COL351: Slides for Lecture Component 23

Thanks to Miles Jones, Russell Impagliazzo, and Sanjoy Dasgupta at UCSD for these slides.

DYNAMIC PROGRAMMING

Dynamic programming is an algorithmic paradigm in which a problem is solved by:

Identifying a collection of subproblems.

Tackling them one by one, smallest first, using the answers to small problems to help figure out larger ones, until they are all solved.

DP STEPS (BEGINNER)

1. Design simple backtracking algorithm
2. Characterize subproblems that can arise in backtracking
3. Simulate backtracking algorithm on subproblems
4. Define array/matrix to hold different subproblems
5. Translate recursion from step 3 in terms of matrix positions: Recursive call becomes array position; return becomes write to array position
6. Invert top-down recursion order to get bottom up order

7: Assemble: Fill in base cases
   In bottom-up order do:
      Use step 5 to fill in each array position
   Return array position corresponding to whole input

DYNAMIC PROGRAMMING STEPS (EXPERT)

Step1: Define the subproblems

Step 2: Define the base cases

Step 3: Express subproblems recursively

Step 4: Order the subproblems

EITHER WAY

1. You MUST explain what each cell of the table/matrix means AS a solution to a subproblem.

That is, clearly define the subproblems.

2. You MUST explain what the recursion is in terms of a LOCAL, COMPLETE case analysis.

That is, explain how subproblems are solved using other, “smaller”, subproblems.

Undocumented dynamic programing is indistinguishable from nonsense. Assumptions about optimal solution almost always wrong.

LONGEST INCREASING SUBSEQUENCE

Given a sequence of distinct positive integers a[1],…,a[n]

An increasing subsequence is a sequence $a[i_1],\ldots,a[i_k]$ such that $i_1<\ldots<i_k$ and $a[i_1]<\ldots<a[i_k]$.

For example: 15, 18, 8, 11, 5, 12, 16, 2, 20, 9, 10, 4

5, 16, 20 is an increasing subsequence.

How long is the longest increasing subsequence?

DYNAMIC PROGRAMMING: EXPERT MODE

What is a suitable notion of subproblem?

For example: 15, 18, 8, 11, 5, 12, 16, 2, 20, 9, 10, 4

DYNAMIC PROGRAMMING: EXPERT MODE

Step1: Define the subproblems

L(k) = length of the longest increasing subsequence ending exactly at position k

Step 2: Base Case

L(1)=1

Step 3: Express subproblems recursively

$L(k) = 1+\max(\{L(i): i < k, a_i < a_k\})$

Step 4: Order the subproblems

Solve them in the order L(1), L(2), L(3), …

Try it out!   a = [15, 18, 8, 11, 5, 12, 16, 2, 20, 9, 10, 4].

LONGEST INCREASING SUBSEQUENCE

Subproblem: L[k] = length of LIS ending exactly at position k

```
L[1] = 1
For k = 2 to n:
        Len = 1
        For i = 1 to k-1:
                If a[i] < a[k] and Len < 1+L[i]:
                        Len = 1+L[i]
        L[k] = Len
return max(L[1], L[2], …, L[n])
```

LONGEST INCREASING SUBSEQUENCE

Given a sequence of distinct positive integers a[1],…,a[n]

An increasing subsequence is a sequence $a[i_1],\ldots,a[i_k]$ such that $i_1<\ldots<i_k$ and $a[i_1]<\ldots<a[i_k]$.

For example: 15, 18, 8, 11, 5, 12, 16, 2, 20, 9, 10, 4

5, 16, 20 is an increasing subsequence.

How long is the longest increasing subsequence?

THE LONG WAY

1. Come up with simple backtracking algorithm
2. Characterize subproblems
3. Define matrix to store answers to the above
4. Simulate BT algorithm on subproblem
5. Replace recursive calls with matrix elements
6. Invert "top-down" order of BT to get "bottom-up" order
7. Assemble into DP algorithm:
   - Fill in base cases into matrix in bottom-up order
   - Use translated recurrence to fill in each matrix element
   - Return "main problem" answer
   - (Trace-back to get corresponding solution)

LONGEST INCREASING SUBSEQUENCE

What is a **local decision**?

More than one possible answer…

LONGEST INCREASING SUBSEQUENCE

What is a local decision?

Version 1: For each element, is it in the subsequence?
Possible answers: Yes, No

Version 2: What is the first element in the subsequence? The second?
Possible answers: 1…n.

Either way, we need to generalize the problem a bit to solve recursively.

FIRST CHOICE, RECURSION

Assume we're only allowed to use entries bigger than V.

(Initially, set V=-1, and branch on whether or not to include A[1].)

We'll just return the length of the LIS.

```
BTLIS1(V, A[1…n])
    If n=0 then return 0
    If n=1 then if A[1] > V then return 1 else return 0
    OUT:= BTLIS(V, A[2..n])  {if we do not include A[1]}
    IF A[1] > V then IN:= 1+BTLIS(A[1],A[2..n]) else IN:= 0
    Return max (IN, OUT)
```

EXAMPLE

A[1:12] = [15, 18, 8, 11, 5, 12, 16, 2, 20, 9, 10, 4]

WHAT DO SUBPROBLEMS LOOK LIKE?

Arrays in subcalls are:

V in subcalls are:

Total number of distinct subcalls:

SUBPROBLEMS

Array A[J..n], where J ranges from 1 to n

V is either -1 or of the form A[K]

To simplify things, define A[0] = -1

Define

L[K,J] = (length of) LIS of A[J..n], with elements > A[K]

SIMULATING RECURRENCE

```
BTLIS(A[K], A[J…n])
        If J=n then if A[K] < A[n] return 1 else return 0
        OUT:= BTLIS(A[K], A[J+1..n])
        IF A[J] > A[K] then IN:= 1 + BTLIS(A[J], A[J+1..n]) else IN:= 0
        Return max (IN, OUT)
```

TRANSLATE RECURRENCE IN TERMS OF MATRIX

```
BTLIS(A[K], A[J…n])
        If J=n then if A[K] < A[n] return 1 else return 0
        OUT:= BTLIS(A[K], A[J+1..n])
        IF A[J] > A[K] then IN:= 1 + BTLIS(A[J], A[J+1..n]) else IN:= 0
        Return max (IN, OUT)
```

Recall: L[K,J] = (length of) LIS of A[J..n], with elements > A[K]

```
        If A[K] < A[n] then L[K,n] := 1 else L[K,n]:=0
        OUT: = L[K,J+1]
        IF A[J] > A[K] then IN:= 1 + L[J,J+1] else IN: = 0
        L[K,J]:= max (IN, OUT)
```

INVERT TOP-DOWN ORDER TO GET BOTTOM-UP ORDER

Recall: L[K,J] = (length of) LIS of A[J..n], with elements > A[K]

As we recurse, J gets incremented, K sometimes increases

Bottom-up: J gets decremented, K any order

FILL IN MATRIX IN BOTTOM UP ORDER

```
A[0] := -1
For K=0 to n-1 do:
        IF A[n] > A[K] then L[K,n] := 1 else L[K,n] := 0
For J=n-1 downto 1 do:
        For K=0 to J-1 do:
                OUT := L[K, J+1]
                IF A[J] > A[K] then IN := 1 + L[J,J+1] else IN := 0
                L[K,J] := max(IN, OUT)
Return L[0,1]
```

Recall: L[K,J] = (length of) LIS of A[J..n], with elements > A[K]

EXAMPLE

A[0:4] = [-1, 15, 8, 11, 2]

	1	**2**	**3**	**4**
0				
1				
2				
3				

Recall: L[K,J] = (length of) LIS of A[J..n], with elements > A[K]

TIME ANALYSIS

```
A[0] := -1
For K=0 to n-1 do:
        IF A[n] > A[K] then L[K,n] := 1 else L[K,n] := 0
For J=n-1 downto 1 do:
        For K=0 to J-1 do:
                OUT := L[K, J+1]
                IF A[J] > A[K] then IN := 1 + L[J,J+1] else IN := 0
                L[K,J] := max(IN, OUT)
Return L[0,1]
```

LONGEST INCREASING SUBSEQUENCE

What is a local decision?

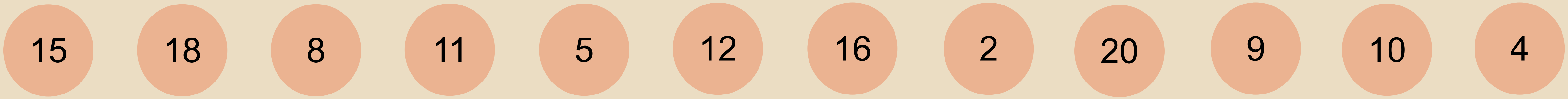
Version 1: For each element, is it in the subsequence?
Possible answers: Yes, No

Version 2:   What is the first element in the subsequence? The second?
Possible answers: 1…n.

Either way, we need to generalize the problem a bit to solve recursively.

ANOTHER VIEW OF LONGEST INCREASING SUBSEQUENCE

Let’s make a DAG out of our example…



WHY DAGS ARE CANONICAL FOR DP

Consider a graph whose vertices are the distinct recursive calls an algorithm makes, and where calls are edges from the subproblem to the main problem.

This graph had better be a DAG or we're in deep trouble!

This graph should be small or DP won't help much.

Bottom-up order = topological sort

BT TO DP: TREES TO DAGS

BT:

Create a tree of possible subproblems, where branching is based on all consistent next choices for local searches

DP:

Make this tree into a DAG by identifying paths that lead to same problems.

Array indices = names for vertices in this DAG

Expert's method: Skip directly to DAG.

VERSION 2, BACKTRACKING

If the current position we've chosen is A[J], what is the next choice?

Possibilities: J+1,…n, none (need to check greater than A[J])

Again, set A[0]=-1 and start J=0

Only counting choices after A[J]

```
BTLIS2(A[J…n]) {LIS of A[J+1..n], assuming we've taken A[J]}
        IF n=J return 0
        Max := 0
        FOR K=J+1 TO n do:
                IF A[K] > A[J] THEN:
                        L:= BTLIS2(A[K..n])
                        IF Max < 1+L THEN Max := 1+L
        Return Max
```

WHAT ARE THE SUB-PROBLEMS?

```
BTLIS2(A[J...n]) {LIS of A[J+1..n], assuming we've taken A[J]}
        IF n=J return 0
        Max := 0
        FOR K=J+1 TO n do:
                IF A[K] > A[J] THEN:
                        L:= BTLIS2(A[K..n])
                        IF Max < 1+L THEN Max := 1+L
        Return Max
```

Again, set A[0]=-1 and start J=0

What are the distinct recursive calls we make throughout this algorithm?

DEFINE ARRAY AND TRANSLATE

Let M[J] = BTLIS2(A[J..n]), J=0…n

REPLACE RECURSION WITH ARRAY

```
BTLIS2(A[J…n]) {LIS of A[J+1..n], assuming we've taken A[J]}
        IF n=J return 0
        Max := 0
        FOR K=J+1 TO n do:
                IF A[K] > A[J] THEN:
                        L:= BTLIS2(A[K..n])
                        IF Max < 1+L THEN Max := 1+L
        Return Max
```

```
M[n] := 0
For J in 0 to n-1:
        Max:=0
        FOR K=J+1 TO n do:
                IF A[K] > A[J] THEN:
                        L:= M[K]
                        IF Max < 1+L THEN Max:= 1+L
        M[J]:= Max
```

IDENTIFY TOP DOWN ORDER

When we make recursive calls, J is:

So bottom up order means J is:

FILL IN ARRAY IN BOTTOM-UP ORDER

```
DPLIS2(A[1..n])
        A[0] :=-1
        M[n] := 0
        FOR J=n-1 downto 0 do:
                Max := 0
                FOR K=J+1 TO n do:
                        IF A[K] > A[J] THEN:
                                L:= M[K]
                                IF Max < 1+L THEN Max:= 1+L
                M[J] := Max
        Return M[0]
```

Recall: M[J] = (length of) LIS of A[J+1..n], assuming we've taken A[J]

EXAMPLE

A: -1, 15, 18, 8, 11, 5, 12, 16, 2, 20, 9, 10, 4

Recall: M[J] = (length of) LIS of A[J+1..n], assuming we've taken A[J]

TIME ANALYSIS

```
DPLIS2(A[1..n])
    A[0] :=-1
    M[n] := 0
    FOR J=n-1 downto 0 do:
        Max := 0
        FOR K=J+1 TO n do:
            IF A[K] > A[J] THEN:
                L:= M[K]
                IF Max < 1+L THEN Max:= 1+L
        M[J] := Max
    Return M[0]
```

CORRECTNESS

Invariant:

M[J] is length of increasing sequence from A[J+1…n] with elements greater than A[J]

Strong induction on n-J

Base case: When J=n, no choices possible, M[n] = 0

Induction step: We try all possible values for first element.