Introduction to Dynamic Programming

Lecture 5

September 10, 2015

5.1: Introduction to Dynamic Programming

Recursion

Reduction:

Reduce one problem to another

Recursion

Recursion is a special case of reduction, where:

1. reduce problem to a *smaller* instance of *itself*, and
2. self-reduction.

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

Recursion

Reduction:

Reduce one problem to another

Recursion

Recursion is a special case of reduction, where:

1. reduce problem to a *smaller* instance of *itself*, and
2. self-reduction.

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

Recursion in Algorithm Design

1. **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.
2. **Divide and Conquer**: Problem reduced to multiple **independent** sub-problems that are solved separately. Conquer step puts together solution for bigger problem.

   Examples: Closest pair, deterministic median selection, quick sort.
3. **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

1. Fibonacci numbers defined by recurrence:
   $F(n) = F(n-1) + F(n-2)$ and $F(0) = 0, F(1) = 1.$
2. ... many interesting properties. A journal *The Fibonacci Quarterly*!
3. Known: $F_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n + \left(\frac{1-\sqrt{5}}{2}\right)^n\right] = \Theta(\phi^n),$
4.
   1. $F(n) = (\phi^n - (1-\phi)^n)/\sqrt{5}$ where ϕ is the golden ratio $\phi = (1+\sqrt{5})/2 \simeq 1.618$.
   2. $\lim_{n\to\infty} F(n+1)/F(n) = \phi$.

Fibonacci Numbers

1. Fibonacci numbers defined by recurrence:
   $F(n) = F(n-1) + F(n-2)$ and $F(0) = 0, F(1) = 1.$
2. ... many interesting properties. A journal *The Fibonacci Quarterly*!
3. Known: $F_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n + \left(\frac{1-\sqrt{5}}{2}\right)^n\right] = \Theta(\phi^n)$,
4.
   1. $F(n) = (\phi^n - (1-\phi)^n)/\sqrt{5}$ where ϕ is the golden ratio $\phi = (1+\sqrt{5})/2 \simeq 1.618$.
   2. $\lim_{n\to\infty} F(n+1)/F(n) = \phi$.

Fibonacci Numbers

1. Fibonacci numbers defined by recurrence:
   $F(n) = F(n-1) + F(n-2)$ and $F(0) = 0, F(1) = 1.$
2. ... many interesting properties. A journal *The Fibonacci Quarterly*!
3. Known: $F_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n + \left(\frac{1-\sqrt{5}}{2}\right)^n\right] = \Theta(\phi^n),$
4.
   1. $F(n) = (\phi^n - (1-\phi)^n)/\sqrt{5}$ where ϕ is the golden ratio $\phi = (1+\sqrt{5})/2 \simeq 1.618.$
   2. $\lim_{n\to\infty} F(n+1)/F(n) = \phi.$

Fibonacci Numbers

1. Fibonacci numbers defined by recurrence:
   $$F(n) = F(n-1) + F(n-2) \text{ and } F(0) = 0, F(1) = 1.$$
2. ... many interesting properties. A journal *The Fibonacci Quarterly*!
3. Known: $F_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n + \left(\frac{1-\sqrt{5}}{2}\right)^n\right] = \Theta(\phi^n)$,
4.
   1. $F(n) = (\phi^n - (1-\phi)^n)/\sqrt{5}$ where ϕ is the golden ratio $\phi = (1+\sqrt{5})/2 \simeq 1.618$.
   2. $\lim_{n\to\infty} F(n+1)/F(n) = \phi$.

Recursive Algorithm for Fibonacci Numbers

1. Question: Given n, compute $F(n)$.

   Fib(n):

   ```
   if (n = 0)
       return 0
   else if (n = 1)
       return 1
   else
       return Fib(n − 1) + Fib(n − 2)
   ```

2. Running time? Let $T(n)$ be the number of additions in Fib(n).
3. $T(n) = T(n-1) + T(n-2) + 1$ and $T(0) = T(1) = 0$
4. Roughly same as $F(n)$

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

The number of additions is exponential in n. Can we do better?

Recursive Algorithm for Fibonacci Numbers

1. Question: Given n, compute $F(n)$.

   Fib(n):

       if $(n = 0)$

           return 0

       else if $(n = 1)$

           return 1

       else

           return Fib$(n-1)$ + **Fib**$(n-2)$

2. Running time? Let $T(n)$ be the number of additions in Fib(n).
3. $T(n) = T(n-1) + T(n-2) + 1$ and $T(0) = T(1) = 0$
4. Roughly same as $F(n)$

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

The number of additions is exponential in n. Can we do better?

Recursive Algorithm for Fibonacci Numbers

1. Question: Given n, compute $F(n)$.

   Fib(n):

   if $(n = 0)$

       return 0

   else if $(n = 1)$

       return 1

   else

       return **Fib**$(n-1)$ + **Fib**$(n-2)$

2. Running time? Let $T(n)$ be the number of additions in Fib(n).
3. $T(n) = T(n-1) + T(n-2) + 1$ and $T(0) = T(1) = 0$
4. Roughly same as $F(n)$

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

The number of additions is exponential in n. Can we do better?

Recursive Algorithm for Fibonacci Numbers

1. Question: Given n, compute $F(n)$.

   Fib(n):

       if $(n = 0)$

           return 0

       else if $(n = 1)$

           return 1

       else

           return **Fib**$(n-1)$ + **Fib**$(n-2)$

2. Running time? Let $T(n)$ be the number of additions in Fib(n).
3. $T(n) = T(n-1) + T(n-2) + 1$ and $T(0) = T(1) = 0$
4. Roughly same as $F(n)$

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

The number of additions is exponential in n. Can we do better?

Recursive Algorithm for Fibonacci Numbers

1. Question: Given n, compute $F(n)$.

   Fib(n):

   if $(n = 0)$

   return 0

   else if $(n = 1)$

   return 1

   else

   return **Fib**$(n-1)$ + **Fib**$(n-2)$

2. Running time? Let $T(n)$ be the number of additions in Fib(n).
3. $T(n) = T(n-1) + T(n-2) + 1$ and $T(0) = T(1) = 0$
4. Roughly same as $F(n)$

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

The number of additions is exponential in n. Can we do better?

An iterative algorithm for Fibonacci numbers

FibIter(n):
    if $(n = 0)$ **then**
        return 0
    if $(n = 1)$ **then**
        return 1
    $F[0] = 0$
    $F[1] = 1$
    for i = 2 to n **do**
        $F[i] \Leftarrow F[i-1] + F[i-2]$
    return $F[n]$

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

An iterative algorithm for Fibonacci numbers

FibIter(n):
 if $(n = 0)$ **then**
  return 0
 if $(n = 1)$ **then**
  return 1
 $F[0] = 0$
 $F[1] = 1$
 for i = 2 to n **do**
  $F[i] \Leftarrow F[i-1] + F[i-2]$
 return $F[n]$

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

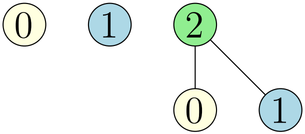
An iterative algorithm for Fibonacci numbers

FibIter(n):
    if $(n = 0)$ **then**
        return 0
    if $(n = 1)$ **then**
        return 1
    $F[0] = 0$
    $F[1] = 1$
    for i = 2 to n **do**
        $F[i] \Leftarrow F[i-1] + F[i-2]$
    return $F[n]$

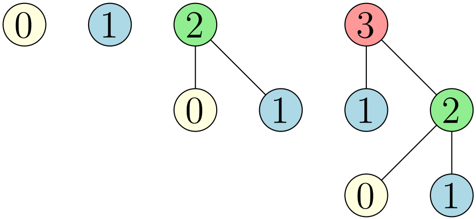
What is the running time of the algorithm? $O(n)$ additions.

Recursion tree for the Recursive Fibonacci

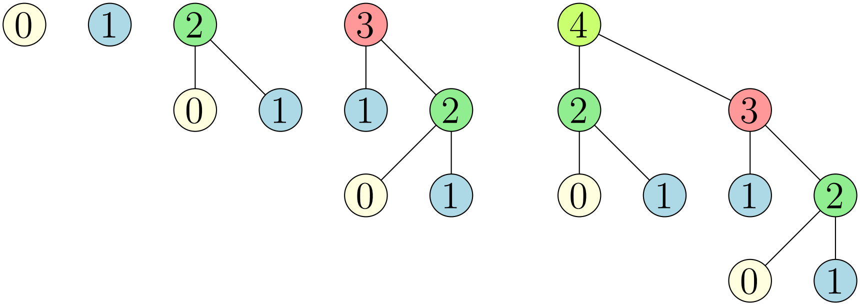
Recursion tree for the Recursive Fibonacci



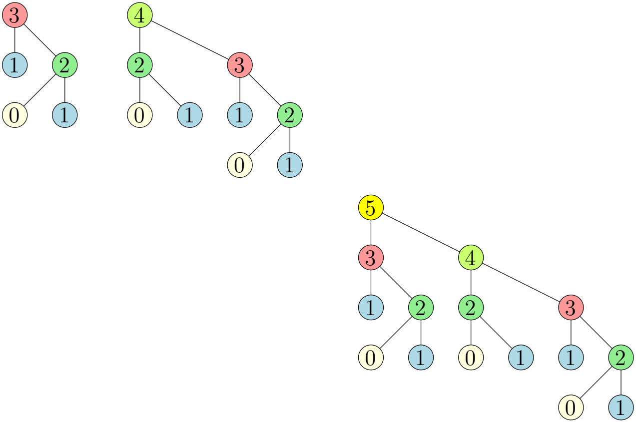
Recursion tree for the Recursive Fibonacci



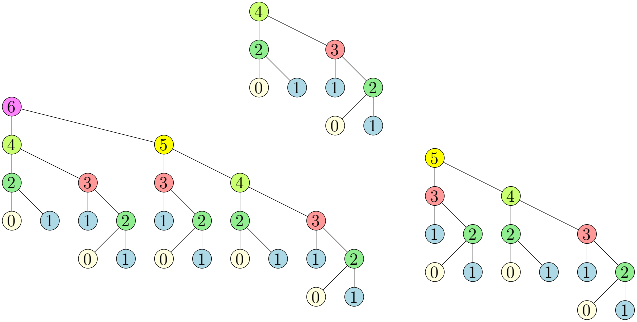
Recursion tree for the Recursive Fibonacci



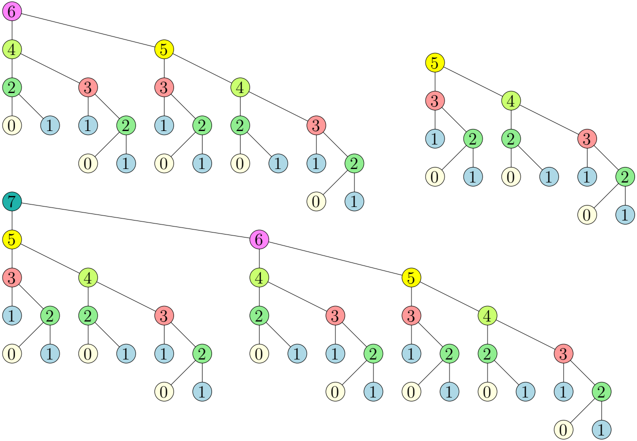
Recursion tree for the Recursive Fibonacci



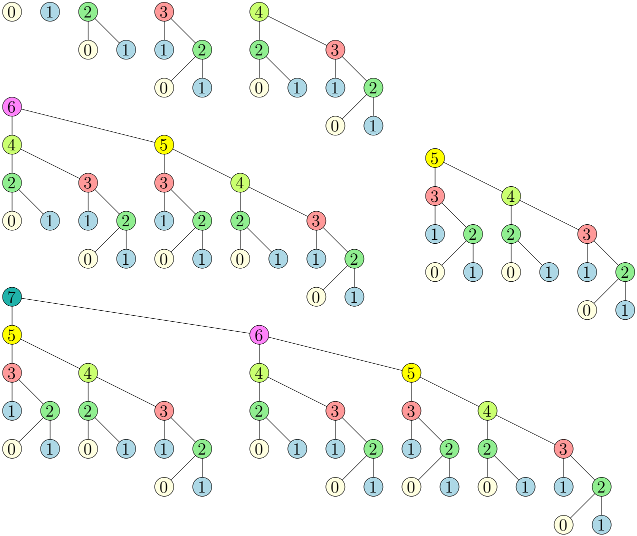
Recursion tree for the Recursive Fibonacci



Recursion tree for the Recursive Fibonacci



Recursion tree for Fibonacci



What is the difference?

1. Recursive/iterative:
   1. Recursive algorithm is computing the same numbers again and again.
   2. Iterative algorithm is storing computed values and building bottom up the final value. Memoization.
2. Dynamic programming:

Dynamic Programming:

Finding a recursion that can be *effectively/efficiently* memoized.

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

What is the difference?

1. Recursive/iterative:
    1. Recursive algorithm is computing the same numbers again and again.
    2. Iterative algorithm is storing computed values and building bottom up the final value. Memoization.
2. Dynamic programming:

Dynamic Programming:

Finding a recursion that can be *effectively/efficiently* memoized.

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

What is the difference?

1. Recursive/iterative:
   1. Recursive algorithm is computing the same numbers again and again.
   2. Iterative algorithm is storing computed values and building bottom up the final value. Memoization.
2. Dynamic programming:

Dynamic Programming:

Finding a recursion that can be *effectively/efficiently* memoized.

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

What is the difference?

1. Recursive/iterative:
    1. Recursive algorithm is computing the same numbers again and again.
    2. Iterative algorithm is storing computed values and building bottom up the final value. Memoization.
2. Dynamic programming:

Dynamic Programming:

Finding a recursion that can be *effectively/efficiently* memoized.

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

Automatic Memoization

1. Recursive version:

```
f(x_1, x_2, ..., x_d):
    CODE
```

2. Recursive version with memoization:

```
g(x_1, x_2, ..., x_d):
    if f already computed for (x_1, x_2, ..., x_d) then
        return value already computed
    NEW_CODE
```

3. NEW_CODE:
   1. Replaces any "**return** α" with
   2. Remember "$f(x_1, \ldots, x_d) = \alpha$"; **return** α.

Automatic Memoization

1. Recursive version:

```
f(x_1, x_2, ..., x_d):
      CODE
```

$f(x_1, x_2, \ldots, x_d)$:
CODE

2. Recursive version with memoization:

$g(x_1, x_2, \ldots, x_d)$:
if f already computed for $(x_1, x_2, \ldots, x_d)$ **then**
return value already computed
NEW_CODE

3. NEW_CODE:
   1. Replaces any "**return** α" with
   2. Remember "$f(x_1, \ldots, x_d) = \alpha$"; **return** α.

Automatic Memoization

1. Recursive version:

```
f(x_1, x_2, ..., x_d):
    CODE
```

   $f(x_1, x_2, \ldots, x_d)$:
   CODE

2. Recursive version with memoization:

   $g(x_1, x_2, \ldots, x_d)$:
   if f already computed for $(x_1, x_2, \ldots, x_d)$ **then**
   return value already computed
   NEW_CODE

3. NEW_CODE:
   1. Replaces any "**return** α" with
   2. Remember "$f(x_1, \ldots, x_d) = \alpha$"; **return** α.

Automatic Memoization

1. **Q:** How to convert recursive algorithm into efficient algorithm?
   ... Without explicitly doing an iterative algorithm?
2. Remember old computations!

   Fib(n) :

       if $(n = 0)$

           return 0

       if $(n = 1)$

           return 1

       if (**Fib**(n) `was previously computed`)

           return `stored value of Fib(n)`

       else

           return **Fib**$(n-1)$ + **Fib**$(n-2)$
3. How do we keep track of previously computed values?
4. Two methods: explicitly and implicitly (via a data structure).

Automatic Memoization

1. **Q:** How to convert recursive algorithm into efficient algorithm?
   ... Without explicitly doing an iterative algorithm?
2. Remember old computations!

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

3. How do we keep track of previously computed values?
4. Two methods: explicitly and implicitly (via a data structure).

Automatic Memoization

1. **Q:** How to convert recursive algorithm into efficient algorithm?
   ... Without explicitly doing an iterative algorithm?
2. Remember old computations!

   Fib(n) :
       if $(n = 0)$
           return 0
       if $(n = 1)$
           return 1
       if (**Fib**(n) `was previously computed`)
           return `stored value of Fib(n)`
       else
           return **Fib**$(n-1)$ + **Fib**$(n-2)$
3. How do we keep track of previously computed values?
4. Two methods: explicitly and implicitly (via a data structure).

Automatic Memoization

1. **Q:** How to convert recursive algorithm into efficient algorithm? ... Without explicitly doing an iterative algorithm?
2. Remember old computations!

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

3. How do we keep track of previously computed values?
4. Two methods: explicitly and implicitly (via a data structure).

Automatic Memoization

1. **Q:** How to convert recursive algorithm into efficient algorithm?
   ... Without explicitly doing an iterative algorithm?
2. Remember old computations!

   Fib(n):
       if $(n = 0)$
           return 0
       if $(n = 1)$
           return 1
       if (**Fib**(n) `was previously computed`)
           return `stored value of Fib(n)`
       else
           return **Fib**$(n-1)$ + **Fib**$(n-2)$
3. How do we keep track of previously computed values?
4. Two methods: explicitly and implicitly (via a data structure).

Automatic Memoization

1. **Q:** How to convert recursive algorithm into efficient algorithm?
   ... Without explicitly doing an iterative algorithm?
2. Remember old computations!

   Fib(n):

       if $(n = 0)$

           return 0

       if $(n = 1)$

           return 1

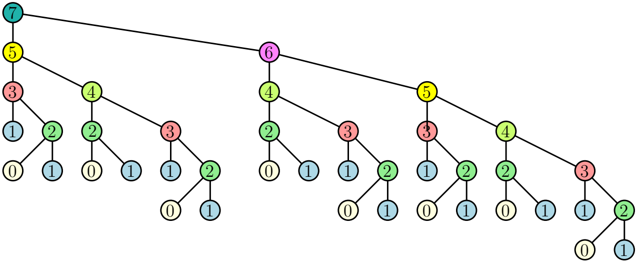
       if (**Fib**(n) `was previously computed`)

           return `stored value of Fib(n)`

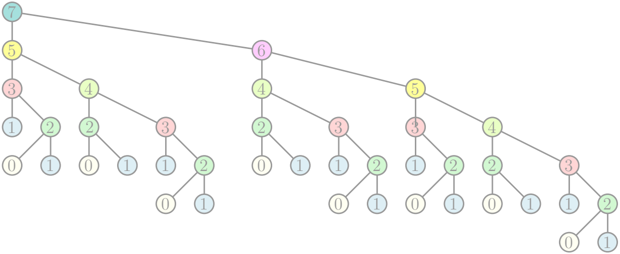
       else

           return **Fib**$(n-1)$ + **Fib**$(n-2)$
3. How do we keep track of previously computed values?
4. Two methods: explicitly and implicitly (via a data structure).

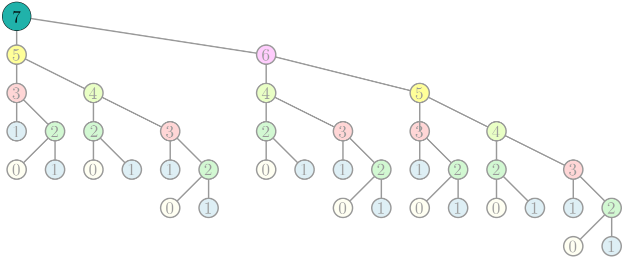
Recursion tree for the memoized Fib...



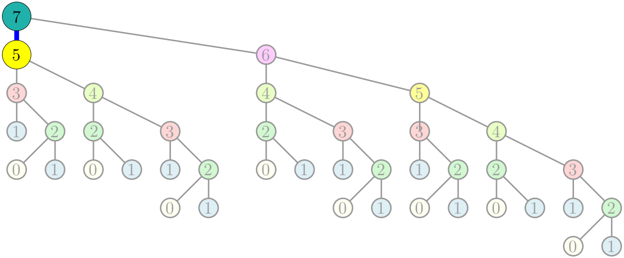
Recursion tree for the memoized Fib...



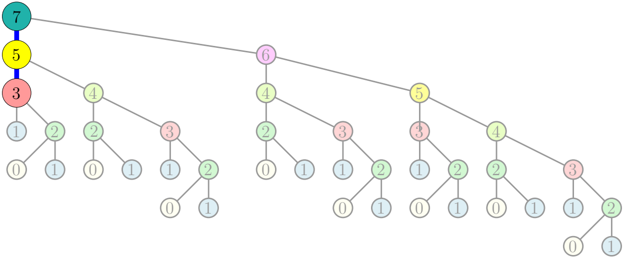
Recursion tree for the memoized Fib...



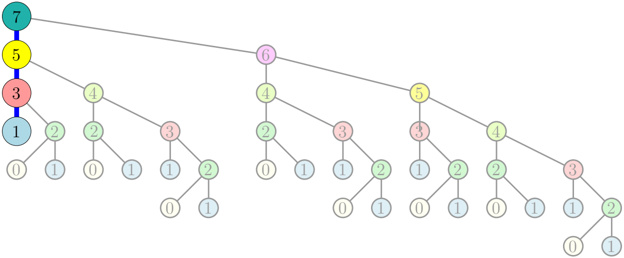
Recursion tree for the memoized Fib...



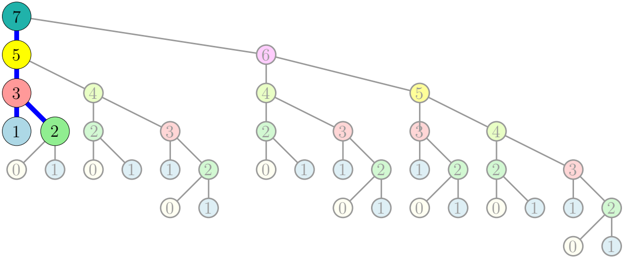
Recursion tree for the memoized Fib...



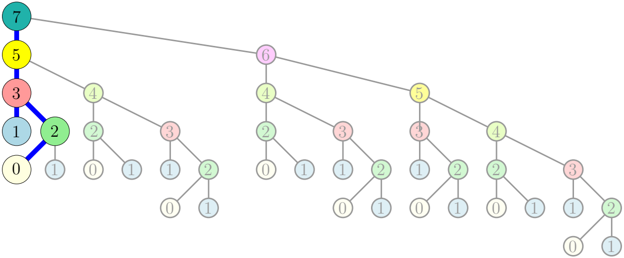
Recursion tree for the memoized Fib...



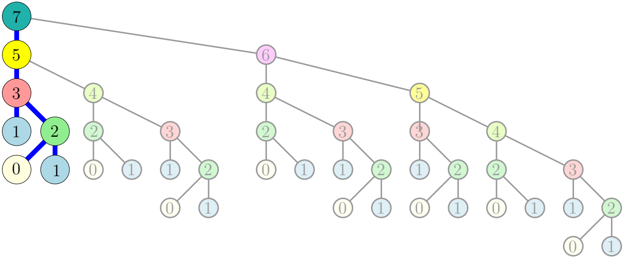
Recursion tree for the memoized Fib...



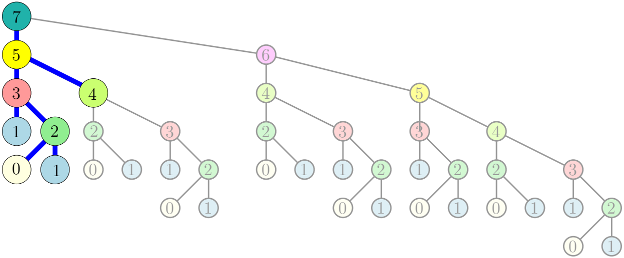
Recursion tree for the memoized Fib...



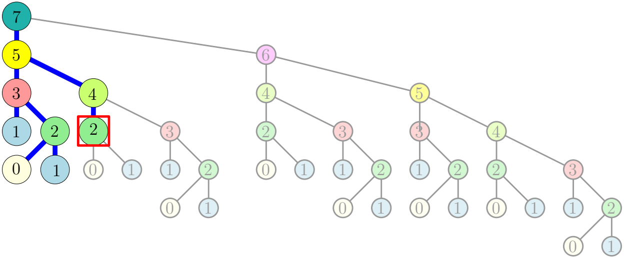
Recursion tree for the memoized Fib...



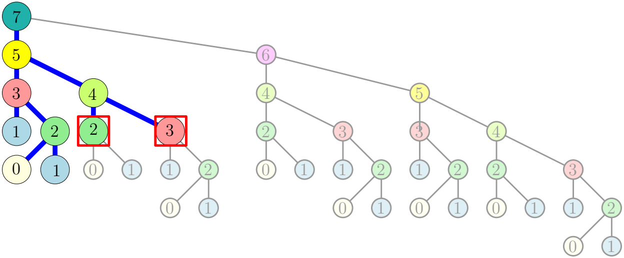
Recursion tree for the memoized Fib...



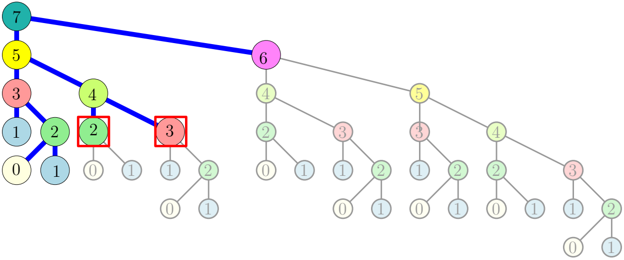
Recursion tree for the memoized Fib...



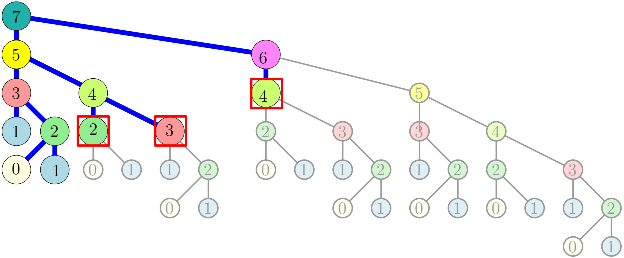
Recursion tree for the memoized Fib...



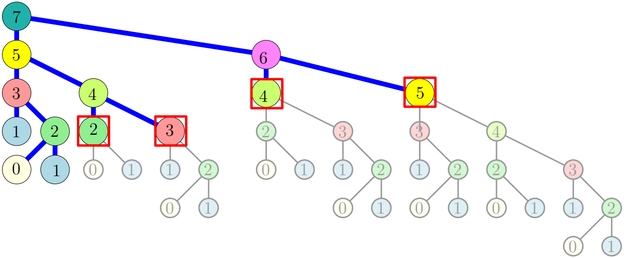
Recursion tree for the memoized Fib...



Recursion tree for the memoized Fib...



Recursion tree for the memoized Fib...



Automatic explicit memoization

1. Initialize table/array M of size n: $M[i] = -1$ for $i = 0, \ldots, n$.
2. Resulting code:

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

3. Need to know upfront the number of subproblems to allocate memory.

Automatic explicit memoization

1. Initialize table/array M of size n: $M[i] = -1$ for $i = 0, \ldots, n$.
2. Resulting code:

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

3. Need to know upfront the number of subproblems to allocate memory.

Automatic explicit memoization

1. Initialize table/array M of size n: $M[i] = -1$ for $i = 0, \ldots, n$.
2. Resulting code:

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

3. Need to know upfront the number of subproblems to allocate memory.

Automatic implicit memoization

Initialize a (dynamic) dictionary data structure D to empty

```
Fib(n):
    if (n = 0)
        return 0
    if (n = 1)
        return 1
    if (n is already in D)
        return value stored with n in D
    val ⇐ Fib(n − 1) + Fib(n − 2)
    Store (n, val) in D
    return val
```

Explicit vs Implicit Memoization

1. Explicit memoization (iterative algorithm) preferred:
   1. analyze problem ahead of time
   2. Allows for efficient memory allocation and access.
2. Implicit (automatic) memoization:
   1. problem structure or algorithm is not well understood.
   2. Need to pay overhead of data-structure.
   3. Functional languages (e.g., LISP) automatically do memoization, usually via hashing based dictionaries.

Explicit vs Implicit Memoization

1. Explicit memoization (iterative algorithm) preferred:
    1. analyze problem ahead of time
    2. Allows for efficient memory allocation and access.
2. Implicit (automatic) memoization:
    1. problem structure or algorithm is not well understood.
    2. Need to pay overhead of data-structure.
    3. Functional languages (e.g., LISP) automatically do memoization, usually via hashing based dictionaries.

Explicit vs Implicit Memoization

1. Explicit memoization (iterative algorithm) preferred:
   1. analyze problem ahead of time
   2. Allows for efficient memory allocation and access.
2. Implicit (automatic) memoization:
   1. problem structure or algorithm is not well understood.
   2. Need to pay overhead of data-structure.
   3. Functional languages (e.g., LISP) automatically do memoization, usually via hashing based dictionaries.

Explicit vs Implicit Memoization

1. Explicit memoization (iterative algorithm) preferred:
    1. analyze problem ahead of time
    2. Allows for efficient memory allocation and access.
2. Implicit (automatic) memoization:
    1. problem structure or algorithm is not well understood.
    2. Need to pay overhead of data-structure.
    3. Functional languages (e.g., LISP) automatically do memoization, usually via hashing based dictionaries.

Explicit vs Implicit Memoization

1. Explicit memoization (iterative algorithm) preferred:
    1. analyze problem ahead of time
    2. Allows for efficient memory allocation and access.
2. Implicit (automatic) memoization:
    1. problem structure or algorithm is not well understood.
    2. Need to pay overhead of data-structure.
    3. Functional languages (e.g., LISP) automatically do memoization, usually via hashing based dictionaries.

Back to Fibonacci Numbers

1. **Q:** Is the iterative algorithm a **polynomial** time algorithm? Does it take $O(n)$ time?
2. input is n and hence input size is $\Theta(\log n)$
3. output is $F(n)$ and output size is $\Theta(n)$. Why?
4. $\implies$ Output sizes exponential in input size...
   ... so no polynomial time algorithm possible!
   1. Running time of iterative algorithm: $\Theta(n)$ additions but number sizes are $O(n)$ bits long!
      $\implies$ Total running time is $O(n^2)$. And $\Theta(n^2)$. Why?
   2. Running time of recursive algorithm $O(n\phi^n)$
      1. Really: $O(\phi^n)$ by careful analysis.
      2. Doubly exponential in input size and exponential even in output size.

Back to Fibonacci Numbers

1. **Q:** Is the iterative algorithm a **polynomial** time algorithm? Does it take $O(n)$ time?
2. input is n and hence input size is $\Theta(\log n)$
3. output is $F(n)$ and output size is $\Theta(n)$. Why?
4. $\implies$ Output sizes exponential in input size... ... so no polynomial time algorithm possible!
   1. Running time of iterative algorithm: $\Theta(n)$ additions but number sizes are $O(n)$ bits long! $\implies$ Total running time is $O(n^2)$. And $\Theta(n^2)$. Why?
   2. Running time of recursive algorithm $O(n\phi^n)$
      1. Really: $O(\phi^n)$ by careful analysis.
      2. Doubly exponential in input size and exponential even in output size.

Back to Fibonacci Numbers

1. **Q:** Is the iterative algorithm a **polynomial** time algorithm? Does it take $O(n)$ time?
2. input is n and hence input size is $\Theta(\log n)$
3. output is $F(n)$ and output size is $\Theta(n)$. Why?
4. $\implies$ Output sizes exponential in input size...
   ... so no polynomial time algorithm possible!
   1. Running time of iterative algorithm: $\Theta(n)$ additions but number sizes are $O(n)$ bits long!
      $\implies$ Total running time is $O(n^2)$. And $\Theta(n^2)$. Why?
   2. Running time of recursive algorithm $O(n\phi^n)$
      1. Really: $O(\phi^n)$ by careful analysis.
      2. Doubly exponential in input size and exponential even in output size.

Back to Fibonacci Numbers

1. **Q:** Is the iterative algorithm a **polynomial** time algorithm? Does it take $O(n)$ time?
2. input is n and hence input size is $\Theta(\log n)$
3. output is $F(n)$ and output size is $\Theta(n)$. Why?
4. $\implies$ Output sizes exponential in input size...
   ... so no polynomial time algorithm possible!
   1. Running time of iterative algorithm: $\Theta(n)$ additions but number sizes are $O(n)$ bits long!
      $\implies$ Total running time is $O(n^2)$. And $\Theta(n^2)$. Why?
   2. Running time of recursive algorithm $O(n\phi^n)$
      1. Really: $O(\phi^n)$ by careful analysis.
      2. Doubly exponential in input size and exponential even in output size.

Back to Fibonacci Numbers

1. **Q:** Is the iterative algorithm a **polynomial** time algorithm? Does it take $O(n)$ time?
2. input is n and hence input size is $\Theta(\log n)$
3. output is $F(n)$ and output size is $\Theta(n)$. Why?
4. $\Longrightarrow$ Output sizes exponential in input size...
   ... so no polynomial time algorithm possible!
   1. Running time of iterative algorithm: $\Theta(n)$ additions but number sizes are $O(n)$ bits long!
      $\Longrightarrow$ Total running time is $O(n^2)$. And $\Theta(n^2)$. Why?
   2. Running time of recursive algorithm $O(n\phi^n)$
      1. Really: $O(\phi^n)$ by careful analysis.
      2. Doubly exponential in input size and exponential even in output size.

Back to Fibonacci Numbers

1. **Q:** Is the iterative algorithm a **polynomial** time algorithm? Does it take $O(n)$ time?
2. input is n and hence input size is $\Theta(\log n)$
3. output is $F(n)$ and output size is $\Theta(n)$. Why?
4. $\implies$ Output sizes exponential in input size...
   ... so no polynomial time algorithm possible!
   1. Running time of iterative algorithm: $\Theta(n)$ additions but number sizes are $O(n)$ bits long!
      $\implies$ Total running time is $O(n^2)$. And $\Theta(n^2)$. Why?
   2. Running time of recursive algorithm $O(n\phi^n)$
      1. Really: $O(\phi^n)$ by careful analysis.
      2. Doubly exponential in input size and exponential even in output size.

Back to Fibonacci Numbers

1. **Q:** Is the iterative algorithm a **polynomial** time algorithm? Does it take $O(n)$ time?
2. input is n and hence input size is $\Theta(\log n)$
3. output is $F(n)$ and output size is $\Theta(n)$. Why?
4. $\implies$ Output sizes exponential in input size...
   ... so no polynomial time algorithm possible!
   1. Running time of iterative algorithm: $\Theta(n)$ additions but number sizes are $O(n)$ bits long!
      $\implies$ Total running time is $O(n^2)$. And $\Theta(n^2)$. Why?
   2. Running time of recursive algorithm $O(n\phi^n)$
      1. Really: $O(\phi^n)$ by careful analysis.
      2. Doubly exponential in input size and exponential even in output size.

Back to Fibonacci Numbers

1. **Q:** Is the iterative algorithm a **polynomial** time algorithm? Does it take $O(n)$ time?
2. input is n and hence input size is $\Theta(\log n)$
3. output is $F(n)$ and output size is $\Theta(n)$. Why?
4. $\implies$ Output sizes exponential in input size...
   ... so no polynomial time algorithm possible!
   1. Running time of iterative algorithm: $\Theta(n)$ additions but number sizes are $O(n)$ bits long!
      $\implies$ Total running time is $O(n^2)$. And $\Theta(n^2)$. Why?
   2. Running time of recursive algorithm $O(n\phi^n)$
      1. Really: $O(\phi^n)$ by careful analysis.
      2. Doubly exponential in input size and exponential even in output size.

Back to Fibonacci Numbers

1. **Q:** Is the iterative algorithm a **polynomial** time algorithm? Does it take $O(n)$ time?
2. input is n and hence input size is $\Theta(\log n)$
3. output is $F(n)$ and output size is $\Theta(n)$. Why?
4. $\Longrightarrow$ Output sizes exponential in input size...
   ... so no polynomial time algorithm possible!
   1. Running time of iterative algorithm: $\Theta(n)$ additions but number sizes are $O(n)$ bits long!
      $\Longrightarrow$ Total running time is $O(n^2)$. And $\Theta(n^2)$. Why?
   2. Running time of recursive algorithm $O(n\phi^n)$
      1. Really: $O(\phi^n)$ by careful analysis.
      2. Doubly exponential in input size and exponential even in output size.

More on fast Fibonacci numbers

$$\begin{pmatrix} y \\ x+y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}.$$

As such,

$$\begin{pmatrix} F_{n-1} \\ F_n \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} F_{n-2} \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^2 \begin{pmatrix} F_{n-3} \\ F_{n-2} \end{pmatrix}$$
$$= \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{n-3} \begin{pmatrix} F_2 \\ F_1 \end{pmatrix}.$$

Thus, computing the nth Fibonacci number can be done by computing $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{n-3}$. Which can be done in $O(\log n)$ time (how?). What is wrong?

5.3: How to come up with dynamic programming algorithm, and how to analyze it

Coming up with dynamic programming algorithm

1. Given a problem...
2. ..define precisely a recursive subproblem.
    1. minimize # of parameters defining recursive subproblem.
    2. For each parameter, try to minimize the number of values it can have.
3. Come up with a recursive algorithm solving the problem, that...
    1. do as little work as possible...
    2. ...except for the recursive calls.
    3. Think on recursive calls as being "free".

Coming up with dynamic programming algorithm

1. Given a problem...
2. ..define precisely a recursive subproblem.
   1. minimize # of parameters defining recursive subproblem.
   2. For each parameter, try to minimize the number of values it can have.
3. Come up with a recursive algorithm solving the problem, that...
   1. do as little work as possible...
   2. ...except for the recursive calls.
   3. Think on recursive calls as being “free”.

Coming up with dynamic programming algorithm

1. Given a problem...
2. ..define precisely a recursive subproblem.
   1. minimize # of parameters defining recursive subproblem.
   2. For each parameter, try to minimize the number of values it can have.
3. Come up with a recursive algorithm solving the problem, that...
   1. do as little work as possible...
   2. ...except for the recursive calls.
   3. Think on recursive calls as being “free”.

Coming up with dynamic programming algorithm

1. Given a problem...
2. ..define precisely a recursive subproblem.
   1. minimize # of parameters defining recursive subproblem.
   2. For each parameter, try to minimize the number of values it can have.
3. Come up with a recursive algorithm solving the problem, that...
   1. do as little work as possible...
   2. ...except for the recursive calls.
   3. Think on recursive calls as being “free”.

Coming up with dynamic programming algorithm

1. Given a problem...
2. ..define precisely a recursive subproblem.
    1. minimize # of parameters defining recursive subproblem.
    2. For each parameter, try to minimize the number of values it can have.
3. Come up with a recursive algorithm solving the problem, that...
    1. do as little work as possible...
    2. ...except for the recursive calls.
    3. Think on recursive calls as being “free”.

Coming up with dynamic programming algorithm

1. Given a problem...
2. ..define precisely a recursive subproblem.
    1. minimize # of parameters defining recursive subproblem.
    2. For each parameter, try to minimize the number of values it can have.
3. Come up with a recursive algorithm solving the problem, that...
    1. do as little work as possible...
    2. ...except for the recursive calls.
    3. Think on recursive calls as being "free".

Analyzing a dynamic programming algorithm

1. S: number of distinct subproblems encountered in recursion.
2. T: amount of work spent in recursive call, ignoring the recursive calls themselves.
3. Using memoization, the resulting dynamic algorithm would need:
    1. $O(S)$ space.
    2. $O(T \cdot S)$ running time.

Computing the number of distinct recursive calls

1. $f(x_1, x_2, \ldots, x_d)$: Recursive function with d parameters.
2. x_i can have n_i different values.
3. Total number of difference recursive calls: $N = n_1 \times n_2 \times \cdots \times n_d$.
4. Memoization would yield a table with $O(N)$ different entries.

Computing the number of distinct recursive calls

1. $f(x_1, x_2, \ldots, x_d)$: Recursive function with d parameters.
2. x_i can have n_i different values.
3. Total number of difference recursive calls: $N = n_1 \times n_2 \times \cdots \times n_d$.
4. Memoization would yield a table with $O(N)$ different entries.

Computing the number of distinct recursive calls

1. $f(x_1, x_2, \ldots, x_d)$: Recursive function with d parameters.
2. x_i can have n_i different values.
3. Total number of difference recursive calls: $N = n_1 \times n_2 \times \cdots \times n_d$.
4. Memoization would yield a table with $O(N)$ different entries.

Computing the number of distinct recursive calls

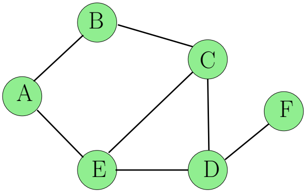
1. $f(x_1, x_2, \ldots, x_d)$: Recursive function with d parameters.
2. x_i can have n_i different values.
3. Total number of difference recursive calls:
   $N = n_1 \times n_2 \times \cdots \times n_d$.
4. Memoization would yield a table with $O(N)$ different entries.

5.4: Brute Force Search, Recursion and Backtracking

Maximum Independent Set in a Graph

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 $(u, v) \notin E$.

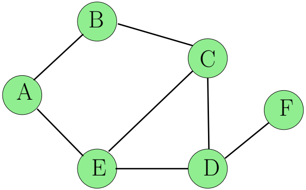


Some independent sets in graph above:

Maximum Independent Set Problem

Input Graph $G = (V, E)$

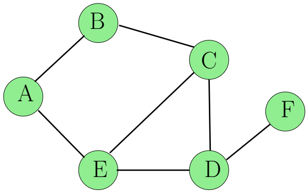
Goal Find maximum sized independent set in G



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

1. What we know:
   1. No *efficient* (polynomial time) algorithm for this problem.
   2. Problem is **NP-Complete**... no polynomial time algorithm
2. Brute force approach...

Brute-force algorithm:

Try all subsets of vertices.

Maximum Weight Independent Set Problem

1. What we know:
   1. No *efficient* (polynomial time) algorithm for this problem.
   2. Problem is **NP-Complete**... no polynomial time algorithm
2. Brute force approach...

Brute-force algorithm:

Try all subsets of vertices.

Brute-force enumeration

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

```
MaxIndSet(G = (V, E)):
  max = 0
  for each subset S ⊆ V do
    check if S is an independent set
    if S is an independent set and w(S) > max then
      max = w(S)
  Output max
```

2. Running time: suppose G has n vertices and m edges
   1. 2^n subsets of V
   2. checking each subset S takes $O(m)$ time
   3. total time is $O(m2^n)$

Brute-force enumeration

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

```
MaxIndSet(G = (V, E)):
    max = 0
    for each subset S ⊆ V do
        check if S is an independent set
        if S is an independent set and w(S) > max then
            max = w(S)
    Output max
```

2. Running time: suppose G has n vertices and m edges
    1. 2^n subsets of V
    2. checking each subset S takes $O(m)$ time
    3. total time is $O(m2^n)$

Brute-force enumeration

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

   MaxIndSet$(G = (V, E))$:
   $max = 0$
   for each subset $S \subseteq V$ **do**
   check if S is an independent set
   if S is an independent set and $w(S) > max$ **then**
   $max = w(S)$
   Output max

2. Running time: suppose G has n vertices and m edges
   1. 2^n subsets of V
   2. checking each subset S takes $O(m)$ time
   3. total time is $O(m2^n)$

Brute-force enumeration

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

   MaxIndSet$(G = (V, E))$:
   $max = 0$
   for each subset $S \subseteq V$ **do**
   check if S is an independent set
   if S is an independent set and $w(S) > max$ **then**
   $max = w(S)$
   Output max

2. Running time: suppose G has n vertices and m edges
   1. 2^n subsets of V
   2. checking each subset S takes $O(m)$ time
   3. total time is $O(m2^n)$

A Recursive Algorithm

1. Let $V = \{v_1, v_2, \ldots, v_n\}$.
2. For a vertex u let $N(u)$ be its neighbors.
3. We have that:

Observation

v_n: Vertex in the graph.
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.*

4. **RecursiveMIS**(G):

    if G `is empty` **then** `Output` 0
    $a =$ **RecursiveMIS**$(G - v_n)$
    $b = w(v_n) +$ **RecursiveMIS**$(G - v_n - N(v_n))$
    `Output` $\max(a, b)$

A Recursive Algorithm

1. Let $V = \{v_1, v_2, \ldots, v_n\}$.
2. For a vertex u let $N(u)$ be its neighbors.
3. We have that:

Observation

v_n: *Vertex in the graph.*
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.*

4. **RecursiveMIS**(G):

   if G `is empty` **then** `Output` 0
   $a =$ **RecursiveMIS**$(G - v_n)$
   $b = w(v_n) +$ **RecursiveMIS**$(G - v_n - N(v_n))$
   `Output` $\max(a, b)$

A Recursive Algorithm

1. Let $V = \{v_1, v_2, \ldots, v_n\}$.
2. For a vertex u let $N(u)$ be its neighbors.
3. We have that:

Observation

v_n: *Vertex in the graph.*

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.*

4. **RecursiveMIS**(G):

    if G `is empty` **then** `Output` 0

    $a =$ **RecursiveMIS**$(G - v_n)$

    $b = w(v_n)$ + **RecursiveMIS**$(G - v_n - N(v_n))$

    `Output` $\max(a, b)$

A Recursive Algorithm

1. Let $V = \{v_1, v_2, \ldots, v_n\}$.
2. For a vertex u let $N(u)$ be its neighbors.
3. We have that:

Observation

v_n: Vertex in the graph.
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.*

4. **RecursiveMIS**(G):

```
if G is empty then Output 0
a = RecursiveMIS(G − v_n)
b = w(v_n) + RecursiveMIS(G − v_n − N(v_n))
Output max(a, b)
```

Recursive Algorithms

..for Maximum Independent Set

1. Running time:
$$T(n) = T(n-1) + T\Big(n - 1 - deg(v_n)\Big) + O(1 + deg(v_n)).$$
2. $deg(v_n)$: degree of v_n. $T(0) = T(1) = 1$.
3. Worst case is when $deg(v_n) = 0$ when the recurrence becomes $T(n) = 2T(n-1) + O(1)$.
4. Solution to this is $T(n) = O(2^n)$.
5. **Improvement:** Over previous running time $O(m2^n)$.

1. Running time:
$$T(n) = T(n-1) + T\Big(n - 1 - deg(v_n)\Big) + O(1 + deg(v_n)).$$
2. $deg(v_n)$: degree of v_n. $T(0) = T(1) = 1$.
3. Worst case is when $deg(v_n) = 0$ when the recurrence becomes $T(n) = 2T(n-1) + O(1)$.
4. Solution to this is $T(n) = O(2^n)$.
5. **Improvement:** Over previous running time $O(m2^n)$.

Recursive Algorithms

..for Maximum Independent Set

1. Running time:
$$T(n) = T(n-1) + T\Big(n - 1 - deg(v_n)\Big) + O(1 + deg(v_n)).$$
2. $deg(v_n)$: degree of v_n. $T(0) = T(1) = 1$.
3. Worst case is when $deg(v_n) = 0$ when the recurrence becomes $T(n) = 2T(n-1) + O(1)$.
4. Solution to this is $T(n) = O(2^n)$.
5. **Improvement:** Over previous running time $O(m2^n)$.

Recursive Algorithms

..for Maximum Independent Set

1. Running time:
$$T(n) = T(n-1) + T\Big(n - 1 - deg(v_n)\Big) + O(1 + deg(v_n)).$$
2. $deg(v_n)$: degree of v_n. $T(0) = T(1) = 1$.
3. Worst case is when $deg(v_n) = 0$ when the recurrence becomes $T(n) = 2T(n-1) + O(1)$.
4. Solution to this is $T(n) = O(2^n)$.
5. **Improvement:** Over previous running time $O(m2^n)$.

Recursive Algorithms

..for Maximum Independent Set

1. Running time:
$$T(n) = T(n-1) + T\Big(n - 1 - deg(v_n)\Big) + O(1 + deg(v_n)).$$
2. $deg(v_n)$: degree of v_n. $T(0) = T(1) = 1$.
3. Worst case is when $deg(v_n) = 0$ when the recurrence becomes $T(n) = 2T(n-1) + O(1)$.
4. Solution to this is $T(n) = O(2^n)$.
5. **Improvement:** Over previous running time $O(m2^n)$.

Backtrack Search via Recursion

1. Recursive algorithm generates a tree of computation where each node is a smaller problem (subproblem)
2. Simple recursive algorithm computes/explores the whole tree blindly in some order.
3. Backtrack search is a way to explore the tree intelligently to prune the search space
    1. Some subproblems may be so simple that we can stop the recursive algorithm and solve it directly by some other method
    2. Memoization to avoid recomputing same problem
    3. Stop the recursion at a subproblem if it is clear that there is no need to explore further.
    4. Leads to a number of heuristics that are widely used in practice although the worst case running time may still be exponential.

5.5: Longest Increasing Subsequence

5.5.1: 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 $1 \leq i_1 < i_2 < \ldots < i_k \leq n$.

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**.

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 $1 \leq i_1 < i_2 < \ldots < i_k \leq n$.

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**.

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 $1 \leq i_1 < i_2 < \ldots < i_k \leq n$.

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

1. Sequence: $6, 3, 5, 2, 7, 8, 1, 9$
2. Subsequence of above sequence: $5, 2, 1$
3. Increasing sequence: $3, 5, 9, 17, 54$
4. Decreasing sequence: $34, 21, 7, 5, 1$
5. Increasing subsequence of the first sequence: $2, 7, 9$.

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

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

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

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

Naïve Enumeration

1. Assume $a_1, a_2, \ldots, a_n$ 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
```

2. Running time: $O(n2^n)$.
3. 2^n subsequences of a sequence of length n and $O(n)$ time to check if a given sequence is increasing.

Naïve Enumeration

1. Assume $a_1, a_2, \ldots, a_n$ 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
```

2. Running time: $O(n2^n)$.
3. 2^n subsequences of a sequence of length n and $O(n)$ time to check if a given sequence is increasing.

Naïve Enumeration

1. Assume $a_1, a_2, \ldots, a_n$ 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
```

2. Running time: $O(n2^n)$.
3. 2^n subsequences of a sequence of length n and $O(n)$ time to check if a given sequence is increasing.

Naïve Enumeration

1. Assume $a_1, a_2, \ldots, a_n$ 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
```

2. Running time: $O(n2^n)$.
3. 2^n subsequences of a sequence of length n and $O(n)$ time to check if a given sequence is increasing.

Naïve Enumeration

1. Assume $a_1, a_2, \ldots, a_n$ 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
```

2. Running time: $O(n2^n)$.
3. 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

1. **Q:** Can we find a recursive algorithm for LIS?
2. Algorithm: $\mathrm{LIS}(A[1..n])$:
   1. Case 1: Does not contain $A[n]$ in which case $\mathrm{LIS}(A[1..n]) = \mathrm{LIS}(A[1..(n-1)])$
   2. Case 2: contains $A[n]$ in which case $\mathrm{LIS}(A[1..n])$ is not so clear.
3. We have:

Observation

if $A[n]$ is in the longest increasing subsequence then all the elements before it must be smaller.

Recursive Approance: Take 1

LIS: Longest increasing subsequence

1. **Q:** Can we find a recursive algorithm for LIS?
2. Algorithm: $\mathrm{LIS}(A[1..n])$:
   1. Case 1: Does not contain $A[n]$ in which case $\mathrm{LIS}(A[1..n]) = \mathrm{LIS}(A[1..(n-1)])$
   2. Case 2: contains $A[n]$ in which case $\mathrm{LIS}(A[1..n])$ is not so clear.
3. We have:

Observation

if $A[n]$ is in the longest increasing subsequence then all the elements before it must be smaller.

Recursive Approach: Take 1

LIS: Longest increasing subsequence

1. **Q:** Can we find a recursive algorithm for LIS?
2. Algorithm: $\text{LIS}(A[1..n])$:
   1. Case 1: Does not contain $A[n]$ in which case $\text{LIS}(A[1..n]) = \text{LIS}(A[1..(n-1)])$
   2. Case 2: contains $A[n]$ in which case $\text{LIS}(A[1..n])$ is not so clear.
3. We have:

Observation

if $A[n]$ is in the longest increasing subsequence then all the elements before it must be smaller.

Recursive Approach: Take 1

LIS: Longest increasing subsequence

1. **Q:** Can we find a recursive algorithm for LIS?
2. Algorithm: $\text{LIS}(A[1..n])$:
   1. Case 1: Does not contain $A[n]$ in which case $\text{LIS}(A[1..n]) = \text{LIS}(A[1..(n-1)])$
   2. Case 2: contains $A[n]$ in which case $\text{LIS}(A[1..n])$ is not so clear.
3. We have:

Observation

if $A[n]$ is in the longest increasing subsequence then all the elements before it must be smaller.

Recursive Approach: Take 1

LIS: Longest increasing subsequence

1. **Q:** Can we find a recursive algorithm for LIS?
2. Algorithm: $\mathrm{LIS}(A[1..n])$:
   1. Case 1: Does not contain $A[n]$ in which case $\mathrm{LIS}(A[1..n]) = \mathrm{LIS}(A[1..(n-1)])$
   2. Case 2: contains $A[n]$ in which case $\mathrm{LIS}(A[1..n])$ is not so clear.
3. We have:

Observation

if $A[n]$ is in the longest increasing subsequence then all the elements before it must be smaller.

Recursive Approach: Take 1

```
algLIS(A[1..n]):
   if (n = 0) then return 0
   m = algLIS(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 ≤ n − 1 *)
   m = max(m, 1 + 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 1

```
algLIS(A[1..n]):
  if (n = 0) then return 0
  m = algLIS(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 ≤ n − 1 *)
  m = max(m, 1 + 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 1

```
algLIS(A[1..n]):
  if (n = 0) then return 0
  m = algLIS(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 ≤ n − 1 *)
  m = max(m, 1 + 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

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

Observation

1. *Case 2: try to find a subsequence in $A[1..(n-1)]$ restricted to numbers $\leq A[n]$.*
2. *Suggests: **LIS_smaller**$(A[1..n], x)$ computes the longest increasing subsequence in A where each number in the sequence is less than x.*

Recursive Approach: Take 2

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

Observation

1. *Case 2: try to find a subsequence in $A[1..(n-1)]$ restricted to numbers $\leq A[n]$.*
2. *Suggests:* **LIS_smaller**$(A[1..n], x)$ *computes the longest increasing subsequence in A where each number in the sequence is less than x.*

Recursive Approach: Take 2

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

Observation

1. *Case 2: try to find a subsequence in $A[1..(n-1)]$ restricted to numbers $\leq A[n]$.*
2. *Suggests:* **LIS_smaller**$(A[1..n], x)$ *computes 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 = LIS_smaller(A[1..(n − 1)], x)
    if (A[n] < x) then
        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: $T(n) \leq 2T(n-1) + O(1)$.

Question: Is there any advantage?

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 = LIS_smaller(A[1..(n − 1)], x)
  if (A[n] < x) then
     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: $T(n) \leq 2T(n-1) + O(1)$.

Question: Is there any advantage?

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 = LIS_smaller(A[1..(n − 1)], x)
    if (A[n] < x) then
        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: $T(n) \leq 2T(n-1) + O(1)$.

Question: Is there any advantage?

Recursive Algorithm: Take 2

Observation

1. *The number of* different *subproblems generated by* **LIS_smaller**$(A[1..n], x)$ *is* $O(n^2)$.
2. Memoization of recursive algorithm: $O(n^2)$ running time!
3. **Q:** What the recursive subproblem generated by **LIS_smaller**$(A[1..n], x)$?
4. Subproblem:
   1. For $0 \le i < n$ **LIS_smaller**$(A[1..i], y)$ where y is either x or one of $A[i+1], \ldots, A[n]$.

Observation

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

Recursive Algorithm: Take 2

1. **Observation**

   The number of different *subproblems generated by* **LIS_smaller**$(A[1..n], x)$ *is* $O(n^2)$.

2. Memoization of recursive algorithm: $O(n^2)$ running time!
3. **Q:** What the recursive subproblem generated by **LIS_smaller**$(A[1..n], x)$?
4. Subproblem:
   1. For $0 \leq i < n$ **LIS_smaller**$(A[1..i], y)$ where y is either x or one of $A[i+1], \ldots, A[n]$.
5. **Observation**

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

Recursive Algorithm: Take 2

1. **Observation**

   The number of different *subproblems generated by* **LIS_smaller**$(A[1..n], x)$ *is* $O(n^2)$.

2. Memoization of recursive algorithm: $O(n^2)$ running time!
3. **Q:** What the recursive subproblem generated by **LIS_smaller**$(A[1..n], x)$?
4. Subproblem:
   1. For $0 \leq i < n$ **LIS_smaller**$(A[1..i], y)$ where y is either x or one of $A[i+1], \ldots, A[n]$.
5. **Observation**

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

Recursive Algorithm: Take 2

Observation

1. *The number of* different *subproblems generated by* **LIS_smaller**$(A[1..n], x)$ *is* $O(n^2)$.
2. Memoization of recursive algorithm: $O(n^2)$ running time!
3. **Q:** What the recursive subproblem generated by **LIS_smaller**$(A[1..n], x)$?
4. Subproblem:
    1. 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

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

Recursive Algorithm: Take 2

Observation

1. *The number of* different *subproblems generated by* **LIS_smaller**$(A[1..n], x)$ *is* $O(n^2)$.
2. Memoization of recursive algorithm: $O(n^2)$ running time!
3. **Q:** What the recursive subproblem generated by **LIS_smaller**$(A[1..n], x)$?
4. Subproblem:
   1. 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

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

Recursive Algorithm: Take 3

Definition

1. **LISEnding**$(A[1..n])$: length of longest increasing sub-sequence that *ends* in $A[n]$.
2. **Q:** Obtain a recursive expression?

Recursive formula

LISEnding$(A[1..n])$

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

Recursive Algorithm: Take 3

Definition

1. **LISEnding**$(A[1..n])$: length of longest increasing sub-sequence that *ends* in $A[n]$.
2. **Q:** Obtain a recursive expression?

Recursive formula

LISEnding$(A[1..n])$

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

Recursive Algorithm: Take 3

Definition

1. **LISEnding**$(A[1..n])$: length of longest increasing sub-sequence that *ends* in $A[n]$.
2. **Q:** Obtain a recursive expression?

Recursive formula

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

Recursive Algorithm: Take 3

```
LIS_ending_alg(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_alg(A[1..i]))
    return m
```

```
LIS(A[1..n]):
        return max_{i=1}^{n} LIS_ending_alg(A[1 . . . i])
```

Question:

How many distinct subproblems generated by LIS_ending_alg($A[1..n]$)? n.

Recursive Algorithm: Take 3

```
LIS_ending_alg(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_alg(A[1..i]))
    return m
```

```
LIS(A[1..n]):
        return max_{i=1}^{n} LIS_ending_alg(A[1 . . . i])
```

Question:

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

Recursive Algorithm: Take 3

```
LIS_ending_alg(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_alg(A[1..i]))
    return m
```

```
LIS(A[1..n]):
        return max_{i=1}^{n} LIS_ending_alg(A[1...i])
```

Question:

How many distinct subproblems generated by LIS_ending_alg($A[1..n]$)? n.

Iterative Algorithm via Memoization

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

```
LIS_ending_alg(A[1..n]):
  Array L[1..n]  (* L[i] = value of LIS_ending_alg(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_alg(A[1..n])
      return the maximum value in L
```

Iterative Algorithm via Memoization

Simplifying:

```
IS(A[1..n]):
 Array L[1..n]  (* L[i] stores the value LISEnding(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: $\Theta(n)$

Iterative Algorithm via Memoization

Simplifying:

```
IS(A[1..n]):
 Array L[1..n]  (* L[i] stores the value LISEnding(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: $\Theta(n)$

Iterative Algorithm via Memoization

Simplifying:

```
LIS(A[1..n]):
 Array L[1..n]  (* L[i] stores the value LISEnding(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: $\Theta(n)$

Iterative Algorithm via Memoization

Simplifying:

```
IS(A[1..n]):
 Array L[1..n]  (* L[i] stores the value LISEnding(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: $\Theta(n)$

Iterative Algorithm via Memoization

Simplifying:

```
LIS(A[1..n]):
 Array L[1..n]  (* L[i] stores the value LISEnding(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: $\Theta(n)$

Iterative Algorithm via Memoization

Simplifying:

```
LIS(A[1..n]):
 Array L[1..n]  (* L[i] stores the value LISEnding(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

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

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

Example

Example

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

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

Example

Example

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

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

Example

Example

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

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

Memoizing LIS_smaller

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

  return L[n,(n+1)]
```

Correctness: Via induction following the recursion (take 2)

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

Memoizing LIS_smaller

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

    return L[n,(n+1)]
```

Correctness: Via induction following the recursion (take 2)

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

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*)
  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**
        $L[i,j] = max(L[i,j], 1 + L[i-1,i])$

  return $L[n,(n+1)]$

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

Memoizing LIS_smaller

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

  return L[n,(n+1)]
```

Correctness: Via induction following the recursion (take 2)

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

Memoizing LIS_smaller

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

  return L[n,(n+1)]
```

Correctness: Via induction following the recursion (take 2)

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

Memoizing LIS_smaller

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

  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

1. $G = (\{s, 1, \ldots, n\}, \{\})$: directed graph.
   1. $\forall i, j$: If $i < j$ and $A[i] < A[j]$ then add the edge $i \rightarrow j$ to G.
   2. $\forall i$: Add $s \rightarrow i$.
2. The graph G is a DAG. LIS corresponds to longest path in G starting at s.
3. 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.

Dynamic Programming

1. Find a "smart" recursion for the problem in which the number of distinct subproblems is small; polynomial in the original problem size.
2. Estimate the number of subproblems, the time to evaluate each subproblem and the space needed to store the value. This gives an upper bound on the total running time if we use automatic memoization.
3. 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. This leads to an explicit algorithm.
4. Optimize the resulting algorithm further

5.6: Weighted Interval Scheduling

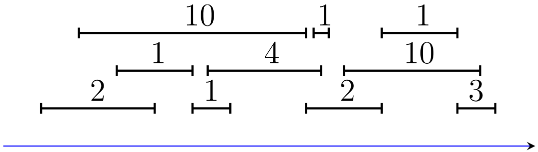
5.6.1: Weighted Interval Scheduling

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.

1. Two jobs with overlapping intervals cannot both be scheduled!

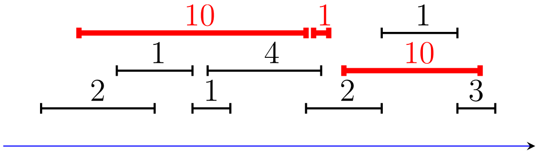


Weighted Interval Scheduling

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.

1. Two jobs with overlapping intervals cannot both be scheduled!



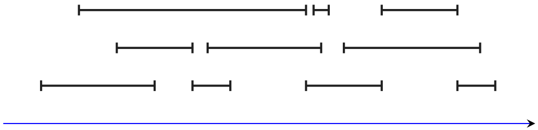
Interval Scheduling

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.

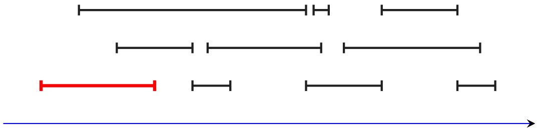
1. Greedy strategy of considering jobs according to finish times produces optimal schedule (to be seen later).



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.

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



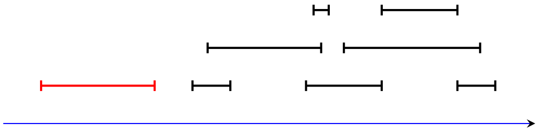
Interval Scheduling

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.

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



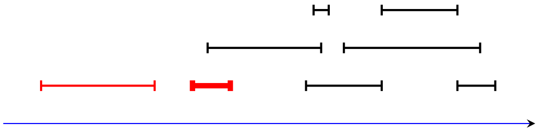
Interval Scheduling

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.

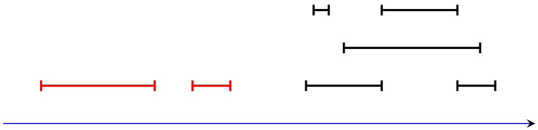
1. Greedy strategy of considering jobs according to finish times produces optimal schedule (to be seen later).



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.

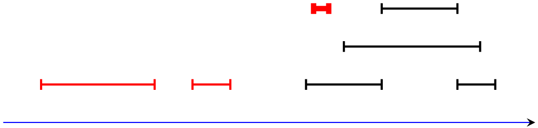
1. Greedy strategy of considering jobs according to finish times produces optimal schedule (to be seen later).



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.

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



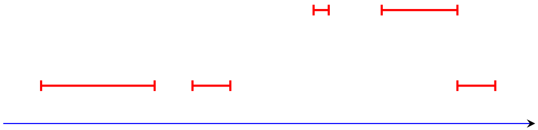
Interval Scheduling

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.

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



Greedy Strategies

1. Earliest finish time first
2. Largest weight/profit first
3. Largest weight to length ratio first
4. Shortest length first
5. ...

None of the above strategies lead to an optimum solution.

Moral: Greedy strategies often don't work!

Greedy Strategies

1. Earliest finish time first
2. Largest weight/profit first
3. Largest weight to length ratio first
4. Shortest length first
5. ...

None of the above strategies lead to an optimum solution.

Moral: Greedy strategies often don't work!

Reduction to...

Max Weight Independent Set Problem

1. Given weighted interval scheduling instance I create an instance of max weight independent set on a graph $G(I)$ as follows.
   1. For each interval i create a vertex v_i with weight w_i.
   2. Add an edge between v_i and v_j if i and j overlap.
2. Claim: max weight independent set in $G(I)$ has weight equal to max weight set of intervals in I that do not overlap

Reduction to...

Max Weight Independent Set Problem

1. Given weighted interval scheduling instance I create an instance of max weight independent set on a graph $G(I)$ as follows.
   1. For each interval i create a vertex v_i with weight w_i.
   2. Add an edge between v_i and v_j if i and j overlap.
2. Claim: max weight independent set in $G(I)$ has weight equal to max weight set of intervals in I that do not overlap

Reduction to...

Max Weight Independent Set Problem

1. There is a reduction from **Weighted Interval Scheduling** to **Independent Set**.
2. Can use structure of original problem for efficient algorithm?
3. **Independent Set** in general is **NP-Complete**.

Reduction to...

Max Weight Independent Set Problem

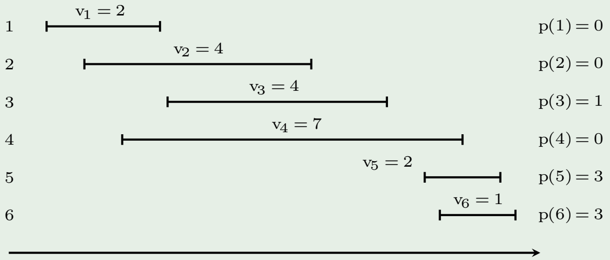
1. There is a reduction from **Weighted Interval Scheduling** to **Independent Set**.
2. Can use structure of original problem for efficient algorithm?
3. **Independent Set** in general is **NP-Complete**.

Conventions

Definition

1. Let the requests be sorted according to finish time, i.e., $i < j$ implies $f_i \leq f_j$
2. 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.*

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

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)} ← Schedule(p(n))
    O_{n-1} ← 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

...

A Recursive Algorithm

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) <- Schedule(p(n))
    O_{n-1} <- 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

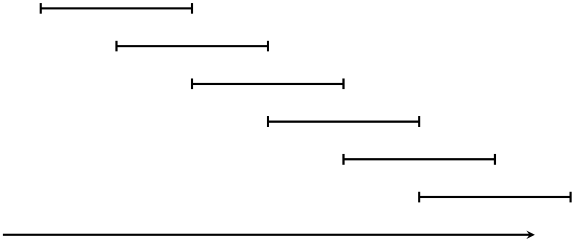


Figure: Bad instance for recursive algorithm

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.

Bad Example

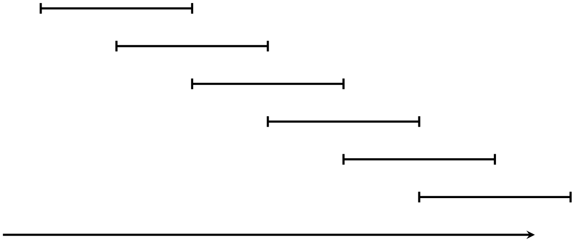


Figure: Bad instance for recursive algorithm

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

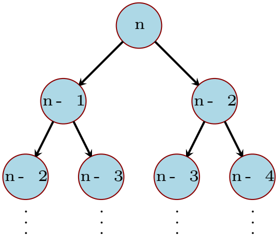


Figure: Label of node indicates size of sub-problem. Tree of sub-problems grows very quickly

Memo(r)ization

Observation

1. *Number of different sub-problems in recursive algorithm is* $O(n)$*; they are* $O_1, O_2, \ldots, O_{n-1}$
2. *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

```
schdlMem(j)
   if j = 0 then return 0
   if M[j] is defined then
      return M[j]
   if M[j] is not defined then
      M[j] = max( w(v_j) + schdlMem(p(j)),
                  schdlMem(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...)

Recursive Solution with Memoization

schdIMem(j)
- **if** $j = 0$ **then return** 0
- **if** $M[j]$ is defined **then** **return** $M[j]$
- **if** $M[j]$ is not defined **then**
  - $M[j] = max\left(\begin{array}{l} w(v_j) + \textbf{schdIMem}(p(j)), \\ \textbf{schdIMem}(j-1) \end{array}\right)$
  - **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...)

Recursive Solution with Memoization

schdIMem(j)
  if $j = 0$ **then return** 0
  if $M[j]$ `is defined` **then** **return** $M[j]$
  if $M[j]$ `is not defined` **then**

$$M[j] = max\left(\begin{array}{l} w(v_j) + \textbf{schdIMem}(p(j)), \\ \textbf{schdIMem}(j-1) \end{array} \right)$$

  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...)

Recursive Solution with Memoization

schdIMem(j)
  if $j = 0$ **then return** 0
  if $M[j]$ **return** `defined` $M[j]$ **then**
  if $M[j]$ `is not defined` **then**
    $M[j] = max\begin{pmatrix} w(v_j) + \textbf{schdIMem}(p(j)), \\ \textbf{schdIMem}(j-1) \end{pmatrix}$
    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...)

Recursive Solution with Memoization

```
schdIMem(j)
   if j = 0 then return 0
   if M[j] is defined then return M[j]
   if M[j] is not defined then
      M[j] = max( w(v_j) + schdIMem(p(j)),
                  schdIMem(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**
$M[i] = \max\Big(w(v_i) + M[p(i)], M[i-1]\Big)$

M: table of subproblems

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

Back to Weighted Interval Scheduling

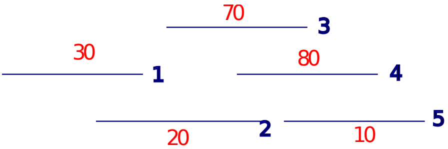
Iterative Solution

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

M: table of subproblems

1. Implicitly dynamic programming fills the values of M.
2. Recursion determines order in which table is filled up.
3. 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

1. 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**
   $\quad M[i] = max\Big(w(v_i) + M[p(i)],\ M[i-1]\Big)$
   $\quad$ **if** $w(v_i) + M[p(i)] < M[i-1]$ **then**
   $\quad\quad S[i] = S[i-1]$
   $\quad$ **else**
   $\quad\quad S[i] = S[p(i)] \cup \{i\}$

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

Computing Solutions + First Attempt

1. 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**
   $\quad M[i] = max\Big(w(v_i) + M[p(i)],\ M[i-1]\Big)$
   $\quad$ **if** $w(v_i) + M[p(i)] < M[i-1]$ **then**
   $\quad\quad S[i] = S[i-1]$
   $\quad$ **else**
   $\quad\quad S[i] = S[p(i)] \cup \{i\}$

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

Computing Solutions + First Attempt

1. 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**
  $M[i] = max\Big(w(v_i) + M[p(i)],\ M[i-1]\Big)$
  if $w(v_i) + M[p(i)] < M[i-1]$ **then**
    $S[i] = S[i-1]$
  else
    $S[i] = S[p(i)] \cup \{i\}$

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

Computing Solutions + First Attempt

1. 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**
   $\quad M[i] = max\Big(w(v_i) + M[p(i)],\ M[i-1]\Big)$
   $\quad$ **if** $w(v_i) + M[p(i)] < M[i-1]$ **then**
   $\quad\quad S[i] = S[i-1]$
   $\quad$ **else**
   $\quad\quad S[i] = S[p(i)] \cup \{i\}$

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

Computing Solutions + First Attempt

1. 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)] ∪ {i}
```

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

Computing Solutions + First Attempt

1. 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**
$M[i] = max\Big(w(v_i) + M[p(i)],\; M[i-1]\Big)$
if $w(v_i) + M[p(i)] < M[i-1]$ **then**
$S[i] = S[i-1]$
else
$S[i] = S[p(i)] \cup \{i\}$

2. Naïvely updating $S[]$ takes $O(n)$ time
3. Total running time is $O(n^2)$
4. 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 (v_j + M[p(j)] > M[j − 1]) then
    return findSolution(p(j)) ∪ {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:

1. Keep track of the *decision* in computing the optimum value of a sub-problem. decision space depends on recursion
2. 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

A generic strategy for computing solutions in dynamic programming:

1. Keep track of the *decision* in computing the optimum value of a sub-problem. decision space depends on recursion
2. 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
        else Decision[i] = 0
S = ∅, i = n
while (i > 0) do
    if (Decision[i] = 1) then
        S = S ∪ {i}
        i = p(i)
    else
        i = i − 1
return S
```

Notes

Notes

Notes

Notes