CSC148 winter 2014

sorting big-ooh

week 10

Danny Heap
heap@cs.toronto.edu
BA4270 (behind elevators)
http://www.cdf.toronto.edu/~heap/148/W14/
416-978-5899

March 19, 2014
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 $n > 8$. 
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? (more on this next slide)
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!
quick sort

idea: choose a pivot; decide where the pivot goes with respect to the rest of the list, repeat on the partitions...
a digression...

what could go wrong?

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

- how many times do we choose the pivot?

- how many steps each time we choose a pivot?
merge sort

idea: divide the list in half, (merge) sort the halves, then merge the sorted results
merge sort performance

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

- how many steps each time we split?
scaling:

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