Week 5 Day 2 Thurday Lecture Notes

Prep:

- Prepare for Quiz 3 (Knapsack, Dynamic Programming, edit distance)
- Recitation Answers have been posted on Canvas.
- Review Unbound Knapsack problem (where taking an item doesn't remove it from the list of available items to take) but instead only reducing the carry weight.
- Pseudocode for Knapsack.

Steps for Designing Dynamic Programming Algorithms:

1. Define the Subproblem, (usually an array of values to compute)
2. Describe how a Subproblem can be solved using smaller Subproblems (Recurrence relation)
3. Set up base cases, use recurrence relation to fill up table. (can also use recursion → Memoization)
   Review: Memoization is being able to cache data as you go to save the time to not solve the save problems over and over.
4. Return the solution of the filled array.

Revisit: Maximum Subarray

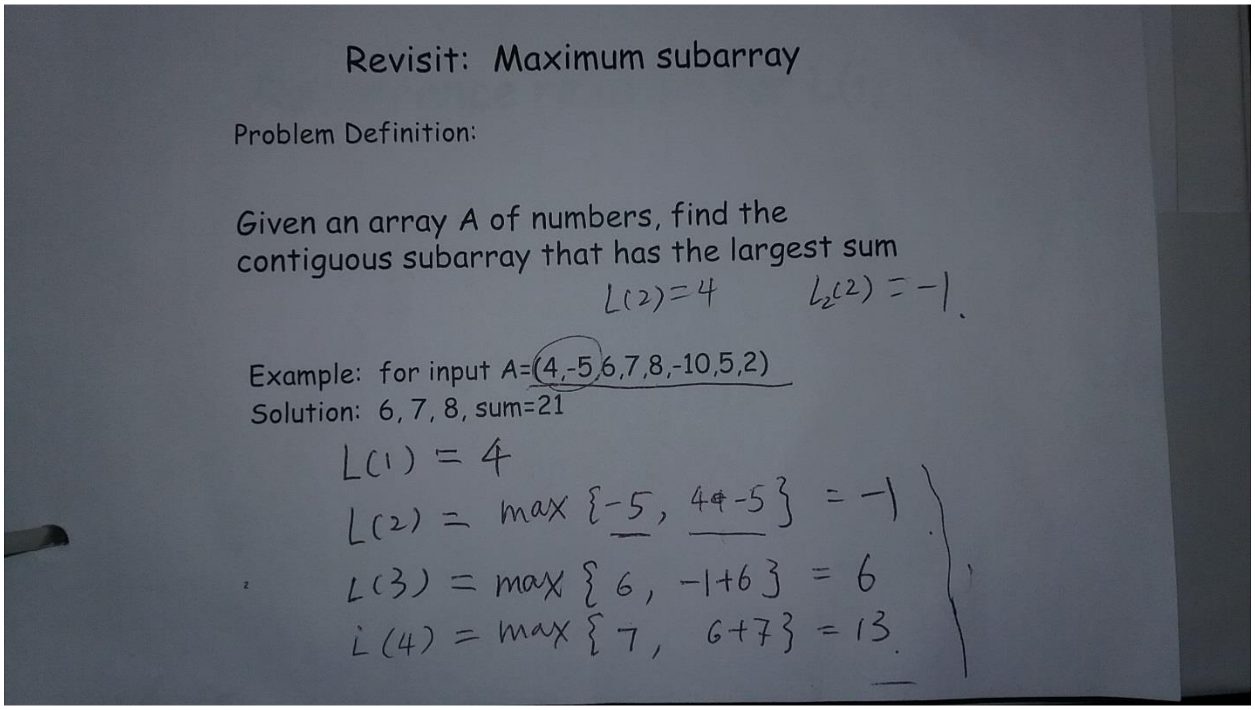
- Given an array A of numbers, find the largest contiguous subarray that has the largest sum.
- Example input: A= (4,-5,6,7,8,-10,5,2)
- Solution: 6,7,8. Giving a Sum of 21.
- Brute Force algorithm. n^2 runtime
- Divide and Conquer: n log n. (divide in half, solve left/right/ and solve the cross part)
- What about a Dynamic Programming Solution?

Two Possible Subproblems:

1. **L(i) = Max subarray for A[1,...,i]**
2. **L(i) = Max subarray ending at position i.**

L(i) Max sum ending at A[i]

MAX of L(i) = A[i] + L(i-1) || A[i]



Revisit: Maximum subarray

Problem Definition:

Given an array A of numbers, find the contiguous subarray that has the largest sum

$L(2)=4$ $L_2(2)=-1$.

Example: for input A=(4,-5,6,7,8,-10,5,2)
Solution: 6, 7, 8, sum=21

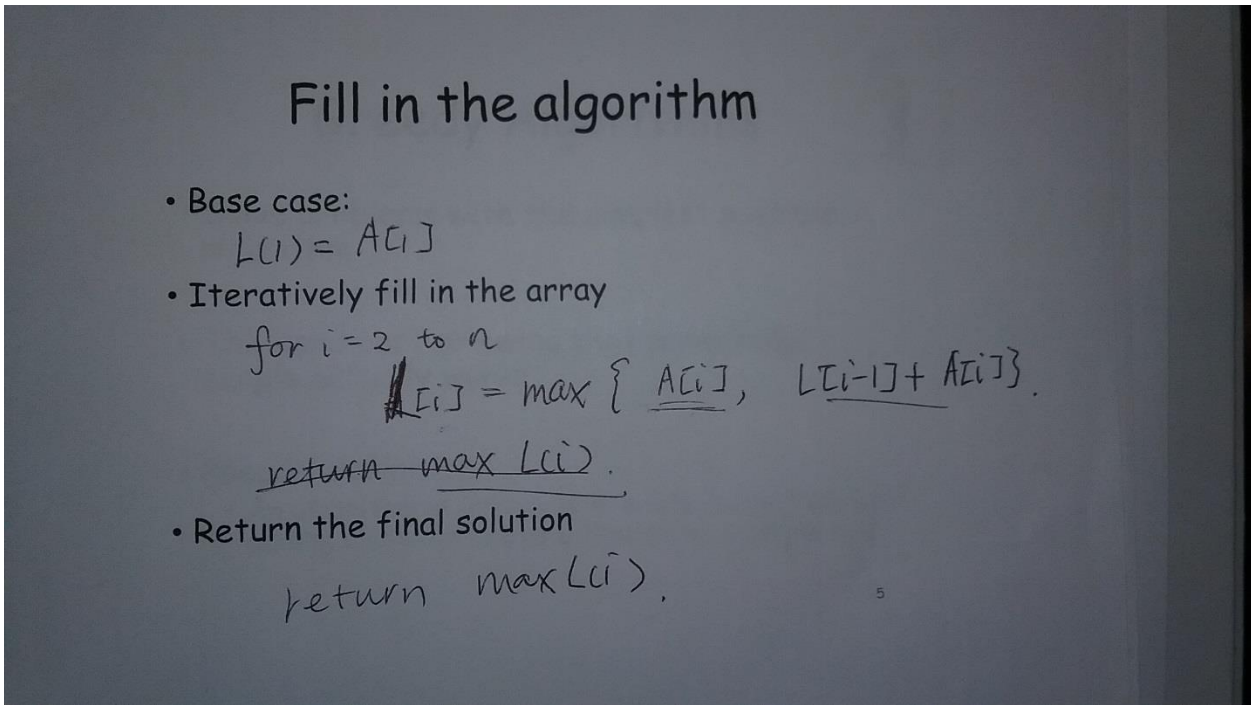
$L(1)=4$

$L(2)=\max\{\underline{-5},\ \underline{4-5}\}=-1$

$L(3)=\max\{6,\ -1+6\}=6$

$L(4)=\max\{7,\ 6+7\}=13$.

Picture of Algorithm. **Fill in the Algorithm**



Fill in the algorithm

- Base case:
  $L(1)=A[1]$
- Iteratively fill in the array

```
for $i=2$ to $n$
        $L[i]=\max\{\underline{A[i]},\ \underline{L[i-1]+A[i]}\}$.
~~return~~ max L(i).
```

- Return the final solution

  return max L(i).

5

End of Material for Lecture 1

Greedy Algorithms (new Lecture)

- Solve problems with the simplest possible algorithm
- Prove that sometimes the greedy solution gives the most optimal solution.

Greedy Algorithm: if it builds its solution by adding one element at a time using a simple rule.

Interval Scheduling:

Given a set of n tasks.

i-th task start and finish time: s(i), f(i)

two tasks are compatible if they don't overlap.

Goal: find the maximum number of mutually compatible tasks.

Just care about the number of jobs. (maximum number of jobs that can be run)

Greedy Template:

- Let T be a set of Tasts.
- Set of independent tasks i
- A is the rule determining the greedy algorithm

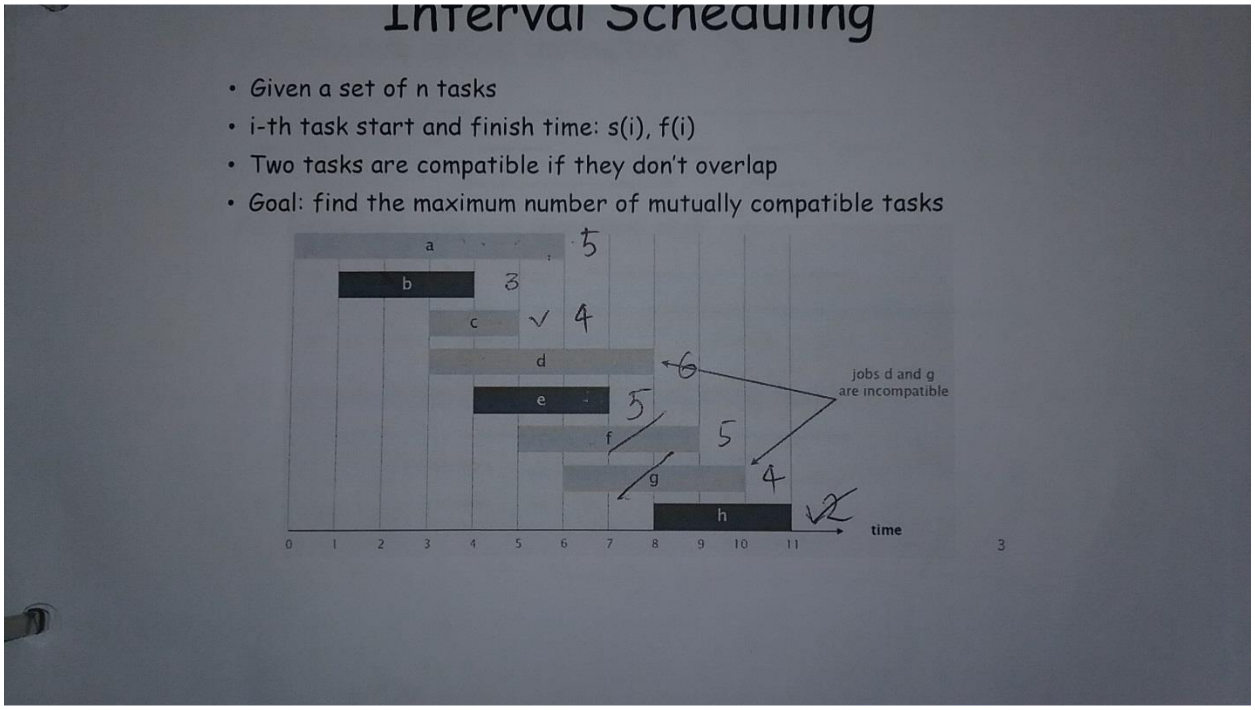
Pseudocode:

```
I = {}
While (T is not empty)
    Select a task t from T by a rule A
    Add t to I
    Remove t and all tasks incompatible with t from T.
```

3 Different Greedy Algorithms:

1. Schedule Earliest staring task (bad)
2. Select Shortest Available task (bad)
3. Select tasks with the fewest conflicting tasks (bad)

- None of these greedy algorithms gives the optimal solution



Interval Scheduling

- Given a set of n tasks
- i-th task start and finish time: s(i), f(i)
- Two tasks are compatible if they don't overlap
- Goal: find the maximum number of mutually compatible tasks

a
5
b
3
c
✓ 4
d
6
e
5
f
5
g
4
h
✓2
jobs d and g are incompatible
time
0 1 2 3 4 5 6 7 8 9 10 11
3

Next Time:

- Review for **Midterm**
- Midterm **Thursday Next week**
- Email Professor about questions.
- **Assignment 2** will be posted soon. (due in 2 weeks)

Ended on Greedy Algorithm Lecture.

<u>**End of Week 5 Day 2Thursday Lecture Notes**</u>

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