CS 473: Algorithms

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

University of Illinois, Urbana-Champaign

Fall 2009
Part I

Introduction to Dynamic Programming
Recursion

Reduction: reduce one problem to another

Recursion: a special case of reduction
- reduce problem to a smaller instance of itself
- self-reduction
Recursion

Reduction: reduce one problem to another

Recursion: a special case of reduction

- reduce problem to a *smaller* instance of *itself*
- self-reduction

- Problem instance of size $n$ is reduced to one or more instances of size $n - 1$ or less.
- For termination, problem instances of small size are solved by some other method as *base cases*
Recurrention in Algorithm Design

- **Tail Recursion**: problem reduced to a *single* recursive call after some work. Easy to convert algorithm into iterative or greedy algorithms. Examples: Interval scheduling, MST algorithms, etc.

- **Divide and Conquer**: problem reduced to multiple *independent* sub-problems that are solved separately. Conquer step puts together solution for bigger problem.

- **Dynamic Programming**: problem reduced to multiple *(typically) dependent or overlapping* sub-problems. Use *memoization* to avoid recomputation of common solutions leading to *iterative bottom-up* algorithm.
Fibonacci Numbers

Fibonacci numbers defined by recurrence:

\[ F(n) = F(n - 1) + F(n - 2) \] and \[ F(0) = 0, F(1) = 1. \]

These numbers have many interesting and amazing properties. A journal *The Fibonacci Quarterly*!

- \[ F(n) = (\phi^n - (1 - \phi)^n)/\sqrt{5} \] where \( \phi \) is the golden ratio \( (1 + \sqrt{5})/2 \approx 1.618. \)
- \( \lim_{n \to \infty} F(n + 1)/F(n) = \phi \)
Fibonacci numbers defined by recurrence:

\[ F(n) = F(n - 1) + F(n - 2) \text{ and } F(0) = 0, F(1) = 1. \]

These numbers have many interesting and amazing properties. A journal *The Fibonacci Quarterly!*

- \[ F(n) = \left( \phi^n - (1 - \phi)^n \right) / \sqrt{5} \text{ where } \phi \text{ is the golden ratio } \frac{1 + \sqrt{5}}{2} \approx 1.618. \]
- \[ \lim_{n \to \infty} F(n + 1)/F(n) = \phi \]

**Question:** Given \( n \), compute \( F(n) \).
Fibonacci Numbers

Recursive Algorithm for Fibonacci Numbers

Fib(n):
    if (n = 0)
        return 0
    else if (n = 1)
        return 1
    else
        return Fib(n-1) + Fib(n-2)

Running time? Let $T(n)$ be the number of additions in Fib(n).

$T(n) = T(n-1) + T(n-2) + 1$ and $T(0) = T(1) = 0$

Roughly same as $F(n)$

$T(n) = \Theta(\phi^n)$

Thus algorithm does exponential in $n$ additions.

Can we do better?

Chekuri
Recursive Algorithm for Fibonacci Numbers

Fib(n):
  if (n = 0)
    return 0
  else if (n = 1)
    return 1
  else
    return Fib(n-1) + Fib(n-2)

Running time? Let $T(n)$ be the number of additions in Fib(n).
Recursive Algorithm for Fibonacci Numbers

Fib(n):
    if (n = 0)
        return 0
    else if (n = 1)
        return 1
    else
        return Fib(n-1) + Fib(n-2)

Running time? Let $T(n)$ be the number of additions in Fib(n).

$T(n) = T(n - 1) + T(n - 2) + 1$ and $T(0) = T(1) = 0$

Roughly same as $F(n)$

$T(n) = \Theta(\phi^n)$

Thus algorithm does exponential in $n$ additions.
Fibonacci Numbers

Recursive Algorithm for Fibonacci Numbers

Fib(n):
    if (n = 0)
        return 0
    else if (n = 1)
        return 1
    else
        return Fib(n-1) + Fib(n-2)

Running time? Let $T(n)$ be the number of additions in Fib(n).

$$T(n) = T(n-1) + T(n-2) + 1 \text{ and } T(0) = T(1) = 0$$

Roughly same as $F(n)$

$$T(n) = \Theta(\phi^n)$$

Thus algorithm does exponential in $n$ additions. Can we do better?
Fibonacci Numbers

An iterative algorithm for Fibonacci numbers

Fib(n):
    if (n = 0)
        return 0
    else if (n = 1)
        return 1
    else
        F[0] = 0
        F[1] = 1
        for i = 2 to n do
            F[i] = F[i-1] + F[i-2]
        return F[n]
An iterative algorithm for Fibonacci numbers

Fib(n):
    if (n = 0)
        return 0
    else if (n = 1)
        return 1
    else
        F[0] = 0
        F[1] = 1
        for i = 2 to n do
            F[i] = F[i-1] + F[i-2]
        return F[n]

What is the running time of the algorithm?

$O(n)$ additions.
An iterative algorithm for Fibonacci numbers

Fib(n):
    if (n = 0)
        return 0
    else if (n = 1)
        return 1
    else
        F[0] = 0
        F[1] = 1
        for i = 2 to n do
            F[i] = F[i-1] + F[i-2]
        return F[n]

What is the running time of the algorithm? $O(n)$ additions.
What is the difference?

- Recursive algorithm is computing the same numbers again and again.
- Iterative algorithm is storing computed values and building bottom up the final value.
What is the difference?

- Recursive algorithm is computing the same numbers again and again.
- Iterative algorithm is storing computed values and building bottom up the final value. **Memoization.**
Fibonacci Numbers

What is the difference?

- Recursive algorithm is computing the same numbers again and again.
- Iterative algorithm is storing computed values and building bottom up the final value. Memoization.

Dynamic Programming: finding a recursion that can be effectively/efficiently memoized

Leads to polynomial time algorithm if number of sub-problems is polynomial in input size.
Fibonacci Numbers

Automatic Memoization

Can we convert recursive algorithm into an efficient algorithm without explicitly doing an iterative algorithm?

Fib(n):
if (n = 0)
    return 0
else if (n = 1)
    return 1
else if (Fib(n) was previously computed)
    return stored value of Fib(n)
else
    return Fib(n-1) + Fib(n-2)

How do we keep track of previously computed values?

Two methods: explicitly and implicitly (via data structure)
Can we convert recursive algorithm into an efficient algorithm without explicitly doing an iterative algorithm?

Fib(n):
    if (n = 0)
        return 0
    else if (n = 1)
        return 1
    else if (Fib(n) was previously computed)
        return stored value of Fib(n)
    else
        return Fib(n-1) + Fib(n-2)
Can we convert recursive algorithm into an efficient algorithm without explicitly doing an iterative algorithm?

Fib(n):
    if (n = 0)
        return 0
    else if (n = 1)
        return 1
    else if (Fib(n) was previously computed)
        return stored value of Fib(n)
    else
        return Fib(n-1) + Fib(n-2)

How do we keep track of previously computed values?
Can we convert recursive algorithm into an efficient algorithm without explicitly doing an iterative algorithm?

Fib(n):
  if (n = 0)
    return 0
  else if (n = 1)
    return 1
  else if (Fib(n) was previously computed)
    return stored value of Fib(n)
  else
    return Fib(n-1) + Fib(n-2)

How do we keep track of previously computed values?
Two methods: explicitly and implicitly (via data structure)
Automatic explicit memoization

Initialize table/array $M$ of size $n$ such that $M[i] = -1$ for $0 \leq i < n$
Automatic explicit memoization

Initialize table/array $M$ of size $n$ such that $M[i] = -1$ for $0 \leq i < n$

Fib(n):
  if (n = 0)
    return 0
  else if (n = 1)
    return 1
  else if (M[n] $\neq$ -1) (* $M[n]$ has stored value of Fib(n) *)
    return M[n]
  else
    M[n] = Fib(n-1) + Fib(n-2)
    return M[n]

Need to know upfront the number of subproblems to allocate memory
Initialize a (dynamic) dictionary data structure $D$ to empty

Fib(n):
    if (n = 0)
        return 0
    else if (n = 1)
        return 1
    else if (n is already in D)
        return value stored with n in D
    else
        val = Fib(n-1) + Fib(n-2)
        Store (n, val) in D
    return val
Explicit vs Implicit Memoization

- Explicit memoization or iterative algorithm preferred if one can analyze problem ahead of time. Allows for efficient memory allocation and access.
- Implicit and automatic memoization used when problem structure or algorithm is either not well understood or in fact unknown to the underlying system
  - need to pay overhead of datastructure
  - Functional languages such as LISP automatically do memoization, usually via hashing based dictionaries.
Is the iterative algorithm a *polynomial* time algorithm? Does it take $O(n)$ time?
Is the iterative algorithm a *polynomial* time algorithm? Does it take $O(n)$ time?

- input is $n$ and hence input size is $\Theta(\log n)$
Is the iterative algorithm a *polynomial* time algorithm? Does it take $O(n)$ time?

- input is $n$ and hence input size is $\Theta(\log n)$
- output is $F(n)$ and output size is $\Theta(n)$. Why?
Is the iterative algorithm a *polynomial* time algorithm? Does it take $O(n)$ time?

- input is $n$ and hence input size is $\Theta(\log n)$
- output is $F(n)$ and output size is $\Theta(n)$. Why?
- Hence output size is exponential in input size so no polynomial time algorithm possible!
Is the iterative algorithm a *polynomial* time algorithm? Does it take $O(n)$ time?

- input is $n$ and hence input size is $\Theta(\log n)$
- output is $F(n)$ and output size is $\Theta(n)$. Why?
- Hence output size is exponential in input size so no polynomial time algorithm possible!

Running time of iterative algorithm: $\Theta(n)$ additions but number sizes are $O(n)$ bits long! Hence total time is $O(n^2)$, in fact $\Theta(n^2)$. Why?
Is the iterative algorithm a *polynomial* time algorithm? Does it take $O(n)$ time?

- input is $n$ and hence input size is $\Theta(\log n)$
- output is $F(n)$ and output size is $\Theta(n)$. Why?
- Hence output size is exponential in input size so no polynomial time algorithm possible!

Running time of iterative algorithm: $\Theta(n)$ additions but number sizes are $O(n)$ bits long! Hence total time is $O(n^2)$, in fact $\Theta(n^2)$. Why?

Running time of recursive algorithm is $O(n\phi^n)$ but can in fact shown to be $O(\phi^n)$ by being careful. Doubly exponential in input size and exponential even in output size.
Part II

Recursion and Brute Force Search
Definition

Given undirected graph $G = (V, E)$ a subset of nodes $S \subseteq V$ is an independent set (also called a stable set) if for there are no edges between nodes in $S$. That is, if $u, v \in S$ then $(u, v) \not\in E$.

Some independent sets in graph above:
Input  Graph  $G = (V, E)$

Goal  Find maximum sized independent set in $G$
Maximum Weight Independent Set Problem

Input: Graph $G = (V, E)$, weights $w(v) \geq 0$ for $v \in V$

Goal: Find maximum weight independent set in $G$
Maximum Weight Independent Set Problem

- No one knows an *efficient* (polynomial time) algorithm for this problem.
- Problem is NP-Complete and it is *believed* that there is no polynomial time algorithm.

A *brute-force* algorithm: try all subsets of vertices.
Algorithm to find the size of the maximum weight independent set.

MaxIndSet($G = (V, E)$):

1. $max = 0$
2. for each subset $S \subseteq V$
   1. check if $S$ is an independent set
   2. if $S$ is an independent set and $w(S) > max$
      1. $max = w(S)$
   endfor
3. Output $max$

Running time: Suppose $G$ has $n$ vertices and $m$ edges. Checking each subset $S$ takes $O(m)$ time.
Total time is $O(m^2 n)$. 

Chekuri CS473ug
Brute-force enumeration

Algorithm to find the size of the maximum weight independent set.

MaxIndSet\( (G = (V, E)) \):
\[
\begin{align*}
  &max = 0 \\
  &\text{for each subset } S \subseteq V \\
  &\quad \text{check if } S \text{ is an independent set} \\
  &\quad \text{if } S \text{ is an independent set and } w(S) > max \\
  &\quad \quad max = w(S) \\
  &\text{endfor} \\
  &\text{Output } max
\end{align*}
\]

Running time: suppose \( G \) has \( n \) vertices and \( m \) edges

- \( 2^n \) subsets of \( V \)
- checking each subset \( S \) takes \( O(m) \) time
- total time is \( O(m2^n) \)
A Recursive Algorithm

Let $V = \{v_1, v_2, \ldots, v_n\}$.
For a vertex $u$ let $N(u)$ be its neighbours.
A Recursive Algorithm

Let $V = \{v_1, v_2, \ldots, v_n\}$.
For a vertex $u$ let $N(u)$ be its neighbours.

Observation

One of the following two cases is true

Case 1  $v_n$ is in some maximum independent set.
Case 2  $v_n$ is in no maximum independent set.
A Recursive Algorithm

Let \( V = \{v_1, v_2, \ldots, v_n\} \).
For a vertex \( u \) let \( N(u) \) be its neighbours.

Observation

One of the following two cases is true

Case 1 \( v_n \) is in some maximum independent set.
Case 2 \( v_n \) is in no maximum independent set.

Recursive-MIS(\( G \)):
If \( G \) is empty, Output 0
\( a = \) Recursive-MIS(\( G - v_n \))
\( b = w(v_n) + \) Recursive-MIS(\( G - v_n - N(v_n) \))
Output \( \max(a, b) \)
Recursive Algorithms of MIS

Running time:

\[ T(n) = \]

where \( \deg(v_1) \) is the degree of \( v_1 \).

\( T(0) = T(1) = 1 \) is base case.

Worst case is when \( \deg(v_n) = 0 \) when the recurrence becomes

\[ T(n) = 2T(n-1) + O(1) \]

Solution to this is \( T(n) = O(2^n) \).

Can we improve this?

Worst case is when \( \deg(v_n) = 0 \). In this case \( v_n \) is in every maximum weight independent set! No need to check!
Recursive Algorithms of MIS

Running time:

\[ T(n) = T(n - 1) + T(n - 1 - \text{deg}(v_1)) + O(1) \]

where \( \text{deg}(v_1) \) is the degree of \( v_1 \). \( T(0) = T(1) = 1 \) is base case.

Worst case is when \( \text{deg}(v_1) = 0 \) when the recurrence becomes

\[ T(n) = 2T(n - 1) + O(1) \]

Solution to this is \( T(n) = O(2^n) \).
Recursive Algorithms of MIS

Running time:

\[ T(n) = T(n-1) + T(n-1 - \text{deg}(v_1)) + O(1) \]

where \( \text{deg}(v_1) \) is the degree of \( v_1 \). \( T(0) = T(1) = 1 \) is base case.

Worst case is when \( \text{dev}(v_1) = 0 \) when the recurrence becomes

\[ T(n) = 2T(n-1) + O(1) \]

Solution to this is \( T(n) = O(2^n) \).
Can we improve this?
Recursive Algorithms of MIS

Running time:

\[ T(n) = T(n-1) + T(n-1 - \text{deg}(v_1)) + O(1) \]

where \( \text{deg}(v_1) \) is the degree of \( v_1 \). \( T(0) = T(1) = 1 \) is base case.

Worst case is when \( \text{dev}(v_1) = 0 \) when the recurrence becomes

\[ T(n) = 2T(n-1) + O(1) \]

Solution to this is \( T(n) = O(2^n) \).
Can we improve this?
Worst case is when \( \text{deg}(v_n) = 0 \). In this case \( v_n \) is in every maximum weight independent set! No need to check!
An Improved Algorithm

Recursive-MIS($G$):

If $G$ is empty, Output 0
$a = \text{Recursive-MIS}(G - v_n)$
If ($deg(v_n) = 0$)
    Output $w(v_n) + a$
Else
    $b = w(v_n) + \text{Recursive-MIS}(G - v_n - N(v_n))$
    Output $\max(a, b)$

Running time:

$$T(n) = \max\{T(n-1), T(n-1) + T(n-2)\} + O(1)$$

Similar to the Fibonacci recurrence. Can show that $T(n) = O(1.618^n)$. 

Chekuri CS473ug
An Improved Algorithm

Recursive-MIS($G$):
  If $G$ is empty, Output 0
  $a =$ Recursive-MIS($G - v_n$)
  If ($\text{deg}(v_n) = 0$)
    Output $w(v_n) + a$
  Else
    $b = w(v_n) + \text{Recursive-MIS}(G - v_n - N(v_n))$
    Output $\max(a, b)$

Running time:

$$T(n) = \max\{T(n - 1), T(n - 1) + T(n - 2)\} + O(1)$$

Similar to the Fibonacci recurrence. Can show that $T(n) = O(1.618^n)$. 
We expressed the optimum solution value on $G$ recursively as a function of the values on two smaller instances.

$$OPT(G) = \max\{OPT(G - v_n), w(v_n) + OPT(G - v_n - N(v_n))\}$$

- Can we memoize the recursive algorithm(s)? Yes.
- Does memoization improve the running time in the worst case? No. Number of sub-problems can be large (can create explicit graphs).
Part III

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!
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

**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

- Recall, greedy strategy of considering jobs according to finish times produces optimal schedule
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
- Recall, greedy strategy of considering jobs according to finish times produces optimal schedule
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

- Recall, greedy strategy of considering jobs according to finish times produces optimal schedule
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

- Recall, greedy strategy of considering jobs according to finish times produces optimal schedule
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

- Recall, greedy strategy of considering jobs according to finish times produces optimal schedule
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

- Recall, greedy strategy of considering jobs according to finish times produces optimal schedule
Greedy Strategy for Weighted Interval Scheduling

- Pick jobs in order of finishing times
- Add job to schedule if it does not conflict with current schedule
Greedy Strategy for Weighted Interval Scheduling

- Pick jobs in order of finishing times
- Add job to schedule if it does not conflict with current schedule
Other Greedy Strategies

- Largest weight/profit first
- Largest weight to length ratio first
- Shortest length first
- ...

None of the above strategies lead to an optimum solution.
Other Greedy Strategies

- 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 a weighted interval scheduling instance $I$, we create an instance of the max weight independent set problem 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: the max weight independent set in $G(I)$ has weight equal to the 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?
Weighted Interval Scheduling

The Problem
Greedy Solution
Recursive Solution
Dynamic Programming
Computing Solutions

Conventions

Definition

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

Define \( p(j) \) to be the largest \( i \) (less than \( j \)) such that job \( i \) and job \( j \) are not in conflict.

Example:

\[
\begin{align*}
  &v_1 = 2 \\
  &v_2 = 4 \\
  &v_3 = 4 \\
  &v_4 = 7 \\
  &v_5 = 2 \\
  &v_6 = 1 \\
  &p(1) = 0 \\
  &p(2) = 0 \\
  &p(3) = 1 \\
  &p(4) = 0 \\
  &p(5) = 3 \\
  &p(6) = 3
\end{align*}
\]
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

<table>
<thead>
<tr>
<th>Request</th>
<th>Start</th>
<th>Finish</th>
<th>$v_i$</th>
<th>$p(i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
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.
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

Notation: $O_i$ 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

Notation: $O_i$ 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 $O_n$

Time Analysis

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

```python
def computeOpt(j):
    if j == 0:
        return 0
    if M[j] is defined:
        return M[j]
    if M[j] is not defined:
        M[j] = max(v[j] + computeOpt(p[j]), computeOpt(j-1))
    return M[j]
```
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(vj + 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 $M[·]$
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.
Recursive Solution with Memoization

\[ \text{computeOpt}(\text{int } j) \]

- 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
  - \( M[j] = \max(v_j + \text{computeOpt}(p(j)), \text{computeOpt}(j-1)) \)
  - 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
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(vj + 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

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

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

\(M\): table of subproblems

- There is always a table in dynamic programming
- 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

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?

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

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

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

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

- **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
- **Total running time is \(O(n^2)\)**
Observation

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

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

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

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

Output \( S \)
Part IV

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
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

**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
Longest Increasing Subsequence

A First Recursive Approach

$L(i)$: length of longest increasing subsequence in $a_1, a_2, \ldots, a_i$. 

Can we write $L(i)$ in terms of $L(1), L(2), \ldots, L(i-1)$?

Case 1: $L(i)$ does not contain $a_i$, then $L(i) = L(i-1)$

Case 2: $L(i)$ contains $a_i$, then $L(i) = \text{?}$

What is the element in the subsequence before $a_i$? If it is $a_j$ then it better be the case that $a_j < a_i$ since we are looking for an increasing sequence. Do we know which $j$? No!

So we try all possibilities: $L(i) = 1 + \max_{j < i \text{ and } a_j < a_i} L(j)$

Is the above correct? No, because we do not know that $L(j)$ corresponds to a subsequence that actually ends at $a_j$!
A First Recursive Approach

$L(i)$: length of longest increasing subsequence in $a_1, a_2, \ldots, a_i$.

Can we write $L(i)$ in terms of $L(1), L(2), \ldots, L(i - 1)$?
A First Recursive Approach

$L(i)$: length of longest increasing subsequence in $a_1, a_2, \ldots, a_i$.

Can we write $L(i)$ in terms of $L(1), L(2), \ldots, L(i - 1)$?

Case 1: $L(i)$ does not contain $a_i$, then $L(i) = L(i - 1)$
Case 2: $L(i)$ contains $a_i$, then $L(i) =$?
A First Recursive Approach

$L(i)$: length of longest increasing subsequence in $a_1, a_2, \ldots, a_i$.

Can we write $L(i)$ in terms of $L(1), L(2), \ldots, L(i-1)$?

Case 1: $L(i)$ does not contain $a_i$, then $L(i) = L(i-1)$

Case 2: $L(i)$ contains $a_i$, then $L(i) =$? What is the element in the subsequence before $a_i$? If it is $a_j$ then it better be the case that $a_j < a_i$ since we are looking for an increasing sequence.
A First Recursive Approach

$L(i)$: length of longest increasing subsequence in $a_1, a_2, \ldots, a_i$.

Can we write $L(i)$ in terms of $L(1), L(2), \ldots, L(i-1)$?

Case 1: $L(i)$ does not contain $a_i$, then $L(i) = L(i-1)$

Case 2: $L(i)$ contains $a_i$, then $L(i) =$? What is the element in the subsequence before $a_i$? If it is $a_j$ then it better be the case that $a_j < a_i$ since we are looking for an increasing sequence. Do we know which $j$? No!
A First Recursive Approach

$L(i)$: length of longest increasing subsequence in $a_1, a_2, \ldots, a_i$.

Can we write $L(i)$ in terms of $L(1), L(2), \ldots, L(i - 1)$?

**Case 1**: $L(i)$ does not contain $a_i$, then $L(i) = L(i - 1)$

**Case 2**: $L(i)$ contains $a_i$, then $L(i) =$? What is the element in the subsequence before $a_i$? If it is $a_j$ then it better be the case that $a_j < a_i$ since we are looking for an increasing sequence. Do we know which $j$? No! So we try all possibilities

$$L(i) = 1 + \max_{j < i \text{ and } a_j < a_i} L(j)$$
A First Recursive Approach

$L(i)$: length of longest increasing subsequence in $a_1, a_2, \ldots, a_i$.

Can we write $L(i)$ in terms of $L(1), L(2), \ldots, L(i - 1)$?

Case 1 : $L(i)$ does not contain $a_i$, then $L(i) = L(i - 1)$
Case 2 : $L(i)$ contains $a_i$, then $L(i) =$? What is the element in the subsequence before $a_i$? If it is $a_j$ then it better be the case that $a_j < a_i$ since we are looking for an increasing sequence. Do we know which $j$? No! So we try all possibilities

$L(i) = 1 + \max_{j<i \text{ and } a_j < a_i} L(j)$

Is the above correct?
A First Recursive Approach

$L(i)$: length of longest increasing subsequence in $a_1, a_2, \ldots, a_i$.

Can we write $L(i)$ in terms of $L(1), L(2), \ldots, L(i - 1)$?

Case 1: $L(i)$ does not contain $a_i$, then $L(i) = L(i - 1)$

Case 2: $L(i)$ contains $a_i$, then $L(i) =$? What is the element in the subsequence before $a_i$? If it is $a_j$ then it better be the case that $a_j < a_i$ since we are looking for an increasing sequence. Do we know which $j$? No! So we try all possibilities

$$L(i) = 1 + \max_{j < i \text{ and } a_j < a_i} L(j)$$

Is the above correct? No, because we do not know that $L(j)$ corresponds to a subsequence that actually ends at $a_i$!
A Correct Recursion

$L(i)$: longest increasing subsequence in $a_1, a_2, \ldots, a_i$ that ends in $a_i$
A Correct Recursion

\[ L(i) : \text{longest increasing subsequence in } a_1, a_2, \ldots, a_i \text{ that ends in } a_i \]

Recursion for \( L(i) \):

\[ L(i) = 1 + \max_{j < i \text{ and } a_j < a_i} L(j) \]
A Correct Recursion

$L(i)$: longest increasing subsequence in $a_1, a_2, \ldots, a_i$ that ends in $a_i$

Recursion for $L(i)$:

$$L(i) = 1 + \max_{j<i \text{ and } a_j < a_i} L(j)$$

Length of the longest increasing subsequence:
A Correct Recursion

\[ L(i) \]: longest increasing subsequence in \( a_1, a_2, \ldots, a_i \) that ends in \( a_i \)

Recursion for \( L(i) \):

\[ L(i) = 1 + \max_{j<i \text{ and } a_j < a_i} L(j) \]

Length of the longest increasing subsequence: \( \max_{i=1}^{n} L(i) \).

How many subproblems?
A Correct Recursion

$L(i)$: longest increasing subsequence in $a_1, a_2, \ldots, a_i$ that ends in $a_i$

Recursion for $L(i)$:

$$L(i) = 1 + \max_{j<i \text{ and } a_j < a_i} L(j)$$

Length of the longest increasing subsequence: $\max_{i=1}^n L(i)$.

How many subproblems? $O(n)$
Running time for Recursion

\[ L(i) = 1 + \max_{j < i \text{ and } a_j < a_i} L(j) \]

\( T(i) \): time to compute \( L[i] \):

\[ T(i) = 1 + \sum_{j=1}^{i-1} T(i - 1) \quad \text{and} \quad T(1) = 1. \]

\( T(n) = \)
Longest Increasing Subsequence

Running time for Recursion

\[ L(i) = 1 + \max_{j < i \text{ and } a_j < a_i} L(j) \]

\( T(i) \): time to compute \( L[i] \):

\[ T(i) = 1 + \sum_{j=1}^{i-1} T(i-1) \quad \text{and} \quad T(1) = 1. \]

\( T(n) = 2^{n-1} \).
Iterative Algorithm via Memoization

Recurrence:

\[ L(i) = 1 + \max_{j < i \text{ and } a_j < a_i} L(j) \]

Iterative algorithm:

for \( i = 1 \) to \( n \) do
  \( L[i] = 1 \)
  for \( j = 1 \) to \( i-1 \) do
    if \( a_j < a_i \) and \( (1+L[j]) > L[i] \) then
      \( L[i] = 1 + L[j] \)

Output \( \max_{i=1}^{n} L[i] \)

Running Time:

\( O(n^2) \)
Iterative Algorithm via Memoization

Recurrence:

\[ L(i) = 1 + \max_{j<i \text{ and } a_j < a_i} L(j) \]

Iterative algorithm:

for \( i = 1 \) to \( n \) do
   \( L[i] = 1 \)
   for \( j = 1 \) to \( i-1 \) do
      if \( a_j < a_i \) and \( 1 + L[j] > L[i] \) then
         \( L[i] = 1 + L[j] \)

Output \( \max_{i=1}^n L[i] \)

Running Time: \( O(n^2) \)
Space:
Iterative Algorithm via Memoization

Recurrence:

\[ L(i) = 1 + \max_{j < i \text{ and } a_j < a_i} L(j) \]

Iterative algorithm:

for \( i = 1 \) to \( n \) do
   \( L[i] = 1 \)
   for \( j = 1 \) to \( i-1 \) do
      if \((a_j < a_i) \text{ and } (1+L[j]) > L[i]\) then
         \( L[i] = 1 + L[j] \)

Output \( \max_{i=1}^{n} L[i] \)

Running Time: \( O(n^2) \)

Space: \( O(n) \)
Computing an Optimum Solution

Keep track of decision when computing $L[i]$.

for $i = 1$ to $n$ do
  $L[i] = 1$
  prev[$i$] = 0 (* 0 is a sentinel value *)
  for $j = 1$ to $i-1$ do
    if ($a_j < a_i$) and ($1+L[j]$) > $L[i]$ then
      $L[i] = 1 + L[j]$
      prev[$i$] = $j$

Output $\max_{i=1}^{n} L[i]$

Exercise: show how to output an increasing sequence of length equal to $L[i]$ using the prev pointers.