Recitation 1

Algorithms

The study of algorithms searches for efficient procedures to solve problems. The goal of this class is to not only teach you how to solve problems, but to teach you to **communicate** to others that a solution to a problem is both **correct** and **efficient**.

- A **problem** is a binary relation connecting problem inputs to correct outputs.
- A (deterministic) **algorithm** is a procedure that maps inputs to single outputs.
- An algorithm **solves** a problem if for every problem input it returns a correct output.

While a problem input may have more than one correct output, an algorithm should only return one output for a given input (it is a function). As an example, consider the problem of finding another student in your recitation who shares the same birthday.

> **Problem:** Given the students in your recitation, return either the names of two students who share the same birthday and year, or state that no such pair exists.

This problem relates one input (your recitation) to one or more outputs comprising birthday-matching pairs of students or one negative result. A problem input is sometimes called an **instance** of the problem. One algorithm that solves this problem is the following.

> **Algorithm:** Maintain an initially empty record of student names and birthdays. Go around the room and ask each student their name and birthday. After interviewing each student, check to see whether their birthday already exists in the record. If yes, return the names of the two students found. Otherwise, add their name and birthday to the record. If after interviewing all students no satisfying pair is found, return that no matching pair exists.

Of course, our algorithm solves a much more general problem than the one proposed above. The same algorithm can search for a birthday-matching pair in **any** set of students, not just the students in your recitation. In this class, we try to solve problems which generalize to inputs that may be arbitrarily large. The birthday matching algorithm can be applied to a recitation of any size. But how can we determine whether the algorithm is correct and efficient?

Correctness

Any computer program you write will have finite size, while an input it acts on may be arbitrarily large. Thus every algorithm we discuss in this class will need to repeat commands in the algorithm via loops or recursion, and we will be able to prove correctness of the algorithm via **induction**. Let's prove that the birthday algorithm is correct.

> *Proof.* Induct on the first k students interviewed. Base case: for $k = 0$, there is no matching pair, and the algorithm returns that there is no matching pair. Alternatively, assume for induction that the algorithm returns correctly for the first k students. If the first k students contain a matching pair, than so does the first $k + 1$ students and the algorithm already returned a matching pair. Otherwise the first k students do not contain a matching pair, so if the $k+1$ students contain a match, the match includes student $k+1$, and the algorithm checks whether the student $k + 1$ has the same birthday as someone already processed. □

Efficiency

What makes a computer program efficient? One program is said to be more **efficient** than another if it can solve the same problem input using fewer resources. We expect that a larger input might take more time to solve than another input having smaller size. In addition, the resources used by a program, e.g. storage space or running time, will depend on both the algorithm used and the machine on which the algorithm is implemented. We expect that an algorithm implemented on a fast machine will run faster than the same algorithm on a slower machine, even for the same input. We would like to be able to compare algorithms, without having to worry about how fast our machine is. So in this class, we compare algorithms based on their **asymptotic performance** relative to problem input size, in order to ignore constant factor differences in hardware performance.

Asymptotic Notation

We can use **asymptotic notation** to ignore constants that do not change with the size of the problem input. $O(f(n))$ represents the set of functions with domain over the natural numbers satisfying the following property.

> **O Notation:** Non-negative function $g(n)$ is in $O(f(n))$ if and only if there exists a positive real number c and positive integer n_0 such that $g(n) \leq c \cdot f(n)$ for all $n \geq n_0$.

This definition upper bounds the **asymptotic growth** of a function for sufficiently large n, i.e., the bound on growth is true even if we were to scale or shift our function by a constant amount. By convention, it is more common for people to say that a function $g(n)$ **is** $O(f(n))$ or **equal to** $O(f(n))$, but what they really mean is set containment, i.e., $g(n) \in O(f(n))$. So since our problem's input size is cn for some constant c, we can forget about c and say the input size is $O(n)$ (**order** n). A similar notation can be used for lower bounds.

> **Ω Notation:** Non-negative function $g(n)$ is in $\Omega(f(n))$ if and only if there exists a positive real number c and positive integer n_0 such that $c \cdot f(n) \leq g(n)$ for all $n \geq n_0$.

When one function both asymptotically upper bounds **and** asymptotically lower bounds another function, we use Θ notation. When $g(n) = \Theta(f(n))$, we say that $f(n)$ represents a **tight bound** on $g(n)$.

> **Θ Notation:** Non-negative $g(n)$ is in $\Theta(f(n))$ if and only if $g(n) \in O(f(n)) \cap \Omega(f(n))$.

We often use shorthand to characterize the asymptotic growth (i.e., **asymptotic complexity**) of common functions, such as those shown in the table below[1]. Here we assume $c \in \Theta(1)$.

Shorthand	Constant	Logarithmic	Linear	Quadratic	Polynomial	Exponential[1]
$\Theta(f(n))$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^c)$	$2^{\Theta(n^c)}$

Linear time is often necessary to solve problems where the entire input must be read in order to solve the problem. However, if the input is already accessible in memory, many problems can be solved in sub-linear time. For example, the problem of finding a value in a sorted array (that has already been loaded into memory) can be solved in logarithmic time via binary search. We focus on polynomial time algorithms in this class, typically for small values of c. There's a big difference between logarithmic, linear, and exponential. If $n = 1000$, $\log n \approx 10^1, n \approx 10^3$, while $2^n \approx 10^{300}$. For comparison, the number of atoms in the universe is estimated around 10^{80}. It is common to use the variable 'n' to represent a parameter that is linear in the problem input size, though this is not always the case. For example, when talking about graph algorithms later in the term, a problem input will be a graph parameterized by vertex set V and edge set E, so a natural input size will be $\Theta(|V| + |E|)$. Alternatively, when talking about matrix algorithms, it is common to let n be the width of a square matrix, where a problem input will have size $\Theta(n^2)$, specifying each element of the $n \times n$ matrix.

[1]Note that exponential $2^{\Theta(n^c)}$ is a convenient abuse of notation meaning $\{2^p \mid p \in \Theta(n^c)\}$.

Model of Computation

In order to precisely calculate the resources used by an algorithm, we need to model how long a computer takes to perform basic operations. Specifying such a set of operations provides a **model of computation** upon which we can base our analysis. In this class, we will use the w-bit **Word-RAM** model of computation, which models a computer as a random access array of machine words called **memory**, together with a **processor** that can perform operations on the memory. A **machine word** is a sequence of w bits representing an integer from the set $\{0, \ldots, 2^w - 1\}$. A Word-RAM processor can perform basic binary operations on two machine words in constant time, including addition, subtraction, multiplication, integer division, modulo, bitwise operations, and binary comparisons. In addition, given a word a, the processor can read or write the word in memory located at address a in constant time. If a machine word contains only w bits, the processor will only be able to read and write from at most 2^w addresses in memory[2]. So when solving a problem on an input stored in n machine words, we will always assume our Word-RAM has a word size of at least $w > \log_2 n$ bits, or else the machine would not be able to access all of the input in memory. To put this limitation in perspective, a Word-RAM model of a byte-addressable 64-bit machine allows inputs up to $\sim 10^{10}$ GB in size.

Data Structure

The running time of our birthday matching algorithm depends on how we store the record of names and birthdays. A **data structure** is a way to store a non-constant amount of data, supporting a set of operations to interact with that data. The set of operations supported by a data structure is called an **interface**. Many data structures might support the same interface, but could provide different performance for each operation. Many problems can be solved trivially by storing data in an appropriate choice of data structure. For our example, we will use the most primitive data structure native to the Word-RAM: the **static array**. A static array is simply a contiguous sequence of words reserved in memory, supporting a static sequence interface:

- `StaticArray(n)`: allocate a new static array of size n initialized to 0 in $\Theta(n)$ time
- `StaticArray.get_at(i)`: return the word stored at array index i in $\Theta(1)$ time
- `StaticArray.set_at(i, x)`: write the word x to array index i in $\Theta(1)$ time

The `get_at(i)` and `set_at(i, x)` operations run in constant time because each item in the array has the same size: one machine word. To store larger objects at an array index, we can interpret the machine word at the index as a memory address to a larger piece of memory. A Python `tuple` is like a static array without `set_at(i, x)`. A Python `list` implements a **dynamic array** (see L02).

[2]For example, on a typical 32-bit machine, each byte (8-bits) is addressable (for historical reasons), so the size of the machine's random-access memory (RAM) is limited to (8-bits)$\times(2^{32}) \approx 4$ GB.

```
1  class StaticArray:
2      def __init__(self, n):
3          self.data = [None] * n
4      def get_at(self, i):
5          if not (0 <= i < len(self.data)): raise IndexError
6          return self.data[i]
7      def set_at(self, i, x):
8          if not (0 <= i < len(self.data)): raise IndexError
9          self.data[i] = x
10
11 def birthday_match(students):
12     '''
13     Find a pair of students with the same birthday
14     Input:  tuple of student (name, bday) tuples
15     Output: tuple of student names or None
16     '''
17     n = len(students)                              # O(1)
18     record = StaticArray(n)                        # O(n)
19     for k in range(n):                             # n
20         (name1, bday1) = students[k]               # O(1)
21         for i in range(k):                         # k     Check if in record
22             (name2, bday2) = record.get_at(i)      # O(1)
23             if bday1 == bday2:                     # O(1)
24                 return (name1, name2)              # O(1)
25         record.set_at(k, (name1, bday1))           # O(1)
26     return None                                    # O(1)
```

Running Time Analysis

Now let's analyze the running time of our birthday matching algorithm on a recitation containing n students. We will assume that each name and birthday fits into a constant number of machine words so that a single student's information can be collected and manipulated in constant time[3]. We step through the algorithm line by line. All the lines take constant time except for lines 8, 9, and 11. Line 8 takes $\Theta(n)$ time to initialize the static array record; line 9 loops at most n times; and line 11 loops through the k items existing in the record. Thus the running time for this algorithm is at most:

$$O(n) + \sum_{k=0}^{n-1}(O(1) + k \cdot O(1)) = O(n^2)$$

This is quadratic in n, which is polynomial! Is this efficient? No! We can do better by using a different data structure for our record. We will spend the first half of this class studying elementary data structures, where each data structure will be tailored to support a different set of operations efficiently.

[3]This is a reasonable restriction, which allows names and birthdays to contain $O(w)$ characters from a constant sized alphabet. Since $w > \log_2 n$, this restriction still allows each student's information to be distinct.

Asymptotics Exercises

1. Have students generate 10 functions and order them based on asymptotic growth.

2. Find a simple, tight asymptotic bound for $\binom{n}{6006}$.

   Solution: Definition yields $n(n-1)\dots(n-6005)$ in the numerator (a degree 6006 polynomial) and $6006!$ in the denominator (constant with respect to n). So $\binom{n}{6006} = \Theta(n^{6006})$.

3. Find a simple, tight asymptotic bound for $\log_{6006}\left(\left(\log\left(n^{\sqrt{n}}\right)\right)^2\right)$.

   Solution: Recall exponent and logarithm rules: $\log ab = \log a + \log b$, $\log\left(a^b\right) = b\log a$, and $\log_a b = \log b / \log a$.

$$\begin{aligned}\log_{6006}\left(\left(\log\left(n^{\sqrt{n}}\right)\right)^2\right) &= \frac{2}{\log 6006}\log\left(\sqrt{n}\log n\right)\\ &= \Theta(\log n^{1/2} + \log\log n) = \Theta(\log n)\end{aligned}$$

4. Show that $2^{n+1} \in \Theta(2^n)$, but that $2^{2^{n+1}} \notin O(2^{2^n})$.

   Solution: In the first case, $2^{n+1} = 2\cdot 2^n$, which is a constant factor larger than 2^n. In the second case, $2^{2^{n+1}} = \left(2^{2^n}\right)^2$, which is definitely more than a constant factor larger than 2^{2^n}.

5. Show that $(\log n)^a = O(n^b)$ for all positive constants a and b.

   Solution: It's enough to show $n^b/(\log n)^a$ limits to ∞ as $n \to \infty$, and this is equivalent to arguing that the **log** of this expression approaches ∞:

$$\lim_{n\to\infty}\log\left(\frac{n^b}{(\log n)^a}\right) = \lim_{n\to\infty}(b\log n - a\log\log n) = \lim_{x\to\infty}(bx - a\log x) = \infty,$$

   as desired.

   Note: for the same reasons, $n^a = O(c^n)$ for any $c > 1$.

6. Show that $(\log n)^{\log n} = \Omega(n)$.

   Solution: Note that $m^m = \Omega(2^m)$, so setting $n = 2^m$ completes the proof.

7. Show that $(6n)! \notin \Theta(n!)$, but that $\log((6n)!) \in \Theta(\log(n!))$.

   Solution: We invoke Sterling's approximation,

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1 + \Theta\left(\frac{1}{n}\right)\right).$$

   Substituting in $6n$ gives an expression that is at least 6^{6n} larger than the original. But taking the logarithm of Sterling's gives $\log(n!) = \Theta(n\log n)$, and substituting in $6n$ yields only constant additional factors.

6.006 Introduction to Algorithms

Spring 2020

Recitation 2

Sequence Interface (L02, L07)

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 (L03-L08)

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. In Problem Set 1, you will extend both Linked Lists and Dynamic arrays to make both first and last dynamic operations $O(1)$ time for each. Notice that none of these data structures support dynamic operations at arbitrary index in sub-linear time. We will learn how to improve this operation in Lecture 7.

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, A, j):                # O(n)
        for k in range(n):
            A[j + k] = self.A[i + k]

    def _copy_backward(self, i, n, A, j):               # O(n)
        for k in range(n - 1, -1, -1):
            A[j + k] = self.A[i + k]

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

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

Dynamic Array Sequence

The array's dynamic sequence operations require linear time with respect to the length of array A. Is there another way to add elements to an array without paying a linear overhead transfer cost each time you add an element? One straight-forward way to support faster insertion would be to over-allocate additional space when you request space for the array. Then, inserting an item would be as simple as copying over the new value into the next empty slot. This compromise trades a little extra space in exchange for constant time insertion. Sounds like a good deal, but any additional allocation will be bounded; eventually repeated insertions will fill the additional space, and the array will again need to be reallocated and copied over. Further, any additional space you reserve will mean less space is available for other parts of your program.

Then how does Python support appending to the end of a length n Python List in worst-case $O(1)$ time? The answer is simple: **it doesn't**. Sometimes appending to the end of a Python List requires $O(n)$ time to transfer the array to a larger allocation in memory, so **sometimes** appending to a Python List takes linear time. However, allocating additional space in the right way can guarantee that any sequence of n insertions only takes at most $O(n)$ time (i.e. such linear time transfer operations do not occur often), so insertion will take $O(1)$ time per insertion **on average**. We call this asymptotic running time **amortized constant time**, because the cost of the operation is amortized (distributed) across many applications of the operation.

To achieve an amortized constant running time for insertion into an array, our strategy will be to allocate extra space in proportion to the size of the array being stored. Allocating $O(n)$ additional space ensures that a linear number of insertions must occur before an insertion will overflow the allocation. A typical implementation of a dynamic array will allocate double the amount of space needed to store the current array, sometimes referred to as **table doubling**. However, allocating any constant fraction of additional space will achieve the amortized bound. Python Lists allocate additional space according to the following formula (from the Python source code written in C):

```
new_allocated = (newsize >> 3) + (newsize < 9 ? 3 : 6);
```

Here, the additional allocation is modest, roughly one eighth of the size of the array being appended (bit shifting the size to the right by 3 is equivalent to floored division by 8). But the additional allocation is still linear in the size of the array, so on average, $n/8$ insertions will be performed for every linear time allocation of the array, i.e. amortized constant time.

What if we also want to remove items from the end of the array? Popping the last item can occur in constant time, simply by decrementing a stored length of the array (which Python does). However, if a large number of items are removed from a large list, the unused additional allocation could occupy a significant amount of wasted memory that will not available for other purposes. When the length of the array becomes sufficiently small, we can transfer the contents of the array to a new, smaller memory allocation so that the larger memory allocation can be freed. How big should this new allocation be? If we allocate the size of the array without any additional allocation, an immediate insertion could trigger another allocation. To achieve constant amortized running time for any sequence of n appends or pops, we need to make sure there remains a linear fraction of unused allocated space when we rebuild to a smaller array, which guarantees that at least $\Omega(n)$ sequential dynamic operations must occur before the next time we need to reallocate memory.

Below is a Python implementation of a dynamic array sequence, including operations `insert_last` (i.e., Python list `append`) and `delete_last` (i.e., Python list `pop`), using table doubling proportions. When attempting to append past the end of the allocation, the contents of the array are transferred to an allocation that is twice as large. When removing down to one fourth of the allocation, the contents of the array are transferred to an allocation that is half as large. Of course Python Lists already support dynamic operations using these techniques; this code is provided to help you understand how **amortized constant** `append` and `pop` could be implemented.

```
class Dynamic_Array_Seq(Array_Seq):
    def __init__(self, r = 2):                          # O(1)
        super().__init__()
        self.size = 0
        self.r = r
        self._compute_bounds()
        self._resize(0)

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

    def __iter__(self):                                 # O(n)
        for i in range(len(self)): yield self.A[i]

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

    def _compute_bounds(self):                          # O(1)
        self.upper = len(self.A)
        self.lower = len(self.A) // (self.r * self.r)

    def _resize(self, n):                               # O(1) or O(n)
        if (self.lower < n < self.upper): return
        m = max(n, 1) * self.r
        A = [None] * m
        self._copy_forward(0, self.size, A, 0)
        self.A = A
        self._compute_bounds()

    def insert_last(self, x):                           # O(1)a
        self._resize(self.size + 1)
        self.A[self.size] = x
        self.size += 1

    def delete_last(self):                              # O(1)a
        self.A[self.size - 1] = None
        self.size -= 1
        self._resize(self.size)

    def insert_at(self, i, x):                          # O(n)
        self.insert_last(None)
        self._copy_backward(i, self.size - (i + 1), self.A, i + 1)
        self.A[i] = x

    def delete_at(self, i):                             # O(n)
        x = self.A[i]
        self._copy_forward(i + 1, self.size - (i + 1), self.A, i)
        self.delete_last()
        return x
                                                        # O(n)
    def insert_first(self, x):  self.insert_at(0, x)
    def delete_first(self):     return self.delete_at(0)
```

Exercises:

- Suppose the next pointer of the last node of a linked list points to an earlier node in the list, creating a cycle. Given a pointer to the head of the list (without knowing its size), describe a linear-time algorithm to find the number of nodes in the cycle. Can you do this while using only constant additional space outside of the original linked list?

  Solution: Begin with two pointers pointing at the head of the linked list: one slow pointer and one fast pointer. The pointers take turns traversing the nodes of the linked list, starting with the fast pointer. On the slow pointer's turn, the slow pointer simply moves to the next node in the list; while on the fast pointer's turn, the fast pointer initially moves to the next node, but then moves on to the next node's next node before ending its turn. Every time the fast pointer visits a node, it checks to see whether it's the same node that the slow pointer is pointing to. If they are the same, then the fast pointer must have made a full loop around the cycle, to meet the slow pointer at some node v on the cycle. Now to find the length of the cycle, simply have the fast pointer continue traversing the list until returning back to v, counting the number of nodes visited along the way.

  To see that this algorithm runs in linear time, clearly the last step of traversing the cycle takes at most linear time, as v is the only node visited twice while traversing the cycle. Further, we claim the slow pointer makes at most one move per node. Suppose for contradiction the slow pointer moves twice away from some node u before being at the same node as the fast pointer, meaning that u is on the cycle. In the same time the slow pointer takes to traverse the cycle from u back to u, the fast pointer will have traveled around the cycle twice, meaning that both pointers must have existed at the same node prior to the slow pointer leaving u, a contradiction.

- Given a data structure implementing the Sequence interface, show how to use it to implement the Set interface. (Your implementation does not need to be efficient.)

  Solution:

```
def Set_from_Seq(seq):
    class set_from_seq:
        def __init__(self):   self.S = seq()
        def __len__(self):    return len(self.S)
        def __iter__(self):   yield from self.S

        def build(self, A):
            self.S.build(A)

        def insert(self, x):
            for i in range(len(self.S)):
                if self.S.get_at(i).key == x.key:
                    self.S.set_at(i, x)
                    return
            self.S.insert_last(x)

```

```
17          def delete(self, k):
18              for i in range(len(self.S)):
19                  if self.S.get_at(i).key == k:
20                      return self.S.delete_at(i)
21
22          def find(self, k):
23              for x in self:
24                  if x.key == k:  return x
25              return None
26
27          def find_min(self):
28              out = None
29              for x in self:
30                  if (out is None) or (x.key < out.key):
31                      out = x
32              return out
33
34          def find_max(self):
35              out = None
36              for x in self:
37                  if (out is None) or (x.key > out.key):
38                      out = x
39              return out
40
41          def find_next(self, k):
42              out = None
43              for x in self:
44                  if x.key > k:
45                      if (out is None) or (x.key < out.key):
46                          out = x
47              return out
48
49          def find_prev(self, k):
50              out = None
51              for x in self:
52                  if x.key < k:
53                      if (out is None) or (x.key > out.key):
54                          out = x
55              return out
56
57          def iter_ord(self):
58              x = self.find_min()
59              while x:
60                  yield x
61                  x = self.find_next(x.key)
62
63      return set_from_seq
```

6.006 Introduction to Algorithms

Spring 2020

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	$\log n$

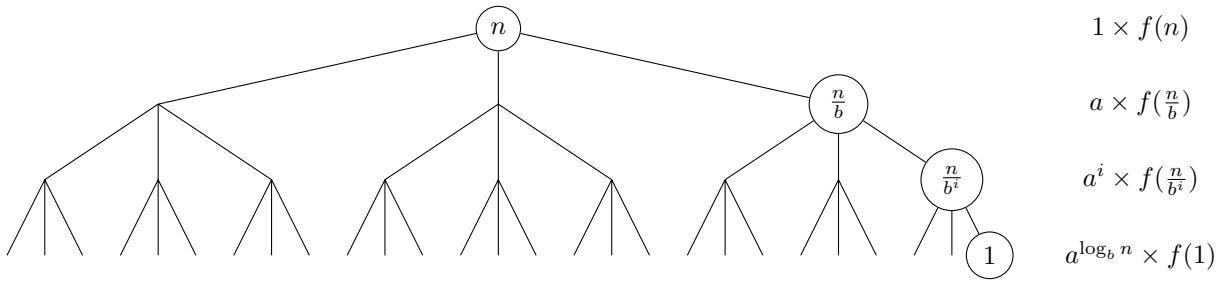
Recurrences

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

6.006 Introduction to Algorithms

Spring 2020

Recitation 4

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	$\log n$

We've learned how to implement a set interface using a sorted array, where query operations are efficient but whose dynamic operations are lacking. Recalling that $\Theta(\log n)$ growth is much closer to $\Theta(1)$ than $\Theta(n)$, a sorted array provides really good performance! But one of the most common operations you will do in programming is to search for something you're storing, i.e., `find(k)`. Is it possible to `find` faster than $\Theta(\log n)$? It turns out that if the only thing we can do to items is to compare their relative order, then the answer is **no**!

Comparison Model

The comparison model of computation acts on a set of **comparable** objects. The objects can be thought of as black boxes, supporting only a set of binary boolean operations called **comparisons** (namely $<, \leq, >, \geq, =$, and $\neq$). Each operation takes as input two objects and outputs a Boolean value, either **True** or **False**, depending on the relative ordering of the elements. A search algorithm operating on a set of n items will return a stored item with a key equal to the input key, or return no item if no such item exists. In this section, we assume that each item has a unique key.

If binary comparisons are the only way to distinguish between stored items and a search key, a deterministic comparison search algorithm can be thought of as a fixed binary **decision tree** representing all possible executions of the algorithm, where each node represents a comparison performed by the algorithm. During execution, the algorithm walks down the tree along a path from the root. For any given input, a comparison sorting algorithm will make some comparison first, the comparison at the root of the tree. Depending on the outcome of this comparison, the computation will then proceed with a comparison at one of its two children. The algorithm repeatedly makes comparisons until a leaf is reached, at which point the algorithm terminates, returning an output to the algorithm. There must be a leaf for each possible output to the algorithm. For search, there are $n + 1$ possible outputs, the n items and the result where no item is found, so there must be at least $n + 1$ leaves in the decision tree. Then the **worst-case number of comparisons** that must be made by any comparison search algorithm will be **the height of the algorithm's decision tree**, i.e., the length of any longest root to leaf path.

Exercise: Prove that the smallest height for any tree on n nodes is $\lceil \lg(n+1) \rceil - 1 = \Omega(\log n)$.

Solution: We show that the maximum number of nodes in any binary tree with height h is $n \leq T(h) = 2^{h+1} - 1$, so $h \geq (\lg(n+1)) - 1$. Proof by induction on h. The only tree of height zero has one node, so $T(0) = 1$, a base case satisfying the claim. The maximum number of nodes in a height-h tree must also have the maximum number of nodes in its two subtrees, so $T(h) = 2T(h-1) + 1$. Substituting $T(h)$ yields $2^{h+1} - 1 = 2(2^h - 1) + 1$, proving the claim. □

A tree with $n+1$ leaves has more than n nodes, so its height is at least $\Omega(\log n)$. Thus the minimum number of comparisons needed to distinguish between the n items is at least $\Omega(\log n)$, and the worst-case running time of any deterministic comparison search algorithm is at least $\Omega(\log n)$! So sorted arrays and balanced BSTs are able to support `find(k)` asymptotically **optimally**, in a comparison model of computation.

Comparisons are very limiting because each operation performed can lead to at most constant branching factor in the decision tree. It doesn't matter that comparisons have branching factor two; any fixed constant branching factor will lead to a decision tree with at least $\Omega(\log n)$ height. If we were not limited to comparisons, it opens up the possibility of faster-than-$O(\log n)$ search. More specifically, if we can use an operation that allows for asymptotically larger than constant $\omega(1)$ branching factor, then our decision tree could be shallower, leading to a faster algorithm.

Direct Access Arrays

Most operations within a computer only allow for constant logical branching, like if statements in your code. However, one operation on your computer allows for non-constant branching factor: specifically the ability to randomly access any memory address in constant time. This special operation allows an algorithm's decision tree to branch with large branching factor, as large as there is space in your computer. To exploit this operation, we define a data structure called a **direct access array**, which is a normal static array that associates a semantic meaning with each array index location: specifically that any item x with key k will be stored at array index k. This statement only makes sense when item keys are integers. Fortunately, in a computer, any thing in memory can be associated with an integer—for example, its value as a sequence of bits or its address in memory—so from now on we will only consider integer keys.

Now suppose we want to store a set of n items, each associated with a **unique** integer key in the **bounded range** from 0 to some large number $u-1$. We can store the items in a length u direct access array, where each array slot i contains an item associated with integer key i, if it exists. To find an item having integer key i, a search algorithm can simply look in array slot i to respond to the search query in **worst-case constant time**! However, order operations on this data structure will be very slow: we have no guarantee on where the first, last, or next element is in the direct access array, so we may have to spend u time for order operations.

Worst-case constant time search comes at the cost of storage space: a direct access array must have a slot available for every possible key in range. When u is very large compared to the number of items being stored, storing a direct access array can be wasteful, or even impossible on modern machines. For example, suppose you wanted to support the set `find(k)` operation on ten-letter names using a direct access array. The space of possible names would be $u \approx 26^{10} \approx 9.5 \times 10^{13}$; even storing a bit array of that length would require 17.6 Terabytes of storage space. How can we overcome this obstacle? The answer is hashing!

```
class DirectAccessArray:
    def __init__(self, u):  self.A = [None] * u   # O(u)
    def find(self, k):      return self.A[k]      # O(1)
    def insert(self, x):    self.A[x.key] = x     # O(1)
    def delete(self, k):    self.A[k] = None      # O(1)
    def find_next(self, k):
        for i in range(k, len(self.A)):           # O(u)
            if A[i] is not None:
                return A[i]
    def find_max(self):
        for i in range(len(self.A) - 1, -1, -1):  # O(u)
            if A[i] is not None:
                return A[i]
    def delete_max(self):
        for i in range(len(self.A) - 1, -1, -1):  # O(u)
            x = A[i]
            if x is not None:
                A[i] = None
                return x
```

Hashing

Is it possible to get the performance benefits of a direct access array while using only linear $O(n)$ space when $n \ll u$? A possible solution could be to store the items in a smaller **dynamic** direct access array, with $m = O(n)$ slots instead of u, which grows and shrinks like a dynamic array depending on the number of items stored. But to make this work, we need a function that maps item keys to different slots of the direct access array, $h(k) : \{0, \ldots, u-1\} \to \{0, \ldots, m-1\}$. We call such a function a **hash function** or a **hash map**, while the smaller direct access array is called a **hash table**, and $h(k)$ is the **hash** of integer key k. If the hash function happens to be injective over the n keys you are storing, i.e. no two keys map to the same direct access array index, then we will be able to support worst-case constant time search, as the hash table simply acts as a direct access array over the smaller domain m.

Unfortunately, if the space of possible keys is larger than the number of array indices, i.e. $m < u$, then any hash function mapping u possible keys to m indices must map multiple keys to the same array index, by the pigeonhole principle. If two items associated with keys k_1 and k_2 hash to the same index, i.e. $h(k_1) = h(k_2)$, we say that the hashes of k_1 and k_2 **collide**. If you don't know in advance what keys will be stored, it is extremely unlikely that your choice of hash function will avoid collisions entirely[1]. If the smaller direct access array hash table can only store one item at each index, when collisions occur, where do we store the colliding items? Either we store collisions somewhere else in the same direct access array, or we store collisions somewhere else. The first strategy is called **open addressing**, which is the way most hash tables are actually implemented, but such schemes can be difficult to analyze. We will adopt the second strategy called **chaining**.

Chaining

Chaining is a collision resolution strategy where colliding keys are stored separately from the original hash table. Each hash table index holds a pointer to a **chain**, a separate data structure that supports the dynamic set interface, specifically operations `find(k)`, `insert(x)`, and `delete(k)`. It is common to implement a chain using a linked list or dynamic array, but any implementation will do, as long as each operation takes no more than linear time. Then to `insert` item x into the hash table, simply insert x into the chain at index $h(x.key)$; and to `find` or `delete` a key k from the hash table, simply find or delete k from the chain at index $h(k)$.

Ideally, we want chains to be small, because if our chains only hold a constant number of items, the dynamic set operations will run in constant time. But suppose we are unlucky in our choice of hash function, and all the keys we want to store has all of them to the same index location, into the same chain. Then the chain will have linear size, meaning the dynamic set operations could take linear time. A good hash function will try to minimize the frequency of such collisions in order to minimize the maximum size of any chain. So what's a good hash function?

Hash Functions

Division Method (bad): The simplest mapping from an integer key domain of size u to a smaller one of size m is simply to divide the key by m and take the remainder: $h(k) = (k \bmod m)$, or in Python, `k % m`. If the keys you are storing are uniformly distributed over the domain, the division method will distribute items roughly evenly among hashed indices, so we expect chains to have small size providing good performance. However, if all items happen to have keys with the same remainder when divided by m, then this hash function will be terrible. Ideally, the performance of our data structure would be **independent** of the keys we choose to store.

[1] If you know all of the keys you will want to store in advance, it is possible to design a hashing scheme that will always avoid collisions between those keys. This idea, called **perfect hashing**, follows from the Birthday Paradox.

Universal Hashing (good): For a large enough key domain u, every hash function will be bad for some set of n inputs[2]. However, we can achieve good **expected** bounds on hash table performance by choosing our hash function **randomly** from a large family of hash functions. Here the expectation is over our choice of hash function, which is independent of the input. **This is not expectation over the domain of possible input keys.** One family of hash functions that performs well is:

$$\mathcal{H}(m,p) = \Big\{ h_{ab}(k) = (((ak+b) \bmod p) \bmod m) \quad a,b \in \{0,\ldots,p-1\} \text{ and } a \neq 0 \Big\},$$

where p is a prime that is larger than the key domain u. A single hash function from this family is specified by choosing concrete values for a and b. This family of hash functions is **universal**[3]: for any two keys, the probability that their hashes will collide when hashed using a hash function chosen uniformly at random from the universal family, is no greater than $1/m$, i.e.

$$\Pr_{h\in\mathcal{H}}\{h(k_i) = h(k_j)\} \leq 1/m, \quad \forall k_i \neq k_j \in \{0,\ldots,u-1\}.$$

If we know that a family of hash functions is universal, then we can upper bound the expected size of any chain, **in expectation over our choice of hash function** from the family. Let X_{ij} be the indicator random variable representing the value 1 if keys k_i and k_j collide for a chosen hash function, and 0 otherwise. Then the random variable representing the number of items hashed to index $h(k_i)$ will be the sum $X_i = \sum_j X_{ij}$ over all keys k_j from the set of n keys $\{k_0,\ldots,k_{n-1}\}$ stored in the hash table. Then the expected number of keys hashed to the chain at index $h(k_i)$ is:

$$\begin{aligned}
\mathop{\mathbb{E}}_{h\in\mathcal{H}}\{X_i\} = \mathop{\mathbb{E}}_{h\in\mathcal{H}}\left\{\sum_j X_{ij}\right\} = \sum_j \mathop{\mathbb{E}}_{h\in\mathcal{H}}\{X_{ij}\} &= 1 + \sum_{j\neq i} \mathop{\mathbb{E}}_{h\in\mathcal{H}}\{X_{ij}\} \\
&= 1 + \sum_{j\neq i} (1) \Pr_{h\in\mathcal{H}}\{h(k_i) = h(k_j)\} + (0) \Pr_{h\in\mathcal{H}}\{h(k_i) \neq h(k_j)\} \\
&\leq 1 + \sum_{j\neq i} 1/m = 1 + (n-1)/m.
\end{aligned}$$

If the size of the hash table is at least linear in the number of items stored, i.e. $m = \Omega(n)$, then the expected size of any chain will be $1 + (n-1)/\Omega(n) = O(1)$, a constant! Thus a hash table where collisions are resolved using chaining, implemented using a randomly chosen hash function from a universal family, will perform dynamic set operations in **expected constant time**, where the expectation is taken over the random choice of hash function, independent from the input keys! Note that in order to maintain $m = O(n)$, insertion and deletion operations may require you to rebuild the direct access array to a different size, choose a new hash function, and reinsert all the items back into the hash table. This can be done in the same way as in dynamic arrays, leading to **amortized bounds for dynamic operations**.

[2] If $u > nm$, every hash function from u to m maps some n keys to the same hash, by the pigeonhole principle.

[3] The proof that this family is universal is beyond the scope of 6.006, though it is usually derived in 6.046.

```
class Hash_Table_Set:
    def __init__(self, r = 200):            # O(1)
        self.chain_set = Set_from_Seq(Linked_List_Seq)
        self.A = []
        self.size = 0
        self.r = r                          # 100/self.r = fill ratio
        self.p = 2**31 - 1
        self.a = randint(1, self.p - 1)
        self._compute_bounds()
        self._resize(0)

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

    def build(self, X):                     # O(n)e
        for x in X: self.insert(x)

    def _hash(self, k, m):                  # O(1)
        return ((self.a * k) % self.p) % m

    def _compute_bounds(self):              # O(1)
        self.upper = len(self.A)
        self.lower = len(self.A) * 100*100 // (self.r*self.r)

    def _resize(self, n):                   # O(n)
        if (self.lower >= n) or (n >= self.upper):
            f = self.r // 100
            if self.r % 100:     f += 1
            # f = ceil(r / 100)
            m = max(n, 1) * f
            A = [self.chain_set() for _ in range(m)]
            for x in self:
                h = self._hash(x.key, m)
                A[h].insert(x)
            self.A = A
            self._compute_bounds()

    def find(self, k):                      # O(1)e
        h = self._hash(k, len(self.A))
        return self.A[h].find(k)

    def insert(self, x):                    # O(1)ae
        self._resize(self.size + 1)
        h = self._hash(x.key, len(self.A))
        added = self.A[h].insert(x)
        if added:   self.size += 1
        return added


```

```
52      def delete(self, k):                    # O(1)ae
53          assert len(self) > 0
54          h = self._hash(k, len(self.A))
55          x = self.A[h].delete(k)
56          self.size -= 1
57          self._resize(self.size)
58          return x
59
60      def find_min(self):                     # O(n)
61          out = None
62          for x in self:
63              if (out is None) or (x.key < out.key):
64                  out = x
65          return out
66
67      def find_max(self):                     # O(n)
68          out = None
69          for x in self:
70              if (out is None) or (x.key > out.key):
71                  out = x
72          return out
73
74      def find_next(self, k):                 # O(n)
75          out = None
76          for x in self:
77              if x.key > k:
78                  if (out is None) or (x.key < out.key):
79                      out = x
80          return out
81
82      def find_prev(self, k):                 # O(n)
83          out = None
84          for x in self:
85              if x.key < k:
86                  if (out is None) or (x.key > out.key):
87                      out = x
88          return out
89
90      def iter_order(self):                   # O(n^2)
91          x = self.find_min()
92          while x:
93              yield x
94              x = self.find_next(x.key)
```

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	$\log n$
Direct Access Array	u	1	1	u	u
Hash Table	$n_{(e)}$	$1_{(e)}$	$1_{(a)(e)}$	n	n

Exercise

Given an unsorted array $A = [a_0, \ldots, a_{n-1}]$ containing n positive integers, the DUPLICATES problem asks whether two integers in the array have the same value.

1) Describe a brute-force **worst-case** $O(n^2)$-time algorithm to solve DUPLICATES.
Solution: Loop through all $\binom{n}{2} = O(n^2)$ pairs of integers from the array and check if they are equal in $O(1)$ time.

2) Describe a **worst-case** $O(n \log n)$-time algorithm to solve DUPLICATES.
Solution: Sort the array in worst-case $O(n \log n)$ time (e.g. using merge sort), and then scan through the sorted array, returning if any of the $O(n)$ adjacent pairs have the same value.

3) Describe an **expected** $O(n)$-time algorithm to solve DUPLICATES.
Solution: Hash each of the n integers into a hash table (implemented using chaining and a hash function chosen randomly from a universal hash family[4]), with insertion taking expected $O(1)$ time. When inserting an integer into a chain, check it against the other integers already in the chain, and return if another integer in the chain has the same value. Since each chain has expected $O(1)$ size, this check takes expected $O(1)$ time, so the algorithm runs in expected $O(n)$ time.

4) If $k < n$ and $a_i \leq k$ for all $a_i \in A$, describe a **worst-case** $O(1)$-time algorithm to solve DUPLICATES.
Solution: If $k < n$, a duplicate always exists, by the pigeonhole principle.

5) If $n \leq k$ and $a_i \leq k$ for all $a_i \in A$, describe a **worst-case** $O(k)$-time algorithm to solve DUPLICATES.
Solution: Insert each of the n integers into a direct access array of length k, which will take worst-case $O(k)$ time to instantiate, and worst-case $O(1)$ time per insert operation. If an integer already exists at an array index when trying to insert, then return that a duplicate exists.

[4]In 6.006, you do not have to specify these details when answering problems. You may simply quote that hash tables can achieve the expected/amortized bounds for operations described in class.

6.006 Introduction to Algorithms

Spring 2020

Recitation 5

Comparison Sorting

Last time we discussed a lower bound on search in a comparison model. We can use a similar analysis to lower bound the worst-case running time of any sorting algorithm that only uses comparisons. There are $n!$ possible outputs to a sorting algorithm: the $n!$ permutations of the items. Then the decision tree for any deterministic sorting algorithm that uses only comparisons must have at least $n!$ leaves, and thus (by the same analysis as the search decision tree) must have height that is at least $\Omega(\log(n!)) = \Omega(n \log n)$ height[1], leading to a running time of at least $\Omega(n \log n)$.

Direct Access Array Sort

Just as with search, if we are **not** limited to comparison operations, it is possible to beat the $\Omega(n \log n)$ bound. If the items to be sorted have **unique** keys from a bounded positive range $\{0, \ldots, u-1\}$ (so $n \leq u$), we can sort them simply by using a direct access array. Construct a direct access array with size u and insert each item x into index $x.key$. Then simply read through the direct access array from left to right returning items as they are found. Inserting takes time $\Theta(n)$ time while initializing and scanning the direct access array takes $\Theta(u)$ time, so this sorting algorithm runs in $\Theta(n + u)$ time. If $u = O(n)$, then this algorithm is linear! Unfortunately, this sorting algorithm has two drawbacks: first, it cannot handle duplicate keys, and second, it cannot handle large key ranges.

```
def direct_access_sort(A):
    "Sort A assuming items have distinct non-negative keys"
    u = 1 + max([x.key for x in A])          # O(n) find maximum key
    D = [None] * u                           # O(u) direct access array
    for x in A:                              # O(n) insert items
        D[x.key] = x
    i = 0
    for key in range(u):                     # O(u) read out items in order
        if D[key] is not None:
            A[i] = D[key]
            i += 1
```

[1] We can prove this directly via Stirling's approximation, $n! \approx \sqrt{2\pi n}(n/e)^n$, or by observing that $n! > (n/2)^{n/2}$.

Counting Sort

To solve the first problem, we simply link a chain to each direct access array index, just like in hashing. When multiple items have the same key, we store them both in the chain associated with their key. Later, it will be important that this algorithm be **stable**: that items with duplicate keys appear in the same order in the output as the input. Thus, we choose chains that will support a sequence **queue interface** to keep items in order, inserting to the end of the queue, and then returning items back in the order that they were inserted.

```
def counting_sort(A):
    "Sort A assuming items have non-negative keys"
    u = 1 + max([x.key for x in A])          # O(n) find maximum key
    D = [[] for i in range(u)]               # O(u) direct access array of chains
    for x in A:                              # O(n) insert into chain at x.key
        D[x.key].append(x)
    i = 0
    for chain in D:                          # O(u) read out items in order
        for x in chain:
            A[i] = x
            i += 1
```

Counting sort takes $O(u)$ time to initialize the chains of the direct access array, $O(n)$ time to insert all the elements, and then $O(u)$ time to scan back through the direct access array to return the items; so the algorithm runs in $O(n + u)$ time. Again, when $u = O(n)$, then counting sort runs in linear time, but this time allowing duplicate keys.

There's another implementation of counting sort which just keeps track of how many of each key map to each index, and then moves each item only once, rather the implementation above which moves each item into a chain and then back into place. The implementation below computes the final index location of each item via cumulative sums.

```
def counting_sort(A):
    "Sort A assuming items have non-negative keys"
    u = 1 + max([x.key for x in A])          # O(n) find maximum key
    D = [0] * u                              # O(u) direct access array
    for x in A:                              # O(n) count keys
        D[x.key] += 1
    for k in range(1, u):                    # O(u) cumulative sums
        D[k] += D[k - 1]
    for x in list(reversed(A)):              # O(n) move items into place
        A[D[x.key] - 1] = x
        D[x.key] -= 1
```

Now what if we want to sort keys from a larger integer range? Our strategy will be to break up integer keys into parts, and then sort each part! In order to do that, we will need a sorting strategy to sort tuples, i.e. multiple parts.

Tuple Sort

Suppose we want to sort tuples, each containing many different keys (e.g. $x.k_1, x.k_2, x.k_3, \ldots$), so that the sort is lexicographic with respect to some ordering of the keys (e.g. that key k_1 is more important than key k_2 is more important than key k_3, etc.). Then **tuple sort** uses a stable sorting algorithm as a subroutine to repeatedly sort the objects, first according to the **least important key**, then the second least important key, all the way up to most important key, thus lexicographically sorting the objects. Tuple sort is similar to how one might sort on multiple rows of a spreadsheet by different columns. However, tuple sort will only be correct if the sorting from previous rounds are maintained in future rounds. In particular, tuple sort requires the subroutine sorting algorithms be stable.

Radix Sort

Now, to increase the range of integer sets that we can sort in linear time, we break each integer up into its multiples of powers of n, representing each item key its sequence of digits when represented in base n. If the integers are non-negative and the largest integer in the set is u, then this base n number will have $\lceil \log_n u \rceil$ digits. We can think of these digit representations as tuples and sort them with tuple sort by sorting on each digit in order from least significant to most significant digit using counting sort. This combination of tuple sort and counting sort is called radix sort. If the largest integer in the set $u \leq n^c$, then radix sort runs in $O(nc)$ time. Thus, if c is constant, then radix sort also runs in linear time!

```
def radix_sort(A):
    "Sort A assuming items have non-negative keys"
    n = len(A)
    u = 1 + max([x.key for x in A])             # O(n) find maximum key
    c = 1 + (u.bit_length() // n.bit_length())
    class Obj: pass
    D = [Obj() for a in A]
    for i in range(n):                          # O(nc) make digit tuples
        D[i].digits = []
        D[i].item = A[i]
        high = A[i].key
        for j in range(c):                      # O(c) make digit tuple
            high, low = divmod(high, n)
            D[i].digits.append(low)
    for i in range(c):                          # O(nc) sort each digit
        for j in range(n):                      # O(n) assign key i to tuples
            D[j].key = D[j].digits[i]
        counting_sort(D)                        # O(n) sort on digit i
    for i in range(n):                          # O(n) output to A
        A[i] = D[i].item
```

We've made a CoffeeScript Counting/Radix sort visualizer which you can find here:
https://codepen.io/mit6006/pen/LqZqrd

Exercises

1) Sort the following integers using a base-10 radix sort.

$$(329, 457, 657, 839, 436, 720, 355) \longrightarrow (329, 355, 436, 457, 657, 720, 839)$$

2) Describe a linear time algorithm to sort n integers from the range $[-n^2, \ldots, n^3]$.
Solution: Add n^2 to each number so integers are all positive, apply Radix sort, and then subtract n^2 from each element of the output.

3) Describe a linear time algorithm to sort a set n of strings, each having k English characters.
Solution: Use tuple sort to repeatedly sort the strings by each character from right to left with counting sort, using the integers $\{0, \ldots, 25\}$ to represent the English alphabet. There are k rounds of counting sort, and each round takes $\Theta(n + 26) = \Theta(n)$ time, thus the algorithm runs in $\Theta(nk)$ time. This running time is linear because the input size is $\Theta(nk)$.

6.006 Introduction to Algorithms

Spring 2020

Recitation 6

Binary Trees

A **binary tree** is a tree (a connected graph with no cycles) of **binary nodes**: a linked node container, similar to a linked list node, having a constant number of fields:

- a pointer to an item stored at the node,
- a pointer to a **parent node** (possibly `None`),
- a pointer to a **left child** node (possibly `None`), and
- a pointer to a **right child** node (possibly `None`).

```
class Binary_Node:
    def __init__(A, x):                    # O(1)
        A.item   = x
        A.left   = None
        A.right  = None
        A.parent = None
        # A.subtree_update()               # wait for R07!
```

Why is a binary node called "binary"? In actuality, a binary node can be connected to **three** other nodes (its parent, left child, and right child), not just two. However, we will differentiate a node's parent from it's children, and so we call the node "binary" based on the number of children the node has.

A binary tree has one node that that is the **root** of the tree: the only node in the tree lacking a parent. All other nodes in the tree can reach the root of the tree containing them by traversing parent pointers. The set of nodes passed when traversing parent pointers from node `<X>` back to the root are called the **ancestors** for `<X>` in the tree. The **depth** of a node `<X>` in the subtree rooted at `<R>` is the length of the path from `<X>` back to `<R>`. The **height** of node `<X>` is the maximum depth of any node in the subtree rooted at `<X>`. If a node has no children, it is called a **leaf**.

Why would we want to store items in a binary tree? The difficulty with a linked list is that many linked-list nodes can be $O(n)$ pointer hops away from the head of the list, so it may take $O(n)$ time to reach them. By contrast, as we've seen in earlier recitations, it is possible to construct a binary tree on n nodes such that no node is more than $O(\log n)$ pointer hops away from the root, i.e., there exist binary trees with logarithmic height. The power of a binary tree structure is if we can keep the height h of the tree low, i.e., $O(\log n)$, and only perform operations on the tree that run in time on the order of the height of the tree, then these operations will run in $O(h) = O(\log n)$ time (which is much closer to $O(1)$ than to $O(n)$).

Traversal Order

The nodes in a binary tree have a natural order based on the fact that we distinguish one child to be left and one child to be right. We define a binary tree's **traversal order** based on the following implicit characterization:

- every node in the left subtree of node <X> comes **before** <X> in the traversal order; and
- every node in the right subtree of node <X> comes **after** <X> in the traversal order.

Given a binary node <A>, we can list the nodes in <A>'s subtree by recursively listing the nodes in <A>'s left subtree, listing <A> itself, and then recursively listing the nodes in <A>'s right subtree. This algorithm runs in $O(n)$ time because every node is recursed on once doing constant work.

```
1    def subtree_iter(A):                    # O(n)
2        if A.left:   yield from A.left.subtree_iter()
3        yield A
4        if A.right:  yield from A.right.subtree_iter()
```

Right now, there is no semantic connection between the items being stored and the traversal order of the tree. Next time, we will provide two different semantic meanings to the traversal order (one of which will lead to an efficient implementation of the Sequence interface, and the other will lead to an efficient implementation of the Set interface), but for now, we will just want to preserve the traversal order as we manipulate the tree.

Tree Navigation

Given a binary tree, it will be useful to be able to navigate the nodes in their traversal order efficiently. Probably the most straight forward operation is to find the node in a given node's subtree that appears first (or last) in traversal order. To find the first node, simply walk left if a left child exists. This operation takes $O(h)$ time because each step of the recursion moves down the tree. Find the last node in a subtree is symmetric.

```
1    def subtree_first(A):                   # O(h)
2        if A.left:  return A.left.subtree_first()
3        else:       return A
4
5    def subtree_last(A):                    # O(h)
6        if A.right: return A.right.subtree_last()
7        else:       return A
```

Given a node in a binary tree, it would also be useful too find the next node in the traversal order, i.e., the node's **successor**, or the previous node in the traversal order, i.e., the node's **predecessor**. To find the successor of a node <A>, if <A> has a right child, then <A>'s successor will be the first node in the right child's subtree. Otherwise, <A>'s successor cannot exist in <A>'s subtree, so we walk up the tree to find the lowest ancestor of <A> such that <A> is in the ancestor's left subtree.

In the first case, the algorithm only walks down the tree to find the successor, so it runs in $O(h)$ time. Alternatively in the second case, the algorithm only walks up the tree to find the successor, so it also runs in $O(h)$ time. The predecessor algorithm is symmetric.

```
    def successor(A):                          # O(h)
        if A.right: return A.right.subtree_first()
        while A.parent and (A is A.parent.right):
            A = A.parent
        return A.parent

    def predecessor(A):                        # O(h)
        if A.left:  return A.left.subtree_last()
        while A.parent and (A is A.parent.left):
            A = A.parent
        return A.parent
```

Dynamic Operations

If we want to add or remove items in a binary tree, we must take care to preserve the traversal order of the other items in the tree. To insert a node <B> before a given node <A> in the traversal order, either node <A> has a left child or not. If <A> does not have a left child, than we can simply add <B> as the left child of <A>. Otherwise, if <A> has a left child, we can add <B> as the right child of the last node in <A>'s left subtree (which cannot have a right child). In either case, the algorithm walks down the tree at each step, so the algorithm runs in $O(h)$ time. Inserting after is symmetric.

```
    def subtree_insert_before(A, B):           # O(h)
        if A.left:
            A = A.left.subtree_last()
            A.right, B.parent = B, A
        else:
            A.left,  B.parent = B, A
        # A.maintain()                         # wait for R07!

    def subtree_insert_after(A, B):            # O(h)
        if A.right:
            A = A.right.subtree_first()
            A.left,  B.parent = B, A
        else:
            A.right, B.parent = B, A
        # A.maintain()                         # wait for R07!
```

To delete the item contained in a given node from its binary tree, there are two cases based on whether the node storing the item is a leaf. If the node is a leaf, then we can simply clear the child pointer from the node's parent and return the node. Alternatively, if the node is not a leaf, we can swap the node's item with the item in the node's successor or predecessor down the tree until the item is in a leaf which can be removed. Since swapping only occurs down the tree, again this operation runs in $O(h)$ time.

```
    def subtree_delete(A):                  # O(h)
        if A.left or A.right:               # A is not a leaf
            if A.left:  B = A.predecessor()
            else:       B = A.successor()
            A.item, B.item = B.item, A.item
            return B.subtree_delete()
        if A.parent:                        # A is a leaf
            if A.parent.left is A:  A.parent.left  = None
            else:                   A.parent.right = None
            # A.parent.maintain()          # wait for R07!
        return A
```

Binary Node Full Implementation

```
class Binary_Node:
    def __init__(A, x):                     # O(1)
        A.item   = x
        A.left   = None
        A.right  = None
        A.parent = None
        # A.subtree_update()                # wait for R07!

    def subtree_iter(A):                    # O(n)
        if A.left:   yield from A.left.subtree_iter()
        yield A
        if A.right:  yield from A.right.subtree_iter()

    def subtree_first(A):                   # O(h)
        if A.left:  return A.left.subtree_first()
        else:       return A

    def subtree_last(A):                    # O(h)
        if A.right: return A.right.subtree_last()
        else:       return A

    def successor(A):                       # O(h)
        if A.right: return A.right.subtree_first()
        while A.parent and (A is A.parent.right):
            A = A.parent
        return A.parent

    def predecessor(A):                     # O(h)
        if A.left:  return A.left.subtree_last()
        while A.parent and (A is A.parent.left):
            A = A.parent
        return A.parent

```

```
34      def subtree_insert_before(A, B):        # O(h)
35          if A.left:
36              A = A.left.subtree_last()
37              A.right, B.parent = B, A
38          else:
39              A.left,  B.parent = B, A
40          # A.maintain()                      # wait for R07!
41
42      def subtree_insert_after(A, B):         # O(h)
43          if A.right:
44              A = A.right.subtree_first()
45              A.left,  B.parent = B, A
46          else:
47              A.right, B.parent = B, A
48          # A.maintain()                      # wait for R07!
49
50      def subtree_delete(A):                  # O(h)
51          if A.left or A.right:
52              if A.left:  B = A.predecessor()
53              else:       B = A.successor()
54              A.item, B.item = B.item, A.item
55              return B.subtree_delete()
56          if A.parent:
57              if A.parent.left is A:  A.parent.left  = None
58              else:                   A.parent.right = None
59               # A.parent.maintain()          # wait for R07!
60          return A
```

Top-Level Data Structure

All of the operations we have defined so far have been within the `Binary_Tree` class, so that they apply to any subtree. Now we can finally define a general Binary Tree data structure that stores a pointer to its root, and the number of items it stores. We can implement the same operations with a little extra work to keep track of the root and size.

```
1   class Binary_Tree:
2       def __init__(T, Node_Type = Binary_Node):
3           T.root = None
4           T.size = 0
5           T.Node_Type = Node_Type
6
7       def __len__(T): return T.size
8       def __iter__(T):
9           if T.root:
10              for A in T.root.subtree_iter():
11                  yield A.item
```

Exercise: Given an array of items $A = (a_0, \ldots, a_{n-1})$, describe a $O(n)$-time algorithm to construct a binary tree T containing the items in A such that (1) the item stored in the i^{th} node of T's traversal order is item a_i, and (2) T has height $O(\log n)$.

Solution: Build T by storing the middle item in a root node, and then recursively building the remaining left and right halves in left and right subtrees. This algorithm satisfies property (1) by definition of traversal order, and property (2) because the height roughly follows the recurrence $H(n) = 1 + H(n/2)$. The algorithm runs in $O(n)$ time because every node is recursed on once doing constant work.

```
def build(X):
    A = [x for x in X]
    def build_subtree(A, i, j):
        c = (i + j) // 2
        root = self.Node_Type(A[c])
        if i < c:                   # needs to store more items in left subtree
            root.left = build_subtree(A, i, c - 1)
            root.left.parent = root
        if c < j:                   # needs to store more items in right subtree
            root.right = build_subtree(A, c + 1, j)
            root.right.parent = root
        return root
    self.root = build_subtree(A, 0, len(A)-1)
```

Exercise: Argue that the following iterative procedure to return the nodes of a tree in traversal order takes $O(n)$ time.

```
def tree_iter(T):
    node = T.subtree_first()
    while node:
        yield node
        node = node.successor()
```

Solution: This procedure walks around the tree traversing each edge of the tree twice: once going down the tree, and once going back up. Then because the number of edges in a tree is one fewer than the number of nodes, the traversal takes $O(n)$ time.

Application: Set

To use a Binary Tree to implement a Set interface, we use the traversal order of the tree to store the items sorted in increasing key order. This property is often called the **Binary Search Tree Property**, where keys in a node's left subtree are less than the key stored at the node, and keys in the node's right subtree are greater than the key stored at the node. Then finding the node containing a query key (or determining that no node contains the key) can be done by walking down the tree, recursing on the appropriate side.

Exercise: Make a Set Binary Tree (Binary Search Tree) by inserting student-chosen items one by one, then searching and/or deleting student-chosen keys one by one.

```
class BST_Node(Binary_Node):
    def subtree_find(A, k):                 # O(h)
        if k < A.item.key:
            if A.left:  return A.left.subtree_find(k)
        elif k > A.item.key:
            if A.right: return A.right.subtree_find(k)
        else:           return A
        return None

    def subtree_find_next(A, k):            # O(h)
        if A.item.key <= k:
            if A.right: return A.right.subtree_find_next(k)
            else:       return None
        elif A.left:
            B = A.left.subtree_find_next(k)
            if B:       return B
        return A

    def subtree_find_prev(A, k):             # O(h)
        if A.item.key >= k:
            if A.left:  return A.left.subtree_find_prev(k)
            else:       return None
        elif A.right:
            B = A.right.subtree_find_prev(k)
            if B:       return B
        return A

    def subtree_insert(A, B):                # O(h)
        if B.item.key < A.item.key:
            if A.left:  A.left.subtree_insert(B)
            else:       A.subtree_insert_before(B)
        elif B.item.key > A.item.key:
            if A.right: A.right.subtree_insert(B)
            else:       A.subtree_insert_after(B)
        else:    A.item = B.item
```

```
class Set_Binary_Tree(Binary_Tree):  # Binary Search Tree
    def __init__(self): super().__init__(BST_Node)

    def iter_order(self): yield from self

    def build(self, X):
        for x in X: self.insert(x)

    def find_min(self):
        if self.root:   return self.root.subtree_first().item

    def find_max(self):
        if self.root:   return self.root.subtree_last().item

    def find(self, k):
        if self.root:
            node = self.root.subtree_find(k)
            if node:    return node.item

    def find_next(self, k):
        if self.root:
            node = self.root.subtree_find_next(k)
            if node:    return node.item

    def find_prev(self, k):
        if self.root:
            node = self.root.subtree_find_prev(k)
            if node:    return node.item

    def insert(self, x):
        new_node = self.Node_Type(x)
        if self.root:
            self.root.subtree_insert(new_node)
            if new_node.parent is None: return False
        else:
            self.root = new_node
        self.size += 1
        return True

    def delete(self, k):
        assert self.root
        node = self.root.subtree_find(k)
        assert node
        ext = node.subtree_delete()
        if ext.parent is None:  self.root = None
        self.size -= 1
        return ext.item
```

6.006 Introduction to Algorithms

Spring 2020

Recitation 7

Balanced Binary Trees

Previously, we discussed binary trees as a general data structure for storing items, without bounding the maximum height of the tree. The ultimate goal will be to keep our tree **balanced**: a tree on n nodes is balanced if its height is $O(\log n)$. Then all the $O(h)$-time operations we talked about last time will only take $O(\log n)$ time.

There are many ways to keep a binary tree balanced under insertions and deletions (Red-Black Trees, B-Trees, 2-3 Trees, Splay Trees, etc.). The oldest (and perhaps simplest) method is called an **AVL Tree**. Every node of an AVL Tree is **height-balanced** (i.e., satisfies the **AVL Property**) where the left and right subtrees of a height-balanced node differ in height by at most 1. To put it a different way, define the **skew** of a node to be the height of its right subtree minus the height of its left subtree (where the height of an empty subtree is -1. Then a node is height-balanced if it's skew is either $-1, 0,$ or 1. A tree is height-balanced if every node in the tree is height-balanced. Height-balance is good because it implies balance!

Exercise: A height-balanced tree is balanced.
Solution: Balanced means that $h = O(\log n)$. Equivalently, balanced means that $\log n$ is lower bounded by $\Omega(h)$ so that $n = 2^{\Omega(h)}$. So if we can show the minimum number of nodes in a height-balanced tree is at least exponential in h, then it must also be balanced. Let $F(h)$ denote the fewest nodes in any height-balanced tree of height h. Then $F(h)$ satisfies the recurrence:

$$F(h) = 1 + F(h-1) + F(h-2) \geq 2F(h-2),$$

since the subtrees of the root's children should also contain the fewest nodes. As base cases, the fewest nodes in a height-balanced tree of height 0 is one, i.e., $F(0) = 1$, while the fewest nodes in a height-balanced tree of height 1 is two, i.e., $F(1) = 2$. Then this recurrence is lower bounded by $F(h) \geq 2^{h/2} = 2^{\Omega(h)}$ as desired.

Rotations

As we add or remove nodes to our tree, it is possible that our tree will become imbalanced. We will want to change the structure of the tree without changing its traversal order, in the hopes that we can make the tree's structure more balanced. We can change the structure of a tree using a local operation called a **rotation**. A rotation takes a subtree that locally looks like one the following two configurations and modifies the connections between nodes in $O(1)$ time to transform it into the other configuration.

```
1            ______<D>__       rotate_right(<D>)      __<B>______
2          __<B>__     <E>             =>            <A>     __<D>__
3         <A>    <C>    / \                           / \    <C>    <E>
4         / \    / \   /___\           <=            /___\   / \    / \
5        /___\  /___\                rotate_left(<B>)          /___\  /___\
```

This operation preserves the traversal order of the tree while changing the depth of the nodes in subtrees `<A>` and `<E>`. Next time, we will use rotations to enforce that a balanced tree stays balanced after inserting or deleting a node.

```
1  def subtree_rotate_right(D):
2      assert D.left
3      B, E = D.left, D.right
4      A, C = B.left, B.right
5      D, B = B, D
6      D.item, B.item = B.item, D.item
7      B.left, B.right = A, D
8      D.left, D.right = C, E
9      if A: A.parent = B
10     if E: E.parent = D
11     # B.subtree_update()
12     # D.subtree_update()
```

```
def subtree_rotate_left(B):  # O(1)
    assert B.right
    A, D = B.left, B.right
    C, E = D.left, D.right
    B, D = D, B
    B.item, D.item = D.item, B.item
    D.left, D.right = B, E
    B.left, B.right = A, C
    if A: A.parent = B
    if E: E.parent = D
    # B.subtree_update()      # wait for R07!
    # D.subtree_update()      # wait for R07!
```

Maintaining Height-Balance

Suppose we have a height-balanced AVL tree, and we perform a single insertion or deletion by adding or removing a leaf. Either the resulting tree is also height-balanced, or the change in leaf has made at least one node in the tree have magnitude of skew greater than 1. In particular, the only nodes in the tree whose subtrees have changed after the leaf modification are ancestors of that leaf (at most $O(h)$ of them), so these are the only nodes whose skew could have changed and they could have changed by at most 1 to have magnitude at most 2. As shown in lecture via a brief case analysis, given a subtree whose root has skew is 2 and every other node in its subtree is height-balanced, we can restore balance to the subtree in at most two rotations. Thus to rebalance the entire tree, it suffices to walk from the leaf to the root, rebalancing each node along the way, performing at most $O(\log n)$ rotations in total. A detailed proof is outlined in the lecture notes and is not repeated here; but the proof may be reviewed in recitation if students would like to see the

full argument. Below is code to implement the rebalancing algorithm presented in lecture.

```
    def skew(A):                              # O(?)
        return height(A.right) - height(A.left)

    def rebalance(A):                         # O(?)
        if A.skew() == 2:
            if A.right.skew() < 0:
                A.right.subtree_rotate_right()
            A.subtree_rotate_left()
        elif A.skew() == -2:
            if A.left.skew() > 0:
                A.left.subtree_rotate_left()
            A.subtree_rotate_right()

    def maintain(A):                          # O(h)
        A.rebalance()
        A.subtree_update()
        if A.parent: A.parent.maintain()
```

Unfortunately, it's not clear how to efficiently evaluate the skew of a a node to determine whether or not we need to perform rotations, because computing a node's height naively takes time linear in the size of the subtree. The code below to compute height recurses on every node in `<A>`'s subtree, so takes at least $\Omega(n)$ time.

```
def height(A):                            # Omega(n)
    if A is None: return -1
    return 1 + max(height(A.left), height(A.right))
```

Rebalancing requires us to check at least $\Omega(\log n)$ heights in the worst-case, so if we want rebalancing the tree to take at most $O(\log n)$ time, we need to be able to evaluate the height of a node in $O(1)$ time. Instead of computing the height of a node every time we need it, we will speed up computation via augmentation: in particular each node stores and maintains the value of its own subtree height. Then when we're at a node, evaluating its height is a simple as reading its stored value in $O(1)$ time. However, when the structure of the tree changes, we will need to update and recompute the height at nodes whose height has changed.

```
def height(A):
    if A:   return A.height
    else:   return -1


    def subtree_update(A):                    # O(1)
        A.height = 1 + max(height(A.left), height(A.right))
```

In the dynamic operations presented in R06, we put commented code to call update on every node whose subtree changed during insertions, deletions, or rotations. A rebalancing insertion or deletion operation only calls `subtree_update` on at most $O(\log n)$ nodes, so as long as updating a

node takes at most $O(1)$ time to recompute augmentations based on the stored augmentations of the node's children, then the augmentations can be maintained during rebalancing in $O(\log n)$ time.

In general, the idea behind **augmentation** is to store additional information at each node so that information can be queried quickly in the future. You've done some augmentation already in PS1, where you augmented a singly-linked list with back pointers to make it faster to evaluate a node's predecessor. To augment the nodes of a binary tree with a **subtree** property `P(<X>)`, you need to:

- clearly define what property of `<X>`'s subtree corresponds to `P(<X>)`, and
- show how to compute `P(<X>)` in $O(1)$ time from the augmentations of `<X>`'s children.

If you can do that, then you will be able to store and maintain that property at each node without affecting the $O(\log n)$ running time of rebalancing insertions and deletions. We've shown how to traverse around a binary tree and perform insertions and deletions, each in $O(h)$ time while also maintaining height-balance so that $h = O(\log n)$. Now we are finally ready to implement an efficient Sequence and Set.

Binary Node Implementation with AVL Balancing

```
def height(A):
    if A:   return A.height
    else:   return -1

class Binary_Node:
    def __init__(A, x):                     # O(1)
        A.item   = x
        A.left   = None
        A.right  = None
        A.parent = None
        A.subtree_update()

    def subtree_update(A):                  # O(1)
        A.height = 1 + max(height(A.left), height(A.right))

    def skew(A):                            # O(1)
        return height(A.right) - height(A.left)

    def subtree_iter(A):                    # O(n)
        if A.left:   yield from A.left.subtree_iter()
        yield A
        if A.right:  yield from A.right.subtree_iter()
```

```
23
24      def subtree_first(A):                    # O(log n)
25          if A.left:  return A.left.subtree_first()
26          else:       return A
27
28      def subtree_last(A):                     # O(log n)
29          if A.right: return A.right.subtree_last()
30          else:       return A
31
32      def successor(A):                        # O(log n)
33          if A.right: return A.right.subtree_first()
34          while A.parent and (A is A.parent.right):
35              A = A.parent
36          return A.parent
37
38      def predecessor(A):                      # O(log n)
39          if A.left:  return A.left.subtree_last()
40          while A.parent and (A is A.parent.left):
41              A = A.parent
42          return A.parent
43
44      def subtree_insert_before(A, B):         # O(log n)
45          if A.left:
46              A = A.left.subtree_last()
47              A.right, B.parent = B, A
48          else:
49              A.left,  B.parent = B, A
50          A.maintain()
51
52      def subtree_insert_after(A, B):          # O(log n)
53          if A.right:
54              A = A.right.subtree_first()
55              A.left,  B.parent = B, A
56          else:
57              A.right, B.parent = B, A
58          A.maintain()
59
60      def subtree_delete(A):                  # O(log n)
61          if A.left or A.right:
62              if A.left:  B = A.predecessor()
63              else:       B = A.successor()
64              A.item, B.item = B.item, A.item
65              return B.subtree_delete()
66          if A.parent:
67              if A.parent.left is A:  A.parent.left  = None
68              else:                   A.parent.right = None
69              A.parent.maintain()
70          return A
71
72
73
```

```
74      def subtree_rotate_right(D):              # O(1)
75          assert D.left
76          B, E = D.left, D.right
77          A, C = B.left, B.right
78          D, B = B, D
79          D.item, B.item = B.item, D.item
80          B.left, B.right = A, D
81          D.left, D.right = C, E
82          if A: A.parent = B
83          if E: E.parent = D
84          B.subtree_update()
85          D.subtree_update()
86
87      def subtree_rotate_left(B):               # O(1)
88          assert B.right
89          A, D = B.left, B.right
90          C, E = D.left, D.right
91          B, D = D, B
92          B.item, D.item = D.item, B.item
93          D.left, D.right = B, E
94          B.left, B.right = A, C
95          if A: A.parent = B
96          if E: E.parent = D
97          B.subtree_update()
98          D.subtree_update()
99
100     def rebalance(A):                         # O(1)
101         if A.skew() == 2:
102             if A.right.skew() < 0:
103                 A.right.subtree_rotate_right()
104             A.subtree_rotate_left()
105         elif A.skew() == -2:
106             if A.left.skew() > 0:
107                 A.left.subtree_rotate_left()
108             A.subtree_rotate_right()
109
110     def maintain(A):                          # O(log n)
111         A.rebalance()
112         A.subtree_update()
113         if A.parent: A.parent.maintain()
```

Application: Set

Using our new definition of `Binary_Node` that maintains balance, the implementation presented in R06 of the `Binary_Tree_Set` immediately supports all operations in $h = O(\log n)$ time, except `build(X)` and `iter()` which run in $O(n \log n)$ and $O(n)$ time respectively. This data structure is what's normally called an **AVL tree**, but what we will call a **Set AVL**.

Application: Sequence

To use a Binary Tree to implement a Sequence interface, we use the traversal order of the tree to store the items in Sequence order. Now we need a fast way to find the i^{th} item in the sequence because traversal would take $O(n)$ time. If we knew how many items were stored in our left subtree, we could compare that size to the index we are looking for and recurse on the appropriate side. In order to evaluate subtree size efficiently, we augment each node in the tree with the size of its subtree. A node's size can be computed in constant time given the sizes of its children by summing them and adding 1.

```
class Size_Node(Binary_Node):
    def subtree_update(A):                  # O(1)
        super().subtree_update()
        A.size = 1
        if A.left:   A.size += A.left.size
        if A.right:  A.size += A.right.size

    def subtree_at(A, i):                   # O(h)
        assert 0 <= i
        if A.left:        L_size = A.left.size
        else:             L_size = 0
        if i < L_size:    return A.left.subtree_at(i)
        elif i > L_size:  return A.right.subtree_at(i - L_size - 1)
        else:             return A
```

Once we are able to find the i^{th} node in a balanced binary tree in $O(\log n)$ time, the remainder of the Sequence interface operations can be implemented directly using binary tree operations. Further, via the first exercise in R06, we can build such a tree from an input sequence in $O(n)$ time. We call this data structure a **Sequence AVL**.

Implementations of both the Sequence and Set interfaces can be found on the following pages. We've made a CoffeeScript Balanced Binary Search Tree visualizer which you can find here:

https://codepen.io/mit6006/pen/NOWddZ

```
class Seq_Binary_Tree(Binary_Tree):
    def __init__(self): super().__init__(Size_Node)

    def build(self, X):
        def build_subtree(X, i, j):
            c = (i + j) // 2
            root = self.Node_Type(A[c])
            if i < c:
                root.left = build_subtree(X, i, c - 1)
                root.left.parent = root
            if c < j:
                root.right = build_subtree(X, c + 1, j)
                root.right.parent = root
            root.subtree_update()
            return root
        self.root = build_subtree(X, 0, len(X) - 1)
        self.size = self.root.size

    def get_at(self, i):
        assert self.root
        return self.root.subtree_at(i).item

    def set_at(self, i, x):
        assert self.root
        self.root.subtree_at(i).item = x

    def insert_at(self, i, x):
        new_node = self.Node_Type(x)
        if i == 0:
            if self.root:
                node = self.root.subtree_first()
                node.subtree_insert_before(new_node)
            else:
                self.root = new_node
        else:
            node = self.root.subtree_at(i - 1)
            node.subtree_insert_after(new_node)
        self.size += 1

    def delete_at(self, i):
        assert self.root
        node = self.root.subtree_at(i)
        ext = node.subtree_delete()
        if ext.parent is None:  self.root = None
        self.size -= 1
        return ext.item

    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)
```

Exercise: Make a Sequence AVL Tree or Set AVL Tree (Balanced Binary Search Tree) by inserting student chosen items one by one. If any node becomes height-imbalanced, rebalance its ancestors going up the tree. Here's a Sequence AVL Tree example that may be instructive (remember to update subtree heights and sizes as you modify the tree!).

```
T = Seq_Binary_Tree()
T.build([10,6,8,5,1,3])
T.get_at(4)
T.set_at(4, -4)
T.insert_at(4, 18)
T.insert_at(4, 12)
T.delete_at(2)
```

Solution:

```
Line #  1    | 2,3     | 4        | 5          | 6            | 7
             |         |          |            |              |
Result  None | ___8__  | ___8___  | ___8_____  | ___8_______  |   __12____
             | 10_ _1_ | 10_ _-4_ | 10_ ___-4_ | 10_  ____-4_ | __6_  __-4_
             |   6 5 3 |   6 5  3 |   6 5__  3 |   6 _12__  3 | 10 5  18  3
             |         |          |      18    |     5  18    |
```

Also labeled with subtree height H, size #:

```
None
____________8H2#6___________
10H1#2_____      _____1H1#3_____
      6H0#1      5H0#1      3H0#1

____________8H2#6___________
10H1#2_____      _____1H1#3_____
      6H0#1      5H0#1      3H0#1

____________8H2#6____________
10H1#2_____      _____-4H1#3_____
      6H0#1      5H0#1       3H0#1

____________8H3#7____________________
10H1#2_____      ____________-4H2#4_____
      6H0#1      5H1#2_______       3H0#1
                      18H0#1

____________8H3#8_____________________________
10H1#2_____            _____________-4H2#5_____
      6H0#1      _____12H1#3_______       3H0#1
                 5H0#1      18H0#1

      ___________12H2#7_____________
______6H1#3_____       ______-4H1#3_____
10H0#1     5H0#1       18H0#1      3H0#1
```

Exercise: Maintain a sequence of n bits that supports two operations, each in $O(\log n)$ time:

- `flip(i)`: flip the bit at index i
- `count_ones_upto(i)`: return the number of bits in the prefix up to index i that are one

Solution: Maintain a Sequence Tree storing the bits as items, augmenting each node `A` with `A.subtree_ones`, the number of 1 bits in its subtree. We can maintain this augmentation in $O(1)$ time from the augmentations stored at its children.

```
def update(A):
    A.subtree_ones = A.item
    if A.left:
        A.subtree_ones += A.left.subtree_ones
    if A.right:
        A.subtree_ones += A.right.subtree_ones
```

To implement `flip(i)`, find the i^{th} node `A` using `subtree_node_at(i)` and flip the bit stored at `A.item`. Then update the augmentation at `A` and every ancestor of `A` by walking up the tree in $O(\log n)$ time.

To implement `count_ones_upto(i)`, we will first define the subtree-based recursive function `subtree_count_ones_upto(A, i)` which returns the number of 1 bits in the subtree of node `A` that are at most index i within `A`'s subtree. Then `count_ones_upto(i)` is symantically equivilent to `subtree_count_ones_upto(T.root, i)`. Since each recursive call makes at most one recursive call on a child, operation takes $O(\log n)$ time.

```
def subtree_count_ones_upto(A, i):
    assert 0 <= i < A.size
    out = 0
    if A.left:
        if i < A.left.size:
            return subtree_count_ones_upto(A.left, i)
        out += A.left.subtree_ones
        i -= A.left.size
    out += A.item
    if i > 0:
        assert A.right
        out += subtree_count_ones_upto(A.right, i - 1)
    return out
```

6.006 Introduction to Algorithms

Spring 2020

Recitation 8

Priority Queues

Priority queues provide a general framework for at least three sorting algorithms, which differ only in the data structure used in the implementation.

algorithm	data structure	insertion	extraction	total
Selection Sort	Array	$O(1)$	$O(n)$	$O(n^2)$
Insertion Sort	Sorted Array	$O(n)$	$O(1)$	$O(n^2)$
Heap Sort	Binary Heap	$O(\log n)$	$O(\log n)$	$O(n \log n)$

Let's look at Python code that implements these priority queues. We start with an abstract base class that has the interface of a priority queue, maintains an internal array A of items, and trivially implements `insert(x)` and `delete_max()` (the latter being incorrect on its own, but useful for subclasses).

```
class PriorityQueue:
    def __init__(self):
        self.A = []

    def insert(self, x):
        self.A.append(x)

    def delete_max(self):
        if len(self.A) < 1:
            raise IndexError('pop from empty priority queue')
        return self.A.pop()                 # NOT correct on its own!

    @classmethod
    def sort(Queue, A):
        pq = Queue()                        # make empty priority queue
        for x in A:                         # n x T_insert
            pq.insert(x)
        out = [pq.delete_max() for _ in A]  # n x T_delete_max
        out.reverse()
        return out
```

Shared across all implementations is a method for sorting, given implementations of `insert` and `delete_max`. Sorting simply makes two loops over the array: one to insert all the elements, and another to populate the output array with successive maxima in reverse order.

Array Heaps

We showed implementations of selection sort and merge sort previously in recitation. Here are implementations from the perspective of priority queues. If you were to unroll the organization of this code, you would have essentially the same code as we presented before.

```
class PQ_Array(PriorityQueue):
    # PriorityQueue.insert already correct: appends to end of self.A
    def delete_max(self):           # O(n)
        n, A, m = len(self.A), self.A, 0
        for i in range(1, n):
            if A[m].key < A[i].key:
                m = i
        A[m], A[n] = A[n], A[m]     # swap max with end of array
        return super().delete_max() # pop from end of array
```

```
class PQ_SortedArray(PriorityQueue):
    # PriorityQueue.delete_max already correct: pop from end of self.A
    def insert(self, *args):            # O(n)
        super().insert(*args)           # append to end of array
        i, A = len(self.A) - 1, self.A  # restore array ordering
        while 0 < i and A[i + 1].key < A[i].key:
            A[i + 1], A[i] = A[i], A[i + 1]
            i -= 1
```

We use `*args` to allow `insert` to take one argument (as makes sense now) or zero arguments; we will need the latter functionality when making the priority queues in-place.

Binary Heaps

The next implementation is based on a binary heap, which takes advantage of the logarithmic height of a complete binary tree to improve performance. The bulk of the work done by these functions are encapsulated by `max_heapify_up` and `max_heapify_down` below.

```
class PQ_Heap(PriorityQueue):
    def insert(self, *args):         # O(log n)
        super().insert(*args)        # append to end of array
        n, A = self.n, self.A
        max_heapify_up(A, n, n - 1)

    def delete_max(self):            # O(log n)
        n, A = self.n, self.A
        A[0], A[n] = A[n], A[0]
        max_heapify_down(A, n, 0)
        return super().delete_max() # pop from end of array
```

Before we define `max_heapify` operations, we need functions to compute parent and child indices given an index representing a node in a tree whose root is the first element of the array. In this implementation, if the computed index lies outside the bounds of the array, we return the input index. Always returning a valid array index instead of throwing an error helps to simplify future code.

```
def parent(i):
    p = (i - 1) // 2
    return p if 0 < i else i

def left(i, n):
    l = 2 * i + 1
    return l if l < n else i

def right(i, n):
    r = 2 * i + 2
    return r if r < n else i
```

Here is the meat of the work done by a max heap. Assuming all nodes in `A[:n]` satisfy the Max-Heap Property except for node `A[i]` makes it easy for these functions to maintain the Node Max-Heap Property locally.

```
def max_heapify_up(A, n, c):            # T(c) = O(log c)
    p = parent(c)                       # O(1) index of parent (or c)
    if A[p].key < A[c].key:             # O(1) compare
        A[c], A[p] = A[p], A[c]         # O(1) swap parent
        max_heapify_up(A, n, p)         # T(p) = T(c/2) recursive call on parent
```

```
def max_heapify_down(A, n, p):          # T(p) = O(log n - log p)
    l, r = left(p, n), right(p, n)      # O(1) indices of children (or p)
    c = l if A[r].key < A[l].key else r # O(1) index of largest child
    if A[p].key < A[c].key:             # O(1) compare
        A[c], A[p] = A[p], A[c]         # O(1) swap child
        max_heapify_down(A, n, c)       # T(c) recursive call on child
```

$O(n)$ Build Heap

Recall that repeated insertion using a max heap priority queue takes time $\sum_{i=0}^{n} \log i = \log n! = O(n \log n)$. We can build a max heap in linear time if the whole array is accessible to you. The idea is to construct the heap in *reverse* level order, from the leaves to the root, all the while maintaining that all nodes processed so far maintain the Max-Heap Property by running `max_heapify_down` at each node. As an optimization, we note that the nodes in the last half of the array are all leaves, so we do not need to run `max_heapify_down` on them.

```
def build_max_heap(A):
    n = len(A)
    for i in range(n // 2, -1, -1): # O(n) loop backward over array
        max_heapify_down(A, n, i)   # O(log n - log i)) fix max heap
```

To see that this procedure takes $O(n)$ instead of $O(n \log n)$ time, we compute an upper bound explicitly using summation. In the derivation, we use Stirling's approximation: $n! = \Theta(\sqrt{n}(n/e)^n)$.

$$T(n) < \sum_{i=0}^{n} (\log n - \log i) = \log\left(\frac{n^n}{n!}\right) = O\left(\log\left(\frac{n^n}{\sqrt{n}(n/e)^n}\right)\right)$$
$$= O(\log(e^n/\sqrt{n})) = O(n \log e - \log \sqrt{n}) = O(n)$$

Note that using this linear-time procedure to build a max heap does not affect the **asymptotic** efficiency of heap sort, because each of n `delete_max` still takes $O(\log n)$ time each. But it is **practically** more efficient procedure to initially insert n items into an empty heap.

In-Place Heaps

To make heap sort **in place**[1] (as well as restoring the in-place property of selection sort and insertion sort), we can modify the base class `PriorityQueue` to take an entire array `A` of elements, and maintain the queue itself in the prefix of the first `n` elements of `A` (where `n <= len(A)`). The `insert` function is no longer given a value to insert; instead, it inserts the item already stored in `A[n]`, and incorporates it into the now-larger queue. Similarly, `delete_max` does not return a value; it merely deposits its output into `A[n]` before decreasing its size. This approach only works in the case where all `n insert` operations come before all `n delete_max` operations, as in priority queue sort.

```
class PriorityQueue:
    def __init__(self, A):
        self.n, self.A = 0, A

    def insert(self):           # absorb element A[n] into the queue
        if not self.n < len(self.A):
            raise IndexError('insert into full priority queue')
        self.n += 1

    def delete_max(self):       # remove element A[n - 1] from the queue
        if self.n < 1:
            raise IndexError('pop from empty priority queue')
        self.n -= 1             # NOT correct on its own!

    @classmethod
    def sort(Queue, A):
        pq = Queue(A)           # make empty priority queue
        for i in range(len(A)): # n x T_insert
            pq.insert()
        for i in range(len(A)): # n x T_delete_max
            pq.delete_max()
        return pq.A
```

This new base class works for sorting via any of the subclasses: `PQ_Array`, `PQ_SortedArray`, `PQ_Heap`. The first two sorting algorithms are even closer to the original selection sort and insertion sort, and the final algorithm is what is normally referred to as **heap sort**.

We've made a CoffeeScript heap visualizer which you can find here:
https://codepen.io/mit6006/pen/KxOpep

[1] Recall that an in-place sort only uses $O(1)$ additional space during execution, so only a constant number of array elements can exist outside the array at any given time.

Exercises

1. Draw the complete binary tree associated with the sub-array array `A[:8]`. Turn it into a max heap via linear time bottom-up heap-ification. Run `insert` twice, and then `delete_max` twice.

```
A = [7, 3, 5, 6, 2, 0, 3, 1, 9, 4]
```

2. How would you find the **minimum** element contained in a **max** heap?

   Solution: A max heap has no guarantees on the location of its minimum element, except that it may not have any children. Therefore, one must search over all $n/2$ leaves of the binary tree which takes $\Omega(n)$ time.

3. How long would it take to convert a **max** heap to a **min** heap?

   Solution: Run a modified `build_max_heap` on the original heap, enforcing a **Min**-Heap Property instead of a Max-Heap Property. This takes linear time. The fact that the original heap was a max heap does not improve the running time.

4. **Proximate Sorting**: An array of **distinct** integers is ***k-proximate*** if every integer of the array is at most k places away from its place in the array after being sorted, i.e., if the ith integer of the unsorted input array is the jth largest integer contained in the array, then $|i - j| \leq k$. In this problem, we will show how to sort a k-proximate array faster than $\Theta(n \log n)$.

   (a) Prove that insertion sort (as presented in this class, without any changes) will sort a k-proximate array in $O(nk)$ time.

   Solution: To prove $O(nk)$, we show that each of the n insertion sort rounds swap an item left by at most $O(k)$. In the original ordering, entries that are $\geq 2k$ slots apart must already be ordered correctly: indeed, if $A[s] > A[t]$ but $t - s \geq 2k$, there is no way to reverse the order of these two items while moving each at most k slots. This means that for each entry $A[i]$ in the original order, fewer than $2k$ of the items $A[0], \ldots, A[i-1]$ are greater than $A[i]$. Thus, on round i of insertion sort when $A[i]$ is swapped into place, fewer than $2k$ swaps are required, so round i requires $O(k)$ time.

   It's possible to prove a stronger bound: that $a_i = A[i]$ is swapped at most k times in round i (instead of $2k$). This is a bit subtle: the final sorted index of a_i is at most k slots away from i by the k-proximate assumption, but a_i might not move to its final position immediately, but may move **past** its final sorted position and then be bumped to the right in future rounds. Suppose for contradiction a loop swaps the pth largest item $A[i]$ to the left by more than k to position $p' < i - k$, past at least k items larger than $A[i]$. Since A is k-proximate, $i - p \leq k$, i.e. $i - k \leq p$, so $p' < p$. Thus at least one item less than $A[i]$ must exist to the right of $A[i]$. Let $A[j]$ be the smallest such item, the qth largest item in sorted order. $A[j]$ is smaller than $k+1$ items to the left of $A[j]$, and no item to the right of $A[j]$ is smaller than $A[j]$, so $q \leq j - (k+1)$, i.e. $j - q \geq k + 1$. But A is k-proximate, so $j - q \leq k$, a contradiction.

   (b) $\Theta(nk)$ is asymptotically faster than $\Theta(n^2)$ when $k = o(n)$, but is not asymptotically faster than $\Theta(n \log n)$ when $k = \omega(\log n)$. Describe an algorithm to sort a k-proximate array in $O(n \log k)$ time, which can be faster (but no slower) than $\Theta(n \log n)$.

   Solution: We perform a variant of heap sort, where the heap only stores $k+1$ items at a time. Build a min-heap H out of $A[0], \ldots, A[k-1]$. Then, repeatedly, insert the next item from A into H, and then store `H.delete_min()` as the next entry in sorted order. So we first call `H.insert(A[k])` followed by `B[0] = H.delete_min()`; the next iteration calls `H.insert(A[k+1])` and `B[1] = H.delete_min()`; and so on. (When there are no more entries to insert into H, do only the `delete_min` step.) B is the sorted answer. This algorithm works because the ith smallest entry in array A must be one of $A[0], A[1], \ldots, A[i+k]$ by the k-proximate assumption, and by the time we're about to write $B[i]$, all of these entries have already been inserted into H (and some also deleted). Assuming entries $B[0], \ldots, B[i-1]$ are correct (by induction), this means the ith smallest value is still in H while all smaller values have already been removed, so this ith smallest value is in fact `H.delete_min()`, and $B[i]$ gets filled correctly. Each heap operation takes time $O(\log k)$ because there are at most $k+1$ items in the heap, so the n insertions and n deletions take $O(n \log k)$ total.

6.006 Introduction to Algorithms

Spring 2020

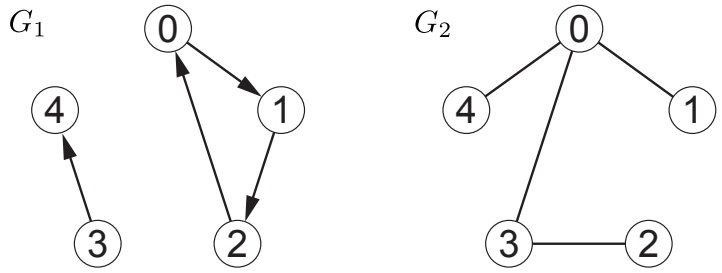
Recitation 9

Graphs

A graph $G = (V, E)$ is a mathematical object comprising a set of **vertices** V (also called nodes) and a set of **edges** E, where each edge in E is a two-element subset of vertices from V. A vertex and edge are **incident** or **adjacent** if the edge contains the vertex. Let u and v be vertices. An edge is **directed** if its subset pair is **ordered**, e.g., (u, v), and **undirected** if its subset pair is **unordered**, e.g., $\{u, v\}$ or alternatively both (u, v) and (v, u). A directed edge $e = (u, v)$ extends from vertex u (e's **tail**) to vertex v (e's **head**), with e an **incoming** edge of v and an **outgoing** edge of u. In an undirected graph, every edge is incoming and outgoing. The **in-degree** and **out-degree** of a vertex v denotes the number of incoming and outgoing edges connected to v respectively. Unless otherwise specified, when we talk about degree, we generally mean out-degree.

As their name suggest, graphs are often depicted **graph**ically, with vertices drawn as points, and edges drawn as lines connecting the points. If an edge is directed, its corresponding line typically includes an indication of the direction of the edge, for example via an arrowhead near the edge's head. Below are examples of a directed graph G_1 and an undirected graph G_2.

$$G_1 = (V_1, E_1) \qquad V_1 = \{0, 1, 2, 3, 4\} \qquad E_1 = \{(0, 1), (1, 2), (2, 0), (3, 4)\}$$
$$G_2 = (V_2, E_2) \qquad V_2 = \{0, 1, 2, 3, 4\} \qquad E_2 = \{\{0, 1\}, \{0, 3\}, \{0, 4\}, \{2, 3\}\}$$



A **path**[1] in a graph is a sequence of vertices $(v_0, \ldots, v_k)$ such that for every ordered pair of vertices (v_i, v_{i+1}), there exists an outgoing edge in the graph from v_i to v_{i+1}. The **length** of a path is the number of edges in the path, or one less than the number of vertices. A graph is called **strongly connected** if there is a path from every node to every other node in the graph. Note that every connected undirected graph is also strongly connected because every undirected edge incident to a vertex is also outgoing. Of the two connected components of directed graph G_1, only one of them is strongly connected.

[1] These are "walks" in 6.042. A "path" in 6.042 does not repeat vertices, which we would call a **simple path**.

Graph Representations

There are many ways to represent a graph in code. The most common way is to store a Set data structure Adj mapping each vertex u to another data structure $\mathrm{Adj}(u)$ storing the **adjacencies** of v, i.e., the set of vertices that are accessible from v via a single outgoing edge. This inner data structure is called an **adjacency list**. Note that we don't store the edge pairs explicitly; we store only the out-going neighbor vertices for each vertex. When vertices are uniquely labeled from 0 to $|V| - 1$, it is common to store the top-level Set Adj within a direct access array of length $|V|$, where array slot i points to the adjacency list of the vertex labeled i. Otherwise, if the vertices are not labeled in this way, it is also common to use a hash table to map each $u \in V$ to $\mathrm{Adj}(u)$. Then, it is common to store each adjacency list $\mathrm{Adj}(u)$ as a simple unordered array of the outgoing adjacencies. For example, the following are adjacency list representations of G_1 and G_2, using a direct access array for the top-level Set and an array for each adjacency list.

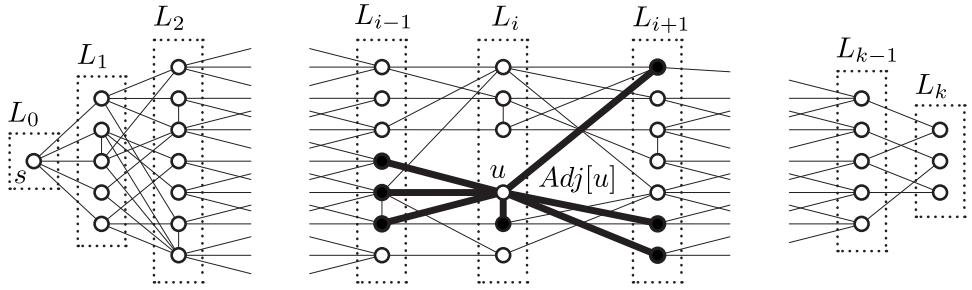
```
A1 = [[1],                  A2 = [[1, 4, 3],            # 0
      [2],                       [0],                   # 1
      [0],                       [3],                   # 2
      [4],                       [0, 2],                # 3
      []]                        [0]]                   # 4
```

Using an array for an adjacency list is a perfectly good data structures if all you need to do is loop over the edges incident to a vertex (which will be the case for all algorithms we will discuss in this class, so will be our default implementation). Each edge appears in any adjacency list at most twice, so the size of an adjacency list representation implemented using arrays is $\Theta(|V| + |E|)$. A drawback of this representation is that determining whether your graph contains a given edge (u, v) might require $\Omega(|V|)$ time to step through the array representing the adjacency list of u or v. We can overcome this obstacle by storing adjacency lists using hash tables instead of regular unsorted arrays, which will support edge checking in expected $O(1)$ time, still using only $\Theta(|V|+|E|)$ space. However, we won't need this operation for our algorithms, so we will assume the simpler unsorted-array-based adjacency list representation. Below are representations of G_1 and G_2 that use a hash table for both the outer Adj Set and the inner adjacency lists $\mathrm{Adj}(u)$, using Python dictionaries:

```
S1 = {0: {1},               S2 = {0: {1, 3, 4},         # 0
      1: {2},                     1: {0},               # 1
      2: {0},                     2: {3},               # 2
      3: {4}}                     3: {0, 2},            # 3
                                  4: {0}}               # 4
```

Breadth-First Search

Given a graph, a common query is to find the vertices reachable by a path from a queried vertex s. A **breadth-first search** (BFS) from s discovers the **level sets** of s: level L_i is the set of vertices reachable from s via a **shortest** path of length i (not reachable via a path of shorter length). Breadth-first search discovers levels in increasing order starting with $i = 0$, where $L_0 = \{s\}$ since the only vertex reachable from s via a path of length $i = 0$ is s itself. Then any vertex reachable from s via a shortest path of length $i + 1$ must have an incoming edge from a vertex whose shortest path from s has length i, so it is contained in level L_i. So to compute level L_{i+1}, include every vertex with an incoming edge from a vertex in L_i, that has not already been assigned a level. By computing each level from the preceding level, a growing frontier of vertices will be explored according to their shortest path length from s.



Below is Python code implementing breadth-first search for a graph represented using index-labeled adjacency lists, returning a parent label for each vertex in the direction of a shortest path back to s. Parent labels (**pointers**) together determine a **BFS tree** from vertex s, containing some shortest path from s to every other vertex in the graph.

```
1  def bfs(Adj, s):                          # Adj: adjacency list, s: starting vertex
2      parent = [None for v in Adj]          # O(V) (use hash if unlabeled)
3      parent[s] = s                         # O(1) root
4      level = [[s]]                         # O(1) initialize levels
5      while 0 < len(level[-1]):             # O(?) last level contains vertices
6          level.append([])                  # O(1) amortized, make new level
7          for u in level[-2]:               # O(?) loop over last full level
8              for v in Adj[u]:              # O(Adj[u]) loop over neighbors
9                  if parent[v] is None:     # O(1) parent not yet assigned
10                     parent[v] = u         # O(1) assign parent from level[-2]
11                     level[-1].append(v)   # O(1) amortized, add to border
12     return parent
```

How fast is breadth-first search? In particular, how many times can the inner loop on lines 9–11 be executed? A vertex is added to any level at most once in line 11, so the loop in line 7 processes each vertex v at most once. The loop in line 8 cycles through all $\deg(v)$ outgoing edges from vertex v. Thus the inner loop is repeated at most $O(\sum_{v \in V} \deg(v)) = O(|E|)$ times. Because the parent array returned has length $|V|$, breadth-first search runs in $O(|V| + |E|)$ time.

Exercise: For graphs G_1 and G_2, conducting a breadth-first search from vertex v_0 yields the parent labels and level sets below.

```
P1 = [0,      L1 = [[0],     P2 = [0,     L2 = [[0],        # 0
      0,            [1],           0,           [1,3,4],    # 1
      1,            [2],           3,           [2],        # 2
      None,         []]            0,           []]         # 3
      None]                        0]                       # 4
```

We can use parent labels returned by a breadth-first search to construct a shortest path from a vertex s to vertex t, following parent pointers from t backward through the graph to s. Below is Python code to compute the shortest path from s to t which also runs in worst-case $O(|V| + |E|)$ time.

```
def unweighted_shortest_path(Adj, s, t):
    parent = bfs(Adj, s)                    # O(V + E) BFS tree from s
    if parent[t] is None:                   # O(1) t reachable from s?
        return None                         # O(1) no path
    i = t                                   # O(1) label of current vertex
    path = [t]                              # O(1) initialize path
    while i != s:                           # O(V) walk back to s
        i = parent[i]                       # O(1) move to parent
        path.append(i)                      # O(1) amortized add to path
    return path[::-1]                       # O(V) return reversed path
```

Exercise: Given an unweighted graph $G = (V, E)$, find a shortest path from s to t having an **odd** number of edges.

Solution: Construct a new graph $G' = (V', E')$. For every vertex u in V, construct two vertices u_E and u_O in V': these represent reaching the vertex u through an even and odd number of edges, respectively. For every edge (u, v) in E, construct the edges (u_E, v_O) and (u_O, v_E) in E'. Run breadth-first search on G' from s_E to find the shortest path from s_E to t_O. Because G' is bipartite between even and odd vertices, even paths from s_E will always end at even vertices, and odd paths will end at odd vertices, so finding a shortest path from s_E to t_O will represent a path of odd length in the original graph. Because G' has $2|V|$ vertices and $2|E|$ edges, constructing G' and running breadth-first search from s_E each take $O(|V| + |E|)$ time.

6.006 Introduction to Algorithms

Spring 2020

Recitation 10

Depth-First Search

A breadth-first search discovers vertices reachable from a queried vertex s level-by-level outward from s. A **depth-first search** (DFS) also finds all vertices reachable from s, but does so by searching undiscovered vertices as deep as possible before exploring other branches. Instead of exploring all neighbors of s one after another as in a breadth-first search, depth-first searches as far as possible from the first neighbor of s before searching any other neighbor of s. Just as with breadth-first search, depth-first search returns a set of parent pointers for vertices reachable from s in the order the search discovered them, together forming a **DFS tree**. However, unlike a BFS tree, a DFS tree will not represent shortest paths in an unweighted graph. (Additionally, DFS returns an order on vertices discovered which will be discussed later.) Below is Python code implementing a recursive depth-first search for a graph represented using index-labeled adjacency lists.

```
1  def dfs(Adj, s, parent = None, order = None): # Adj: adjacency list, s: start
2      if parent is None:                        # O(1) initialize parent list
3          parent = [None for v in Adj]          # O(V) (use hash if unlabeled)
4          parent[s] = s                         # O(1) root
5          order = []                            # O(1) initialize order array
6      for v in Adj[s]:                          # O(Adj[s]) loop over neighbors
7          if parent[v] is None:                 # O(1) parent not yet assigned
8              parent[v] = s                     # O(1) assign parent
9              dfs(Adj, v, parent, order)        # Recursive call
10     order.append(s)                           # O(1) amortized
11     return parent, order
```

How fast is depth-first search? A recursive `dfs` call is performed only when a vertex does not have a parent pointer, and is given a parent pointer immediately before the recursive call. Thus `dfs` is called on each vertex at most once. Further, the amount of work done by each recursive search from vertex v is proportional to the out-degree $deg(v)$ of v. Thus, the amount of work done by depth-first search is $O(\sum_{v \in V} deg(v)) = O(|E|)$. Because the parent array returned has length $|V|$, depth-first search runs in $O(|V| + |E|)$ time.

Exercise: Describe a graph on n vertices for which BFS and DFS would first visit vertices in the same order.

Solution: Many possible solutions. Two solutions are a chain of vertices from v, or a star graph with an edge from v to every other vertex.

Full Graph Exploration

Of course not all vertices in a graph may be reachable from a query vertex s. To search all vertices in a graph, one can use depth-first search (or breadth-first search) to explore each connected component in the graph by performing a search from each vertex in the graph that has not yet been discovered by the search. Such a search is conceptually equivalent to adding an auxiliary vertex with an outgoing edge to every vertex in the graph and then running breadth-first or depth-first search from the added vertex. Python code searching an entire graph via depth-first search is given below.

```
def full_dfs(Adj):                          # Adj: adjacency list
    parent = [None for v in Adj]            # O(V) (use hash if unlabeled)
    order = []                              # O(1) initialize order list
    for v in range(len(Adj)):               # O(V) loop over vertices
        if parent[v] is None:               # O(1) parent not yet assigned
            parent[v] = v                   # O(1) assign self as parent (a root)
            dfs(Adj, v, parent, order)      # DFS from v (BFS can also be used)
    return parent, order
```

For historical reasons (primarily for its connection to topological sorting as discussed later) **depth-first search** is often used to refer to both a method to search a graph from a specific vertex, **and** as a method to search an entire (as in `graph_explore`). You may do the same when answering problems in this class.

DFS Edge Classification

To help prove things about depth-first search, it can be useful to classify the edges of a graph in relation to a depth-first search tree. Consider a graph edge from vertex u to v. We call the edge a **tree edge** if the edge is part of the DFS tree (i.e. `parent[v] = u`). Otherwise, the edge from u to v is not a tree edge, and is either a **back edge**, **forward edge**, or **cross edge** depending respectively on whether: u is a descendant of v, v is a descendant of u, or neither are descendants of each other, in the DFS tree.

Exercise: Draw a graph, run DFS from a vertex, and classify each edge relative to the DFS tree. Show that forward and cross edges cannot occur when running DFS on an undirected graph.

Exercise: How can you identify back edges computationally?

Solution: While performing a depth-first search, keep track of the set of ancestors of each vertex in the DFS tree during the search (in a direct access array or a hash table). When processing neighbor v of s in `dfs(Adj, s)`, if v is an ancestor of s, then (s, v) is a back edge, and certifies a cycle in the graph.

Topological Sort

A directed graph containing no directed cycle is called a **directed acyclic graph** or a DAG. A **topological sort** of a directed acyclic graph $G = (V, E)$ is a linear ordering of the vertices such that for each edge (u, v) in E, vertex u appears before vertex v in the ordering. In the `dfs` function, vertices are added to the `order` list in the order in which their recursive DFS call finishes. If the graph is acyclic, the order returned by `dfs` (or `graph_search`) is the **reverse** of a topological sort order. Proof by cases. One of `dfs(u)` or `dfs(v)` is called first. If `dfs(u)` was called before `dfs(v)`, `dfs(v)` will start and end before `dfs(u)` completes, so v will appear before u in `order`. Alternatively, if `dfs(v)` was called before `dfs(u)`, `dfs(u)` cannot be called before `dfs(v)` completes, or else a path from v to u would exist, contradicting that the graph is acyclic; so v will be added to `order` before vertex u. Reversing the order returned by DFS will then represent a topological sort order on the vertices.

Exercise: A high school contains many student organization, each with its own hierarchical structure. For example, the school's newspaper has an editor-in-chief who oversees all students contributing to the newspaper, including a food-editor who oversees only students writing about school food. The high school's principal needs to line students up to receive diplomas at graduation, and wants to recognize student leaders by giving a diploma to student a before student b whenever a oversees b in any student organization. Help the principal determine an order to give out diplomas that respects student organization hierarchy, or prove to the principal that no such order exists.

Solution: Construct a graph with one vertex per student, and a directed edge from student a to b if student a oversees student b in some student organization. If this graph contains a cycle, the principal is out of luck. Otherwise, a topological sort of the students according to this graph will satisfy the principal's request. Run DFS on the graph (exploring the whole graph as in `graph_explore`) to obtain an order of DFS vertex finishing times in $O(|V| + |E|)$ time. While performing the DFS, keep track of the ancestors of each vertex in the DFS tree, and evaluate if each new edge processed is a back edge. If a back edge is found from vertex u to v, follow parent pointers back to v from u to obtain a directed cycle in the graph to prove to the principal that no such order exists. Otherwise, if no cycle is found, the graph is acyclic and the order returned by DFS is the reverse of a topological sort, which may then be returned to the principal.

We've made a CoffeeScript graph search visualizer which you can find here:
https://codepen.io/mit6006/pen/dqeKEN

6.006 Introduction to Algorithms

Spring 2020

Recitation 11

Weighted Graphs

For many applications, it is useful to associate a numerical **weight** to edges in a graph. For example, a graph modeling a road network might weight each edge with the length of a road corresponding to the edge, or a graph modeling an online dating network might contain edges from one user to another weighted by directed attraction. A **weighted graph** is then a graph $G = (V, E)$ together with a **weight function** $w : E \to \mathbb{R}$, mapping edges to real-valued weights. In practice, edge weights will often not be represented by a separate function at all, preferring instead to store each weight as a value in an adjacency matrix, or inside an edge object stored in an adjacency list or set. For example, below are randomly weighted adjacency set representations of the graphs from Recitation 11. A function to extract such weights might be: `def w(u,v): return W[u][v].`

```
W1 = [0: {1: -2},          W2 = {0: {1: 1, 3: 2, 4: -1},    # 0
      1: {2:  0},                1: {0: 1},                  # 1
      2: {0:  1},                2: {3: 0},                  # 2
      3: {4:  3}}                3: {0: 2, 2: 0},            # 3
                                 4: {0: -1}}                 # 4
```

Now that you have an idea of how weights could be stored, for the remainder of this class you may simply assume that a weight function w can be stored using $O(|E|)$ space, and can return the weight of an edge in constant time[1]. When referencing the weight of an edge $e = (u, v)$, we will often use the notation $w(u, v)$ interchangeably with $w(e)$ to refer to the weight of an edge.

Exercise: Represent graphs W_1 and W_2 as adjacency matrices. How could you store weights in an adjacency list representation?

Weighted Shortest Paths

A **weighted path** is simply a path in a weighted graph as defined in Recitation 11, where the **weight of the path** is the sum of the weights from edges in the path. Again, we will often abuse our notation: if $\pi = (v_1, \ldots, v_k)$ is a weighted path, we let $w(\pi)$ denote the path's weight $\sum_{i=1}^{k-1} w(v_i, v_{i+1})$. The (single source) **weighted shortest paths** problem asks for a lowest weight path to every vertex v in a graph from an input source vertex s, or an indication that no lowest weight path exists from s to v. We already know how to solve the weighted shortest paths problem on graphs for which all edge weights are positive and are equal to each other: simply run breadth-first search from s to minimize the number of edges traversed, thus minimizing path weight. But when edges have different and/or non-positive weights, breadth-first search cannot be applied directly.

[1] We will typically only be picky with the distinction between worst-case and expected bounds when we want to test your understanding of data structures. Hash tables perform well in practice, so use them!

In fact, when a graph contains a **cycle** (a path starting and ending at the same vertex) that has negative weight, then some shortest paths might not even exist, because for any path containing a vertex from the negative weight cycle, a shorter path can be found by adding a tour around the cycle. If any path from s to some vertex v contains a vertex from a negative weight cycle, we will say the shortest path from s to v is undefined, with weight $-\infty$. If no path exists from s to v, then we will say the shortest path from s to v is undefined, with weight $+\infty$. In addition to breadth-first search, we will present three additional algorithms to compute single source shortest paths that cater to different types of weighted graphs.

Weighted Single Source Shortest Path Algorithms

Restrictions		SSSP Algorithm	
Graph	Weights	Name	Running Time $O(\cdot)$
General	Unweighted	BFS	$\lvert V\rvert + \lvert E\rvert$
DAG	Any	DAG Relaxation	$\lvert V\rvert + \lvert E\rvert$
General	Any	Bellman-Ford	$\lvert V\rvert \cdot \lvert E\rvert$
General	Non-negative	Dijkstra	$\lvert V\rvert \log \lvert V\rvert + \lvert E\rvert$

Relaxation

We've shown you one view of relaxation in lecture. Below is another framework by which you can view DAG relaxation. As a general algorithmic paradigm, a **relaxation** algorithm searches for a solution to an optimization problem by starting with a solution that is not optimal, then iteratively improves the solution until it becomes an optimal solution to the original problem. In the single source shortest paths problem, we would like to find the weight $\delta(s, v)$ of a shortest path from source s to each vertex v in a graph. As a starting point, for each vertex v we will initialize an upper bound estimate $d(v)$ on the shortest path weight from s to v, $+\infty$ for all $d(s, v)$ except $d(s, s) = 0$. During the relaxation algorithm, we will repeatedly **relax** some path estimate $d(s, v)$, decreasing it toward the true shortest path weight $\delta(s, v)$. If ever $d(s, v) = \delta(s, v)$, we say that estimate $d(s, v)$ is fully relaxed. When all shortest path estimates are fully relaxed, we will have solved the original problem. Then an algorithm to find shortest paths could take the following form:

```
def general_relax(Adj, w, s):        # Adj: adjacency list, w: weights, s: start
    d = [float('inf') for _ in Adj] # shortest path estimates d(s, v)
    parent = [None for _ in Adj]    # initialize parent pointers
    d[s], parent[s] = 0, s          # initialize source
    while True:                     # repeat forever!
        relax some d[v] ??          # relax a shortest path estimate d(s, v)
    return d, parent                # return weights, paths via parents
```

There are a number of problems with this algorithm, not least of which is that it never terminates! But if we can repeatedly decrease each shortest path estimates to fully relax each $d(s, v)$, we will have found shortest paths. How do we 'relax' vertices and when do we stop relaxing?

To relax a shortest path estimate $d(s, v)$, we will relax **an incoming edge** to v, from another vertex u. If we maintain that $d(s, u)$ always upper bounds the shortest path from s to u for all $u \in V$, then the true shortest path weight $\delta(s, v)$ can't be larger than $d(s, u) + w(u, v)$ or else going to u along a shortest path and traversing the edge (u, v) would be a shorter path[2]. Thus, if at any time $d(s, u) + w(u, v) < d(s, v)$, we can relax the edge by setting $d(s, v) = d(s, u) + w(u, v)$, strictly improving our shortest path estimate.

```
1 def try_to_relax(Adj, w, d, parent, u, v):
2     if d[v] > d[u] + w(u, v):    # better path through vertex u
3         d[v] = d[u] + w(u, v)    # relax edge with shorter path found
4         parent[v] = u
```

If we only change shortest path estimates via relaxation, than we can prove that the shortest path estimates will never become smaller than true shortest paths.

> **Safety Lemma:** Relaxing an edge maintains $d(s, v) \geq \delta(s, v)$ for all $v \in V$.
>
> *Proof.* We prove a stronger statement, that for all $v \in V$, $d(s, v)$ is either infinite or the weight of some path from s to v (so cannot be larger than a shortest path). This is true at initialization: each $d(s, v)$ is $+\infty$, except for $d(s) = 0$ corresponding to the zero-length path. Now suppose at some other time the claim is true, and we relax edge (u, v). Relaxing the edge decreases $d(s, v)$ to a finite value $d(s, u) + w(u, v)$, which by induction is a length of a path from s to v: a path from s to u and the edge (u, v). □

If ever we arrive at an assignment of all shortest path estimates such that no edge in the graph can be relaxed, then we can prove that shortest path estimates are in fact shortest path distances.

> **Termination Lemma:** If no edge can be relaxed, then $d(s, v) \leq \delta(s, v)$ for all $v \in V$.
>
> *Proof.* Suppose for contradiction $\delta(s, v) < d(s, v)$ so that there is a shorter path π from s to v. Let (a, b) be ther first edge of π such that $d(b) > \delta(s, b)$. Then edge (a, b) can be relaxed, a contradiction. □

So, we can change lines 5-6 of the general relaxation algorithm to repeatedly relax edges from the graph until no edge can be further relaxed.

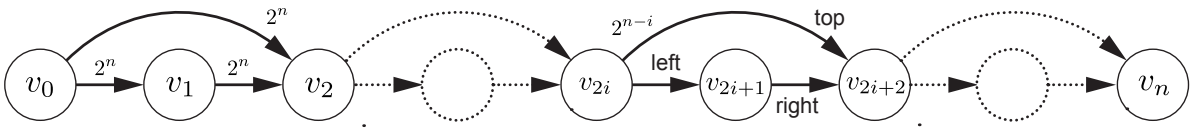
```
1     while some_edge_relaxable(Adj, w, d):
2         (u, v) = get_relaxable_edge(Adj, w, d)
3         try_to_relax(Adj, w, d, parent, u, v)
```

It remains to analyze the running time of this algorithm, which cannot be determined unless we provide detail for how this algorithm chooses edges to relax. If there exists a negative weight cycle in the graph reachable from s, this algorithm will never terminate as edges along the cycle could be relaxed forever. But even for acyclic graphs, this algorithm could take exponential time.

[2]This is a special case of the **triangle inequality**: $\delta(a, c) \leq \delta(a, b) + \delta(b, c)$ for all $a, b, c \in V$.

Exponential Relaxation

How many modifying edge relaxations could occur in an acyclic graph before all edges are fully relaxed? Below is a weighted directed graph on $2n + 1$ vertices and $3n$ edges for which the relaxation framework could perform an **exponential** number of modifying relaxations, if edges are relaxed in a bad order.



This graph contains n sections, with section i containing three edges, (v_{2i}, v_{2i+1}), (v_{2i}, v_{2i+2}), and (v_{2i+1}, v_{2i+2}), each with weight 2^{n-i}; we will call these edges within a section, **left**, **top**, and **right** respectively. In this construction, the lowest weight path from v_0 to v_i is achieved by traversing top edges until v_i's section is reached. Shortest paths from v_0 can easily by found by performing only a linear number of modifying edge relaxations: relax the top and left edges of each successive section. However, a bad relaxation order might result in many more modifying edge relaxations.

To demonstrate a bad relaxation order, initialize all minimum path weight estimates to ∞, except $d(s, s) = 0$ for source $s = v_0$. First relax the left edge, then the right edge of section 0, updating the shortest path estimate at v_2 to $d(s, v_2) = 2^n + 2^n = 2^{n+1}$. In actuality, the shortest path from v_0 to v_2 is via the top edge, i.e., $\delta(s, v_2) = 2^n$. But before relaxing the top edge of section 0, recursively apply this procedure to fully relax the remainder of the graph, from section 1 to $n - 1$, computing shortest path estimates based on the incorrect value of $d(s, v_2) = 2^{n+1}$. Only then relax the top edge of section 0, after which $d(s, v_2)$ is modified to its correct value 2^n. Lastly, fully relax sections 1 through $n - 1$ one more time recursively, to their correct and final values.

How many modifying edge relaxations are performed by this edge relaxation ordering? Let $T(n)$ represent the number of modifying edge relaxations performed by the procedure on a graph containing n sections, with recurrence relation given by $T(n) = 3 + 2T(n - 2)$. The solution to this recurrence is $T(n) = O(2^{n/2})$, exponential in the size of the graph. Perhaps there exists some edge relaxation order requiring only a **polynomial** number of modifying edge relaxations?

DAG Relaxation

In a directed acyclic graph (DAG), there can be no negative weight cycles, so eventually relaxation must terminate. It turns out that relaxing each outgoing edge from every vertex exactly once in a topological sort order of the vertices, correctly computes shortest paths. This shortest paths algorithm is sometimes called **DAG Relaxation**.

```
def DAG_Relaxation(Adj, w, s):      # Adj: adjacency list, w: weights, s: start
    _, order = dfs(Adj, s)          # run depth-first search on graph
    order.reverse()                 # reverse returned order
    d = [float('inf') for _ in Adj] # shortest path estimates d(s, v)
    parent = [None for _ in Adj]    # initialize parent pointers
    d[s], parent[s] = 0, s          # initialize source
    for u in order:                 # loop through vertices in topo sort
        for v in Adj[u]:            # loop through out-going edges of u
            try_to_relax(Adj, w, d, parent, u, v)  # try to relax edge from u to v
    return d, parent                # return weights, paths via parents
```

Claim: The DAG Relaxation algorithm computes shortest paths in a directed acyclic graph.

Proof. We prove that at termination, $d(s, v) = \delta(s, v)$ for all $v \in V$. First observe that Safety ensures that a vertex not reachable from s will retain $d(s, v) = +\infty$ at termination. Alternatively, consider any shortest path $\pi = (v_1, \ldots, v_m)$ from $v_1 = s$ to any vertex $v_m = v$ reachable from s. The topological sort order ensures that edges of the path are relaxed in the order in which they appear in the path. Assume for induction that before edge $(v_i, v_{i+1}) \in \pi$ is relaxed, $d(s, v_i) = \delta(s, v_i)$. Setting $d(s, s) = 0$ at the start provides a base case. Then relaxing edge (v_i, v_{i+1}) sets $d(s, v_{i+1}) = \delta(s, v_i) + w(v_i, v_{i+1}) = \delta(s, v_{i+1})$, as sub-paths of shortest paths are also shortest paths. Thus the procedure constructs shortest path weights as desired. Since depth-first search runs in linear time and the loops relax each edge exactly once, this algorithm takes $O(|V| + |E|)$ time. □

Exercise: You have been recruited by MIT to take part in a new part time student initiative where you will take only one class per term. You don't care about graduating; all you really want to do is to take 19.854, Advanced Quantum Machine Learning on the Blockchain: Neural Interfaces, but are concerned because of its formidable set of prerequisites. MIT professors will allow you take any class as long as you have taken **at least one** of the class's prerequisites prior to taking the class. But passing a class without all the prerequsites is difficult. From a survey of your peers, you know for each class and prerequisite pair, how many hours of stress the class will demand. Given a list of classes, prerequisites, and surveyed stress values, describe a linear time algorithm to find a sequence of classes that minimizes the amount of stress required to take 19.854, never taking more than one prerequisite for any class. You may assume that every class is offered every semester.

Solution: Build a graph with a vertex for every class and a directed edge from class a to class b if b is a prerequisite of a, weighted by the stress of taking class a after having taken class b as a prerequisite. Use topological sort relaxation to find the shortest path from class 19.854 to every other class. From the classes containing no prerequisites (sinks of the DAG), find one with minimum total stress to 19.854, and return its reversed shortest path.

6.006 Introduction to Algorithms

Spring 2020

Recitation 12

Bellman-Ford

In lecture, we presented a version of Bellman-Ford[1] based on graph duplication and DAG Relaxation that solves SSSPs in $O(|V||E|)$ time and space, and can return a negative-weight cycle reachable on a path from s to v, for any vertex v with $\delta(s, v) = -\infty$.

The original Bellman-Ford algorithm is easier to state but is a little less powerful. It solves SSSPs in the same time using only $O(|V|)$ space, but only detects whether a negative-weight cycle exists (will not return such a negative weight cycle). It is based on the relaxation framework discussed in R11. The algorithm is straight-forward: initialize distance estimates, and then relax every edge in the graph in $|V|-1$ rounds. The claim is that: if the graph does not contain negative-weight cycles, $d(s, v) = \delta(s, v)$ for all $v \in V$ at termination; otherwise if any edge still relaxable (i.e., still violates the triangle inequality), the graph contains a negative weight cycle. A Python implementation of the Bellman-Ford algorithm is given below.

```
1  def bellman_ford(Adj, w, s):          # Adj: adjacency list, w: weights, s: start
2      # initialization
3      infinity = float('inf')           # number greater than sum of all + weights
4      d =  [infinity for _ in Adj]      # shortest path estimates d(s, v)
5      parent = [None for _ in Adj]      # initialize parent pointers
6      d[s], parent[s] = 0, s            # initialize source
7      # construct shortest paths in rounds
8      V = len(Adj)                      # number of vertices
9      for k in range(V - 1):            # relax all edges in (V - 1) rounds
10         for u in range(V):            # loop over all edges (u, v)
11             for v in Adj[u]:          # relax edge from u to v
12                 try_to_relax(Adj, w, d, parent, u, v)
13     # check for negative weight cycles accessible from s
14     for u in range(V):                # Loop over all edges (u, v)
15         for v in Adj[u]:
16             if d[v] > d[u] + w(u,v):  # If edge relax-able, report cycle
17                 raise Exception('Ack! There is a negative weight cycle!')
18     return d, parent
```

This algorithm has the same overall structure as the general relaxation paradigm, but limits the order in which edges can be processed. In particular, the algorithm relaxes every edge of the graph (lines 10-12), in a series of $|V| - 1$ rounds (line 9). The following lemma establishes correctness of the algorithm.

[1]This algorithm is called **Bellman-Ford** after two researchers who independently proposed the same algorithm in different contexts.

Lemma 1 *At the end of relaxation round i of Bellman-Ford, $d(s, v) = \delta(s, v)$ for any vertex v that has a shortest path from s to v which traverses at most i edges.*

Proof. Proof by induction on round i. At the start of the algorithm (at end of round 0), the only vertex with shortest path from s traversing at most 0 edges is vertex s, and Bellman-Ford correctly sets $d(s, s) = 0 = \delta(s, s)$. Now suppose the claim is true at the end of round $i - 1$. Let v be a vertex containing a shortest path from s traversing at most i edges. If v has a shortest path from s traversing at most $i-1$ edges, $d(s, v) = \delta(s, v)$ prior to round i, and will continue to hold at the end of round i by the upper-bound property[2] Alternatively, $d(s, v) \neq \delta(s, v)$ prior to round i, and let u be the second to last vertex visited along some shortest path from s to v which traverses exactly i edges. Some shortest path from s to u traverses at most $i - 1$ edges, so $d(s, u) = \delta(s, u)$ prior to round i. Then after the edge from u to v is relaxed during round i, $d(s, v) = \delta(s, v)$ as desired. □

If the graph does not contain negative weight cycles, some shortest path is simple, and contains at most $|V|-1$ edges as it traverses any vertex of the graph at most once. Thus after $|V|-1$ rounds of Bellman-Ford, $d(s, v) = \delta(s, v)$ for every vertex with a simple shortest path from s to v. However, if after $|V| - 1$ rounds of relaxation, some edge (u, v) still violates the triangle inequality (lines 14-17), then there exists a path from s to v using $|V|$ edges which has lower weight than all paths using fewer edges. Such a path cannot be simple, so it must contain a negative weight cycle.

This algorithm runs $|V|$ rounds, where each round performs a constant amount of work for each edge in the graph, so Bellman-Ford runs in $O(|V||E|)$ time. Note that lines 10-11 actually take $O(|V| + |E|)$ time to loop over the entire adjacency list structure, even for vertices adjacent to no edge. If the graph contains isolated vertices that are not S, we can just remove them from Adj to ensure that $|V| = O(|E|)$. Note that if edges are processed in a topological sort order with respect to a shortest path tree from s, then Bellman-Ford will correctly compute shortest paths from s after its first round; of course, it is not easy to find such an order. However, for many graphs, significant savings can be obtained by stopping Bellman-Ford after any round for which no edge relaxation is modifying.

Note that this algorithm is different than the one presented in lecture in two important ways:

- The original Bellman-Ford only keeps track of one 'layer' of $d(s, v)$ estimates in each round, while the lecture version keeps track of $d_k(s, v)$ for $k \in \{0, \ldots, |V|\}$, which can be then used to construct negative-weight cycles.
- A distance estimate $d(s, v)$ in round k of original Bellman-Ford does not necessarily equal $d_k(s, v)$, the k-edge distance to v computed in the lecture version. This is because the original Bellman-Ford may relax multiple edges along a shortest path to v in a single round, while the lecture version relaxes at most one in each level. In other words, distance estimate $d(s, v)$ in round k of original Bellman-Ford is never larger than $d_k(s, v)$, but it may be much smaller and converge to a solution quicker than the lecture version, so may be faster in practice.

[2]Recall that the Safety Lemma from Recitation 11 ensures that relaxation maintains $\delta(s, v) \leq d(s, v)$ for all v.

Exercise: Alice, Bob, and Casey are best friends who live in different corners of a rural school district. During the summer, they decide to meet every Saturday at some intersection in the district to play tee-ball. Each child will bike to the meeting location from their home along dirt roads. Each dirt road between road intersections has a level of **fun** associated with biking along it in a certain direction, depending on the incline and quality of the road, the number of animals passed, etc. Road fun-ness may be positive, but could also be be negative, e.g. when a road is difficult to traverse in a given direction, or passes by a scary dog, etc. The children would like to: choose a road intersection to meet and play tee-ball that maximizes the total fun of all three children in **reaching** their chosen meeting location; or alternatively, abandon tee-ball altogether in favor of biking, if a loop of roads exists in their district along which they can bike all day with ever increasing fun. Help the children organize their Saturday routine by finding a tee-ball location, or determining that there exists a continuously fun bike loop in their district (for now, you do not have to find such a loop). You may assume that each child can reach any road in the district by bike.

Solution: Construct a graph on road intersections within the district, as well as the locations a, b, and c of the homes of the three children, with a directed edge from one vertex to another if there is a road between them traversable in that direction by bike, weighted by negative fun-ness of the road. If a negative weight cycle exists in this graph, such a cycle would represent a continuously fun bike loop. To check for the existence of any negative weight cycle in the graph, run Bellman-Ford from vertex a. If Bellman-Ford detects a negative weight cycle by finding an edge (u, v) that can be relaxed in round $|V|$, return that a continuously fun bike loop exists. Alternatively, if no negative weight cycle exists, minimal weighted paths correspond to bike routes that maximize fun. Running Bellman-Ford from vertex a then computes shortest paths $d(s, v)$ from a to each vertex v in the graph. Run Bellman-Ford two more times, once from vertex b and once from vertex c, computing shortest paths values $d(b, v)$ and $d(c, v)$ respectively for each vertex v in the graph. Then for each vertex v, compute the sum $d(a, v) + d(b, v) + d(c, v)$. A vertex that minimizes this sum will correspond to a road intersection that maximizes total fun of all three children in reaching it. This algorithm runs Bellman-Ford three times and then compares a constant sized sum at each vertex, so this algorithm runs in $O(|V||E|)$ time.

6.006 Introduction to Algorithms

Spring 2020

Recitation 13

Dijkstra's Algorithm

Dijkstra is possibly the most commonly used weighted shortest paths algorithm; it is asymptotically faster than Bellman-Ford, but only applies to graphs containing non-negative edge weights, which appear often in many applications. The algorithm is fairly intuitive, though its implementation can be more complicated than that of other shortest path algorithms. Think of a weighted graph as a network of pipes, each with non-negative length (weight). Then turn on a water faucet at a source vertex s. Assuming the water flowing from the faucet traverses each pipe at the same rate, the water will reach each pipe intersection vertex in the order of their shortest distance from the source. Dijkstra's algorithm discretizes this continuous process by repeatedly relaxing edges from a vertex whose minimum weight path estimate is smallest among vertices whose out-going edges have not yet been relaxed. In order to efficiently find the smallest minimum weight path estimate, Dijkstra's algorithm is often presented in terms of a minimum priority queue data structure. Dijkstra's running time then depends on how efficiently the priority queue can perform its supported operations. Below is Python code for Dijkstra's algorithm in terms of priority queue operations.

```
1  def dijkstra(Adj, w, s):
2      d = [float('inf') for _ in Adj]          # shortest path estimates d(s, v)
3      parent = [None for _ in Adj]             # initialize parent pointers
4      d[s], parent[s] = 0, s                   # initialize source
5      Q = PriorityQueue()                      # initialize empty priority queue
6      V = len(Adj)                             # number of vertices
7      for v in range(V):                       # loop through vertices
8          Q.insert(v, d[v])                    # insert vertex-estimate pair
9      for _ in range(V):                       # main loop
10         u = Q.extract_min()                  # extract vertex with min estimate
11         for v in Adj[u]:                     # loop through out-going edges
12             try_to_relax(Adj, w, d, parent, u, v)
13             Q.decrease_key(v, d[v])          # update key of vertex
14     return d, parent
```

This algorithm follows the same structure as the general relaxation framework. Lines 2-4 initialize shortest path weight estimates and parent pointers. Lines 5-7 initialize a priority queue with all vertices from the graph. Lines 8-12 comprise the main loop. Each time the loop is executed, line 9 removes a vertex from the queue, so the queue will be empty at the end of the loop. The vertex u processed in some iteration of the loop is a vertex from the queue whose shortest path weight estimate is smallest, from among all vertices not yet removed from the queue. Then, lines 10-11 relax the out-going edges from u as usual. However, since relaxation may reduce the shortest path weight estimate $d(s, v)$, vertex v's key in the queue must be updated (if it still exists in the queue); line 12 accomplishes this update.

Why does Dijkstra's algorithm compute shortest paths for a graph with non-negative edge weights? The key observation is that shortest path weight estimate of vertex u equals its actual shortest path weight $d(s,u) = \delta(s,u)$ when u is removed from the priority queue. Then by the upper-bound property, $d(s,u) = \delta(s,u)$ will still hold at termination of the algorithm. A proof of correctness is described in the lecture notes, and will not be repeated here. Instead, we will focus on analyzing running time for Dijkstra implemented using different priority queues.

Exercise: Construct a weighted graph with non-negative edge weights, and apply Dijkstra's algorithm to find shortest paths. Specifically list the key-value pairs stored in the priority queue after each iteration of the main loop, and highlight edges corresponding to constructed parent pointers.

Priority Queues

An important aspect of Dijkstra's algorithm is the use of a priority queue. The priority queue interface used here differs slightly from our presentation of priority queues earlier in the term. Here, a priority queue maintains a set of key-value pairs, where vertex v is a value and $d(s,v)$ is its key. Aside from empty initialization, the priority queue supports three operations: `insert(val, key)` adds a key-value pair to the queue, `extract_min()` removes and returns a value from the queue whose key is minimum, and `decrease_key(val, new_key)` which reduces the key of a given value stored in the queue to the provided `new_key`. The running time of Dijkstra depends on the running times of these operations. Specifically, if T_i, T_e, and T_d are the respective running times for inserting a key-value pair, extracting a value with minimum key, and decreasing the key of a value, the running time of Dijkstra will be:

$$T_{Dijkstra} = O(|V| \cdot T_i + |V| \cdot T_e + |E| \cdot T_d).$$

There are many different ways to implement a priority queue, achieving different running times for each operation. Probably the simplest implementation is to store all the vertices and their current shortest path estimate in a dictionary. A hash table of size $O(|V|)$ can support expected constant time $O(1)$ insertion and decrease-key operations, though to find and extract the vertex with minimum key takes linear time $O(|V|)$. If the vertices are indices into the vertex set with a linear range, then we can alternatively use a direct access array, leading to worst case $O(1)$ time insertion and decrease-key, while remaining linear $O(|V|)$ to find and extract the vertex with minimum key. In either case, the running time for Dijkstra simplifies to:

$$T_{Dict} = O(|V|^2 + |E|).$$

This is actually quite good! If the graph is dense, $|E| = \Omega(|V|^2)$, this implementation is linear in the size of the input! Below is a Python implementation of Dijkstra using a direct access array to implement the priority queue.

```
class PriorityQueue:                        # Hash Table Implementation
    def __init__(self):                     # stores keys with unique labels
        self.A = {}

    def insert(self, label, key):           # insert labeled key
        self.A[label] = key

    def extract_min(self):                  # return a label with minimum key
        min_label = None
        for label in self.A:
            if (min_label is None) or (self.A[label] < self.A[min_label].key):
                min_label = label
        del self.A[min_label]
        return min_label

    def decrease_key(self, label, key):     # decrease key of a given label
        if (label in self.A) and (key < self.A[label]):
            self.A[label] = key
```

If the graph is sparse, $|E| = O(|V|)$, we can speed things up with more sophisticated priority queue implementations. We've seen that a binary min heap can implement insertion and extract-min in $O(\log n)$ time. However, decreasing the key of a value stored in a priority queue requires finding the value in the heap in order to change its key, which naively could take linear time. However, this difficulty is easily addressed: each vertex can maintain a pointer to its stored location within the heap, or the heap can maintain a mapping from values (vertices) to locations within the heap (you were asked to do this in Problem Set 5). Either solution can support finding a given value in the heap in constant time. Then, after decreasing the value's key, one can restore the min heap property in logarithmic time by re-heapifying the tree. Since a binary heap can support each of the three operations in $O(\log |V|)$ time, the running time of Dijkstra will be:

$$T_{Heap} = O((|V| + |E|) \log |V|).$$

For sparse graphs, that's $O(|V| \log |V|)$! For graphs in between sparse and dense, there is an even more sophisticated priority queue implementation using a data structure called a **Fibonacci Heap**, which supports amortized $O(1)$ time insertion and decrease-key operations, along with $O(\log n)$ minimum extraction. Thus using a Fibonacci Heap to implement the Dijkstra priority queue leads to the following worst-case running time:

$$T_{FibHeap} = O(|V| \log |V| + |E|).$$

We won't be talking much about Fibonacci Heaps in this class, but they're theoretically useful for speeding up Dijkstra on graphs that have a number of edges asymptotically in between linear and quadratic in the number of graph vertices. You may quote the Fibonacci Heap running time bound whenever you need to argue the running time of Dijkstra when solving theory questions.

```
class Item:
    def __init__(self, label, key):
        self.label, self.key = label, key

class PriorityQueue:                        # Binary Heap Implementation
    def __init__(self):                     # stores keys with unique labels
        self.A = []
        self.label2idx = {}

    def min_heapify_up(self, c):
        if c == 0: return
        p = (c - 1) // 2
        if self.A[p].key > self.A[c].key:
            self.A[c], self.A[p] = self.A[p], self.A[c]
            self.label2idx[self.A[c].label] = c
            self.label2idx[self.A[p].label] = p
            self.min_heapify_up(p)

    def min_heapify_down(self, p):
        if p >= len(self.A): return
        l = 2 * p + 1
        r = 2 * p + 2
        if l >= len(self.A): l = p
        if r >= len(self.A): r = p
        c = l if self.A[r].key > self.A[l].key else r
        if self.A[p].key > self.A[c].key:
            self.A[c], self.A[p] = self.A[p], self.A[c]
            self.label2idx[self.A[c].label] = c
            self.label2idx[self.A[p].label] = p
            self.min_heapify_down(c)

    def insert(self, label, key):           # insert labeled key
        self.A.append(Item(label, key))
        idx = len(self.A) - 1
        self.label2idx[self.A[idx].label] = idx
        self.min_heapify_up(idx)

    def extract_min(self):                  # remove a label with minimum key
        self.A[0], self.A[-1] = self.A[-1], self.A[0]
        self.label2idx[self.A[0].label] = 0
        del self.label2idx[self.A[-1].label]
        min_label = self.A.pop().label
        self.min_heapify_down(0)
        return min_label

    def decrease_key(self, label, key):     # decrease key of a given label
        if label in self.label2idx:
            idx = self.label2idx[label]
            if key < self.A[idx].key:
                self.A[idx].key = key
                self.min_heapify_up(idx)
```

Fibonacci Heaps are not actually used very often in practice as it is more complex to implement, and results in larger constant factor overhead than the other two implementations described above. When the number of edges in the graph is known to be at most linear (e.g., planar or bounded degree graphs) or at least quadratic (e.g. complete graphs) in the number of vertices, then using a binary heap or dictionary respectively will perform as well asymptotically as a Fibonacci Heap.

We've made a JavaScript Dijkstra visualizer which you can find here:
https://codepen.io/mit6006/pen/BqqXWM

Exercise: CIA officer Mary Cathison needs to drive to meet with an informant across an unwelcome city. Some roads in the city are equipped with government surveillance cameras, and Mary will be detained if cameras from more than one road observe her car on the way to her informant. Mary has a map describing the length of each road and the locations and ranges of surveillance cameras. Help Mary find the shortest drive to reach her informant, being seen by at most one surveillance camera along the way.

Solution: Construct a graph having two vertices $(v, 0)$ and $(v, 1)$ for every road intersection v within the city. Vertex (v, i) represents arriving at intersection v having already been spotted by exactly i camera(s). For each road from intersection u to v: add two directed edges from $(u, 0)$ to $(v, 0)$ and from $(u, 1)$ to $(v, 1)$ if traveling on the road will not be visible by a camera; and add one directed edge from $(u, 0)$ to $(v, 1)$ if traveling on the road will be visible. If s is Mary's start location and t is the location of the informant, any path from $(s, 0)$ to $(t, 0)$ or $(t, 1)$ in the constructed graph will be a path visible by at most one camera. Let n be the number of road intersections and m be the number of roads in the network. Assuming lengths of roads are positive, use Dijkstra's algorithm to find the shortest such path in $O(m + n \log n)$ time using a Fibonacci Heap for Dijkstra's priority queue. Alternatively, since the road network is likely planar and/or bounded degree, it may be safe to assume that $m = O(n)$, so a binary heap could be used instead to find a shortest path in $O(n \log n)$ time.

6.006 Introduction to Algorithms

Spring 2020

Recitation 14

Single Source Shortest Paths Review

We've learned four algorithms to solve the single source shortest paths (SSSP) problem; they are listed in the table below. Then, to solve shortest paths problems, you must first define or construct a graph related to your problem, and then running an SSSP algorithm on that graph in a way that solves your problem. Generally, you will want to use the fastest SSSP algorithm that solves your problem. Bellman-Ford applies to any weighted graph but is the slowest of the four, so we prefer the other algorithms whenever they are applicable.

Restrictions		SSSP Algorithm	
Graph	Weights	Name	Running Time $O(\cdot)$
General	Unweighted	BFS	$\lvert V\rvert + \lvert E\rvert$
DAG	Any	DAG Relaxation	$\lvert V\rvert + \lvert E\rvert$
General	Non-negative	Dijkstra	$\lvert V\rvert \log \lvert V\rvert + \lvert E\rvert$
General	Any	Bellman-Ford	$\lvert V\rvert \cdot \lvert E\rvert$

We presented these algorithms with respect to the SSSP problem, but along the way, we also showed how to use these algorithms to solve other problems. For example, we can also **count connected components** in a graph using Full-DFS or Full-BFS, **topologically sort** vertices in a DAG using DFS, and **detect negative weight cycles** using Bellman-Ford.

All Pairs Shortest Paths

Given a weighted graph $G = (V, E, w)$, the (weighted) All Pairs Shortest Paths (APSP) problem asks for the minimum weight $\delta(u, v)$ of any path from u to v for every pair of vertices $u, v \in V$. To make the problem a little easier, if there exists a negative weight cycle in G, our algorithm is not required to return any output. A straight-forward way to solve this problem is to reduce to solving an SSSP problem $|V|$ times, once from each vertex in V. This strategy is actually quite good for special types of graphs! For example, suppose we want to solve ASPS on an unweighted graph that is **sparse** (i.e. $|E| = O(|V|)$). Running BFS from each vertex takes $O(|V|^2)$ time. Since we need to return a value $\delta(u, v)$ for each pair of vertices, any ASPS algorithm requires at least $\Omega(V^2)$ time, so this algorithm is optimal for graphs that are **unweighted** and **sparse**. However, for general graphs, possibly containing negative weight edges, running Bellman-Ford $|V|$ times is quite slow, $O(|V|^2|E|)$, a factor of $|E|$ larger than the output. By contrast, if we have a graph that only has non-negative weights, applying Dijkstra $|V|$ times takes $O(|V|^2 \log |V| + |V||E|)$ time. On a sparse graph, running Dijkstra $|V|$ times is only a $\log |V|$ factor larger than the output, while $|V|$ times Bellman-Ford is a linear $|V|$ factor larger. Is it possible to solve the APSP problem on general weighted graphs faster than $O(|V|^2|E|)$?

Johnson's Algorithm

The idea behind Johnson's Algorithm is to reduce the ASPS problem on a graph with **arbitrary edge weights** to the ASPS problem on a graph with **non-negative edge weights**. The algorithm does this by re-weighting the edges in the original graph to non-negative values in such a way so that shortest paths in the re-weighted graph are also shortest paths in the original graph. Then finding shortest paths in the re-weighted graph using $|V|$ times Dijkstra will solve the original problem. How can we re-weight edges in a way that preserves shortest paths? Johnson's clever idea is to assign each vertex v a real number $h(v)$, and change the weight of each edge (a, b) from $w(a, b)$ to $w'(a, b) = w(a, b) + h(a) - h(b)$, to form a new weight graph $G' = (V, E, w')$.

Claim: A shortest path $(v_1, v_2, \ldots, v_k)$ in G' is also a shortest path in G from v_1 to v_k.

Proof. Let $w(\pi) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$ be the weight of path π in G. Then weight of π in G' is:

$$\begin{aligned}\sum_{i=1}^{k-1} w'(v_i, v_{i+1}) &= \sum_{i=1}^{k-1} \Big(w(v_i, v_{i+1}) + h(v_i) - h(v_{i+1}) \Big) \\ &= \Big(\sum_{i=1}^{k-1} w(v_i, v_{i+1}) \Big) + \Big(\sum_{i=1}^{k-1} h(v_i) \Big) - \Big(\sum_{i=1}^{k-1} h(v_{i+1}) \Big) = w(\pi) + h(v_1) - h(v_k).\end{aligned}$$

So, since each path from v_1 to v_k is increased by the same number $h(v_1) - h(v_k)$, shortest paths remain shortest. □

It remains to find a vertex assignment function h, for which all edge weights $w'(a, b)$ in the modified graph are non-negative. Johnson's defines h in the following way: add a new node x to G with a directed edge from x to v for each vertex $v \in V$ to construct graph G^*, letting $h(v) = \delta(x, v)$. This assignment of h ensures that $w'(a, b) \geq 0$ for every edge (a, b).

Claim: If $h(v) = \delta(x, v)$ and $h(v)$ is finite, then $w'(a, b) = w(a, b) + h(a) - h(b) \geq 0$ for every edge $(a, b) \in E$.

Proof. The claim is equivalent to claiming $\delta(x, b) \leq w(a, b) + \delta(x, a)$ for every edge $(a, b) \in E$, i.e. the minimum weight of any path from x to b in G^* is not greater than the minimum weight of any path from x to a than traversing the edge from a to b, which is true by definition of minimum weight. (This is simply a restatement of the triangle inequality.) □

Johnson's algorithm computes $h(v) = \delta(x, v)$, negative minimum weight distances from the added node x, using Bellman-Ford. If $\delta(x, v) = -\infty$ for any vertex v, then there must be a negative weight cycle in the graph, and Johnson's can terminate as no output is required. Otherwise, Johnson's can re-weight the edges of G to $w'(a, b) = w(a, b) + h(a) - h(b) \geq 0$ into G' containing only positive edge weights. Since shortest paths in G' are shortest paths in G, we can run Dijkstra $|V|$ times on G' to find a single source shortest paths distances $\delta'(u, v)$ from each vertex u in G'. Then we can compute each $\delta(u, v)$ by setting it to $\delta'(u, v) - \delta(x, u) + \delta(x, y)$. Johnson's takes $O(|V||E|)$ time to run Bellman-Ford, and $O(|V|(|V| \log |V| + |E|))$ time to run Dijkstra $|V|$ times, so this algorithm runs in $O(|V|^2 \log |V| + |V||E|)$ time, asymptotically better than $O(|V|^2|E|)$.

6.006 Introduction to Algorithms

Spring 2020

Recitation 15

Dynamic Programming

Dynamic Programming generalizes Divide and Conquer type recurrences when subproblem dependencies form a directed acyclic graph instead of a tree. Dynamic Programming often applies to optimization problems, where you are maximizing or minimizing a single scalar value, or counting problems, where you have to count all possibilities. To solve a problem using dynamic programming, we follow the following steps as part of a recursive problem solving framework.

How to Solve a Problem Recursively (SRT BOT)

1. **Subproblem** definition subproblem $x \in X$
   - Describe the meaning of a subproblem **in words**, in terms of parameters
   - Often subsets of input: prefixes, suffixes, contiguous subsequences
   - Often record partial state: add subproblems by incrementing some auxiliary variables
2. **Relate** subproblem solutions recursively $x(i) = f(x(j), \ldots)$ for one or more $j < i$
3. **Topological order** to argue relation is acyclic and subproblems form a DAG
4. **Base** cases
   - State solutions for all (reachable) independent subproblems where relation doesn't apply/work
5. **Original problem**
   - Show how to compute solution to original problem from solutions to subproblems
   - Possibly use parent pointers to recover actual solution, not just objective function
6. **Time** analysis
   - $\sum_{x \in X} \text{work}(x)$, or if $\text{work}(x) = O(W)$ for all $x \in X$, then $|X| \cdot O(W)$
   - $\text{work}(x)$ measures **nonrecursive** work in relation; treat recursions as taking $O(1)$ time

Implementation

Once subproblems are chosen and a DAG of dependencies is found, there are two primary methods for solving the problem, which are functionally equivalent but are implemented differently.

- A **top down** approach evaluates the recursion starting from roots (vertices incident to no incoming edges). At the end of each recursive call the calculated solution to a subproblem is recorded into a memo, while at the start of each recursive call, the memo is checked to see if that subproblem has already been solved.
- A **bottom up** approach calculates each subproblem according to a topological sort order of the DAG of subproblem dependencies, also recording each subproblem solution in a memo so it can be used to solve later subproblems. Usually subproblems are constructed so that a topological sort order is obvious, especially when subproblems only depend on subproblems having smaller parameters, so performing a DFS to find this ordering is usually unnecessary.

Top down is a recursive view, while Bottom up unrolls the recursion. Both implementations are valid and often used. Memoization is used in both implementations to remember computation from previous subproblems. While it is typical to memoize all evaluated subproblems, it is often possible to remember (memoize) fewer subproblems, especially when subproblems occur in 'rounds'.

Often we don't just want the value that is optimized, but we would also like to return a path of subproblems that resulted in the optimized value. To reconstruct the answer, we need to maintain auxiliary information in addition to the value we are optimizing. Along with the value we are optimizing, we can maintain parent pointers to the subproblem or subproblems upon which a solution to the current subproblem depends. This is analogous to maintaining parent pointers in shortest path problems.

Exercise: Simplified Blackjack

We define a simplified version of the game **blackjack** between one **player** and a **dealer**. A **deck of cards** is an ordered sequence of n cards $D = (c_1, \ldots, c_n)$, where each card c_i is an integer between 1 and 10 inclusive (unlike in real blackjack, aces will always have value 1). Blackjack is played in **rounds**. In one round, the dealer will draw the top two cards from the deck (initially c_1 and c_2), then the player will draw the next two cards (initially c_3 and c_4), and then the player may either choose to draw or not draw one additional card (a hit).

The player wins the round if the **value** of the player's hand (i.e., the sum of cards drawn by the player in the round) is ≤ 21 and exceeds the value of the dealer's hand; otherwise, the player loses the round. The game ends when a round ends with fewer than 5 cards remaining in the deck. Given a deck of n cards with a **known order**, describe an $O(n)$-time algorithm to determine the maximum number of rounds the player can win by playing simplified blackjack with the deck.

Solution:

1. **Subproblems**
   - Choose suffixes
   - $x(i)$: maximum rounds player can win by playing blackjack using cards $(c_i, \ldots, c_n)$
2. **Relate**
   - Guess whether the player hits or not
   - Dealer's hand always has value $c_i + c_{i+1}$
   - Player's hand will have value either:
     - $c_{i+2} + c_{i+3}$ (no hit, 4 cards used in round), or
     - $c_{i+2} + c_{i+3} + c_{i+4}$ (hit, 5 cards used in round)
   - Let $w(d, p)$ be the round result given hand values d and p (dealer and player)
     - player win: $w(d, p) = 1$ if $d < p \leq 21$
     - player loss: $w(d, p) = 0$ otherwise (if $p \leq d$ or $21 < p$)
   - $x(i) = \max\{w(c_i+c_{i+1}, c_{i+2}+c_{i+3})+x(i+4), w(c_i+c_{i+1}, c_{i+2}+c_{i+3}+c_{i+4})+x(i+5)\}$
   - (for $n - (i-1) \geq 5$, i.e., $i \leq n-4$)
3. **Topo**
   - Subproblems $x(i)$ only depend on strictly larger i, so acyclic
4. **Base**
   - $x(n-3) = x(n-2) = x(n-1) = x(n) = x(n+1) = 0$
   - (not enough cards for another round)
5. **Original**
   - Solve $x(i)$ for $i \in \{1, \ldots, n+1\}$, via recursive top down or iterative bottom up
   - $x(1)$: the maximum rounds player can win by playing blackjack with the full deck
6. **Time**
   - # subproblems: $n + 1$
   - work per subproblem: $O(1)$
   - $O(n)$ running time

Exercise: Text Justification

Text Justification is the problem of fitting a sequence of n space separated words into a column of lines with constant width s, to minimize the amount of white-space between words. Each word can be represented by its width $w_i < s$. A good way to minimize white space in a line is to minimize **badness** of a line. Assuming a line contains words from w_i to w_j, the badness of the line is defined as $b(i, j) = (s - (w_i + \ldots + w_j))^3$ if $s > (w_i + \ldots + w_j)$, and $b(i, j) = \infty$ otherwise. A good text justification would then partition words into lines to minimize the sum total of badness over all lines containing words. The cubic power heavily penalizes large white space in a line. Microsoft Word uses a greedy algorithm to justify text that puts as many words into a line as it can before moving to the next line. This algorithm can lead to some really bad lines. LaTeX on the other hand formats text to minimize this measure of white space using a dynamic program. Describe an $O(n^2)$ algorithm to fit n words into a column of width s that minimizes the sum of badness over all lines.

Solution:

1. **Subproblems**
   - Choose suffixes as subproblems
   - $x(i)$: minimum badness sum of formatting the words from w_i to w_{n-1}
2. **Relate**
   - The first line must break at some word, so try all possibilities
   - $x(i) = \min\{b(i, j) + x(j + 1) \mid i \leq j < n\}$
3. **Topo**
   - Subproblems $x(i)$ only depend on strictly larger i, so acyclic
4. **Base**
   - $x(n) = 0$ badness of justifying zero words is zero
5. **Original**
   - Solve subproblems via recursive top down or iterative bottom up
   - Solution to original problem is $x(0)$
   - Store parent pointers to reconstruct line breaks
6. **Time**
   - # subproblems: $O(n)$
   - work per subproblem: $O(n^2)$

- $O(n^3)$ running time
- Can we do even better?

Optimization

- Computing badness $b(i, j)$ could take linear time!
- If we could pre-compute and remember each $b(i, j)$ in $O(1)$ time, then:
- work per subproblem: $O(n)$
- $O(n^2)$ running time

Pre-compute all $b(i, j)$ in $O(n^2)$, also using dynamic programming!

1. **Subproblems**
   - $\boxed{x(i, j)\text{: sum of word lengths } w_i \text{ to } w_j}$
2. **Relate**
   - $x(i, j) = \sum_k w_k$ takes $O(j - i)$ time to compute, slow!
   - $x(i, j) = x(i, j - 1) + w_j$ takes $O(1)$ time to compute, faster!
3. **Topo**
   - Subproblems $x(i, j)$ only depend on strictly smaller $j - i$, so acyclic
4. **Base**
   - $x(i, i) = w_i$ for all $0 \leq i < n$, just one word
5. **Original**
   - Solve subproblems via recursive top down or iterative bottom up
   - Compute each $b(i, j) = (s - x(i, j))^3$ in $O(1)$ time
6. **Time**
   - # subproblems: $O(n^2)$
   - work per subproblem: $O(1)$
   - $O(n^2)$ running time

6.006 Introduction to Algorithms

Spring 2020

Recitation 16

Dynamic Programming Exercises

Max Subarray Sum

Given an array A of n integers, what is the largest sum of any **nonempty** subarray?
(in this class, **subarray** always means a contiguous sequence of elements)
Example: `A = [-9, 1, -5, 4, 3, -6, 7, 8, -2]`, largest `subsum` is 16.

Solution: We could brute force in $O(n^3)$ by computing the sum of each of the $O(n^2)$ subarrays in $O(n)$ time. We can get a faster algorithm by noticing that the subarray with maximum sum must end somewhere. Finding the maximum subarray ending at a particular location k can be computed in $O(n)$ time by scanning to the left from k, keeping track of a rolling sum, and remembering the maximum along the way; since there are n ending locations, this algorithm runs in $O(n^2)$ time. We can do even faster by recognizing that each successive scan to the left is redoing work that has already been done in earlier scans. Let's use dynamic programming to reuse this work!

1. **Subproblems**
   - $x(k)$: the max subarray sum ending at $A[k]$
   - Prefix subproblem, but with condition, like Longest Increase Subsequence
   - (Exercise: reformulate in terms of suffixes instead of prefixes)
2. **Relate**
   - Maximizing subarray ending at k either uses item $k-1$ or it doesn't
   - If it doesn't, then subarray is just $A[k]$
   - Otherwise, $k-1$ is used, and we should include the maximum subarray ending at $k-1$
   - $x(k) = \max\{A[k], A[k] + x(k-1)\}$
3. **Topo. Order**
   - Subproblems $x(k)$ only depend on strictly smaller k, so acyclic
4. **Base**
   - $x(0) = A[0]$ (since subarray must be nonempty)
5. **Original**
   - Solve subproblems via recursive top down or iterative bottom up

- Solution to original is max of all subproblems, i.e., $\max\{x(k) \mid k \in \{0, \ldots, n-1\}\}$
- Subproblems are used twice: when computing the next larger, and in the final max

6. **Time**
   - # subproblems: $O(n)$
   - work per subproblem: $O(1)$
   - time to solve original problem: $O(n)$
   - $O(n)$ time in total

```python
# bottom up implementation
def max_subarray_sum(A):
    x = [None for _ in A]                    # memo
    x[0] = A[0]                              # base case
    for k in range(1, len(A)):               # iteration
        x[k] = max(A[k], A[k] + x[k - 1])    # relation
    return max(x)                            # original
```

Edit Distance

A plagiarism detector needs to detect the similarity between two texts, string A and string B. One measure of similarity is called **edit distance**, the minimum number of **edits** that will transform string A into string B. An edit may be one of three operations: delete a character of A, replace a character of A with another letter, and insert a character between two characters of A. Describe a $O(|A||B|)$ time algorithm to compute the edit distance between A and B.

Solution:

1. **Subproblems**
   - Approach will be to modify A until its last character matches B
   - $x(i, j)$: minimum number of edits to transform prefix up to $A(i)$ to prefix up to $B(j)$
   - (Exercise: reformulate in terms of suffixes instead of prefixes)
2. **Relate**
   - If $A(i) = B(j)$, then match!
   - Otherwise, need to edit to make last element of A equal to $B(j)$
   - Edit is either an insertion, replace, or deletion **(Guess!)**
   - Deletion removes $A(i)$
   - Insertion adds $B(j)$ to end of A, then removes it and $B(j)$

- Replace changes $A(i)$ to $B(j)$ and removes both $A(i)$ and $B(j)$
- $x(i,j) = \begin{cases} x(i-1,j-1) & \text{if } A(i) = B(i) \\ 1 + \min(x(i-1,j), x(i,j-1), x(i-1,j-1)) & \text{otherwise} \end{cases}$

3. **Topo. Order**
   - Subproblems $x(i,j)$ only depend on strictly smaller i and j, so acyclic
4. **Base**
   - $x(i,0) = i$, $x(0,j) = j$ (need many insertions or deletions)
5. **Original**
   - Solve subproblems via recursive top down or iterative bottom up
   - Solution to original problem is $x(|A|, |B|)$
   - (Can store parent pointers to reconstruct edits transforming A to B)
6. **Time**
   - # subproblems: $O(n^2)$
   - work per subproblem: $O(1)$
   - $O(n^2)$ running time

```python
def edit_distance(A, B):
    x = [[None] * len(A) for _ in range(len(B))]  # memo
    x[0][0] = 0                                   # base cases
    for i in range(1, len(A)):
        x[i][0] = x[i - 1][0] + 1                 # delete A[i]
    for j in range(1, len(B)):
        x[0][j] = x[0][j - 1] + 1                 # insert B[j] into A
    for i in range(1, len(A)):                    # dynamic program
        for j in range(1, len(B)):
            if A[i] == B[j]:
                x[i][j] = x[i - 1][j - 1]         # matched! no edit needed
            else:                                 # edit needed!
                ed_del = 1 + x[i - 1][j]          # delete A[i]
                ed_ins = 1 + x[i][j - 1]          # insert B[j] after A[i]
                ed_rep = 1 + x[i - 1][j - 1]      # replace A[i] with B[j]
                x[i][j] = min(ed_del, ed_ins, ed_rep)
    return x[len(A) - 1][len(B) - 1]
```

Exercise: Modify the code above to return a minimal sequence of edits to transform string A into string B. (Note, the base cases in the above code are computed individually to make reconstructing a solution easier.)

6.006 Introduction to Algorithms

Spring 2020

Recitation 17

Treasureship!

The new boardgame Treasureship is played by placing 2×1 ships within a $2 \times n$ rectangular grid. Just as in regular battleship, each 2×1 ship can be placed either horizontally or vertically, occupying exactly 2 grid squares, and each grid square may only be occupied by a single ship. Each grid square has a positive or negative integer value, representing how much treasure may be acquired or lost at that square. You may place as many ships on the board as you like, with the score of a placement of ships being the value sum of all grid squares covered by ships. Design an efficient dynamic-programming algorithm to determine a placement of ships that will maximize your total score.

Solution:

1. **Subproblems**

   - The game board has n columns of height 2 (alternatively 2 rows of width n)
   - Let $v(x, y)$ denote the grid value at row y column x, for $y \in \{1, 2\}$ and $x \in \{1, \ldots, n\}$
   - Guess how to cover the right-most square(s) in an optimal placement
   - Can either:
     - not cover,
     - place a ship to cover vertically, or
     - place a ship to cover horizontally.
   - After choosing an option, the remainder of the board may not be a rectangle
   - Right side of board looks like one of the following cases:

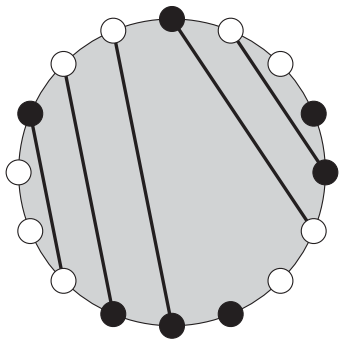
```
(0) #####   (+1) #####   (-1) ####    (+2) #####   (-2) ###    row 2
    #####        ####         #####        ###          #####  row 1
```

   - Exists optimal placement where no two ships aligned horizontally on top of each other
   - Proof: cover instead by two vertical ships next to each other!
   - So actually only need first three cases above: 0, $+1$, -1
   - Let $s(i, j)$ represent game board subset containing columns 1 to i of row 1, and columns 1 to $i + j$ of row 2, for $j \in \{0, +1, -1\}$
   - $\boxed{x(i, j)\text{: maximum score, only placing ships on board subset } s(i, j)}$
   - for $i \in \{0, \ldots, n\}$, $j \in \{0, +1, -1\}$

2. **Relate**
   - If $j = +1$, can cover right-most square with horizontal ship or leave empty
   - If $j = -1$, can cover right-most square with horizontal ship or leave empty
   - If $j = 0$, can cover column i with vertical ship or not cover one of right-most squares
   - $x(i,j) = \begin{cases} \max\{v(i,1) + v(i-1,1) + x(i-2,+1), x(i-1,0)\} & \text{if } j = -1 \\ \max\{v(i+1,2) + v(i,2) + x(i,-1), x(i,0)\} & \text{if } j = +1 \\ \max\{v(i,1) + v(i,2) + x(i-1,0), x(i,-1), x(i-1,+1)\} & \text{if } j = 0 \end{cases}$
3. **Topo**
   - Subproblems $x(i,j)$ only depend on strictly smaller $2i + j$, so acyclic.
4. **Base**
   - $s(i,j)$ contains $2i + j$ grid squares
   - $x(i,j) = 0$ if $2i + j < 2$ (can't place a ship if fewer than 2 squares!)
5. **Original**
   - Solution is $x(n,0)$, the maximum considering all grid squares.
   - Store parent pointers to reconstruct ship locations
6. **Time**
   - # subproblems: $O(n)$
   - work per subproblem $O(1)$
   - $O(n)$ running time

Wafer Power

A start-up is working on a new electronic circuit design for highly-parallel computing. Evenly-spaced along the perimeter of a circular wafer sits n ports for either a power source or a computing unit. Each computing unit needs energy from a power source, transferred between ports via a wire etched into the top surface of the wafer. However, if a computing unit is connected to a power source that is too close, the power can overload and destroy the circuit. Further, no two etched wires may cross each other. The circuit designer needs an automated way to evaluate the effectiveness of different designs, and has asked you for help. Given an arrangement of power sources and computing units plugged into the n ports, describe an $O(n^3)$-time dynamic programming algorithm to match computing units to power sources by etching non-crossing wires between them onto the surface of the wafer, in order to maximize the number of powered computing units, where wires may not connect two adjacent ports along the perimeter. Below is an example wafer, with non-crossing wires connecting computing units (white) to power sources (black).



Solution:

1. **Subproblems**

   - Let $(a_1, \ldots, a_n)$ be the ports cyclically ordered counter-clockwise around the wafer, where ports a_1 and a_n are adjacent
   - Let a_i be True if the port is a computing unit, and False if it is a power source
   - Want to match opposite ports connected by non-crossing wires
   - If match across the wafer, need to independently match ports on either side (substrings!)
   - $x(i, j)$: maximum number of matchings, restricting to ports a_k for all $k \in \{i, \ldots, j\}$
   - for $i \in \{1, \ldots, n\}$, $j \in \{i-1, \ldots, n\}$
   - $j - i + 1$ is number of ports in substring (allow $j = i - 1$ as an empty substring)

2. **Relate**
   - Guess what port to match with **first port** in substring. Either:
     - first port does not match with anything, try to match the rest;
     - first port matches a port in the middle, try to match each side independently.
   - Non-adjacency condition restricts possible matchings between i and some port t:
     - if $(i, j) = (1, n)$, can't match i with last port n or 2, so try $t \in \{3, \ldots, n-1\}$
     - otherwise, just can't match i with $i+1$, so try $t \in \{i+2, \ldots, j\}$
   - Let $m(i, j) = 1$ if $a_i \neq a_j$ and $m(i, j) = 0$ otherwise (ports of opposite type match)
   - $x(1, n) = \max\{x(2, n)\} \cup \{m(1, t) + x(2, t-1) + x(t+1, n) \mid t \in \{3, \ldots, n-1\}\}$
   - $x(i, j) = \max\{x(i+1, j)\} \cup \{m(i, t) + x(i+1, t-1) + x(t+1, j) \mid t \in \{i+2, \ldots, j\}\}$
3. **Topo**
   - Subproblems $x(i, j)$ only depend on strictly smaller $j - i$, so acyclic
4. **Base**
   - $x(i, j) = 0$ for any $j - i + 1 \in \{0, 1, 2\}$ (no match within 0, 1, or 2 adjacent ports)
5. **Original**
   - Solve subproblems via recursive top down or iterative bottom up.
   - Solution to original problem is $x(1, n)$.
   - Store parent pointers to reconstruct matching (e.g., choice of t or no match at each step)
6. **Time**
   - # subproblems: $O(n^2)$
   - work per subproblem: $O(n)$
   - $O(n^3)$ running time

6.006 Introduction to Algorithms

Spring 2020

Recitation 18: Subset Sum Variants

Subset Sum Review

- Input: Set of n positive integers $A[i]$
- Output: Is there subset $A' \subset A$ such that $\sum_{a \in A'} a = S$?
- Can solve with dynamic programming in $O(nS)$ time

Subset Sum

1. **Subproblems**
   - Here we'll try 1-indexed prefixes for comparison
   - $x(i, j)$: True if can make sum j using items 1 to i, False otherwise
2. **Relate**
   - Is last item i in a valid subset? (Guess!)
   - If yes, then try to sum to $j - A[i] \geq 0$ using remaining items
   - If no, then try to sum to j using remaining items
   - $x(i, j) = \text{OR} \left\{ \begin{array}{ll} x(i-1, j - A[i]) & \text{if } j \geq A[i] \\ x(i-1, j) & \text{always} \end{array} \right\}$
   - for $i \in \{0, \ldots, n\}, j \in \{0, \ldots, S\}$
3. **Topo**
   - Subproblems $x(i, j)$ only depend on strictly smaller i, so acyclic
4. **Base**
   - $x(i, 0) =$ True for $i \in \{0, \ldots, n\}$ (trivial to make zero sum!)
   - $x(0, j) =$ False for $j \in \{1, \ldots, S\}$ (impossible to make positive sum from empty set)
5. **Original**
   - Solve subproblems via recursive top down or iterative bottom up
   - Maximum evaluated expression is given by $x(n, S)$
6. **Time**
   - (# subproblems: $O(nS)$) $\times$ (work per subproblem $O(1)$) $= O(nS)$ running time.

Exercise: Partition - Given a set of n positive integers A, describe an algorithm to determine whether A can be partitioned into two non-intersecting subsets A_1 and A_2 of equal sum, i.e. $A_1 \cap A_2 = \emptyset$ and $A_1 \cup A_2 = A$ such that $\sum_{a \in A_1} a = \sum_{a \in A_2} a$.
Example: $A = \{1, 4, 3, 12, 19, 21, 22\}$ has partition $A_1 = \{1, 19, 21\}$, $A_2 = \{3, 4, 12, 22\}$.

Solution: Run subset sum dynamic program with same A and $S = \frac{1}{2} \sum_{a \in A} a$.

Exercise: Close Partition - Given a set of n positive integers A, describe an algorithm to find a partition of A into two non-intersecting subsets A_1 and A_2 such that the difference between their respective sums are minimized.

Solution: Run subset sum dynamic program as above, but evaluate for every $S' \in \{0, \ldots, \frac{1}{2} \sum_{a \in A} a\}$, and return the largest S' such that the subset sum dynamic program returns true. Note that this still only takes $O(nS)$ time: $O(nS)$ to compute all subproblems, and then $O(nS)$ time again to loop over the subproblems to find the max true S'.

Exercise: Can you adapt subset sum to work with negative integers?

Solution: Same as subset sum (see L19), but we allow calling subproblems with larger j. But now instead of solving $x(i, j)$ only in the range $i \in \{0, \ldots, n\}, j \in \{0, \ldots, S\}$ as in positive subset sum, we allow j to range from $j_{min} = \sum_{a \in A, a < 0} a$ (smallest possible j) to $j_{max} = \sum_{a \in A, a > 0} a$ (largest possible j).

$$x(i, j) = \text{OR}\{x(i-1, j - A[i]), x(i-1, j)\} \text{ (note } j_{min} \leq j - A[i] \leq j_{max} \text{ is always true)}$$

Subproblem dependencies are still acyclic because $x(i, j)$ only depend on strictly smaller i. Base cases are $x(0, 0) =$ True and $x(0, j) =$ False if $j \neq 0$. Running time is then proportional to number of constant work subproblems, $O(n(j_{max} - j_{min}))$.

Alternatively, you can convert to an equivalent instance of positive subset sum and solve that. Choose large number $Q > \max(|S|, \sum_{a \in A} |a|)$. Add $2Q$ to each integer in A to form A', and append the value $2Q$, $n - 1$ times to the end of A'. Every element of A' is now positive, so solve positive subset sum with $S' = S + n(2Q)$. Because $(2n - 1)Q < S' < (2n + 1)Q$, any satisfying subset will contain exactly n integers from A' since the sum of any fewer would have sum no greater than $(n-1)2Q + \sum_{a \in A} |a| < (2n-1)Q$, and sum of any more would have sum no smaller than $(n + 1)2Q - \sum_{a \in A} |a| > (2n + 1)Q$. Further, at least one integer in a satisfying subset of A' corresponds to an integer of A since S' is not divisible by $2Q$. If A' has a subset B' summing to S', then the items in A corresponding to integers in B' will comprise a nonempty subset that sums to S. Conversely, if A has a subset B that sums to S, choosing the k elements of A' corresponding the integers in B and $n - k$ of the added $2Q$ values in A' will comprise a subset B' that sums to S'.

This is an example of a **reduction**: we show how to use a black-box to solve positive subset sum to solve general subset sum. However, this reduction does lead to a weaker pseudopolynomial time bound of $O(n(S + 2nQ))$ than the modified algorithm presented above.

0-1 Knapsack

- Input: Knapsack with size S, want to fill with items each item i has size s_i and value v_i.
- Output: A subset of items (may take 0 or 1 of each) with $\sum s_i \leq S$ maximizing value $\sum v_i$
- (Subset sum same as 0-1 Knapsack when each $v_i = s_i$, deciding if total value S achievable)
- Example: Items $\{(s_i, v_i)\} = \{(6,6), (9,9), (10,12)\}, S = 15$
- Solution: Subset with max value is all items except the last one (greedy fails)

1. **Subproblems**
   - Idea: Is last item in an optimal knapsack? (Guess!)
   - If yes, get value v_i and pack remaining space $S - s_i$ using remaining items
   - If no, then try to sum to S using remaining items
   - $x(i, j)$: maximum value by packing knapsack of size j using items 1 to i
2. **Relate**
   - $x(i,j) = \max \left\{ \begin{array}{ll} v_i + x(i-1, j-s_i) & \text{if } j \geq s_i \\ x(i-1,j) & \text{always} \end{array} \right\}$
   - for $i \in \{0, \ldots, n\}, j \in \{0, \ldots, S\}$
3. **Topo**
   - Subproblems $x(i, j)$ only depend on strictly smaller i, so acyclic
4. **Base**
   - $x(i, 0) = 0$ for $i \in \{0, \ldots, n\}$ (zero value possible if no more space)
   - $x(0, j) = 0$ for $j \in \{1, \ldots, S\}$ (zero value possible if no more items)
5. **Original**
   - Solve subproblems via recursive top down or iterative bottom up
   - Maximum evaluated expression is given by $x(n, S)$
   - Store parent pointers to reconstruct items to put in knapsack
6. **Time**
   - # subproblems: $O(nS)$
   - work per subproblem $O(1)$
   - $O(nS)$ running time

Exercise: Close Partition (Alternative solution)

Solution: Given integers A, solve a 0-1 Knapsack instance with $s_i = v_i = A[i]$ and $S = \frac{1}{2}\sum_{a \in A} a$, where the subset returned will be one half of a closest partition.

Exercise: Unbounded Knapsack - Same problem as 0-1 Knapsack, except that you may take as many of any item as you like.

Solution: The O-1 Knapsack formulation works directly except for a small change in relation, where i will not be decreased if it is taken once, where the topological order strictly decreases $i+j$ with each recursive call.

$$x(i,j) = \max \left\{ \begin{array}{ll} v_i + x(i, j - s_i) & \text{if } j \geq s_i \\ x(i-1, j) & \text{always} \end{array} \right\}$$

An equivalent formulation reduces subproblems to expand work done per subproblem:

1. **Subproblems:**
   - $x(j)$: maximum value by packing knapsack of size j using the provided items
2. **Relate:**
   - $x(j) = \max\{v_i + x(j - s_i) \mid i \in \{1, \ldots, n\} \text{ and } s_i \leq j\} \cup \{0\}, \quad \text{for } j \in \{0, \ldots, S\}$
3. **Topo**
   - Subproblems $x(j)$ only depend on strictly smaller j, so acyclic
4. **Base**
   - $x(0) = 0$ (no space to pack!)
5. **Original**
   - Solve subproblems via recursive top down or iterative bottom up
   - Maximum evaluated expression is given by $x(S)$
   - Store parent pointers to reconstruct items to put in knapsack
6. **Time**
   - # subproblems: $O(S)$
   - work per subproblem $O(n)$
   - $O(nS)$ running time

We've made CoffeeScript visualizers solving subset sum and 0-1 Knapsack:
https://codepen.io/mit6006/pen/JeBvKe
https://codepen.io/mit6006/pen/VVEPod

6.006 Introduction to Algorithms

Spring 2020

Recitation 19: Complexity

0-1 Knapsack Revisited

- 0-1 Knapsack
  - Input: Knapsack with volume S, want to fill with items: item i has size s_i and value v_i.
  - Output: A subset of items (may take 0 or 1 of each) with $\sum s_i \leq S$ maximizing $\sum v_i$
  - Solvable in $O(nS)$ time via dynamic programming
- How does running time compare to input?
  - What is size of input? If numbers written in binary, input has size $O(n \log S)$ bits
  - Then $O(nS)$ runs in exponential time compared to the input
  - If numbers polynomially bounded, $S = n^{O(1)}$, then dynamic program is polynomial
  - This is called a **pseudopolynomial** time algorithm
- Is 0-1 Knapsack solvable in polynomial time when numbers not polynomially bounded?
- No if **P** $\neq$ **NP**. What does this mean? (More Computational Complexity in 6.045 and 6.046)

Decision Problems

- **Decision problem**: assignment of inputs to No (0) or Yes (1)
- Inputs are either **No instances** or **Yes instances** (i.e. satisfying instances)

Problem	Decision
s-t Shortest Path	Does a given G contain a path from s to t with weight at most d?
Negative Cycle	Does a given G contain a negative weight cycle?
Longest Path	Does a given G contain a **simple** path with weight at least d?
Subset Sum	Does a given set of integers A contain a subset with sum S?
Tetris	Can you survive a given sequence of pieces?
Chess	Can a player force a win from a given board?
Halting problem	Does a given computer program terminate for a given input?

- **Algorithm/Program**: constant length code (working on a word-RAM with $\Omega(\log n)$-bit words) to solve a problem, i.e., it produces correct output for every input and the length of the code is independent of the instance size
- Problem is **decidable** if there exists a program to solve the problem in finite time

Decidability

- Program is finite string of bits, problem is function $p : \mathbb{N} \to \{0, 1\}$, i.e. infinite string of bits
- (# of programs $|\mathbb{N}|$, countably infinite) $\ll$ (# of problems $|\mathbb{R}|$, uncountably infinite)
- (Proof by Cantor's diagonal argument, probably covered in 6.042)
- Proves that most decision problems not solvable by any program (undecidable)
- e.g. the Halting problem is undecidable (many awesome proofs in 6.045)
- Fortunately most problems we think of are algorithmic in structure and are decidable

Decidable Problem Classes

R	problems decidable in finite time	'R' comes from recursive languages
EXP	problems decidable in exponential time $2^{n^{O(1)}}$	most problems we think of are here
P	problems decidable in polynomial time $n^{O(1)}$	efficient algorithms, the focus of this class

- These sets are distinct, i.e. **P** $\subsetneq$ **EXP** $\subsetneq$ **R** (via time hierarchy theorems, see 6.045)

Nondeterministic Polynomial Time (NP)

- **P** is the set of decision problems for which there is an algorithm A such that for every instance I of size n, A on I runs in poly(n) time and solves I correctly
- **NP** is the set of decision problems for which there is an algorithm V, a "verifier", that takes as input an instance I of the problem, and a "certificate" bit string of length polynomial in the size of I, so that:
  - V always runs in time polynomial in the size of I,
  - if I is a YES-instance, then there is some certificate c so that V on input (I, c) returns YES, and
  - if I is a NO-instance, then no matter what c is given to V together with I, V will always output NO on (I, c).
- You can think of the certificate as a proof that I is a YES-instance. If I is actually a NO-instance then no proof should work.

Problem	Certificate	Verifier
s-t Shortest Path	A path P from s to t	Adds the weights on P and checks if $\leq d$
Negative Cycle	A cycle C	Adds the weights on C and checks if < 0
Longest Path	A path P	Checks if P is a **simple** path with weight at least d
Subset Sum	A set of items A'	Checks if $A' \in A$ has sum S
Tetris	Sequence of moves	Checks that the moves allow survival

- **P** $\subset$ **NP** (if you can solve the problem, the solution is a certificate)
- **Open:** Does **P** $=$ **NP**? **NP** $=$ **EXP**?
- Most people think **P** $\subsetneq$ **NP** ($\subsetneq$ **EXP**), i.e.,t generating solutions harder than checking
- If you prove either way, people will give you lots of money. ($1M Millennium Prize)
- Why do we care? If can show a problem is hardest problem in **NP**,
  then problem cannot be solved in polynomial time if **P** $\neq$ **NP**
- How do we relate difficulty of problems? Reductions!

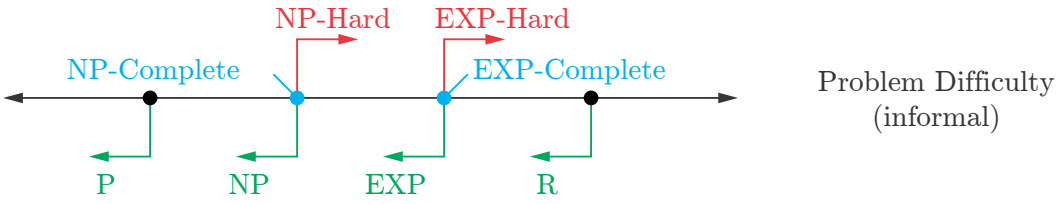
Reductions

- Suppose you want to solve problem A
- One way to solve is to convert A into a problem B you know how to solve
- Solve using an algorithm for B and use it to compute solution to A
- This is called a **reduction** from problem A to problem B ($A \to B$)
- Because B can be used to solve A, B is at least as hard ($A \leq B$)
- General algorithmic strategy: reduce to a problem you know how to solve

A	Conversion	B
Unweighted Shortest Path	Give equal weights	Weighted Shortest Path
Product Weighted Shortest Path	Logarithms	Sum Weighted Shortest Path
Sum Weighted Shortest Path	Exponents	Product Weighted Shortest Path

- Problem A is **NP-Hard** if every problem in **NP** is polynomially reducible to A
- i.e. A is at least as hard as (can be used to solve) every problem in **NP** ($X \leq A$ for $X \in$ **NP**)
- **NP-Complete = NP** $\cap$ **NP-Hard**

- All **NP-Complete** problems are equivalent, i.e. reducible to each other
- First **NP-Complete**? Every decision problem reducible to satisfying a logical circuit.
- Longest Path, Tetris are **NP-Complete**, Chess is **EXP-Complete**



0-1 Knapsack is NP-Hard

- Reduce known NP-Hard Problem to 0-1 Knapsack: **Partition**
  - Input: List of n numbers a_i
  - Output: Does there exist a partition into two sets with equal sum?
- Reduction: $s_i = v_i = a_i$, $S = \frac{1}{2}\sum a_i$
- 0-1 Knapsack at least as hard as Partition, so since Partition is **NP-Hard**, so is 0-1 Knapsack
- 0-1 Knapsack in **NP**, so also **NP-Complete**

6.006 Introduction to Algorithms

Spring 2020