CS 473: Algorithms

Chandra Chekuri
chekuri@cs.illinois.edu
3228 Siebel Center

University of Illinois, Urbana-Champaign

Fall 2010
Part I

Longest Increasing Subsequence
Sequences

**Definition**

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

**Definition**

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

**Definition**

A sequence is *increasing* if \( a_1 < a_2 < \ldots < a_n \). It is *non-decreasing* if \( a_1 \leq a_2 \leq \ldots \leq a_n \). Similarly *decreasing* and *non-increasing*.

**Example**

- Sequence: 6, 3, 5, 2, 7, 8, 1
- Subsequence: 5, 2, 1
Longest Increasing Subsequence Problem

**Input**  A sequence of numbers $a_1, a_2, \ldots, a_n$

**Goal**  Find an *increasing subsequence* $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ of maximum length
Input  A sequence of numbers $a_1, a_2, \ldots, a_n$

Goal  Find an increasing subsequence $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ 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
Näive Enumeration

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

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

Running time: $O(n^2)$. There are $2^n$ subsequences of a sequence of length $n$ and $O(n)$ time to check if a given sequence is increasing.
Naive Enumeration

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

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

1. $max = 0$
2. for each subsequence $B$ of $A$ do
   1. if $B$ is increasing and $|B| > max$ then
      1. $max = |B|$
   end for
3. Output $max$

Running time:

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

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

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

max = 0

for each subsequence $B$ of $A$ do

if $B$ is increasing and $|B| >$ max then

max = $|B|$

end for

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

Can we find a recursive algorithm for LIS?

\[ \text{LIS}(A[1..n]) : \]
Can we find a recursive algorithm for LIS?

LIS(A[1..n]):

Case 1: does not contain $a_n$ in which case
LIS(A[1..n]) = LIS(A[1..(n − 1)])

Case 2: contains $a_n$ in which case LIS(A[1..n]) is
Can we find a recursive algorithm for LIS?

LIS(A[1..n]):

Case 1: does not contain $a_n$ in which case
LIS(A[1..n]) = LIS(A[1..(n − 1)])

Case 2: contains $a_n$ in which case LIS(A[1..n]) is not so clear.
Recursive Approach: Take 1

Can we find a recursive algorithm for LIS?

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

**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**: if \( a_n \) is in the longest increasing subsequence then all the elements before it must be smaller.
Recursive Approach: Take 1

Can we find a recursive algorithm for LIS?

LIS(A[1..n]):
Can we find a recursive algorithm for LIS?

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

**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
Recursive Approach: Take 1

Can we find a recursive algorithm for LIS?

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.
Longest Increasing Subsequence

Recursive Approach: Take 1

Can we find a recursive algorithm for LIS?

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

**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**: If \( a_n \) is in the longest increasing subsequence then all the elements before it must be smaller.
Recursive Approach: Take 1

LIS(A[1..n]):
    if (n = 0) return 0
    m = LIS(A[1..(n-1))
    B is subsequence of 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+LIS(B[1..h]))
    Output m
Recursive Approach: Take 1

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

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

\[\text{LIS}(A[1..n]):\]
\[
\begin{align*}
\text{if } (n = 0) & \text{ return 0} \\
\text{m} &= \text{LIS}(A[1..(n-1)]) \\
B & \text{ is subsequence of } A[1..(n-1)] \text{ with only elements less than } a_n \\
(* & \text{ let } h \text{ be size of } B, \ h \leq n-1 *) \\
\text{m} &= \max(\text{m}, 1+\text{LIS}(B[1..h])) \\
\text{Output } m
\end{align*}
\]

Recursion for running time: \(T(n) \leq 2T(n-1) + O(n).\)
\(T(n) = O(2^n).\) Easy to see that it 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 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 \text{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) return 0
    m = LIS-smaller($A[1..(n-1)], x$)
    if (A[n] < x)
        m = max(m, 1+LIS-smaller($A[1..(n-1)], A[n]$))
    Output m

LIS($A[1..n]$):
    return LIS-smaller($A[1..n], \infty$)
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) return 0
    m = LIS-smaller(A[1..(n-1)], x)
    if (A[n] < x)
        m = max(m, 1+LIS-smaller(A[1..(n-1)], A[n]))
    Output m

LIS(A[1..n]):
    return LIS-smaller(A[1..n], ∞)

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

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 \(x\)

\[
\text{LIS-smaller}(A[1..n], x):
\]
\[
\begin{align*}
&\text{if (n = 0) return 0} \\
&m = \text{LIS-smaller}(A[1..(n-1)], x) \\
&\text{if (A[n] < x)} \\
&m = \max(m, 1 + \text{LIS-smaller}(A[1..(n-1)], A[n])) \\
&\text{Output m}
\end{align*}
\]

LIS\((A[1..n])\):
return LIS-smaller\((A[1..n], \infty)\)

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

**Question:** Is there any advantage?
Observation: The number of different subproblems generated by 
LIS-smaller(A[1..n], x) is $O(n^2)$. 
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!
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($A[1..n], x$)?
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 $A[i + 1], \ldots, A[n]$. 
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 $A[i+1], \ldots, A[n]$.

**Observation:** previous recursion also generates only $O(n^2)$ subproblems. Slightly harder to see.
Recursive Algorithm: Take 3


**Question:** can we obtain a recursive expression?
Recursive Algorithm: Take 3


**Question:** can we obtain a recursive expression?

\[
\text{LIS-Ending}(A[1..n]) = \max_{i: A[i] < A[n]} (1 + \text{LIS-Ending}(A[1..i]))
\]
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]) do
            m = max(m, 1+LIS-Ending(A[1..i]))
    Output m

LIS(A[1..n]):
    return $\max_{i=1}^{n} LIS$-Ending(A[1..i])
Longest Increasing Subsequence

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]) do
         m = max(m, 1+LIS-Ending(A[1..i]))
   
   Output m

LIS(A[1..n]):
   return \( \max_{i=1}^{n} \) LIS-Ending(A[1..n], \( \infty \))

Question: How many distinct subproblems generated by
LIS-Ending(A[1..n])?
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]) do
      m = max(m, 1+LIS-Ending(A[1..i]))
  
  Output m

LIS(A[1..n]):
  return \max_{i=1}^{n} LIS-Ending(A[1..n], \infty)

**Question:** How many distinct subproblems generated by LIS-Ending(A[1..n])? n.
Iterative Algorithm via Memoization

Compute the values LIS-Ending(A[1..i]) iteratively in a bottom up fashion.

LIS-Ending(A[1..n]):
   Array L[1..n] (* L[i] stores the value 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])
Output m
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])
Output m

Correctness: Via induction following the recursion
Iterative Algorithm via Memoization

Simplifying:

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

Correctness: Via induction following the recursion

Running time:
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])

Output m

Correctness: Via induction following the recursion

Running time: $O(n^2)$,
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])
Output m

Correctness: Via induction following the recursion
Running time: \(O(n^2)\), Space:
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])
    Output m

Correctness: Via induction following the recursion

Running time: $O(n^2)$, Space: $\Theta(n)$
Longest Increasing Subsequence

Example

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


$L[1] = 1$
$L[2] = 1$
$L[3] = 2$
$L[4] = 1$
$L[5] = 3$
$L[6] = 4$

$Q[7] = 1$
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 the Second Recursive Algorithm

LIS(A[1..n]):
A[n+1] = ∞ (* 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])
      L[i,j] = max(L[i,j], 1+L[i-1,i])

Output L[n,(n+1)]
Longest Increasing Subsequence

Memoizing the Second Recursive Algorithm

LIS(A[1..n]):
    A[n+1] = ∞ (* add a sentinel at the end *)
    Array L[(n+1),(n+1)] (* two-dimensional array*)
(* L[i,j] for j ≥ 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])
                L[i,j] = max(L[i,j], 1+L[i-1,i])
    Output L[n,(n+1)]

Correctness: Via induction following the recursion (take 2)
Memoizing the Second Recursive Algorithm

LIS(A[1..n]):

A[n+1] = ∞ (* add a sentinel at the end *)
Array L[(n+1),(n+1)] (* two-dimensional array*)
(* L[i,j] for j ≥ 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])
            L[i,j] = max(L[i,j], 1+L[i-1,i])

Output L[n,(n+1)]

Correctness: Via induction following the recursion (take 2)

Running time:
LIS(A[1..n]):
A[n+1] = ∞ (* add a sentinel at the end *)
Array L[(n+1),(n+1)] (* two-dimensional array*)
(* L[i,j] for j ≥ 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])
      L[i,j] = max(L[i,j], 1+L[i-1,i])

Output L[n,(n+1)]

**Correctness:** Via induction following the recursion (take 2)
**Running time:** $O(n^2)$,
Memoizing the Second Recursive Algorithm

\[
\text{LIS}(A[1..n]):
\]
\[
A[n+1] = \infty \quad (* \text{add a sentinel at the end} *)
\]

* Array \( L[(n+1),(n+1)] \) (* two-dimensional array*)

(* \( L[i,j] \) for \( j \geq i \) stores the value \( \text{LIS-Smaller}(A[1..i],A[j]) \) *)

\[
\text{for } j = 1 \text{ to } n+1 \text{ do}
\]
\[
L[0,j] = 0
\]

\[
\text{for } i = 1 \text{ to } n+1 \text{ do}
\]
\[
\text{for } j = i \text{ to } n+1 \text{ do}
\]
\[
L[i,j] = L[i-1,j]
\]
\[
\text{if } (A[i] < A[j])
\]
\[
L[i,j] = \max(L[i,j], 1+L[i-1,i])
\]

Output \( L[n,(n+1)] \)

**Correctness:** Via induction following the recursion (take 2)

**Running time:** \( O(n^2) \), **Space:**
Memoizing the Second Recursive Algorithm

LIS(A[1..n]):
A[n+1] = \infty (\text{* add a sentinel at the end *})
Array L[(n+1),(n+1)] (\text{* 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])
            L[i,j] = \max(L[i,j], 1+L[i-1,i])

Output L[n,(n+1)]

**Correctness:** Via induction following the recursion (take 2)
**Running time:** \(O(n^2)\), **Space:** \(\Theta(n^2)\)
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!
**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

**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
Interval Scheduling

**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)
Interval Scheduling

**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)
Interval Scheduling

**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)
Interval Scheduling

**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)
Interval Scheduling

**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)
Interval Scheduling

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.
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 vi with weight wi. Add an edge between vi and vj 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.

We do not know an efficient (polynomial time) algorithm for independent set! Can we take advantage of the interval structure to find an efficient algorithm?
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.
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.

We do not know an efficient (polynomial time) algorithm for independent set! Can we take advantage of the interval structure to find an 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**

<table>
<thead>
<tr>
<th>Task</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1</td>
<td>2</td>
</tr>
<tr>
<td>Task 2</td>
<td>4</td>
</tr>
<tr>
<td>Task 3</td>
<td>4</td>
</tr>
<tr>
<td>Task 4</td>
<td>7</td>
</tr>
<tr>
<td>Task 5</td>
<td>2</td>
</tr>
<tr>
<td>Task 6</td>
<td>1</td>
</tr>
</tbody>
</table>

- $p(1) = 0$
- $p(2) = 0$
- $p(3) = 1$
- $p(4) = 0$
- $p(5) = 3$
- $p(6) = 3$
Conventions

**Definition**

- Let the requests be sorted according to finish time, i.e., \( i < j \) implies \( f_i \leq f_j \)
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
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

\begin{align*}
1 & \quad v_1 = 2 \\
2 & \quad v_2 = 4 \\
3 & \quad v_3 = 4 \\
4 & \quad v_4 = 7 \\
5 & \quad v_5 = 2 \\
6 & \quad v_6 = 1
\end{align*}

\begin{align*}
p(1) &= 0 \\
p(2) &= 0 \\
p(3) &= 1 \\
p(4) &= 0 \\
p(5) &= 3 \\
p(6) &= 3
\end{align*}
Towards a Recursive Solution

**Observation**

Consider an optimal schedule $O$

**Case** $n \in O$

None of the jobs between $n$ and $p(n)$ can be scheduled. Moreover $O$ must contain an optimal schedule for the first $p(n)$ jobs.
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 $n \not\in \mathcal{O}$ $\mathcal{O}$ 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.

Recursively compute $O_{p(n)}$
Recursively compute $O_{n-1}$
If $(O_{p(n)} + v_n < O_{n-1})$ then
\[ O_n = O_{n-1} \]
else
\[ O_n = O_{p(n)} + v_n \]
Output $O_n$
A Recursive Algorithm

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

Recursively compute $O_{p(n)}$
Recursively compute $O_{n-1}$
If $(O_{p(n)} + \nu_n < O_{n-1})$ then
   $O_n = O_{n-1}$
else
   $O_n = O_{p(n)} + \nu_n$
Output $O_n$

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

\[ T(n) = T(n - 1) + T(n - 2) + O(1) \]
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 $O_1, O_2, ..., O_{n−1}$. 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.
Observation

- *Number of different sub-problems in recursive algorithm is*
Memo(r)ization

Observation

- Number of different sub-problems in recursive algorithm is \( O(n) \); they are \( O_1, O_2, \ldots, O_{n-1} \)
Memo(r)ization

Observation

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

Observation

- Number of different sub-problems in recursive algorithm is $O(n)$; they are $O_1, O_2, \ldots, O_{n-1}$
- 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

```plaintext
computeOpt(int 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(v_j + computeOpt(p(j)), computeOpt(j-1))
    return M[j]
```
Recursive Solution with Memoization

```java
computeOpt(int 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(v_j + computeOpt(p(j)), computeOpt(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 $M[\cdot]$. 

Chekuri
CS473
Recursive Solution with Memoization

computeOpt(int 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(v_j + computeOpt(p(j)), computeOpt(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 $M[\cdot]$ is filled.
Recursive Solution with Memoization

\[
\text{computeOpt}(\text{int } j) \\
\text{if } j = 0 \text{ then return } 0 \\
\text{if } M[j] \text{ is defined then } (* \text{ sub-problem already solved } *) \\
\quad \text{return } M[j] \\
\text{if } M[j] \text{ is not defined then} \\
\quad M[j] = \max(v_j + \text{computeOpt}(p(j)), \text{computeOpt}(j-1)) \\
\quad \text{return } M[j]
\]

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 \(M[\cdot]\) is filled; at the end all entries of \(M[\cdot]\) are filled.
Recursive Solution with Memoization

```java
computeOpt(int 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(v_j + computeOpt(p(j)), computeOpt(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 $M[\cdot]$ is filled; at the end all entries of $M[\cdot]$ are filled
- So total time is $O(n)$
Automatic Memoization

Fact

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

\[ M[0] = 0 \]
\[
\text{for } i = 1 \text{ to } n \\
\quad \quad M[i] = \max(v_i + M[p(i)], M[i-1])
\]
Iterative Solution

\[ M[0] = 0 \]
\[
\text{for } i = 1 \text{ to } n \\
M[i] = \max(v_i + M[p(i)], M[i-1])
\]

\( M \): 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
Weighted Interval Scheduling

The Problem
Greedy Solution
Recursive Solution
Dynamic Programming
Computing Solutions

Example

\[ p(5) = 2, \ p(4) = 1, \ p(3) = 1, \ p(2) = 0, \ p(1) = 0 \]
Memoization + Recursion/Iteration allows one to compute the optimal value. What about the actual schedule?
Memoization + Recursion/Iteration allows one to compute the optimal value. What about the actual schedule?

\[
\begin{align*}
M[0] &= 0 \\
S[0] &= \text{empty schedule} \\
\text{for } i &= 1 \text{ to } n \\
M[i] &= \max(v_i + M[p(i)], M[i-1]) \\
S[i] &= v_i + M[p(i)] < M[i-1] ? S[i-1] : S[p(i)] \cup \{i\}
\end{align*}
\]
Memoization + Recursion/Iteration allows one to compute the optimal value. What about the actual schedule?

\[ M[0] = 0 \]
\[ S[0] \text{ is empty schedule} \]

\[ \text{for } i = 1 \text{ to } n \]
\[ M[i] = \max(v_i + M[p(i)], M[i-1]) \]
\[ S[i] = v_i + M[p(i)] < M[i-1] ? S[i-1] : S[p(i)] \cup \{i\} \]
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 \( n \)
  
  \[
  M[i] = \max(v_i + M[p(i)], M[i-1])
  \]
  
  \[
  S[i] = v_i + M[p(i)] < M[i-1] ? S[i-1] : S[p(i)] \cup \{i\}
  \]

- Naïvely updating \( S[] \) takes \( O(n) \) time
Memoization + Recursion/Iteration allows one to compute the optimal value. What about the actual schedule?

\[
\begin{align*}
M[0] & = 0 \\
S[0] & \text{ is empty schedule} \\
\text{for } i & = 1 \text{ to } n \\
M[i] & = \max(v_i + M[p(i)], M[i-1]) \\
S[i] & = v_i + M[p(i)] < M[i-1] \ ? \ S[i-1] : \ S[p(i)] \cup \{i\}
\end{align*}
\]

- Naïvely updating \( S[] \) takes \( O(n) \) time
- Total running time is \( O(n^2) \)
Observation

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

```java
findSolution(int j)
    if (j=0) then return empty schedule
    if ($v_j + M[p(j)] > M[j-1]$) 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 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 $M[i]$? Whether to include $i$ or not.
Computing Implicit Solutions

\[
\begin{align*}
M[0] &= 0 \\
\text{for } i &= 1 \text{ to } n \\
M[i] &= \max(v_i + M[p(i)], M[i-1]) \\
\text{if } (v_i + M[p(i)] > M[i-1]) \\
\quad \text{Decision}[i] &= 1 \quad (* \ 1 \text{ means } i \text{ included in solution } M[i] \ *) \\
\text{else} \\
\quad \text{Decision}[i] &= 0 \quad (* \ 0 \text{ means } i \text{ not included in solution } M[i] \ *) \\
\end{align*}
\]

\[
S = \emptyset, \ i = n \\
\text{While } (i > 0) \text{ do} \\
\quad \text{if } (\text{Decision}[i] == 1) \\
\quad \quad S = S \cup i \\
\quad \quad i = p(i) \\
\quad \text{else} \\
\quad \quad i = i-1 \\
\]

Output $S$