Announcements

Final lab this week, no prep for next week.

Posted: Assignment 2 FAQ, office hour schedule, sample tests, melody contest.

CSSU.ca/outreach
Splitting lists, divide-and-conquer

1. **Divide** the input list into smaller lists.
2. **Recurse** on each smaller list.
3. **Combine** the results of each recursive call.
mergesort and quicksort
Running time demo

mergesort, quicksort, and insertion sort
Worksheet!

How do we analyse the running time of recursive algorithms in general? (Not just for trees.)

Two key parts:
- how long do the non-recursive parts take?
- what is the structure of the recursive calls?
def mergesort(lst):
    if len(lst) < 2:
        return lst[:]
    else:
        mid = len(lst) // 2
        left = lst[:mid]
        right = lst[mid:]
        left_sorted = mergesort(left)
        right_sorted = mergesort(right)
        return _merge(left_sorted, right_sorted)

- length $n$
- non-rec part takes $O(n)$ time

$\frac{n}{2} + \frac{n}{2} = n$
(list length 8)

(len 4)

(len 2)

(len 1)
\[ n + n + n + \ldots + n = \Theta(n \log n) \]

\[ \text{len n} \]

\[ \approx \log n \text{ levels} \]
def quicksort(lst):
    if len(lst) < 2:
        return lst[:]
    else:
        pivot = lst[0]
        smaller, bigger = _partition(lst[1:], pivot)
        smaller_sorted = quicksort(smaller)
        bigger_sorted = quicksort(bigger)
        return smaller_sorted + [pivot] + bigger_sorted
IF always
pick median

$O(n \log n)$
If always pick min

\[ n \]
\[ n-1+1 \]
\[ n-2+1 \]
\[ n-3+1 \]

\[ \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix} \]
\[ O(n^2) \]

* len 0 list takes 1 step
Quicksort: in theory, a mixed bag

If we always choose a pivot that’s an *approximate median*, then the two partitions are roughly equal, and the running time is $O(n \log(n))$.

If we always choose a pivot that’s an approximate min/max, then they two partitions are very unequal, and the running time is $O(n^2)$. 
The limitations of Big-Oh

Big-Oh notation is a simplification of running time analysis, and allows us to ignore constants when analysing efficiency.

But constants can make a difference, too!

\( O(n \log n) \) vs. \( O(n \log n) \) vs. \( O(n^2) \)
Assignment 2 due tonight!

Check the FAQ on the course forum.
Run the sample tests, and your own tests.
Polish and run PyTA to check your work.
Check your remaining grace tokens.
Office hours will be busy. Come prepared with questions; the more specific you are, the more we can help. 😊
In-place quicksort

MUTATING THE INPUT LIST IN A SPACE-EFFICIENT WAY.
The key helper: in-place partition

```
10 | 7 | 20 | 30 | 3 | 6
```

```
10 <= | ??? | >
```

```
10 | 7 | 3 | 6 | 20 | 30
```
def quicksort(lst):
    if len(lst) < 2:
        return lst[:]
    else:
        pivot = lst[0]

        smaller, bigger = _partition(lst[1:], pivot)

        smaller_sorted = quicksort(smaller)
        bigger_sorted = quicksort(bigger)

        return smaller_sorted + [pivot] + bigger_sorted
Simulating slicing with indexes

We often want to operate on just part of a list:
- \( f(lst[start:end]) \)

Rather than create a new list object, we pass in the indexes:
- \( f(lst, \text{start}, \text{end}) \)
Simulating slicing with indexes

_in_place_partition(lst) \rightarrow
_in_place_partition(lst, start, end)

quicksort(lst) \rightarrow
_in_place_quicksort(lst, start, end)
Lessons in efficiency

A CASE STUDY IN COMPARING SORTING ALGORITHMS
1. Big-Oh describes behaviour as input size grows
2. An algorithm can be “good on average” and “bad in the worst case”
3. Big-Oh is *not* good at predicting behaviour on small inputs.
4. Saving space doesn’t always mean saving time!
5. Hard work doesn’t always mean saving time, either!
6. But sometimes hard work pays off.*
RecursionError and the limitation of recursion