Lecture 14 - Use-It-or-Lose-It

Eric A. Autry

Last time: Sorting Algorithms

This time: Use-It-or-Lose-It and Dynamic Programming

Next time: DFS/BFS

HW 5 due today.

Project 1 due next Friday, the 2nd.

HW 6 assigned today, due next Thursday, the 1st.
(very short - only one problem)

Sorting Algorithms

- Selection Sort, $O(n^2)$
- Insertion Sort, $O(n^2)$ worst/average, $O(n)$ best
- Bubble Sort, $O(n^2)$ worst/average, $O(n)$ best
- Merge Sort, $O(n \log n)$
- Quick Sort, $O(n \log n)$ best/average, $O(n^2)$ worst
- Heap Sort, $O(n \log n)$

In-Place Algorithms

We say that an algorithm is an **in-place** algorithm if all of the operations are performed directly in the input data,

i.e., no extra storage space is required.

- Ex: Bubble Sort swaps elements of A and never has to use any outside storage arrays.
- Note: Selection Sort, Insertion Sort, Bubble Sort, and Quick Sort can all be implemented in-place.
- Note: Merge Sort and Heap Sort **cannot** be implemented in-place and still maintain their runtimes.
- If we made Merge Sort in-place, the merging step would require $O(n^2)$ operations, giving an overall $O(n^2)$ runtime.
  - Well, technically Merge Sort can be implemented in-place and maintain its runtime, but this requires a very complicated algorithm.
- There is a trade-off between runtime and necessary storage space!

Project 1 - Pair Programming

You are allowed to work in pairs for this project. If you elect to work with a partner:

- You should submit only one version of your final code and report. Have one partner upload the code and report on their Sakai site. Make sure that both partner's names are clearly indicated at the top of both the code and the report.
- When work is being done on this project, both partners are required to be physically present at the machine in question, with one partner typing ('driving') and the other partner watching ('navigating').
- You should split your time equally between driving and navigating to ensure that both partners spend time coding.
- You are allowed to discuss this project with other students in the class, but the code you submit must be written by you and your partner alone.

Project 1 - Commenting Code

The guidelines for 'style points':

- Your program file should have a header stating the name of the program, the author(s), and the date.
- All functions need a comment stating: the name of the function, what the function does, what the inputs are, and what the outputs are.
- Every major block of code should have a comment explaining what the block of code is doing. This need not be every line, and it is left to your discretion to determine how frequently you place these comments. However, you should have enough comments to clearly explain the behavior of your code.
- Please limit yourself to 80 characters in a line of code. In python, you can use the symbol \ to indicate a line break/continuation.

Project 1 - Python 3.7.1 and IDE

We will be using Python 3.7.1 to grade your projects.

If you create your code in an older version of python, **it is your responsibility** to ensure that your code is compatible with version 3.7.1.

If we run your code in python 3.7.1 and it throws errors, you will lose points.

'But it worked in my version of python' is **not** an acceptable excuse.

- To edit your python code, you may use whatever editor you wish (emacs, vim, gedit, python IDLE, etc).
- PyCharm has a nice tutorial included with it, which is nice for students who have not used python before. But it is not required.

Longest Common Subsequence (LCS)

NOTE: the example for this problem was incorrect for the morning section!

Problem definition:

- We will consider two strings `S1` and `S2`.
- We want to find the longest possible substring that can be made out of elements **included** in order in both strings.

Ex:

```
S1 = ' s t p l a m'
       0 1 2 3 4 5 6
S2 = ' z s p q r a m'
```

Longest common subsequence: `'spam'`

Application: genetics

Longest Common Subsequence (LCS)

- Note on pseudo-code: we will index the same as in Python.
- Index starting from 0.
- `S[k:j]` is the subtring from `k` (inclusive) to `j` (exclusive).
- Ex: `S='abcde'`, then `S[1:3] = 'bc'`

```
def LCS(S1[0:i], S2[0:j]):
    # Returns the length of the LCS for the
    # substring of S1 from index 0 to i-1 and
    # the substring of S2 from index 0 to j-1.
```

Longest Common Subsequence (LCS)

Idea: compare the last letter in each string and use recursion.

Base Case: what if one of the strings is empty?

If the strings aren't empty, what are **all** of the possible situations?

- The last letters are the same, and **must** be part of the LCS.
  Specific example 1: `S1 = spam` and `S2 = plum` $\Rightarrow$ `pm`
- They are not the same, but the 'm' in 'spam' **IS** in the LCS. (Use-it)
  Specific example 2: `S1 = spam` and `S2 = same` $\Rightarrow$ `sam`
- They are not the same, but the 'm' in 'spam' **IS NOT** in the LCS. (Lose-it)
  Specific example 3: `S1 = spam` and `S2 = spot` $\Rightarrow$ `sp`

We will consider **all** of these possibilities, and keep whichever gives us the largest LCS!

Longest Common Subsequence (LCS)

```
def LCS(S1[0:i], S2[0:j]):
    # Returns the length of the LCS for the
    # substring of S1 from index 0 to i-1 and
    # the substring of S2 from index 0 to j-1.

# Base Case
if (i == 0) or (j == 0):
    return 0

# They match so are part of the LCS!
elif S1[i-1] == S2[j-1]:
    return 1 + LCS(S1[0:i-1], S2[0:j-1])

else:
    # They don't match, so one must go!
    A = LCS(S1[0:i], S2[0:j-1]) # Use-it (keep S1)
    B = LCS(S1[0:i-1], S2[0:j]) # Lose-it (cut S1)
    return max(A,B) # Return whichever was better.
```

Longest Common Subsequence (LCS)

What is the runtime?

Worst case: nothing matches and we have maximum number of recursions (2 at each step instead of 1 if we had a match).

Recurrence relation for the worst case:

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

How do we analyze this with both n and m? Easier to consider:

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

Let's unroll this recurrence relation:

$$= \Big(T(n,n-2) + T(n-1,n-1)\Big) + \Big(T(n-1,n-1) + T(n-2,n)\Big)$$

$$\geq\ 2T(n-1,n-1)\ \geq\ 2^2T(n-2,n-2)\ \geq \cdots \geq\ 2^nT(1,1)\ \in O(2^n).$$

Brief Return to Fibonacci

Think back to our Fibonacci function:

```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n-1) + fib(n-2)
```

The problem: we had a lot of repeated recursive calls.

Idea: store values in an array as we compute them

```
F = [fib(1), fib(2), fib(3), fib(4), ..., fib(n)]
```

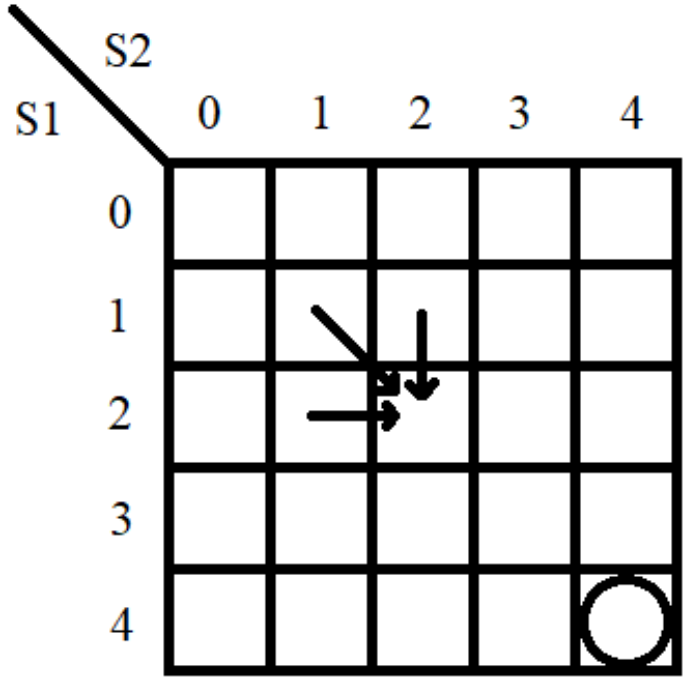
How to compute `fib(k)`? Just add `F[k-1]+F[k-2]`!

This approach is called **Dynamic Programming**, where we store the solutions in a 'DP table' and use that table to iteratively build up the solution from bottom-up.

Longest Common Subsequence (LCS)

For LCS, how many different recursive calls are made? $n \times m$

So store the results in an $n \times m$ DP table.



- Fill the table from top left to bottom right.
- Start with first row/column.
- Then fill in diagonals (or 2nd row/col, then 3rd row/col, etc).
- $O(n \cdot m)$

Ex: `LCS('spam', 'plum')`

Longest Common Subsequence (LCS)

What are the possible recursive calls from `LCS('spam', 'plum')`?

`('','')`	`('','p')`	`('','pl')`	`('','plu')`	`('','plum')`
`('s','')`	`('s','p')`	`('s','pl')`	`('s','plu')`	`('s','plum')`
`('sp','')`	`('sp','p')`	`('sp','pl')`	`('sp','plu')`	`('sp','plum')`
`('spa','')`	`('spa','p')`	`('spa','pl')`	`('spa','plu')`	`('spa','plum')`
`('spam','')`	`('spam','p')`	`('spam','pl')`	`('spam','plu')`	`('spam','plum')`

As we did with the Fibonacci numbers, we store the results of these calls in our DP table!

For each value, we use the recursive code to determine what it should be. But instead of actually making the different recursive calls, we simply look up their values that are **already in our table**.

Longest Common Subsequence (LCS)

(‘’,‘’)	(‘’,‘p’)	(‘’,‘pl’)	(‘’,‘plu’)	(‘’,‘plum’)
(‘s’,‘’)	(‘s’,‘p’)	(‘s’,‘pl’)	(‘s’,‘plu’)	(‘s’,‘plum’)
(‘sp’,‘’)	(‘sp’,‘p’)	(‘sp’,‘pl’)	(‘sp’,‘plu’)	(‘sp’,‘plum’)
(‘spa’,‘’)	(‘spa’,‘p’)	(‘spa’,‘pl’)	(‘spa’,‘plu’)	(‘spa’,‘plum’)
(‘spam’,‘’)	(‘spam’,‘p’)	(‘spam’,‘pl’)	(‘spam’,‘plu’)	(‘spam’,‘plum’)

Ex: what is the value of the call `LCS('sp','plu')`?

```
else:
    # They don't match, so one must go!
    A = LCS(S1[0:i], S2[0:j-1]) # Use-it (keep S1)
    B = LCS(S1[0:i-1], S2[0:j]) # Lose-it (cut S1)
    return max(A,B) # Return whichever was better.
```

So we take the maximum of `LCS('sp','pl')` and `LCS('s','plu')`, which are already filled in!

Longest Common Subsequence (LCS)

(‘’,‘’)	(‘’,‘p’)	(‘’,‘pl’)	(‘’,‘plu’)	(‘’,‘plum’)
(‘s’,‘’)	(‘s’,‘p’)	(‘s’,‘pl’)	(‘s’,‘plu’)	(‘s’,‘plum’)
(‘sp’,‘’)	(‘sp’,‘p’)	(‘sp’,‘pl’)	(‘sp’,‘plu’)	(‘sp’,‘plum’)
(‘spa’,‘’)	(‘spa’,‘p’)	(‘spa’,‘pl’)	(‘spa’,‘plu’)	(‘spa’,‘plum’)
(‘spam’,‘’)	(‘spam’,‘p’)	(‘spam’,‘pl’)	(‘spam’,‘plu’)	(‘spam’,‘plum’)

Ex: what is the value of the call `LCS('sp','p')`?

```
# They match so are part of the LCS!
elif S1[i-1] == S2[j-1]:
    return 1 + LCS(S1[0:i-1], S2[0:j-1])
```

So we return 1 plus the value of `LCS('s','')`, which we already filled in!

Longest Common Subsequence (LCS)

```
('','')      ('','p')      ('','pl')      ('','plu')      ('','plum')
('s','')     ('s','p')     ('s','pl')     ('s','plu')     ('s','plum')
('sp','')    ('sp','p')    ('sp','pl')    ('sp','plu')    ('sp','plum')
('spa','')   ('spa','p')   ('spa','pl')   ('spa','plu')   ('spa','plum')
('spam','')  ('spam','p')  ('spam','pl')  ('spam','plu')  ('spam','plum')
```

For this problem, the resulting table is:

```
    ''p l u m
   ----------
'' |0 0 0 0 0
s  |0 0 0 0 0
p  |0 1 1 1 1
a  |0 1 1 1 1
m  |0 1 1 1 2
```

Longest Common Subsequence (LCS)

Not that we have our table, how can we reconstruct the actual solution (i.e., how can we actually report what the LCS is)?

When we are filling in our table, we will keep track of where each entry came from: L if we took the value from the left, U if we took the value from above, and D if we took the value from the diagonal.

So our solution now looks like:

	``	p	l	u	m
``	0	0	0	0	0
s	0	0	0	0	0
p	0	1	1	1	1
a	0	1	1	1	1
m	0	1	1	1	2

	``	p	l	u	m
``	0	0	0	0	0
s	0	L	L	L	L
p	0	D	L	L	L
a	0	U	L	L	L
m	0	U	L	L	D

Longest Common Subsequence (LCS)

We will now start at the bottom right corner, and work our way backwards through the table to reconstruct the solution.

	`''	p	l	u	m
`''	0	0	0	0	0
s	0	0	0	0	0
p	0	1	1	1	1
a	0	1	1	1	1
m	0	1	1	1	2

	`''	p	l	u	m
`''	0	0	0	0	0
s	0	L	L	L	L
p	0	D	L	L	L
a	0	U	L	L	L
m	0	U	L	L	D

- If we see a 0, then we are in a base case and no letters left.
- If we see an L, then the letter from this column was not used.
- If we see a U, then the letter from this row was not used.
- If we see a D **and** we **did** add one, then we used the letter from this column and this row.

Longest Common Subsequence (LCS)

	''	p	l	u	m
''	0	0	0	0	0
s	0	0	0	0	0
p	0	1	1	1	1
a	0	1	1	1	1
m	0	1	1	1	2

	''	p	l	u	m
''	0	0	0	0	0
s	0	L	L	L	L
p	0	D	L	L	L
a	0	U	L	L	L
m	0	U	L	L	D

- Start at bottom right. We see D and $+1$, so `'m'` is in LCS. Move diagonally to the upper-left.
- We see an L. The `'u'` was not in LCS. Move left.
- We see an L. The `'l'` was not in LCS. Move left.
- We see a U. The `'a'` was not in LCS. Move up.
- We see a D and $+1$, so `'p'` is in LCS. Move diagonally to the upper-left.
- We see a 0. Nothing else in the LCS.

So the LCS was `'pm'` as expected!

Longest Common Subsequence (LCS)

Note that there may be more than one way to fill in the table. What happens if there was a tie between several of the cases?

For example, we could have filled out the table as:

	''	p	l	u	m
''	0	0	0	0	0
s	0	0	0	0	0
p	0	1	1	1	1
a	0	1	1	1	1
m	0	1	1	1	2

	''	p	l	u	m
''	0	0	0	0	0
s	0	U	U	U	U
p	0	D	L	L	L
a	0	U	U	U	U
m	0	U	U	U	D

When there are ties, the reconstructed solution *may* change, but the *optimality* will not. There might be multiple solutions to the problem, but each solution will have the same length.

Because ties don't affect the optimality of the answer, just pick a rule for tie-breaking.

Edit Distance (ED)

Objective: turn the string `'spam'` into the string `'slime'` using as few insertions, deletions, and substitutions as possible.

```
'spam' ->    (insert 'e')     -> 'spame'
       -> (sub 'i' for 'a') -> 'spime'
       -> (sub 'l' for 'p') -> 'slime'
```

How about `'libate'` to `'flub'`?

```
'libate' ->    (insert 'f')     -> 'flibate'
         ->    (delete 'e')     -> 'flibat'
         ->    (delete 't')     -> 'fliba'
         ->    (delete 'a')     -> 'flib'
         -> (sub 'u' for 'i') -> 'flub'
```

Application: genetics

Edit Distance (ED)

Idea: compare the last letter in each string and use recursion.

Base Case: what if one of the strings is empty?

If the strings aren't empty, what are **all** of the possible situations?

- The last letters are the same, so no editing necessary. Move on to the next letter! (Use-it)
  Specific example 1: `S1 = aaab` and `S2 = baab`
- They are not the same, and it's best to delete. (Lose-it #1)
  Specific example 2: `S1 = aaab` and `S2 = aaa`
- They are not the same, and it's best to insert. (Lose-it #2)
  Specific example 3: `S1 = aaa` and `S2 = aaab`
- They are not the same, and it's best to sub. (Lose-it #3)
  Specific example 4: `S1 = aaab` and `S2 = aaaa`

We will consider **all** of these possibilities, and keep whichever gives us the smallest ED!

Edit Distance (ED)

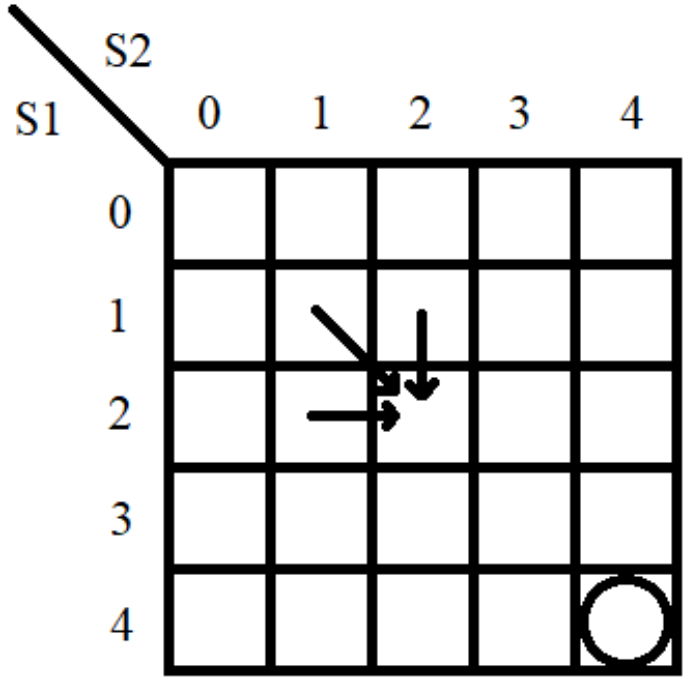
```
def ED(S1[0:i], S2[0:j]):
    # Returns the ED between S1 and S2.

    # Base Case (one of them is empty).
    if (i == 0) or (j == 0):
        return max(i, j)

    # Use It (they match so no edit needed).
    elif S1[i-1] == S2[j-1]:
        return ED(S1[0:i-1], S2[0:j-1])

    # Lose It (they didn't match so edit).
    else:
        A = ED(S1[0:i-1], S2[0:j]) # Delete
        B = ED(S1[0:i], S2[0:j-1]) # Insert
        C = ED(S1[0:i-1], S2[0:j-1]) # Sub
        return 1 + min(A, B, C)
```

- Note for the 'Sub' case, they now match. So no more editing necessary and we can move on to the next letter!

Edit Distance (ED)

For ED, how many different recursive calls are made? $n \times m$

So store the results in an $n \times m$ DP table.



- Fill the table from top left to bottom right.
- Start with first row/column.
- Then fill in diagonals (or 2nd row/col, then 3rd row/col, etc).
- $O(n \cdot m)$

Ex: `ED('spam', 'pims')`

Edit Distance (ED)

Ex: `ED('spam', 'pims')`

One possible way to fill in the table is:

	''	p	i	m	s
''	0	1	2	3	4
s	1	1	2	3	3
p	2	1	2	3	4
a	3	2	2	3	4
m	4	3	3	2	3

	''	p	i	m	s
''	0	1	2	3	4
s	1	D	L	L	D
p	2	D	L	L	L
a	3	U	D	L	L
m	4	U	U	D	L

The rules for reconstructing the solution are:

- Top row: insert the remaining letters of `'pims'`.
- First column: delete the preceding letters of `'spam'`.
- L: insert the letter for this column
- U: delete the letter for this row
- D (+1): change letter for this row to the one for this column
- D (+0): match, no edit required

Edit Distance (ED)

Ex: `ED('spam', 'pims')`

	''	p	i	m	s
''	0	1	2	3	4
s	1	1	2	3	3
p	2	1	2	3	4
a	3	2	2	3	4
m	4	3	3	2	3

	''	p	i	m	s
''	0	1	2	3	4
s	1	D	L	L	D
p	2	D	L	L	L
a	3	U	D	L	L
m	4	U	U	D	L

- Start at bottom right. See L, insert `'s'` at the end. Move left.
- See D $(+0)$, match the `'m'`. Move diagonally.
- See D $(+1)$, change `'a'` into `'i'`. Move diagonally.
- See D $(+0)$, match the `'p'`. Move diagonally.
- See 1, base case in first column: remove the leading `'s'`.

Edit Distance (ED)

Ex: `ED('spam', 'pims')`

- Start at bottom right. See L, insert `'s'` at the end. Move left.
- See D $(+0)$, match the `'m'`. Move diagonally.
- See D $(+1)$, change `'a'` into `'i'`. Move diagonally.
- See D $(+0)$, match the `'p'`. Move diagonally.
- See 1, base case in first column: remove the leading `'s'`.

Working backwards through the string `'spam'`:

```
'spam' ->    (insert 's')     -> 'spams'
       -> (sub 'i' for 'a') -> 'spims'
       ->    (delete 's')     -> 'pims'
```

It took us 3 edits, which is what the DP table told us!

Returning Change

What is the fewest number of coins required to make 42 cents?

- 1 Quarter
- 1 Dime
- 1 Nickel
- 2 Pennies

For a total of 5 coins.

What if I want to use weird coins worth 2 cents, 7 cents, 21 cents, and 38 cents?

Returning Change

```
def change(A, coins[0,i]):
    # Returns the fewest number of coins
    # required to make change for amount A.

    # Base Case 1: no change needed.
    if A == 0: return 0

    # Base Case 2: no coins to use.
    elif i == 0: return ∞

    # Base Case 3: Largest coin too big.
    elif coins[i-1] > A:
        return change(A, coins[0:i-1])
```

Returning Change

```
def change(A, coins[0,i]):
    # Returns the fewest number of coins
    # required to make change for amount A.

    # Base Cases
    ...
    # Base Cases

    else:
        # Let's do lose-it first...

        # Don't use any more of the largest coin.
        loseit = change(A, coins[0:i-1])

        # Use one of the largest coins.
        useit = 1 + change(A-coins[i-1],coins[0:i])

        # Return the fewest number of coins.
        return min(loseit, useit)
```

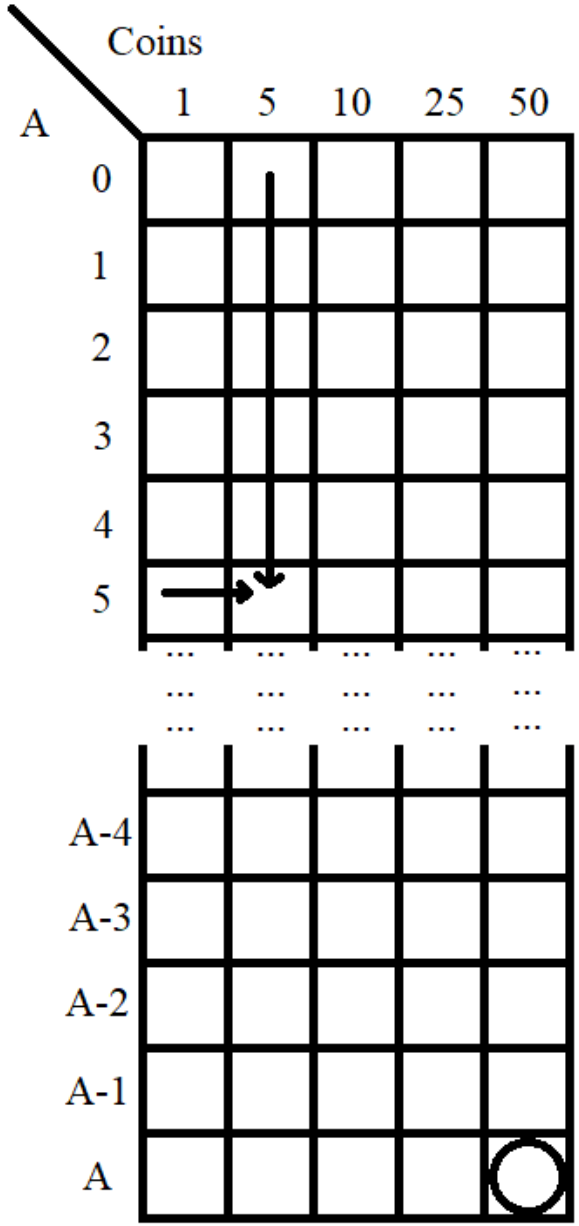
Returning Change

For change, how many different recursive calls are made? Could be up to A... (exact change using only pennies)

So store the results in an DP table with A down the rows, and coins across the columns!

- Fill the table from top left to bottom right.
- Start with first row/column.
- Then fill in one column at a time, left to right.
- $O(A \cdot n)$

Returning Change



Ex: `coins(7, [1,3,5])`

Returning Change

Ex: `coins(7, [1,3,5])`

Note that now, when we consider the use-it case, we need to look multiple rows above.

For example, if we are at the 5th row and 2nd column, the use-it case looks at the 2nd row and 2nd column.

We will still record this with a U, understanding that how far up depends on which column we are in.

Also: **I made a mistake in lecture. If we cannot check the use-it case, we were actually in the 'special base case' where the coin was too big. This meant we removed the coin from the list, i.e., we looked left in the table.**

Returning Change

	1	3	5
0	0	0	0
1	1	1	1
2	2	2	2
3	3	1	1
4	4	2	2
5	5	3	1
6	6	2	2
7	7	3	3

	1	3	5
0	0	0	0
1	1	L	L
2	2	L	L
3	3	U	L
4	4	U	L
5	5	U	U
6	6	U	U
7	7	U	U

- Start in the bottom right. See U for a coin worth 5, so use a 5-coin. Move up 5.
- See L. Move left.
- See L. Move left.
- See U for a coin worth 1, so use a 1-coin. Move up 1.
- See U for a coin worth 1, so use a 1-coin. Move up 1.
- See 0, base case so we are done.

Returning Change

Ex: `coins(7, [1,3,5])`

- Start in the bottom right. See U for a coin worth 5, so use a 5-coin. Move up 5.
- See L. Move left.
- See L. Move left.
- See U for a coin worth 1, so use a 1-coin. Move up 1.
- See U for a coin worth 1, so use a 1-coin. Move up 1.
- See 0, base case so we are done.

So when asked for 7 change, we gave one 5-coin and two 1-coins for a total of 3 coins!