Objects

Announcements

Dictionaries

```
{'Dem': 0}
```

Dictionary Comprehensions

```
{<key exp>: <value exp> for <name> in <iter exp> if <filter exp>}
```

```
Short version: {<key exp>: <value exp> for <name> in <iter exp>}
```

Example: Multiples

Implement **multiples**, which takes two lists of positive numbers **s** and **factors**. It returns a dictionary in which each element of factors is a key, and the value for each key is a list of the elements of **s** that are multiples of the key.

```
def multiples(s, factors):
    """Create a dictionary where each factor is a key and each value
    is the elements of s that are multiples of the key.

    >>> multiples([3, 4, 5, 6, 7, 8], [2, 3])
    {2: [4, 6, 8], 3: [3, 6]}
    >>> multiples([1, 2, 3, 4, 5], [2, 5, 8])
    {2: [2, 4], 5: [5], 8: []}
    """

    return {d: [x for x in ________ if ___________] for d in ________}
```

Example: Multiples

Implement **multiples**, which takes two lists of positive numbers **s** and **factors**. It returns a dictionary in which each element of factors is a key, and the value for each key is a list of the elements of **s** that are multiples of the key.

```
def multiples(s, factors):
    """Create a dictionary where each factor is a key and each value
    is the elements of s that are multiples of the key.

    >>> multiples([3, 4, 5, 6, 7, 8], [2, 3])
    {2: [4, 6, 8], 3: [3, 6]}
    >>> multiples([1, 2, 3, 4, 5], [2, 5, 8])
    {2: [2, 4], 5: [5], 8: []}
    """

    return {d: [x for x in ____s____ if ____x % d == 0____] for d in ____factors____}
```

Data Classes

Objects

Python is an object-oriented language

- Objects represent information
- They consist of data and behavior, bundled together to create abstractions

Dictionaries are objects:

- A correspondence between keys and values is what a dict represent: `d = {3: 9, 4: 16}`
- `d.get(4)` (which evaluates to 16) is part of how a dict behaves
- **get** is called a method: it's a function invoked on the dictionary
- Its type is dict
- When you print a dict, the format looks like a dict

All values in Python are objects: each has data as well as behavior determined by its type

Class Statements Create User-Defined Types

A class statement creates a new type; the **@dataclass** decorator simplifies class statements

In a data class, each name with a type hint creates an attribute for instances of the type

```
from dataclasses import dataclass

@dataclass
class Line:
    slope: float
    intercept: float
    def __str__(self):
        return format_line(self)

def parallel(c: Line, d: Line) -> bool:
    return c.slope == d.slope

def format_line(c: Line) -> str:
    return 'y = ' + str(c.slope) + 'x + ' + str(c.intercept)

 OR return f'y = {c.slope}x + {c.intercept}'
```

__str__ determines what happens when you call **print** or **str** on a **Line**

```
>>> c = Line(3, 4)
>>> c.slope
3
>>> c.intercept
4
>>> c
Line(slope=3, intercept=4)
>>> type(c) == Line
True
>>> isinstance(c, Line)
True
>>> print(c)
y = 3x + 4
```

Partitions with Restrictions

Recursion so far

```
double_eights(s: list[int]) -> bool:
  Strategy: Check if the first two elements are both 8s
            Call double_eights on everything except the first element

streak(n: int) -> bool:
      Return whether n is a dice integer in which all digits the same
  Strategy: Check if last digit is a dice integer, and matches the previous
            Call streak on everything except the last digit

reverse(s: list) -> list:
  Strategy: Get the first element into place
            Call reverse on the rest
```

Deal with one item or digit; recurse for the rest

```
count_partitions(n: int, m: int) -> int:
      Return how many ways we can count to n, using pieces of up to size m
  Strategy: Use a piece of size m; recurse for the rest
                                  AND
            Don't use any pieces of size m; recurse for the rest
```

Tree recursion: Make a SMALL choice; for each option, recurse

Count Partitions Review

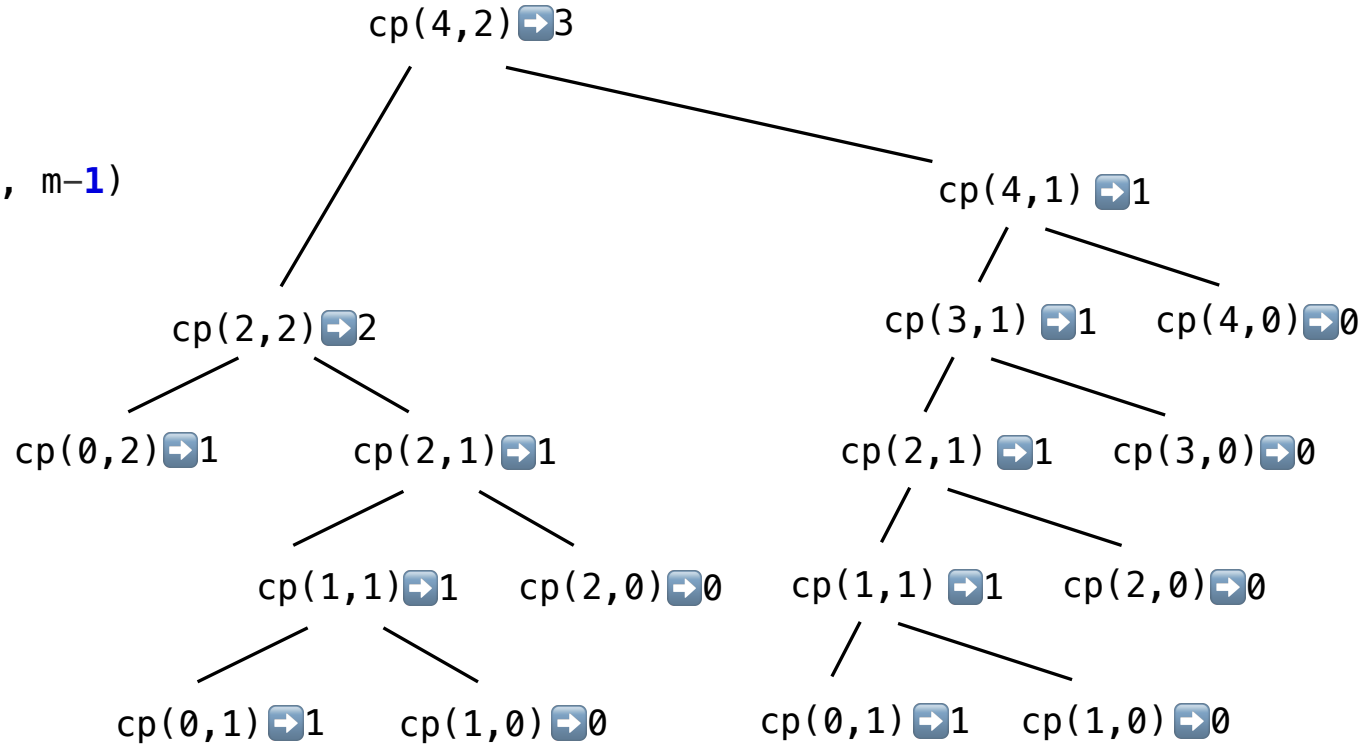
count_partitions

cp(n, m): count the ways to make n by summing positive pieces up to m in increasing order

```
def cp(n, m):
    if n == 0:
        return 1
    elif n < 0 or m == 0:
        return 0
    return cp(n-m, m) + cp(n, m-1)
```

cp(4, 2) ➡️ 3 ways

- choose at least 1 m=2
  - **cp(4-2, 2)** ➡️ 2 ways
    - 2+2 = 4
    - 1+1+2 = 4
- choose no more m=2:
  - **cp(4, 1)** ➡️ 1 way
    - 1+1+1+1 = 4



Count Partitions Variants

count_partitions

cp(n, m): count the ways to make n by summing positive pieces up to m in increasing order

```
def cp(n, m):
    if n == 0:
        return 1
    elif n < 0 or m == 0:
        return 0
    return cp(n-m, m) + cp(n, m-1)
```

cp(4, 2) ➡️ 3 ways

- choose at least 1 m=2
- **cp(4-2, 2)** ➡️ 2 ways
  - 2+2 = 4
  - 1+1+2 = 4
- choose no more m=2:
- **cp(4, 1)** ➡️ 1 way
  - 1+1+1+1 = 4

Adding base cases can skip work

```
elif m == 1:
    return 1
```

Avoiding base cases by varying recursive calls

```
def cp(n, m):
    if n == 0:
        return 1
    elif n < 0 or m == 0:
        return 0
    if m > n:
        return cp(n, n)
    return cp(n-m, m) + cp(n, m-1)
```

Tracking State

count_partitions

cp(n, m): count the ways to make n by summing positive pieces up to m in increasing order using **at most 3 total pieces** in the sum: 2 + 2 + 2 is fine, but 1 + 1 + 2 + 2 is not

Original implementation:

```
def cp(n, m):
    if n == 0:
        return 1
    elif n < 0 or m == 0:
        return 0
    return cp(n-m, m) + cp(n, m-1)
```

There is no way to add a base case for using too many pieces, because we don't know how many pieces we've used.

Revised implementation:

```
def cp_3total(n, m):
    return cpk(n, m, 3)

def cpk(n, m, k):
    if n == 0:
        return 1
    elif n < 0 or m == 0:
        return 0
    elif k == 0:
        return 0
    return cpk(n-m, m, k-1) + cpk(n, m-1, k)
```

At most **k** pieces total

Lesson: To add a constraint, add arguments that track whether the constraint was violated

Discussion Question

count_partitions

cp(n, m): count the ways to make n by summing positive pieces up to m in increasing order using **each piece at most 3 times:** 1 + 1 + 2 + 2 is fine, but 1 + 1 + 1 + 1 + 2 is not

Original implementation:

```
def cp(n, m):
    if n == 0:
        return 1
    elif n < 0 or m == 0:
        return 0
    return cp(n-m, m) + cp(n, m-1)
```

There is no way to add a base case for using too many pieces, because we don't know how many pieces we've used.

Revised implementation:

```
def cp_3_of_each_piece(n, m):
    return cpk(n, m, 3, 3)

def cpk(n, m, k, t):
    if n == 0:
        return 1
    elif n < 0 or m == 0:
        return 0

    no_m = cpk(_____, _____, _____, _____)
    if k == 0:
        return no_m
    return cpk(n-m, m, k-1, t) + no_m
```

At most **k** of **m** & at most **t** of each smaller piece

Lesson: To add a constraint, add arguments that track whether the constraint was violated

Discussion Question

count_partitions

cp(n, m): count the ways to make n by summing positive pieces up to m in increasing order using **each piece at most 3 times**: 1 + 1 + 2 + 2 is fine, but 1 + 1 + 1 + 1 + 2 is not

Original implementation:

```
def cp(n, m):
    if n == 0:
        return 1
    elif n < 0 or m == 0:
        return 0
    return cp(n-m, m) + cp(n, m-1)
```

There is no way to add a base case for using too many pieces, because we don't know how many pieces we've used.

Revised implementation:

```
def cp_3_of_each_piece(n, m):
    return cpk(n, m, 3, 3)

def cpk(n, m, k, t):
    if n == 0:
        return 1
    elif n < 0 or m == 0:
        return 0

    no_m = cpk(  n  , m-1 , t 3 , t  )
    if k == 0:
        return no_m
    return cpk(n-m, m, k-1, t) + no_m
```

At most **k** of **m** & at most t 3 of each smaller piece

Lesson: To add a constraint, add arguments that track whether the constraint was violated

More Tree Recursion Practice

Spring 2023 Midterm 2 Question 5(a) [modified a bit]

Definition. When parking vehicles in a row, a motorcycle takes up 1 parking spot and a car takes up 2 adjacent parking spots. A string of length n can represent n adjacent parking spots using % for a motorcycle, <> for a car, and . for an empty spot.

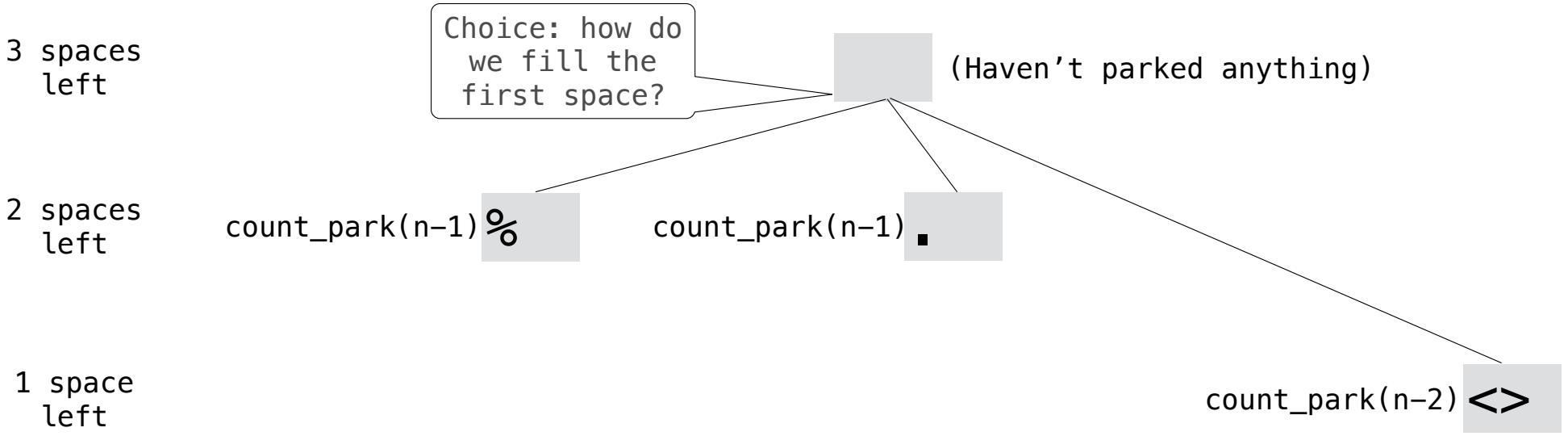
For example: '.%%.<><>' (Thanks to the Berkeley Math Circle for introducing this question.)

Implement **count_park**, which returns the number of ways that vehicles can be parked in n adjacent parking spots for positive integer n. Some or all spots can be empty.

```
def count_park(n):
    """Count the ways to park cars and motorcycles in n adjacent spots.
    >>> count_park(1)  # '.' or '%'
    2
    >>> count_park(2)  # '..', '.%', '%.', '%%', or '<>'
    5
    >>> count_park(4)  # some examples: '<><>', '.%%.', '%<>%', '%.<>'
    29
    """
```

We haven't parked anything yet. What's a first choice we can make?

Spring 2023 Midterm 2 Question 5(a) [modified a bit]



Spring 2023 Midterm 2 Question 5(a) [modified a bit]

Definition. When parking vehicles in a row, a motorcycle takes up 1 parking spot and a car takes up 2 adjacent parking spots. A string of length n can represent n adjacent parking spots using % for a motorcycle, <> for a car, and . for an empty spot.

For example: '.%%.<><>' (Thanks to the Berkeley Math Circle for introducing this question.)

Implement **count_park**, which returns the number of ways that vehicles can be parked in n adjacent parking spots for positive integer n. Some or all spots can be empty.

```
def count_park(n):
    """Count the ways to park cars and motorcycles in n adjacent spots.
    >>> count_park(1)  # '.' or '%'
    2
    >>> count_park(2)  # '..', '.%', '%.', '%%', or '<>'
    5
    >>> count_park(4)  # some examples: '<><>', '.%%.', '%<>%', '%.<>'
    29
    """
    if n < 0:
        return ____________
    elif n == 0:
        return ____________
    else:
        return ____________________________________________________
```

count_park(3):

%%%
%%.
%.%
%..
%<>
.%%
.%.
..%
...
.<>
<>%
<>.

Spring 2023 Midterm 2 Question 5(a) [modified a bit]

Definition. When parking vehicles in a row, a motorcycle takes up 1 parking spot and a car takes up 2 adjacent parking spots. A string of length n can represent n adjacent parking spots using % for a motorcycle, <> for a car, and . for an empty spot.

For example: '.%%.<><>' (Thanks to the Berkeley Math Circle for introducing this question.)

Implement **count_park**, which returns the number of ways that vehicles can be parked in n adjacent parking spots for positive integer n. Some or all spots can be empty.

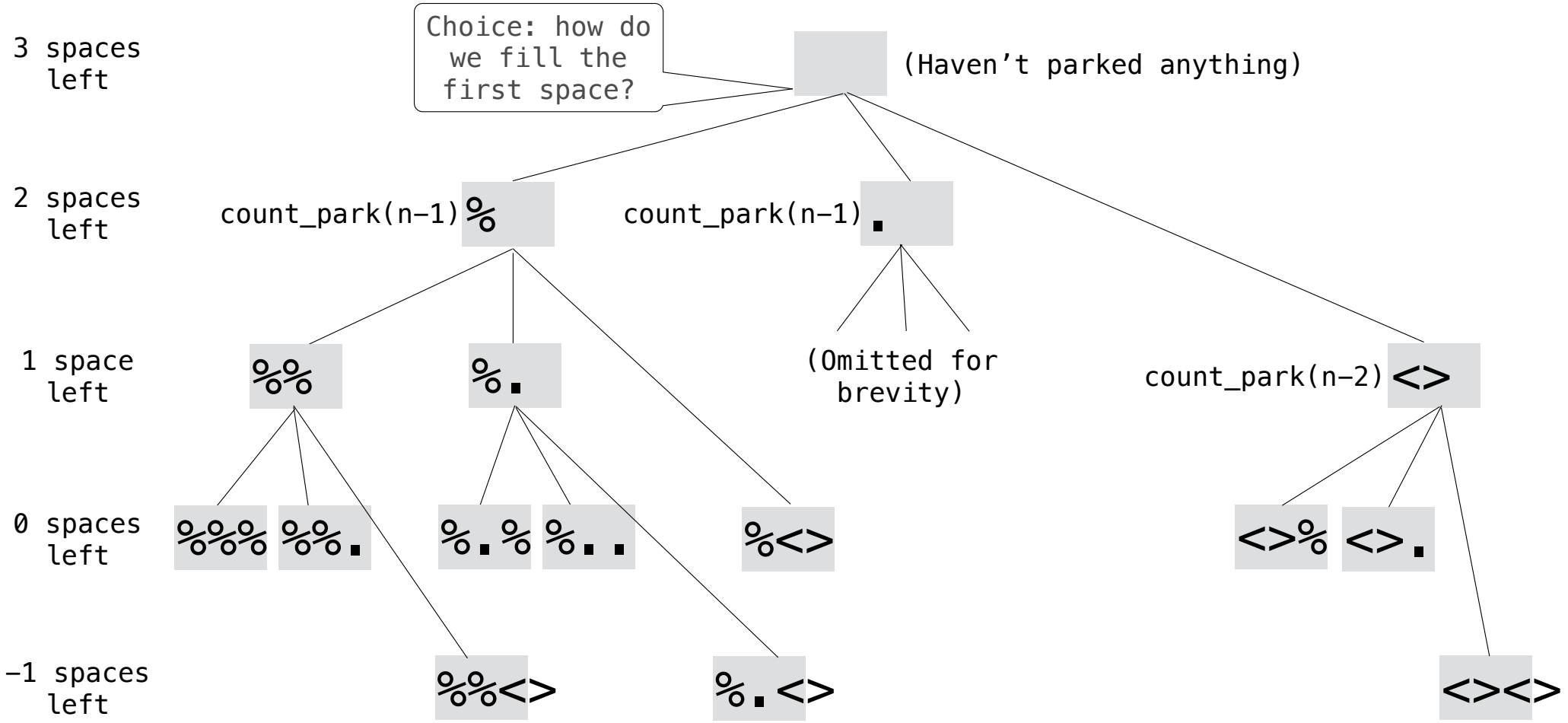
```
def count_park(n):
    """Count the ways to park cars and motorcycles in n adjacent spots.
    >>> count_park(1)  # '.' or '%'
    2
    >>> count_park(2)  # '..', '.%', '%.', '%%', or '<>'
    5
    >>> count_park(4)  # some examples: '<><>', '.%%.', '%<>%', '%.<>'
    29
    """
    if n < 0:
        return ____________
    elif n == 0:
        return ____________
    else:
        return count_park(n-1) + count_park(n-1) + count_park(n-2)
```

One way to think about these base cases: which recursive calls lead to these cases, and what should their values be?

count_park(3):

%%%
%%.
%.%
%..
%<>
.%%
.%.
..%
...
.<>
<>%
<>.

Spring 2023 Midterm 2 Question 5(a) [modified a bit]



Spring 2023 Midterm 2 Question 5(a) [modified a bit]

Definition. When parking vehicles in a row, a motorcycle takes up 1 parking spot and a car takes up 2 adjacent parking spots. A string of length n can represent n adjacent parking spots using % for a motorcycle, <> for a car, and . for an empty spot.

For example: '.%%.<><>' (Thanks to the Berkeley Math Circle for introducing this question.)

Implement **count_park**, which returns the number of ways that vehicles can be parked in n adjacent parking spots for positive integer n. Some or all spots can be empty.

```
def count_park(n):
    """Count the ways to park cars and motorcycles in n adjacent spots.
    >>> count_park(1)  # '.' or '%'
    2
    >>> count_park(2)  # '..', '.%', '%.', '%%', or '<>'
    5
    >>> count_park(4)  # some examples: '<><>', '.%%.', '%<>%', '%.<>'
    29
    """
    if n < 0:
        return 0
    elif n == 0:
        return 1
    else:
        return count_park(n-1) + count_park(n-1) + count_park(n-2)
```

One way to think about these base cases: which recursive calls lead to these cases, and what should their values be?

count_park(3):

%%%
%%.
%.%
%..
%<>
.%%
.%.
..%
...
.<>
<>%
<>.

Spring 2023 Midterm 2 Question 5(a) [modified a bit]