Recitation 3

Recall that in Recitation 2 we reduced the Set interface to the Sequence Interface (we simulated one with the other). This directly provides a Set data structure from an array (albeit a poor one).

Data Structure	Operations $O(\cdot)$				
	Container	Static	Dynamic	Order	
	`build(X)`	`find(k)`	`insert(x)` `delete(k)`	`find_min()` `find_max()`	`find_prev(k)` `find_next(k)`
Array	n	n	n	n	n

We would like to do better, and we will spend the next five lectures/recitations trying to do exactly that! One of the simplest ways to get a faster Set is to store our items in a **sorted** array, where the item with the smallest key appears first (at index 0), and the item with the largest key appears last. Then we can simply binary search to find keys and support Order operations! This is still not great for dynamic operations (items still need to be shifted when inserting or removing from the middle of the array), but finding items by their key is much faster! But how do we get a sorted array in the first place?

Data Structure	Operations $O(\cdot)$				
	Container	Static	Dynamic	Order	
	`build(X)`	`find(k)`	`insert(x)` `delete(k)`	`find_min()` `find_max()`	`find_prev(k)` `find_next(k)`
Sorted Array	?	$\log n$	n	1	$\log n$

```
class Sorted_Array_Set:
    def __init__(self):     self.A = Array_Seq()   # O(1)
    def __len__(self):      return len(self.A)     # O(1)
    def __iter__(self):     yield from self.A      # O(n)
    def iter_order(self):   yield from self        # O(n)

    def build(self, X):                            # O(?)
        self.A.build(X)
        self._sort()

    def _sort(self):                               # O(?)
        ??

    def _binary_search(self, k, i, j):             # O(log n)
        if i >= j:          return i
        m = (i + j) // 2
        x = self.A.get_at(m)
        if x.key > k:       return self._binary_search(k, i, m - 1)
        if x.key < k:       return self._binary_search(k, m + 1, j)
```

```
20          return m
21
22      def find_min(self):                             # O(1)
23          if len(self) > 0:       return self.A.get_at(0)
24          else:                   return None
25
26      def find_max(self):                             # O(1)
27          if len(self) > 0:       return self.A.get_at(len(self) - 1)
28          else:                   return None
29
30      def find(self, k):                              # O(log n)
31          if len(self) == 0:      return None
32          i = self._binary_search(k, 0, len(self) - 1)
33          x = self.A.get_at(i)
34          if x.key == k:          return x
35          else:                   return None
36
37      def find_next(self, k):                         # O(log n)
38          if len(self) == 0:      return None
39          i = self._binary_search(k, 0, len(self) - 1)
40          x = self.A.get_at(i)
41          if x.key > k:           return x
42          if i + 1 < len(self):   return self.A.get_at(i + 1)
43          else:                   return None
44
45      def find_prev(self, k):                         # O(log n)
46          if len(self) == 0:      return None
47          i = self._binary_search(k, 0, len(self) - 1)
48          x = self.A.get_at(i)
49          if x.key < k:           return x
50          if i > 0:               return self.A.get_at(i - 1)
51          else:                   return None
52
53      def insert(self, x):                            # O(n)
54          if len(self.A) == 0:
55              self.A.insert_first(x)
56          else:
57              i = self._binary_search(x.key, 0, len(self.A) - 1)
58              k = self.A.get_at(i).key
59              if k == x.key:
60                  self.A.set_at(i, x)
61                  return False
62              if k > x.key:   self.A.insert_at(i, x)
63              else:           self.A.insert_at(i + 1, x)
64          return True
65
66      def delete(self, k):                            # O(n)
67          i = self._binary_search(k, 0, len(self.A) - 1)
68          assert self.A.get_at(i).key == k
69          return self.A.delete_at(i)
```

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:

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

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):                  # 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
```

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
        j = i                               # O(1) initialize pointer
        while j > 0 and A[j] < A[j - 1]:    # O(i) loop over prefix
            A[j - 1], A[j] = A[j], A[j - 1] # O(1) swap
            j = j - 1                       # O(1) decrement j
```

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, a = 0, b = None):          # Sort sub-array A[a:b]
    if b is None:                            # O(1) initialize
        b = len(A)                           # O(1)
    if 1 < b - a:                            # O(1) size k = b - a
        c = (a + b + 1) // 2                 # O(1) compute center
        merge_sort(A, a, c)                  # T(k/2) recursively sort left
        merge_sort(A, c, b)                  # T(k/2) recursively sort right
        L, R = A[a:c], A[c:b]                # O(k) copy
        i, j = 0, 0                          # O(1) initialize pointers
        while a < b:                         # O(n)
            if (j >= len(R)) or (i < len(L) and L[i] < R[j]): # O(1) check side
                A[a] = L[i]                  # O(1) merge from left
                i = i + 1                    # O(1) decrement left pointer
            else:
                A[a] = R[j]                  # O(1) merge from right
                j = j + 1                    # O(1) decrement right pointer
            a = a + 1                        # O(1) decrement merge pointer
```

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`

Build a Sorted Array

With an algorithm to sort our array in $\Theta(n \log n)$, we can now complete our table! We sacrifice some time in building the data structure to speed up order queries. This is a common technique called **preprocessing**.

Data Structure	Operations $O(\cdot)$				
	Container	Static	Dynamic	Order	
	`build(X)`	`find(k)`	`insert(x)` `delete(k)`	`find_min()` `find_max()`	`find_prev(k)` `find_next(k)`
Array	n	n	n	n	n
Sorted Array	$n \log n$	$\log n$	n	1	$\log n$

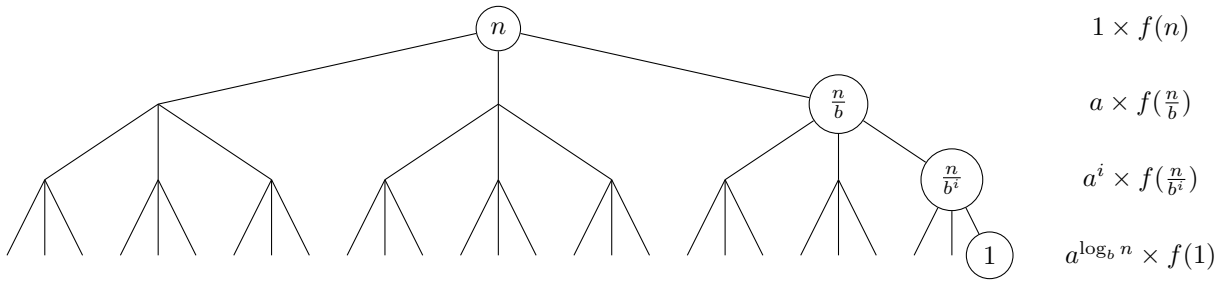
Recurrences

There are three primary methods for solving recurrences:

- **Substitution:** Guess a solution and substitute to show the recurrence holds.
- **Recursion Tree:** Draw a tree representing the recurrence and sum computation at nodes. This is a very general method, and is the one we've used in lecture so far.
- **Master Theorem:** A general formula to solve a large class of recurrences. It is useful, but can also be hard to remember.

Master Theorem

The **Master Theorem** provides a way to solve recurrence relations in which recursive calls decrease problem size by a constant factor. Given a recurrence relation of the form $T(n) = aT(n/b) + f(n)$ and $T(1) = \Theta(1)$, with branching factor $a \geq 1$, problem size reduction factor $b > 1$, and asymptotically non-negative function $f(n)$, the Master Theorem gives the solution to the recurrence by comparing $f(n)$ to $a^{\log_b n} = n^{\log_b a}$, the number of leaves at the bottom of the recursion tree. When $f(n)$ grows asymptotically faster than $n^{\log_b a}$, the work done at each level decreases geometrically so the work at the root dominates; alternatively, when $f(n)$ grows slower, the work done at each level increases geometrically and the work at the leaves dominates. When their growth rates are comparable, the work is evenly spread over the tree's $O(\log n)$ levels.



case	solution	conditions
1	$T(n) = \Theta(n^{\log_b a})$	$f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$
2	$T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$	$f(n) = \Theta(n^{\log_b a} \log^k n)$ for some constant $k \geq 0$
3	$T(n) = \Theta(f(n))$	$f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$ and $af(n/b) < cf(n)$ for some constant $0 < c < 1$

The Master Theorem takes on a simpler form when $f(n)$ is a polynomial, such that the recurrence has the from $T(n) = aT(n/b) + \Theta(n^c)$ for some constant $c \geq 0$.

case	solution	conditions	intuition
1	$T(n) = \Theta(n^{\log_b a})$	$c < \log_b a$	Work done at leaves dominates
2	$T(n) = \Theta(n^c \log n)$	$c = \log_b a$	Work balanced across the tree
3	$T(n) = \Theta(n^c)$	$c > \log_b a$	Work done at root dominates

This special case is straight-forward to prove by substitution (this can be done in recitation). To apply the Master Theorem (or this simpler special case), you should state which case applies, and show that your recurrence relation satisfies all conditions required by the relevant case. There are even stronger (more general) formulas[1] to solve recurrences, but we will not use them in this class.

Exercises

1. Write a recurrence for binary search and solve it.

   Solution: $T(n) = T(n/2) + O(1)$ so $T(n) = O(\log n)$ by case 2 of Master Theorem.

2. $T(n) = T(n-1) + O(1)$

   Solution: $T(n) = O(n)$, length n chain, $O(1)$ work per node.

3. $T(n) = T(n-1) + O(n)$

   Solution: $T(n) = O(n^2)$, length n chain, $O(k)$ work per node at height k.

4. $T(n) = 2T(n-1) + O(1)$

   Solution: $T(n) = O(2^n)$, height n binary tree, $O(1)$ work per node.

5. $T(n) = T(2n/3) + O(1)$

   Solution: $T(n) = O(\log n)$, length $\log_{3/2}(n)$ chain, $O(1)$ work per node.

6. $T(n) = 2T(n/2) + O(1)$

   Solution: $T(n) = O(n)$, height $\log_2 n$ binary tree, $O(1)$ work per node.

7. $T(n) = T(n/2) + O(n)$

   Solution: $T(n) = O(n)$, length $\log_2 n$ chain, $O(2^k)$ work per node at height k.

8. $T(n) = 2T(n/2) + O(n \log n)$

   Solution: $T(n) = O(n \log^2 n)$ (special case of Master Theorem does not apply because $n \log n$ is not polynomial), height $\log_2 n$ binary tree, $O(k \cdot 2^k)$ work per node at height k.

9. $T(n) = 4T(n/2) + O(n)$

   Solution: $T(n) = O(n^2)$, height $\log_2 n$ degree-4 tree, $O(2^k)$ work per node at height k.

[1] `http://en.wikipedia.org/wiki/Akra-Bazzi_method`