```
int main(int argc, char** argv)
{
  char c = 0;
  char* commands = "ads pq"; // key commands: "left,right,rotate,confirm,pause,quit"
  int speed = 2; // sets max moves per row
  int moves_to_go = 2;
  int full = 0; // whether board is full
  init(); // initialize board an tetrominoes


  cur =
  state[
  state[

  system
  printf

  while
    {
      dr
```

MAC122 - PRINCÍPIOS DE DESENVOLVIMENTO DE ALGORITMOS

Recursão

```
      // process user action
      c = getchar();   // get new action
      if (c == commands[0] && !intersect(cur, state[0]-1, state[1])) state[0]--; // move left
      if (c == commands[1] && !intersect(cur, state[0]+1, state[1])) state[0]++; // move right
      if (c == commands[2] && !intersect(cur->rotated, state[0], state[1])) cur = cur->rotated;
      if (c == commands[3]) moves_to_go=0;

      // scroll down
      if (!moves_to_go--)
        {
          if (intersect(cur,state[0],state[1]+1)) // if tetromino intersected with sth
            {
              cramp_tetromino();
              remove_complete_lines();
              cur = &tetrominoes[rand() % NUM_POSES];
              state[0] = (WIDTH - cur->width)/2;
```

Prove que $\log_a n$ é $O(\log_b n)$ para quaisquer $a, b > 1$

Prove que $\log_a n$ é $O(\log_b n)$ para quaisquer $a, b > 1$

Precisamos mostrar que existem constantes c e n_0 tais que

$$\log_a n < c \cdot \log_b n \quad \text{para todo } n > n_0$$

Prove que $\log_a n$ é $O(\log_b n)$ para quaisquer $a, b > 1$

Precisamos mostrar que existem constantes c e n_0 tais que

$$\log_a n < c \cdot \log_b n \quad \text{para todo } n > n_0$$

Fatos:

1. $\log_a n = \frac{\log_b n}{\log_b a}$

Prove que $\log_a n$ é $O(\log_b n)$ para quaisquer $a, b > 1$

Precisamos mostrar que existem constantes c e n_0 tais que

$$\log_a n < c \cdot \log_b n \quad \text{para todo } n > n_0$$

Fatos:

1. $\log_a n = \frac{\log_b n}{\log_b a}$
2. $x < y \Leftrightarrow \log x < \log y$

Prove que $\log_a n$ é $O(\log_b n)$ para quaisquer $a, b > 1$

Precisamos mostrar que existem constantes c e n_0 tais que

$$\log_a n < c \cdot \log_b n \quad \text{para todo } n > n_0$$

Fatos:

1. $\log_a n = \frac{\log_b n}{\log_b a}$
2. $x < y \Leftrightarrow \log x < \log y$
3. $\log_b n > 0$ para $n > b$

Prove que $\log_a n$ é $O(\log_b n)$ para quaisquer $a, b > 1$

Precisamos mostrar que existem constantes c e n_0 tais que

$$\log_a n < c \cdot \log_b n \quad \text{para todo } n > n_0$$

Fatos:

1. $\log_a n = \frac{\log_b n}{\log_b a}$
2. $x < y \Leftrightarrow \log x < \log y$
3. $\log_b n > 0$ para $n > b$

Prove que $\log_a n$ é $O(\log_b n)$ para quaisquer $a, b > 1$

Precisamos mostrar que existem constantes c e n_0 tais que

$$\log_a n < c \cdot \log_b n \quad \text{para todo } n > n_0$$

Fatos:

1. $\log_a n = \frac{\log_b n}{\log_b a}$
2. $x < y \Leftrightarrow \log x < \log y$
3. $\log_b n > 0$ para $n > b$

Portanto, $\log_a n < (\frac{1}{\log_b a} + \epsilon) \log_b n$ para $n > b$ e $\epsilon > 0$

Prove que $\log_a n$ é $O(\log_b n)$ para quaisquer $a, b > 1$

Precisamos mostrar que existem constantes c e n_0 tais que

$$\log_a n < c \cdot \log_b n \quad \text{para todo } n > n_0$$

Fatos:

1. $\log_a n = \frac{\log_b n}{\log_b a}$
2. $x < y \Leftrightarrow \log x < \log y$
3. $\log_b n > 0$ para $n > b$

Portanto, $\log_a n < (\frac{1}{\log_b a} + \epsilon) \log_b n$ para $n > b$ e $\epsilon > 0$

$\Rightarrow$ base de logaritmo é irrelevante para crescimento assintótico

Prove que $\log(n!)$ é $O(n \log n)$

Prove que $\log(n!)$ é $O(n \log n)$

Precisamos mostrar que existem constantes c e n_0 tais que

$$\log n! < c \cdot n \log n \,, \quad \text{para todo } n > n_0$$

Prove que $\log(n!)$ é $O(n \log n)$

Precisamos mostrar que existem constantes c e n_0 tais que

$$\log n! < c \cdot n \log n\,, \quad \text{para todo } n > n_0$$

Fatos:

$$\log xy = \log x + \log y$$

Prove que $\log(n!)$ é $O(n \log n)$

Precisamos mostrar que existem constantes c e n_0 tais que

$$\log n! < c \cdot n \log n \,, \quad \text{para todo } n > n_0$$

Fatos:

$$\log xy = \log x + \log y$$

$$\begin{aligned} \log n! &= \log 1 + \log 2 + \cdots + \log n \\ &< \log n + \log n + \cdots + \log n \\ &= n \log n \end{aligned}$$

Prove que $\log(n!)$ é $O(n \log n)$

Precisamos mostrar que existem constantes c e n_0 tais que

$$\log n! < c \cdot n \log n\,, \quad \text{para todo } n > n_0$$

Fatos:

$$\log xy = \log x + \log y$$

$$\begin{aligned}\log n! &= \log 1 + \log 2 + \cdots + \log n \\ &< \log n + \log n + \cdots + \log n \\ &= n \log n\end{aligned}$$

Portanto $\log n! < 1 \cdot n \log n$ para $n > n_0 = 1$ $\square$

Prove que $\log n$ é $O(\sqrt{n})$

Prove que $\log n$ é $O(\sqrt{n})$

Primeiro, note que como $\log n$ é $O(\ln n)$, temos que $\log n$ é $O(\sqrt{n}) \Leftrightarrow \ln n$ é $O(\sqrt{n})$

Prove que $\log n$ é $O(\sqrt{n})$

Primeiro, note que como $\log n$ é $O(\ln n)$, temos que $\log n$ é $O(\sqrt{n}) \Leftrightarrow \ln n$ é $O(\sqrt{n})$

Considere $f(n) = \sqrt{n} - \ln(n)$.
Se $f(n) > 0$ então $\ln(n) < \sqrt{n}$.

Prove que $\log n$ é $O(\sqrt{n})$

Primeiro, note que como $\log n$ é $O(\ln n)$, temos que $\log n$ é $O(\sqrt{n}) \Leftrightarrow \ln n$ é $O(\sqrt{n})$

Considere $f(n) = \sqrt{n} - \ln(n)$.
Se $f(n) > 0$ então $\ln(n) < \sqrt{n}$.

Temos que

$$\frac{d}{dn} f(n) = \frac{1}{2\sqrt{n}} - \frac{1}{n} = \frac{\sqrt{n} - 2}{2n} > 0 \text{ para } n > 4$$

Prove que $\log n$ é $O(\sqrt{n})$

Primeiro, note que como $\log n$ é $O(\ln n)$, temos que $\log n$ é $O(\sqrt{n}) \Leftrightarrow \ln n$ é $O(\sqrt{n})$

Considere $f(n) = \sqrt{n} - \ln(n)$.
Se $f(n) > 0$ então $\ln(n) < \sqrt{n}$.

Temos que

$$\frac{d}{dn} f(n) = \frac{1}{2\sqrt{n}} - \frac{1}{n} = \frac{\sqrt{n} - 2}{2n} > 0 \text{ para } n > 4$$

Como $f(e^2) = e - 2 > 0$, $f(n) > 0$ para $n > n_0 = 9$ $\square$

Prove que $n \log n$ é $O(n^{1.5})$

Prove que $n \log n$ é $O(n^{1.5})$

Fato: Se $h_1(n)$ é $O(f(n))$ e $h_2(n)$ é $O(g(n))$ então

$$h(n) = h_1(n)h_2(n) \text{ é } O\big(f(n)g(n)\big)$$

Prove que $n \log n$ é $O(n^{1.5})$

Fato: Se $h_1(n)$ é $O(f(n))$ e $h_2(n)$ é $O(g(n))$ então

$$h(n) = h_1(n)h_2(n) \text{ é } O\big(f(n)g(n)\big)$$

- Seja $h_1(n) = n$, $h_2(n) = \log n$ e $h(n) = n \log n$

Prove que $n \log n$ é $O(n^{1.5})$

Fato: Se $h_1(n)$ é $O(f(n))$ e $h_2(n)$ é $O(g(n))$ então

$$h(n) = h_1(n)h_2(n) \text{ é } O\big(f(n)g(n)\big)$$

- Seja $h_1(n) = n$, $h_2(n) = \log n$ e $h(n) = n \log n$
- $h_1(n)$ é $O(n)$

Prove que $n \log n$ é $O(n^{1.5})$

Fato: Se $h_1(n)$ é $O(f(n))$ e $h_2(n)$ é $O(g(n))$ então

$$h(n) = h_1(n)h_2(n) \text{ é } O\big(f(n)g(n)\big)$$

- Seja $h_1(n) = n$, $h_2(n) = \log n$ e $h(n) = n \log n$
- $h_1(n)$ é $O(n)$
- $h_2(n)$ é $O(\sqrt{n}) = O(n^{1/2})$

Prove que $n \log n$ é $O(n^{1.5})$

Fato: Se $h_1(n)$ é $O(f(n))$ e $h_2(n)$ é $O(g(n))$ então

$$h(n) = h_1(n)h_2(n) \text{ é } O\big(f(n)g(n)\big)$$

- Seja $h_1(n) = n$, $h_2(n) = \log n$ e $h(n) = n \log n$
- $h_1(n)$ é $O(n)$
- $h_2(n)$ é $O(\sqrt{n}) = O(n^{1/2})$

Prove que $n \log n$ é $O(n^{1.5})$

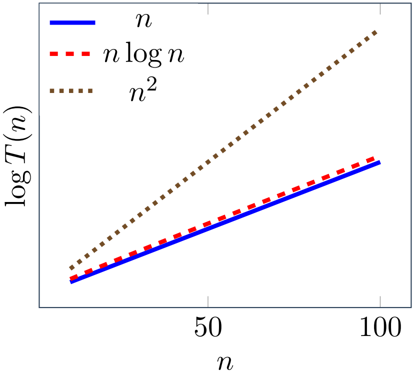
Fato: Se $h_1(n)$ é $O(f(n))$ e $h_2(n)$ é $O(g(n))$ então

$$h(n) = h_1(n)h_2(n) \text{ é } O\big(f(n)g(n)\big)$$

- Seja $h_1(n) = n$, $h_2(n) = \log n$ e $h(n) = n \log n$
- $h_1(n)$ é $O(n)$
- $h_2(n)$ é $O(\sqrt{n}) = O(n^{1/2})$

Portanto, $h(n)$ é $O(n \cdot n^{1/2}) = O(n^{1.5})$ $\square$

Classe	Ordem
constante	$O(1)$
logarítmica	$O(\log n)$
linear	$O(n)$
linearítmico	$O(n \log n)$
quadrática	$O(n^2)$
cúbico	$O(n^3)$
polinomial	$O(n^k)$
exponencial	$O(c^n)$



Função recursiva

chama a si mesma

```
int f(int n) {
  if (n == 0) return 0;
  return f(n - 1) + 1;
}
```

```
int g(int n) {
  if (n == 0) return 0;
  return g(n + 1) + 1;
}
```

Função recursiva

chama a si mesma

```
int f(int n) {
  if (n == 0) return 0;
  return f(n - 1) + 1;
}
```

```
int g(int n) {
  if (n == 0) return 0;
  return g(n + 1) + 1;
}
```

- Funções recursivas podem levar a execuções infinitas chama a si mesma

```
int f(int n) {
  if (n == 0) return 0;
  return f(n - 1) + 1;
}
```

```
int g(int n) {
  if (n == 0) return 0;
  return g(n + 1) + 1;
}
```

- Funções recursivas podem levar a execuções infinitas
  - `f` termina sse `n > 0`, `g` termina sse `n < 0`

FUNÇÃO RECURSIVA

chama a si mesma

```
int f(int n) {
  if (n == 0) return 0;
  return f(n - 1) + 1;
}
```

```
int g(int n) {
  if (n == 0) return 0;
  return g(n + 1) + 1;
}
```

- Funções recursivas podem levar a execuções infinitas
  - `f` termina sse `n > 0`, `g` termina sse `n < 0`
- Uma função recursiva termina se

chama a si mesma

```
int f(int n) {
  if (n == 0) return 0;
  return f(n - 1) + 1;
}
```

```
int g(int n) {
  if (n == 0) return 0;
  return g(n + 1) + 1;
}
```

- Funções recursivas podem levar a execuções infinitas
  - `f` termina sse `n > 0`, `g` termina sse `n < 0`
- Uma função recursiva termina se
  1. Existe uma condição de parada, e

FUNÇÃO RECURSIVA

chama a si mesma

```c
int f(int n) {
  if (n == 0) return 0;
  return f(n - 1) + 1;
}
```

```c
int g(int n) {
  if (n == 0) return 0;
  return g(n + 1) + 1;
}
```

- Funções recursivas podem levar a execuções infinitas
  - `f` termina sse `n > 0`, `g` termina sse `n < 0`
- Uma função recursiva termina se
  1. Existe uma condição de parada, e
  2. A cada recursão a função se aproxima da condição de chamada

Certos problemas exibem uma estrutura recursiva, significando que podemos resolvê-los pela seguinte abordagem:

```
se (instância é pequena)
  resolva-a diretamente
caso contrário
  divida o problema em problemas menores
  resolva cada subproblema chamando essa rotina
  construa a solução do problema a partir das soluções
```

Qualquer algoritmo que siga essa abordagem é chamado algoritmo recursivo

```
/* Algoritmo recursivo para computar n! */
int Fatorial (int n) {
  int f;
  /* se instância for pequena */
  if (n == 0 || n == 1) return 1; /* resolve diretamente */
  /* caso contrário, divide em problemas menores */
  f = Fatorial(n-1);
  /* e constrói solução */
  return n * f;
}

/* Algoritmo iterativo para computar n! */
int Fatorial (int n) {
  int res = 1;
  while (n) res *= n--;
  return res;
}
```

Funções matemáticas nos inteiros definidas por identidades com a função ocorrendo nos dois lados da igualdade

Funções matemáticas nos inteiros definidas por identidades com a função ocorrendo nos dois lados da igualdade

Exemplo:

- Definição não recursiva de fatorial:

$$n! = 1 \cdot 2 \cdot \cdots \cdot n$$

Funções matemáticas nos inteiros definidas por identidades com a função ocorrendo nos dois lados da igualdade

Exemplo:

- Definição não recursiva de fatorial:

$$n! = 1 \cdot 2 \cdot \dots \cdot n$$

- Definição recursiva de fatorial usando relação de recorrência:

$$n! = \begin{cases} n \cdot (n-1)!, & \text{se } n > 1 \\ 1, & \text{se } n \leq 1 \end{cases}$$

Funções matemáticas nos inteiros definidas por identidades com a função ocorrendo nos dois lados da igualdade

Exemplo:

- Definição não recursiva de fatorial:

$$n! = 1 \cdot 2 \cdot \cdots \cdot n$$

- Definição recursiva de fatorial usando relação de recorrência:

$$n! = \begin{cases} n \cdot (n-1)!, & \text{se } n > 1 \\ 1, & \text{se } n \leq 1 \end{cases}$$

Funções matemáticas nos inteiros definidas por identidades com a função ocorrendo nos dois lados da igualdade

Exemplo:

- Definição não recursiva de fatorial:

$$n! = 1 \cdot 2 \cdot \cdots \cdot n$$

- Definição recursiva de fatorial usando relação de recorrência:

$$n! = \begin{cases} n \cdot (n-1)!, & \text{se } n > 1 \\ 1, & \text{se } n \leq 1 \end{cases}$$

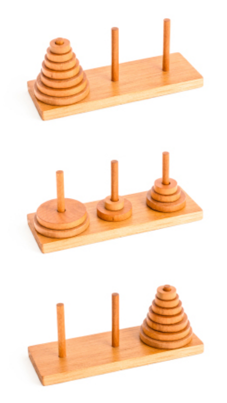
Definição recursiva sugere abordagem recursiva

NÚMEROS DE FIBONACCI

$$F_n = \begin{cases} F_{n-1} + F_{n-2}, & \text{se } n > 2 \\ 1, & \text{se } n = 1 \text{ ou } 2 \end{cases}$$

```
/* Algoritmo recursivo */
int Fibonacci (int n) {
  int f1, f2;
  if (n <= 2) return 1; /* resolve instância pequena */
  /* resolve subproblemas e constrói solução */
  f1 = Fibonacci (n - 1);
  f2 = Fibonacci (n - 2)
  return f1 + f2;
}

/* Algoritmo iterativo */
int Fibonacci (int n) {
  int x = 1, y = 1, z, i;
  for (i = 3; i <= n; i++) {
     z = x;   x = x + y;   y = x;
  }
  return x;
}
```



- n discos, 3 agulhas
- Disco maior não pode ser colocado sobre disco menor
- Discos começam na agulha 1
- Objetivo: mover todos os discos para agulha 3

Para mover disco da base da agulha 1 para agulha 3 faça:

- Mova todos os outros discos para a agulha 2 (aux)
- Mova o disco maior para a agulha 3 (destino)
- Mova todos os outros discos para a agulha 3

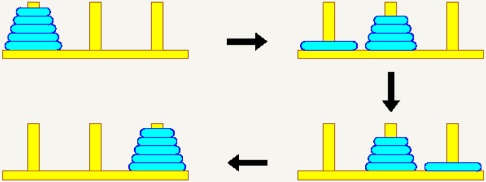


imagem: `http://www.dcs.warwick.ac.uk/~sgm/open/hanoi.html`

```c
void Hanoi (int n, char origem, char dest, char aux) {
 if (n == 0) return;
 Hanoi (n-1, origem, aux, dest);
 printf ("Mova disco %d de %c para %c\n", n, origem, dest);
 Hanoi (n-1, aux, dest, origem);
}

int main() {
  Hanoi (4, 'A', 'C', 'B');
}
```

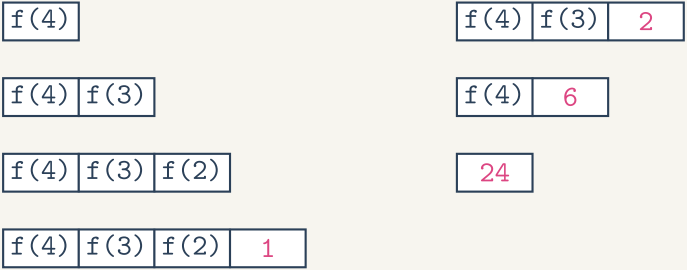
Torres de Hanoi: Algoritmo Recursivo

```
Mova disco 1 de A para B
Mova disco 2 de A para C
Mova disco 1 de B para C
Mova disco 3 de A para B
Mova disco 1 de C para A
Mova disco 2 de C para B
Mova disco 1 de A para B
Mova disco 4 de A para C
Mova disco 1 de B para C
Mova disco 2 de B para A
Mova disco 1 de C para A
Mova disco 3 de B para C
Mova disco 1 de A para B
Mova disco 2 de A para C
Mova disco 1 de B para C
```

Pilha de execução

Chamadas a funções recursivas são armazenas na pilha de execução

```
int Fatorial(int n) {
    if (n == 0 || n == 1) return 1;
    int f = Fatorial(n-1);
    return n*f;
}
printf("%d", Fatorial(4)); /* 24 */
```



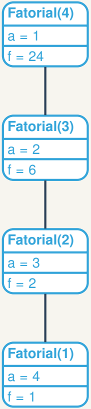
Maior número de chamadas recursivas feitas sem retornar; Igual ao maior tamanho da pilha de execução

```
int Fatorial (int n) {
  int f;
  if (n == 0 || n == 1) return 1;
  f = Fatorial (n - 1);
  return n * f;
}
```

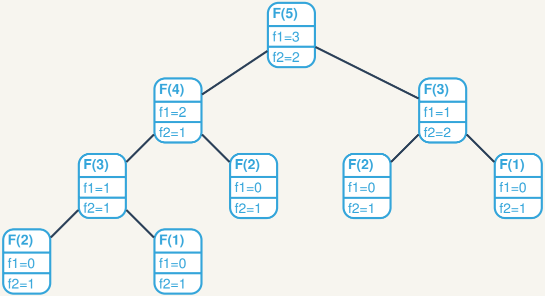
Profundidade de recursão determina (limite inferior na) complexidade de espaço do algoritmo

PROFUNDIDADE DE RECURSÃO: EXEMPLO

```c
int Fatorial(int n) {
  static int a = 0; /* mede profundidade da recursão */
  int f;
 a++; /* aumenta profundidade */
  if (n < 2) { f = 1; }
  else f = n * Fatorial (n - 1);
 a--; /* diminui profundidade */
  return f;
}
```



```c
int Fibonacci (int n) {
  static int a = 0; /* mede profundidade */
  int f1, f2;
  a++; /* aumenta profundidade */
  if (n <= 2) { f1 = 0; f2 = 1; }
  else { f1 = Fibonacci(n - 1); f2 = Fibonacci(n - 2); }
  a--; /* diminui profundidade */
  return f1 + f2;
}
```



```
int Fatorial (int n) {
  if (n ==0 || n == 1) return 1;
  return n * Fatorial (n - 1);
}
```

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

tempo total normalizado
pelo tempo de uma iteração

```
int Fatorial (int n) {
  if (n ==0 || n == 1) return 1;
  return n * Fatorial (n - 1);
}
```

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

tempo total normalizado pelo tempo de uma iteração

```
int Fatorial (int n) {
  if (n ==0 || n == 1) return 1;
  return n * Fatorial (n - 1);
}
```

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

tempo total normalizado
pelo tempo de uma iteração

```c
int Fatorial (int n) {
  if (n ==0 || n == 1) return 1;
  return n * Fatorial (n - 1);
}
```

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

tempo total normalizado
pelo tempo de uma iteração

Complexidade de funções recursivas

```
int Fatorial (int n) {
  if (n ==0 || n == 1) return 1;
  return n * Fatorial (n - 1);
}
```

$$
\begin{aligned}
T(n) &= T(n-1) + 1 && \text{tempo total normalizado} \\
&= T(n-2) + 1 + 1 && \text{pelo tempo de uma iteração} \\
&\;\;\vdots \\
&= T(1) + (n-1) = n && \text{é } O(n) \text{ complexidade linear}
\end{aligned}
$$

```
int Fatorial (int n) {
  if (n ==0 || n == 1) return 1;
  return n * Fatorial (n - 1);
}
```

$$
\begin{aligned}
T(n) &= T(n-1) + 1 && \text{tempo total normalizado} \\
&= T(n-2) + 1 + 1 && \text{pelo tempo de uma iteração} \\
&\;\vdots \\
&= T(1) + (n-1) = n && \text{é } O(n) \text{ complexidade linear}
\end{aligned}
$$

Para casa: provar por indução que $T(n) = n$

```
int Fatorial (int n) {
  if (n == 0 || n == 1) return 1;
  return n * Fatorial (n - 1);
}

int Fatorial (int n) {
  int res = 1;
  while (n) res *= n--;
  return res;
}
```

- Versões iterativa e recursiva possuem complexidade de tempo linear, $O(n)$
- Versão recursiva possui sobrecarga de chamadas recursivas (tempo $O(1)$ e espaço $O(n)$)

```
int Fibonacci(int n) {
  if (n == 1 || n == 2) return 1;
  return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

Tempo de execução $T(n) =?$

$$F_n = F_{n-1} + F_{n-2}$$

$$
\begin{aligned}
F_n &= F_{n-1} + F_{n-2} \\
F_{n-1} &= F_{n-2} + F_{n-3}
\end{aligned}
$$

$$
\begin{aligned}
F_n &= F_{n-1} + F_{n-2} \\
F_{n-1} &= F_{n-2} + F_{n-3} \\
F_{n-2} &= F_{n-3} + F_{n-4}
\end{aligned}
$$

$$
\begin{aligned}
F_n &= F_{n-1} + F_{n-2} \\
F_{n-1} &= F_{n-2} + F_{n-3} \\
F_{n-2} &= F_{n-3} + F_{n-4} \\
\vdots \quad & \quad \vdots
\end{aligned}
$$

$$
\begin{aligned}
F_n &= F_{n-1} + F_{n-2} \\
F_{n-1} &= F_{n-2} + F_{n-3} \\
F_{n-2} &= F_{n-3} + F_{n-4} \\
&\vdots \qquad \vdots \\
F_3 &= F_2 + F_1 \\
F_2 &= F_1 \\
F_1 &= 1
\end{aligned}
$$

$$
\begin{aligned}
F_n &= F_{n-1} + F_{n-2} \\
F_{n-1} &= F_{n-2} + F_{n-3} \\
F_{n-2} &= F_{n-3} + F_{n-4} \\
&\vdots \\
F_3 &= F_2 + F_1 \\
F_2 &= F_1 \\
F_1 &= 1
\end{aligned}
$$

$$
\begin{aligned}
F_n &= F_{n-1} + F_{n-2} \\
F_{n-1} &= F_{n-2} + F_{n-3} \\
F_{n-2} &= F_{n-3} + F_{n-4} \\
&\vdots \qquad \vdots \\
F_3 &= F_2 + F_1 \\
F_2 &= F_1 \\
F_1 &= 1
\end{aligned}
$$

$$\sum_{i=1}^{n} F_i = 1 + 2F_1 + \cdots + 2F_{n-2} + F_{n-1}$$

$$
\begin{aligned}
F_n &= F_{n-1} + F_{n-2} \\
F_{n-1} &= F_{n-2} + F_{n-3} \\
F_{n-2} &= F_{n-3} + F_{n-4} \\
&\vdots \qquad \vdots \\
F_3 &= F_2 + F_1 \\
F_2 &= F_1 \\
F_1 &= 1
\end{aligned}
$$

$$
\begin{aligned}
\sum_{i=1}^{n} F_i &= 1 + 2F_1 + \cdots + 2F_{n-2} + F_{n-1} \\
&= 1 + F_{n-1} + 2\sum_{i=1}^{n-2} F_i
\end{aligned}
$$

$$
\begin{aligned}
F_n &= F_{n-1} + F_{n-2} \\
F_{n-1} &= F_{n-2} + F_{n-3} \\
F_{n-2} &= F_{n-3} + F_{n-4} \\
&\;\vdots \qquad \vdots \\
F_3 &= F_2 + F_1 \\
F_2 &= F_1 \\
F_1 &= 1
\end{aligned}
\qquad \Rightarrow \boxed{\sum_{i=1}^{n-2} F_i = F_n - 1}
$$

$$
\begin{aligned}
\sum_{i=1}^{n} F_i &= 1 + 2F_1 + \cdots + 2F_{n-2} + F_{n-1} \\
&= 1 + F_{n-1} + 2\sum_{i=1}^{n-2} F_i
\end{aligned}
$$

Prove que $F_n < 2^n$

Prove que $F_n < 2^n$

Indução matemática:

- Base: $F_1 = 1 < 2^1 = 2$

Prove que $F_n < 2^n$

Indução matemática:

- Base: $F_1 = 1 < 2^1 = 2$
- Passo indutivo: Assuma que $F_n < 2^n$.

Prove que $F_n < 2^n$

Indução matemática:

- Base: $F_1 = 1 < 2^1 = 2$
- Passo indutivo: Assuma que $F_n < 2^n$. Então

$$F_{n+1} = F_n + F_{n-1} < 2F_n < 2 \cdot 2^n = 2^{n+1} \quad \square$$

Prove que $F_n < 2^n$

Indução matemática:

- Base: $F_1 = 1 < 2^1 = 2$
- Passo indutivo: Assuma que $F_n < 2^n$. Então

$$F_{n+1} = F_n + F_{n-1} < 2F_n < 2 \cdot 2^n = 2^{n+1} \quad \square$$

$\Rightarrow F_n$ é $O(2^n)$

```
int Fibonacci(int n) {
  if (n == 1 || n == 2) return 1;
  return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

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

```
int Fibonacci(int n) {
  if (n == 1 || n == 2) return 1;
  return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

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

```
int Fibonacci(int n) {
  if (n == 1 || n == 2) return 1;
  return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

$$
\begin{aligned}
T(n) &= T(n-1) + T(n-2) + 1 \\
&= 2T(n-2) + T(n-3) + 1 + 1 \\
&= 3T(n-3) + 2T(n-4) + 1 + 1 + 2
\end{aligned}
$$

```c
int Fibonacci(int n) {
  if (n == 1 || n == 2) return 1;
  return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

$$
\begin{aligned}
T(n) &= T(n-1) + T(n-2) + 1 \\
&= 2T(n-2) + T(n-3) + 1 + 1 \\
&= 3T(n-3) + 2T(n-4) + 1 + 1 + 2 \\
&= 5T(n-4) + 3T(n-5) + 1 + 1 + 2 + 3
\end{aligned}
$$

```c
int Fibonacci(int n) {
  if (n == 1 || n == 2) return 1;
  return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

$$
\begin{aligned}
T(n) &= T(n-1) + T(n-2) + 1 \\
&= 2T(n-2) + T(n-3) + 1 + 1 \\
&= 3T(n-3) + 2T(n-4) + 1 + 1 + 2 \\
&= 5T(n-4) + 3T(n-5) + 1 + 1 + 2 + 3 \\
&\vdots
\end{aligned}
$$

```
int Fibonacci(int n) {
  if (n == 1 || n == 2) return 1;
  return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

$$\begin{aligned}
T(n) &= T(n-1) + T(n-2) + 1 \\
&= 2T(n-2) + T(n-3) + 1 + 1 \\
&= 3T(n-3) + 2T(n-4) + 1 + 1 + 2 \\
&= 5T(n-4) + 3T(n-5) + 1 + 1 + 2 + 3 \\
&\vdots \\
&= \boxed{AT(2) + BT(1) + C}?
\end{aligned}$$

```
int Fibonacci(int n) {
  if (n == 1 || n == 2) return 1;
  return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

$$
\begin{aligned}
T(n) &= T(n-1) + T(n-2) + 1 \\
&= 2T(n-2) + T(n-3) + 1 + 1 \\
&= 3T(n-3) + 2T(n-4) + 1 + 1 + 2 \\
&= 5T(n-4) + 3T(n-5) + 1 + 1 + 2 + 3 \\
&\ \vdots \\
&= \boxed{F_{n-1}T(2) + F_{n-2}T(1) + \sum_{i=1}^{n-2} F_i}
\end{aligned}
$$

```c
int Fibonacci(int n) {
  if (n == 1 || n == 2) return 1;
  return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

$$\begin{aligned}
T(n) &= T(n-1) + T(n-2) + 1 \\
&= 2T(n-2) + T(n-3) + 1 + 1 \\
&= 3T(n-3) + 2T(n-4) + 1 + 1 + 2 \\
&= 5T(n-4) + 3T(n-5) + 1 + 1 + 2 + 3 \\
&\vdots \\
&= \boxed{F_{n-1} + F_{n-2} + F_n - 1}
\end{aligned}$$

Complexidade

```c
int Fibonacci(int n) {
  if (n == 1 || n == 2) return 1;
  return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

$$
\begin{aligned}
T(n) &= T(n-1) + T(n-2) + 1 \\
&= 2T(n-2) + T(n-3) + 1 + 1 \\
&= 3T(n-3) + 2T(n-4) + 1 + 1 + 2 \\
&= 5T(n-4) + 3T(n-5) + 1 + 1 + 2 + 3 \\
&\vdots \\
&= \boxed{F_n + F_n - 1} = 2F_n - 1
\end{aligned}
$$

```c
int Fibonacci(int n) {
  if (n == 1 || n == 2) return 1;
  return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

$$
\begin{aligned}
T(n) &= T(n-1) + T(n-2) + 1 \\
&= 2T(n-2) + T(n-3) + 1 + 1 \\
&= 3T(n-3) + 2T(n-4) + 1 + 1 + 2 \\
&= 5T(n-4) + 3T(n-5) + 1 + 1 + 2 + 3 \\
&\vdots \\
&= \boxed{F_n + F_n - 1} = 2F_n - 1
\end{aligned}
$$

```
int Fibonacci(int n) {
  if (n == 1 || n == 2) return 1;
  return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

$$\boxed{T(n) = 2F_n - 1 \text{ é } O(2^n)}$$

```c
int Fibonacci(int n) {
  if (n == 1 || n == 2) return 1;
  return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

$$\boxed{T(n) = 2F_n - 1 \text{ é } O(2^n)}$$

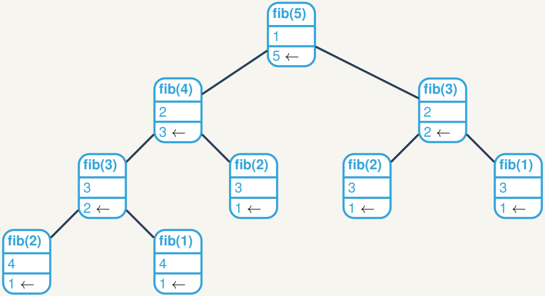
$\Rightarrow$ Complexidade exponencial em n

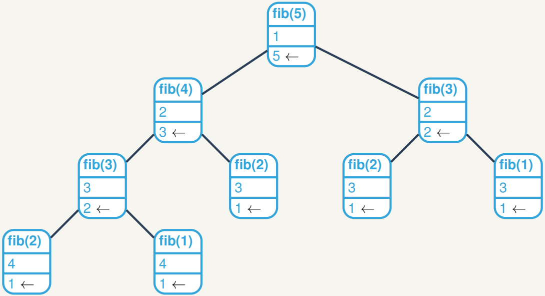
```
int Fibonacci(int n) {
  if (n == 1 || n == 2) return 1;
  return Fibonacci(n - 1) + Fibonacci(n - 2);
}

int Fibonacci(int n) {
  int x = 1, y = 1, z, i;
  for (i = 3; i <= n; i++) {
     z = x;   x = x + y;   y = x;
  }
  return x;
}
```

- Versão iterativa: tempo $O(n)$, espaço $O(1)$
- Versão recursiva: tempo $O(2^n)$, espaço $O(n)$

Memorização






```
int FibM(int n, int v[]) {
  if (v[n] == 0) v[n] = FibM(n - 1,v) + FibM(n - 2,v);
  return v[n];
}
int Fibonacci(int n) {
  int *v = calloc(n,sizeof(int)); v[1] = v[2] = 1;
  return FibM(n, v);
}
```

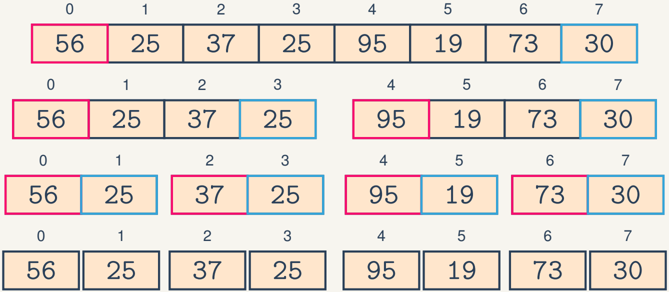
```c
int FibM(int n, int v[]) {
  if (v[n] == 0) v[n] = FibM(n - 1,v) + FibM(n - 2,v);
  return v[n];
}
int Fibonacci(int n) {
  int *v = calloc(n,sizeof(int)); v[1] = v[2] = 1;
  return FibM(n, v);
}
```

- Versão iterativa: tempo $O(n)$, espaço $O(1)$
- Versão recursiva ingênua: tempo $O(2^n)$, espaço $O(n)$
- Versão recursiva com mem.: tempo $O(n)$, espaço $O(n)$

```c
void Hanoi (int n, char origem, char dest, char aux) {
 if (n == 0) return;
 Hanoi (n-1, origem, aux, dest);
 printf ("Mova disco %d de %c para %c\n", n, origem, dest);
 Hanoi (n-1, aux, dest, origem);
}
```

$$
\begin{aligned}
T(n) &= 2T(n-1) + 1 \\
&= 2^2 T(n-2) + 2 \\
&\;\vdots \\
&= 2^n T(0) + 2^n - 1 = 2^n - 1 \qquad \text{exponencial}
\end{aligned}
$$

```
int Maximo (int v[], int e, int d) {
  int x, y;   int c = (e+d)/2;
  if (e == d) return v[e];
  x = Maximo (v, e, c);
  y = Maximo (v, c + 1, d);
  if (x > y) return x; else return y;
}
```



Maximizador Recursivo: Correção

```
int Maximo (int v[], int e, int d) {
  int x, y;   int c = (e+d)/2;
  if (e == d) return v[e];
  x = Maximo (v, e, c);
  y = Maximo (v, c + 1, d);
  if (x > y) return x; else return y;
}
```

Prova por indução no tamanho do vetor $n = \mathtt{d} - \mathtt{e} + 1 = 2^k$

- Base: para $k = 0$, função retorna máximo
- Passo indutivo: se função é correta para 2^k, então função é correta para 2^{k+1}

```
int Maximo (int v[], int e, int d) {
  int x, y;    int c = (e+d)/2;
  if (e == d) return v[e];
  x = Maximo (v, e, c);
  y = Maximo (v, c + 1, d);
  if (x > y) return x; else return y;
}
```

T(n) = ?

Dica: assuma que $n = 2^k$ por conveniência

```c
int Maximo (int v[], int e, int d) {
  int x, y;   int c = (e+d)/2;
  if (e == d) return v[e];
  x = Maximo (v, e, c);
  y = Maximo (v, c + 1, d);
  if (x > y) return x; else return y;
}
```

Assumindo que $n = 2^k$:

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

EXPONENCIAÇÃO

$$x^n = \begin{cases} x \cdot x^{n-1}, & \text{se } n > 1 \\ 1, & \text{se } n = 0 \end{cases}$$

```
/* Computa x elevado a n */
double Potencia1 (double x, int n) {
  if (n == 0) return 1;
  return x*Potencia1 (x, n-1)
}
```

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

$$x^n = \begin{cases} (x^{n/2})^2, & \text{se } n > 1 \text{ e par} \\ (x^{\lfloor n/2 \rfloor})^2 x, & \text{se } n > 1 \text{ e ímpar} \\ 1, & \text{se } n = 0 \end{cases}$$

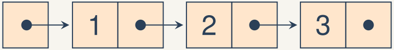
```
/* Computa x elevado a n */
double Potencia2 (double x, int n) {
  double y;
  if (n == 0) return 1;
  y = Potencia2 (x, n/2);
  if (n % 2 == 0) return y*y;
  else return y*y*x;
}
```

```
double Potencia2 (double x, int n) {
  double y;
  if (n == 0) return 1;
  y = Potencia2 (x, n/2);
  if (n % 2 == 0) return y*y;
  else return y*y*x;
}
```

Vamos assumir que $n = 2^k$:

$$T(2^k) = T(2^{k-1}) + 1 = T(2^{k-2}) + 2 = \cdots = T(1) + k$$

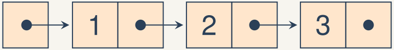
Portanto, $T(n) = O(\log n)$



Uma lista encadeada pode ser definida de forma indutiva como:

- Uma celula vazia (`NULL`) é uma lista encadeada
- Uma célula contendo um ponteiro para uma lista encadeada é uma lista encadeada
- nada mais é uma lista encadeada

Lista Encadeada Como Tipo de Dado Recursivo

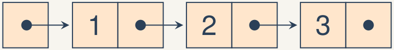


Uma lista encadeada pode ser definida de forma indutiva como:

- Uma celula vazia (`NULL`) é uma lista encadeada
- Uma célula contendo um ponteiro para uma lista encadeada é uma lista encadeada
- nada mais é uma lista encadeada

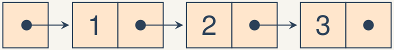
Definição recursiva leva naturalmente a algoritmos recursivos

Tamanho de Lista Recursivo



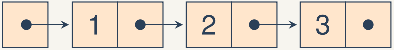

```
int Tamanho (celula *lista) {
    /* Implementação
        recursiva? */
}
```

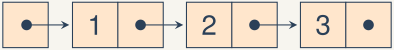
Tamanho de Lista Recursivo




```
int Tamanho (celula *lista) {
   if (lista == NULL) return 0;
   return 1 + Tamanho (lista->prox);
}
```

Busca em Lista Recursivo






```
int Busca (int x, celula *lista) {
   if (lista == NULL) return NULL;
   if (lista->valor == x) return lista;
   return Busca (x, lista->prox);
}
```

PARA CASA

Exercícios 11A-11C