Recitation 2

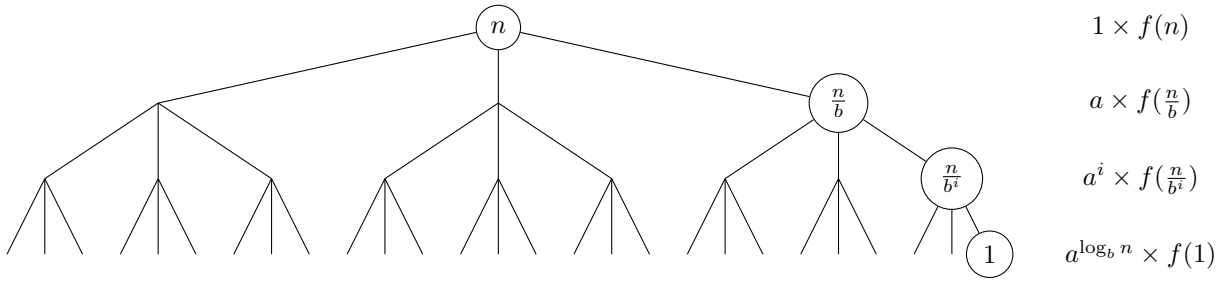
Recurrences

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.
- **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. 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. Note that case 2 of the simplified Master Theorem applies with $c = 0$ and $a = 1$.

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

   Solution: $T(n) = O(n)$, height $\log_2 n$ binary tree, $O(1)$ work per node. Case 1 of the simplified MT with $c = 0$ and $a = b = 2$.

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. Case 3 of the simplified MT with $a = c = 1$ and $b = 2$.

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

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

   Solution: $T(n) = O(n \log^2 n)$, height $\log_2 n$ binary tree, $O(k \cdot 2^k)$ work per node at height k. Case 2 of the general version of the MT with $a = b = 2$ and $k = 1$.

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. Case 1 of the special MT with $a = 4$, $b = 2$ and $c = 1$.

Sequence Interface

Sequences maintain a collection of items in an **extrinsic** order, where each item stored has a **rank** in the sequence, including a first item and a last item. By extrinsic, we mean that the first item is 'first', not because of what the item is, but because some external party put it there. Sequences are generalizations of stacks and queues, which support a subset of sequence operations.

Container	`build(X)`	given an iterable `X`, build sequence from items in `X`
	`len()`	return the number of stored items
Static	`iter_seq()`	return the stored items one-by-one in sequence order
	`get_at(i)`	return the i^{th} item
	`set_at(i, x)`	replace the i^{th} item with x
Dynamic	`insert_at(i, x)`	add x as the i^{th} item
	`delete_at(i)`	remove and return the i^{th} item
	`insert_first(x)`	add x as the first item
	`delete_first()`	remove and return the first item
	`insert_last(x)`	add x as the last item
	`delete_last()`	remove and return the last item

(Note that `insert_` / `delete_` operations change the rank of all items after the modified item.)

Set Interface

By contrast, Sets maintain a collection of items based on an **intrinsic** property involving what the items are, usually based on a unique **key**, `x.key`, associated with each item `x`. Sets are generalizations of dictionaries and other intrinsic query databases.

Container	`build(X)`	given an iterable `X`, build set from items in `X`
	`len()`	return the number of stored items
Static	`find(k)`	return the stored item with key `k`
Dynamic	`insert(x)`	add `x` to set (replace item with key `x.key` if one already exists)
	`delete(k)`	remove and return the stored item with key `k`
Order	`iter_ord()`	return the stored items one-by-one in key order
	`find_min()`	return the stored item with smallest key
	`find_max()`	return the stored item with largest key
	`find_next(k)`	return the stored item with smallest key larger than `k`
	`find_prev(k)`	return the stored item with largest key smaller than `k`

(Note that `find` operations return `None` if no qualifying item exists.)

Sequence Implementations

Here, we will discuss three data structures to implement the sequence interface.

Data Structure	Operation, Worst Case $O(\cdot)$				
	Container	Static	Dynamic		
	`build(X)`	`get_at(i)` `set_at(i,x)`	`insert_first(x)` `delete_first()`	`insert_last(x)` `delete_last()`	`insert_at(i, x)` `delete_at(i)`
Array	n	1	n	n	n
Linked List	n	n	1	n	n
Dynamic Array	n	1	n	$1_{(a)}$	n

Array Sequence

Computer memory is a finite resource. On modern computers many processes may share the same main memory store, so an operating system will assign a fixed chunk of memory addresses to each active process. The amount of memory assigned depends on the needs of the process and the availability of free memory. For example, when a computer program makes a request to store a variable, the program must tell the operating system how much memory (i.e. how many bits) will be required to store it. To fulfill the request, the operating system will find the available memory in the process's assigned memory address space and reserve it (i.e. allocate it) for that purpose until it is no longer needed. Memory management and allocation is a detail that is abstracted away by many high level languages including Python, but know that whenever you ask Python to store something, Python makes a request to the operating system behind-the-scenes, for a fixed amount of memory in which to store it.

Now suppose a computer program wants to store two arrays, each storing ten 64-bit words. The program makes separate requests for two chunks of memory (640 bits each), and the operating system fulfills the request by, for example, reserving the first ten words of the process's assigned address space to the first array A, and the second ten words of the address space to the second array B. Now suppose that as the computer program progresses, an eleventh word w needs to be added to array A. It would seem that there is no space near A to store the new word: the beginning of the process's assigned address space is to the left of A and array B is stored on the right. Then how can we add w to A? One solution could be to shift B right to make room for w, but tons of data may already be reserved next to B, which you would also have to move. Better would be to simply request eleven new words of memory, copy A to the beginning of the new memory allocation, store w at the end, and free the first ten words of the process's address space for future memory requests.

A fixed-length array is the data structure that is the underlying foundation of our model of computation (you can think of your computer's memory as a big fixed-length array that your operating system allocates from). Implementing a sequence using an array, where index i in the array corresponds to item i in the sequence allows `get_at` and `set_at` to be $O(1)$ time because of our

random access machine. However, when deleting or inserting into the sequence, we need to move items and resize the array, meaning these operations could take linear-time in the worst case. Below is a full Python implementation of an array sequence.

```
class Array_Seq:
    def __init__(self):                                 # O(1)
        self.A = []
        self.size = 0

    def __len__(self):      return self.size            # O(1)
    def __iter__(self):     yield from self.A           # O(n) iter_seq

    def build(self, X):                                 # O(n)
        self.A = [a for a in X]  # pretend this builds a static array
        self.size = len(self.A)

    def get_at(self, i):    return self.A[i]            # O(1)
    def set_at(self, i, x): self.A[i] = x               # O(1)

    def _copy_forward(self, i, n, B, j):                # O(n)
        # copies n values starting in position i to
        # B starting in position j
        for k in range(n):
            B[j + k] = self.A[i + k]

    def insert_at(self, i, x):                          # O(n)
        n = len(self)
        B = [None] * (n + 1)
        self._copy_forward(0, i, B, 0)
        B[i] = x
        self._copy_forward(i, n - i, B, i + 1)
        self.build(B)

    def delete_at(self, i):                             # O(n)
        n = len(self)
        B = [None] * (n - 1)
        self._copy_forward(0, i, B, 0)
        x = self.A[i]
        self._copy_forward(i + 1, n - i - 1, B, i)
        self.build(B)
        return x
                                                        # O(n)
    def insert_first(self, x):  self.insert_at(0, x)
    def delete_first(self):     return self.delete_at(0)
    def insert_last(self, x):   self.insert_at(len(self), x)
    def delete_last(self):      return self.delete_at(len(self) - 1)
```

Linked List Sequence

A **linked list** is a different type of data structure entirely. Instead of allocating a contiguous chunk of memory in which to store items, a linked list stores each item in a node, `node`, a constant-sized container with two properties: `node.item` storing the item, and `node.next` storing the memory address of the node containing the next item in the sequence.

```
class Linked_List_Node:
    def __init__(self, x):                          # O(1)
        self.item = x
        self.next = None

    def later_node(self, i):                        # O(i)
        if i == 0:  return self
        assert self.next
        return self.next.later_node(i - 1)
```

Such data structures are sometimes called **pointer-based** or **linked** and are much more flexible than array-based data structures because their constituent items can be stored anywhere in memory. A linked list stores the address of the node storing the first element of the list called the **head** of the list, along with the linked list's size, the number of items stored in the linked list. It is easy to add an item after another item in the list, simply by changing some addresses (i.e. relinking pointers). In particular, adding a new item at the front (head) of the list takes $O(1)$ time. However, the only way to find the i^{th} item in the sequence is to step through the items one-by-one, leading to worst-case linear time for `get_at` and `set_at` operations. Below is a Python implementation of a full linked list sequence.

```
class Linked_List_Seq:
    def __init__(self):                             # O(1)
        self.head = None
        self.size = 0

    def __len__(self):  return self.size            # O(1)

    def __iter__(self):                             # O(n) iter_seq
        node = self.head
        while node:
            yield node.item
            node = node.next

    def build(self, X):                             # O(n)
        for a in reversed(X):
            self.insert_first(a)

    def get_at(self, i):                            # O(i)
        node = self.head.later_node(i)
        return node.item

```

```
22      def set_at(self, i, x):                             # O(i)
23          node = self.head.later_node(i)
24          node.item = x
25
26      def insert_first(self, x):                          # O(1)
27          new_node = Linked_List_Node(x)
28          new_node.next = self.head
29          self.head = new_node
30          self.size += 1
31
32      def delete_first(self):                             # O(1)
33          x = self.head.item
34          self.head = self.head.next
35          self.size -= 1
36          return x
37
38      def insert_at(self, i, x):                          # O(i)
39          if i == 0:
40              self.insert_first(x)
41              return
42          new_node = Linked_List_Node(x)
43          node = self.head.later_node(i - 1)
44          new_node.next = node.next
45          node.next = new_node
46          self.size += 1
47
48      def delete_at(self, i):                             # O(i)
49          if i == 0:
50              return self.delete_first()
51          node = self.head.later_node(i - 1)
52          x = node.next.item
53          node.next = node.next.next
54          self.size -= 1
55          return x
56                                                          # O(n)
57      def insert_last(self, x):   self.insert_at(len(self), x)
58      def delete_last(self):      return self.delete_at(len(self) - 1)
```