

CT-234



Estruturas de Dados,
Análise de Algoritmos e
Complexidade Estrutural

Carlos Alberto Alonso Sanches

CT-234



2) Algoritmos recursivos

Indução matemática, recursão, recorrências

Indução matemática

- Uma proposição $P(X)$ pode ser provada por *indução matemática* (ou *indução finita*) do seguinte modo:
 - **Base:** Comprovamos que P é verdadeira para os casos básicos ($X=0$ ou $X=1$, por exemplo).
 - **Hipótese indutiva:** Supomos que P seja verdadeira para o caso genérico $X=N$.
 - **Passo indutivo:** Demonstramos que P também é verdadeira para o caso $X=N+1$.
- Ideia: como a proposição vale para o caso inicial e o passo é correto, então essa proposição também será válida para todos os casos subsequentes.

Uma imagem

- *Proposição:* numa sequência de peças de dominó que estejam em pé, suficientemente próximas entre si, se a primeira caiu então todas caíram.
- *Prova por indução:*
 - **Base:** A primeira peça caiu (por definição).
 - **Hipótese indutiva:** Supomos que a n -ésima tenha caído.
 - **Passo indutivo:** Como a n -ésima peça caiu e ela está suficientemente perto da seguinte, então a $(n+1)$ -ésima peça também terá caído.

Um exemplo

- Demonstre que, para $x > 1$ e $n > 0$, $x^n - 1$ é divisível por $x - 1$.
- Prova por indução em n :
 - **Base:** Para $n=1$, $x^1 - 1$ é divisível por $x - 1$.
 - **Hipótese indutiva:** Para um valor qualquer $n > 1$, supomos que $x^n - 1$ seja divisível por $x - 1$, para todo $x > 0$ natural.
 - **Passo indutivo:**
 - Sabemos que $x^{n+1} - 1 = x^{n+1} - x + x - 1 = x(x^n - 1) + (x - 1)$.
 - Pela hipótese de indução, a primeira parcela é divisível por $x - 1$.
 - Como sabemos que a segunda também é, o passo está provado.

Exercícios



- Demonstre por indução matemática:
 - $n^3 + 2n$ é divisível por 3, para $n \geq 0$.
 - $2^0 + 2^1 + 2^2 + \dots + 2^n = 2^{n+1} - 1$, para $n \geq 0$.
 - $2^{-1} + 2^{-2} + 2^{-3} + \dots + 2^{-n} < 1$, para $n > 0$.
 - $n^2 < 2^n$, para $n > 4$.
 - A representação binária de um número inteiro $n > 0$ tem exatamente $\lfloor \lg n \rfloor + 1$ bits.

Definições recursivas ou indutivas



- Em uma *definição recursiva*, uma classe de objetos relacionados é definida em termos desses próprios objetos.
- Uma definição recursiva envolve:
 - Uma **base**, onde um ou mais objetos elementares são definidos.
 - Um **passo indutivo**, onde objetos subsequentes são definidos em termos de objetos já conhecidos.

Um exemplo



- Definição recursiva dos números naturais:
 - **Base:** o número 0 está em \mathbb{N} .
 - **Passo indutivo:** se n está em \mathbb{N} , então $n + 1$ também está.
- O conjunto dos números naturais é o menor conjunto que satisfaz as condições acima.

Outro exemplo



- As expressões numéricas são comumente definidas de forma recursiva:
 - **Base:** Todos os operandos atômicos (números, variáveis, etc.) são expressões numéricas.
 - **Passo indutivo:** Se E_1 e E_2 são expressões numéricas então $(E_1 + E_2)$, $(E_1 - E_2)$, $(E_1 \cdot E_2)$, (E_1 / E_2) e $(-E_1)$ também são.

Algoritmos recursivos

- Recursão (ou recursividade) é uma técnica de programação no qual um procedimento (função, método, etc.) pode chamar a si mesmo.
- Algoritmos recursivos possuem uma clara analogia com o método indutivo:
 - **Base:** Uma entrada elementar, que pode ser resolvida diretamente.
 - **Parte indutiva:** Chamadas a si mesmo, mas com entradas mais simples.
- A ideia é aproveitar a solução de um ou mais subproblemas para resolver todo o problema.

Um exemplo clássico

- Algoritmo recursivo para o cálculo de fatorial:

$$0! = 1! = 1$$
$$n! = n \cdot (n-1)!, n > 1$$

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

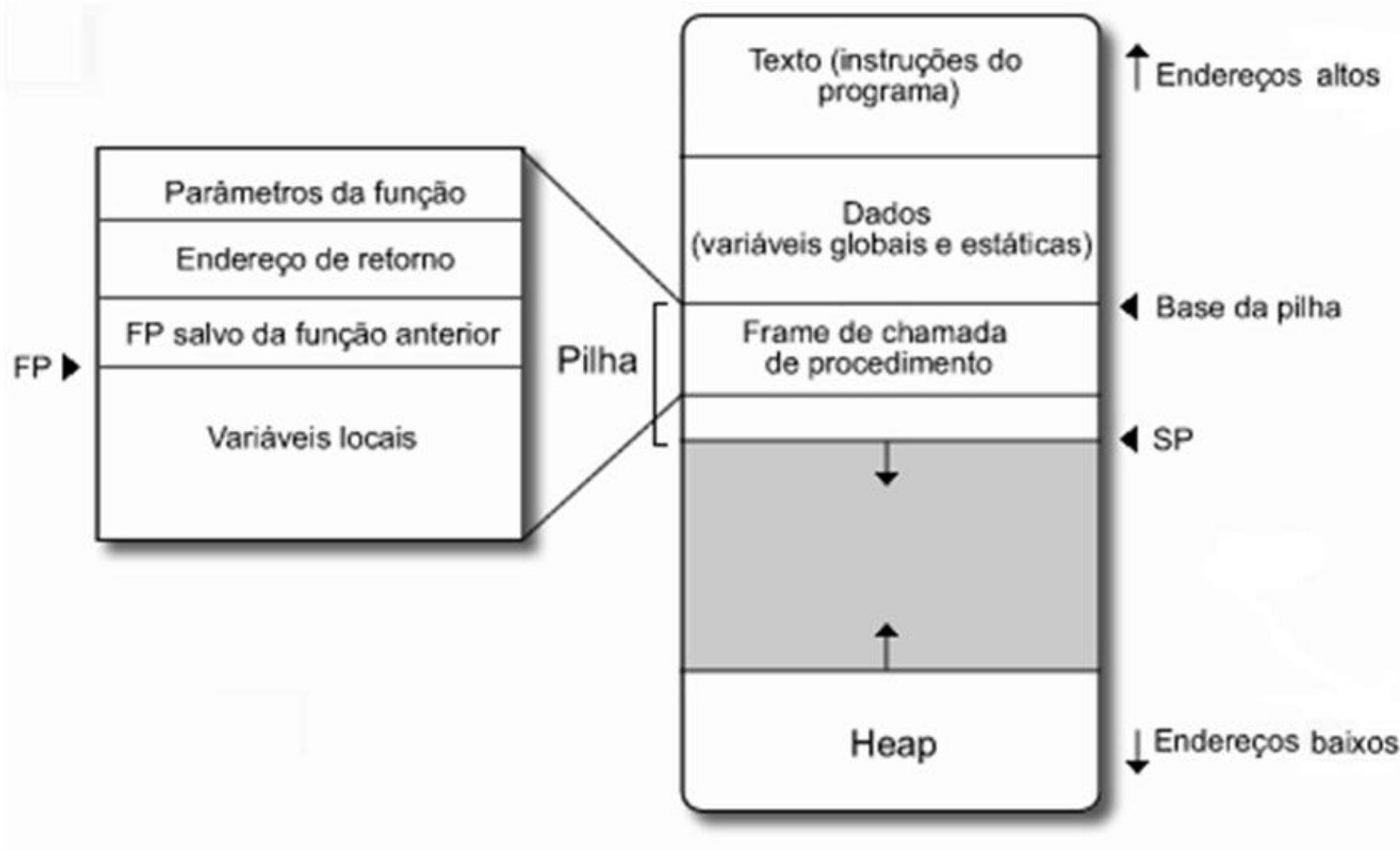
- Execução de `fat(4)`:

call ↓		
<code>fat(4)</code>		return 24 ↑
call ↓		
<code>fat(3)</code>		return 6 ↑
call ↓		
<code>fat(2)</code>		return 2 ↑
call ↓		
<code>fat(1)</code>	return 1 ↑	

Mecanismo da recursão

- Durante a execução de um programa, quando um procedimento é chamado, é preciso guardar o contexto atual de processamento (valores de parâmetros e variáveis locais, endereço de retorno, etc.) para que depois seja possível recomeçar de onde se parou.
- Deseja-se que o último procedimento interrompido seja o primeiro a recomeçar a sua execução. Por isso, o sistema operacional utiliza uma *pilha de execução*, alocada na memória.
- Portanto, os algoritmos recursivos poderiam ser escritos de uma forma iterativa: basta utilizar uma pilha explícita, que simule o gerenciamento realizado pelo sistema operacional.

Pilha de execução



Alocação estática versus dinâmica

- Variáveis estáticas: têm endereço determinado em *tempo de compilação*
 - São previstas antes da compilação do programa
 - Ocupam uma área de dados do programa, determinada na compilação
 - Existem durante toda a execução do programa
- Variáveis dinâmicas: têm endereço determinado em *tempo de execução*
 - São alocadas de uma área extra da memória, chamada *heap*, através de funções específicas (`malloc`, `new`, etc.)
 - Sua eventual existência depende do programa, e seu endereço precisa ser armazenado em um ponteiro
 - Exigem uma política de administração da memória

Análise da complexidade de tempo

- Seja $T(n)$ o tempo de pior caso de fat(n) :
 - Base: $T(0) = T(1) = a$
 - Parte indutiva: $T(n) = T(n-1) + b$, $n > 1$
- Cálculos:
 - $T(2) = T(1) + b = a + b$
 - $T(3) = T(2) + b = a + 2b$
 - $T(4) = T(3) + b = a + 3b$
 - Generalizando: $T(n) = a + (n-1)b$
 - Portanto: $T(n) = \Theta(n)$

Um algoritmo iterativo equivalente

- Costuma-se calcular o fatorial de um número natural n da seguinte maneira:

```
int fat(int n) {  
    int f = 1;  
    while (n > 0)  
        f *= n--;  
    return f;  
}
```

- É fácil constatar que o tempo de pior caso desse algoritmo iterativo é também $\Theta(n)$, ou seja, tem a mesma complexidade de tempo que a sua versão recursiva.
- No entanto, é mais rápido... Por quê?
- E com relação às complexidades de espaço?

Exercício

- O programa recursivo abaixo calcula a soma dos números naturais entre 1 e n, onde $n > 0$:

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

- Simule a sua execução para a entrada $n = 5$, mostrando a pilha de chamadas.

Outro exemplo clássico

- Algoritmo recursivo para encontrar o n-ésimo número de Fibonacci:

$$F_0 = F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2}, \quad n > 1$$

```
int Fib(int n) {  
    if (n==0 || n==1) return 1;  
    return Fib(n-1) + Fib(n-2);  
}
```

- Equivalente iterativo:

```
int Fib(int n) {  
    if (n==0 || n==1) return 1;  
    int f1=1, f2=1, f3;  
    for (int i=2; i<=n; i++) {  
        f3 = f1 + f2;  
        f1 = f2;  
        f2 = f3;  
    }  
    return f3;  
}
```

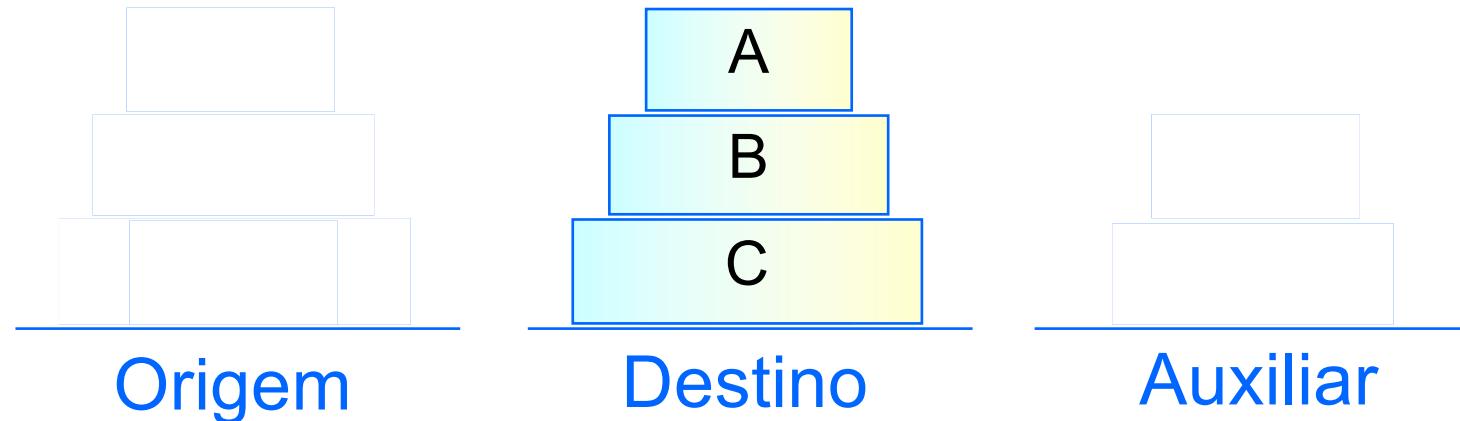
Será que têm a mesma complexidade de tempo?

Análise da complexidade de tempo

- É fácil constatar que o algoritmo iterativo gasta tempo $\Theta(n)$.
- Seja $T(n)$ o tempo de pior caso do algoritmo recursivo:
 - **Base:** $T(1) = a$
 - **Parte indutiva:** $T(n) = T(n-1) + T(n-2) + b$, $n > 1$
- Como $T(n-1) > T(n-2)$, sabemos que $T(n) > 2T(n-2)$
- Repetindo:
 - $T(n) > 2T(n-2) > 2(2T(n-2-2)) = 4T(n-4)$
 - $T(n) > 4T(n-4) > 4(2T(n-4-2)) = 8T(n-6)$
 - Generalizando: $T(n) > 2^i T(n-2i)$, para $i > 0$
 - Consideremos o caso $n-2i=1$, ou seja, $i=(n-1)/2$:
 - $T(n) > 2^{(n-1)/2} T(1)$
 - $T(n) = \Omega(2^{n/2})$

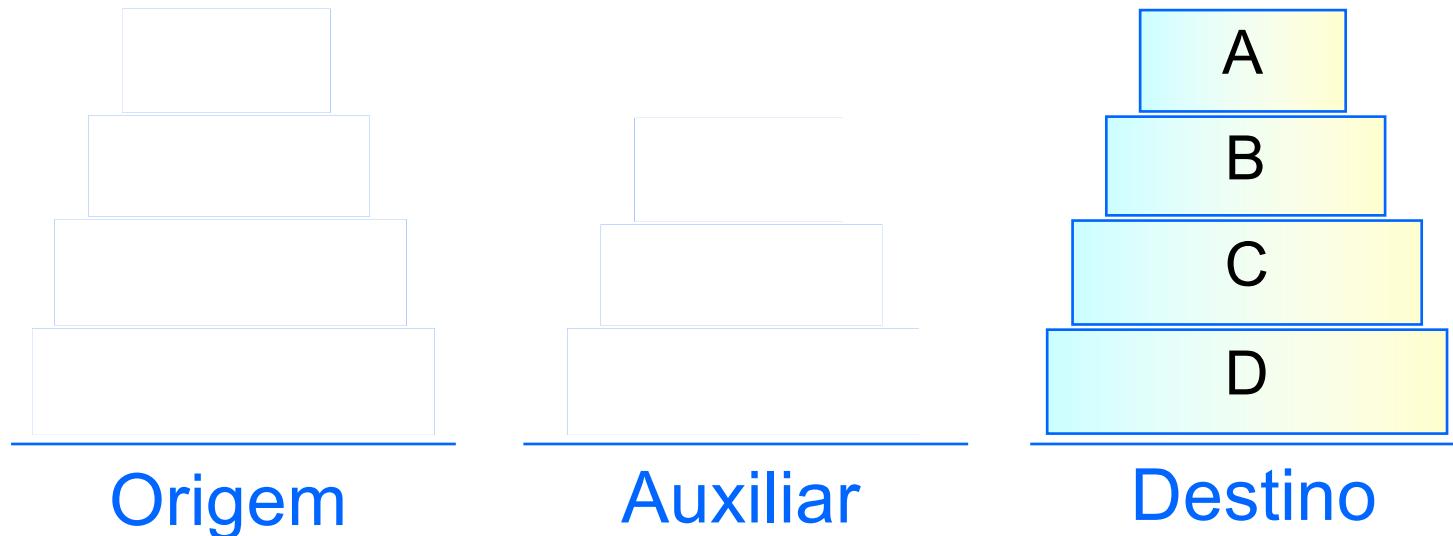
Torres de Hanoi

- É um exemplo clássico da aplicabilidade da recursão.
- Deseja-se mover n discos, um de cada vez, de uma torre de origem para outra de destino, usando uma terceira auxiliar, sem nunca colocar um disco maior sobre outro menor.
- Exemplo para 3 discos:



Torres de Hanoi

- E se, ao invés de 3 discos, fossem 4?
- Vamos usar o que já sabemos fazer, ou seja, mover 3 discos seguindo as regras do jogo.
- Em seguida, o restante fica fácil...



Solução recursiva



- Mova os $n-1$ discos de cima de *Origem* para *Auxiliar* (recursivamente)
- Mova o maior disco de *Origem* para *Destino*
- Mova os $n-1$ discos de *Auxiliar* para *Destino* (recursivamente)

Solução

- Mover n discos da torre org para a torre $dest$, utilizando aux como auxiliar:

```
void hanoi(int n, org, dest, aux) {  
    if (n==1)  
        print("Mova de ", org, "para ", dest);  
    else {  
        hanoi(n-1, org, aux, dest);  
        print("Mova de ", org, "para ", dest);  
        hanoi(n-1, aux, dest, org);  
    }  
}
```

- Complexidade de tempo:

- $T(1) = a$
- $T(n) = 2T(n-1) + a, n > 1$

Complexidade de tempo

- Desenvolvendo $T(n) = 2T(n-1) + a$:
 - $T(n) = 2(2T(n-2) + a) + a = 2^2T(n-2) + 2^1a + a$
 - $T(n) = 2^3T(n-3) + 2^2a + 2^1a + a$
 - $T(n) = 2^4T(n-4) + 2^3a + 2^2a + 2^1a + a$
- Generalizando: $T(n) = 2^i T(n-i) + 2^{i-1}a + \dots + 2^0a$, $i > 0$
- Para $n-i=1$, ou seja, $i=n-1$:
 - $T(n) = 2^{n-1}a + 2^{n-2}a + \dots + 2^0a$
 - $T(n) = (2^n - 1)a = \Theta(2^n)$

Vantagens *versus* desvantagens



- A recursão deve ser utilizada com critério: não há regras gerais.
- Usualmente, é menos eficiente que o seu equivalente iterativo (devido ao *overhead* da pilha de execução), mas essa diferença nem sempre é decisiva.
 - Em determinados compiladores, há implementações otimizadas para chamadas recursivas no final do código da função (*tail recursion*). Neste caso, é possível evitar o crescimento da pilha de execução.
- A sua transformação em uma versão iterativa nem sempre é trivial.
- Muitas vezes, é vantajosa em clareza, legibilidade e simplicidade de código.

Exercícios



- Resolva com algoritmos recursivos:
 - Imprimir os n primeiros números naturais em ordem crescente.
 - Idem, mas em ordem decrescente.
 - Encontrar o valor máximo presente em um vetor.
 - Verificar se um determinado valor está ou não presente em um vetor.
 - Calcular a soma dos valores armazenados em um vetor.
 - Inverter a ordem dos valores armazenados em um vetor.

Outros exercícios

- Dado um número natural, imprimir recursivamente a sua representação binária.
- (*Busca binária*) Dado um vetor ordenado de tamanho n , verificar se um determinado elemento está ou não presente.
- (*Gray code*) Gerar recursivamente todas as representações de n bits, de tal modo que, entre duas sucessivas, haja um único bit distinto.
- Torres de Saigon: idem a Hanoi, mas com 2 torres auxiliares.
- Pesquisar *análise sintática recursiva*.

Recorrências

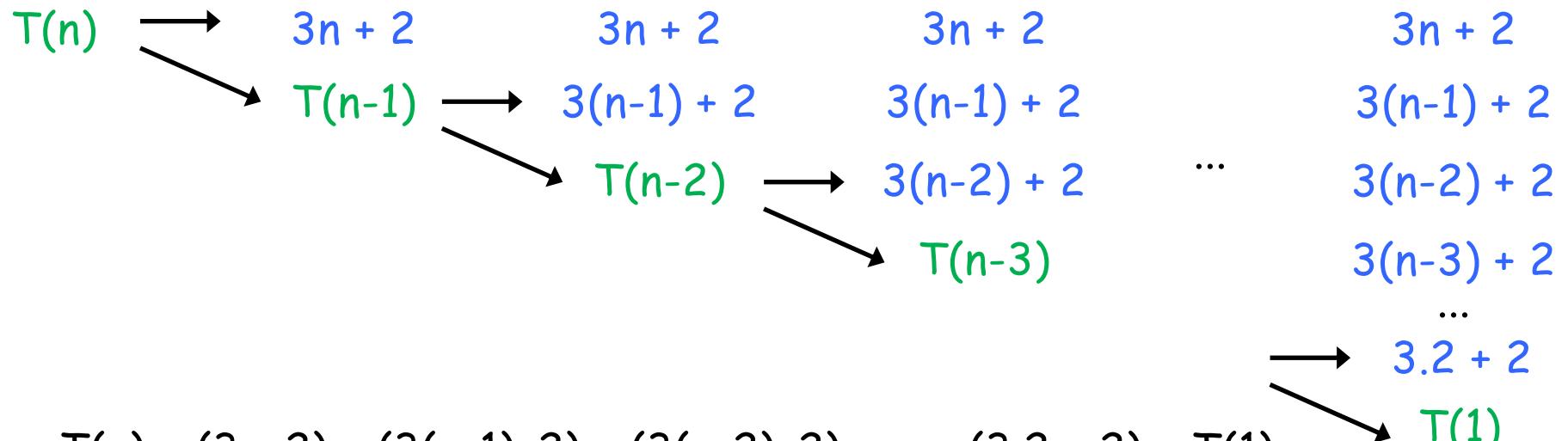


- Recorrência é uma equação ou desigualdade que descreve uma função em termos de si mesma, mas com entradas menores.
- Como a complexidade de tempo de um algoritmo recursivo é expressa através de uma recorrência, é preciso determiná-la efetivamente.
- “Resolvemos” uma recorrência quando conseguimos eliminar as referências a si mesma.
- Melhores técnicas: uso de árvore de recorrência, iterações e substituição de variáveis.

Exemplo 1

$$T(1) = 1$$

$$T(n) = T(n-1) + 3n + 2, n > 1$$



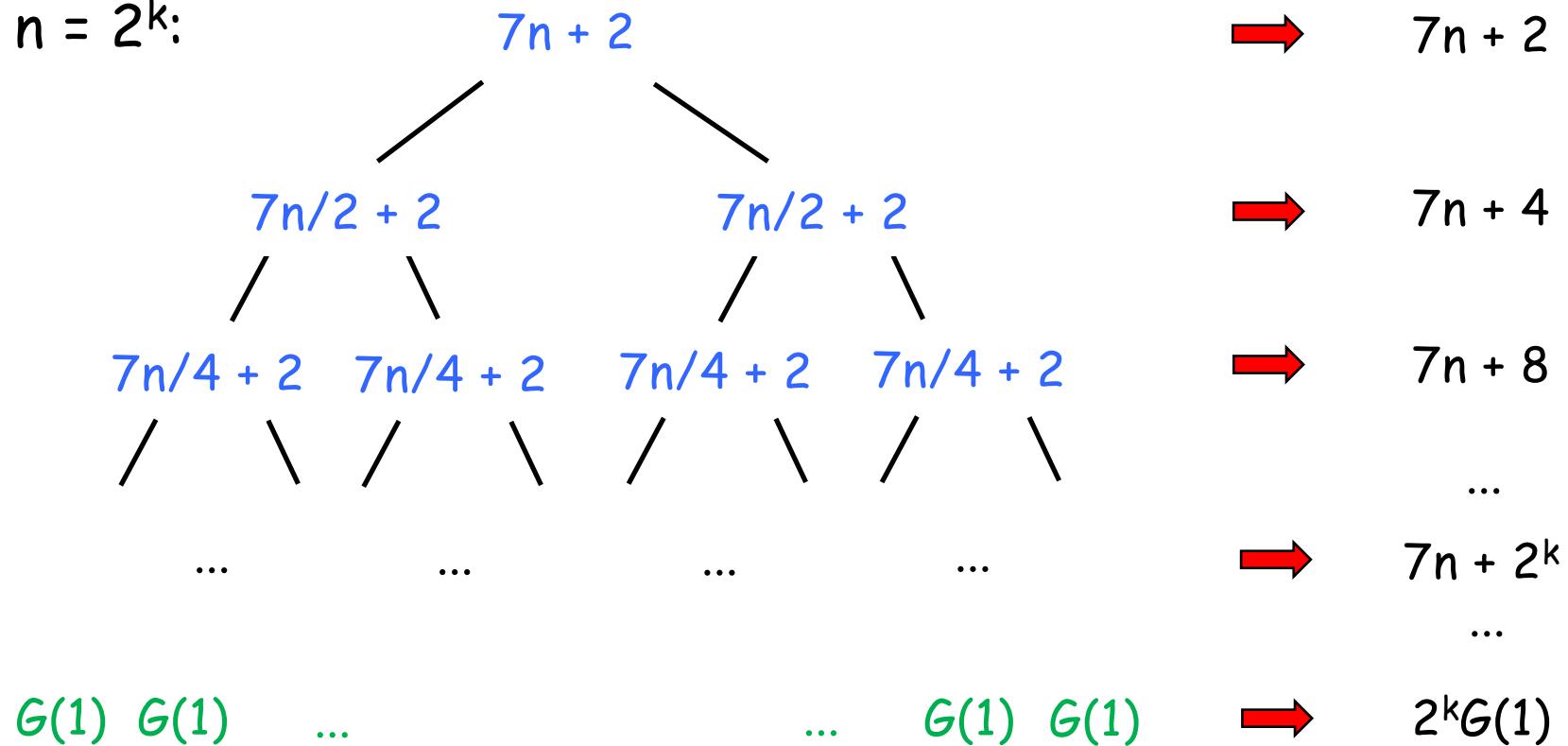
- $T(n) = (3n+2) + (3(n-1)+2) + (3(n-2)+2) + \dots + (3.2 + 2) + T(1)$
- $T(n) = 3(n + n-1 + n-2 + \dots + 2) + 2(n-1) + 1$
- $T(n) = 3(n+2)(n-1)/2 + 2n - 1$
- $T(n) = 3n^2/2 + 7n/2 - 4$

Exemplo 2.a

$$G(1) = 1$$

$$G(n) = 2G(n/2) + 7n + 2, \text{ onde } n=2, 4, \dots, 2^i, \dots$$

Com $n = 2^k$:



Exemplo 2.a (continuação)



- $G(n) = (7n+2) + (7n+4) + \dots + (7n+2^k) + 2^k$, onde $k=\lg n$
- $G(n) = 7nk + (2+4+\dots+2^k) + 2^k$
- $G(n) = 7n.\lg n + (2^{k+1} - 2) + 2^k$
- $G(n) = 7n.\lg n + 3n - 2$

Exemplo 2.b

- Novamente, $G(1) = 1$ e $G(n) = 2G(n/2) + 7n + 2$
- Por indução, pode-se demonstrar que $G(n) \leq 9n \lg n$ para $n = 2, 4, \dots, 2^i$:
 - **Base:** Para $n=2$, $G(2) = 2G(1) + 7.2 + 2 = 2 + 14 + 2 = 18$. Portanto, $G(2) \leq 9.2 \lg 2$.
 - **Passo indutivo:**
 - $G(n) = 2G(n/2) + 7n + 2$
 - $G(n) \leq 2.9(n/2) \lg (n/2) + 7n + 2$ (h.i. vale porque $2 \leq n/2 < n$)
 - $G(n) \leq 9n(\lg n - 1) + 7n + 2$
 - $G(n) \leq 9n \lg n - 2n + 2$
 - $G(n) < 9n \lg n$, pois $n > 2$.

Exemplo 2.c

- Caso se deseje apenas a ordem, basta considerar $G(1) = 1$ e $G(n) = 2G(n/2) + \Theta(n)$ e iterar substituições:
 - $G(n) = 2(2G(n/4) + \Theta(n/2)) + \Theta(n) = 4G(n/4) + 2\Theta(n)$
 - $G(n) = 4(2G(n/8) + \Theta(n/4)) + 2\Theta(n) = 8G(n/8) + 3\Theta(n)$
 - Generalizando: $G(n) = 2^k G(n/2^k) + k\Theta(n)$
 - Para $n = 2^k$, ou seja, $k = \lg n$:
 - $G(n) = nG(1) + \lg n \cdot \Theta(n)$
 - $G(n) = \Theta(n \log n)$

Exemplo 3

$$T(1) = 1$$

$$T(n) = 2T(\lfloor n^{1/2} \rfloor) + \lg n, n > 1$$

- Vamos considerar apenas o caso em que n é potência de 2
- Troca de variáveis: $n = 2^m$ com m par, ou seja, $m = \lg n$
- $T(2^m) = 2T(2^{m/2}) + m$
- Seja $S(m) = T(2^m)$
- $S(m) = 2S(m/2) + m$
- Pelo exemplo 2, sabemos que $S(m) = \Theta(m \cdot \log m)$ quando $m = 2^k$
- Portanto, $T(n) = T(2^m) = S(m) = \Theta(m \cdot \log m) = \Theta(\log n \cdot (\log \log n))$

Exercícios



- Resolva as recorrências:

- $T(1) = 1$ e $T(n) = T(n-1) + 1$, $n > 1$.
- $T(1) = 1$ e $T(n) = T(n-1) + n$, $n > 1$.
- $T(0) = 0$, $T(1) = 1$ e $T(n) = T(n-2) + 2n + 1$, $n > 1$.
- $T(1) = 1$ e $T(n) = T(\lfloor n/2 \rfloor) + n$, $n > 1$.
- $T(1) = 1$ e $T(n) = T(\lfloor n/3 \rfloor) + T(\lfloor n/4 \rfloor) + kn$, $n > 1$.