Análise da Complexidade de Algoritmos Recursivos II

Joaquim Madeira

26/03/2020

Sumário

- Recap
- Generalização: algoritmos exponenciais
- Pesquisa binária – versão recursiva
- The Master Theorem
- The Smoothness Rule
- Exercício adicional
- Sugestões de leitura

Recapitulação

Decrease-And-Conquer

- Exploit the relationship between
  - A solution to a given problem instance
  - A solution to a smaller instance of the same problem

- General framework (Top-Down)
  - Identify ONE similar and smaller problem instance
  - The smaller instance is solved recursively
  - Solutions for smaller instances are processed to get the solution of the original problem, if needed

Decrease-And-Conquer

- How does instance size decrease ?
- Decrease by a constant factor
  - n ; n / 2 ; n / 4; …
  - n ; n / 3 ; n / 9; …
- Decrease by a constant
  - n ; n - 1 ; n – 2 ; …
- Variable-size decrease
  - Size reduction pattern varies from iteration to iteration

Decrease by a Constant Factor

- Reduce instance size by a constant factor in each iteration
  - Usually, decrease by halving !

$$T(1) = c$$

$$T(n) = T(n / b) + f(n)$$

- Examples ?

Decrease by a Constant

- Reduce instance size by a constant in each iteration
  - Usually, decrease by one !

$$T(1) = c$$

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

- Examples ?

Divide-And-Conquer

- The best-known algorithm design technique

- General framework
  - Divide a problem instance into (two or more) similar, smaller instances
  - The smaller instances are solved recursively
  - Solutions for smaller instances are combined to get the solution of the original problem, if needed

Divide-And-Conquer

- In each subdivision step, the smaller instances should have approx. the same size !

- All smaller problem instances have to be solved !!

- When do we stop the subdivision process ?
  - Base cases ? Just one or more ?
  - Smaller instances might be solved by another algorithm

Divide-And-Conquer

- This recursive strategy can be implemented
  - Using recursive functions / procedures (obvious solution !)
  - Iteratively, using a stack, queue, etc.
    - Choose which sub-problem to solve next !!
- Problems ?
  - Recursion is slow !
    - Solve small instances using other algorithms
  - Not the best approach for simple problems !
  - Sub-problems might overlap !
    - Reuse previous results / solutions – Dynamic Programming !

Divide-And-Conquer

- Did you work out this example from our last class?
- Computing b^n using $b^n = b^{n \text{ div } 2} \times b^{(n+1) \text{ div } 2}$
- Number of multiplications ?

$$M(n) = M(n \text{ div } 2) + M((n+1) \text{ div } 2) + 1$$

Divide-And-Conquer

- What if n is a power of 2 ?
  - $n = 2^k$, $k = \log_2 n$

$$M(n) = M(n / 2) + M(n / 2) + 1 = 2\,M(n / 2) + 1 = \ldots$$

- Closed formula ? Complexity order ?

- Is it better than the direct algorithm ?

Homework – Searching an Array

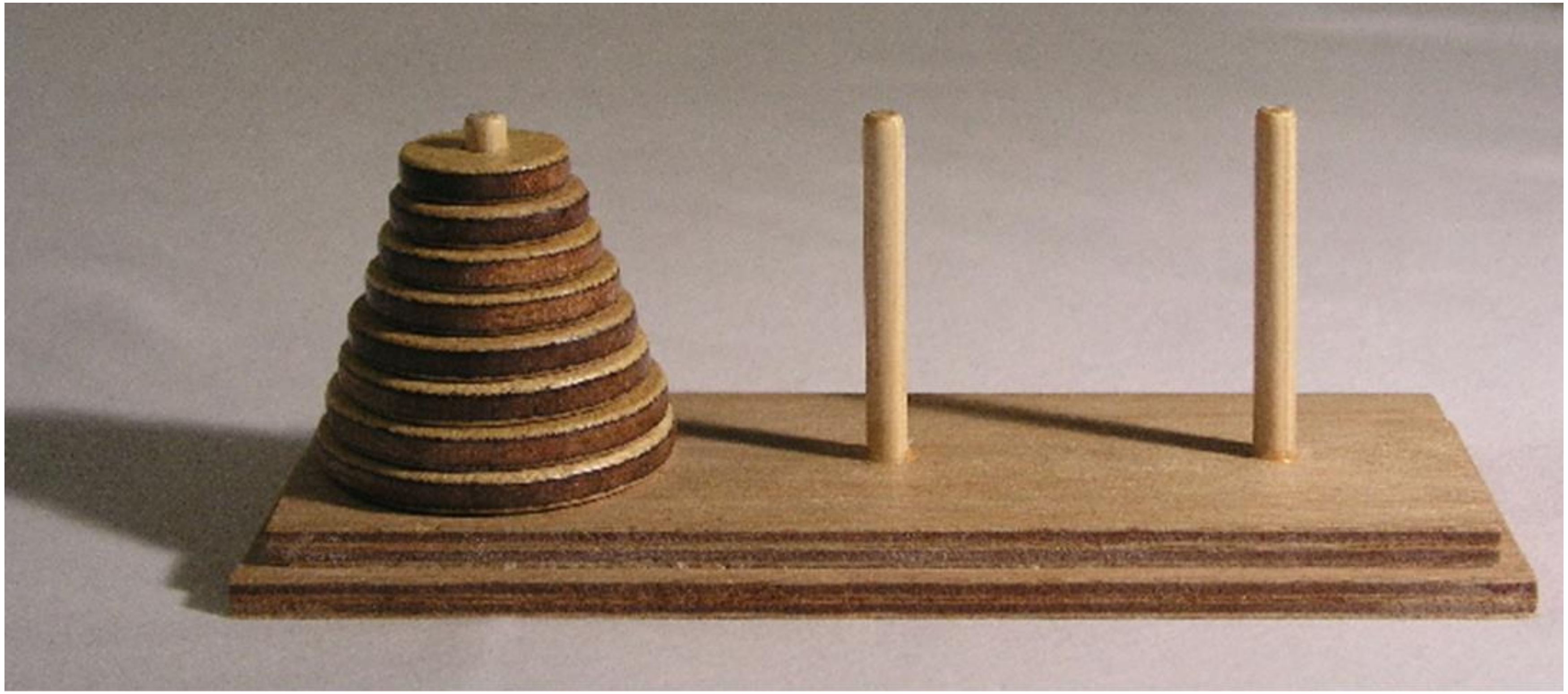
- Given a n-element array of integers
- Get the max value!

- Divide-And-Conquer
  - Get max value of left sub-array : ((n+1) div 2) elements
  - Get max value of right sub-array : (n div 2) elements
  - Compare the two and return the largest

- How many comparisons?

Generalização:
algoritmos exponenciais

As Torres de Hanói



[Wikipedia]

Função recursiva

```
// torresDeHanoi('A', 'B', 'C', 8);

void torresDeHanoi(char origem, char auxiliar, char destino, int n) {
  if (n == 1) {
    contadorGlobalMovs++;
    moverDisco(origem, destino);  // Imprime o movimento
    return;
  }
  // Divide-and-Conquer
  torresDeHanoi(origem, destino, auxiliar, n - 1);

  contadorGlobalMovs++;
  moverDisco(origem, destino);

  torresDeHanoi(auxiliar, origem, destino, n - 1);
}
```

Nº de movimentos realizados

$M(1) = 1$

$M(n) = M(n-1) + 1 + M(n-1) = 1 + 2\ M(n-1)$

$M(n) = 1 + 2\ M(n-1) = 1 + 2 \times (1 + 2\ M(n-2)\ ) = 1 + 2 + 4\ M(n-2) = \ldots$

$M(n) = 2^0 + 2^1 + 2^2 + \ldots\ 2^{k-1} + 2^k\ M(n-k)$

Caso de base: $M(1) = 1\ ;\ k = n - 1$

$M(n) = 2^0 + 2^1 + 2^2 + \ldots\ 2^{n-2} + 2^{n-1} = 2^n - 1$ $\boldsymbol{M(n) \in \mathcal{O}(2^n)}$

Padrão de comportamento

$$T(1) = b$$

$$T(n) = a \times T(n - c) + d$$

- a : nº de subproblemas a resolver em cada passo
- b : nº de operações / tempo para o caso de base
- c : diminuição do tamanho do problema
- d : nº de operações / tempo de processamento a cada passo

Decrease-and-Conquer

$$T(1) = b$$

$$T(n) = a \times T(n - c) + d$$

- $a = 1$

$$T(n) = b + d \times (n - 1) / c \qquad \boldsymbol{T(n) \in \mathcal{O}(n)}$$

- Aplica-se a algum exemplo anterior?
- Sugestão: fazer o desenvolvimento

Divide-and-Conquer

$$T(1) = b$$

$$T(n) = a \times T(n - c) + d$$

- $a > 1$

$$T(n) = d/(1 - a) + (b - d/(1 - a)) \times \mathbf{a}^{\mathbf{(n - 1) / c}}$$

$$\boldsymbol{T(n) \in \mathcal{O}(a^{\frac{n}{c}})}$$

- Aplica-se às Torres de Hanói? Verificar!
- Sugestão: fazer o desenvolvimento

Pesquisa binária – versão recursiva

Binary Search

- Given a sorted array of n elements : A[left..right]

- Search value / key X : index ?

- Idea
  - Compare A[middle] with X
  - If equal, return middle
  - If larger, recursively search in A[left..middle - 1]
  - If smaller, recursively search in A[middle + 1..right]

Binary Search

- How to compute middle ?
  - Be sure to avoid overflow ! Shifting !

- How many comparisons per iteration ?
  - Try using just one comparison !

- How to report a non-existing value / key ?
  - Signed vs. unsigned !

Binary Search

- Let’s do it !

Binary Search

```c
int pesqBinRec(int* v, int esq, int dir, int valor) {
  unsigned int meio;

  if (esq > dir) return -1;

  meio = (esq + dir) / 2;

  contadorComps++;
  if (v[meio] == valor) {
    return meio;
  }
  contadorComps++;
  if (v[meio] > valor) {
    return pesqBinRec(v, esq, meio - 1, valor);
  }

  return pesqBinRec(v, meio + 1, dir, valor);
}
```

Binary Search

- Best case ?
  - Just 1 iteration

- Worst case ?
  - Always select the largest partition !
  - Odd vs. even number of elements ?
  - When do we always have equal-sized partitions ?

- Try to obtain a closed formula for the number of iterations !!

Binary Search

- Let’s do it !

Binary Search

- $n = 2^k$
- esq = 0 dir = $2^k - 1$ meio = $2^{k-1} - 1$
- Pior caso: escolher sempre a partição da direita
  - É a maior das duas !!

W(1) = 2

W(n) = 2 + W(n/2) = 4 + W(n/4) = 6 + W(n/8) = ...

W(n) = 2 x k + W(1) = 2 + 2 log n

$$\boldsymbol{W(n) \in \mathcal{O}(\log_2 n)}$$

Extra – The Fake-Coin Problem

- Given n identically looking coins
- Find the one that is a fake !
- Use only a balance scale !
- The fake coin is lighter than a genuine one !
- Efficient algorithm ?

Extra – Ternary Search

- Given a sorted array of n elements : A[left..right]
- Search value / key X : index ?

- Idea
  - Compare A[leftThird] with X
  - If equal, return leftThird
  - If larger, recursively search in A[left..leftThird - 1]
  - Compare A[rightThird] with X
  - If equal, return rightThird
  - If larger, rec. search in A[leftThird + 1..rightThird - 1]
  - If smaller, recursively search in A[rightThird + 1..right]

General Recurrence

- Assume that $n = b^k$, $k \geq 1$

- Number of operations, i.e., execution time

$$T(n) = a\, T(n / b) + f(n)$$

  - a : number of smaller instances (integer, $a \geq 1$)
  - b : size factor (integer, $b \geq 2$)
  - $f(n)$: number of ops (or time) for defining smaller instances and/or combining their results

The Master Theorem

The Master Theorem

- Given a recurrence, for $n = b^k$, $k \geq 1$

$$T(1) = c \quad \text{and} \qquad T(n) = a\,T(n / b) + f(n)$$

where $a \geq 1$, $b \geq 2$, $c > 0$

- Theorem : If $f(n)$ in $\Theta(n^d)$, where $d \geq 0$, then

$$T(n) \text{ in } \Theta(n^d), \text{ if } a < b^d$$

$$T(n) \text{ in } \Theta(n^d \log n), \text{ if } a = b^d$$

$$T(n) \text{ in } \Theta(n^{\log_b a}), \text{ if } a > b^d$$

The Master Theorem

- Allows directly obtaining complexity order, given a recurrence
  - But not a closed formula for the number of ops !
- Similar results hold for the $O(n)$ and $\Omega(n)$ notations
- Example

$$M(n) = 2\ M(n / 2) + 1$$

$f(n) = 1$, $f(n)$ in $\Theta(n^0)$, $d = 0$

$a = 2$, $b = 2$, $a > b^d$

$M(n)$ in $\Theta(n)$

The Smoothness Rule

Smooth Functions

- Eventually non-decreasing function

$$f(n_1) \leq f(n_2), \text{ for any } n_2 > n_1 \geq n_0$$

- Smooth function
  1) f(n) is eventually non-decreasing
  2) f(2n) in $\Theta(f(n))$

- Examples
  - log n , n, n log n and n^k are smooth functions
  - a^n is not !!

The Smoothness Rule

- Let $T(n)$ be an eventually non-decreasing function.
- And let $f(n)$ be a smooth function.

- If $T(n)$ in $\Theta(f(n))$ for values of n that are powers of b, where $b \geq 2$,

- Then $T(n)$ in $\Theta(f(n))$.

- Analogous results for $O(n)$ and $\Omega(n)$!!

- This is very good news !!

Decrease by a Constant Factor

- Reduce instance size by a constant factor in each iteration

$$T(1) = c$$

$$T(n) = T(n / b) + f(n)$$

- Complexity ?
  - T(n) in $\Theta(\log n)$, if $f(n)$ = constant
  - T(n) in $\Theta(n)$, if $f(n)$ in $\Theta(n)$

- Examples ?

Exercício adicional

Mais um algoritmo – Decrease-And-Conquer

- Desenvolva uma função para calcular b^n usando

$$b^n = b^{n \text{ div } 2} \times b^{n \text{ div } 2} \text{ , se n é par}$$
$$b^n = b \times b^{(n-1) \text{ div } 2} \times b^{(n-1) \text{ div } 2} \text{ , se n é impar}$$

- Fazer uma só chamada recursiva em cada passo !!
- Quais são os casos de base ?

- Quantas multiplicações são efetuadas ?
- Qual é a ordem de complexidade ?

Sugestões de leitura

Sugestões de leitura

- A. Levitin, Introduction to the Design and Analysis of Algorithms, 3rd Edition, 2012
  - Capítulo 4: secção 4.4
  - Capítulo 5: secção 5.4
  - Apêndice B