

Dynamic Programming

Lecture 8

February 15, 2011

Part I

Longest Increasing Subsequence

Sequences

Definition

Sequence: an ordered list a_1, a_2, \dots, a_n . **Length** of a sequence is number of elements in the list.

Definition

a_{i_1}, \dots, a_{i_k} is a **subsequence** of a_1, \dots, a_n if $1 \leq i_1 < \dots < i_k \leq n$.

Definition

A sequence is **increasing** if $a_1 < a_2 < \dots < a_n$. It is **non-decreasing** if $a_1 \leq a_2 \leq \dots \leq a_n$. Similarly **decreasing** and **non-increasing**.

Sequences

Example...

Example

- Sequence: 6, 3, 5, 2, 7, 8, 1
- Subsequence: 5, 2, 1
- Increasing sequence: 3, 5, 9
- Increasing subsequence: 2, 7, 8

Longest Increasing Subsequence Problem

Input A sequence of numbers a_1, a_2, \dots, a_n

Goal Find an *increasing subsequence* $a_{i_1}, a_{i_2}, \dots, a_{i_k}$ of maximum length

Example

- Sequence: 6, 3, 5, 2, 7, 8, 1
- Increasing subsequences: 6, 7, 8 and 3, 5, 7, 8 and 2, 7 etc
- Longest increasing subsequence: 3, 5, 7, 8

Naïve Enumeration

Assume a_1, a_2, \dots, a_n is contained in an array A

```
algLISNaive( $A[1..n]$ ):  
   $max = 0$   
  for each subsequence  $B$  of  $A$  do  
    if  $B$  is increasing and  $|B| > max$  then  
       $max = |B|$   
  
  Output  $max$ 
```

Running time: $O(n2^n)$.

2^n subsequences of a sequence of length n and $O(n)$ time to check if a given sequence is increasing.

Recursive Approach: Take 1

LIS: Longest increasing subsequence

Can we find a recursive algorithm for LIS?

LIS($\mathbf{A}[1..n]$):

- Case 1: does not contain a_n in which case
 $\text{LIS}(\mathbf{A}[1..n]) = \text{LIS}(\mathbf{A}[1..(n-1)])$
- Case 2: contains a_n in which case $\text{LIS}(\mathbf{A}[1..n])$ is not so clear.

Observation: if a_n is in the longest increasing subsequence then all the elements before it must be smaller.

Recursive Approach: Take 1

```
algLIS( $\mathbf{A}[1..n]$ ):  
  if ( $n = 0$ ) then return 0  
   $m = \text{algLIS}(\mathbf{A}[1..(n-1)])$   
   $B$  is subsequence of  $\mathbf{A}[1..(n-1)]$  with  
    only elements less than  $a_n$   
  (* let  $h$  be size of  $B$ ,  $h \leq n-1$  *)  
   $m = \max(m, 1 + \text{algLIS}(B[1..h]))$   
  Output  $m$ 
```

Recursion for running time: $T(n) \leq 2T(n-1) + O(n)$.
Easy to see that $T(n)$ is $O(n2^n)$.

Recursive Approach: Take 2

LIS($A[1..n]$):

- Case 1: does not contain a_n in which case
 $\text{LIS}(A[1..n]) = \text{LIS}(A[1..(n-1)])$
- Case 2: contains a_n in which case $\text{LIS}(A[1..n])$ is not so clear.

Observation

For second case we want to find a subsequence in $A[1..(n-1)]$ that is restricted to numbers less than a_n . This suggests that a more general problem is **LIS_smaller**($A[1..n], x$) which gives the longest increasing subsequence in A where each number in the sequence is less than x .

Recursive Approach: Take 2

LIS_smaller($A[1..n], x$) : length of longest increasing subsequence in $A[1..n]$ with all numbers in subsequence less than x

```
LIS_smaller( $A[1..n], x$ ) :  
  if ( $n = 0$ ) then return 0  
   $m = \text{LIS\_smaller}(A[1..(n-1)], x)$   
  if ( $A[n] < x$ ) then  
     $m = \max(m, 1 + \text{LIS\_smaller}(A[1..(n-1)], A[n]))$   
  Output  $m$ 
```

```
LIS( $A[1..n]$ ) :  
  return LIS_smaller( $A[1..n], \infty$ )
```

Recursion for running time: $T(n) \leq 2T(n-1) + O(1)$.

Question: Is there any advantage?

Recursive Algorithm: Take 2

Observation: The number of *different* subproblems generated by **LIS_smaller**($A[1..n], x$) is $O(n^2)$. Memoization the recursive algorithm leads to an $O(n^2)$ running time!

Question: What are the recursive subproblem generated by **LIS_smaller**($A[1..n], x$)?

- For $0 \leq i < n$ **LIS_smaller**($A[1..i], y$) where y is either x or one of $A[i + 1], \dots, A[n]$.

Observation: previous recursion also generates only $O(n^2)$ subproblems. Slightly harder to see.

Recursive Algorithm: Take 3

LISEnding($A[1..n]$): length of longest increasing sub-sequence that *ends* in $A[n]$.

Question: can we obtain a recursive expression?

$$\mathbf{LISEnding}(A[1..n]) = \max_{i:A[i] < A[n]} (1 + \mathbf{LISEnding}(A[1..i]))$$

Recursive Algorithm: Take 3

```
LIS_ending(A[1..n]):  
  if (n = 0) return 0  
  m = 1  
  for i = 1 to n - 1 do  
    if (A[i] < A[n]) then  
      m = max(m, 1 + LIS_ending(A[1..i]))  
  
  return m
```

```
LIS(A[1..n]):  
  return maxi=1n LIS_ending(A[1...i])
```

Question:

How many distinct subproblems generated by **LISEnding**(**A**[1..*n*])?
n.

Iterative Algorithm via Memoization

Compute the values **LISEnding**(**A**[1..*i*]) iteratively in a bottom up fashion.

```
LIS_ending(A[1..n]):  
  Array L[1..n]  
  (* L[i] = value of LIS_ending(A[1..i]) *)  
  for i = 1 to n do  
    L[i] = 1  
    for j = 1 to i - 1 do  
      if (A[j] < A[i]) do  
        L[i] = max(L[i], 1 + L[j])  
  return L
```

```
LIS(A[1..n]):  
  L = LIS_ending(A[1..n])  
  return the maximum value in L
```

Iterative Algorithm via Memoization

Simplifying:

```
LIS(A[1..n]):  
  Array L[1..n]  
    (* L[i] stores the value LIS-Ending(A[1..i]) *)  
  m = 0  
  for i = 1 to n do  
    L[i] = 1  
    for j = 1 to i - 1 do  
      if (A[j] < A[i]) do  
        L[i] = max(L[i], 1 + L[j])  
    m = max(m, L[i])  
  return m
```

Correctness: Via induction following the recursion

Running time: $O(n^2)$, **Space:** $\Theta(n)$

Example

Example

- Sequence: 6, 3, 5, 2, 7, 8, 1
- Longest increasing subsequence: 3, 5, 7, 8

- $L[i]$ is value of longest increasing subsequence ending in $A[i]$
- Recursive algorithm computes $L[i]$ from $L[1]$ to $L[i-1]$
- Iterative algorithm builds up the values from $L[1]$ to $L[n]$

Memoizing LIS_smaller

```
LIS( $A[1..n]$ ):  
   $A[n+1] = \infty$  (* add a sentinel at the end *)  
  Array  $L[(n+1), (n+1)]$  (* two-dimensional array*)  
    (*  $L[i, j]$  for  $j \geq i$  stores the value LIS_smaller( $A[1..i], A[j]$ ) *)  
  for  $j = 1$  to  $n+1$  do  
     $L[0, j] = 0$   
  for  $i = 1$  to  $n+1$  do  
    for  $j = i$  to  $n+1$  do  
       $L[i, j] = L[i-1, j]$   
      if ( $A[i] < A[j]$ ) then  
         $L[i, j] = \max(L[i, j], 1 + L[i-1, j])$   
  
  return  $L[n, (n+1)]$ 
```

Correctness: Via induction following the recursion (take 2)

Running time: $O(n^2)$, **Space:** $\Theta(n^2)$

Longest increasing subsequence

Another way to get quadratic time algorithm

- $G = (\{s, 1, \dots, n\}, \{\})$: directed graph.
 - $\forall i, j$: If $i < j$ and $A[i] < A[j]$ then add the edge $i \rightarrow j$ to G .
 - $\forall i$: Add $s \rightarrow i$.
- The graph G is a **DAG**. **LIS** corresponds to longest path in G starting at s .
- We know how to compute this in $O(|V(G)| + |E(G)|) = O(n^2)$.

Comment: One can compute **LIS** in $O(n \log n)$ time with a bit more work.

- Find a “smart” recursion for the problem in which the number of distinct subproblems is small; polynomial in the original problem size.
- Eliminate recursion and find an iterative algorithm to compute the problems bottom up by storing the intermediate values in an appropriate data structure; need to find the right way or order the subproblem evaluation.
- Estimate the number of subproblems, the time to evaluate each subproblem and the space needed to store the value. Evaluate the total running time.
- Optimize the resulting algorithm further

Part II

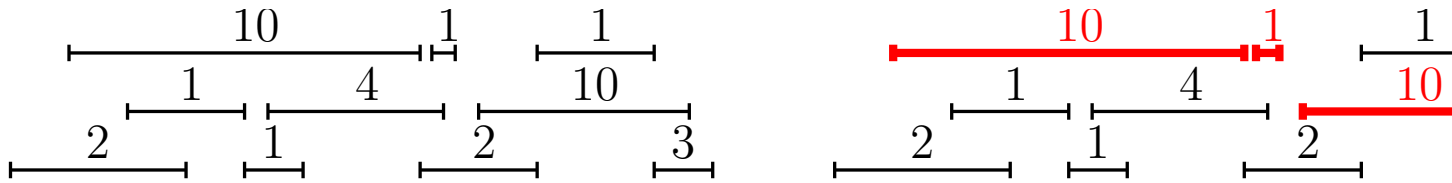
Weighted Interval Scheduling

Weighted Interval Scheduling

Input A set of jobs with start times, finish times and *weights* (or profits).

Goal Schedule jobs so that total weight of jobs is maximized.

- Two jobs with overlapping intervals cannot both be scheduled!



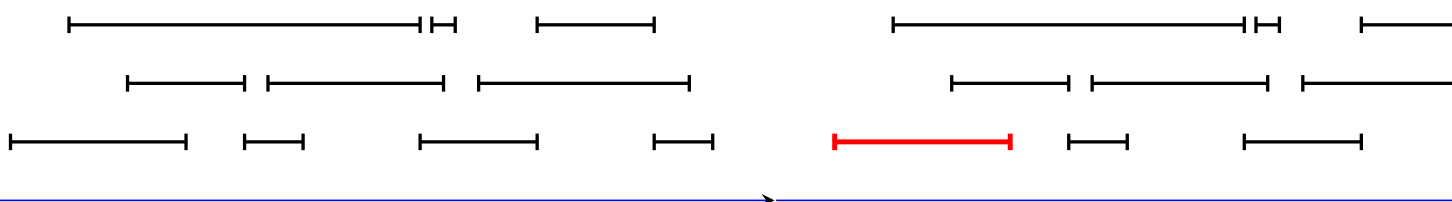
Interval Scheduling

Greedy Solution

Input A set of jobs with start and finish times to be scheduled on a resource; special case where all jobs have weight **1**.

Goal Schedule as many jobs as possible.

- Greedy strategy of considering jobs according to finish times produces optimal schedule (to be seen later).



Greedy Strategies

- Earliest finish time first
- Largest weight/profit first
- Largest weight to length ratio first
- Shortest length first
- ...

None of the above strategies lead to an optimum solution.

Moral: Greedy strategies often don't work!

Reduction to...

Max Weight Independent Set Problem

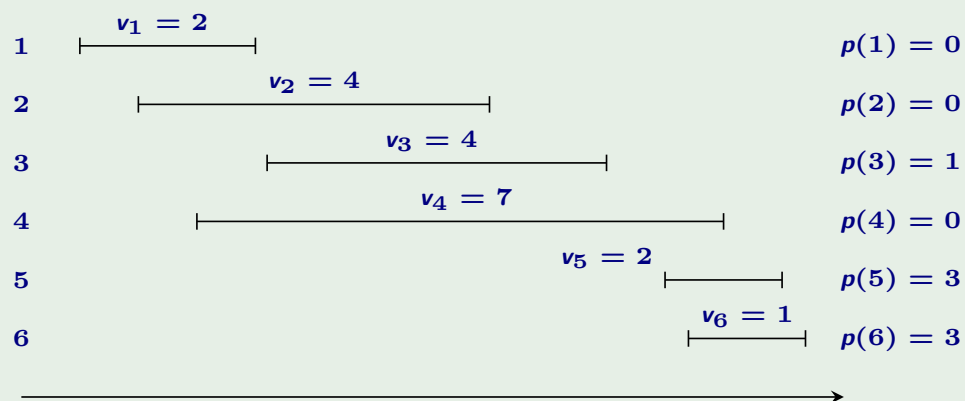
- Given weighted interval scheduling instance I create an instance of max weight independent set on a graph $G(I)$ as follows.
 - For each interval i create a vertex v_i with weight w_i .
 - Add an edge between v_i and v_j if i and j overlap.
- **Claim:** max weight independent set in $G(I)$ has weight equal to max weight set of intervals in I that do not overlap
- **Independent Set** in general is **NP-COMplete**.
- Can use structure of original problem for efficient algorithm?

Conventions

Definition

- Let the requests be sorted according to finish time, i.e., $i < j$ implies $f_i \leq f_j$
- Define $p(j)$ to be the largest i (less than j) such that job i and job j are not in conflict

Example



Towards a Recursive Solution

Observation

Consider an optimal schedule \mathcal{O}

Case $n \in \mathcal{O}$: None of the jobs between n and $p(n)$ can be scheduled. Moreover \mathcal{O} must contain an optimal schedule for the first $p(n)$ jobs.

Case $n \notin \mathcal{O}$: \mathcal{O} is an optimal schedule for the first $n - 1$ jobs.

A Recursive Algorithm

Let O_i be value of an optimal schedule for the first i jobs.

```
Schedule( $n$ ):  
  if  $n = 0$  then return 0  
  if  $n = 1$  then return  $w(v_1)$   
   $O_{p(n)} \leftarrow$  Schedule( $p(n)$ )  
   $O_{n-1} \leftarrow$  Schedule( $n - 1$ )  
  if ( $O_{p(n)} + w(v_n) < O_{n-1}$ ) then  
     $O_n = O_{n-1}$   
  else  
     $O_n = O_{p(n)} + w(v_n)$   
  return  $O_n$ 
```

Time Analysis

Running time is $T(n) = T(p(n)) + T(n - 1) + O(1)$ which is ...

Bad Example

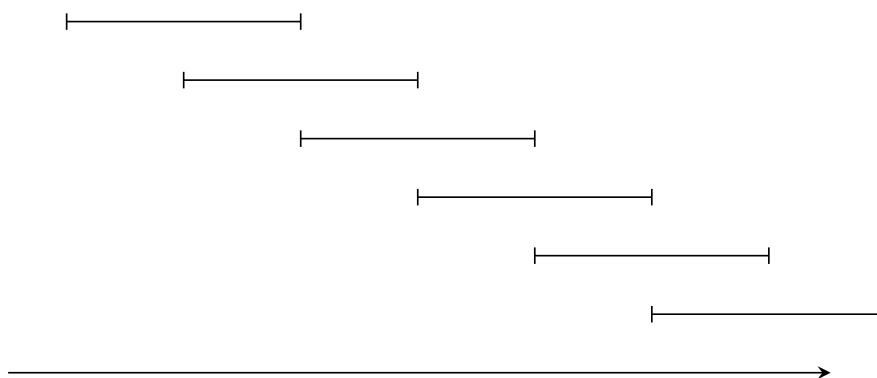


Figure: Bad instance for recursive algorithm

Running time on this instance is

$$T(n) = T(n - 1) + T(n - 2) + O(1) = \Theta(\phi^n)$$

where $\phi \approx 1.618$ is the golden ratio.

Analysis of the Problem

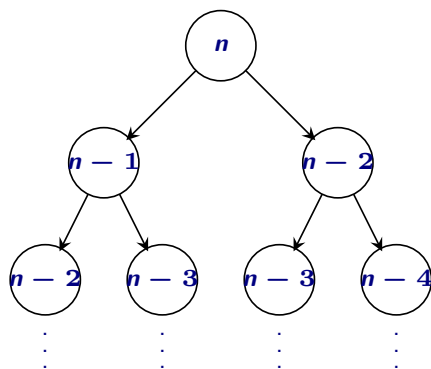


Figure: Label of node indicates size of sub-problem. Tree of sub-problems grows very quickly

Memo(r)ization

Observation

- Number of different sub-problems in recursive algorithm is $O(n)$; they are O_1, O_2, \dots, O_{n-1}
- Exponential time is due to recomputation of solutions to sub-problems

Solution

Store optimal solution to different sub-problems, and perform recursive call **only** if not already computed.

Recursive Solution with Memoization

```
schdMem(j)
```

```
  if j = 0 then return 0
```

```
  if M[j] is defined then (* sub-problem already solved *)  
    return M[j]
```

```
  if M[j] is not defined then
```

```
    M[j] = max(w(vj) + schdMem(p(j)), schdMem(j - 1))
```

```
    return M[j]
```

Time Analysis

- Each invocation, $O(1)$ time plus: either return a computed value, or generate 2 recursive calls and fill one $M[\cdot]$
- Initially no entry of $M[]$ is filled; at the end all entries of $M[]$ are filled
- So total time is $O(n)$ (Assuming input is presorted...)

Automatic Memoization

Fact

Many functional languages (like LISP) automatically do memoization for recursive function calls!

Back to Weighted Interval Scheduling

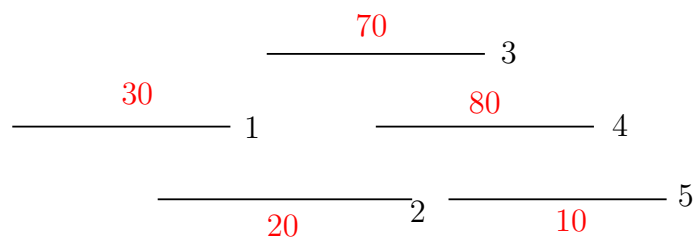
Iterative Solution

```
 $M[0] = 0$   
for  $i = 1$  to  $n$  do  
     $M[i] = \max(w(v_i) + M[p(i)], M[i - 1])$ 
```

M : table of subproblems

- Implicitly dynamic programming fills the values of **M** .
- Recursion determines order in which table is filled up.
- Think of decomposing problem first (recursion) and then worry about setting up table — this comes naturally from recursion.

Example



$$p(5) = 2, p(4) = 1, p(3) = 1, p(2) = 0, p(1) = 0$$

Computing Solutions + First Attempt

- Memoization + Recursion/Iteration allows one to compute the optimal value. What about the actual schedule?

```
 $M[0] = 0$   
 $S[0]$  is empty schedule  
for  $i = 1$  to  $n$  do  
     $M[i] = \max(w(v_i) + M[p(i)], M[i - 1])$   
    if  $w(v_i) + M[p(i)] < M[i - 1]$  then  
         $S[i] = S[i - 1]$   
    else  
         $S[i] = S[p(i)] \cup \{i\}$ 
```

- Naïvely updating $S[]$ takes $O(n)$ time
- Total running time is $O(n^2)$
- Using pointers and linked lists running time can be improved to $O(n)$.

Computing Implicit Solutions

Observation

Solution can be obtained from $M[]$ in $O(n)$ time, without any additional information

```
findSolution(  $j$  )  
    if ( $j = 0$ ) then return empty schedule  
    if ( $w_j + M[p(j)] > M[j - 1]$ ) then  
        return findSolution( $p(j)$ )  $\cup$   $\{j\}$   
    else  
        return findSolution( $j - 1$ )
```

Makes $O(n)$ recursive calls, so findSolution runs in $O(n)$ time.

Computing Implicit Solutions

A generic strategy for computing solutions in dynamic programming:

- Keep track of the *decision* in computing the optimum value of a sub-problem. decision space depends on recursion
- Once the optimum values are computed, go back and use the decision values to compute an optimum solution.

Question: What is the decision in computing $M[i]$?

A: Whether to include i or not.

Computing Implicit Solutions

```
 $M[0] = 0$ 
for  $i = 1$  to  $n$  do
   $M[i] = \max(v_i + M[p(i)], M[i - 1])$ 
  if  $(v_i + M[p(i)] > M[i - 1])$  then
     $Decision[i] = 1$  (* 1:  $i$  included in solution  $M[i]$  *)
  else
     $Decision[i] = 0$  (* 0:  $i$  not included in solution  $M[i]$  *)

 $S = \emptyset$ ,  $i = n$ 
while  $(i > 0)$  do
  if  $(Decision[i] = 1)$  then
     $S = S \cup \{i\}$ 
     $i = p(i)$ 
  else
     $i = i - 1$ 
return  $S$ 
```