Solving Problems by Searching in Python: Extra exercises

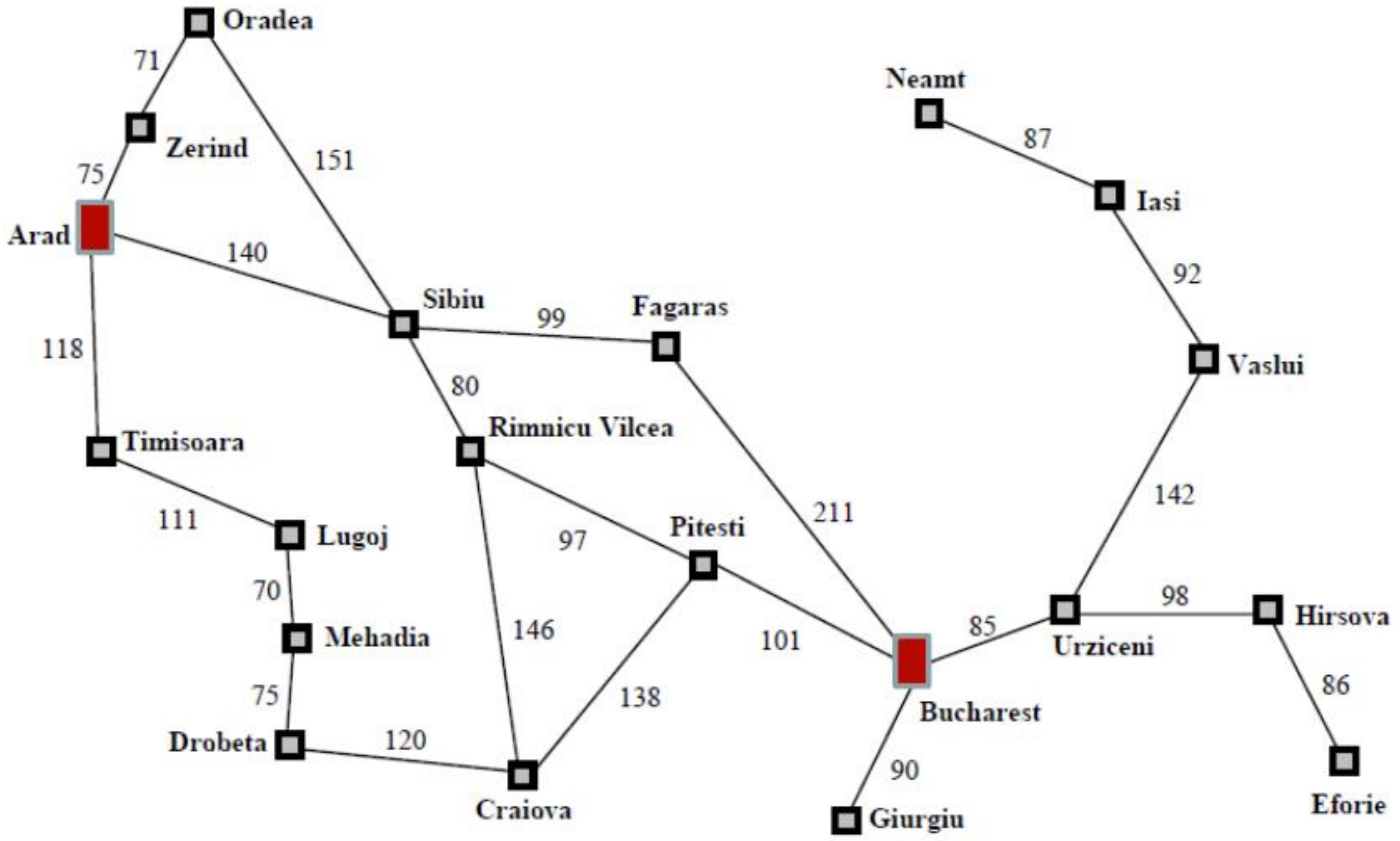
Gloria Beraldo (gloria.beraldo@unipd.it)
Department of Information Engineering, University of Padova

Topics:

- Route Finding Problems
- Map class
- Pancake Sorting Problems
- Gap heuristic
- CountCalls
- Report

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Route Finding Problems



In a RouteProblem, the states are names of "cities" (or other locations), like `'A'` for Arad.

The actions are also city names; `'Z'` is the action to move to city `'Z'`.

The layout of cities is given by a separate data structure, a Map, which is a graph where there are vertexes (cities), links between vertexes, distances (costs) of those links (if not specified, the default is 1 for every link), and optionally the 2D (x, y) location of each city can be specified.

A RouteProblem takes this Map as input and allows actions to move among linked cities. The default heuristic is straight-line distance to the goal, or is uniformly zero if locations were not given.

Map

Map, which is a graph where there are **vertexes** (cities), **links** between vertexes, **distances (costs)** of those links (if not specified, the default is 1 for every link), and **optionally** the 2D **(x, y) location** of each city can be specified.

```
class Map:
    """A map of places in a 2D world: a graph with vertexes and links between them.
    In `Map(links, locations)`, `links` can be either [(v1, v2)...] pairs,
    or a {(v1, v2): distance...} dict. Optional `locations` can be {v1: (x, y)}
    If `directed=False` then for every (v1, v2) link, we add a (v2, v1) link."""

    def __init__(self, links, locations=None, directed=False):
        if not hasattr(links, 'items'): # Distances are 1 by default
            links = {link: 1 for link in links}
        if not directed:
            for (v1, v2) in list(links):
                links[v2, v1] = links[v1, v2]
        self.distances = links
        self.neighbors = multimap(links)
        self.locations = locations or defaultdict(lambda: (0, 0))


def multimap(pairs) -> dict:
    "Given (key, val) pairs, make a dict of {key: [val,...]}."
    result = defaultdict(list)
    for key, val in pairs:
        result[key].append(val)
    return result
```

hasattr() function returns True if the specified object has the specified attribute, otherwise False.

https://docs.python.org/3/library/functions.html#hasattr

If it is missing items (e.g., the costs of the links), this means that link-cost is 1 per default

If is not directed, for every (v1, v2) link, we add a (v2, v1) link

The neighbors is represented as the multimap namely a dict (see the second function)

The 2D locations can be given or are represented by a dict with (0,0)

Map

The neighbors is represented as the multimap namely a dict (see the second function)

```
def multimap(pairs) -> dict:
    "Given (key, val) pairs, make a dict of {key: [val,...]}."
    result = defaultdict(list)
    for key, val in pairs:
        result[key].append(val)
    return result
```

```
from collections import defaultdict, deque, Counter
result = defaultdict(list)
print(result)
links =  {('O', 'Z'):  71, ('O', 'S'): 151, ('A', 'Z'): 75, ('A', 'S'): 140, ('A', 'T'): 118, ('L', 'T'): 111, ('L', 'M'):  70, ('D', 'M'): 75, ('C', 'D'): 120,
multimap(links)
```

```
defaultdict(<class 'list'>, {})
defaultdict(list,
            {'O': ['Z', 'S'],
             'A': ['Z', 'S', 'T'],
             'L': ['T', 'M'],
             'D': ['M'],
             'C': ['D', 'R', 'P'],
             'R': ['S'],
             'F': ['S'],
             'B': ['F', 'P', 'G', 'U'],
             'H': ['U'],
             'E': ['H'],
             'U': ['V'],
             'I': ['V', 'N'],
             'P': ['R']})
```

Route Finding Problems

```python
class RouteProblem(Problem):
    """A problem to find a route between locations on a `Map`.
    Create a problem with RouteProblem(start, goal, map=Map(...)}).
    States are the vertexes in the Map graph; actions are destination states."""

    def actions(self, state):
        """The places neighboring `state`."""
        return self.map.neighbors[state]

    def result(self, state, action):
        """Go to the `action` place, if the map says that is possible."""
        return action if action in self.map.neighbors[state] else state

    def action_cost(self, s, action, s1):
        """The distance (cost) to go from s to s1."""
        return self.map.distances[s, s1]

    def h(self, node):
        "Straight-line distance between state and the goal."
        locs = self.map.locations
        return straight_line_distance(locs[node.state], locs[self.goal])


def straight_line_distance(A, B):
    "Straight-line distance between two points."
    return sum(abs(a - b)**2 for (a, b) in zip(A, B)) ** 0.5
```

zip() function produces tuples with an item from each one
https://docs.python.org/3/library/functions.html#zip

The actions are also city names; `'Z'` is the action to move to city `'Z'`.

```
defaultdict(list,
            {'O': ['Z', 'S'],
             'A': ['Z', 'S', 'T'],
             'L': ['T', 'M'],
             'D': ['M'],
             'C': ['D', 'R', 'P'],
             'R': ['S'],
             'F': ['S'],
             'B': ['F', 'P', 'G', 'U'],
             'H': ['U'],
             'E': ['H'],
             'U': ['V'],
             'I': ['V', 'N'],
             'P': ['R']})
```

The cities where it is possible to move (i.e., destinations) from a current city = state. They are stored in the Map (neighbors).

The result method returns the action if the action (i.e., destinations) are inside the Map otherwise returns the state (i.e., current city)

The action_cost method returns the cost of applying the action from the current state. In this example the cost is the distance.

The default heuristic is straight-line distance to the goal, or is uniformly zero if locations were not given.

In h method (i.e., heuristic) the straight-line distance is computed as specified in the problem description

Route Finding Problems

```
class Map:
    """A map of places in a 2D world: a graph with vertexes and links between them.
    In `Map(links, locations)`, `links` can be either [(v1, v2)...] pairs,
    or a {(v1, v2): distance...} dict. Optional `locations` can be {v1: (x, y)}
    If `directed=False` then for every (v1, v2) link, we add a (v2, v1) link."""

    def __init__(self, links, locations=None, directed=False):
```

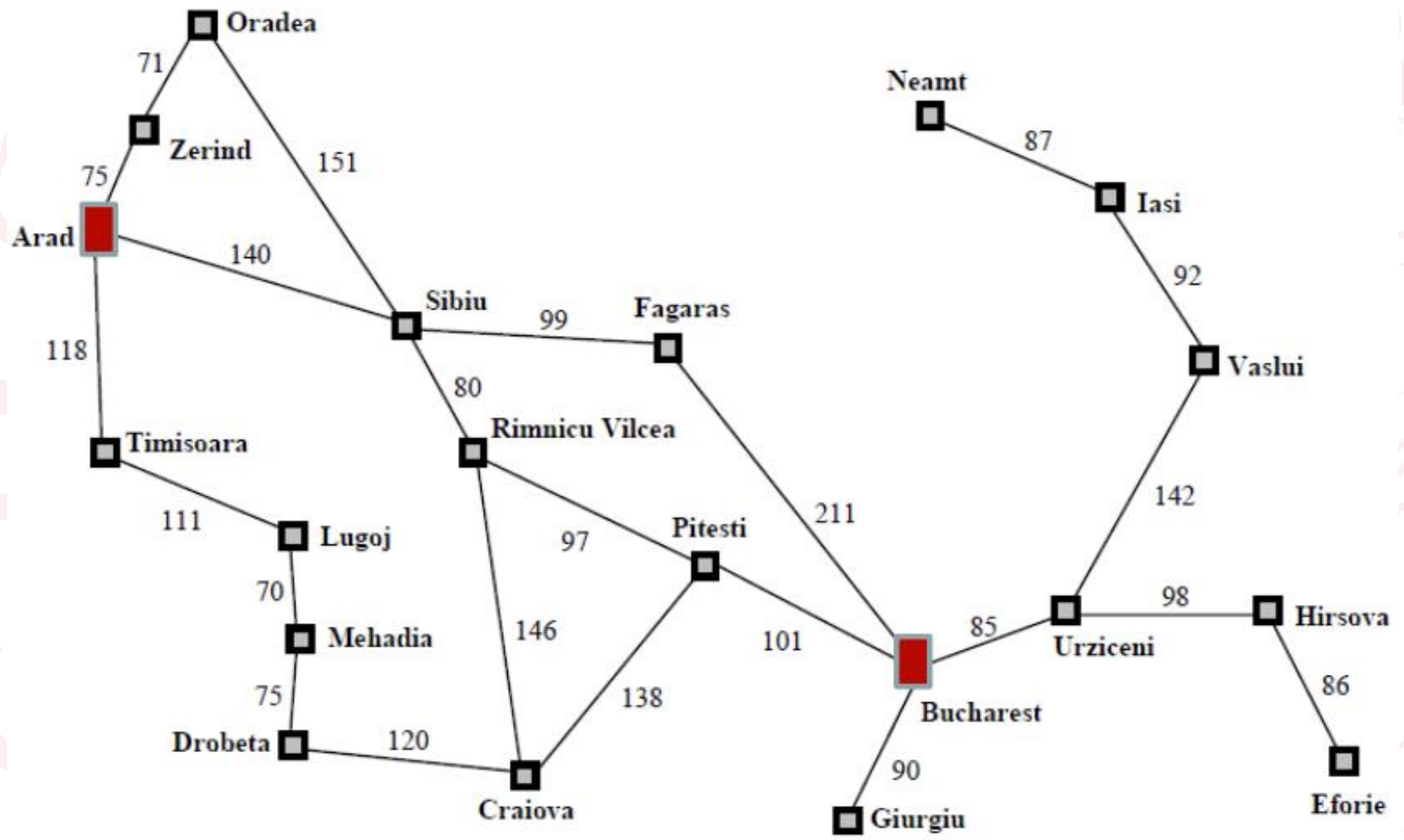
```
romania = Map(
    {('O', 'Z'):  71, ('O', 'S'): 151, ('A', 'Z'): 75, ('A', 'S'): 140, ('A', 'T'): 118,
     ('L', 'T'): 111, ('L', 'M'):  70, ('D', 'M'): 75, ('C', 'D'): 120, ('C', 'R'): 146,
     ('C', 'P'): 138, ('R', 'S'):  80, ('F', 'S'): 99, ('B', 'F'): 211, ('B', 'P'): 101,
     ('B', 'G'):  90, ('B', 'U'):  85, ('H', 'U'): 98, ('E', 'H'):  86, ('U', 'V'): 142,
     ('I', 'V'):  92, ('I', 'N'):  87, ('P', 'R'): 97},
    {'A': ( 76, 497), 'B': (400, 327), 'C': (246, 285), 'D': (160, 296), 'E': (558, 294),
     'F': (285, 460), 'G': (368, 257), 'H': (548, 355), 'I': (488, 535), 'L': (162, 379),
     'M': (160, 343), 'N': (407, 561), 'O': (117, 580), 'P': (311, 372), 'R': (227, 412),
     'S': (187, 463), 'T': ( 83, 414), 'U': (471, 363), 'V': (535, 473), 'Z': (92, 539)})


r0 = RouteProblem('A', 'A', map=romania)
r1 = RouteProblem('A', 'B', map=romania)
r2 = RouteProblem('N', 'L', map=romania)
r3 = RouteProblem('E', 'T', map=romania)
r4 = RouteProblem('O', 'M', map=romania)
```

LINKS

locations and directed are set as the default values

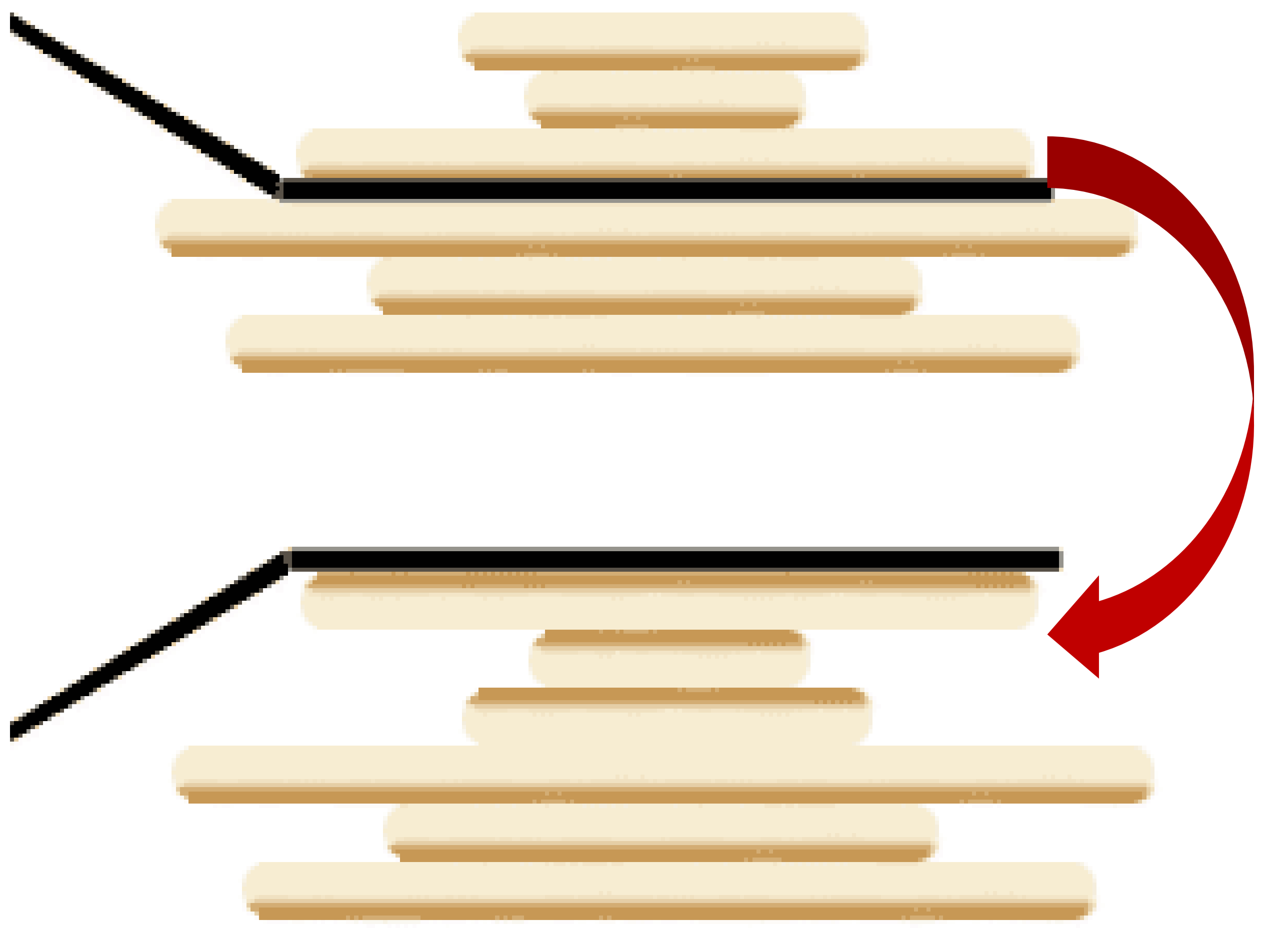
initial state

goal



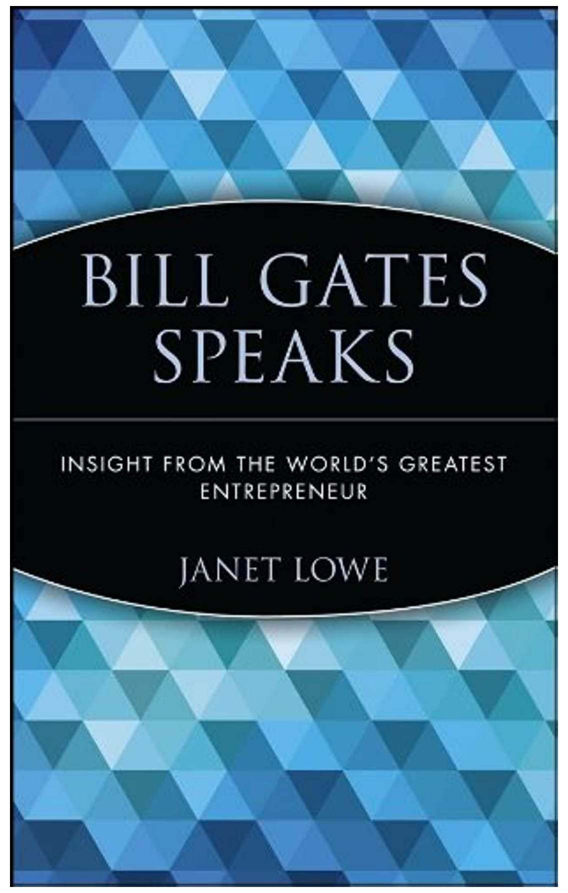
Pancake Sorting Problems

Given a stack of pancakes of various sizes, can you sort them into a stack of decreasing sizes, largest on bottom to smallest on top? You have a spatula with which you can flip the top i pancakes.



This is shown below for i = 3; on the top the spatula grabs the first three pancakes; on the bottom we see them flipped.

How many flips will it take to get the whole stack sorted? → **SORTING PROBLEM**

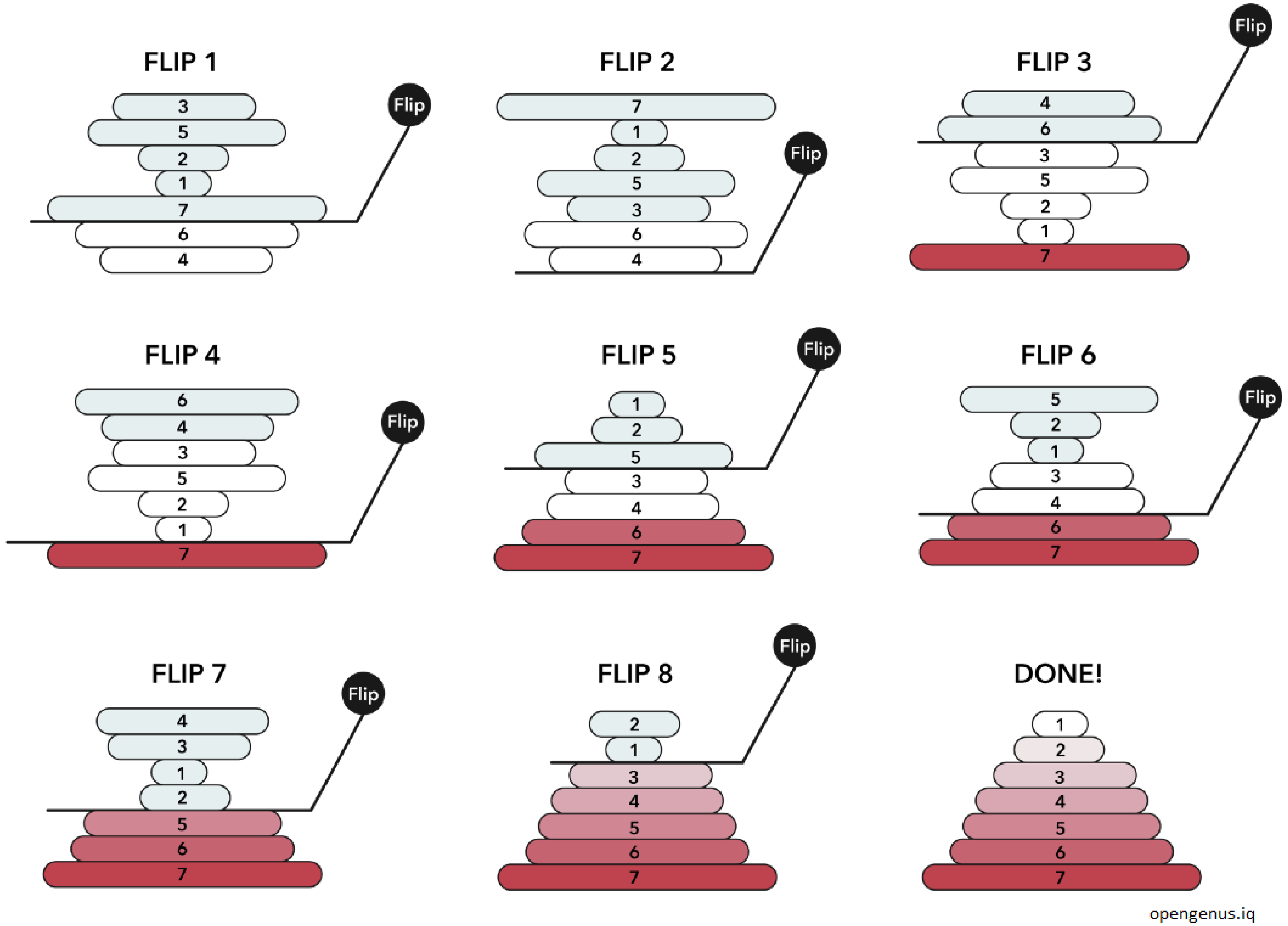


Pag 19

https://www.npr.org/templates/story/story.php?storyId=92236781

Source: Neil Jones and Pavel Pevzner, 2004 “Introduction to BioInformatics Algorithms”.

Pancake Sorting Problems



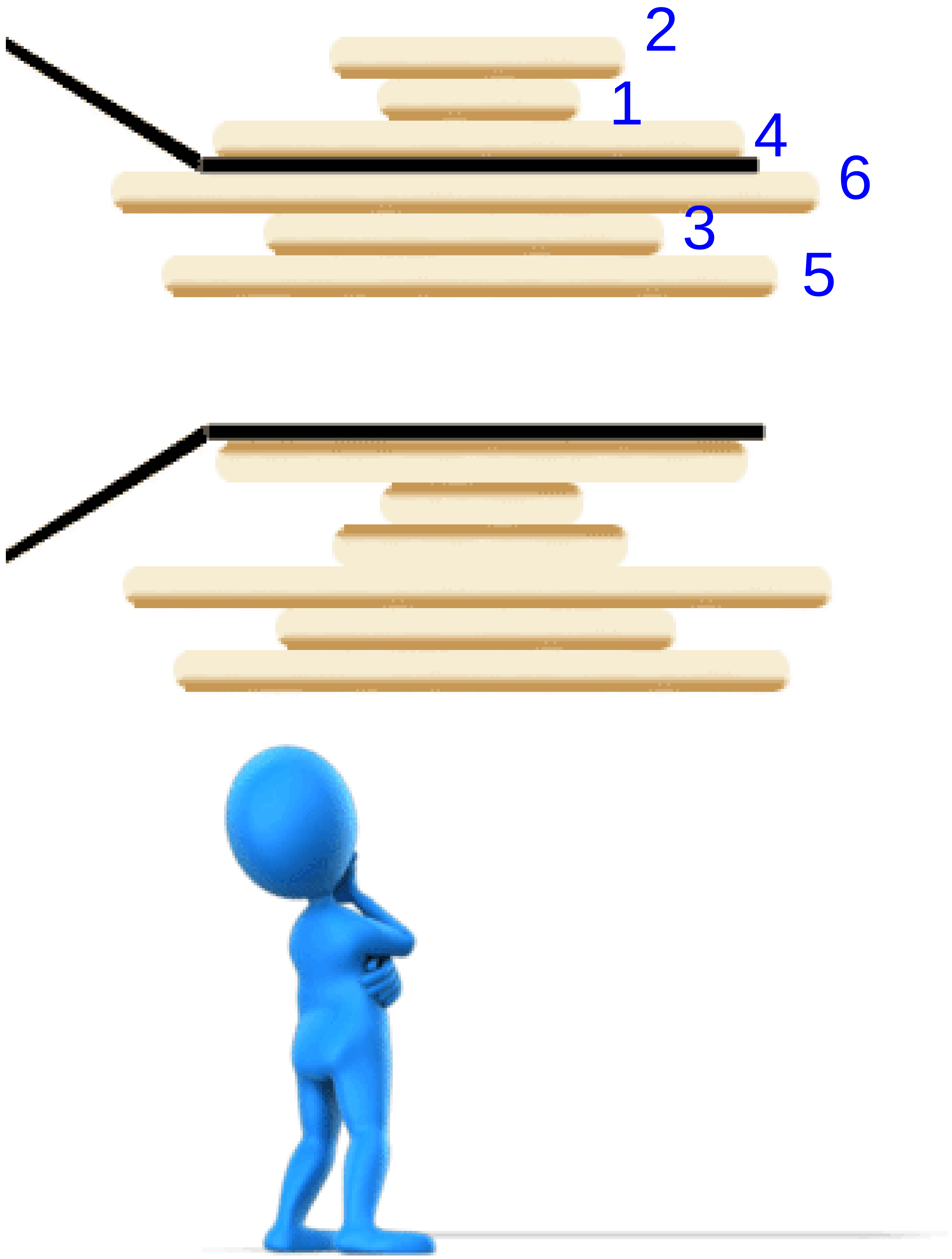
A reasonable heuristic for this problem is:
the *gap heuristic*

If we look at neighboring pancakes, if, say, the 2nd smallest is next to the 3rd smallest, that's good; they should stay next to each other.

But if the 2nd smallest is next to the 4th smallest, that's bad: we will require at least one move to separate them and insert the 3rd smallest between them.

The gap heuristic counts the number of neighbors that have a gap like this.

Pancake Sorting Problems



The gap heuristic counts the number of neighbors that have a gap like this.

In our specification of the problem, pancakes are ranked by size: the smallest is 1, the 2nd smallest 2, and so on, and

The representation of a state is a tuple of these rankings, from the top to the bottom pancake.

Thus the goal state is always (1, 2, ..., `*n*`)

The initial (top) state in the diagram above is: (2, 1, 4, 6, 3, 5).

Pancake Sorting Problems

```python
class PancakeProblem(Problem):
    """A PancakeProblem the goal is always `tuple(range(1, n+1))`, where the
    initial state is a permutation of `range(1, n+1)`. An act is the index `i`
    of the top `i` pancakes that will be flipped."""

    def __init__(self, initial):
        self.initial, self.goal = tuple(initial), tuple(sorted(initial))

    def actions(self, state): return range(2, len(state) + 1)

    def result(self, state, i): return state[:i][::-1] + state[i:]

    def h(self, node):
        "The gap heuristic."
        s = node.state
        return sum(abs(s[i] - s[i - 1]) > 1 for i in range(1, len(s)))
```

state: (2, 1, 4, 6, 3, 5). i=3

state[:i]

(4, 1, 2)

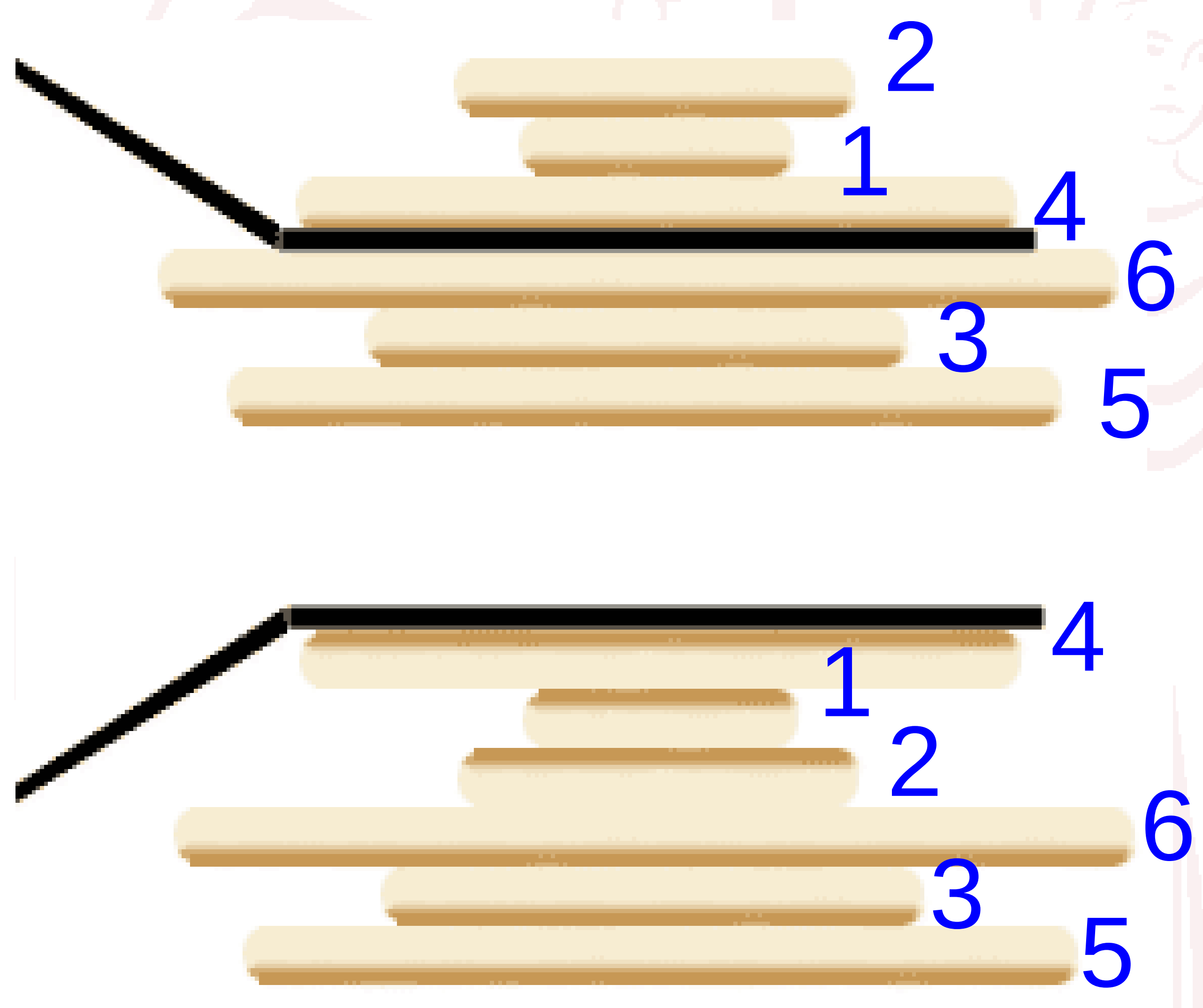
state[:i][::-1]

The representation of a state is a tuple of these rankings, from the top to the bottom pancake.

The initial state is the tuple representing the pancake stack

The goal is the sorted tuple

https://docs.python.org/3/howto/sorting.html



Pancake Sorting Problems

```python
class PancakeProblem(Problem):
    """A PancakeProblem the goal is always `tuple(range(1, n+1))`, where the
    initial state is a permutation of `range(1, n+1)`. An act is the index `i`
    of the top `i` pancakes that will be flipped."""

    def __init__(self, initial):
        self.initial, self.goal = tuple(initial), tuple(sorted(initial))

    def actions(self, state): return range(2, len(state) + 1)

    def result(self, state, i): return state[:i][::-1] + state[i:]

    def h(self, node):
        "The gap heuristic."
        s = node.state
        return sum(abs(s[i] - s[i - 1]) > 1 for i in range(1, len(s)))
```

state: (2, 1, 4, 6, 3, 5)   i=3

state[i:]

(4, 1, 2) + (6, 3, 5)

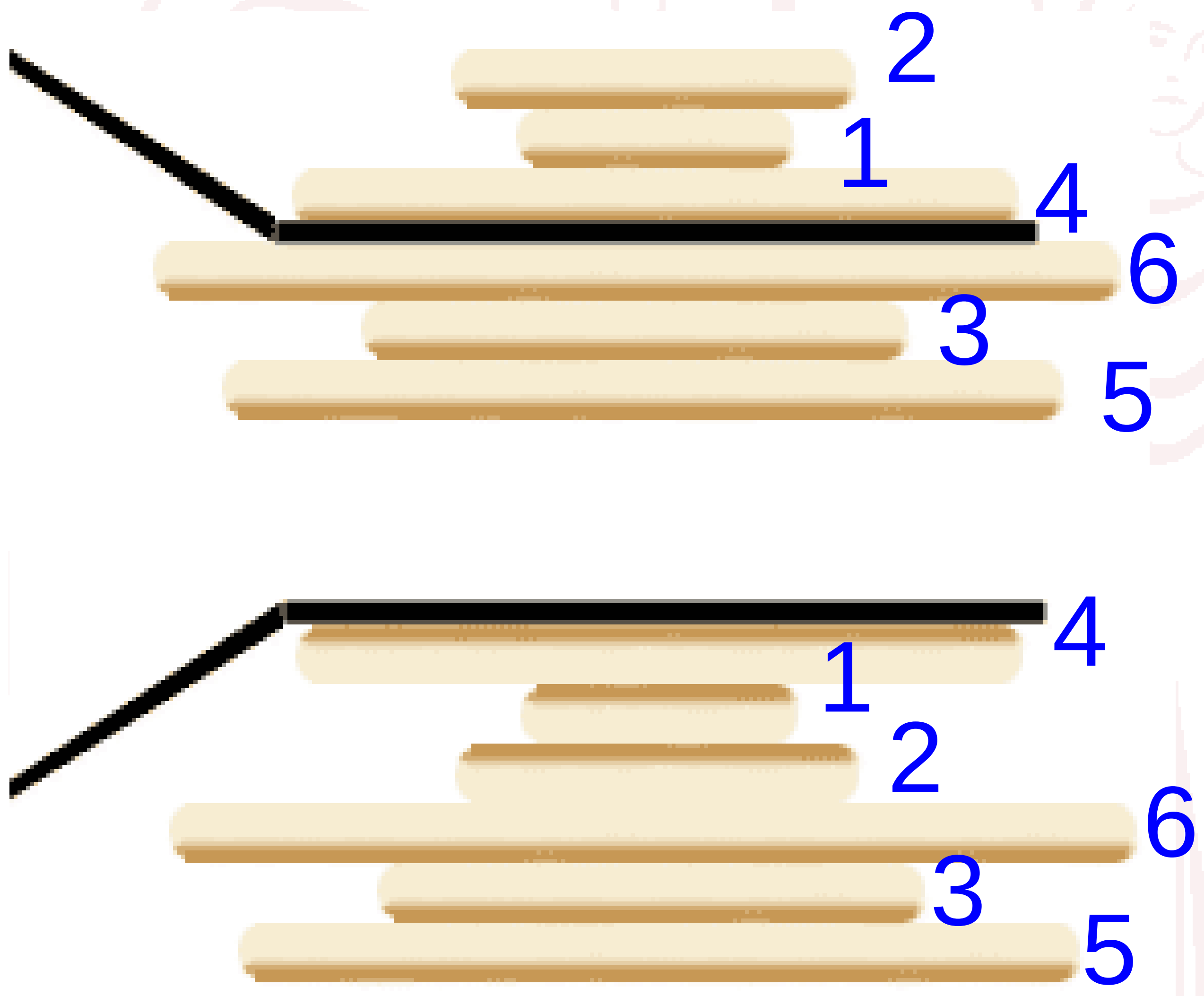
state[:i][::-1] + state[i:]

The representation of a state is a tuple of these rankings, from the top to the bottom pancake.

The initial state is the tuple representing the pancake stack

The goal is the sorted tuple

https://docs.python.org/3/howto/sorting.html



Pancake Sorting Problems

```python
class PancakeProblem(Problem):
    """A PancakeProblem the goal is always `tuple(range(1, n+1))`, where the
    initial state is a permutation of `range(1, n+1)`. An act is the index `i`
    of the top `i` pancakes that will be flipped."""

    def __init__(self, initial):
        self.initial, self.goal = tuple(initial), tuple(sorted(initial))

    def actions(self, state): return range(2, len(state) + 1)

    def result(self, state, i): return state[:i][::-1] + state[i:]

    def h(self, node):
        "The gap heuristic."
        s = node.state
        return sum(abs(s[i] - s[i - 1]) > 1 for i in range(1, len(s)))
```

The gap heuristic counts the number of neighbors that have a gap like this.

The operation is equivalent to the following:

```python
count = 0
for i in range(1, len(s)):
  diff = abs(s[i] - s[i-1])
  if diff > 1:
    count+=1
```

$(\underbrace{2, 1}_{1}, 4, 6, 3, 5).$

Pancake Sorting Problems

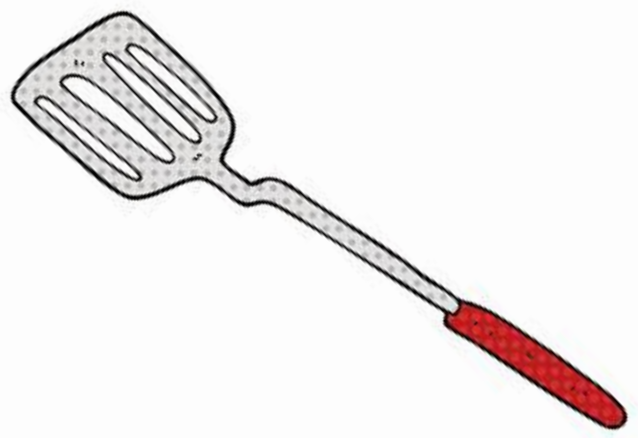
```python
class PancakeProblem(Problem):
    """A PancakeProblem the goal is always `tuple(range(1, n+1))`, where the
    initial state is a permutation of `range(1, n+1)`. An act is the index `i`
    of the top `i` pancakes that will be flipped."""

    def __init__(self, initial):
        self.initial, self.goal = tuple(initial), tuple(sorted(initial))

    def actions(self, state): return range(2, len(state) + 1)

    def result(self, state, i): return state[:i][::-1] + state[i:]

    def h(self, node):
        "The gap heuristic."
        s = node.state
        return sum(abs(s[i] - s[i - 1]) > 1 for i in range(1, len(s)))
```

The gap heuristic counts the number of neighbors that have a gap like this.

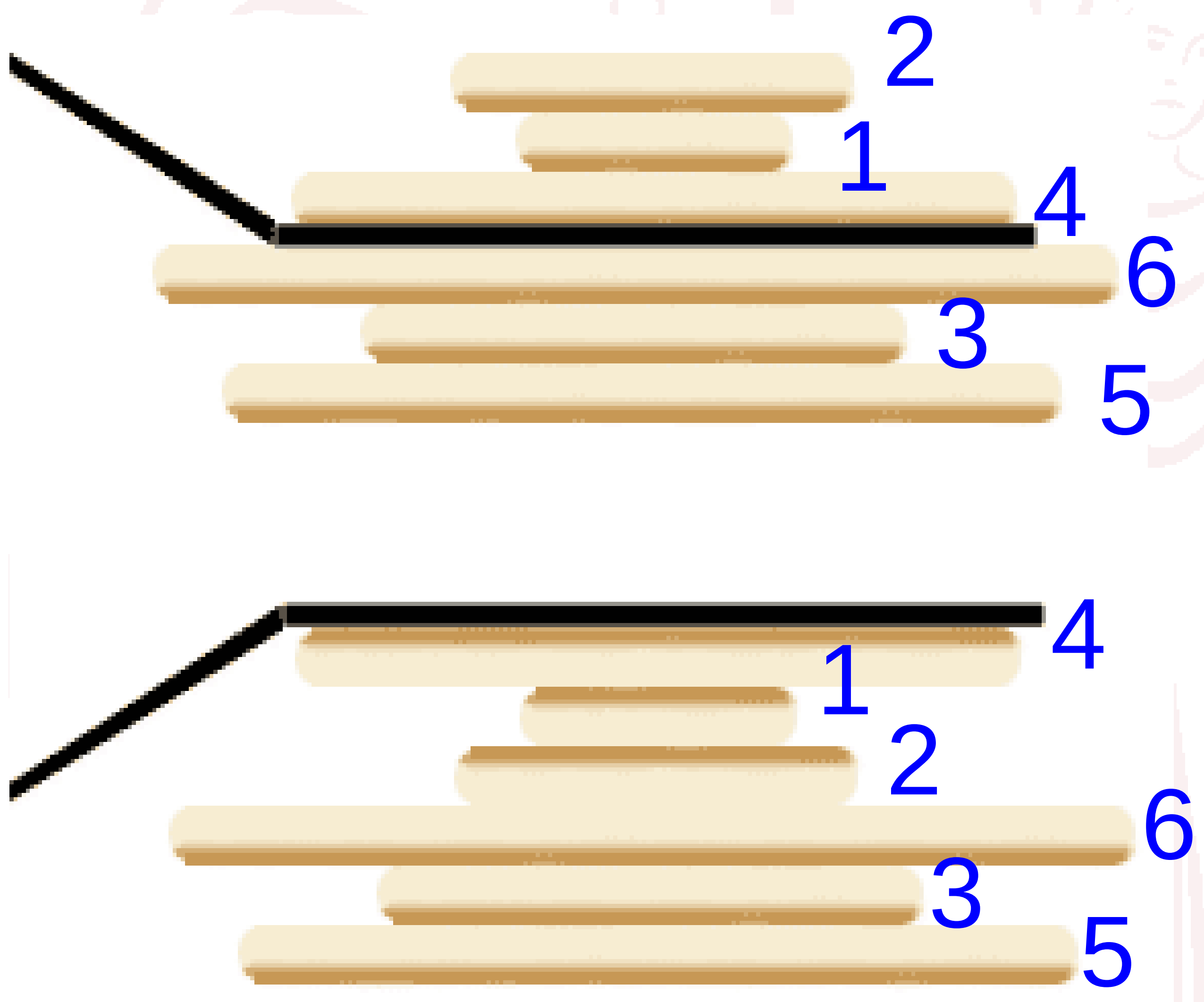
The operation is equivalent to the following:

```python
count = 0
for i in range(1, len(s)):
  diff = abs(s[i] - s[i-1])
  if diff > 1:
    count+=1
```

$(2, \underbrace{1, 4}_{3}, 6, 3, 5).$



+1



Pancake Sorting Problems

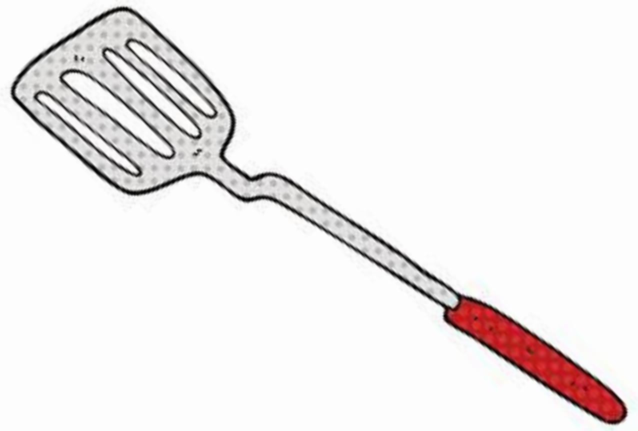
```python
class PancakeProblem(Problem):
    """A PancakeProblem the goal is always `tuple(range(1, n+1))`, where the
    initial state is a permutation of `range(1, n+1)`. An act is the index `i`
    of the top `i` pancakes that will be flipped."""

    def __init__(self, initial):
        self.initial, self.goal = tuple(initial), tuple(sorted(initial))

    def actions(self, state): return range(2, len(state) + 1)

    def result(self, state, i): return state[:i][::-1] + state[i:]

    def h(self, node):
        "The gap heuristic."
        s = node.state
        return sum(abs(s[i] - s[i - 1]) > 1 for i in range(1, len(s)))
```

The gap heuristic counts the number of neighbors that have a gap like this.

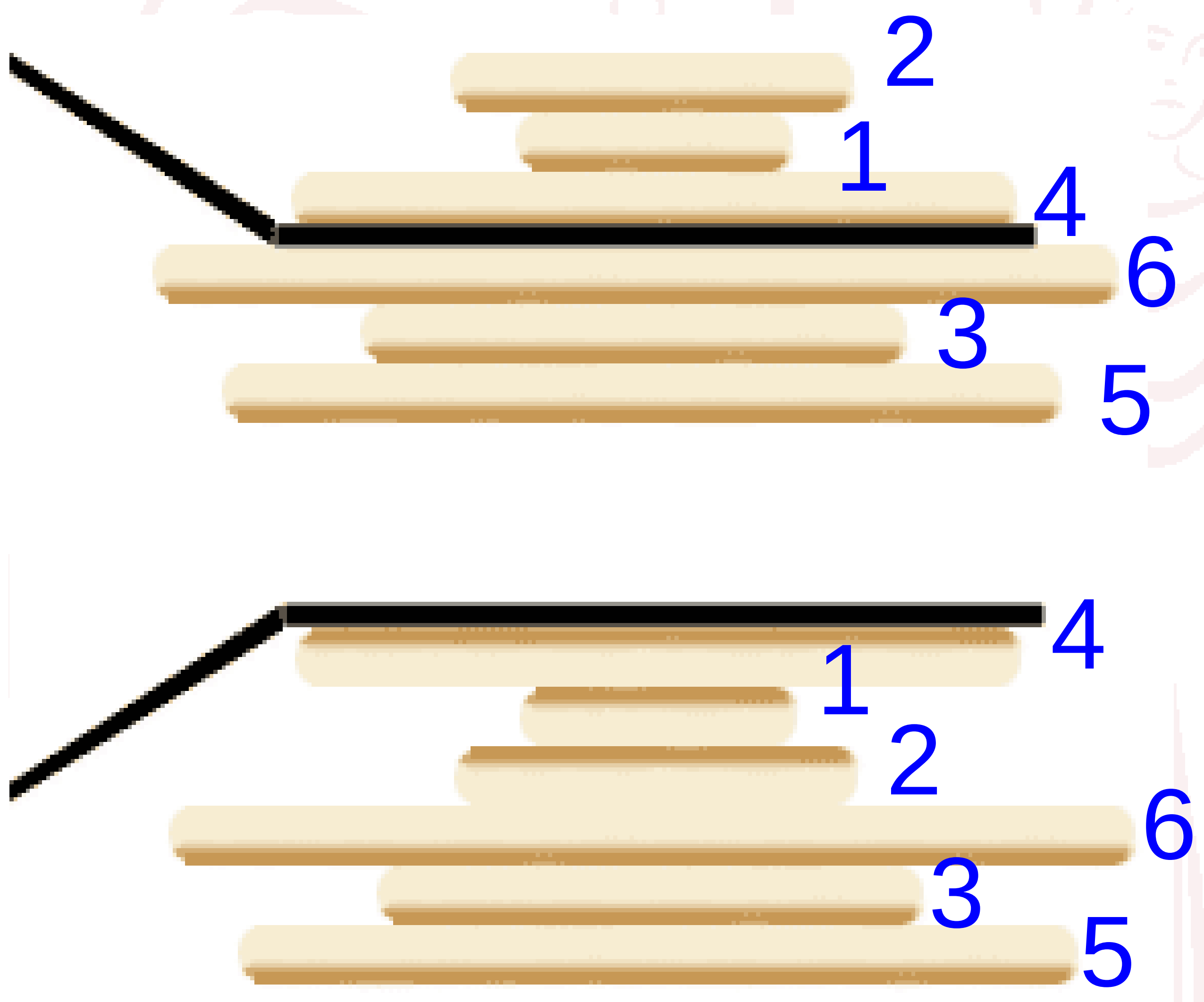
The operation is equivalent to the following:

```python
count = 0
for i in range(1, len(s)):
  diff = abs(s[i] - s[i-1])
  if diff > 1:
    count+=1
```

(2, $\underbrace{1, 4}_{2}$, 6, 3, 5).



+2



Pancake Sorting Problems

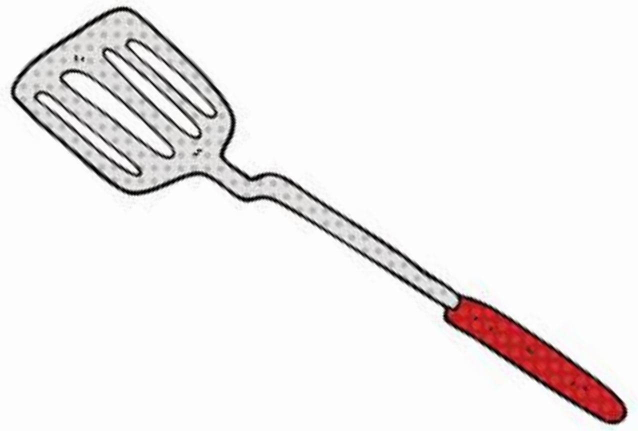
```python
class PancakeProblem(Problem):
    """A PancakeProblem the goal is always `tuple(range(1, n+1))`, where the
    initial state is a permutation of `range(1, n+1)`. An act is the index `i`
    of the top `i` pancakes that will be flipped."""

    def __init__(self, initial):
        self.initial, self.goal = tuple(initial), tuple(sorted(initial))

    def actions(self, state): return range(2, len(state) + 1)

    def result(self, state, i): return state[:i][::-1] + state[i:]

    def h(self, node):
        "The gap heuristic."
        s = node.state
        return sum(abs(s[i] - s[i - 1]) > 1 for i in range(1, len(s)))
```

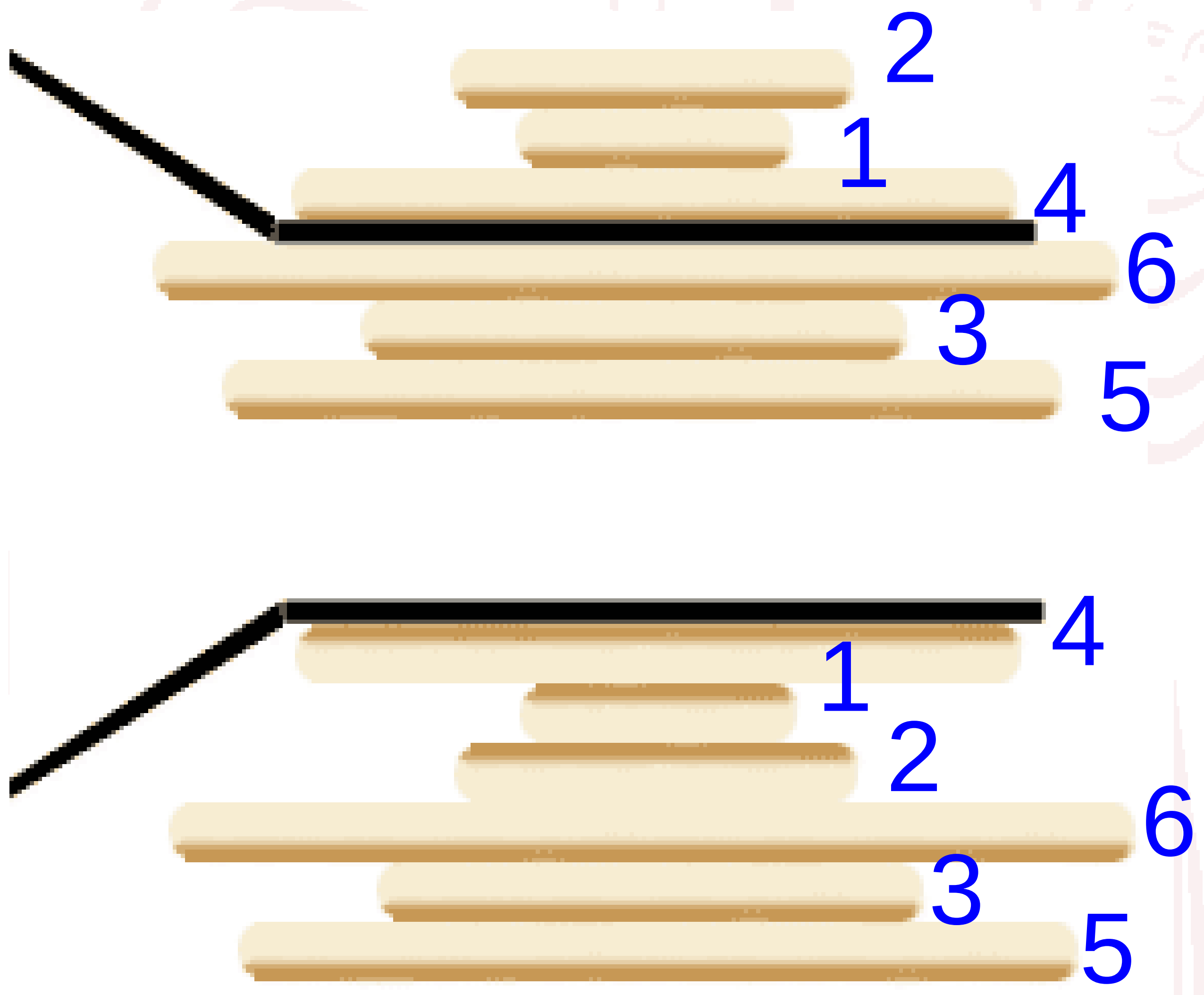
The gap heuristic counts the number of neighbors that have a gap like this.

The operation is equivalent to the following:

```python
count = 0
for i in range(1, len(s)):
  diff = abs(s[i] - s[i-1])
  if diff > 1:
    count+=1
```

$(2, 1, \underbrace{4, 6,}_{3} 3, 5).$





Pancake Sorting Problems

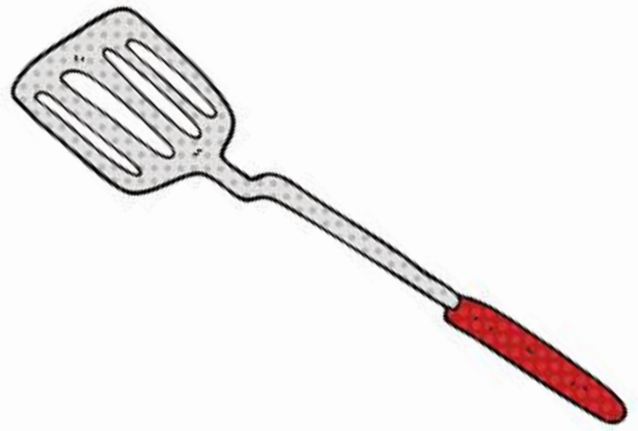
```python
class PancakeProblem(Problem):
    """A PancakeProblem the goal is always `tuple(range(1, n+1))`, where the
    initial state is a permutation of `range(1, n+1)`. An act is the index `i`
    of the top `i` pancakes that will be flipped."""

    def __init__(self, initial):
        self.initial, self.goal = tuple(initial), tuple(sorted(initial))

    def actions(self, state): return range(2, len(state) + 1)

    def result(self, state, i): return state[:i][::-1] + state[i:]

    def h(self, node):
        "The gap heuristic."
        s = node.state
        return sum(abs(s[i] - s[i - 1]) > 1 for i in range(1, len(s)))
```

The gap heuristic counts the number of neighbors that have a gap like this.

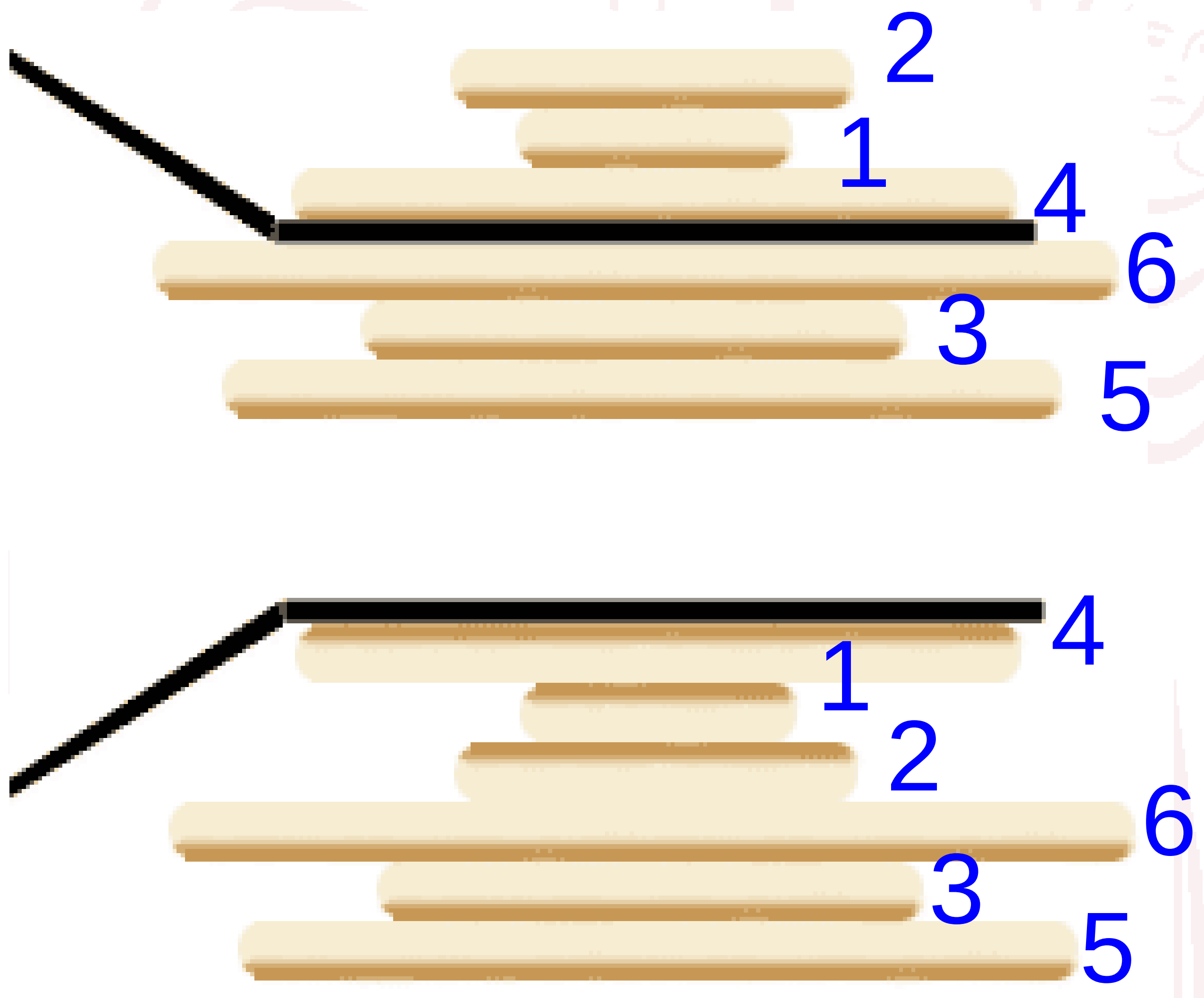
The operation is equivalent to the following:

```python
count = 0
for i in range(1, len(s)):
  diff = abs(s[i] - s[i-1])
  if diff > 1:
    count+=1
```

$(2, 1, 4, 6, \underbrace{3, 5}_{2}).$



+4



CountCalls

We'll use CountCalls to wrap a Problem object in such a way that calls to its methods are delegated to the original problem, but each call increments a counter.

```
class CountCalls:
    """Delegate all attribute gets to the object, and count them in ._counts"""
    def __init__(self, obj):
        self._object = obj
        self._counts = Counter()

    def __getattr__(self, attr):
        "Delegate to the original object, after incrementing a counter."
        self._counts[attr] += 1
        return getattr(self._object, attr)
```

https://docs.python.org/3/library/collections.html#collections.Counter

Called when the default attribute access fails with an AttributeError (either __getattribute__() raises an AttributeError because *name* is not an instance attribute or an attribute in the class tree for self; or __get__() of a *name* property raises AttributeError).

https://docs.python.org/3/reference/datamodel.html#object.__getattr__

Report

Now let's gather some metrics on how well each algorithm does.

```
def report(searchers, problems, verbose=True):
    """Show summary statistics for each searcher (and on each problem unless verbose is false)."""
    for searcher in searchers:
        print(searcher.__name__ + ':')
        total_counts = Counter()
        for p in problems:
            print(p)
            prob   = CountCalls(p)
            soln   = searcher(prob)
            counts = prob._counts;
            counts.update(actions=len(soln), cost=soln.path_cost)
            total_counts += counts
            if verbose: report_counts(counts, str(p)[:40])
        report_counts(total_counts, 'TOTAL\n')

def report_counts(counts, name):
    """Print one line of the counts report."""
    print('{:9,d} nodes |{:9,d} goal |{:5.0f} cost |{:8,d} actions | {}'.format(
          counts['result'], counts['is_goal'], counts['cost'], counts['actions'], name))
```

searchers is an iterable object (e.g., list)

Report

Now let's gather some metrics on how well each algorithm does.

```
def report(searchers, problems, verbose=True):
    """Show summary statistics for each searcher (and on each problem unless verbose is false)."""
    for searcher in searchers:
        print(searcher.__name__ + ':')
        total_counts = Counter()
        for p in problems:
            print(p)
            prob   = CountCalls(p)
            soln   = searcher(prob)
            counts = prob._counts;
            counts.update(actions=len(soln), cost=soln.path_cost)
            total_counts += counts
            if verbose: report_counts(counts, str(p)[:40])
        report_counts(total_counts, 'TOTAL\n')

def report_counts(counts, name):
    """Print one line of the counts report."""
    print('{:9,d} nodes |{:9,d} goal |{:5.0f} cost |{:8,d} actions | {}'.format(
        counts['result'], counts['is_goal'], counts['cost'], counts['actions'], name))
```

problems is an iterable object (e.g., list)

Report

Now let's gather some metrics on how well each algorithm does.

```
def report(searchers, problems, verbose=True):
    """Show summary statistics for each searcher (and on each problem unless verbose is false)."""
    for searcher in searchers:
        print(searcher.__name__ + ':')
        total_counts = Counter()
        for p in problems:
            print(p)
            prob   = CountCalls(p)
            soln   = searcher(prob)
            counts = prob._counts;
            counts.update(actions=len(soln), cost=soln.path_cost)
            total_counts += counts
            if verbose: report_counts(counts, str(p)[:40])
        report_counts(total_counts, 'TOTAL\n')

def report_counts(counts, name):
    """Print one line of the counts report."""
    print('{:9,d} nodes |{:9,d} goal |{:5.0f} cost |{:8,d} actions | {}'.format(
          counts['result'], counts['is_goal'], counts['cost'], counts['actions'], name))
```

solve the problem using the *searcher*

Report

Now let's gather some metrics on how well each algorithm does.

```
def report(searchers, problems, verbose=True):
    """Show summary statistics for each searcher (and on each problem unless verbose is false)."""
    for searcher in searchers:
        print(searcher.__name__ + ':')
        total_counts = Counter()
        for p in problems:
            print(p)
            prob   = CountCalls(p)
            soln   = searcher(prob)
            counts = prob._counts;
            counts.update(actions=len(soln), cost=soln.path_cost)
            total_counts += counts
            if verbose: report_counts(counts, str(p)[:40])
        report_counts(total_counts, 'TOTAL\n')

def report_counts(counts, name):
    """Print one line of the counts report."""
    print('{:9,d} nodes |{:9,d} goal |{:5.0f} cost |{:8,d} actions | {}'.format(
          counts['result'], counts['is_goal'], counts['cost'], counts['actions'], name))
```

update([iterable-or-mapping])
Elements are counted from an *iterable* or added-in from another *mapping* (or counter).
https://docs.python.org/3/library/collections.html#collections.Counter

Questions