Algorithms Week 5

Ljubomir Perković, DePaul University

One of the most famous sequences of numbers is the Fibonacci sequence:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

One of the most famous sequences of numbers is the Fibonacci sequence:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

The sequence can be defined recursively:

$$F(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ F(n-1) + F(n-2) & \text{if } n \geq 2 \end{cases}$$

One of the most famous sequences of numbers is the Fibonacci sequence:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

The sequence can be defined recursively:

$$F(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ F(n-1) + F(n-2) & \text{if } n \geq 2 \end{cases}$$

The Fibonacci numbers have various interesting properties, including being related to the golden ratio and its conjugate:

$$\phi = \frac{1+\sqrt{5}}{2} \approx 1.61803 \qquad \hat{\phi} = \frac{1-\sqrt{5}}{2} \approx -0.61803$$

You could prove by induction that $F(n) = \frac{\phi^n - \hat{\phi}^n}{\sqrt{5}}$.

Fibonacci number recursive algorithm

The obvious algorithm for computing $F(n)$:


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

Fibonacci number recursive algorithm

The obvious algorithm for computing $F(n)$:


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


What is the running time $T(n)$ of this algorithm on input n?

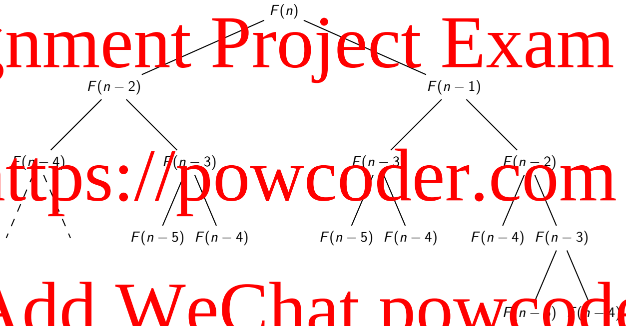
The obvious algorithm for computing $F(n)$:


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

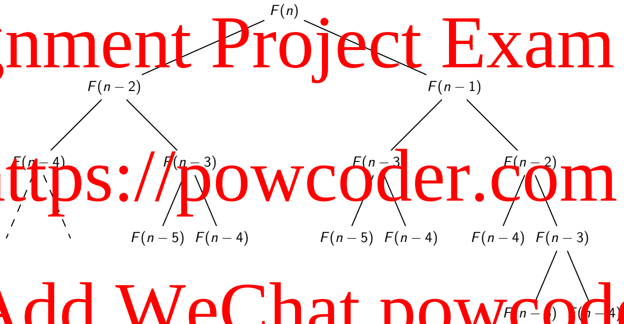

What is the running time $T(n)$ of this algorithm on input n?

$$T(n) = T(n-1) + T(n-2) + \Theta(1).$$

Consider the recursion tree for $F(n)$:

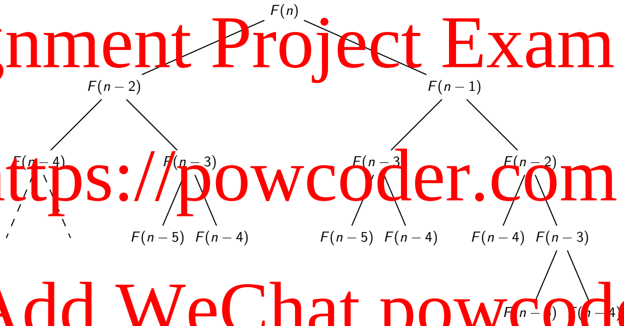


Consider the recursion tree for $F(n)$:



Insights:

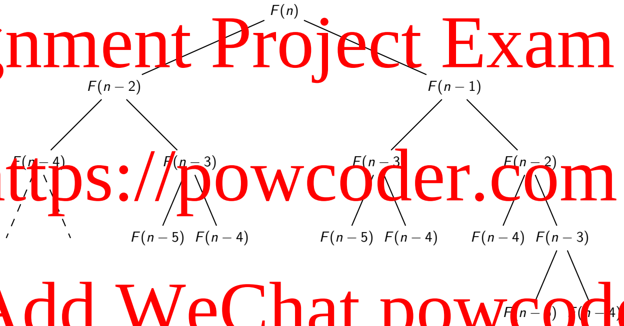
Consider the recursion tree for $F(n)$:



Insights:

- The number of recursive calls at depth i is 2^i

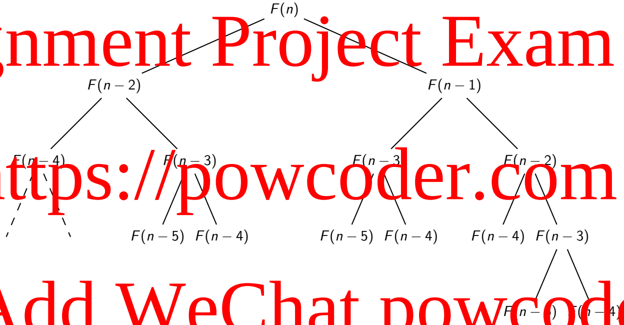
Consider the recursion tree for $F(n)$:



Insights:

- The number of recursive calls at depth i is 2^i
- The shallowest leaf is the leftmost one, with depth $\frac{n}{2}$.

Consider the recursion tree for $F(n)$:



Insights:

- The number of recursive calls at depth i is 2^i
- The shallowest leaf is the leftmost one, with depth $\frac{n}{2}$.

Thus the total number of recursive calls is $\Omega(2^{\frac{n}{2}})$, or exponential.

In fact, the solution to the recursion

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

is

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

In fact, the solution to the recursion

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

is

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

The reason the recursive algorithm is so slow is because the same recursive calls are recomputed over and over.

How do we avoid making all these duplicate recursive calls?

How do we avoid making all these duplicate recursive calls?

- One approach is to augment the recursive algorithm by storing the values returned by the recursive calls.

How do we avoid making all these duplicate recursive calls?

- One approach is to augment the recursive algorithm by storing the values returned by the recursive calls.
- Then, at the beginning of each recursive call, check to see if the value we want already exists.

How do we avoid making all these duplicate recursive calls?

- One approach is to augment the recursive algorithm by storing the values returned by the recursive calls.
- Then, at the beginning of each recursive call, check to see if the value we want already exists.

- If it does, then we re-use it. Otherwise we compute it.

How do we avoid making all these duplicate recursive calls?

- One approach is to augment the recursive algorithm by storing the values returned by the recursive calls.
- Then, at the beginning of each recursive call, check to see if the value we want already exists.

- If it does, then we re-use it. Otherwise we compute it.

This technique is called memoization.

How do we avoid making all these duplicate recursive calls?

- One approach is to augment the recursive algorithm by storing the values returned by the recursive calls.
- Then, at the beginning of each recursive call, check to see if the value we want already exists.
- If it does, then we re-use it. Otherwise we compute it.

This technique is called memoization. It:

- maintains the familiar, recursive, top-down structure of the algorithm
- but without the exponential costs of re-computing all the values.

Memoized algorithm for Fibonacci numbers:

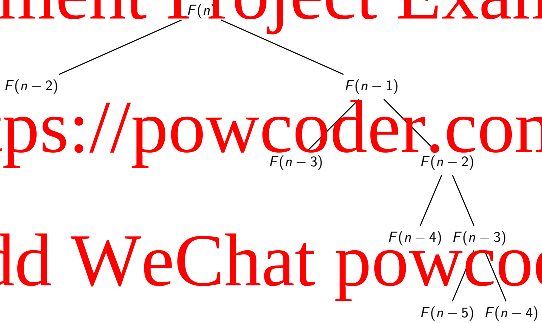

```
// F[0..n] is a global array
MemFib(n)
  if n = 0 or n = 1 then
    F[n] ← 1
  else if F[n] undefined
    F[n] ← MemFib(n-1) + MemFib(n-2)
  return F[n]
```

Memoized algorithm for Fibonacci numbers:


```
// F[0..n] is a global array
MemFib(n)
  if n = 0 or n = 1 then
    F[n] ← 1
  else if F[n] undefined
    F[n] ← MemFib(n-1) + MemFib(n-2)
  return F[n]
```


Running Time?

Running time of MemFib



Running time of MemFib

Memoized algorithm for Fibonacci numbers:


```
// F[0..n] is a global array
MemFib(n)
  if n = 0 or n = 1 then
    F[n] ← 1
  else if F[n] undefined
    F[n] ← MemFib(n-1) + MemFib(n-2)
  return F[n]
```


Running Time? $O(n)$!

Memoized algorithm for Fibonacci numbers:


```
// F[0..n] is a global array
MemFib(n)
  if n = 0 or n = 1 then
    F[n] ← 1
  else if F[n] undefined
    F[n] ← MemFib(n-1) + MemFib(n-2)
  return F[n]
```


Running Time? $O(n)$! An exponential speedup!

Rather than using a top-down approach to reach the bottom of the recursion and then compute Fibonacci numbers bottom-up, dynamic programming does the bottom-up approach directly and iteratively.

Rather than using a top-down approach to reach the bottom of the recursion and then compute Fibonacci numbers bottom-up, dynamic programming does the bottom-up approach directly and iteratively.

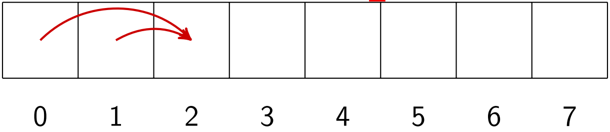
To compute $F(n)$:

0	1	2	3	4	5	6	7

Rather than using a top-down approach to reach the bottom of the recursion and then compute Fibonacci numbers bottom-up, dynamic programming does the bottom-up approach directly and iteratively.

To compute $F(n)$:

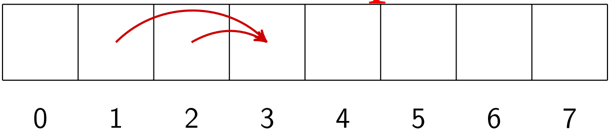
- We start by computing $F(2)$ from $F(0) = 1$ and $F(1) = 1$.



Rather than using a top-down approach to reach the bottom of the recursion and then compute Fibonacci numbers bottom-up, dynamic programming does the bottom-up approach directly and iteratively.

To compute $F(n)$:

- We start by computing $F(2)$ from $F(0) = 1$ and $F(1) = 1$.
- Then we compute $F(3)$ from $F(2)$ and $F(1)$,

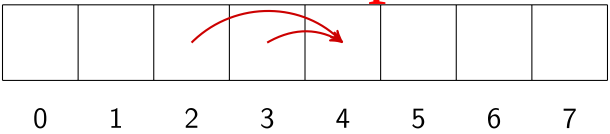


Dynamic Programming

Rather than using a top-down approach to reach the bottom of the recursion and then compute Fibonacci numbers bottom-up, dynamic programming does the bottom-up approach directly and iteratively.

To compute $F(n)$:

- We start by computing $F(2)$ from $F(0) = 1$ and $F(1) = 1$.
- Then we compute $F(3)$ from $F(2)$ and $F(1)$,
- and then $F(4)$ from $F(3)$ and $F(2)$, and so on . . .



```
IterFib(n)
  F[0] ← 1
  F[1] ← 1
  for i ← 2 to n do
    F[i] ← F[i-1] + F[i-2]
  return F[n]
```

```
IterFib(n)
  F[0] ← 1
  F[1] ← 1
  for i ← 2 to n do
    F[i] ← F[i-1] + F[i-2]
  return F[n]
```

Running time?

```
IterFib(n)
  F[0] ← 1
  F[1] ← 1
  for i ← 2 to n do
    F[i] ← F[i-1] + F[i-2]
  return F[n]
```

Running time? Clearly $\Theta(n)$.

```
IterFib(n)
  F[0] ← 1
  F[1] ← 1
  for i ← 2 to n do
    F[i] ← F[i-1] + F[i-2]
  return F[n]
```


Running time? Clearly $\Theta(n)$.

Note that this algorithm uses $\Theta(n)$ space. Can you modify it so it uses $\Theta(1)$ space?

The basic idea behind dynamic programming is recursion without repetition.

To develop a dynamic algorithm:

The basic idea behind dynamic programming is recursion without repetition.

To develop a dynamic algorithm:

1. Formulate the problem recursively

The basic idea behind dynamic programming is recursion without repetition.

To develop a dynamic algorithm:

1. Formulate the problem recursively
   1. Describe the problem that you want to solve recursively

The basic idea behind dynamic programming is recursion without repetition.

To develop a dynamic algorithm:

1. Formulate the problem recursively
   a. Describe the problem that you want to solve recursively
   b. Give a clear recursive formula or algorithm

The basic idea behind dynamic programming is recursion without repetition.

To develop a dynamic algorithm:

1. Formulate the problem recursively
   a. Describe the problem that you want to solve recursively
   b. Give a clear recursive formula or algorithm
2. Build solutions to your recurrence from the bottom up

The basic idea behind dynamic programming is recursion without repetition.

To develop a dynamic algorithm:

1. Formulate the problem recursively
   a. Describe the problem that you want to solve recursively
   b. Give a clear recursive formula or algorithm
2. Build solutions to your recurrence from the bottom up
   a. Identify the subproblems

The basic idea behind dynamic programming is recursion without repetition.

To develop a dynamic algorithm:

1. Formulate the problem recursively
   a. Describe the problem that you want to solve recursively
   b. Give a clear recursive formula or algorithm
2. Build solutions to your recurrence from the bottom up
   a. Identify the subproblems
   b. Choose a memoization data structure

The basic idea behind dynamic programming is recursion without repetition.

To develop a dynamic algorithm:

1. Formulate the problem recursively
   a. Describe the problem that you want to solve recursively
   b. Give a clear recursive formula or algorithm
2. Build solutions to your recurrence from the bottom up
   a. Identify the subproblems
   b. Choose a memoization data structure
   c. Identify dependencies

The basic idea behind dynamic programming is recursion without repetition.

To develop a dynamic algorithm:

1. Formulate the problem recursively
   a. Describe the problem that you want to solve recursively
   b. Give a clear recursive formula or algorithm
2. Build solutions to your recurrence from the bottom up
   a. Identify the subproblems
   b. Choose a memoization data structure
   c. Identify dependencies
   d. Find a good evaluation order

The basic idea behind dynamic programming is recursion without repetition.

To develop a dynamic algorithm:

1. Formulate the problem recursively
   a. Describe the problem that you want to solve recursively
   b. Give a clear recursive formula or algorithm

2. Build solutions to your recurrence from the bottom up
   a. Identify the subproblems
   b. Choose a memoization data structure
   c. Identify dependencies
   d. Find a good evaluation order
   e. Write down the algorithm

The basic idea behind dynamic programming is recursion without repetition.

To develop a dynamic algorithm:

1. Formulate the problem recursively
   a. Describe the problem that you want to solve recursively
   b. Give a clear recursive formula or algorithm

2. Build solutions to your recurrence from the bottom up
   a. Identify the subproblems
   b. Choose a memoization data structure
   c. Identify dependencies
   d. Find a good evaluation order
   e. Write down the algorithm
   f. Analyze space and running time

Input: A sequence of characters stored in $A[1..n]$.

Output: True if A can be segmented into a sequence of words, False otherwise.

Input: A sequence of characters stored in $A[1..n]$.

Output: True if A can be segmented into a sequence of words, False otherwise.

Given: Function $IsWord(w)$ that returns *True* if sequence of characters w is a word, *False* otherwise.

Input: A sequence of characters stored in $A[1..n]$.

Output: True if A can be segmented into a sequence of words, False otherwise.

Given: Function *IsWord*(w) that returns *True* if sequence of characters w is a word, *False* otherwise.

Example: If sequence A consists of characters

BOTHEARTHANDSATURNSPIN

and *IsWord*(w) is True if w is a word in English, then:

Input: A sequence of characters stored in $A[1..n]$.

Output: True if A can be segmented into a sequence of words, False otherwise.

Given: Function *IsWord*(w) that returns *True* if sequence of characters w is a word, *False* otherwise.

Example: If sequence A consists of characters

BOTHEARTHANDSATURNSPIN

and *IsWord*(w) is True if w is a word in English, then:

- Output *True* because BOTH·EARTH·AND·SATURN·SPIN is a valid segmentation of A

Input: A sequence of characters stored in $A[1..n]$.

Output: True if A can be segmented into a sequence of words, False otherwise.

Given: Function *IsWord(w)* that returns *True* if sequence of characters w is a word, *False* otherwise.

Example: If sequence A consists of characters

BOTHEARTHANDSATURNSPIN

and *IsWord(w)* is True if w is a word in English, then:

- Output *True* because BOTH·EARTH·AND·SATURN·SPIN is a valid segmentation of A
- By the way, BOT·HEART·HANDS·AT·URNS·PIN is another valid segmentation of A.

Dynamic Programming algorithm development, step 1

Formulate the problem recursively

a. Describe the problem that you want to solve recursively

Formulate the problem recursively

a. Describe the problem that you want to solve recursively
b. Give a clear recursive formula or algorithm

Formulate the problem recursively

a. Describe the problem that you want to solve recursively
b. Give a clear recursive formula or algorithm


```
// Is the suffix A[i..n] Splittable?
Splittable(i):
  if i > n
    return True
  for j ← i to n
    if IsWord(i, j) and Splittable(j + 1)
        return True
  return False
```

Dynamic Programming algorithm development, step 1

Formulate the problem recursively

a. Describe the problem that you want to solve recursively
b. Give a clear recursive formula or algorithm


```
// Is the suffix A[i..n] Splittable?
Splittable(i):
  if i > n
    return True
  for j ← i to n
    if IsWord(i, j) and Splittable(j + 1)
        return True
  return False
```

Running time?

Formulate the problem recursively

a. Describe the problem that you want to solve recursively
b. Give a clear recursive formula or algorithm


```
// Is the suffix A[i..n] Splittable?
Splittable(i):
  if i > n
    return True
  for j ← i to n
    if IsWord(i, j) and Splittable(j + 1)
        return True
  return False
```


Running time? $O(2^n)$

Note that there are:

- Only $n + 1$ different ways to call the recursive function `Splittable(i)`, one for each value of i between 1 and $n + 1$

Note that there are:

- Only $n+1$ different ways to call the recursive function `Splittable(i)`, one for each value of i between 1 and $n+1$

- Only $O(n^2)$ different ways to call `IsWord(i, j)`, one for each pair (i, j) such that $1 \leq i \leq j \leq n$

Note that there are:

- Only $n+1$ different ways to call the recursive function `Splittable(i)`, one for each value of i between 1 and $n+1$
- Only $O(n^2)$ different ways to call `IsWord(i, j)`, one for each pair (i, j) such that $1 \leq i \leq j \leq n$

"Why are we spending exponential time computing only a polynomial amount of stuff???"

Note that there are:

- Only $n+1$ different ways to call the recursive function `Splittable(i)`, one for each value of i between 1 and $n+1$
- Only $O(n^2)$ different ways to call `IsWord(i, j)`, one for each pair (i, j) such that $1 \leq i \leq j \leq n$

"Why are we spending exponential time computing only a polynomial amount of stuff???"

For example:

BLUE	**STEM**	**UNIT**	**ROBOT**	HEARTHANDSATURNSPIN

BLUEST	**EMU**	**NITRO**	**BOT**	HEARTHANDSATURNSPIN

Dynamic programming algorithm development step 2

Build solutions to your recurrence from the bottom up

a. Each recursive subproblem is `Splittable(i)` with i between 1 and $n+1$ ...

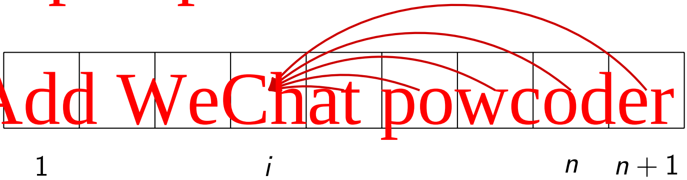
Dynamic programming algorithm development step 2

Build solutions to your recurrence from the bottom up

a. Each recursive subproblem is `Splittable(i)` with i between 1 and $n+1$ ...
b. ... so we can memoize the function `Splittable` into an array `SplitTable[1..n+1]`.

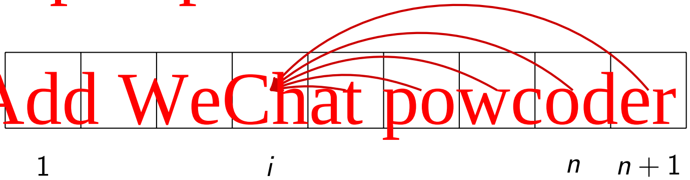
Build solutions to your recurrence from the bottom up

- (a) Each recursive subproblem is `Splittable(i)` with i between 1 and $n+1$ ...
- (b) ... so we can memoize the function `Splittable` into an array `SplitTable[1..n+1]`.
- (c) Each subproblem `Splittable(i)` depends only on results of subproblems `Splittable(j)` where `j > i` ...



Build solutions to your recurrence from the bottom up

a. Each recursive subproblem is `Splittable(i)` with i between 1 and $n+1$ ...
b. ... so we can memoize the function `Splittable` into an array `SplitTable[1..n+1]`.
c. Each subproblem `Splittable(i)` depends only on results of subproblems `Splittable(j)` where `j > i` ...

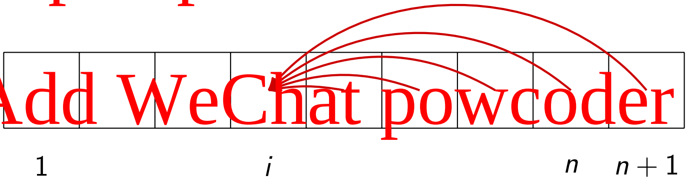


d. ... so we should be filling the array in decreasing index order, starting with `SplitTable[n+1] = True`.

Dynamic programming algorithm development step 2

Build solutions to your recurrence from the bottom up

a. Each recursive subproblem is `Splittable(i)` with i between 1 and $n+1$ ...
b. ... so we can memoize the function `Splittable` into an array `SplitTable[1..n+1]`.
c. Each subproblem `Splittable(i)` depends only on results of subproblems `Splittable(j)` where `j > i` ...



d. ... so we should be filling the array in decreasing index order, starting with `SplitTable[n+1] = True`.
e. The algorithm is on the next slide ...
f. Also on next slide, the running time analysis ...

Text segmentation dynamic programming algorithm

```
// Is A[1..n] splittable?
FastSplittable(A[1..n]):
  SplitTable[n + 1] ← True
  for i ← n down to 1
    SplitTable[i] ← False
    for j ← i to n
      if IsWord(i, j) and SplitTable[j + 1]
        SplitTable[i] ← True
  return SplitTable[1]
```

Text segmentation dynamic programming algorithm


```
// Is A[1..n] splittable?
FastSplittable(A[1..n]):
  SplitTable[n + 1] ← True
  for i ← n down to 1
    SplitTable[i] ← False
    for j ← i to n
      if IsWord(i, j) and SplitTable[j + 1]
        SplitTable[i] ← True
  return SplitTable[1]
```


Running time?

Text segmentation dynamic programming algorithm


```
// Is A[1..n] splittable?
FastSplittable(A[1..n]):
  SplitTable[n + 1] ← True
  for i ← n down to 1
    SplitTable[i] ← False
    for j ← i to n
      if IsWord(i, j) and SplitTable[j + 1]
        SplitTable[i] ← True
  return SplitTable[1]
```


Running time? $O(n^2)$

For any sequence S,

2	6	1	4	2	4	2	9	5	3	5	7	8	3

For any sequence S, a subsequence of S is another sequence obtained from S by deleting zero or more elements, without changing the order of the remaining elements.

2	**6**	**1**	4	2	**4**	2	**9**	5	3	**5**	**7**	8	3

Longest Increasing Subsequence

For any sequence S, a subsequence of S is another sequence obtained from S by deleting zero or more elements, without changing the order of the remaining elements.

2	6	**1**	4	**2**	**4**	2	9	**5**	3	5	**7**	**8**	3

Problem: Given a sequence of integers S find the Longest Increasing Subsequence (LIS) of S.

For any sequence S, a subsequence of S is another sequence obtained from S by deleting zero or more elements, without changing the order of the remaining elements.

2	6	**1**	4	**2**	**4**	2	9	**5**	3	5	**7**	**8**	3

Problem: Given a sequence of integers S find the Longest Increasing Subsequence (LIS) of S.

Input: Integer array $A[1..n]$.

Output: Longest possible sequence of indices $1 \le i_1 < i_2 < \cdots < i_l \le n$ such that $A[i_1] < A[i_2] < \cdots < A[i_l]$.

Example: See above.

Formulate the problem recursively

a. Describe the problem that you want to solve recursively

Dynamic programming algorithm development step 1

Formulate the problem recursively

a. Describe the problem that you want to solve recursively
b. Give a clear recursive formula or algorithm

Dynamic programming algorithm development step 1

Formulate the problem recursively

- ⓐ Describe the problem that you want to solve recursively
- ⓑ Give a clear recursive formula or algorithm


```
// return length of LIS of A[j..n] s.t.
// every element is larger than A[i]
LISbigger(i, j):
  if j > n
    return 0
  else if A[i] ≥ A[j]
    return LISbigger(i, j + 1)
  else
    skip ← LISbigger(i, j + 1)
    take ← LISbigger(j, j + 1) + 1
    return max{skip, take}
```

Formulate the problem recursively

a. Describe the problem that you want to solve recursively
b. Give a clear recursive formula or algorithm


```
// return length of LIS of A[j..n] s.t.
// every element is larger than A[i]
LISbigger(i, j):
  if j > n
    return 0
  else if A[i] ≥ A[j]
    return LISbigger(i, j + 1)
  else
    skip ← LISbigger(i, j + 1)
    take ← LISbigger(j, j + 1) + 1
    return max{skip, take}
```


Running time?

Dynamic programming algorithm development step 1

Formulate the problem recursively

ⓐ Describe the problem that you want to solve recursively

ⓑ Give a clear recursive formula or algorithm


```
// return length of LIS of A[j..n] s.t.
// every element is larger than A[i]
LISbigger(i, j):
  if j > n
    return 0
  else if A[i] ≥ A[j]
    return LISbigger(i, j + 1)
  else
    skip ← LISbigger(i, j + 1)
    take ← LISbigger(j, j + 1) + 1
    return max{skip, take}
```


Running time? $O(2^n)$

Dynamic programming algorithm development step 2

To develop a dynamic algorithm:

1. Formulate the problem recursively
   a. Describe the problem that you want to solve recursively
   b. Give a clear recursive formula or algorithm

2. Build solutions to your recurrence from the bottom up
   a. Identify the subproblems
   b. Choose a memoization data structure
   c. Identify dependencies
   d. Find a good evaluation order
   e. Write down the algorithm
   f. Analyze space and running time

Dynamic programming algorithm development step 2

Build solutions to your recurrence from the bottom up

1. The subproblems are `LISbigger(i, j + 1)` and `LISbigger(j, j + 1)` with indices i and j with values between 0 and n

Build solutions to your recurrence from the bottom up

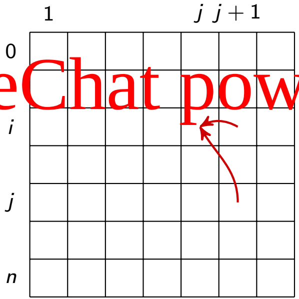
a. The subproblems are `LISbigger(i, j + 1)` and `LISbigger(j, j + 1)` with indices i and j with values between 0 and n

b. We can memoize the results of these subproblems into a two-dimensional array `LISbigger[0..n, 1..n+1]`.

Dynamic programming algorithm development step 2

Build solutions to your recurrence from the bottom up

a. The subproblems are `LISbigger(i, j + 1)` and `LISbigger(j, j + 1)` with indices i and j with values between 0 and n
b. We can memoize the results of these subproblems into a two-dimensional array `LISbigger[0..n, 1..n+1]`.
c. Each entry `LISbigger[i, j]` is filled in using entries `LISbigger[i, j+1]` and `LISbigger[j, j+1]`.



Dynamic programming algorithm development step 2

Build solutions to your recurrence from the bottom up

a. The subproblems are `LISbigger(i, j + 1)` and `LISbigger(j, j + 1)` with indices i and j with values between 0 and n

b. We can memoize the results of these subproblems into a two-dimensional array `LISbigger[0..n, 1..n+1]`.
c. Each entry `LISbigger[i, j]` is filled in using entries `LISbigger[i, j+1]` and `LISbigger[j, j+1]`.

d. Fill in the entries of the two-dimensional table column-by-column, right-to-left.

Dynamic programming algorithm development step 2

Build solutions to your recurrence from the bottom up

a. The subproblems are `LISbigger(i, j + 1)` and `LISbigger(j, j + 1)` with indices i and j with values between 0 and n
b. We can memoize the results of these subproblems into a two-dimensional array `LISbigger[0..n, 1..n+1]`.
c. Each entry `LISbigger[i, j]` is filled in using entries `LISbigger[i, j+1]` and `LISbigger[j, j+1]`.
d. Fill in the entries of the two-dimensional table column-by-column, right-to-left.
e. Iterative algorithm on the slide...
f. Running time analysis on the next slide...

```
// return length of LIS of A[1..n]
FastLIS(A[1 .. n]):
  A[0] ← −∞
  for i ← 0 to n
    LISbigger[i, n + 1] ← 0
  for j ← n down to 1
    for i ← 0 to j − 1
      keep ← 1 + LISbigger[j, j + 1]
      skip ← LISbigger[i, j + 1]
      if A[i] ≥ A[j]
        LISbigger[i, j] ← skip
      else
        LISbigger[i, j] ← max{keep, skip}
  return LISbigger[0, 1]
```

FastLIS

```
// return length of LIS of A[1..n]
FastLIS(A[1 .. n]):
  A[0] ← −∞
  for i ← 0 to n
    LISbigger[i, n + 1] ← 0
  for j ← n down to 1
    for i ← 0 to j − 1
      keep ← 1 + LISbigger[j, j + 1]
      skip ← LISbigger[i, j + 1]
      if A[i] ≥ A[j]
        LISbigger[i, j] ← skip
      else
        LISbigger[i, j] ← max{keep, skip}
  return LISbigger[0, 1]
```


Running time?

```
// return length of LIS of A[1..n]
FastLIS(A[1 .. n]):
  A[0] ← −∞
  for i ← 0 to n
    LISbigger[i, n + 1] ← 0
  for j← n down to 1
    for i ← 0 to j − 1
      keep ← 1 + LISbigger[j, j + 1]
      skip ← LISbigger[i, j + 1]
      if A[i] ≥ A[j]
        LISbigger[i, j] ← skip
      else
        LISbigger[i, j] ← max{keep, skip}
  return LISbigger[0, 1]
```


Running time? $O(n^2)$