CS483 Design and Analysis of Algorithms

Chapter 2 Divide and Conquer Algorithms

Instructor: Fei Li

lifei@cs.gmu.edu with subject: CS483

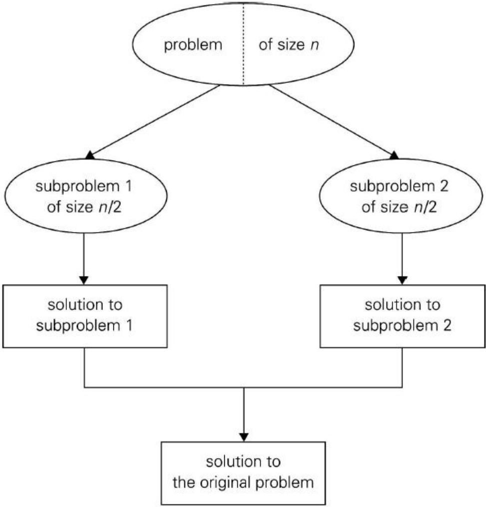
Office hours:
Room 5326, Engineering Building, Thursday 4:30pm - 6:30pm or by appointments

Course web-site:
http://www.cs.gmu.edu/∼lifei/teaching/cs483_fall09

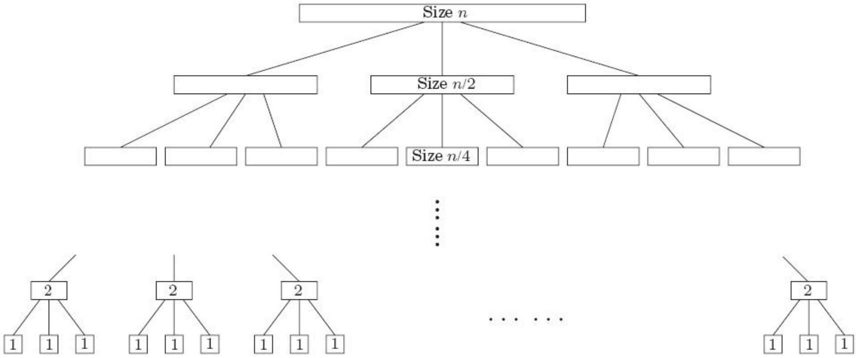
Figures unclaimed are from books "Algorithms" and "Introduction to Algorithms"

Divide and Conquer Algorithms

1. Breaking the problem into subproblems of the same type
2. Recursively solving these subproblems
3. Appropriately combining their answers



Divide-and-Conquer Recurrence



Divide-and-Conquer Recurrence

- Divide the problems into b smaller instances; a of them need to be solved. $f(n)$ is the time spent on dividing and merging

Divide-and-Conquer Recurrence

- Divide the problems into b smaller instances; a of them need to be solved. $f(n)$ is the time spent on dividing and merging
- $$T(n) = a \cdot T(n/b) + f(n)$$

Divide-and-Conquer Recurrence

- Divide the problems into b smaller instances; a of them need to be solved. $f(n)$ is the time spent on dividing and merging
- $$T(n) = a \cdot T(n/b) + f(n)$$
- 1. **The iteration method**

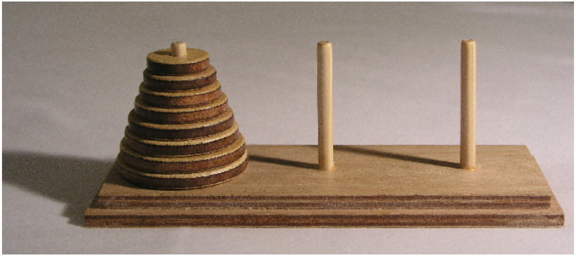
     Expand (iterate) the recurrence and express it as a summation of terms depending only on n and the initial conditions
  2. **The substitution method**
     - Guess the form of the solution
     - Use mathematical induction to find the constants
  3. **Master Theorem** ($T(n) = a \cdot T(n/b) + f(n)$)

Iteration Method — Examples

- $n!$

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

- **Tower of Hanoi**



http://en.wikipedia.org/wiki/Tower_of_Hanoi

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

Iteration — Example

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

$$\begin{aligned} T(n) &= T(n-1) + 1 \\ &= (T(n-2) + 1) + 1 \\ &= T(n-2) + 2 \\ \cdots & \quad \cdots \\ &= T(n-i) + i \\ \cdots & \quad \cdots \\ &= T(0) + n = n \end{aligned}$$

- **Tower of Hanoi** $(T(n) = 2 \cdot T(n-1) + 1)$?

Iteration — Example

Tower of Hanoi $(T(n) = 2 \cdot T(n-1) + 1)$

$$
\begin{aligned}
T(n) &= 2 \cdot T(n-1) + 1 \\
&= 2 \cdot (2 \cdot T(n-2) + 1) + 1 \\
&= 2^2 \cdot T(n-2) + 2 + 1 \\
\cdots & \quad \cdots \\
&= 2^i \cdot T(n-i) + 2^{i-1} + \cdots + 1 \\
\cdots & \quad \cdots \\
&= 2^{n-1} \cdot T(1) + 2^{n-1} + 2^{n-1} + \cdots + 1 \\
&= 2^{n-1} \cdot T(1) + \sum_{i=0}^{n-2} 2^i \\
&= 2^{n-1} + 2^{n-1} - 1 \\
&= 2^n - 1
\end{aligned}
$$

Substitution Method — Count Number of Bits

- **Count number of bits** ($T(n) = T(\lfloor n/2 \rfloor) + 1$)

Substitution Method — Count Number of Bits

- **Count number of bits** ($T(n) = T(\lfloor n/2 \rfloor) + 1$)
- Guess $T(n) \leq \log n$.

$$
\begin{aligned}
T(n) &= T(\lfloor n/2 \rfloor) + 1 \\
&\leq \log(\lfloor n/2 \rfloor) + 1 \\
&\leq \log(n/2) + 1 \\
&\leq (\log n - \log 2) + 1 \\
&\leq \log n - 1 + 1 \\
&= \log n
\end{aligned}
$$

Substitution Method — Tower of Hanoi

- **Tower of Hanoi** ($T(n) = 2 \cdot T(n-1) + 1$)

Substitution Method — Tower of Hanoi

- **Tower of Hanoi** ($T(n) = 2 \cdot T(n-1) + 1$)
- Guess $T(n) \leq 2^n$

$$\begin{aligned} T(n) &= 2 \cdot T(n-1) + 1 \\ &\leq 2 \cdot 2^{n-1} + 1 \\ &\leq 2^n + 1, \quad \text{wrong!} \end{aligned}$$

Substitution Method — Tower of Hanoi

- **Tower of Hanoi** $(T(n) = 2 \cdot T(n-1) + 1)$
- Guess $T(n) \leq 2^n$

$$\begin{aligned} T(n) &= 2 \cdot T(n-1) + 1 \\ &\leq 2 \cdot 2^{n-1} + 1 \\ &\leq 2^n + 1, \quad \text{wrong!} \end{aligned}$$

- Guess $T(n) \leq 2^n - 1$

$$\begin{aligned} T(n) &= 2 \cdot T(n-1) + 1 \\ &\leq 2 \cdot (2^{n-1} - 1) + 1 \\ &= 2^n - 2 + 1 \\ &= 2^n - 1, \quad \text{correct!} \end{aligned}$$

Substitution Method — Extension F_n

- **Fibonacci Numbers** ($F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$)

Substitution Method — Extension F_n

- **Fibonacci Numbers** $(F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2})$
- $F_{n-2} < F_{n-1} < F_n, \forall n \geq 1$

Substitution Method — Extension F_n

- **Fibonacci Numbers** $(F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2})$
- $F_{n-2} < F_{n-1} < F_n, \forall n \geq 1$
- Assume $2^{n-1} < F_n < 2^n$
  Guess $F_n = c \cdot \phi^n$, $1 < \phi < 2$

Substitution Method — Extension F_n

- **Fibonacci Numbers** $(F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2})$
- $F_{n-2} < F_{n-1} < F_n, \forall n \geq 1$
- Assume $2^{n-1} < F_n < 2^n$
  Guess $F_n = c \cdot \phi^n$, $1 < \phi < 2$
- $$\begin{aligned} c \cdot \phi^n &= c \cdot \phi^{n-1} + c \cdot \phi^{n-2} \\ \phi^2 &= \phi + 1 \\ \phi &= \frac{1 \pm \sqrt{5}}{2} \end{aligned}$$

Substitution Method — Extension F_n

- **Fibonacci Numbers** $(F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2})$
- $F_{n-2} < F_{n-1} < F_n, \forall n \geq 1$
- Assume $2^{n-1} < F_n < 2^n$
  Guess $F_n = c \cdot \phi^n$, $1 < \phi < 2$
-

$$\begin{aligned} c \cdot \phi^n &= c \cdot \phi^{n-1} + c \cdot \phi^{n-2} \\ \phi^2 &= \phi + 1 \\ \phi &= \frac{1 \pm \sqrt{5}}{2} \end{aligned}$$

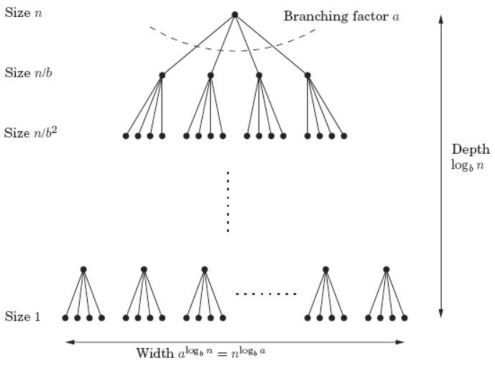
- **General solution:** $F_n = c_1 \cdot \phi_1^n + c_2 \cdot \phi_2^n$
  $F_1 = 0,\ F_2 = 1$

$$F_n = \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^n - \frac{1}{\sqrt{5}}(\frac{1-\sqrt{5}}{2})^n$$

Master Theorem

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

Figure 2.3 Each problem of size n is divided into a subproblems of size n/b.



Master Theorem

$$T(n) = a \cdot T(n/b) + f(n), \quad a \geq 1, b > 1 \text{ be constants.}$$

Theorem

We interpret n/b to mean either $\lceil n/b \rceil$ or $\lfloor n/b \rfloor$.
If $f(n) \in \Theta(n^d)$, where $d \geq 0$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

$$T(n) = \begin{cases} \Theta(f(n)) & \text{if } f(n) = \Omega(n^{\log_b a + \epsilon}) \text{ and if } a \cdot f(n/b) \leq c \cdot f(n) \\ & \quad \text{for some constant } c < 1 \text{ and all sufficiently large } n \\ \Theta(n^{\log_b a} \cdot \log n) & \text{if } f(n) = \Theta(n^{\log_b a}) \\ \Theta(n^{\log_b a}) & \text{if } f(n) = O(n^{\log_b a - \epsilon}) \text{ for some constant } \epsilon > 0 \end{cases}$$

1. $T(n) = 4 \cdot T(n/2) + n = ?$
2. $T(n) = 4 \cdot T(n/2) + n^2 = ?$
3. $T(n) = 4 \cdot T(n/2) + n^3 = ?$

Chapter 2 of DPV — Divide and Conquer Algorithms

1. Mergesort
2. Medians
3. Matrix Multiplication

Mergesort

Mergesort

- Given an array of n numbers, sort the elements in non-decreasing order

Mergesort

- Given an array of n numbers, sort the elements in non-decreasing order
- `function mergesort(A[n])`

```
if (n = 1)
    return A;
else
```

$B = A[1, \ldots, \lceil \frac{n}{2} \rceil];$
$C = A[\lceil \frac{n}{2} \rceil + 1, \ldots, n];$

```
    mergesort(B);
    mergesort(A);
    merge(B, C, A);
```

Mergesort

- Given an array of n numbers, sort the elements in non-decreasing order
- `function mergesort(A[n])`

```
if (n = 1)
    return A;
else
```

$$B = A[1, \ldots, \lceil \frac{n}{2} \rceil];$$
$$C = A[\lceil \frac{n}{2} \rceil + 1, \ldots, n];$$

```
    mergesort(B);
    mergesort(A);
    merge(B, C, A);
```

- Is this algorithm complete?

Mergesort

Merge two sorted arrays, B and C and put the result in A

```
function merge(B[1, .. p], C[1, .. q], A[1, .. p + q])

    i = 1; j = 1;

    for (k = 1, 2, .. p + 1)
        if (B[i] < C[j])
            A[k] = B[i];
            i = i + 1;
        else
            A[k] = C[j];
            j = j + 1;
```

24, 11, 91, 10, 22, 32, 22, 3, 7, 99

$$T(n) = 2 \cdot T(n/2) + O(n) = O(n \cdot \log n)$$

Medians

The *median* is a single representative value of a list of numbers: half of them are larger and half of them are smaller; less sensitive to outliers

$$S := \{2,\ 36,\ 5,\ 21,\ 8,\ 13,\ 11,\ 20,\ 5,\ 4,\ 1\}$$

Selection problem:

- **Input**: A list of number S; an integer k.
- **Output**: The k-th smallest element of S.

$$\text{selection}(S, 8) = ?$$

Medians

The *median* is a single representative value of a list of numbers: half of them are larger and half of them are smaller; less sensitive to outliers

$$S := \{2,\ 36,\ 5,\ 21,\ 8,\ 13,\ 11,\ 20,\ 5,\ 4,\ 1\}$$

Let us split at $v = 5$

$$\begin{aligned} S_L &= \{2,\ 4,\ 1\} \\ S_v &= \{5,\ 5\} \\ S_R &= \{36,\ 21,\ 8,\ 13,\ 11,\ 20\} \end{aligned}$$

$$\text{selection}(S, 8) = \text{selection}(S_R, 3).$$

$$\text{selection}(S, k) = \begin{cases} \text{selection}(S_L, k) & \text{if } k \le |S_L| \\ v & \text{if } |S_L| < k \le |S_L| + |S_v| \\ \text{selection}(S_R, k - |S_L| - |S_v|) & \text{if } k > |S_L| + |S_v|. \end{cases}$$

Medians

The *median* is a single representative value of a list of numbers: half of them are larger and half of them are smaller; less sensitive to outliers

$$S := \{2,\ 36,\ 5,\ 21,\ 8,\ 13,\ 11,\ 20,\ 5,\ 4,\ 1\}$$

$$\text{selection}(S, 8) = \text{selection}(S_R, 3).$$

$$\text{selection}(S, k) = \begin{cases} \text{selection}(S_L, k) & \text{if } k \le |S_L| \\ v & \text{if } |S_L| < k \le |S_L| + |S_v| \\ \text{selection}(S_R, k - |S_L| - |S_v|) & \text{if } k > |S_L| + |S_v|. \end{cases}$$

If $|S_L| \approx |S_R|$ (i.e., pick up v to be the median),

$$T(n) = T(n/2) + O(n)$$

Pick up v randomly from S

Medians

v is *good* if it lies within 25% and 75% of the array it is chosen

Lemma

On average a fair coin needs to be tossed two times before a "heads" is seen

Proof.

$E = 1 + \frac{1}{2} \cdot E$ □

Remark

v has 50% chance of being in-between [25%, 75%]. We need to pick v twice to make it good

Theorem

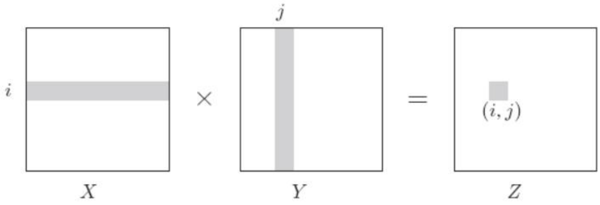
$$T(n) \leq T((3/4) \cdot n) + O(n) = O(n)$$

Matrix Multiplication

The product of two $n \times n$ matrices X and Y is a third $n \times n$ matrix $Z = XY$, with (i, j)th entry

$$Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}.$$

To make it more visual, Z_{ij} is the dot product of the ith row of X with the jth column of Y:



Matrix Multiplication

1. Matrix Multiplication (by definition):

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$= \begin{bmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{12} \cdot B_{22} \end{bmatrix}$$

Matrix Multiplication

1. Matrix Multiplication (by definition):

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$= \begin{bmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{12} \cdot B_{22} \end{bmatrix}$$

2. $$T(n) = 8 \cdot T(\frac{n}{2}) + O(n) = O(n^3)$$

Time complexity of the brute-force algorithm is $O(n^3)$

Matrix Multiplication

1. Strassen's Matrix Multiplication:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

- $m_1 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$
- $m_2 = (A_{21} + A_{22}) \cdot B_{11}$
- $m_3 = A_{11} \cdot (B_{12} - B_{22})$
- $m_4 = A_{22} \cdot (B_{21} - B_{11})$
- $m_5 = (A_{11} + A_{12}) \cdot B_{22}$
- $m_6 = (A_{21} - A_{11}) \cdot (B_{11} + B_{12})$
- $m_7 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$

Matrix Multiplication

1. Strassen's Matrix Multiplication:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

   - $m_1 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$
   - $m_2 = (A_{21} + A_{22}) \cdot B_{11}$
   - $m_3 = A_{11} \cdot (B_{12} - B_{22})$
   - $m_4 = A_{22} \cdot (B_{21} - B_{11})$
   - $m_5 = (A_{11} + A_{12}) \cdot B_{22}$
   - $m_6 = (A_{21} - A_{11}) \cdot (B_{11} + B_{12})$
   - $m_7 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$

2. $$T(n) = 7 \cdot T(\frac{n}{2}) + O(n) = O(\log n^{\log_2 7}) \approx O(n^{2.81})$$

Time complexity of the brute-force algorithm is $O(n^3)$