Week 4 Tuesday Lecture Notes

Before Class:

- **Readings 1 and 2** (You can also watch the videos)
- Introduction to **Dynamic Programming**
- Work on **Implementation 1 Due Thursday**.

Discussion Implementation 1:

- Make sure that the readme file explains how the TA can run the code.
- Asymptotic complexity (worst case scenario)
- Ask questions to TA/Instructor.
- Due Thursday at Midnight.

Fibonacci Numbers:

- Base Case: if $n = 0$ or 1.
- Else return fib-recur(n-1) + fib-recur(n-2)
- $T(n) = T(n-1) + T(n-2) + C$
- Run time?
- $O(2^n)$, $\Omega(2^{n/2})$
- **Slowest Speed:**
    - $T(n) <= T(n-1) + T(n-1) + C = 2T(n-1)+C$
    - Through Recursion = $T(n) = \mathbf{O(2^n)}$
- **Fastest Speed:**
    - $T(n) > T(n-2) + T(n-2) + C$
    - $2T(n-2)+C. = O(2^{n/2})$
    - Through Recursion = $T(n) = \mathbf{\Omega(2^{n/2})}$

Why is it so slow?

There's allot of repetition in this Algorithm.

Repeated Computation Slows down algorithm.

Memoization: a speed up technique that stores allot of repeated function calls.

The runtime for this is $O(n)$. Since you only need to compute each Fibonacci number once.

Iterative Version:

Bottom-up: iterative Version

```
function fib-iter(n)
  f[0]=0
  f[1]=1
  for i=2 to n
      f[i]=f[i-1]+f[i-2]
  return f[n]
```

Runtime?

- Similar to Memoization but also has a runtime of O(n)
- No advantage to run Recursive version over Iterative Version.

Dynamic Programming:

- Very Powerful.
- Common Interview Questions
- Core Technique for calculating runtime of an algorithm.
- Natural Language Parsing, Dynamic Programming is a key component.
- Really focused on “planning over time”.

When to use Dynamic Programming?

- **Optimal Substructures:** Solutions can be defined using solutions of smaller problems (Divide and Conquer)

- **Subproblems are overlapping**: Apparent repeated Sub-problems that are computed multiple times.

Designing with Dynamic Programming:

1. You have a big problem but the solver can only solve smaller problems. How can you use that small problem solver for the bigger problem? (Recursion)
2. Start Small and build up. (Bottom up Design = **Iterative**)
3. Might have to trace back to subproblems that were already computed (Memoization)

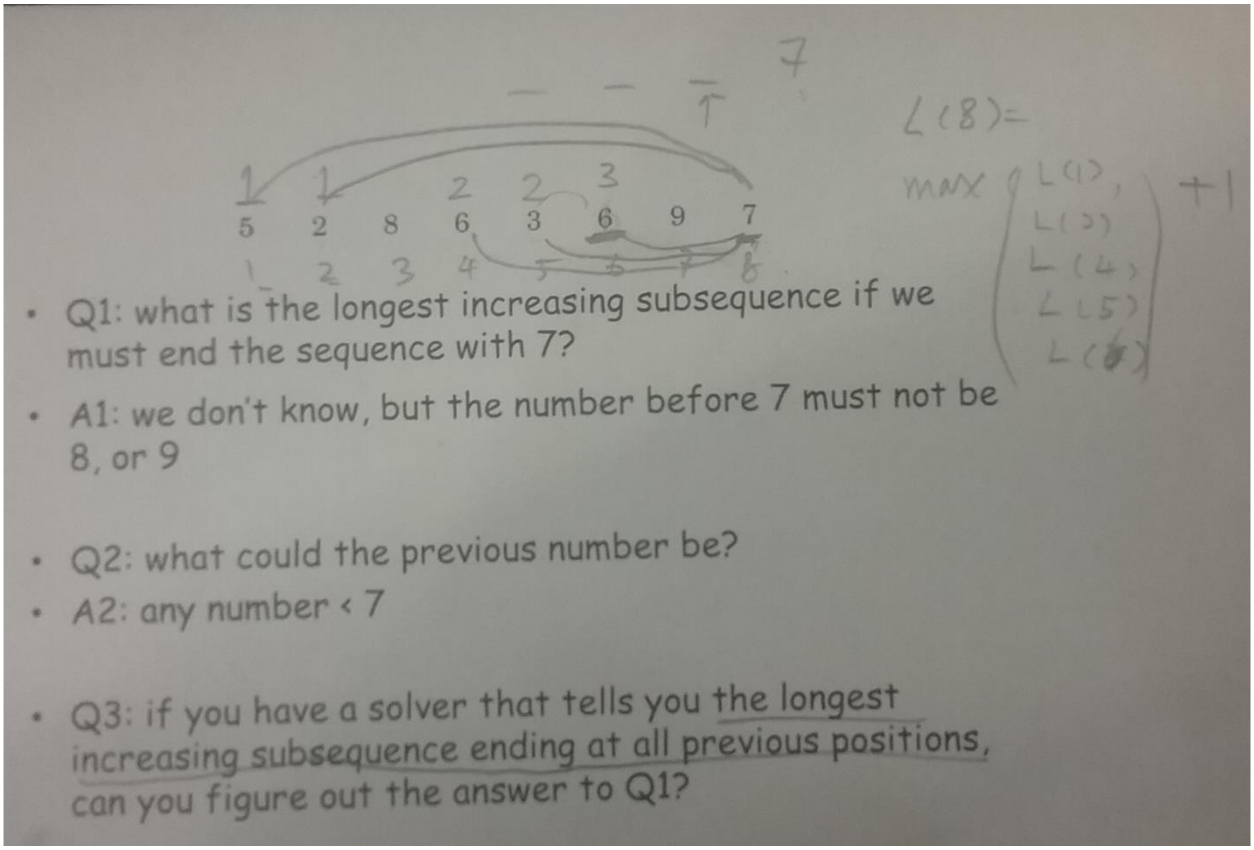
Longest Increasing Subsequences Example:

- Find the Longest Increasing Subsequence.
- Does not need to be contiguous.
- Example: we have a sequences (5, 2, 8, 4, 9)
- Increasing Subsequences include:
  - 5,8,9
  - 2,4,9
  - 2,8,9
  - 8,9
  - 4,9
  - Ect…
- There can be multiple subsequences with the longest series.
- Brute Force Approach: $O(2^n)$
- Can we do **Divide and Conquer**? No because we can jump across points and we are not solving for a contiguous series.

Longest Subsequence Example Continued: (5 2 8 6 3 6 9 7)

- What is we must end with a sequence with a specific number? (7)
- From this we know that any number larger than 7 can't be part of the subseries.
- If you have a solver that tells you the longest increasing subsequence **Ending at all previous positions**, can you figure out the longest subsequence ending at 7?

See Picture Example of Solver*



- **Optimal Substructure** solution that ends at 7 can be computed by the optimum solution ending at all of the previous numbers + 1.

Building the Solution to the Subsequence:

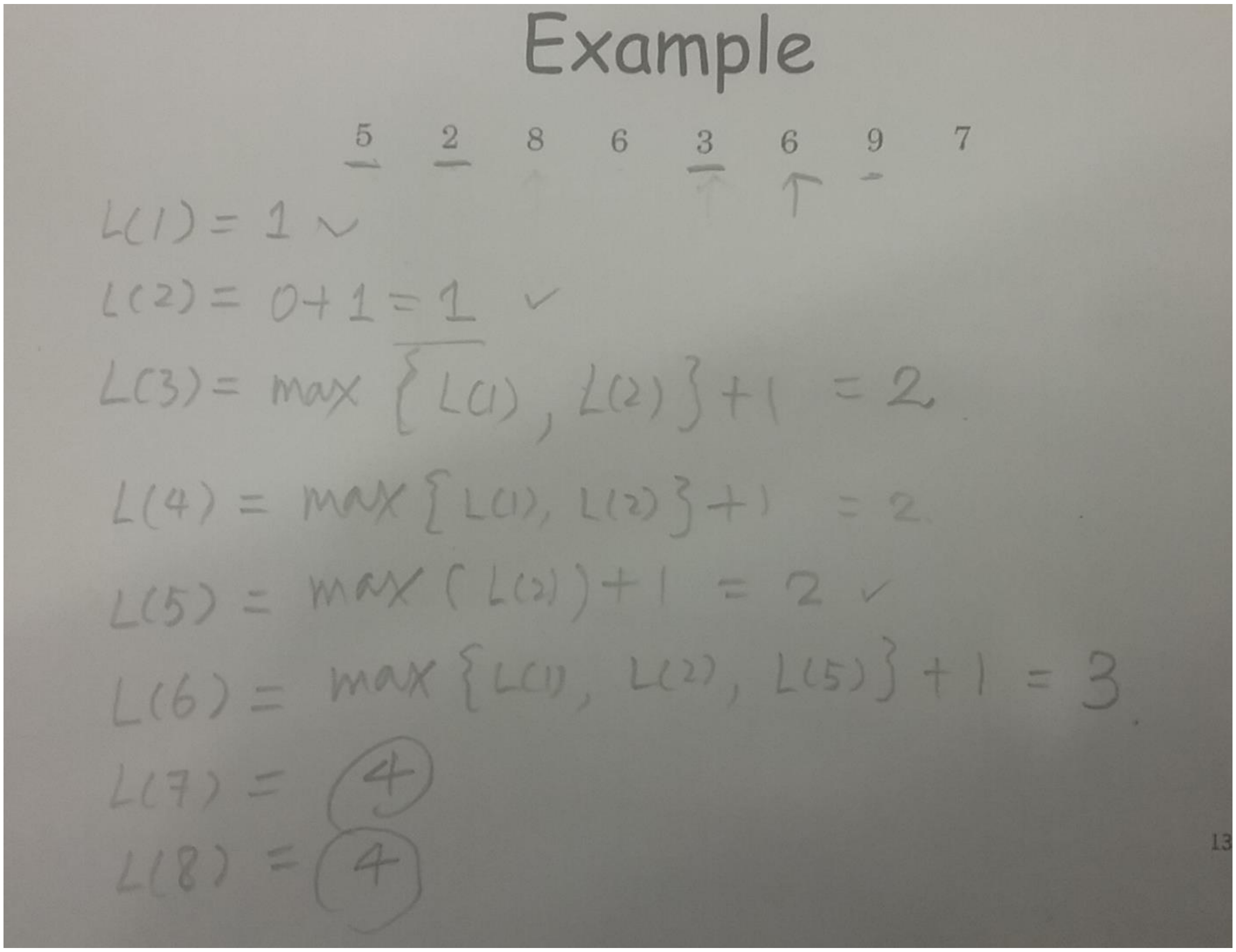
Building our solution

Let L[i] be the length of a longest increasing subsequence ending at position i

$$L[1] = 1$$

$$L[i] = \max_{j:1 \le j < i, a_j < a_i} L[j] + 1 \text{ for } i = 2, \dots, n$$

Overall solution: $\max_i L[i]$



This example is based on this example with the algorithm of Building Our Solution from the previous Page.

Our initial sequence of numbers is (5 2 8 6 3 6 9 7)

We are trying to compute the longest increasing sequence.

L(1) = 1

- We start with the very first value of the series: L(1).
- It contains the value 5.
- Since there are no previous sequences we can set the length up to this point as L[j]+1. There was no previous L[j]. So 0+1 = 1

L(2) = 0+1 = 1

- Since there are no smaller values of L before L(2). 5 is **NOT** less than 2. There are no subsequences that we can add this series to. L[j] = 0. 0+1 = 1.

Reminder: Our initial sequence of numbers is (5 2 8 6 3 6 9 7)

L(3) = MAX {L(1), L(2)} +1 = 2

- L(3) has a value of 8. 8 is Greater than both 5 and 2. Both L(1) and L(2) had a MAX length of 1. So L[j] = 1. 1+1 = 2.

This Trend continues for the rest of the values up to L(8) which contains the value 7.

The Longest increasing subsequence has a length of 4 and can be pathed in 2 different ways:

(2, 3, 6, 9) or (2, 3, 6, 7)

Iterative Algorithm:

Iterative algorithm

```
LIS (A,n)
L[1]=1
for i=2 to n
        L[i]=1
        for j=1 to i-1
                if a_j < a_i and L[i] < L[j]+1
                        L[i]= L[j]+1
Lis_max=1
for i=1 to n
        if L[i]>Lis_max  Lis_max = L[i]
Return Lis_max
```

$$L[i] = \max_{j:1 \le j < i, a_j < a_i} L[j] + 1$$

return $\max_i L[i]$

Run time?

- This runtime is n^2. Because you have to run through for each value of the series, and then again to compare to the max of each previous value.

Ending Notes:

- **Implementation Assignment 1:** Due Midnight on Thursday 2/2/17
- **Contact TA/Instructor** with questions about the implementation.
- **NO RECITATION** this week since there is no quiz (assignment instead)
- **Review** Dynamic Programming

Finished Dynamic Programming Lecture (W4D1)

<u>End of Week 4 Tuesday Lecture Notes</u>

~Information composed by Notetaker Scott Russell for CS 325 **DAS** student