

Max Product

Write a function that takes in a list and returns the maximum product that can be formed using non-consecutive elements of the list. All numbers in the input list are greater than or equal to 1.

```
A tip for finding a recursive process:
                                                       1. Pick an example: s = [5, 10, 5, 10, 5]
def max product(s):
                                                       2.Write down what recursive calls will do:
    """Return the maximum product that can be
                                                        - \max_{product([10, 5, 10, 5]) \rightarrow 10 * 10
    formed using non-consecutive elements of s.
                                                        - max product([5, 10, 5])
                                                                                        \rightarrow 5 * 5
                                                        - max product([10, 5])
                                                                                        → 10
    >>> max_product([10, 3, 1, 9, 2]) # 10 * 9
                                                        - max product([5])
                                                                                        → 5
    90
                                                       3.Which one helps build the result?
    >>> max product([5, 10, 5, 10, 5]) # 5 * 5 * 5
    125
    >>> max_product([])
                                           Either include s[0] but not s[1], OR
    1
                                                    Don't include s[0]
    .....
    if len(s) == 0:
                                                  Choose the larger of:
        return 1
                            multiplying s[0] by the max_product of s[2:] (skipping s[1]) OR
    elif len(s) == 1:
                                            just the max product of s[1:] (skipping s[0])
        return s[0]
    else:
                                   max(s[0] * max_product(s[2:]), max_product(s[1:]))
        return ____
```

Sum Fun

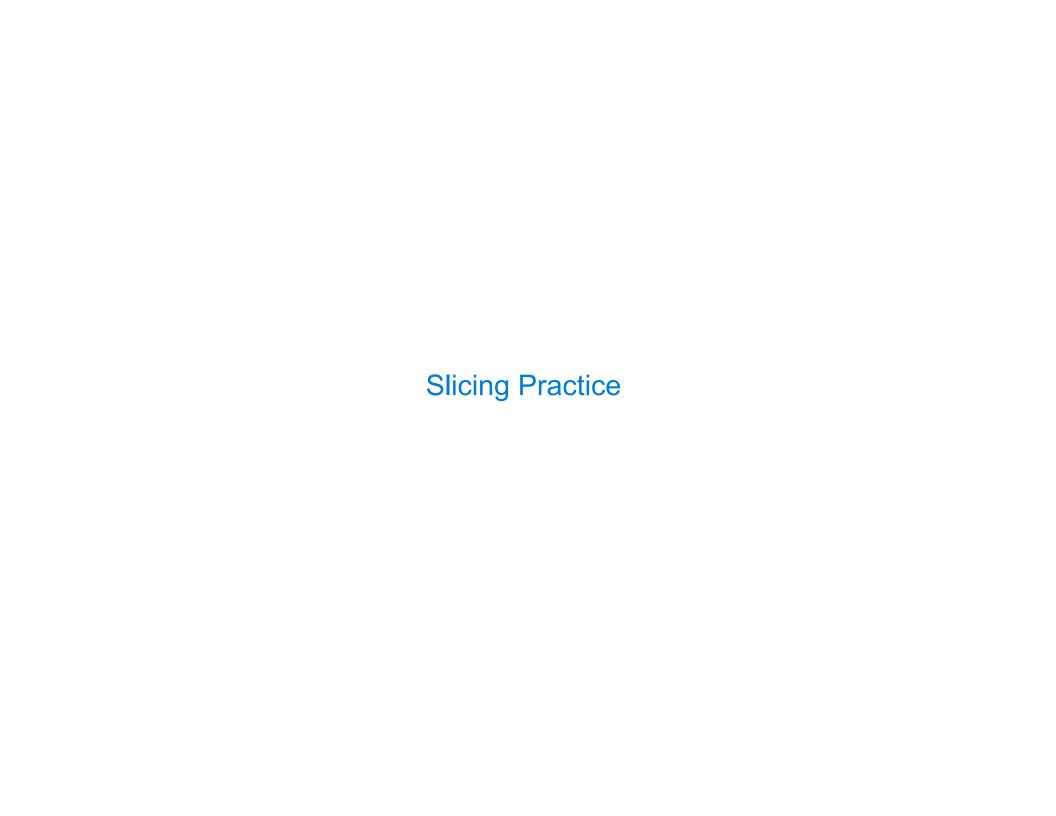
```
Implement sums(n, m), which takes a total n and maximum m. It returns a list of all lists:

    that sum to n,

    that contain only positive numbers up to m, and

                                                                  [1, 3, 1] = [1] + [3, 1]
• in which no two adjacent numbers are the same.
                                                                  [2, 1, 2] = [2] + [1, 2]
                                                                  [2, 3] = [2] + [3]
>>> sums(5, 3)
                                                                  [3, 2] = [3] + [2]
[[1, 3, 1], [2, 1, 2], [2, 3], [3, 2]]
                                                                 \frac{1}{1} = [1] + [1], 3
>>> sums(5, 5)
[[1, 3, 1], [1, 4], [2, 1, 2], [2, 3], [3, 2], [4, 1], [5]]
                                                                 \frac{1}{2} = [1] + [2, 2]
def sums(n, m):
    if n < 0:
        return []
    if n == 0:
        sums to zero = [] # The only way to sum to zero using positives
        return [sums_to_zero] # Return a list of all the ways to sum to zero
    result = []
    for k in range(1, m + 1): [k]+rest
                                                    sums(n-k,m)
                                                                                k != rest[0]
        result = result + [ _____ for rest in _____ if rest == [] or _____ ]
    return result
```

5



Spring 2023 Midterm 2 Question

Definition. A prefix sum of a sequence of numbers is the sum of the first n elements for some positive length n.

(a) (4.0 points)

def prefix(s):

Implement prefix, which takes a list of numbers s and returns a list of the prefix sums of s in increasing order of the length of the prefix.



Parking

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 **park**, which <u>returns a list</u> of all the ways, represented as strings, that vehicles can be parked in n adjacent parking spots for positive integer n. Spots can be empty.

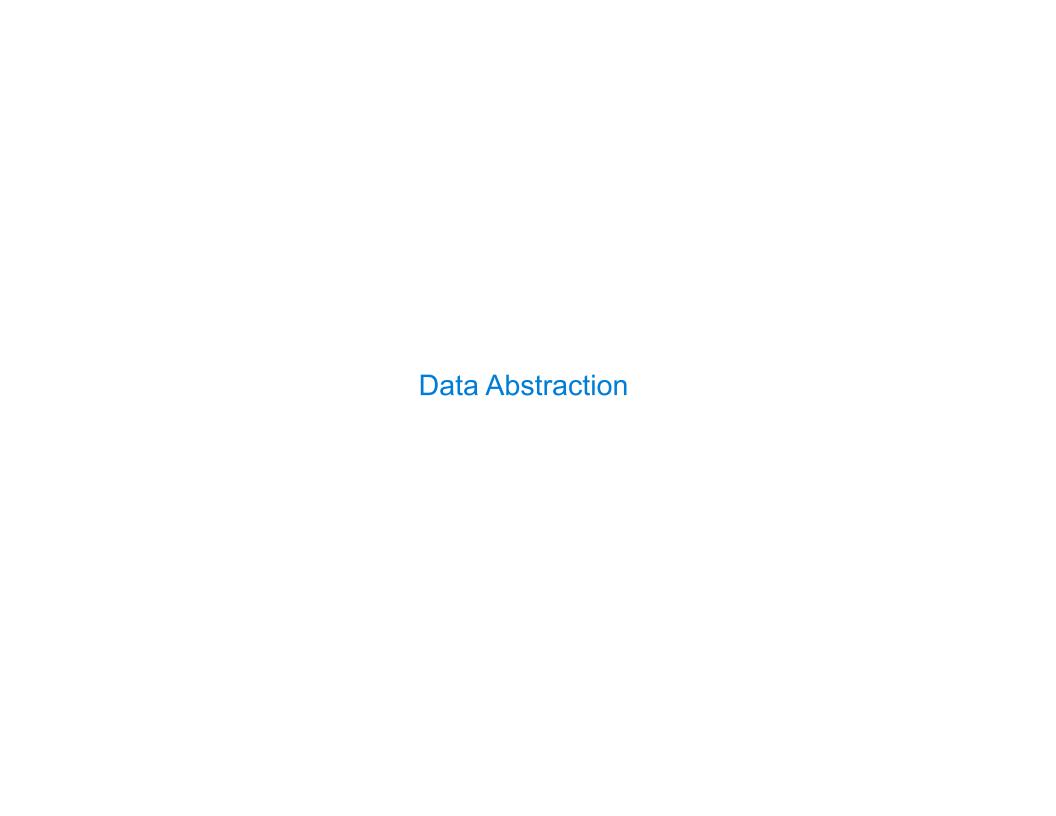
```
park(3):
def park(n):
    """Return the ways to park cars and motorcycles in n adjacent spots.
                                                                                        %%%
    >>> park(1)
                                                                                         %%.
    ['%', '.']
                                                                                        %.%
    >>> park(2)
                                                                                        %..
    ['%', '%.', '.%', '...', '<>']
                                                                                         %<>
    >>> len(park(4)) # some examples: '<><>', '.%%.', '%<>%', '%.<>'
                                                                                         .%%
    29
                                                                                         .%.
    .....
    if n < 0:
        return
                                                                                         .<>
    elif n == 0:
                                                                                         <>%
        return ['']
                                                                                         <>.
    else:
        return _____ ['%'+s for s in park(n-1)] + ['.'+s for s in park(n-1)] + ['<>'+s for s in park(n-2)]
```

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>}
```



Data Abstraction

```
A small set of functions enforce an abstraction barrier between representation and use

How data are represented (as some underlying list, dictionary, etc.)

How data are manipulated (as whole values with named parts)

E.g., refer to the parts of a line (affine function) called f:

*slope(f) instead of f[0] or f['slope']

*y_intercept(f) instead of f[1] or f['y_intercept']

Why? Code becomes easier to read & revise; later you could represent a line f as two points instead of a [slope, intercept] pair without changing code that uses lines.
```