Lecture 8: More Dynamic Programming

Tim LaRock

larock.t@northeastern.edu

bit.ly/cs3000syllabus

Business

Keep working on homework 2!

- Ask questions early if you are stuck!

Take home midterm 1 will be next Wednesday through Friday (more at the end)

Today

Brief correction on yesterday's lecture

Dynamic Programming

- Subset Sum
- Edit Distance

Correct the record: 2 mistakes

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

```
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

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

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

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

Correct the record: 2 mistakes

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

```
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

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

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

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

This is wrong!
I was not careful
and made an
error!

Correct the record: 2 mistakes

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

```
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

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

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

Correct the record: 2 mistakes

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

```
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?

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

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

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

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

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

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

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

Correct the record: 2 mistakes

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

```
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?

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

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

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

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

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

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

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

This is wrong!

Correct the record: 2 mistakes

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

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

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

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

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

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

The important point is that just counting to f_n would be twice as fast!

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

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

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

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

Today: Dynamic Programming Subset Sum

Subset Sum

We are given a set of n positive integers $X = \{x_{1,} x_{2,} \ldots, x_n\}$ and a target integer value T. We want to find a subset $Y \subseteq X$ such that the sum of the elements $\sum_{x_i \in Y} x_i = T$.

Our problem: For a given T and X, does such a Y exist?

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

```
SubsetSum(X[1..n], i, T):
  If T = 0:
    return True
  ElseIf T < 0 or i = 0:
    return False
  Else:
    with ← SubsetSum(X, i-1, T – X[i])
    wout ← SubsetSum(X, i-1, T)
    return with OR wout
```

Subset Sum

We are given a set of n positive integers $X = \{x_{1,} x_{2,} \ldots, x_n\}$ and a target integer value T. We want to find a subset $Y \subseteq X$ such that the sum of the elements $\sum_{x_i \in Y} x_i = T$.

Our problem: For a given T and X, does such a Y exist?

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

```
SubsetSum(X[1..n], i, T):
  If T = 0:
    return True
  ElseIf T < 0 or i = 0:
    return False
  Else:
    with ← SubsetSum(X, i-1, T – X[i])
    wout ← SubsetSum(X, i-1, T)
    return with OR wout
```

- We know our algorithm is correct, but it is very slow
- Let’s reformulate it with dynamic programming

Formulating Subset Sum for Dynamic Programming

```
SubsetSum(X[1..n], i, T):
  If T = 0:
    return True
  ElseIf T < 0 or i = 0:
    return False
  Else:
    with ← SubsetSum(X, i-1, T – X[i])
    wout ← SubsetSum(X, i-1, T)
    return with OR wout
```

Formulating Subset Sum for Dynamic Programming

What are our subproblems?

```
SubsetSum(X[1..n], i, T):
  If T = 0:
    return True
  ElseIf T < 0 or i = 0:
    return False
  Else:
    with ← SubsetSum(X, i-1, T – X[i])
    wout ← SubsetSum(X, i-1, T)
    return with OR wout
```

Formulating Subset Sum for Dynamic Programming

What are our subproblems?

What data structure can we use for memoization?

```
SubsetSum(X[1..n], i, T):
  If T = 0:
    return True
  ElseIf T < 0 or i = 0:
    return False
  Else:
    with ← SubsetSum(X, i-1, T – X[i])
    wout ← SubsetSum(X, i-1, T)
    return with OR wout
```

Formulating Subset Sum for Dynamic Programming

What are our subproblems?

What data structure can we use for memoization?

Which subproblems depend on each other, and what evaluation order does this imply?

```
SubsetSum(X[1..n], i, T):
  If T = 0:
    return True
  ElseIf T < 0 or i = 0:
    return False
  Else:
    with ← SubsetSum(X, i-1, T – X[i])
    wout ← SubsetSum(X, i-1, T)
    return with OR wout
```

Formulating Subset Sum for Dynamic Programming

What are our subproblems?

What data structure can we use for memoization?

Which subproblems depend on each other, and what evaluation order does this imply?

What are the space/time requirements?

```
SubsetSum(X[1..n], i, T):
  If T = 0:
    return True
  ElseIf T < 0 or i = 0:
    return False
  Else:
    with ← SubsetSum(X, i-1, T – X[i])
    wout ← SubsetSum(X, i-1, T)
    return with OR wout
```

Formulating Subset Sum for Dynamic Programming

What are our subproblems?

What data structure can we use for memoization?

Which subproblems depend on each other, and what evaluation order does this imply?

What are the space/time requirements?

```
SubsetSum(X[1..n], i, T):
  If T = 0:
    return True
  ElseIf T < 0 or i = 0:
    return False
  Else:
    with ← SubsetSum(X, i-1, T – X[i])
    wout ← SubsetSum(X, i-1, T)
    return with OR wout
```

What are our subproblems?

Define a boolean function $SubSum(i, t)$ that returns $True$ if and only if there is a subset of $X[i..n]$ sums to t.

Formulating Subset Sum for Dynamic Programming

What are our subproblems?

What data structure can we use for memoization?

Which subproblems depend on each other, and what evaluation order does this imply?

What are the space/time requirements?

```
SubsetSum(X[1..n], i, T):
  If T = 0:
    return True
  ElseIf T < 0 or i = 0:
    return False
  Else:
    with ← SubsetSum(X, i-1, T – X[i])
    wout ← SubsetSum(X, i-1, T)
    return with OR wout
```

What are our subproblems?

Define a boolean function $SubSum(i, t)$ that returns $True$ if and only if there is a subset of $X[i..n]$ sums to t.

At an arbitrary iteration $1 \leq i \leq n + 1$ and $t \leq T$

Formulating Subset Sum for Dynamic Programming

What are our subproblems?

What data structure can we use for memoization?

Which subproblems depend on each other, and what evaluation order does this imply?

What are the space/time requirements?

```
SubsetSum(X[1..n], i, T):
  If T = 0:
    return True
  ElseIf T < 0 or i = 0:
    return False
  Else:
    with ← SubsetSum(X, i-1, T – X[i])
    wout ← SubsetSum(X, i-1, T)
    return with OR wout
```

What are our subproblems?

Define a boolean function $SubSum(i, t)$ that returns $True$ if and only if there is a subset of $X[i..n]$ sums to t.

At an arbitrary iteration $1 \leq i \leq n + 1$ and $t \leq T$

$$SubSum(i, t) = \begin{cases} \\ \\ \end{cases}$$

Formulating Subset Sum for Dynamic Programming

What are our subproblems?

What data structure can we use for memoization?

Which subproblems depend on each other, and what evaluation order does this imply?

What are the space/time requirements?

```
SubsetSum(X[1..n], i, T):
  If T = 0:
    return True
  ElseIf T < 0 or i = 0:
    return False
  Else:
    with ← SubsetSum(X, i-1, T – X[i])
    wout ← SubsetSum(X, i-1, T)
    return with OR wout
```

What are our subproblems?

Define a boolean function $SubSum(i, t)$ that returns $True$ if and only if there is a subset of $X[i..n]$ sums to t.

At an arbitrary iteration $1 \leq i \leq n + 1$ and $t \leq T$

$$SubSum(i,t) = \begin{cases} True & if\ t = 0 \\ & \\ & \end{cases}$$

Formulating Subset Sum for Dynamic Programming

What are our subproblems?

What data structure can we use for memoization?

Which subproblems depend on each other, and what evaluation order does this imply?

What are the space/time requirements?

```
SubsetSum(X[1..n], i, T):
  If T = 0:
    return True
  ElseIf T < 0 or i = 0:
    return False
  Else:
    with ← SubsetSum(X, i-1, T – X[i])
    wout ← SubsetSum(X, i-1, T)
    return with OR wout
```

What are our subproblems?

Define a boolean function $SubSum(i, t)$ that returns $True$ if and only if there is a subset of $X[i..n]$ sums to t.

At an arbitrary iteration $1 \leq i \leq n + 1$ and $t \leq T$

$$SubSum(i, t) = \begin{cases} True & if\ t = 0 \\ False & if\ i > n \\ & \\ & \end{cases}$$

Formulating Subset Sum for Dynamic Programming

What are our subproblems?

What data structure can we use for memoization?

Which subproblems depend on each other, and what evaluation order does this imply?

What are the space/time requirements?

```
SubsetSum(X[1..n], i, T):
  If T = 0:
    return True
  ElseIf T < 0 or i = 0:
    return False
  Else:
    with ← SubsetSum(X, i-1, T – X[i])
    wout ← SubsetSum(X, i-1, T)
    return with OR wout
```

What are our subproblems?

Define a boolean function $SubSum(i, t)$ that returns $True$ if and only if there is a subset of $X[i..n]$ sums to t.

At an arbitrary iteration $1 \leq i \leq n + 1$ and $t \leq T$

$$SubSum(i,t) = \begin{cases} True & if\ t = 0 \\ False & if\ i > n \\ SubSum(i+1,t) & if\ t < X[i] \\ & \end{cases}$$

Formulating Subset Sum for Dynamic Programming

What are our subproblems?

What data structure can we use for memoization?

Which subproblems depend on each other, and what evaluation order does this imply?

What are the space/time requirements?

```
SubsetSum(X[1..n], i, T):
  If T = 0:
    return True
  ElseIf T < 0 or i = 0:
    return False
  Else:
    with ← SubsetSum(X, i-1, T – X[i])
    wout ← SubsetSum(X, i-1, T)
    return with OR wout
```

What are our subproblems?

Define a boolean function $SubSum(i,t)$ that returns $True$ if and only if there is a subset of $X[i..n]$ sums to t.

At an arbitrary iteration $1 \le i \le n+1$ and $t \le T$

$$SubSum(i,t) = \begin{cases} True & if\ t = 0 \\ False & if\ i > n \\ SubSum(i+1,t) & if\ t < X[i] \\ SubSum(i+1,t) \vee SubSum(i+1, t+X[i]) & otherwise \end{cases}$$

Formulating Subset Sum for Dynamic Programming

What are our subproblems?

At an arbitrary iteration $1 \le i \le n+1$ and $t \le T$

$$SS(i,t) = \begin{cases} True & if\ t = 0 \\ False & if\ i > n \\ SubSum(i+1,t) & if\ t < X[i] \\ SubSum(i+1,t) \lor SubSum(i+1, t - X[i]) & otherwise \end{cases}$$

```
SubsetSum(X[1..n], i, T):
  If T = 0:
    return True
  ElseIf T < 0 or i = 0:
    return False
  Else:
    with ← SubsetSum(X, i-1, T – X[i])
    wout ← SubsetSum(X, i-1, T)
    return with OR wout
```

What data structure can we use for memoization?

Which subproblems depend on each other, and what evaluation order does this imply?

What are the space/time requirements?

What data structure can we use for memoization?

Formulating Subset Sum for Dynamic Programming

What are our subproblems?

At an arbitrary iteration $1 \le i \le n+1$ and $t \le T$

$$SS(i,t) = \begin{cases} True & if\ t = 0 \\ False & if\ i > n \\ SubSum(i+1,t) & if\ t < X[i] \\ SubSum(i+1,t) \lor SubSum(i+1,t-X[i]) & otherwise \end{cases}$$

```
SubsetSum(X[1..n], i, T):
  If T = 0:
    return True
  ElseIf T < 0 or i = 0:
    return False
  Else:
    with ← SubsetSum(X, i-1, T – X[i])
    wout ← SubsetSum(X, i-1, T)
    return with OR wout
```

What data structure can we use for memoization?

Which subproblems depend on each other, and what evaluation order does this imply?

What are the space/time requirements?

What data structure can we use for memoization?

We can fill a 2 dimensional array with the following dimensions:
$S[1..n+1, 0..T] = SubSum(i,t)$

	$S(1,0)$	$S(1,1)$	$S(1,2)$	$S(1,3)$
	$S(2,0)$	$S(2,1)$	$S(2,2)$	$S(2,3)$
n	$S(3,0)$	$S(3,1)$	$S(3,2)$	$S(3,3)$
	$S(4,0)$	$S(4,1)$	$S(4,2)$	$S(4,3)$
		T		

Formulating Subset Sum for Dynamic Programming

What are our subproblems?

At an arbitrary iteration $1 \leq i \leq n+1$ and $t \leq T$

$$SS(i,t) = \begin{cases} True & if\ t = 0 \\ False & if\ i > n \\ SubSum(i+1,t) & if\ t < X[i] \\ SubSum(i+1,t) \lor SubSum(i+1, t - X[i]) & otherwise \end{cases}$$

What data structure can we use for memoization?

We can fill a 2 dimensional array with the following dimensions:

$S[1..n+1, 0..T] = SubSum(i,t)$

Which subproblems depend on each other, and what evaluation order does this imply?

What are the space/time requirements?

```
SubsetSum(X[1..n], i, T):
  If T = 0:
    return True
  ElseIf T < 0 or i = 0:
    return False
  Else:
    with ← SubsetSum(X, i-1, T – X[i])
    wout ← SubsetSum(X, i-1, T)
    return with OR wout
```

	$S(1,0)$	$S(1,1)$	$S(1,2)$	$S(1,3)$
	$S(2,0)$	$S(2,1)$	$S(2,2)$	$S(2,3)$
n	$S(3,0)$	$S(3,1)$	$S(3,2)$	$S(3,3)$
	$S(4,0)$	$S(4,1)$	$S(4,2)$	$S(4,3)$

T

Formulating Subset Sum for Dynamic Programming

What are our subproblems?

At an arbitrary iteration $1 \le i \le n+1$ and $t \le T$

$$SS(i,t) = \begin{cases} True & if\ t = 0 \\ False & if\ i > n \\ SubSum(i+1,t) & if\ t < X[i] \\ SubSum(i+1,t)\ \vee SubSum(i+1, t - X[i]) & otherwise \end{cases}$$

What data structure can we use for memoization?

We can fill a 2 dimensional array with the following dimensions:
$S[1..n+1, 0..T] = SubSum(i,t)$

Which subproblems depend on each other, and what evaluation order does this imply?

$SubSum(i,t)$ can depend on $SubSum(i+1,t)$ and $SubSum(i+1, t - X[i])$. So we can start at the bottom of the table and work up.

What are the space/time requirements?

```
SubsetSum(X[1..n], i, T):
  If T = 0:
    return True
  ElseIf T < 0 or i = 0:
    return False
  Else:
    with ← SubsetSum(X, i-1, T – X[i])
    wout ← SubsetSum(X, i-1, T)
    return with OR wout
```

n (rows), T (columns)

$S(1,0)$	$S(1,1)$	$S(1,2)$	$S(1,3)$
$S(2,0)$	$S(2,1)$	$S(2,2)$	$S(2,3)$
$S(3,0)$	$S(3,1)$	$S(3,2)$	$S(3,3)$
$S(4,0)$	$S(4,1)$	$S(4,2)$	$S(4,3)$

Formulating Subset Sum for Dynamic Programming

What are our subproblems?

At an arbitrary iteration $1 \leq i \leq n + 1$ and $t \leq T$

$$SS(i,t) = \begin{cases} True & if\ t = 0 \\ False & if\ i > n \\ SubSum(i+1, t) & if\ t < X[i] \\ SubSum(i+1,t)\ \vee SubSum(i+1, t - X[i]) & otherwise \end{cases}$$

What data structure can we use for memoization?

We can fill a 2 dimensional array with the following dimensions:
$S[1..n+1, 0..T] = SubSum(i,t)$

Which subproblems depend on each other, and what evaluation order does this imply?

$SubSum(i,t)$ can depend on $SubSum(i+1,t)$ and $SubSum(i+1, t - X[i])$. So we can start at the bottom of the table and work up.

What are the space/time requirements?

```
SubsetSum(X[1..n], i, T):
  If T = 0:
    return True
  ElseIf T < 0 or i = 0:
    return False
  Else:
    with ← SubsetSum(X, i-1, T – X[i])
    wout ← SubsetSum(X, i-1, T)
    return with OR wout
```

End ↑

n

$S(1,0)$	$S(1,1)$	$S(1,2)$	$S(1,3)$
$S(2,0)$	$S(2,1)$	$S(2,2)$	$S(2,3)$
$S(3,0)$	$S(3,1)$	$S(3,2)$	$S(3,3)$
$S(4,0)$	$S(4,1)$	$S(4,2)$	$S(4,3)$

T

Start

Formulating Subset Sum for Dynamic Programming

What are our subproblems?

At an arbitrary iteration $1 \leq i \leq n+1$ and $t \leq T$

$$SS(i,t) = \begin{cases} True & if\ t = 0 \\ False & if\ i > n \\ SubSum(i+1,t) & if\ t < X[i] \\ SubSum(i+1,t) \lor SubSum(i+1, t - X[i]) & otherwise \end{cases}$$

What data structure can we use for memoization?

We can fill a 2 dimensional array with the following dimensions: $S[1..n+1, 0..T] = SubSum(i,t)$

Which subproblems depend on each other, and what evaluation order does this imply?

$SubSum(i,t)$ can depend on $SubSum(i+1,t)$ and $SubSum(i+1, t - X[i])$. So we can start at the bottom of the table and work up.

What are the space/time requirements?

```
SubsetSum(X[1..n], i, T):
  If T = 0:
    return True
  ElseIf T < 0 or i = 0:
    return False
  Else:
    with ← SubsetSum(X, i-1, T – X[i])
    wout ← SubsetSum(X, i-1, T)
    return with OR wout
```

What are the space/time requirements?

n	$S(1,0)$	$S(1,1)$	$S(1,2)$	$S(1,3)$
	$S(2,0)$	$S(2,1)$	$S(2,2)$	$S(2,3)$
	$S(3,0)$	$S(3,1)$	$S(3,2)$	$S(3,3)$
	$S(4,0)$	$S(4,1)$	$S(4,2)$	$S(4,3)$

T

Space requirement is $O(nT)$

Formulating Subset Sum for Dynamic Programming

What are our subproblems?

At an arbitrary iteration $1 \leq i \leq n+1$ and $t \leq T$

$$SS(i,t) = \begin{cases} True & if\ t = 0 \\ False & if\ i > n \\ SubSum(i+1,t) & if\ t < X[i] \\ SubSum(i+1,t) \lor SubSum(i+1, t - X[i]) & otherwise \end{cases}$$

What data structure can we use for memoization?

We can fill a 2 dimensional array with the following dimensions: $S[1..n+1, 0..T] = SubSum(i,t)$

Which subproblems depend on each other, and what evaluation order does this imply?

$SubSum(i,t)$ can depend on $SubSum(i+1,t)$ and $SubSum(i+1, t - X[i])$. So we can start at the bottom of the table and work up.

What are the space/time requirements?

Space: $O(nT)$

```
SubsetSum(X[1..n], i, T):
  If T = 0:
    return True
  ElseIf T < 0 or i = 0:
    return False
  Else:
    with ← SubsetSum(X, i-1, T – X[i])
    wout ← SubsetSum(X, i-1, T)
    return with OR wout
```

What are the space/time requirements?

	$S(1,0)$	$S(1,1)$	$S(1,2)$	$S(1,3)$
	$S(2,0)$	$S(2,1)$	$S(2,2)$	$S(2,3)$
n	$S(3,0)$	$S(3,1)$	$S(3,2)$	$S(3,3)$
	$S(4,0)$	$S(4,1)$	$S(4,2)$	$S(4,3)$

T

Formulating Subset Sum for Dynamic Programming

What are our subproblems?

At an arbitrary iteration $1 \leq i \leq n+1$ and $t \leq T$

$$SS(i,t) = \begin{cases} True & if\ t = 0 \\ False & if\ i > n \\ SubSum(i+1,t) & if\ t < X[i] \\ SubSum(i+1,t)\ \vee SubSum(i+1, t-X[i]) & otherwise \end{cases}$$

What data structure can we use for memoization?

We can fill a 2 dimensional array with the following dimensions: $S[1..n+1, 0..T] = SubSum(i,t)$

Which subproblems depend on each other, and what evaluation order does this imply?

$SubSum(i,t)$ can depend on $SubSum(i+1,t)$ and $SubSum(i+1,t-X[i])$. So we can start at the bottom of the table and work up.

What are the space/time requirements?

Space: $O(nT)$
Time: $O(nT)$

```
SubsetSum(X[1..n], i, T):
  If T = 0:
    return True
  ElseIf T < 0 or i = 0:
    return False
  Else:
    with ← SubsetSum(X, i-1, T – X[i])
    wout ← SubsetSum(X, i-1, T)
    return with OR wout
```

What are the space/time requirements?

n (rows) × T (columns):

$S(1,0)$	$S(1,1)$	$S(1,2)$	$S(1,3)$
$S(2,0)$	$S(2,1)$	$S(2,2)$	$S(2,3)$
$S(3,0)$	$S(3,1)$	$S(3,2)$	$S(3,3)$
$S(4,0)$	$S(4,1)$	$S(4,2)$	$S(4,3)$

Using our evaluation order, we can fill the table in constant time per update, so the time complexity is also $O(nT)$

Formulating Subset Sum for Dynamic Programming

What are our subproblems?

At an arbitrary iteration $1 \leq i \leq n+1$ and $t \leq T$

$$SS(i,t) = \begin{cases} True & if\ t = 0 \\ False & if\ i > n \\ SubSum(i+1,t) & if\ t < X[i] \\ SubSum(i+1,t)\ \lor SubSum(i+1,t-X[i]) & otherwise \end{cases}$$

What data structure can we use for memoization?

We can fill a 2 dimensional array with the following dimensions: $S[1..n+1, 0..T] = SubSum(i,t)$

Which subproblems depend on each other, and what evaluation order does this imply?

$SubSum(i,t)$ can depend on $SubSum(i+1,t)$ and $SubSum(i+1,t-X[i])$. So we can start at the bottom of the table and work up.

What are the space/time requirements?

Space: $O(nT)$
Time: $O(nT)$

```
FastSubsetSum(X[1..n], T):
  S[n+1, 0] ← True
  for t ← 1 to T:
    S[n+1, t] ← False
  for i ← n down to 1:
    S[i, 0] ← True
    for t ← 1 to X[i] − 1:
      S[i, t] ← S[i+1, t]
    for t ← X[i] to T:
      S[i, t] ← S[i+1, t] ∨ S[i+1, t − X[i]]
  return S[1, T]
```

Formulating Subset Sum for Dynamic Programming

What are our subproblems?

At an arbitrary iteration $1 \leq i \leq n+1$ and $t \leq T$

$$SS(i,t) = \begin{cases} True & if\ t = 0 \\ False & if\ i > n \\ SubSum(i+1,t) & if\ t < X[i] \\ SubSum(i+1,t)\ \vee SubSum(i+1,t-X[i]) & otherwise \end{cases}$$

What data structure can we use for memoization?

We can fill a 2 dimensional array with the following dimensions: $S[1..n+1, 0..T] = SubSum(i,t)$

Which subproblems depend on each other, and what evaluation order does this imply?

$SubSum(i,t)$ can depend on $SubSum(i+1,t)$ and $SubSum(i+1,t-X[i])$. So we can start at the bottom of the table and work up.

What are the space/time requirements?

Space: $O(nT)$
Time: $O(nT)$

```
FastSubsetSum(X[1..n], T):
  S[n+1, 0] ← True
  for t ← 1 to T:
    S[n+1, t] ← False
  for i ← n down to 1:
    S[i, 0] ← True
    for t ← 1 to X[i] − 1:
      S[i, t] ← S[i+1, t]
    for t ← X[i] to T:
      S[i, t] ← S[i+1, t] ∨ S[i+1, t − X[i]]
  return S[1, T]
```

Let's see an example

Dynamic Programming Subset Sum Example

```
FastSubsetSum(X[1..n], T):
  S[n+1, 0] ← True
  for t ← 1 to T:
    S[n+1, t] ← False
  for i ← n down to 1:
    S[i, 0] ← True
    for t ← 1 to X[i] − 1:
      S[i, t] ← S[i+1, t]
    for t ← X[i] to T:
      S[i, t] ← S[i+1, t] ∨ S[i+1, t − X[i]]
  return S[1, T]
```

$X = [1,2,3], T = 3$

$S(1,0)$	$S(1,1)$	$S(1,2)$	$S(1,3)$
$S(2,0)$	$S(2,1)$	$S(2,2)$	$S(2,3)$
$S(3,0)$	$S(3,1)$	$S(3,2)$	$S(3,3)$
$S(4,0)$	$S(4,1)$	$S(4,2)$	$S(4,3)$

Dynamic Programming Subset Sum Example

```
FastSubsetSum(X[1..n], T):
  S[n+1, 0] ← True
  for t ← 1 to T:
    S[n+1, t] ← False
  for i← n down to 1:
    S[i, 0] ← True
    for t ← 1 to X[i] − 1:
      S[i, t] ← S[i+1, t]
    for t ← X[i] to T:
      S[i, t] ← S[i+1, t] ∨ S[i+1, t − X[i]]
  return S[1, T]
```

$X = [1,2,3], T = 3$

$S(1,0)$	$S(1,1)$	$S(1,2)$	$S(1,3)$
$S(2,0)$	$S(2,1)$	$S(2,2)$	$S(2,3)$
$S(3,0)$	$S(3,1)$	$S(3,2)$	$S(3,3)$
T	F	F	F

Dynamic Programming Subset Sum Example

```
FastSubsetSum(X[1..n], T):
  S[n+1, 0] ← True
  for t ← 1 to T:
    S[n+1, t] ← False
  for i ← n down to 1:
    S[i, 0] ← True
    for t ← 1 to X[i] − 1:
      S[i, t] ← S[i+1, t]
    for t ← X[i] to T:
      S[i, t] ← S[i+1, t] ∨ S[i+1, t − X[i]]
  return S[1, T]
```

$X = [1,2,3], T = 3$

	$S(1,0)$	$S(1,1)$	$S(1,2)$	$S(1,3)$
	$S(2,0)$	$S(2,1)$	$S(2,2)$	$S(2,3)$
X[i] = 3	T	$S(3,1)$	$S(3,2)$	$S(3,3)$
	T	F	F	F

Dynamic Programming Subset Sum Example

```
FastSubsetSum(X[1..n], T):
  S[n+1, 0] ← True
  for t ← 1 to T:
    S[n+1, t] ← False
  for i ← n down to 1:
    S[i, 0] ← True
    for t ← 1 to X[i] − 1:
      S[i, t] ← S[i+1, t]
    for t ← X[i] to T:
      S[i, t] ← S[i+1, t] ∨ S[i+1, t − X[i]]
  return S[1, T]
```

$X = [1,2,3], T = 3$

	$S(1,0)$	$S(1,1)$	$S(1,2)$	$S(1,3)$
	$S(2,0)$	$S(2,1)$	$S(2,2)$	$S(2,3)$
X[i] = 3	T	F	F	$S(3,3)$
	T	F	F	F

Dynamic Programming Subset Sum Example

```
FastSubsetSum(X[1..n], T):
  S[n+1, 0] ← True
  for t ← 1 to T:
    S[n+1, t] ← False
  for i← n down to 1:
    S[i, 0] ← True
    for t ← 1 to X[i] − 1:
      S[i, t] ← S[i+1, t]
    for t ← X[i] to T:
      S[i, t] ← S[i+1, t] ∨ S[i+1, t − X[i]]
  return S[1, T]
```

$X = [1,2,3], T = 3$

	$S(1,0)$	$S(1,1)$	$S(1,2)$	$S(1,3)$
	$S(2,0)$	$S(2,1)$	$S(2,2)$	$S(2,3)$
X[i] = 3	T	F	F	T
	T	F	F	F

Dynamic Programming Subset Sum Example

```
FastSubsetSum(X[1..n], T):
  S[n+1, 0] ← True
  for t ← 1 to T:
    S[n+1, t] ← False
  for i ← n down to 1:
    S[i, 0] ← True
    for t ← 1 to X[i] − 1:
      S[i, t] ← S[i+1, t]
    for t ← X[i] to T:
      S[i, t] ← S[i+1, t] ∨ S[i+1, t − X[i]]
  return S[1, T]
```

$X = [1,2,3], T = 3$

	$S(1,0)$	$S(1,1)$	$S(1,2)$	$S(1,3)$
X[i] = 2	$S(2,0)$	$S(2,1)$	$S(2,2)$	$S(2,3)$
	T	F	F	T
	T	F	F	F

Dynamic Programming Subset Sum Example

```
FastSubsetSum(X[1..n], T):
  S[n+1, 0] ← True
  for t ← 1 to T:
    S[n+1, t] ← False
  for i ← n down to 1:
    S[i, 0] ← True
    for t ← 1 to X[i] − 1:
      S[i, t] ← S[i+1, t]
    for t ← X[i] to T:
      S[i, t] ← S[i+1, t] ∨ S[i+1, t − X[i]]
  return S[1, T]
```

$X = [1,2,3], T = 3$

	$S(1,0)$	$S(1,1)$	$S(1,2)$	$S(1,3)$
X[i] = 2	T	F	$S(2,2)$	$S(2,3)$
	T	F	F	T
	T	F	F	F

Dynamic Programming Subset Sum Example

```
FastSubsetSum(X[1..n], T):
  S[n+1,0] ← True
  for t ← 1 to T:
    S[n+1,t] ← False
  for i ← n down to 1:
    S[i,0] ← True
    for t ← 1 to X[i] − 1:
      S[i,t] ← S[i+1,t]
    for t ← X[i] to T:
      S[i,t] ← S[i+1,t] ∨ S[i+1,t − X[i]]
  return S[1,T]
```

$$X = [1,2,3], T = 3$$

	$S(1,0)$	$S(1,1)$	$S(1,2)$	$S(1,3)$
X[i] = 2	T	F	T	$S(2,3)$
	T	F	F	T
	T	F	F	F

Dynamic Programming Subset Sum Example

```
FastSubsetSum(X[1..n], T):
   S[n+1, 0] ← True
   for t ← 1 to T:
      S[n+1, t] ← False
   for i ← n down to 1:
      S[i, 0] ← True
      for t ← 1 to X[i] − 1:
         S[i, t] ← S[i+1, t]
      for t ← X[i] to T:
         S[i, t] ← S[i+1, t] ∨ S[i+1, t − X[i]]
   return S[1, T]
```

$X = [1,2,3], T = 3$

	$S(1,0)$	$S(1,1)$	$S(1,2)$	$S(1,3)$
X[i] = 2	T	F	T	T
	T	F	F	T
	T	F	F	F

Dynamic Programming Subset Sum Example

```
FastSubsetSum(X[1..n], T):
   S[n+1, 0] ← True
   for t ← 1 to T:
      S[n+1, t] ← False
   for i ← n down to 1:
      S[i, 0] ← True
      for t ← 1 to X[i] − 1:
         S[i, t] ← S[i+1, t]
      for t ← X[i] to T:
         S[i, t] ← S[i+1, t] ∨ S[i+1, t − X[i]]
   return S[1, T]
```

$X = [1,2,3], T = 3$

X[i] = 1	T	$S(1,1)$	$S(1,2)$	$S(1,3)$
	T	F	T	T
	T	F	F	T
	T	F	F	F

Dynamic Programming Subset Sum Example

```
FastSubsetSum(X[1..n], T):
   S[n+1, 0] ← True
   for t ← 1 to T:
      S[n+1, t] ← False
   for i ← n down to 1:
      S[i, 0] ← True
      for t ← 1 to X[i] − 1:
         S[i, t] ← S[i+1, t]
      for t ← X[i] to T:
         S[i, t] ← S[i+1, t] ∨ S[i+1, t − X[i]]
   return S[1, T]
```

$X = [1,2,3], T = 3$

X[i] = 1	T	$S(1,1)$	$S(1,2)$	$S(1,3)$
	T	F	T	T
	T	F	F	T
	T	F	F	F

Dynamic Programming Subset Sum Example

```
FastSubsetSum(X[1..n], T):
   S[n+1, 0] ← True
   for t ← 1 to T:
      S[n+1, t] ← False
   for i ← n down to 1:
      S[i, 0] ← True
      for t ← 1 to X[i] − 1:
         S[i, t] ← S[i+1, t]
      for t ← X[i] to T:
         S[i, t] ← S[i+1, t] ∨ S[i+1, t − X[i]]
   return S[1, T]
```

$X = [1,2,3], T = 3$

X[i] = 1	T	$S(1,1)$	$S(1,2)$	$S(1,3)$
	T	F	T	T
	T	F	F	T
	T	F	F	F

Since $S[1,3]$ checks $S[2,3]$ which is $True$, we know we have a solution!

Dynamic Programming Subset Sum Example

```
FastSubsetSum(X[1..n], T):
  S[n+1, 0] ← True
  for t ← 1 to T:
    S[n+1, t] ← False
  for i ← n down to 1:
    S[i, 0] ← True
    for t ← 1 to X[i] − 1:
      S[i, t] ← S[i+1, t]
    for t ← X[i] to T:
      S[i, t] ← S[i+1, t] ∨ S[i+1, t − X[i]]
  return S[1, T]
```

$X = [1,2,3], T = 3$

X[i] = 1	T	T	T	**T**
	T	F	T	T
	T	F	F	T
	T	F	F	F

Since $S[1,3]$ checks $S[2,3]$ which is $True$, we know we have a solution!

Subset Sum Wrap

What are our subproblems?

At an arbitrary iteration $1 \leq i \leq n+1$ and $t \leq T$

$$SS(i,t) = \begin{cases} True & if\ t = 0 \\ False & if\ i > n \\ SubSum(i+1,t) & if\ t < X[i] \\ SubSum(i+1,t) \lor SubSum(i+1, t - X[i]) & otherwise \end{cases}$$

What data structure can we use for memoization?

We can fill a 2 dimensional array with the following dimensions: $S[1..n+1, 0..T] = SubSum(i,t)$

Which subproblems depend on each other, and what evaluation order does this imply?

$SubSum(i,t)$ can depend on $SubSum(i+1,t)$ and $SubSum(i+1, t - X[i])$. So we can start at the bottom of the table and work up.

What are the space/time requirements?

Space: $O(nT)$
Time: $O(nT)$

```
FastSubsetSum(X[1..n], T):
  S[n+1, 0] ← True
  for t ← 1 to T:
    S[n+1, t] ← False
  for i ← n down to 1:
    S[i, 0] ← True
    for t ← 1 to X[i] − 1:
      S[i, t] ← S[i+1, t]
    for t ← X[i] to T:
      S[i, t] ← S[i+1, t] ∨ S[i+1, t − X[i]]
  return S[1, T]
```

Is FastSubsetSum *always* faster than the recursive version?

Subset Sum Wrap

What are our subproblems?

At an arbitrary iteration $1 \leq i \leq n+1$ and $t \leq T$

$$SS(i,t) = \begin{cases} True & if\ t = 0 \\ False & if\ i > n \\ SubSum(i+1,t) & if\ t < X[i] \\ SubSum(i+1,t)\ \vee SubSum(i+1,t-X[i]) & otherwise \end{cases}$$

What data structure can we use for memoization?

We can fill a 2 dimensional array with the following dimensions:
$S[1..n+1, 0..T] = SubSum(i,t)$

Which subproblems depend on each other, and what evaluation order does this imply?

$SubSum(i,t)$ can depend on $SubSum(i+1,t)$ and $SubSum(i+1,t-X[i])$. So we can start at the bottom of the table and work up.

What are the space/time requirements?

Space: $O(nT)$
Time: $O(nT)$

```
FastSubsetSum(X[1..n], T):
  S[n+1,0] ← True
  for t ← 1 to T:
    S[n+1,t] ← False
  for i← n down to 1:
    S[i,0] ← True
    for t ← 1 to X[i] − 1:
      S[i,t] ← S[i+1,t]
    for t ← X[i] to T:
      S[i,t] ← S[i+1,t] ∨ S[i+1,t − X[i]]
  return S[1,T]
```

Is FastSubsetSum *always* faster than the recursive version?

No! If $T \gg 2^n$, the recursive version is actually faster!

Edit Distance

The *edit distance* between two strings is the minimum number of insertions, deletions, and substitutions that will transform one string into the other.

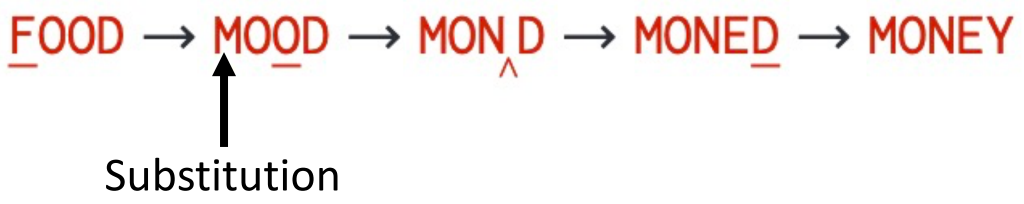
Edit Distance

The *edit distance* between two strings is the minimum number of insertions, deletions, and substitutions that will transform one string into the other.

FOOD → MOOD → MON^D → MONED → MONEY

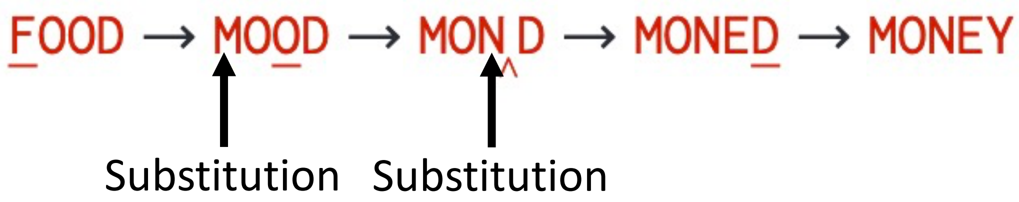
Edit Distance

The *edit distance* between two strings is the minimum number of insertions, deletions, and substitutions that will transform one string into the other.



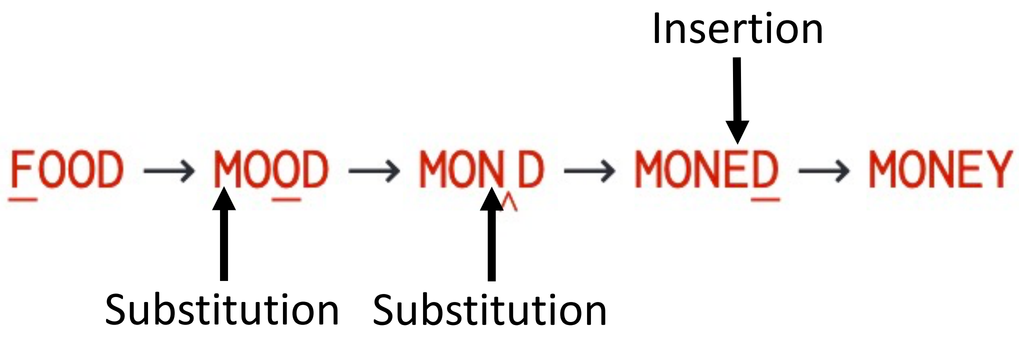
Edit Distance

The *edit distance* between two strings is the minimum number of insertions, deletions, and substitutions that will transform one string into the other.



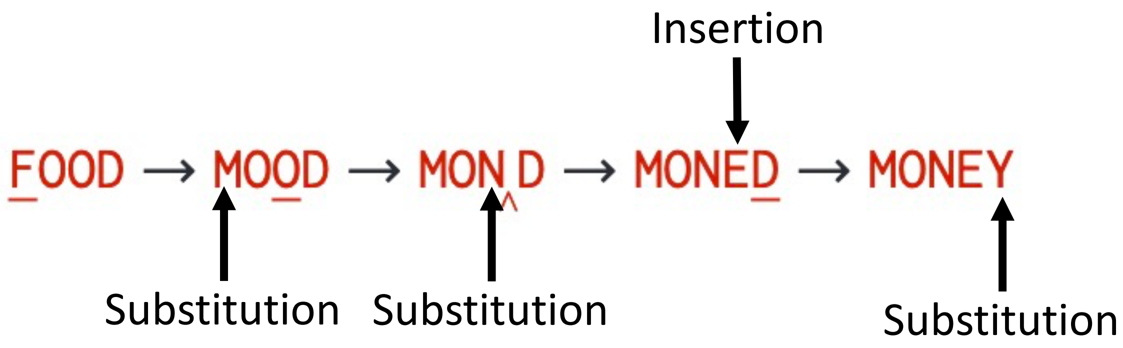
Edit Distance

The *edit distance* between two strings is the minimum number of insertions, deletions, and substitutions that will transform one string into the other.



Edit Distance

The *edit distance* between two strings is the minimum number of insertions, deletions, and substitutions that will transform one string into the other.



Edit Distance

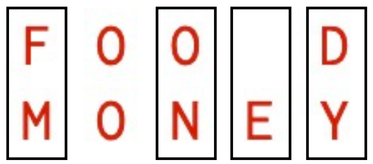
The *edit distance* between two strings is the minimum number of insertions, deletions, and substitutions that will transform one string into the other.

```
F O O   D
M O N E Y
```

Alternative: Align the strings and count the differences

Edit Distance

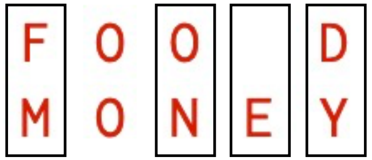
The *edit distance* between two strings is the minimum number of insertions, deletions, and substitutions that will transform one string into the other.



Alternative: Align the strings and count the differences

Edit Distance

The *edit distance* between two strings is the minimum number of insertions, deletions, and substitutions that will transform one string into the other.



$EditDistance(\text{food}, \text{money}) = 4$

Alternative: Align the strings and count the differences

Formulating a recursive edit distance

The *edit distance* between two strings is the minimum number of insertions, deletions, and substitutions that will transform one string into the other.

```
F O O   D
M O N E Y
```

Formulating a recursive edit distance

The *edit distance* between two strings is the minimum number of insertions, deletions, and substitutions that will transform one string into the other.

```
F O O   D
M O N E Y
```

What should our subproblems be?

Formulating a recursive edit distance

The *edit distance* between two strings is the minimum number of insertions, deletions, and substitutions that will transform one string into the other.

```
F O O   D
M O N E Y
```

What should our subproblems be?

- Imagine that we have this alignment representation for the optimal edit distance

Formulating a recursive edit distance

The *edit distance* between two strings is the minimum number of insertions, deletions, and substitutions that will transform one string into the other.

F	O	O		D
M	O	N	E	Y

What should our subproblems be?

- Imagine that we have this alignment representation for the optimal edit distance
- Remove the last column

Formulating a recursive edit distance

The *edit distance* between two strings is the minimum number of insertions, deletions, and substitutions that will transform one string into the other.

F	O	O		D
M	O	N	E	Y

What should our subproblems be?

- Imagine that we have this alignment representation for the optimal edit distance
- Remove the last column
- What must be true of the remaining prefixes?

Formulating a recursive edit distance

The *edit distance* between two strings is the minimum number of insertions, deletions, and substitutions that will transform one string into the other.

```
F O O   D
M O N E Y
```

What should our subproblems be?

- Imagine that we have this alignment representation for the optimal edit distance
- Remove the last column
- What must be true of the remaining prefixes?
    - They must also be optimal!

Formulating a recursive edit distance

The *edit distance* between two strings is the minimum number of insertions, deletions, and substitutions that will transform one string into the other.

For any two input strings $A[1..n]$ and $B[1..m]$, let

$$Edit(i,j)$$

denote the edit distance between prefixes $A[1..i]$ and $B[1..j]$. We need to compute $Edit(n,m)$.

1	2	3		4
F	O	O		D
M	O	N	E	Y
1	2	3	4	5

What should our subproblems be?

- Imagine that we have this alignment representation for the optimal edit distance
- Remove the last column
- What must be true of the remaining prefixes?
  - They must also be optimal!

Formulating a recursive edit distance

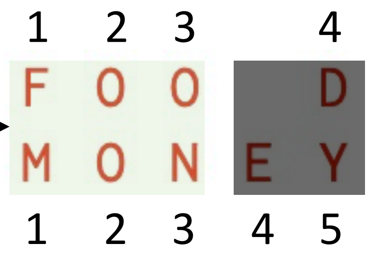
The *edit distance* between two strings is the minimum number of insertions, deletions, and substitutions that will transform one string into the other.

For any two input strings $A[1..n]$ and $B[1..m]$, let

$$Edit(i, j)$$

denote the edit distance between prefixes $A[1..i]$ and $B[1..j]$. We need to compute $Edit(n, m)$.

Decisions here do not depend on what was already computed!



What should our subproblems be?

- Imagine that we have this alignment representation for the optimal edit distance
- Remove the last column
- What must be true of the remaining prefixes?
  - They must also be optimal!

Formulating a recursive edit distance

Each call to $Edit(i, j)$ makes a decision about how to align the last column in the substring.
There are three possibilities:

```
A L G O R   I   T H M
A L   T R U I S T I C
```

Formulating a recursive edit distance

Each call to $Edit(i, j)$ makes a decision about how to align the last column in the substring.
There are three possibilities:

Arbitrary Case

1. Insertion

$$Edit(i, j-1) + 1$$

2. Deletion

3. Substitution

Formulating a recursive edit distance

Each call to $Edit(i,j)$makes a decision about how to align the last column in the substring.
There are three possibilities:

Arbitrary Case

1. Insertion

$$Edit(i, j-1)+1$$

---

2. Deletion

$$Edit(i-1, j)+1$$

---

3. Substitution

Formulating a recursive edit distance

Each call to $Edit(i, j)$ makes a decision about how to align the last column in the substring.
There are three possibilities:

Arbitrary Case

1. Insertion

$Edit(i, j-1) + 1$

---

2. Deletion

$Edit(i-1, j) + 1$

---

3. Substitution

$Edit(i-1, j-1) + 1$, if $A[i] \neq B[j]$

$Edit(i-1, j-1)$, if $A[i] = B[j]$

Formulating a recursive edit distance

Each call to $Edit(i, j)$makes a decision about how to align the last column in the substring.
There are three possibilities:

```
A L G O R   I   T H M
A L   T R U I S T I C
```

	Arbitrary Case	**Base Case**
1. Insertion 	$Edit(i, j-1)+1$	$Edit(0, j) = j$ to insert gaps in $A[0..j]$
2. Deletion 	$Edit(i-1, j)+1$	$Edit(i, 0) = i$ to delete characters from $B[0..i]$
3. Substitution  	$Edit(i-1, j-1)+1$, if $A[i] \neq B[j]$ $Edit(i-1, j-1)$, if $A[i] = B[j]$	

Formulating a recursive edit distance

Each call to $Edit(i, j)$makes a decision about how to align the last column in the substring.
There are three possibilities:

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \begin{Bmatrix} Edit(i, j-1) + 1 \\ Edit(i-1, j) + 1 \\ Edit(i-1, j-1) + [A[i] \neq B[j]] \end{Bmatrix} & \text{otherwise} \end{cases}$$

	Arbitrary Case	**Base Case**
1. Insertion	$Edit(i, j-1) + 1$	$Edit(0, j) = j$ to insert gaps in $A[0..j]$
2. Deletion	$Edit(i-1, j) + 1$	$Edit(i, 0) = i$ to delete characters from $B[0..i]$
3. Substitution	$Edit(i-1, j-1) + 1$, if $A[i] \neq B[j]$ $Edit(i-1, j-1)$, if $A[i] = B[j]$	

1. Insertion

2. Deletion

3. Substitution

Next Time

We will formulate a dynamic programming algorithm for Edit Distance and discuss the Knapsack Problem.

$$
Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \left\{ \begin{array}{c} Edit(i, j-1) + 1 \\ Edit(i-1, j) + 1 \\ Edit(i-1, j-1) + [A[i] \neq B[j]] \end{array} \right\} & \text{otherwise} \end{cases}
$$

Wrap up

No new reading assignment (still chapter 3 of Erickson)

- If you didn't follow our Edit Distance discussion, read 3.7 before Monday!

Work on homework 2! Ask questions on Piazza.

Midterm next Weds 8PM – Fri 8PM.

- Topics will be everything we have done so far:
  - Asymptotic analysis and Divide and Conquer (including recursion, backtracking, and dynamic programming)
- If there are things you have struggled with, strategize sooner rather than later about how you will review them before Wednesday!

Enjoy your weekend!