Divide-and-Conquer

Overview

A divide-and-conquer algorithm has three main steps:

1. Break the problem into smaller subproblems
2. Recursively solve the subproblems
3. Combine the subproblem solutions to solve the original problem

Divide-and-Conquer: Multiplication

pretended this is constant time

n-digit

- How do we do binary multiplication?

$$
\begin{array}{r}
10011 \\
\times\ 1110 \\
\hline
0000 \\
10010 \\
1100100 \\
\hline
\end{array}
$$

n dig.

n dig.

n^2 pairs

$O(n)$ space

- What is the running time of this method?

$\boxed{O(n^2)}$ n is # digits

Divide-and-Conquer: Multiplication v. 2

real + imaginary $\quad i = \sqrt{-1} \quad$ a, b, c, d real

- Complex number multiplication: $\quad$ i imaginary

$$(a + bi)(c + di) = ac - bd + (bc + ad)i$$

$$bi \cdot di = b \cdot d \cdot i^2 = -bd$$

- 4 multiplications
- Can we do it with fewer multiplications?

Divide-and-Conquer: Multiplication v. 2

- Complex number multiplication:

$$(a + bi)(c + di) = ac - bd + (bc + ad)i$$

1001
1010

4 mult
2 adds
1 subtraction

real mult. $O(n^2)$
real add/sub $O(n)$

or

$$(a + bi)(c + di) = ac - bd + ((a + b)(c + d) - ac - bd)i$$

3 multiplications!

3 mult.
3 add
3 sub

Divide-and-Conquer: Multiplication v. 2

- Complex number multiplication can be done with 3 multiplications, instead of 4
- But this is a constant factor improvement- how does it help us in big-O terms?

original method $= 4 \cdot O(n^2) + 3 \cdot O(n)$

new method $= 3 \cdot O(n^2) + 6 \cdot O(n)$

both are $O(n^2)$

Divide-and-Conquer: Multiplication v. 3

- Multiplying binary numbers:

x, y n digits

$$x = \boxed{x_L}\boxed{x_R} = (2^{n/2}x_L + x_R)$$

$$y = \boxed{y_L}\boxed{y_R} = (2^{n/2}y_L + y_R).$$

$\mathrm{Prod}(x,y)$

$$xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) =$$

$\mathrm{Prod}(x_L,y_L)$ $\mathrm{Prod}(x_L,y_R)$ $\mathrm{Prod}(x_R,y_L)$ $\mathrm{Prod}(x_R,y_R)$

$$2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R.$$

Divide-and-Conquer: Multiplication v. 3

other work = $O(n)$

- Multiplying binary numbers:

$$xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) =$$

Prod(x,y)

$$2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R.$$

- Describe the running time as a recurrence relation:

T(n) = running time of Prod(x,y) when x,y both have length n

$T(n) = aT(\frac{n}{b}) + O(n^d) = 4T(\frac{n}{2}) + O(n^1)$

- What is the running time?

$a = 4$ $\quad d = 1$

$b = 2$

$\log_2 4 = 2 > 1$

$\boxed{O(n^2)}$

Divide-and-Conquer: Karatsuba Method for Multiplication

- Multiplying binary numbers:

$x_L y_L \quad x_R y_R$

$(x_L + x_R)(y_L + y_R)$

$$xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) =$$
$$2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R.$$

- Use Gauss' trick!

$$x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R.$$

- Now what is the recurrence relation? Running time?

$T(n) = 3T(\frac{n}{2}) + O(n^1) \qquad O(n^{1.6})$

$\log_b a = \log_2 3 \approx 1.6 > 1$

Divide-and-Conquer: Karatsuba Method for Multiplication

$\boxed{x_R y_R}$

- Multiplying binary numbers:

$$xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) =$$

n $\quad n/2 \quad n \quad n/2$

$$2^n\, x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R.$$

$S(n/2)$ $\quad$ $S(n/2)$

n $\quad$ n

- Use Gauss' trick!

$$x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R.$$

$S(n/2)$

- Now what is the recurrence relation? Running time?

$S(n)$ = space required by Prod for two #s of len. n

$$S(n) = 3S\left(\frac{n}{2}\right) + O(n) = O(n^{1.6})$$

Divide-and-Conquer: Binary Search

- Given a sorted array A and a target value k, find the index of value k in A

Divide-and-Conquer: Binary Search

Divide-and-Conquer: MergeSort

- Given an input array of real numbers, we want to output the array in sorted order
- MergeSort idea:
  - Split the array in half
  - Sort each half
  - Merge the two halves together

Divide-and-Conquer: MergeSort

n size of array

```
function mergesort(a[1...n])
Input:  An array of numbers a[1...n]
Output:  A sorted version of this array

if n > 1:
  return merge(mergesort(a[1...⌊n/2⌋]), mergesort(a[⌊n/2⌋+1...n]))
else:
  return a
```

merge([1, 2, 13] [4, 6, 8])

x y

1 ∘ merge([2, 13], [4, 6, 8])

2 ∘ merge([13], [4, 6, 8])

Divide-and-Conquer: MergeSort

- How do we merge?

sorted

$$
\begin{array}{l}
\underline{\texttt{function merge}}(x[1\ldots k], y[1\ldots l]) \\
\texttt{if } k=0\texttt{:} \quad \texttt{return } y[1\ldots l] \\
\texttt{if } l=0\texttt{:} \quad \texttt{return } x[1\ldots k] \\
\texttt{if } x[1]\leq y[1]\texttt{:} \\
\quad \texttt{return } x[1]\circ \texttt{merge}(x[2\ldots k], y[1\ldots l]) \\
\texttt{else:} \\
\quad \texttt{return } y[1]\circ \texttt{merge}(x[1\ldots k], y[2\ldots l])
\end{array}
$$

Divide-and-Conquer: MergeSort

- What is the running time of this function?

define $m = k + l$

```
function merge(x[1...k], y[1...l])
if k = 0:  return y[1...l]
if l = 0:  return x[1...k]
if x[1] ≤ y[1]:
    return x[1] ∘ merge(x[2...k], y[1...l])
else:
    return y[1] ∘ merge(x[1...k], y[2...l])
```

$T(m) = 1 \cdot T(m-1) + O(1) = O(m)$

$T(n_1, n_2) = T(k, l)$

$T(m)$ = running time of merge on input of total size m $= O(m)$

Divide-and-Conquer: MergeSort

- What is the running time of MergeSort?

```
function mergesort(a[1...n])
Input:  An array of numbers a[1...n]
Output:  A sorted version of this array

if n > 1:
  return merge(mergesort(a[1...⌊n/2⌋]), mergesort(a[⌊n/2⌋+1...n]))
else:
  return a
```

$O(n)$

$n/2 \quad + \quad n/2 \quad = n$

$T(n)$ = running time of MS on input of size n

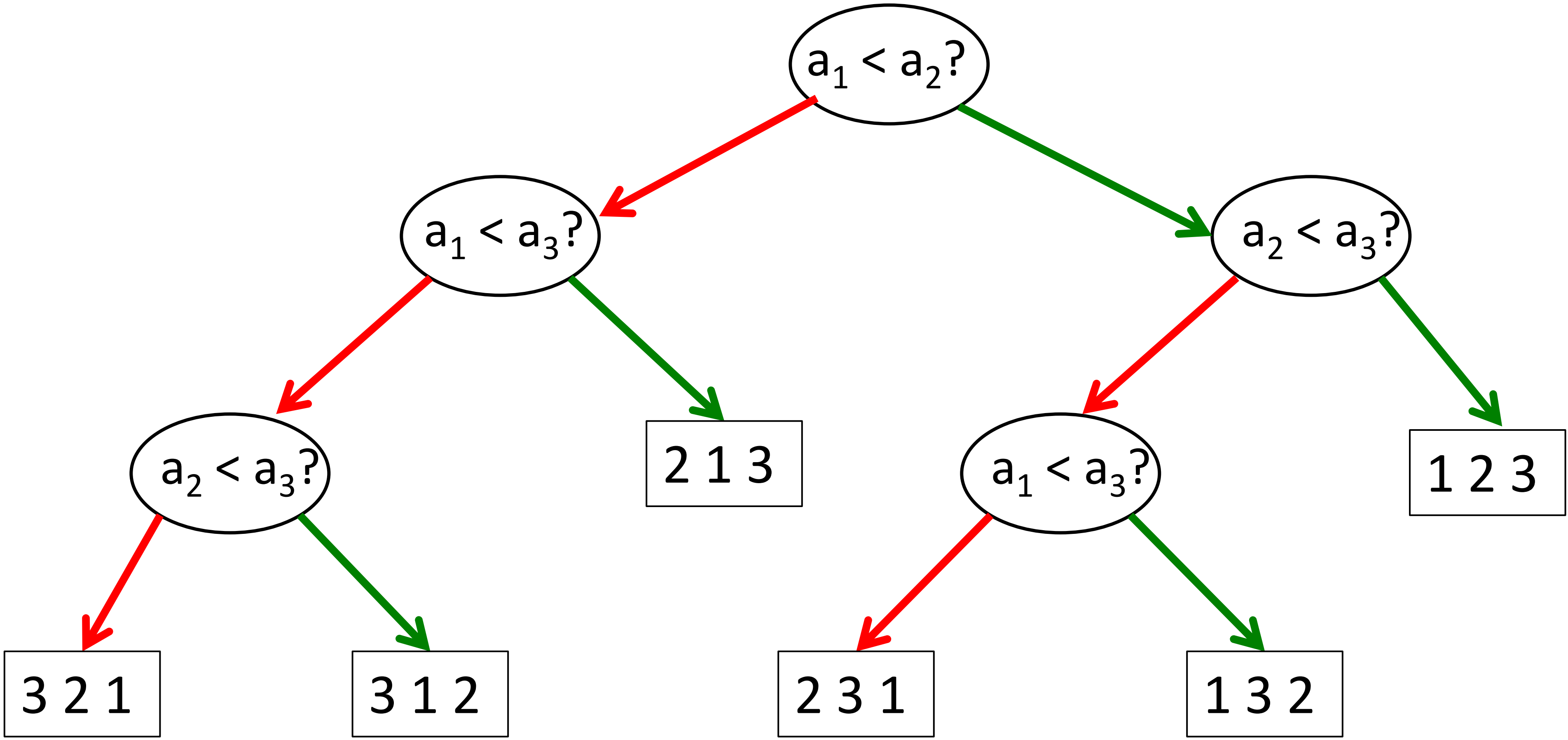
$\log_2 2 = 1 = d$

$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n^1)$ $\quad \boxed{O(n \log n)}$

How Well Can We Do?

- A sort algorithm can be represented as a binary tree
- The leaves of the tree are the possible inputs
- The internal nodes of the tree are the comparison operations

How Well Can We Do?



(Specific comparisons vary by algorithm)

How Well Can We Do?

- Depth of tree is the worst-case running time!
- For array of size *n*, how many leaf nodes?

- For binary tree, what is minimum depth?

In-Class Exercise

- Find the non-duplicate
  - Given a **sorted** array *S* of integers. Each value present appears exactly twice in *S*, except for one (so if there are *k* distinct values, the length is $2k - 1$). Find the non-duplicate value in better than O(n) time.

appears once

0 0 1 1 2 3 3 4 4

1 2 3 4 5 6 7 8 9

In-Class Exercise

before non-d: first copy of each # is on odd idx,
second copy is on even idx

non-d: it is on an odd idx

after non-d: first copy is on even idx
second copy is on odd idx

sketch: Find the odd idx closest to midpoint. ~~If~~ Check value to right.
If they are equal, the non-d is in right half. Recurse on right half.
Otherwise, check value to left. If same, recurse on left. Otherwise, return that value.

$T(n) = 1 \cdot T(\frac{n}{2}) + O(n^0)$

$\log_2 1 = 0 \leq 0$

$O(\log n)$