Lecture 7: Dynamic Programming

Tim LaRock

larock.t@northeastern.edu

bit.ly/cs3000syllabus

Business

Homework 2 is out, due Tuesday May 19 11:59PM Boston time on Canvas

We are working on grading homework 1, will share solutions once the grades are complete

It is totally fine to look ahead in the slides, but please give others a minute to try to answer questions I ask before writing what you saw later

Today

Dynamic Programming

- Fibonacci Numbers
- Text Segmentation Revisited

Fibonacci Numbers

$$f_n \begin{cases} 0 & if\ n = 0 \\ 1 & if\ n = 1 \\ f_{n-1} + f_{n-2} & otherwise \end{cases}$$

Fibonacci Numbers: Recursion

$$f_n \begin{cases} 0 & if\ n = 0 \\ 1 & if\ n = 1 \\ f_{n-1} + f_{n-2} & otherwise \end{cases}$$

```
Fib(n):
  If n = 0:
    return 0
  ElseIf n = 1:
    return 1
  Else:
    return Fib(n − 1) + Fib(n − 2)
```

Fibonacci Numbers: Recurrence Relation

$$f_n \begin{cases} 0 & if\ n = 0 \\ 1 & if\ n = 1 \\ f_{n-1} + f_{n-2} & otherwise \end{cases}$$

What does the recurrence relation $T(n)$ look like?

$$T(n) = T(n-1) + T(n-2) + O(1)$$

```
Fib(n):
  If n = 0:
    return 0
  ElseIf n = 1:
    return 1
  Else:
    return Fib(n − 1) + Fib(n − 2)
```

Fibonacci Numbers: Recurrence Relation

$$f_n \begin{cases} 0 & if\ n = 0 \\ 1 & if\ n = 1 \\ f_{n-1} + f_{n-2} & otherwise \end{cases}$$

What does the recurrence relation $T(n)$ look like?

$$T(0) = 1, T(1) = 1$$

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

```
Fib(n):
  If n = 0:
    return 0
  ElseIf n = 1:
    return 1
  Else:
    return Fib(n − 1) + Fib(n − 2)
```

Fibonacci Numbers: Recurrence Relation

$$f_n \begin{cases} 0 & if\ n = 0 \\ 1 & if\ n = 1 \\ f_{n-1} + f_{n-2} & otherwise \end{cases}$$

```
Fib(n):
  If n = 0:
    return 0
  ElseIf n = 1:
    return 1
  Else:
    return Fib(n − 1) + Fib(n − 2)
```

What does the recurrence relation $T(n)$ look like?

$$T(0) = 1, T(1) = 1$$

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

First, if we squint and assume $n \to \infty$ we might see

$$T(n) = T(n-1) + T(n-1) + 1$$

Fibonacci Numbers: Recurrence Relation

$$f_n \begin{cases} 0 & if\ n = 0 \\ 1 & if\ n = 1 \\ f_{n-1} + f_{n-2} & otherwise \end{cases}$$

```
Fib(n):
  If n = 0:
    return 0
  ElseIf n = 1:
    return 1
  Else:
    return Fib(n − 1) + Fib(n − 2)
```

What does the recurrence relation $T(n)$ look like?

$$T(0) = 1, T(1) = 1$$

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

First, if we squint and assume $n \to \infty$ we might see

$$T(n) = T(n-1) + T(n-1) + 1$$

$$T(n) = 2T(n-1) + 1 \leq 2 \cdot 2^n$$

$$\leq O(2^{n+1})$$

Fibonacci Numbers: Recurrence Relation

$$f_n \begin{cases} 0 & if\ n = 0 \\ 1 & if\ n = 1 \\ f_{n-1} + f_{n-2} & otherwise \end{cases}$$

What does the recurrence relation $T(n)$ look like?

$$T(0) = 1, T(1) = 1$$

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

```
Fib(n):
  If n = 0:
    return 0
  ElseIf n = 1:
    return 1
  Else:
    return Fib(n − 1) + Fib(n − 2)
```

Fibonacci Numbers: Recurrence Relation

$$f_n \begin{cases} 0 & if\ n = 0 \\ 1 & if\ n = 1 \\ f_{n-1} + f_{n-2} & otherwise \end{cases}$$

What does the recurrence relation $T(n)$ look like?

$$T(0) = 1, T(1) = 1$$

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

$$T(2) = T(1) + T(0) + 1 = 3$$

```
Fib(n):
  If n = 0:
    return 0
  ElseIf n = 1:
    return 1
  Else:
    return Fib(n − 1) + Fib(n − 2)
```

Fibonacci Numbers: Recurrence Relation

$$f_n \begin{cases} 0 & if\ n = 0 \\ 1 & if\ n = 1 \\ f_{n-1} + f_{n-2} & otherwise \end{cases}$$

```
Fib(n):
  If n = 0:
    return 0
  ElseIf n = 1:
    return 1
  Else:
    return Fib(n − 1) + Fib(n − 2)
```

What does the recurrence relation $T(n)$ look like?

$$T(0) = 1, T(1) = 1$$

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

$T(2) = T(1) + T(0) + 1 = 3$

$2+1=3$

$Fib(3) = Fib(2) + Fib(1) = 2$

$1 + 1 = 2$

Fibonacci Numbers: Recurrence Relation

$$f_n \begin{cases} 0 & if\ n = 0 \\ 1 & if\ n = 1 \\ f_{n-1} + f_{n-2} & otherwise \end{cases}$$

```
Fib(n):
  If n = 0:
    return 0
  ElseIf n = 1:
    return 1
  Else:
    return Fib(n − 1) + Fib(n − 2)
```

What does the recurrence relation $T(n)$ look like?

$$T(0) = 1, T(1) = 1$$

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

$T(2) = T(1) + T(0) + 1 = 3$

$T(3) = T(2) + T(1) + 1 = 5$

3 + 1 + 1

$Fib(3) = Fib(2) + Fib(1) = 2$

$Fib(4) = Fib(3) + Fib(2) = 3$

Fibonacci Numbers: Recurrence Relation

$$f_n \begin{cases} 0 & if\ n = 0 \\ 1 & if\ n = 1 \\ f_{n-1} + f_{n-2} & otherwise \end{cases}$$

```
Fib(n):
  If n = 0:
    return 0
  ElseIf n = 1:
    return 1
  Else:
    return Fib(n − 1) + Fib(n − 2)
```

What does the recurrence relation $T(n)$ look like?

$$T(0) = 1, T(1) = 1$$

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

$T(2) = T(1) + T(0) + 1 = 3$	$Fib(3) = Fib(2) + Fib(1) = 2$
$T(3) = T(2) + T(1) + 1 = 5$	$Fib(4) = Fib(3) + Fib(2) = 3$
$T(4) = T(3) + T(2) + 1 = 9$	$Fib(5) = Fib(4) + Fib(3) = 5$

Fibonacci Numbers: Recurrence Relation

$$f_n \begin{cases} 0 & if\ n = 0 \\ 1 & if\ n = 1 \\ f_{n-1} + f_{n-2} & otherwise \end{cases}$$

```
Fib(n):
  If n = 0:
    return 0
  ElseIf n = 1:
    return 1
  Else:
    return Fib(n − 1) + Fib(n − 2)
```

What does the recurrence relation $T(n)$ look like?

$$T(0) = 1, T(1) = 1$$

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

$T(2) = T(1) + T(0) + 1 = 3$	$Fib(3) = Fib(2) + Fib(1) = 2$
$T(3) = T(2) + T(1) + 1 = 5$	$Fib(4) = Fib(3) + Fib(2) = 3$
$T(4) = T(3) + T(2) + 1 = 9$	$Fib(5) = Fib(4) + Fib(3) = 5$

$$T(2) = 2Fib(2+1) - 1 = 3$$

$$T(3) = 2Fib(3+1) - 1 = 5$$

$$T(4) = 2Fib(4+1) - 1 = 9$$

Fibonacci Numbers: Recurrence Relation

$$f_n \begin{cases} 0 & if\ n = 0 \\ 1 & if\ n = 1 \\ f_{n-1} + f_{n-2} & otherwise \end{cases}$$

```
Fib(n):
  If n = 0:
    return 0
  ElseIf n = 1:
    return 1
  Else:
    return Fib(n − 1) + Fib(n − 2)
```

What does the recurrence relation $T(n)$ look like?

$$T(0) = 1, T(1) = 1$$

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

$T(2) = T(1) + T(0) + 1 = 3$	$Fib(3) = Fib(2) + Fib(1) = 2$
$T(3) = T(2) + T(1) + 1 = 5$	$Fib(4) = Fib(3) + Fib(2) = 3$
$T(4) = T(3) + T(2) + 1 = 9$	$Fib(5) = Fib(4) + Fib(3) = 5$

$$T(2) = 2Fib(2+1) - 1 = 3$$

$$T(3) = 2Fib(3+1) - 1 = 5$$

$$T(4) = 2Fib(4+1) - 1 = 9$$

$$T(n) = 2f_{n+1} - 1$$

Fibonacci Numbers: Recurrence Relation

$$f_n \begin{cases} 0 & if\ n = 0 \\ 1 & if\ n = 1 \\ f_{n-1} + f_{n-2} & otherwise \end{cases}$$

```
Fib(n):
  If n = 0:
    return 0
  ElseIf n = 1:
    return 1
  Else:
    return Fib(n − 1) + Fib(n − 2)
```

What does the recurrence relation $T(n)$ look like?

$$T(0) = 1, T(1) = 1$$

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

$$T(2) = T(1) + T(0) + 1 = 3$$

$$T(3) = T(2) + T(1) + 1 = 5$$

$$T(4) = T(3) + T(2) + 1 = 9$$

$$Fib(3) = Fib(2) + Fib(1) = 2$$

$$Fib(4) = Fib(3) + Fib(2) = 3$$

$$Fib(5) = Fib(4) + Fib(3) = 5$$

$$T(2) = 2Fib(2+1) - 1 = 3$$

$$T(3) = 2Fib(3+1) - 1 = 5$$

$$T(4) = 2Fib(4+1) - 1 = 9$$

$$T(n) = 2f_{n+1} - 1 \rightarrow 2T(n+1) \leq O(2^{n+1})$$

Fibonacci Numbers: Recurrence Relation

$$f_n \begin{cases} 0 & if\ n = 0 \\ 1 & if\ n = 1 \\ f_{n-1} + f_{n-2} & otherwise \end{cases}$$

```
Fib(n):
  If n = 0:
    return 0
  ElseIf n = 1:
    return 1
  Else:
    return Fib(n − 1) + Fib(n − 2)
```

What does the recurrence relation $T(n)$ look like?

$$T(0) = 1, T(1) = 1$$

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

$T(2) = T(1) + T(0) + 1 = 3$	$Fib(3) = Fib(2) + Fib(1) = 2$
$T(3) = T(2) + T(1) + 1 = 5$	$Fib(4) = Fib(3) + Fib(2) = 3$
$T(4) = T(3) + T(2) + 1 = 9$	$Fib(5) = Fib(4) + Fib(3) = 5$

$$T(2) = 2Fib(2+1) - 1 = 3$$

$$T(3) = 2Fib(3+1) - 1 = 5$$

$$T(4) = 2Fib(4+1) - 1 = 9$$

$$T(n) = 2f_{n+1} - 1 \rightarrow 2T(n+1) \leq O(2^{n+1})$$

Exponential in n is *very* slow for such a simple function!

Memoization

```
Fib(n):
  If n = 0:
    return 0
  ElseIf n = 1:
    return 1
  Else:
    return Fib(n − 1) + Fib(n − 2)
```

Memo(r)ization

```
Fib(n):
  If n = 0:
    return 0
  ElseIf n = 1:
    return 1
  Else:
    return Fib(n − 1) + Fib(n − 2)
```

$Fib(k) \quad k < n-2$

- $Fib(n)$ is very slow because we are recomputing the same values over and over again!

Memo(r)ization

```
Fib(n):
  If n = 0:
    return 0
  ElseIf n = 1:
    return 1
  Else:
    return Fib(n − 1) + Fib(n − 2)
```

- $Fib(n)$ is very slow because we are recomputing the same values over and over again!

- What if instead we save each value we compute so that we can access it in constant time?

Memo(r)ization

```
Fib(n):
  If n = 0:
    return 0
  ElseIf n = 1:
    return 1
  Else:
    return Fib(n − 1) + Fib(n − 2)
```

```
MemFib(n):
  If n = 0:
    return 0
  ElseIf n = 1:
    return 1
  Else:
    If F[n] is undefined:
      F[n] = MemFib(n − 1) + MemFib(n − 2)
    return F[n]
```

- $Fib(n)$ is very slow because we are recomputing the same values over and over again!
- What if instead we save each value we compute so that we can access it in constant time?
- Keep a global table $F[i]$ that stores results and use stored results where possible

Memo(r)ization

```
Fib(n):
  If n = 0:
    return 0
  ElseIf n = 1:
    return 1
  Else:
    return Fib(n − 1) + Fib(n − 2)
```

```
MemFib(n):
  If n = 0:
    return 0
  ElseIf n = 1:
    return 1
  Else:
    If F[n] is undefined:
      F[n] = MemFib(n − 1) + MemFib(n − 2)
    return F[n]
```

- $Fib(n)$ is very slow because we are recomputing the same values over and over again!
- What if instead we save each value we compute so that we can access it in constant time?
- Keep a global table $F[i]$ that stores results and use stored results where possible
- How is the table filled? And what implication does this have for the runtime?

Memo(r)ization

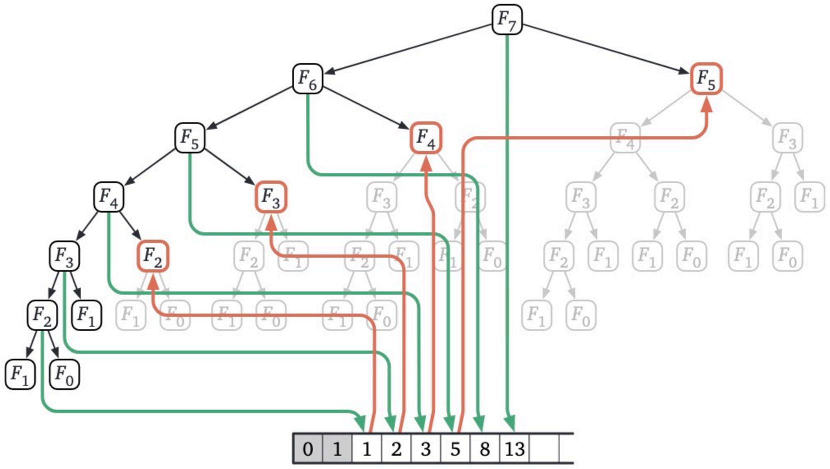


Figure 3.2. The recursion tree for F_7 trimmed by memoization. Downward green arrows indicate writing into the memoization array; upward red arrows indicate reading from the memoization array.

$MemFib(n)$:
  If $n = 0$:
    return 0
  ElseIf $n = 1$:
    return 1
  Else:
    If $F[n]$ is undefined:
      $\mathrm{F[n]} = MemFib(n-1) + MemFib(n-2)$
    return $F[n]$

Memo(r)ization

$O(2^n)$

How many additions?

$O(n)$

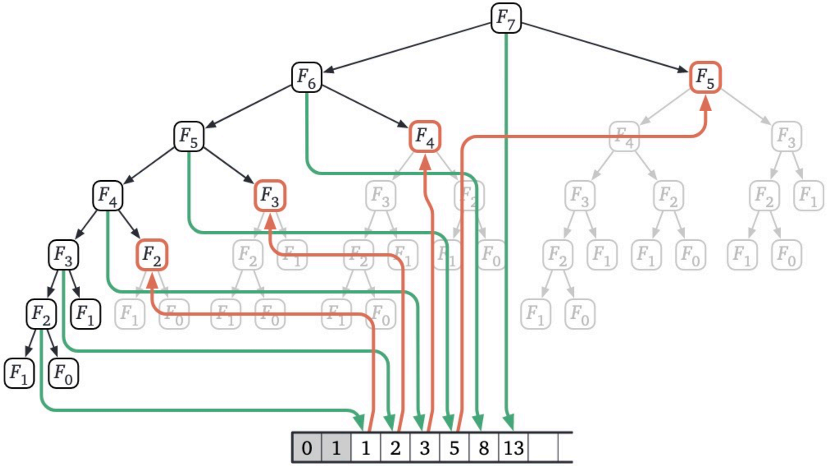


Figure 3.2. The recursion tree for F_7 trimmed by memoization. Downward green arrows indicate writing into the memoization array; upward red arrows indicate reading from the memoization array.

```
MemFib(n):
  If n = 0:
    return 0
  ElseIf n = 1:
    return 1
  Else:
    If F[n] is undefined:
      F[n] = MemFib(n − 1) + MemFib(n − 2)
    return F[n]
```

There has to be a better way!

Enter: Dynamic programming

Dynamic Programming

```
MemFib(n):
  If n = 0:
    return 0
  ElseIf n = 1:
    return 1
  Else:
    If F[n] is undefined:
      F[n] = MemFib(n − 1) + MemFib(n − 2)
    return F[n]
```

- The execution order and runtime of $MemFib(n)$ implies a simpler way to compute Fibonacci numbers

Dynamic Programming

```
MemFib(n):
  If n = 0:
    return 0
  ElseIf n = 1:
    return 1
  Else:
    If F[n] is undefined:
      F[n] = MemFib(n − 1) + MemFib(n − 2)
    return F[n]
```

```
IterFib(n):
  F[0] ← 0
  F[1] ← 1
  for i from 2..n
      F[i] ← F[i − 1] + F[i − 2]
  return F[n]
```

- The execution order and runtime of $MemFib(n)$ implies a simpler way to compute Fibonacci numbers
- What if we just like....filled F explicitly?

Dynamic Programming

```
MemFib(n):
  If n = 0:
    return 0
  ElseIf n = 1:
    return 1
  Else:
    If F[n] is undefined:
      F[n] = MemFib(n − 1) + MemFib(n − 2)
    return F[n]
```

```
IterFib(n):
  F[0] ← 0
  F[1] ← 1
  for i from 2..n
      F[i] ← F[i − 1] + F[i − 2]
  return F[n]
```

- The execution order and runtime of $MemFib(n)$ implies a simpler way to compute Fibonacci numbers
- What if we just like….filled F explicitly?
- Now the execution is clearly $O(n)$!
- Note: We could save memory here. How?

Dynamic Programming

- Formalized by Richard Bellman at RAND in the ‘50s
  - Bellman apparently named it “dynamic programming” to obscure his research from his bosses.
  - Programming does not refer to computers, but scheduling: for example designing the “program” of a performance or event, or filling a TV schedule

- General pattern: Recursion without repetition
  - Store solutions of intermediate problems to be reused later!
  - Finding a correct recurrence that can be memoized is vital
    - If your recurrence is wrong or can’t be memoized, you will go in circles!

Dynamic Programming Process

There are 3 main steps to developing dynamic programming solutions:

1. Find the right recurrence
    - Formalize the problem carefully
    - Find a recursive solution (could be pseudocode or just a relation)

Dynamic Programming Process

There are 3 main steps to developing dynamic programming solutions:

1. Find the right recurrence
   - Formalize the problem carefully
   - Find a recursive solution (could be pseudocode or just a relation)
2. Build solutions to the recurrence from the bottom up
   - What are the subproblems that need solving?
   - What data structure can I use to access them correctly and quickly?
   - Which subproblems depend on each other?
   - What order should the subproblems be executed in?

Dynamic Programming Process

There are 3 main steps to developing dynamic programming solutions:

1. Find the right recurrence
    - Formalize the problem carefully
    - Find a recursive solution (could be pseudocode or just a relation)
2. Build solutions to the recurrence from the bottom up
    - What are the subproblems that need solving?
    - What data structure can I use to access them correctly and quickly?
    - Which subproblems depend on each other?
    - What order should the subproblems be executed in?
3. Prove it!

Text Segmentation Revisited

$T(n) = 2T(n-1) + c \leq O(2^n)$

```
Splittable(A[1..n], i):
  If i > n:
    return True
  Else:
    j ← i
    for j to n:
      If IsWord(i,j):
        If Splittable(A[1..n], j + 1):
          return True

  return False
```

Problem: Given an array $A[1..n]$ representing a sequence of n characters without spaces, determine whether the array can be subdivided into a sequence of *words.*

True or False

Assume we are given a function $IsWord(i, j)$. This function assumes A is a global variable and returns True if the subarray $A[i..j]$ is a word in the language of the sequence.

- This allows us to avoid passing subarrays as arguments to functions.

Text Segmentation Revisited

Where are we wasting computation?

$T(n) = 2T(n-1) + c \le O(2^n)$

```
Splittable(A[1..n], i):
  If i > n:
    return True
  Else:
    j ← i
    for j to n:
      If IsWord(i, j):
        If Splittable(A[1..n], j + 1):
          return True

  return False
```

Text Segmentation Revisited

$T(n) = 2T(n-1) + c \ \leq O(2^n)$

```
Splittable(A[1..n], i):
  If i > n:
    return True
  Else:
    j ← i
    for j to n:
      If IsWord(i, j):
        If Splittable(A[1..n], j + 1):
          return True

  return False
```

Where are we wasting computation?

For a fixed $A[1..n]$, how many ways can we call $Splittable(A, i)$?

$n-1$ $O(n)$

Text Segmentation Revisited

$T(n) = 2T(n-1) + c \ \leq O(2^n)$

```
Splittable(A[1..n], i):
  If i > n:
    return True
  Else:
    j ← i
    for j to n:
      If IsWord(i,j):
        If Splittable(A[1..n], j+1):
          return True

  return False
```

Where are we wasting computation?

For a fixed $A[1..n]$, how many ways can we call $Splittable(A, i)$?

$O(n)$

Text Segmentation Revisited

$T(n) = 2T(n-1) + c \le O(2^n)$

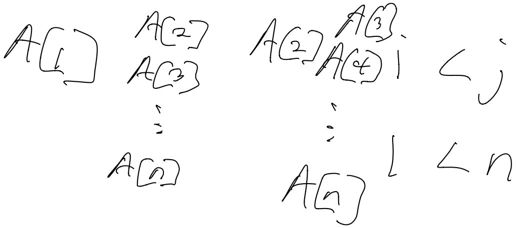
```
Splittable(A[1..n], i):
  If i > n:
    return True
  Else:
    j ← i
    for j to n:
      If IsWord(i, j):
        If Splittable(A[1..n], j + 1):
          return True

  return False
```

Where are we wasting computation?

For a fixed $A[1..n]$, how many ways can we call $Splittable(A, i)$?

$O(n)$

For all indices $1 \le i \le j \le n$, how many times can we call $IsWord(i, j)$?



Text Segmentation Revisited

$T(n) = 2T(n-1) + c \le O(2^n)$

```
Splittable(A[1..n], i):
  If i > n:
    return True
  Else:
    j ← i
    for j to n:
      If IsWord(i, j):
        If Splittable(A[1..n], j + 1):
          return True

  return False
```

Where are we wasting computation?

For a fixed $A[1..n]$, how many ways can we call $Splittable(A, i)$?

$O(n)$

For all indices $1 \le \mathrm{i} \le j \le n$, how many times can we call $IsWord(i, j)$?

$O(n^2)$

We are spending exponential time computing polynomial amounts of stuff!

Dynamic Programming Approach

SplitTable[1, ..., n+1]

```
FastSplittable(A[1..n]):
   SplitTable[n + 1] ← True

   for i from n to 1:
      SplitTable[i] ← False
      for j from i to n:
         If IsWord(i,j) AND SplitTable[j + 1]:
            SplitTable[i] ← True

   return SplitTable[1]
```

Dynamic Programming Approach

$n = 18$

t	h	e	b	r	o	w	n	f	o	x	i	s	q	u	i	c	k

$i = 18$ ↑

$j = 18$ ↑

```
FastSplittable(A[1..n]):
   SplitTable[n + 1] ← True

   for i from n to 1:
      SplitTable[i] ← False
      for j from i to n:
         If IsWord(i, j) AND SplitTable[j + 1]:
            SplitTable[i] ← True

   return SplitTable[1]
```

SplitTable

t	h	e	b	r	o	w	n	f	o	x	i	s	q	u	i	c	k	
F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	T

$j + 1 = 19$ ↑

Dynamic Programming Approach

```
FastSplittable(A[1..n]):
   SplitTable[n + 1] ← True

   for i from n to 1:
      SplitTable[i] ← False
      for j from i to n:
         If IsWord(i, j) AND SplitTable[j + 1]:
            SplitTable[i] ← True

   return SplitTable[1]
```

$n = 18$

t	h	e	b	r	o	w	n	f	o	x	i	s	q	u	i	c	k

$i = 17$

$j = 17$

SplitTable

t	h	e	b	r	o	w	n	f	o	x	i	s	q	u	i	c	k	
F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	T

$j + 1 = 18$

Dynamic Programming Approach

$n = 18$

t	h	e	b	r	o	w	n	f	o	x	i	s	q	u	i	c	k

$i = 17$

$j = 18$

```
FastSplittable(A[1..n]):
   SplitTable[n + 1] ← True

   for i from n to 1:
      SplitTable[i] ← False
      for j from i to n:
         If IsWord(i,j) AND SplitTable[j + 1]:
            SplitTable[i] ← True

   return SplitTable[1]
```

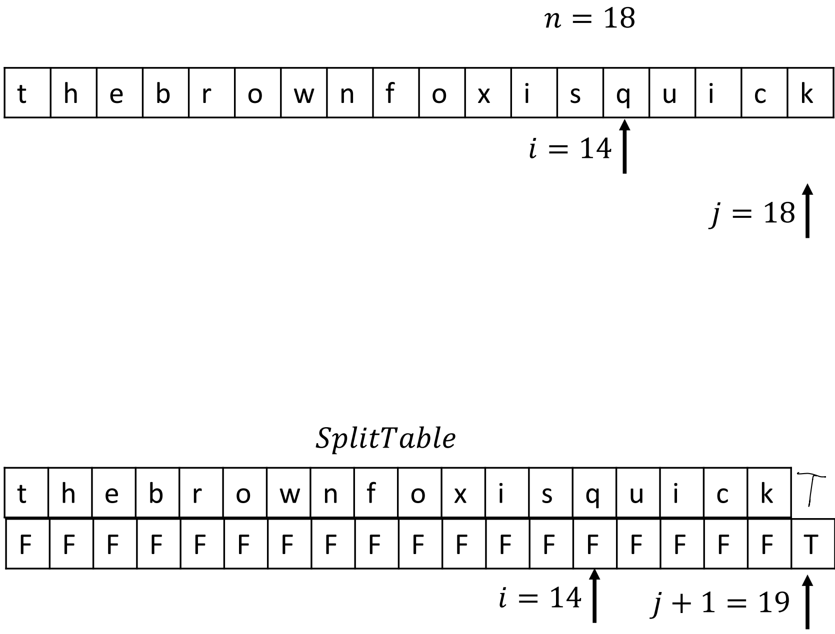
SplitTable

t	h	e	b	r	o	w	n	f	o	x	i	s	q	u	i	c	k	
F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	T

$j + 1 = 19$

Dynamic Programming Approach

```
FastSplittable(A[1..n]):
   SplitTable[n + 1] ← True

   for i from n to 1:
      SplitTable[i] ← False
      for j from i to n:
         If IsWord(i, j) AND SplitTable[j + 1]:
            SplitTable[i] ← True

   return SplitTable[1]
```



Dynamic Programming Approach

$n = 18$

t	h	e	b	r	o	w	n	f	o	x	i	s	q	u	i	c	k

$i = 14$ ↑

$j = 18$ ↑

$IsWord(14{,}18)$ is $True$!

```
FastSplittable(A[1..n]):
   SplitTable[n + 1] ← True

   for i from n to 1:
      SplitTable[i] ← False
      for j from i to n:
         If IsWord(i,j) AND SplitTable[j + 1]:
            SplitTable[i] ← True

   return SplitTable[1]
```

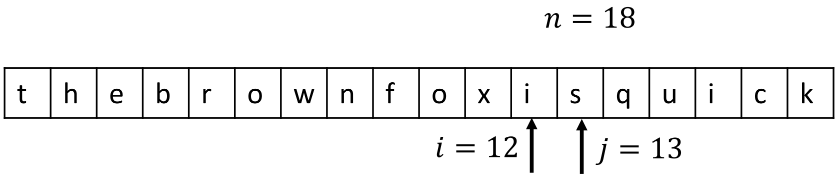
SplitTable

t	h	e	b	r	o	w	n	f	o	x	i	s	q	u	i	c	k	
F	F	F	F	F	F	F	F	F	F	F	F	F	T	F	F	F	F	T

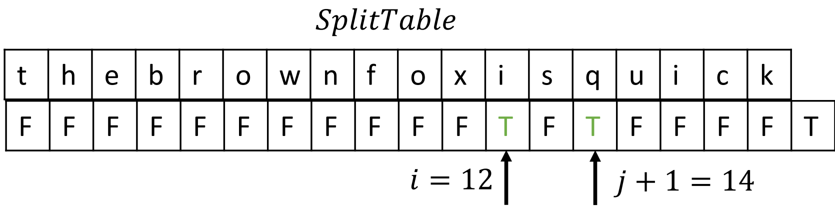
$i = 14$ ↑ $j + 1 = 19$ ↑

Dynamic Programming Approach

```
FastSplittable(A[1..n]):
   SplitTable[n + 1] ← True

   for i from n to 1:
      SplitTable[i] ← False
      for j from i to n:
         If IsWord(i, j) AND SplitTable[j + 1]:
            SplitTable[i] ← True

   return SplitTable[1]
```



$IsWord(12,13)$ is $True$!



Dynamic Programming Approach

```
FastSplittable(A[1..n]):
   SplitTable[n + 1] ← True

   for i from n to 1:
      SplitTable[i] ← False
      for j from i to n:
         If IsWord(i,j) AND SplitTable[j + 1]:
            SplitTable[i] ← True

   return SplitTable[1]
```

$n = 18$

t	h	e	b	r	o	w	n	f	o	x	i	s	q	u	i	c	k

$i = 10$ ↑ ↑ $j = 11$

$IsWord(10,11)$ is $True$!

$SplitTable$

t	h	e	b	r	o	w	n	f	o	x	i	s	q	u	i	c	k	
F	F	F	F	F	F	F	F	F	T	F	T	F	T	F	F	F	F	T

$i = 10$ ↑ ↑ $j + 1 = 12$

Dynamic Programming Approach

```
FastSplittable(A[1..n]):
    SplitTable[n + 1] ← True

    for i from n to 1:
        SplitTable[i] ← False
        for j from i to n:
            If IsWord(i,j) AND SplitTable[j + 1]:
                SplitTable[i] ← True

    return SplitTable[1]
```

$n = 18$

t	h	e	b	r	o	w	n	f	o	x	i	s	q	u	i	c	k

$i = 9$ ↑ ↑ $j = 11$

$IsWord(9,11)$ is $True$!

$SplitTable$

t	h	e	b	r	o	w	n	f	o	x	i	s	q	u	i	c	k	
F	F	F	F	F	F	F	F	T	T	F	T	F	T	F	F	F	F	T

$i = 9$ ↑ ↑ $j + 1 = 12$

Dynamic Programming Approach

$n = 18$

t	h	e	b	r	o	w	n	f	o	x	i	s	q	u	i	c	k

$i = 1$ ↑ ↑ $j = 3$

```
FastSplittable(A[1..n]):
   SplitTable[n + 1] ← True

   for i from n to 1:
      SplitTable[i] ← False
      for j from i to n:
         If IsWord(i,j) AND SplitTable[j + 1]:
            SplitTable[i] ← True

   return SplitTable[1]
```

$IsWord(1,3)$ is $True$!

$SplitTable$

t	h	e	b	r	o	w	n	f	o	x	i	s	q	u	i	c	k	
F	F	F	T	F	T	F	F	T	T	F	T	F	T	F	F	F	F	T

$i = 1$ ↑ ↑ $j + 1 = 4$

nfoxisquick

Dynamic Programming Approach

```
FastSplittable(A[1..n]):
   SplitTable[n + 1] ← True

   for i from n to 1:
      SplitTable[i] ← False
      for j from i to n:
         If IsWord(i,j) AND SplitTable[j + 1]:
            SplitTable[i] ← True

   return SplitTable[1]
```

$n = 18$

t	h	e	b	r	o	w	n	f	o	x	i	s	q	u	i	c	k

$i = 1$ ↑ ↑ $j = 3$

~~The brown fox ...~~

$IsWord(1,3)$ is $True$!

The

$SplitTable$

t	h	e	b	r	o	w	n	f	o	x	i	s	q	u	i	c	k	
T	F	F	T	F	T	F	F	T	T	F	T	F	T	F	F	F	F	T

$i = 1$ ↑ ↑ $j + 1 = 4$

If T propagates all the way back to $i = 1$, we have a segmentation!

FastSplittable Analysis

$n \cdot n = n^2$

t	h	e	b	r	o	w	n	f	o	x	i	s	q	u	i	c	k	
T	F	F	T	F	T	F	F	T	T	F	T	F	T	F	F	F	F	T

```
FastSplittable(A[1..n]):
   SplitTable[n + 1] ← True

   for i from n to 1:
      SplitTable[i] ← False
      for j from i to n:
         If IsWord(i,j) AND SplitTable[j + 1]:
            SplitTable[i] ← True

   return SplitTable[1]
```

If T propagates all the way back to $i = 1$, we have a segmentation!

Previously we had the recurrence:

$$T(n) = 2T(n-1) + c \ \leq O(2^n)$$

I argue we can just read off the running time of $FastSplittable$ from the pseudocode

$O(n^2)$

Wrap up

Work on homework 2! Due Tuesday night at midnight.

Next time:

- Subset Sum revisited
- Edit Distance
- Knapsack problem

No new reading assignment (Chapter 3 Erickson)