Warmup (L23)

- Write a recursive function to get the prime factors of a number.
- Descent condition: Divide out factors, so number approaches 1. (Ignore non-positive numbers)
- Base case: 1 has no factors
- Recursive thought process: Find a factor and divide it out. Assume that factorizing that smaller number is already done. Append our newly found factor.
- Note: Finding a single factor should involve a loop, but finding *every* factor should recurse.

Sorted search

- If you’re looking at a library shelf for a book by a specific author, you don’t go one-by-one through every book
- If the list is sorted, you can make a decent guess as to where the value is
- Guesswork is human behavior, so in programming, we split the list in half and look in the appropriate half

Binary search

- Descent condition: List gets smaller (approaches empty list)
- Base case: Empty list doesn’t have it
- Recursive thought process: Look at an item in the middle. If it’s what you’re looking for, we’re done. Otherwise, by comparing it to what you’re looking for, you can choose which half the item you’re looking for might be in. Assume that the search is already implemented for any smaller list, so for either half.

Binary search

```
def sortedSearch(lst: list, needle: typing.Any) -> bool:
    if len(lst) == 0:
        return False
    midpoint = len(lst) // 2
    if lst[midpoint] == needle:
        return True
    elif lst[midpoint] < needle:
        return sortedSearch(lst[midpoint+1:], needle)
    else: # lst[midpoint] > needle
        return sortedSearch(lst[:midpoint], needle)
```

Understanding binary search

Once again, let’s add some prints to understand what’s happening in our code

Is recursion always about lists?

- We’re about to see a bunch of examples with lists
- Some programming languages are designed for you to use recursion everywhere
- Python is not
- Divide-and-conquer is the usual reason for recursion in Python, which means we need something to divide
- So, recursion is usually for dividable lists

Other ways of dividing

CS114 (M10)

“Divide” and conquer

- It’s common to divide in half…
- but it’s not necessary.
- Let’s consider a case where a different division is necessary

Excursion

- Consider recording how far a bunch of samples deviated from an expected mean
- If your expected mean is correct, then over a long period, this should add to about zero
- The longest subsequence that's off by some threshold is the longest *excursion*

Longest excursion

- Descent condition: List gets smaller (approaches empty list)
- Base case: Empty list has no excursion (longest excursion length is zero)
- Recursion thought process: If I have a list from 0 to *n* that does *not* exceed my threshold, *any* sublist could. If I assume all shorter lists are already implemented, then checking (0, *n*-1) and (1, *n*) will cover every sublist.

Longest excursion

```python
def longestExcursion(
    lst: list[float], threshold: float
) -> int:
    if len(lst) == 0:
        return 0
    s = sum(lst)
    if s > threshold or s < -threshold:
        return len(lst)
    l = longestExcursion(lst[:-1], threshold)
    r = longestExcursion(lst[1:], threshold)
    if l > r:
        return l
    else:
        return r
```

This recursion is bad…

- Let’s add prints to see how `longestExcursion` behaves
- It sure is looking at the same list over and over again…
- Let’s draw what’s happening on the board to see why it’s happening
- But, doing this with loops would be *really* hard!

In-lecture quiz (L23)

- https://student.cs.uwaterloo.ca/~cs114/F25/quiz/
- Q1: How many numbers does this print?

```
def strange(n: int) -> int:
    print(n)
    if n <= 0:
        return 0
    n = strange(n // 10)
    print(n)
    return n + 1
strange(1234)
```

A. None, an error, or it recurses forever
B. 4
C. 5
D. 8
E. 9

In-lecture quiz (L23)

- https://student.cs.uwaterloo.ca/~cs114/F25/quiz/
- Q2: How many numbers does this print?

```
def strange(n: int) -> int:
    print(n)
    if n >= 10000:
        return 0
    n = strange(n * 10)
    print(n)
    return n + 1
strange(1234)
```

A. None, an error, or it recurses forever
B. 2
C. 3
D. 4
E. 5

Avoiding repeats

CS114 (M10)

Avoiding repeats

- The problem with `longestExcursion` was that it checked the same sublists over and over again
- For example, with [1, 2, 3], both [1, 2] and [2, 3] would then check [2].
- In this case, the solution is simple: Once you start going right, only go right
    - (Or once you start going left, only go left)

```
def longestExcursionPrime(
    lst: list[float], threshold: float, right: bool
) -> int:
    if len(lst) == 0:
        return 0
    s = sum(lst)
    if s > threshold or s < -threshold:
        return len(lst)
    r = longestExcursionPrime(lst[1:], threshold, True)
    if not right:
        l = longestExcursion(lst[:-1], threshold, False)
        if l > r:
            return l
        else:
            return r
    else:
        return r

def longestExcursion(
    lst: list[float], threshold: float
) -> int:
    return longestExcursionPrime(lst, threshold, False)
```

```
def longestExcursionPrime(
    lst: list[float], threshold: float, right: bool
) -> int:
    if len(lst) == 0:
        return 0
    s = sum(lst)
```

It would be ugly to make the `longestExcursion` function take this weird extra parameter that's only meant to fix a recursion problem, so I made a helper function that carries the extra parameter instead, and made *that* recurse.

```
    else:
        return r

def longestExcursion(
    lst: list[float], threshold: float
) -> int:
    return longestExcursionPrime(lst, threshold, False)
```

```
def longestExcursionPrime(
    lst: list[float], threshold: float, right: bool
) -> int:
    if len(lst) == 0:
        return 0
    s = sum(lst)
```

“Prime” is just my naming standard for helper functions like this.
You’ll see lots of other names.

```
        if l > r:
            return l
        else:
            return r
    else:
        return r

def longestExcursion(
    lst: list[float], threshold: float
) -> int:
    return longestExcursionPrime(lst, threshold, False)
```

Avoiding repeats

- We had the same problem with Fibonacci, but that one is very hard to solve
- There’s no universal solution to avoiding repetition with recursion
- Try to think about a small case all the way through, and use that to work out big cases

Insertion sort

CS114 (M10)

Insertion sort

- Keep a list sorted by inserting things in their correct location instead of appending to the end
- Like binary search, but instead of checking if the value is there, *put* it there
- Can’t slice the list, since that makes a new list; we want to insert the value!

Insertion sort

- Descent condition: Segment of list we’re considering gets smaller (upperbound minus lowerbound approaches zero)
- Base case: upperbound == lowerbound (insert here)
- Recursive thought process: Look at a middle point. Insert either before or after that middle point. Assume inserting into a smaller range is done.

```python
def insertSortedPrime(
    lst: list, val: typing.Any, lb: int, ub: int
) -> int:
    if ub == lb:
        lst.insert(lb, val)
        return lb
    midpoint = (ub - lb) // 2 + lb
    mid = lst[midpoint]
    if val < mid:
        return insertSortedPrime(lst, val, lb, midpoint)
    else:
        return insertSortedPrime(lst, val, midpoint+1, ub)

def insertSorted(lst: list, val: typing.Any) -> int:
    return insertSortedPrime(lst, val, 0, len(lst))
```

Merge sort

CS114 (M10)

How do you sort?

- How do `lst.sort` and `sorted` actually work?
- There are many algorithms, but we'll show you a classic one: merge sort
- Intuition: It's much easier to merge two lists that are already sorted than it is to sort a list

- This part isn’t recursive

```
def merge(a: list, b: list) -> list:
    r = []
    while len(a) > 0 and len(b) > 0:
        if b[0] < a[0]:
            r.append(b[0])
            b.pop(0)
        else:
            r.append(a[0])
            a.pop(0)
    return r + a + b
```

Where’s the recursion?

- Descent condition: List becomes smaller (approaches length 1 or empty)
- Base case: An empty list or a list of length 1 is sorted
- Recursive thought process: Assume we can already sort half the list. Sort both halves, then merge the sorted lists.

Merge sort

```
def mergeSort(lst: list) -> list:
    if len(lst) <= 1:
        return lst
    midpoint = len(lst) // 2
    l = mergeSort(lst[:midpoint])
    r = mergeSort(lst[midpoint:])
    return merge(l, r)
```

Is merge sort any good?

- Merge sort duplicates things a lot
- In terms of efficiency (next module), it’s actually the same as any other sort

Module summary

CS114 (M10)

Module summary

- Recursion:
  - Base case
  - Recursive case
  - Descent condition
- Divide-and-conquer for lists
  - Binary search
  - Longest excursion/avoiding repeats
  - Insertion sort
  - Merge sort