EECS 376 Discussion 3

Sec 27: Th 5:30-6:30 DOW 1017

IA: Eric Khiu

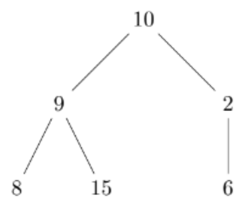
Slide deck available at course drive/Discussion/Slides/Eric Khiu

From Last Time:
Binary Tree Local Max Clarification

5. **Divide and Conquer**

A complete binary tree is a binary tree in which every level, except possibly the last is completely filled and all nodes in the last level are as far left as possible.

Consider a complete binary tree $T = (V, E, r)$ rooted at r where each vertex is labelled with a distinct integer. A vertex $v \in V$ is called a *local maximum* if the label of v is greater than the label of each of its neighbors.



In this example, the local maxima are 10, 15, and 6.

Suppose you are given such a tree where the labelling is given implicitly, i.e., the only way to determine the label of the vertex v is to visit v and query for the vertex label. Provide an algorithm that computes a local maximum of T with using $O(\log(|V|))$ vertex label queries.

Input: Complete, rooted, vertex labelled, binary tree $T = (V, E, r)$
Output: Local maximum v^*

```
function ComputeLocalMaximum(T = (V, E, r))
    if the label of r is greater than both of its children's or r has no children then
        return r
    else
        Let r' be a child of of r with a label greater than r
        Let T' be the complete, rooted, vertex labelled, binary tree induced by r'
        Compute ComputeLocalMaximum(T' = (V', E', r'))
```

- Goal: Return *any* local maximum
- Q: Is it possible that the tree has multiple local maximums?
  - Yes, it's possible
- Q: What if both children are greater than the root?
  - Recurse into any will work
- See Piazza @188 for details

Announcement

- Homework review videos
  - See [Media Gallery] of the Canvas page
  - Please watch them before submitting regrade request
  - You should still read the written solutions
  - Feedback form here
- Quick survey: Preference on future discussions:
  - More content review?
  - More worksheet problems?
  - Hard problems review from past homework?

Agenda

- More on Divide and Conquer
- Karatsuba Algorithm
- Dynamic Programming

Divide and Conquer

Lecture Notes

Recap: Divide and Conquer

- A divide and conquer algorithm usually consists of
  - Base case(s)
  - Dividing the problems
  - Recursive calls
  - Combining results
- General form to apply Master Theorem: $T(n) = kT\left(\frac{n}{b}\right) + O(n^d)$
  - k = number of recursive calls
  - n/b = size of subproblems
  - $O(n^d)$ = cost of dividing and combining the results

Correctness Proof

- Similar idea to prove by induction

Prove by Induction	Correctness Proof for D&C
Prove that $P(0)$ is true	Prove that the base case(s) is/ are correct
Assuming $P(k)$ is true, prove $P(k+1)$ is true	Assuming recursive calls on smaller inputs return correct answer, prove that the current call is correct Extra: Briefly justify you are making recursive calls under correct <u>condition</u> and with correct <u>input</u>

Worksheet: Array Local Min

1. **Array Local Minimum**

   Let $A[1,\ldots,n]$ be an array of distinct integers. The integers in the array are not sorted in any particular order. A cell $A[i]$ is a *local minimum* if $A[i] < A[i-1]$ and $A[i] < A[i+1]$. (If $i = 1$ or $i = n$, it only needs to be smaller than the adjacent cell.)

   Devise a divide and conquer algorithm to find the local minimum in this array in $O(\log n)$ time.

- Consider
  - What are the base case(s)?
  - How to divide the problem?
  - When to make the recursive calls?
  - What is the input to the recursive call?

Worksheet: Array Local Min

1. **Array Local Minimum**

   Let $A[1, \ldots, n]$ be an array of distinct integers. The integers in the array are not sorted in any particular order. A cell $A[i]$ is a *local minimum* if $A[i] < A[i-1]$ and $A[i] < A[i+1]$. (If $i = 1$ or $i = n$, it only needs to be smaller than the adjacent cell.)

   Devise a divide and conquer algorithm to find the local minimum in this array in $O(\log n)$ time.

- Consider
  - What are the base case(s)? Array size = 1
  - How to divide the problem? Start at the middle, recurse into left/ right subarray
  - When to make the recursive calls? If at least one of the neighbor is smaller than the middle element of the array
  - What is the input to the recursive call?
    - Left subarray if middle > left neighbor
    - Right subarray if middle < right neighbor

Worksheet: Array Local Min- Solution

1. **Array Local Minimum**

   Let $A[1,\ldots,n]$ be an array of distinct integers. The integers in the array are not sorted in any particular order. A cell $A[i]$ is a *local minimum* if $A[i] < A[i-1]$ and $A[i] < A[i+1]$. (If $i = 1$ or $i = n$, it only needs to be smaller than the adjacent cell.)

   Devise a divide and conquer algorithm to find the local minimum in this array in $O(\log n)$ time.

Input: An array $A[1,...,n]$ of distinct integers
Output: LOCAL-MIN(A)

```
function LOCAL-MIN(A[1...n])
    if size(A) == 1 then
        return A[1]
    middle = size(A)/2
    if A[middle] < A[middle - 1] and A[middle] < A[middle + 1] then
        return A[middle]
    if A[middle] > A[middle - 1] then
        return LOCAL-MIN(A[1, ..., middle - 1])
    else
        return LOCAL-MIN(A[middle + 1, ..., n])
```

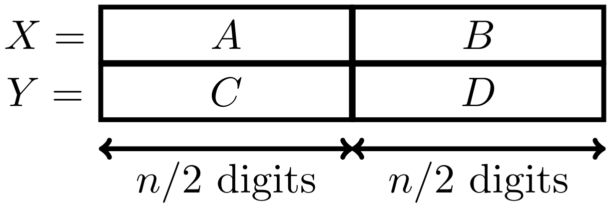
- $T\left(\frac{n}{2}\right) + O(1)$
- Master Theorem says $O(\log n)$

Karatsuba Algorithm

Lecture Notes

Karatsuba Algorithm: Big Idea

- We want to multiply two numbers X and Y. Each has n digits. Naïve way: $O(n^2)$
- Rewrite X and Y as follows:
  - $X = A \cdot 10^{\frac{n}{2}} + B$
  - $Y = C \cdot 10^{\frac{n}{2}} + D$



- Expand $X \cdot Y$ as follows:
  - $X \cdot Y = \left(A \cdot 10^{\frac{n}{2}} + B\right)\left(C \cdot 10^{\frac{n}{2}} + D\right) = AC \cdot 10^n + (AD + BC) \cdot 10^{\frac{n}{2}} + BD$
- Observation: The multiplications AC, AD, BC, BD are smaller versions of the original problem- we can use Divide and Conquer!
- Karatsuba Algorithm: Clever “preparations” to make the conquer step faster

Karatsuba Algorithm

- The Karatsuba Algorithm for Decimal Multiplication is as follows:

```
function KARATSUBA(x, y)
    // n here represents the number of digits in the decimal representation of x
    if n = 1 then
        return x · y
    else
        Write x as a · 10^{n/2} + b
        Write y as c · 10^{n/2} + d
        M_1 ← KARATSUBA(a, c)
        M_2 ← KARATSUBA(b, d)
        M_3 ← KARATSUBA(a + b, c + d)
        return M_1 · 10^n + (M_3 − M_1 − M_2) · 10^{n/2} + M_2
```

- Remember we wanted $AC \cdot 10^n + (AD + BC) \cdot 10^{\frac{n}{2}} + BD$
  - Algebra says $M_3 - M_1 - M_2 = AD + BC$
- Recurrence: $3T\left(\frac{n}{2}\right) + O(n) = O\left(n^{\log_2 3}\right) = O(n^{1.585})$
  - Better than $O(n^2)$

Q: What are the "clever preparations"?

$M_1 = AC$
$M_2 = BD$
$M_3 = (A + B)(C + D)$

Karatsuba Algorithm: Exercise

- Compute 37×76
- $n = 4$ (number of digits)
- $A = 3, B = 7, C = 7, D = 6$

Algorithm 1 The Karatsuba algorithm for Decimal Multiplication

Input: integers x, y, which are both n-digit numbers with $n \geq 1$

Output: $x \cdot y$

```
function KARATSUBA(x, y)
    // n here represents the number of digits in the decimal representation of x
    if n = 1 then
        return x · y
    else
        Write x as a · 10^(n/2) + b
        Write y as c · 10^(n/2) + d
        M1 ← KARATSUBA(a, c)
        M2 ← KARATSUBA(b, d)
        M3 ← KARATSUBA(a + b, c + d)
        return M1 · 10^n + (M3 − M1 − M2) · 10^(n/2) + M2
```

- $M_1 = AC = 3 \cdot 7 = 21$
- $M_2 = BD = 7 \cdot 6 = 42$
- $M_3 = (A + B)(C + D) = (3 + 7)(7 + 6) = 130$

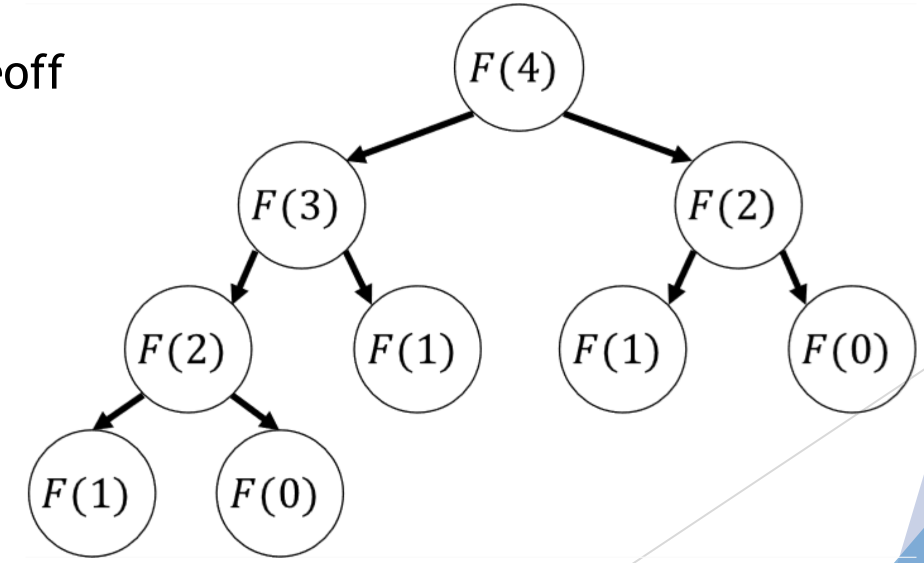
- $37 \times 76 = 21 \cdot 10^2 + (130 - 21 - 42) \times 10 + 42 = 2100 + 67 + 42 = 2812$

Dynamic Programming

Lecture Notes

Dynamic Programming: Big Idea

- In D&C, we divide a problem into a smaller versions of the same problem
- However, for some problems, this recursive subdivision may result in encountering many instances of the exact same problem
- Wouldn’t it be nice if we remember our solution of duplicated problems so that we don’t have to re-solve them?
- Classic debate: Memory-runtime tradeoff
- In DP, we trade memory for runtime



Divide and Conquer vs DP

Divide and Conquer	Dynamic Programming
Divide original problem to smaller version(s) of the same problem	
Non-overlapping subproblems	Overlapping subproblems
Subproblems usually scale down by a constant: $T(n) \rightarrow T\left(\frac{n}{2}\right) \rightarrow T\left(\frac{n}{4}\right) \rightarrow \cdots$	Subproblems don't usually scale down: $T(n) \rightarrow T(n-1) \rightarrow T(n-2) \rightarrow \cdots$
Optimal substructure: The solution is correct for this (scaled-down) portion	Optimal substructure: The solution is correct up to this point
Always top-town	Top-down, memoization, bottom-up
Often less time efficient	Often more time efficient- especially with bottom-up

- Q: Why is MergeSort a D&C algorithm rather than a DP algorithm?
  - No overlapping subproblems- once a subarray is sorted we never have to sort it again

Useful Notations for Recurrence Relations

- Big sum/ Big product ($\sum$, $\prod$)
- Min/ max ($\min, \max$)
- Argmin/ argmax ($\operatorname{argmin}, \operatorname{argmax}$)
- Big AND/ Big OR ($\wedge$, $\vee$)

DP Cookbook

- Write recurrence
  - Choose the subject of recurrence
  - Base case(s)
  - Form <u>optimal</u> sub-<u>solution</u> (“up to this point”)
- Size of table (Dimensions? Range of each dimensions?)
- To fill in cell, which other cells do I look at?
- Which cell(s) contain the final answer?
- Reconstructing solution: Follow arrows from final answer to base case

Longest Increasing Subsequence (LIS)

- Given a sequence of numbers S, an increasing subsequence of S is a subsequence such that the elements are in strictly increasing order.
  - One LIS of S = [0, 8, 4, 12, 5, 6, 3] would be [0, 4, 5, 6]
- Problem: Return the length of LIS of a given sequence
- Q: Why can’t we solve this using divide and conquer?
  - Overlapping subproblem!

Longest Increasing Subsequence (LIS)

- Given a sequence of numbers S, an increasing subsequence of S is a subsequence such that the elements are in strictly increasing order.
  - One LIS of S = [0, 8, 4, 12, 5, 6, 3] would be [0, 4, 5, 6]
- Problem: Return the length of LIS of a given sequence
- **Step 1: Choosing subject of recurrence**
  - Attempt 1: L[i] = Length of LIS on subarray S[1, …, i]
  - Problem: The subsequence chosen can be *very* ambiguous
    - Example: In [0, 8, 4, 12, 5], L[5] = 3, do we want [0, 4, 5] or [0, 8, 12]?
    - Problematic because when we consider the next element, 6, we can only append it to [0, 4, 5] and have L[6] = 4. If my L[5] means [0, 8, 12], then my L[6] is still 3
    - Reminder: In DP, we don't want to re-solve the subproblem again

Longest Increasing Subsequence

- Given a sequence of numbers S, an increasing subsequence of S is a subsequence such that the elements are in strictly increasing order.
  - One LIS of S = [0, 8, 4, 12, 5, 6, 3] would be [0, 4, 5, 6]
- Problem: Return the length of LIS of a given sequence
- **Step 1: Choosing subject of recurrence**
  - Attempt 2: L[i] = Length LIS on subarray A[1, …, i] that ends at A[i]
    - Using the same example: [0, 8, 4, 12, 5], L[5] = 3
    - When we consider *any future* element at position k, we only have to check if it's greater than L[i] to decide if we want to append S[k] to that LIS ending at S[i]

Longest Increasing Subsequence

- Given a sequence of numbers S, an increasing subsequence of S is a subsequence such that the elements are in strictly increasing order.
  - One LIS of S = [0, 8, 4, 12, 5, 6, 3] would be [0, 4, 5, 6]
- Problem: Return the length of LIS of a given sequence
- **Step 2: Determining base case**
  - L[i] = Length LIS on subarray A[1, …, i] that ends at A[i]
  - Base case: i = 1: L[i] = 1 (only one element)

Longest Increasing Subsequence

- Given a sequence of numbers S, an increasing subsequence of S is a subsequence such that the elements are in strictly increasing order.
  - One LIS of S = [0, 8, 4, 12, 5, 6, 3] would be [0, 4, 5, 6]
- Problem: Return the length of LIS of a given sequence
- **Step 3: Forming optimal sub-solution**
  - L[i] = Length LIS on subarray A[1, …, i] that ends at A[i]
  - (1) We can append S[i] to *any* subsequence ending at S[j], j < i
  - (2) Only append if it is an increasing subsequence, i.e., S[j] < S[i]
  - Which subsequence to append to? The longest one! (across all j < i)

Longest Increasing Subsequence

- Given a sequence of numbers S, an increasing subsequence of S is a subsequence such that the elements are in strictly increasing order.
  - One LIS of S = [0, 8, 4, 12, 5, 6, 3] would be [0, 4, 5, 6]
- Problem: Return the length of LIS of a given sequence
- Putting everything together,

$$L(i) = 1 + \begin{cases} 0 & \text{if } i = 1 \\ \max_{j<i}\{L(j) : S[j] < S[i]\} & \text{otherwise} \end{cases}$$

- Or equivalently

$$L(i) = \begin{cases} 1 & \text{if } i = 1 \\ 1 + \max_{j<i}\{L(j) : S[j] < S[i]\} & \text{otherwise} \end{cases}$$

Longest Increasing Subsequence

- Given a sequence of numbers S, an increasing subsequence of S is a subsequence such that the elements are in strictly increasing order.
  - One LIS of S = [0, 8, 4, 12, 5, 6, 3] would be [0, 4, 5, 6]
- Problem: Return the length of LIS of a given sequence
- Recurrence relation:

$$L(i) = \begin{cases} 1 & \text{if } i = 1 \\ 1 + \max_{j<i}\{L(j) : S[j] < S[i]\} & \text{otherwise} \end{cases}$$

- What's next? From cookbook:
  - Size of table (Dimensions? Range of each dimensions?)
  - To fill in cell, which other cells do I look at?
  - Which cell(s) contain the final answer?

Longest Increasing Subsequence

- Given a sequence of numbers S, an increasing subsequence of S is a subsequence such that the elements are in strictly increasing order.
  - One LIS of S = [0, 8, 4, 12, 5, 6, 3] would be [0, 4, 5, 6]
- Problem: Return the length of LIS of a given sequence
- Recurrence relation:

$$L(i) = \begin{cases} 1 & \text{if } i = 1 \\ 1 + \max_{j<i}\{L(j) : S[j] < S[i]\} & \text{otherwise} \end{cases}$$

- What's next? From cookbook:
  - Size of table (Dimensions? Range of each dimensions?) n x 1
  - To fill in cell, which other cells do I look at? L[j] for all j < i
  - Which cell(s) contain the final answer? L[n]

Longest Increasing Subsequence

- Given a sequence of numbers S, an increasing subsequence of S is a subsequence such that the elements are in strictly increasing order.
  - One LIS of S = [0, 8, 4, 12, 5, 6, 3] would be [0, 4, 5, 6]
- Problem: Return the length of LIS of a given sequence
- Bottom-up solution:

$$L(i) = \begin{cases} 1 & \text{if } i = 1 \\ 1 + \max_{j<i}\{L(j) : S[j] < S[i]\} & \text{otherwise} \end{cases}$$

Q: How does the max expression translate to algorithm?

Use a variable to keep track of current max

LIS$(A[1..n])$: // table implementation of LCS
allocate $L[0..n]$
$L[0] \leftarrow 0$
for $i = 1..n$: // fill table
  $l \leftarrow 0$
  for $j = 1..i-1$:
    if $S[j] < S[i]$: $l \leftarrow \max\{l, L[j]\}$
  $L[i] \leftarrow l + 1$
return $\max_{1 \le i \le n} L[i]$

Runtime: $O(n^2)$

Worksheet: Binary Strings (if time)

Binary Strings

Let $\#C(\ell)$ denote the number of binary strings with length ℓ that have no consecutive occurrences of a 1. For example $\#C(3) = 5$; we can list all binary strings of length 3 and determine by inspection that only the strings 000, 001, 010, 100, and 101 have no consecutive occurrences of a 1.

(c) Show that $\#C(\ell) = \#C(\ell - 1) + \#C(\ell - 2)$ for $\ell \geq 2$. Derive a recursive algorithm to compute $\#C(\ell)$.

Hint: You will need to add base cases.

Solution: Observe that there for a string of length $\ell \geq 2$, it must either begin with a 1 or a 0. If it begins with a 1, then the next bit must be a 0, and the remaining string has length $\ell - 2$ and must have no consecutive occurrences of a 1. If it begins with a 0, then the next bit may be any bit, and the remaining string has length $\ell - 1$ and must have no consecutive occurrences of a 1.

As such, we have that $\#C(\ell) = \#C(\ell - 1) + \#C(\ell - 2)$ for $\ell \geq 2$.

If we want to compute this recursively, we only need to add base cases. Observe that $\#C$ is only defined for non-negative integers, so our base cases should be $\#C(0) = 1$ and $\#C(1) = 2$.

Input: Unsigned integer n

function $\#C(n)$
    if $n = 0$ **then**
        return 1
    if $n = 1$ **then**
        return 2
    return $\#C(n-1) + \#C(n-2)$
=0

Worksheet: Binary Strings (if time)

Binary Strings

Let $\#C(\ell)$ denote the number of binary strings with length ℓ that have no consecutive occurrences of a 1. For example $\#C(3) = 5$; we can list all binary strings of length 3 and determine by inspection that only the strings 000, 001, 010, 100, and 101 have no consecutive occurrences of a 1.

(d) Use the bottom-up-table approach to improve your recursive algorithm.

```
Input: Nonnegative integer n
function #C(n)
    DP ← a table of n integers
    DP[0] ← 1
    DP[1] ← 2
    for i = 2 to n do
        DP[i] ← DP[i − 1] + DP[i − 2]
    return DP[n]
```

Next Time

- More on Dynamic Programming
- Recursive top-down vs memorization vs bottom up