05 – Análise de Algoritmos (parte 1)

SCC5900 - Projeto de Algoritmos

Material gentilmente cedido pelo Prof. Moacir Ponti Jr.

Modificado por Joao Batista Neto

www.icmc.usp.br/~jbatista

Instituto de Ciências Matemáticas e de Computação – USP

2011/1

Sumário

- Considere a função abaixo que realiza exponenciação a^b

```
int exp1(int a, int b) {
    int res = 1;
    while (b > 0) {
       res *= a;
       b -= 1;
    }
    return res;
}
```

Analisando funções para exponenciação

- Considere a função abaixo que realiza exponenciação a^b

```
int exp1(int a, int b) {
    int res = 1;
    while (b > 0) {
       res *= a;
       b -= 1;
    }
    return res;
}
```

- Qual a complexidade dessa função?

Analisando funções para exponenciação

- Considere a função abaixo que realiza exponenciação a^b de maneira recursiva

```
int exp2(int a, int b) {
    if (b == 1)
       return a;
    else
       return a*exp2(a, b-1);
}
```

Analisando funções para exponenciação

- Considere a função abaixo que realiza exponenciação a^b de maneira recursiva

```
int exp2(int a, int b) {
    if (b == 1)
        return a;
    else
        return a*exp2(a, b-1);
}
```

- Qual a complexidade dessa função?

Analisando funções para exponenciação

- Considere a função abaixo que realiza exponenciação a^b de maneira recursiva

```
int exp2(int a, int b) {
    if (b == 1)
       return a;
    else
       return a*exp2(a, b-1);
}
```

- Qual a complexidade dessa função?
- Apesar de funcionar como uma repetição, a resolução não é tão trivial assim!

Sumário

Recorrências

- Para analisar o consumo de tempo de um algoritmo recursivo é necessário **resolver uma recorrência**.
- Uma recorrência é uma expressão que dá o valor de uma função em termos dos valores "anteriores" da mesma função.

(FEOFILOFF, 2010)

Recorrências

- Para analisar o consumo de tempo de um algoritmo recursivo é necessário **resolver uma recorrência**.
- Uma recorrência é uma expressão que dá o valor de uma função em termos dos valores "anteriores" da mesma função.
- Exemplo:

$$F(n) = F(n-1) + 3n + 2 \quad (1)$$

é uma recorrência que dá o valor de $F(n)$ em termos de $F(n-1)$.

(FEOFILOFF, 2010)

Recorrências

- Para analisar o consumo de tempo de um algoritmo recursivo é necessário **resolver uma recorrência**.
- Uma recorrência é uma expressão que dá o valor de uma função em termos dos valores "anteriores" da mesma função.
- Exemplo:

$$F(n) = F(n-1) + 3n + 2 \tag{1}$$

  é uma recorrência que dá o valor de $F(n)$ em termos de $F(n-1)$.
- Uma recorrência pode ser vista como um algoritmo recursivo que calcula uma função a partir de um "valor inicial"
- Mais quais os valores de n?

(FEOFILOFF, 2010)

- Para analisar o consumo de tempo de um algoritmo recursivo é necessário **resolver uma recorrência**.
- Uma recorrência é uma expressão que dá o valor de uma função em termos dos valores "anteriores" da mesma função.
- Exemplo:

$$F(n) = F(n-1) + 3n + 2 \tag{1}$$

  é uma recorrência que dá o valor de $F(n)$ em termos de $F(n-1)$.
- Uma recorrência pode ser vista como um algoritmo recursivo que calcula uma função a partir de um "valor inicial"
- Mais quais os valores de n?
- Podemos supor, por exemplo, que $n = 2, 3, 4, 5, \cdots$, e que $F(1) = 1$ como valor inicial.

(FEOFILOFF, 2010)

Recorrências

- Uma recorrência é satisfeita por muitas funções diferentes — uma para cada valor inicial.
- As funções no entanto são, em geral, do mesmo "tipo".
- Interessam geralmente funções definidas nos números naturais, mas podem ser definidas em outros conjuntos: naturais maiores que 99, as potências inteiras de 2, potências inteiras de $1\frac{1}{2}$, etc.

(FEOFILOFF, 2010)

Recorrências

- Uma recorrência é satisfeita por muitas funções diferentes — uma para cada valor inicial.
- As funções no entanto são, em geral, do mesmo "tipo".
- Interessam geralmente funções definidas nos números naturais, mas podem ser definidas em outros conjuntos: naturais maiores que 99, as potências inteiras de 2, potências inteiras de $1\frac{1}{2}$, etc.

Resolver uma recorrência é ...

- ...encontrar uma **fórmula fechada** que dê o valor diretamente em termos de seu parâmetro.
  - Geralmente, uma combinação de polinômios, quocientes de polinômios, logaritmos, exponenciais, etc.

(FEOFILOFF, 2010)

Recorrências

- Considere:

$$F(n) = F(n-1) + 3n + 2 \tag{2}$$

- E suponha que $n \in \{2, 3, 4, \cdots\}$
- Há uma infinidade de funções F que satisfazem a recorrência com valor inicial $F(1)$ diferentes ($F(1) = 1$, $F(1) = 10$, etc.).

(FEOFILOFF, 2010)

- Considere:

$$F(n) = F(n-1) + 3n + 2 \tag{2}$$

- E suponha que $n \in \{2, 3, 4, \cdots\}$
- Há uma infinidade de funções F que satisfazem a recorrência com valor inicial $F(1)$ diferentes ($F(1) = 1$, $F(1) = 10$, etc.).
  - De modo mais geral, é evidente que para cada número i existe uma (e apenas uma) função F definida sobre $\{1, 2, 3, 4, \cdots\}$ que tem valor inicial $F(1) = i$ e satisfaz a recorrência acima.

(FEOFILOFF, 2010)

Recorrências

- Considere:

$$F(n) = F(n-1) + 3n + 2 \tag{2}$$

- E suponha que $n \in \{2, 3, 4, \cdots\}$
- Há uma infinidade de funções F que satisfazem a recorrência com valor inicial $F(1)$ diferentes ($F(1) = 1$, $F(1) = 10$, etc.).
    - De modo mais geral, é evidente que para cada número i existe uma (e apenas uma) função F definida sobre $\{1, 2, 3, 4, \cdots\}$ que tem valor inicial $F(1) = i$ e satisfaz a recorrência acima.
- Gostaríamos de obter uma fórmula fechada para a recorrência. Como fazer?

(FEOFILOFF, 2010)

Analisando funções para exponenciação

- Considere (novamente) a função abaixo que realiza exponenciação a^b de maneira recursiva

```
   int exp2(int a, int b) {
1      if (b == 1)
2          return a;
       else
3          return a*exp2(a, b-1);
   }
```

- Seja $T(b)$ uma função de complexidade onde b é o número de vezes que termos que multiplicar a base para obter a exponenciação.
  - O custo das linhas 1 e 2 é $O(1)$.

Analisando funções para exponenciação

- Considere (novamente) a função abaixo que realiza exponenciação a^b de maneira recursiva

```
  int exp2(int a, int b) {
1     if (b == 1)
2         return a;
      else
3         return a*exp2(a, b-1);
  }
```

- Seja $T(b)$ uma função de complexidade onde b é o número de vezes que termos que multiplicar a base para obter a exponenciação.
  - O custo das linhas 1 e 2 é $O(1)$.
  - Quantas vezes a linha 3 será executada? — quantas chamadas recursivas serão necessárias?

Analisando funções para exponenciação

```
    int exp2(int a, int b) {
1       if (b == 1)
2           return a;
        else
3           return a*exp2(a, b-1);
    }
```

- Podemos encontrar uma relação de recorrência para $T(b)$: temos 1 comparação, 1 multiplicação e 1 subtração e 1 retorno:

$$T(b) = 4 + T(b-1) \tag{3}$$

Analisando funções para exponenciação

```
    int exp2(int a, int b) {
1       if (b == 1)
2           return a;
        else
3           return a*exp2(a, b-1);
    }
```

- Podemos encontrar uma relação de recorrência para $T(b)$: temos 1 comparação, 1 multiplicação e 1 subtração e 1 retorno:

$$T(b) = 4 + T(b-1) \tag{3}$$

- Isso significa que temos 3 operações mais uma chamada recursiva que deverá processar uma entrada de tamanho $b-1$.

Analisando funções para exponenciação

$$T(b) = 4 + T(b-1)$$
$$T(b) = 4 + (4 + T(b-2))$$
$$\dots$$
$$T(b) = 4k + T(b-k)$$

Analisando funções para exponenciação

$$T(b) = 4 + T(b-1)$$
$$T(b) = 4 + (4 + T(b-2))$$
$$\dots$$
$$T(b) = 4k + T(b-k)$$

- Quando termina?

Analisando funções para exponenciação

$$
\begin{aligned}
T(b) &= 4 + T(b-1) \\
T(b) &= 4 + (4 + T(b-2)) \\
&\dots \\
T(b) &= 4k + T(b-k)
\end{aligned}
$$

- Quando termina?
- Quando alcanço o caso base, ou seja $b - k = 1$, ou $k = b - 1$

Analisando funções para exponenciação

$$
\begin{aligned}
T(b) &= 4 + T(b-1) \\
T(b) &= 4 + (4 + T(b-2)) \\
&\dots \\
T(b) &= 4k + T(b-k)
\end{aligned}
$$

- Quando termina?
- Quando alcanço o caso base, ou seja $b - k = 1$, ou $k = b - 1$
- "Abusando" da matemática e substituindo:

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

Analisando funções para exponenciação

```
  int exp2(int a, int b) {
1     if (b == 1)
2         return a;
      else
3         return a*exp2(a, b-1);
  }
```

- Qual seria então a complexidade de exp2?

```
    int exp2(int a, int b) {
1       if (b == 1)
2           return a;
        else
3           return a*exp2(a, b-1);
    }
```

- Qual seria então a complexidade de exp2?
- Como $T(b) = 4b - 2$, podemos dizer que é $O(b)$, ou seja, linear.
- Observação: aqui usamos b para facilitar o entendimento, mas quer dizer também o tamanho da entrada, nesse caso relativo ao tamanho do expoente.

Uma função para exponenciação que não é linear

- Podemos melhorar a performance do algoritmo que calcula a^b utilizando propriedades matemáticas:

Uma função para exponenciação que não é linear

- Podemos melhorar a performance do algoritmo que calcula a^b utilizando propriedades matemáticas:

1. Se b for par, então $a^b = (a \cdot a)^{b/2}$
   - perceba que se b for par, reduzi o problema pela metade $(b/2)$.

Uma função para exponenciação que não é linear

- Podemos melhorar a performance do algoritmo que calcula a^b utilizando propriedades matemáticas:

1. Se b for par, então $a^b = (a \cdot a)^{b/2}$
   - perceba que se b for par, reduzi o problema pela metade $(b/2)$.
2. Se b for ímpar, então $a^b = a \cdot (a^{b-1})$

Uma função para exponenciação que não é linear

- Podemos melhorar a performance do algoritmo que calcula a^b utilizando propriedades matemáticas:

1. Se b for par, então $a^b = (a \cdot a)^{b/2}$
   - perceba que se b for par, reduzi o problema pela metade $(b/2)$.
2. Se b for ímpar, então $a^b = a \cdot (a^{b-1})$
   - perceba que mesmo no caso ímpar, no próximo passo teremos b par, e podemos utilizar a mesma propriedade acima.

Analisando funções para exponenciação

```
    int exp3(int a, int b) {
1       if (b == 1)
2           return a;
3       if ((b % 2) == 0)
4           return exp3(a*a, b/2);
        else
5           return a*exp3(a, b-1);
    }
```

- Qual a ordem de crescimento de exp3?

Analisando funções para exponenciação

```
int exp3(int a, int b) {
   if (b == 1)
      return a;
   if ((b % 2) == 0)
      return exp3(a*a, b/2);
   else
      return a*exp3(a, b-1);
}
```

- Qual a ordem de crescimento de `exp3`?
- Para b par, temos 1 comparação, 1 operação de resto, outra comparação, 1 multiplicação, 1 divisão e o retorno $=$ 6 operações, mais a quantidade de operações necessárias para resolver $T(b/2)$.

Analisando funções para exponenciação

```
int exp3(int a, int b) {
    if (b == 1)
        return a;
    if ((b % 2) == 0)
        return exp3(a*a, b/2);
    else
        return a*exp3(a, b-1);
}
```

- Qual a ordem de crescimento de exp3?
- Para b par, temos 1 comparação, 1 operação de resto, outra comparação, 1 multiplicação, 1 divisão e o retorno = 6 operações, mais a quantidade de operações necessárias para resolver $T(b/2)$.
- Então no caso em que b é par: $T(b) = 6 + T(b/2)$

```
int exp3(int a, int b) {
    if (b == 1)
        return a;
    if ((b % 2) == 0)
        return exp3(a*a, b/2);
    else
        return a*exp3(a, b-1);
}
```

- Para b ímpar, temos 1 comparação, 1 operação de resto, outra comparação, 1 multiplicação, 1 subtração e o retorno = 6 operações, mais a quantidade de operações necessárias para resolver $T(b-1)$.

Analisando funções para exponenciação

```
    int exp3(int a, int b) {
1       if (b == 1)
2           return a;
3       if ((b % 2) == 0)
4           return exp3(a*a, b/2);
        else
5           return a*exp3(a, b-1);
    }
```

- Para b ímpar, temos 1 comparação, 1 operação de resto, outra comparação, 1 multiplicação, 1 subtração e o retorno = 6 operações, mais a quantidade de operações necessárias para resolver $T(b-1)$.
- Então no caso em que b é ímpar: $T(b) = 6 + T(b-1)$

Uma função para exponenciação que não é linear

- Assim, temos, para `exp3`:

Uma função para exponenciação que não é linear

- Assim, temos, para `exp3`:

- b par: $T(b) = 6 + T(b/2)$
- b ímpar: $T(b) = 6 + T(b-1)$
- mas como no próximo passo do caso ímpar, estaremos no caso par, então:

Uma função para exponenciação que não é linear

- Assim, temos, para `exp3`:

- b par: $T(b) = 6 + T(b/2)$
- b ímpar: $T(b) = 6 + T(b-1)$
- mas como no próximo passo do caso ímpar, estaremos no caso par, então:
- b ímpar: $T(b) = 6 + (6 + T\left(\frac{b-1}{2}\right))$

Uma função para exponenciação que não é linear

- Assim, temos, para `exp3`:

- b par: $T(b) = 6 + T(b/2)$
- b ímpar: $T(b) = 6 + T(b-1)$
- mas como no próximo passo do caso ímpar, estaremos no caso par, então:
- b ímpar: $T(b) = 6 + (6 + T\left(\frac{b-1}{2}\right))$

- podemos aproximar $T(b)$ por um limite superior,

$$T(b) = 12 + T\left(\frac{b-1}{2}\right)$$
$$\approx 12 + T\left(\frac{b}{2}\right)$$

Uma função para exponenciação que não é linear

- a cada chamada recorrente, o problema é dividido pela metade:

$$
\begin{aligned}
T(b) &= 12 + T\left(\frac{b}{2}\right) \\
&= 12 + 12 + T\left(\frac{b}{4}\right) \\
&= 12 + 12 + 12 + T\left(\frac{b}{8}\right) \\
&= 12k + T\left(\frac{b}{2^k}\right)
\end{aligned}
$$

Uma função para exponenciação que não é linear

- a cada chamada recorrente, o problema é dividido pela metade:

$$\begin{aligned} T(b) &= 12 + T\left(\frac{b}{2}\right) \\ &= 12 + 12 + T\left(\frac{b}{4}\right) \\ &= 12 + 12 + 12 + T\left(\frac{b}{8}\right) \\ &= 12k + T\left(\frac{b}{2^k}\right) \end{aligned}$$

- o caso base ocorre quando: $b/2^k = 1$

Uma função para exponenciação que não é linear

- a cada chamada recorrente, o problema é dividido pela metade:

$$
\begin{aligned}
T(b) &= 12 + T\left(\frac{b}{2}\right) \\
&= 12 + 12 + T\left(\frac{b}{4}\right) \\
&= 12 + 12 + 12 + T\left(\frac{b}{8}\right) \\
&= 12k + T\left(\frac{b}{2^k}\right)
\end{aligned}
$$

- o caso base ocorre quando: $b/2^k = 1$
- ou seja:

$$
\begin{aligned}
b &= 2^k \\
k &= \log_2 b
\end{aligned}
$$

Analisando funções para exponenciação

```
int exp3(int a, int b) {
    if (b == 1)
        return a;
    if ((b % 2) == 0)
        return exp3(a*a, b/2);
    else
        return a*exp3(a, b-1);
}
```

- A complexidade de `exp3`, desconsiderando constantes, é $O(\log b)$

Analisando funções para exponenciação

```
int exp3(int a, int b) {
    if (b == 1)
        return a;
    if ((b % 2) == 0)
        return exp3(a*a, b/2);
    else
        return a*exp3(a, b-1);
}
```

- A complexidade de exp3, desconsiderando constantes, é $O(\log b)$
- Em exp1 e exp2, o problema era reduzido em 1 unidade a cada etapa — um sinal de que eram lineares.

Analisando funções para exponenciação

```
    int exp3(int a, int b) {
1       if (b == 1)
2           return a;
3       if ((b % 2) == 0)
4           return exp3(a*a, b/2);
        else
5           return a*exp3(a, b-1);
    }
```

- A complexidade de exp3, desconsiderando constantes, é $O(\log b)$
- Em exp1 e exp2, o problema era reduzido em 1 unidade a cada etapa — um sinal de que eram lineares.
- Em exp3 o problema é dividido por um fator (2) a cada etapa — característico de algoritmos de complexidade logaritmica.

Sumário

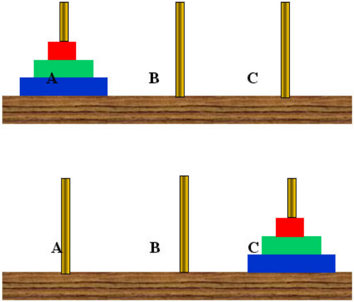
1. Análise de algoritmos

2. Relações de Recorrências
   - Definições e exemplos
   - Torres de Hanói

3. Método de substituição

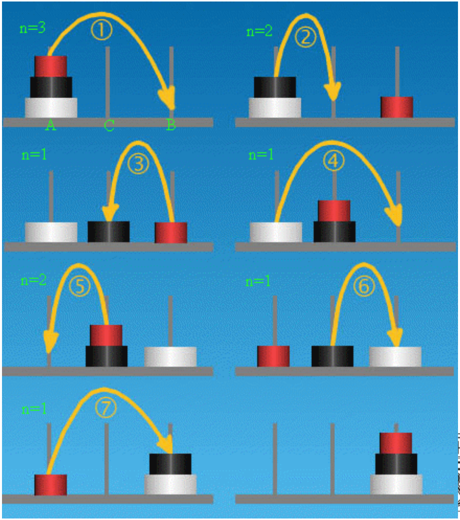
Torres de Hanói

- Problema que consiste em três postes e um número de discos de diferentes tamanhos que podem deslizar pelos postes.
- É preciso mover os discos de um poste a outro seguindo as seguintes regras: a) mover um disco de cada vez e b) um disco maior não pode ficar sobre um disco menor.

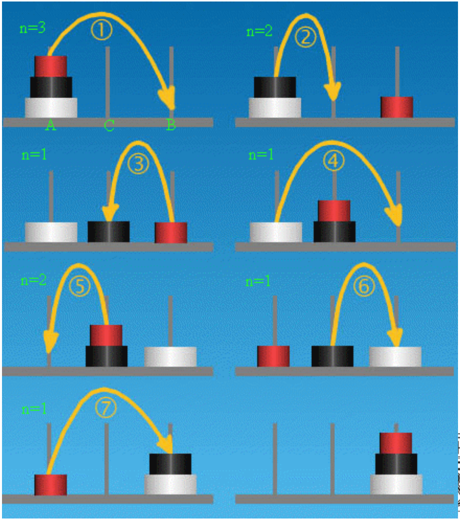


Torres de Hanói

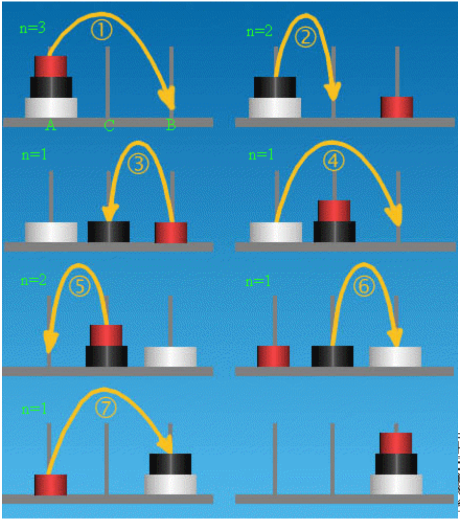
- Tentar resolver esse problema é um bom exercício de como pensar recursivamente
- É preciso entender o caso base e como reduzir o problema a uma instância menor.



- Tentar resolver esse problema é um bom exercício de como pensar recursivamente
- É preciso entender o caso base e como reduzir o problema a uma instância menor.
- A estratégia básica é:
  1. Mover $n - 1$ discos do poste origem para o intermediário

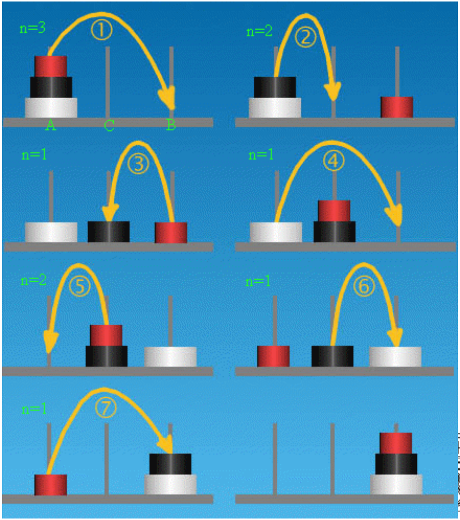


- Tentar resolver esse problema é um bom exercício de como pensar recursivamente
- É preciso entender o caso base e como reduzir o problema a uma instância menor.
- A estratégia básica é:
  1. Mover $n-1$ discos do poste origem para o intermediário
  2. Mover 1 disco (disco base) do poste origem para o destino



Torres de Hanói

- Tentar resolver esse problema é um bom exercício de como pensar recursivamente
- É preciso entender o caso base e como reduzir o problema a uma instância menor.
- A estratégia básica é:
  1. Mover $n - 1$ discos do poste origem para o intermediário
  2. Mover 1 disco (disco base) do poste origem para o destino
  3. Mover $n - 1$ discos do poste intermediário para o destino



Torres de Hanói

```
  void Hanoi(int tam, char ori, char des, char interm) {
1   if (tam == 1)
2      printf("Mova disco de %c para %c\n", ori, des);
    else {
3      Hanoi(tam-1, ori, interm, des);
4      Hanoi(1, ori, des, interm);
5      Hanoi(tam-1, interm, des, ori);
    }
  }
```

- Qual é a ordem de crescimento para esse algoritmo? ($T(1) = 2$)

Torres de Hanói

```
  void Hanoi(int tam, char ori, char des, char interm) {
1   if (tam == 1)
2      printf("Mova disco de %c para %c\n", ori, des);
    else {
3      Hanoi(tam-1, ori, interm, des);
4      Hanoi(1, ori, des, interm);
5      Hanoi(tam-1, interm, des, ori);
    }
  }
```

- Qual é a ordem de crescimento para esse algoritmo? ($T(1) = 2$)
- Para encontrar uma fórmula temos: uma comparação, uma movimentação de 1 disco e duas movimentações de $n-1$ discos.

Torres de Hanói

```
void Hanoi(int tam, char ori, char des, char interm) {
    if (tam == 1)
        printf("Mova disco de %c para %c\n", ori, des);
    else {
        Hanoi(tam-1, ori, interm, des);
        Hanoi(1, ori, des, interm);
        Hanoi(tam-1, interm, des, ori);
    }
}
```

- Qual é a ordem de crescimento para esse algoritmo? ($T(1) = 2$)
- Para encontrar uma fórmula temos: uma comparação, uma movimentação de 1 disco e duas movimentações de $n-1$ discos.
- $T(n) = 1 + T(1) + 2 \cdot T(n-1)$

Torres de Hanói

Fórmula básica

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

Fórmula básica

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

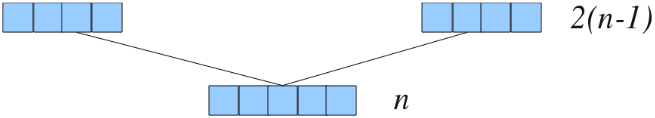
expandindo...

$$
\begin{aligned}
T(n) &= 3 + 2 \cdot 3 + 4 \cdot T(n-2) \\
&= 3 + 2 \cdot 3 + 4 \cdot 3 + 8 \cdot T(n-3) \\
&\dots \\
T(n) &= 3(1 + 2 + \cdots + 2^{k-1}) + 2^k T(n-k)
\end{aligned}
$$

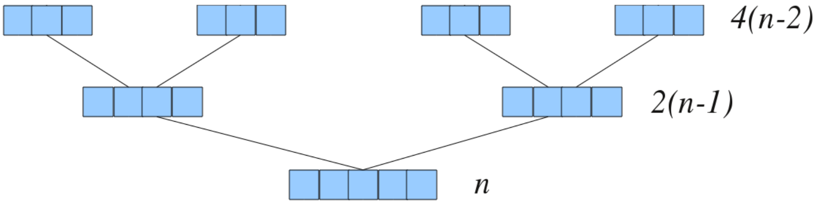
Torres de Hanói

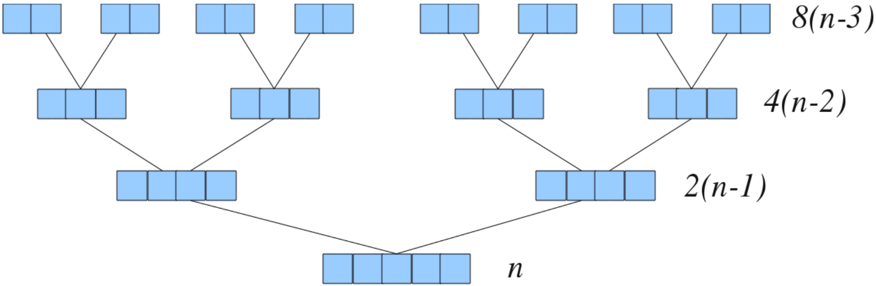
n

Torres de Hanói

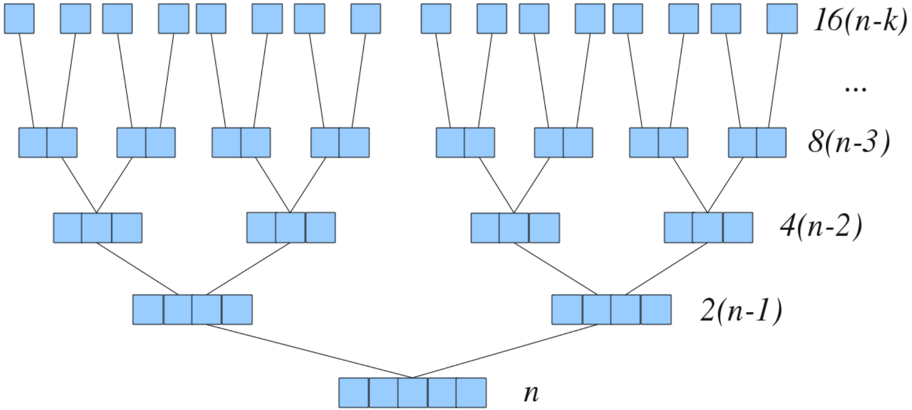


Torres de Hanói




8(n-3)
4(n-2)
2(n-1)
n

Torres de Hanói



- Considere $T(n) = 3(1 + 2 + \cdots + 2^{k-1}) + 2^k T(n-k)$.
- Veja que, para $n = 5$, e desconsiderando o primero termo (soma constante), teremos:

$$\begin{aligned} T(5) &= 2^k(5-k) \\ &= 2^4(5-4) \\ &= 2^4 \end{aligned}$$

Torres de Hanói

- Considere $T(n) = 3(1 + 2 + \cdots + 2^{k-1}) + 2^k T(n-k)$.
- Veja que, para $n = 5$, e desconsiderando o primero termo (soma constante), teremos:

$$\begin{aligned} T(5) &= 2^k(5-k) \\ &= 2^4(5-4) \\ &= 2^4 \end{aligned}$$

- Podemos dizer que $T(n) \approx 2^{n-1}$.

Torres de Hanói

- Considere $T(n) = 3(1 + 2 + \cdots + 2^{k-1}) + 2^k T(n-k)$.
- Veja que, para $n = 5$, e desconsiderando o primero termo (soma constante), teremos:

$$\begin{aligned} T(5) &= 2^k(5-k) \\ &= 2^4(5-4) \\ &= 2^4 \end{aligned}$$

- Podemos dizer que $T(n) \approx 2^{n-1}$.
- A complexidade do problema é de ordem exponencial, mais especificamente $O(2^n)$.

- Considere $T(n) = 3(1 + 2 + \cdots + 2^{k-1}) + 2^k T(n-k)$.
- Veja que, para $n = 5$, e desconsiderando o primero termo (soma constante), teremos:

$$\begin{aligned} T(5) &= 2^k(5-k) \\ &= 2^4(5-4) \\ &= 2^4 \end{aligned}$$

- Podemos dizer que $T(n) \approx 2^{n-1}$.
- A complexidade do problema é de ordem exponencial, mais especificamente $O(2^n)$.
- Ao olharmos de forma superficial, pareceria linear. No entanto, a cada passo o problema é dividido em **duas** partes menores, o que fez grande diferença.

Torres de Hanói

- Problema inventado por Édouard Lucas em 1883, com base em uma lenda (inventada por ele ou que o inspirou?).

Torres de Hanói

- Problema inventado por Édouard Lucas em 1883, com base em uma lenda (inventada por ele ou que o inspirou?).
- O criador do universo, no início dos tempos criou em Hanoi uma grande sala com três postes. Num dos postes colocou 64 discos dourados de tamanhos diferentes, do maior para o menor.
- Os sacerdotes de Hanói, criados na mesma época, de acordo com a lenda, realizam movimentos com os discos de um poste para outro seguindo as duas regras do problema.

Torres de Hanói

- Problema inventado por Édouard Lucas em 1883, com base em uma lenda (inventada por ele ou que o inspirou?).
- O criador do universo, no início dos tempos criou em Hanoi uma grande sala com três postes. Num dos postes colocou 64 discos dourados de tamanhos diferentes, do maior para o menor.
- Os sacerdotes de Hanói, criados na mesma época, de acordo com a lenda, realizam movimentos com os discos de um poste para outro seguindo as duas regras do problema.
- Segundo a estória, quando o último movimento do quebra-cabeças for feito, o mundo chegara ao fim.

Sumário

Método de substituição

- Existem muitos métodos para se obter uma fórmula fechada para recorrências. Um dos mais utilizados é o método de substituição.

Método de substituição

- Existem muitos métodos para se obter uma fórmula fechada para recorrências. Um dos mais utilizados é o método de substituição.
- É também conhecido como “expandir, conjecturar e verificar”.
- Consiste em duas etapas:
  1. Pressupor a formula da solução (expandir e conjecturar),
  2. Usar indução matemática para mostrar que a solução funciona.

Método de substituição

- Existem muitos métodos para se obter uma fórmula fechada para recorrências. Um dos mais utilizados é o método de substituição.
- É também conhecido como "expandir, conjecturar e verificar".
- Consiste em duas etapas:
  1. Pressupor a formula da solução (expandir e conjecturar),
  2. Usar indução matemática para mostrar que a solução funciona.
- O nome vem da substituição do palpite pela função resposta.
- Pode-se ajustar o palpite para encontrar funções mais exatas.
- Pode ser usado para estabelecer limites superiores ou inferiores sobre uma recorrência.

Método de substituição

- Existem muitos métodos para se obter uma fórmula fechada para recorrências. Um dos mais utilizados é o método de substituição.
- É também conhecido como "expandir, conjecturar e verificar".
- Consiste em duas etapas:
  1. Pressupor a formula da solução (expandir e conjecturar),
  2. Usar indução matemática para mostrar que a solução funciona.
- O nome vem da substituição do palpite pela função resposta.
- Pode-se ajustar o palpite para encontrar funções mais exatas.
- Pode ser usado para estabelecer limites superiores ou inferiores sobre uma recorrência.
- As análises que fizemos até agora contemplam as partes de "expandir e conjecturar". No entanto, ainda precisamos verificar por indução se o "palpite" está correto.

Método de substituição

- Queremos provar que um dado $T(n)$ é verdadeiro para $n \geq 1$.
- Utilizamos para a indução o caso base, $T(1)$, supomos $T(n)$ e provamos por hipótese que $T(n-1)$ é verdadeiro.

Método de substituição

- Queremos provar que um dado $T(n)$ é verdadeiro para $n \geq 1$.
- Utilizamos para a indução o caso base, $T(1)$, supomos $T(n)$ e provamos por hipótese que $T(n-1)$ é verdadeiro.
- Exemplo (exp2): tínhamos a fórmula $T(n) = 4 + T(n-1)$ e chegamos à fórmula fechada $T(n) = 4n - 2$.
  - Para $n = 1$ é fácil ver que a fórmula está correta, pois $T(1) = 2$.

Método de substituição

- Queremos provar que um dado $T(n)$ é verdadeiro para $n \geq 1$.
- Utilizamos para a indução o caso base, $T(1)$, supomos $T(n)$ e provamos por hipótese que $T(n-1)$ é verdadeiro.
- Exemplo (exp2): tínhamos a fórmula $T(n) = 4 + T(n-1)$ e chegamos à fórmula fechada $T(n) = 4n - 2$.
    - Para $n = 1$ é fácil ver que a fórmula está correta, pois $T(1) = 2$.
    - Agora, tome $n > 1$ e suponha que a fórmula fechada acima vale com $n - 1$ no lugar de n.

$$\begin{aligned} T(n) &= 4 + T(n-1) \\ &= 4 + [4(n-1) - 2] \\ &= 4 + [4n - 6] \\ &= 4n - 2 \end{aligned}$$

Método de substituição

- Queremos provar que um dado $T(n)$ é verdadeiro para $n \geq 1$.
- Utilizamos para a indução o caso base, $T(1)$, supomos $T(n)$ e provamos por hipótese que $T(n-1)$ é verdadeiro.
- Exemplo (exp2): tínhamos a fórmula $T(n) = 4 + T(n-1)$ e chegamos à fórmula fechada $T(n) = 4n - 2$.
  - Para $n = 1$ é fácil ver que a fórmula está correta, pois $T(1) = 2$.
  - Agora, tome $n > 1$ e suponha que a fórmula fechada acima vale com $n - 1$ no lugar de n.

$$\begin{aligned} T(n) &= 4 + T(n-1) \\ &= 4 + [4(n-1) - 2] \\ &= 4 + [4n - 6] \\ &= 4n - 2 \end{aligned}$$

  - dessa forma, provamos por indução que nosso palpite é verdadeiro e, portanto, exp2 é $O(n)$.

Bibliografia

- ZIVIANI, N. **Projeto de algoritmos**: com implementações em Pascal e C. (seção 1.4). 2.ed. Thomson, 2004.
- CORMEN, T.H. et al. **Algoritmos: Teoria e Prática** (Capítulo 4). Campus. 2002.
- FEOFILOFF, P. **Recorrências**. Disponível em: `http://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/recorrencias.html`.