Outline

assignment # 2 questions

more big-oh, better sorts
is_regex(s)

Returns True if the string s is a valid regular expression, False otherwise. Think about...

- simplest expressions — how can you check for these and reject many strings?

- binary expressions — | and . — how can you check for these? How can you break up the remainder of the string so that you can check it?

- unary expressions — * — how can you check for these? How can you break up the remainder of the string so that you can check it?
all_regex_permutations(s)

Returns a set (could be empty) of permutations of s that are valid regular expressions. Think about...

- how to produce a set of permutations? There is lots of code laying about, including in week 4 of this course’s calendar

- filter out any permutation that isn’t a regex — it would sure be nice to have some code that could test whether a string were a regex...

- a string of length n has n-factorial permutations — producing an impractically large set for n > 8.
  → We will only test your code on strings of length ≤ 8.
**regex_match(r, s)**

Returns **True** if string s matches the regular expression equivalent to the tree rooted at r, **False** otherwise. Think about...

- you may assume that r is an instance of one of the specialized regular expression tree classes in regextree.py
- what are the simplest cases of string s to consider?
- if the symbol at the root of r is a |, what do you need to check?
- if the symbol at the root of r is a ., what do you need to check?
- if the symbol at the root of r is a *, what do you need to check? **This is the hardest case; complete the others first.**

(more on this next slide)
debugging regex_match tip

doctests only using 1 2 e .
doctests only using 1 2 e |
doctests only using 1 2 e *
doctests only using 1 2 e | .
doctests only using 1 2 e . *
doctests only using 1 2 e | *
doctests using all the symbols etc
star regexes...

The handout says that a string $s$ matches a regular expression $r^*$ (where $r$ is the child regular expression) if and only if:

- $s$ is the empty string — pretty easy to check OR

- $s = s_1 + s_2 + \cdots + s_k$ where each $s_i$ matches the child regular expression $r$. This seems harder to check — so many ways to break up $s$!

- equivalently (why?) $s = s_1 + s_2$, where $s_1$ matches the child regular expression $r$ and $s_2$ matches $r^*$ — now you only have to check every possible way to break $s$ into two pieces.
build_regex_tree(r)

Return the regular expression tree equivalent to the valid (we promise) regular expression regex. Think about:

- very similar thinking to is_regex

- instead of checking whether regex is a regular expression (you are guaranteed that it is), you have to break it into a few pieces to determine which sort of regular expression tree, and provide input strings to form its children (if any)

- strangely, that’s all there is to do!
a digression...

what could go wrong?

def f(n: int, L: list=[]) -> list:
    L.append(n)
    return L

>>> f(10)
[10]

>>> f(9)
[10,9]

or

>>> X = [[]]*3

>>> X[0].append(1)

>>> X
[[1],[1],[1]]
quick sort

idea:

- somehow choose a pivot element
- move everything smaller than the pivot to one list (call it left) and everything larger than the pivot to another list (call it right).
- quicksort the sublists left and right (two recursive calls)
- now sorted list is left followed by the pivot followed by right
quick sort code

def quick(L):
    if len(L) > 1:
        # there are much better ways of choosing the pivot!
        pivot = L[0]
        smaller_than_pivot = [x for x in L[1:] if x < pivot]
        larger_than_pivot = [x for x in L[1:] if x >= pivot]
        return (quick(smaller_than_pivot) +
                [pivot] +
                quick(larger_than_pivot))
    else:
        return L
quick sort performance

- how many times do we choose the pivot?

  $\mathcal{O}(n)$

  more specifically $n +$ some constant

- how many steps each time we choose a pivot?
  linear in the size of the sublist... which gets smaller after each recursive call
merge sort

idea:

- divide the list in half
- mergesort the two halves (two recursive calls)
- merge the two sorted halves in linear time
merge code

```python
def merge(L1:list, L2:list) -> list:
    """return merge of L1 and L2
    NOTE: modifies L1 and L2"

decreasing_from_largest = []
while L1 and L2:
    if L1[-1] > L2[-1]:
        decreasing_from_largest.append(L1.pop())
    else:
        decreasing_from_largest.append(L2.pop())

decreasing_from_largest.reverse()
return L1 + L2 + decreasing_from_largest
```
merge sort code

def merge_sort(L):
    """Produce copy of L in non-decreasing order"

    >>> merge_sort([1, 5, 3, 4, 2])
    [1, 2, 3, 4, 5]
    """
    if len(L) < 2:
        return L
    else:
        left_sublist = L[:len(L)//2]
        right_sublist = L[len(L)//2:]
        return merge(merge_sort(left_sublist),
                      merge_sort(right_sublist))
merge sort performance

- how many times do we split the list in half?

\[ \mathcal{O}(n) \]

more specifically \( n + \) some constant

- how many steps each time we split?
linear in the size of the sublist... which has size \( \approx n/2^d \)
when we’re \( d \) function calls deep into the recursion.
how do we know merge sort runs in time $O(n \log n)$?

- Splitting a size $n$ list into two halves takes constant time or time $O(n)$ depending on the data structure.
- Merging two sorted lists of size $n/2$ each takes time $O(n)$.
- So the split/merge tasks together run in linear time.
- Which means there are constants $c_0$, $d$ such that $c_0 n + d$ is an upper bound on the runtime.
- Let $c = c_0 + d$. Then $c \geq c_0 n + d$ for all $n \geq 1$.
- So $cn$ is also a bound on the runtime for the split/merge tasks.
- We do the split/merge tasks once on a size $n$ list (the input) - takes time $cn$.
- We do those tasks 2 times on size $n/2$ sublists - takes time $2(c(n/2)) = cn$.
- We do those tasks 4 times on size $n/4$ sublists - takes time $4(c(n/4)) = cn$.
- ...
how do we know merge sort runs in time $O(n \log n)$?

- So $cn$ is also a bound on the runtime for the split/merge tasks.
- We do the split/merge tasks once on a size $n$ sublist (the input) - takes time $cn$.
- We do the split/merge tasks 2 times on size $n/2$ sublists - takes time $2(c(n/2)) = cn$.
- We do the split/merge tasks $2^d$ times on size $n/2^d$ sublists - takes time $2^d(c(n/2^d)) = cn$.

And that is all the work we do!

When $d = \log n$, the sublists have size 1, in which case we don’t do any more recursive calls.

So runtime =

$$
\sum_{d=1}^{\log n} \text{(time spent on size } n/2^d \text{ lists)} = \sum_{d=1}^{\log n} c n = c n \log n - c n
$$
scaling:

How well do these various sorts perform as the size of the problem (list length) increases? Time and compare.