Data Abstraction
Announcements
Discussion 4
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.

def max_product(s):
    """Return the maximum product that can be formed using non-consecutive elements of s."
    if len(s) == 0:
        return 1
    elif len(s) == 1:
        return s[0]
    else:
        return max(s[0] * max_product(s[2:]), max_product(s[1:]))

A tip for finding a recursive process:
1. Pick an example: s = [5, 10, 5, 10, 5]
2. Write down what recursive calls will do:
   - max_product([10, 5, 10, 5]) → 10 * 10
   - max_product([5, 10, 5]) → 5 * 5
   - max_product([10, 5]) → 10
   - max_product([5]) → 5
3. Which one helps build the result?

Either include s[0] but not s[1], OR
Don't include s[0]

Choose the larger of:
multiplying s[0] by the max_product of s[2:] (skipping s[1]) OR
just the max_product of s[1:] (skipping s[0])

max(s[0] * max_product(s[2:]), max_product(s[1:]))
### 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
- in which no two adjacent numbers are the same.

```python
def sums(n, m):
    if n < 0:
        return []
    if n == 0:
        sums_to_zero = []
        return [sums_to_zero]
    result = []
    for k in range(1, m + 1):
        [k]+rest = [k] + rest
        sums(n-k,m) = sums(n-k,m)
        k != rest[0] = k != rest[0]
        result = result + [_________ for rest in ________ if rest == [] or ________]
    return result
```

```python
>>> sums(5, 3)
[[1, 3, 1], [2, 1, 2], [2, 3], [3, 2]]

>>> sums(5, 5)
[[1, 3, 1], [1, 4], [2, 1, 2], [2, 3], [3, 2], [4, 1], [5]]
```
Slicing Practice
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)

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.

```python
def prefix(s):
    """Return a list of all prefix sums of list $s$.
    ""
    sum(s[:k+1]) range(len(s))
    return [________ for k in _______]
    (a) (b)

>>> prefix([1, 2, 3, 0, 4, 5])
[1, 3, 6, 6, 10, 15]
>>> prefix([2, 2, 2, 0, -5, 5])
[2, 4, 6, 6, 1, 6]
""
```

ii. (1.0 pt) Fill in blank (b).

- $s$
- $[s]$
- $s[1:]$
- `range(s)`
- `range(len(s))`
Tree Recursion with Strings
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 returns a list 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.

```python
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 \text{exp}>: <value \text{exp}> \text{for} <name> \text{in} <\text{iter}\ \text{exp}> \text{if} <\text{filter}\ \text{exp}>\}\n
Short version: \{<key \text{exp}>: <value \text{exp}> \text{for} <name> \text{in} <\text{iter}\ \text{exp}>\}
Data Abstraction
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.