COMPSCI 330: Algorithms **Fall 2012**

Duke University **Instructor: Chittu Tripathy**

Lecture 04: Divide and Conquer

Sep 07, 2012

- Divide and Conquer
  - Merge Sort
  - Binary Search
  - Prune and Search Method
  - Powering a Number
  - Computing Fibonacci Numbers
  - Matrix Multiplication
  - Strassen's Matrix Multiplication

These notes are hand-written, unedited and sketchy. They are primarily used for, and based on my lectures.

If you find any bug, impreciseness, or a rare poor-/mis-interpretation of facts, please let me know. I will be grateful for any additional comments you have that are intended to make the quality of the notes better.

Please note that I will provide my hand-written lecture notes only for a subset of my lectures, *not for all lectures.* Therefore, it is your responsibility to attend all the lectures, take notes regularly, and ask me and/or the TAs if you have any questions.

Thank you!
--Chittu

Written using an Intuos 5 Touch Pen Tablet and Microsoft Office OneNote

Divide and Conquer: In politics: Divide and rule / Divide et impera.

- One of the powerful techniques for algorithm design
- Leads to naturally recursive algorithms
- Analysis uses recurrences

Algorithm: Divide-and-Conquer

Step 1: Divide the problem instance into a set of smaller subproblems
Step 2: Conquer the subproblems by solving them recursively
Step 3: Combine the solutions to the subproblems to obtain solution to the original problem

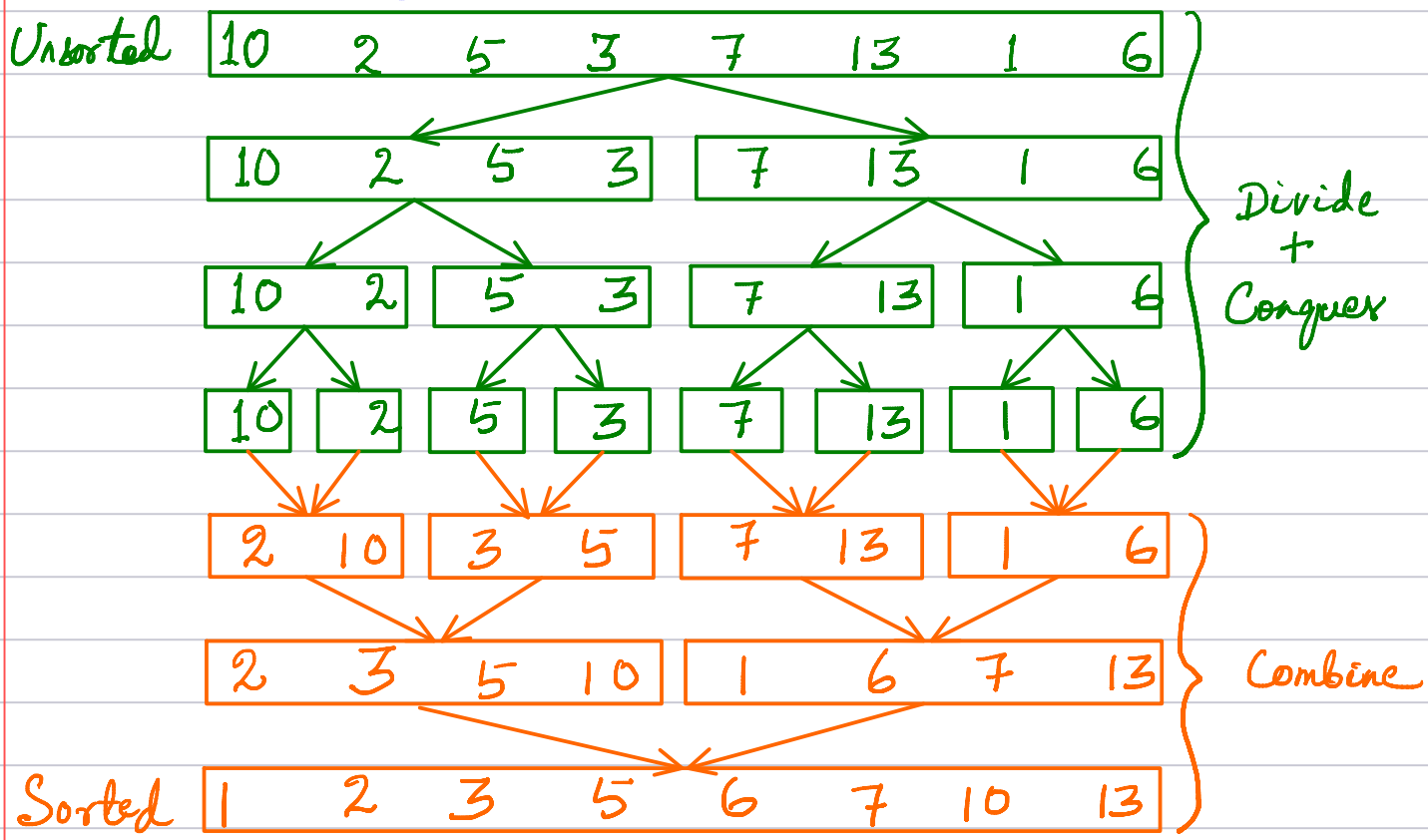
Merge Sort:

Given: An array $A[p \cdots r]$ of n elements for which $<, =, >$ relations are defined

Goal: Sort A in nondecreasing order.

Idea: Divide and Conquer

1. If $n=1$ then do nothing
2. Divide A into two halves and recursively sort each half (Conquer)
3. Merge two sorted arrays (Combine)




```
MERGE-SORT(A, p, r)
1. if p < r
     q ← ⌊(r-p)/2⌋                    // Divide: Trivial
     MERGE-SORT(A, p, q)              } Conquer
     MERGE-SORT(A, q+1, r)
     MERGE(A, p, q, r)                // Combine: Θ(n) for MERGE
```

Time: $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \lg n)$

MERGE2(A, p, q, B, r, s) // Merge two sorted arrays A[p...q] and B[r...s]

1. if p > q return B
2. if r > s return A
3. if A[p] ≤ B[r]
4. return A[p] ∘ MERGE2(A, p+1, q, B, r, s) // ∘ is concatenation
5. else return B[r] ∘ MERGE2(A, p, q, B, r+1, s)

Note: MERGE2 is implemented using an auxiliary array of size which is the sum of the sizes of A and B.

We can use MERGE2 to implement our MERGE procedure.

MERGE(A, p, q, r)

1. C[p...r] ← MERGE2(A, p, q, A, q+1, r)
2. Copy C back to A

Exercise: MERGE-SORT can be implemented in a bottom-up iterative manner using a queue. Design an algorithm for the iterative MERGE-SORT.

Exercise: The MERGE2 procedure can be implemented as a simple iterative procedure. It is helpful to recall the two-fingers algorithm we briefly discussed during the lecture. Design an iterative algorithm for MERGE2.

Binary Search:

Given: a sorted array A of n elements, and a key x

Goal: Check if the key x is present in A. If so, then return the corresponding index of the element in A; Otherwise, report error.

2 4 6 9 10 14 17 Search: x = 10

2 4 6 9 10 14 17

~~2 4 6 9~~ 10 14 17

~~2 4 6 9~~ 10 ~~14 17~~

~~2 4 6 9~~ 10 ~~14 17~~

```
REC-BIN-SRCH(A, x, i, j)
1. if i > j return "key not found"
2. mid ← ⌊(i+j)/2⌋
3. if A[mid] = x
4.     return mid
5. else if A[mid] > x
       return REC-BIN-SRCH(A, x, i, mid-1)
   else return REC-BIN-SRCH(A, x, mid+1, j)
```

Exercise: Give an iterative version of binary search.

Divide: Check the middle element

Conquer: Throw away half of the array, and search for x in the other half.

Combine: Trivial. Do nothing

Time: $T(n) = T(n/2) + \Theta(1) = \Theta(\lg n)$

Note: - In binary search we always throw away half of the array and search in the other half.

- A divide and conquer algorithm which throws away a fraction of the input and recursively work on the other part of the input is called a "Prune and Search" algorithm

- Binary search is an example.

Prune and Search:

- A special type of divide an conquer in which we throw away (i.e. eliminate from further considerations, a.k.a. PRUNE) a fraction of the input and recurse on the remaining part of the input.
- Example: Binary search, where half of the array is pruned in each step giving us the recurrence

$$T(n) = T(n/2) + \Theta(1)$$

- General Prune and Search Recurrence:

$$T(n) = T(\alpha n) + f(n) \Rightarrow T(n) = \begin{cases} \Theta(\lg n), & \text{if } f(n) = \Theta(1) \\ \Theta(f(n)), & \text{otherwise.} \end{cases}$$

$0 < \alpha < 1$

$f(n)$: cost of pruning

- Pruning factor: $1 - \alpha$
- Input size for recursive calls: $n, \alpha n, \alpha^2 n, \ldots, \Theta(1)$.

- More General Prune and Search Recurrence:

Theorem: Let $T(n) = T(\alpha_1 n) + T(\alpha_2 n) + \cdots + T(\alpha_k n) + f(n)$ for constants $0 < \alpha_1, \ldots, \alpha_k < 1$ s.t. $\alpha_1 + \alpha_2 + \cdots + \alpha_k < 1$. Then the solution to $T(n)$ is given by

$$T(n) = \begin{cases} \Theta(\lg n), & \text{if } f(n) = \Theta(1) \\ \Theta(f(n)), & \text{otherwise.} \end{cases}$$

- We will see one more example of prune and search when we design an algorithm for finding median (or i-th smallest element in an array).

Powering a Number: Computing a^n, $n \in \mathbb{N}$.

$$a^n = \underbrace{a * a * \cdots * a}_{n \text{ } a\text{'s}}$$

Algorithm 1: naively multiply from left to right.

```
ITERATIVE-POWER(a, n)
1. result ← 1
2. for i ← 1 to n
3.     result ← result * a
4. return result
```

#multiplications

Time: $T(n) = \Theta(n)$.

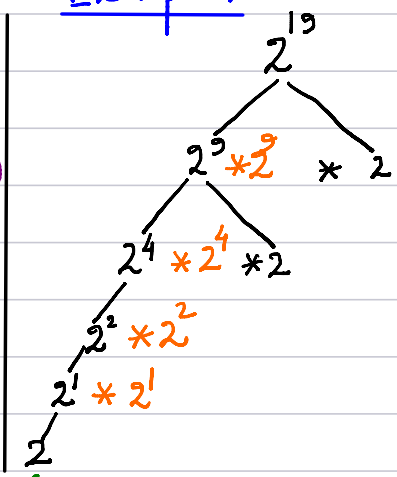
Perpetual Question: Can we do better?

Idea: Let's try divide-and-conquer — split n into two halves, compute $a^{\lfloor n/2 \rfloor}$, save it (i.e. Memoize) to use once more to compute a^n.

Algorithm 2: Use the above idea

```
RECURSIVE-POWER(a, n)
1. if n = 1    return a
2. if n is even
       memo ← RECURSIVE-POWER(a, n/2)
       return memo * memo
   else // i.e. n is odd
       memo ← RECURSIVE-POWER(a, (n-1)/2)
       return memo * memo * a
```

Example:



Time: $T(n) = T(n/2) + \Theta(1) = \Theta(\lg n)$.

#multiplications

Computing Fibonacci Numbers:

- We have seen in the last lecture:

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

Time: $T(n) = \Omega\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$ ← Exponential!

- Previous lectures: we've seen one exponential-time algorithm, one linear-time algorithm with memoization, and one linear-time algorithm without memoization for computing n-th Fibonacci number.

Perpetual Question: Can we do better?

From Homework $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$. (proof by induction)

Is it not equivalent to computing a^n where a is a 2×2 matrix?

Since two 2×2 matrices can be multiplied in $O(1)$ time, the time complexity of computing n-th Fibonacci number is:

$$T(n) = 2T(n/2) + O(1) = \Theta(\lg n),$$

which follows directly from how we computed a^n previously.

Matrix Multiplication:

Input: $\begin{pmatrix} a & b \\ c & d \end{pmatrix} * \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{pmatrix}$

$A_{n\times n} * B_{n\times n} = C_{n\times n}$

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot a_{kj}$$

Algorithm 1: High-School Matrix Multiplication Code:

```
MULT1(A_nxn, B_nxn)
  for i ← 1 to n
    for j ← 1 to n
      c_ij ← 0
      for k ← 1 to n
        c_ij ← c_ij + a_ik * b_kj
```

Running Time: $T(n) = \Theta(n^3)$.

Perpetual Question: can we do better?

Idea: Try divide-and-conquer.

$$\underbrace{\left\{\begin{pmatrix} a & b \\ c & d \end{pmatrix}}_{n/2 \times n/2}\right._{n\times n} * \begin{pmatrix} e & f \\ g & h \end{pmatrix}_{n\times n} = \begin{pmatrix} a*e+b*g & a*f+b*h \\ c*e+d*g & c*f+d*h \end{pmatrix}_{n\times n}$$

D-&-C-MULT(A, B)

Exercise for you!

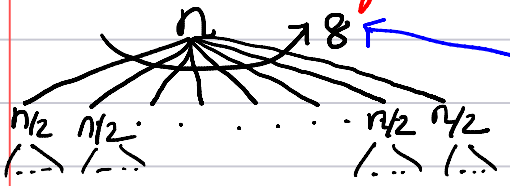
Obs: # multiplications = 8

additions = 4

Size of a, ..., h reduces by half

Running Time: $T(n) = 8T(n/2) + \Theta(n^2) = \Theta(n^3)$.

What can be a possible way of bringing down the running time?



Somehow can we make it 7 or less at the expenses of doing more (constant #) number of additions?

Volker Strassen showed it is possible in 1969!

Strassen's Matrix Multiplication: (1969)

Idea: Replace a multiplication by additions.

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} * \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} r=ae+bg & s=af+bh \\ t=ce+dg & u=cf+dh \end{pmatrix}$$

$$A \quad * \quad B \quad = \quad C$$

$$\begin{aligned} p_1 &= a(f-h) \\ p_2 &= (a+b)h \\ p_3 &= (c+d)e \\ p_4 &= d(g-e) \\ p_5 &= (a+d)(e+h) \\ p_6 &= (b-d)(g+h) \\ p_7 &= (a-c)(e+f) \end{aligned}$$

$$\begin{aligned} r &= p_5 + p_4 - p_2 + p_6 \\ &= ae + bg \\ s &= p_1 + p_2 \\ &= af + bh \\ t &= p_3 + p_4 \\ &= ce + dg \\ u &= p_1 + p_5 - p_3 - p_7 \\ &= cf + dh \end{aligned}$$

Total: #multiplication = 7

#addition/subtraction = 18

So intricate that it makes us wonder how Strassen discovered it!

Algorithm:

- **Divide:** Divide A and B into 8 submatrices $a, \dots, h$ of dimensions $\frac{n}{2} \times \frac{n}{2}$.
- **Conquer:** Recursively multiply the 7 submatrices
- **Combine:** Write C by doing the add/sub operations to compute r, s, t and u.

Time: $T(n) = 7\,T(n/2) + \Theta(n^2) = \Theta(n^{\lg 7}) = \Theta(n^{2.808})$

Aside: Coppersmith–Winograd bound: $O(n^{2.376})$

Recently, in 2011, Vassilevska Williams gave a bound of $O(n^{2.3727})$.

Trivial lower bound: $\Omega(n^2)$. Can this gap be closed?