Changes to quiz close dates

Because you’re mostly preparing for the test now, the close dates for quizzes 7 and 8 have been moved on 5 days and 3 days respectively.

- Quiz 7 closes Wednesday, 12 May
- Quiz 8 closes Monday, 17 May

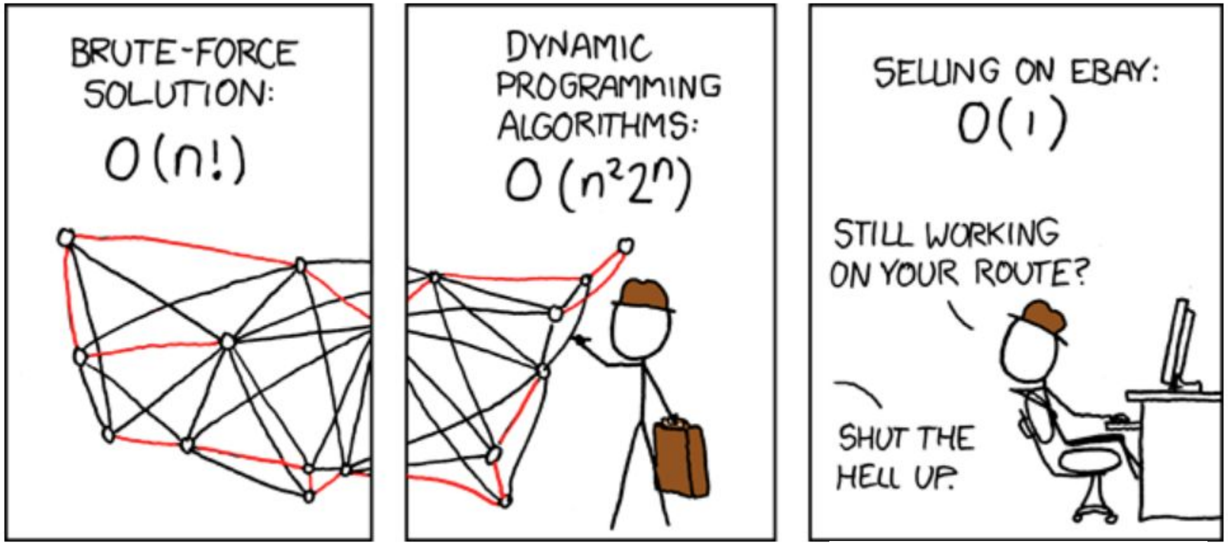
Beyond that quizzes will revert to the usual pattern

- Open Tuesday
- Close Friday the following week

COSC 262
Algorithms

Optimization Methods Part 2

Dynamic Programming



https://xkcd.com/399/

Richard Lobb and R Mukundan
Department of Computer Science and Software Engineering
University of Canterbury

What does the name mean?

Coined by Richard Bellman in the 1950s. He said:

“... it's impossible to use the word *dynamic* in a pejorative sense … 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.”

Top-Down Recurrence

Consider the recurrence relation for the Fibonacci sequence:

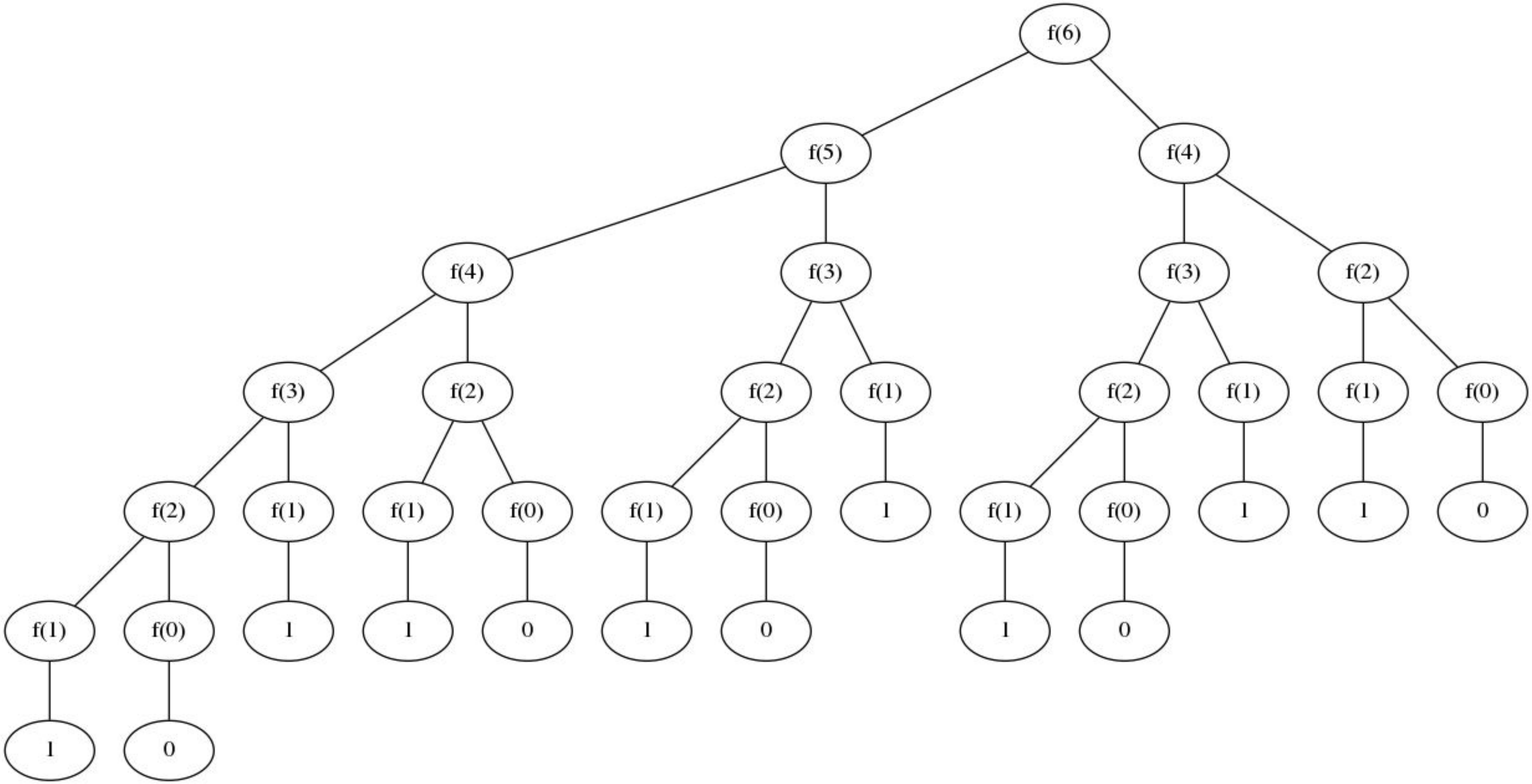
$$f(n) = f(n-1) + f(n-2)$$

```
def fibonacci(n):
    """ Recursive evaluation of Fibonacci sequence """
    if n < 2:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

This is not an efficient procedure as there are many duplicate recursive calls.

Call graph

Trace of the recursive calculation of *f*(6):



Order of complexity of $f(n)$: $\Theta(\varphi^n)$, $\varphi = (1+\sqrt{5})/2$

Top-Down Recurrence, cached

“Memoisation”: Storing calculated values for future use, so that the function need not be called again for those values.

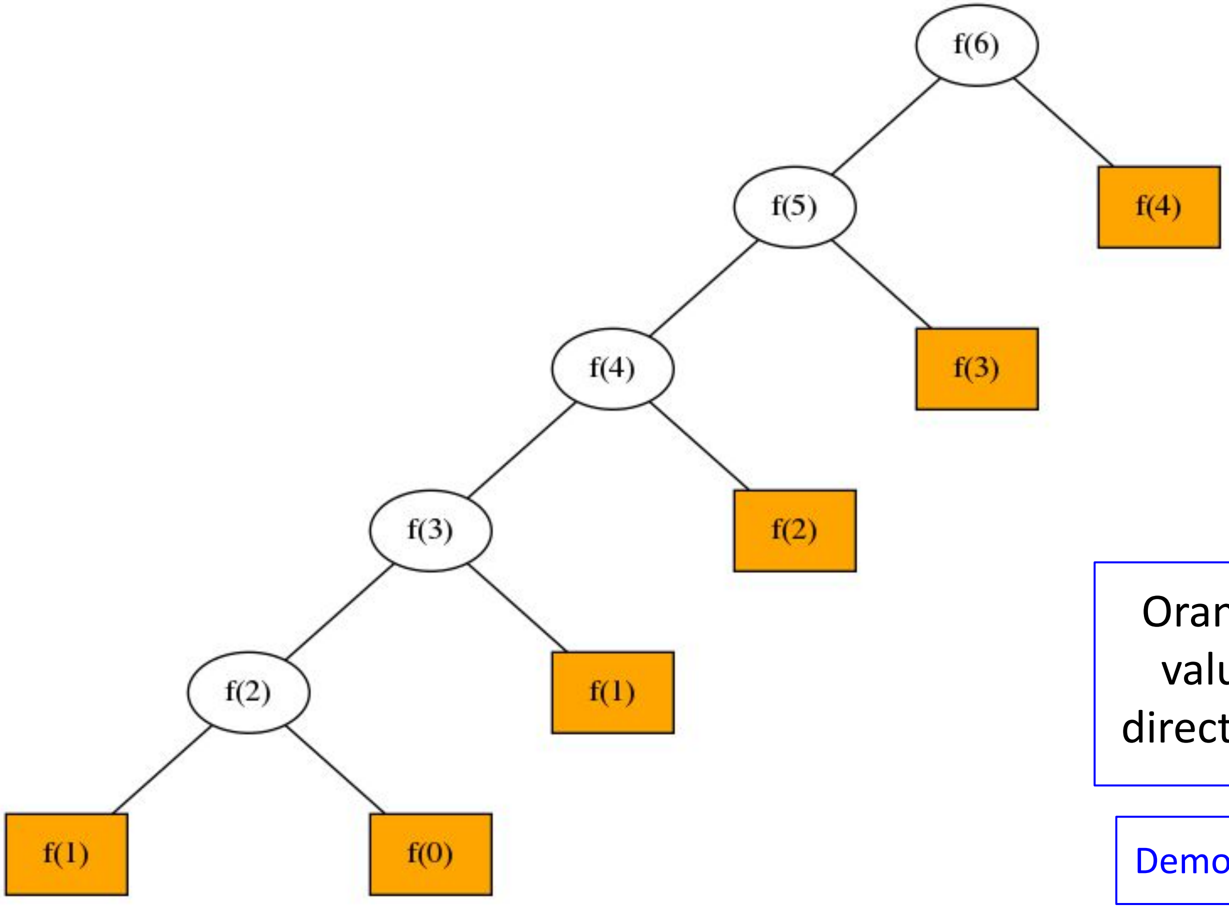
```
""" Fibonacci sequence using memoisation"""

fib = {0:0, 1:1}  # Cache of memoised answers

def fibonacci_m(n):
    """ Recursive evaluation of Fibonacci sequence """
    if n not in fib:
        fib[n] = fibonacci_m(n-1) + fibonacci_m(n-2)
    return fib[n]
```

Top-Down Recurrence

Trace of the calculation of *f*(6) using memoisation:



Orange nodes are values returned directly from cache.

Demo in Python tutor

Order of complexity of *f*(*n*) now: $\Theta(n)$

Interlude: Pythonic memoisation

Memoisation is a useful design pattern.

Can be generalised with a function like:

```
def memoise(f):
    """ Return a memoised version of function f """
    known = {}
    def new_func(*params):
        if params not in known:
            known[params] = f(*params)
        return known[params]
    return new_func
```

We can then follow our non-memoised fibonacci definition by

```
fibonacci = memoise(fibonacci)
```

Notes on *memoise*

To understand the *memoise* function you need to know:

- Functions are objects too.
  - Can be put in lists, passed as parameters, returned from functions, etc
- Function objects are created by *def* statements or *lambda* expressions.
- defs can appear anywhere a statement is valid.
  - So you can have *nested* functions.
- A nested function can "see" its parent's scope.
  - That scope persists even after the parent function returns!
    - The function plus its persisted scope is called a *closure*.
- The *parameter* `*params` captures all parameters as a tuple.
- An *argument* `*params` expands that tuple into a list of args.

Pythonic memoisation: decorators

But nicer is to apply memoise as a *decorator*:

```
# Insert memoise function here, as before. Then ...

@memoise
def fibonacci(n):
    """ Recursive evaluation of Fibonacci sequence """
    if n < 2:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

@memoise is a “decorator expression”, which applies the decorator function (memoise) to the immediately following function, (re)binding the function name to the result of the application, i.e. *fibonacci = memoise(fibonacci)*.

functools.lru_cache

But why re-invent the wheel?

Better to use Python’s own memoise function: *lru_cache*

```
from functools import lru_cache

@lru_cache(maxsize=None)  # None => ∞
def fibonacci(n):
    """ Recursive evaluation of Fibonacci sequence """
    if n < 2:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

“lru” = “least recently used”, referring to the strategy for discarding cache entries to make space for new ones when maxsize is limited.

But ...

Many exercise in COSC262 will require you to implement your own caching because:

1. That way you understand exactly what's happening
2. Custom caches (e.g. using a list indexed by an int rather than a dictionary) are likely to be more efficient
3. You may want to inspect the cache afterwards, e.g. for back-trace
4. If you have to implement DP in another language you'll probably have to implement your own cache anyway

Interlude 2: mutable default parameter

Pylint objects to mutable default parameters, e.g. [] or {}. Here’s why:

```
def appended(item, dest=[]):  # Empty list as default is flagged as bad
    """Append item to given destination list and return it"""
    dest.append(item)
    return dest

print(appended(4, [1, 2, 3]))
print(appended(10))
print(appended(20))
```

Outputs:

```
[1, 2, 3, 4]
[10]
[10, 20]
```

⬅ !

Reason: the default parameter value/object is created when the function is first defined, not when it’s first needed. There’s only one such object, and it’s reused whenever it’s needed.

Relevance to DP:

function definitions like *my_memoised_func(param1, param2, cache={})* are error prone.

Interludes end

Bottom-Up Enumeration

The Fibonacci sequence can be evaluated in a sequence from *i* = 1..*n*, storing all values.

```
def fibonacci_i(n):
    """ Sequential evaluation of Fibonacci sequence"""
    fib = [0]*(n+1)
    fib[1] = 1
    for i in range(2, n+1):
        fib[i] = fib[i-1] + fib[i-2]
    return fib[n]
```

Time complexity: $\Theta(n)$, Space complexity: $\Theta(n)$

Bottom-Up Enumeration, take #2

The previous method can be further improved, storing only two values:

```
def fibonacci_i(n):
      """ Iterative evaluation of Fibonacci sequence """
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a
```

Time complexity: $\Theta(n)$, Space complexity: $\Theta(1)$

Dynamic Programming ("DP")

- An algorithm design method that can be used to solve optimisation programs in which a solution to a problem can be expressed in terms of solutions to a set of overlapping sub-problems [i.e. a *recursive* formulation].
- Two different implementation methods
  - **Top-down Dynamic Programming** ("top-down DP")
    - Write the solution using recursion + memoisation
  - **Bottom-up Dynamic Programming** ("bottom-up DP")
    - Build a table of solutions starting with the simplest subproblems.
    - Fill out the table case by case, combining known sub-solutions, until the target problem is reached
    - "Classic DP"

The Minimum Cost Path problem

Array $W_{i,j}$

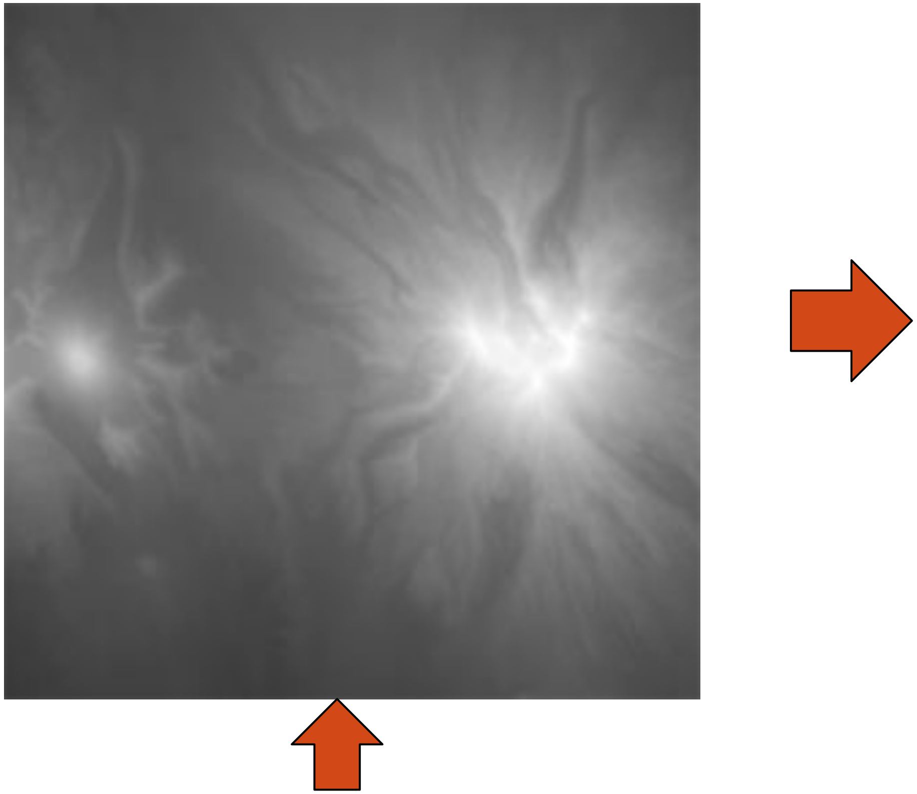
Col Row	0	1	2	3	4
0	6	7	4	7	8
1	7	6	1	1	4
2	3	5	7	8	2
3	2	6	7	0	2
4	7	3	5	6	1

We have an n x n grid with a "weight" $W_{i,j}$ at each square (i,j). You have to move from anywhere in the top row to anywhere in the bottom row, one square at time, downwards or diagonally-downwards.

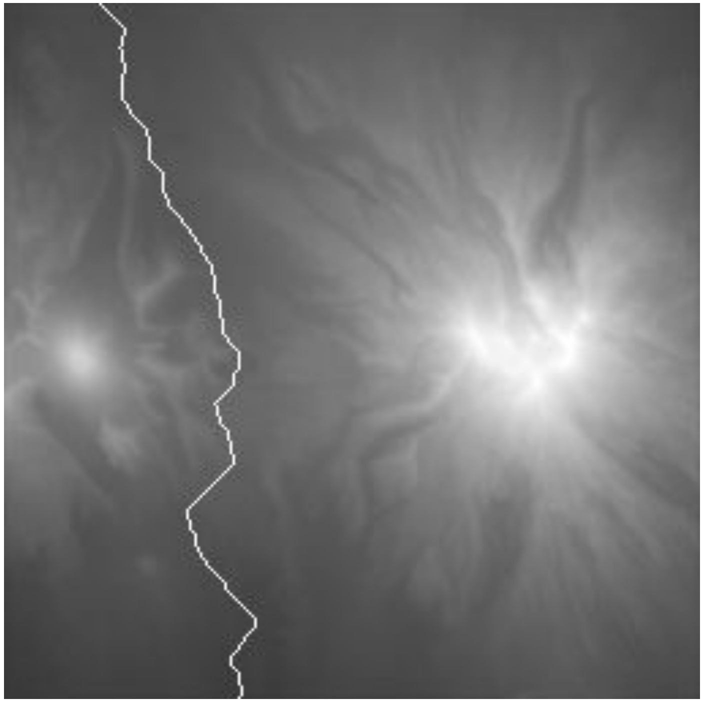
Find the path that minimises the total cost, defined as the sum of the weights on the path.

Example: Tongariro National Park

Suppose weight/cost of a bit of terrain is its height (?!)
Find minimum cost path from top to bottom (or vice versa).



Mt. Ruapehu and Mt. Ngauruhoe
Height Map



Recurrence formula

- Let $C_{i,j}$ be the optimal cost of getting to square (i,j)
- Then:

$$C_{i,j} = \begin{cases} W_{i,j} & i = 0 \\ W_{i,j} + \min\limits_{k \in \{j-1, j, j+1\}} \{C_{i-1,k}\} & \text{otherwise} \end{cases}$$

- Required solution is: $\min\limits_{j} \{C_{n-1,j}\}$
- But: recursive implementation involves overlapping subproblems.
- So … either memoise the function or use bottom-up DP.

Top-down DP (pseudocode)

```
cache = empty

function cell_cost(row, col)
    if (row, col) is outside the grid
        return ∞
    else if row = 0    # Top row?
        return weight(row, col)
    else if (row, col) in cache
        return answer from cache
    else:
        answer = min(cell_cost(row - 1, col - 1),
                     cell_cost(row - 1, col),
                     cell_cost(row - 1, col + 1)) + weight(row, col)
        store answer in cache
        return answer

grid_cost = min(cell_cost(bottom_row, col), col=0,1, … num_cols-1)
```

Bottom-up DP solution

Construct a table of all $C_{i,j}$ working up from simplest case, i.e., row-wise from top row to bottom row. Check 3 entry routes into each cell, take cheapest.

$$C_{i,j} = \begin{cases} W_{i,j} & i = 0 \\ W_{i,j} + \min\limits_{k \in \{j-1,j,j+1\}} \{C_{i-1,k}\} & \text{otherwise} \end{cases}$$

	0	1	2	3	4
0	6	7	4	7	8
1	7	6	1	1	4
2	3	5	7	8	2
3	2	6	7	0	2
4	7	3	5	6	1

Cell weights

	0	1	2	3	4
0	6	7	4	7	8
1	?	?	?	?	?
2	?	?	?	?	?
3	?	?	?	?	?
4	?	?	?	?	?

Cell costs (= DP Table), row 0

	0	1	2	3	4
0	6	7	4	7	8
1	13	10	5	5	11
2	?	?	?	?	?
3	?	?	?	?	?
4	?	?	?	?	?

Cell costs (= DP table), rows 0 & 1

etc

Interlude: 2D tables in Python

Example code to create and fill a table with the (meaningless) recurrence equation:

$$w_{i,j} = \begin{cases} 0 & i = 0 \\ 1 + w_{i-1,j} & i \neq 0, j < i \\ 2 + w_{i-1,j-i} & \text{otherwise} \end{cases}$$

```
def filled_table(n_rows, n_cols):
   """Make the table, fill it and return it"""

   table = [n_cols * [0] for row in range(n_rows)]

   # Now fill it using the recurrence equation.
   # Row 0 is already zero, so don't really need to do it, but to make the
   # code as general as possible, we'll plug the zeros in (again).
   for i in range(n_rows):  # For each row
       for j in range(n_cols):   # For each column
           # Evaluate the recurrence equation to fill in the table value.
           if i == 0:
               table[i][j] = 0   # Base case (redundant, since already 0)
           elif j < i:
               table[i][j] = 1 + table[i - 1][j]
           else:
               table[i][j] = 2 + table[i - 1][j - i]

   return table
```

Interlude: 2D tables in Python (cont’d)

```
def print_table(table):
    """Pretty(ish) print of table, row by row"""
    n_rows = len(table)
    n_cols = len(table[0])
    for i in range(n_rows):
        print(f"{i}:", end='')
        for j in range(n_cols):
            print(f"{table[i][j]:5}", end='')
        print() # Line break between rows

def main():
    # Demo by making and printing the table
    table = filled_table(4, 5)
    print_table(table)

main()
```

Output

```
0:    0    0    0    0    0
1:    1    2    2    2    2
2:    2    3    3    4    4
3:    3    4    4    4    5
```

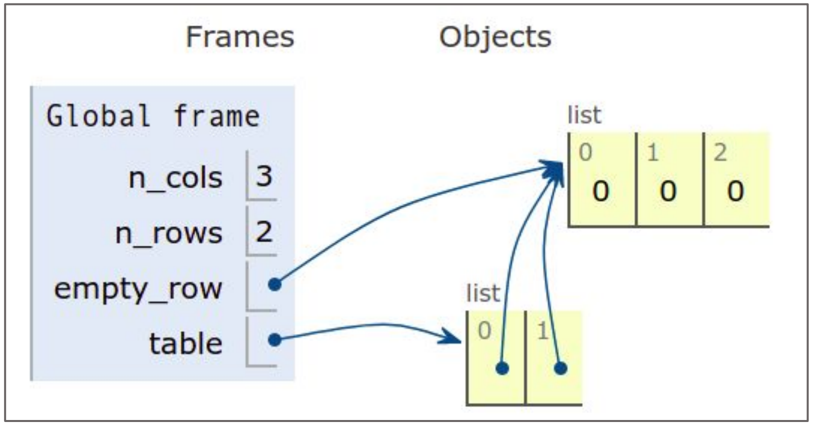
Interlude: 2D tables in Python (cont’d)

How *not* to initialise a 2D table:

```
empty_row = n_cols * [0]
table = n_rows * [empty_row]

# Proof it’s bad:
table[0][0] = 99 # Change 1 cell
print(table)
```

Run it in Python tutor (pythontutor.com):



Prints

```
[[99, 0, 0], [99, 0, 0], [99, 0, 0], [99, 0, 0]]
```

All rows changed!

Reason: all elements of *table* reference the same *empty_row* object

NB: `table = [empty_row for row in range(n_rows)]` fails too

Interlude to the interlude: for enthusiasts only!

2D tables in *numpy*

Example code to create and fill a table with the (meaningless) recurrence equation:

$$w_{i,j} = \begin{cases} 0 & i = 0 \\ 1 + w_{i-1,j} & i \neq 0, j < i \\ 2 + w_{i-1,j-i} & \text{otherwise} \end{cases}$$

```
import numpy as np  # If you have it installed!
def filled_table2(n_rows, n_cols):
    """Make the table, fill it and return it"""

    table = np.zeros((n_rows, n_cols), dtype=int)   # A numpy table/matrix.

    # Now fill it using the recurrence equation.
    # Row 0 is already zero, so don't really need to do it, but to make the
    # code as general as possible, we'll plug the zeros in (again).
    for i in range(n_rows):  # For each row
        for j in range(n_cols):   # For each column
            # Evaluate the recurrence equation to fill in the table value.
            if i == 0:
                table[i, j] = 0   # Base case (redundant, since already 0)
            elif j < i:
                table[i, j] = 1 + table[i - 1, j]
            else:
                table[i, j] = 2 + table[i - 1, j - i]

    return table
```

Interlude to the interlude, for enthusiasts only, continues

2D tables in Python (cont’d)

```
def main():
    # Demo by making and printing the table
    table = filled_table2(4, 5)
    print(table)
main()
```

Output

```
[[0 0 0 0 0]
 [1 2 2 2 2]
 [2 3 3 4 4]
 [3 4 4 4 5]]
```

This is how numpy formats 2D arrays.
No special function needed now.

Interlude(s) end.

Backtracking to find the solution

- Backtrack from bottom row back to top
- Choose cheapest entry path for each cell

- *Or* record predecessor cells during construction
  - E.g. make cell-cost a tuple (cell_cost, predecessor_col)
  - Often this is easier

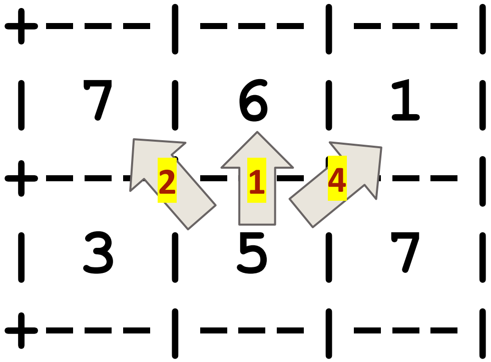
```
      0    1    2    3    4
   +----|----|----|----|----+
0  | 6  | 7  |  4 |  7 |  8 |
   +----|----|----|----|----+
1  | 13 | 10 |  5 |  5 | 11 |
   +----|----|----|----|----+
2  | 13 | 10 | 12 | 13 |  7 |
   +----|----|----|----|----+
3  | 12 | 16 | 17 |  7 |  9 |
   +----|----|----|----|----+
4  | 19 | 15 | 12 | 13 |  8 |
   +----+----+----+----+----+
```

```
      0    1    2    3    4
   +----|----|----|----|----+
0  |6,- |7,- |4,- |7,- |8,- |
   +----|----|----|----|----+
1  |13,0|10,2|5,2 |5,2 |11,3|
   +----|----|----|----|----+
2  |13,0|10,2|12,2|13,1|7,3 |
   +----|----|----|----|----+
3  |12,1|16,1|17,1|7,4 |9,4 |
   +----|----|----|----|----+
4  |19,0|15,0|12,3|13,3|8,3 |
   +----+----+----+----+----+
```

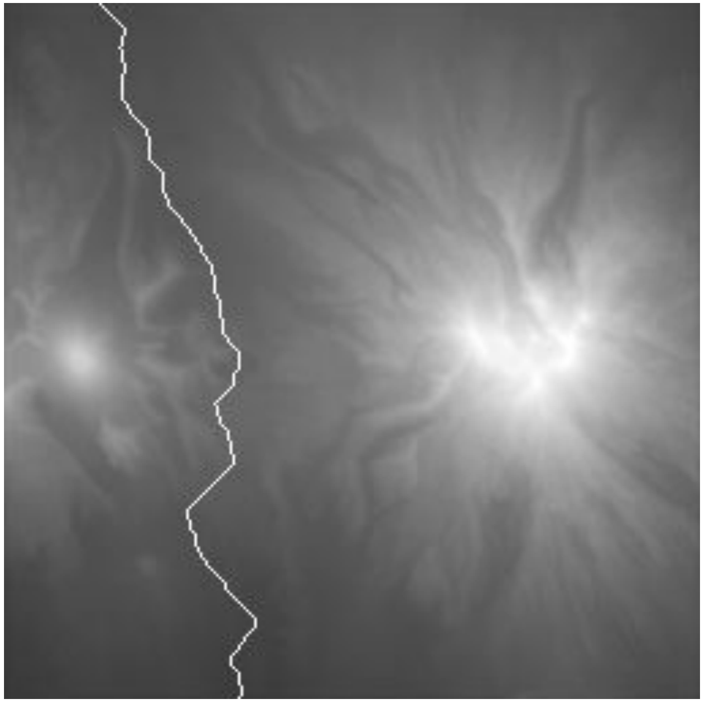
NB: these are *cell-cost* grid, not cell-weight grids.

Exercise

- The path in the Tongariro example (a height field) minimised the sum of the heights.
- More realistically for tramping we would want to minimise the sum of the absolute values of the *changes* in height from one cell to the next.
    - ○ Raise change to some power to penalise large steps

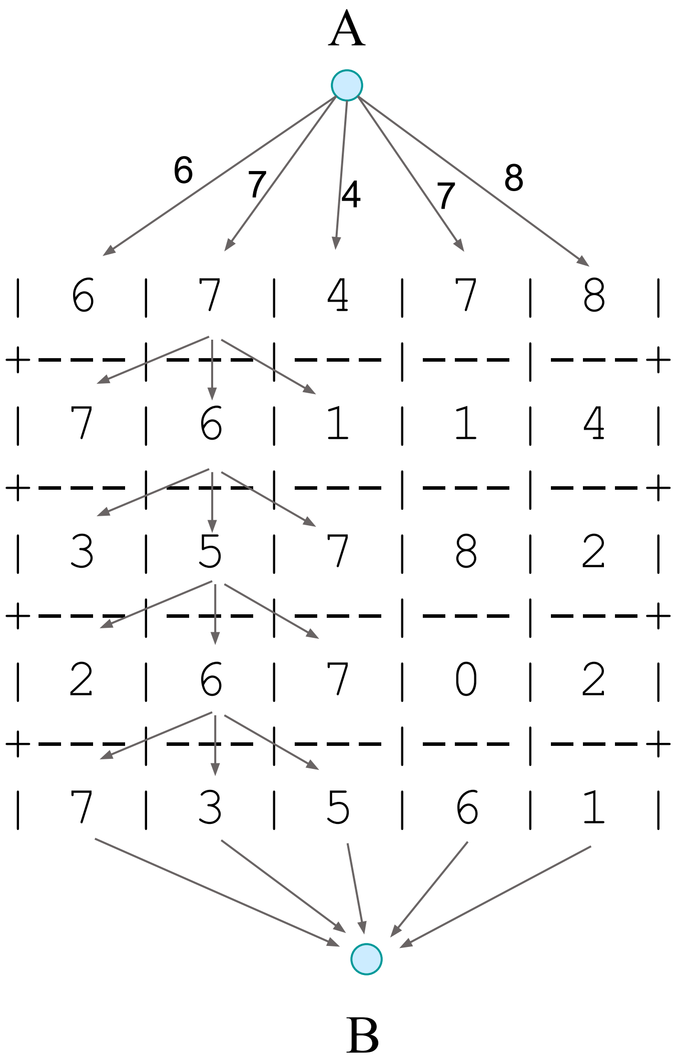


```
+---|---|---|
| 7 | 6 | 1 |
+---|---|---|
| 3 | 5 | 7 |
+---|---|---|
```



- Can you generalise the algorithm to do this?

Dijkstra shortest path as an alternative?



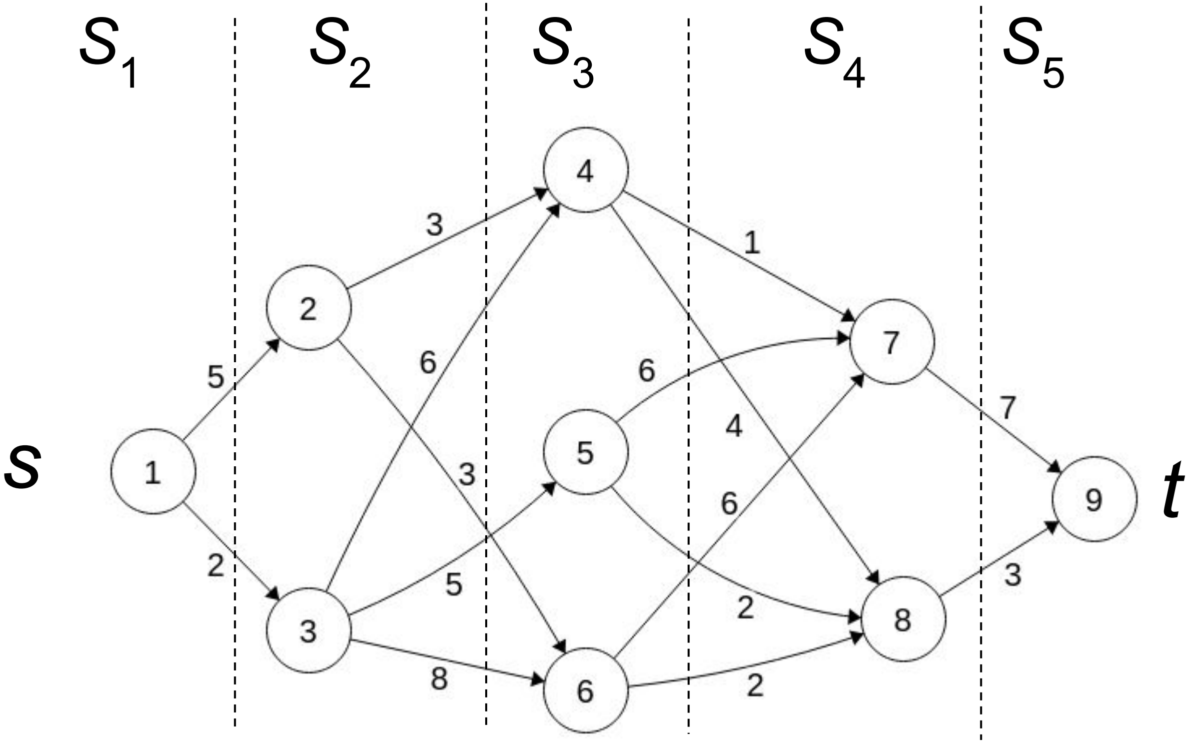
```
| 6 | 7 | 4 | 7 | 8 |
+---|---|---|---|---+
| 7 | 6 | 1 | 1 | 4 |
+---|---|---|---|---+
| 3 | 5 | 7 | 8 | 2 |
+---|---|---|---|---+
| 2 | 6 | 7 | 0 | 2 |
+---|---|---|---|---+
| 7 | 3 | 5 | 6 | 1 |
```

- The problem can also be done with a graph algorithm as on the left.
  - Edges are shown only for the second column
- Move weights to the edges.
- Apply Dijkstra shortest path algorithm.
- BUT: harder and less efficient
  - How much less efficient?

A generalisation: multistage graphs

Dijkstra algorithm is unnecessarily complicated and inefficient in this case, which is a *multistage graph*.

Consider the following multistage graph (*V*, *G*) where vertices are partitioned into $n \geq 2$ disjoint sets S_k ($1 \leq k \leq n$), n = 5

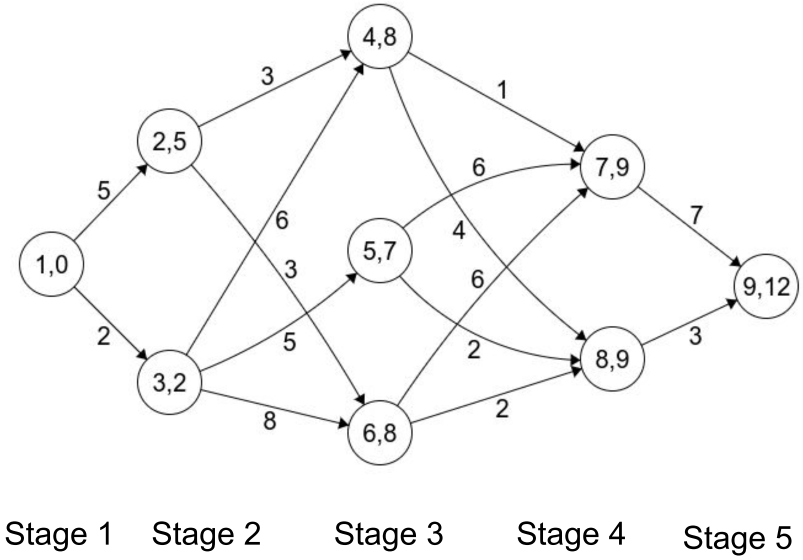


Multistage graph: Each stage has connections only to the next stage.

Goal: find the minimum cost path from *s* to *t*.

Multistage graphs (cont'd)

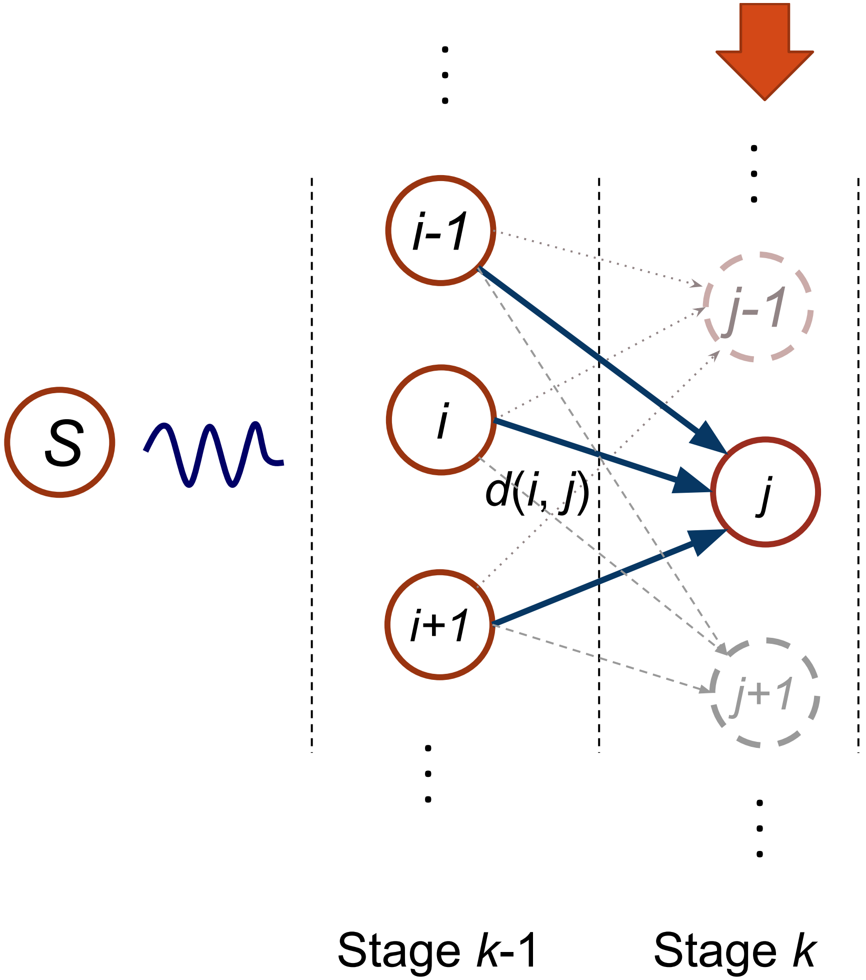
- The minimum cost for each stage is calculated starting from stage 2.
  - Cost appended to node label
- Table shows node cost and (in brackets) predecessor node.
- Minimum distance: **12**



Stage S_2 Nodes 2, 3	**Stage S_3** Nodes 4, 5, 6	**Stage S_4** Nodes 7, 8	**Stage S_5** Node 9
5 (1)	8 (2)	9 (4)	12 (8)
2 (1)	7 (3)	9 (5)	
	8 (2)		

Multistage graphs: recurrence formula

$$\text{for } j \in S_k \quad \text{cost}(j) = \min_{i \in S_{k-1}} \{\text{cost}(i) + d(i,j)\}$$



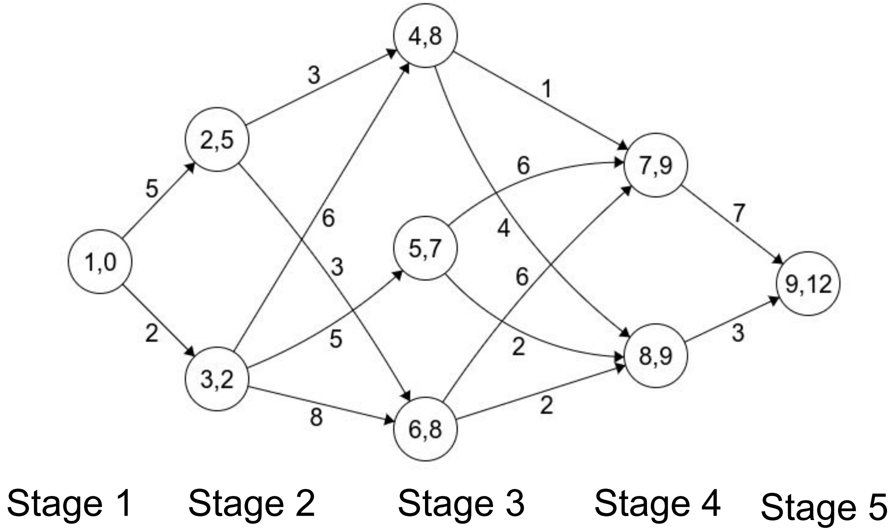
Algorithm visits each node exactly once.

It inspects each incoming edge to each node exactly once.

So it’s O(V + E) where V is number of vertices, E is number of edges.

How does this compare with Dijkstra’s algorithm?

Tracing back the minimum path



Stage S_2 Nodes 2, 3	**Stage S_3** Nodes 4, 5, 6	**Stage S_4** Nodes 7, 8	**Stage S_5** Node 9
5 (1)	8 (2)	9 (4)	12 (8)
2 (1)	7 (3)	9 (5)	
	8 (2)		

Predecessors shown in parentheses.

- We can get the minimum cost path by tracing back the sequence that gave the minimum distance, from the last node.
- Minimum cost path = 1 3 5 8 9

Remind me what DP *is* again?

Dynamic Programming (DP) is a method for solving optimisation problems that have *optimal substructure* and where the number of possible subproblems of the original problem is computationally tractable due to *overlapping subproblems* and a sufficiently small problem space.

It involves either:

1. **Top-down DP**: expressing the solution to the original problem recursively, using memoisation to record subproblem solutions as they're encountered, or
2. **Bottom-up DP:** using a recurrence equation ("recursion") to compute the solutions to all possible subproblems, starting from the simplest and working up to the required solution.

Optimal substructure

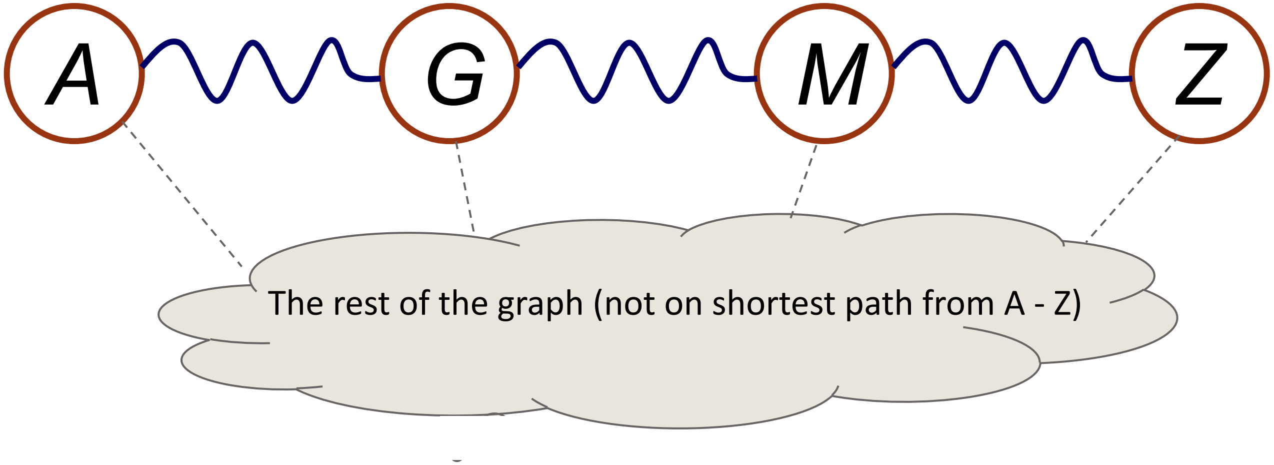
If an optimization problem can be solved using optimal solutions of its sub-problems, then it has optimal substructure.

- Let a solution *S* to an optimization problem be given in terms of solutions S_1, S_2, …. , S_n to smaller instances of the problem.
- Then “*S* is an optimal solution to the problem instance” implies “S_1, S_2, …., S_n are optimal solutions to their associated problem instances”

Optimal substructure: example

Example - Shortest path problem:

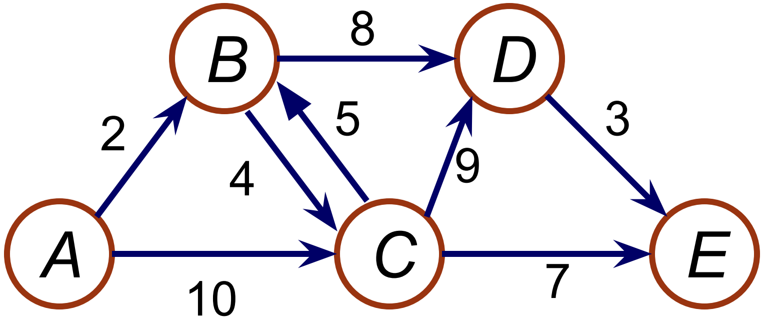
Let the shortest path from node *A* to node *Z* be given in terms of nodes *G*, *M*:



Then, each segment (AG, GM, MZ) of the path in the solution is also a shortest path between the corresponding nodes.

Not optimal substructure

Example of a problem that does **not** have optimal substructure – Longest simple path problem.



Longest simple path from *A* to *E* ? $A \longrightarrow C \longrightarrow B \longrightarrow D \longrightarrow E$

But, $B \longrightarrow D$ is **not** the longest path between *B* and *D;* the longest path is $B \longrightarrow C \longrightarrow D$.

Overlapping sub-problems

- Solutions of sub-problems can be reused several times in the computation of the optimal solution of the whole problem.
- Example – Fibonacci sequence (This is not an optimization problem, but illustrates the concept of overlapping sub-problems): $C(n) = C(n\text{-}1) + C(n\text{-}2)$
- Sub-problem solutions can be stored in a table (“cache”) so that they don’t have to be recomputed every time they are needed.
- In addition to finding the minimum cost, we also often require the set of elements (nodes, items etc) that yield the optimal solution.
  - Usually involves tracing back from the solution to the starting point.

Properties of the recursive solution

For the recursion *soln(param1, param2, …)* to be useful, we require

card(param1) * card(param2) * … * card(param*n*) <= N

where:

- *card* is the "cardinality" of the parameter, meaning how many different values it might possibly take
- N is an estimate of how many low-level operations you can afford

LHS is the size of the search space. If small enough, overlapping subproblems are assured (or it's a trivial exhaustive search).

Rules of thumb:

C language, 1 second allowed: N ≈ 10^7 - 10^8.

Python 10 - 50 times slower.

When is a DP approach likely to work?

It can be hard to recognise a DP problem. But some common properties are:

1. The problem is combinatorial. A brute force solution involves a series of “either I do this, or I do that” decisions.
2. The problem involves only integers or slices of fixed strings with well-defined bounds, giving a finite domain to be searched.
3. You're looking for an *optimal* solution in the search space.

If those conditions are satisfied you can then start trying to find a recursive solution *soln(param1, param2, …)*

- Recursion => building the solution to a problem from solutions to subproblems (“optimal substructure”)
- Expect to have *overlapping subproblems*

Coin changing problem

- We saw earlier that a greedy approach does not always give an optimal solution to the coin changing problem.
- It’s a combinatorial optimisation problem over a bounded integer domain.
- So … DP to the rescue?
- We need to find a recursive formulation for the function *coins_reqd(amount, coinage)* which returns the minimum number of coins required.
  - Later we’ll extend it to return the actual coin list

Coin changing recursions

Reminder: we are assuming the existence of a 1 unit coin.

Consider recursively solving *coins_reqd(395, [1, 10, 25])*

Two different recursions might come to mind:

1. Either the solution has at least one 25c coin (the last coin) or it has none. So:
   a. Remove one 25c coin (a cost of 1 coin) and recurse to solve *1 + coins_reqd(370, [1, 10, 25])*.
   b. Recurse to solve *coins_reqd(395, [1, 10])*
   c. Choose whichever outcome is best.

Cont'd on next slide

Coin Changing Recursions (cont’d)

2. The best solution certainly has at least one 1c coin **or** one 10c coin **or** one 25c coin (assuming amount > 0). So
    a. Recurse to solve 1 + *coins_reqd(394, [1, 10, 25])*
    b. Recurse to solve 1 + *coins_reqd(385, [1, 10, 25])*
    c. Recurse to solve 1 + *coins_reqd(370, [1, 10, 25])*
    d. See which of the above three gives the best result.

- Both recursions work and can be memoised or implemented as bottom-up DP.
    - Equivalent temporal complexity
- Method 2 is the standard approach because only one of the two parameters changes.
    - Cache is simpler (a list rather than a table or dictionary) and more space efficient. [Q: what are the two space complexities?]
- We’ll study just method 2.

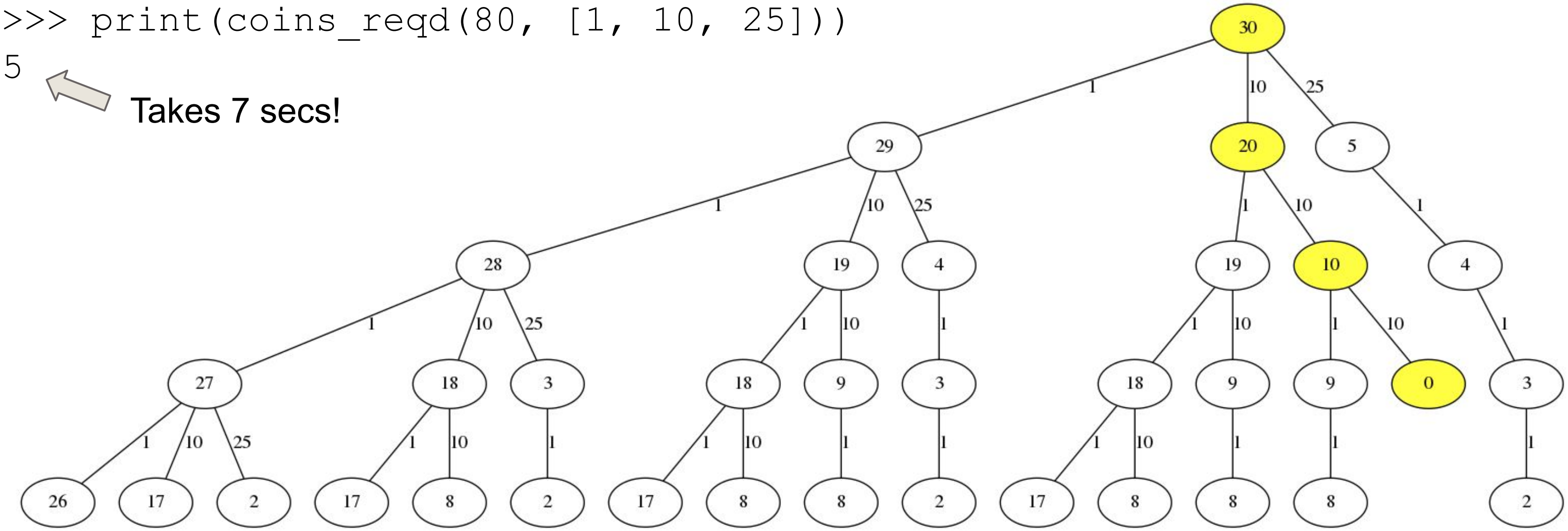
Method 2: pure recursive solution

```
INFINITY = float('inf')
def coins_reqd(value, coinage):
    """The minimum number of coins to represent value"""
    if value == 0:
        return 0
    elif value < 0:
        return INFINITY  # No solution possible
    else:
        return 1 + min(coins_reqd(value - coin, coinage) for coin in coinage)
```

```
>>> print(coins_reqd(30, [1, 10, 25]))
3
>>> print(coins_reqd(80, [1, 10, 25]))
5
```

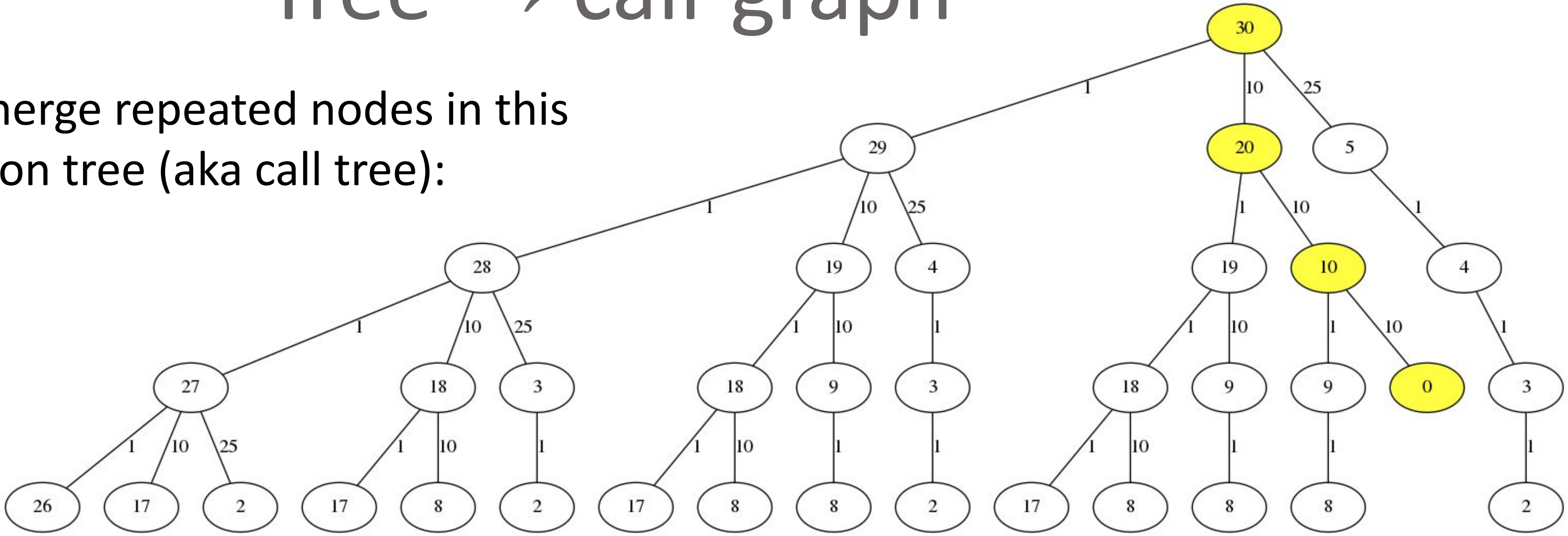
Takes 7 secs!

Recursion tree (top 5 levels, of 30).
Negative nodes pruned.



Tree → call-graph

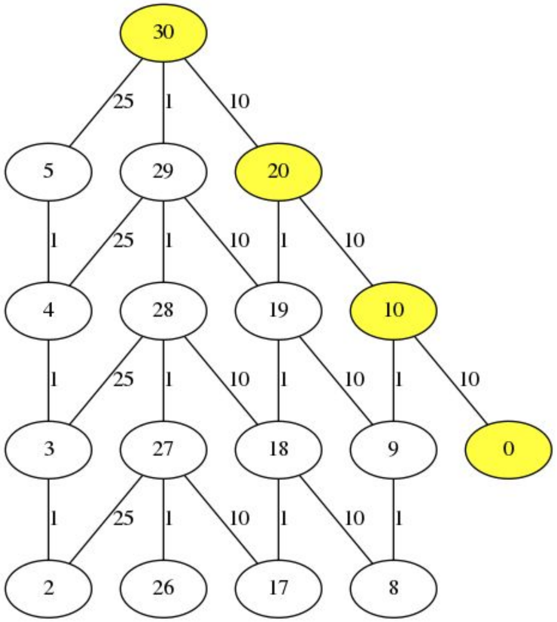
If we merge repeated nodes in this recursion tree (aka call tree):



… we get this call graph (top 5 levels only are shown).

Shows the overlapping subproblems.

How many nodes in the full call graph?



Coin changing: top-down DP

Just memoise it, right?

First try:

```
from functools import lru_cache
INFINITY = float('inf')

@lru_cache(None)  # None means maxsize = infinity
def coins_reqd(value, coinage):
    """The minimum number of coins to represent value"""
    if value == 0:
        return 0
    elif value < 0:
        return INFINITY  # No solution possible
    else:
        return 1 + min(coins_reqd(value - coin, coinage) for coin in coinage)
```

```
Traceback (most recent call last):
  File "junk.py", line 23, in <module>
    print(coins_reqd(80, [1, 10, 25]))
TypeError: unhashable type: 'list'
```

Same happens with our own *memoise* decorator function

Solution 0: Use a tuple

- Use a tuple instead of a list for the coinage!
- But … including the entire coin-list tuple as an element of the cache key tuple every time is very inefficient.
  - Involves hashing the coin list on every call (O(*n*) on each call), where *n* is the number of coins
- So let's not do that!

Solution 1: code the cache explicitly

```
INFINITY = float('inf')

def coins_reqd(value, coinage, cache=None):
    """The minimum number of coins to represent value"""
    if cache is None:
        cache = (value + 1) * [None]  # Initialise cache on first call
    if value == 0:
        return 0
    elif value < 0:
        return INFINITY  # No solution possible
    else:
        if cache[value] is None:
            cache[value] = 1 + min(coins_reqd(value - coin, coinage, cache) for coin in coinage)
        return cache[value]
```

We're using a simple list instead of a dictionary for the cache. Faster.

```
>>> print(coins_reqd(30, [1, 10, 25]))
3
>>> print(coins_reqd(80, [1, 10, 25]))
5
```

Takes 0.5 msecs
(c.f. 7 sec before;
10,000 times faster)

Solution 2: a variant

```
INFINITY = float('inf')

def coins_reqd(value, coinage):
    """The minimum number of coins to represent value"""

    cache = (value + 1) * [None]

    def coins_recursive(value):
        """ The recursive function with coinage and cache in outer scope """
        if value == 0:
            return 0
        elif value < 0:
            return INFINITY  # No solution possible
        else:
            if cache[value] is None:
                cache[value] = 1 + min(coins_recursive(value - coin) for coin in coinage)
            return cache[value]

    return coins_recursive(value)
```

Solution 3: another variant

```python
from functools import lru_cache
INFINITY = float('inf')

def coins_reqd(value, coinage):
    """The minimum number of coins to represent value"""

    @lru_cache(maxsize=None)  # OK now because coinage is in outer scope
    def coins_recursive(value):
        """ The simplified recursive function."""
        if value == 0:
            return 0
        elif value < 0:
            return INFINITY  # No solution possible
        else:
            return 1 + min(coins_recursive(value - coin) for coin in coinage)

    return coins_recursive(value)
```

Bottom-up DP solution

- BUT: top-down DP gives stack overflow ("recursion limit exceeded") for large amounts.
- Setting a higher recursion limit sometimes works but Python may then crash.
- Solution is to switch to bottom up DP
    - i.e., build a table of the subproblem solutions from the simplest up to the target problem.
    - Bottom-up DP is nearly always much faster, too.
- For this we need a recurrence equation.

The recurrence equation

We can express the previous recursion mathematically as a recurrence equation:

$$N_{amt} = \begin{cases} 0 & amt = 0 \\ 1 + \min\limits_{c \in C, c \leq amt} \{N_{amt-c}\} & amt > 0 \end{cases}$$

where N_{amt} is the minimum number of coins chosen from coinage C needed to represent a given total *amt*.

Coin Changing: bottom-up DP

```
def coins_reqd(value, coinage):
    """The minimum number of coins to represent value, assuming there is
        a 1-unit coin."""
    num_coins = [0] * (value + 1)
    for amt in range(1, value + 1):
        num_coins[amt] = 1 + min(num_coins[amt - c] for c in coinage if  amt >=  c)
    return num_coins[value]
```

```
>>> coins_reqd([1, 10, 25], 30))
>>> 3
```

Performance: O(value * len(coinage))
Memory: O(value)

Resulting num_coins array (N[i]), a 1-row DP table:

```
   i:  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
N[i]:  0  1  2  3  4  5  6  7  8  9 10  1  2  3  4  5  6  7  8  9 10  2  3  4  5  6  1  2  3  4  5  3
```

$$N_{amt} = \begin{cases} 0 & amt = 0 \\ 1 + \min\limits_{c \in C, c \le amt} \{N_{amt-c}\} & amt > 0 \end{cases}$$

Coin Changing: bottom-up DP take #2

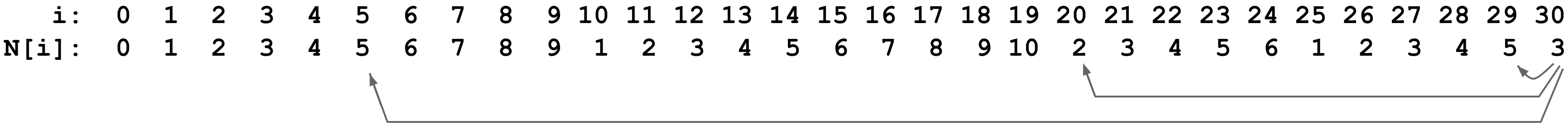
Since the use of a list comprehension seems to cause some confusion, here's a version without a list comprehension.

```
def coins_reqd(value, coinage):
    """A version that doesn't use a list comprehension"""
    numCoins = [0] * (value + 1)
    for amt in range(1, value + 1):
        minimum = None
        for c in coinage:
            if amt >= c:
                coin_count = numCoins[amt - c]  # Num coins required to solve for amt - c
                if minimum is None or coin_count < minimum:
                    minimum = coin_count
        numCoins[amt] = 1 + minimum
    return numCoins[value]
```

Performance: O(value * len(coinage))
Memory: O(value)

```
>>> coins_reqd([1, 10, 25], 30))
>>> 3
```

Resulting num_coins array (N[i]), a 1-row DP table:



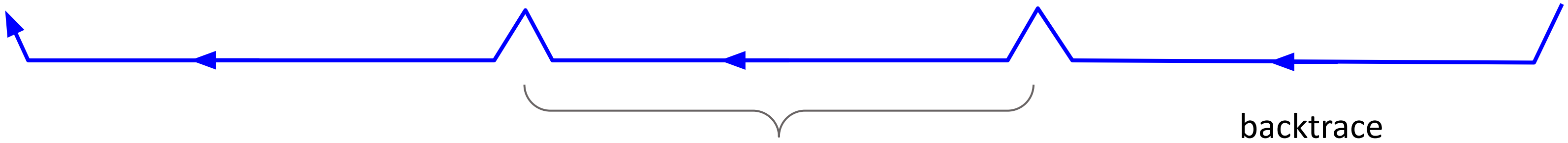
```
   i:  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
N[i]:  0  1  2  3  4  5  6  7  8  9  1  2  3  4  5  6  7  8  9 10  2  3  4  5  6  1  2  3  4  5  3
```

Printing the actual coins

- The above program returns only the minimum number of coins.
- To find the actual coins, back-track either:
  - By checking for the minimum of the potential predecessors i.e. check N[amt - c] for all coins c, or
  - By recording explicit predecessor links (= "which coin used") during table construction.

Recording the predecessors (Pred[i]) is probably easier:

```
      i:  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
   N[i]:  0  1  2  3  4  5  6  7  8  9  1  2  3  4  5  6  7  8  9 10  2  3  4  5  6  1  2  3  4  5  3
Pred[i]:  0  0  1  2  3  4  5  6  7  8  0 10 11 12 13 14 15 16 17 18 10 20 21 22 23  0 25 26 27 28 20
```

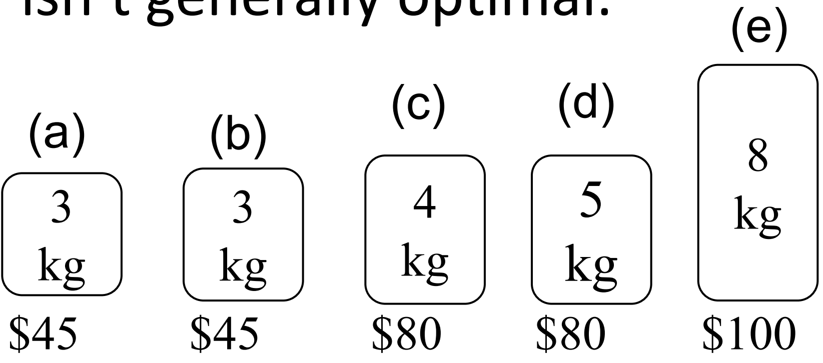


backtrace

hop lengths are coin values (3 x 10-coin hops)

The 0/1 Knapsack

- A thief robbing a store finds n items; the i^{th} item has value v_i dollars, and weighs w_i kgs (v_i , w_i are integers). He wants to carry as valuable a load as possible, but he can carry at most W kgs. Which items should he take?
- Assume items indivisible: called the “0/1 knapsack problem”
- A greedy strategy isn’t generally optimal.
- Example:



- Greedy solution using only v_i : (e) = $100

  or using v_i/w_i : (c) + (d) = $160
- Optimal solution: (a)+(b)+(c) = $170

0/1 Knapsack: is it a DP problem?

Checking the criteria in the slide "When is DP likely to work?":

1. The problem *is* combinatorial.
    - You either pick up each item or you don't, giving 2^n possibilities.
2. It involves integer parameters (weights and values). These are *probably* well bounded
3. It is clearly an *optimisation* problem.

It ticks all the boxes so it **might** be solvable with DP.

But … how to find the recursion?

0/1 Knapsack: finding the recursion

Put yourself in the thief's position, looking at item (v_i, w_i)

How do you decide whether to take it or not?

If you pick it up, you have a smaller effective knapsack that can hold only (W - w_i) kg. It's worth \$$v_i$. Recurse with the smaller knapsack and one fewer item to see what *that's* worth.

If you don't pick it up, you still have a knapsack that can hold W kg, but it's worth \$0. Recurse with the same capacity but one less item available.

0/1 Knapsack: finding the recursion (cont’d)

That leads to the following recursion for the maximum achievable value:

Item	Value ($)	Weight (kg)
1	45	3
2	45	3
3	80	4
4	80	5
5	100	8

```
function max_value(items, n, capacity)
   # Return the maximum value achievable with the first
   #  n items and the given capacity of knapsack
   if  n == 0 or capacity == 0 # Base case
      return 0
   else if weight(items[n - 1]) > capacity  # Item too heavy to fit?
      return max_value(items, n - 1, capacity)
   else
      # Try both with and without the last item. Return the best of the two outcomes.
      return max(max_value(items, n - 1, capacity),
                     value(items[n - 1]) + max_value(items, n - 1, capacity - weight(items[n - 1]))
```

Python code: see weekly quiz

A note on the recursion

- It might seem more natural to cast the recursion in the form “either I take the first item, or I don’t. Recurse on the rest”.
- That works fine, too.
    - You can of course take the items in any order
- But the approach of considering the n^{th} item and recursing with the first $n - 1$ items leads much more naturally into a bottom-up DP solution (coming soon).

0/1 Knapsack recurrence

We can express the algorithm as a recurrence equation as follows.

$$\text{weights} = w_1, w_2, \ldots, w_n$$

$$\text{values} = v_1, v_2, \ldots, v_n$$

Let V(i, C) denote the maximum total value achievable with a knapsack of capacity *C* using only items 1, 2 … *i*. Then:

Item	Value ($)	Weight (kg)
1	45	3
2	45	3
3	80	4
4	80	5
5	100	8

$$V(i,C) = \begin{cases} 0 & i = 0 \\ V(i-1,C) & i > 0, w_i > C \\ max(V(i-1,C), v_i + V(i-1,C-w_i)) & i > 0, w_i \leq C \end{cases}$$

Explanation of recurrence.

$$V(i,C) = \begin{cases} 0 & i = 0 \\ V(i-1,C) & i > 0, w_i > C \\ max(V(i-1,C), v_i + V(i-1, C - w_i)) & i > 0, w_i \leq C \end{cases}$$

The three cases are:

1. If *i* = 0, we have no items, therefore total value = 0.
2. If w_i > capacity C the i^{th} item won't fit in the knapsack, so we can't use it.
3. Otherwise, we try **with** and **without** the i^{th} item and see which gives the best outcome. $C - w_i$ is the capacity if we do use the i^{th} item. In both cases we recurse to get the best result with the remaining *i* - 1 items.

A note on notation

The above recurrence equation follows the 1-origin convention for writing recurrences (often more natural/convenient):

$$\text{weights} = w_1, w_2, \ldots, w_n$$
$$\text{values} = v_1, v_2, \ldots, v_n$$

n is the *length* of the sequences **and also** the index of their last elements.

With 0-origin subscripting, the recurrence is:

$$\text{weights} = w_0, w_1, w_2, \ldots, w_{n-1}$$
$$\text{values} = v_0, v_1, v_2, \ldots, v_{n-1}$$

$$V(i, C) = \begin{cases} 0 & i < 0 \\ V(i-1, C) & i \geq 0, w_i > C \\ max(V(i-1, C), v_i + V(i-1, C - w_i)) & i \geq 0, w_i \leq C \end{cases}$$

When translating 1-origin recurrences to Python, keep in mind that *i* is a *length*, not an index. [It’s *both* in a 1-origin system]

Top-down knapsack

Just memoise the function *max_value(items, n, capacity)*

Can :

- use a dictionary indexed by the tuple (n, capacity)
    - Cached value of function call is *cache[(n, capacity)]*
- or use a table with *n* rows and *capacity* columns as the cache
    - `cache = [capacity * [-1] for row in range(n)]`
    - Cached value of function call is *cache[n][capacity]*
        - `cache[i][c] == -1` => (i, c) not in cache
- or use *lru_cache* decorator, but only if the parameter *items* can be moved to an outer scope
    - Cache is hidden.

Exercise

Given a class *Item* as follows,

```
class Item:
    """An item to (maybe) put in a knapsack"""
    def __init__(self, value, weight):
        self.value = value
        self.weight = weight
```

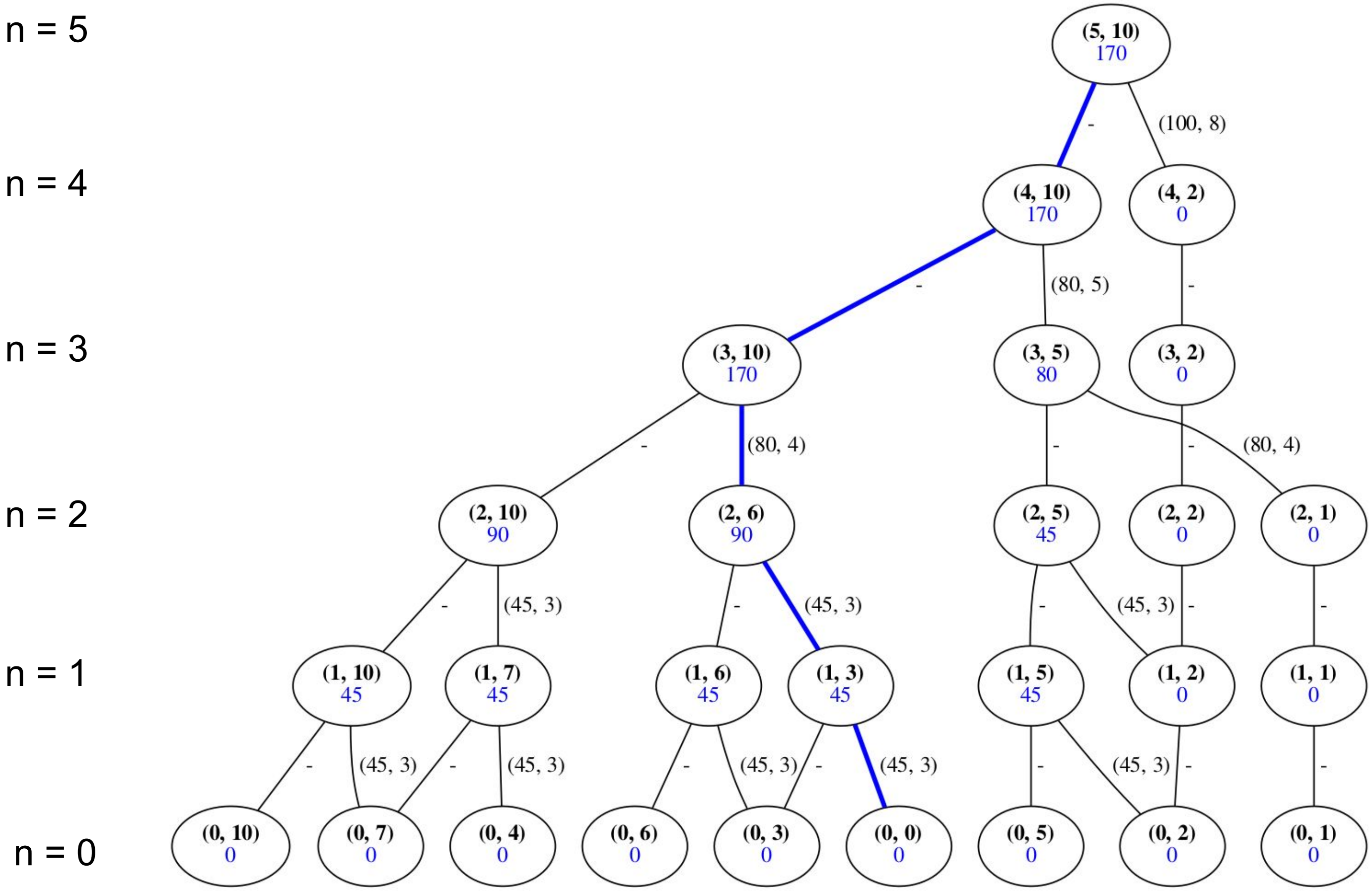
write a recursive top-down DP (i.e. memoised) function *max_value(items, capacity)* which returns the maximum value achievable with a given knapsack size using the given list of items. You may assume that there will fewer than 500 items and that the capacity will be less than 500.

See lab quiz.

0/1 Knapsack: top-down DP call graph

Showing all calls to *max_value* with params (n, capacity).
Number in blue is return value from the call, i.e. the value of the node.
Edge labels show (value, weight) picked up.

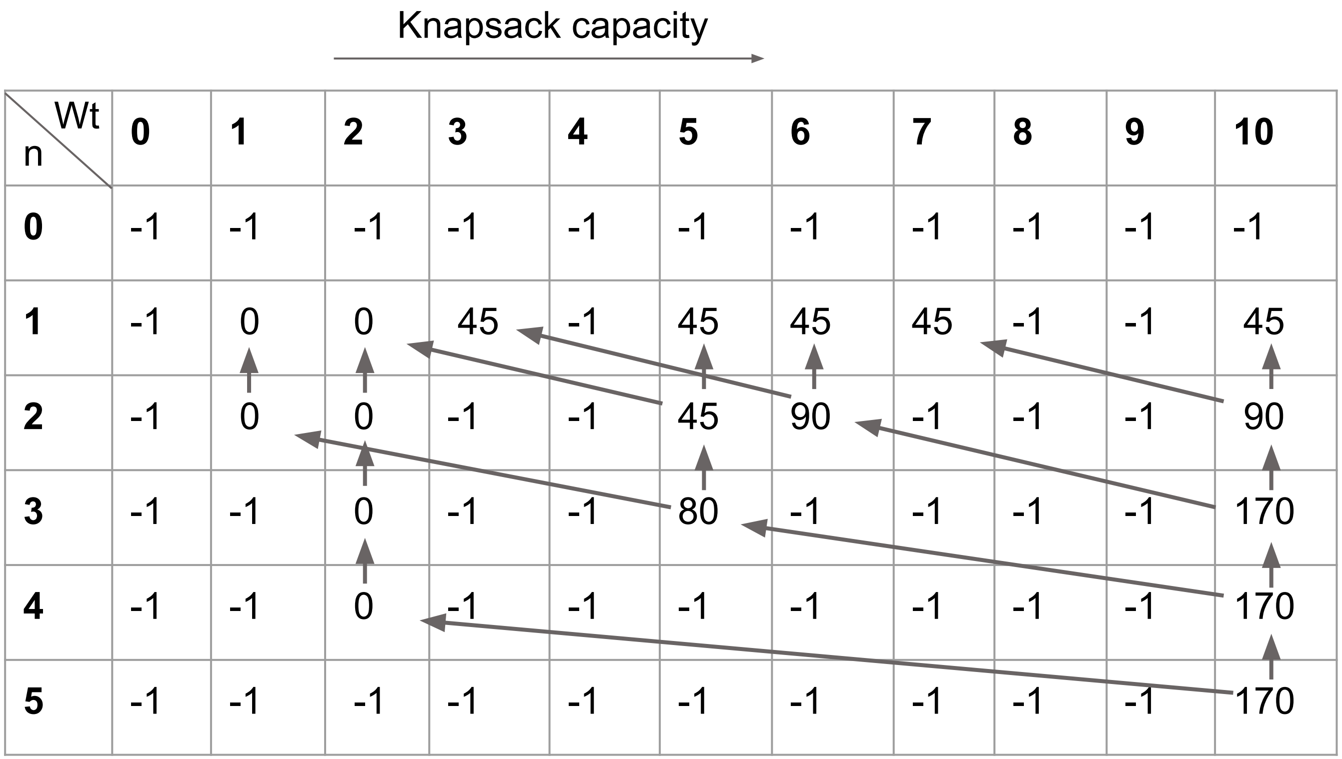
Item	Value ($)	Weight (kg)
1	45	3
2	45	3
3	80	4
4	80	5
5	100	8



Memoisation cache values

If use a table intialised to -1 as cache and print it at end, get ...

Item	Value ($)	Wt (kg)
1	45	3
2	45	3
3	80	4
4	80	5
5	100	8



n \ Wt	0	1	2	3	4	5	6	7	8	9	10
0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
1	-1	0	0	45	-1	45	45	45	-1	-1	45
2	-1	0	0	-1	-1	45	90	-1	-1	-1	90
3	-1	-1	0	-1	-1	80	-1	-1	-1	-1	170
4	-1	-1	0	-1	-1	-1	-1	-1	-1	-1	170
5	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	170

NB: this is just the values of all nodes in the call graph.

$$V(i,j) = \max\ \{V(i-1,j),\ \ V(i-1,\ j-w_i)+v_i\}$$

0/1 Knapsack: **bottom-up DP**

Just fill out the whole table row-wise, from n = 0.

Item	Value ($)	Wt (kg)
1	45	3
2	45	3
3	80	4
4	80	5
5	100	8

Knapsack capacity →

Wt n	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	45	45	45	45	45	45	45	45
2	0	0	0	45	45	45	90	90	90	90	90
3	0	0	0	45	80	80	90	125	125	125	170
4	0	0	0	45	80	80	90	125	125	160	170
5	0	0	0	45	80	80	90	125	125	160	170

$$V(i,j) = \max\ \{V(i-1,j),\ \ V(i-1,\ j-w_i)+v_i\}$$

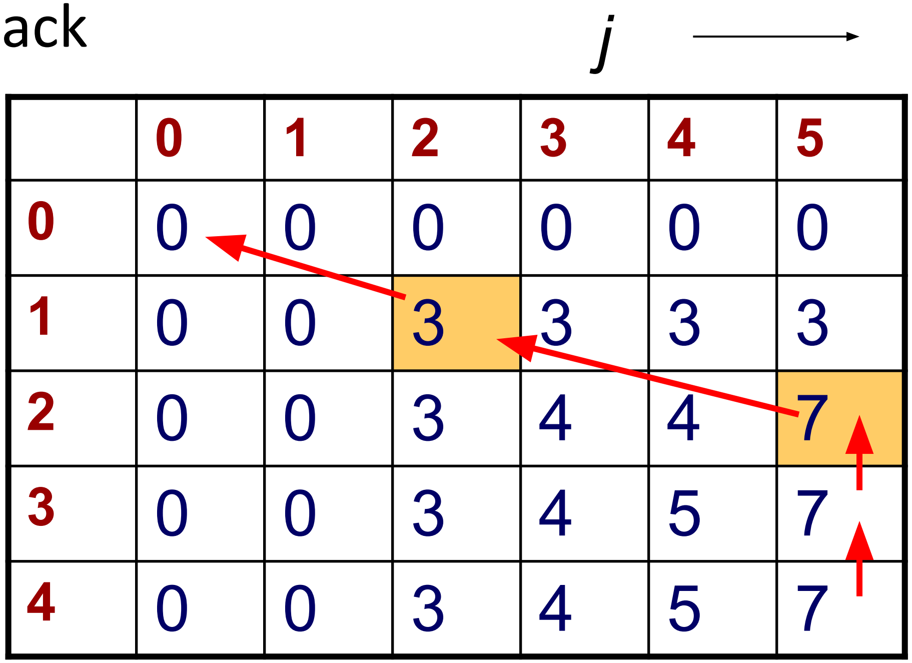
Complexity, given *n* items and capacity *W*, is O(*nW*)

Back-tracking to get the solution

```
Start from C[n][W]  (i=n,  j=W)
while i, j > 0:
  if C(i, j) ≠ C(i−1 , j):
    item i was added to the knapsack
    j = j −w_i
  i = i −1
```

Complexity: $O(n)$

v_i	w_i
3	2
4	3
5	4
6	5



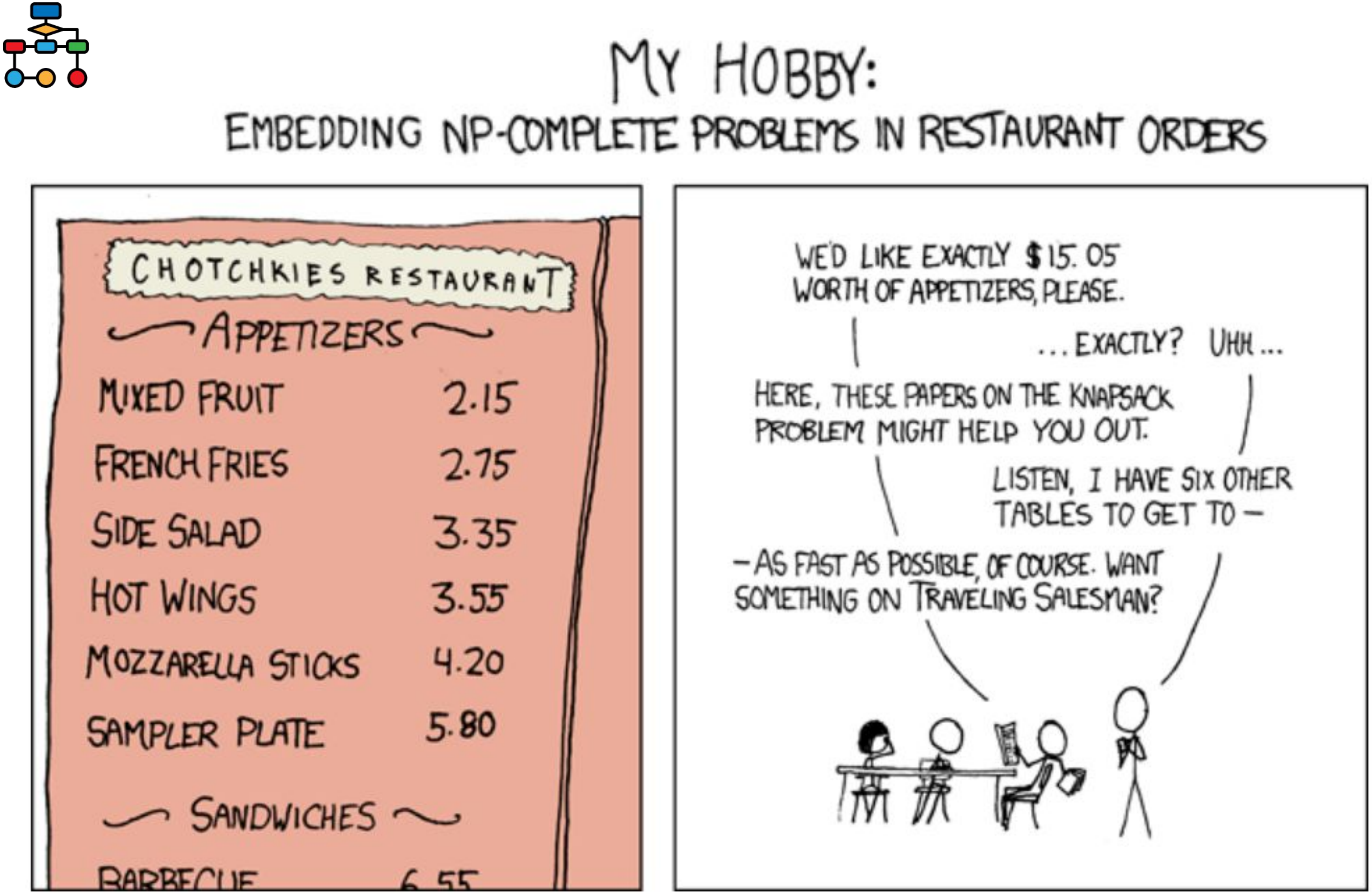
	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$$C(i, j) = \max\ \{C(i-1, j),\ \ C(i-1,\ j - w_i) + v_i\ \}$$

Items added: 1, 2

Notes on complexity

- Time complexity, given n items and capacity W, is $\Theta(nW)$
- Space complexity also $\Theta(nW)$
- BUT with bottom-up method, if we only want the maximum value, not the items picked up, we need keep only the last 2 rows => space complexity $\Theta(W)$
- The algorithm only works with weights that are **integers** and **sufficiently small**.
- The algorithm is said to run in ***pseudo-polynomial time*** as it's polynomial in the capacity of the knapsack not just the number of items.


MY HOBBY:
EMBEDDING NP-COMPLETE PROBLEMS IN RESTAURANT ORDERS
CHOTCHKIES RESTAURANT
APPETIZERS
MIXED FRUIT 2.15
FRENCH FRIES 2.75
SIDE SALAD 3.35
HOT WINGS 3.55
MOZZARELLA STICKS 4.20
SAMPLER PLATE 5.80
SANDWICHES
BARBECUE 6.55
WE'D LIKE EXACTLY $15.05 WORTH OF APPETIZERS, PLEASE.
...EXACTLY? UHH...
HERE, THESE PAPERS ON THE KNAPSACK PROBLEM MIGHT HELP YOU OUT.
LISTEN, I HAVE SIX OTHER TABLES TO GET TO –
– AS FAST AS POSSIBLE, OF COURSE. WANT SOMETHING ON TRAVELING SALESMAN?

https://xkcd.com/287/

A challenge: the subset sum problem

Another classic DP problem

Given a set of integers, positive and negative, can you find a non-empty subset that sums to zero?

Similar to 0/1 knapsack.

Hint: Like 0/1 knapsack and many other DP problems, the approach is to ***generalise***.

In this case to: "Can you find a subset that sums *to a value S*?".

Step 1: write a recursive implementation of the boolean function

```
subset_sum_exists(numbers, n, required_sum)
```

which returns true iff there exists a subset within the first *n* elements of numbers that sums to *required_sum*

You do the rest :-)

Longest Common Subsequence (LCS)

Problem statement:

Given two strings *A* and *B* of lengths *n* and *m* respectively, determine the longest subsequence (LCS) that is common to both *A* and *B*.

(Elements in a subsequence do not need to be contiguous)

Example:

A = `glorious`

B = `algorithms`

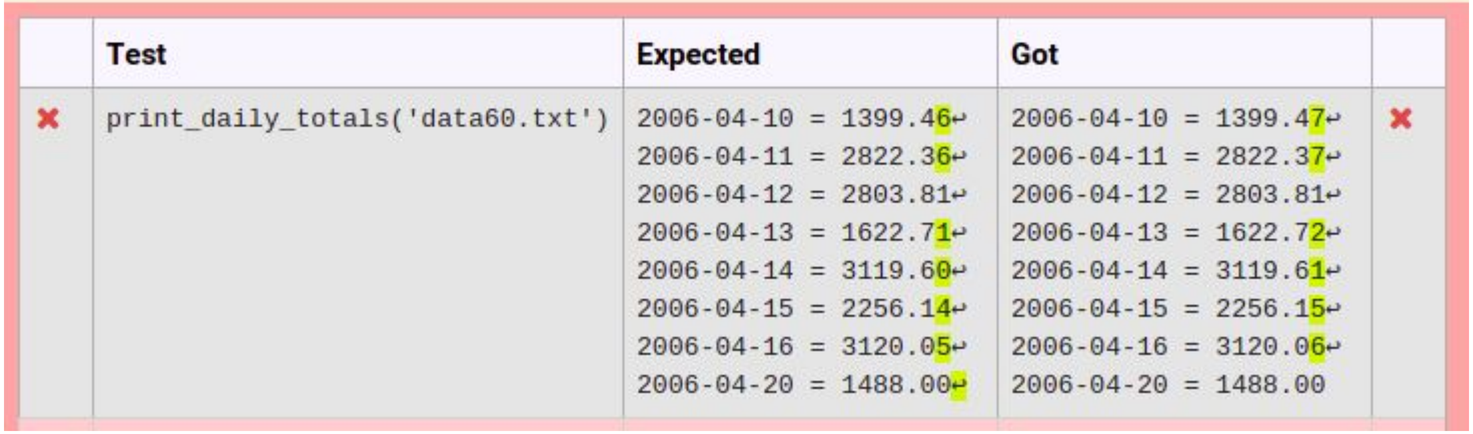
Longest common subsequence: `goris`

More than one solution may exist: `loris`

Example use

Quiz server “Show differences” button.

All text not in the LCS of Expected and Got is highlighted.



	Test	Expected	Got	
✗	print_daily_totals('data60.txt')	2006-04-10 = 1399.46↵ 2006-04-11 = 2822.36↵ 2006-04-12 = 2803.81↵ 2006-04-13 = 1622.71↵ 2006-04-14 = 3119.60↵ 2006-04-15 = 2256.14↵ 2006-04-16 = 3120.05↵ 2006-04-20 = 1488.00↵	2006-04-10 = 1399.47↵ 2006-04-11 = 2822.37↵ 2006-04-12 = 2803.81↵ 2006-04-13 = 1622.72↵ 2006-04-14 = 3119.61↵ 2006-04-15 = 2256.15↵ 2006-04-16 = 3120.06↵ 2006-04-20 = 1488.00	✗

LCS: How to solve?

Brute-force method:

- Enumerate 2^n-1 subsequences of *A*, and for each subsequence, determine if it is also a subsequence of *B* in $O(m)$ time.
- Clearly, the running time of the algorithm is $O(m2^n)$, which is exponential.

But it's a combinatorial optimisation problem so perhaps solvable by DP if:

- We can find a recursive solution, and ...
- the recursion involves integer parameters over a small enough domain

LCS: a simple recursive solution

```python
def lcs(s1, s2):
    if s1 == '' or s2 == '':
        return ''
    elif s1[-1] == s2[-1]: # Last chars match
        return lcs(s1[:-1], s2[:-1]) + s1[-1]
    else:
        # Drop last char of each string in turn.
        # Choose best outcome.
        soln1 = lcs(s1[:-1], s2)
        soln2 = lcs(s1, s2[:-1])
        if len(soln1) > len(soln2):
            return soln1
        else:
            return soln2
```

Exercise (lab quiz): add memoisation

Properties of the recursive LCS program

We need to memoise it to avoid exponential complexity. Then:

- Cache need n^2 entries (assuming both strings of length n).
  - That’s ok (maybe).
- But:
  - Every call involves slicing a string
  - Slicing is O(n) in time
  - So algorithm is $O(n^3)$ in time => slow
- And:
  - Cache has to store a string value for each key *and* key is a pair of strings too
  - O(n) memory reqd per value (and n^2 values)
  - So $O(n^3)$ in space, too => gigabytes reqd for kilobyte strings

Keeping cache size manageable

- DP problems are always limited by the size of the cache (a.k.a. DP table).
  - If it's too big, it's too slow to compute *and* takes too much memory.
- So we don't want to store structured data like strings or lists in it.
- Hence, we usually need to re-cast the DP problem in a form with an integer answer.
  - "What is the maximum value we can get into a knapsack?"
- We then use a trace-back through the table to get the answer we really want.
  - "What is the best set of items to put in our knapsack?"

Re-casting the LCS problem in *int* form

Change the problem statement to *find the **length** of the LCS*

- We'll come back to finding the actual LCS itself later

Let $L_{i,j}$ denote the *length* of the LCS of

$$A = a_1\ a_2\ a_3 \ldots a_i$$
$$B = b_1\ b_2\ b_3 \ldots b_j$$

Note 1-origin subscripts. When using Python, *i* and *j* are the **lengths** of the substrings.

If *i*=0, or *j*=0, then $L_{i,j} = 0$.

If $a_i = b_j$, this char is in LCS. Recurse on the rest.

Otherwise choose best outcome of discarding either a_i or b_j

$L_{i,j}$ recurrence

The recurrence relation for $L_{i,j}$ is thus:

$$L_{i,j} = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ L_{i-1,j-1} + 1 & a_i = b_j \\ max(L_{i,j-1}, L_{i-1,j}) & a_i \neq b_j \end{cases}$$

Case 1: base case

Case 2: last characters same, so both are in the LCS

Case 3: discard each of a_i and b_j in turn, recurse on what's left, pick best of the two outcomes.

Exercise: write a top-down DP solution for this.

Graph of recursive calls

A: "them"

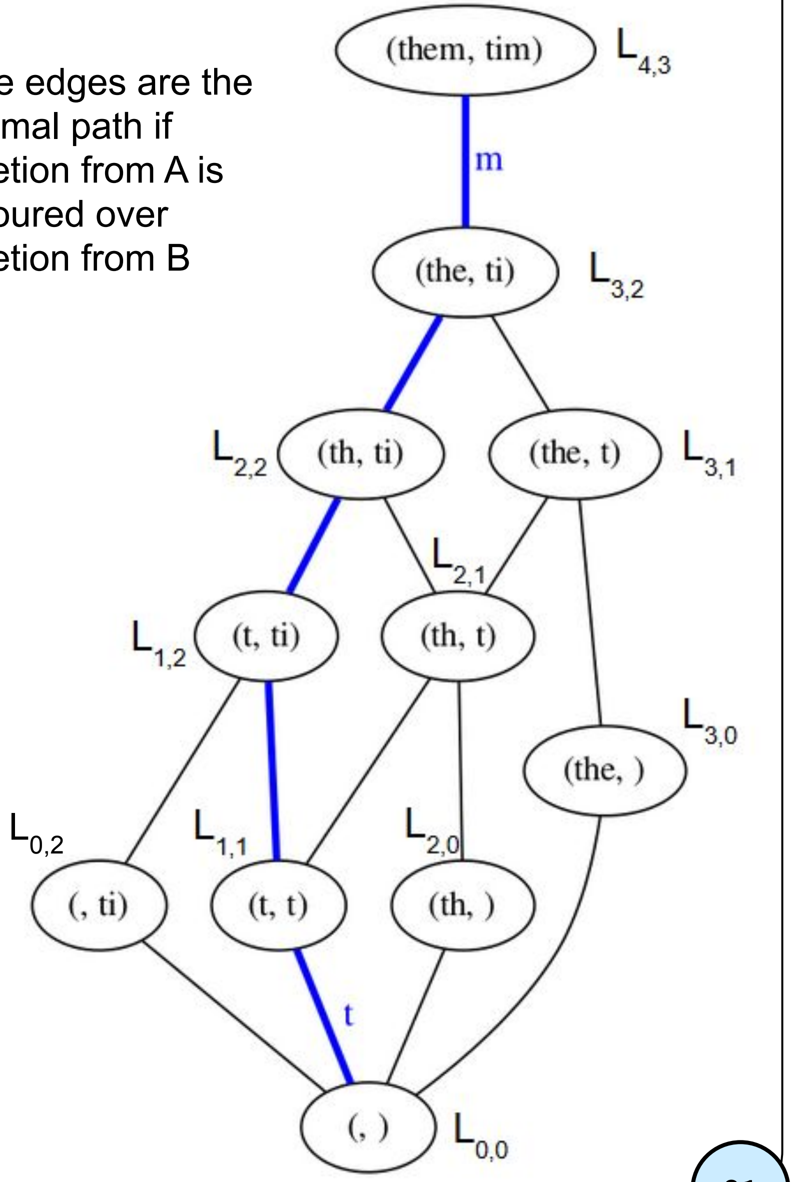
B: "tim"

LCS = "tm"

Note overlapping subproblems.

Q: what are the *values* of all $L_{i,j}$?

Blue edges are the optimal path if deletion from A is favoured over deletion from B



$$L_{i,j} = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ L_{i-1,j-1} + 1 & a_i = b_j \\ max(L_{i,j-1}, L_{i-1,j}) & a_i \neq b_j \end{cases}$$

LCS: bottom-up DP approach

Build a table of $L_{i,j}$ row by row, by using the recurrence relation.

$$\begin{array}{cccccc} & 1 & 2 & 3 & 4 & \\ A = & t & h & e & m & (n{=}4) \end{array}$$

$$\begin{array}{ccccc} & 1 & 2 & 3 & \\ B = & t & i & m & (m{=}3) \end{array}$$

$$L_{i,j} = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ L_{i-1,j-1} + 1 & a_i = b_j \\ max(L_{i,j-1}, L_{i-1,j}) & a_i \neq b_j \end{cases}$$

Remember: 1-origin subscripts, hence *lengths*.

$j \rightarrow$

$i \downarrow$

			t	i	m
		0	1	2	3
	0	0	0	0	0
t	1	0	1	1	1
h	2	0	1	1	1
e	3	0	1	1	1
m	4	0	1	1	2

Another (longer) example

A = *x y x x z x* (*n*=6)
1 2 3 4 5 6

B = *z x z y y z x x* (*m*=8)
1 2 3 4 5 6 7 8

$$L_{i,j} = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ L_{i-1,j-1} + 1 & a_i = b_j \\ max(L_{i,j-1}, L_{i-1,j}) & a_i \neq b_j \end{cases}$$

j →

i ↓

			z	*x*	*z*	*y*	*y*	*z*	*x*	*x*
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
	0	0	0	0	0	0	0	0	0	0
x	**1**	0								
y	**2**	0								
x	**3**	0								
x	**4**	0								
z	**5**	0								
x	**6**	0								

What are all the values in the empty squares?

Answer

A = *x y x x z x* (*n*=6)
1 2 3 4 5 6

B = *z x z y y z x x* (*m*=8)
1 2 3 4 5 6 7 8

$$L_{i,j} = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ L_{i-1,j-1} + 1 & a_i = b_j \\ max(L_{i,j-1}, L_{i-1,j}) & a_i \neq b_j \end{cases}$$

j →

				z	*x*	*z*	*y*	*y*	*z*	*x*	*x*
			0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
		0	0	0	0	0	0	0	0	0	0
	x	**1**	0	0	1	1	1	1	1	1	1
	y	**2**	0	0	1	1	2	2	2	2	2
i ↓	*x*	**3**	0	0	1	1	2	2	2	3	3
	x	**4**	0	0	1	1	2	2	2	3	4
	z	**5**	0	1	1	2	2	2	3	3	4
	x	**6**	0	1	2	2	2	2	3	4	4

Lots of words

Top down DP is often easier to write down and think about, at least for people who have got the hang of recursion. It works well for fairly small problems where the recursion depth limit isn't exceeded. But there's a further constraint that applies to both top down and bottom up DP: the total cache (or DP table) size must be *sufficiently small*.

The term "sufficiently small" is a bit tricky because it involves both time and space. A 1500 x 1500 table of *int*s is sufficiently small to compute in a relatively small amount of time (say less than a couple of seconds in Python) and is certainly small enough to fit in memory (a few megabytes). However a 1500 x 1500 table of *strings* is likely to be problematic if strings can be a 1k or more in length. Now we're talking *gigabytes* of RAM. Computation time is likely to be much higher, too, as string slicing or copying is taking place for each cell computation. The problem becomes much worse again if the table elements are longish lists of strings (lines).

So ... with larger problems we **must** use caches/tables of a simple data type, e.g. *int*. This is what you get anyway if the function type is *int*, e.g. "compute the *length* of the longest common subsequence" rather than "compute the longest common subsequence. But if you're asked for the actual best solution (e.g. the longest common subsequence) you have a problem. You can't store the actual result in the table because the table will be too big. So you must do it in two steps: compute the table of lengths, then trace backwards in the table to determine the actual best solution.

It would be possible to run a memoised recursive solution to the problem "compute the *length* of the longest common subsequence", then inspect the cache and trace back to get the solution but it's usually much easier and faster to switch to a bottom-up DP approach. That also avoids the possibility of exceeding the recursion depth since there's now no recursion taking place.

Retrieving the actual sequence

The length of the longest common subsequence is given by $L_{n,m}$ (= 4 in the previous example)

The subsequence can be retrieved from the table by tracing back from the final cell:

Start from position [*n*, *m*].

Initialize subsequence *S* to empty.

If, at [*i*, *j*], $a_i = b_j$, then add a_i to the beginning of *S*, and move to position [*i*-1, *j*-1].

If $a_i \neq b_j$, move to whichever of [*i*-1, *j*] or [*i*, *j*-1] contains the maximum value.

OR: build a separate predecessor table, as for *giving change* problem

Longest Common Subsequence

B

				z	*x*	*z*	*y*	*y*	*z*	*x*	*x*
			0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
		0	0	0	0	0	0	0	0	0	0
	x	**1**	0	0	1	1	1	1	1	1	1
	y	**2**	0	0	1	1	2	2	2	2	2
A	*x*	**3**	0	0	1	1	2	2	2	3	3
	x	**4**	0	0	1	1	2	2	2	3	4
	z	**5**	0	1	1	2	2	2	3	3	4
	x	**6**	0	1	2	2	2	2	3	4	4

Solution: *x y x x*

Longest Common Subsequence

Another Solution:

B

			z	*x*	*z*	*y*	*y*	*z*	*x*	*x*
A		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
	0	0	0	0	0	0	0	0	0	0
x	**1**	0	0	1	1	1	1	1	1	1
y	**2**	0	0	1	1	2	2	2	2	2
x	**3**	0	0	1	1	2	2	2	3	3
x	**4**	0	0	1	1	2	2	2	3	4
z	**5**	0	1	1	2	2	2	3	3	4
x	**6**	0	1	2	2	2	2	3	4	4

Solution: *x y z x*

←

LCS: Complexity, Applications

An optimal solution to the LCS problem can be found in *O*(*nm*) time and space.

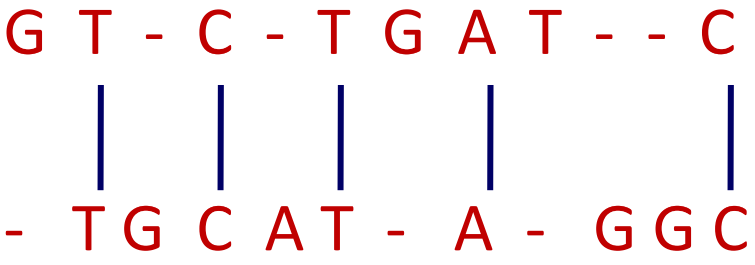
LCS problem has many applications, e.g.

- Text processing ("Show differences" button)
- Bioinformatics - DNA sequence comparison and alignment:

String 1: GTCTGATC

String 2: TGCATAGGC

Longest Common Sequence: TCTAC



Longest **Increasing** Subsequence (LIS)

Used in *Patience diff* algorithm - an improved (?) *diff* (see next slide)

Problem: given a sequence of some orderable objects, find the longest increasing subsequence within it.

A simple LCS solution: create another sequence by sorting the elements in ascending order (after removing duplicates), and find the longest common subsequence between the two sequences.

Example:

```
Seq1: 3  5  2  0  7  5  11  4  6     8
Seq2: 0  2  3  4  5  5    6  7  8  11
```

$O(n^2)$ in both space and time

Longest increasing subsequence: 0 4 6 8 (or 0 5 6 8 or 3 5 7 11)

Easy. But … we'll return to this problem soon, with a better solution.

The Linux *diff* program

original:

```
This part of the
document has stayed the
same from version to
version.  It shouldn't
be shown if it doesn't
change.  Otherwise, that
would not be helping to
compress the size of the
changes.

This paragraph contains
text that is outdated.
It will be deleted in the
near future.

It is important to spell
check this dokument. On
the other hand, a
misspelled word isn't
the end of the world.
Nothing in the rest of
this paragraph needs to
be changed. Things can
be added after it.
```

new:

```
This is an important
notice! It should
therefore be located at
the beginning of this
document!

This part of the
document has stayed the
same from version to
version.  It shouldn't
be shown if it doesn't
change.  Otherwise, that
would not be helping to
compress the size of the
changes.

It is important to spell
check this document. On
the other hand, a
misspelled word isn't
the end of the world.
Nothing in the rest of
this paragraph needs to
be changed. Things can
be added after it.

This paragraph contains
important new additions
to this document.
```

The command **diff original new** produces the following *normal diff output*:

```
0a1,6
> This is an important
> notice! It should
> therefore be located at
> the beginning of this
> document!
>
11,15d16
< This paragraph contains
< text that is outdated.
< It will be deleted in the
< near future.
<
17c18
< check this dokument. On
---
> check this document. On
24a26,29
>
> This paragraph contains
> important new additions
> to this document.
```

https://en.wikipedia.org/wiki/Diff

Edit distance: a close relative of LCS

The *diff* algorithm

We have to transform a source string $A = a_1 \ldots a_n$ into a target string $B = b_1 \ldots b_m$ using a minimum number of operations of the following types. This number is the **edit distance** between the two strings.

- Insertion
- Deletion
- Substitution ← LCS doesn’t consider this possibility

We assign a unit cost to each of the above operations, and use dynamic programming to find the minimum cost $D(n, m)$.

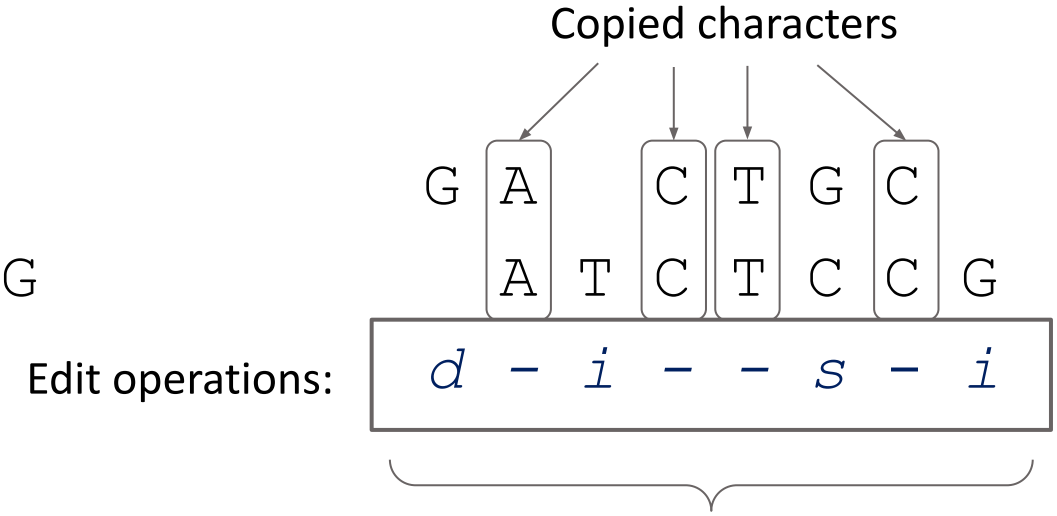
As for LCS, we’re looking for a recurrence that solves for an integer value - the edit distance. We’ll find the sequence of edit operations by backtracking through the decision tree.

Edit distance example

What is the cost of editing A -> B?

$A = \texttt{GACTGC}$

$B = \texttt{ATCTCCG}$



	G	A		C	T	G	C	
		A	T	C	T	C	C	G
Edit operations:	*d*	–	*i*	–	–	*s*	–	*i*

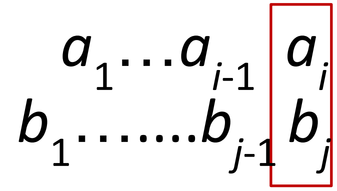
Edit distance = 4

Note that there are 2 LCS solutions: ACTC and ACTG. But the latter would lead to an edit distance of 5. LCS alone is not a sufficient algorithm.

Finding the recurrence

Let $D_{i,j}$ denote the edit distance between two sub-strings of A and B given by $a_1 \ldots a_i$ and $b_1 \ldots b_j$.

If $a_i = b_j$, then $D_{i,j} = D_{i-1,j-1}$ (Alignment/Copy)



$a_1 \ldots a_{i-1}\ \boxed{\begin{matrix} a_i \\ b_j \end{matrix}}$

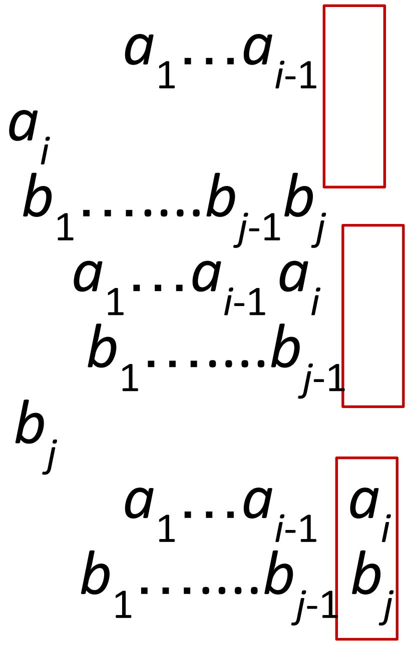
$b_1 \ldots\ldots b_{j-1}$

If $a_i \neq b_j$, we have to consider 3 cases:

$D_{i,j} = D_{i-1,j} + 1$ (Deletion)

$D_{i,j} = D_{i,j-1} + 1$ (Insertion)

$D_{i,j} = D_{i-1,j-1} + 1$ (Substitution)



$a_1 \ldots a_{i-1}$

a_i

$b_1 \ldots\ldots b_{j-1} b_j$

$a_1 \ldots a_{i-1} a_i$

$b_1 \ldots\ldots b_{j-1}$

b_j

$a_1 \ldots a_{i-1}\ a_i$

$b_1 \ldots\ldots b_{j-1}\ b_j$

The edit distance recurrence

Recurrence relations:

$D_{0,0} = 0$ Base case - two empty strings

Redundant

$D_{i,0} = i$ Delete all *i* characters from A

$D_{0,j} = j$ Insert all *j* characters of B into A

Warning: 1-origin, so *i* and *j* are string **lengths** in Python

If $a_i = b_j$, $D_{i,j} = D_{i-1,\, j-1}$ (Alignment/Copy)

else

$$D_{i,j} = 1 + \min \begin{cases} D_{i-1,j} & \text{(Deletion)} \\ D_{i,\, j-1} & \text{(Insertion)} \\ D_{i-1,\, j-1} & \text{(Substitution)} \end{cases}$$

Exercise: how does the equation for Di,j change if the different operations have different costs?

Edit distance - recursive code

```
def cost(a, b, na=None, nb=None):
    """ Cost of converting first na chars of a into first nb chars of b"""
    if na is None:  # Top level call - both na and nb will be None
        na = len(a)
        nb = len(b)
    if na == 0 or nb == 0:
        # One string is empty - n insertions or deletions reqd
        return max(na, nb)
    elif a[na - 1] == b[nb - 1]:  # Do last chars match?
        return cost(a, b, na - 1, nb - 1)  # Yes - this is the align/copy case
    else:  # Last chars don't match
        # Must delete last a, insert last b or replace last a with last b
        delete_cost = 1 + cost(a, b, na - 1, nb)
        insert_cost = 1 + cost(a, b, na, nb - 1)
        replace_cost = 1 + cost(a, b, na - 1, nb - 1)
        return min([delete_cost, insert_cost, replace_cost])
```

```
>>> print(cost("GACTGC", "ATCTCCG"))
4
```

Top-down DP: you do (i.e. add caching)

Graph of recursion

a = "them", b = "tim"

Compare with LCS graph

Note extra edges for substitution

Edge labels:

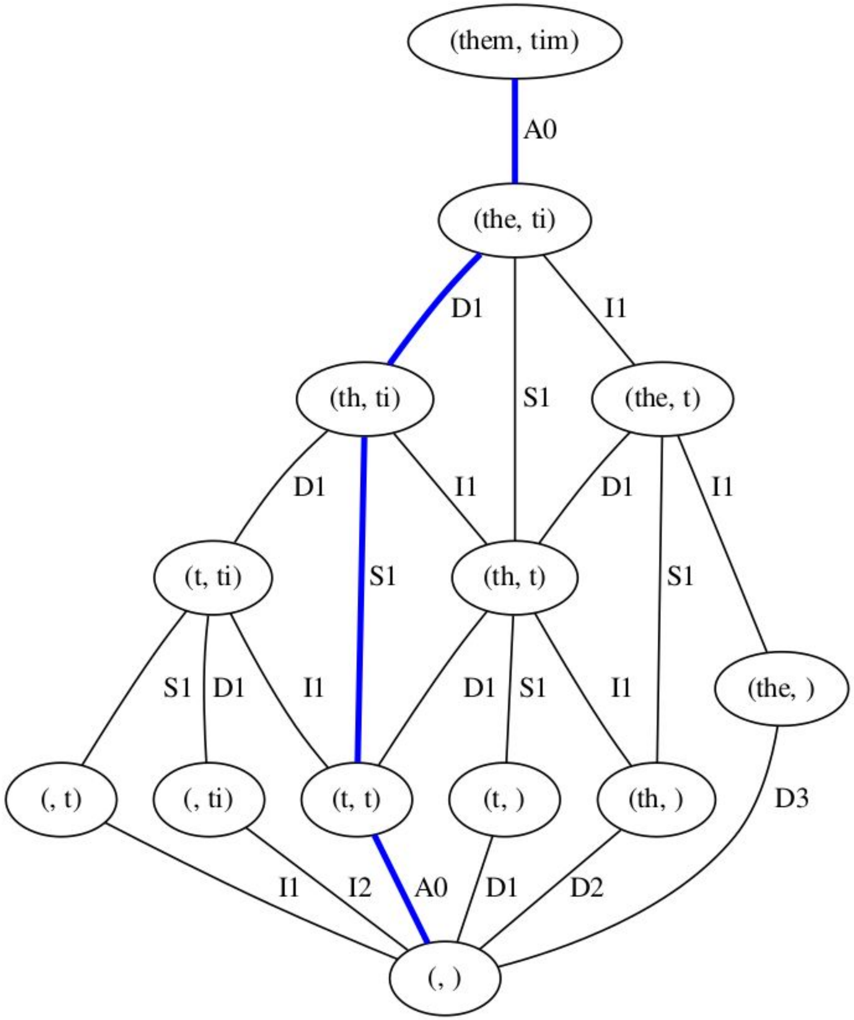
A = Align/Copy

D = Delete

I = Insert

S = Substitute

Followed by cost



Q: What is the edit distance from "them" to "tim"?

Bottom-up DP: fill out the table

$$D_{i,j} = \begin{cases} 0 & i = j = 0 \\ i & j = 0 \\ j & i = 0 \\ D_{i-1,j-1} & a_i = b_j \\ 1 + min(D_{i-1,j}, D_{i,j-1}, D_{i-1,j-1}) & otherwise \end{cases}$$

j →

i ↓

			t	*i*	*m*
		0	**1**	**2**	**3**
	0	0	1	2	3
t	**1**	1	0	1	2
h	**2**	2	1	1	2
e	**3**	3	2	2	2
m	**4**	4	3	3	2

Exercise: fill in the table

$$D_{i,j} = \begin{cases} 0 & i = j = 0 \\ i & j = 0 \\ j & i = 0 \\ D_{i-1,j-1} & a_i = b_j \\ 1 + min(D_{i-1,j}, D_{i,j-1}, D_{i-1,j-1}) & otherwise \end{cases}$$

n = 6
m = 7

$j \longrightarrow$

$i \downarrow$

			A	*T*	*C*	*T*	*C*	*C*	*G*
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
	0								
G	**1**								
A	**2**								
C	**3**								
T	**4**								
G	**5**								
C	**6**								

The answer

$$D_{i,j} = \begin{cases} 0 & i = j = 0 \\ i & j = 0 \\ j & i = 0 \\ D_{i-1,j-1} & a_i = b_j \\ 1 + min(D_{i-1,j}, D_{i,j-1}, D_{i-1,j-1}) & otherwise \end{cases}$$

n = 6
m = 7

j →

i ↓

			A	*T*	*C*	*T*	*C*	*C*	*G*
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
	0	0	1	2	3	4	5	6	7
G	**1**	1	1	2	3	4	5	6	6
A	**2**	2	1	2	3	4	5	6	7
C	**3**	3	2	2	2	3	4	5	6
T	**4**	4	3	2	3	2	3	4	5
G	**5**	5	4	3	3	3	3	4	4
C	**6**	6	5	4	3	4	3	3	4

Edit Distance: trace-back

Trace-back: Start from the cell $D_{n,m}$. Repeatedly: check if $a_i = b_j$. If so, it's an Align/Copy operation. Move to (i - 1, j - 1). Otherwise, find minimum of neighbours (i-1, j), (i-1, j-1), (i, j-1).

Move to that cell, recording the operation as follows:

(i-1, j) : Deletion
(i, j-1) : Insertion
(i-1, j-1) : Substitution

j

i

			A	T	C	T	C	C	G
		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
G	1	1	1	2	3	4	5	6	6
A	2	2	1	2	3	4	5	6	7
C	3	3	2	2	2	3	4	5	6
T	4	4	3	2	3	2	3	4	5
G	5	5	4	3	3	3	3	4	4
C	6	6	5	4	3	4	3	3	4

cell (i, j-1) : Insertion

G	A		C	T	G	C	
	A	T	C	T	C	C	G
D		I			S		I

Another trace-back solution

If substitution is preferred over insertion we get:

			A	T	C	T	C	C	G
		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
G	1	1	1	2	3	4	5	6	6
A	2	2	1	2	3	4	5	6	7
C	3	3	2	2	2	3	4	5	6
T	4	4	3	2	3	2	3	4	5
G	5	5	4	3	3	3	3	4	4
C	6	6	5	4	3	4	3	3	4

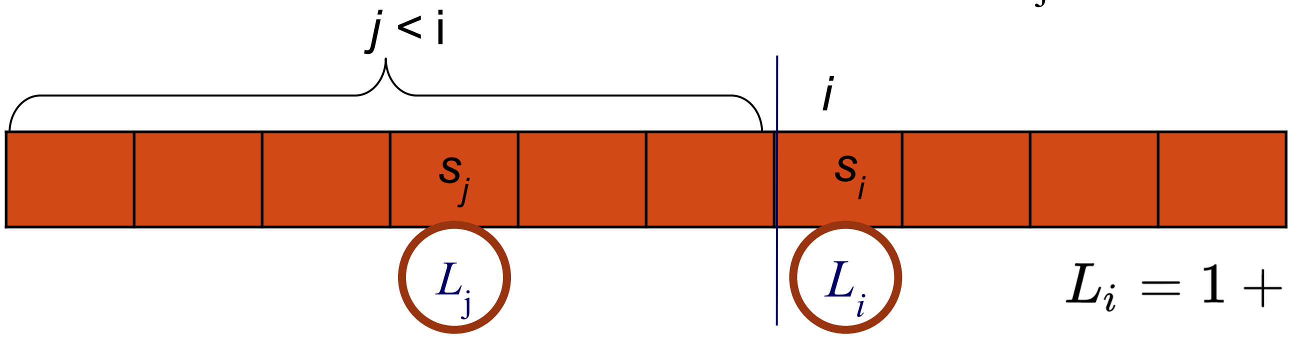
G	A	C	T	G	C	
A	T	C	T	C	C	G
S	S			S		I

Longest Increasing Subsequence (LIS): a better solution

The LCS solution to the Longest Increasing Subsequence problem requires an $(n+1)$ x $(n+1)$ table for computing the optimal subsequence. $O(n^2)$ in both space and time.

Can we formulate a recurrence relation for the subsequence using only the given sequence, and a one-dimensional table?

- Let the sequence be $s_1 s_2 \ldots s_n$
- Let L_j be the length of the longest increasing subsequence up to position *j* **that includes** s_j.
- For a given *i*, we search in the region *j* < *i*, and select *all* $s_j < s_i$, and take the maximum of the corresponding L_j values and add 1.



$$L_i = 1 + \max_{j<i, s_j<s_i} (L_j)$$

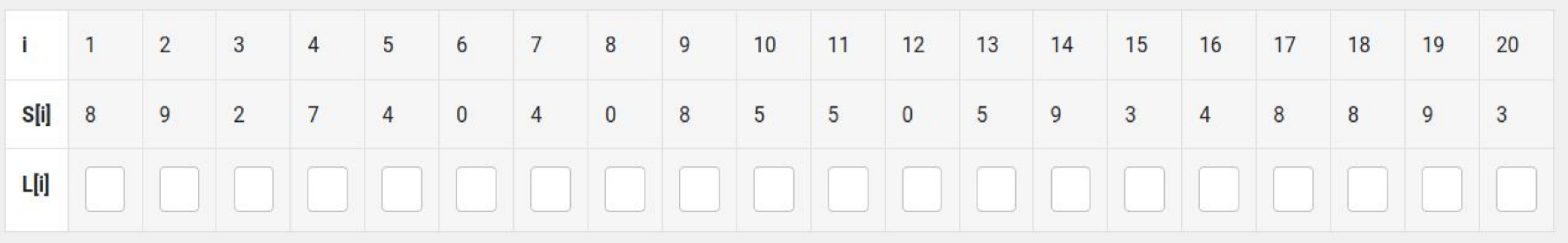
LIS Recurrence

We can write the following recurrence relation for L_i:

$$L_i = \begin{cases} 1 & i = 1 \text{ or } s_j >= s_i \text{ for all } 0 < j < i \\ 1 + \max\limits_{0<j<i, s_j<s_i} (L_j) & otherwise \end{cases}$$

Warning: 1-origin subscripts.

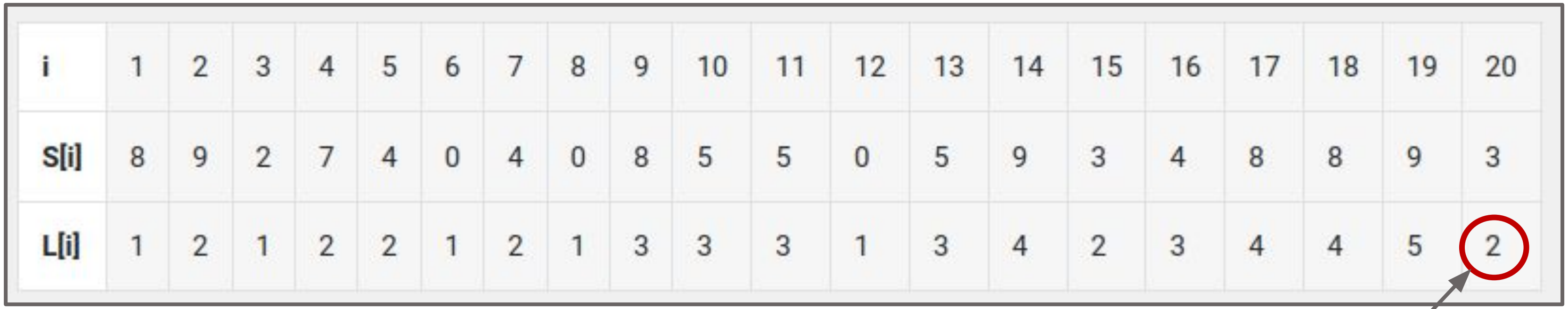
Example: *int* sequence = 8 9 2 7 4 0 4 0 8 5 5 0 5 9 3 4 8 8 9 3



i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
S[i]	8	9	2	7	4	0	4	0	8	5	5	0	5	9	3	4	8	8	9	3
L[i]	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□

Exercise: what are all the L[i] values?

And the answer is ...



i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
S[i]	8	9	2	7	4	0	4	0	8	5	5	0	5	9	3	4	8	8	9	3
L[i]	1	2	1	2	2	1	2	1	3	3	3	1	3	4	2	3	4	4	5	2

The winner! Best sequence has length 5

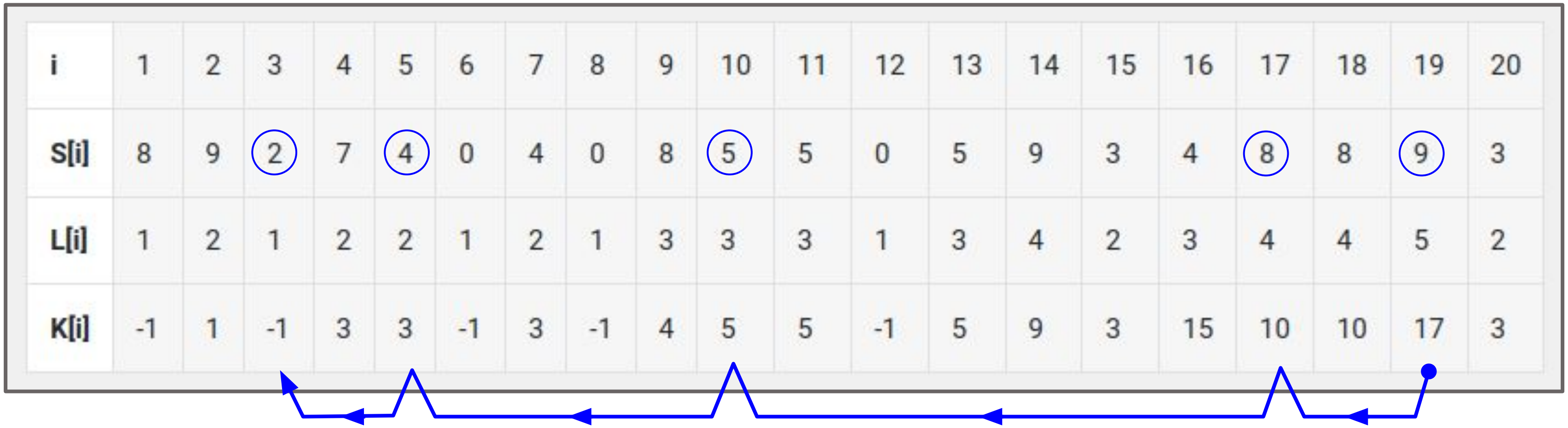
NB: have to search L to find the winner. It’s not generally at the end.

$$L_i = \begin{cases} 1 & i = 1 \text{ or } s_j >= s_i \text{ for all } 0 < j < i \\ 1 + \max\limits_{0<j<i, s_j<s_i} (L_j) & otherwise \end{cases}$$

Warning: 1-origin subscripts.

Back-tracing to get the longest seq

To simplify back-tracing, define K[i] to be the value of j that yielded the maximum when computing L[i]. This is the *predecessor*. -1 denotes “no predecessor”.



i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
S[i]	8	9	2	7	4	0	4	0	8	5	5	0	5	9	3	4	8	8	9	3
L[i]	1	2	1	2	2	1	2	1	3	3	3	1	3	4	2	3	4	4	5	2
K[i]	-1	1	-1	3	3	-1	3	-1	4	5	5	-1	5	9	3	15	10	10	17	3

So best sequence is [2, 4, 5, 8, 9]

An *n* log *n* solution to LIS (enthusiasts only - not part of course)

First LIS algorithm (using LCS) was O(n^2) in both space and time.

The improved algorithm is O(n) in space but still O(n^2) in time.

Can we do better?

Answer: yes.

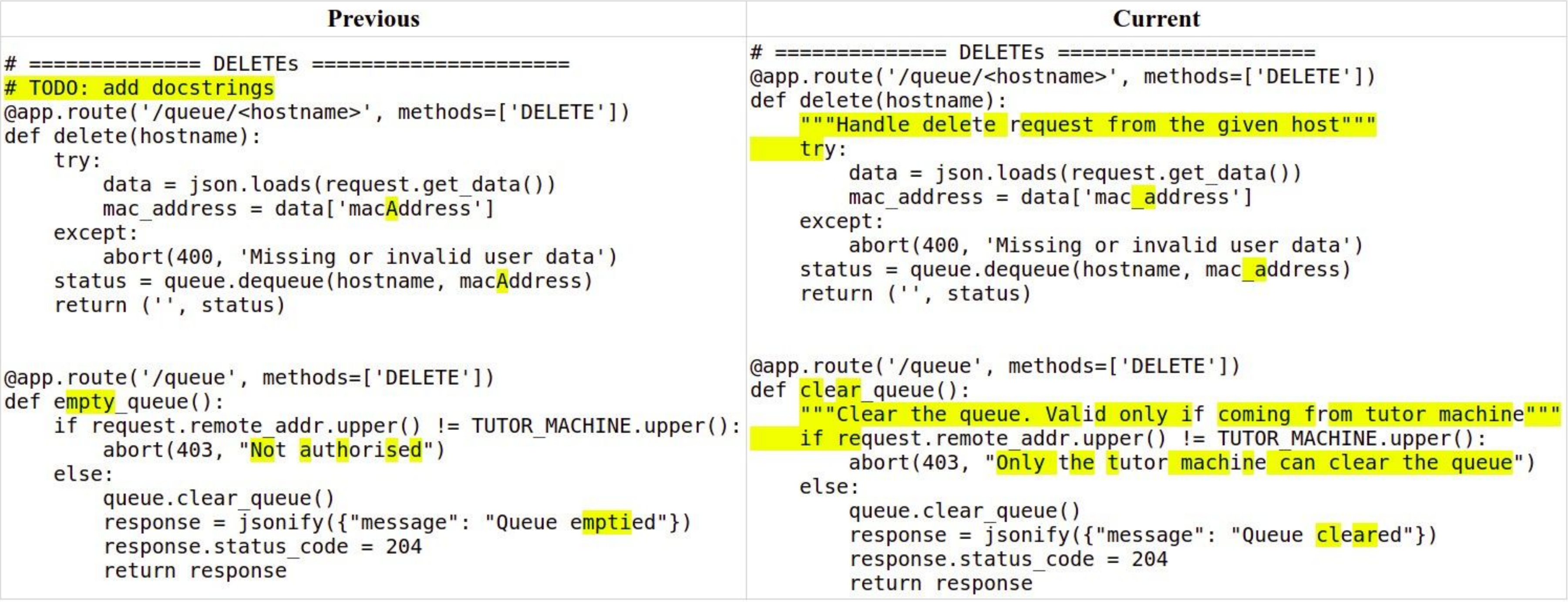
It turns out that the last elements of the optimal sequences of lengths 1, 2, 3, … are in increasing order at all times. So to make best use of each new S_i we can binary search in a list of such terminators rather than searching all S_j , $j < i$.

Gives O(n log n) in time.

See https://en.wikipedia.org/wiki/Longest_increasing_subsequence

Intro to assignment 2

Quiz 9, Q3. LCS, top-down DP



Previous

```
# ============== DELETEs =====================
# TODO: add docstrings
@app.route('/queue/<hostname>', methods=['DELETE'])
def delete(hostname):
    try:
        data = json.loads(request.get_data())
        mac_address = data['macAddress']
    except:
        abort(400, 'Missing or invalid user data')
    status = queue.dequeue(hostname, macAddress)
    return ('', status)


@app.route('/queue', methods=['DELETE'])
def empty_queue():
    if request.remote_addr.upper() != TUTOR_MACHINE.upper():
        abort(403, "Not authorised")
    else:
        queue.clear_queue()
        response = jsonify({"message": "Queue emptied"})
        response.status_code = 204
        return response
```

Current

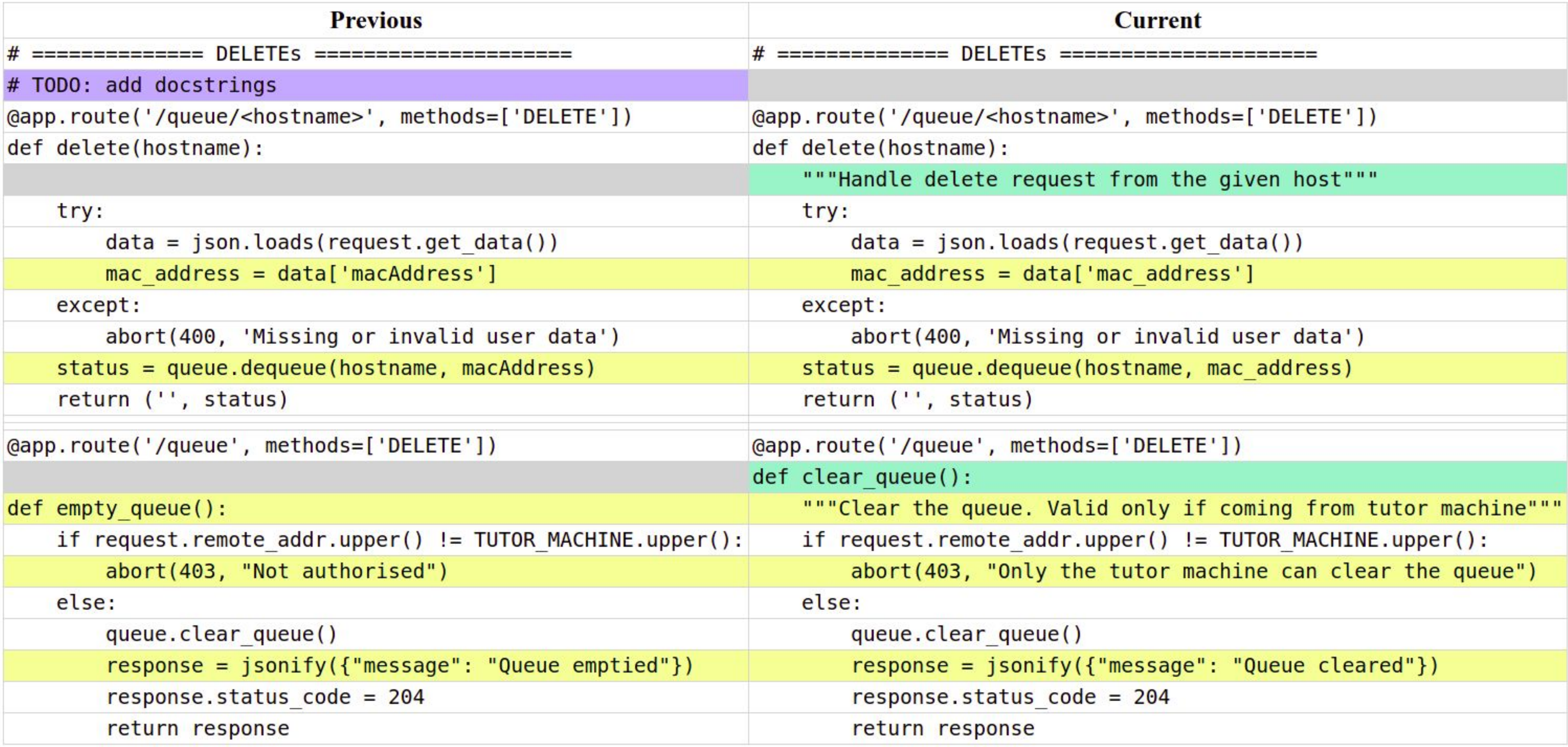
```
# ============== DELETEs =====================
@app.route('/queue/<hostname>', methods=['DELETE'])
def delete(hostname):
    """Handle delete request from the given host"""
    try:
        data = json.loads(request.get_data())
        mac_address = data['mac_address']
    except:
        abort(400, 'Missing or invalid user data')
    status = queue.dequeue(hostname, mac_address)
    return ('', status)


@app.route('/queue', methods=['DELETE'])
def clear_queue():
    """Clear the queue. Valid only if coming from tutor machine"""
    if request.remote_addr.upper() != TUTOR_MACHINE.upper():
        abort(403, "Only the tutor machine can clear the queue")
    else:
        queue.clear_queue()
        response = jsonify({"message": "Queue cleared"})
        response.status_code = 204
        return response
```

Then: Assignment 2, Q1. LCS, bottom-up DP
(same output)

Assignment 2, cont'd

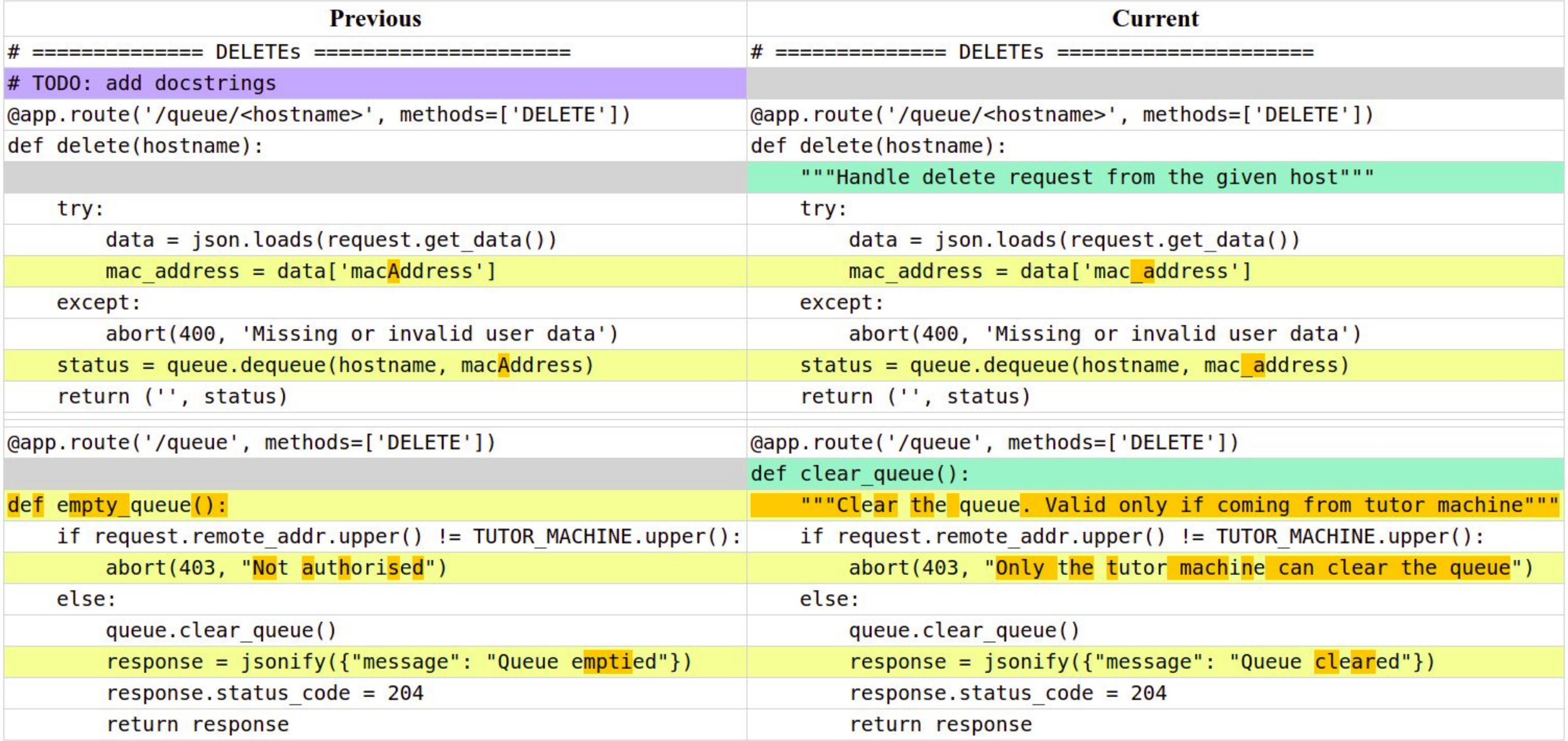
Assignment 2, Q3. Edit distance, bottom-up DP (line-based not character-based)



Previous	Current
# ============== DELETEs =====================	# ============== DELETEs =====================
# TODO: add docstrings	
@app.route('/queue/<hostname>', methods=['DELETE'])	@app.route('/queue/<hostname>', methods=['DELETE'])
def delete(hostname):	def delete(hostname):
	"""Handle delete request from the given host"""
try:	try:
data = json.loads(request.get_data())	data = json.loads(request.get_data())
mac_address = data['macAddress']	mac_address = data['mac_address']
except:	except:
abort(400, 'Missing or invalid user data')	abort(400, 'Missing or invalid user data')
status = queue.dequeue(hostname, macAddress)	status = queue.dequeue(hostname, mac_address)
return ('', status)	return ('', status)
@app.route('/queue', methods=['DELETE'])	@app.route('/queue', methods=['DELETE'])
	def clear_queue():
def empty_queue():	"""Clear the queue. Valid only if coming from tutor machine"""
if request.remote_addr.upper() != TUTOR_MACHINE.upper():	if request.remote_addr.upper() != TUTOR_MACHINE.upper():
abort(403, "Not authorised")	abort(403, "Only the tutor machine can clear the queue")
else:	else:
queue.clear_queue()	queue.clear_queue()
response = jsonify({"message": "Queue emptied"})	response = jsonify({"message": "Queue cleared"})
response.status_code = 204	response.status_code = 204
return response	return response

Assignment 2, cont’d

Assignment 2, Q3. Edit distance, bottom-up DP on lines, character-based LCS on changed lines



Previous	Current
# ============== DELETEs ======================	# ============== DELETEs ======================
# TODO: add docstrings	
@app.route('/queue/<hostname>', methods=['DELETE'])	@app.route('/queue/<hostname>', methods=['DELETE'])
def delete(hostname):	def delete(hostname):
	"""Handle delete request from the given host"""
try:	try:
data = json.loads(request.get_data())	data = json.loads(request.get_data())
mac_address = data['macAddress']	mac_address = data['mac_address']
except:	except:
abort(400, 'Missing or invalid user data')	abort(400, 'Missing or invalid user data')
status = queue.dequeue(hostname, macAddress)	status = queue.dequeue(hostname, mac_address)
return ('', status)	return ('', status)
@app.route('/queue', methods=['DELETE'])	@app.route('/queue', methods=['DELETE'])
	def clear_queue():
def empty_queue():	"""Clear the queue. Valid only if coming from tutor machine"""
if request.remote_addr.upper() != TUTOR_MACHINE.upper():	if request.remote_addr.upper() != TUTOR_MACHINE.upper():
abort(403, "Not authorised")	abort(403, "Only the tutor machine can clear the queue")
else:	else:
queue.clear_queue()	queue.clear_queue()
response = jsonify({"message": "Queue emptied"})	response = jsonify({"message": "Queue cleared"})
response.status_code = 204	response.status_code = 204
return response	return response