Funções Recursivas

Estrutura de Dados — QXD0010

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

Universidade Federal do Ceará
Campus Quixadá

2º semestre/2025

O que é recursão?

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Definição

- **Recursão** é um conceito em programação e matemática no qual uma função (ou definição) faz referência a si mesma para resolver um problema.

O que é recursão?

Definição

- **Recursão** é um conceito em programação e matemática no qual uma função (ou definição) faz referência a si mesma para resolver um problema.

- 👉 Em termos simples: uma **função recursiva** é uma função que se chama dentro de si mesma, geralmente com uma entrada de tamanho menor ou mais simples, até alcançar uma condição de parada (ou caso base), que interrompe o processo.

Por que estudar recursão?

- Uma maneira diferente de pensar sobre problemas
- Técnica de programação poderosa
- Leva a um código curto, elegante e simples (quando bem usado)
- Muitas linguagens de programação (linguagens "funcionais", como Scheme, Clojure e Haskell) usam exclusivamente recursão (não existem laços nessas linguagens)
- Muitas estruturas têm natureza recursiva:
  - Estruturas de dados lineares
  - Funções matemáticas: fatorial, máximo divisor comum, etc.
  - Sistema de arquivos do computador

Exercício 1: Situação hipotética

Suponha que você esteja esperando em uma fila muito longa. Você quer determinar quantas pessoas estão na sua frente, mas você não pode ver e você não tem permissão para se mover.

Cada pessoa só tem permissão para falar com a pessoa que está à sua frente e atrás de si.

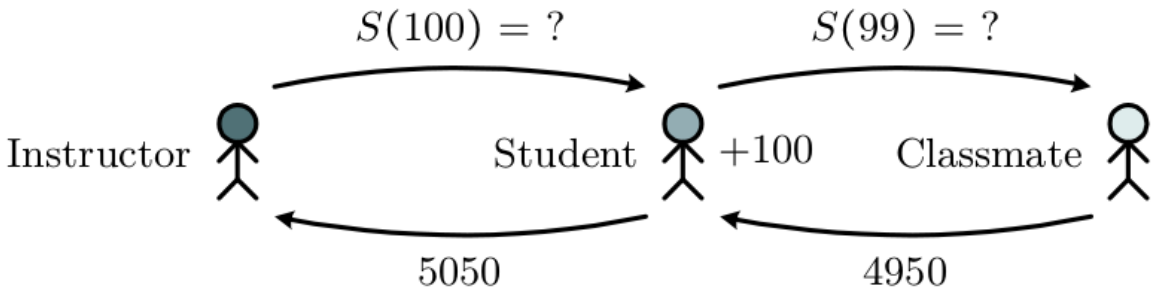
Como determinar quantas pessoas estão na sua frente?

Solução

Algoritmo recursivo para saber o número de pessoas na minha frente em uma fila:

- Se houver uma pessoa na minha frente, então pergunto a ela quantas pessoas estão na frente dela e espero ela responder.
  - Quando ela responder com o valor N, então respondo $N + 1$
- Se não houver ninguém na minha frente, respondo 0.

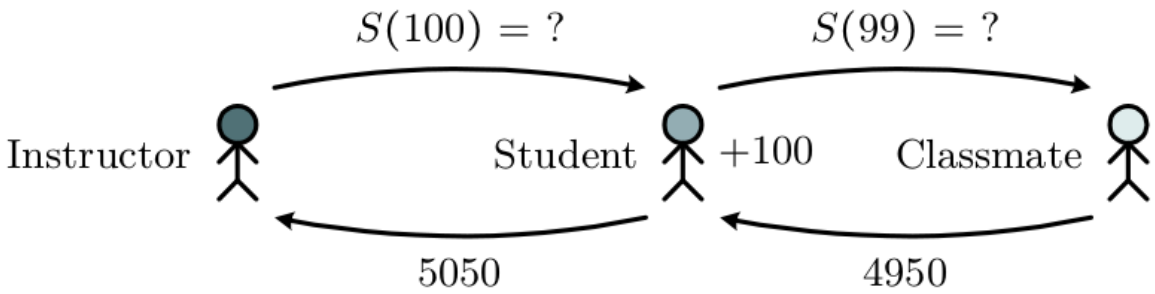
Problema: soma dos n primeiros inteiros positivos



Experimento em uma sala de aula onde o professor pede a um estudante para calcular a soma dos 100 primeiros inteiros positivos.

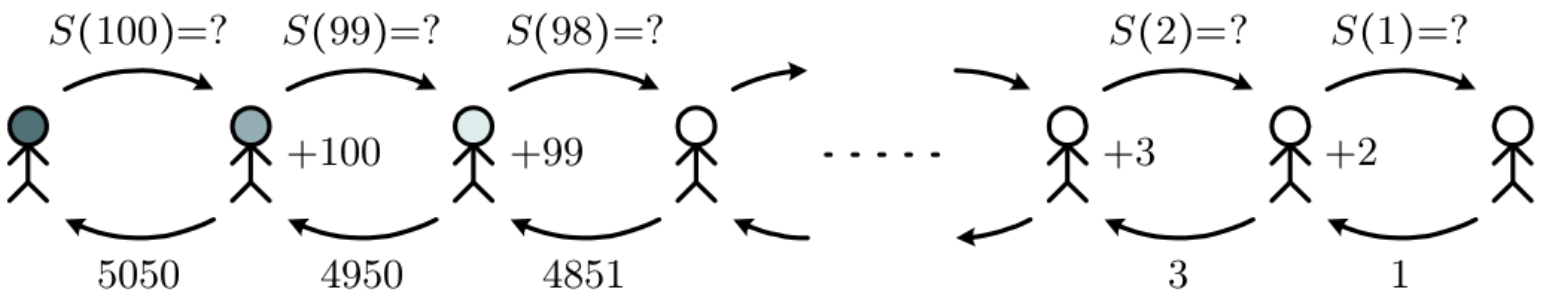
A **função** $S(n)$ representa a soma dos n primeiros inteiros positivos.

Problema: soma dos n primeiros inteiros positivos



Experimento em uma sala de aula onde o professor pede a um estudante para calcular a soma dos 100 primeiros inteiros positivos.

A **função** $S(n)$ representa a soma dos n primeiros inteiros positivos.



Recursão

- Dá para ver que a soma dos n primeiros inteiros positivos pode ser definida de maneira recursiva, conforme a função abaixo:

$$S(n) = \begin{cases} 1 & \text{se } n = 1; \\ n + S(n-1) & \text{se } n \geq 2. \end{cases}$$

- Problemas ou funções que podem ser definidos desta forma possuem uma estrutura recursiva.

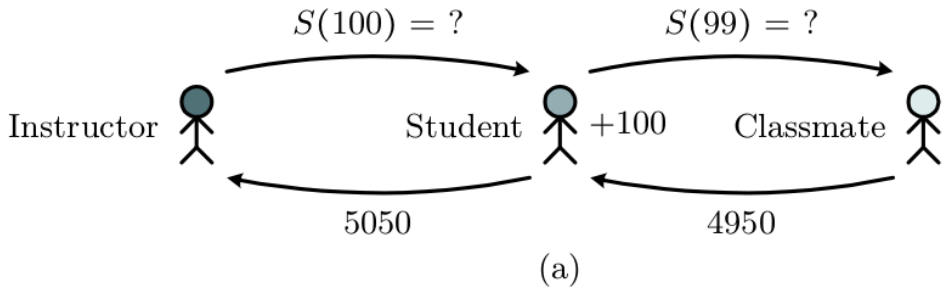
Algoritmo recursivo

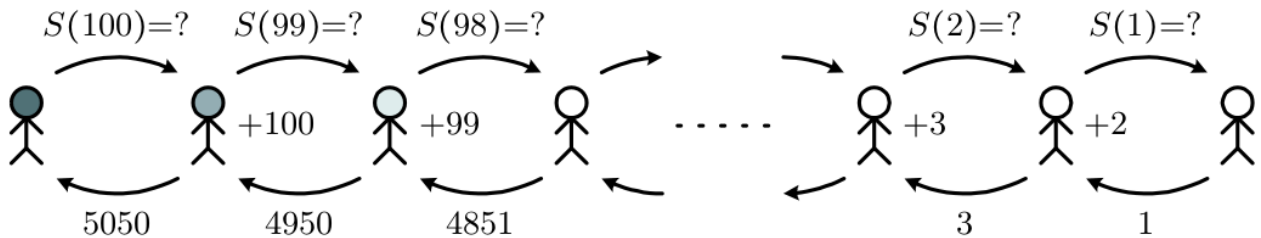
A fim de resolver um problema que tem estrutura recursiva é natural aplicar o seguinte método:

```
Se a instância em questão for 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.
```

Casos da Recursão

Todo algoritmo recursivo envolve pelo menos 2 casos:

- **Caso Base**: resolve instâncias pequenas diretamente.

- **Caso Geral**: resolve instâncias de tamanho grande de forma recursiva.
  - reduzimos o problema à instâncias menores do mesmo problema.
  - chama a função recursivamente para essas instâncias menores.
  - combinamos o resultado da computação dessas funções a fim de obter o resultado da instância original.

Casos da Recursão

Todo algoritmo recursivo envolve pelo menos 2 casos:

- **Caso Base**: resolve instâncias pequenas diretamente.

- **Caso Geral**: resolve instâncias de tamanho grande de forma recursiva.
  - reduzimos o problema à instâncias menores do mesmo problema.
  - chama a função recursivamente para essas instâncias menores.
  - combinamos o resultado da computação dessas funções a fim de obter o resultado da instância original.

Alguns algoritmos recursivos têm mais do que um caso base ou caso geral, mas todos eles têm pelo menos um de cada.

Uma parte crucial da programação recursiva é identificar estes 2 casos.

Exercício 2

- Escreva, em C, uma função recursiva que calcule a soma dos n primeiros inteiros positivos, onde o inteiro positivo n é dado como entrada.
  - Não esqueça: para ser recursiva, a função deve chamar a si mesma pelo menos uma vez.

- Dica: é sempre bom documentar a sua função recursiva antes de começar a escrevê-la.
  - Documentar é escrever um comentário acima da definição da função especificando qual é a **entrada** esperada pela função e qual é a **saída** prometida. O que a função de fato faz dada uma entrada de um certo tamanho.

Exercício 3

O algoritmo abaixo imprime n asteriscos em uma linha.

```
1 // Imprime n asteriscos em uma linha
2 // Pre-condicao: n >= 0
3 void print_stars(int n) {
4     for(int i = 0; i < n; ++i) {
5         cout << "*";
6     }
7     cout << endl;
8 }
```

- Escreva uma versão recursiva desta função.
  - Resolva o problema sem usar laços.
  - **Dica:** sua solução deveria imprimir somente um asterisco por vez.

Um caso base para o exercício 3

- Quais os casos a considerar?
  - Qual o menor número de asteriscos que sei imprimir sem necessidade de laços?

```
1  // Imprime n asteriscos em uma linha
2  // Pre-condicao: n >= 0
3  void print_stars(int n) {
4      if(n == 0) {
5          cout << '\n';
6      }
7      else if(n == 1) {
8          cout << "*";
9      }
10     else {
11         ...
12     }
13 }
```

Lidando com mais casos (sem usar laços)

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  // Imprime n asteriscos em uma linha
2  // Pre-condicao: n >= 0
3  void print_stars(int n) {
4      if(n == 0) {
5          cout << '\n';
6      } else if(n == 1) {
7          cout << '*' << '\n';
8      } else if(n == 2) {
9          cout << '*';
10         cout << '*' << '\n';
11     } else if(n == 3) {
12         cout << '*';
13         cout << '*';
14         cout << '*' << '\n';
15     } else ...
16 }
```

Aproveitando o padrão de repetição

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  // Imprime n asteriscos em uma linha
2  // Pre-condicao: n >= 0
3  void print_stars(int n) {
4      if(n == 0) {
5          cout << '\n';
6      } else if(n == 1) {
7          cout << "*" << '\n';
8      } else if(n == 2) {
9          cout << "*";
10         print_stars(1);     // imprime "*"
11     } else if(n == 3) {
12         cout << "*";
13         print_stars(2);     // imprime "**"
14     } else if(n == 4) {
15         cout << "*";
16         print_stars(3);     // imprime "***"
17     } else ...
18 }
```

Usando recursão apropriadamente

(Versão Final)

```
1  // Imprime n asteriscos em uma linha
2  // Pre-condicao: n >= 0
3  void print_stars(int n) {
4      if(n == 0) {
5          cout << '\n';
6      } else {
7          cout << "*";
8          print_stars(n-1);
9      }
10 }
```

Palíndromos

O que é um palíndromo?

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Definição

Um **palíndromo** é uma palavra, frase, número ou sequência que pode ser lida da mesma forma da esquerda para a direita e da direita para a esquerda, desconsiderando espaços, acentos e pontuação.

Palíndromo — Definição matemática

Definição matemática

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

Palíndromo — Definição matemática

Definição matemática

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

- **Exercício:** Escreva um algoritmo recursivo que determina se uma string é um palíndromo.

Palíndromos


```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdbool.h>
4
5  /**
6   * A função ehPalindromo retorna true se o parâmetro
7   * palavra for um palíndromo; false, caso contrário.
8   * A palavra é uma string que começa no índice ini
9   * e termina no índice fim.
10  */
11 bool ehPalindromo(char palavra[], int ini, int fim) {
12     if(ini >= fim)
13         return true;
14     return (palavra[ini] == palavra[fim]) &&
15             ehPalindromo(palavra, ini+1, fim-1);
16 }
```

Palíndromos

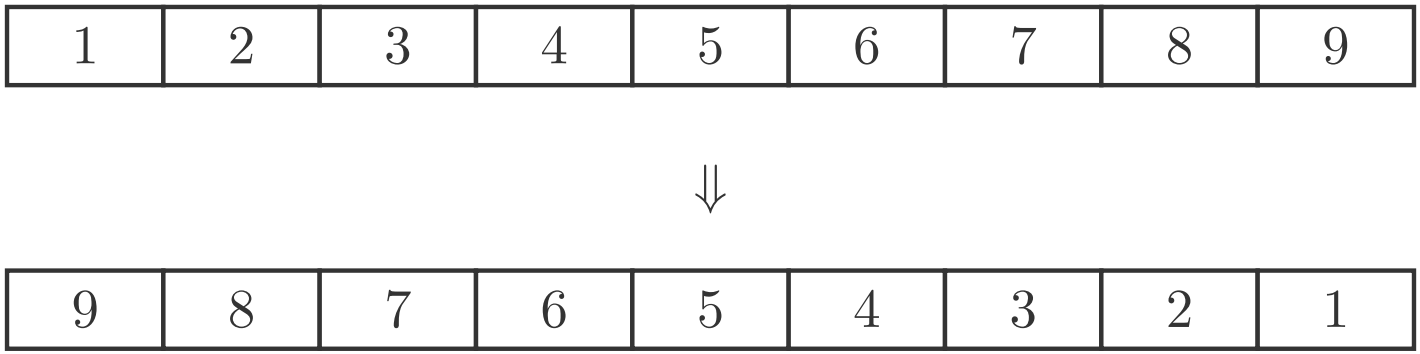

```c
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

/**
 * A função ehPalindromo retorna true se o parâmetro
 * palavra for um palíndromo; false, caso contrário.
 * A palavra é uma string que começa no índice ini
 * e termina no índice fim.
 */
bool ehPalindromo(char palavra[], int ini, int fim) {
    if(ini >= fim)
        return true;
    return (palavra[ini] == palavra[fim]) &&
            ehPalindromo(palavra, ini+1, fim-1);
}

int main() {
    char str[] = "sopapos";
    printf("%d\n", ehPalindromo(str, 0, strlen(str)-1));
    return 0;
}
```

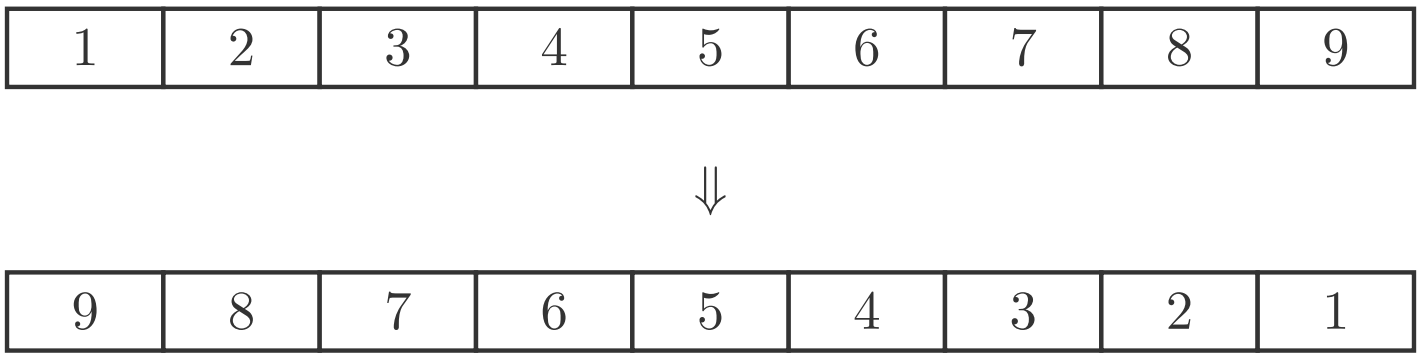
Inversão de vetor

Exemplo: Inversão dos elementos de um vetor

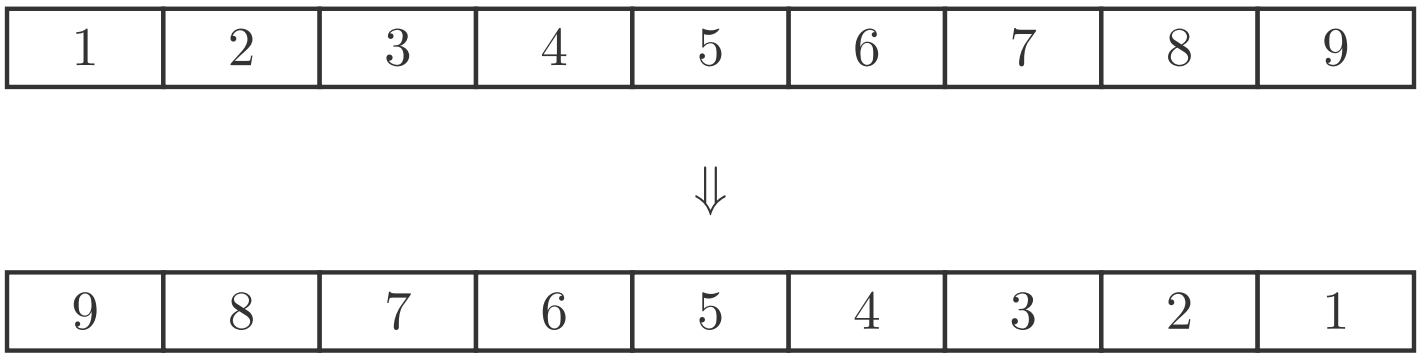


- **Problema:** Inverter os elementos de um vetor usando uma função recursiva.

Solução

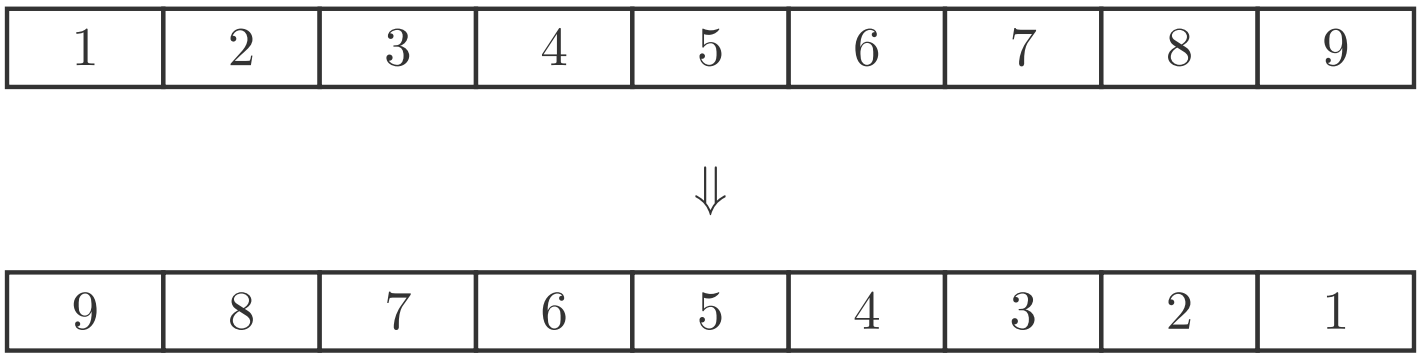


Solução




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

Solução




```
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 }
```

Definições Recursivas

Definições recursivas

Universidade Federal do Ceará
Campus Quixadá

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 }
```

Como provar que meu algoritmo recursivo está correto?

Algoritmos Recursivos

Para produzir um algoritmo recursivo, usamos o seguinte método:

```
Se a instância em questão for 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.
```

Provando que função recursiva está correta

- **Passo 1:** Escreva o que a função deve fazer.
  - Determine os parâmetros da função e o significado de cada um.
  - Sua função deve resolver um problema de tamanho n.
  - Dada a entrada, deixe claro quem é a saída.

Provando que função recursiva está correta

- **Passo 1:** Escreva o que a função deve fazer.
  - Determine os parâmetros da função e o significado de cada um.
  - Sua função deve resolver um problema de tamanho n.
  - Dada a entrada, deixe claro quem é a saída.

- **Passo 2:** Verifique se a função faz o que deveria quando a entrada tem um tamanho n pequeno (Caso Base).

Provando que função recursiva está correta

- **Passo 1:** Escreva o que a função deve fazer.
  - Determine os parâmetros da função e o significado de cada um.
  - Sua função deve resolver um problema de tamanho n.
  - Dada a entrada, deixe claro quem é a saída.

- **Passo 2:** Verifique se a função faz o que deveria quando a entrada tem um tamanho n pequeno (Caso Base).

- **Passo 3:** (Caso Geral) Imagine que a entrada tem um tamanho n grande e suponha que a função fará o que dela se espera se no lugar de n tivermos um problema de tamanho menor que n. Sob esta hipótese, verifique que a função faz o que dela se espera.

Se as verificações das etapas 2 e 3 retornarem respostas positivas, então sua função recursiva está correta.

Elemento máximo em vetor

Elemento máximo de um vetor

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

Elemento máximo de um vetor

- Gostaríamos de determinar o valor de um elemento máximo de um vetor $A[0 \ldots n-1]$.
- Como projetar uma solução recursiva?
- Quem é o caso de parada? Quem é o caso geral?

Elemento máximo de um vetor

Uma possível 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 }
```

Busca em vetor ordenado (Busca Binária)

Busca em vetor ordenado

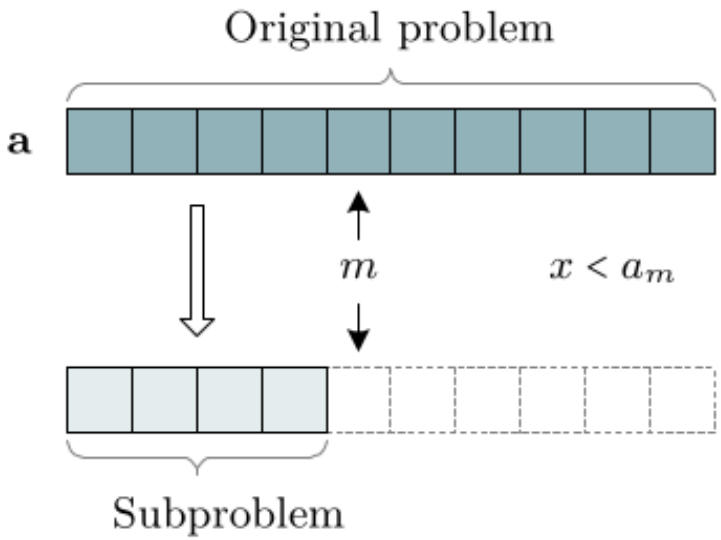
Construa um algoritmo recursivo que resolva o seguinte problema:

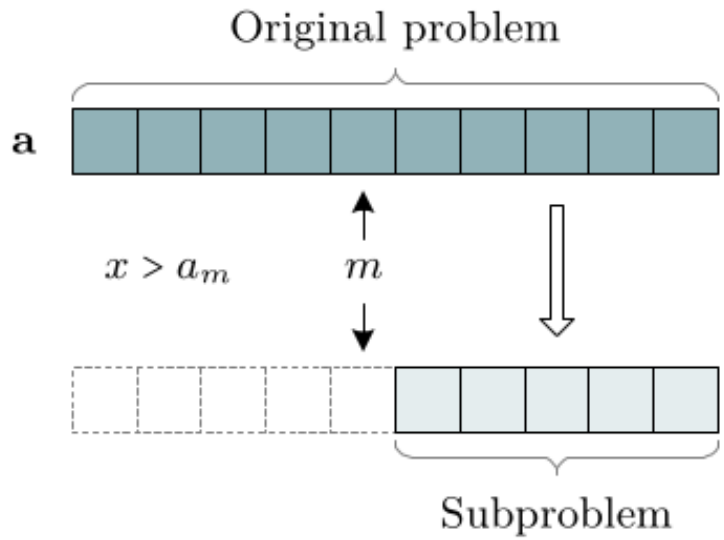
- Temos um vetor com n inteiros ordenados em ordem crescente e gostaríamos de determinar se um dado valor está ou não no vetor.
  - Se o valor estiver no vetor, deve ser retornado o índice da sua posição no vetor.
  - Se o valor não estiver no vetor, deve ser retornado -1.

1	3	4	4	5	8	10	13	19

Decomposição do problema

Busca em vetor ordenado





Algoritmo: Busca Binária

Busca em vetor ordenado


```
1  /**
2   * Se key estiver no array A entao retorna o seu indice.
3   * Caso contrario, retorna -1
4   */
5  int busca_binaria(int A[], int key, int lower, int upper) {
6      if(lower > upper)
7          return -1;
8      int m = (lower + upper) / 2;
9      if(key == A[m])
10         return m;
11     if(key < A[m])
12         return busca_binaria(A, key, lower, m-1);
13     else
14         return busca_binaria(A, key, m+1, upper);
15 }
```

Recursão versus Iteração

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, ...

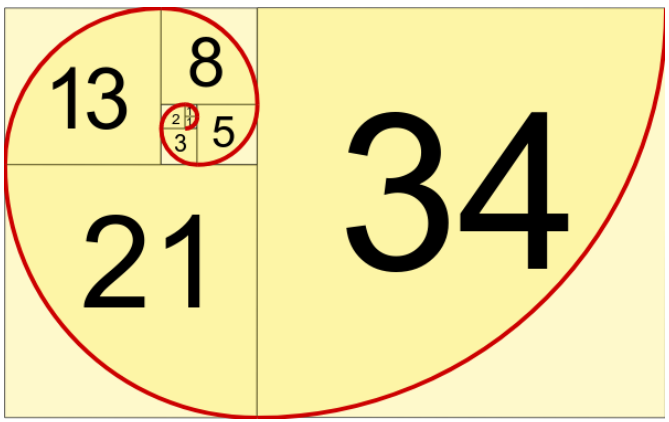
$$F(n) = \begin{cases} n, & \text{se } 0 \leq n \leq 1; \\ F(n-1) + F(n-2), & \text{se } n \geq 2. \end{cases}$$

Sequência de Fibonacci

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

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

- 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.



- Exercício: Escreva uma versão iterativa da função de Fibonacci.

Fibonacci: recursivo vs. iterativo

Sequência de Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, $\ldots$

Fibonacci: recursivo vs. iterativo

Sequência de Fibonacci: $0, 1, 1, 2, 3, 5, 8, 13, \ldots$

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
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: $0, 1, 1, 2, 3, 5, 8, 13, \ldots$

```c
int fib_rec(int n) {
  if (n < 2)
    return n;
  else
    return fib_rec(n-2)+
    fib_rec(n-1);
}
```

```c
int fib_iterativo(int n) {
  int ant, atual, prox, i;
  ant = 0;
  atual = 1;
  for (i = 2; i <= n; i++) {
    prox = ant + atual;
    ant = atual;
    atual = prox;
  }
  return (n > 0) ? atual : ant;
}
```

Fibonacci: recursivo vs. iterativo

Sequência de Fibonacci: $0, 1, 1, 2, 3, 5, 8, 13, \ldots$

```c
int fib_rec(int n) {
  if (n < 2)
    return n;
  else
    return fib_rec(n-2)+
    fib_rec(n-1);
}
```

```c
int fib_iterativo(int n) {
  int ant, atual, prox, i;
  ant = 0;
  atual = 1;
  for (i = 2; i <= n; i++) {
    prox = ant + atual;
    ant = atual;
    atual = prox;
  }
  return (n > 0) ? atual : ant;
}
```

Número de operações:

Fibonacci: recursivo vs. iterativo

Sequência de Fibonacci: $0, 1, 1, 2, 3, 5, 8, 13, \ldots$

```c
int fib_rec(int n) {
  if (n < 2)
    return n;
  else
    return fib_rec(n-2)+
    fib_rec(n-1);
}
```

```c
int fib_iterativo(int n) {
  int ant, atual, prox, i;
  ant = 0;
  atual = 1;
  for (i = 2; i <= n; i++) {
    prox = ant + atual;
    ant = atual;
    atual = prox;
  }
  return (n > 0) ? atual : ant;
}
```

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

Fibonacci: recursivo vs. iterativo

Sequência de Fibonacci: $0, 1, 1, 2, 3, 5, 8, 13, \ldots$

```c
int fib_rec(int n) {
  if (n < 2)
    return n;
  else
    return fib_rec(n-2)+
    fib_rec(n-1);
}
```

```c
int fib_iterativo(int n) {
  int ant, atual, prox, i;
  ant = 0;
  atual = 1;
  for (i = 2; i <= n; i++) {
    prox = ant + atual;
    ant = atual;
    atual = prox;
  }
  return (n > 0) ? atual : ant;
}
```

Número de operações:

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

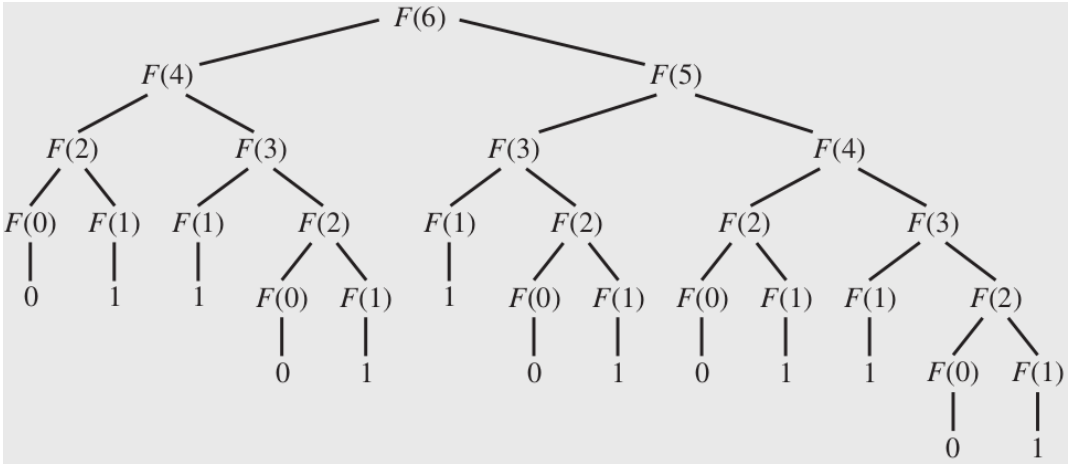
n	10	20	30	40	50	60
1.6^n	110	12089	1329227	146150163	16069380442	$1{,}766847065 \times 10^{12}$

1 operação $\approx 1/10^6$ segundos

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

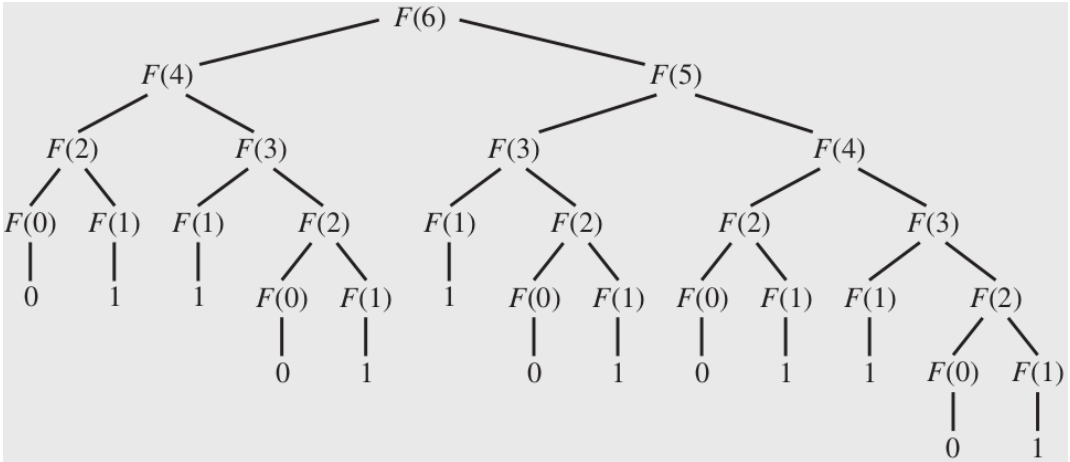
Fibonacci: recursivo vs. iterativo

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:

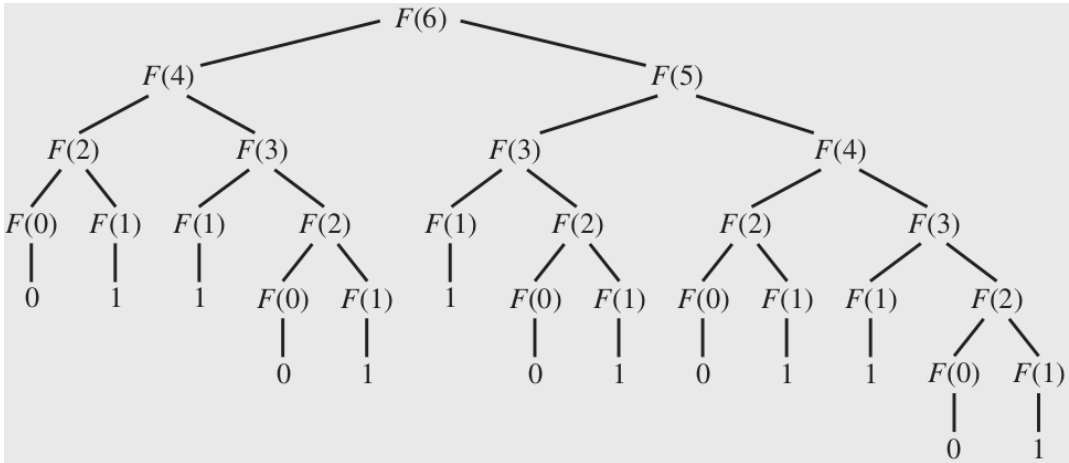


- O número de chamadas à função recursiva `F(n)` cresce rapidamente mesmo para número bem pequenos da série.

Fibonacci: recursivo vs. iterativo

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

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

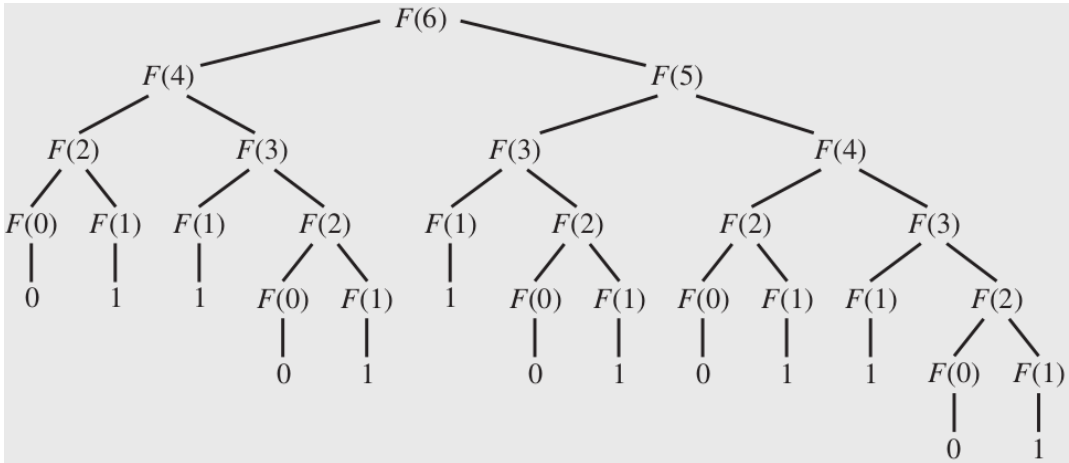


- O número de chamadas à função recursiva `F(n)` cresce rapidamente mesmo para número bem pequenos da série.
- Por exemplo, o fibonacci de 30 exige aproximadamente $2.692.537$ chamadas à função.

Fibonacci: recursivo vs. iterativo

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

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.

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.

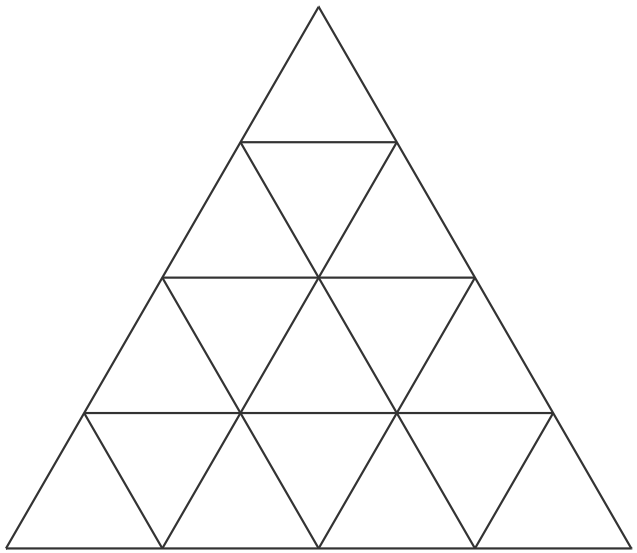
Segue uma amostra de valores:

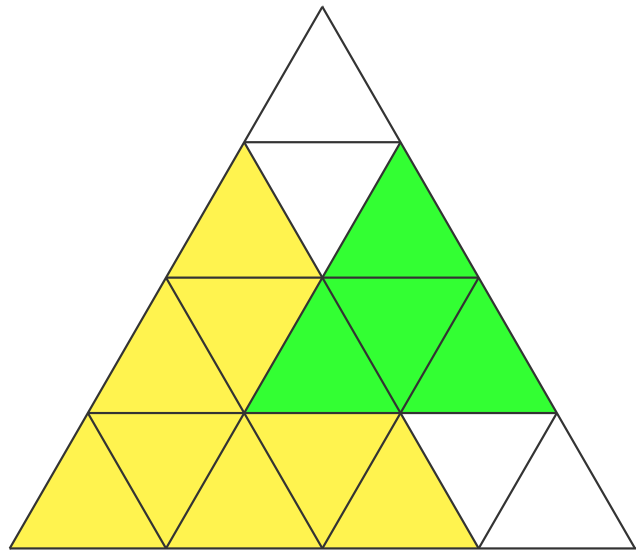
n	15	16	31	32	63	64	127	128	255	256
$\lfloor \log_2 n \rfloor$	3	4	4	5	5	6	6	7	7	8

Apêndice

Problema dos Triângulos

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ



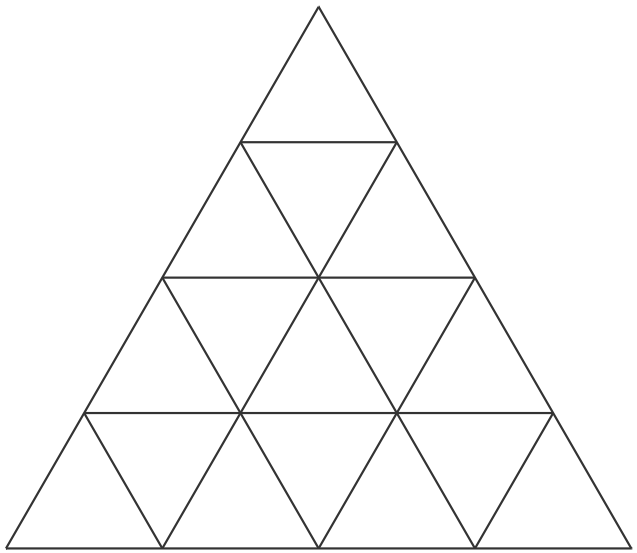


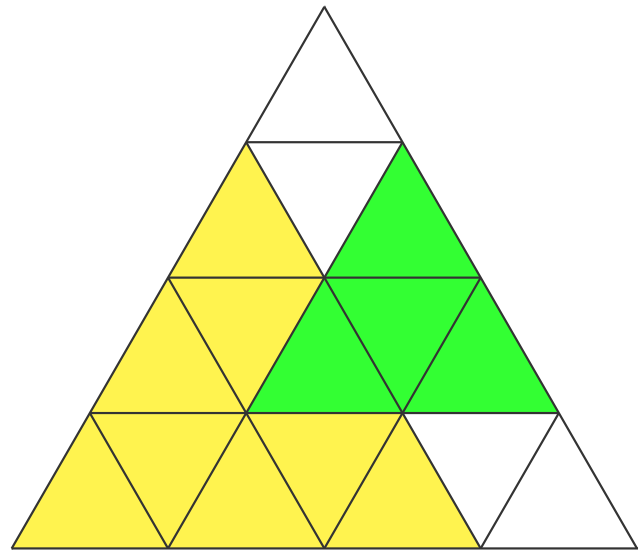
Quantos triângulos em pé (tamanhos variados) podemos encontrar em uma grade de triângulos com altura n?

- No exemplo, a grade tem altura 4

Problema dos Triângulos

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ



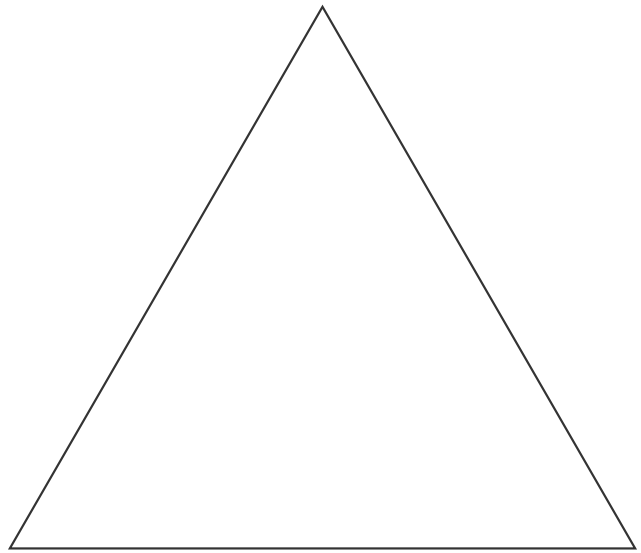


Quantos triângulos em pé (tamanhos variados) podemos encontrar em uma grade de triângulos com altura n?

- No exemplo, a grade tem altura 4

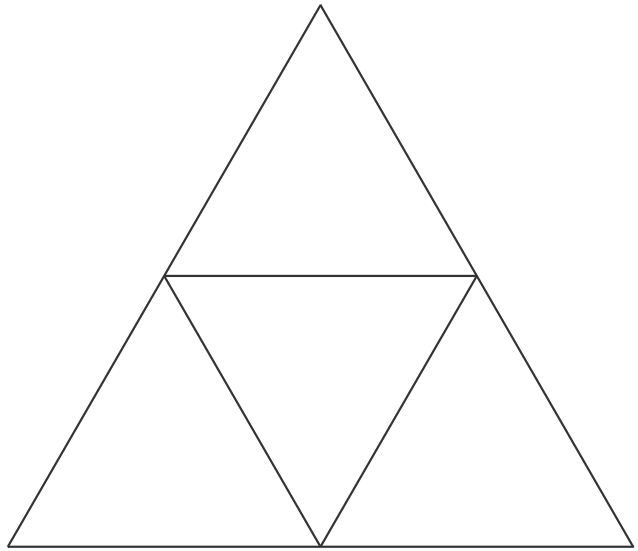
Vamos definir como $t(n)$ o número de triângulos para a altura n.

Triângulos



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

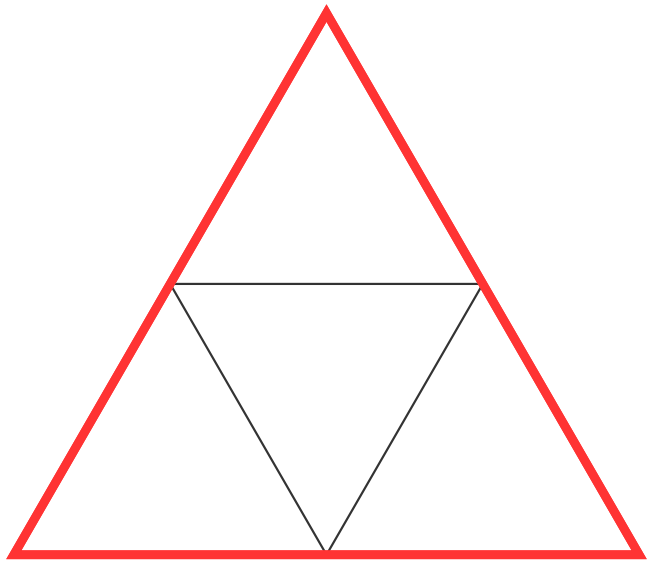
Triângulos



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

- 2 com o vértice superior
- 2 outros triângulos

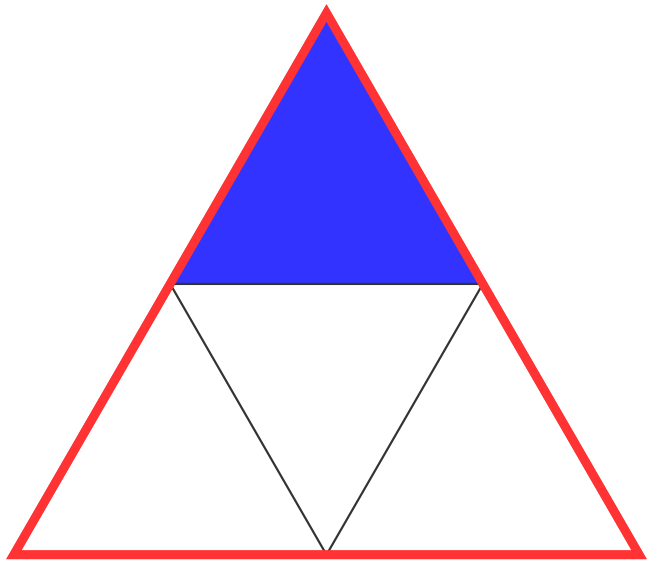
Triângulos



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

- 2 com o vértice superior
- 2 outros triângulos

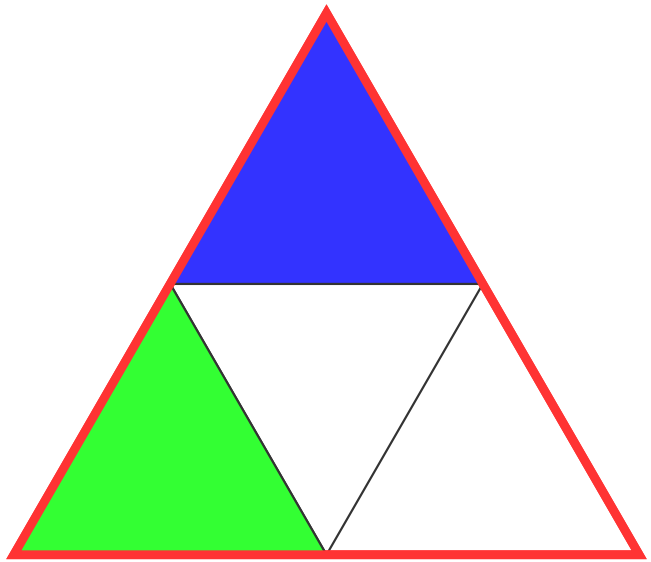
Triângulos



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

- 2 com o vértice superior
- 2 outros triângulos

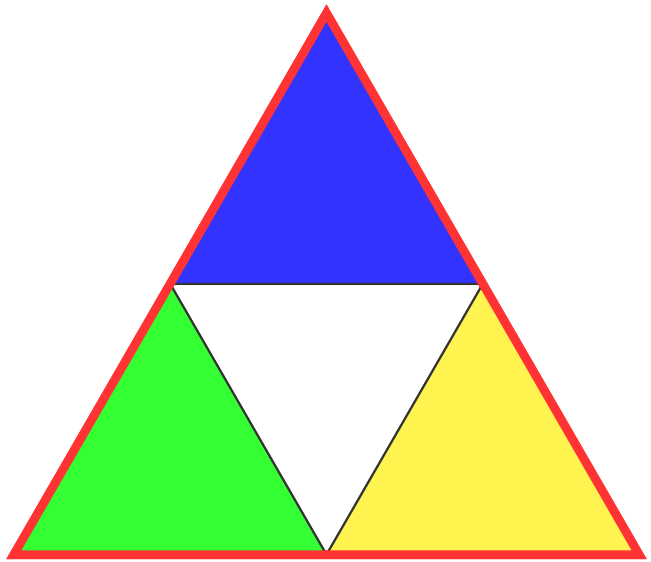
Triângulos



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

- 2 com o vértice superior
- 2 outros triângulos

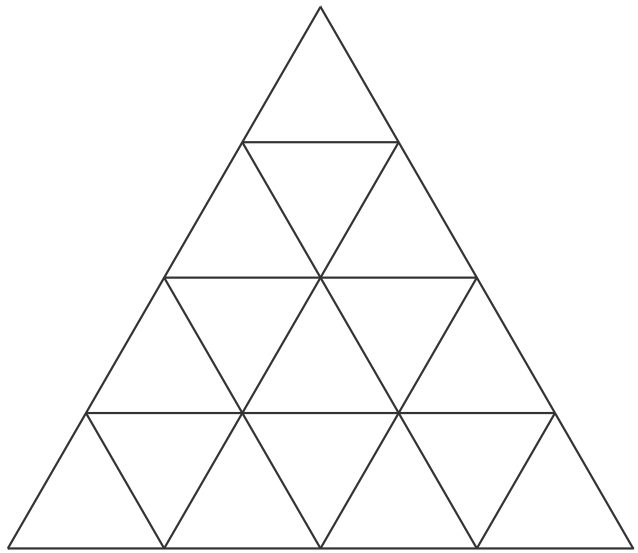
Triângulos



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

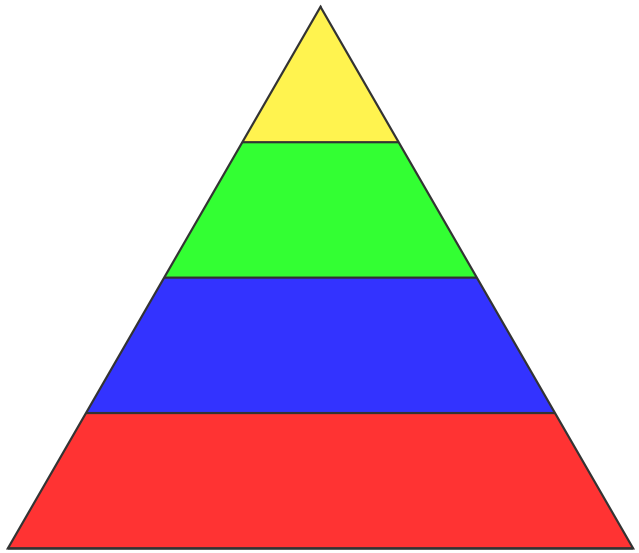
- 2 com o vértice superior
- 2 outros triângulos

Triângulos



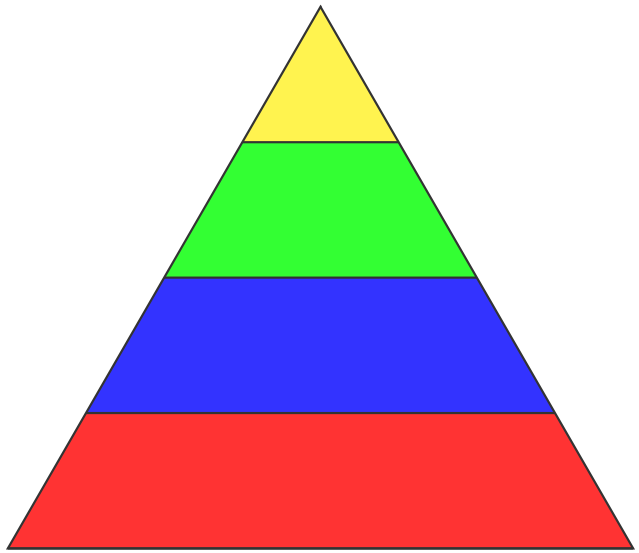
Podemos encontrar algum padrão para $n = 4$?

Triângulos



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

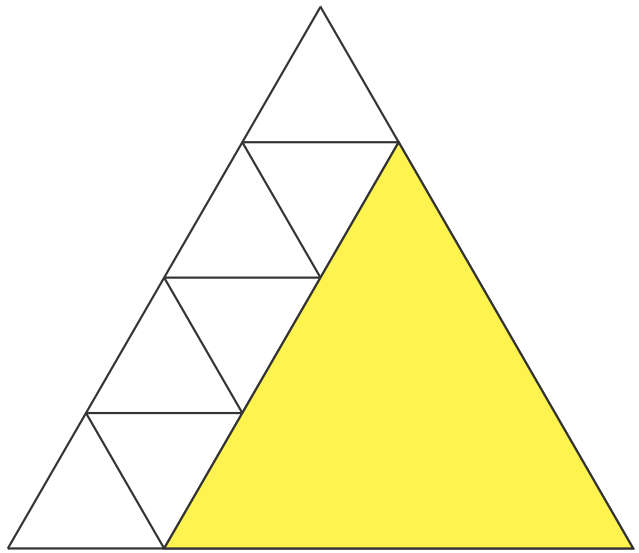
Triângulos



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

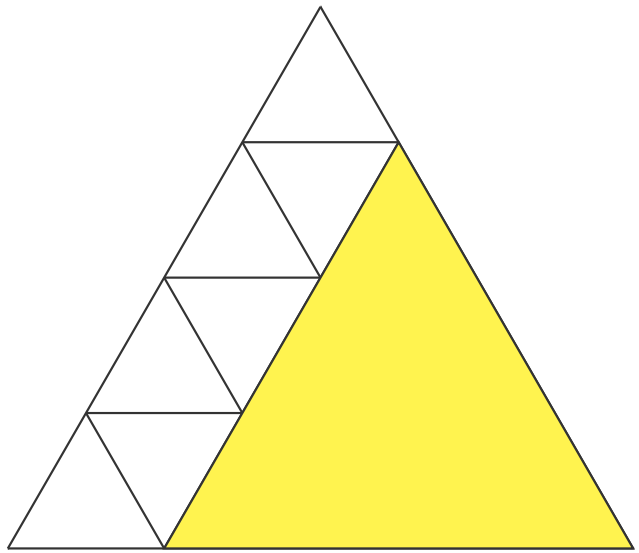
Além desses, quantos outros triângulos faltam?

Triângulos



Faltam os triângulos do lado direito e os do lado esquerdo

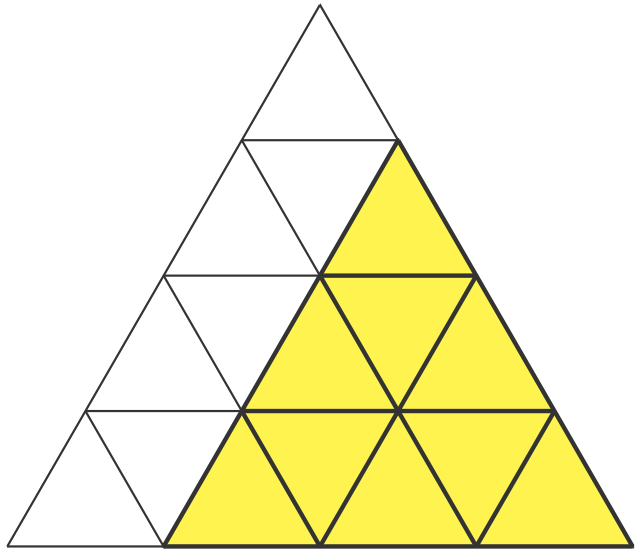
Triângulos



Faltam os triângulos do lado direito e os do lado esquerdo

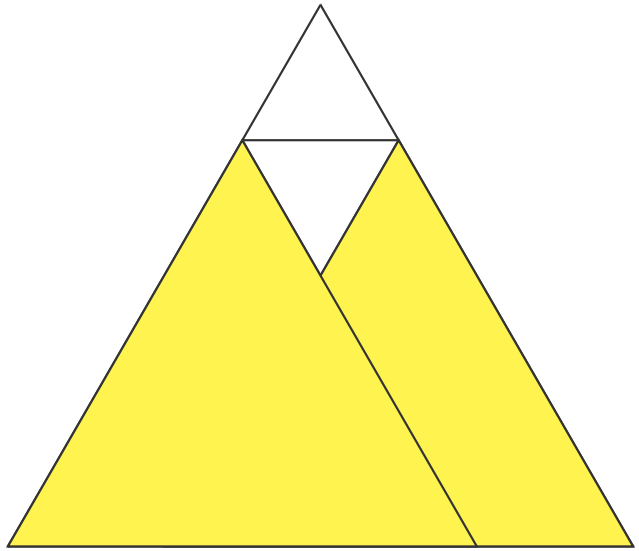
Mas como calcular o número de triângulos de um lado?

Triângulos



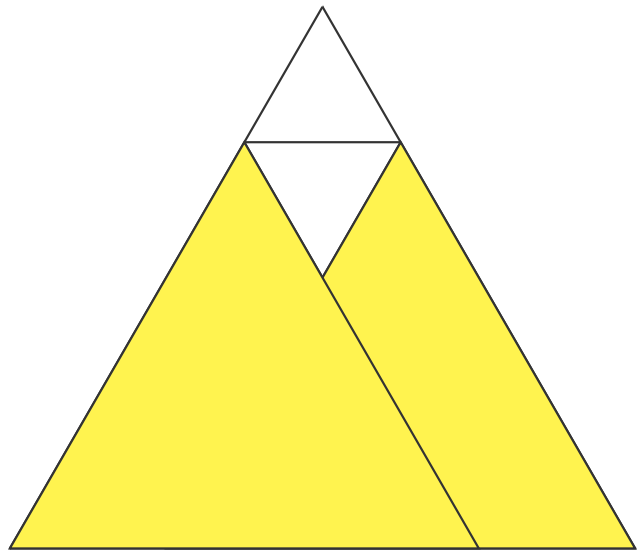
Recaímos no mesmo problema anterior mas agora para $n = 3$

Triângulos



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

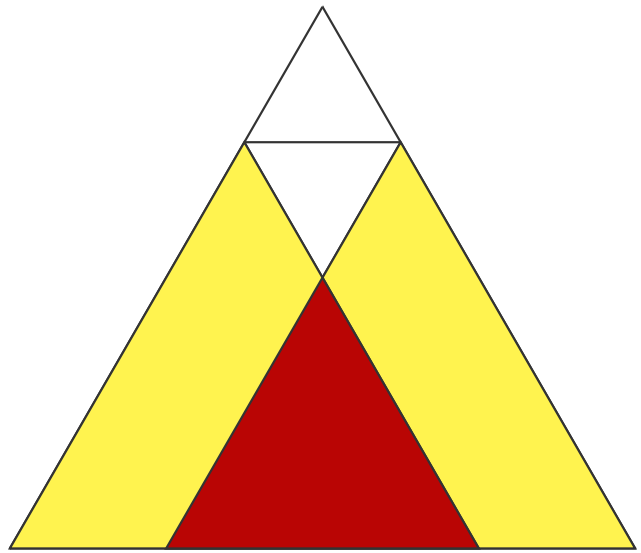
Triângulos



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

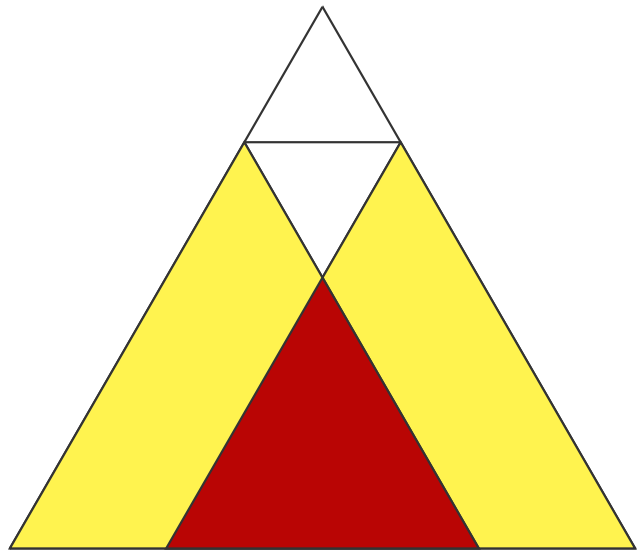
Triângulos



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

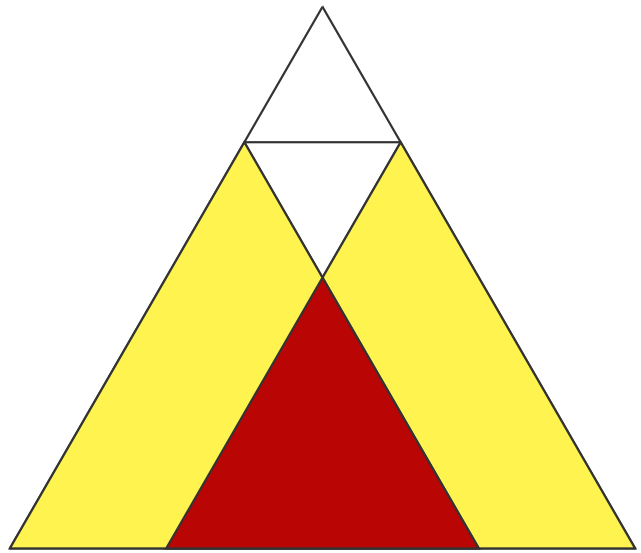


Suponha que já sabemos: $t(1) = 1, t(2) = 4, t(3) = 10$

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

$t(4) =$

Triângulos

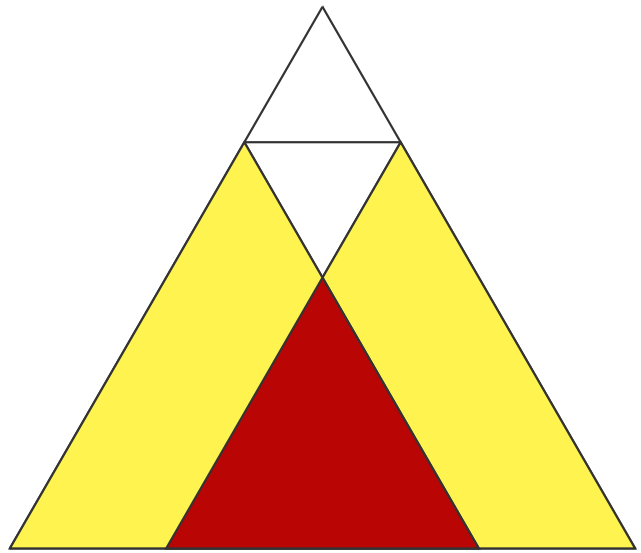


Suponha que já sabemos: $t(1) = 1, t(2) = 4, t(3) = 10$

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

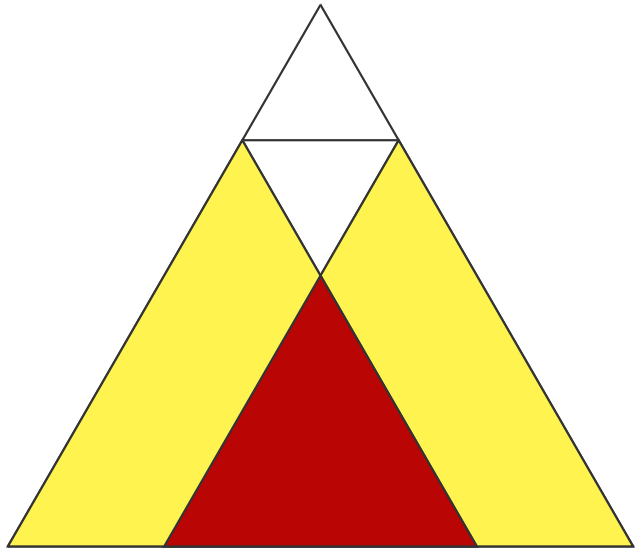


Suponha que já sabemos: $t(1) = 1, t(2) = 4, t(3) = 10$

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

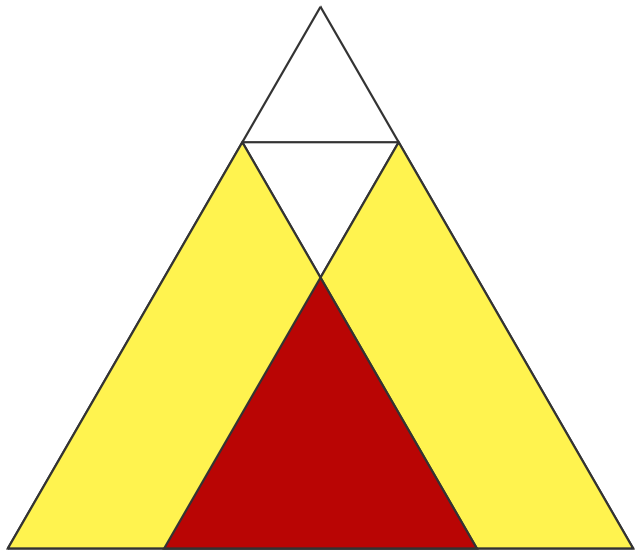
$t(4) = 4 + t(3) + t(3) - t(2) = 20$

Triângulos



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

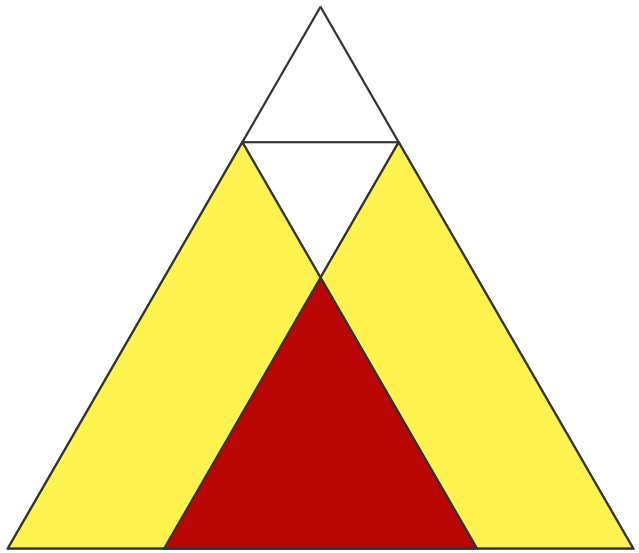
Triângulos



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

- Se $n = 0$, então $t(n) = 0$

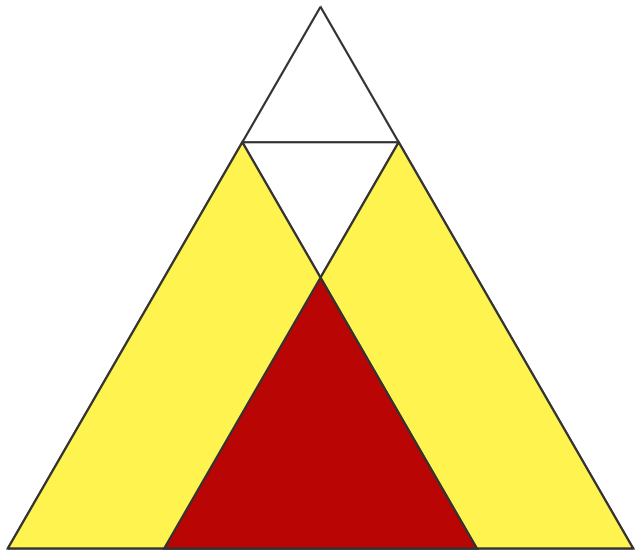
Triângulos



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 }
```

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

FIM