Recitation 3

Sorting

Sorting an array `A` of comparable items into increasing order is a common subtask of many computational problems. Insertion sort and selection sort are common sorting algorithms for sorting small numbers of items because they are easy to understand and implement. Both algorithms are **incremental** in that they maintain and grow a sorted subset of the items until all items are sorted. The difference between them is subtle:

- **Insertion sort** maintains and grows a subset of the **first** `i` input items in sorted order.
- **Selection sort** maintains and grows a subset the **largest** `i` items in sorted order.

Insertion Sort

Here is a Python implementation of insertion sort. Having already sorted sub-array `A[:i]`, the algorithm repeatedly swaps item `A[i]` with the item to its left until the left item is no larger than `A[i]`. As can be seen from the code, insertion sort can require $\Omega(n^2)$ comparisons and $\Omega(n^2)$ swaps in the worst case.

```
def insertion_sort(A):
    '''Insertion sort array A'''
    for i in range(1, len(A)):              # O(n) loop over array
        for j in range(i, 0, -1):           # O(i) loop over sub-array
            if A[j - 1] <= A[j]:            # O(1) check if swap needed
                break                       # O(1) no swap needed
            A[j - 1], A[j] = A[j], A[j - 1] # O(1) swap
```

Selection Sort

Here is a Python implementation of selection sort. Having already sorted the largest items into sub-array `A[i+1:]`, the algorithm repeatedly scans the array for the largest item not yet sorted and swaps it with item `A[i]`. As can be seen from the code, selection sort can require $\Omega(n^2)$ comparisons, but will perform at most $O(n)$ swaps in the worst case.

```
def selection_sort(A):
    '''Selection sort array A'''
    for i in range(len(A) - 1, 0, -1):      # O(n) loop over array
        m = i                               # O(1) initial index of max
        for j in range(i, 0, -1):           # O(i) search for max in A[:i]
            if A[m] < A[j]:                 # O(1) check for larger value
                m = j                       # O(1) new max found
        A[m], A[i] = A[i], A[m]             # O(1) swap
```

In-place and Stability

Both insertion sort and selection sort are **in-place** algorithms, meaning they can each be implemented using at most a constant amount of additional space. The only operations performed on the array are comparisons and swaps between pairs of elements. Insertion sort is **stable**, meaning that items having the same value will appear in the sort in the same order as they appeared in the input array. By comparison, this implementation of selection sort is not stable. For example, the input $(2, 1, 1')$ would produce the output $(1', 1, 2)$.

Merge Sort

In lecture, we introduced **merge sort**, an asymptotically faster algorithm for sorting large numbers of items. The algorithm recursively sorts the left and right half of the array, and then merges the two halves in linear time. The recurrence relation for merge sort is then $T(n) = 2T(n/2) + \Theta(n)$, which solves to $T(n) = \Theta(n \log n)$. An $\Theta(n \log n)$ asymptotic growth rate is **much closer** to linear than quadratic, as $\log n$ grows exponentially slower than n. In particular, $\log n$ grows slower than any polynomial n^ε for $\varepsilon > 0$.

```
def merge_sort(A, l = 0, r = 0, temp = None):
    '''Merge sort sub-array A[l:r] using temp as auxiliary storage'''
    if temp is None:                          # O(1) initial call
        r, temp = len(A), [None] * len(A)     # O(n) allocate temp
    if 1 < r - l:                             # O(1) size k = r - l
        c = (l + r + 1) // 2                  # O(1) compute center
        merge_sort(A, l, c, temp)             # T(k/2) recursively sort left
        merge_sort(A, c, r, temp)             # T(k/2) recursively sort right
        # merge ranges A[l:c] and A[c:r]
        t, p1, p2 = l, l, c                   # O(1) initialize pointers
        while t != r:                         # O(k) fill temp storage
            if p2 == r or (p1 < c and A[p1] < A[p2]):   # O(1) check side
                temp[t] = A[p1]               # O(1) merge from left
                p1 += 1                       # O(1) increment left pointer
            else:
                temp[t] = A[p2]               # O(1) merge from right
                p2 += 1                       # O(1) increment right pointer
            t += 1                            # O(1) increment temp pointer
        A[l:r] = temp[l:r]                    # O(k) copy back to array
```

Merge sort uses a linear amount of temporary storage (`temp`) when combining the two halves, so it is **not in-place**. While there exist algorithms that perform merging using no additional space, such implementations are substantially more complicated than the merge sort algorithm. Whether merge sort is stable depends on how an implementation breaks ties when merging. The above implementation is not stable, but it can be made stable with only a small modification. Can you modify the implementation to make it stable? We've made CoffeeScript visualizers for the merge step of this algorithm, as well as one showing the recursive call structure. You can find them here:
`https://codepen.io/mit6006/pen/RYJdOG` `https://codepen.io/mit6006/pen/wEXOOq`