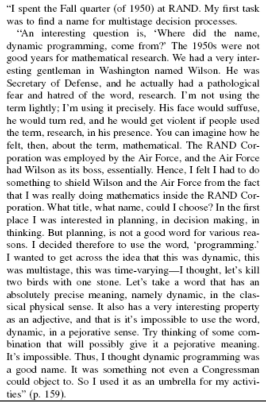
Dynamic Programming continued

David Kauchak
cs302
Spring 2013

Admin

- No office hours tomorrow
- Assignments 14 **AND** 15 will be made available today
  - assignment 15 will be programming a DP algorithm, so look at this sooner than later

Where did "dynamic programming" come from?



"I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes.

"An interesting question is, 'Where did the name, dynamic programming, come from?' The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, 'programming.' I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying—I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities" (p. 159).

Richard Bellman On the Birth of Dynamic Programming

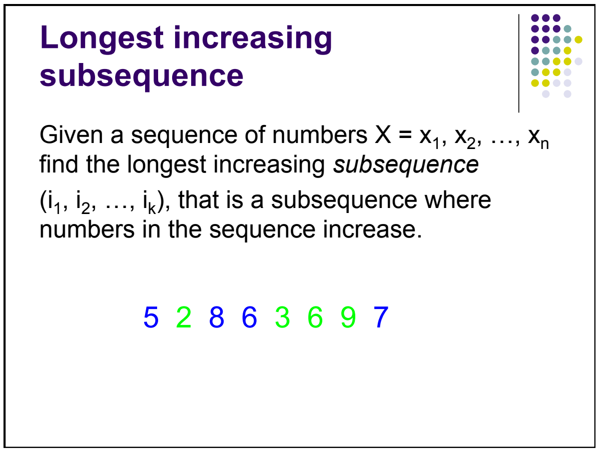
Stuart Dreyfus

http://www.eng.tau.ac.il/~ami/cd/or50/1526-5463-2002-50-01-0048.pdf

Longest increasing subsequence

Given a sequence of numbers X = $x_1, x_2, \ldots, x_n$ find the longest increasing *subsequence* $(i_1, i_2, \ldots, i_k)$, that is a subsequence where numbers in the sequence increase.

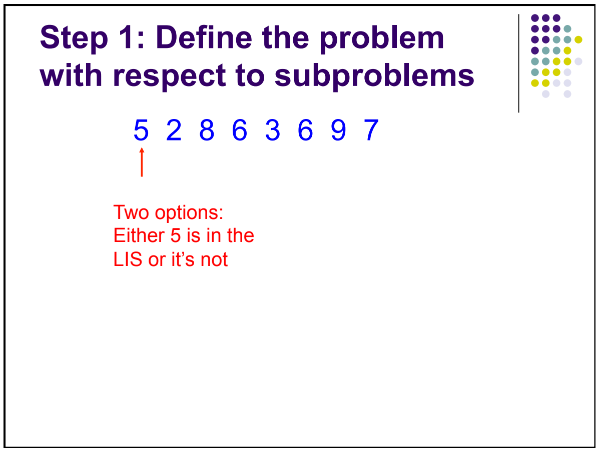
5 2 8 6 3 6 9 7



Longest increasing subsequence

Given a sequence of numbers $X = x_1, x_2, \ldots, x_n$ find the longest increasing *subsequence* $(i_1, i_2, \ldots, i_k)$, that is a subsequence where numbers in the sequence increase.

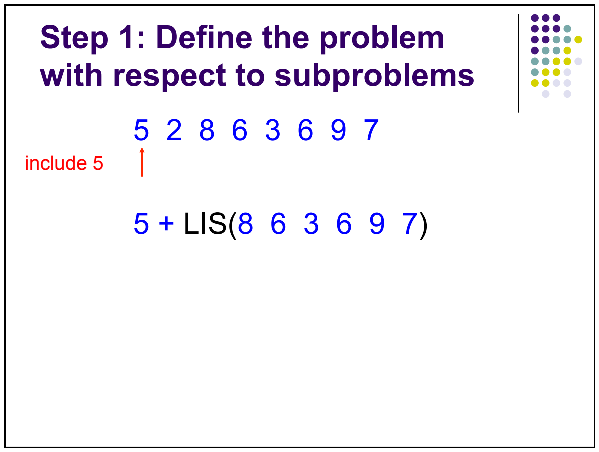
5 2 8 6 3 6 9 7



Step 1: Define the problem with respect to subproblems

5 2 8 6 3 6 9 7

Two options:
Either 5 is in the
LIS or it's not

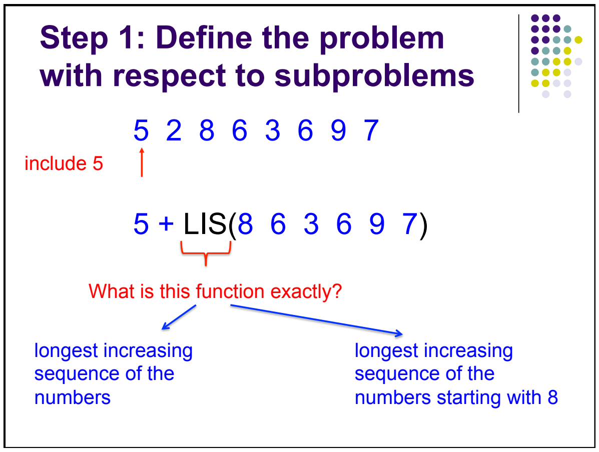


Step 1: Define the problem with respect to subproblems

5 2 8 6 3 6 9 7

include 5

5 + LIS(8 6 3 6 9 7)



Step 1: Define the problem with respect to subproblems

5 2 8 6 3 6 9 7

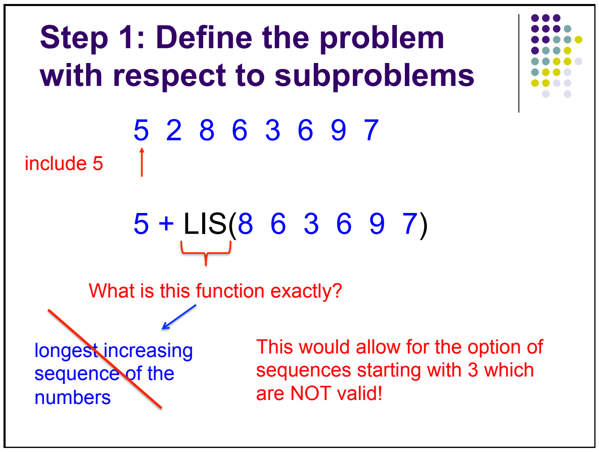
include 5

5 + LIS(8 6 3 6 9 7)

What is this function exactly?

longest increasing sequence of the numbers

longest increasing sequence of the numbers starting with 8



Step 1: Define the problem with respect to subproblems

5 2 8 6 3 6 9 7

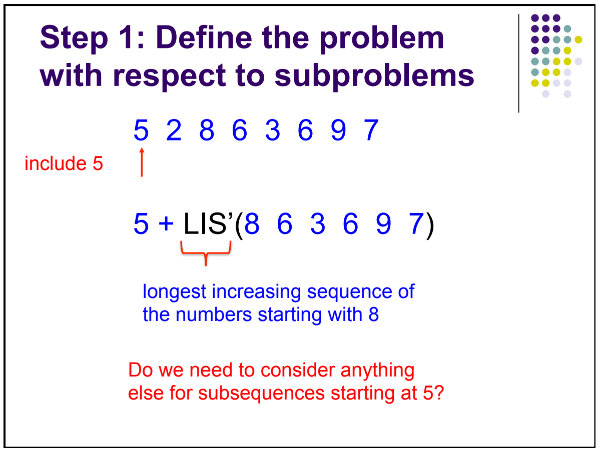
include 5 ↑

$5 + \underbrace{\text{LIS}}(8\ \ 6\ \ 3\ \ 6\ \ 9\ \ 7)$

What is this function exactly?

~~longest increasing sequence of the numbers~~

This would allow for the option of sequences starting with 3 which are NOT valid!



Step 1: Define the problem with respect to subproblems

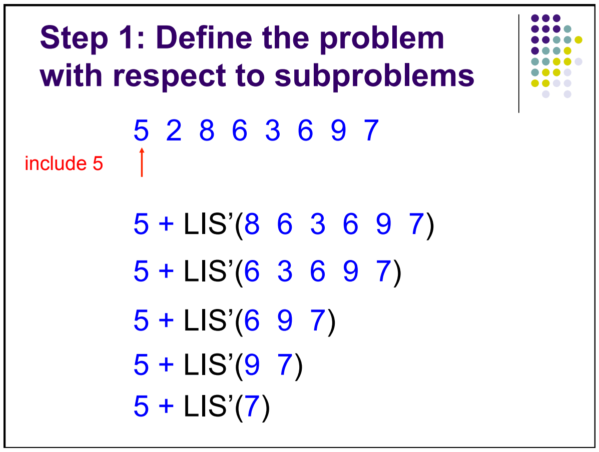
5 2 8 6 3 6 9 7

include 5 ↑

$5 + \underbrace{\text{LIS'}}(8\ \ 6\ \ 3\ \ 6\ \ 9\ \ 7)$

longest increasing sequence of the numbers starting with 8

Do we need to consider anything else for subsequences starting at 5?



Step 1: Define the problem with respect to subproblems

5 2 8 6 3 6 9 7

include 5 ↑

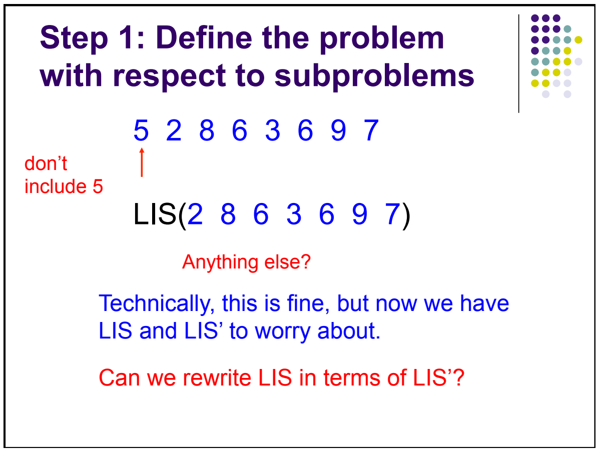
5 + LIS’(8 6 3 6 9 7)

5 + LIS’(6 3 6 9 7)

5 + LIS’(6 9 7)

5 + LIS’(9 7)

5 + LIS’(7)



Step 1: Define the problem with respect to subproblems

5 2 8 6 3 6 9 7

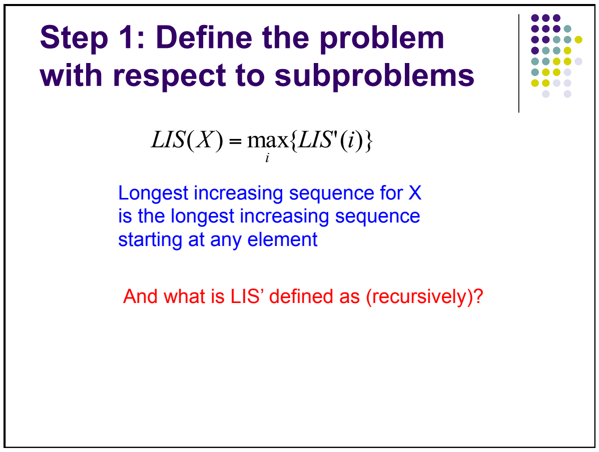
don’t include 5 ↑

LIS(2 8 6 3 6 9 7)

Anything else?

Technically, this is fine, but now we have LIS and LIS’ to worry about.

Can we rewrite LIS in terms of LIS’?

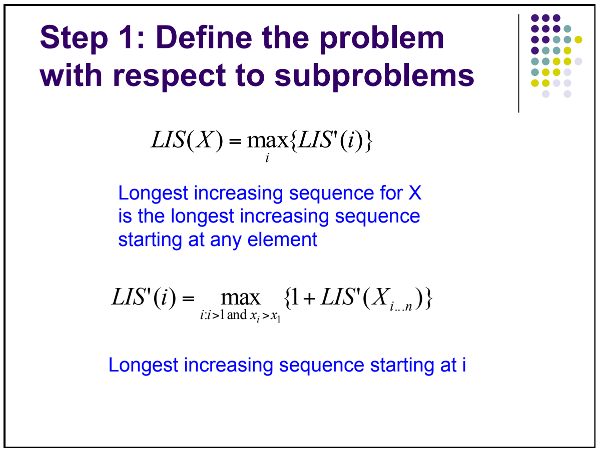


Step 1: Define the problem with respect to subproblems

$$LIS(X) = \max_i \{LIS'(i)\}$$

Longest increasing sequence for X is the longest increasing sequence starting at any element

And what is LIS’ defined as (recursively)?



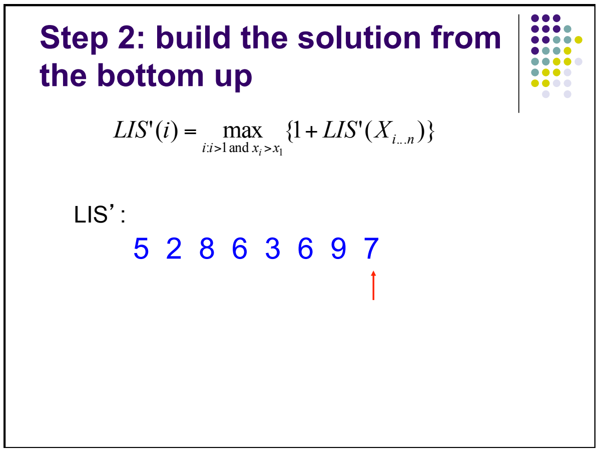
Step 1: Define the problem with respect to subproblems

$$LIS(X) = \max_i \{LIS'(i)\}$$

Longest increasing sequence for X is the longest increasing sequence starting at any element

$$LIS'(i) = \max_{i:i>1 \text{ and } x_i > x_1} \{1 + LIS'(X_{i...n})\}$$

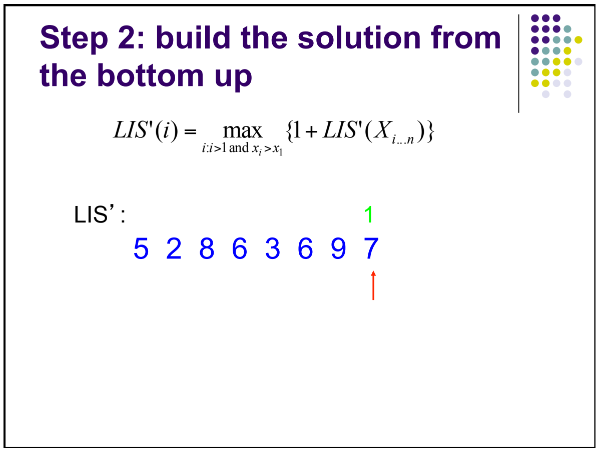
Longest increasing sequence starting at i



Step 2: build the solution from the bottom up

$$LIS'(i) = \max_{i:i>1 \text{ and } x_i > x_1} \{1 + LIS'(X_{i...n})\}$$

LIS’ :								
	5	2	8	6	3	6	9	7
								↑



Step 2: build the solution from the bottom up

$$LIS'(i) = \max_{i:i>1 \text{ and } x_i > x_1} \{1 + LIS'(X_{i...n})\}$$

LIS’ :								1
	5	2	8	6	3	6	9	7
								↑

Step 2: build the solution from the bottom up

$$LIS'(i) = \max_{i:i>1 \text{ and } x_i > x_1} \{1 + LIS'(X_{i...n})\}$$

LIS' :								1
	5	2	8	6	3	6	9	7
							↑	

Step 2: build the solution from the bottom up

$$LIS'(i) = \max_{i:i>1 \text{ and } x_i > x_1} \{1 + LIS'(X_{i...n})\}$$

LIS' :							1	1
	5	2	8	6	3	6	9	7
							↑	

Step 2: build the solution from the bottom up

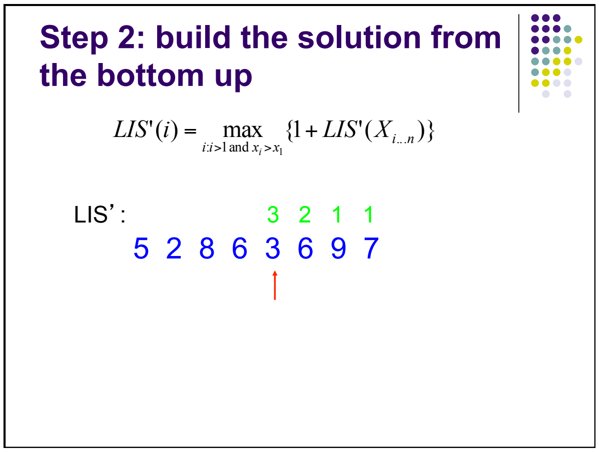
$$LIS'(i) = \max_{i:i>1 \text{ and } x_i > x_1} \{1 + LIS'(X_{i...n})\}$$

LIS' :							1	1
	5	2	8	6	3	6	9	7
						↑		

Step 2: build the solution from the bottom up

$$LIS'(i) = \max_{i:i>1 \text{ and } x_i > x_1} \{1 + LIS'(X_{i...n})\}$$

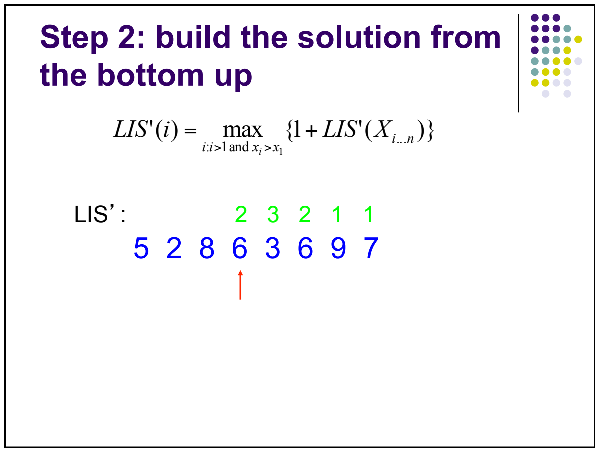
LIS' :						2	1	1
	5	2	8	6	3	6	9	7
						↑		



Step 2: build the solution from the bottom up

$$LIS'(i) = \max_{i: i>1 \text{ and } x_i > x_1} \{1 + LIS'(X_{i \ldots n})\}$$

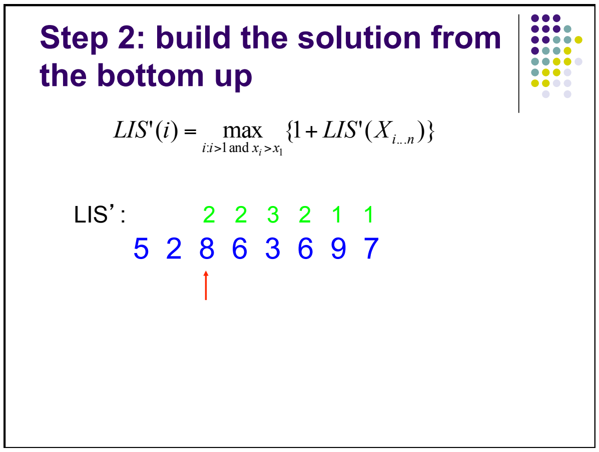
LIS’ :					3	2	1	1
	5	2	8	6	3	6	9	7
					↑			



Step 2: build the solution from the bottom up

$$LIS'(i) = \max_{i: i>1 \text{ and } x_i > x_1} \{1 + LIS'(X_{i \ldots n})\}$$

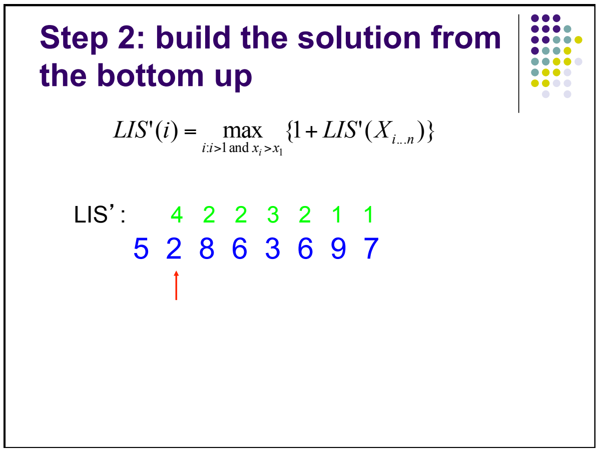
LIS’ :				2	3	2	1	1
	5	2	8	6	3	6	9	7
				↑				



Step 2: build the solution from the bottom up

$$LIS'(i) = \max_{i: i>1 \text{ and } x_i > x_1} \{1 + LIS'(X_{i \ldots n})\}$$

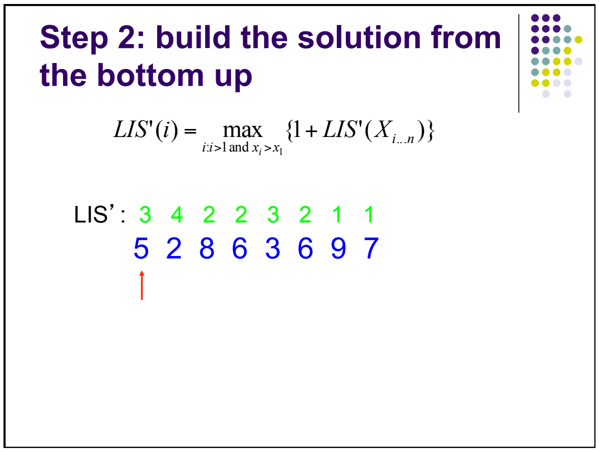
LIS’ :			2	2	3	2	1	1
	5	2	8	6	3	6	9	7
			↑					



Step 2: build the solution from the bottom up

$$LIS'(i) = \max_{i: i>1 \text{ and } x_i > x_1} \{1 + LIS'(X_{i \ldots n})\}$$

LIS’ :		4	2	2	3	2	1	1
	5	2	8	6	3	6	9	7
		↑						

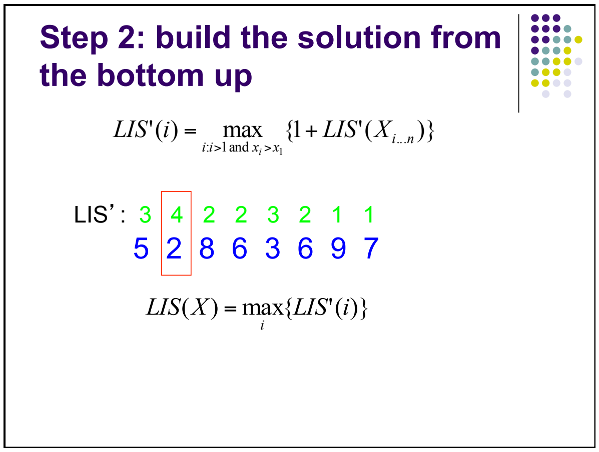


Step 2: build the solution from the bottom up

$$LIS'(i) = \max_{i:i>1 \text{ and } x_i > x_1} \{1 + LIS'(X_{i...n})\}$$

LIS': 3 4 2 2 3 2 1 1

5 2 8 6 3 6 9 7



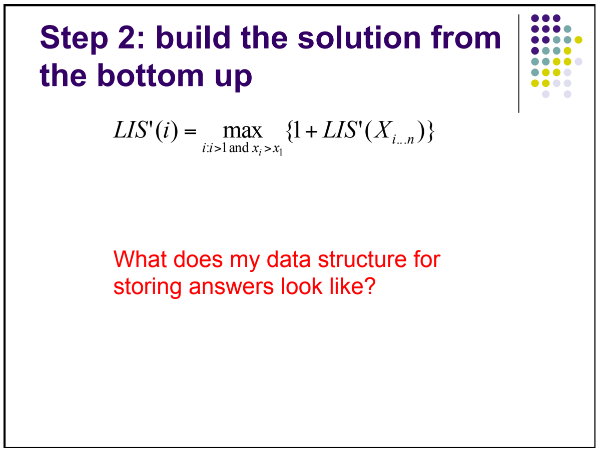
Step 2: build the solution from the bottom up

$$LIS'(i) = \max_{i:i>1 \text{ and } x_i > x_1} \{1 + LIS'(X_{i...n})\}$$

LIS': 3 4 2 2 3 2 1 1

5 2 8 6 3 6 9 7

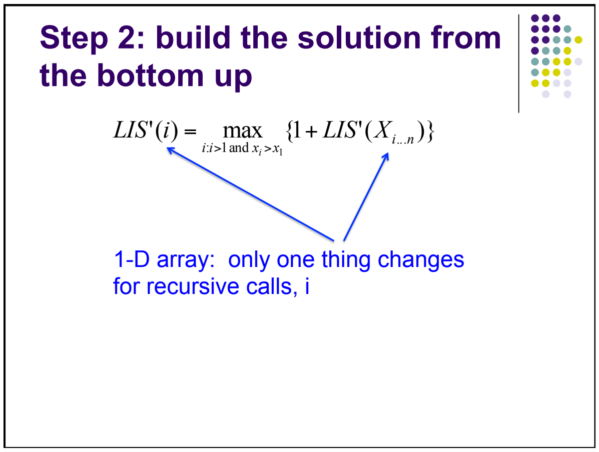
$$LIS(X) = \max_{i} \{LIS'(i)\}$$



Step 2: build the solution from the bottom up

$$LIS'(i) = \max_{i:i>1 \text{ and } x_i > x_1} \{1 + LIS'(X_{i...n})\}$$

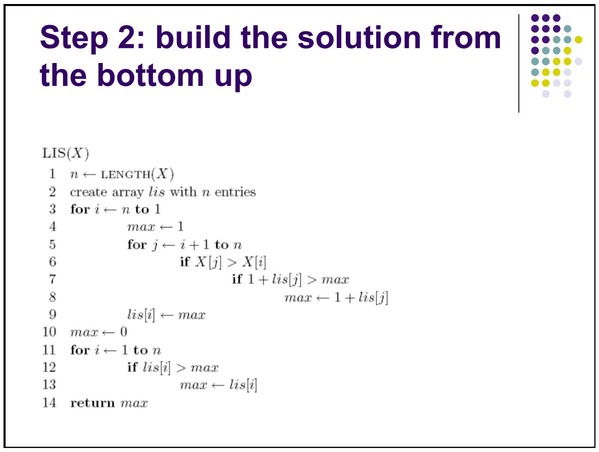
What does my data structure for storing answers look like?



Step 2: build the solution from the bottom up

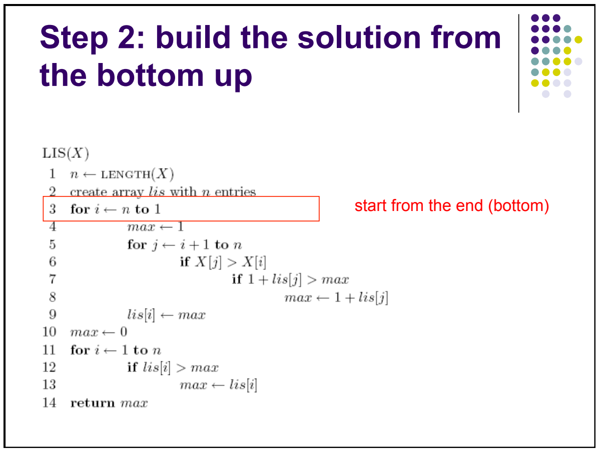
$$LIS'(i) = \max_{i:i>1 \text{ and } x_i > x_1} \{1 + LIS'(X_{i...n})\}$$

1-D array: only one thing changes for recursive calls, i



Step 2: build the solution from the bottom up

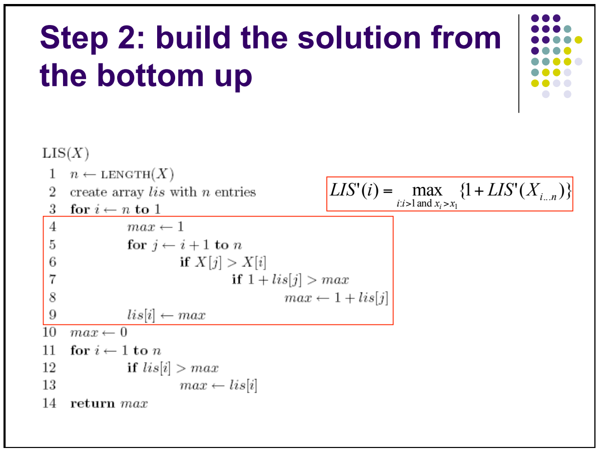
```
LIS($X$)
  $n \leftarrow$ LENGTH($X$)
  create array $lis$ with $n$ entries
  for $i \leftarrow n$ to 1
      $max \leftarrow 1$
      for $j \leftarrow i+1$ to $n$
          if $X[j] > X[i]$
              if $1 + lis[j] > max$
                  $max \leftarrow 1 + lis[j]$
      $lis[i] \leftarrow max$
  $max \leftarrow 0$
  for $i \leftarrow 1$ to $n$
      if $lis[i] > max$
          $max \leftarrow lis[i]$
  return $max$
```



Step 2: build the solution from the bottom up

```
LIS($X$)
  $n \leftarrow$ LENGTH($X$)
  create array $lis$ with $n$ entries
  for $i \leftarrow n$ to 1
      $max \leftarrow 1$
      for $j \leftarrow i+1$ to $n$
          if $X[j] > X[i]$
              if $1 + lis[j] > max$
                  $max \leftarrow 1 + lis[j]$
      $lis[i] \leftarrow max$
  $max \leftarrow 0$
  for $i \leftarrow 1$ to $n$
      if $lis[i] > max$
          $max \leftarrow lis[i]$
  return $max$
```

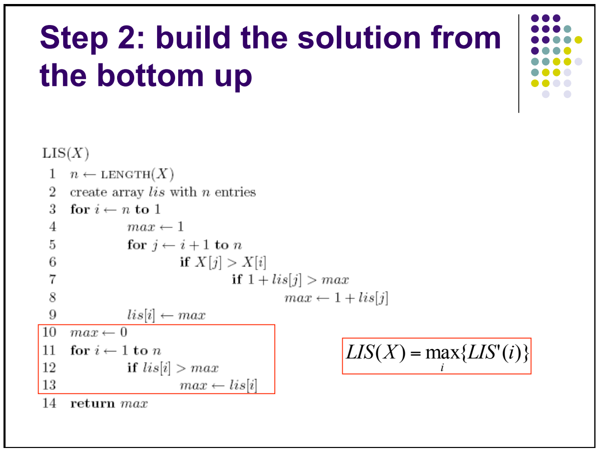
start from the end (bottom)



Step 2: build the solution from the bottom up

```
LIS($X$)
  $n \leftarrow$ LENGTH($X$)
  create array $lis$ with $n$ entries
  for $i \leftarrow n$ to 1
      $max \leftarrow 1$
      for $j \leftarrow i+1$ to $n$
          if $X[j] > X[i]$
              if $1 + lis[j] > max$
                  $max \leftarrow 1 + lis[j]$
      $lis[i] \leftarrow max$
  $max \leftarrow 0$
  for $i \leftarrow 1$ to $n$
      if $lis[i] > max$
          $max \leftarrow lis[i]$
  return $max$
```

$$\boxed{LIS'(i) = \max_{i:i>1 \text{ and } x_i > x_1} \{1 + LIS'(X_{i...n})\}}$$



Step 2: build the solution from the bottom up

```
LIS($X$)
  $n \leftarrow$ LENGTH($X$)
  create array $lis$ with $n$ entries
  for $i \leftarrow n$ to 1
      $max \leftarrow 1$
      for $j \leftarrow i+1$ to $n$
          if $X[j] > X[i]$
              if $1 + lis[j] > max$
                  $max \leftarrow 1 + lis[j]$
      $lis[i] \leftarrow max$
  $max \leftarrow 0$
  for $i \leftarrow 1$ to $n$
      if $lis[i] > max$
          $max \leftarrow lis[i]$
  return $max$
```

$$\boxed{LIS(X) = \max_{i}\{LIS'(i)\}}$$

Step 2: build the solution from the bottom up

```
LIS($X$)
$n \leftarrow$ LENGTH($X$)
create array $lis$ with $n$ entries
for $i \leftarrow n$ to 1
    $max \leftarrow 1$
    for $j \leftarrow i+1$ to $n$
        if $X[j] > X[i]$
            if $1 + lis[j] > max$
                $max \leftarrow 1 + lis[j]$
    $lis[i] \leftarrow max$
$max \leftarrow 0$
for $i \leftarrow 1$ to $n$
    if $lis[i] > max$
        $max \leftarrow lis[i]$
return $max$
```

initialization?

Running time?

```
LIS($X$)
$n \leftarrow$ LENGTH($X$)
create array $lis$ with $n$ entries
for $i \leftarrow n$ to 1
    $max \leftarrow 1$
    for $j \leftarrow i+1$ to $n$
        if $X[j] > X[i]$
            if $1 + lis[j] > max$
                $max \leftarrow 1 + lis[j]$
    $lis[i] \leftarrow max$
$max \leftarrow 0$
for $i \leftarrow 1$ to $n$
    if $lis[i] > max$
        $max \leftarrow lis[i]$
return $max$
```

$\Theta(n^2)$

Another solution

Can we use LCS to solve this problem?

5 2 8 6 3 6 9 7

LCS

2 3 5 6 6 7 8 9

Another solution

Can we use LCS to solve this problem?

5 2 8 6 3 6 9 7

LCS

2 3 5 6 6 7 8 9

Memoization

Sometimes it can be a challenge to write the function in a bottom-up fashion

Memoization:

- Write the recursive function top-down
- Alter the function to check if we've already calculated the value
- If so, use the pre-calculate value
- If not, do the recursive call(s)

Memoized fibonacci

```
Fibonacci(n)
if n = 1 or n = 2
        return 1
else
        return Fibonacci(n − 1) + Fibonacci(n − 2)
```

```
Fibonacci-Memoized(n)
fib[1] ← 1
fib[2] ← 1
for i ← 3 to n
        fib[i] ← ∞
return Fib-Lookup(n)
```

```
Fib-Lookup(n)
if fib[n] < ∞
        return fib[n]
x ← Fib-Lookup(n − 1) + Fib-Lookup(n − 2)
if x < fib[n]
        fib[n] ← x
return fib[n]
```

Memoized fibonacci

```
Fibonacci(n)
if n = 1 or n = 2
        return 1
else
        return Fibonacci(n − 1) + Fibonacci(n − 2)
```

```
Fibonacci-Memoized(n)
fib[1] ← 1
fib[2] ← 1
for i ← 3 to n
        fib[i] ← ∞
return Fib-Lookup(n)
```

```
Fib-Lookup(n)
if fib[n] < ∞
        return fib[n]
fib[n] ← Fib-Lookup(n − 1) + Fib-Lookup(n − 2)
return fib[n]
```

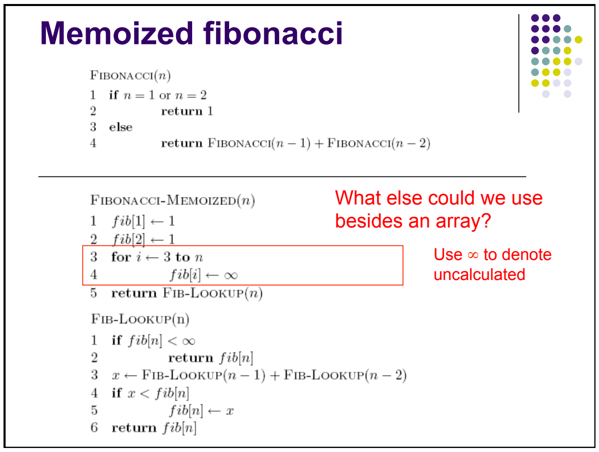
Memoized fibonacci

```
Fibonacci(n)
if n = 1 or n = 2
        return 1
else
        return Fibonacci(n − 1) + Fibonacci(n − 2)
```

```
Fibonacci-Memoized(n)
fib[1] ← 1
fib[2] ← 1
for i ← 3 to n
        fib[i] ← ∞
return Fib-Lookup(n)
```

Use ∞ to denote uncalculated

```
Fib-Lookup(n)
if fib[n] < ∞
        return fib[n]
x ← Fib-Lookup(n − 1) + Fib-Lookup(n − 2)
if x < fib[n]
        fib[n] ← x
return fib[n]
```



Memoized fibonacci

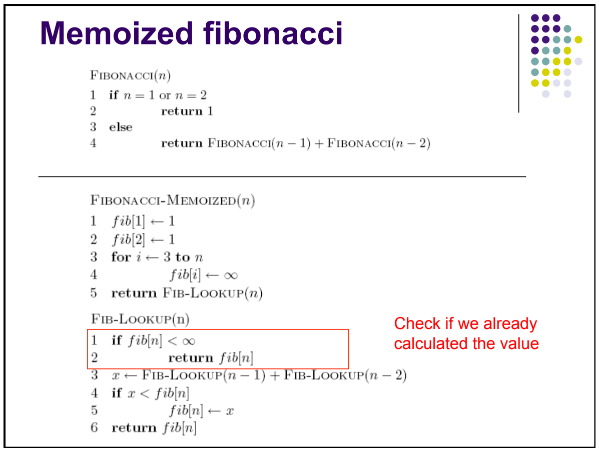
```
FIBONACCI($n$)
if $n = 1$ or $n = 2$
        return 1
else
        return FIBONACCI($n-1$) $+$ FIBONACCI($n-2$)
```

```
FIBONACCI-MEMOIZED($n$)
$fib[1] \leftarrow 1$
$fib[2] \leftarrow 1$
for $i \leftarrow 3$ to $n$
        $fib[i] \leftarrow \infty$
return FIB-LOOKUP($n$)
```

What else could we use besides an array?

Use ∞ to denote uncalculated

```
FIB-LOOKUP(n)
if $fib[n] < \infty$
        return $fib[n]$
$x \leftarrow$ FIB-LOOKUP($n-1$) $+$ FIB-LOOKUP($n-2$)
if $x < fib[n]$
        $fib[n] \leftarrow x$
return $fib[n]$
```



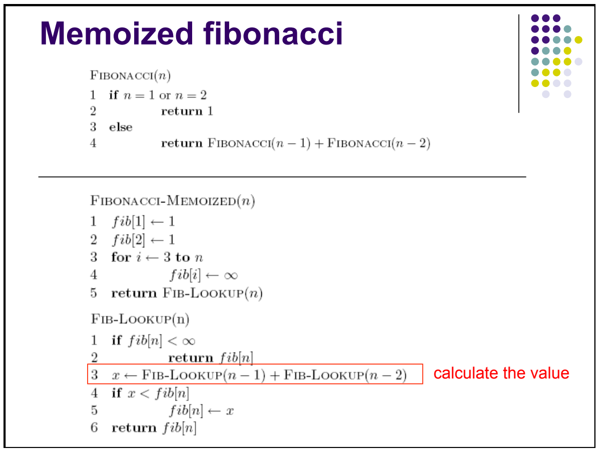
Memoized fibonacci

```
FIBONACCI($n$)
if $n = 1$ or $n = 2$
        return 1
else
        return FIBONACCI($n-1$) $+$ FIBONACCI($n-2$)
```

```
FIBONACCI-MEMOIZED($n$)
$fib[1] \leftarrow 1$
$fib[2] \leftarrow 1$
for $i \leftarrow 3$ to $n$
        $fib[i] \leftarrow \infty$
return FIB-LOOKUP($n$)
```

```
FIB-LOOKUP(n)
if $fib[n] < \infty$
        return $fib[n]$
$x \leftarrow$ FIB-LOOKUP($n-1$) $+$ FIB-LOOKUP($n-2$)
if $x < fib[n]$
        $fib[n] \leftarrow x$
return $fib[n]$
```

Check if we already calculated the value



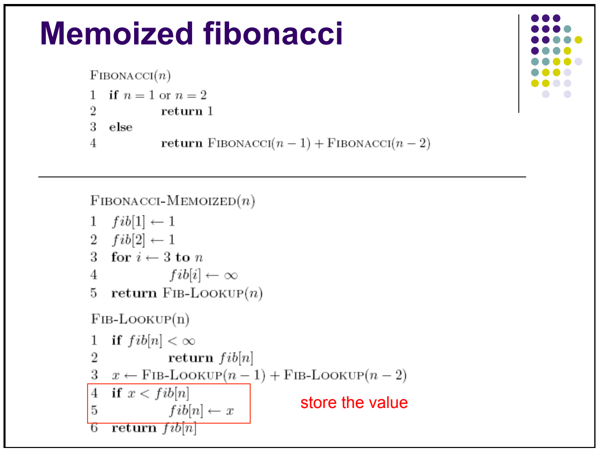
Memoized fibonacci

```
FIBONACCI($n$)
if $n = 1$ or $n = 2$
        return 1
else
        return FIBONACCI($n-1$) $+$ FIBONACCI($n-2$)
```

```
FIBONACCI-MEMOIZED($n$)
$fib[1] \leftarrow 1$
$fib[2] \leftarrow 1$
for $i \leftarrow 3$ to $n$
        $fib[i] \leftarrow \infty$
return FIB-LOOKUP($n$)
```

```
FIB-LOOKUP(n)
if $fib[n] < \infty$
        return $fib[n]$
$x \leftarrow$ FIB-LOOKUP($n-1$) $+$ FIB-LOOKUP($n-2$)
if $x < fib[n]$
        $fib[n] \leftarrow x$
return $fib[n]$
```

calculate the value



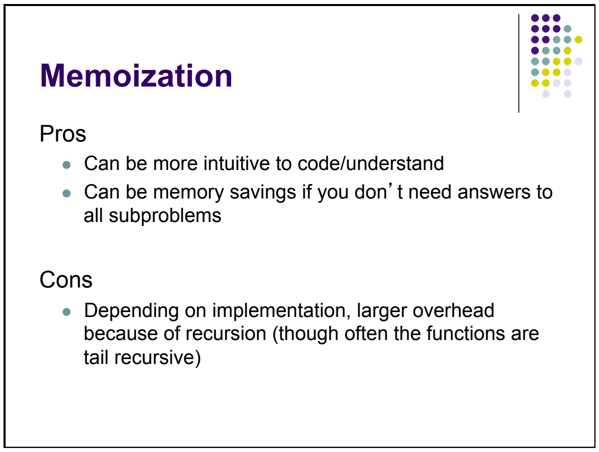
Memoized fibonacci

```
FIBONACCI($n$)
if $n = 1$ or $n = 2$
        return 1
else
        return FIBONACCI($n-1$) $+$ FIBONACCI($n-2$)
```

```
FIBONACCI-MEMOIZED($n$)
$fib[1] \leftarrow 1$
$fib[2] \leftarrow 1$
for $i \leftarrow 3$ to $n$
        $fib[i] \leftarrow \infty$
return FIB-LOOKUP($n$)
```

```
FIB-LOOKUP(n)
if $fib[n] < \infty$
        return $fib[n]$
$x \leftarrow$ FIB-LOOKUP($n-1$) $+$ FIB-LOOKUP($n-2$)
if $x < fib[n]$
        $fib[n] \leftarrow x$
return $fib[n]$
```

store the value



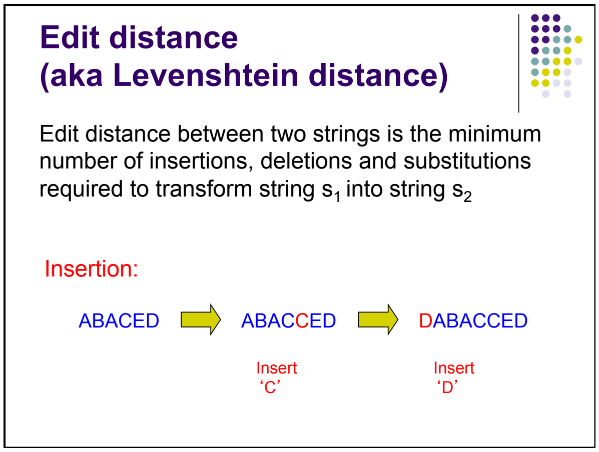
Memoization

Pros

- Can be more intuitive to code/understand
- Can be memory savings if you don' t need answers to all subproblems

Cons

- Depending on implementation, larger overhead because of recursion (though often the functions are tail recursive)



Edit distance (aka Levenshtein distance)

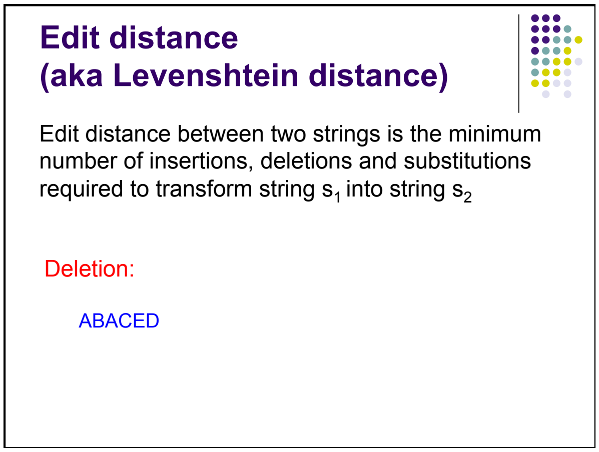
Edit distance between two strings is the minimum number of insertions, deletions and substitutions required to transform string s_1 into string s_2

Insertion:

ABACED → ABACCED → DABACCED

Insert 'C'

Insert 'D'

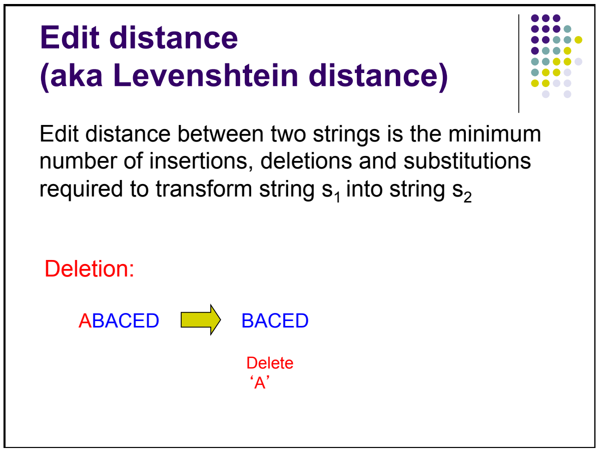


Edit distance (aka Levenshtein distance)

Edit distance between two strings is the minimum number of insertions, deletions and substitutions required to transform string s_1 into string s_2

Deletion:

ABACED



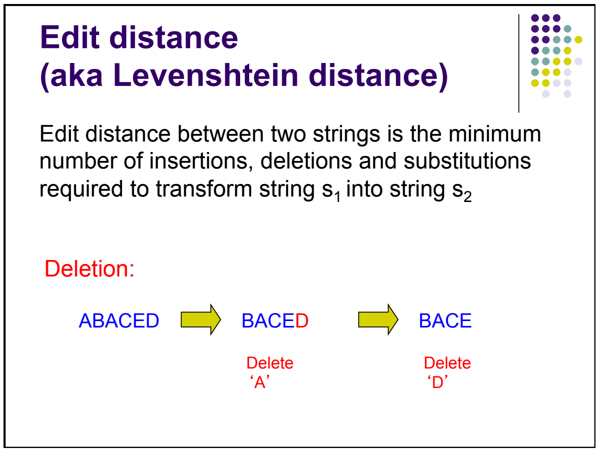
Edit distance (aka Levenshtein distance)

Edit distance between two strings is the minimum number of insertions, deletions and substitutions required to transform string s_1 into string s_2

Deletion:

ABACED → BACED

Delete 'A'



Edit distance (aka Levenshtein distance)

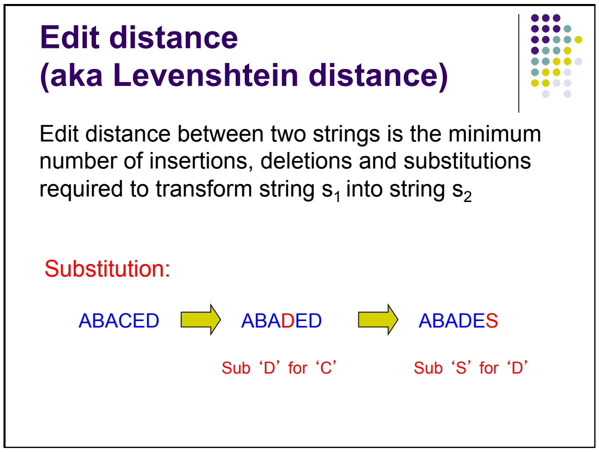
Edit distance between two strings is the minimum number of insertions, deletions and substitutions required to transform string s_1 into string s_2

Deletion:

ABACED → BACED → BACE

Delete 'A'

Delete 'D'



Edit distance (aka Levenshtein distance)

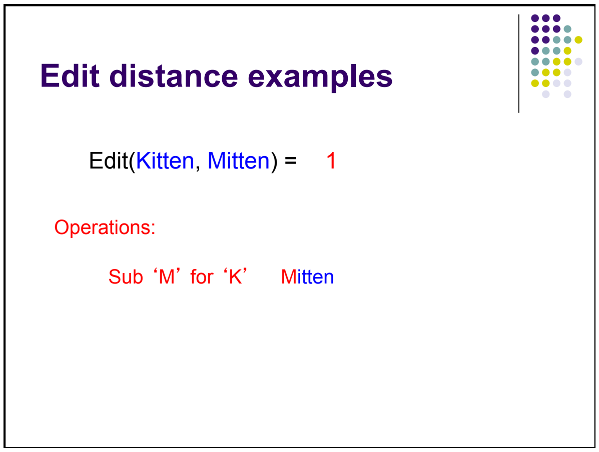
Edit distance between two strings is the minimum number of insertions, deletions and substitutions required to transform string s_1 into string s_2

Substitution:

ABACED → ABADED → ABADES

Sub 'D' for 'C'

Sub 'S' for 'D'

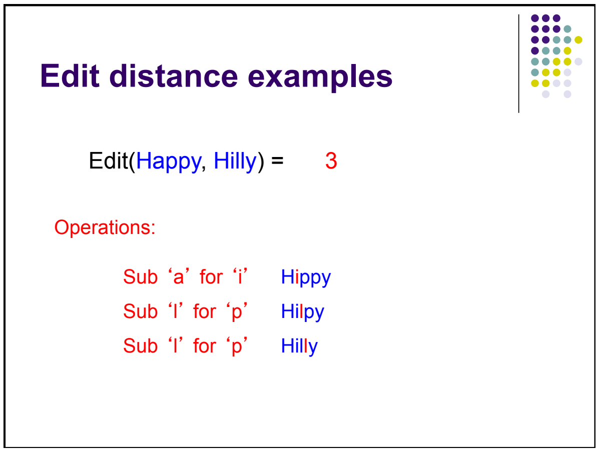


Edit distance examples

Edit(Kitten, Mitten) = 1

Operations:

Sub 'M' for 'K'	Mitten

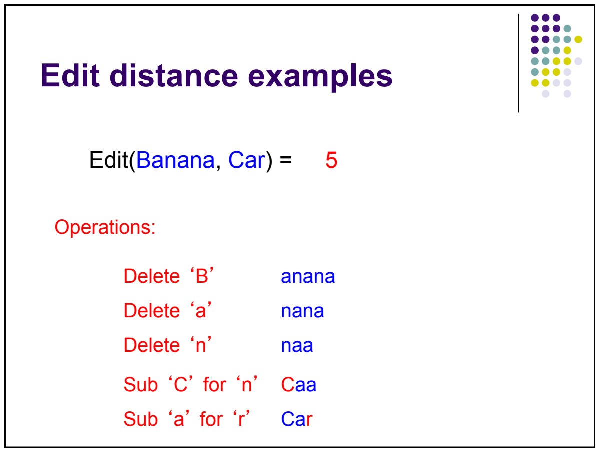


Edit distance examples

Edit(Happy, Hilly) = 3

Operations:

Sub 'a' for 'i'	Hippy
Sub 'l' for 'p'	Hilpy
Sub 'l' for 'p'	Hilly

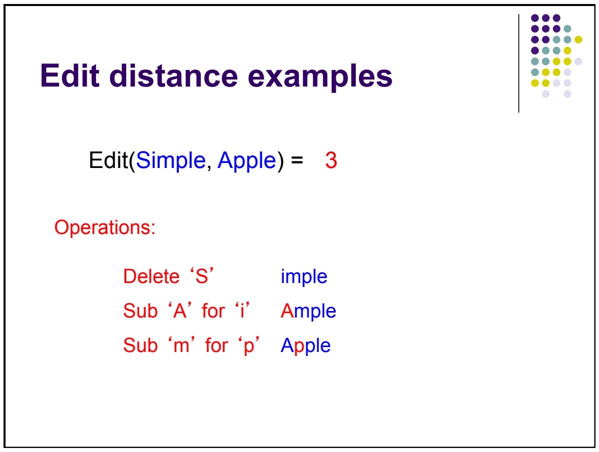


Edit distance examples

Edit(Banana, Car) = 5

Operations:

Delete ‘B’	anana
Delete ‘a’	nana
Delete ‘n’	naa
Sub ‘C’ for ‘n’	Caa
Sub ‘a’ for ‘r’	Car

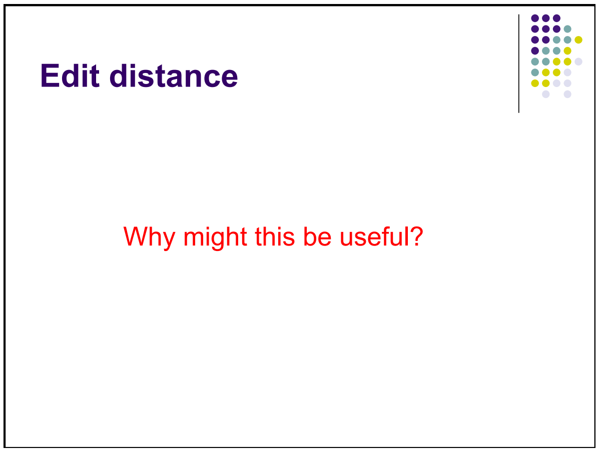


Edit distance examples

Edit(Simple, Apple) = 3

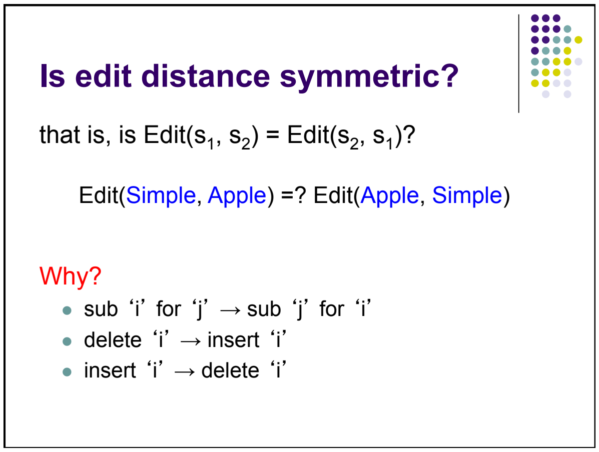
Operations:

Delete ‘S’	imple
Sub ‘A’ for ‘i’	Ample
Sub ‘m’ for ‘p’	Apple



Edit distance

Why might this be useful?



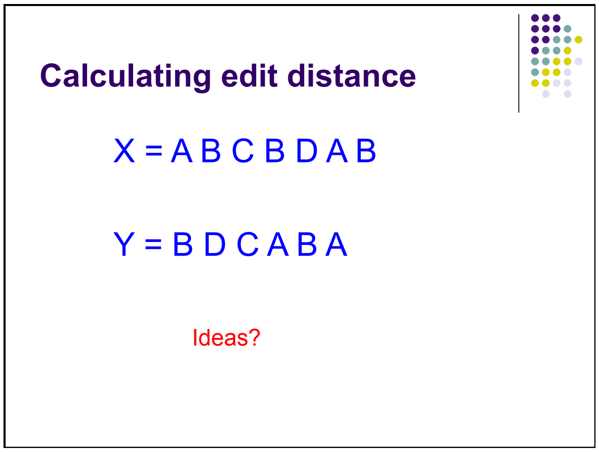
Is edit distance symmetric?

that is, is Edit(s_1, s_2) = Edit(s_2, s_1)?

Edit(Simple, Apple) =? Edit(Apple, Simple)

Why?

- sub ‘i’ for ‘j’ → sub ‘j’ for ‘i’
- delete ‘i’ → insert ‘i’
- insert ‘i’ → delete ‘i’

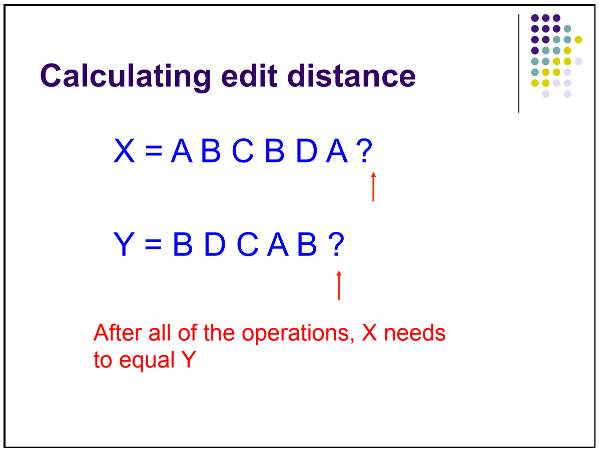


Calculating edit distance

X = A B C B D A B

Y = B D C A B A

Ideas?

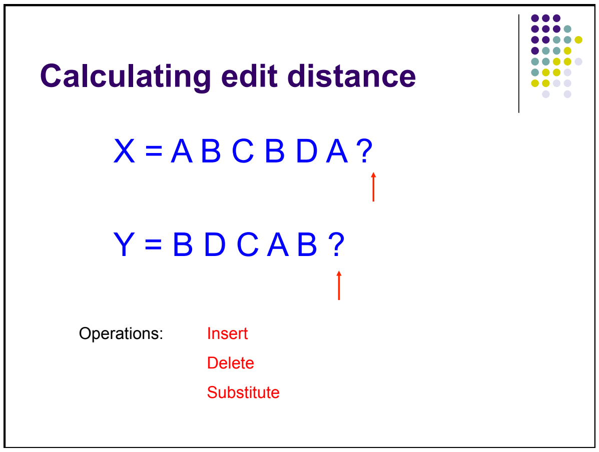


Calculating edit distance

X = A B C B D A ?

Y = B D C A B ?

After all of the operations, X needs to equal Y

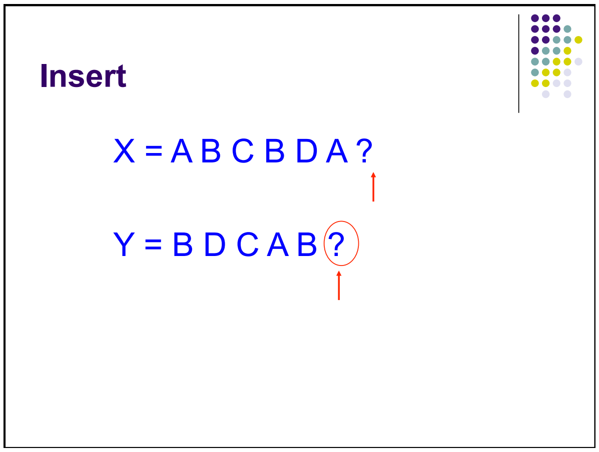


Calculating edit distance

X = A B C B D A ?

Y = B D C A B ?

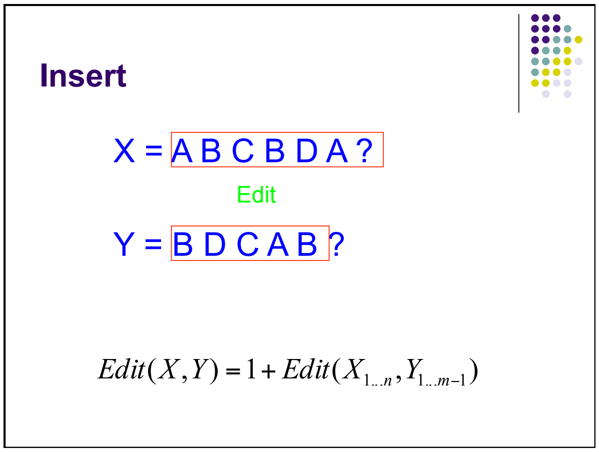
Operations: Insert
Delete
Substitute



Insert

X = A B C B D A ?

Y = B D C A B ?



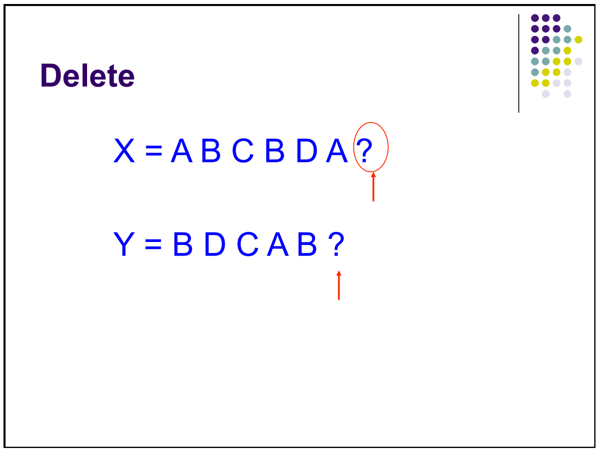
Insert

X = A B C B D A ?

Edit

Y = B D C A B ?

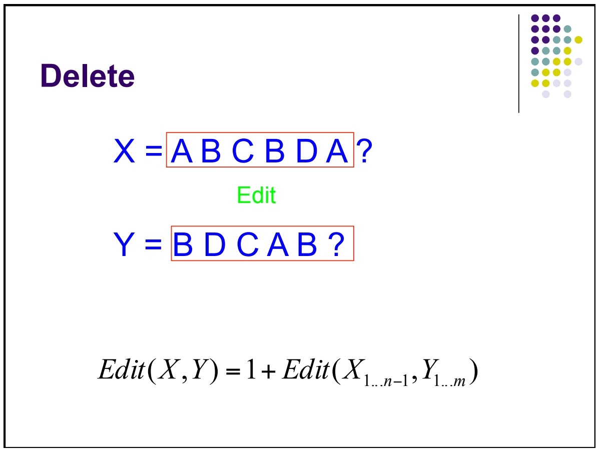
$$Edit(X,Y) = 1 + Edit(X_{1...n}, Y_{1...m-1})$$



Delete

X = A B C B D A ?

Y = B D C A B ?



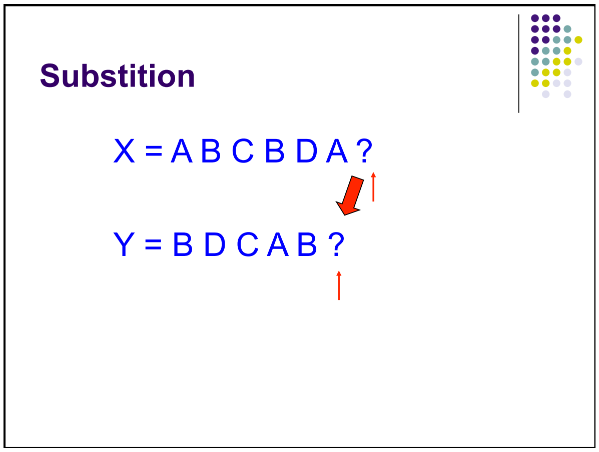
Delete

X = A B C B D A ?

Edit

Y = B D C A B ?

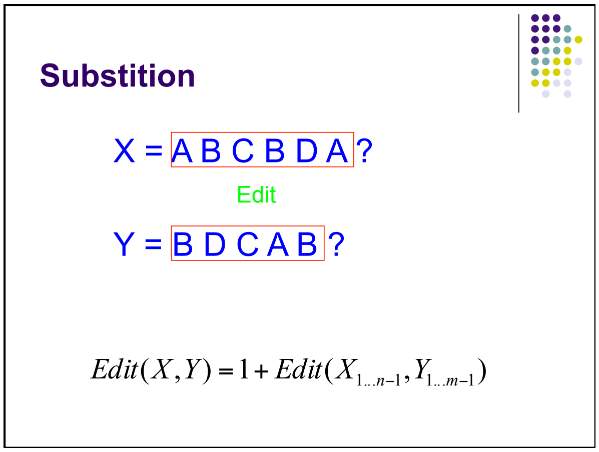
$$Edit(X,Y) = 1 + Edit(X_{1...n-1}, Y_{1...m})$$



Substition

X = A B C B D A ?

Y = B D C A B ?



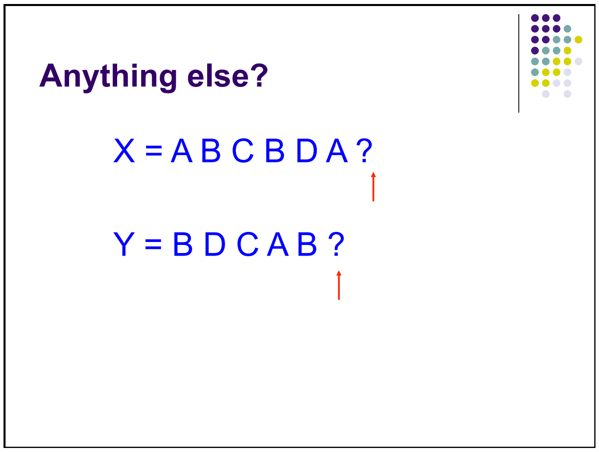
Substition

X = A B C B D A ?

Edit

Y = B D C A B ?

$$Edit(X,Y) = 1 + Edit(X_{1\ldots n-1}, Y_{1\ldots m-1})$$



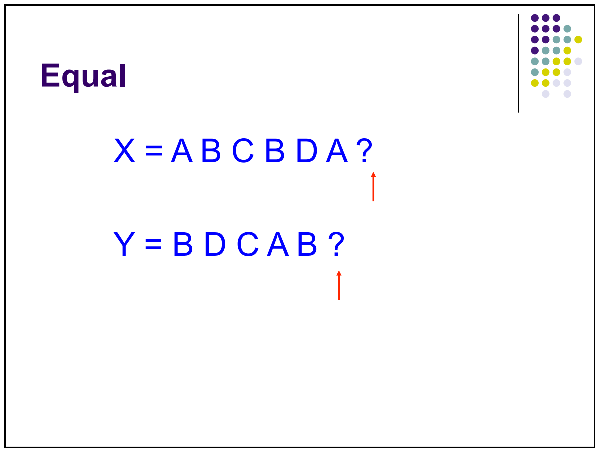
Anything else?

X = A B C B D A ?

↑

Y = B D C A B ?

↑



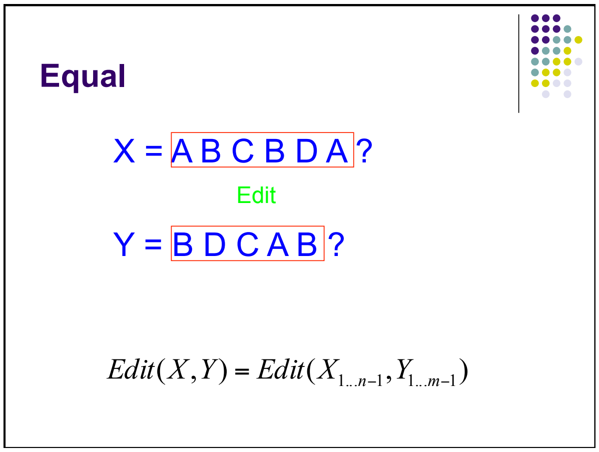
Equal

X = A B C B D A ?

↑

Y = B D C A B ?

↑



Equal

X = A B C B D A ?

Edit

Y = B D C A B ?

$$Edit(X,Y) = Edit(X_{1\ldots n-1}, Y_{1\ldots m-1})$$

Combining results

Insert: $Edit(X,Y) = 1 + Edit(X_{1\ldots n}, Y_{1\ldots m-1})$

Delete: $Edit(X,Y) = 1 + Edit(X_{1\ldots n-1}, Y_{1\ldots m})$

Substitute: $Edit(X,Y) = 1 + Edit(X_{1\ldots n-1}, Y_{1\ldots m-1})$

Equal: $Edit(X,Y) = Edit(X_{1\ldots n-1}, Y_{1\ldots m-1})$

Combining results

$$Edit(X,Y) = \min \begin{cases} 1 + Edit(X_{1\ldots n}, Y_{1\ldots m-1}) & \text{insertion} \\ 1 + Edit(X_{1\ldots n-1}, Y_{1\ldots m}) & \text{deletion} \\ Diff(x_n, y_m) + Edit(X_{1\ldots n-1}, Y_{1\ldots m-1}) & \text{equal/substitution} \end{cases}$$

```
$\textsc{Edit}(X, Y)$
$m \leftarrow length[X]$
$n \leftarrow length[Y]$
for $i \leftarrow 0$ to $m$
        $d[i, 0] \leftarrow i$
for $j \leftarrow 0$ to $n$
        $d[0, j] \leftarrow j$
for $i \leftarrow 1$ to $m$
        for $j \leftarrow 1$ to $n$
                $d[i, j] = min(1 + d[i-1, j],$
                        $1 + d[i, j-1],$
                        $\textsc{diff}(x_i, y_j) + d[i-1, j-1])$
return $d[m, n]$
```

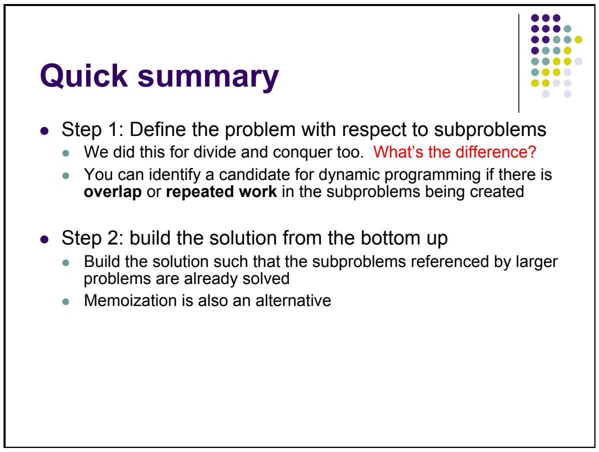
Running time

$\Theta(nm)$

```
$\textsc{Edit}(X, Y)$
$m \leftarrow length[X]$
$n \leftarrow length[Y]$
for $i \leftarrow 0$ to $m$
        $d[i, 0] \leftarrow i$
for $j \leftarrow 0$ to $n$
        $d[0, j] \leftarrow j$
for $i \leftarrow 1$ to $m$
        for $j \leftarrow 1$ to $n$
                $d[i, j] = min(1 + d[i-1, j],$
                        $1 + d[i, j-1],$
                        $\textsc{diff}(x_i, y_j) + d[i-1, j-1])$
return $d[m, n]$
```

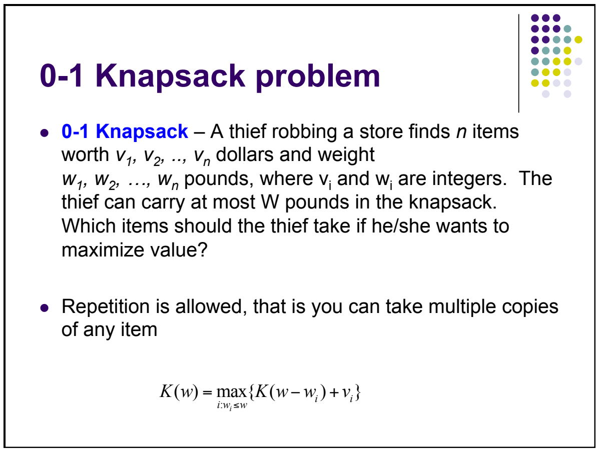
Variants

- Only include insertions and deletions
  - What does this do to substitutions?
- Include swaps, i.e. swapping two adjacent characters counts as one edit
- Weight insertion, deletion and substitution differently
- Weight **specific** character insertion, deletion and substitutions differently
- Length normalize the edit distance



Quick summary

- Step 1: Define the problem with respect to subproblems
  - We did this for divide and conquer too. What's the difference?
  - You can identify a candidate for dynamic programming if there is **overlap** or **repeated work** in the subproblems being created
- Step 2: build the solution from the bottom up
  - Build the solution such that the subproblems referenced by larger problems are already solved
  - Memoization is also an alternative



0-1 Knapsack problem

- **0-1 Knapsack** – A thief robbing a store finds *n* items worth $v_1, v_2, .., v_n$ dollars and weight $w_1, w_2, …, w_n$ pounds, where v_i and w_i are integers. The thief can carry at most W pounds in the knapsack. Which items should the thief take if he/she wants to maximize value?

- Repetition is allowed, that is you can take multiple copies of any item

$$K(w) = \max_{i:w_i \le w}\{K(w - w_i) + v_i\}$$