Análise de Algoritmos

Alternative
Big O Notation

$O(1) = O(\text{yeah})$

$O(\log n) = O(\text{nice})$

$O(n) = O(\text{k})$

$O(n^2) = O(\text{my})$

$O(2^n) = O(\text{no})$

$O(n!) = O(\text{mg})$

$O(n^n) = O(\text{sh*t!})$

Análise do Tempo de Execução

- Comando de atribuição, de leitura ou de escrita: $O(1)$
- Sequência de comandos: determinado pelo maior tempo de execução de qualquer comando da sequência
- Comando de decisão: tempo dos comandos dentro do comando condicional, mais tempo para avaliar a condição.
- Anel: soma do tempo de execução do corpo do anel mais o tempo de avaliar a condição para terminação (geralmente $O(1)$), multiplicado pelo número de iterações.
- Procedimentos não recursivos: cada um deve ser computado separadamente um a um, iniciando com os que não chamam outros procedimentos. Avaliam-se então os que chamam os já avaliados (utilizando os tempos desses). O processo é repetido até chegar no programa principal.

Programa 1.5 *Programa para ordenar*

```
    procedure Ordena (var A: TipoVetor);
    { ordena o vetor A em ordem ascendente }
    var  i, j, min, x: integer;
    begin
(1)   for i := 1 to n-1 do
        begin
(2)     min := i;
(3)     for j:= i+1 to n do
(4)       if A[j] < A[min]
(5)       then min := j;
        { troca A[min] e A[i] }
(6)     x := A[min];
(7)     A[min] := A[i];
(8)     A[i] := x;
        end;
```

Operações Notação O

Tabela 1.2 Operações com a notação O

$$
\begin{array}{rcll}
f(n) & = & O(f(n)) & \\
c \times O(f(n)) & = & O(f(n)) & c = constante \\
O(f(n)) + O(f(n)) & = & O(f(n)) & \\
O(O(f(n)) & = & O(f(n)) & \\
O(f(n)) + O(g(n)) & = & O(max(f(n), g(n))) & \\
O(f(n))O(g(n)) & = & O(f(n)g(n)) & \\
f(n)O(g(n)) & = & O(f(n)g(n)) &
\end{array}
$$

Loop Interno

- O anel mais interno contém um comando de decisão que, por sua vez, contém apenas um comando de atribuição.
- Número de iterações $n - i$.
- O tempo combinado para executar uma vez o anel composto pelas linhas (3), (4) e (5) é $O(\max(1, 1, 1)) = O(1)$.
- Tempo gasto no anel é $O((n - i) \times 1) = O(n - i)$.

Loop Externo

- O corpo do anel mais externo contém, além do anel interno, os comandos de atribuição nas linhas (2), (6), (7) e (8). O(max(1, (n – i), 1, 1, 1)) = O(n – i).
- A linha (1) é executada n – 1 vezes
- Tempo total do programa:
  - $\sum_{i=1}^{n-1}(n - i) = \frac{n(n-1)}{2}$

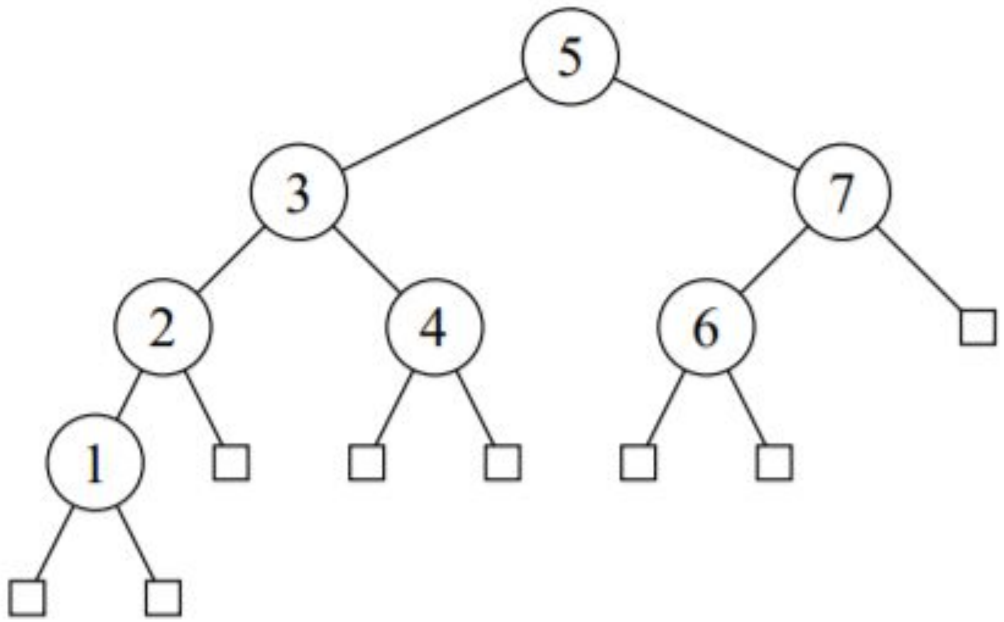
```
void p1 (int n)
{
    int i, j, k;

    for (i=0; i<n; i++)
        for (j=0; j<n; j++) {
            C[i][j]=0;
            for (k=n-1; k>=0; k--)
                C[i][j]=C[i][j]+A[i][k]*B[k][j];
        }
}
```

Recursividade

- Um procedimento que chama a si mesmo, direta ou indiretamente, é dito recursivo.
- Recursividade permite descrever algoritmos de forma mais clara e concisa, especialmente problemas recursivos por natureza ou que utilizam estruturas recursivas.
- Normalmente, as funções recursivas são divididas em duas partes
  - Chamada recursiva
  - Condição de parada

Recursividade

```
procedure Central (p: TipoApontador);
begin
  if p <> nil
  then begin
        Central (p^.Esq);
        writeln (p^.Reg.Chave);
        Central (p^.Dir);
        end;
end;
```



```c
#include<stdio.h>
int fat (int n) {
   if (n<=0)
      return (1);
   else
      return (n * fat(n-1));
}

int main(){
    int f;
    f = fat(5);
    printf("%d",f);
    return (0);
}
```

Recursividade

- A complexidade de tempo do fatorial recursivo é O(n).
  - Qual é a função de recorrência?
- Complexidade de espaço também é O(n), devido a pilha de execução.

```c
#include<stdio.h>
int fat (int n) {
    if (n==1)
        return (1);
    else
        return (n * fat(n-1));
}

int main(){
     int f;
     f = fat(6);
     printf("%d",f);
     return (0);
}
```

Recursividade

- Qual a complexidade de tempo desse algoritmo?
- Qual a complexidade de espaço?

```
#include<stdio.h>

int main(){
 int f,n;
 f = 1;
 n = 5;
 while(n > 0){
     f = f * n;
     n--;
  }
  printf("Fatorial %d", f);
}
```

Recursividade

- Assim como as estruturas de controle de repetição, recursão pode iterar infinitamente;
- Uma exigência fundamental é que a chamada recursiva a um procedimento P esteja sujeita a uma condição B;
- Wirth, 1976
  - $P \equiv if\ B\ then\ C[S_i, P]$
- Uma forma simples de garantir a terminação é associar um parâmetro *n* para *P* e chamar *P* recursivamente com *n*− *1*.
  - $P \equiv if\ n > 0\ \ then\ C[S_i, n\ \ - 1]$

Tipos de Recursividade

- Direta
- Indireta
- Cauda
- Não Cauda

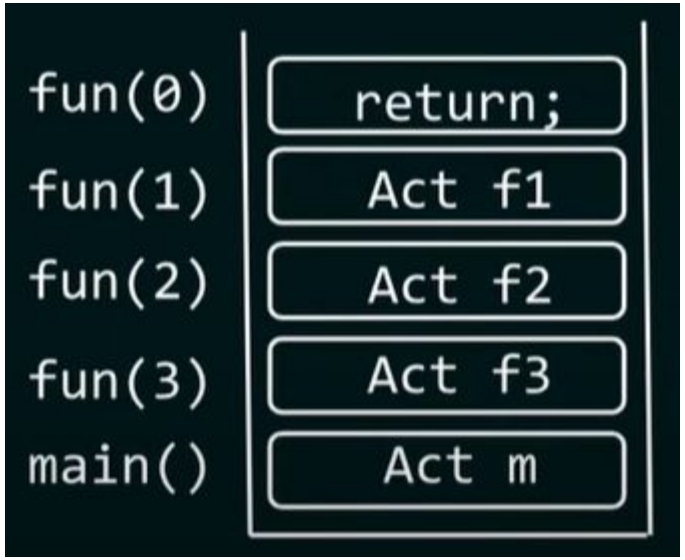
Tipos de Recursividade

- Se uma chamada recursiva em uma função é a última expressão na função, ela é chamada de **recursão em cauda (tail recursive).**
- Não é necessário armazenar o valor anterior.

```c
#include<stdio.h>
void fun(int n){
  if (n == 0)
     return;
  else
     printf(" %d ",n);
  fun(n-1);
}

int main() {
  fun(3);
  return 0;
}
```

Tipos de Recursividade

- https://pythontutor.com/visualize.html#mode=edit



Tipos de Recursividade

- Se uma chamada recursiva em uma função **não é** a última expressão na função, ela é chamada de **recursão sem cauda (non-tail recursive).**
- Depois de retornar, ainda há operação a fazer.

```c
#include<stdio.h>
void fun(int n){
  if (n == 0)
    return;
  fun(n-1);
  printf(" %d ",n);

}

int main() {
  fun(3);
  return 0;
}
```

Tipos de Recursividade

- Qual valor será impresso?
- Esta função é tail ou non-tail?

```c
#include<stdio.h>
int fun(int n){
  if (n == 1)
    return 0;
  else
    return 1 + fun(n/2);


}

int main() {
   printf(" %d ",fun(8));
  return 0;
}
```

Recursividade

- Qual será o resultado da execução do programa?
- Fibo é caudal ou não caudal?
- Qual é a função de recorrência?

```c
#include<stdio.h>
static int cont = 0;
int fibo (int n) {

   cont++;

   if (n < 2)
      return n;
    else
        return fibo(n-1)+fibo(n-2);
}

int main(){
    int aux = 5;
    printf("%d \n", fibo(aux));
    printf("Chamadas da Funcao %d\n", cont);
}
```

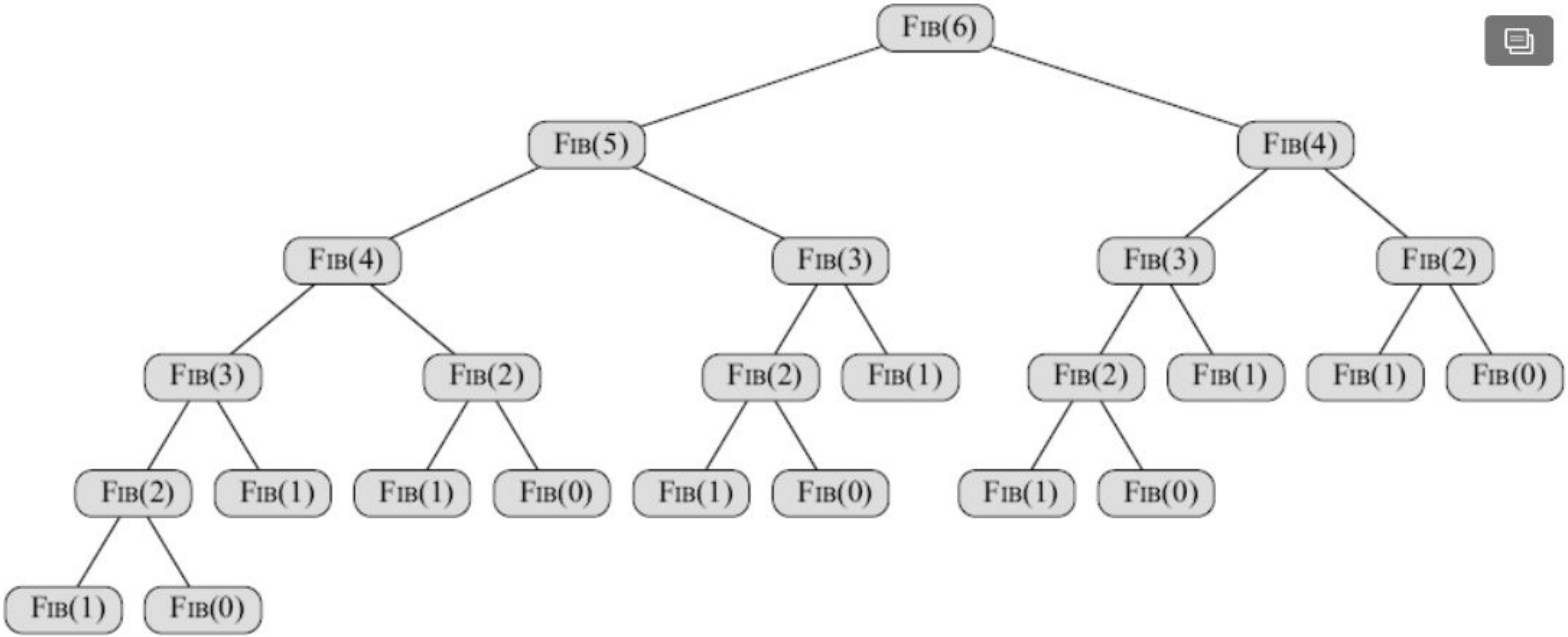


Figura 27.1 A árvore de instâncias de procedimento recursivo para o cálculo de FIB(6). Cada instância de FIB que tenha os mesmos argumentos realiza o mesmo trabalho para produzir o mesmo resultado, proporcionando um modo ineficiente, mas interessante de calcular números de Fibonacci.

Recursividade

-
  - f(0) = 0
  - f(1) = 1
  - f(n) = f(n-1)+f(n-2); para n>=2
- Formula fechada
  - $f(n) = \frac{1}{\sqrt{5}} \phi^n - (-(-\phi^{-n})); \phi = (1 + \sqrt{5})/2$
  - https://youtu.be/VftBeWtzzbE (Demonstração da fórmula fechada)
- Implemente a versão iterativa de Fibonacci. Qual é a ordem de complexidade?

Recorrência Linear de Primeira Ordem

- Formato
  - $S(n) = C*S(n-1) + g(n)$
- Fórmula
  - $S(n) = c^{n-1}\, a_1 + \sum_{i=2}^{n} c^{n-i}\, g(i)$
- Exemplo
  - $F(1) = 2;$
  - $F(n) = 2f(n-1)$
  - $C = 2;\ g(n) = 0;\ a_1=2;$

- $F(n) = c^{n-1}\, a_1 + \sum_{i=2}^{n} c^{n-i}\, g(i)$
- $F(n) = 2^{n-1}2 + \sum_{i=2}^{n} 2^{n-i}\, 0$
- $F(n) = 2^n$

Recorrência Linear de Segunda Ordem

- O n-ésimo termo depende dos dois termos anteriores. Tem a seguinte forma:
  - $S(n) = C_1S(n-1) + C_2s(n-2)$
- São necessários 2 casos básicos.
- Fórmula
  - $S(n) = pr_1^{n-1} + qr_2^{n-1}$
- r_1 e r_2 são raízes da equação característica
  - $t^2 - c_1t - c_2$
- Cálculo de p e q
  - $p + q = S(1)$
  - $pr_1 + qr_2 = S(2)$

- Exemplo
  - $S(n) = 2S(n-1) + 3S(n-2)$
  - $S(1) = 3$
  - $S(2) = 1$
  - Equação Característica
    - $T^2 – 2T – 3$
    - Raízes:
      - $r_1 = 3$ e $r_2 = -1$
  - Sistema de equações
    - $p+q = S(1)$ $p+q=3$
    - $pr_1 + qr_2 = S(2)$ $3p – q = 1$
    - $p =1$ e $q = 2$
  - $Sn = pr_1^{n-1} + qr_2^{n-1}$ $Sn = 1*3^{n-1} + 2*(-1)^{n-1}$

Recorrência Linear de Segunda Ordem

- Se as equações características tiveram raízes iguais?
  - p = S(1)
  - pr + qr = S(2)
- Fórmula
  - $S(n) = pr^{n-1} + q(n-1)r^{n-1}$

Teorema Mestre

- $T(n) = a \times T\left(\frac{n}{b}\right) + f(n)$
  - A ➔ número de subproblemas (a>=1)
  - B ➔ Tamanho do subproblema (b > 1)
  - F(n) ➔ não negativa
- Neste caso, não estamos achando a forma fechada da recorrência, mas sim seu comportamento assintótico

Teorema Mestre

- Se $f(n) = O(n^{\log_b a-\varepsilon})$ para alguma constante $\varepsilon > 0$ então $T(n) = \theta(n^{\log_b a})$
  - $f(n) < n^{\log_b a} \rightarrow T(n) = \theta(n^{\log_b a})$
- Se $f(n) = \theta(n^{\log_b a})$ para alguma constante $\varepsilon > 0$ então $T(n) = \theta(n^{\log_b a}\ f(n))$
  - $f(n) = n^{\log_b a} \rightarrow T(n) = \theta(f(n) * \log_b n)$
- Se $f(n) = \Omega(n^{\log_b a+\varepsilon})$ para alguma constante $\varepsilon > 0$ então $T(n) = \theta(f(n))$
  - $f(n) > n^{\log_b a} \rightarrow T(n) = \theta(f(n))$

Teorema Mestre

- Exemplo
  - $T(n) = 2*T(n/2) + 3$
  - $T(n) = 2*T(n/2) + n$
  - $T(n) = 2*T(n/2) + n^2$