inteiro i tal que $i \leq x < i + 1$. O piso de x é denotado por $\lfloor x \rfloor$.

Escreva uma função recursiva que receba um inteiro positivo n e calcule $\lfloor \log_2 n \rfloor$, ou seja, o piso do logaritmo de n na base 2.

Segue uma amostra de valores:

n	15	16	31	32	63	64	127	128	255	256
$\lfloor \log_2 n \rfloor$	3	4	4	5	5	6	6	7	7	8

FIM

