

Lecture 1: Introduction

The goal of this class is to teach you to **solve** computation problems, and to **communicate** that your solutions are **correct** and **efficient**.

Problem

- Binary relation from **problem inputs** to **correct outputs**
- Usually don't specify every correct output for all inputs (too many!)
- Provide a verifiable **predicate** (a property) that correct outputs must satisfy
- 6.006 studies problems on large general input spaces
- Not general: small input instance
 - **Example:** In this room, is there a pair of students with same birthday?
- General: arbitrarily large inputs
 - **Example:** Given any set of n students, is there a pair of students with same birthday?
 - If birthday is just one of 365, for $n > 365$, answer always true by pigeon-hole
 - Assume resolution of possible birthdays exceeds n (include year, time, etc.)

Algorithm

- Procedure mapping each input to a **single** output ( deterministic)
- Algorithm **solves** a problem if it returns a correct output for every problem input
- **Example:** An algorithm to solve birthday matching
 - Maintain a **record** of names and birthdays (initially empty)
 - Interview each student in some order
 - * If birthday exists in record, return found pair!
 - * Else add name and birthday to record
 - Return None if last student interviewed without success

Correctness

- Programs/algorithms have fixed size, so how to prove correct?
- For small inputs, can use case analysis
- For arbitrarily large inputs, algorithm must be **recursive** or loop in some way
- Must use **induction** (why recursion is such a key concept in computer science)
- **Example:** Proof of correctness of birthday matching algorithm
 - Induct on k : the number of students in record
 - **Hypothesis:** if first k contain match, returns match before interviewing student $k + 1$
 - **Base case:** $k = 0$, first k contains no match
 - Assume for induction hypothesis holds for $k = k'$, and consider $k = k' + 1$
 - If first k' contains a match, already returned a match by induction
 - Else first k' do not have match, so if first $k' + 1$ has match, match contains $k' + 1$
 - Then algorithm checks directly whether birthday of student $k' + 1$ exists in first k' \square

Efficiency

- How fast does an algorithm produce a correct output?
 - Could measure time, but want performance to be machine independent
 - **Idea!** Count number of fixed-time operations algorithm takes to return
 - Expect to depend on size of input: larger input suggests longer time
 - Size of input is often called ‘ n ’, but not always!
 - Efficient if returns in **polynomial time** with respect to input
 - Sometimes no efficient algorithm exists for a problem! (See L20)
- Asymptotic Notation: ignore constant factors and low order terms
 - Upper bounds (O), lower bounds (Ω), tight bounds (Θ) $\in, =$, is, order
 - Time estimate below based on one operation per cycle on a 1 GHz single-core machine
 - Particles in universe estimated $< 10^{100}$

input	constant	logarithmic	linear	log-linear	quadratic	polynomial	exponential
n	$\Theta(1)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n^c)$	$2^{\Theta(n^c)}$
1000	1	≈ 10	1000	$\approx 10,000$	1,000,000	1000^c	$2^{1000} \approx 10^{301}$
Time	1 ns	10 ns	$1\mu\text{s}$	$10\mu\text{s}$	1 ms	10^{3c-9} s	10^{281} millenia

Model of Computation

- Specification for what operations on the machine can be performed in $O(1)$ time
- Model in this class is called the **Word-RAM**
- **Machine word:** block of w bits (w is word size of a w -bit Word-RAM)
- **Memory:** Addressable sequence of machine words
- **Processor** supports many **constant time** operations on a $O(1)$ number of words (**integers**):
 - **integer** arithmetic: $(+, -, *, //, \%)$
 - **logical** operators: $(\&\&, ||, !, ==, <, >, \leq, \geq)$
 - **(bitwise** arithmetic: $(\&, |, \ll, \gg, \dots)$)
 - Given word a , can **read** word at address a , **write** word to address a
- Memory address must be able to access every place in memory
 - Requirement: $w \geq \#$ bits to represent largest memory address, i.e., $\log_2 n$
 - 32-bit words \rightarrow max ~ 4 GB memory, 64-bit words \rightarrow max ~ 16 exabytes of memory
- **Python** is a more complicated model of computation, implemented on a Word-RAM

Data Structure

- A **data structure** is a way to store non-constant data, that supports a set of operations
- A collection of operations is called an **interface**
 - **Sequence:** Extrinsic order to items (first, last, n th)
 - **Set:** Intrinsic order to items (queries based on item keys)
- Data structures may implement the same interface with different performance
- **Example: Static Array** - fixed width slots, fixed length, static sequence interface
 - `StaticArray(n)`: allocate static array of size n initialized to 0 in $\Theta(n)$ time
 - `StaticArray.get_at(i)`: return word stored at array index i in $\Theta(1)$ time
 - `StaticArray.set_at(i, x)`: write word x to array index i in $\Theta(1)$ time
- Stored word can hold the address of a larger object
- Like Python `tuple` plus `set_at(i, x)`, Python `list` is a **dynamic array** (see L02)

```

1 def birthday_match(students):
2     """
3         Find a pair of students with the same birthday
4         Input: tuple of student (name, bday) tuples
5         Output: tuple of student names or None
6     """
7     n = len(students)                      # O(1)
8     record = StaticArray(n)                 # O(n)
9     for k in range(n):                     # n
10        (name1, bday1) = students[k]        # O(1)
11        # Return pair if bday1 in record
12        for i in range(k):                # k
13            (name2, bday2) = record.get_at(i) # O(1)
14            if bday1 == bday2:              # O(1)
15                return (name1, name2)       # O(1)
16            record.set_at(k, (name1, bday1)) # O(1)
17    return None                           # O(1)

```

Example: Running Time Analysis

- Two loops: outer $k \in \{0, \dots, n - 1\}$, inner is $i \in \{0, \dots, k\}$
- Running time is $O(n) + \sum_{k=0}^{n-1}(O(1) + k \cdot O(1)) = O(n^2)$
- Quadratic in n is **polynomial**. Efficient? Use different data structure for record!

How to Solve an Algorithms Problem

1. Reduce to a problem you already know (use data structure or algorithm)

Search Problem (Data Structures)	Sort Algorithms	Shortest Path Algorithms
Static Array (L01)	Insertion Sort (L03)	Breadth First Search (L09)
Linked List (L02)	Selection Sort (L03)	DAG Relaxation (L11)
Dynamic Array (L02)	Merge Sort (L03)	Depth First Search (L10)
Sorted Array (L03)	Counting Sort (L05)	Topological Sort (L10)
Direct-Access Array (L04)	Radix Sort (L05)	Bellman-Ford (L12)
Hash Table (L04)	AVL Sort (L07)	Dijkstra (L13)
Balanced Binary Tree (L06-L07)	Heap Sort (L08)	Johnson (L14)
Binary Heap (L08)		Floyd-Warshall (L18)

2. Design your own (recursive) algorithm

- Brute Force
- Decrease and Conquer
- Divide and Conquer
- **Dynamic Programming** (L15-L19)
- Greedy / Incremental

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

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. \square

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 below¹. 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.

¹Note 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 $\{0, \dots, 2^w - 1\}$. 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 memory². 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).

²For 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 $(8\text{-bits}) \times (2^{32}) \approx 4 \text{ GB}$.

```

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 time³. 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.

³This 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 $\log_{6006} \left((\log(n^{\sqrt{n}}))^2 \right)$.

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

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

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, $2^{2^{n+1}} = (2^{2^n})^2$, 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 $n \rightarrow \infty$, and this is equivalent to arguing that the **log** of this expression approaches ∞ :

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

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.

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

Lecture 2: Data Structures

Data Structure Interfaces

- A **data structure** is a way to store data, with algorithms that support **operations** on the data
- Collection of supported operations is called an **interface** (also API or ADT)
- Interface is a **specification**: what operations are supported (the problem!)
- Data structure is a **representation**: how operations are supported (the solution!)
- In this class, two main interfaces: **Sequence** and **Set**

Sequence Interface (L02, L07)

- Maintain a sequence of items (order is **extrinsic**)
- Ex: $(x_0, x_1, x_2, \dots, x_{n-1})$ (zero indexing)
- (use n to denote the number of items stored in the data structure)
- Supports sequence operations:

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

- Special case interfaces:

stack | `insert_last(x)` and `delete_last()`
queue | `insert_last(x)` and `delete_first()`

Set Interface (L03-L08)

- Sequence about **extrinsic** order, set is about **intrinsic** order
- Maintain a set of items having **unique keys** (e.g., item x has key $x.key$)
- (Set or multi-set? We restrict to unique keys for now.)
- Often we let key of an item be the item itself, but may want to store more info than just key
- Supports set operations:

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

- Special case interfaces:
dictionary | set without the Order operations
- In recitation, you will be asked to implement a Set, given a Sequence data structure.

Array Sequence

- Array is great for static operations! `get_at(i)` and `set_at(i, x)` in $\Theta(1)$ time!
- But not so great at dynamic operations...
- (For consistency, we maintain the invariant that array is full)
- Then inserting and removing items requires:
 - reallocating the array
 - shifting all items after the modified item

Data Structure	Operation, Worst Case $O(\cdot)$				
	Container	Static	Dynamic		
		<code>build(x)</code>	<code>get_at(i)</code> <code>set_at(i, x)</code>	<code>insert_first(x)</code> <code>delete_first()</code>	<code>insert_last(x)</code> <code>delete_last()</code>
Array		n	1	n	n

Linked List Sequence

- Pointer data structure (this is **not** related to a Python “list”)
- Each item stored in a **node** which contains a pointer to the next node in sequence
- Each node has two fields: `node.item` and `node.next`
- Can manipulate nodes simply by relinking pointers!
- Maintain pointers to the first node in sequence (called the head)
- Can now insert and delete from the front in $\Theta(1)$ time! Yay!
- (Inserting/deleting efficiently from back is also possible; you will do this in PS1)
- But now `get_at(i)` and `set_at(i, x)` each take $O(n)$ time... :(
- Can we get the best of both worlds? Yes! (Kind of...)

Data Structure	Operation, Worst Case $O(\cdot)$				
	Container	Static	Dynamic		
		<code>build(X)</code>	<code>get_at(i)</code>	<code>insert_first(x)</code>	<code>insert_last(x)</code>
Linked List		n	n	1	n
					n

Dynamic Array Sequence

- Make an array efficient for **last** dynamic operations
- Python “list” is a dynamic array
- **Idea!** Allocate extra space so reallocation does not occur with every dynamic operation
- **Fill ratio:** $0 \leq r \leq 1$ the ratio of items to space
- Whenever array is full ($r = 1$), allocate $\Theta(n)$ extra space at end to fill ratio r_i (e.g., $1/2$)
- Will have to insert $\Theta(n)$ items before the next reallocation
- A single operation can take $\Theta(n)$ time for reallocation
- However, any sequence of $\Theta(n)$ operations takes $\Theta(n)$ time
- So each operation takes $\Theta(1)$ time “on average”

Amortized Analysis

- Data structure analysis technique to distribute cost over many operations
- Operation has **amortized cost** $T(n)$ if k operations cost at most $\leq kT(n)$
- “ $T(n)$ amortized” roughly means $T(n)$ “on average” over many operations
- Inserting into a dynamic array takes **$\Theta(1)$ amortized time**
- More amortization analysis techniques in 6.046!

Dynamic Array Deletion

- Delete from back? $\Theta(1)$ time without effort, yay!
- However, can be very wasteful in space. Want size of data structure to stay $\Theta(n)$
- **Attempt:** if very empty, resize to $r = 1$. Alternating insertion and deletion could be bad...
- **Idea!** When $r < r_d$, resize array to ratio r_i where $r_d < r_i$ (e.g., $r_d = 1/4$, $r_i = 1/2$)
- Then $\Theta(n)$ cheap operations must be made before next expensive resize
- Can limit extra space usage to $(1 + \varepsilon)n$ for any $\varepsilon > 0$ (set $r_d = \frac{1}{1+\varepsilon}$, $r_i = \frac{r_d+1}{2}$)
- Dynamic arrays only support dynamic **last** operations in $\Theta(1)$ time
- Python List `append` and `pop` are amortized $O(1)$ time, other operations can be $O(n)$!
- (Inserting/deleting efficiently from front is also possible; you will do this in PS1)

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

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

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	<code>build(x)</code> <code>len()</code>	given an iterable <code>x</code> , build sequence from items in <code>x</code> return the number of stored items
Static	<code>iter_seq()</code> <code>get_at(i)</code> <code>set_at(i, x)</code>	return the stored items one-by-one in sequence order return the i^{th} item replace the i^{th} item with x
Dynamic	<code>insert_at(i, x)</code> <code>delete_at(i)</code> <code>insert_first(x)</code> <code>delete_first()</code> <code>insert_last(x)</code> <code>delete_last()</code>	add x as the i^{th} item remove and return the i^{th} item add x as the first item remove and return the first item add x as the last item 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	<code>build(x)</code> <code>len()</code>	given an iterable <code>x</code> , build set from items in <code>x</code> return the number of stored items
Static	<code>find(k)</code>	return the stored item with key <code>k</code>
Dynamic	<code>insert(x)</code> <code>delete(k)</code>	add <code>x</code> to set (replace item with key <code>x.key</code> if one already exists) remove and return the stored item with key <code>k</code>
Order	<code>iter_ord()</code> <code>find_min()</code> <code>find_max()</code> <code>find_next(k)</code> <code>find_prev(k)</code>	return the stored items one-by-one in key order return the stored item with smallest key return the stored item with largest key return the stored item with smallest key larger than <code>k</code> return the stored item with largest key smaller than <code>k</code>

(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	Container	Operation, Worst Case $O(\cdot)$			
		Static	Dynamic		
		build(x)	get_at(i) set_at(i, x)	insert_first(x) delete_first()	insert_last(x) delete_last()
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.

```

1  class Array_Seq:
2      def __init__(self):                                     # O(1)
3          self.A = []
4          self.size = 0
5
6      def __len__(self):         return self.size           # O(1)
7      def __iter__(self):       yield from self.A          # O(n) iter_seq
8
9      def build(self, X):                           # O(n)
10         self.A = [a for a in X]    # pretend this builds a static array
11         self.size = len(self.A)
12
13     def get_at(self, i):        return self.A[i]          # O(1)
14     def set_at(self, i, x):    self.A[i] = x             # O(1)
15
16     def _copy_forward(self, i, n, A, j):                 # O(n)
17         for k in range(n):
18             A[j + k] = self.A[i + k]
19
20     def _copy_backward(self, i, n, A, j):                 # O(n)
21         for k in range(n - 1, -1, -1):
22             A[j + k] = self.A[i + k]
23
24     def insert_at(self, i, x):                           # O(n)
25         n = len(self)
26         A = [None] * (n + 1)
27         self._copy_forward(0, i, A, 0)
28         A[i] = x
29         self._copy_forward(i, n - i, A, i + 1)
30         self.build(A)
31
32     def delete_at(self, i):                            # O(n)
33         n = len(self)
34         A = [None] * (n - 1)
35         self._copy_forward(0, i, A, 0)
36         x = self.A[i]
37         self._copy_forward(i + 1, n - i - 1, A, i)
38         self.build(A)
39         return x                                      # O(n)
40
41     def insert_first(self, x):   self.insert_at(0, x)
42     def delete_first(self):    return self.delete_at(0)
43     def insert_last(self, x):  self.insert_at(len(self), x)
44     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.

```

1  class Linked_List_Node:
2      def __init__(self, x):                                # O(1)
3          self.item = x
4          self.next = None
5
6      def later_node(self, i):                                # O(i)
7          if i == 0: return self
8          assert self.next
9          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.

```

1  class Linked_List_Seq:
2      def __init__(self):                                     # O(1)
3          self.head = None
4          self.size = 0
5
6      def __len__(self):   return self.size                   # O(1)
7
8      def __iter__(self):                                     # O(n) iter_seq
9          node = self.head
10         while node:
11             yield node.item
12             node = node.next
13
14     def build(self, X):                                    # O(n)
15         for a in reversed(X):
16             self.insert_first(a)
17
18     def get_at(self, i):                                   # O(i)
19         node = self.head.later_node(i)
20         return node.item
21

```

```

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):

```
1 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.

```
1  class Dynamic_Array_Seq(Array_Seq):
2      def __init__(self, r = 2):                      # O(1)
3          super().__init__()
4          self.size = 0
5          self.r = r
6          self._compute_bounds()
7          self._resize(0)
8
9      def __len__(self):    return self.size           # O(1)
10
11     def __iter__(self)::
12         for i in range(len(self)): yield self.A[i]
13
14     def build(self, X):                           # O(n)
15         for a in X: self.insert_last(a)
16
17     def _compute_bounds(self):                     # O(1)
18         self.upper = len(self.A)
19         self.lower = len(self.A) // (self.r * self.r)
20
21     def _resize(self, n):                         # O(1) or O(n)
22         if (self.lower < n < self.upper): return
23         m = max(n, 1) * self.r
24         A = [None] * m
25         self._copy_forward(0, self.size, A, 0)
26         self.A = A
27         self._compute_bounds()
28
29     def insert_last(self, x):                     # O(1)a
30         self._resize(self.size + 1)
31         self.A[self.size] = x
32         self.size += 1
33
34     def delete_last(self):                        # O(1)a
35         self.A[self.size - 1] = None
36         self.size -= 1
37         self._resize(self.size)
38
39     def insert_at(self, i, x):                   # O(n)
40         self.insert_last(None)
41         self._copy_backward(i, self.size - (i + 1), self.A, i + 1)
42         self.A[i] = x
43
44     def delete_at(self, i):                      # O(n)
45         x = self.A[i]
46         self._copy_forward(i + 1, self.size - (i + 1), self.A, i)
47         self.delete_last()
48         return x                                # O(n)
49
50     def insert_first(self, x):    self.insert_at(0, x)
51     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:

```

1  def Set_from_Seq(seq):
2      class set_from_seq:
3          def __init__(self):    self.S = seq()
4          def __len__(self):   return len(self.S)
5          def __iter__(self):  yield from self.S
6
7          def build(self, A):
8              self.S.build(A)
9
10         def insert(self, x):
11             for i in range(len(self.S)):
12                 if self.S.get_at(i).key == x.key:
13                     self.S.set_at(i, x)
14                     return
15             self.S.insert_last(x)
16

```

```
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
```

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

Problem Session 1

Problem 1-1. Asymptotic behavior of functions

For each of the following sets of five functions, order them so that if f_a appears before f_b in your sequence, then $f_a = O(f_b)$. If $f_a = O(f_b)$ and $f_b = O(f_a)$ (meaning f_a and f_b could appear in either order), indicate this by enclosing f_a and f_b in a set with curly braces. For example, if the functions are:

$$f_1 = n, \quad f_2 = \sqrt{n}, \quad f_3 = n + \sqrt{n},$$

the correct answers are $(f_2, \{f_1, f_3\})$ or $(f_2, \{f_3, f_1\})$.

a)	d)
$f_1 = (\log n)^{2019}$	$f_1 = 2^n$
$f_2 = n^2 \log(n^{2019})$	$f_2 = n^3$
$f_3 = n^3$	$f_3 = \binom{n}{n/2}$
$f_4 = 2.019^n$	$f_4 = n!$
$f_5 = n \log n$	$f_5 = \binom{n}{3}$

Solution:

- a. $(f_1, f_5, f_2, f_3, f_4)$. This order follows directly from the claim in R01 that $(\log n)^a = o(n^b)$ for all positive constants a and b , as well as standard logarithm and exponentiation manipulations.
- b. $(\{f_2, f_5\}, f_3, f_1, f_4)$. This order follows from the definition of the binomial coefficient and Stirling's approximation. The trickiest one is $f_3 = \Theta(2^n/\sqrt{n})$ (by repeated use of Stirling), which grows slower than f_1 .

Problem 1-2. Given a data structure D supporting the four first/last sequence operations:

$D.\text{insert_first}(x)$, $D.\text{delete_first}()$, $D.\text{insert_last}(x)$, $D.\text{delete_last}()$, each in $O(1)$ time, describe algorithms to implement the following higher-level operations in terms of the lower-level operations. Recall that `delete` operations return the deleted item.

- (a) `swap_ends(D)`: Swap the first and last items in the sequence in $O(1)$ time.

Solution: Swapping the first and last items in the list can be performed by simply deleting both ends in $O(1)$ time, and then inserting them back in the opposite order, also in $O(1)$ time. This algorithm is correct by the definitions of these operations.

```

1 def swap_ends(D):
2     x_first = D.delete_first()
3     x_last = D.delete_last()
4     D.insert_first(x_last)
5     D.insert_last(x_first)

```

- (b) `shift_left(D, k)`: Move the first k items in order to the end of the sequence in D in $O(k)$ time. (After, the k^{th} item should be last and the $(k + 1)^{\text{st}}$ item should be first.)

Solution: To implement `shift_left(D, 1)`, delete the first item and insert it into the last position in $O(1)$ time. The list maintains the relative ordering of all items in the sequence, except has moved the first item behind all the others, so `shift_left(D, 1)` is correct. Then to implement `shift_left(D, k)`, move the first item to the last position as above, and then recursively call `shift_left(D, k - 1)` until reaching base case `shift_left(D, 1)`. By induction, if `shift_left(D, k - 1)` is correct, moving the first item to the last position restores correctness. `shift_left(D, k)` runs in $O(k)$ time because it makes $O(k)$ recursive calls until reaching the base case, doing constant work per call.

```

1 def shift_left(D, k):
2     if (k < 1) or (k > len(D) - 1):
3         return
4     x = D.delete_first()
5     D.insert_last(x)
6     shift_left(D, k - 1)

```

Problem 1-3. Double-Ended Sequence Operations

A dynamic array can implement a Sequence interface supporting worst-case $O(1)$ -time indexing as well as insertion and removal of items at the back of the array in amortized constant time. However, insertion and deletion at the front of a dynamic array are not efficient as every entry must be shifted to maintain the sequential order of entries, taking linear time.

On the other hand, a linked-list data structure can be made to support insertion and removal operations at both ends in worst-case $O(1)$ time (see PS1), but at the expense of linear-time indexing.

Show that we can have the best of both worlds: design a data structure to store a sequence of items that supports **worst-case** $O(1)$ -time index lookup, as well as **amortized** $O(1)$ -time insertion and removal at both ends. Your data structure should use $O(n)$ space to store n items.

Solution: There are many possible solutions. One solution uses two-stacks to implement the deque, where care needs to be taken when popping from an empty stack, or pushing to a full stack. An alternative approach would be to store the queued items in the middle of an array rather than at the front, leaving a linear number of extra slots at both the beginning and end whenever rebuilding occurs, guaranteeing that linear time rebuilding only occurs once every $\Omega(n)$ operations.

For example, whenever reallocating space to store a sequence of n elements, copy them to the middle of a length $m = 3n$ array. To insert or remove an item to the beginning or end of the sequence, add or remove an element at the appropriate end in constant time. If no free slot is exists during an insertion, at least a linear number of insertions must have happened since the last rebuild, so we can afford to rebuild the array. If removing an item brings the ratio n/m of items to array size to below $1/6$, at least $m/6 = \Omega(n)$ removals must have occurred since the last rebuild, so we can again afford to rebuild the array.

A linear number of operations between expensive linear time rebuilds ensures that each dynamic operation takes at most amortized $O(1)$ time. To support array indexing in constant time, we maintain the index location i of the left-most item in the array and the number of items n stored in the array, both of which can be maintained in worst-case constant time per update. To access the j^{th} item stored in the queue sequence using zero-indexing, confirm that $i + j < n$ and return the item at index $i + j$ of the array container in worst-case constant time.

Problem 1-4. Jen & Berry's

Jen drives her ice cream truck to her local elementary school at recess. All the kids rush to line up in front of her truck. Jen is overwhelmed with the number of students (there are $2n$ of them), so she calls up her associate, Berry, to bring his ice cream truck to help her out. Berry soon arrives and parks at the other end of the line of students. He offers to sell to the last student in line, but the other students revolt in protest: “The last student was last! This is unfair!”

The students decide that the fairest way to remedy the situation would be to have the back half of the line (the n kids furthest from Jen) reverse their order and queue up at Berry's truck, so that the last kid in the original line becomes the last kid in Berry's line, with the $(n + 1)^{\text{st}}$ kid in the original line becoming Berry's first customer.

- (a)** Given a linked list containing the names of the $2n$ kids, in order of the original line formed in front of Jen's truck (where the first node contains the name of the first kid in line), describe an $O(n)$ -time algorithm to modify the linked list to reverse the order of the last half of the list. Your algorithm should not make any new linked list nodes or instantiate any new non-constant-sized data structures during its operation.

Solution: Reverse the order of the last half of the nodes in a list in three stages:

- find the n^{th} node a in the sequence (the end of Jen's line)
- for each node x from the $(n + 1)^{\text{st}}$ node b to the $(2n)^{\text{th}}$ node c , change the next pointer of x to point to the node before it in the original sequence
- change the next pointer of a and b to point to c and nothing respectively

Finding the n^{th} node requires traversing next pointers $n - 1$ times from the head of the list, which can be done in $O(n)$ time via a simple loop. We can compute n by halving the size of the list (which is guaranteed to be even).

To change the next pointers of the last half of the sequence, we can maintain pointers to the current node x and the node before it x_p , initially b and a respectively. Then, record the node x_n after x , relink x to point to the x_p , the node before x in $O(1)$ time. Then we can change the current node to x_n and the node before it to x , maintaining the desired properties for the next node to relink. Repeating n times, relinks all n nodes in the last half of the sequence in $O(n)$ time.

Lastly, by remembering nodes a , b , and c while the algorithm traverses the list, means that changing the exceptional next pointers at the front and back of the last half of the list takes $O(1)$, leading to an $O(n)$ time algorithm overall.

(b) Write a Python function `reorder_students(L)` that implements your algorithm.

Solution:

```
1 def reorder_students(L):
2     n = len(L) // 2           # find the n-th node
3     a = L.head
4     for _ in range(n - 1):
5         a = a.next
6     b = a.next               # relink next pointers of last half
7     x_p, x, x_p = a, b
8     for _ in range(n):
9         x_n = x.next
10        x.next = x_p
11        x_p, x = x, x_n
12    c = x_p
13    a.next = c               # relink front and back of last half
14    b.next = None
15    return
```

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

Lecture 3: Sorting

Set Interface (L03-L08)

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

- Storing items in an array in arbitrary order can implement a (not so efficient) set
- Stored items sorted increasing by key allows:
 - faster find min/max (at first and last index of array)
 - faster finds via binary search: $O(\log n)$

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

- But how to construct a sorted array efficiently?

Sorting

- Given a sorted array, we can leverage binary search to make an efficient set data structure.
- Input:** (static) array A of n numbers
- Output:** (static) array B which is a sorted permutation of A
 - Permutation:** array with same elements in a different order
 - Sorted:** $B[i - 1] \leq B[i]$ for all $i \in \{1, \dots, n\}$
- Example: $[8, 2, 4, 9, 3] \rightarrow [2, 3, 4, 8, 9]$
- A sort is **destructive** if it overwrites A (instead of making a new array B that is a sorted version of A)
- A sort is **in place** if it uses $O(1)$ extra space (implies destructive: in place \subseteq destructive)

Permutation Sort

- There are $n!$ permutations of A , at least one of which is sorted
- For each permutation, check whether sorted in $\Theta(n)$
- Example: $[2, 3, 1] \rightarrow \{[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]\}$

```

1 def permutation_sort(A):
2     '''Sort A'''
3     for B in permutations(A):           # O(n!)
4         if is_sorted(B):               # O(n)
5             return B                 # O(1)

```

- permutation_sort analysis:
 - Correct by case analysis: try all possibilities (Brute Force)
 - Running time: $\Omega(n! \cdot n)$ which is **exponential** :(

Solving Recurrences

- Substitution:** Guess a solution, replace with representative function, recurrence holds true
- Recurrence Tree:** Draw a tree representing the recursive calls and sum computation at nodes
- Master Theorem:** A formula to solve many recurrences (R03)

Selection Sort

- Find a largest number in prefix $A[:i + 1]$ and swap it to $A[i]$
- Recursively sort prefix $A[:i]$
- Example: $[8, 2, 4, 9, 3], [8, 2, 4, 3, 9], [3, 2, 4, 8, 9], [3, 2, 4, 8, 9], [2, 3, 4, 8, 9]$

```

1 def selection_sort(A, i = None):          # T(i)
2     '''Sort A[:i + 1]'''
3     if i is None: i = len(A) - 1           # O(1)
4     if i > 0:                           # O(1)
5         j = prefix_max(A, i)             # S(i)
6         A[i], A[j] = A[j], A[i]          # O(1)
7         selection_sort(A, i - 1)         # T(i - 1)
8
9 def prefix_max(A, i):                   # S(i)
10    '''Return index of maximum in A[:i + 1]'''
11    if i > 0:                           # O(1)
12        j = prefix_max(A, i - 1)         # S(i - 1)
13        if A[i] < A[j]:                # O(1)
14            return j                     # O(1)
15        return i                       # O(1)

```

- `prefix_max` analysis:
 - Base case: for $i = 0$, array has one element, so index of max is i
 - Induction: assume correct for i , maximum is either the maximum of $A[:i]$ or $A[i]$, returns correct index in either case. \square
 - $S(1) = \Theta(1), S(n) = S(n - 1) + \Theta(1)$
 - * Substitution: $S(n) = \Theta(n), cn = \Theta(1) + c(n - 1) \implies 1 = \Theta(1)$
 - * Recurrence tree: chain of n nodes with $\Theta(1)$ work per node, $\sum_{i=0}^{n-1} 1 = \Theta(n)$
- `selection_sort` analysis:
 - Base case: for $i = 0$, array has one element so is sorted
 - Induction: assume correct for i , last number of a sorted output is a largest number of the array, and the algorithm puts one there; then $A[:i]$ is sorted by induction \square
 - $T(1) = \Theta(1), T(n) = T(n - 1) + \Theta(n)$
 - * Substitution: $T(n) = \Theta(n^2), cn^2 = \Theta(n) + c(n - 1)^2 \implies c(2n - 1) = \Theta(n)$
 - * Recurrence tree: chain of n nodes with $\Theta(i)$ work per node, $\sum_{i=0}^{n-1} i = \Theta(n^2)$

Insertion Sort

- Recursively sort prefix $A[:i]$
- Sort prefix $A[:i + 1]$ assuming that prefix $A[:i]$ is sorted by repeated swaps
- Example: $[8, 2, 4, 9, 3], [2, 8, 4, 9, 3], [2, 4, 8, 9, 3], [2, 4, 8, 9, 3], [2, 3, 4, 8, 9]$

```

1 def insertion_sort(A, i = None):          # T(i)
2     '''Sort A[:i + 1]'''
3     if i is None: i = len(A) - 1           # O(1)
4     if i > 0:                           # O(1)
5         insertion_sort(A, i - 1)          # T(i - 1)
6         insert_last(A, i)                # S(i)
7
8 def insert_last(A, i):                   # S(i)
9     '''Sort A[:i + 1] assuming sorted A[:i]'''
10    if i > 0 and A[i] < A[i - 1]:        # O(1)
11        A[i], A[i - 1] = A[i - 1], A[i]   # O(1)
12        insert_last(A, i - 1)            # S(i - 1)

```

- `insert_last` analysis:
 - Base case: for $i = 0$, array has one element so is sorted
 - Induction: assume correct for i , if $A[i] \geq A[i - 1]$, array is sorted; otherwise, swapping last two elements allows us to sort $A[:i]$ by induction \square
 - $S(1) = \Theta(1), S(n) = S(n - 1) + \Theta(1) \implies S(n) = \Theta(n)$
- `insertion_sort` analysis:
 - Base case: for $i = 0$, array has one element so is sorted
 - Induction: assume correct for i , algorithm sorts $A[:i]$ by induction, and then `insert_last` correctly sorts the rest as proved above \square
 - $T(1) = \Theta(1), T(n) = T(n - 1) + \Theta(n) \implies T(n) = \Theta(n^2)$

Merge Sort

- Recursively sort first half and second half (may assume power of two)
- Merge sorted halves into one sorted list (two finger algorithm)
- Example: $[7, 1, 5, 6, 2, 4, 9, 3], [1, 7, 5, 6, 2, 4, 3, 9], [1, 5, 6, 7, 2, 3, 4, 9], [1, 2, 3, 4, 5, 6, 7, 9]$

```

1  def merge_sort(A, a = 0, b = None):                      # T(b - a = n)
2      '''Sort A[a:b]'''                                     # O(1)
3      if b is None: b = len(A)                             # O(1)
4      if 1 < b - a:                                       # O(1)
5          c = (a + b + 1) // 2                            # T(n / 2)
6          merge_sort(A, a, c)                           # T(n / 2)
7          merge_sort(A, c, b)                           # T(n / 2)
8          L, R = A[a:c], A[c:b]                         # O(n)
9          merge(L, R, A, len(L), len(R), a, b)           # S(n)
10
11 def merge(L, R, A, i, j, a, b):                         # S(b - a = n)
12     '''Merge sorted L[:i] and R[:j] into A[a:b]'''       # O(1)
13     if a < b:                                         # O(1)
14         if (j <= 0) or (i > 0 and L[i - 1] > R[j - 1]): # O(1)
15             A[b - 1] = L[i - 1]                          # O(1)
16             i = i - 1                                  # O(1)
17         else:                                         # O(1)
18             A[b - 1] = R[j - 1]                          # O(1)
19             j = j - 1                                  # O(1)
20     merge(L, R, A, i, j, a, b - 1)                      # S(n - 1)

```

- merge analysis:
 - Base case: for $n = 0$, arrays are empty, so vacuously correct
 - Induction: assume correct for n , item in $A[r]$ must be a largest number from remaining prefixes of L and R , and since they are sorted, taking largest of last items suffices; remainder is merged by induction \square
 - $S(0) = \Theta(1), S(n) = S(n - 1) + \Theta(1) \implies S(n) = \Theta(n)$
- merge_sort analysis:
 - Base case: for $n = 1$, array has one element so is sorted
 - Induction: assume correct for $k < n$, algorithm sorts smaller halves by induction, and then `merge` merges into a sorted array as proved above. \square
 - $T(1) = \Theta(1), T(n) = 2T(n/2) + \Theta(n)$
 - * Substitution: Guess $T(n) = \Theta(n \log n)$
 $cn \log n = \Theta(n) + 2c(n/2) \log(n/2) \implies cn \log(2) = \Theta(n)$
 - * Recurrence Tree: complete binary tree with depth $\log_2 n$ and n leaves, level i has 2^i nodes with $O(n/2^i)$ work each, total: $\sum_{i=0}^{\log_2 n} (2^i)(n/2^i) = \sum_{i=0}^{\log_2 n} n = \Theta(n \log n)$

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

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	$\log n$

```

1  class Sorted_Array_Set:
2      def __init__(self):
3          self.A = Array_Seq() # O(1)
4      def __len__(self):
5          return len(self.A) # O(1)
6      def __iter__(self):
7          yield from self.A # O(n)
8      def iter_order(self):
9          yield from self # O(n)
10
11     def build(self, X): # O(?)
12         self.A.build(X)
13         self._sort()
14
15     def _sort(self): # O(?)
16         ???
17
18     def _binary_search(self, k, i, j): # O(log n)
19         if i >= j:
20             return i
21         m = (i + j) // 2
22         x = self.A.get_at(m)
23         if x.key > k:
24             return self._binary_search(k, i, m - 1)
25         if x.key < k:
26             return self._binary_search(k, m + 1, j)

```

```

20         return m
21
22     def find_min(self):
23         if len(self) > 0:           # O(1)
24             return self.A.get_at(0)
25         else:
26
26     def find_max(self):
27         if len(self) > 0:           # O(1)
28             return self.A.get_at(len(self) - 1)
29         else:
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.

```

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

```

1 def insertion_sort(A):                      # Insertion sort array A
2     for i in range(1, len(A)):               # O(n) loop over array
3         j = i                               # O(1) initialize pointer
4         while j > 0 and A[j] < A[j - 1]:    # O(i) loop over prefix
5             A[j - 1], A[j] = A[j], A[j - 1] # O(1) swap
6             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 $\varepsilon > 0$.

```

1 def merge_sort(A, a = 0, b = None):
2     if b is None:
3         b = len(A)
4     if 1 < b - a:
5         c = (a + b + 1) // 2
6         merge_sort(A, a, c)           # Sort sub-array A[a:c]
7         merge_sort(A, c, b)          # O(1) initialize
8         L, R = A[a:c], A[c:b]       # O(1) size k = b - a
9         i, j = 0, 0                 # O(1) compute center
10        while a < b:
11            if (j >= len(R)) or (i < len(L)) and L[i] < R[j]: # T(k/2) recursively sort left
12                A[a] = L[i]                                         # T(k/2) recursively sort right
13                i = i + 1                                         # O(1) copy
14            else:
15                A[a] = R[j]                                         # O(1) initialize pointers
16                j = j + 1                                         # O(1) merge from left
17                a = a + 1                                         # O(1) decrement left pointer
18        else:
19            A[a] = R[j]                                         # O(1) merge from right
20            j = j + 1                                         # O(1) decrement right pointer
21        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_prev(k)
Array	n		n		n		n
Sorted Array	$n \log n$		$\log n$		n		1
							$\log n$

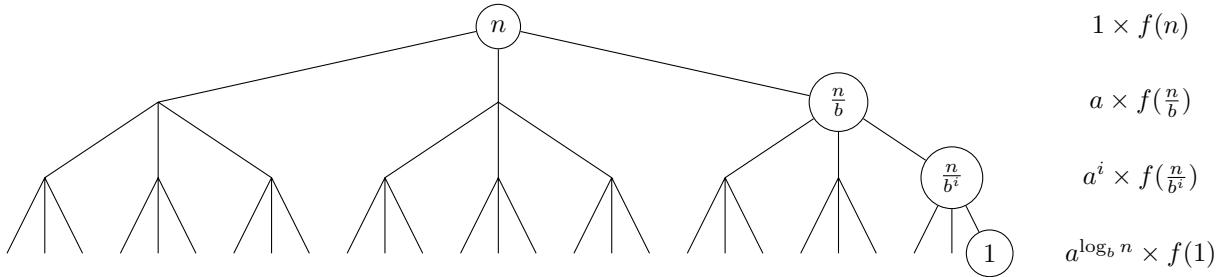
Recurrences

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 form $T(n) = aT(n/b) + \Theta(n^c)$ 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) formulas¹ 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 .

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

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

Lecture 4: Hashing

Review

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	$\log n$

- **Idea!** Want faster search and dynamic operations. Can we `find(k)` faster than $\Theta(\log n)$?
- Answer is no (lower bound)! (But actually, yes...!?)

Comparison Model

- In this model, assume algorithm can only differentiate items via comparisons
- **Comparable items:** black boxes only supporting comparisons between pairs
- Comparisons are $<$, \leq , $>$, \geq , $=$, \neq , outputs are binary: True or False
- **Goal:** Store a set of n comparable items, support `find(k)` operation
- Running time is **lower bounded** by # comparisons performed, so count comparisons!

Decision Tree

- Any algorithm can be viewed as a **decision tree** of operations performed
- An internal node represents a **binary comparison**, branching either True or False
- For a comparison algorithm, the decision tree is binary (draw example)
- A leaf represents algorithm termination, resulting in an algorithm **output**
- A **root-to-leaf path** represents an **execution of the algorithm** on some input
- Need at least one leaf for each **algorithm output**, so search requires $\geq n + 1$ leaves

Comparison Search Lower Bound

- What is worst-case running time of a comparison search algorithm?
 - running time \geq # comparisons \geq max length of any root-to-leaf path \geq height of tree
 - What is minimum height of any binary tree on $\geq n$ nodes?
 - Minimum height when binary tree is complete (all rows full except last)
 - Height $\geq \lceil \lg(n + 1) \rceil - 1 = \Omega(\log n)$, so running time of any comparison sort is $\Omega(\log n)$
 - Sorted arrays achieve this bound! Yay!
 - More generally, height of tree with $\Theta(n)$ leaves and max branching factor b is $\Omega(\log_b n)$
 - To get faster, need an operation that allows super-constant $\omega(1)$ branching factor. How??
-

Direct Access Array

- Exploit Word-RAM $O(1)$ time random access indexing! Linear branching factor!
- **Idea!** Give item **unique** integer key k in $\{0, \dots, u - 1\}$, store item in an array at index k
- Associate a meaning with each index of array
- If keys fit in a machine word, i.e. $u \leq 2^w$, worst-case $O(1)$ find/dynamic operations! Yay!
- 6.006: assume input numbers/strings fit in a word, unless length explicitly parameterized
- Anything in computer memory is a binary integer, or use (static) 64-bit address in memory
- But space $O(u)$, so really bad if $n \ll u$... :(
- **Example:** if keys are ten-letter names, for one bit per name, requires $26^{10} \approx 17.6$ TB space
- How can we use less space?

Hashing

- **Idea!** If $n \ll u$, map keys to a smaller range $m = \Theta(n)$ and use smaller direct access array
- **Hash function:** $h(k) : \{0, \dots, u - 1\} \rightarrow \{0, \dots, m - 1\}$ (also hash map)
- Direct access array called **hash table**, $h(k)$ called the **hash** of key k
- If $m \ll u$, no hash function is injective by pigeonhole principle

- Always exists keys a, b such that $h(a) = h(b) \rightarrow \text{Collision!} \quad :($
 - Can't store both items at same index, so where to store? Either:
 - store somewhere else in the array (**open addressing**)
 - * complicated analysis, but common and practical
 - store in another data structure supporting dynamic set interface (**chaining**)
-

Chaining

- **Idea!** Store collisions in another data structure (a chain)
 - If keys roughly evenly distributed over indices, chain size is $n/m = n/\Omega(n) = O(1)!$
 - If chain has $O(1)$ size, all operations take $O(1)$ time! Yay!
 - If not, many items may map to same location, e.g. $h(k) = \text{constant}$, chain size is $\Theta(n) \quad :($
 - Need good hash function! So what's a good hash function?
-

Hash Functions

Division (bad):
$$h(k) = (k \bmod m)$$

- Heuristic, good when keys are uniformly distributed!
- m should avoid symmetries of the stored keys
- Large primes far from powers of 2 and 10 can be reasonable
- Python uses a version of this with some additional mixing
- If $u \gg n$, every hash function will have some input set that will create $O(n)$ size chain
- **Idea!** Don't use a fixed hash function! Choose one randomly (but carefully)!

Universal (good, theoretically): $h_{ab}(k) = (((ak + b) \bmod p) \bmod m)$

- Hash Family $\mathcal{H}(p, m) = \{h_{ab} \mid a, b \in \{0, \dots, p-1\} \text{ and } a \neq 0\}$
- Parameterized by a fixed prime $p > u$, with a and b chosen from range $\{0, \dots, p-1\}$
- \mathcal{H} is a **Universal** family: $\Pr_{h \in \mathcal{H}} \{h(k_i) = h(k_j)\} \leq 1/m \quad \forall k_i \neq k_j \in \{0, \dots, u-1\}$
- Why is universality useful? Implies short chain lengths! (in expectation)
- X_{ij} indicator random variable over $h \in \mathcal{H}$: $X_{ij} = 1$ if $h(k_i) = h(k_j)$, $X_{ij} = 0$ otherwise
- Size of chain at index $h(k_i)$ is random variable $X_i = \sum_j X_{ij}$
- Expected size of chain at index $h(k_i)$

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

- Since $m = \Omega(n)$, load factor $\alpha = n/m = O(1)$, so $O(1)$ **in expectation!**

Dynamic

- If n/m far from 1, rebuild with new randomly chosen hash function for new size m
- Same analysis as dynamic arrays, cost can be **amortized** over many dynamic operations
- So a hash table can implement dynamic set operations in expected amortized $O(1)$ time! :)

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

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

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. \square

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!

```

1  class DirectAccessArray:
2      def __init__(self, u):    self.A = [None] * u      # O(u)
3      def find(self, k):      return self.A[k]        # O(1)
4      def insert(self, x):   self.A[x.key] = x        # O(1)
5      def delete(self, k):   self.A[k] = None         # O(1)
6      def find_next(self, k):
7          for i in range(k, len(self.A)):             # O(u)
8              if A[i] is not None:
9                  return A[i]
10     def find_max(self):
11         for i in range(len(self.A) - 1, -1, -1):  # O(u)
12             if A[i] is not None:
13                 return A[i]
14     def delete_max(self):
15         for i in range(len(self.A) - 1, -1, -1):  # O(u)
16             x = A[i]
17             if x is not None:
18                 A[i] = None
19                 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, $h(k) : \{0, \dots, u-1\} \rightarrow \{0, \dots, m-1\}$. 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 entirely¹. 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.

¹If 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 inputs². 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:

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

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.

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

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 $\{k_0, \dots, k_{n-1}\}$ stored in the hash table. Then the expected number of keys hashed to the chain at index $h(k_i)$ is:

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

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**.

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

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

```

1  class Hash_Table_Set:
2      def __init__(self, r = 200):                      # O(1)
3          self.chain_set = Set_from_Seq(Linked_List_Seq)
4          self.A = []
5          self.size = 0
6          self.r = r                                     # 100/self.r = fill ratio
7          self.p = 2**31 - 1
8          self.a = randint(1, self.p - 1)
9          self._compute_bounds()
10         self._resize(0)
11
12     def __len__(self):   return self.size           # O(1)
13     def __iter__(self):                         # O(n)
14         for X in self.A:
15             yield from X
16
17     def build(self, X):                          # O(n)e
18         for x in X: self.insert(x)
19
20     def _hash(self, k, m):                      # O(1)
21         return ((self.a * k) % self.p) % m
22
23     def _compute_bounds(self):                  # O(1)
24         self.upper = len(self.A)
25         self.lower = len(self.A) * 100*100 // (self.r*self.r)
26
27     def _resize(self, n):                      # O(n)
28         if (self.lower >= n) or (n >= self.upper):
29             f = self.r // 100
30             if self.r % 100:        f += 1
31             # f = ceil(r / 100)
32             m = max(n, 1) * f
33             A = [self.chain_set() for _ in range(m)]
34             for x in self:
35                 h = self._hash(x.key, m)
36                 A[h].insert(x)
37             self.A = A
38             self._compute_bounds()
39
40     def find(self, k):                         # O(1)e
41         h = self._hash(k, len(self.A))
42         return self.A[h].find(k)
43
44     def insert(self, x):                      # O(1)ae
45         self._resize(self.size + 1)
46         h = self._hash(x.key, len(self.A))
47         added = self.A[h].insert(x)
48         if added:    self.size += 1
49         return added
50
51

```

```
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 $A = [a_0, \dots, a_{n-1}]$ 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 family⁴), 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.

⁴In 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.

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

Problem Session 2

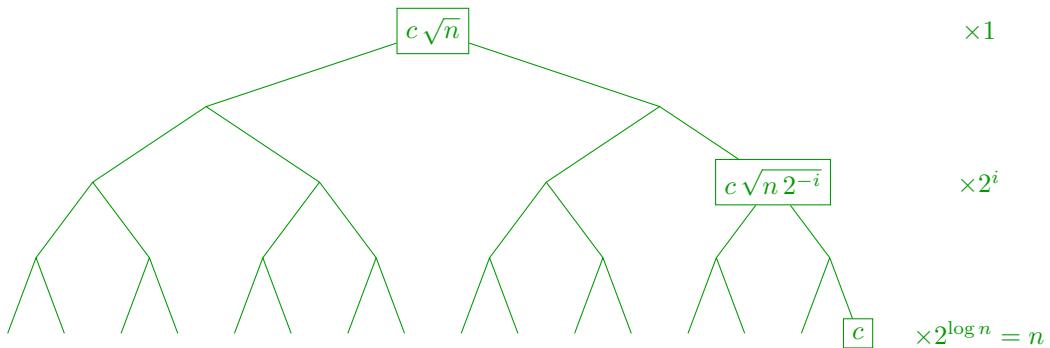
Problem 1-1. Solving recurrences

Derive solutions to the following recurrences in two ways: via a recursion tree **and** via Master Theorem. A solution should include the tightest upper and lower bounds that the recurrence will allow. Assume $T(1) \in \Theta(1)$.

(a) $T(n) = 2T\left(\frac{n}{2}\right) + O(\sqrt{n})$

Solution: $T(n) \in \Theta(n)$ by case 1 of the Master Theorem, since:

$$O(\sqrt{n}) \subseteq O(n) = O(n^{\log_2 2}).$$

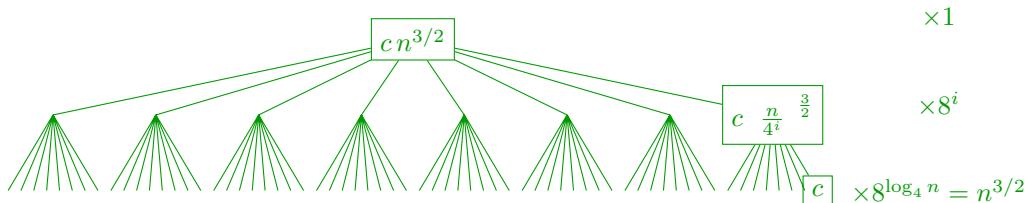


Drawing a tree, there are 2^i vertices at depth i each doing at most $c\sqrt{n}2^{-i}$ work, so the total work at depth i is at most $c2^{i/2}\sqrt{n}$. Summing over the entire tree, the total work is at most $\sum_{i=0}^{\log n} c2^{i/2}\sqrt{n} = c\sqrt{n}\sum_{i=0}^{\log n} 2^{i/2} = cn\frac{\sqrt{2}}{\sqrt{2}-1} \in O(n)$. Since $\Theta(1)$ work is done at each leaf, and there are n leaves, the total work is also $\Omega(n)$ leading to $\Theta(n)$ running time.

(b) $T(n) = 8T\left(\frac{n}{4}\right) + O(n\sqrt{n})$

Solution: $T(n) \in O(n^{3/2} \log n)$ by case 2 of the Master Theorem, since:

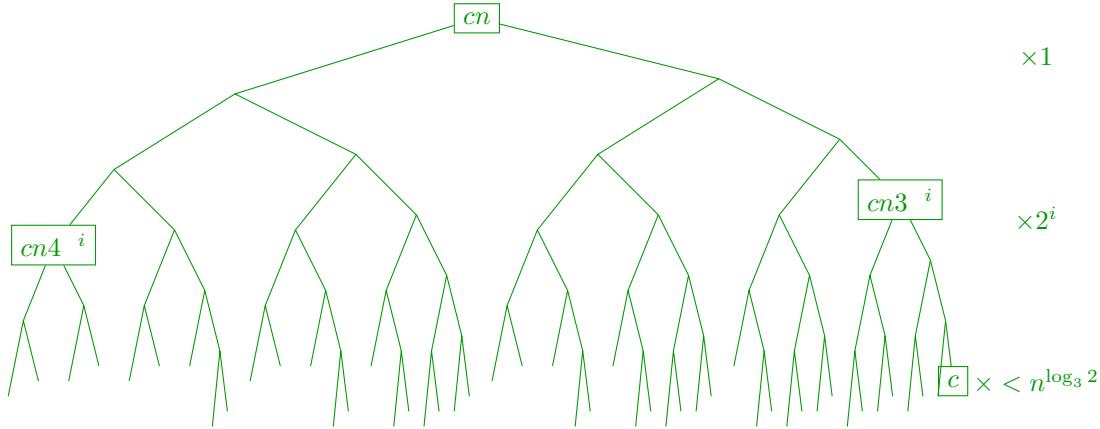
$$O(n\sqrt{n}) = O(n^{3/2}) = O(n^{\log_4 8}).$$



Drawing a tree, there are 8^i vertices at depth i each doing at most $c(n4^{-i})^{3/2} = cn^{3/2}8^{-i}$ work, so the total work at depth i is at most $cn^{3/2}$. Summing over the entire tree, the total work is at most $\sum_{i=0}^{\log_4 n} cn^{3/2} = \frac{c}{2}n^{3/2} \log n \in O(n^{3/2} \log n)$.

(c) $T(n) = T(\frac{n}{3}) + T(\frac{n}{4}) + \Theta(n)$ assuming $T(a) < T(b)$ for all $a < b$

Solution: By monotonicity, $T(n) \leq 2T(\frac{n}{3}) + \Theta(n)$. Then $T(n) \in O(n)$ by case 3 of the Master Theorem, since $\Theta(n) \subset \Omega(n^{\log_3 2+\epsilon})$ for some $\epsilon > 0$ (e.g. $\epsilon = 1/3$). On the other hand, if we ignore the recursive component, $T(n) \in \Omega(n)$. Combining the two gives $T(n) \in \Theta(n)$.



Drawing a tree, there are 2^i vertices at depth i , $\binom{i}{j}$ of which do $cn3^{-j}4^{j-i}$ work (as long as $3^j4^{j-i} \leq n$). As a crude upper bound, this is at most $cn3^{-i}$, so the total work at depth i is at most $cn\left(\frac{2}{3}\right)^i$. Summing over the entire tree, the total work is at most $\sum_{i=0}^{\infty} cn\left(\frac{2}{3}\right)^i = 3cn \in O(n)$. We have cn work at the root, giving a lower bound of $\Omega(n)$. Combining the two gives a total of $\Theta(n)$.

Problem 1-2. Stone Searching

Sanos is a supervillain on an intergalactic quest in search of an ancient and powerful artifact called the Thoul Stone. Unfortunately she has no idea what planet the stone is on. The universe is composed of an infinite number of planets, each identified by a unique positive integer. On each planet is an oracle who, after some persuasion, will tell Sanos whether or not the Thoul Stone is on a planet having a strictly higher planet identifier than their own. Interviewing every oracle in the universe would take forever, and Sanos wants to find the Thoul Stone quickly. Supposing the Thoul Stone resides on planet k , describe an algorithm to help Sanos find the Thoul Stone by interviewing at most $O(\log k)$ oracles.

Solution: First observe that if we could find a planet with identifier $x > k$ that is not too much larger than k (specifically, $x = \Theta(k)$), then we would be done, as binary searching the planets from 1 to $x - 1$ would find the value of k by visiting at most $O(\log x) = O(\log k)$ oracles. It remains to find such a planet x .

To find x , instruct Sanos to visit planets 2^i starting at $i = 0$ until an oracle on planet $x = 2^{i^*}$ first tells Sanos that $x > k$. Since x is the first planet for which $2^i > k$, then $x/2 < k$ and $x < 2k = \Theta(k)$ as desired. To reach planet $x = 2^{i^*}$, Sanos interviews $i^* = \lceil \log_2 k \rceil = \Theta(\log k)$ oracles, so to find k , this algorithm interviews at most $O(\log k)$ oracles as desired.

Problem 1-3. Collage Collating

Fodoby is a company that makes customized software tools for creative people. Their newest software, Ottoshop, helps users make collages by allowing them to overlay images on top of each other in a single document. Describe a database to keep track of the images in a given document which supports the following operations:

1. `make_document()`: construct an empty document containing no images
2. `import_image(x)`: add an image with unique integer ID x to the top of the document
3. `display()`: return an array of the document's image IDs in order from bottom to top
4. `move_below(x, y)`: move the image with ID x directly below the image with ID y

Operation (1) should run in worst-case $O(1)$ time, operations (2) and (3) should each run in worst-case $O(n)$ time, while operation (4) should run in worst-case $O(\log n)$ time, where n is the number of images contained in a document at the time of the operation.

Solution: This database requires us to maintain a sequence of images ordered extrinsically, but also support searching intrinsically for images based on their ID. So, we will implement the database with a combination of both a sequence data structure, specifically a **doubly linked list** (as implemented in PS1-4) storing image IDs, and a set data structure, specifically a sorted array storing pairs (x, v_x) sorted by x values, where x is the ID of an image, and v_x is a pointer to the linked list node containing x .

To implement `make_document()`, simply initialize an empty linked list L and an empty sorted array S , each in $O(1)$ time. There is no output to this operation, so it is trivially correct.

To implement `import_image(x)`, add x to the front of L in node v_x in $O(1)$ time and add (x, v_x) to S in $O(n)$ time. Delegating to these data structures ensures that x is added to front of the sequence stored in L , and that S now contains (x, v_x) and remains sorted after insertion.

To implement `display()`, construct and return an array by iterating the items of L in sequence order which can be done in $O(n)$.

To implement `move_below(x, y)`, use binary search to find pairs (x, v_x) and (y, v_y) in S , each in $O(\log n)$ time. Then we can remove node v_x from L in $O(1)$ time and insert it after node v_y , also in $O(1)$ time, by relinking pointers. For completeness, here is one way to relink the pointers in a doubly linked list:

```

1 def relink(S, vx, vy):
2     if vx.prev: vx.prev.next = vx.next
3     else: S.head = vx.next
4     if vx.next: vx.next.prev = vx.prev
5     else: S.tail = vx.prev
6     vx.prev = vy
7     vx.next = vy.next
8     if vy.next: vy.next.prev = vx
9     else: S.tail = vx
10    vy.next = vx

```

Problem 1-4. Brick Blowing

Porkland is a community of pigs who live in n houses lined up along one side of a long, straight street running east to west. Every house in Porkland was built from straw and bricks, but some houses were built with more bricks than others. One day, a wolf arrives in Porkland and all the pigs run inside their homes to hide. Unfortunately for the pigs, this wolf is extremely skilled at blowing down pig houses, aided by a strong wind already blowing from west to east. If the wolf blows in an easterly direction on a house containing b bricks, that house will fall down, along with every house east of it containing strictly fewer than b bricks. For every house in Porkland, the wolf wants to know its **damage**, i.e., the number of houses that would fall were he to blow on it in an easterly direction.

- (a) Suppose $n = 10$ and the number of bricks in each house in Porkland from west to east is $[34, 57, 70, 19, 48, 2, 94, 7, 63, 75]$. Compute for this instance the damage for every house in Porkland.

Solution: $[4, 5, 6, 3, 3, 1, 4, 1, 1, 1]$

- (b) A house in Porkland is **special** if it either (1) has no easterly neighbor or (2) its adjacent neighbor to the east contains at least as many bricks as it does. Given an array containing the number of bricks in each house of Porkland, describe an $O(n)$ -time algorithm to return the damage for every house in Porkland **when all but one house** in Porkland is special.

Solution: Maintain an array D of the same size as the input array H to store updated damages, where the i^{th} item of D is an integer representing the number of damages counted so far. To add damage to the i^{th} house, add to the value at $D[i]$ in $O(1)$ time. As each house will itself fall down when blown on, initialize every element of D to 1 in $O(n)$ time, and count other damages using the following algorithm.

If exactly one house (say the h^{th} house) in Porkland is not special, that means the subarray A from the east-most house to h non-strictly monotonically increases, as does the subarray B from the $(h + 1)^{\text{th}}$ house to the west-most house. We can find h in $O(n)$ time via a linear scan. The damage for any house in subarray B is 1, as no house to the west contains strictly fewer bricks, so these values are set correctly at initialization.

It remains to compute the damage for the houses in A . Use a two-finger algorithm starting with one index i at the beginning of A ($i = 0$) and another index j at the beginning of B ($j = 0$). Then repeat the following process until $i = |A|$: if $j < |B|$ and house $A[i]$ has strictly more bricks than $B[j]$, then increase j by 1; otherwise, add j to the damage at $D[i]$ and increase i by 1. This loop halts when $i + j = |A| + |B| = n$, and $i + j$ increases by one in each iteration. Since the work done in each iteration is $O(1)$, this algorithm runs in $O(n)$ time.

To prove that this algorithm correctly computes the damage for each house in A we first prove that the loop above maintains the invariant that at the start of each iteration, $A[i] > B[k]$ for all $k \in \{0, \dots, j - 1\}$. This property implies the algorithm computes damage correctly for each house in A : the algorithm updates the damage for $A[i]$ when $A[i] \leq B[j]$, so the claim implies the houses west of $A[i]$ with strictly fewer bricks are exactly the houses $H = \{B[k] \mid k \in \{0, \dots, j - 1\}\}$ where $|H| = j$; so the damage blowing on house $A[i]$ is $D[i] = j + 1$ as recorded.

To prove the claim, we induct on $i + j$. When $i + j = 0$, the claim is vacuously true as the set of possible k is empty. Now assume for induction that the claim holds for some $i + j$. If $A[i] > B[j]$, then increasing j by one directly maintains the invariant since $A[i] > B[k]$ for $k \in \{0, \dots, j - 1\}$ by induction. Alternatively, if $A[i] \leq B[j]$, then increasing i by 1 also maintains the induction hypothesis since $A[i + 1] > A[i]$, proving the claim.

- (c) Given an array containing the number of bricks in each house of Porkland, describe an $O(n \log n)$ -time algorithm to return the damage for every house in Porkland.

Solution: We modify merge sort to record all damages that occur between houses within each subarray before every merge. Since we will be moving brick values in H from their original locations, we will replace each brick value $b_i = H[i]$ with tuples $H[i] = (b_i, i)$, to keep track which house is associated with b_i .

As in (b), initialize a damages array D to 1s. Then, recursively sort and record damages that would occur between houses in the first half of H , and then do the same for the second half. Next use the $O(n)$ -time algorithm from part (b) to count the damages that would occur between one house in the first half and one house in the second half, and then use the merge step of merge sort to combine the two sorted halves into one sorted array in $O(n)$ time. Since both (b) and merge take $O(n)$ time, the recurrence for this algorithm is the same as merge sort, yielding an $O(n \log n)$ running time.

Now we prove this algorithm correctly records the damage of every house in a given subarray with any other house in the subarray (in addition to sorting the subarray), by inducting on the size of the subarray. When the subarray has size 1, there is exactly one damage between houses within that subarray, and the initialization step records it. Alternatively, assume for induction that the claim is true for all $k < n$. By induction, the algorithm correctly records all damages between houses within the first half of the subarray, and also all damages between houses within the second half. It remains to record damages between houses in the left half with houses on the right half.

Fortunately, since the first and last halves of the subarray are sorted, the algorithm in (b) counts exactly those damages. Then sorting using the merge step of merge sort maintains the invariant as desired.

Note that the merge step and the algorithm in part (b) are both two finger algorithms that traverse from the starts of the same two subarrays. Our implementation for (d) utilizes this observation to record the damages with the merge step of merge sort, rather than separately.

- (d) Write a Python function `get_damages` that implements your algorithm.

Solution:

```

1  def get_damages(H):
2      D = [1 for _ in H]
3      H2 = [(H[i], i) for i in range(len(H))]
4      def merge_sort(A, a = 0, b = None):
5          if b is None:    b = len(A)
6          if 1 < b - a:
7              c = (a + b + 1) // 2
8              merge_sort(A, a, c)
9              merge_sort(A, c, b)
10             i, j, L, R = 0, 0, A[a:c], A[c:b]
11             while a < b:
12                 if (j >= len(R)) or (i < len(L) and L[i][0] <= R[j][0]):
13                     D[L[i][1]] += j
14                     A[a] = L[i]
15                     i += 1
16                 else:
17                     A[a] = R[j]
18                     j += 1
19                     a += 1
20             merge_sort(H2)
21     return D

```

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

Lecture 5: Linear Sorting

Review

- Comparison search lower bound: any decision tree with n nodes has height $\geq \lceil \lg(n+1) \rceil - 1$
- Can do faster using random access indexing: an operation with linear branching factor!
- **Direct access array** is fast, but may use a lot of space ($\Theta(u)$)
- Solve space problem by mapping (**hashing**) key space u down to $m = \Theta(n)$
- **Hash tables** give **expected** $O(1)$ time operations, **amortized** if dynamic
- Expectation input-independent: choose hash function randomly from **universal** hash family
- Data structure overview!
- Last time we achieved faster find. Can we also achieve faster sort?

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

Comparison Sort Lower Bound

- Comparison model implies that algorithm decision tree is binary (constant branching factor)
 - Requires # leaves $L \geq \#$ possible outputs
 - Tree height lower bounded by $\Omega(\log L)$, so worst-case running time is $\Omega(\log L)$
 - To sort array of n elements, # outputs is $n!$ permutations
 - Thus height lower bounded by $\log(n!) \geq \log((n/2)^{n/2}) = \Omega(n \log n)$
 - So merge sort is optimal in comparison model
 - Can we exploit a direct access array to sort faster?
-

Direct Access Array Sort

- **Example:** [5, 2, 7, 0, 4]
- Suppose all keys are **unique** non-negative integers in range $\{0, \dots, u - 1\}$, so $n \leq u$
- Insert each item into a direct access array with size u in $\Theta(n)$
- Return items in order they appear in direct access array in $\Theta(u)$
- Running time is $\Theta(u)$, which is $\Theta(n)$ if $u = \Theta(n)$. Yay!

```

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

```

- What if keys are in larger range, like $u = \Omega(n^2) < n^2$?
- **Idea!** Represent each key k by tuple (a, b) where $k = an + b$ and $0 \leq b < n$
- Specifically $a = \lfloor k/n \rfloor < n$ and $b = (k \bmod n)$ (just a 2-digit base- n number!)
- This is a built-in Python operation $(a, b) = \text{divmod}(k, n)$
- **Example:** $[17, 3, 24, 22, 12] \Rightarrow [(3,2), (0,3), (4,4), (4,2), (2,2)] \Rightarrow [32, 03, 44, 42, 22]_{(n=5)}$
- How can we sort tuples?

Tuple Sort

- Item keys are tuples of equal length, i.e. item $x.key = (x.k_1, x.k_2, x.k_3, \dots)$.
- Want to sort on all entries **lexicographically**, so first key k_1 is most significant
- How to sort? **Idea!** Use other **auxiliary sorting algorithms** to separately sort each key
- (Like sorting rows in a spreadsheet by multiple columns)
- What order to sort them in? Least significant to most significant!
- **Exercise:** $[32, 03, 44, 42, 22] \Rightarrow [42, 22, 32, 03, 44] \Rightarrow [03, 22, 32, 42, 44]_{(n=5)}$

- **Idea!** Use tuple sort with **auxiliary direct access array sort** to sort tuples (a, b) .
- **Problem!** Many integers could have the same a or b value, even if input keys distinct
- Need sort allowing **repeated keys** which preserves input order
- Want sort to be **stable**: repeated keys appear in output in same order as input
- Direct access array sort cannot even sort arrays having repeated keys!
- Can we modify direct access array sort to admit multiple keys in a way that is stable?

Counting Sort

- Instead of storing a single item at each array index, store a chain, just like hashing!
- For stability, chain data structure should remember the order in which items were added
- Use a **sequence** data structure which maintains insertion order
- To insert item x , `insert_last` to end of the chain at index $x.key$
- Then to sort, read through all chains in sequence order, returning items one by one

```

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

```

Radix Sort

- **Idea!** If $u < n^2$, use tuple sort with **auxiliary counting sort** to sort tuples (a, b)
- Sort least significant key b , then most significant key a
- Stability ensures previous sorts stay sorted
- Running time for this algorithm is $O(2n) = O(n)$. Yay!
- If every key $< n^c$ for some positive $c = \log_n(u)$, every key has at most c digits base n
- A c -digit number can be written as a c -element tuple in $O(c)$ time
- We sort each of the c base- n digits in $O(n)$ time
- So tuple sort with **auxiliary counting sort** runs in $O(cn)$ time in total
- If c is constant, so each key is $\leq n^c$, this sort is linear $O(n)$!

```

1 def radix_sort(A):
2     "Sort A assuming items have non-negative keys"
3     n = len(A)
4     u = 1 + max([x.key for x in A])           # O(n) find maximum key
5     c = 1 + (u.bit_length() // n.bit_length())
6     class Obj: pass
7     D = [Obj() for a in A]                     # O(nc) make digit tuples
8     for i in range(n):                         # O(nc) make digit tuple
9         D[i].digits = []
10        D[i].item = A[i]
11        high = A[i].key
12        for j in range(c):                     # O(c) make digit tuple
13            high, low = divmod(high, n)
14            D[i].digits.append(low)
15        for i in range(c):                   # O(nc) sort each digit
16            for j in range(n):               # O(n) assign key i to tuples
17                D[j].key = D[j].digits[i]
18            counting_sort(D)             # O(n) sort on digit i
19        for i in range(n):                 # O(n) output to A
20            A[i] = D[i].item

```

Algorithm	Time $O(\cdot)$	In-place?	Stable?	Comments
Insertion Sort	n^2	Y	Y	$O(nk)$ for k -proximate
Selection Sort	n^2	Y	N	$O(n)$ swaps
Merge Sort	$n \log n$	N	Y	stable, optimal comparison
Counting Sort	$n + u$	N	Y	$O(n)$ when $u = O(n)$
Radix Sort	$n + n \log_n(u)$	N	Y	$O(n)$ when $u = O(n^c)$

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

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 $\Omega(\log(n!)) = \Omega(n \log n)$ height¹, 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 $\{0, \dots, u - 1\}$ (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.

```

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

```

¹We 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.

```

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

```

1 def counting_sort(A):
2     "Sort A assuming items have non-negative keys"
3     u = 1 + max([x.key for x in A])           # O(n) find maximum key
4     D = [0] * u                                # O(u) direct access array
5     for x in A:                                # O(n) count keys
6         D[x.key] += 1
7         for k in range(1, u):                  # O(u) cumulative sums
8             D[k] += D[k - 1]
9         for x in list(reversed(A)):            # O(n) move items into place
10            A[D[x.key] - 1] = x
11            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. $x.k_1, x.k_2, x.k_3, \dots$), 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!

```

1 def radix_sort(A):
2     "Sort A assuming items have non-negative keys"
3     n = len(A)
4     u = 1 + max([x.key for x in A])           # O(n) find maximum key
5     c = 1 + (u.bit_length() // n.bit_length())
6     class Obj: pass
7     D = [Obj() for a in A]
8     for i in range(n):                         # O(nc) make digit tuples
9         D[i].digits = []
10        D[i].item = A[i]
11        high = A[i].key
12        for j in range(c):                     # O(c) make digit tuple
13            high, low = divmod(high, n)
14            D[i].digits.append(low)
15        for i in range(c):                   # O(nc) sort each digit
16            for j in range(n):               # O(n) assign key i to tuples
17                D[j].key = D[j].digits[i]
18                counting_sort(D)          # O(n) sort on digit i
19            for i in range(n):             # O(n) output to A
20                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 $[-n^2, \dots, n^3]$.

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 $\{0, \dots, 25\}$ 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 $\Theta(nk)$.

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

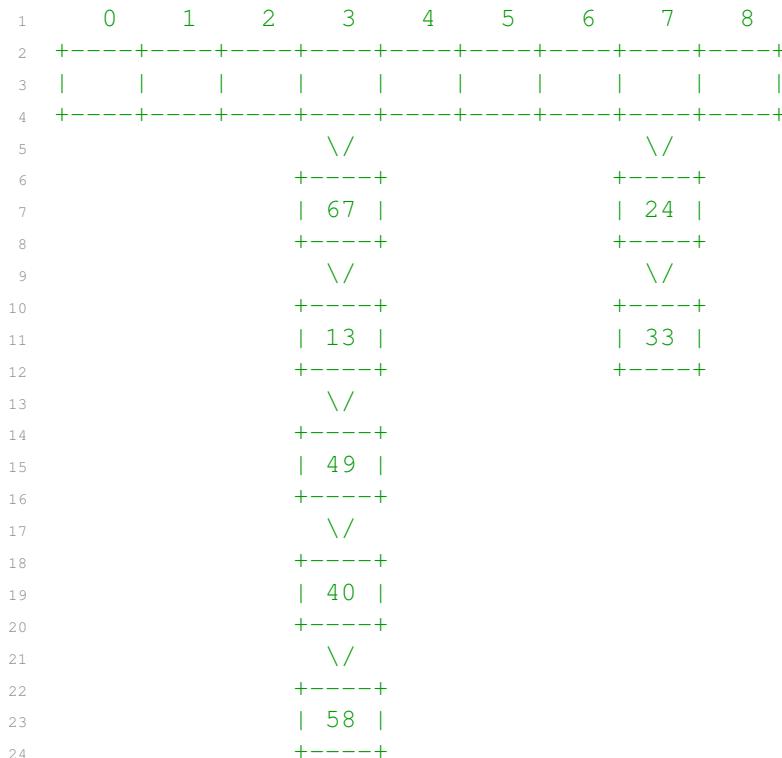
For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

Problem Session 3

Problem 3-1. Hash It Out

Insert integer keys $A = [67, 13, 49, 24, 40, 33, 58]$ in order into a hash table of size 9 using the hash function $h(k) = (11k + 4) \bmod 9$. Collisions should be resolved via chaining, where collisions are stored at the end of a chain. Draw a picture of the hash table after all keys have been inserted.

Solution:



Problem 3-2. Hash Sequence

Hash tables are not only useful for implementing Set operations; they can be used to implement Sequences as well! (Recall the Set and Sequence interfaces were defined in Lecture and Recitation 02.) Given a hash table, describe how to use it as a black-box¹ (using only its Set interface operations) to implement the Sequence interface such that:

- `build(A)` runs in expected $O(n)$ time,
- `get_at` and `set_at` each run in expected $O(1)$ time,

¹By black-box, we mean you should not modify the inner workings of the data structure or algorithm.

- `insert_at` and `delete_at` each run in expected $O(n)$ time, and
- the four dynamic first/last operations each run in amortized expected $O(1)$ time.

Solution: To use a hash table H to implement the Sequence operations, store each Sequence item x in an object b with key $b.key$ and value $b.val = x$, and we will store these keyed objects in the hash table. We also maintain the lowest key s stored in the hash table, to maintain invariant that the n stored objects have keys $(s, \dots, s + n - 1)$, where the i^{th} item in the Sequence is stored in the object with key $s + i$.

To implement `build(A)`, for each item x_i in $A = (x_0, \dots, x_{n-1})$ construct its keyed object b , initially with key $b.key = i$, in worst-case $O(1)$ time; then insert it into the hash table using Set `insert(b)` in expected $O(1)$ time, for an expected total of $O(n)$ time. Initializing $s = 0$ ensures the invariant is satisfied.

To implement `get_at(i)`, return the value of the stored object with key $s + i$ using Set `find(s + i)` in expected $O(1)$ time, which is correct by the invariant. Similarly, to implement `set_at(i, x)`, find the object with key $s + i$ using `find(s + i)` and change its value to x , also in expected $O(1)$ time.

To implement `insert_at(i, x)`, for each j from $s + n - 1$ down to $s + i$, remove the object b with key j using `delete(j)` in expected $O(1)$ time, change its key to $j + 1$ in worst-case $O(1)$ time, and then insert the object with `insert(b)` in expected $O(1)$ time. Then, construct a keyed object b' with value x and key $s + i$, and insert with `insert(b')` in expected $O(1)$ time, for an expected total of $O(n)$ time. This operation restores the invariant for each affected item.

Similarly, to implement `delete_at(i)`, remove the object b' stored at $s + i$ with `delete(s + i)` in expected $O(1)$ time; then for each j from $s + i + 1$ to $s + n - 1$, remove the object b with key j using `delete(j)` in expected $O(1)$ time, change its key to $j - 1$ in worst-case $O(1)$ time, and then insert the object with `insert(b)` in expected $O(1)$ time. Then return the value of object b' , for an expected total of $O(n)$ time. This operation returns the correct value by the invariant, and restores the invariant for each affected item.

To implement `insert_last(x)` or `delete_last()`, simply reduce to `insert_at(s + n)` or `delete_at(s + n - 1)` in expected $O(1)$ time since no objects need to be shifted.

To implement `insert_first(x)`, construct a keyed object b with value x and key $s - 1$ and insert it with `insert(b)` in expected $O(1)$ time. Then setting the stored value of s to $s - 1$ restores the invariant. Similarly for `delete_first()`, remove the object with key s using `delete(s)` in expected $O(1)$ time, and return the value of the object. Then setting the stored value of s to $s + 1$ restores the invariant.

Problem 3-3. Critter sort

Ashley Getem collects and trains Pocket Critters to fight other Pocket Critters in battle. She has collected n Critters in total, and she keeps track of a variety of statistics for each Critter C_i . Describe **efficient**² algorithms to sort Ashley's Critters based on each of the following keys:

²By “efficient”, we mean that faster correct algorithms will receive more points than slower ones.

- (a) Species ID: an integer x_i between $-n$ and n (negative IDs are grumpy)

Solution: These integers are in a linearly bounded range, but are not positive. So take worst-case $O(n)$ time to add n to each critter's ID so that $x_i \leq 2n = u$ for all i , sort them using counting sort in worst-case $O(n+2n) = O(n)$ time, and then subtract n from each ID, again in worst-case $O(n)$ time.

- (b) Name: a unique string s_i containing at most $10\lceil \lg n \rceil$ English letters

Solution: Let's assume that each string is stored sequentially in a contiguous chunk of memory, in an encoding such that the numerical representation of each character is bounded above by some constant number k (e.g., 26 for efficient English letter encoding, or 256 for byte encoding), where the numerical representation of one character c_i is smaller than that of another character c_j if c_i comes before c_j in the English alphabet. Then each string can be thought of as an integer between 0 and $u = k^{10\lceil \lg n \rceil} = O(n^{10\lg k}) = n^{O(1)}$, stored in a constant number of machine words, so can be sorted using radix sort in worst-case $O(n + n \log_n n^{O(1)}) = O(n)$ time.

Alternatively, if each character in the each string s_i is stored in its own machine word, then the input has size $\Theta(n \log n)$. For each string, compute its $n^{O(1)}$ numerical representation by direct computation in $O(\log n)$ arithmetic computations (which can each be performed in $O(1)$ time, since each intermediate representation fits into at most 10 machine words). Then sort using radix-sort as above. Computing the numerical representations of the strings takes $O(n \log n)$ time, which is linear in the size of the input.

- (c) Number of fights fought: a positive integer f_i under n^2

Solution: These integers are in a polynomially bounded range $u = n^2$, so sort them using radix-sort in worst-case $O(n + n \log_n n^2) = O(n)$ time.

- (d) Win fraction: ratio w_i/f_i where $w_i \leq f_i$ is the number of fights won

Solution: Non-integer division may yield a number that requires an infinite number of digits to represent, so we cannot compute these numbers directly. Solutions attempting to compute and compare such numbers without accounting for precision should not be awarded any points. We present two solutions here.

The first solution uses an optimal comparison sorting algorithm like merge sort to sort the win fractions in worst-case $O(n \log n)$ time. Two win ratios w_1/f_1 and w_2/f_2 can be compared via cross multiplication, since $w_1/f_1 < w_2/f_2$ if and only if $w_1f_2 < w_2f_1$. This solution done correctly is worth 4/5 points.

The second solution is more tricky. The idea will be to scale the ratios sufficiently such that when they are not equal, their integer parts also not equal. First, for each win number, compute $w'_i = w_i \cdot n^4$ in $O(1)$ time. Then compute $p_i = \lfloor w'_i/f_i \rfloor$ in $O(1)$ time via integer division, where $w'_i = p_i \cdot f_i + q_i$ for $q_i = w'_i \bmod f_i$. Then, since each p_i is a positive integer bounded by $O(n^6)$, we can sort by p_i in worst-case $O(n + n \log_n n^6) = O(n)$ time via radix sort.

Now we must prove that sorting by p_i is equivalent to sorting by w_i/f_i . It suffices to prove that $w_i/f_i - w_j/f_j > 0$ is true if and only if $p_i - p_j > 0$ is true. Without loss of

generality, assume that $d_w = w_i/f_i - w_j/f_j > 0$. Since

$$d_w n^4 = w'_i/f_i - w'_j/f_j = (p_i + q_i/f_i) - (p_j + q_j/f_j) = (p_i - p_j) + (q_i/f_i - q_j/f_j),$$

it suffices to show that $p_i - p_j = d_w n^4 - q_w > 0$ where $q_w = q_i/f_i - q_j/f_j$. First, q_w is maximized when:

$$q_w = \frac{n^2 - 2}{n^2 - 1} - \frac{0}{1} < 1,$$

while $d_w n^4$ is minimized when:

$$d_w n^4 = \left(\frac{1}{n^2 - 2} - \frac{1}{n^2 - 1} \right) n^4 = \frac{n^4}{n^4 - 3n^2 + 2} > 1,$$

so $d_w n^4 - q_w > 0$ as desired.

Problem 3-4. College Construction

MIT has employed Gank Frehry to build a new wing of the Stata Center to house the new College of Computing. MIT wants the new building be as tall as possible, but Cambridge zoning laws limit all new buildings to be no higher than positive integer height h . As an innovative architect, Frehry has decided to build the new wing by stacking two giant aluminum cubes on top of each other, into which rooms will be carved. However, Frehry's supplier of aluminum cubes can only make cubes with a limited set of positive integer side lengths $S = \{s_0, \dots, s_{n-1}\}$. Help Frehry purchase cubes for the new building.

- (a) Assuming the input S fits within $\Theta(n)$ machine words, describe an **expected** $O(n)$ -time algorithm to determine whether there exist a pair of side lengths in S that exactly sum to h .

Solution: It suffices to check for each s_i whether $(h - s_i) \in S$. Naively, we could perform this check by comparing $h - s_i$ against all $s_j \in S - \{s_i\}$, which would take $O(n)$ time for each s_i , leading to $O(n^2)$ running time. We can speed up this algorithm by first storing the elements of S in a hash table H so that looking up each $h - s_i$ can be done quickly. For each $s_i \in S$, insert s_i into H in expected $O(1)$ time. Now all unique values that occur in S appear in H , so for each s_i , check whether $h - s_i$ appears in H in expected $O(1)$ time. (If the supplier can build only one of each block size, we can also check that $h - s_i \neq s_i$.) Building the hash table and then checking for matches each take expected $O(n)$ time, so this algorithm runs in $O(n)$ time. This brute force algorithm is correct because each s_i satisfies $s_i + k_i = h$ for exactly one integer k_i , and we check all possible (s_i, k_i) .

- (b) Unfortunately for Frehry, there is no pair of side lengths in S that sum exactly to h . Assuming that $h = 600n^6$, describe a **worst-case** $O(n)$ -time algorithm to return a pair of side lengths in S whose sum is closest to h without going over.

Solution: We do not know whether all $s_i \in S$ are polynomially bounded in n ; but we do know that h is. If some $s_i \geq h$, it can certainly not be part of a pair of positive side

lengths from S that sum to under h . So first perform a linear scan of S and remove all $s_i \geq h$ to construct set S' . Now the integers in S' are each upper bounded by $O(n^6)$, so we can sort them in worst-case $O(n + n \log_n n^6)$ time using radix-sort, and store the output in an array A .

Now we can sweep the sorted list using a two-finger algorithm similar to the merge step in merge sort to find a pair with the largest sum at most h , if such a pair exists. Specifically, initialize indices $i = 0$ and $j = |S'| - 1$, and repeat the following procedure, keeping track of the largest sum t found so far initialized to zero. If $A[i] + A[j] \leq h$, then if $t < A[i] + A[j]$, you have found a better pair, so set $t = A[i] + A[j]$; regardless $A[k] + A[j] < t$ for all $k \leq i$, so increase i by one. Otherwise if $A[i] + A[j] > h$, then $A[i] + A[\ell] > h$ for all $\ell \geq j$, so decrease j by one. If $j < i$ (or $j = i$ and we want distinct s_i, s_j), then return False. This loop maintains the invariant that at the start of each loop, we have confirmed that $A[k] + A[\ell] \geq t$ for all $k \leq i \leq j \leq \ell$ for which $A[k] + A[\ell] \leq h$, so the algorithm is correct. Since each iteration of the loop takes $O(1)$ time and decreases $j - i$ decrease by one, and $j - i = |S'| - 1$ starts positive and ends when $j - i < 0$, this procedure takes at most $O(n)$ time in the worst case.

Problem 3-5. Po- k -er Hands

Meff Ja is a card shark who enjoys playing card games. He has found an unusual deck of cards, where each of the n cards in the deck is marked with a lowercase letter from the 26-character English alphabet. We represent a deck of cards as a sequence of letters, where the first letter corresponds to the top of the deck. Meff wants to play a game of Po- k -er with you. To begin the game, he deals you a Po- k -er hand of k cards in the following way:

1. The deck D starts in a pile face down in a known order.
2. Meff **cuts** the deck uniformly at random at some location $i \in \{0, \dots, n - 1\}$, i.e., move the top i cards in order to the bottom of the deck.
3. Meff then deals you the top k cards from the top of the cut deck.
4. You **sort** your k cards alphabetically, resulting in your Po- k -er **hand**.

Let $P(D, i, k)$ be the Po- k -er hand resulting from cutting a deck D at location i . Then cutting deck $D = \text{'abcdabc'}$ at location 2 would result in the deck 'cdbcab' , which would then yield the Po-4-er hand $P(D, 2, 4) = \text{'bccd'}$. From a given starting deck, many hands are possible depending on where the deck is cut. Meff wants to know the **most likely** Po- k -er hand for a given deck. Given that the most likely Po- k -er hand is not necessarily unique, Meff always prefers the lexicographically smallest hand.

- (a) Describe a data structure that can be built in $O(n)$ time from a deck D of n cards and integer k , after which it can support `same(i, j)`: a constant-time operation which returns True if $P(D, i, k) = P(D, j, k)$ and False otherwise.

Solution: We build a direct access array mapping each index $i \in \{0, \dots, n - 1\}$ to a frequency table of the letters in hand $P(D, i, k)$, specifically a direct access array A of length 26 where $A[j]$ corresponds to the number of times the $(j + 1)$ th letter of the English alphabet occurs in the hand. The frequency table of hand $P(D, 0, k)$ can be computed in $O(k)$ time by simply looping through the cards in the hand and adding them to the frequency table. Then given the frequency table of $P(D, i, k)$, we can compute the frequency table of $P(D, i + 1, k)$ in constant time by subtracting one from letter $D[i]$ and adding one to letter $D[i + k]$. Building the above hash table then takes $O(k) + nO(1) = O(n)$ time. To support $\text{same}(i, j)$, look up indices i and j in the direct access array in constant time. If the corresponding frequency tables are the same, then the hands must also match. We can check if they match in worst-case constant time since each frequency has constant length (i.e., 26), so this operation takes worst-case $O(1)$ time. Students may use a hash table to achieve expected $O(1)$ time.

- (b) Given a deck of n cards, describe an $O(n)$ -time algorithm to find the most likely Po- k -er hand, breaking ties lexicographically. State whether your algorithm's running time is worst-case, amortized, and/or expected.

Solution: Build the data structure from part (a) in worst-case $O(n)$ time, specifically a direct access array of hand frequency tables. Now, compute the frequency of each hand directly: loop through the direct access array and add each hand frequency table to a hash table T mapping to value 1; if a hand table h already exists in T , increase $T[h]$ by 1. This procedure performs one hash table operation for each of the n hand tables, so it runs in expected $O(n)$ time. Next, find the largest frequency of any hand directly by looping through all hands in T , keeping track of f the largest frequency seen in worst-case $O(n)$ time. Then, construct a list of hand tables with frequency f directly by looping through all hands in T again, appending to the end of a dynamic array A every hand table that has frequency f , also in worst-case $O(n)$ time. The lexicographically first hand will be the one whose hand frequency table is lexicographically last (e.g., $(1, 0, \dots) > (0, 1, \dots)$ but ' $a\dots$ ' < ' $b\dots$ '), so loop through the hand tables and keep track of the lexicographically last hand table t in worst-case $O(n)$ time. Lastly, convert hand table t back into a hand by concatenating k letters in order based on their frequency in worst-case $O(k)$ time, and then return the hand. Then in total, this procedure runs in expected $O(n)$ time.

We can reduce to **worst-case** $O(n)$ time using radix sort instead of a hash table to count the frequencies of hand tables. Namely, we apply tuple/radix sort to the data structure from part (a). Each hand frequency table consists of 26 numbers between 0 and n , so we can treat them as a base- $(n + 1)$ integer of 26 digits. Sorting by each digit from least to most significant, we put the hand frequency tables into lexically increasing order. Now a single scan through the array, at each step checking whether the hand frequency table matches the previous, lets us compute the frequency of each table. A scan of these occurrence frequencies lets us find the maximum frequency f , and another scan of the array lets us find the lexicographically last hand with frequency f .

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

Lecture 6: Binary Trees I

Previously and New Goal

Sequence Data Structure	Container	Operations $O(\cdot)$		
		Static	Dynamic	
	build(X)	get_at(i) set_at(i, x)	insert_first(x) delete_first()	insert_last(x) delete_last()
Array	n	1	n	n
Linked List	n	n	1	n
Dynamic Array	n	1	n	$1_{(a)}$
Goal	n	$\log n$	$\log n$	$\log n$

Set Data Structure	Container	Operations $O(\cdot)$			Order
		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
Goal	$n \log n$	$\log n$	$\log n$	$\log n$	$\log n$

How? Binary Trees!

- Pointer-based data structures (like Linked List) can achieve **worst-case** performance
- Binary tree is pointer-based data structure with three pointers per node
- Node representation: `node.{item, parent, left, right}`
- **Example:**

1	_____<A>_____	node <A> <C> <D> <E> <F>
2	_______ <C>	item A B C D E F
3	__<D>_____ <E>	parent - <A> <A> <D>
4	<F>	left <C> - <F> - -
5		right <C> <D> - - - -

Terminology

- The **root** of a tree has no parent (**Ex:** $\langle A \rangle$)
 - A **leaf** of a tree has no children (**Ex:** $\langle C \rangle$, $\langle E \rangle$, and $\langle F \rangle$)
 - Define **depth**($\langle X \rangle$) of node $\langle X \rangle$ in a tree rooted at $\langle R \rangle$ to be length of path from $\langle X \rangle$ to $\langle R \rangle$
 - Define **height**($\langle X \rangle$) of node $\langle X \rangle$ to be max depth of any node in the **subtree** rooted at $\langle X \rangle$
 - **Idea:** Design operations to run in $O(h)$ time for root height h , and maintain $h = O(\log n)$
 - A binary tree has an inherent order: its **traversal order**
 - every node in node $\langle X \rangle$'s left subtree is **before** $\langle X \rangle$
 - every node in node $\langle X \rangle$'s right subtree is **after** $\langle X \rangle$
 - List nodes in traversal order via a recursive algorithm starting at root:
 - Recursively list left subtree, list self, then recursively list right subtree
 - Runs in $O(n)$ time, since $O(1)$ work is done to list each node
 - **Example:** Traversal order is ($\langle F \rangle$, $\langle D \rangle$, $\langle B \rangle$, $\langle E \rangle$, $\langle A \rangle$, $\langle C \rangle$)
 - Right now, traversal order has no meaning relative to the stored items
 - Later, assign semantic meaning to traversal order to implement Sequence/Set interfaces
-

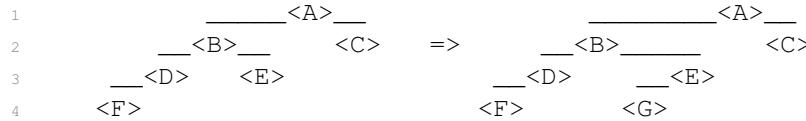
Tree Navigation

- **Find first** node in the traversal order of node $\langle X \rangle$'s subtree (last is symmetric)
 - If $\langle X \rangle$ has left child, recursively return the first node in the left subtree
 - Otherwise, $\langle X \rangle$ is the first node, so return it
 - Running time is $O(h)$ where h is the height of the tree
 - **Example:** first node in $\langle A \rangle$'s subtree is $\langle F \rangle$
- **Find successor** of node $\langle X \rangle$ in the traversal order (predecessor is symmetric)
 - If $\langle X \rangle$ has right child, return first of right subtree
 - Otherwise, return lowest ancestor of $\langle X \rangle$ for which $\langle X \rangle$ is in its left subtree
 - Running time is $O(h)$ where h is the height of the tree
 - **Example:** Successor of: $\langle B \rangle$ is $\langle E \rangle$, $\langle E \rangle$ is $\langle A \rangle$, and $\langle C \rangle$ is None

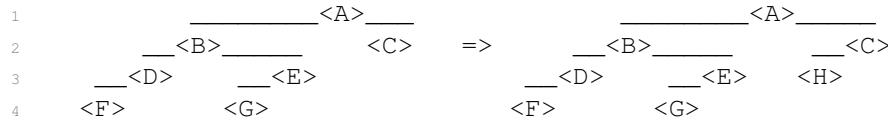
Dynamic Operations

- Change the tree by a single item (only add or remove leaves):
 - add a node after another in the traversal order (before is symmetric)
 - remove an item from the tree
- **Insert** node $\langle Y \rangle$ after node $\langle X \rangle$ in the traversal order
 - If $\langle X \rangle$ has no right child, make $\langle Y \rangle$ the right child of $\langle X \rangle$
 - Otherwise, make $\langle Y \rangle$ the left child of $\langle X \rangle$'s successor (which cannot have a left child)
 - Running time is $O(h)$ where h is the height of the tree

- **Example:** Insert node $\langle G \rangle$ before $\langle E \rangle$ in traversal order

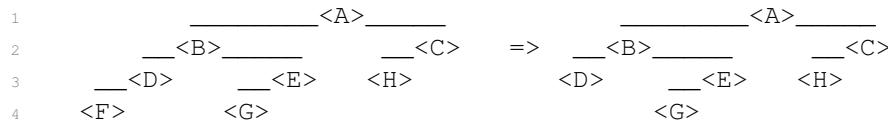


- **Example:** Insert node $\langle H \rangle$ after $\langle A \rangle$ in traversal order

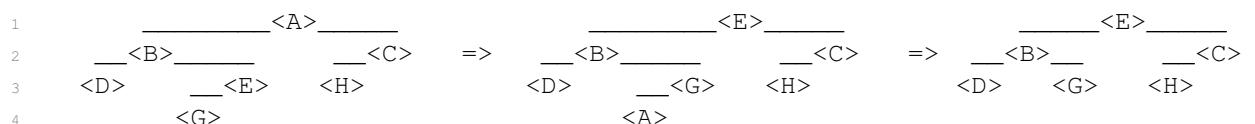


- **Delete** the item in node $\langle X \rangle$ from $\langle X \rangle$'s subtree

- If $\langle X \rangle$ is a leaf, detach from parent and return
- Otherwise, $\langle X \rangle$ has a child
 - * If $\langle X \rangle$ has a left child, swap items with the predecessor of $\langle X \rangle$ and recurse
 - * Otherwise $\langle X \rangle$ has a right child, swap items with the successor of $\langle X \rangle$ and recurse
- Running time is $O(h)$ where h is the height of the tree
- **Example:** Remove $\langle F \rangle$ (a leaf)



- **Example:** Remove $\langle A \rangle$ (not a leaf, so first swap down to a leaf)



Application: Set

- **Idea! Set Binary Tree** (a.k.a. **Binary Search Tree / BST**):
Traversal order is sorted order increasing by key
 - Equivalent to **BST Property**: for every node, every key in left subtree \leq node's key \leq every key in right subtree
 - Then can find the node with key k in node $\langle x \rangle$'s subtree in $O(h)$ time like binary search:
 - If k is smaller than the key at $\langle x \rangle$, recurse in left subtree (or return `None`)
 - If k is larger than the key at $\langle x \rangle$, recurse in right subtree (or return `None`)
 - Otherwise, return the item stored at $\langle x \rangle$
 - Other Set operations follow a similar pattern; see recitation
-

Application: Sequence

- **Idea! Sequence Binary Tree**: Traversal order is sequence order
- How do we find i^{th} node in traversal order of a subtree? Call this operation `subtree_at(i)`
- Could just iterate through entire traversal order, but that's bad, $O(n)$
- However, if we could compute a subtree's **size** in $O(1)$, then can solve in $O(h)$ time
 - How? Check the size n_L of the left subtree and compare to i
 - If $i < n_L$, recurse on the left subtree
 - If $i > n_L$, recurse on the right subtree with $i' = i - n_L - 1$
 - Otherwise, $i = n_L$, and you've reached the desired node!
- Maintain the size of each node's subtree at the node via **augmentation**
 - Add `node.size` field to each `node`
 - When adding new leaf, add $+1$ to `a.size` for all ancestors a in $O(h)$ time
 - When deleting a leaf, add -1 to `a.size` for all ancestors a in $O(h)$ time
- Sequence operations follow directly from a fast `subtree_at(i)` operation
- Naively, `build(x)` takes $O(nh)$ time, but can be done in $O(n)$ time; see recitation

So Far

Set 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)
Binary Tree	$n \log n$	h	h	h	h
Goal	$n \log n$	$\log n$	$\log n$	$\log n$	$\log n$

Sequence Data Structure	Operations $O(\cdot)$				
	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)
Binary Tree	n	h	h	h	h
Goal	n	$\log n$	$\log n$	$\log n$	$\log n$

Next Time

- Keep a binary tree **balanced** after insertion or deletion
- Reduce $O(h)$ running times to $O(\log n)$ by keeping $h = O(\log n)$

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

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`).

```

1 class Binary_Node:
2     def __init__(A, x):                      # O(1)
3         A.item   = x
4         A.left   = None
5         A.right  = None
6         A.parent = None
7         # 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 its children, and so we call the node “binary” based on the number of children the node has.

A binary tree has one node 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 $\langle x \rangle$ back to the root are called the **ancestors** for $\langle x \rangle$ in the tree. The **depth** of a node $\langle x \rangle$ in the subtree rooted at $\langle R \rangle$ is the length of the path from $\langle x \rangle$ back to $\langle R \rangle$. The **height** of node $\langle x \rangle$ is the maximum depth of any node in the subtree rooted at $\langle x \rangle$. 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 $\langle A \rangle$ comes **before** $\langle A \rangle$ in the traversal order; and
- every node in the right subtree of node $\langle A \rangle$ comes **after** $\langle A \rangle$ in the traversal order.

Given a binary node $\langle A \rangle$, we can list the nodes in $\langle A \rangle$'s subtree by recursively listing the nodes in $\langle A \rangle$'s left subtree, listing $\langle A \rangle$ itself, and then recursively listing the nodes in $\langle A \rangle$'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 $\langle A \rangle$, if $\langle A \rangle$ has a right child, then $\langle A \rangle$'s successor will be the first node in the right child's subtree. Otherwise, $\langle A \rangle$'s successor cannot exist in $\langle A \rangle$'s subtree, so we walk up the tree to find the lowest ancestor of $\langle A \rangle$ such that $\langle A \rangle$ 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.

```

1  def successor(A):                      # O(h)
2      if A.right: return A.right.subtree_first()
3      while A.parent and (A is A.parent.right):
4          A = A.parent
5      return A.parent
6
7  def predecessor(A):                     # O(h)
8      if A.left:  return A.left.subtree_last()
9      while A.parent and (A is A.parent.left):
10         A = A.parent
11     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 $\langle B \rangle$ before a given node $\langle A \rangle$ in the traversal order, either node $\langle A \rangle$ has a left child or not. If $\langle A \rangle$ does not have a left child, than we can simply add $\langle B \rangle$ as the left child of $\langle A \rangle$. Otherwise, if $\langle A \rangle$ has a left child, we can add $\langle B \rangle$ as the right child of the last node in $\langle A \rangle$'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.

```

1  def subtree_insert_before(A, B):           # O(h)
2      if A.left:
3          A = A.left.subtree_last()
4          A.right, B.parent = B, A
5      else:
6          A.left, B.parent = B, A
7      # A.maintain()                         # wait for R07!
8
9  def subtree_insert_after(A, B):             # O(h)
10     if A.right:
11         A = A.right.subtree_first()
12         A.left, B.parent = B, A
13     else:
14         A.right, B.parent = B, A
15     # 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.

```

1  def subtree_delete(A):                      # O(h)
2      if A.left or A.right:                   # A is not a leaf
3          if A.left:  B = A.predecessor()
4          else:      B = A.successor()
5          A.item, B.item = B.item, A.item
6          return B.subtree_delete()
7      if A.parent:                          # A is a leaf
8          if A.parent.left is A: A.parent.left = None
9          else:                  A.parent.right = None
10         # A.parent.maintain()           # wait for R07!
11         return A

```

Binary Node Full Implementation

```

1  class Binary_Node:
2      def __init__(A, x):                      # O(1)
3          A.item   = x
4          A.left   = None
5          A.right  = None
6          A.parent = None
7          # A.subtree_update()             # wait for R07!
8
9      def subtree_iter(A):                     # O(n)
10         if A.left:   yield from A.left.subtree_iter()
11         yield A
12         if A.right:  yield from A.right.subtree_iter()
13
14     def subtree_first(A):                   # O(h)
15         if A.left:   return A.left.subtree_first()
16         else:       return A
17
18     def subtree_last(A):                   # O(h)
19         if A.right:  return A.right.subtree_last()
20         else:       return A
21
22     def successor(A):                      # O(h)
23         if A.right: return A.right.subtree_first()
24         while A.parent and (A is A.parent.right):
25             A = A.parent
26         return A.parent
27
28     def predecessor(A):                   # O(h)
29         if A.left:  return A.left.subtree_last()
30         while A.parent and (A is A.parent.left):
31             A = A.parent
32         return A.parent
33

```

```

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 $A = (a_0, \dots, a_{n-1})$, 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.

```

1 def build(X):
2     A = [x for x in X]
3     def build_subtree(A, i, j):
4         c = (i + j) // 2
5         root = self.Node_Type(A[c])
6         if i < c:                      # needs to store more items in left subtree
7             root.left = build_subtree(A, i, c - 1)
8             root.left.parent = root
9         if c < j:                      # needs to store more items in right subtree
10            root.right = build_subtree(A, c + 1, j)
11            root.right.parent = root
12        return root
13    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.

```

1 def tree_iter(T):
2     node = T.subtree_first()
3     while node:
4         yield node
5         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.

```

1  class BST_Node(Binary_Node):
2      def subtree_find(A, k):                      # O(h)
3          if k < A.item.key:
4              if A.left:   return A.left.subtree_find(k)
5          elif k > A.item.key:
6              if A.right: return A.right.subtree_find(k)
7          else:           return A
8          return None
9
10     def subtree_find_next(A, k):                  # O(h)
11         if A.item.key <= k:
12             if A.right: return A.right.subtree_find_next(k)
13             else:       return None
14         elif A.left:
15             B = A.left.subtree_find_next(k)
16             if B:       return B
17         return A
18
19     def subtree_find_prev(A, k):                  # O(h)
20         if A.item.key >= k:
21             if A.left:   return A.left.subtree_find_prev(k)
22             else:       return None
23         elif A.right:
24             B = A.right.subtree_find_prev(k)
25             if B:       return B
26         return A
27
28     def subtree_insert(A, B):                     # O(h)
29         if B.item.key < A.item.key:
30             if A.left:   A.left.subtree_insert(B)
31             else:       A.subtree_insert_before(B)
32         elif B.item.key > A.item.key:
33             if A.right: A.right.subtree_insert(B)
34             else:       A.subtree_insert_after(B)
35         else:       A.item = B.item

```

```
1 class Set_Binary_Tree(Binary_Tree): # Binary Search Tree
2     def __init__(self): super().__init__(BST_Node)
3
4     def iter_order(self): yield from self
5
6     def build(self, X):
7         for x in X: self.insert(x)
8
9     def find_min(self):
10        if self.root: return self.root.subtree_first().item
11
12    def find_max(self):
13        if self.root: return self.root.subtree_last().item
14
15    def find(self, k):
16        if self.root:
17            node = self.root.subtree_find(k)
18            if node: return node.item
19
20    def find_next(self, k):
21        if self.root:
22            node = self.root.subtree_find_next(k)
23            if node: return node.item
24
25    def find_prev(self, k):
26        if self.root:
27            node = self.root.subtree_find_prev(k)
28            if node: return node.item
29
30    def insert(self, x):
31        new_node = self.Node_Type(x)
32        if self.root:
33            self.root.subtree_insert(new_node)
34            if new_node.parent is None: return False
35        else:
36            self.root = new_node
37        self.size += 1
38        return True
39
40    def delete(self, k):
41        assert self.root
42        node = self.root.subtree_find(k)
43        assert node
44        ext = node.subtree_delete()
45        if ext.parent is None: self.root = None
46        self.size -= 1
47        return ext.item
```

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

Lecture 7: Binary Trees II: AVL

Last Time and Today's Goal

Sequence Data Structure	Operations $O(\cdot)$				
	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)
Binary Tree	n	h	h	h	h
AVL Tree	n	$\log n$	$\log n$	$\log n$	$\log n$

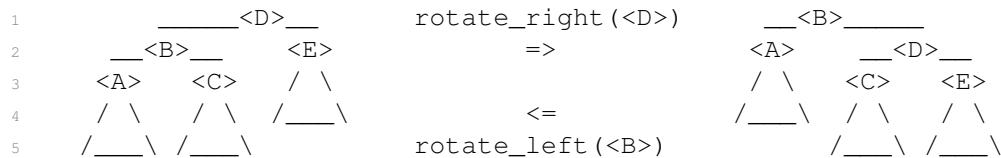
Set Data Structure	Operations $O(\cdot)$				
	Container	Static	Dynamic	Order	
	build(x)	find(k)	insert(x) delete(k)	find_min() find_max()	find_prev(k) find_next(k)
Binary Tree	$n \log n$	h	h	h	h
AVL Tree	$n \log n$	$\log n$	$\log n$	$\log n$	$\log n$

Height Balance

- How to maintain height $h = O(\log n)$ where n is number of nodes in tree?
- A binary tree that maintains $O(\log n)$ height under dynamic operations is called **balanced**
 - There are many balancing schemes (Red-Black Trees, Splay Trees, 2-3 Trees, ...)
 - First proposed balancing scheme was the **AVL Tree** (Adelson-Velsky and Landis, 1962)

Rotations

- Need to reduce height of tree without changing its traversal order, so that we represent the same sequence of items
- How to change the structure of a tree, while preserving traversal order? **Rotations!**



- A rotation relinks $O(1)$ pointers to modify tree structure and maintains traversal order

Rotations Suffice

- **Claim:** $O(n)$ rotations can transform a binary tree to any other with same traversal order.
 - **Proof:** Repeatedly perform last possible right rotation in traversal order; resulting tree is a canonical chain. Each rotation increases depth of the last node by 1. Depth of last node in final chain is $n - 1$, so at most $n - 1$ rotations are performed. Reverse canonical rotations to reach target tree. \square
 - Can maintain height-balance by using $O(n)$ rotations to fully balance the tree, but slow :(
 - We will keep the tree balanced in $O(\log n)$ time per operation!
-

AVL Trees: Height Balance

- AVL trees maintain **height-balance** (also called the **AVL Property**)
 - A node is **height-balanced** if heights of its left and right subtrees differ by at most 1
 - Let **skew** of a node be the height of its right subtree minus that of its left subtree
 - Then a node is height-balanced if its skew is $-1, 0$, or 1
- **Claim:** A binary tree with height-balanced nodes has height $h = O(\log n)$ (i.e., $n = 2^{\Omega(h)}$)
- **Proof:** Suffices to show fewest nodes $F(h)$ in any height h tree is $F(h) = 2^{\Omega(h)}$

$$F(0) = 1, F(1) = 2, F(h) = 1 + F(h-1) + F(h-2) \geq 2F(h-2) \implies F(h) \geq 2^{h/2} \quad \square$$

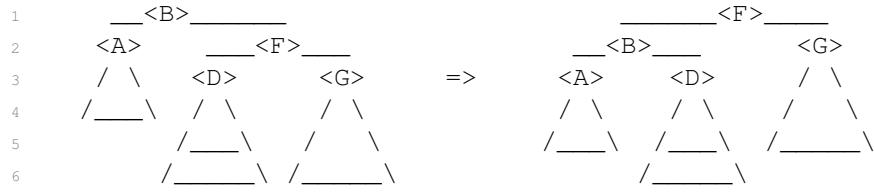
- Suppose adding or removing leaf from a height-balanced tree results in imbalance
 - Only subtrees of the leaf's ancestors have changed in height or skew
 - Heights changed by only ± 1 , so skews still have magnitude ≤ 2
 - **Idea:** Fix height-balance of ancestors starting from leaf up to the root
 - Repeatedly rebalance lowest ancestor that is not height-balanced, wlog assume skew 2

- **Local Rebalance:** Given binary tree node $\langle B \rangle$:

- whose skew 2 and
- every other node in $\langle B \rangle$'s subtree is height-balanced,
- then $\langle B \rangle$'s subtree can be made height-balanced via one or two rotations
- (after which $\langle B \rangle$'s height is the same or one less than before)

- **Proof:**

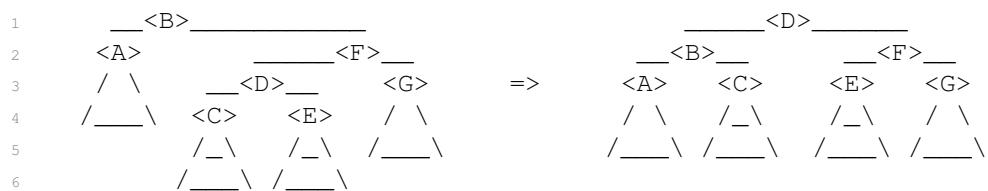
- Since skew of $\langle B \rangle$ is 2, $\langle B \rangle$'s right child $\langle F \rangle$ exists
- **Case 1:** skew of $\langle F \rangle$ is 0 or **Case 2:** skew of $\langle F \rangle$ is 1
 - * Perform a left rotation on $\langle B \rangle$



- * Let $h = \text{height}(\langle A \rangle)$. Then $\text{height}(\langle G \rangle) = h + 1$ and $\text{height}(\langle D \rangle)$ is $h + 1$ in Case 1, h in Case 2
- * After rotation:
 - the skew of $\langle B \rangle$ is either 1 in Case 1 or 0 in Case 2, so $\langle B \rangle$ is height balanced
 - the skew of $\langle F \rangle$ is -1 , so $\langle F \rangle$ is height balanced
 - the height of $\langle B \rangle$ before is $h + 3$, then after is $h + 3$ in Case 1, $h + 2$ in Case 2

-
- **Case 3:** skew of $\langle F \rangle$ is -1 , so the left child $\langle D \rangle$ of $\langle F \rangle$ exists

- * Perform a right rotation on $\langle F \rangle$, then a left rotation on $\langle B \rangle$



- * Let $h = \text{height}(\langle A \rangle)$. Then $\text{height}(\langle G \rangle) = h$ while $\text{height}(\langle C \rangle)$ and $\text{height}(\langle E \rangle)$ are each either h or $h - 1$
- * After rotation:
 - the skew of $\langle B \rangle$ is either 0 or -1 , so $\langle B \rangle$ is height balanced
 - the skew of $\langle F \rangle$ is either 0 or 1, so $\langle F \rangle$ is height balanced
 - the skew of $\langle D \rangle$ is 0, so D is height balanced
 - the height of $\langle B \rangle$ is $h + 3$ before, then after is $h + 2$

- **Global Rebalance:** Add or remove a leaf from height-balanced tree T to produce tree T' . Then T' can be transformed into a height-balanced tree T'' using at most $O(\log n)$ rotations.
 - **Proof:**
 - Only ancestors of the affected leaf have different height in T' than in T
 - Affected leaf has at most $h = O(\log n)$ ancestors whose subtrees may have changed
 - Let $\langle x \rangle$ be lowest ancestor that is not height-balanced (with skew magnitude 2)
 - If a leaf was added into T :
 - * Insertion increases height of $\langle x \rangle$, so in Case 2 or 3 of Local Rebalancing
 - * Rotation decreases subtree height: balanced after one rotation
 - If a leaf was removed from T :
 - * Deletion decreased height of one child of $\langle x \rangle$, not $\langle x \rangle$, so only imbalance
 - * Could decrease height of $\langle x \rangle$ by 1; parent of $\langle x \rangle$ may now be imbalanced
 - * So may have to rebalance every ancestor of $\langle x \rangle$, but at most $h = O(\log n)$ of them
 - So can maintain height-balance using only $O(\log n)$ rotations after insertion/deletion!
 - But requires us to evaluate whether possibly $O(\log n)$ nodes were height-balanced
-

Computing Height

- How to tell whether node $\langle x \rangle$ is height-balanced? Compute heights of subtrees!
- How to compute the height of node $\langle x \rangle$? Naive algorithm:
 - Recursively compute height of the left and right subtrees of $\langle x \rangle$
 - Add 1 to the max of the two heights
 - Runs in $\Omega(n)$ time, since we recurse on every node :(
- **Idea:** Augment each node with the height of its subtree! (Save for later!)
- Height of $\langle x \rangle$ can be computed in $O(1)$ time from the heights of its children:
 - Look up the stored heights of left and right subtrees in $O(1)$ time
 - Add 1 to the max of the two heights
- During dynamic operations, we must **Maintain** our augmentation as the tree changes shape
- Recompute subtree augmentations at every node whose subtree changes:
 - Update relinked nodes in a rotation operation in $O(1)$ time (ancestors don't change)
 - Update all ancestors of an inserted or deleted node in $O(h)$ time by walking up the tree

Steps to Augment a Binary Tree

- In general, to augment a binary tree with a **subtree property** P , you must:
 - State the subtree property $P(<X>)$ you want to store at each node $<X>$
 - Show how to compute $P(<X>)$ from the augmentations of $<X>$'s children in $O(1)$ time
 - Then stored property $P(<X>)$ can be maintained without changing dynamic operation costs
-

Application: Sequence

- For sequence binary tree, we needed to know subtree **sizes**
 - For just inserting/deleting a leaf, this was easy, but now need to handle rotations
 - Subtree size is a subtree property, so can maintain via augmentation
 - Can compute size from sizes of children by summing them and adding 1
-

Conclusion

- Set AVL trees achieve $O(\lg n)$ time for all set operations, except $O(n \log n)$ time for build and $O(n)$ time for iter
 - Sequence AVL trees achieve $O(\lg n)$ time for all sequence operations, except $O(n)$ time for build and iter
-

Application: Sorting

- Any Set data structure defines a sorting algorithm: build (or repeatedly insert) then iter
- For example, Direct Access Array Sort from Lecture 5
- AVL Sort is a new $O(n \lg n)$ -time sorting algorithm

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

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 its 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.

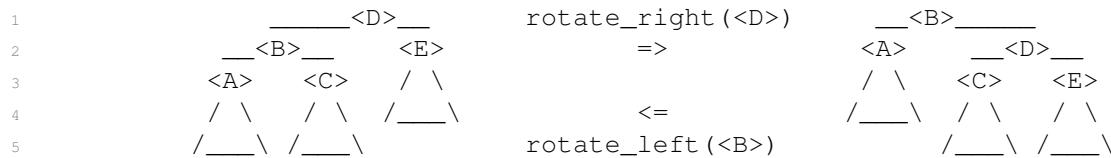
Solution: Balanced means that $h = O(\log n)$. Equivalently, balanced means that $\log n$ is lower bounded by $\Omega(h)$ so that $n = 2^{\Omega(h)}$. So if we can show the minimum number of nodes in a height-balanced tree is at least exponential in h , then it must also be balanced. Let $F(h)$ denote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



This operation preserves the traversal order of the tree while changing the depth of the nodes in subtrees $\langle A \rangle$ and $\langle E \rangle$. 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.

```

1  def skew(A):                      # O(?)  
2      return height(A.right) - height(A.left)  
3  
4  def rebalance(A):                 # O(?)  
5      if A.skew() == 2:  
6          if A.right.skew() < 0:  
7              A.right.subtree_rotate_right()  
8          A.subtree_rotate_left()  
9      elif A.skew() == -2:  
10         if A.left.skew() > 0:  
11             A.left.subtree_rotate_left()  
12         A.subtree_rotate_right()  
13  
14  def maintain(A):                # O(h)  
15      A.rebalance()  
16      A.subtree_update()  
17      if A.parent: A.parent.maintain()

```

Unfortunately, it's not clear how to efficiently evaluate the skew of 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 $\langle A \rangle$'s subtree, so takes at least $\Omega(n)$ time.

```

1  def height(A):                  # Omega(n)  
2      if A is None: return -1  
3      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.

```

1  def height(A):  
2      if A:    return A.height  
3      else:   return -1  
  
1  def subtree_update(A):           # O(1)  
2      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

```

1  def height(A):
2      if A:    return A.height
3      else:   return -1
4
5  class Binary_Node:
6      def __init__(A, x):                      # O(1)
7          A.item   = x
8          A.left   = None
9          A.right  = None
10         A.parent = None
11         A.subtree_update()
12
13     def subtree_update(A):                  # O(1)
14         A.height = 1 + max(height(A.left), height(A.right))
15
16     def skew(A):                          # O(1)
17         return height(A.right) - height(A.left)
18
19     def subtree_iter(A):                  # O(n)
20         if A.left:   yield from A.left.subtree_iter()
21         yield A
22         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.

```

1  class Size_Node(Binary_Node):
2      def subtree_update(A):                      # O(1)
3          super().subtree_update()
4          A.size = 1
5          if A.left:    A.size += A.left.size
6          if A.right:   A.size += A.right.size
7
8      def subtree_at(A, i):                      # O(h)
9          assert 0 <= i
10         if A.left:        L_size = A.left.size
11         else:            L_size = 0
12         if i < L_size:    return A.left.subtree_at(i)
13         elif i > L_size:  return A.right.subtree_at(i - L_size - 1)
14         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>

```
1  class Seq_Binary_Tree(Binary_Tree):
2      def __init__(self): super().__init__(Size_Node)
3
4      def build(self, X):
5          def build_subtree(X, i, j):
6              c = (i + j) // 2
7              root = self.Node_Type(A[c])
8              if i < c:
9                  root.left = build_subtree(X, i, c - 1)
10                 root.left.parent = root
11             if c < j:
12                 root.right = build_subtree(X, c + 1, j)
13                 root.right.parent = root
14             root.subtree_update()
15             return root
16         self.root = build_subtree(X, 0, len(X) - 1)
17         self.size = self.root.size
18
19     def get_at(self, i):
20         assert self.root
21         return self.root.subtree_at(i).item
22
23     def set_at(self, i, x):
24         assert self.root
25         self.root.subtree_at(i).item = x
26
27     def insert_at(self, i, x):
28         new_node = self.Node_Type(x)
29         if i == 0:
30             if self.root:
31                 node = self.root.subtree_first()
32                 node.subtree_insert_before(new_node)
33             else:
34                 self.root = new_node
35         else:
36             node = self.root.subtree_at(i - 1)
37             node.subtree_insert_after(new_node)
38         self.size += 1
39
40     def delete_at(self, i):
41         assert self.root
42         node = self.root.subtree_at(i)
43         ext = node.subtree_delete()
44         if ext.parent is None: self.root = None
45         self.size -= 1
46         return ext.item
47
48     def insert_first(self, x): self.insert_at(0, x)
49     def delete_first(self): return self.delete_at(0)
50     def insert_last(self, x): self.insert_at(len(self), x)
51     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!).

```

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

```

Solution:

```

1 Line # 1      | 2,3      | 4          | 5          | 6          | 7
2           |           |           |           |           |
3 Result   None | ____8____ | ____8____ | ____8_____ | ____8_____ | ____12_____
4           | 10_ _1_ | 10_ _-4_ | 10_ _-4_ | 10_ _-4_ | 6_ _-4_
5           | 6 5 3 | 6 5 3 | 6 5_ 3 | 6 _12_ 3 | 10 5 18 3
6           |           |           | 18         | 5 18       |
7
8 Also labeled with subtree height H, size #:
9
10 None
11 _____8H2#6_____
12 10H1#2_____     _____1H1#3_____
13      6H0#1        5H0#1        3H0#1
14
15 _____8H2#6_____
16 10H1#2_____     _____1H1#3_____
17      6H0#1        5H0#1        3H0#1
18
19 _____8H2#6_____
20 10H1#2_____     _____-4H1#3_____
21      6H0#1        5H0#1        3H0#1
22
23 _____8H3#7_____
24 10H1#2_____     _____-4H2#4_____
25      6H0#1        5H1#2_____    3H0#1
26          18H0#1
27
28 _____8H3#8_____
29 10H1#2_____     _____-4H2#5_____
30      6H0#1        5H0#1        3H0#1
31          18H0#1
32
33 _____12H2#7_____
34 6H1#3_____     _____-4H1#3_____
35 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_up to(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 $A.\text{subtree_ones}$, the number of 1 bits in its subtree. We can maintain this augmentation in $O(1)$ time from the augmentations stored at its children.

```

1 def update(A):
2     A.subtree_ones = A.item
3     if A.left:
4         A.subtree_ones += A.left.subtree_ones
5     if A.right:
6         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 $A.\text{item}$. 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_up to(i)`, we will first define the subtree-based recursive function `subtree_count_ones_up to(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_up to(i)` is symmetrically equivalent to `subtree_count_ones_up to(T.root, i)`. Since each recursive call makes at most one recursive call on a child, operation takes $O(\log n)$ time.

```

1 def subtree_count_ones_up to(A, i):
2     assert 0 <= i < A.size
3     out = 0
4     if A.left:
5         if i < A.left.size:
6             return subtree_count_ones_up to(A.left, i)
7         out += A.left.subtree_ones
8         i -= A.left.size
9     out += A.item
10    if i > 0:
11        assert A.right
12        out += subtree_count_ones_up to(A.right, i - 1)
13    return out

```

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

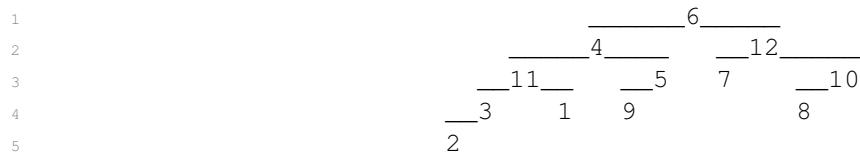
Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

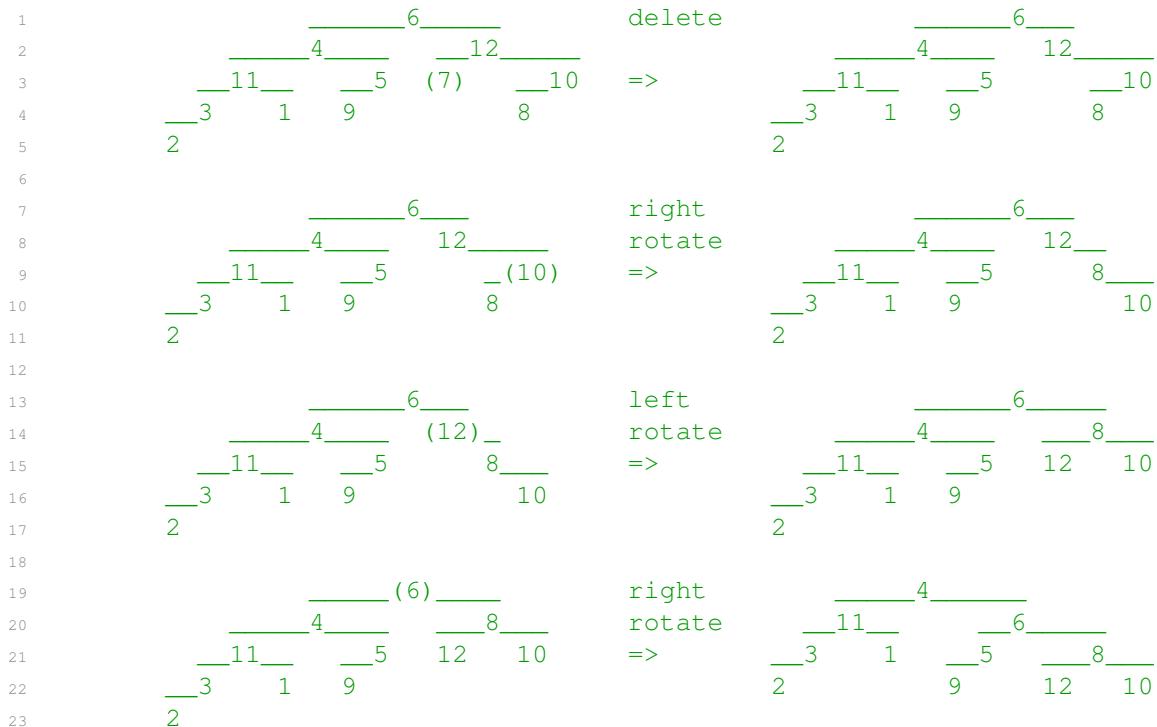
Problem Session 4

Problem 4-1. Sequence Rotations

Below is a Sequence AVL Tree T . Perform operation $T.\text{delete_at}(8)$ and draw the tree after each rotation operation performed during the operation.



Solution:



Problem 4-2. Fick Nury

Fick Nury directs an elite group of n superheroes called the Revengers. He has heard that supervillain Sanos is making trouble on a distant planet and needs to decide whether to confront her. Fick surveys the Revengers and compiles a list of n polls, where each poll is a tuple matching a different Revenger's name with their integer opinion on the topic. Opinion $+s$ means they are for confronting Sanos with strength s , while opinion $-s$ means they are against confronting Sanos with strength s . Fick wants to generate a list containing the names of the $\log n$ Revengers having the strongest opinions (breaking ties arbitrarily), so he can meet with them to discuss. For this problem, assume that the record containing the polls is **read-only** access controlled (the material is classified), so any computation must be written to alternative memory.

- (a) Describe an $O(n)$ -time algorithm to generate Fick's list.

Solution: Build a maximum priority queue containing all Revengers (together with their names). The priority queue stores at most n Revengers, so should take up no more than $O(n)$ space. Key each Revenger r_i with opinion s_i on the pair $(|s_i|, i)$ to make keys unique. Then delete the maximum keyed Revenger $\log n$ times and return the results. The running time of this algorithm is $B(n) + (\log n) \cdot D$ where $B(n)$ is the time to build a priority queue of size n and D is the time for delete max. We can achieve the desired running time by using a binary max-heap as our priority queue, since build takes $O(n)$ time and deletions take amortized $O(\log n)$ time.

- (b) Now suppose Fick's computer is only allowed to write to at most $O(\log n)$ space. Describe an $O(n \log \log n)$ -time algorithm to generate Fick's list.

Solution: Build a minimum priority queue containing the opinions of the first $\log n$ Revengers in the list (together with their names). Key each Revenger r_i with opinion s_i on the pair $(|s_i|, i)$ to make keys unique. Then repeat the following procedure for the remaining Revengers, while maintaining the invariant that after processing Revenger i , the priority queue contains the $\log n$ Revengers from the first i having the highest absolute opinion, which is true for $i = \log n$ and would solve our problem when $i = n$: (1) delete the minimum Revenger r^* from the priority queue, (2) compare the key of r^* to that of the i^{th} Revenger r_i , and (3) insert whichever is larger into the priority queue. If $r_i < r^*$, then r_i is smaller than every Revenger in the priority queue so is not in the top $\log n$. Alternatively, if $r^* < r_i$, then r_i is larger than the k^{th} largest found so far. In either case, adding the larger to the priority queue maintains the invariant. In all cases, the priority queue stores at most $\log n$ Revengers, so should take up no more than $O(\log n)$ space. The running time of this algorithm is $B(\log n) + (n - \log n) \cdot (I + D)$ where $B(\log n)$ is the time to build a priority queue of size $\log n$ and I and D are the time for insertion and delete min respectively. Thus we can use either a Set AVL tree or a min-heap as our priority queue to achieve $O(n \log \log n)$ running time, using at most $O(\log n)$ space.

Problem 4-3. SCLR

Stormen, Ceiserson, Livest, and Rein are four academics who wrote a very popular textbook in computer science, affectionately known as SCLR. They just found k first editions in their offices, and want to auction them off online for charity. Each bidder in the auction has a unique integer bidder ID and can bid some positive integer amount for a single copy (but may increase or decrease their bid while the auction is live). Describe a database supporting the following operations, assuming n is the number of bidders in the database at the time of the operation. For each operation, state whether your running time is worst-case, expected, and/or amortized.

<code>new_bid(d, b)</code>	record a new bidder ID d with bid b in $O(\log n)$ time
<code>update_bid(d, b)</code>	update the bid of existing bidder ID d to bid b in $O(\log n)$ time
<code>get_revenue()</code>	return revenue from selling to the current k highest bidders in $O(1)$ time

Solution: First, observe that operation `update_bid` requires finding and modifying the bid of a bidder given their ID, so maintain a dictionary containing the bidders keyed by bidder ID (call it the bidder dictionary). Either a hash table or Set AVL tree will work for this purpose, where using a hash table will yield amortized expected running times for the first two operations.

We will also need to keep track of an ordering of bids stored: in particular the sum of the k highest at any given time. To do this, maintain two priority queues: a min priority queue containing the k highest bids, and a max priority queue containing the remaining $n - k$ bids. Because we need $O(\log n)$ performance on priority queue insertions or deletions, we can use either Set AVL trees or binary heaps to implement the priority queues, where using binary heaps will yield amortized running times for the first two operations.

In order to find where each bidder is in the priority queues, we store with each bidder in the bidder dictionary a cross-linking pointer to the bidder's bid in one of the priority queues (i.e., its AVL node or index in a binary heap). In what follows, we will assume a Set AVL tree is used to implement the bidder dictionary and the priority queues. In addition, maintain the sum B of all bids in the min priority queue so that we can return it in constant time.

To implement `new_bid(d, b)`, remove the smallest bid b' with bidder ID d' from the min priority queue in $O(\log n)$ time, and decrease B by its bid amount b' . Then compare b to b' . Whichever is larger is among the k highest bidders seen so far while the other is not, so insert the larger into the min priority queue in $O(\log n)$ time while increasing B by its bid; and insert the smaller into the max priority queue, also in $O(\log n)$ time. In addition, add d to the bidder dictionary pointing to its bid location in a priority queue, and update the pointer associated with bidder d' , each in $O(\log n)$ time. This procedure maintains the invariants of the data structures presented, and runs in $O(\log n)$ total time in the worst case.

To implement `update_bid(d, b)`, find existing bidder d 's bid location in the priority queues using the bidder dictionary in $O(\log n)$ time, and then remove its record from its priority queue, also in $O(\log n)$ time. If the min priority queue has only $k - 1$ bids, remove the max from the max priority queue and insert it into the min priority queue in $O(\log n)$ time. While doing this, recompute B in $O(1)$ time and update the cross-linked pointers in $O(\log n)$ time. Now that the bidder with ID d has been removed, we can now reinsert it as a new bidder with updated bid b and

insert using `new_bid` as above. This operation performs a constant number of calls to $O(\log n)$ time operations, so this operation runs in worst-case $O(\log n)$ time.

To implement `get_revenue()`, simply return the stored value of B in worst-case $O(1)$ time.

Problem 4-4. Receiver Roster

Coach Bell E. Check is trying to figure out which of her football receivers to put in her starting lineup. In each game, Coach Bell wants to start the receivers who have the highest **performance** (the average number of points made in the games they have played), but has been having trouble because her data is incomplete, though interns do often add or correct data from old and new games. Each receiver is identified with a unique positive integer jersey number, and each game is identified with a unique integer time. Describe a database supporting the following operations, each in **worst-case** $O(\log n)$ time, where n is the number of games in the database at the time of the operation. Assume that n is always larger than the number of receivers on the team.

<code>record(g, r, p)</code>	record p points for jersey r in game g
<code>clear(g, r)</code>	remove any record that jersey r played in game g
<code>ranked_receiver(k)</code>	return the jersey with the k^{th} highest performance

Solution: First, observe that operations require finding and modifying records for a receiver given a jersey number in $O(\log n)$ time, so maintain dictionary containing the receivers keyed by unique jersey (call it the jersey dictionary). Since we need to achieve worst-case $O(\log n)$ running time, we cannot afford the expected performance of a hash table, so we implement the dictionary using a Set AVL tree.

For each receiver, we will need to find and update their games by game ID, so for each receiver in the jersey dictionary, we will maintain a pointer to their own Set AVL tree containing that receiver's games keyed by game ID (call this a receiver's game dictionary). With each receiver's game dictionary, we will maintain the number of games they've played and the total number of points they've scored to date. We can compare the performance of two jerseys from their respective number of games and points via cross multiplication.

Lastly, to find the k^{th} highest performing receiver, we maintain a separate Set AVL tree on the receivers keyed by performance, augmenting each node with the size of its subtree (call this the performance tree). We showed in lecture how to maintain subtree size in $O(1)$ time, so we can maintain this augmentation. Each node of the jersey dictionary will store a cross-linking pointer to the node in the performance tree corresponding to that player. Since we use Set AVL trees for all data structures, Set operations run in worst-case $O(\log n)$ time.

To implement `record(g, r, p)`, find player r 's game dictionary D in the receiver dictionary in $O(\log n)$ time. If game g is in D , update its stored points to p in $O(\log n)$ time and update the total number of points stored with r 's game dictionary in $O(1)$ time. Otherwise, insert the record of game g into D in $O(\log n)$ time, and update the number of games and total points stored in $O(1)$ time. The performance of r may have changed, so find the node corresponding to r in the performance tree, remove the receiver's performance from the tree, update its performance, and then reinsert into the tree all in $O(\log n)$ time. This operation maintains the semantics of our data structures in worst-case $O(\log n)$ time.

To implement `clear(g, r)`, find player r 's game dictionary D in the receiver dictionary as before and remove g (assuming it exists). Identically to above, maintain the stored number of games and total points, and update the performance tree together in worst-case $O(\log n)$ time.

To implement `ranked_receiver(k)`, find the k^{th} highest performance in the performance tree by using the subtree size augmentation: if the size of a node's right subtree is k or larger, recursively find in the right subtree; if the size of the node's right subtree is $k - 1$, then return the jersey stored at the current node; otherwise the size of the node's right subtree is $k' < k - 1$, recurse in the node's left subtree to find its subtree's k'^{th} highest performing player. This recursive algorithm only walks down the tree, so it runs in worst-case $O(\log n)$ time.

Problem 4-5. Warming Weather

Gal Ore is a scientist who studies climate. As part of her research, she often needs to query the maximum temperature the earth has observed within a particular date range in history, based on a growing set of measurements which she collects and adds to frequently. Assume that temperatures and dates are integers representing values at some consistent resolution. Help Gal evaluate such range queries efficiently by implementing a database supporting the following operations.

<code>record_temp(t, d)</code>	record a measurement of temperature t on date d
<code>max_in_range(d1, d2)</code>	return max temperature observed between dates d_1 and d_2 inclusive

To solve this problem, we will store temperature measurements in an AVL tree with binary search tree semantics keyed by date, where each node A stores a measurement $A.\text{item}$ with a date property $A.\text{item}.key$ and temperature property $A.\text{item}.temp$.

- (a) To help evaluate the desired range query, we will augment each node with:

$A.\text{max_temp}$, the maximum temperature stored in A 's subtree; and both $A.\text{min_date}$ and $A.\text{max_date}$, the minimum and maximum dates stored in A 's subtree respectively. Describe a $O(1)$ -time algorithm to compute the value of these augmentations on node A , assuming all other nodes in A 's subtree have already been correctly augmented.

Solution: Each of these augmentations can be computed in $O(1)$ time via the algorithms below:

Augmentation $A.\text{max_temp}$ can be computed by taking the max of $A.\text{item}.temp$, $A.\text{left}.\text{max_temp}$, and $A.\text{right}.\text{max_temp}$ (when they exist).

Augmentation $A.\text{min_date}$ is $A.\text{left}.\text{min_date}$ if A has a left child, and $A.\text{item}.key$ otherwise.

Augmentation $A.\text{max_date}$ is $A.\text{right}.\text{max_date}$ if A has a right child, and $A.\text{item}.key$ otherwise.

- (b) A subtree **partially overlaps** an inclusive date range if the subtree contains at least one measurement that is within the range **and** at least one measurement that is outside the range. Given an inclusive date range, prove that for any binary search tree containing measurements keyed by dates, there is at most one node in the tree whose left and right subtrees both partially overlap the range.

Solution: Let's say that a node **branches** on a range if both its left and right subtrees both partially overlap the range. Then the question asks us to prove that at most one node in a binary search tree branches on a given range. First, observe that any node that branches is within the range; otherwise the range would not be continuous. Suppose for contradiction that two distinct nodes p and q branch on range (d_1, d_2) . Let x be the lowest common ancestor of p and q . Since p and q are in range, then x is too since x appears between p and q in the traversal order. At least one of p and q is not x , so without loss of generality, assume p is not x and that p is in the left subtree of x . Then p and x are both in range, but the right subtree of p is between p and x in traversal order and contains keys that are not in range, a contradiction.

- (c) Let `subtree_max_in_range(A, d1, d2)` be the maximum temperature of any measurement stored in node A 's subtree with a date between d_1 and d_2 inclusive (returning `None` if no measurements exist in the range). Assuming the tree has been augmented as in part (a), describe a **recursive** algorithm to compute the value of `subtree_max_in_range(A, d1, d2)`. If h is the height of A 's subtree, your algorithm should run in $O(h)$ time when A partially overlaps the range, and in $O(1)$ time otherwise.

Solution: Given node A and range inclusive (d_1, d_2) , the maximum temperature of any measurement stored in A 's subtree is either the temperature at the node itself, or it is in A 's left or right subtree. Because we have augmented by the min and max dates within each node's subtree, we can evaluate whether the dates in A :

1. is disjoint from the range ($(A.\text{max_date} < d_1) \text{ or } (d_2 < A.\text{min_date})$),
2. fully overlaps the range ($d_1 \leq A.\text{min_date}$ and $A.\text{max_date} \leq d_2$), or
3. otherwise partially overlaps the range,

in $O(1)$ time via a constant number of comparisons. In case (1), no measurements in the subtree are within range, so we can return `None` in $O(1)$ time. In case (2), all measurements in the subtree are within range, so we can return `A.max_temp` in $O(1)$ time, which is correct by definition of the augmentation. Lastly, in case (3), recursively compute the max temperature in range for each of A 's child subtrees and return the max between them and the temperature at A (since the temperature at A must be in range). This algorithm takes at most $O(h)$ time. To see this, observe that the claim in part (b) ensures that in all but one case (3) node, a recursive call in at least one of its child subtrees will take $O(1)$ time. So the shape of the recursive calls will be the union of at most two paths from the root to another node in the tree, touching at most $O(h)$ nodes along the way, doing at most $O(1)$ work at each.

- (d) Describe a database to implement operations `record_temp(t, d)` and `max_in_range(d1, d2)`, each in **worst-case** $O(\log n)$ time, where n is the number of unique dates of measurements stored in the database at the time of the operation.

Solution: Store the measurements in a Set AVL tree keyed by date. Augment the tree as in part (a), which can be maintained at each node in $O(1)$ time from the augmenta-

tions of the node's children, without affecting the running time of the other AVL tree operations.

To implement `record_temp(t, d)`, check to see whether date d already exists in the tree. If it does, delete the measurement from the tree, and keep whichever of the two measurements has higher temperature. Then in either case, insert the new measurement into the tree in worst-case $O(\log n)$ time. This procedure maintains the invariant that the temperature stored at date d is the highest temperature recorded for that day.

To implement `max_in_range(d1, d2)`, we reduce to part (c) by simply returning `subtree_max_in_range(T.root, d1, d2)`, where `T.root` is the stored root node of the tree. This algorithm is correct by (c), and runs in worst-case $O(\log n)$ time since the height of an Set AVL tree is $O(\log n)$.

- (e) Implement your database in the Python class `Temperature_DB` extending the `Set_AVL_Tree` class provided; you will only need to implement parts (a) and (c) from above.

Solution:

```

1  from Set_AVL_Tree import BST_Node, Set_AVL_Tree
2
3  class Measurement:
4      def __init__(self, temp, date):
5          self.key = date
6          self.temp = temp
7
8      def __str__(self): return "%s,%s" % (self.key, self.temp)
9
10 class Temperature_DB_Node(BST_Node):
11     def subtree_update(A):
12         super().subtree_update()
13         A.max_temp = A.item.temp
14         A.min_date = A.max_date = A.item.key
15         if A.left:
16             A.min_date = A.left.min_date
17             A.max_temp = max(A.max_temp, A.left.max_temp)
18         if A.right:
19             A.max_date = A.right.max_date
20             A.max_temp = max(A.max_temp, A.right.max_temp)
21

```

```

22     def subtree_max_in_range(A, d1, d2):
23         if (A.max_date < d1) or (d2 < A.min_date):      return None
24         if (d1 <= A.min_date) and (A.max_date <= d2): return A.max_temp
25         t = None
26         if d1 <= A.item.key <= d2:
27             t = A.item.temp
28         if A.left:
29             t_left = A.left.subtree_max_in_range(d1, d2)
30             if t_left:
31                 if t:    t = max(t, t_left)
32                 else:   t = t_left
33         if A.right:
34             t_right = A.right.subtree_max_in_range(d1, d2)
35             if t_right:
36                 if t:    t = max(t, t_right)
37                 else:   t = t_right
38         return t
39
40 class Temperature_DB(Set_AVL_Tree):
41     def __init__(self):
42         super().__init__(Temperature_DB_Node)
43
44     def record_temp(self, t, d):
45         try:
46             m = self.delete(d)
47             t = max(t, m.temp)
48         except: pass
49         self.insert(Measurement(t, d))
50
51     def max_in_range(self, d1, d2):
52         return self.root.subtree_max_in_range(d1, d2)

```

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

Lecture 8: Binary Heaps

Priority Queue Interface

- Keep track of many items, quickly access/remove the most important
 - Example: router with limited bandwidth, must prioritize certain kinds of messages
 - Example: process scheduling in operating system kernels
 - Example: discrete-event simulation (when is next occurring event?)
 - Example: graph algorithms (later in the course)
- Order items by key = priority so **Set interface** (not Sequence interface)
- Optimized for a particular subset of Set operations:

<code>build(x)</code>	build priority queue from iterable x
<code>insert(x)</code>	add item x to data structure
<code>delete_max()</code>	remove and return stored item with largest key
<code>find_max()</code>	return stored item with largest key
- (Usually optimized for max or min, not both)
- Focus on `insert` and `delete_max` operations: `build` can repeatedly `insert`; `find_max()` can `insert(delete_min())`

Priority Queue Sort

- Any priority queue data structure translates into a sorting algorithm:
 - `build(A)`, e.g., insert items one by one in input order
 - Repeatedly `delete_min()` (or `delete_max()`) to determine (reverse) sorted order
- All the hard work happens inside the data structure
- Running time is $T_{\text{build}} + n \cdot T_{\text{delete_max}} \leq n \cdot T_{\text{insert}} + n \cdot T_{\text{delete_max}}$
- Many sorting algorithms we've seen can be viewed as priority queue sort:

Priority Queue Data Structure	Operations $O(\cdot)$			Priority Queue Sort	
	<code>build(A)</code>	<code>insert(x)</code>	<code>delete_max()</code>	Time	In-place?
Dynamic Array	n	$1_{(a)}$	n	n^2	Y
Sorted Dynamic Array	$n \log n$	n	$1_{(a)}$	n^2	Y
Set AVL Tree	$n \log n$	$\log n$	$\log n$	$n \log n$	N
Goal	n	$\log n_{(a)}$	$\log n_{(a)}$	$n \log n$	Y

Selection Sort
 Insertion Sort
 AVL Sort
 Heap Sort

Priority Queue: Set AVL Tree

- Set AVL trees support `insert(x)`, `find_min()`, `find_max()`, `delete_min()`, and `delete_max()` in $O(\log n)$ time per operation
 - So priority queue sort runs in $O(n \log n)$ time
 - This is (essentially) AVL sort from Lecture 7
 - Can speed up `find_min()` and `find_max()` to $O(1)$ time via subtree augmentation
 - But this data structure is complicated and resulting sort is not in-place
 - Is there a simpler data structure for just priority queue, and in-place $O(n \lg n)$ sort?
YES, binary heap and heap sort
 - Essentially implement a Set data structure on top of a Sequence data structure (array), using what we learned about binary trees
-

Priority Queue: Array

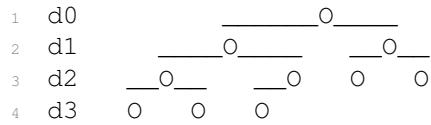
- Store elements in an **unordered** dynamic array
 - `insert(x)`: append x to end in amortized $O(1)$ time
 - `delete_max()`: find max in $O(n)$, swap max to the end and remove
 - `insert` is quick, but `delete_max` is slow
 - Priority queue sort is selection sort! (plus some copying)
-

Priority Queue: Sorted Array

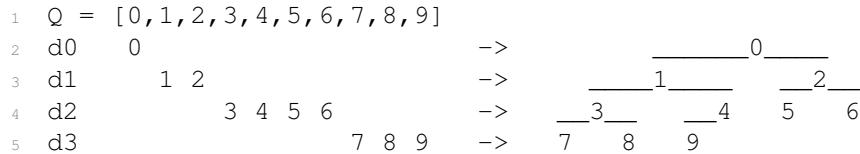
- Store elements in a **sorted** dynamic array
- `insert(x)`: append x to end, swap down to sorted position in $O(n)$ time
- `delete_max()`: delete from end in $O(1)$ amortized
- `delete_max` is quick, but `insert` is slow
- Priority queue sort is insertion sort! (plus some copying)
- Can we find a compromise between these two array priority queue extremes?

Array as a Complete Binary Tree

- **Idea:** interpret an array as a complete binary tree, with maximum 2^i nodes at depth i except at the largest depth, where all nodes are **left-aligned**



- Equivalently, complete tree is filled densely in reading order: root to leaves, left to right
- Perspective: **bijection** between arrays and complete binary trees



- Height of complete tree perspective of array of n item is $\lceil \lg n \rceil$, so **balanced** binary tree
-

Implicit Complete Tree

- Complete binary tree structure can be **implicit** instead of storing pointers
- Root is at index 0
- Compute neighbors by index arithmetic:

$$\begin{aligned} \text{left}(i) &= 2i + 1 \\ \text{right}(i) &= 2i + 2 \\ \text{parent}(i) &= \left\lfloor \frac{i - 1}{2} \right\rfloor \end{aligned}$$

Binary Heaps

- **Idea:** keep larger elements higher in tree, but only locally
 - **Max-Heap Property** at node i : $Q[i] \geq Q[j]$ for $j \in \{\text{left}(i), \text{right}(i)\}$
 - **Max-heap** is an array satisfying max-heap property at all nodes
 - **Claim:** In a max-heap, every node i satisfies $Q[i] \geq Q[j]$ for **all nodes** j in $\text{subtree}(i)$
 - Proof:
 - Induction on $d = \text{depth}(j) - \text{depth}(i)$
 - Base case: $d = 0$ implies $i = j$ implies $Q[i] \geq Q[j]$ (in fact, equal)
 - $\text{depth}(\text{parent}(j)) - \text{depth}(i) = d - 1 < d$, so $Q[i] \geq Q[\text{parent}(j)]$ by induction
 - $Q[\text{parent}(j)] \geq Q[j]$ by Max-Heap Property at $\text{parent}(j)$ □
 - In particular, max item is at root of max-heap
-

Heap Insert

- Append new item x to end of array in $O(1)$ amortized, making it next leaf i in reading order
- `max_heapify_up(i)`: swap with parent until Max-Heap Property
 - Check whether $Q[\text{parent}(i)] \geq Q[i]$ (part of Max-Heap Property at $\text{parent}(i)$)
 - If not, swap items $Q[i]$ and $Q[\text{parent}(i)]$, and recursively `max_heapify_up($\text{parent}(i)$)`
- Correctness:
 - Max-Heap Property guarantees all nodes \geq descendants, except $Q[i]$ might be $>$ some of its ancestors (unless i is the root, so we're done)
 - If swap necessary, same guarantee is true with $Q[\text{parent}(i)]$ instead of $Q[i]$
- Running time: height of tree, so $\Theta(\log n)!$

Heap Delete Max

- Can only easily remove last element from dynamic array, but max key is in root of tree
 - So swap item at root node $i = 0$ with last item at node $n - 1$ in heap array
 - $\text{max_heapify_down}(i)$: swap root with larger child until Max-Heap Property
 - Check whether $Q[i] \geq Q[j]$ for $j \in \{\text{left}(i), \text{right}(i)\}$ (Max-Heap Property at i)
 - If not, swap $Q[i]$ with $Q[j]$ for child $j \in \{\text{left}(i), \text{right}(i)\}$ with maximum key, and recursively $\text{max_heapify_down}(j)$
 - Correctness:
 - Max-Heap Property guarantees all nodes \geq descendants, except $Q[i]$ might be $<$ some descendants (unless i is a leaf, so we're done)
 - If swap is necessary, same guarantee is true with $Q[j]$ instead of $Q[i]$
 - Running time: height of tree, so $\Theta(\log n)!$
-

Heap Sort

- Plugging max-heap into priority queue sort gives us a new sorting algorithm
 - Running time is $O(n \log n)$ because each `insert` and `delete_max` takes $O(\log n)$
 - But often include two improvements to this sorting algorithm:
-

In-place Priority Queue Sort

- Max-heap Q is a prefix of a larger array A , remember how many items $|Q|$ belong to heap
- $|Q|$ is initially zero, eventually $|A|$ (after inserts), then zero again (after deletes)
- `insert()` absorbs next item in array at index $|Q|$ into heap
- `delete_max()` moves max item to end, then abandons it by decrementing $|Q|$
- In-place priority queue sort with Array is exactly Selection Sort
- In-place priority queue sort with Sorted Array is exactly Insertion Sort
- In-place priority queue sort with binary Max Heap is **Heap Sort**

Linear Build Heap

- Inserting n items into heap calls `max_heapify_up(i)` for i from 0 to $n - 1$ (root down):

$$\text{worst-case swaps} \approx \sum_{i=0}^{n-1} \text{depth}(i) = \sum_{i=0}^{n-1} \lg i = \lg(n!) \geq (n/2) \lg(n/2) = \Omega(n \lg n)$$

- **Idea!** Treat full array as a complete binary tree from start, then `max_heapify_down(i)` for i from $n - 1$ to 0 (leaves up):

$$\text{worst-case swaps} \approx \sum_{i=0}^{n-1} \text{height}(i) = \sum_{i=0}^{n-1} (\lg n - \lg i) = \lg \frac{n^n}{n!} = \Theta\left(\lg \frac{n^n}{\sqrt{n}(n/e)^n}\right) = O(n)$$

- So can build heap in $O(n)$ time
 - (Doesn't speed up $O(n \lg n)$ performance of heap sort)
-

Sequence AVL Tree Priority Queue

- Where else have we seen linear build time for an otherwise logarithmic data structure? Sequence AVL Tree!
 - Store items of priority queue in Sequence AVL Tree in **arbitrary order** (insertion order)
 - Maintain max (and/or min) augmentation:
`node.max` = pointer to node in subtree of `node` with maximum key
 - This is a subtree property, so constant factor overhead to maintain
 - `find_min()` and `find_max()` in $O(1)$ time
 - `delete_min()` and `delete_max()` in $O(\log n)$ time
 - `build(A)` in $O(n)$ time
 - Same bounds as binary heaps (and more)
-

Set vs. Multiset

- While our Set interface assumes no duplicate keys, we can use these Sets to implement Multisets that allow items with duplicate keys:
 - Each item in the Set is a Sequence (e.g., linked list) storing the Multiset items with the same key, which is the key of the Sequence
- In fact, without this reduction, binary heaps and AVL trees work directly for duplicate-key items (where e.g. `delete_max` deletes *some* item of maximum key), taking care to use \leq constraints (instead of $<$ in Set AVL Trees)

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

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).

```

1  class PriorityQueue:
2      def __init__(self):
3          self.A = []
4
5      def insert(self, x):
6          self.A.append(x)
7
8      def delete_max(self):
9          if len(self.A) < 1:
10              raise IndexError('pop from empty priority queue')
11          return self.A.pop()                      # NOT correct on its own!
12
13     @classmethod
14     def sort(Queue, A):
15         pq = Queue()                           # make empty priority queue
16         for x in A:                          # n x T_insert
17             pq.insert(x)
18         out = [pq.delete_max() for _ in A]    # n x T_delete_max
19         out.reverse()
20         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.

```

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

1 class PQ_SortedArray(PriorityQueue):
2     # PriorityQueue.delete_max already correct: pop from end of self.A
3     def insert(self, *args):                  # O(n)
4         super().insert(*args)                # append to end of array
5         i, A = len(self.A) - 1, self.A    # restore array ordering
6         while 0 < i and A[i + 1].key < A[i].key:
7             A[i + 1], A[i] = A[i], A[i + 1]
8             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.

```

1  class PQ_Heap(PriorityQueue):
2      def insert(self, *args):          # O(log n)
3          super().insert(*args)        # append to end of array
4          n, A = self.n, self.A
5          max_heapify_up(A, n, n - 1)
6
7      def delete_max():               # O(log n)
8          n, A = self.n, self.A
9          A[0], A[n] = A[n], A[0]
10         max_heapify_down(A, n, 0)
11         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.

```

1  def parent(i):
2      p = (i - 1) // 2
3      return p if 0 < i else i
4
5  def left(i, n):
6      l = 2 * i + 1
7      return l if l < n else i
8
9  def right(i, n):
10     r = 2 * i + 2
11     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.

```

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

1 def max_heapify_down(A, n, p):          # T(p) = O(log n - log p)
2     l, r = left(p, n), right(p, n)    # O(1) indices of children (or p)
3     c = l if A[r].key < A[l].key else r # O(1) index of largest child
4     if A[p].key < A[c].key:            # O(1) compare
5         A[c], A[p] = A[p], A[c]       # O(1) swap child
6         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.

```

1 def build_max_heap(A):
2     n = len(A)
3     for i in range(n // 2, -1, -1): # O(n) loop backward over array
4         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: $n! = \Theta(\sqrt{n}(n/e)^n)$.

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

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.

```

1  class PriorityQueue:
2      def __init__(self, A):
3          self.n, self.A = 0, A
4
5      def insert(self):           # absorb element A[n] into the queue
6          if not self.n < len(self.A):
7              raise IndexError('insert into full priority queue')
8          self.n += 1
9
10     def delete_max(self):       # remove element A[n - 1] from the queue
11         if self.n < 1:
12             raise IndexError('pop from empty priority queue')
13         self.n -= 1           # NOT correct on its own!
14
15     @classmethod
16     def sort(Queue, A):
17         pq = Queue(A)        # make empty priority queue
18         for i in range(len(A)): # n x T_insert
19             pq.insert()
20         for i in range(len(A)): # n x T_delete_max
21             pq.delete_max()
22         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>

¹Recall 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.

```
1 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 i th integer of the unsorted input array is the j th 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 $A[0], \dots, A[i-1]$ 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 p th 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 q th 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 $A[0], \dots, A[k - 1]$. Then, repeatedly, insert the next item from A into H , and then store $H.\text{delete_min}()$ as the next entry in sorted order. So we first call $H.\text{insert}(A[k])$ followed by $B[0] = H.\text{delete_min}()$; the next iteration calls $H.\text{insert}(A[k+1])$ and $B[1] = H.\text{delete_min}()$; 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 i th smallest entry in array A must be one of $A[0], A[1], \dots, A[i+k]$ 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 $B[0], \dots, B[i-1]$ are correct (by induction), this means the i th smallest value is still in H while all smaller values have already been removed, so this i th smallest value is in fact $H.\text{delete_min}()$, 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.

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

Lecture 9: Breadth-First Search

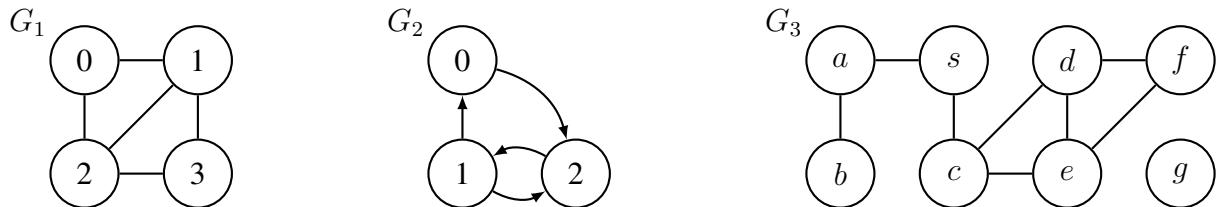
New Unit: Graphs!

- Quiz 1 next week covers lectures L01 - L08 on Data Structures and Sorting
- Today, start new unit, lectures L09 - L14 on Graph Algorithms

Graph Applications

- Why? Graphs are everywhere!
- any network system has direct connection to graphs
- e.g., road networks, computer networks, social networks
- the state space of any discrete system can be represented by a transition graph
- e.g., puzzle & games like Chess, Tetris, Rubik's cube
- e.g., application workflows, specifications

Graph Definitions



- Graph $G = (V, E)$ is a set of vertices V and a set of pairs of vertices $E \subseteq V \times V$.
- **Directed** edges are ordered pairs, e.g., (u, v) for $u, v \in V$
- **Undirected** edges are unordered pairs, e.g., $\{u, v\}$ for $u, v \in V$ i.e., (u, v) and (v, u)
- In this class, we assume all graphs are **simple**:
 - **edges are distinct**, e.g., (u, v) only occurs once in E (though (v, u) may appear), and
 - edges are **pairs of distinct vertices**, e.g., $u \neq v$ for all $(u, v) \in E$
 - Simple implies $|E| = O(|V|^2)$, since $|E| \leq \binom{|V|}{2}$ for undirected, $\leq 2 \binom{|V|}{2}$ for directed

Neighbor Sets/Adjacencies

- The **outgoing neighbor set** of $u \in V$ is $\text{Adj}^+(u) = \{v \in V \mid (u, v) \in E\}$
- The **incoming neighbor set** of $u \in V$ is $\text{Adj}^-(u) = \{v \in V \mid (v, u) \in E\}$
- The **out-degree** of a vertex $u \in V$ is $\deg^+(u) = |\text{Adj}^+(u)|$
- The **in-degree** of a vertex $u \in V$ is $\deg^-(u) = |\text{Adj}^-(u)|$
- For undirected graphs, $\text{Adj}^-(u) = \text{Adj}^+(u)$ and $\deg^-(u) = \deg^+(u)$
- Dropping superscript defaults to outgoing, i.e., $\text{Adj}(u) = \text{Adj}^+(u)$ and $\deg(u) = \deg^+(u)$

Graph Representations

- To store a graph $G = (V, E)$, we need to store the outgoing edges $\text{Adj}(u)$ for all $u \in V$
- First, need a Set data structure Adj to map u to $\text{Adj}(u)$
- Then for each u , need to store $\text{Adj}(u)$ in another data structure called an **adjacency list**
- Common to use **direct access array** or **hash table** for Adj , since want lookup fast by vertex
- Common to use **array** or **linked list** for each $\text{Adj}(u)$ since usually only iteration is needed¹
- For the common representations, Adj has size $\Theta(|V|)$, while each $\text{Adj}(u)$ has size $\Theta(\deg(u))$
- Since $\sum_{u \in V} \deg(u) \leq 2|E|$ by handshaking lemma, graph storable in $\Theta(|V| + |E|)$ space
- Thus, for algorithms on graphs, **linear time** will mean $\Theta(|V| + |E|)$ (linear in size of graph)

Examples

- Examples 1 and 2 assume vertices are labeled $\{0, 1, \dots, |V| - 1\}$, so can use a direct access array for Adj , and store $\text{Adj}(u)$ in an array. Example 3 uses a hash table for Adj .

Ex 1 (Undirected)	Ex 2 (Directed)	Ex 3 (Undirected)
$G1 = [$	$G2 = [$	$G3 = \{$
$[2, 1], \ # 0$	$[2], \ # 0$	$a: [s, b], \ b: [a],$
$[2, 0, 3], \ # 1$	$[2, 0], \ # 1$	$s: [a, c], \ c: [s, d, e],$
$[1, 3, 0], \ # 2$	$[1], \ # 2$	$d: [c, e, f], \ e: [c, d, f],$
$[1, 2], \ # 3$]	$f: [d, e], \ g: []$
]		}

- Note that in an undirected graph, connections are symmetric as every edge is outgoing twice

¹A hash table for each $\text{Adj}(u)$ can allow checking for an edge $(u, v) \in E$ in $O(1)_{(e)}$ time

Paths

- A **path** is a sequence of vertices $p = (v_1, v_2, \dots, v_k)$ where $(v_i, v_{i+1}) \in E$ for all $1 \leq i < k$.
- A path is **simple** if it does not repeat vertices²
- The **length** $\ell(p)$ of a path p is the number of edges in the path
- The **distance** $\delta(u, v)$ from $u \in V$ to $v \in V$ is the minimum length of any path from u to v , i.e., the length of a **shortest path** from u to v
(by convention, $\delta(u, v) = \infty$ if u is not connected to v)

Graph Path Problems

- There are many problems you might want to solve concerning paths in a graph:

- **SINGLE_PAIR_REACHABILITY(G, s, t)**: is there a path in G from $s \in V$ to $t \in V$?
- **SINGLE_PAIR_SHORTEST_PATH(G, s, t)**: return distance $\delta(s, t)$, and a shortest path in $G = (V, E)$ from $s \in V$ to $t \in V$
- **SINGLE_SOURCE_SHORTEST_PATHS(G, s)**: return $\delta(s, v)$ for all $v \in V$, and a **shortest-path tree** containing a shortest path from s to every $v \in V$ (defined below)

- Each problem above is **at least as hard** as every problem above it
(i.e., you can use a black-box that solves a lower problem to solve any higher problem)
- We won't show algorithms to solve all of these problems
- Instead, show one algorithm that solves the **hardest** in $O(|V| + |E|)$ time!

Shortest Paths Tree

- How to return a shortest path from source vertex s for every vertex in graph?
- Many paths could have length $\Omega(|V|)$, so returning every path could require $\Omega(|V|^2)$ time
- Instead, for all $v \in V$, store its **parent** $P(v)$: second to last vertex on a shortest path from s
- Let $P(s)$ be null (no second to last vertex on shortest path from s to s)
- Set of parents comprise a **shortest paths tree** with $O(|V|)$ size!
(i.e., reversed shortest paths back to s from every vertex reachable from s)

²A path in 6.006 is a “walk” in 6.042. A “path” in 6.042 is a simple path in 6.006.

Breadth-First Search (BFS)

- How to compute $\delta(s, v)$ and $P(v)$ for all $v \in V$?
 - Store $\delta(s, v)$ and $P(v)$ in Set data structures mapping vertices v to distance and parent
 - (If no path from s to v , do not store v in P and set $\delta(s, v)$ to ∞)
 - **Idea!** Explore graph nodes in increasing order of distance
 - **Goal:** Compute **level sets** $L_i = \{v \mid v \in V \text{ and } d(s, v) = i\}$ (i.e., all vertices at distance i)
 - Claim: Every vertex $v \in L_i$ must be adjacent to a vertex $u \in L_{i-1}$ (i.e., $v \in \text{Adj}(u)$)
 - Claim: No vertex that is in L_j for some $j < i$, appears in L_i
 - **Invariant:** $\delta(s, v)$ and $P(v)$ have been computed correctly for all v in any L_j for $j < i$
-

- Base case ($i = 1$): $L_0 = \{s\}$, $\delta(s, s) = 0$, $P(s) = \text{None}$
 - Inductive Step: To compute L_i :
 - for every vertex u in L_{i-1} :
 - * for every vertex $v \in \text{Adj}(u)$ that does not appear in any L_j for $j < i$:
 - . add v to L_i , set $\delta(s, v) = i$, and set $P(v) = u$
 - Repeatedly compute L_i from L_j for $j < i$ for increasing i until L_i is the empty set
 - Set $\delta(s, v) = \infty$ for any $v \in V$ for which $\delta(s, v)$ was not set
-
- Breadth-first search correctly computes all $\delta(s, v)$ and $P(v)$ by induction
 - Running time analysis:
 - Store each L_i in data structure with $\Theta(|L_i|)$ -time iteration and $O(1)$ -time insertion (i.e., in a dynamic array or linked list)
 - Checking for a vertex v in any L_j for $j < i$ can be done by checking for v in P
 - Maintain δ and P in Set data structures supporting dictionary ops in $O(1)$ time (i.e., direct access array or hash table)
 - Algorithm adds each vertex u to ≤ 1 level and spends $O(1)$ time for each $v \in \text{Adj}(u)$
 - Work upper bounded by $O(1) \times \sum_{u \in V} \deg(u) = O(|E|)$ by handshake lemma
 - Spend $\Theta(|V|)$ at end to assign $\delta(s, v)$ for vertices $v \in V$ not reachable from s
 - So breadth-first search runs in linear time! $O(|V| + |E|)$
 - Run breadth-first search from s in the graph in Example 3

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

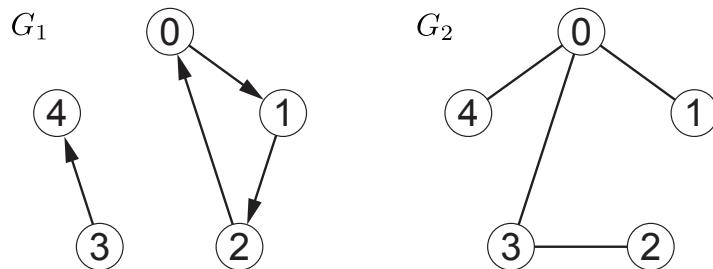
Recitation 9

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 **graphically**, 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 .

$$\begin{array}{lll} G_1 = (V_1, E_1) & V_1 = \{0, 1, 2, 3, 4\} & E_1 = \{(0, 1), (1, 2), (2, 0), (3, 4)\} \\ G_2 = (V_2, E_2) & V_2 = \{0, 1, 2, 3, 4\} & E_2 = \{\{0, 1\}, \{0, 3\}, \{0, 4\}, \{2, 3\}\} \end{array}$$



A **path**¹ in a graph is a sequence of vertices (v_0, \dots, 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.

¹These 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 $\text{Adj}(u)$ 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 $\text{Adj}(u)$. Then, it is common to store each adjacency list $\text{Adj}(u)$ 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.

```

1   A1 = [[1],           A2 = [[1, 4, 3],          # 0
2       [2],           [0],             # 1
3       [0],           [3],             # 2
4       [4],           [0, 2],          # 3
5       []]]           [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 $\text{Adj}(u)$, using Python dictionaries:

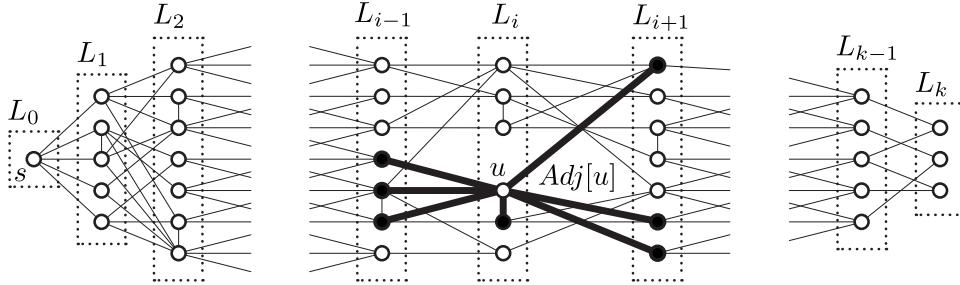
```

1   S1 = {0: {1},           S2 = {0: {1, 3, 4},          # 0
2       1: {2},           1: {0},             # 1
3       2: {0},           2: {3},             # 2
4       3: {4}}           3: {0, 2},          # 3
5                           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):
2     parent = [None for v in Adj]
3     parent[s] = s
4     level = [[s]]
5     while 0 < len(level[-1]):
6         level.append([])
7         for u in level[-2]:
8             for v in Adj[u]:
9                 if parent[v] is None:
10                     parent[v] = u
11                     level[-1].append(v)
12
# Adj: adjacency list, s: starting vertex
# O(V) (use hash if unlabeled)
# O(1) root
# O(1) initialize levels
# O(?) last level contains vertices
# O(1) amortized, make new level
# O(?) loop over last full level
# O(|Adj[u]|) loop over neighbors
# O(1) parent not yet assigned
# O(1) assign parent from level[-2]
# O(1) amortized, add to border
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.

1	P1 = [0,	L1 = [[0],	P2 = [0,	L2 = [[0],	# 0
2	0,	[1],	0,	[1,3,4],	# 1
3	1,	[2],	3,	[2],	# 2
4	None,	[]]	0,	[]]	# 3
5	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.

```

1 def unweighted_shortest_path(Adj, s, t):
2     parent = bfs(Adj, s)                      # O(V + E) BFS tree from s
3     if parent[t] is None:                     # O(1) t reachable from s?
4         return None                          # O(1) no path
5     i = t                                     # O(1) label of current vertex
6     path = [t]                                # O(1) initialize path
7     while i != s:                            # O(V) walk back to s
8         i = parent[i]                         # O(1) move to parent
9         path.append(i)                        # O(1) amortized add to path
10    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.

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

Quiz 1 Review

High Level

- Need to solve large problems n with constant-sized code, **correctly** and **efficiently**
- Analyzing running time: **How to count?**
 - **Asymptotics**
 - **Recurrences** (substitution, tree method, Master Theorem)
 - **Model of computation:** Word-RAM, Comparison
- How to solve an algorithms problem
 - Reduce to a problem you know how to solve
 - * Use an algorithm you know (e.g. **sort**)
 - * Use a data structure you know (e.g. **search**)
 - Design a new recursive algorithm (harder, mostly in 6.046)
 - * Brute Force
 - * Decrease & Conquer
 - * Divide & Conquer (like merge sort)
 - * Dynamic Programming (later in 6.006!)
 - * Greedy/Incremental

Algorithm: Sorting

Reduce your problem to a problem you already know how to solve using known algorithms. You should know **how** each of these sorting algorithms are implemented, as well as be able to **choose** the right algorithm for a given task.

Algorithm	Time $O(\cdot)$	In-place?	Stable?	Comments
Insertion Sort	n^2	Y	Y	$O(nk)$ for k -proximate
Selection Sort	n^2	Y	N	$O(n)$ swaps
Merge Sort	$n \log n$	N	Y	stable, optimal comparison
AVL Sort	$n \log n$	N	Y	good if also need dynamic
Heap Sort	$n \log n$	Y	N	low space, optimal comparison
Counting Sort	$n + u$	N	Y	$O(n)$ when $u = O(n)$
Radix Sort	$n + n \log_n u$	N	Y	$O(n)$ when $u = O(n^c)$

Data Structures

Reduce your problem to using a data structure storing a set of items, supporting certain search and dynamic operations efficiently. You should know **how** each of these data structures implement the operations they support, as well as be able to **choose** the right data structure for a given task.

Sequence data structures support **extrinsic** operations that maintain, query, and modify an externally imposed order on items.

Sequence Data Structure	Operations $O(\cdot)$				
	Container	Static	Dynamic		
		build(x)	get_at(i) set_at(i, x)	insert_first(x) delete_first()	insert_last(x) delete_last()
Array	n	1	n	n	n
Linked List	n	n	1	n	n
Dynamic Array	n	1	n	$1_{(a)}$	n
Sequence AVL	n	$\log n$	$\log n$	$\log n$	$\log n$

Set data structures support **intrinsic** operations that maintain, query, and modify a set of items based on what the items are, i.e., based on the **unique key** associated with each item.

Set 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	u	1	1	u	u
Hash Table	$n_{(e)}$	$1_{(e)}$	$1_{(a)(e)}$	n	n
Set AVL	$n \log n$	$\log n$	$\log n$	$\log n$	$\log n$

Priority Queues support a limited number of Set operations.

Priority Queue Data Structure	Operations $O(\cdot)$			
	build(x)	insert(x)	delete_max()	find_max()
Dynamic Array	n	$1_{(a)}$	n	n
Sorted Dyn. Array	$n \log n$	n	$1_{(a)}$	1
Set AVL	$n \log n$	$\log n$	$\log n$	$\log n$
Binary Heap	n	$\log n_{(a)}$	$\log n_{(a)}$	1

Problem Solving

Testing Strategies

- Read every problem first, rank them in the order of your confidence
- For most problems, you can receive $\geq 50\%$ of points in two sentences or less
- Probably better to do half the problems well than all the problems poorly

Types of problems

Type	Internals	Externals	Tests understanding of:
Mechanical	Y	N	how core material works
Reduction	N	Y	how to apply core material
Modification	Y	Y	how to adapt core material (augmentation, divide & conquer, amortization, etc.)

Questions to ask:

- Is this a Mechanical, Reduction, or Modification type problem?
- Is this problem about data structures? sorting? both?
- If data structures, do you need to support **Sequence** ops? **Set** ops? both?
- If stuck, is there an easy way to get a correct but inefficient algorithm?

Question yourself if you are:

- Trying to compute decimals, rationals, or real numbers
- Using Radix sort for every answer
- Augmenting a binary tree with something other than a subtree property

Data Structures Problems

- First solve using Sorting or Set/Sequence interfaces, choose algorithm/data structure after
- Describe all data structure(s) used (including **what data they store**) and their invariants
- Implement **every operation** we ask for in terms of your data structures
- Separate and label parts of your solution!

Problem 1. Restaurant Lineup (S19 Q1)

Popular restaurant Criminal Seafood does not take reservations, but maintains a wait list where customers who have been on the wait list longer are seated earlier. Sometimes customers decide to eat somewhere else, so the restaurant must remove them from the wait list. Assume each customer has a different name, and no two customers are added to the wait list at the exact same time. Design a database to help Criminal Seafood maintain its wait list supporting the following operations, each in $O(1)$ time. State whether each operation running time is worst-case, amortized, and/or expected.

<code>build()</code>	initialize an empty database
<code>add_name (x)</code>	add name x to the back of the wait list
<code>remove_name (x)</code>	remove name x from the wait list
<code>seat ()</code>	remove and return the name of the customer from the front of the wait list

Solution: Maintain a doubly-linked list containing customers on the wait list in order, maintaining a pointer to the front of the linked list corresponding to the front of the wait list, and a pointer to the back of the linked list corresponding to the back of the wait list. Also maintain a hash table mapping each customer name to the linked list node containing that customer. To implement `add_name (x)`, create a new linked list node containing name x and add it to the back of the linked list in worst-case $O(1)$ time. Then add name x to the hash table pointing to the newly created node in amortized expected $O(1)$ time. To implement `remove_name (x)`, lookup name x in the hash table in and remove the mapped node from the linked list in expected $O(1)$ time. Lastly, to implement `seat ()`, remove the node from the front of the linked list containing name x , remove name x from the hash table, and then return x , in amortized expected $O(1)$ time.

Problem 2. Rainy Research (S19 Q1)

Mether Wan is a scientist who studies global rainfall. Mether often receives data measurements from a large set of deployed sensors. Each collected data measurement is a triple of integers (r, ℓ, t) , where r is a positive amount of rainfall measured at latitude ℓ at time t . The **peak rainfall** at latitude ℓ **since** time t is the maximum rainfall of any measurement at latitude ℓ measured at a time greater than or equal to t (or zero if no such measurement exists). Describe a database that can store Mether's sensor data and support the following operations, each in worst-case $O(\log n)$ time where n is the number of measurements in the database at the time of the operation.

<code>build()</code>	initialize an empty database
<code>record_data(r, ℓ, t)</code>	add a rainfall measurement r at latitude ℓ at time t
<code>peak_rainfall(ℓ, t)</code>	return the peak rainfall at latitude ℓ since time t

Solution: Maintain a Set AVL tree L storing distinct measurement latitudes, where each latitude ℓ maps to a rainfall Set AVL tree $R(\ell)$ containing all measurement triples with latitude ℓ , keyed by time. We only store nodes associated with measurements, so the height of each Set AVL tree is bounded by $O(\log n)$. For each rainfall tree, augment each node p with the maximum rainfall $p.m$ of any measurement within p 's subtree. This augmentation can be maintained in constant time at a node p by taking the maximum of the rainfall at p and the augmented maximums of p 's left and right children (if they exist); thus this augmentation can be maintained without effecting the asymptotic running time of standard AVL tree operations.

To implement `record_data(r, ℓ, t)`, search L for latitude ℓ in worst-case $O(\log n)$ time. If ℓ does not exist in L , insert a new node corresponding to ℓ mapping to a new empty rainfall Set AVL tree, also in $O(\log n)$ time. In either case, insert the measurement triple to $R(\ell)$, for a total running time of worst-case $O(\log n)$.

To implement `peak_rainfall(ℓ, t)`, search L for latitude ℓ in worst-case $O(\log n)$ time. If ℓ does not exist, return zero. Otherwise, perform a one-sided range query on $R(\ell)$ to find the peak rainfall at latitude ℓ since time t . Specifically, let $\text{peak}(v, t)$ be the maximum rainfall of any measurement in node v 's subtree measured at time $\geq t$ (or zero if v is not a node):

$$\text{peak}(v, t) = \begin{cases} \max \{v.item.r, v.right.m, \text{peak}(v.left, t)\} & \text{if } v.t \geq t \\ \text{peak}(v.right, t) & \text{if } v.t < t \end{cases}.$$

Then peak rainfall is simply $\text{peak}(v, t)$ with v being the root of the tree, which can be computed using at most $O(\log n)$ recursive calls. So this operation runs in worst-case $O(\log n)$ time.

Note, this problem can also be solved where each latitude AVL tree is keyed by rainfall, augmenting nodes with maximum time in subtree. We leave this as an exercise to the reader.

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

Solution: Quiz 1

- Do not open this quiz booklet until directed to do so. Read all the instructions on this page.
- When the quiz begins, write your name on the top of every page of this quiz booklet.
- You have 120 minutes to earn a maximum of 120 points. Do not spend too much time on any one problem. Skim them all first, and attack them in the order that allows you to make the most progress.
- **You are allowed one double-sided letter-sized sheet with your own notes.** No calculators, cell phones, or other programmable or communication devices are permitted.
- Write your solutions in the space provided. Pages will be scanned and separated for grading. If you need more space, write “Continued on S1” (or S2, S3) and continue your solution on the referenced scratch page at the end of the exam.
- Do not spend time and paper rederiving facts that we have presented in lecture or recitation. Simply cite them.
- When writing an algorithm, a **clear** description in English will suffice. Pseudo-code is not required. Be sure to argue that your **algorithm is correct**, and analyze the **asymptotic running time of your algorithm**. Even if your algorithm does not meet a requested bound, you **may** receive partial credit for inefficient solutions that are correct.
- **Pay close attention to the instructions for each problem.** Depending on the problem, partial credit may be awarded for incomplete answers.

Problem	Parts	Points
1: Information	2	2
2: Frequentest	2	8
3: Haphazard Heaps	2	10
4: Transforming Trees	2	10
5: Sorting Sock	4	20
6: Triple Sum	1	15
7: Where am i ?	1	15
8: Methane Menace	1	20
9: Vapor Invite	1	20
Total		120

Name: _____

MIT Kerberos Username: _____

Problem 1. [2 points] **Information** (2 parts)

- (a) [1 point] Write your name and email address on the cover page.

Solution: OK!

- (b) [1 point] Write your name at the top of each page.

Solution: OK!

Problem 2. [8 points] **Frequentest** (2 parts)

The following two Python functions correctly solve the problem: given an array X of n positive integers, where the maximum integer in X is k , return the integer that appears the most times in X . Assume: a Python `list` is implemented using a dynamic array; a Python `dict` is implemented using a hash table which randomly chooses hash functions from a universal hash family; and `max(X)` returns the maximum integer in array X in worst-case $O(|X|)$ time. For each function, state its **worst-case** and **expected** running times **in terms of n and k** .

(a) [4 points]

```
def frequentest_a(X):
    k = max(X)
    H = {}
    for x in X:
        H[x] = 0
    best = X[0]
    for x in X:
        H[x] += 1
        if H[x] > H[best]:
            best = x
    return best
```

i. Worst-case:

Solution: $O(n^2)$

ii. Expected:

Solution: $O(n)$

(b) [4 points]

```
def frequentest_b(X):
    k = max(X)
    A = []
    for i in range(k + 1):
        A.append(0)
    best = X[0]
    for x in X:
        A[x] += 1
        if A[x] > A[best]:
            best = x
    return best
```

i. Worst-case:

Solution: $O(n + k)$

ii. Expected:

Solution: $O(n + k)$

Problem 3. [10 points] **Haphazard Heap** (3 parts)

Array $[A, B, C, D, E, F, G, H, I, J]$ represents a **binary min-heap** containing 10 items, where the key of each item is a **distinct** integer. State which item(s) in the array could have the key with:

- (a) the smallest integer

Solution: *A*

- (b) the third smallest integer

Solution: *B, C, D, E, F, G*

Common Mistakes: Incorrectly assuming third smallest must be in either *B* or *C*.

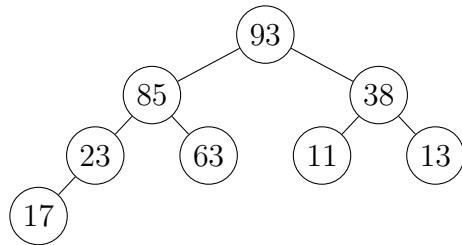
- (c) the largest integer

Solution: *F, G, H, I, J*

Common Mistakes: Thinking the largest integer must be in the bottom level
(instead of in any leaf)

Problem 4. [10 points] **Transforming Trees** (2 parts)

The tree below contains 8 items, where each stored item is an integer which is its own key.



- (a)** [6 points] Suppose the tree drawn above is the implicit tree of a binary max-heap H . State the array representation of H , **first before** and **then after** performing the operation $H.\text{delete_max}()$.

Solution:

Before: [93, 85, 38, 23, 63, 11, 13, 17]

After: [85, 63, 38, 23, 17, 11, 13]

Common Mistakes:

- Not building heap correctly (e.g., confusing heap order with traversal order)
- Not correctly deleting min (by swapping root with last leaf and heapifying down)

- (b)** [4 points] Suppose instead that the original tree drawn above is a Sequence AVL Tree S (note Sequence data structures are zero-indexed). The items in **the leaves** of S in traversal order are (17, 63, 11, 13). Perform operation $S.\text{delete_at}(3)$ on S including any rotations, and then **list the items** stored in **the leaves** of S in traversal order, after the operation has completed. (You do not need to draw the tree.)

Solution: (17, 85, 11, 13)**Common Mistakes:**

- Indexing by heap order instead of traversal order (or using 1-indexing)
- Listing full traversal order (rather than **just the leaves**)
- Deleting the item but not rebalancing the tree to satisfy the AVL Property

Problem 5. [20 points] **Sorting Sock** (4 parts)

At Wog Hearts School of Wizcraft and Witcherdry, n incoming students are sorted into four houses by an ancient magical artifact called the Sorting Sock. The Sorting Sock first sorts the n students by each of the four houses' attributes and then uses the results to make its determinations. For each of the following parts, state and justify what type of sort would be most efficient. (By "efficient", we mean that faster correct algorithms will receive more points than slower ones.)

- (a) [5 points] For House Puffle Huff, students must be sorted by **friend number**, i.e., how many of the other $n - 1$ incoming students they are friends with, which can be determined in $O(1)$ time.

Solution: Friend numbers are non-negative integers less than n , so we can use counting sort to sort the students in worst-case $O(n)$ time. (Radix sort also works with the same running time.) Since we have to compute friend number for each student, any algorithm will take at least $\Omega(n)$ time, so this is optimal.

Common Mistakes: Using a sort that is not $O(n)$

- (b) [5 points] For House Craven Law, students must be sorted by the weight of their books. Book weights cannot be measured precisely, but the Sorting Sock has a **scale** that can determine in $O(1)$ time whether one set of books has total weight greater than, less than, or equal to another set of books.

Solution: A scale weighing is a comparison with a constant number of outcomes, so the comparison sort $\Omega(n \log n)$ lower bound applies. So we cannot do better than by using an worst-case $O(n \log n)$ sorting algorithm, e.g., merge sort, using the scale to compare one student's books against another's.

Common Mistakes: Insufficient justification for why $O(n \log n)$ is optimal

- (c) [5 points] For House Driven Gore, students must be sorted by *bravery*, which can't be directly measured or quantified, but for any set of students, the Sorting Sock can determine the bravest among them in $O(1)$ time, e.g., by presenting the students with a scary situation.

Solution: We can't quantify bravery, so we can't hope to use any integer-based algorithms. However, the Sorting Sock can find a student of maximum bravery in $O(1)$ time, so we repeatedly find and select a bravest student among all previously unselected students in worst-case $O(n)$ time, which is again optimal. (This is priority queue sort, using the Sorting Sock as the priority queue to find the maximum.)

Common Mistakes: Arguing a $\Omega(n \log n)$ lower bound

- (d) [5 points] For House Leather Skin, students must be sorted by their *magical lineage*: how many of a student's ancestors within the previous $3\lceil \log n \rceil + 4$ generations were magical. Recall that humans, magical or not, always have two parents in the previous generation, unlike binary tree nodes which have at most one. Assume the Sorting Sock can compute the magical lineage of a student in $O(1)$ time.

Solution: Each student has at most 2^k ancestors in the k th generation preceding. Thus the number of wizard ancestors will be a non-negative number bounded above by $\sum_{i=1}^{3\lceil \log n \rceil + 4} 2^k < 2^{3(\log n+1)+5} = 2^8 2^{3\log n} = 2^8 n^{3\log 2} = O(n^c)$ for any $c > 3 \log 2$. Thus we can use radix sort to sort the students by their magical lineage in worst-case $O(n)$ time, which is again optimal.

Common Mistakes:

- Claiming $3\lceil \log n \rceil + 4$ ancestors instead of $O(2^{3\lceil \log n \rceil + 4})$
- Saying that $2^{3\lceil \log n \rceil + 4} = O(n)$ and using counting sort

Problem 6. [15 points] **Triple Sum**

Given three arrays A, B, C , each containing n integers, give an $O(n^2)$ -time algorithm to find whether some $a \in A$, some $b \in B$, and some $c \in C$ have zero sum, i.e., $a + b + c = 0$. State whether your running time is worst-case, expected, and/or amortized.

Solution: We present both expected and worst-case solutions (both were accepted for full points).

Expected Time

For each pair of numbers $(a, b) \in A \times B$, store $a + b$ in a hash table H . Then return Yes if $-c$ appears in H for any $c \in C$, and return No otherwise.

Correctness: If any $-c$ appears in H for any $c \in C$, then $-c = a' + b'$ for some $(a', b') \in A \times B$ so $a' + b' + c = 0$. Otherwise, there is no $c \in C$ for which $-c = a' + b'$ for any $(a', b') \in A \times B$.

Running Time: There are $O(n^2)$ pairs in $A \times B$, so inserting them into H takes expected $O(n^2)$ time. Then checking whether each $-c$ appears in H takes expected $O(1)$ time each, and expected $O(n)$ in total. So this algorithm runs in expected $O(n^2)$ time.

Continued on scratch paper S1 for worst-case solution...

Common Mistakes:

- Using counting/radix sort or creating a direct access array (no bound on u so not efficient)
- Saying n insertions into a hash table gives an amortized bound
- Checking all triples in $\Omega(n^3)$ time

Problem 7. [15 points] **Where Am i?**

Given a Sequence AVL Tree T containing n nodes, and a pointer to a node v from T , describe an $O(\log n)$ -time algorithm to return the (zero-indexed) index i of node v in the traversal order of T . (Recall that every node u in a Sequence AVL Tree T stores an item $u.item$, parent $u.parent$, left child $u.left$, right child $u.right$, subtree height $u.height$, and subtree size $u.size$.)

Solution: Our algorithm will be to walk up the tree from v to the root r of the Sequence AVL Tree, counting the nodes preceding v in the traversal order along the way, since the number of nodes preceding v in the tree is equivalent to v 's (zero-indexed) index.

Let $\#_v(u)$ be the number of vertices preceding v in a vertex u 's subtree, where v is in the subtree of u . Then $\#_v(v) = v.left.size$ if v has a left child and zero otherwise; and can be computed in $O(1)$ time. Then, for every ancestor of u starting from v , we compute $\#_v(u.parent)$ from $\#_v(u)$. There are two cases:

- Case 1, u is the left child of $u.parent$: then all the nodes preceding v in the subtree of $u.parent$ are in the subtree of u , so set $\#_v(u.parent) = \#_v(u)$.
- Case 2, u is the right child of $u.parent$: then all nodes in the left subtree of $u.parent$ precede v (as does u), so set $\#_v(u.parent) = 1 + u.parent.left.size + \#_v(u)$.

Then return $\#_v(r)$, since this is the number of nodes preceding v in r 's subtree (i.e., the entire tree). Correctness is argued within the algorithm description. This algorithm spends worst-case $O(1)$ work for each ancestor of v , so since the Sequence AVL Tree is balanced, the number of ancestors is bounded by $O(\log n)$, and the algorithm runs in worst-case $O(\log n)$ time.

Common Mistakes:

- Finding v given i instead of finding i given v
- Breaking early (e.g., as soon as node is a left child, instead of continuing up tree)
- Walking down the tree from root (assuming the way to go to find v)

Problem 8. [20 points] **Methane Menace**

FearBird is a supervillain who has been making small holes in the methane gas pipe network of **mahtoG City**. The network consists of n pipes, each labeled with a distinct positive integer. A hole i is designated by a pair of positive integers (p_i, d_i) , where p_i denotes the label of the pipe containing the hole, and d_i is a positive integer representing the *distance* of the hole from the *front* of pipe p_i . Assume any two holes in the same pipe p_i will be at different distances from the front of p_i . When a new hole (p_i, d_i) is spotted, the city receives a *report* of the hole to keep track of. The city will periodically patch holes using the following priority scheme:

- if each pipe contains at most one hole, patch any hole (if one exists);
- otherwise, among pairs of holes (p_i, d_i) and (p_j, d_j) appearing on the **same pipe**, i.e., $p_i = p_j$, identify any pair with smallest distance $|d_i - d_j|$ between them, and patch one of them.

Describe a database supporting the following operations, where k is the number of recorded but unpatched holes in the network at the time of the operation. State whether your running times are worst-case, expected, and/or amortized.

<code>initialize(H)</code>	Initialize the database with n holes $H = \{(p_0, d_0), \dots, (p_{n-1}, d_{n-1})\}$, with one hole on each pipe, in $O(n)$ time
<code>report(p_i, d_i)</code>	Record existence of a hole in pipe p_i at distance d_i in $O(\log k)$ time
<code>patch()</code>	Patch any hole that follows the priority scheme above in $O(\log k)$ time

Solution: To implement the database, maintain the following data structures:

- A Set AVL tree T_p for each pipe p containing all the unpatched holes in p keyed by hole distance
- A Hash Table D mapping each pipe p to its tree T_p
- A Binary Min Heap Q containing each consecutive pair of holes (p, d_1, d_2) appearing on the same pipe p with key being the distance $|d_2 - d_1|$ between them, and any lonely holes (p, d) (holes that are alone on their pipes) with key ∞ (when multiple stored items have the same key, we store them in a Hash Table keyed by (p, d_1, d_2) or (p, d))
- A Hash Table C mapping each consecutive hole pair (p, d_1, d_2) or lonely hole (p, d) , to their location in Q .

Some parenthetical notes on this solution:

- A Set AVL Tree can be used for C or Q to achieve identical bounds.
- A solution without augmentation was intended, so our solution does not use it. But it is also possible to use augmentation:
 - e.g., to combine C and Q into a single Set AVL Tree keyed the same as C but storing a pointer to the min distance in subtree; or,
 - e.g., on each T_p to maintain the min distance within the pipe (to compute this augmentation efficiently, one would either need to maintain the distance to each hole's successor/predecessor (not a subtree property), or augment by min/max distance in subtree to be correct)

Operation descriptions: Continued on scratch paper S2...

Problem 9. [20 points] **Vapor Invite**

Vapor is an online gaming platform with n users. Each user has a unique positive integer **ID** d_i and an updatable **status**, which can be either active or inactive. Every day, Vapor will post online an **active range**: a pair of positive integers (a, b) with the property that **every user** having an ID d_i contained in the range (i.e., with $a \leq d_i \leq b$) **must be active**. Vapor wants to post an active range containing as many active users as possible, and invite them to play in a special tournament. Describe a database supporting the following **worst-case** operations:

<code>build(D)</code>	Initialize the database with user IDs $D = \{d_0, \dots, d_{n-1}\}$, setting all user statuses initially to active, in $O(n \log n)$ time
<code>toggle_status(d_i)</code>	Toggle the status of the user with ID d_i , e.g., from active to inactive or vice versa, in $O(\log n)$ time
<code>big_active_range()</code>	Return an active range (a, b) containing the largest number of active users possible in $O(1)$ time

Solution: To implement the database, maintain a single Set AVL Tree T containing each user ID and their status, keyed by ID. In addition, augment each node x in T with four subtree properties:

- $x.size$: the number of IDs in the subtree (as discussed in lecture).
- $x.suffix = (d, m)$: the smallest ID d in the subtree for which each of the m IDs $d' \geq d$ in the subtree is active, or None. Computable in $O(1)$ time as either the suffix (d_R, m_R) of the right subtree, or if m_R is equal to the size of the right subtree and x is active, return the suffix (d_L, m_L) of the left subtree but add $m_R + 1$ to m_L (or $(x.key, m_R + 1)$ if left suffix is None).
- $x.prefix = (d, m)$: the largest ID d in the subtree for which each of the m IDs $d' \leq d$ in the subtree is active, or None. Computable in $O(1)$ time as either the prefix (d_L, m_L) of the right subtree, or if m_L is equal to the size of the left subtree and x is active, return the prefix (d_R, m_R) of the right subtree but add $m_L + 1$ to m_R (or $(x.key, m_L + 1)$ if right prefix is None).
- $x.substring = (a, b, m)$: a, b are IDs from the subtree where each of the m IDs d in the subtree with $a \leq d \leq b$ is active and m is maximized. Computable in $O(1)$ time by taking the max of the substring within either left or right subtree, or the substring spanning the two subtrees if x is active. Specifically, consider the substrings of left and right subtrees, (a_L, b_L, m_L) and (a_R, b_R, m_R) respectively, and then if x is active, consider the suffix of the left subtree (d_L, m'_L) and the prefix of the right subtree (d_R, m'_R) . Then depending on which of $(m_L, m'_L + 1 + m'_R, m_R)$ is the largest, return (a_L, b_L, m_L) , $(d_L, d_R, m'_L + 1 + m'_R)$, or (a_R, b_R, m_R) respectively.

To implement `build(D)`, build the Set AVL Tree T in worst-case $O(n \log n)$ time, maintaining the custom augmentations during each insertion.

To implement `toggle_states(di)`, remove d_i from T in worst-case $O(\log n)$ time, toggle its status in $O(1)$ time, and then re-insert d_i into T in worst-case $O(\log n)$ time (again, maintaining augmentations).

To implement `big_active_range()`, simply return the substring augmentation at the root in worst-case $O(1)$ time, which is correct by the definition of our augmentation.

Common Mistakes: See scratch paper S3...

SCRATCH PAPER 1. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write “Continued on S1” on the problem statement’s page.

Solution: (Problem 6 continued...)

Worst-case Solution

Sort A and B increasing using merge sort. We give a two-finger algorithm to determine whether any $a \in A$ and $b \in B$ sum to a given $-c$. Doing this for each $c \in C$ directly determines whether a, b, c sum to 0.

Start with an index $i = 0$ into A and an index $j = n - 1$ into B and repeat the following procedure until either $i = n$ or $j = -1$:

- If $A[i] + B[j] > -c$, increase i .
- If $A[i] + B[j] < -c$, decrease j .
- Otherwise, $A[i] + B[j] = -c$, so return Yes.

If this procedure terminates without returning Yes, return No.

Correctness: We first prove the claim that the loop maintains the invariant that at the start of a loop, that no $A[i'] + B[j'] = -c$ for any $0 \leq i' < i$ or any $n - 1 \geq j' > j$. Proof by induction on $k = i - j$. This invariant is trivially true at the start when $k = i - j = -n + 1$. Assume the claim is true for all $k < k^* = i^* - j^*$. If we are at the start of a loop with $i = i^*$ and $j = j^*$, we cannot have yet returned, so at the start of the previous loop, there are two cases:

- Case 1, $(i, j) = (i^* - 1, j)$: $i^* - 1 - j = k^* - 1 < k^*$, so by induction, no $A[i'] + B[j'] = -c$ for any $i' < i^* - 1$ or $j' > j^*$. But we increased i when $A[i^* - 1] + B[j^*] > -c$, so since $B[j^*] \geq B[j']$ for all $j' > j^*$, then $A[i^* - 1] + B[j'] > -c$ for all $j' > j^*$, restoring the invariant.
- Case 2, $(i, j) = (i^*, j + 1)$: $i^* - (j + 1) = k^* - 1 < k^*$, so by induction, no $A[i'] + B[j'] = -c$ for any $i' < i^*$ or $j' > j^* + 1$. But we decreased j when $A[i^*] + B[j^* + 1] < -c$, so since $A[i^*] \leq A[i']$ for all $i' < i^*$, then $A[i'] + B[j^* + 1] < -c$ for all $i' < i^*$, restoring the invariant.

So the claim is true. Then if the algorithm terminates without returning Yes, either $i = n$ or $j = -1$, so the claim implies that no integer from either A or B respectively can be in a triple with c that sums to zero.

Running Time: Sorting A and B takes worst-case $O(n \log n)$ time. We perform the two-finger algorithm n times. A single two-finger algorithm takes worst-case $O(n)$: every loop does $O(1)$ work and either increases i or decreases j , so since the loop terminates when either $i = n$ or $j = -1$, the loop executes at most $2n = O(n)$ times. So the algorithm runs in worst-case $O(n^2)$ time in total.

SCRATCH PAPER 2. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write “Continued on S2” on the problem statement’s page.

Solution: (Problem 8 continued...)

To implement `initialize(H)`, initialize empty D and C , and then for each $(p, d) \in H$, construct an empty Set AVL Tree T_p , insert d into T_p , insert p into D mapping to T_p . Then build a hash table D_∞ on every $(p, d) \in H$, store it in Q with key ∞ , and then insert each (p, d) into C mapping to D_∞ . Building D_∞ takes expected $O(n)$ time, and for each (p, d) , this procedure takes $O(1)$ time (expected in the case of inserting into D). So it takes expected $O(n)$ time in total, and maintains the invariants of the database directly. (Note that we could not use a Set AVL Tree for D , as it could take $\Omega(n \log n)$ time to construct.)

To implement `report(p, d)`, lookup p in D to find T_p , and then insert d into T_p .

- If d has no predecessor or successor in T_p , then insert (p, d) into Q with key ∞ and insert (p, d) into C mapping to its location in Q .
- Otherwise d has a predecessor or successor in T_p .
 - If d has a predecessor d_1 and successor d_2 , lookup (p, d_1, d_2) in C to find it in Q , and then remove (p, d_1, d_2) from both C and Q .
 - Otherwise, it has one of them d' , so lookup (p, d') in C to find it in Q and then remove (p, d') from both C and Q .
 - In either case, if d has a predecessor d_1 , add (p, d_1, d) to Q with key $|d - d_1|$ and add (p, d_1, d) to C pointing to its location in Q ;
 - and if d has a successor d_2 , add (p, d, d_2) to Q with key $|d_2 - d|$ and add (p, d, d_2) to C pointing to its location in Q .

This procedure does a constant number of worst-case $O(\log k)$ time or amortized expected $O(1)$ time operations, so this operation runs in amortized expected $O(\log k)$ time, and maintains the invariants of the database by removing any consecutive pairs or lonely holes if they are no longer consecutive or lonely, and adding any new consecutive pairs or lonely holes that may have been introduced.

To implement `patch()`, delete the minimum item from Q containing one or two holes on pipe p , remove it from C , lookup p in D to find T_p , and remove the relevant holes from T_p . This procedure does a constant number of worst-case $O(\log k)$ time or amortized expected $O(1)$ time operations, so this operation runs in amortized expected $O(\log k)$ time, and maintains the invariants of the database directly. It is correct because Q exactly implements the requested priority scheme: consecutive pairs with smaller key have higher priority, and will only remove a lonely hole if there are no consecutive pairs having finite key contained in Q .

Common Mistakes: See scratch page S3...

SCRATCH PAPER 3. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write “Continued on S3” on the problem statement’s page.

Common Mistakes: (For Problem 8)

- Taking a minimum in some pipe rather than a minimum over all pipes
- Not prioritizing pipes having more than one hole over those with only one
- Storing an AVL or heap on all pipes which cannot be maintained within the time bounds
- Attempting an augmentation-based solution incorrectly
- Claiming $O(\log n)$ -time insertion into a sorted array
- Initializing a direct access array of non-polynomially bounded size
- Using Sequence AVL Trees instead of Set AVL Trees

Common Mistakes: (For Problem 9)

- Assuming the max range always goes through the root
- Claiming a substring augmentation without showing how to maintain it in $O(1)$ time
- Using 2 AVL Trees, one of active and one of inactive (doesn’t help to find largest range).
- Substring augmentation maintenance doesn’t consider substrings containing the root

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

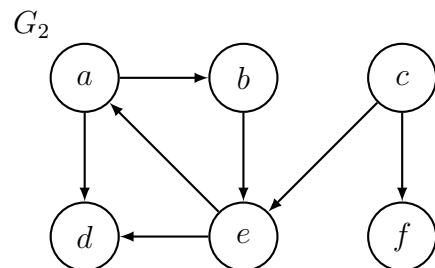
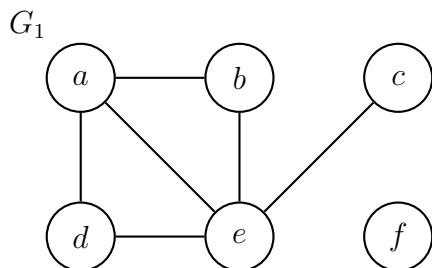
For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

Lecture 10: Depth-First Search

Previously

- Graph definitions (directed/undirected, simple, neighbors, degree)
- Graph representations (Set mapping vertices to adjacency lists)
- Paths and simple paths, path length, distance, shortest path
- Graph Path Problems
 - Single_Pair_Reachability (G, s, t)
 - Single_Source_Reachability (G, s)
 - Single_Pair_Shortest_Path (G, s, t)
 - Single_Source_Shortest_Paths (G, s) (SSSP)
- Breadth-First Search (BFS)
 - algorithm that solves Single Source Shortest Paths
 - with appropriate data structures, runs in $O(|V| + |E|)$ time (linear in input size)

Examples



Depth-First Search (DFS)

- Searches a graph from a vertex s , similar to BFS
 - Solves Single Source Reachability, **not** SSSP. Useful for solving other problems (later!)
 - Return (not necessarily shortest) parent tree of parent pointers back to s
-
- **Idea!** Visit outgoing adjacencies recursively, but never revisit a vertex
 - i.e., follow any path until you get stuck, backtrack until finding an unexplored path to explore
 - $P(s) = \text{None}$, then run $\text{visit}(s)$, where
 - $\text{visit}(u)$:
 - for every $v \in \text{Adj}(u)$ that does not appear in P :
 - * set $P(v) = u$ and recursively call $\text{visit}(v)$
 - (DFS finishes visiting vertex u , for use later!)
-
- **Example:** Run DFS on G_1 and/or G_2 from a

Correctness

- **Claim:** DFS visits v and correctly sets $P(v)$ for every vertex v reachable from s
- **Proof:** induct on k , for claim on only vertices within distance k from s
 - Base case ($k = 0$): $P(s)$ is set correctly for s and s is visited
 - Inductive step: Consider vertex v with $\delta(s, v) = k' + 1$
 - Consider vertex u , the second to last vertex on some shortest path from s to v
 - By induction, since $\delta(s, u) = k'$, DFS visits u and sets $P(u)$ correctly
 - While visiting u , DFS considers $v \in \text{Adj}(u)$
 - Either v is in P , so has already been visited, or v will be visited while visiting u
 - In either case, v will be visited by DFS and will be added correctly to P

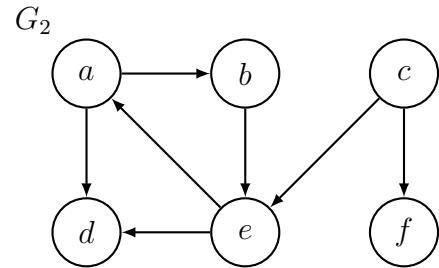
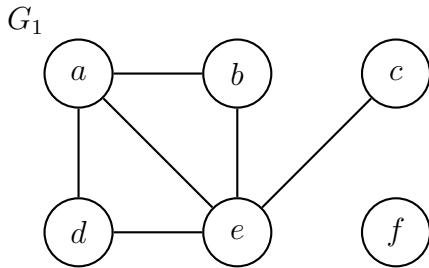
□

Running Time

- Algorithm visits each vertex u at most once and spends $O(1)$ time for each $v \in \text{Adj}(u)$
- Work upper bounded by $O(1) \times \sum_{u \in V} \deg(u) = O(|E|)$
- Unlike BFS, not returning a distance for each vertex, so DFS runs in $O(|E|)$ time

Full-BFS and Full-DFS

- Suppose want to explore entire graph, not just vertices reachable from one vertex
 - **Idea!** Repeat a graph search algorithm A on any unvisited vertex
-
- Repeat the following until all vertices have been visited:
 - Choose an arbitrary unvisited vertex s , use A to explore all vertices reachable from s
-
- We call this algorithm **Full- A** , specifically Full-BFS or Full-DFS if A is BFS or DFS
 - Visits every vertex once, so both Full-BFS and Full-DFS run in $O(|V| + |E|)$ time
 - **Example:** Run Full-DFS/Full-BFS on G_1 and/or G_2



Graph Connectivity

- An **undirected** graph is **connected** if there is a path connecting every pair of vertices
- In a directed graph, vertex u may be reachable from v , but v may not be reachable from u
- Connectivity is more complicated for directed graphs (we won't discuss in this class)
- **Connectivity** (G): is undirected graph G connected?
- **Connected_Components** (G): given undirected graph $G = (V, E)$, return partition of V into subsets $V_i \subseteq V$ (**connected components**) where each V_i is connected in G and there are no edges between vertices from different connected components
- Consider a graph algorithm A that solves Single Source Reachability
- **Claim:** A can be used to solve Connected Components
- **Proof:** Run Full- A . For each run of A , put visited vertices in a connected component □

Topological Sort

- A **Directed Acyclic Graph (DAG)** is a directed graph that contains no directed cycle.
- A **Topological Order** of a graph $G = (V, E)$ is an ordering f on the vertices such that: every edge $(u, v) \in E$ satisfies $f(u) < f(v)$.
- **Exercise:** Prove that a directed graph admits a topological ordering if and only if it is a DAG.
- How to find a topological order?
- A **Finishing Order** is the order in which a Full-DFS **finishes visiting** each vertex in G
- **Claim:** If $G = (V, E)$ is a DAG, the reverse of a finishing order is a topological order
- **Proof:** Need to prove, for every edge $(u, v) \in E$ that u is ordered before v , i.e., the visit to v finishes before visiting u . Two cases:
 - If u visited before v :
 - * Before visit to u finishes, will visit v (via (u, v) or otherwise)
 - * Thus the visit to v finishes before visiting u
 - If v visited before u :
 - * u can't be reached from v since graph is acyclic
 - * Thus the visit to v finishes before visiting u

□

Cycle Detection

- Full-DFS will find a topological order if a graph $G = (V, E)$ is acyclic
- If reverse finishing order for Full-DFS is not a topological order, then G must contain a cycle
- Check if G is acyclic: for each edge (u, v) , check if v is before u in reverse finishing order
- Can be done in $O(|E|)$ time via a hash table or direct access array
- To return such a cycle, maintain the set of **ancestors** along the path back to s in Full-DFS
- **Claim:** If G contains a cycle, Full-DFS will traverse an edge from v to an ancestor of v .
- **Proof:** Consider a cycle $(v_0, v_1, \dots, v_k, v_0)$ in G
 - Without loss of generality, let v_0 be the first vertex visited by Full-DFS on the cycle
 - For each v_i , before visit to v_i finishes, will visit v_{i+1} and finish
 - Will consider edge (v_i, v_{i+1}) , and if v_{i+1} has not been visited, it will be visited now
 - Thus, before visit to v_0 finishes, will visit v_k (for the first time, by v_0 assumption)
 - So, before visit to v_k finishes, will consider (v_k, v_0) , where v_0 is an ancestor of v_k

□

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

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
12 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.

```

1 def full_dfs(Adj):                                # Adj: adjacency list
2     parent = [None for v in Adj]                  # O(V) (use hash if unlabeled)
3     order = []                                     # O(1) initialize order list
4     for v in range(len(Adj)):                     # O(V) loop over vertices
5         if parent[v] is None:                      # O(1) parent not yet assigned
6             parent[v] = v                          # O(1) assign self as parent (a root)
7             dfs(Adj, v, parent, order)            # DFS from v (BFS can also be used)
8
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. $\text{parent}[v] = u$). 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>

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

Lecture 11: Weighted Shortest Paths

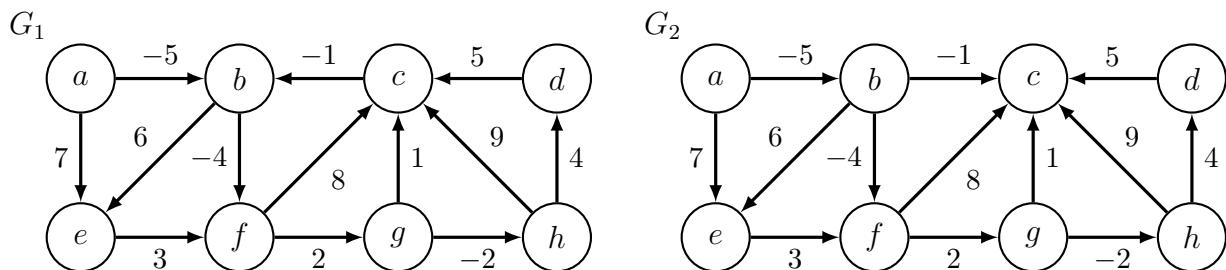
Review

- Single-Source Shortest Paths with BFS in $O(|V| + |E|)$ time (return distance per vertex)
- Single-Source Reachability with BFS or DFS in $O(|E|)$ time (return only reachable vertices)
- Connected components with Full-BFS or Full-DFS in $O(|V| + |E|)$ time
- Topological Sort of a DAG with Full-DFS in $O(|V| + |E|)$ time
- **Previously:** distance = number of edges in path **Today:** generalize meaning of distance

Weighted Graphs

- A **weighted graph** is a graph $G = (V, E)$ together with a weight function $w : E \rightarrow \mathbb{Z}$
- i.e., assigns each edge $e = (u, v) \in E$ an integer weight: $w(e) = w(u, v)$
- Many applications for edge weights in a graph:
 - distances in road network
 - latency in network connections
 - strength of a relationship in a social network
- Two common ways to represent weights computationally:
 - Inside graph representation: store edge weight with each vertex in adjacency lists
 - Store separate Set data structure mapping each edge to its weight
- We assume a representation that allows querying the weight of an edge in $O(1)$ time

Examples



Weighted Paths

- The **weight** $w(\pi)$ of a path π in a weighted graph is the sum of weights of edges in the path
- The **(weighted) shortest path** from $s \in V$ to $t \in V$ is path of minimum weight from s to t
- $\delta(s, t) = \inf\{w(\pi) \mid \text{path } \pi \text{ from } s \text{ to } t\}$ is the **shortest-path weight** from s to t
- (Often use “distance” for shortest-path weight in weighted graphs, not number of edges)
- As with unweighted graphs:
 - $\delta(s, t) = \infty$ if no path from s to t
 - Subpaths of shortest paths are shortest paths (or else could splice in a shorter path)
- Why infimum not minimum? Possible that no finite-length minimum-weight path exists
- When? Can occur if there is a negative-weight cycle in the graph, Ex: (b, f, g, c, b) in G_1
- A **negative-weight cycle** is a path π starting and ending at same vertex with $w(\pi) < 0$
- $\delta(s, t) = -\infty$ if there is a path from s to t through a vertex on a negative-weight cycle
- If this occurs, don’t want a shortest path, but may want the negative-weight cycle

Weighted Shortest Paths Algorithms

- Next four lectures: algorithms to find shortest-path weights in weighted graphs
- (No parent pointers: can reconstruct shortest paths tree in linear time after. Next page!)
- Already know one algorithm: Breadth-First Search! Runs in $O(|V| + |E|)$ time when, e.g.:
 - graph has positive weights, and all weights are the same
 - graph has positive weights, and sum of all weights at most $O(|V| + |E|)$
- For general weighted graphs, we don’t know how to solve SSSP in $O(|V| + |E|)$ time
- But if your graph is a **Directed Acyclic Graph** you can!

Restrictions		SSSP Algorithm		
Graph	Weights	Name	Running Time $O(\cdot)$	Lecture
General	Unweighted	BFS	$ V + E $	L09
DAG	Any	DAG Relaxation	$ V + E $	L11 (Today!)
General	Any	Bellman-Ford	$ V \cdot E $	L12
General	Non-negative	Dijkstra	$ V \log V + E $	L13

Shortest-Paths Tree

- For BFS, we kept track of parent pointers during search. Alternatively, compute them after!
 - If know $\delta(s, v)$ for all vertices $v \in V$, can construct shortest-path tree in $O(|V| + |E|)$ time
 - For weighted shortest paths from s , only need parent pointers for vertices v with finite $\delta(s, v)$
-
- Initialize empty P and set $P(s) = \text{None}$
 - For each vertex $u \in V$ where $\delta(s, u)$ is finite:
 - For each outgoing neighbor $v \in \text{Adj}^+(u)$:
 - * If $P(v)$ not assigned and $\delta(s, v) = \delta(s, u) + w(u, v)$:
 - . There exists a shortest path through edge (u, v) , so set $P(v) = u$
 - Parent pointers may traverse cycles of zero weight. Mark each vertex in such a cycle.
 - For each unmarked vertex $u \in V$ (including vertices later unmarked):
 - For each $v \in \text{Adj}^+(u)$ where v is marked and $\delta(s, v) = \delta(s, u) + w(u, v)$:
 - * Unmark vertices in cycle containing v by traversing parent pointers from v
 - * Set $P(v) = u$, breaking the cycle
 - **Exercise:** Prove this algorithm correctly computes parent pointers in linear time
 - Because we can compute parent pointers afterward, we focus on computing distances

DAG Relaxation

- **Idea!** Maintain a distance estimate $d(s, v)$ (initially ∞) for each vertex $v \in V$, that always upper bounds true distance $\delta(s, v)$, then gradually lowers until $d(s, v) = \delta(s, v)$
- When do we lower? When an edge violates the triangle inequality!
- **Triangle Inequality:** the shortest-path weight from u to v cannot be greater than the shortest path from u to v through another vertex x , i.e., $\delta(u, v) \leq \delta(u, x) + \delta(x, v)$ for all $u, v, x \in V$
- If $d(s, v) > d(s, u) + w(u, v)$ for some edge (u, v) , then triangle inequality is violated :-(
- Fix by lowering $d(s, v)$ to $d(s, u) + w(u, v)$, i.e., **relax** (u, v) to satisfy violated constraint
- **Claim:** Relaxation is **safe**: maintains that each $d(s, v)$ is weight of a path to v (or ∞) $\forall v \in V$
- **Proof:** Assume $d(s, v')$ is weight of a path (or ∞) for all $v' \in V$. Relaxing some edge (u, v) sets $d(s, v)$ to $d(s, u) + w(u, v)$, which is the weight of a path from s to v through u . \square

- Set $d(s, v) = \infty$ for all $v \in V$, then set $d(s, s) = 0$
 - Process each vertex u in a topological sort order of G :
 - For each outgoing neighbor $v \in \text{Adj}^+(u)$:
 - * If $d(s, v) > d(s, u) + w(u, v)$:
 - relax edge (u, v) , i.e., set $d(s, v) = d(s, u) + w(u, v)$
-

- **Example:** Run DAG Relaxation from vertex a in G_2

Correctness

- **Claim:** At end of DAG Relaxation: $d(s, v) = \delta(s, v)$ for all $v \in V$
- **Proof:** Induct on k : $d(s, v) = \delta(s, v)$ for all v in first k vertices in topological order
 - Base case: Vertex s and every vertex before s in topological order satisfies claim at start
 - Inductive step: Assume claim holds for first k' vertices, let v be the $(k' + 1)^{\text{th}}$
 - Consider a shortest path from s to v , and let u be the vertex preceding v on path
 - u occurs before v in topological order, so $d(s, u) = \delta(s, u)$ by induction
 - When processing u , $d(s, v)$ is set to be no larger (\leq) than $\delta(s, u) + w(u, v) = \delta(s, v)$
 - But $d(s, v) \geq \delta(s, v)$, since relaxation is safe, so $d(s, v) = \delta(s, v)$ □
- Alternatively:
 - For any vertex v , DAG relaxation sets $d(s, v) = \min\{d(s, u) + w(u, v) \mid u \in \text{Adj}^-(v)\}$
 - Shortest path to v must pass through some incoming neighbor u of v
 - So if $d(s, u) = \delta(s, u)$ for all $u \in \text{Adj}^-(v)$ by induction, then $d(s, v) = \delta(s, v)$ □

Running Time

- Initialization takes $O(|V|)$ time, and Topological Sort takes $O(|V| + |E|)$ time
- Additional work upper bounded by $O(1) \times \sum_{u \in V} \deg^+(u) = O(|E|)$
- Total running time is linear, $O(|V| + |E|)$

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

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** $w : E \rightarrow \mathbb{R}$, 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]`.

```

1   W1 = [0: {1: -2},           W2 = {0: {1: 1, 3: 2, 4: -1},    # 0
2     1: {2: 0},                  1: {0: 1},                 # 1
3     2: {0: 1},                  2: {3: 0},                 # 2
4     3: {4: 3}}                3: {0: 2, 2: 0},            # 3
5                           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 time¹. 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 $\pi = (v_1, \dots, v_k)$ 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.

¹We 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	$ V + E $
DAG	Any	DAG Relaxation	$ V + E $
General	Any	Bellman-Ford	$ V \cdot E $
General	Non-negative	Dijkstra	$ V \log V + E $

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:

```

1 def general_relax(Adj, w, s):      # Adj: adjacency list, w: weights, s: start
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     while True:                  # repeat forever!
6         relax some d[v] ??       # relax a shortest path estimate d(s, v)
7     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 path². 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) . \square

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. \square

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)

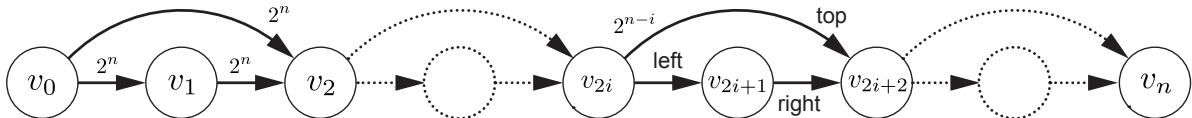
```

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.

²This 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 be 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**.

```

1 def DAG_Relaxation(Adj, w, s):      # Adj: adjacency list, w: weights, s: start
2     _, order = dfs(Adj, s)           # run depth-first search on graph
3     order.reverse()                 # reverse returned order
4     d = [float('inf') 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     for u in order:               # loop through vertices in topo sort
8         for v in Adj[u]:          # loop through out-going edges of u
9             try_to_relax(Adj, w, d, parent, u, v) # try to relax edge from u to v
10    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 $\pi = (v_1, \dots, v_m)$ 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. \square

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 prerequisites 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.

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

Problem Session 5

Problem 5-1. Graph Radius

In any undirected graph $G = (V, E)$, the **eccentricity** $\epsilon(u)$ of a vertex $u \in V$ is the shortest distance to its farthest vertex v , i.e., $\epsilon(u) = \max\{\delta(u, v) \mid v \in V\}$. The **radius** $R(G)$ of an undirected graph $G = (V, E)$ is the smallest eccentricity of any vertex, i.e., $R(G) = \min\{\epsilon(u) \mid u \in V\}$.

- (a) Given connected undirected graph G , describe an $O(|V||E|)$ -time algorithm to determine the radius of G .

Solution: Compute $R(G)$ directly: run breadth-first search from each node u and calculate $\epsilon(u)$ as the maximum of $\delta(u, v)$ for each $v \in V$. Then return the minimum of $\epsilon(u)$ for all $u \in V$. Each vertex is connected to an edge because G is connected so $|V| = O(|E|)$. Each BFS takes $O(|E|)$ time, leading to $O(|V||E|)$ time in total.

- (b) Given connected undirected graph G , describe an $O(|E|)$ -time algorithm to determine an upper bound R^* on the radius of G , such that $R(G) \leq R^* \leq 2R(G)$.

Solution: Use breadth-first search to find $\epsilon(u)$ for any $u \in V$ in $O(|E|)$ time. We claim that $R(G) \leq \epsilon(u) \leq 2R(G)$, so we can choose $R^* = \epsilon(u)$. First, $\epsilon(u) \geq R(G)$ because $R(G)$ is the minimum $\epsilon(v)$ over all $v \in V$. Second, $2R(G) \geq \epsilon(u)$ because if a vertex x has eccentricity $R(G)$, we can construct a path from u to any other vertex v by concatenating a shortest path from u to x and a shortest path from x to v , both of which have length at most $R(G)$; thus $\epsilon(u)$ cannot be greater than $2R(G)$.

Problem 5-2. Internet Investigation

MIT has heard complaints regarding the speed of their WiFi network. The network consists of r routers, some of which are marked as **entry points** which are connected to the rest of the internet. Some pairs of routers are directly connected to each other via bidirectional wires. Each wire w_i between two routers has a known length ℓ_i measured in a positive integer number of feet. The **latency** of a router in the network is proportional to the minimum feet of wire a signal from the router must pass through to reach an entry point. Assume the latency of every router is finite and there is at most $100r$ feet of wire in the entire network. Given a schematic of the network depicting all routers and the lengths of all wires, describe an $O(r)$ -time algorithm to determine the sum total latency, summed over all routers in the network.

Solution: Construct undirected unweighted graph G in the following way. Construct r vertices, one associated with each router in the network. Then, for each wire w_i connecting between routers a_i and b_i with length ℓ_i , add an unweighted chain of ℓ_i edges between the vertices associated with routers a_i and b_i . For each wire w_i , this process adds ℓ_i edges and $\ell_i - 1$ vertices to the graph. Now, if there were exactly one entry point, we could run breadth-first search from it to every other node in the graph, and the shortest path from the entry point to each router would be equal to the router's latency, by definition.

In order to search from all entry points at once, we add an additional node s (sometimes called a super node) to the graph with an edge to every entry point, and compute shortest paths from s using breadth-first search. Now, the shortest path from s to a router is one more than the latency at the router, so to return the total latency, we can sum the shortest path distance over all routers and subtract r . The graph G has at most $r + 100r + 1 = O(r)$ nodes and at most $100r + r = O(r)$ edges, so the graph takes $O(r)$ time to construct.

Breadth-first search runs in linear time in the size of the graph, so also takes $O(r)$ time. Lastly, computing the sum just requires looping over the $O(r)$ vertices and summing shortest paths to vertices that are routers. Thus the algorithm runs in $O(r)$ time.

Problem 5-3. Quadwizard Quest

Wizard Potry Harter and her three wizard friends have been tasked with searching every room of a Labyrinth for magical artifacts. The Labyrinth consists of n rooms, where each room has at most four doors leading to other rooms. Assume all doors begin closed and every room in the Labyrinth is reachable from a specified entry room by traversing doors between rooms. Some doors are protected by evil enchantments that must be **disenchanted** before they can be opened; but all other doors may be opened freely. Given a map of the Labyrinth marking each door as enchanted or not, describe an $O(n)$ -time algorithm to determine the minimum number of doors that must be disenchanted in order to visit every room of the Labyrinth, beginning from the entry room.

Solution: Construct a graph G with a vertex associated with each of the n rooms in the Labyrinth and an edge between two rooms if there is a door that is **not** enchanted connecting them. For any room in this graph, if the wizards can reach a room corresponding to vertex v , the wizards can search the rooms associated with every vertex in v 's connected component without having to disenchant any door. Since every room is reachable from the entry room, it suffices to identify the connected components of G , and then repeatedly disenchant doors that would connect two disconnected components. If there are k connected components of G , the wizards must disenchant $k - 1$ doors to visit every room. So, run either Full breadth-first search or Full depth-first search to count the number of connected components of G and return one less. G has n vertices and at most $4n$ edges, so counting the number of connected components in G will take at most $O(n)$ time for either algorithm.

Problem 5-4. Purity Atlantic

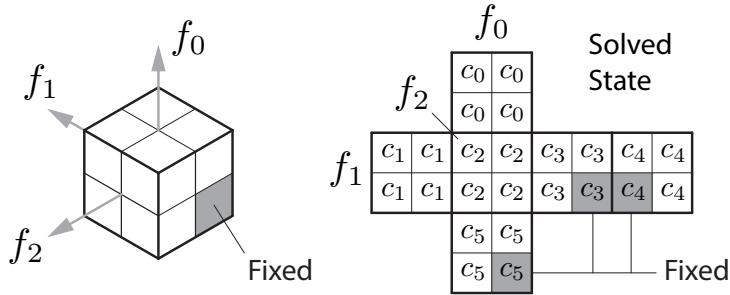
Brichard Ranson is the founder of Purity Atlantic, an international tour company that specializes in planning luxury honeymoon getaways for newlywed couples. To book a customized tour, a couple submits their home city, and the names of three touring cities they would like to visit during their honeymoon. Then Purity will arrange all accommodations, including a **flight itinerary**: a sequence of flights from their home to each touring city (in any order), then returning back to their home. Unfortunately, it's not always possible to fly directly between any two cities, so multiple flights may be required. While cost and time are not a factor, couples prefer to minimize the number of direct flights they will have to take during their honeymoon. Given a list of c cities and a list of all f available direct flights, where each direct flight is specified by an ordered pair of cities (origin, destination), describe an efficient algorithm to determine a flight itinerary for a given couple that minimizes the number of direct flights they will have to take.

Solution: Let the directed **distance** from city a to city b be the fewest direct flights needed to reach b from a (or infinite if b is not reachable from a). Given the home city and three touring cities, if we could compute the $2\binom{4}{2} = 12 = O(1)$ pairwise directed distances between all pairs of the four input cities, then we could find an itinerary of that minimizes the number of direct flights starting at home, visiting the three cities, and returning in $O(1)$ time, by comparing the fewest flights needed for each of the $3! = 6 = O(1)$ permutations of touring cities, each of which can be computed in $O(1)$ time (if none yield a finite distance, then return that no itinerary is possible). To compute these pairwise directed distances, create a graph G with a vertex associated with each of the c cities and a directed edge for each of the f flights from the vertex associated with the origin to the vertex associated with the destination. Then, for each of the four input cities, breadth-first search to find the directed distance between it and the other three cities, storing parent

pointers to remember a shortest path for each pair. Each of the four breadth-first searches takes $O(c + f)$ time, so the 12 pairwise distance can be all computed in $O(c + f)$ time, and a minimizing itinerary can be returned by concatenating the corresponding remembered shortest paths. Thus the entire algorithm also runs in $O(c + f)$ time. This algorithm is efficient because any correct algorithm must examining the entire graph.

Problem 5-5. Pocket Cube

A Pocket Cube¹ is a smaller $2 \times 2 \times 2$ variant of the traditional $3 \times 3 \times 3$ Rubik's cube, consisting of eight corner cubes, each with a different color on its three visible faces. The **solved** configuration is when each 2×2 face of the Pocket Cube is monochromatic. We reference each color c_i with an index $i \in \{0, \dots, 5\}$. Without loss of generality, we fix the position and orientation of one of the corner cubes and only allow single-turn rotations about the normals of the three faces of the Pocket Cube $\{f_0, f_1, f_2\}$ that do not contain the fixed corner cube; specifically, a **move** is described by tuple (j, s) corresponding to a single-turn rotation of face f_j , clockwise when $s = 1$ and counterclockwise when $s = -1$. Breadth-first search can be used to solve puzzles like the Pocket Cube by searching a graph whose vertices are possible configurations of the puzzle, with an edge between two configurations if one can be reached from the other via a single move. Instead of storing these adjacencies explicitly, one can compute the neighbors of a given configuration by applying all possible single moves to the configuration.



- (a) Argue that the number of distinct configurations of a Pocket Cube is less than 12 million (try to get as tight a bound as you can using combinatorics).

Solution: If one of the corner cubes is fixed, the remaining seven corner cubes may exist in $7!$ permutations, while each corner cube may rotate independently to any of three rotations. So the number of configurations is upper bounded by $7!3^7 = 11022480$.

- (b) State the max and min degree of any vertex in the Pocket Cube graph.

Solution: Three sides may be rotated, and each may be rotated clockwise or counterclockwise. So each configuration has exactly $3 \times 2 = 6$ configurations neighboring it.

- (c) In your problem set template is code that fully explores the Pocket Cube graph from a given configuration using breadth-first search, and then returns a sequence of moves that solves the Pocket Cube (assuming the solved configuration is reachable). However, this solver is very slow². Run the code provided and state the number of configurations the search explores. How does this number compare to your upper bound from part (a)?

Solution: The provided BFS searches configurations. This is exactly one third of the configurations estimated by part (a). In fact, the space of configurations turns out to be three disconnected

¹http://en.wikipedia.org/wiki/Pocket_Cube

²Please note that the code requires a couple minutes and considerable memory (over 400 Mb) to complete.

components of equal size. A representative configuration contained in each component can be achieved by rotating a single corner cube of the solved state to each of its three rotations.

- (d) State the **max number of moves** w needed to solve any solvable Pocket Cube.

Solution: This can be found directly from the code output: the diameter is equal to the number of frontiers visited minus 1, i.e., 14.

- (e) Let N_i be the number of Pocket Cube configurations reachable within i moves of a particular configuration. The code provided visits N_w configurations (which is larger than 3 million). Describe an algorithm to find a shortest sequence of moves to solve any Pocket Cube configuration (or return no such sequence exists) that visits no more than $2N_{\lceil w/2 \rceil}$ configurations (which is less than 90 thousand).

Solution: Run BFS from both the query configuration and the solved configuration, but alternating exploring frontiers from each. Store parent pointers to the configuration preceded by each explored configuration. After exploring each frontier, check whether a configuration in the new frontier has been explored by the other BFS. If it has been explored by the other BFS, construct the path from the query configuration to the solved configuration through the overlapping node, by following parent pointers from one BFS to the query configuration, and following parent pointers from the other BFS to the solved configuration. Because we alternate exploring frontiers, the lengths of the paths found by each BFS differ by at most one, so each has length at most $\lceil w/2 \rceil$. Then each BFS visits at most $N_{\lceil w/2 \rceil}$ configuration, as desired.

- (f) Rewrite the `solve(config)` function in the template code provided, based on your algorithm from part (e).

Solution:

```

1 def solve(config):
2     # Return a sequence of moves to solve config, or None if not possible
3     def check(frontier, parent):
4         for f in frontier:
5             if f in parent:
6                 return f
7         return None
8     parent_c, frontier_c = {config: None}, [config]
9     parent_s, frontier_s = {SOLVED: None}, [SOLVED]
10    middle = check(frontier_c, parent_s)
11    while middle is None:
12        frontier_c = explore_frontier(frontier_c, parent_c)
13        middle = check(frontier_c, parent_s)
14        if middle: break
15        frontier_s = explore_frontier(frontier_s, parent_s)
16        middle = check(frontier_s, parent_c)
17    if middle:
18        path_c = path_to_config(middle, parent_c)
19        path_s = path_to_config(middle, parent_s)
20        path_s.pop()
21        path_s.reverse()
22        return moves_from_path(path_c + path_s)
23    return None

```

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

Lecture 12: Bellman-Ford

Previously

- Weighted graphs, shortest-path weight, negative-weight cycles
- Finding shortest-path tree from shortest-path weights in $O(|V| + |E|)$ time
- DAG Relaxation: algorithm to solve SSSP on a weighted DAG in $O(|V| + |E|)$ time
- SSSP for graph with negative weights
 - Compute $\delta(s, v)$ for all $v \in V$ ($-\infty$ if v reachable via negative-weight cycle)
 - If a negative-weight cycle reachable from s , return one

Warmups

- **Exercise 1:** Given undirected graph G , return whether G contains a negative-weight cycle
- **Solution:** Return **Yes** if there is an edge with negative weight in G in $O(|E|)$ time :o
 - So for this lecture, we restrict our discussion to **directed graphs**
- **Exercise 2:** Given SSSP algorithm A that runs in $O(|V|(|V| + |E|)$ time, show how to use it to solve SSSP in $O(|V||E|)$ time
- **Solution:** Run BFS or DFS to find the vertices reachable from s in $O(|E|)$ time
 - Mark each vertex v not reachable from s with $\delta(s, v) = \infty$ in $O(|V|)$ time
 - Make graph $G' = (V', E')$ with only vertices reachable from s in $O(|V| + |E|)$ time
 - Run A from s in G' .
 - G' is connected, so $|V'| = O(|E'|) = O(|E|)$ so A runs in $O(|V||E|)$ time
- Today, we will find a SSSP algorithm with this running time that works for general graphs!

Restrictions		SSSP Algorithm		
Graph	Weights	Name	Running Time $O(\cdot)$	Lecture
General	Unweighted	BFS	$ V + E $	L09
DAG	Any	DAG Relaxation	$ V + E $	L11
General	Any	Bellman-Ford	$ V \cdot E $	L12 (Today!)
General	Non-negative	Dijkstra	$ V \log V + E $	L13

Simple Shortest Paths

- If graph contains cycles and negative weights, might contain negative-weight cycles : (
- If graph does not contain negative-weight cycles, shortest paths are simple!
- **Claim 1:** If $\delta(s, v)$ is finite, there exists a shortest path to v that is **simple**
- **Proof:** By contradiction:
 - Suppose no simple shortest path; let π be a shortest path with fewest vertices
 - π not simple, so exists cycle C in π ; C has non-negative weight (or else $\delta(s, v) = -\infty$)
 - Removing C from π forms path π' with fewer vertices and weight $w(\pi') \leq w(\pi)$ \square
- Since simple paths cannot repeat vertices, finite shortest paths contain at most $|V| - 1$ edges

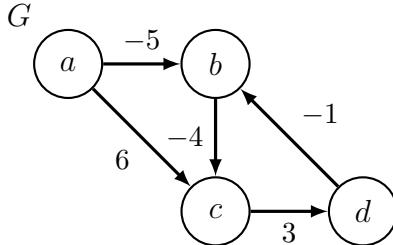
Negative Cycle Witness

- **k -Edge Distance** $\delta_k(s, v)$: the minimum weight of any path from s to v using $\leq k$ edges
- **Idea!** Compute $\delta_{|V|-1}(s, v)$ and $\delta_{|V|}(s, v)$ for all $v \in V$
 - If $\delta(s, v) \neq -\infty$, $\delta(s, v) = \delta_{|V|-1}(s, v)$, since a shortest path is simple (or nonexistent)
 - If $\delta_{|V|}(s, v) < \delta_{|V|-1}(s, v)$
 - * there exists a shorter non-simple path to v , so $\delta_{|V|}(s, v) = -\infty$
 - * call v a (negative cycle) **witness**
 - However, there may be vertices with $-\infty$ shortest-path weight that **are not witnesses**
- **Claim 2:** If $\delta(s, v) = -\infty$, then v is reachable from a witness
- **Proof:** Suffices to prove: every negative-weight cycle reachable from s contains a witness
 - Consider a negative-weight cycle C reachable from s
 - For $v \in C$, let $v' \in C$ denote v 's predecessor in C , where $\sum_{v \in C} w(v', v) < 0$
 - Then $\delta_{|V|}(s, v) \leq \delta_{|V|-1}(s, v') + w(v', v)$ (RHS weight of some path on $\leq |V|$ vertices)
 - So $\sum_{v \in C} \delta_{|V|}(s, v) \leq \sum_{v \in C} \delta_{|V|-1}(s, v') + \sum_{v \in C} w(v', v) < \sum_{v \in C} \delta_{|V|-1}(s, v)$
 - If C contains no witness, $\delta_{|V|}(s, v) \geq \delta_{|V|-1}(s, v)$ for all $v \in C$, a contradiction \square

Bellman-Ford

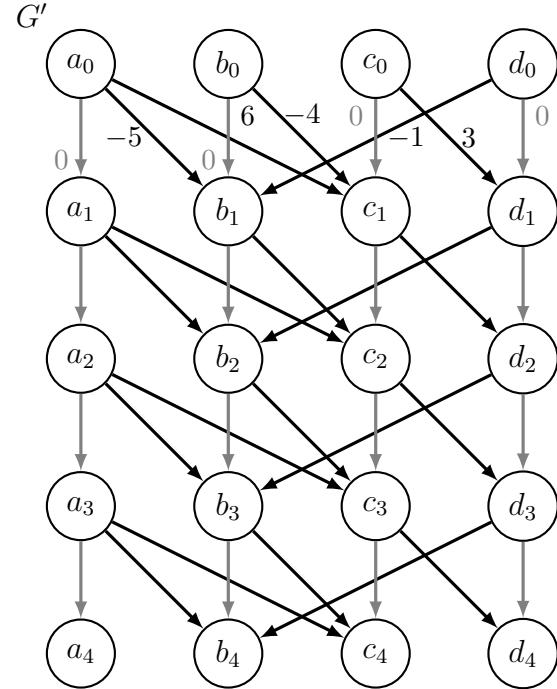
- **Idea!** Use **graph duplication**: make multiple copies (or levels) of the graph
 - $|V| + 1$ levels: vertex v_k in level k represents reaching vertex v from s using $\leq k$ edges
 - If edges only increase in level, resulting graph is a DAG!
-
- Construct new DAG $G' = (V', E')$ from $G = (V, E)$:
 - G' has $|V|(|V| + 1)$ vertices v_k for all $v \in V$ and $k \in \{0, \dots, |V|\}$
 - G' has $|V|(|V| + |E|)$ edges:
 - * $|V|$ edges (v_{k-1}, v_k) for $k \in \{1, \dots, |V|\}$ of weight zero for each $v \in V$
 - * $|V|$ edges (u_{k-1}, v_k) for $k \in \{1, \dots, |V|\}$ of weight $w(u, v)$ for each $(u, v) \in E$
 - Run DAG Relaxation on G' from s_0 to compute $\delta(s_0, v_k)$ for all $v_k \in V'$
 - For each vertex: set $d(s, v) = \delta(s_0, v_{|V|-1})$
 - For each witness $u \in V$ where $\delta(s_0, u_{|V|}) < \delta(s_0, u_{|V|-1})$:
 - For each vertex v reachable from u in G :
 - * set $d(s, v) = -\infty$

Example



$\delta(a_0, v_k)$

$k \setminus v$	a	b	c	d
0	0	∞	∞	∞
1	0	-5	6	∞
2	0	-5	-9	9
3	0	-5	-9	-6
4	0	-7	-9	-6
$\delta(a, v)$	0	$-\infty$	$-\infty$	$-\infty$



Correctness

- **Claim 3:** $\delta(s_0, v_k) = \delta_k(s, v)$ for all $v \in V$ and $k \in \{0, \dots, |V|\}$
- **Proof:** By induction on k :
 - Base case: true for all $v \in V$ when $k = 0$ (only v_0 reachable from s_0 is $v = s$)
 - Inductive Step: Assume true for all $k < k'$, prove for $k = k'$
$$\begin{aligned}\delta(s_0, v_{k'}) &= \min\{\delta(s_0, u_{k'-1}) + w(u_{k'-1}, v_{k'}) \mid u_{k'-1} \in \text{Adj}^-(v_{k'})\} \\ &= \min\{\{\delta(s_0, u_{k'-1}) + w(u, v) \mid u \in \text{Adj}^-(v)\} \cup \{\delta(s_0, v_{k'-1})\}\} \\ &= \min\{\{\delta_{k'-1}(s, u) + w(u, v) \mid u \in \text{Adj}^-(v)\} \cup \{\delta_{k'-1}(s, v)\}\} \quad (\text{by induction}) \\ &= \delta_{k'}(s, v)\end{aligned}$$
□
- **Claim 4:** At the end of Bellman-Ford $d(s, v) = \delta(s, v)$
- **Proof:** Correctly computes $\delta_{|V|-1}(s, v)$ and $\delta_{|V|}(s, v)$ for all $v \in V$ by Claim 3
 - If $\delta(s, v) \neq -\infty$, correctly sets $d(s, v) = \delta_{|V|-1}(s, v) = \delta(s, v)$
 - Then sets $d(s, v) = -\infty$ for any v reachable from a witness; correct by Claim 2□

Running Time

- G' has size $O(|V|(|V| + |E|))$ and can be constructed in as much time
- Running DAG Relaxation on G' takes linear time in the size of G'
- Does $O(1)$ work for each vertex reachable from a witness
- Finding reachability of a witness takes $O(|E|)$ time, with at most $O(|V|)$ witnesses: $O(|V||E|)$
- (Alternatively, connect **super node** x to witnesses via 0-weight edges, linear search from x)
- Pruning G at start to only subgraph reachable from s yields $O(|V||E|)$ -time algorithm

Extras: Return Negative-Weight Cycle or Space Optimization

- **Claim 5:** Shortest $s_0 - v_{|V|}$ path π for any witness v contains a negative-weight cycle in G
- **Proof:** Since π contains $|V| + 1$ vertices, must contain at least one cycle C in G
 - C has negative weight (otherwise, remove C to make path π' with fewer vertices and $w(\pi') \leq w(\pi)$, contradicting witness v)□
- Can use just $O(|V|)$ space by storing only $\delta(s_0, v_{k-1})$ and $\delta(s_0, v_k)$ for each k from 1 to $|V|$
- Traditionally, Bellman-Ford stores only one value per vertex, attempting to relax every edge in $|V|$ rounds; but estimates do not correspond to k -Edge Distances, so analysis trickier
- But these space optimizations don't return a negative weight cycle

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

Recitation 12

Bellman-Ford

In lecture, we presented a version of Bellman-Ford¹ 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.

¹This 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 property² 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. \square

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 $k \in \{0, \dots, |V|\}$, 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.

²Recall 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 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.

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

Problem Session 6

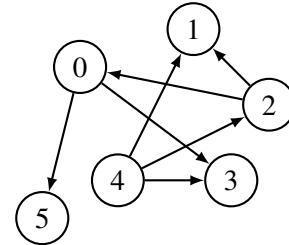
Problem 6-1. Topological Training

Please answer the following questions about the unweighted directed graph G below, whose vertex set is $\{0, 1, 2, 3, 4, 5\}$.

- (a) State a topological ordering of G . Then state and **justify** the number of distinct topological orderings of G .
- (b) State a single directed edge that could be added to G to construct a simple¹ graph with no topological ordering. Then state and **justify** the number of distinct single edges that could be added to G to construct a simple graph with no topological ordering.

Solution: (a) Vertices 4 and 2 must be the first and second vertex in any topological order, since there is a directed path from 4 through 2 to every other vertex in the graph. For the remaining four vertices, the ordering of 0, 3, and 5 are independent from 1. Vertices 0, 3, and 5 have two possible orderings: $(0, 3, 5)$ and $(0, 5, 3)$, while 1 can be placed in one of four positions relative to either ordering. Thus there are 8 distinct topological orders of G . Specifically:

$$(4, 2, 1, 0, 3, 5) \quad (4, 2, 0, 1, 3, 5) \quad (4, 2, 0, 3, 1, 5) \quad (4, 2, 0, 3, 5, 1) \\ (4, 2, 1, 0, 5, 3) \quad (4, 2, 0, 1, 5, 3) \quad (4, 2, 0, 5, 1, 3) \quad (4, 2, 0, 5, 3, 1)$$



Solution: (b) A topological ordering will not exist, if and only if the resulting graph contains a cycle. All vertices are reachable from 4, so we can add any edge from vertices 0, 1, 2, 3, or 5 to 4 to create a cycle. All vertices except 4 are reachable from 2, so we can add any edge from vertices 0, 1, 3, or 5 to 2 to create a cycle. Vertices 3 and 5 are reachable from 0, so we can add either edge to 0. No vertices are reachable from 1, 3, or 5, so no edge can be added to them to construct a cycle. Thus there are **eleven** such directed edges, specifically:

$$\{(0, 4), (1, 4), (2, 4), (3, 4), (5, 4)(0, 2), (1, 2), (3, 2), (5, 2), (3, 0), (5, 0)\}.$$

Problem 6-2. Never Return

Bimsa is a young lioness whose evil uncle has just banished her from the land of Honor Stone, proclaiming: “Run away, Bimsa. Run away, and never return.” Bimsa has knowledge of all landmarks and trails in and around Honor Stone. For each landmark, she knows its x and y coordinates and whether it is inside or outside of Honor Stone. For each trail, she knows its positive integer length and the two landmarks it directly connects. Each landmark connects to at most five trails, each trail may be traversed in either direction, and every landmark can be reached along trails from the **gorge**, the landmark where Bimsa is now. Bimsa wants to leave Honor Stone quickly, while only traversing trails in a way that **never returns**: traversing a trail

¹A simple graph has no self-loops (i.e., each edge connects two different vertices) and has no multi-edges (i.e., there can be at most one directed edge from vertex a to vertex b , though a directed edge from b to a may exist).

from landmark a to landmark b never returns if the distance² from a to the gorge is strictly smaller than the distance from b to the gorge. If there are n landmarks in Honor Stone, describe an $O(n)$ -time algorithm to determine a shortest route that never returns, from the gorge to any landmark outside of Honor Stone.

Solution: The ‘never return’ constraint ensures that we can only visit landmarks in strictly increasing distance from the gorge, so if we construct a graph on landmarks with an edge for each trail only in a direction which increases distance from the gorge, this graph will be acyclic. Let $d(a)$ denote the **squared** Euclidean distance from the gorge to landmark a (squared distances are integers that are computable in $O(1)$ time). Construct a graph G with a vertex for each landmark either in Honor Stone or neighboring a landmark in Honor Stone. Then for each trail directly connecting landmarks a and b , add either directed edge (a, b) if $d(a) < d(b)$, directed edge (b, a) if $d(a) > d(b)$, or no edge if $d(a) = d(b)$. Since there are at most $O(n)$ landmarks in Honor Stone or neighboring a landmark in Honor Stone, and each landmark is connected to at most a constant number of trails, there are at most $O(n)$ trials, so this graph can be constructed in $O(n)$ time. Then every directed path in G from the gorge satisfies the ‘never return’ condition, so we just need to find any shortest route in G from the gorge to any vertex corresponding to a landmark that is outside Honor Stone. Solve single source shortest paths from the gorge (storing parent pointers to allow paths to be reconstructed), and return the shortest path to any vertex corresponding to a landmark that is outside Honor Stone. Since G is a DAG, we can solve single source shortest paths using DAG relaxation in $O(n)$ time as desired.

Problem 6-3. DigBuild

Software company Jomang is developing **DigBuild**, a new 3D voxel-based exploration game. The game world features n collectable block types, each identified with a unique integer from 1 to n . Some of these block types can be converted into other block types: a conversion (d_1, b_1, d_2, b_2) allows d_1 blocks of type b_1 to be converted into d_2 blocks of type b_2 ; each conversion is animated in the game by repeatedly dividing blocks in half, then recombining them, so d_1 and d_2 are always constrained to be integer powers of two. Jomang wants to randomly generate allowable game conversions to increase replayability, but wants to disallow sets of game conversions through which players can generate infinite numbers of blocks via conversion. Describe an $O(n^3)$ -time algorithm to determine whether a given a set of $\lfloor \frac{1}{5}n^2 \rfloor$ conversions should be disallowed. Assume that a starting world contains D blocks of each type, where D is the product of all d_i appearing in any conversion. Given positive integer x , assume that its bit-length, $\log_2 x$, can be computed in $O(1)$ time.

Solution: Construct a graph G with a vertex for every block type and a directed edge for every conversion (d_1, b_1, d_2, b_2) , specifically the edge from b_1 to b_2 with weight $\lg(d_1) - \lg(d_2)$. Then the negative weight of any directed path from b_s to b_t in G corresponds to the base-2 logarithm of the fractional increase of the amount of blocks of type b_t that would result by performing each conversion along the path from some amount of block of type b_s (since $\lg \prod_i \frac{d_{i,2}}{d_{i,1}} = \sum_i (\lg(d_{i,2}) - \lg(d_{i,1}))$). Thus, given a sufficiently large number of starting blocks, there exists a negative-weight simple cycle in G if and only if a user could convert block types along these conversions to generate an unbounded amount of blocks of any block type reachable from the cycle. Since D upper bounds the base-2 logarithm of the sum of all edge weights in G , any simple cycle has absolute weight less than $\lg D$, so D blocks of each type will allow a block type on a negative-weight simple cycle to be converted around the cycle, without ever running out of blocks during the conversion. We can detect whether G contains a negative-weight simple cycle using Bellman-Ford in $O(|V||E|)$ time. Since G has n vertices and $\Theta(n^2)$ edges, Bellman-Ford runs in $O(n^3)$ time, as desired.

²We mean Euclidean distance between landmark (x_1, y_1) and landmark (x_2, y_2) , i.e., $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$.

Problem 6-4. Tipsy Tannister

Lyrion Tannister is a greedy, drunken nobleman residing in Prince's Pier, the capital of Easteros, who wants to travel to the northern town of Summerrise. Lyrion has a map depicting all towns and roads in Easteros, where each road allows direct travel between a pair of towns, and each town is connected to at most seven roads. Each road is marked with the non-negative number of gold pieces he will be able to collect in taxes by traveling along that road in either direction (this number may be zero). Every town has a tavern, and Lyrion has marked each town with the positive number of gold pieces he would spend if he were to drink there. If Lyrion ever runs out of money on his trip, he will just leave a debt marker³ indicating the number of gold pieces he owes. After leaving the tavern in Prince's Pier with no gold and zero debt, Lyrion's policy will be to drink at the tavern of every third town he visits along his route until reaching Summerrise. Given Lyrion's map depicting the n towns in Easteros, describe an $O(n^2)$ -time algorithm to compute the maximum amount of gold he can have (minus debts incurred) upon arriving in Summerrise, traveling from Prince's Pier along a route that follows his drinking policy.

Solution: Given Lyrion's drinking policy, it would be helpful to know at any given time how many towns ago Lyrion last visited a tavern. To keep track of this information, we will make a graph that has three vertices for each town: one vertex associated with arriving at the town having drunk at a town i towns before, either 1, 2, or 3 towns before. Construct a graph G with a vertex $v_{t,i}$ for each town t and for each $i \in \{1, 2, 3\}$. Then for each road connecting town a to town b , add directed edges $(v_{a,i}, v_{b,(i \bmod 3)+1})$ and $(v_{b,i}, v_{a,(i \bmod 3)+1})$ for each $i \in \{1, 2, 3\}$:

- when $i \in \{1, 2\}$, Lyrion does not drink in the town he is coming from, so weight the edge with the negative of the amount of gold he would collect in taxes on that road, while
- when $i = 3$, Lyrion drinks in the town he is coming from, so weight the edge with the amount he will spend at that town's tavern, minus the amount of gold he would collect in taxes on that road.

Let p and s denote the vertices associated with Prince's Pier and Summerrise respectively, and remove all outgoing edges from the three vertices associated with s (since Lyrion stops as soon as he arrives in Summerrise). If w_p is the cost of drinking at the tavern in Prince's Pier, then the weight of any directed path in G from vertex $v_{p,3}$ will be w_p less than the amount of taxes minus expenses that Lyrion will have by traversing along the roads corresponding to edges in the path while following his drinking policy. G has $O(n)$ vertices and $O(n)$ edges (since each vertex is connected to at most a constant number of edges), so this graph can be built in $O(n)$ time. Then, since edges in G may have positive or negative weights, we can use Bellman-Ford to compute the weight of the shortest path from p to s in G in $O(n^2)$ time, which will be the negative of the maximum amount of gold (or debt) he can have upon arriving in Summerrise. If Bellman-Ford returns a weight of $-\infty$ for $d(p, s)$, there is no finite upper bound on the amount Lyrion can arrive with, while if Bellman-Ford returns ∞ for $d(p, s)$, Summerrise cannot be reached along roads, so Lyrion should just stay home.

³The marker will be readily accepted because Tannisters always pay their debts.

Problem 6-5. Cloud Computing

Azrosoft Micure is a cloud computing company where users can upload **computing jobs** to be executed remotely. The company has a large number of identical cloud computers available to run code, many more than the number of pieces of code in any single job. Any piece of code may be run on any available computer at any time. Each computing job consists of a code list and a dependency list.

A **code list** C is an array of code pairs $(f, t) \in C$, where string f is the file name of a piece of code, and t is the positive integer number of microseconds needed for that code to complete when assigned to a cloud computer. Assume file names are short and can be read in $O(1)$ time.

A **dependency list** D is an array of dependency pairs $(f_1, f_2) \in D$, where f_1 and f_2 are distinct file names that appear in C . A dependency pair (f_1, f_2) indicates that the piece of code named f_1 must be completed before the piece of code named f_2 can begin. Assume that every file name exists in some dependency pair.

- (a) A job (C, D) can be **completed** if every piece of code in C can be completed while respecting the dependencies in D . Given job (C, D) , describe an $O(|D|)$ -time algorithm to decide whether the job can be completed.

Solution: Construct a graph G with a vertex for each piece of code in C and a directed edge from f_1 to f_2 for each dependency pair $(f_1, f_2) \in D$. Also add an auxiliary node s and add a directed edge (s, f) to every other vertex. A job can be completed as long as G does not contain any cycle. We can do this by running a depth-first search (DFS) from s , checking whether the reverse order of DFS finishing times is a topological order (i.e., check to make sure no edge in the graph goes against this order). Since $|C| \leq 2|D|$, G has size $O(|D|)$ and DFS can be run in $O(|D|)$ time. Checking each edge against this order can also be done in $O(1)$ time per edge, so this algorithm takes $O(|D|)$ time as desired.

- (b) Azrosoft Micure wants to know how fast they can complete a given job. Given a job (C, D) , describe an $O(|D|)$ -time algorithm to determine the minimum number of microseconds that would be needed to complete the job (or return that the job cannot be completed).

Solution: We can use the same graph as in part (a), adding weights corresponding to running times of pieces of code. Let $t(f)$ denote the time for code f to complete. Then for each edge (a, f) in G , weight it by $-t(f)$. Then the length of the shortest path from s to any other vertex in G will be equal to longest time for any piece of code to be completed. If G has a cycle, we can break as in (a); otherwise, G is a DAG, and we can run DAG Relaxation to compute the shortest path in $O(|D|)$ time.

- (c) Write a Python function `min_time(C, D)` that implements your algorithm from (b).

Solution:

```

1 def dfs(Adj, s, parent = None, order = None):      # code from R10
2     if parent is None:
3         parent = [None for v in Adj]
4         parent[s] = s
5         order = []
6     for v in Adj[s]:
7         if parent[v] is None:
8             parent[v] = s
9             dfs(Adj, v, parent, order)
10    order.append(s)
11    return parent, order
12
13 def topo_shortest_paths(Adj, w, s):                  # code from R11
14     _, order = dfs(Adj, s)
15     order.reverse()
16     d = [float('inf')] for _ in Adj
17     parent = [None for _ in Adj]
18     d[s], parent[s] = 0, s
19     for u in order:
20         for v in Adj[u]:
21             if d[v] > d[u] + w(u, v):
22                 d[v] = d[u] + w(u, v)
23                 parent[v] = u
24     return d, parent
25
26 def min_time(C, D):
27     n = len(C)
28     file_idx, file_time = {}, {}
29     for i in range(n):                                # label files from 1 to n
30         f, t = C[i]
31         file_idx[f], file_time[i + 1] = i + 1, t
32     Adj, w = [[] for _ in range(n + 1)], {}          # construct dependency graph
33     for f1, f2 in D:
34         i1, i2 = file_idx[f1], file_idx[f2]
35         Adj[i1].append(i2)
36         w[(i1, i2)] = -file_time[i2]
37     Adj[0] = list(range(1, n + 1))                   # add supernode
38     for i in range(1, n + 1):
39         w[(0, i)] = -file_time[i]
40     _, order = dfs(Adj, 0)                            # check for cycles
41     order.reverse()
42     rank = [None] * (n + 1)
43     for i in range(n + 1):
44         rank[order[i]] = i
45     for v1 in range(n + 1):
46         for v2 in Adj[v1]:
47             if rank[v1] > rank[v2]:
48                 return None
49     dist, _ = topo_shortest_paths(Adj, lambda u,v: w[(u,v)], 0)
50     return -min(dist)                                # compute min time

```

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

Lecture 13: Dijkstra's Algorithm

Review

- Single-Source Shortest Paths on weighted graphs
- Previously: $O(|V| + |E|)$ -time algorithms for small positive weights or DAGs
- Last time: Bellman-Ford, $O(|V||E|)$ -time algorithm for **general graphs** with **negative weights**
- Today: faster for **general graphs** with **non-negative edge weights**, i.e., for $e \in E$, $w(e) \geq 0$

Restrictions		SSSP Algorithm		
Graph	Weights	Name	Running Time $O(\cdot)$	Lecture
General	Unweighted	BFS	$ V + E $	L09
DAG	Any	DAG Relaxation	$ V + E $	L11
General	Any	Bellman-Ford	$ V \cdot E $	L12
General	Non-negative	Dijkstra	$ V \log V + E $	L13 (Today!)

Non-negative Edge Weights

- **Idea!** Generalize BFS approach to weighted graphs:
 - Grow a sphere centered at source s
 - Repeatedly explore closer vertices before further ones
 - But how to explore closer vertices if you don't know distances beforehand? : (
- **Observation 1:** If weights non-negative, monotonic distance increase along shortest paths
 - i.e., if vertex u appears on a shortest path from s to v , then $\delta(s, u) \leq \delta(s, v)$
 - Let $V_x \subset V$ be the subset of vertices reachable within distance $\leq x$ from s
 - If $v \in V_x$, then any shortest path from s to v only contains vertices from V_x
 - Perhaps grow V_x one vertex at a time! (but growing for every x is slow if weights large)
- **Observation 2:** Can solve SSSP fast if given order of vertices in increasing distance from s
 - Remove edges that go against this order (since cannot participate in shortest paths)
 - May still have cycles if zero-weight edges: repeatedly collapse into single vertices
 - Compute $\delta(s, v)$ for each $v \in V$ using DAG relaxation in $O(|V| + |E|)$ time

Dijkstra's Algorithm

- Named for famous Dutch computer scientist **Edsger Dijkstra** (actually Dijkstra!)

11 August 1982
 prof. dr. Edsger W. Dijkstra
 Burroughs Research Fellow

- Idea!** Relax edges from each vertex in increasing order of distance from source s
- Idea!** Efficiently find next vertex in the order using a data structure
- Changeable Priority Queue Q** on items with keys and unique IDs, supporting operations:

$Q.\text{build}(X)$	initialize Q with items in iterator X
$Q.\text{delete_min}()$	remove an item with minimum key
$Q.\text{decrease_key}(id, k)$	find stored item with ID id and change key to k
- Implement by **cross-linking** a Priority Queue Q' and a Dictionary D mapping IDs into Q'
- Assume vertex IDs are integers from 0 to $|V| - 1$ so can use a direct access array for D
- For brevity, say item x is the tuple $(x.id, x.key)$

- Set $d(s, v) = \infty$ for all $v \in V$, then set $d(s, s) = 0$
- Build changeable priority queue Q with an item $(v, d(s, v))$ for each vertex $v \in V$
- While Q not empty, delete an item $(u, d(s, u))$ from Q that has minimum $d(s, u)$

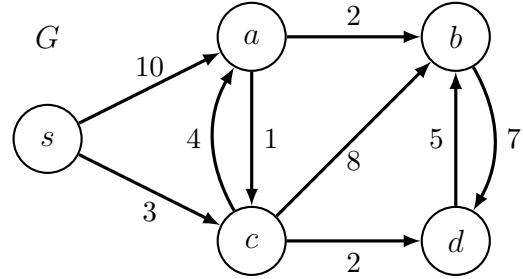
- For vertex v in outgoing adjacencies $\text{Adj}^+(u)$:

- * If $d(s, v) > d(s, u) + w(u, v)$:
 - Relax edge (u, v) , i.e., set $d(s, v) = d(s, u) + w(u, v)$
 - Decrease the key of v in Q to new estimate $d(s, v)$

- Run Dijkstra on example

Example

Delete v from Q	s	a	b	c	d
s	0	∞	∞	∞	∞
c		10	∞	3	∞
d		7	11		5
a		7	10		
b			9		
$\delta(s, v)$	0	7	9	3	5



Correctness

- **Claim:** At end of Dijkstra's algorithm, $d(s, v) = \delta(s, v)$ for all $v \in V$
- **Proof:**
 - If relaxation sets $d(s, v)$ to $\delta(s, v)$, then $d(s, v) = \delta(s, v)$ at the end of the algorithm
 - * Relaxation can only decrease estimates $d(s, v)$
 - * Relaxation is safe, i.e., maintains that each $d(s, v)$ is weight of a path to v (or ∞)
 - Suffices to show $d(s, v) = \delta(s, v)$ when vertex v is removed from Q
 - * Proof by induction on first k vertices removed from Q
 - * Base Case ($k = 1$): s is first vertex removed from Q , and $d(s, s) = 0 = \delta(s, s)$
 - * Inductive Step: Assume true for $k < k'$, consider k' 'th vertex v' removed from Q
 - * Consider some shortest path π from s to v' , with $w(\pi) = \delta(s, v')$
 - * Let (x, y) be the first edge in π where y is not among first $k' - 1$ (perhaps $y = v'$)
 - * When x was removed from Q , $d(s, x) = \delta(s, x)$ by induction, so:
 - $d(s, y) \leq \delta(s, x) + w(x, y)$ relaxed edge (x, y) when removed x
 - $= \delta(s, y)$ subpaths of shortest paths are shortest paths
 - $\leq \delta(s, v')$ non-negative edge weights
 - $\leq d(s, v')$ relaxation is safe
 - $\leq d(s, y)$ v' is vertex with minimum $d(s, v')$ in Q

$$\begin{aligned}
d(s, y) &\leq \delta(s, x) + w(x, y) && \text{relaxed edge } (x, y) \text{ when removed } x \\
&= \delta(s, y) && \text{subpaths of shortest paths are shortest paths} \\
&\leq \delta(s, v') && \text{non-negative edge weights} \\
&\leq d(s, v') && \text{relaxation is safe} \\
&\leq d(s, y) && v' \text{ is vertex with minimum } d(s, v') \text{ in } Q
\end{aligned}$$

- * So $d(s, v') = \delta(s, v')$, as desired □

Running Time

- Count operations on changeable priority queue Q , assuming it contains n items:

Operation	Time	Occurrences in Dijkstra
$Q.\text{build}(X)$ ($n = X $)	B_n	1
$Q.\text{delete_min}()$	M_n	$ V $
$Q.\text{decrease_key}(id, k)$	D_n	$ E $

- Total running time is $O(B_{|V|} + |V| \cdot M_{|V|} + |E| \cdot D_{|V|})$
- Assume pruned graph to search only vertices reachable from the source, so $|V| = O(|E|)$

Priority Queue Q' on n items	Q Operations $O(\cdot)$			Dijkstra $O(\cdot)$ $n = V = O(E)$
	build(X)	delete_min()	decrease_key(id, k)	
Array	n	n	1	$ V ^2$
Binary Heap	n	$\log n_{(a)}$	$\log n$	$ E \log V $
Fibonacci Heap	n	$\log n_{(a)}$	$1_{(a)}$	$ E + V \log V $

- If graph is **dense**, i.e., $|E| = \Theta(|V|^2)$, using an Array for Q' yields $O(|V|^2)$ time
- If graph is **sparse**, i.e., $|E| = \Theta(|V|)$, using a Binary Heap for Q' yields $O(|V| \log |V|)$ time
- A Fibonacci Heap is theoretically good in all cases, but is not used much in practice
- We won't discuss Fibonacci Heaps in 6.006 (see 6.854 or CLRS chapter 19 for details)
- You should assume Dijkstra runs in $O(|E| + |V| \log |V|)$ time when using in theory problems

Summary: Weighted Single-Source Shortest Paths

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

- What about All-Pairs Shortest Paths?
- Doing a SSSP algorithm $|V|$ times is actually pretty good, since output has size $O(|V|^2)$
- Can do better than $|V| \cdot O(|V| \cdot |E|)$ for general graphs with negative weights (next time!)

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

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

```

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.

```

1  class PriorityQueue:                      # Hash Table Implementation
2      def __init__(self):                  # stores keys with unique labels
3          self.A = {}
4
5      def insert(self, label, key):        # insert labeled key
6          self.A[label] = key
7
8      def extract_min(self):              # return a label with minimum key
9          min_label = None
10         for label in self.A:
11             if (min_label is None) or (self.A[label] < self.A[min_label].key):
12                 min_label = label
13         del self.A[min_label]
14         return min_label
15
16     def decrease_key(self, label, key):  # decrease key of a given label
17         if (label in self.A) and (key < self.A[label].key):
18             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:

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

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:

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

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.

```

1  class Item:
2      def __init__(self, label, key):
3          self.label, self.key = label, key
4
5  class PriorityQueue:                      # Binary Heap Implementation
6      def __init__(self):                  # stores keys with unique labels
7          self.A = []
8          self.label2idx = {}
9
10     def min_heapify_up(self, c):
11         if c == 0: return
12         p = (c - 1) // 2
13         if self.A[p].key > self.A[c].key:
14             self.A[c], self.A[p] = self.A[p], self.A[c]
15             self.label2idx[self.A[c].label] = c
16             self.label2idx[self.A[p].label] = p
17             self.min_heapify_up(p)
18
19     def min_heapify_down(self, p):
20         if p >= len(self.A): return
21         l = 2 * p + 1
22         r = 2 * p + 2
23         if l >= len(self.A): l = p
24         if r >= len(self.A): r = p
25         c = l if self.A[r].key > self.A[l].key else r
26         if self.A[p].key > self.A[c].key:
27             self.A[c], self.A[p] = self.A[p], self.A[c]
28             self.label2idx[self.A[c].label] = c
29             self.label2idx[self.A[p].label] = p
30             self.min_heapify_down(c)
31
32     def insert(self, label, key):        # insert labeled key
33         self.A.append(Item(label, key))
34         idx = len(self.A) - 1
35         self.label2idx[self.A[idx].label] = idx
36         self.min_heapify_up(idx)
37
38     def extract_min(self):            # remove a label with minimum key
39         self.A[0], self.A[-1] = self.A[-1], self.A[0]
40         self.label2idx[self.A[0].label] = 0
41         del self.label2idx[self.A[-1].label]
42         min_label = self.A.pop().label
43         self.min_heapify_down(0)
44         return min_label
45
46     def decrease_key(self, label, key): # decrease key of a given label
47         if label in self.label2idx:
48             idx = self.label2idx[label]
49             if key < self.A[idx].key:
50                 self.A[idx].key = key
51                 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.

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

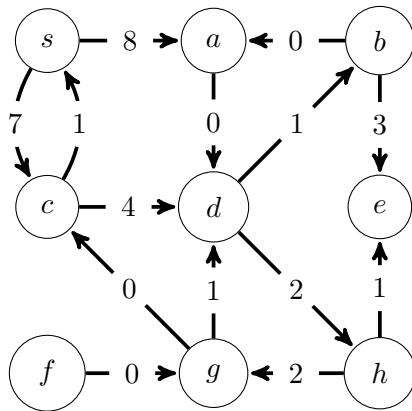
Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

Problem Session 7

Problem 7-1. Dijkstra Practice

- (a) Run Dijkstra on the following graph from vertex s to every vertex in $V = \{a, b, c, d, e, f, g, h, s\}$. Write down (1) the minimum-weight path weight $\delta(s, v)$ for each vertex $v \in V$, and (2) the order that vertices are removed from Dijkstra's queue.



Solution:

Vertex v	a	b	c	d	e	f	g	h	s
$\delta(s, v)$	8	9	7	8	11	∞	12	10	0
Removal order	3	5	2	4	7	9	8	6	1

Nodes are processed in increasing order of $\delta(s, v)$, which for this graph is unique. The processing order is not always unique; if two vertices have the same shortest path distance, it is possible that Dijkstra could process them in either order, depending on how the implementation of Dijkstra breaks ties in the priority queue. While a and d have the same shortest path distance from s , d is only reachable through a , so a must be processed first.

- (b) Change the weight of edge (g, c) to -6 . Identify a vertex v for which running Dijkstra from s as in part (a) would change the shortest path estimate to v at a time when v is not in Dijkstra's queue.

Solution: If the weight of edge (g, c) were changed to -6 , running Dijkstra's would remove vertices from the queue in the same order as in part (a). But then when g is removed from the queue, c would have already been popped from the queue and would change the shortest path estimate to c from 7 to 6, so c has the requested property.

Problem 7-2. Weighted Graph Radius

In a weighted directed graph $G = (V, E)$, the **weighted eccentricity** $\epsilon(u)$ of a vertex $u \in V$ is the shortest weighted distance to its farthest vertex v , i.e., $\epsilon(u) = \max\{\delta(u, v) \mid v \in V\}$. The **weighted radius** $R(G)$ of a weighted directed graph $G = (V, E)$ is the smallest eccentricity of any vertex, i.e., $R(G) = \min\{\epsilon(u) \mid u \in V\}$. Given a weighted directed graph G , where edge weights may be positive or negative but G contains no negative-weight cycle, describe an $O(|V|^3)$ -time algorithm to determine the graph's weighted radius (compare with Problem 2 from PS5).

Solution: Use Johnson's algorithm to compute all-pairs shortest paths distances $\delta(a, b)$ for every ordered distinct pair of vertices (a, b) in $O(|V||E| + |V|^2 \log |V|) = O(|V|^3)$ time (none of which are $-\infty$ since the graph contains no negative-weight cycles). Then, simply compute $\epsilon(u)$ directly for each $u \in V$, by computing a maximum in $O(|V|)$ time per vertex and $O(|V|^2)$ time in total, and then find the smallest $\epsilon(u)$ for $u \in V$ in $O(|V|)$ time. This algorithm is correct because we are performing the requested computation directly and Johnson's computes every $\delta(a, b)$ correctly for any graph not containing negative-weight cycles. The running time for this algorithm is bounded by Johnson's, so runs in $O(|V|^3)$ in total.

The Floyd-Warshall algorithm could also be used in combination with Bellman-Ford, but since we have not yet talked about this algorithm, students would need to fully describe and analyze Floyd-Warshall in order to use it.

Problem 7-3. Under Games

Atniss Keverdeen is a rebel spy who has been assigned to go on a reconnaissance mission to the mansion of the tyrannical dictator, President Rain. To limit exposure, she has decided to travel via an underground sewer network. She has a map of the sewer, composed of n bidirectional pipes which connect to each other at junctions. Each junction connects to at most four pipes, and every junction is reachable from every other junction via the sewer network. Every pipe is marked with its positive integer length, while some junctions are marked as containing identical motion sensors, any of which will be able to sense Atniss if her distance to that sensor (as measured along pipes in the sewer network) is too close. Unfortunately, Atniss does not know the sensitivity of the sensors. Describe an $O(n \log n)$ -time algorithm to find a path along pipes from a given entrance junction to the junction below President Rain's mansion that stays as far from motion sensors as possible.

Solution: Construct a graph G with a vertex for every junction in the sewer network and an undirected edge between junction a and junction b with weight w if a and b are connected by a pipe with length w . Since each junction connects to at most a constant number of pipes, the number of edges and vertices in this graph are both upper bounded by $O(n)$. Let s be the vertex associated with the entrance junction, and let t be the junction below President Rain's mansion. Define G_k to be the subgraph of G on only the vertices whose distance to any sensor is strictly larger than k . Our goal will be to find k^* such that s and t are in the same connected component in G_{k^*} , but s and t are not in the same connected component in G_{k^*+1} , i.e., k^* is the maximum sensor sensitivity for which it is still possible for Atniss to reach t from s undetected. If we could find k^* , then we could just return any path from s to t in G_{k^*} .

To find k^* , first label each junction with its shortest distance to any sensor. To do this, construct a new graph G' from G by adding an auxiliary vertex x and an undirected edge of weight zero from x to every vertex in G marked as containing a motion sensor, and then run Dijkstra from x in $O(n \log n)$ time. Then $\delta(x, v)$ corresponds to the shortest distance labels from v to any sensor as desired. Then we can construct G_k for any k in $O(n)$ time by looping through vertices, removing any vertex v for which $\delta(v, k) \leq k$ (also removing any edge adjacent v). Further, we can check whether t is reachable from s in G_k in $O(n)$ time

via breath-first search or depth-first search. We could find k^* in $O(nk^*)$ time by simply constructing G_k for every $k \in \{1, \dots, k^* + 1\}$, but we can find k^* faster via binary search. Sort the vertices by their distance $\delta(x, v)$ from any sensor in $O(n \log n)$ time using any optimal comparison sort algorithm (e.g., merge sort), and let d_i be the i^{th} largest distance in this sorted order for $i \in \{1, \dots, n\}$, where d_1 is the smallest and d_n is the largest. Then construct G_{d_i} for $i = \lceil n/2 \rceil$ and check whether t is reachable from s in G_{d_i} in $O(n)$ time. If it is, recurse for $i > \lceil n/2 \rceil$, and if not, recurse for $i < \lceil n/2 \rceil$. The binary search proceeds in $O(\log n)$ iterations that each take at most $O(n)$ time until finding the smallest $d_i = d^*$ such that t is not reachable from s . Then t is reachable from s for $k < d^*$ but not for $k \geq d^*$, so $k^* = d^* - 1$. Finally, we can use either breadth-first search or depth-first search while storing parent pointers to reconstruct some path from s to t in G_{k^*} in $O(n)$ time. If $k^* = 0$, every path from s to t goes through a junction containing a sensor, so any path from s to t may be returned.

Problem 7-4. Critter Collection

Ashley Getem (from PS3) is trying to walk from Trundle Town to Blue Bluff, which are both clearings in the Tanko region. She has a map of all clearings and two-way trails in Tanko. Each of the n clearings connects to at most five trails, while each trail t directly connects two clearings and is marked with its length ℓ_t and capacity c_t of critters living on it, both positive integers. Ashley is a compulsive collector and will collect every critter she comes across by throwing an empty Pocket Sphere at it (which will fill the Pocket Sphere so it can no longer be used). If she encounters a critter without an empty Pocket Sphere, she will be sad. Whenever Ashley reaches a clearing, all critters on all trails will respawn to max capacity. Some clearings contain stores where Ashley can buy empty Pocket Spheres, and deposit full ones. Ashley has more money than she knows what to do with, but her backpack can only hold k Pocket Spheres at a time. Given Ashley's map, describe an $O(nk \log(nk))$ -time algorithm to return the shortest route for Ashley to walk from Trundle Town (with a backpack full of empty Pocket Spheres) to Blue Bluff without ever being sad, or return that sadness is unavoidable.

Solution: Construct a graph $G = (V, E)$ with $k+1$ vertices for each clearing, where vertex $v_{c,i}$ corresponds to being at clearing c while having i empty pocket spheres. Then, for each directed pair of clearings (a, b) for which a and b are connected by a trail t , having length ℓ_t and critters, add the following directed edges:

- If clearing a does not contain a store, add an edge of weight ℓ_t from $v_{a,i}$ to $v_{b,i-c_t}$ for every $i \in \{c_t, \dots, k\}$, since Ashley will use up c_t pocket spheres by traversing the trail.
- Otherwise, if clearing a contains a store, add an edge of weight ℓ_t from $v_{a,i}$ to $v_{b,k-c_t}$ for every $i \in \{0, \dots, k\}$, since it is never bad for Ashley to completely fill her backpack with empty pocket spheres whenever she leaves a store.

G has $(k+1)n = O(kn)$ vertices and at most k edges per trail. Since there are at most five trails per clearing, there are at most $5kn = O(kn)$ edges in G , so G has size $O(nk)$. Let s be the vertex associated with Trundle Town and t be the vertex associated with Blue Bluff. Then any path from $v_{s,k}$ to any vertex $v_{t,i}$ for $i \in \{0, \dots, k\}$ in G avoids sadness, and every path that avoids sadness corresponds to a path in G . Since the weights in G are non-negative, we can run Dijkstra, storing parent pointers to reconstruct a shortest route from s to t that avoids sadness in $O(nk \log(nk))$ time. If the shortest path weight to every $v_{t,i}$ in G is infinite, then return that sadness is unavoidable.

Problem 7-5. Shipping Servers

The video streaming service UsTube has decided to relocate across country, and needs to ship their servers by truck from San Francisco, CA to Cambridge, MA. They will pay third-party trucking companies to transport servers from city to city. An intern at UsTube has compiled a list R of all n available trucking routes; each available route $r_i \in R$ is a tuple (s_i, t_i, w_i, c_i) where s_i and t_i are respectively the names¹ of the starting and ending cities of the trucking route, w_i is the positive integer weight capacity of the truck, and c_i is the positive integer cost of shipping along that route (cost is the same for shipping any weight from 0 to w_i). Note that the existence of a shipping route from s_i to t_i does not imply a shipping route from t_i to s_i . Some of UsTube's servers are too heavy to fit on any truck, so they will need to transfer them to smaller servers for the move. Assume that it is possible to ship some finite weight from San Francisco to Cambridge via some routes in R . Help UsTube evaluate their shipping options.

- (a) (Useful Digression)** Given a weighted path π , its **bottleneck** is the minimum weight of any edge along the path. Given a directed graph containing vertices s and t , let $b(s, t)$ denote the maximum bottleneck of any path from s to t , and let $I(t)$ denote the set of incoming neighbors of t . Argue that $b(s, t) \geq \min(b(s, v), w(v, t))$ for every $v \in I(t)$, and that $b(s, t) = \min(b(s, v^*), w(v^*, t))$ for at least one $v^* \in I(t)$.

Solution: First, we argue that $b(s, t) \geq \min(b(s, v), w(v, t))$ for every $v \in I(t)$. Suppose for contradiction that there exist two vertices s and t such that $b(s, t) < \min(b(s, v), w(v, t))$. Then there exists some path from s to v for which the minimum weight of any edge along the path is strictly greater than $b(s, t)$. So extending that path along the edge from v to t would result in a path from s to t for which every edge along the path has weight greater than $b(s, t)$, i.e., a larger bottleneck, contradicting that $b(s, t)$ is the maximum bottleneck over all paths.

Next, we argue that $b(s, t) = \min(b(s, v^*), w(v^*, t))$ for at least one $v^* \in I(t)$. Suppose for contradiction that $b(s, t) > \min(b(s, v), w(v, t))$ for every $v \in I(t)$. Then there exists a path π from s to t whose bottleneck equals $b(s, t)$. Let v be the vertex appearing before t along the path. The bottleneck of π cannot be the weight of the edge (v, t) or else $b(s, t) = w(v, t)$; but the bottleneck can also not occur for some other edge in the path from s to v in π , or else $b(s, t) = b(s, v)$, a contradiction.

- (b)** Assuming that the number of cities appearing in any of the n trucking routes is less than $3\sqrt{n}$, describe an $O(n)$ -time algorithm to return both: (1) the weight w^* of the largest single server that can be shipped from San Francisco to Cambridge via a sequence of trucking routes, and (2) the minimum cost to ship such a server with weight w^* .

Solution: Suppose we knew w^* . Then we could construct a graph $G_c = (V_c, E_c)$ with a vertex for every city and a directed edge from vertex s_i to t_i weighted by c_i for every trucking route (s_i, t_i, w_i, c_i) for which $w_i \geq w^*$, and then run Dijkstra from the vertex corresponding to San Francisco to find the minimum cost of shipping weight w^* to Cambridge. This graph has n edges and at most $O(\sqrt{n})$ vertices (by the problem statement), so Dijkstra runs in $O(n)$ time, even when a direct access array or hash table is used for Dijkstra's priority queue.

To find w^* , we construct a graph $G_w = (V_w, E_w)$, similar to G_c , with a vertex for every city and a directed edge from vertex s_i to t_i weighted by w_i for every trucking route (s_i, t_i, w_i, c_i) , and let s and t be the vertices associated with San Francisco and Cambridge respectively. The problem is then to determine the maximum bottleneck of any path from s to t . We can modify

¹Assume names are ASCII strings that can each be read in constant time.

Dijkstra to compute the bottleneck by replacing shortest path distance estimates with bottleneck estimates $b_s(v)$. Specifically, initialize all bottleneck estimates to zero (since not shipping anything achieves this), except for $b_s(s)$ which we set equal to positive infinity, since we have no limits at the start; and then add them all to a maximum priority queue. Then, repeatedly remove the vertex with largest bottleneck estimate from the priority queue and relax all outgoing edges from it. To relax an edge (u, v) with weight w_i , set $b_s(v) = \max(\min(b_s(u), w_i), b_s(v))$ (either a path through u to v improves on the estimate for $b_s(v)$ or it does not).

To prove that this modification to Dijkstra is correct, we can follow an identical proof structure as the one provided in the Lecture 13 notes, proving correctness of normal Dijkstra. Specifically, we can prove the analogue of Claim 2: for every $v \in V_w$, when v is removed from Dijkstra's queue, $b_s(v) = b(s, v)$. We omit the full proof here. For this problem, students can receive full points for a description of a correct description, without a full proof of correctness.

After the modified Dijkstra has completed, $w^* = b_s(t) = b(s, t)$ by definition of w^* . Since we only changed the relaxation step from one constant time operation to another, and follow the same structure as Dijkstra on a graph with $O(n)$ edges and $O(\sqrt{n})$ vertices, this modified Dijkstra also runs in $O(n)$ time (using a direct access array or hash table for Dijkstra's priority queue).

- (c) Write a Python function `ship_server_stats(R, s, t)` that implements your algorithm from (b).

Solution:

```

1 def dijkstra(Adj, w, s):           # modified from R13
2     d = [float('inf') for _ in Adj]
3     d[s] = 0
4     Q = [i for i in range(len(Adj))]    # direct access as priority queue
5     while len(Q) > 0:
6         u = Q[0]                      # find min distance in Q
7         for v in Q:
8             if d[v] < d[u]:
9                 u = v
10        Q.remove(u)                  # remove min
11        for v in Adj[u]:            # relax outgoing edges
12            d[v] = min(d[v], d[u] + w(u, v))
13    return d
14
15 def dijkstra_bottleneck(Adj, w, s):   # modified from R13
16     d = [0 for _ in Adj]              # initially vertices not reached
17     d[s] = float('inf')              # source has arbitrary capacity
18     Q = [i for i in range(len(Adj))]
19     while len(Q) > 0:
20         u = Q[0]                    # find max capacity in Q
21         for v in Q:
22             if d[v] > d[u]:
23                 u = v
24         Q.remove(u)
25         for v in Adj[u]:          # relax outgoing edges
26             d[v] = max(d[v], min(d[u], w(u, v)))
27    return d
28
29 def ship_server_stats(R, s, t):
30     n = 0
31     city_idx = {}                   # label cities with integers
32     for (_s, _t, _w, _c) in R:
33         for city in (_s, _t):
34             if city not in city_idx:
35                 city_idx[city] = n
36             n += 1
37     Adj = [[] for i in range(n)]      # construct graph
38     w_w, w_c = {}, {}
39     for (_s, _t, _w, _c) in R:
40         si, ti = city_idx[_s], city_idx[_t]
41         Adj[si].append(ti)
42         w_w[(si, ti)], w_c[(si, ti)] = _w, _c
43     si, ti = city_idx[s], city_idx[t]  # find max bottleneck weight
44     w = dijkstra_bottleneck(Adj, lambda u,v: w_w[(u,v)], si)[ti]
45     for i in range(n):               # remove edges above bottleneck
46         for j in Adj[i]:
47             if w_w[(i, j)] < w:
48                 w_c[(i, j)] = float('inf')
49     c = dijkstra(Adj, lambda u,v: w_c[(u,v)], si)[ti] # compute min length
50     return w, c

```

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

Lecture 14: Johnson's Algorithm

Previously

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

All-Pairs Shortest Paths (APSP)

- **Input:** directed graph $G = (V, E)$ with weights $w : E \rightarrow \mathbb{Z}$
- **Output:** $\delta(u, v)$ for all $u, v \in V$, or abort if G contains negative-weight cycle
- Useful when understanding whole network, e.g., transportation, circuit layout, supply chains...
- Just doing a SSSP algorithm $|V|$ times is actually pretty good, since output has size $O(|V|^2)$
 - $|V| \cdot O(|V| + |E|)$ with BFS if weights positive and bounded by $O(|V| + |E|)$
 - $|V| \cdot O(|V| + |E|)$ with DAG Relaxation if acyclic
 - $|V| \cdot O(|V| \log |V| + |E|)$ with Dijkstra if weights non-negative or graph undirected
 - $|V| \cdot O(|V| \cdot |E|)$ with Bellman-Ford (general)
- **Today:** Solve APSP in any weighted graph in $|V| \cdot O(|V| \log |V| + |E|)$ time

Approach

- **Idea:** Make all edge weights non-negative while **preserving shortest paths!**
- i.e., reweight G to G' with no negative weights, where a shortest path in G is shortest in G'
- If non-negative, then just run Dijkstra $|V|$ times to solve APSP
- **Claim:** Can compute distances in G from distances in G' in $O(|V|(|V| + |E|))$ time
 - Compute shortest-path tree from distances, for each $s \in V'$ in $O(|V| + |E|)$ time (L11)
 - Also shortest-paths tree in G , so traverse tree with DFS while also computing distances
 - Takes $O(|V| \cdot (|V| + |E|))$ time (which is less time than $|V|$ times Dijkstra)
- But how to make G' with non-negative edge weights? Is this even possible??
- **Claim:** Not possible if G contains a negative-weight cycle
- **Proof:** Shortest paths are simple if no negative weights, but not if negative-weight cycle \square
- Given graph G with negative weights but no negative-weight cycles,
can we make edge weights non-negative while preserving shortest paths?

Making Weights Non-negative

- **Idea!** Add negative of smallest weight in G to every edge! All weights non-negative! :)
- **FAIL:** Does not preserve shortest paths! Biases toward paths traversing fewer edges :(
- **Idea!** Given vertex v , add h to all **outgoing edges** and subtract h from all **incoming edges**
- **Claim:** Shortest paths are preserved under the above reweighting
- **Proof:**
 - Weight of every path starting at v changes by h
 - Weight of every path ending at v changes by $-h$
 - Weight of a path passing through v **does not change** (locally) \square
- This is a very general and useful trick to transform a graph while preserving shortest paths!

- Even works with multiple vertices!
- Define a **potential function** $h : V \rightarrow \mathbb{Z}$ mapping each vertex $v \in V$ to a potential $h(v)$
- Make graph G' : same as G but edge $(u, v) \in E$ has weight $w'(u, v) = w(u, v) + h(u) - h(v)$
- **Claim:** Shortest paths in G are also shortest paths in G'
- **Proof:**
 - Weight of path $\pi = (v_0, \dots, v_k)$ in G is $w(\pi) = \sum_{i=1}^k w(v_{i-1}, v_i)$
 - Weight of π in G' is: $\sum_{i=1}^k w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i) = w(\pi) + h(v_0) - h(v_k)$
 - (Sum of h 's telescope, since there is a positive and negative $h(v_i)$ for each interior i)
 - Every path from v_0 to v_k changes by the same amount
 - So any shortest path will still be shortest

□

Making Weights Non-negative

- Can we find a potential function such that G' has no negative edge weights?
- i.e., is there an h such that $w(u, v) + h(u) - h(v) \geq 0$ for every $(u, v) \in E$?
- Re-arrange this condition to $h(v) \leq h(u) + w(u, v)$, looks like **triangle inequality!**
- **Idea!** Condition would be satisfied if $h(v) = \delta(s, v)$ and $\delta(s, v)$ is finite for some s
- But graph may be disconnected, so may not exist any such vertex s ... : (
- **Idea!** Add a new vertex s with a directed 0-weight edge to every $v \in V$! :)
- $\delta(s, v) \leq 0$ for all $v \in V$, since path exists a path of weight 0
- **Claim:** If $\delta(s, v) = -\infty$ for any $v \in V$, then the original graph has a negative-weight cycle
- **Proof:**
 - Adding s does not introduce new cycles (s has no incoming edges)
 - So if reweighted graph has a negative-weight cycle, so does the original graph
- Alternatively, if $\delta(s, v)$ is finite for all $v \in V$:
 - $w'(u, v) = w(u, v) + h(u) - h(v) \geq 0$ for every $(u, v) \in E$ by triangle inequality!
 - New weights in G' are non-negative while preserving shortest paths!

□

Johnson's Algorithm

- Construct G_x from G by adding vertex x connected to each vertex $v \in V$ with 0-weight edge
- Compute $\delta_x(x, v)$ for every $v \in V$ (using Bellman-Ford)
- If $\delta_x(x, v) = -\infty$ for any $v \in V$:
 - Abort (since there is a negative-weight cycle in G)
- Else:
 - Reweight each edge $w'(u, v) = w(u, v) + \delta_x(x, u) - \delta_x(x, v)$ to form graph G'
 - For each $u \in V$:
 - * Compute shortest-path distances $\delta'(u, v)$ to all v in G' (using Dijkstra)
 - * Compute $\delta(u, v) = \delta'(u, v) - \delta_x(x, u) + \delta_x(x, v)$ for all $v \in V$

Correctness

- Already proved that transformation from G to G' preserves shortest paths
- Rest reduces to correctness of Bellman-Ford and Dijkstra
- Reducing from **Signed APSP** to **Non-negative APSP**
- Reductions save time! No induction today! :)

Running Time

- $O(|V| + |E|)$ time to construct G_x
- $O(|V||E|)$ time for Bellman-Ford
- $O(|V| + |E|)$ time to construct G'
- $O(|V| \cdot (|V| \log |V| + |E|))$ time for $|V|$ runs of Dijkstra
- $O(|V|^2)$ time to compute distances in G from distances in G'
- $O(|V|^2 \log |V| + |V||E|)$ time in total

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

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	$ V + E $
DAG	Any	DAG Relaxation	$ V + E $
General	Non-negative	Dijkstra	$ V \log V + E $
General	Any	Bellman-Ford	$ V \cdot E $

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 APSP 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 APSP 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, \dots, 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:

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

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. \square

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.) \square

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 $\delta'(u, v) - \delta(x, u) + \delta(x, v)$. 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 $O(|V|^2|E|)$.

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

Quiz 3 Review

Scope

- Quiz 1 & Quiz 2 material fair game but explicitly **not emphasized**
 - 4 lectures on dynamic programming, L15-L18, 2 Problem Sets, PS7-PS8
 - Recursive framework (SRTBOT)
 - Dynamic Programming: subproblem dependencies **overlap**, forming a DAG
-

Recursive Framework (SRT BOT)

1. **Subproblem** definition
 - Describe meaning of subproblem **in words**, in terms of parameters $x \in X$
 - Often subsets of input: prefixes, suffixes, contiguous substrings of a sequence
 - Often multiply possible subsets across multiple inputs
 - Often “remember” information by maintaining some auxiliary state (expansion)
 - Often expand based on state of integers in problem space (pseudopolynomial?)
2. **Relate** recursively
 - Relate subproblem solutions recursively $x(i) = f(x(j), \dots)$ for one or more $j < i$
 - Identify question about subproblem solution whose answer would let you reduce to smaller subproblem(s), then locally brute-force all possible answers to the question
3. **Topological order**
 - Argue relation is acyclic (defines “smaller” subproblems), subproblems form a DAG
4. **Base cases**
 - State solutions for all (reachable) independent subproblems where relation breaks down
5. **Original** problem
 - Show how to compute solution to original problem from solutions to subproblem(s)
 - 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

Problem 1. Future Investing (adapted from S18 PS9)

Tiffany Bannen stumbles upon a lottery chart dropped by a time traveler from the future, which lists winning lottery numbers and positive integer cash payouts for the next n days. Tiffany wants to use this information to make money, but is worried if she plays winning numbers every day, lottery organizers will get suspicious. As such, she decides to play the lottery **infrequently**: at most **twice in any seven day period**. Describe a $O(n)$ -time algorithm to determine the maximum amount of lottery winnings Tiff can win in the next n days by playing the lottery infrequently.

Solution:**1. Subproblems**

- Let $L(i)$ be the cash payout of playing the lottery on day $i \in \{1, \dots, n\}$
- Need to keep track of most recent two plays (or equivalently, restrictions on future plays)
- $x(i, j)$: maximum lottery winnings playing on suffix of days from i to n , assuming play on day i and next allowable play is on day $i + j$
- for $i \in \{1, \dots, n\}$ and $j \in \{1, \dots, 6\}$

2. Relate

- Tiffany will play again on some day in future. Guess!
- It is never optimal to go 11 days without playing the lottery, as playing on the 6th day would be valid and strictly increase winnings
- The next play can be on day $i + k$ for $k \in \{j, \dots, 11\}$
- If next play on $i + k$ for $k \in \{1, \dots, 6\}$, next allowable play is on day $i + 7$
- If next play on $i + k$ for $k \in \{7, \dots, 11\}$, next allowable play is on day $i + k + 1$
- $x(i, j) = L(i) + \max\{x(i + k, \max\{1, 7 - k\}) \mid k \in \{i, \dots, 11\} \text{ and } i + k \leq n\}$

3. Topo

- Subproblems $x(i, j)$ only depend on strictly larger i , so acyclic

4. Base

- $x(n, j) = L(i)$ for all $j \in \{1, \dots, 6\}$

5. Original

- Solve subproblems via recursive top down or iterative bottom up
- Guess first play (within first seven days)
- Solution to original problem is $\max\{x(i, 1) \mid i \in \{1, \dots, 7\}\}$
- (Can store parent pointers to reconstruct days played)

6. Time

- # subproblems: $6n$
- work per subproblem: $O(1)$
- work for original: $O(1)$
- $O(n)$ running time

Problem 2. Oh Charlie, Where Art Thou? (adapted from S18 PS9)

A wealthy family, Alice, Bob, and their young son Charlie are sailing around the world when they encounter a massive storm. Charlie is thrown overboard, presumed drowned. Twenty years later, a man comes to Alice and Bob claiming to be Charlie. Alice and Bob are excited, but skeptical. Alice and Bob order a DNA matching test from the genetic testing company 46AndThee. Given three length- n DNA sequences from Alice, Bob, and Charlie, the testing center will determine **ancestry** as follows: if Charlie's DNA can be partitioned into two (not necessarily contiguous) subsequences of equal length, where one is a subsequence of Alice's DNA, and the other is a subsequence of Bob's DNA, the Charlie is their son. For example, suppose Alice's DNA is AATT and Bob's DNA is CCGG. If Charlie's DNA were CATG, he would be matched as their son, since CATG can be partitioned into disjoint subsequences CG and AT which are subsequences of Alice and Bob's DNA respectively. However, Charlie would be found to be an imposter if his DNA were AGTC. Describe an $O(n^4)$ -time algorithm to determine whether Charlie is a fraud.

Solution:

1. Subproblems

- Let A , B , and C be the relevant length- n DNA sequences from Alice, Bob, and Charlie.
- Want to match **some** characters of A and B to **all** characters of C
- $x(i, j, k_i, k_j)$: true if can match a length- k_i subsequence of suffix $A[i :]$ and a length- k_j characters from prefix $B[j :]$ to all characters in suffix $C[(n - k_i - k_j) :]$ (the suffix containing the last $k_i + k_j$ characters), and false otherwise.
- for $i, j \in \{0, \dots, n\}$ and $k_i, k_j \in \{0, \dots, n/2\}$ (assume n is even)

2. Relate

- Must match character $C[i]$; if $A[i] = C[i]$ or $B[i] = C[i]$ recurse on remainder
- Alternatively, do not use either $A[i]$ or $B[i]$

$$\bullet \quad x(i, j, k_i, k_j) = \text{OR} \left\{ \begin{array}{ll} x(i+1, j, k_i+1, k_j) & \text{if } A[i] = C[n - k_i - k_j] \text{ and } k_i > 0, \\ x(i, j+1, k_i, k_j+1) & \text{if } B[j] = C[n - k_i - k_j] \text{ and } k_j > 0, \\ x(i+1, j, k_i, k_j) & \text{if } i < n, \\ x(i, j+1, k_i, k_j) & \text{if } j < n \end{array} \right\}$$

3. Topo

- Subproblem $x(i, j, k_i, k_j)$ only depends on strictly smaller $i + j$, so acyclic

4. Base

- $x(n, n, 0, 0)$ is true (all matched!)
- $x(n, j, k_i, k_j)$ false if $k_i > 0$ (no more characters in A)
- $x(i, n, k_i, k_j)$ false if $k_j > 0$ (no more characters in B)

5. Original

- Solve subproblems via recursive top down or iterative bottom up
- Solution to original problem is $x(n, n, n/2, n/2)$

6. Time

- # subproblems: $O(n^4)$
- work per subproblem: $O(1)$
- $O(n^4)$ running time

Problem 3. Sweet Tapas (adapted from S18 Final)

Obert Atkins is having dinner at an upscale tapas bar, where he will order many small plates. There are n plates of food on the menu, where information for plate i is given by a triple of non-negative integers (v_i, c_i, s_i) : the plate's volume v_i , calories c_i , and sweetness $s_i \in \{0, 1\}$ (the plate is sweet if $s_i = 1$ and not sweet if $s_i = 0$). Obert is on a **diet**: he wants to eat no more than k calories during his meal, but wants to fill his stomach as much as possible. He also wants to order exactly $s < n$ sweet plates, without purchasing the same dish twice. Describe an $O(nks)$ -time algorithm to find the maximum volume of food Obert can eat given his diet.

Solution:

1. Subproblems

- $x(i, j, s')$: maximum volume of food possible purchasing a suffix of plates p_i to p_{n-1} , using at most j calories, ordering exactly s' sweet plates
- for $i \in \{0, \dots, n\}$, $j \in \{0, \dots, k\}$, $s' \in \{0, \dots, s\}$

2. Relate

- Either order plate p_i or not. Guess!
- If order p_i , get v_i in volume but use c_i calories
- If p_i is sweet, need to order one fewer sweet plate

$$\bullet x(i, j, s') = \max \begin{cases} v_i + x(i+1, j - c_i, s' - s_i) & \text{if } c_i \leq j \text{ and } s_i \leq s', \\ x(i+1, j, s') & \text{always} \end{cases}$$

3. Topo

- Subproblems $x(i, j, s')$ only depend on strictly larger i , so acyclic

4. Base

- $x(n, j, 0) = 0$ and any j (no more plates to eat)
- $x(n, j, s') = -\infty$ for $s' > 0$ and any j (no more plates, but still need to eat sweet)

5. Original

- Solution given by $x(0, k, s)$

6. Time

- # subproblems: $O(nks)$
- work per subproblem: $O(1)$
- $O(nks)$ running time

Problem 4. Gokemon Po (adapted from S18 Final)

Kash Etchum wants to play Gokemon Po, a new augmented reality game. The goal is to catch a set of n monsters who reside at specific locations in her town. Monsters must be obtained in a specific order: before Kash can obtain monster m_i , she must have already obtained all monsters m_j for $j < i$. To obtain monster m_i , Kash may either purchase the monster in-game for positive integer c_i dollars, or she may catch m_i for free from that monster's location. If Kash is not at the monster's location, she will have to pay a ride share service to drive her there. The **minimum possible cost** to transport from the location of monster m_i to the location of monster m_j via ride sharing is denoted by the positive integer $s(i, j)$. Given the price lists of in-game purchases and ride sharing trips between all pairs of monsters, describe an $O(n^2)$ -time algorithm to determine the minimum amount of money Kash must spend in order to catch all n monsters, assuming that she starts at the location of monster m_1 .

Solution:**1. Subproblems**

- $x(i, j)$: min cost of catching monsters m_i to m_n starting at location m_j for $j \leq i$

2. Relate

- If at location of monster, catch it for free!
- Otherwise, either acquire monster m_i by purchasing or ride-sharing to location. Guess!
- If purchase spend c_i dollars, else need to ride share to m_i from m_j

$$\bullet \quad x(i, j) = \begin{cases} x(i + 1, j) & \text{if } j = i \\ \min\{c_i + x(i + 1, j), s(j, i) + x(i, i)\} & \text{otherwise} \end{cases}$$

3. Topo

- Subproblems $x(i, j)$ only depend on strictly larger $i + j$ so acyclic

4. Base

- $x(n + 1, j) = 0$ for any j (no cost to collect no monsters)

5. Original

- Solution given by $x(1, 1)$

6. Time

- # subproblems: $O(n^2)$
- work per subproblem: $O(1)$
- $O(n^2)$ running time

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

Solution: Quiz 2

- Do not open this quiz booklet until directed to do so. Read all the instructions on this page.
- When the quiz begins, write your name on the top of every page of this quiz booklet.
- You have 90 minutes to earn a maximum of 90 points. Do not spend too much time on any one problem. Skim them all first, and attack them in the order that allows you to make the most progress.
- **You are allowed two double-sided letter-sized sheet with your own notes.** No calculators, cell phones, or other programmable or communication devices are permitted.
- Write your solutions in the space provided. Pages will be scanned and separated for grading. If you need more space, write “Continued on S1” (or S2, S3) and continue your solution on the referenced scratch page at the end of the exam.
- Do not waste time and paper rederiving facts that we have studied in lecture, recitation, or problem sets. Simply cite them.
- When writing an algorithm, a **clear** description in English will suffice. Pseudo-code is not required. Be sure to argue that your **algorithm is correct**, and analyze the **asymptotic running time of your algorithm**. Even if your algorithm does not meet a requested bound, you **may** receive partial credit for inefficient solutions that are correct.
- **Pay close attention to the instructions for each problem.** Depending on the problem, partial credit may be awarded for incomplete answers.

Problem	Parts	Points
1: Information	2	2
2: Exact Edges	1	16
3: Color Cost	1	18
4: Orkscapade	1	18
5: Count Cycles	1	18
6: Bellham's Fjord	1	18
Total		90

Name: _____

School Email: _____

Problem 1. [2 points] **Information** (2 parts)

- (a) [1 point] Write your name and email address on the cover page.

Solution: OK!

- (b) [1 point] Write your name at the top of each page.

Solution: OK!

Problem 2. [16 points] **Exact Edges**

Given a weighted, directed graph $G = (V, E, w)$ with positive and negative edge weights, and given a particular vertex $v \in V$, describe an $O(k|E|)$ -time algorithm to return the minimum weight of any cycle containing vertex v that also has exactly k edges, or return that no such cycle exists. Recall that a cycle may repeat vertices/edges.

Solution: Assume all vertices in G are reachable from v so that $|V| = O(|E|)$; otherwise, run BFS or DFS to solve single source reachability from v , and replace G with the subgraph reachable from v in $O(|E|)$ time. Construct a new graph $G' = (V', E')$ with:

- $k + 1$ vertices for each vertex $v \in V$: specifically v_i for all $i \in \{0, \dots, k\}$; and
- k edges for each edge $(u, v) \in E$: specifically edges (u_{i-1}, v_i) for all $i \in \{1, \dots, k\}$.

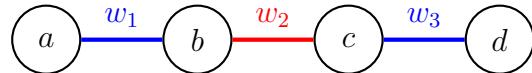
Graph G' has $(k + 1)|V| = O(k|E|)$ vertices and $k|E|$ edges in $k + 1$ layers, and has the property that paths from v_0 to v_k have a one-to-one correspondence with cycles through v in G of the same weight containing exactly k edges, since exactly one edge is traversed with each increase in layer. So solve SSSP from v_0 in G' to return the minimum weight path to v_k . Since edges in G' always increase in subscript, G' is a DAG, so we can solve SSSP using DAG relaxation in linear time with respect to the size of G' . So together with initial pruning, this algorithm takes $O(k|E|)$ time in total.

Common Mistakes:

- Using BFS to detect cycles in a directed graph (need DFS)
- Trying to enumerate all paths or cycles of length k (may be exponential)
- Using Bellman-Ford without removing zero-weight edges (may use fewer than k edges)
- Finding cycles using $k - 1$ edges, not k edges
- Trying to find negative-weight cycles (wrong problem)
- Not running a reachability algorithm to prune to graph reachable from v

Problem 3. [18 points] Color Cost

A **3-color labeling** of a graph maps each edge to either red, green, or blue. The **color cost** of a 3-color labeled path is its path weight plus a positive integer w_c every time the path changes color. For example, in the graph below (having four vertices $\{a, b, c, d\}$, a blue edge (a, b) with weight w_1 , a red edge (b, c) with weight w_2 , and another blue edge (c, d) with weight w_3) the path (a, b, c, d) has color cost $w_1 + w_2 + w_3 + 2w_c$, because the path changes color twice.



Given a 3-color labeling $c : E \rightarrow \{\text{red, green, blue}\}$ of a connected, weighted, undirected graph $G = (V, E, w)$ containing only **positive edge weights**, and given two vertices $s, t \in V$, describe an **efficient** algorithm to return a path from s to t having minimum color cost.

(By “efficient”, we mean that faster correct algorithms will receive more points than slower ones.)

Solution: Construct a new graph $G' = (V', E')$ with:

- 3 vertices for each vertex $v \in V$: specifically v_i for $i \in \{\text{red, green, blue}\}$ corresponding to arriving at vertex v via an edge with color i ;
- (vertex-edges) 3 undirected edges for each vertex $v \in V$: specifically $\{v_{\text{red}}, v_{\text{blue}}\}$, $\{v_{\text{green}}, v_{\text{red}}\}$, and $\{v_{\text{blue}}, v_{\text{green}}\}$ of weight w_c ; and
- (edge-edges) 1 undirected edge for each undirected edge $\{u, v\} \in E$ of weight w and color $c(u, v)$: specifically undirected edge $\{u_{c(u,v)}, v_{c(u,v)}\}$ with weight w .

Graph G' has $3|V|$ vertices and $3|V| + |E|$ edges, and has the property that the minimum weight of any path in G' from any vertex s_i to any vertex t_j for $i, j \in \{\text{red, green, blue}\}$ is equal to the minimum color cost of any 3-color labeled path in G from s to t , as switching colors at a vertex requires traversing an edge of weight w_c . So solve SSSP three times, once from each vertex s_i and find the minimum weight of any path to any t_j , and then return a minimum path by constructing parent pointers as shown in lecture. Since this graph only has positive edge weights, we can solve SSSP using Dijkstra in $O(|V| + |E| + |V| \log |V|) = O(|E| + |V| \log |V|)$ time.

Note that you can avoid running Dijkstra three times via a supernode, but this only reduces work by a constant factor. Also, one can also avoid adding vertex-edges by adding three edges for each edge connected and weighted appropriately, but these edges will need to be **directed** toward a vertex labeled with the same color as the corresponding edge.

Common Mistakes:

- Incorrectly trying to modify Dijkstra to keep track of state
- Failing to identify (or identifying incorrectly) a source from which to run SSSP
- Weighting or directing edges in a duplicated graph incorrectly

Problem 4. [18 points] **Orkscapade**

Ranger Raargorn needs to deliver a message from her home town of Tina's Mirth to the town of Riverdell, but the towns of Midgard have been overrun by an army of k Orks. Raargorn has a map of the n towns and $3n$ roads in Midgard, where each road connects a pair of towns in both directions. Scouts have determined the number of Orks $r_i \geq 1$ stationed in each town i (there is **at least one Ork** stationed in each town). Describe an $O(k)$ -time algorithm to find a path from Tina's Mirth to Riverdell on which Raargorn will encounter the fewest total Orks in towns along the way. **Partial credit** will be awarded for slower correct algorithms, e.g., $O(k \log k)$ or $O(nk)$.

Solution: Construct a graph $G = (V, E)$ with:

- a chain of r_i vertices (v_1, \dots, v_{r_i}) connected by $r_i - 1$ edges for each town v , i.e., unweighted directed edge (v_i, v_{i+1}) for all $i \in \{1, \dots, r_i - 1\}$; and
- two unweighted directed edges (u_{r_u}, v_1) and (v_{r_v}, u_1) for each road between towns u and v .

Graph G has $\sum_v r_v = k$ vertices and $2(3n) + \sum_v (r_v - 1) = 5n + k$ edges. Since there is at least one Ork in each town, $k \geq n$, so G has size $O(k)$. Let s and t correspond to the towns of Tina's Mirth and Riverdell respectively. Graph G has the property that any path from s_1 to t_{r_t} corresponds to a path from Tina's Mirth to Riverdell crossing edges equal to the number of Orks encountered in towns along the way, since for any road connecting towns u and v , going from u_1 to v_1 requires traversing r_v edges in G . So solve unweighted SSSP from s_1 to t_{r_t} using BFS in $O(k)$ time, and return the sequence of towns visited along the found shortest path by following parent pointers.

Common Mistakes:

- Not directing edges with vertex weight, or otherwise not clearly defining a graph
- Expanding weights on all edges, leading to an $O(nk)$ expansion
- (e.g., a town with $\Theta(k)$ Orks may connect to $\Theta(n)$ roads)
- Using Dijkstra without modification to achieve an $O(k \log k)$ -time algorithm
- Expanding vertex weights incorrectly (path length r_i instead of $r_i - 1$)
- Finding shortest paths in a BFS or DFS tree (may not contain shortest paths)

Problem 5. [18 points] **Count Cycles**

A **cycle-sparse** graph is any weighted directed simple graph $G = (V, E, w)$ for which every vertex $v \in V$ is reachable from at most one simple¹ negative-weight cycle in G . Given a cycle-sparse graph, describe an $O(|V|^3)$ -time algorithm to return the number of negative-weight cycles in G .

Solution: Construct a new graph G' by adding a supernode x to G with a zero-weight directed edge (x, v) for each $v \in V$. Then run SSSP from x in G' using Bellman-Ford to label each vertex $v \in V$ with its shortest path distance $\delta(x, v)$. For each $v \in V$, $\delta(x, v) = -\infty$ if and only if v is reachable from a negative-weight cycle in G (since adding x does not add or remove any cycles). Further, for any directed edge (u, v) , if $\delta(x, u) = \delta(x, v) = -\infty$, then both u and v are each reachable from the same simple negative-weight cycle (since v is reachable from u and each vertex is reachable from at most one simple negative-weight cycle).

So, construct a new graph G'' on only the vertices $v \in V$ where $\delta(x, v) = -\infty$ in G' , with an **undirected** edge between u and v in G'' if they share a directed edge in G . Graph G'' has the property that the number of connected components in G'' equals the number of negative-weight cycles in G , so count and return the number of connected components in G'' using Full-BFS or Full-DFS. This algorithm takes $O(|V| + |E|)$ time to construct G' , $O(|V||E|)$ time to run Bellman-Ford, $O(|V| + |E|)$ time to construct G'' , and then $O(|V| + |E|)$ time to count connected components in G'' , leading to an $O(|V||E|) = O(|V|^3)$ running time in total.

Common Mistakes:

- General lack of precision when describing algorithm
- Trying to enumerate all paths or cycles (may be exponential)
- Repeatedly running Bellman-Ford, generally yielding $|V| \cdot O(|V||E|) = O(|V|^4)$ time
- Stating that $|E| = O(|V|)$
- Confusing connected components with strongly connected components

¹Recall a cycle is simple if visits any vertex at most once.

Problem 6. [18 points] **Bellham's Fjord**

Gralexandra Bellham wants to drive her electric car in Norway from location s in Oslo to a scenic Fjord at location t . She has a map of the n locations in Norway and the $O(n)$ one-way roads directly connecting pairs of them.

- Each location x is marked with its (positive or negative) integer **height** $h(x)$ above sea-level.
- Her car has **regenerative braking** allowing it to generate energy while going downhill. Each road from location x to y is marked with the integer energy $J(x, y)$ that the electric car will either spend (positive) or generate (negative) while driving along it.
- By the laws of physics, $J(x, y)$ is always strictly greater than the difference in **potential energy** between locations x and y , i.e., $J(x, y) > m \cdot g \cdot (h(y) - h(x))$, where m and g are the mass of the car and the acceleration due to gravity respectively.

Her car battery has very large **energy capacity** $b > 2nk$ where k is the maximum $|J(x, y)|$ of any road. Assuming she departs s at half capacity, $\lfloor b/2 \rfloor$, describe an $O(n \log n)$ -time algorithm to determine the maximum amount of energy Bellham can have in her battery upon reaching t .

Partial credit will be awarded for slower correct algorithms, e.g., $O(n^2)$.

Solution: Construct graph G with a vertex for each of the n locations in Norway and a directed edge for each of the $O(n)$ roads: specifically for each road from location u to v , add directed edge (u, v) weighted by $J(u, v)$. Then $\lfloor b/2 \rfloor$ minus the weight of a minimum-weight path from s to t in G would correspond to the maximum energy Bellham could have upon reaching t ; or at least it would be if she did not either exceed or exhaust her tank along the way.

First we show that every minimum-weight path from s to t in G is simple. It suffices to show that every directed cycle in G has positive weight. Consider cycle $(c_0, \dots, c_{k-1}, c_k = c_0)$. C has weight $\sum_{i=1}^k J(c_{i-1}, c_i) > \sum_{i=1}^k mg(h(c_i) - h(c_{i-1})) = 0$, as desired.

Any simple path in G traverses at most $n - 1$ edges, so the magnitude of its weight is at most $(n - 1)k < b/2$. Thus $\lfloor b/2 \rfloor$ minus the weight of any simple path in G will always be > 0 and $< b$ (so Bellham cannot exhaust or exceed her tank by driving on a simple path from s to t).

Lastly, we find the weight of a minimum-weight path from s to t by solving SSSP. Unfortunately using Bellman-Ford takes $O(n^2)$ time which is too slow. However, we can re-weight edges in G to be positive while preserving shortest paths by exploiting the provided vertex potentials, similar to Johnson's algorithm. Specifically, create new graph G' , identical to G , except change the weight of each edge (u, v) to $J(u, v) - mg(h(v) - h(u)) > 0$. This transformation preserves shortest paths since the weight of each path from, e.g., a to b changes by the same amount, namely by $mg(h(b) - h(a))$. So run Dijkstra from s to find the minimum weight D of any path to t in G' , and return $\lfloor b/2 \rfloor - (D - mg(h(b) - h(a)))$.

Constructing G takes $O(n)$ time, reweighting to G' also takes $O(n)$ time, and then running Dijkstra from s in G' takes $O(n \log n)$ time, leading to $O(n \log n)$ time in total.

Common mistakes continued on S1.

SCRATCH PAPER 1. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write “Continued on S1” on the problem statement’s page.

Common Mistakes: (for Problem 6)

- Using Bellman-Ford directly yielding $O(n^2)$ -time algorithm (eligible for half the points)
- Running Bellman-Ford to find new weights, still leading to $O(n^2)$ -time
- Using $-J$ on weights instead of J
- b -times graph duplication (inefficient, b may be much larger than n)

SCRATCH PAPER 2. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write “Continued on S2” on the problem statement’s page.

SCRATCH PAPER 3. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write “Continued on S3” on the problem statement’s page.

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

Lecture 15: Recursive Algorithms

How to Solve an Algorithms Problem (Review)

- Reduce to a problem you already know (use data structure or algorithm)

Search Data Structures	Sort Algorithms	Graph Algorithms
Array	Insertion Sort	Breadth First Search
Linked List	Selection Sort	DAG Relaxation (DFS + Topo)
Dynamic Array	Merge Sort	Dijkstra
Sorted Array	Counting Sort	Bellman-Ford
Direct-Access Array	Radix Sort	Johnson
Hash Table	AVL Sort	
AVL Tree	Heap Sort	
Binary Heap		

- Design your own **recursive** algorithm

- Constant-sized program to solve arbitrary input
- Need looping or recursion, analyze by induction
- Recursive function call: vertex in a graph, directed edge from $A \rightarrow B$ if B calls A
- Dependency graph of recursive calls must be acyclic (if can terminate)
- Classify based on shape of graph

Class	Graph
Brute Force	Star
Decrease & Conquer	Chain
Divide & Conquer	Tree
Dynamic Programming	DAG
Greedy/Incremental	Subgraph

- Hard part is thinking inductively to construct recurrence on subproblems
- How to solve a problem recursively (**SRT BOT**)
 1. **Subproblem** definition
 2. **Relate** subproblem solutions recursively
 3. **Topological order** on subproblems (\Rightarrow subproblem DAG)
 4. **Base** cases of relation
 5. **Original** problem solution via subproblem(s)
 6. **Time** analysis

Merge Sort in SRT BOT Framework

- Merge sorting an array A of n elements can be expressed in SRT BOT as follows:
 - Subproblems: $S(i, j) = \text{sorted array on elements of } A[i : j] \text{ for } 0 \leq i \leq j \leq n$
 - Relation: $S(i, j) = \text{merge}(S(i, m), S(m, j))$ where $m = \lfloor (i + j)/2 \rfloor$
 - Topo. order: Increasing $j - i$
 - Base cases: $S(i, i + 1) = [A[i]]$
 - Original: $S(0, n)$
 - Time: $T(n) = 2T(n/2) + O(n) = O(n \lg n)$
 - In this case, subproblem DAG is a tree (divide & conquer)
-

Fibonacci Numbers

- Suppose we want to compute the n th Fibonacci number F_n
- Subproblems: $F(i) = \text{the } i\text{th Fibonacci number } F_i \text{ for } i \in \{0, 1, \dots, n\}$
- Relation: $F(i) = F(i - 1) + F(i - 2)$ (definition of Fibonacci numbers)
- Topo. order: Increasing i
- Base cases: $F(0) = 0, F(1) = 1$
- Original prob.: $F(n)$

```

1 def fib(n):
2     if n < 2: return n                      # base case
3     return fib(n - 1) + fib(n - 2)          # recurrence

```

- Divide and conquer implies a tree of **recursive calls** (draw tree)
- Time: $T(n) = T(n - 1) + T(n - 2) + O(1) > 2T(n - 2)$, $T(n) = \Omega(2^{n/2})$ exponential... :(
- Subproblem $F(k)$ computed more than once! ($F(n - k)$ times)
- Can we avoid this waste?

Re-using Subproblem Solutions

- Draw subproblem dependencies as a DAG
- To solve, either:
 - **Top down:** record subproblem solutions in a memo and re-use (**recursion + memoization**)
 - **Bottom up:** solve subproblems in topological sort order (usually via loops)
- For Fibonacci, $n + 1$ subproblems (vertices) and $< 2n$ dependencies (edges)
- Time to compute is then $O(n)$ additions

```

1 # recursive solution (top down)
2 def fib(n):
3     memo = {}
4     def F(i):
5         if i < 2: return i                      # base cases
6         if i not in memo:                      # check memo
7             memo[i] = F(i - 1) + F(i - 2)    # relation
8         return memo[i]
9     return F(n)                                # original

1 # iterative solution (bottom up)
2 def fib(n):
3     F = {}
4     F[0], F[1] = 0, 1                         # base cases
5     for i in range(2, n + 1):                  # topological order
6         F[i] = F[i - 1] + F[i - 2]            # relation
7     return F[n]                                # original

```

- A subtlety is that Fibonacci numbers grow to $\Theta(n)$ bits long, potentially \gg word size w
- Each addition costs $O(\lceil n/w \rceil)$ time
- So total cost is $O(n\lceil n/w \rceil) = O(n + n^2/w)$ time

Dynamic Programming

- Weird name coined by Richard Bellman
 - Wanted government funding, needed cool name to disguise doing mathematics!
 - Updating (dynamic) a plan or schedule (program)
 - Existence of recursive solution implies decomposable subproblems¹
 - Recursive algorithm implies a graph of computation
 - Dynamic programming if subproblem dependencies **overlap** (DAG, in-degree > 1)
 - “Recurse but re-use” (Top down: record and lookup subproblem solutions)
 - “Careful brute force” (Bottom up: do each subproblem in order)
 - Often useful for **counting/optimization** problems: almost trivially correct recurrences
-

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 substrings of a sequence
 - Often record partial state: add subproblems by incrementing some auxiliary variables
2. **Relate** subproblem solutions recursively $x(i) = f(x(j), \dots)$ 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 breaks down
5. **Original problem**
 - Show how to compute solution to original problem from solutions to subproblem(s)
 - 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

¹This property often called **optimal substructure**. It is a property of recursion, not just dynamic programming

DAG Shortest Paths

- Recall the DAG SSSP problem: given a DAG G and vertex s , compute $\delta(s, v)$ for all $v \in V$
 - Subproblems: $\delta(s, v)$ for all $v \in V$
 - Relation: $\delta(s, v) = \min\{\delta(s, u) + w(u, v) \mid u \in \text{Adj}^-(v)\} \cup \{\infty\}$
 - Topo. order: Topological order of G
 - Base cases: $\delta(s, s) = 0$
 - Original: All subproblems
 - Time: $\sum_{v \in V} O(1 + |\text{Adj}^-(v)|) = O(|V| + |E|)$
 - DAG Relaxation computes the same min values as this dynamic program, just
 - step-by-step (if new value $<$ min, update min via edge relaxation), and
 - from the perspective of u and $\text{Adj}^+(u)$ instead of v and $\text{Adj}^-(v)$
-

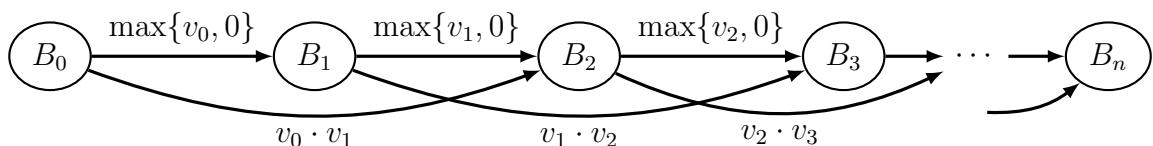
Bowling

- Given n pins labeled $0, 1, \dots, n - 1$
- Pin i has **value** v_i
- Ball of size similar to pin can hit either
 - 1 pin i , in which case we get v_i points
 - 2 adjacent pins i and $i + 1$, in which case we get $v_i \cdot v_{i+1}$ points
- Once a pin is hit, it can't be hit again (removed)
- Problem: Throw zero or more balls to maximize total points
- Example: $[-1, \boxed{1}, \boxed{1}, \boxed{1}, \boxed{9, 9}, \boxed{3}, \boxed{-3, -5}, \boxed{2, 2}]$

Bowling Algorithms

- Let's start with a more familiar divide-and-conquer algorithm:
 - Subproblems: $B(i, j) = \max\{v_m \cdot v_{m+1} + B(i, m) + B(m+2, j), B(i, m+1) + B(m+1, j)\}$ for $0 \leq i \leq j \leq n$
 - Relation:
 - * $m = \lfloor (i+j)/2 \rfloor$
 - * Either hit m and $m+1$ together, or don't
 - * $B(i, j) = \max\{v_m \cdot v_{m+1} + B(i, m) + B(m+2, j), B(i, m+1) + B(m+1, j)\}$
 - Topo. order: Increasing $j - i$
 - Base cases: $B(i, i) = 0, B(i, i+1) = \max\{v_i, 0\}$
 - Original: $B(0, n)$
 - Time: $T(n) = 4T(n/2) + O(1) = O(n^2)$
- This algorithm works but isn't very fast, and doesn't generalize well (e.g., to allow for a bigger ball that hits three balls at once)

- Dynamic programming algorithm: use suffixes
 - Subproblems: $B(i) = \max\{v_i + B(i+1), v_i \cdot v_{i+1} + B(i+2)\}$ for $0 \leq i \leq n$
 - Relation:
 - * Locally brute-force what could happen with first pin (original pin i): skip pin, hit one pin, hit two pins
 - * Reduce to smaller suffix and recurse, either $B(i+1)$ or $B(i+2)$
 - * $B(i) = \max\{B(i+1), v_i + B(i+1), v_i \cdot v_{i+1} + B(i+2)\}$
 - Topo. order: Decreasing i (for $i = n, n-1, \dots, 0$)
 - Base cases: $B(n) = B(n+1) = 0$
 - Original: $B(0)$
 - Time: (assuming memoization)
 - * $\Theta(n)$ subproblems $\cdot \Theta(1)$ work in each
 - * $\Theta(n)$ total time
- Fast and easy to generalize!
- Equivalent to maximum-weight path in Subproblem DAG:



Bowling Code

- Converting a SRT BOT specification into code is automatic/straightforward
- Here's the result for the Bowling Dynamic Program above:

```

1 # recursive solution (top down)
2 def bowl(v):
3     memo = {}
4     def B(i):
5         if i >= len(v): return 0           # base cases
6         if i not in memo:               # check memo
7             memo[i] = max(B(i+1),        # relation: skip pin i
8                               v[i] + B(i+1),    # OR bowl pin i separately
9                               v[i] * v[i+1] + B(i+2)) # OR bowl pins i and i+1 together
10        return memo[i]
11    return B(0)                      # original

1 # iterative solution (bottom up)
2 def bowl(v):
3     B = {}
4     B[len(v)] = 0                   # base cases
5     B[len(v)+1] = 0
6     for i in reversed(range(len(v))): # topological order
7         B[i] = max(B[i+1],          # relation: skip pin i
8                       v[i] + B(i+1),    # OR bowl pin i separately
9                       v[i] * v[i+1] + B(i+2)) # OR bowl pins i and i+1 together
10    return B[0]                      # original

```

How to Relate Subproblem Solutions

- The general approach we're following to define a relation on subproblem solutions:
 - Identify a question about a subproblem solution that, if you knew the answer to, would reduce to “smaller” subproblem(s)
 - * In case of bowling, the question is “how do we bowl the first couple of pins?”
 - Then locally brute-force the question by trying all possible answers, and taking the best
 - * In case of bowling, we take the max because the problem asks to maximize
 - Alternatively, we can think of correctly guessing the answer to the question, and directly recursing; but then we actually check all possible guesses, and return the “best”
- The key for efficiency is for the question to have a small (polynomial) number of possible answers, so brute forcing is not too expensive
- Often (but not always) the nonrecursive work to compute the relation is equal to the number of answers we're trying

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

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 $x(i) = f(x(j), \dots)$ 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 $D = (c_1, \dots, c_n)$, 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, \dots, 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 $i \in \{1, \dots, n + 1\}$, 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 $b(i, j) = (s - (w_i + \dots + w_j))^3$ if $s > (w_i + \dots + w_j)$, 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. L^AT_EX 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

- $x(i, j)$: sum of word lengths w_i 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

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

Lecture 16: Dyn. Prog. Subproblems

Dynamic Programming Review

- Recursion where subproblem dependencies **overlap**, forming DAG
 - “Recurse but re-use” (Top down: record and lookup subproblem solutions)
 - “Careful brute force” (Bottom up: do each subproblem in order)
-

Dynamic Programming Steps (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 substrings of a sequence
 - Often multiply possible subsets across multiple inputs
 - Often record partial state: add subproblems by incrementing some auxiliary variables
2. **Relate** subproblem solutions recursively $x(i) = f(x(j), \dots)$ for one or more $j < i$
 - Identify a question about a subproblem solution that, if you knew the answer to, reduces the subproblem to smaller subproblem(s)
 - Locally brute-force all possible answers to the question
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 breaks down
5. **Original problem**
 - Show how to compute solution to original problem from solutions to subproblem(s)
 - 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

Longest Common Subsequence (LCS)

- Given two strings A and B , find a longest (not necessarily contiguous) subsequence of A that is also a subsequence of B .
- Example: $A = \text{hieroglyphology}$, $B = \text{michaelangelo}$
- Solution: `hello` or `he glo` or `i ello` or `ie glo`, all length 5
- Maximization problem on length of subsequence

1. Subproblems

- $x(i, j) = \text{length of longest common subsequence of suffixes } A[i :] \text{ and } B[j :]$
- For $0 \leq i \leq |A|$ and $0 \leq j \leq |B|$

2. Relate

- Either first characters match or they don't
- If first characters match, some longest common subsequence will use them
- (if no LCS uses first matched pair, using it will only improve solution)
- (if an LCS uses first in $A[i]$ and not first in $B[j]$, matching $B[j]$ is also optimal)
- If they do not match, they cannot both be in a longest common subsequence
- Guess** whether $A[i]$ or $B[j]$ is not in LCS
- $$x(i, j) = \begin{cases} x(i + 1, j + 1) + 1 & \text{if } A[i] = B[j] \\ \max\{x(i + 1, j), x(i, j + 1)\} & \text{otherwise} \end{cases}$$
- (draw subset of all rectangular grid dependencies)

3. Topological order

- Subproblems $x(i, j)$ depend only on strictly larger i or j or both
- Simplest order to state: Decreasing $i + j$
- Nice order for bottom-up code: Decreasing i , then decreasing j

4. Base

- $x(i, |B|) = x(|A|, j) = 0$ (one string is empty)

5. Original problem

- Length of longest common subsequence of A and B is $x(0, 0)$
- Store parent pointers to reconstruct subsequence
- If the parent pointer increases both indices, add that character to LCS

6. Time

- # subproblems: $(|A| + 1) \cdot (|B| + 1)$
- work per subproblem: $O(1)$
- $O(|A| \cdot |B|)$ running time

```
1 def lcs(A, B):
2     a, b = len(A), len(B)
3     x = [[0] * (b + 1) for _ in range(a + 1)]
4     for i in reversed(range(a)):
5         for j in reversed(range(b)):
6             if A[i] == B[j]:
7                 x[i][j] = x[i + 1][j + 1] + 1
8             else:
9                 x[i][j] = max(x[i + 1][j], x[i][j + 1])
10    return x[0][0]
```

Longest Increasing Subsequence (LIS)

- Given a string A , find a longest (not necessarily contiguous) subsequence of A that strictly increases (lexicographically).
- Example: $A = \text{carbohydrate}$
- Solution: abort , of length 5
- Maximization problem on length of subsequence
- Attempted solution:
 - Natural subproblems are prefixes or suffixes of A , say suffix $A[i :]$
 - Natural question about LIS of $A[i :]$: is $A[i]$ in the LIS? (2 possible answers)
 - But then how do we recurse on $A[i + 1 :]$ and guarantee increasing subsequence?
 - Fix: add **constraint** to subproblems to give enough structure to achieve increasing property

1. Subproblems

- $x(i) = \text{length of longest increasing subsequence of suffix } A[i :] \text{ that includes } A[i]$
- For $0 \leq i \leq |A|$

2. Relate

- We're told that $A[i]$ is in LIS (first element)
- Next question: what is the *second* element of LIS?
 - Could be any $A[j]$ where $j > i$ and $A[j] > A[i]$ (so increasing)
 - Or $A[i]$ might be the *last* element of LIS
- $x(i) = \max\{1 + x(j) \mid i < j < |A|, A[j] > A[i]\} \cup \{1\}$

3. Topological order

- Decreasing i

4. Base

- No base case necessary, because we consider the possibility that $A[i]$ is last

5. Original problem

- What is the first element of LIS? **Guess!**
- Length of LIS of A is $\max\{x(i) \mid 0 \leq i < |A|\}$
- Store parent pointers to reconstruct subsequence

6. Time

- # subproblems: $|A|$
- work per subproblem: $O(|A|)$
- $O(|A|^2)$ running time
- Exercise: speed up to $O(|A| \log |A|)$ by doing only $O(\log |A|)$ work per subproblem, via AVL tree augmentation

```
1 def lis(A):
2     a = len(A)
3     x = [1] * a
4     for i in reversed(range(a)):
5         for j in range(i, a):
6             if A[j] > A[i]:
7                 x[i] = max(x[i], 1 + x[j])
8     return max(x)
```

Alternating Coin Game

- Given sequence of n coins of value v_0, v_1, \dots, v_{n-1}
- Two players (“me” and “you”) take turns
- In a turn, take first or last coin among remaining coins
- My goal is to maximize total value of my taken coins, where I go first
- First solution exploits that this is a **zero-sum game**: I take all coins you don’t

1. Subproblems

- Choose subproblems that correspond to the state of the game
- For every contiguous subsequence of coins from i to j , $0 \leq i \leq j < n$
- $x(i, j) = \text{maximum total value I can take starting from coins of values } v_i, \dots, v_j$

2. Relate

- I must choose either coin i or coin j (**Guess!**)
- Then it’s your turn, so you’ll get value $x(i+1, j)$ or $x(i, j-1)$, respectively
- To figure out how much value I get, subtract this from total coin values
- $x(i, j) = \max\{v_i + \sum_{k=i+1}^j v_k - x(i+1, j), v_j + \sum_{k=i}^{j-1} v_k - x(i, j-1)\}$

3. Topological order

- Increasing $j - i$

4. Base

- $x(i, i) = v_i$

5. Original problem

- $x(0, n-1)$
- Store parent pointers to reconstruct strategy

6. Time

- # subproblems: $\Theta(n^2)$
- work per subproblem: $\Theta(n)$ to compute sums
- $\Theta(n^3)$ running time
- Exercise: speed up to $\Theta(n^2)$ time by precomputing all sums $\sum_{k=i}^j v_k$ in $\Theta(n^2)$ time, via dynamic programming (!)

- Second solution uses **subproblem expansion**: add subproblems for when you move next

1. Subproblems

- Choose subproblems that correspond to the full state of the game
- Contiguous subsequence of coins from i to j , and which player p goes next
- $x(i, j, p) = \text{maximum total value I can take when player } p \in \{\text{me, you}\} \text{ starts from coins of values } v_i, \dots, v_j$

2. Relate

- Player p must choose either coin i or coin j (**Guess!**)
- If $p = \text{me}$, then I get the value; otherwise, I get nothing
- Then it's the other player's turn
- $x(i, j, \text{me}) = \max\{v_i + x(i + 1, j, \text{you}), v_j + x(i, j - 1, \text{you})\}$
- $x(i, j, \text{you}) = \min\{x(i + 1, j, \text{me}), x(i, j - 1, \text{me})\}$

3. Topological order

- Increasing $j - i$

4. Base

- $x(i, i, \text{me}) = v_i$
- $x(i, i, \text{you}) = 0$

5. Original problem

- $x(0, n - 1, \text{me})$
- Store parent pointers to reconstruct strategy

6. Time

- # subproblems: $\Theta(n^2)$
- work per subproblem: $\Theta(1)$
- $\Theta(n^2)$ running time

Subproblem Constraints and Expansion

- We've now seen two examples of constraining or expanding subproblems
- If you find yourself lacking information to check the desired conditions of the problem, or lack the natural subproblem to recurse on, try subproblem constraint/expansion!
- More subproblems and constraints give the relation more to work with, so can make DP more feasible
- Usually a trade-off between number of subproblems and branching/complexity of relation
- More examples next lecture

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

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., $\max\{x(k) \mid k \in \{0, \dots, n-1\}\}$
- 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

```

1 # bottom up implementation
2 def max_subarray_sum(A):
3     x = [None for _ in A]           # memo
4     x[0] = A[0]                   # base case
5     for k in range(1, len(A)):    # iteration
6         x[k] = max(A[k], A[k] + x[k - 1]) # relation
7     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 $B(j)$
- $x(i, j) = \begin{cases} x(i - 1, j - 1) & \text{if } A(i) = B(j) \\ 1 + \min(x(i - 1, j), x(i, j - 1), x(i - 1, j - 1)) & \text{otherwise} \end{cases}$

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

```

1 def edit_distance(A, B):
2     x = [[None] * len(A) for _ in range(len(B))] # memo
3     x[0][0] = 0 # base cases
4     for i in range(1, len(A)): # delete A[i]
5         x[i][0] = x[i - 1][0] + 1
6     for j in range(1, len(B)): # insert B[j] into A
7         x[0][j] = x[0][j - 1] + 1
8     for i in range(1, len(A)): # dynamic program
9         for j in range(1, len(B)):
10            if A[i] == B[j]: # matched! no edit needed
11                x[i][j] = x[i - 1][j - 1]
12            else: # edit needed!
13                ed_del = 1 + x[i - 1][j] # delete A[i]
14                ed_ins = 1 + x[i][j - 1] # insert B[j] after A[i]
15                ed_rep = 1 + x[i - 1][j - 1] # replace A[i] with B[j]
16                x[i][j] = min(ed_del, ed_ins, ed_rep)
17
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.)

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

Problem Session 8

Problem 8-1. Sunny Studies

Tim the Beaver needs to study for exams, but it's getting warmer, and Tim wants to spend more time outside. Tim enjoys being outside more when the weather is warmer: specifically, if the temperature outside is t integer units above zero, Tim's happiness will increase by t after spending the day outside (with a decrease in happiness when t is negative). On each of the n days until finals, Tim will either study or play outside (never both on the same day). In order to stay on top of coursework, Tim resolves never to play outside more than two days in a row. Given a weather forecast estimating temperature for the next n days, describe an $O(n)$ -time dynamic programming algorithm to determine which days Tim should study in order to increase happiness the most.

Solution:

1. Subproblems

- Let $t(i)$ be the temperature on day i
- $x(i)$: the maximum happiness increase possible during days i to n

2. Relate

- Guess whether studying or playing on day i will yield more happiness
- If study, no change in happiness, but may study or play on the next day, $x(i + 1)$
- If play, change happiness by $t(i)$ and either
 - study on day $i + 1$, $t(i) + x(i + 2)$, or
 - play on day $i + 1$ and study on $i + 2$, $t(i) + t(i + 1) + x(i + 3)$
- $x(i) = \max\{x(i + 1), t(i) + x(i + 2), t(i) + t(i + 1) + x(i + 3)\}$ for $i \in \{1, \dots, n - 1\}$

3. Topo. Order

- Subproblem $x(i)$ only depends on subproblems with strictly larger i , so acyclic

4. Base

- $x(n) = \max\{0, t(n)\}$, if only one day, either play or study
- $x(n + 1) = x(n + 2) = 0$, if no more days, can't increase happiness

5. Original

- $x(1)$ is the maximum happiness increase possible from day 1 to day n
- Store parent pointers to reconstruct days studied

6. Time

- # subproblems: $x(i)$ for $i \in \{1, \dots, n + 2\}$, $n + 2 = O(n)$
- work per subproblem: $O(1)$ (constant branching factor)
- $O(n)$ running time

Problem 8-2. Difffing Data

Operating system Menix has a `diff` utility that can compare files. A **file** is an ordered sequence of strings, where the i^{th} string is called the i^{th} **line** of the file. A single **change** to a file is either:

- the insertion of a single new line into the file;
- the removal of a single line from the file; or
- swapping two adjacent lines in the file.

In Menix, swapping two lines is cheap, as they are already in the file, but inserting or deleting a line is expensive. A **diff** from a file A to a file B is any sequence of changes that, when applied in sequence to A will transform it into B , under the conditions that any line may be swapped at most once and any pair of swapped lines appear adjacent in A and adjacent in B . Given two files A and B , each containing exactly n lines, describe an $O(kn + n^2)$ -time algorithm to return a diff from A to B that minimizes the number of changes that are **not swaps**, assuming that any line from either file is at most k ASCII characters long.

Solution:

1. Subproblems

- First, use a hash table to assign each unique line a number in $O(kn)$ time
- Now each line can be compared to others in $O(1)$ time
- $x(i, j)$: the minimum non-swap changes to transform $A[:i]$ into $B[:j]$

2. Relate

- If $A[i] = B[j]$, then recurse on remainder
- Otherwise maximize a last change applied:
 - $A[i]$ is deleted
 - an insertion matches with $B[j]$
 - the last two in $A[:i]$ are swapped to match the last two in $B[:j]$
- if $A[i] = A[j]$, $x(i, j) = x(i - 1, j - 1)$
- otherwise, $x(i, j) = \min \left\{ \begin{array}{ll} 1 + x(i - 1, j) & \text{delete} \\ 1 + x(i, j - 1) & \text{insert} \\ x(i - 2, j - 2) & \text{swap if } A[i] = B[j - 1] \text{ and } A[i - 1] = B[j] \end{array} \right\}$

3. Topo. Order

- Subproblem $x(i, j)$ only depends on subproblems with strictly smaller $i + j$, so acyclic

4. Base

- $x(0, 0) = 0$, all lines converted
- $x(i, 0) = i$, must delete remainder
- $x(0, j) = j$, must insert remainder

5. Original

- $x(n, n, 0)$ by definition
- Store parent pointers to reconstruct which changes were made (for cases where $A[i] \neq A[j]$, remember whether a deletion, insertion, or swap occurred)

6. Time

- pre-processing: $O(kn)$
- # subproblems: $(n + 1)^2 = O(n^2)$, $x(i, j)$ for $i, j \in \{0, 1, \dots, n\}$
- work per subproblem: $O(1)$

- $O(n^2)$ running time

Problem 8-3. Building Blocks

Saggie Mimpson is a toddler who likes to build block towers. Each of her blocks is a 3D rectangular prism, where each block b_i has a positive integer width w_i , height h_i , and length ℓ_i , and she has at least three of each type of block. Each block may be **oriented** so that any opposite pair of its rectangular faces may serve as its **top** and **bottom** faces, and the **height** of the block in that orientation is the distance between those faces. Saggie wants to construct a tower by stacking her blocks as high as possible, but she can only stack an oriented block b_i on top of another oriented block b_j if the dimensions of the bottom of block b_i are strictly smaller¹ than the dimensions of the top of block b_j . Given the dimensions of each of her n blocks, describe an $O(n^2)$ -time algorithm to determine the height of the tallest tower Saggie can build from her blocks.

Solution:

1. Subproblems

- Each block may be used in one of three vertical orientations
- (without loss of generality, block bases can always be stacked with the base's shorter side pointed in one direction)
- Because the stacking requirement requires base dimensions to strictly decrease, any optimal tower may use any block type at most three times (once in each orientation)
- Sort dimensions of each block and remove duplicates (e.g., in $O(n)$ time with hash table)
- For each block type with sorted dimensions $a \leq b \leq c$, construct three block orientations (a, b, c) , (a, c, b) , (b, c, a) (last dimension corresponding to height), and add them to an oriented block list in $O(n)$ time
- (triplicating block types allowable since there are at least three of each type)
- Sort lexicographically (first dimension most significant) in $O(n \log n)$ time
- Re-number sorted list, where block i has oriented dimensions (w_i, ℓ_i, h_i) with $w_i \leq \ell_i$
- Any stackable tower of oriented blocks must be a subsequence of this sorted list since the w_i are sorted, though not every subsequence of this list comprises a valid tower, since the ℓ_i are not necessarily sorted
- $x(i)$: the maximum tower height of any tower that uses block i and any subset of the remaining blocks 1 through $i - 1$

2. Relate

- Guess the next lower block in the tower, only from blocks with strictly smaller ℓ_i
- $x(i) = h_i + \max\{0\} \cup \{x(j) \mid j \in \{1, \dots, i - 1\} \text{ s.t. } \ell_i < \ell_j\}$

3. Topo. Order

- Subproblem $x(i)$ only depends on subproblems with strictly smaller i , so acyclic

4. Base

- $x(1) = h_1$, since the maximum in the recurrence is trivially zero

5. Original

¹If the bottom of block b_i has dimensions $p \times q$ and the top of block b_j has dimensions $s \times t$, then b_i can be stacked on b_j in this orientation if either: $p < s$ and $q < t$; or $p < t$ and $q < s$.

- Some block must be the top block, so compare all possible top blocks
- $\max\{x(i) \mid i \in \{1, \dots, n\}\}$

6. Time

- pre-processing: $O(n \log n)$
- # subproblems: $n, x(i)$ for $i \in \{1, \dots, n\}$
- work per subproblem: $O(n)$
- computing the solution: $O(n)$
- $O(n^2)$ running time
- Note that this problem can be solved in $O(n \log n)$ with a similar optimization as in Longest Increasing Subsequence from Recitation 15.

Problem 8-4. Princess Plum

Princess Plum is a video game character collecting mushrooms in a digital haunted forest. The forest is an $n \times n$ square grid where each grid square contains either a tree, mushroom, or is empty. Princess Plum can move from one grid square to another if the two squares share an edge, but she cannot enter a grid square containing a tree. Princess Plum starts in the upper left grid square and wants to reach her home in the bottom right grid square². The haunted forest is scary, so she wants to reach home via a **quick path**: a route from start to home that goes through at most $2n - 1$ grid squares (including start and home). If Princess Plum enters a square with a mushroom, she will pick it up. Let k be the maximum number of mushrooms she could pick up along any quick path, and let a quick path be **optimal** if she could pick up k mushrooms along that path.

- (a) [15 points] Given a map of the forest grid, describe an $O(n^2)$ -time algorithm to return the number of distinct optimal quick paths through the forest, assuming that some quick path exists.

Solution:

1. Subproblems

- Let upper left grid square be $(1, 1)$ and bottom right grid square be (n, n)
- Let $F[i][j]$ denote the contents of grid square (i, j)
- Define two types of subproblems: one for optimal mushrooms and one to count number of paths
 - $k(i, j)$: the maximum number of mushrooms to reach grid square (i, j) from grid square $(1, 1)$ on a path touching $i + j - 1$ squares
 - $x(i, j)$: the number of paths from $(1, 1)$ to (i, j) collecting $k(i, j)$ mushrooms and touching $i + j - 1$ squares

2. Relate

- A path touching $i + j - 1$ squares ending at (i, j) must extend a path touching $i + j - 2$ squares to either its left or upper neighbor
- If $F[i][j]$ contains a tree:
 - There are no paths to (i, j)
 - $k(i, j) = -\infty$ and $x(i, j) = 0$

²Assume that both the start and home grid squares are empty.

- Otherwise:

- Let $m(i, j)$ be 1 if $F[i][j]$ is a mushroom and 0 otherwise

- $k(i, j) = m(i, j) + \max(k(i - 1, j), k(i, j - 1))$

$$x(i, j) = \sum \begin{cases} 0 & \text{always} \\ x(i - 1, j) & \text{if } k(i - 1, j) + m(i, j) = k(i, j) \\ x(i, j - 1) & \text{if } k(i, j - 1) + m(i, j) = k(i, j) \end{cases}$$

3. Topo. Order

- Subproblem $k(i, j)$ only depends on k subproblems with strictly smaller $i + j$, so acyclic
- Subproblem $x(i, j)$ only depends on x subproblems with strictly smaller $i + j$ (and k subproblems which do not depend on x subproblems), so acyclic

4. Base

- $k(1, 1) = 0$, no mushrooms to start
- $x(1, 1) = 1$, one path at start
- negative grid squares impossible
- $k(0, i) = k(i, 0) = -\infty$ for $i \in \{0, \dots, n\}$
- $x(0, i) = x(i, 0) = 0$ for $i \in \{0, \dots, n\}$

5. Original

- $x(n, n)$ by definition

6. Time

- # subproblems: $2(n + 1)^2 = O(n^2)$, $k(i, j)$ and $x(i, j)$ for $i, j \in \{0, 1, \dots, n\}$
- work per subproblem: $O(1)$
- $O(n^2)$ running time

- (b) [25 points] Write a Python function `count_paths(F)` that implements your algorithm from (a).

Solution:

```

1 def count_paths(F):
2     n = len(F)
3     K = [[-float('inf')]] * (n + 1) for _ in range(n + 1)]      # init K memo
4     X = [[0]] * (n + 1) for _ in range(n + 1)]                # init X memo
5     for i in range(1, n + 1):                                # bottom-up dynamic program
6         for j in range(1, n + 1):
7             if F[i - 1][j - 1] == 't':                      # base case
8                 continue
9             if i == 1 and j == 1:                            # base case
10                K[1][1], X[1][1] = 0, 1
11                continue
12             if F[i - 1][j - 1] == 'm':                      m = 1
13             else:                                         m = 0
14             K[i][j] = m + max(K[i - 1][j], K[i][j - 1])
15             if K[i - 1][j] + m == K[i][j]: X[i][j] += X[i - 1][j]
16             if K[i][j - 1] + m == K[i][j]: X[i][j] += X[i][j - 1]
17     return X[n][n]

```

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

Lecture 17: Dyn. Prog. III

Dynamic Programming Steps (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 substrings of a sequence
 - Often multiply possible subsets across multiple inputs
 - Often record partial state: add subproblems by incrementing some auxiliary variables
 2. **Relate** subproblem solutions recursively $x(i) = f(x(j), \dots)$ for one or more $j < i$
 - Identify a question about a subproblem solution that, if you knew the answer to, reduces the subproblem to smaller subproblem(s)
 - Locally brute-force all possible answers to the question
 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 breaks down
 5. **Original problem**
 - Show how to compute solution to original problem from solutions to subproblem(s)
 - 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
-

Recall: DAG Shortest Paths [L15]

- Subproblems: $\delta(s, v)$ for all $v \in V$
- Relation: $\delta(s, v) = \min\{\delta(s, u) + w(u, v) \mid u \in \text{Adj}^-(v)\} \cup \{\infty\}$
- Topo. order: Topological order of G

Single-Source Shortest Paths Revisited

1. Subproblems

- Expand subproblems to add information to make acyclic!
(an example we've already seen of subproblem expansion)
- $\delta_k(s, v) = \text{weight of shortest path from } s \text{ to } v \text{ using at most } k \text{ edges}$
- For $v \in V$ and $0 \leq k \leq |V|$

2. Relate

- Guess last edge (u, v) on shortest path from s to v
- $\delta_k(s, v) = \min\{\delta_{k-1}(s, u) + w(u, v) \mid (u, v) \in E\} \cup \{\delta_{k-1}(s, v)\}$

3. Topological order

- Increasing k : subproblems depend on subproblems only with strictly smaller k

4. Base

- $\delta_0(s, s) = 0$ and $\delta_0(s, v) = \infty$ for $v \neq s$ (no edges)
- (draw subproblem graph)

5. Original problem

- If has finite shortest path, then $\delta(s, v) = \delta_{|V|-1}(s, v)$
- Otherwise some $\delta_{|V|}(s, v) < \delta_{|V|-1}(s, v)$, so path contains a negative-weight cycle
- Can keep track of parent pointers to subproblem that minimized recurrence

6. Time

- # subproblems: $|V| \times (|V| + 1)$
- Work for subproblem $\delta_k(s, v)$: $O(\deg_{\text{in}}(v))$

$$\sum_{k=0}^{|V|} \sum_{v \in V} O(\deg_{\text{in}}(v)) = \sum_{k=0}^{|V|} O(|E|) = O(|V| \cdot |E|)$$

This is just **Bellman-Ford!** (computed in a slightly different order)

All-Pairs Shortest Paths: Floyd–Warshall

- Could define subproblems $\delta_k(u, v) = \text{minimum weight of path from } u \text{ to } v \text{ using at most } k \text{ edges}$, as in Bellman–Ford
- Resulting running time is $|V|$ times Bellman–Ford, i.e., $O(|V|^2 \cdot |E|) = O(|V|^4)$
- Know a better algorithm from L14: Johnson achieves $O(|V|^2 \log |V| + |V| \cdot |E|) = O(|V|^3)$
- Can achieve $\Theta(|V|^3)$ running time (matching Johnson for dense graphs) with a simple dynamic program, called **Floyd–Warshall**
- Number vertices so that $V = \{1, 2, \dots, |V|\}$

1. Subproblems

- $d(u, v, k) = \text{minimum weight of a path from } u \text{ to } v \text{ that only uses vertices from } \{1, 2, \dots, k\} \cup \{u, v\}$
- For $u, v \in V$ and $1 \leq k \leq |V|$

2. Relate

- $x(u, v, k) = \min\{x(u, k, k-1) + x(k, v, k-1), x(u, v, k-1)\}$
- Only constant branching! No longer guessing previous vertex/edge

3. Topological order

- Increasing k : relation depends only on smaller k

4. Base

- $x(u, u, 0) = 0$
- $x(u, v, 0) = w(u, v)$ if $(u, v) \in E$
- $x(u, v, 0) = \infty$ if none of the above

5. Original problem

- $x(u, v, |V|)$ for all $u, v \in V$

6. Time

- $O(|V|^3)$ subproblems
- Each $O(1)$ work
- $O(|V|^3)$ in total
- Constant number of dependencies per subproblem brings the factor of $O(|E|)$ in the running time down to $O(|V|)$.

Arithmetic Parenthesization

- Input: arithmetic expression $a_0 *_1 a_1 *_2 a_2 \cdots *_{n-1} a_{n-1}$
where each a_i is an integer and each $*_i \in \{+, \times\}$
- Output: Where to place parentheses to maximize the evaluated expression
- Example: $7 + 4 \times 3 + 5 \rightarrow ((7) + (4)) \times ((3) + (5)) = 88$
- Allow **negative** integers!
- Example: $7 + (-4) \times 3 + (-5) \rightarrow ((7) + ((-4) \times ((3) + (-5)))) = 15$

1. Subproblems

- Sufficient to maximize each subarray? No! $(-3) \times (-3) = 9 > (-2) \times (-2) = 4$
- $x(i, j, \text{opt}) = \text{opt value obtainable by parenthesizing } a_i *_i+1 \cdots *_j-1 a_j-1$
- For $0 \leq i < j \leq n$ and $\text{opt} \in \{\min, \max\}$

2. Relate

- Guess location of outermost parentheses / last operation evaluated
- $x(i, j, \text{opt}) = \text{opt} \{x(i, k, \text{opt}') *_k x(k, j, \text{opt}'') \mid i < k < j; \text{opt}', \text{opt}'' \in \{\min, \max\}\}$

3. Topological order

- Increasing $j - i$: subproblem $x(i, j, \text{opt})$ depends only on strictly smaller $j - i$

4. Base

- $x(i, i + 1, \text{opt}) = a_i$, only one number, no operations left!

5. Original problem

- $X(0, n, \max)$
- Store parent pointers (two!) to find parenthesization (forms binary tree!)

6. Time

- # subproblems: less than $n \cdot n \cdot 2 = O(n^2)$
- work per subproblem $O(n) \cdot 2 \cdot 2 = O(n)$
- $O(n^3)$ running time

Piano Fingering

- Given sequence t_0, t_1, \dots, t_{n-1} of n **single** notes to play with right hand (will generalize to multiple notes and hands later)
- Performer has right-hand fingers $1, 2, \dots, F$ ($F = 5$ for most humans)
- Given metric $d(t, f, t', f')$ of **difficulty** of transitioning from note t with finger f to note t' with finger f'
 - Typically a sum of penalties for various difficulties, e.g.:
 - $1 < f < f'$ and $t > t'$ is uncomfortable
 - Legato (smooth) play requires $t \neq t'$ (else infinite penalty)
 - Weak-finger rule: prefer to avoid $f' \in \{4, 5\}$
 - $\{f, f'\} = \{3, 4\}$ is annoying
- Goal: Assign fingers to notes to minimize total difficulty
- First attempt:

1. Subproblems

- $x(i) = \text{minimum total difficulty for playing notes } t_i, t_{i+1}, \dots, t_{n-1}$

2. Relate

- Guess first finger: assignment f for t_i
- $x(i) = \min\{x(i+1) + d(t_i, f, t_{i+1}, ?) \mid 1 \leq f \leq F\}$
- Not enough information to fill in $?$
- Need to know which finger at the start of $x(i+1)$
- But different starting fingers could hurt/help both $x(i+1)$ and $d(t_i, f, t_{i+1}, ?)$
- Need a table mapping start fingers to optimal solutions for $x(i+1)$
- I.e., need to expand subproblems with start condition

- Solution:

1. Subproblems

- $x(i, f) = \text{minimum total difficulty for playing notes } t_i, t_{i+1}, \dots, t_{n-1} \text{ starting with finger } f \text{ on note } t_i$
- For $0 \leq i < n$ and $1 \leq f \leq F$

2. Relate

- Guess next finger: assignment f' for t_{i+1}
- $x(i, f) = \min\{x(i + 1, f') + d(t_i, f, t_{i+1}, f') \mid 1 \leq f' \leq F\}$

3. Topological order

- Decreasing i (any f order)

4. Base

- $x(n - 1, f) = 0$ (no transitions)

5. Original problem

- $\min\{x(0, f) \mid 1 \leq f \leq F\}$

6. Time

- $\Theta(n \cdot F)$ subproblems
- $\Theta(F)$ work per subproblem
- $\Theta(n \cdot F^2)$
- No dependence on the number of different notes!

Guitar Fingering

- Up to $S = \text{number of strings}$ different ways to play the same note
- Redefine “finger” to be tuple (finger playing note, string playing note)
- Throughout algorithm, F gets replaced by $F \cdot S$
- Running time is thus $\Theta(n \cdot F^2 \cdot S^2)$

Multiple Notes at Once

- Now suppose t_i is a set of notes to play at time i
- Given a bigger transition difficulty function $d(t, f, t', f')$
- Goal: fingering $f_i : t_i \rightarrow \{1, 2, \dots, F\}$ specifying how to finger each note (including which string for guitar) to minimize $\sum_{i=1}^{n-1} d(t_{i-1}, f_{i-1}, t_i, f_i)$
- At most T^F choices for each fingering f_i , where $T = \max_i |t_i|$
 - $T \leq F = 10$ for normal piano (but there are exceptions)
 - $T \leq S$ for guitar
- $\Theta(n \cdot T^F)$ subproblems
- $\Theta(T^F)$ work per subproblem
- $\Theta(n \cdot T^{2F})$ time
- $\Theta(n)$ time for $T, F \leq 10$

Video Game Applications

- Guitar Hero / Rock Band
 - $F = 4$ (and 5 different notes)
- Dance Dance Revolution
 - $F = 2$ feet
 - $T = 2$ (at most two notes at once)
 - Exercise: handle sustained notes, using “where each foot is” (on an arrow or in the middle) as added state for suffix subproblems

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

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 $x \in \{1, \dots, n\}$
- 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:

1	(0) #####	(+1) #####	(-1) ####	(+2) #####	(-2) ###	row 2
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 $j \in \{0, +1, -1\}$
- $x(i, j)$: maximum score, only placing ships on board subset $s(i, j)$
- for $i \in \{0, \dots, n\}$, $j \in \{0, +1, -1\}$

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
- $$\bullet \quad x(i, j) = \begin{cases} \max\{v(i, 1) + v(i - 1, 1) + x(i - 2, +1), x(i - 1, 0)\} & \text{if } j = -1 \\ \max\{v(i + 1, 2) + v(i, 2) + x(i, -1), x(i, 0)\} & \text{if } j = +1 \\ \max\{v(i, 1) + v(i, 2) + x(i - 1, 0), x(i, -1), x(i - 1, +1)\} & \text{if } j = 0 \end{cases}$$

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

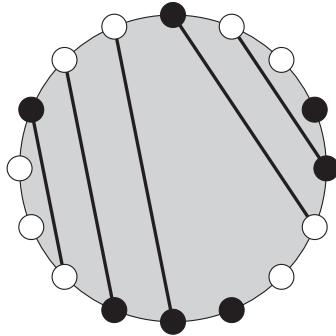
- Solution is $x(n, 0)$, the maximum considering all grid squares.
- Store parent pointers to reconstruct ship locations

6. Time

- # subproblems: $\mathcal{O}(n)$
- work per subproblem $\mathcal{O}(1)$
- $\mathcal{O}(n)$ 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, \dots, 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 $k \in \{i, \dots, j\}$
- for $i \in \{1, \dots, n\}, j \in \{i - 1, \dots, n\}$
- $j - i + 1$ 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 $t \in \{3, \dots, n-1\}$
 - otherwise, just can't match i with $i+1$, so try $t \in \{i+2, \dots, j\}$
- Let $m(i, j) = 1$ if $a_i \neq a_j$ and $m(i, j) = 0$ otherwise (ports of opposite type match)
- $x(1, n) = \max\{x(2, n)\} \cup \{m(1, t) + x(2, t-1) + x(t+1, n) \mid t \in \{3, \dots, n-1\}\}$
- $x(i, j) = \max\{x(i+1, j)\} \cup \{m(i, t) + x(i+1, t-1) + x(t+1, j) \mid t \in \{i+2, \dots, j\}\}$

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

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

Lecture 18: Pseudopolynomial

Dynamic Programming Steps (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 substrings of a sequence
- Often multiply possible subsets across multiple inputs
- Often record partial state: add subproblems by incrementing some auxiliary variables
- Often smaller integers than a given integer (**today's focus**)

2. Relate subproblem solutions recursively $x(i) = f(x(j), \dots)$ for one or more $j < i$

- Identify a question about a subproblem solution that, if you knew the answer to, reduces the subproblem to smaller subproblem(s)
- Locally brute-force all possible answers to the question

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 breaks down

5. Original problem

- Show how to compute solution to original problem from solutions to subproblem(s)
- 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

Rod Cutting

- Given a rod of length L and value $v(\ell)$ of rod of length ℓ for all $\ell \in \{1, 2, \dots, L\}$
- Goal: Cut the rod to maximize the value of cut rod pieces
- Example: $L = 7$, $v_{\ell} = [0, 1, 10, 13, 18, 20, 31, 32]$
- Maybe greedily take most valuable per unit length?
- Nope! $\arg \max_{\ell} v[\ell]/\ell = 6$, and partitioning $[6, 1]$ yields 32 which is not optimal!
- Solution: $v[2] + v[2] + v[3] = 10 + 10 + 13 = 33$
- Maximization problem on value of partition

1. Subproblems

- $x(\ell)$: maximum value obtainable by cutting rod of length ℓ
- For $\ell \in \{0, 1, \dots, L\}$

2. Relate

- First piece has some length p (**Guess!**)
- $x(\ell) = \max\{v(p) + x(\ell - p) \mid p \in \{1, \dots, \ell\}\}$
- (draw dependency graph)

3. Topological order

- Increasing ℓ : Subproblems $x(\ell)$ depend only on strictly smaller ℓ , so acyclic

4. Base

- $x(0) = 0$ (length-zero rod has no value!)

5. Original problem

- Maximum value obtainable by cutting rod of length L is $x(L)$
- Store choices to reconstruct cuts
- If current rod length ℓ and optimal choice is ℓ' , remainder is piece $p = \ell - \ell'$
- (maximum-weight path in subproblem DAG!)

6. Time

- # subproblems: $L + 1$
- work per subproblem: $O(\ell) = O(L)$
- $O(L^2)$ running time

Is This Polynomial Time?

- **(Strongly) polynomial time** means that the running time is bounded above by a constant-degree polynomial in the **input size** measured in words
- In Rod Cutting, input size is $L + 1$ words (one integer L and L integers in v)
- $O(L^2)$ is a constant-degree polynomial in $L + 1$, so YES: (strongly) polynomial time

```

1 # recursive
2 x = []
3 def cut_rod(l, v):
4     if l < 1:    return 0                      # base case
5     if l not in x:                           # check memo
6         for piece in range(1, l + 1):          # try piece
7             x_ = v[piece] + cut_rod(l - piece, v) # recurrence
8             if (l not in x) or (x[l] < x_):      # update memo
9                 x[l] = x_
10    return x[l]

1 # iterative
2 def cut_rod(L, v):
3     x = [0] * (L + 1)                         # base case
4     for l in range(L + 1):                     # topological order
5         for piece in range(1, l + 1):           # try piece
6             x_ = v[piece] + x[l - piece]        # recurrence
7             if x[l] < x_:                      # update memo
8                 x[l] = x_
9
10    return x[L]

1 # iterative with parent pointers
2 def cut_rod_pieces(L, v):
3     x = [0] * (L + 1)                         # base case
4     parent = [None] * (L + 1)                  # parent pointers
5     for l in range(1, L + 1):                 # topological order
6         for piece in range(1, l + 1):           # try piece
7             x_ = v[piece] + x[l - piece]        # recurrence
8             if x[l] < x_:                      # update memo
9                 x[l] = x_
10                parent[l] = l - piece           # update parent
11
12    l, pieces = L, []
13    while parent[l] is not None:               # walk back through parents
14        piece = l - parent[l]
15        pieces.append(piece)
16        l = parent[l]
17
18    return pieces

```

Subset Sum

- Input: Sequence of n positive integers $A = \{a_0, a_1, \dots, a_{n-1}\}$
- Output: Is there a subset of A that sums exactly to T ? (i.e., $\exists A' \subseteq A$ s.t. $\sum_{a \in A'} a = T$?)
- Example: $A = (1, 3, 4, 12, 19, 21, 22)$, $T = 47$ allows $A' = \{3, 4, 19, 21\}$
- Optimization problem? Decision problem! Answer is YES or NO, TRUE or FALSE
- In example, answer is YES. However, answer is NO for some T , e.g., 2, 6, 9, 10, 11, ...

1. Subproblems

- $x(i, t) = \boxed{\text{does any subset of } A[i :] \text{ sum to } t?}$
- For $i \in \{0, 1, \dots, n\}$, $t \in \{0, 1, \dots, T\}$

2. Relate

- Idea: Is first item a_i in a valid subset A' ? (Guess!)
- If yes, then try to sum to $t - a_i \geq 0$ using remaining items
- If no, then try to sum to t using remaining items
- $x(i, t) = \text{OR} \begin{cases} x(i+1, t - A[i]) & \text{if } t \geq A[i] \\ x(i+1, t) & \text{always} \end{cases}$

3. Topological order

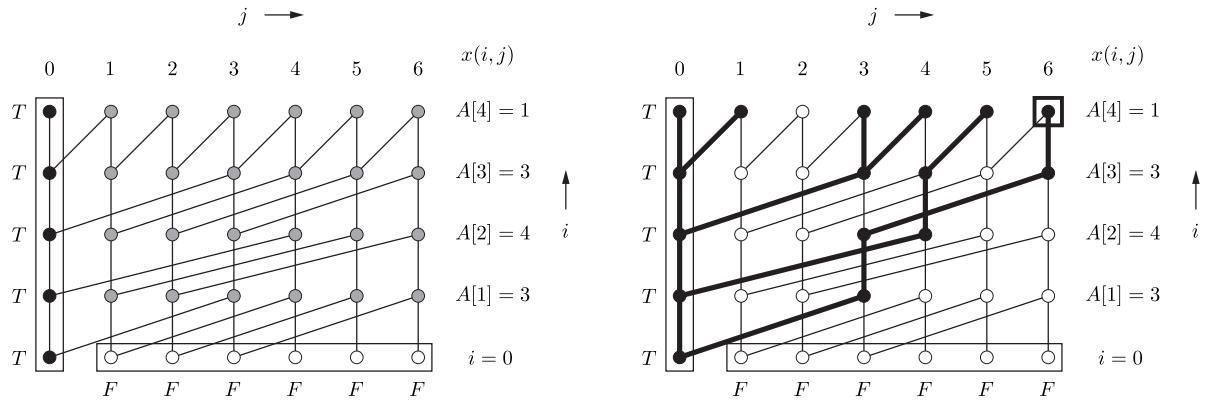
- Subproblems $x(i, t)$ only depend on strictly larger i , so acyclic
- Solve in order of decreasing i

4. Base

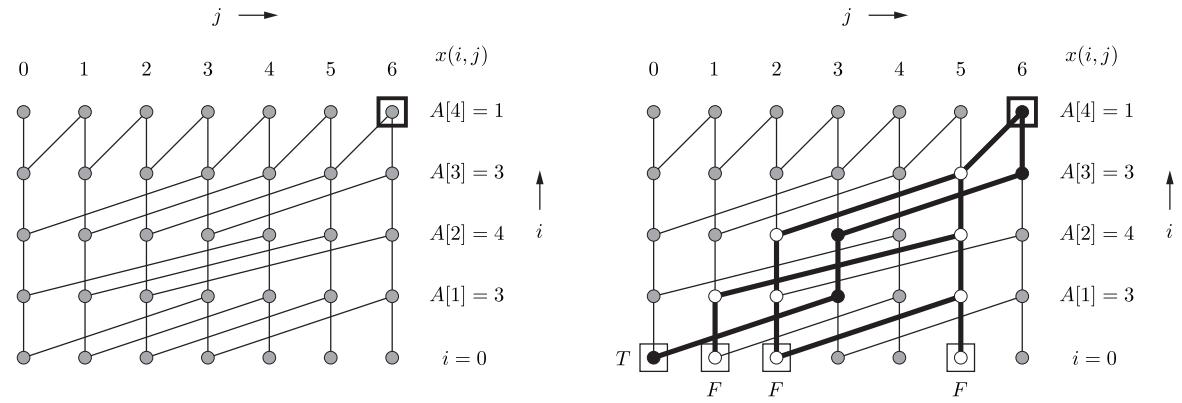
- $x(i, 0) = \text{YES}$ for $i \in \{0, \dots, n\}$ (space packed exactly!)
- $x(n, t) = \text{NO}$ for $j \in \{1, \dots, T\}$ (no more items available to pack)

5. Original problem

- Original problem given by $x(0, T)$
- Example: $A = (3, 4, 3, 1)$, $T = 6$ solution: $A' = (3, 3)$
- Bottom up: Solve all subproblems (Example has 35)



- Top down: Solve only **reachable** subproblems (Example, only 14!)



6. Time

- # subproblems: $O(nT)$, $O(1)$ work per subproblem, $O(nT)$ time

Is This Polynomial?

- Input size is $n + 1$: one integer T and n integers in A
- Is $O(nT)$ bounded above by a polynomial in $n + 1$? NO, not necessarily
- On w -bit word RAM, $T \leq 2^w$ and $w \geq \lg(n + 1)$, but we don't have an upper bound on w
- E.g., $w = n$ is not unreasonable, but then running time is $O(n2^n)$, which is **exponential**

Pseudopolynomial

- Algorithm has **pseudopolynomial time**: running time is bounded above by a constant-degree polynomial in input size and input integers
- Such algorithms are polynomial in the case that integers are polynomially bounded in input size, i.e., $n^{O(1)}$ (same case that Radix Sort runs in $O(n)$ time)
- Counting sort $O(n + u)$, radix sort $O(n \log_n u)$, direct-access array build $O(n + u)$, and Fibonacci $O(n)$ are all pseudopolynomial algorithms we've seen already
- Radix sort is actually **weakly polynomial** (a notion in between strongly polynomial and pseudopolynomial): bounded above by a constant-degree polynomial in the input size measured in bits, i.e., in the logarithm of the input integers
- Contrast with Rod Cutting, which was polynomial
 - Had pseudopolynomial dependence on L
 - But luckily had $\geq L$ input integers too
 - If only given subset of sellable rod lengths (Knapsack Problem, which generalizes Rod Cutting and Subset Sum — see recitation), then algorithm would have been only pseudopolynomial

Complexity

- Is Subset Sum solvable in polynomial time when integers are not polynomially bounded?
- No if $P \neq NP$. What does that mean? Next lecture!

Main Features of Dynamic Programs

- Review of examples from lecture
- **Subproblems:**
 - **Prefix/suffixes:** Bowling, LCS, LIS, Floyd–Warshall, Rod Cutting (coincidentally, really Integer subproblems), Subset Sum
 - **Substrings:** Alternating Coin Game, Arithmetic Parenthesization
 - **Multiple sequences:** LCS
 - **Integers:** Fibonacci, Rod Cutting, Subset Sum
 - * **Pseudopolynomial:** Fibonacci, Subset Sum
 - **Vertices:** DAG shortest paths, Bellman–Ford, Floyd–Warshall
- **Subproblem constraints/expansion:**
 - **Nonexpansive constraint:** LIS (include first item)
 - **$2 \times$ expansion:** Alternating Coin Game (who goes first?), Arithmetic Parenthesization (min/max)
 - **$\Theta(1) \times$ expansion:** Piano Fingering (first finger assignment)
 - **$\Theta(n) \times$ expansion:** Bellman–Ford (# edges)
- **Relation:**
 - **Branching** = # dependant subproblems in each subproblem
 - **$\Theta(1)$ branching:** Fibonacci, Bowling, LCS, Alternating Coin Game, Floyd–Warshall, Subset Sum
 - **$\Theta(\text{degree})$ branching** (source of $|E|$ in running time): DAG shortest paths, Bellman–Ford
 - **$\Theta(n)$ branching:** LIS, Arithmetic Parenthesization, Rod Cutting
 - **Combine multiple solutions (not path in subproblem DAG):** Fibonacci, Floyd–Warshall, Arithmetic Parenthesization
- **Original problem:**
 - **Combine multiple subproblems:** DAG shortest paths, Bellman–Ford, Floyd–Warshall, LIS, Piano Fingering

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

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 $i \in \{0, \dots, n\}, j \in \{0, \dots, S\}$

3. Topo

- Subproblems $x(i, j)$ only depend on strictly smaller i , so acyclic

4. Base

- $x(i, 0) = \text{True}$ for $i \in \{0, \dots, n\}$ (trivial to make zero sum!)
- $x(0, j) = \text{False}$ for $j \in \{1, \dots, S\}$ (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 $S' \in \{0, \dots, \frac{1}{2} \sum_{a \in A} a\}$, 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 $i \in \{0, \dots, n\}, j \in \{0, \dots, S\}$ 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).

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

Subproblem dependencies are still acyclic because $x(i, j)$ only depend on strictly smaller i . Base cases are $x(0, 0) = \text{True}$ and $x(0, j) = \text{False}$ 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 to 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

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

• for $i \in \{0, \dots, n\}$, $j \in \{0, \dots, S\}$

3. Topo

- Subproblems $x(i, j)$ only depend on strictly smaller i , so acyclic

4. Base

- $x(i, 0) = 0$ for $i \in \{0, \dots, n\}$ (zero value possible if no more space)
- $x(0, j) = 0$ for $j \in \{1, \dots, S\}$ (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:

- $x(j) = \max\{v_i + x(j - s_i) \mid i \in \{1, \dots, n\} \text{ and } s_i \leq j\} \cup \{0\}$, for $j \in \{0, \dots, S\}$

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>

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

Lecture 19: Complexity

Decision Problems

- **Decision problem:** assignment of inputs to YES (1) or NO (0)
- Inputs are either **NO inputs** or **YES inputs**

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 Simple 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 in given board?
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 (constant) string of bits, i.e., a nonnegative integer $\in \mathbb{N}$.
Problem is function $p : \mathbb{N} \rightarrow \{0, 1\}$, i.e., infinite string of bits.
- (# of programs $|\mathbb{N}|$, countably infinite) \ll (# of problems $|\mathbb{R}|$, uncountably infinite)
- (Proof by Cantor's diagonalization 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 Decision Problems

- | | | |
|------------|-------------------------------------------------------|-------------------------------------------------|
| 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., $\mathbf{P} \subsetneq \mathbf{EXP} \subsetneq \mathbf{R}$ (via time hierarchy theorems, see 6.045)
 - E.g., Chess is in $\mathbf{EXP} \setminus \mathbf{P}$

Nondeterministic Polynomial Time (**NP**)

- **P** is the set of decision problems for which there is an algorithm A such that, for every input I of size n , A on I runs in $\text{poly}(n)$ time and solves I correctly
- **NP** is the set of decision problems for which there is a **verification** algorithm V that takes as input an input 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 input, then there is some certificate c so that V outputs YES on input (I, c) ; and
 - if I is a NO input, then no matter what certificate c we choose, V always output NO on input (I, c) .
- You can think of the certificate as a **proof** that I is a YES input.
If I is actually a NO input, 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 whether $\leq d$
Negative Cycle	A cycle C	Adds the weights on C and checks whether < 0
Longest Simple Path	A path P	Checks whether P is a simple path with weight $\geq d$
Subset Sum	A set of items A'	Checks whether $A' \in A$ has sum S
Tetris	Sequence of moves	Checks that the moves allow survival

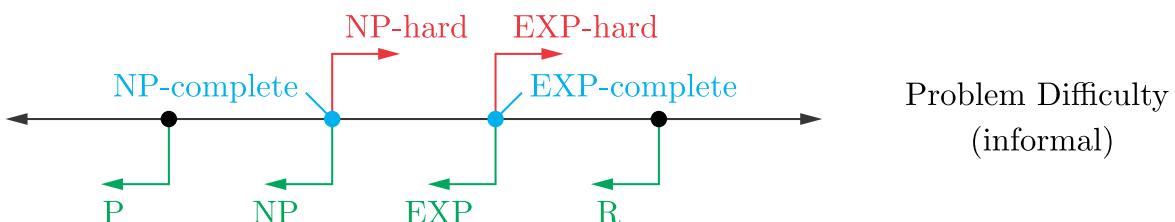
- **P ⊆ NP**: The verifier V just solves the instance ignoring any certificate
- **NP ⊆ EXP**: Try all possible certificates! At most $2^{n^{O(1)}}$ of them, run verifier V on all
- **Open**: Does **P = NP?** **NP = EXP?**
- Most people think **P ⊈ NP** (\subsetneq EXP), i.e., generating solutions harder 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 ≠ 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 ($A \rightarrow B$)
- Because B can be used to solve A , B is **at least as hard** as A ($A \leq B$)
- General algorithmic strategy: reduce to a problem you know how to solve

A	Conversion	B
Unweighted Shortest Path	Give equal weights	Weighted Shortest Path
Integer-weighted Shortest Path	Subdivide edges	Unweighted Shortest Path
Longest Path	Negate weights	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 $X \in \text{NP}$)
- **NP-complete** = $\text{NP} \cap \text{NP-hard}$
- All **NP-complete** problems are equivalent, i.e., reducible to each other
- First **NP-complete** problem? Every decision problem reducible to satisfying a logical circuit, a problem called “Circuit SAT”.
- Longest Simple Path and Tetris are **NP-complete**, so if any problem is in $\text{NP} \setminus \text{P}$, these are
- Chess is **EXP-complete**: in **EXP** and reducible from every problem in **EXP** (so $\notin \text{P}$)



Examples of NP-complete Problems

- Subset Sum from L18 (“weakly NP-complete” which is what allows a pseudopolynomial-time algorithm, but no polynomial algorithm unless $\mathbf{P} = \mathbf{NP}$)
- 3-Partition: given n integers, can you divide them into triples of equal sum? (“strongly NP-complete”: no pseudopolynomial-time algorithm unless $\mathbf{P} = \mathbf{NP}$)
- Rectangle Packing: given n rectangles and a target rectangle whose area is the sum of the n rectangle areas, pack without overlap
 - Reduction from 3-Partition to Rectangle Packing: transform integer a_i into $1 \times a_i$ rectangle; set target rectangle to $n/3 \times (\sum_i a_i) / 3$
- Jigsaw puzzles: given n pieces with possibly ambiguous tabs/pockets, fit the pieces together
 - Reduction from Rectangle Packing: use uniquely matching tabs/pockets to force building rectangles and rectangular boundary; use one ambiguous tab/pocket for all other boundaries
- Longest common subsequence of n strings
- Longest simple path in a graph
- Traveling Salesman Problem: shortest path that visits all vertices of a given graph (or decision version: is minimum weight $\leq d$)
- Shortest path amidst obstacles in 3D
- 3-coloring given graph (but 2-coloring $\in \mathbf{P}$)
- Largest clique in a given graph
- SAT: given a Boolean formula (made with AND, OR, NOT), is it every true?
E.g., x AND NOT x is a NO input
- Minesweeper, Sudoku, and most puzzles
- Super Mario Bros., Legend of Zelda, Pokémon, and most video games are **NP-hard** (many are harder)

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

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 $p : \mathbb{N} \rightarrow \{0, 1\}$, 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. $\mathbf{P} \subsetneq \mathbf{EXP} \subsetneq \mathbf{R}$ (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 $\text{poly}(n)$ 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 ⊂ NP** (if you can solve the problem, the solution is a certificate)
- **Open:** Does **P = NP?** **NP = EXP?**
- Most people think **P ⊂ NP** (\subsetneq EXP), i.e., 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 ≠ NP**
- How do we relate difficulty of problems? Reductions!

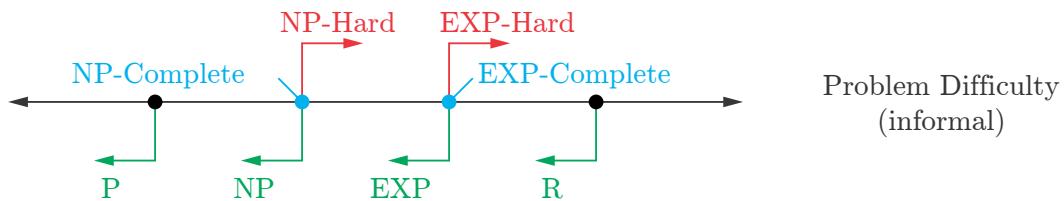
Reductions

- 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 ($A \rightarrow B$)
- 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 $X \in \text{NP}$)
- **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**

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

Problem Session 9

Problem 9-1. Coin Crafting

Ceal Naffrey is a thief in desperate need of money. He recently acquired n identical gold coins. Each coin has distinctive markings that would easily identify them as stolen if sold. However, using his amateur craftsman skills, Ceal can melt down gold coins to craft other golden objects. Ceal has a buyer willing to purchase golden objects at different rates, but will only purchase one of any object. Ceal has compiled a list of the n golden objects, listing both the positive integer **purchase price** the buyer would be willing to pay for each object and each object's positive integer **melting number**: the number of gold coins that would need to be melted to craft that object. Given this list, describe an efficient algorithm to determine the maximum revenue that Ceal could make, by melting down his coins to craft into golden objects to sell to his buyer.

Solution:

1. Subproblems

- Label each craftable object with a unique integer from 1 to n
- Let p_i be the purchase price of object j , with k_i its melting number
- $x(i, j)$: the maximum revenue possible from i coins, being able to craft any of the objects from the objects from 1 to j

2. Relate

- Guess whether or not to craft object j
- If $i < k_i$, object j cannot be crafted
- If object j is not crafted, may recurse on remaining items
- If object j is crafted, receive p_i in revenue and lose k_i coins
- $$x(i, j) = \begin{cases} x(i, j - 1) & \text{if } i < k_i \\ \max(p_i + x(i - k_i, j - 1), x(i, j - 1)) & \text{otherwise} \end{cases}$$

3. Topo. Order

- Subproblem $x(i, j)$ only depends on subproblems with strictly smaller j , so acyclic

4. Base

- If there are not more coins, or no more items, cannot gain any revenue
- $x(0, j) = 0$ for $j \in \{0, \dots, n\}$
- $x(i, 0) = 0$ for $i \in \{0, \dots, n\}$

5. Original

- $x(n, n)$ is the maximum revenue possible from n coins, being able to craft any of the n objects, as requested

6. Time

- # subproblems: $(n + 1)^2 = O(n^2)$, $x(i, j)$ for $i, j \in \{0, 1, \dots, n\}$
- work per subproblem: $O(1)$
- $O(n^2)$ running time

Problem 9-2. Career Fair Optimization

Tim the Beaver always attends the career fair, not to find a career, but to collect free swag. There are n booths at the career fair, each giving out one known type of swag. To collect a single piece of swag from booth i , having integer coolness c_i and integer weight w_i , requires standing in line at that booth for integer t_i minutes. After obtaining a piece of swag from one booth, it will take Tim exactly 1 minute to get back in line at the same booth or any other. Tim's backpack can hold at most weight b in swag; but at any time Tim may spend integer h minutes to run home, empty the backpack, and return to the fair, taking 1 additional minute to get back in a line. Given that the career fair lasts exactly k minutes, describe an $O(nbk)$ -time algorithm to determine the maximum total coolness of swag Tim can collect during the career fair.

Solution:

1. Subproblems

- $x(i, j)$: the maximum total coolness of swag collectible in the next i minutes with j weight remaining in Tim's backpack (where Tim is at the career fair and may immediately stand in line)

2. Relate

- Tim can either:
 - Collect no more swag
 - Stand in a line and collect a piece of swag
 - Go home and empty the backpack

$$\bullet \quad x(i, j) = \max \begin{cases} 0 & \text{always} \\ c_{k'} + x(i - t_{k'}, j - w_{k'}) & \text{for } k' \in \{1, \dots, n\} \text{ where } w_{k'} \leq j, t_{k'} < i \\ x(i - h, b) & \text{when } i > h \end{cases}$$

3. Topo. Order

- Subproblem $x(i, j)$ only depends on subproblems with strictly smaller i , so acyclic

4. Base

- Tim needs time to collect swag, so nothing possible if time is zero
- $x(0, j) = 0$ for $j \in \{0, \dots, b\}$

5. Original

- $x(k, b)$ is the maximum total coolness of swag collectible in k minutes, starting with an empty backpack, as desired.

6. Time

- # subproblems: $\leq (k+1)(b+1) = O(kb)$, $x(i, j)$ for $i \in \{0, \dots, k\}$, $j \in \{0, \dots, b\}$
- work per subproblem: $O(n)$
- $O(nbk)$ running time, which is **psuedo-polynomial** in b and k

Problem 9-3. Protein Parsing

Prof. Leric Ander's lab performs experiments on DNA. After experimenting on any **strand of DNA** (a sequence of nucleotides, either A, C, G, or T), the lab will cut it up so that any useful protein markers can be used in future experiments. Ander's lab has compiled a list P of known protein markers, where each **protein marker** corresponds to a sequence of at most k nucleotides. A **division** of a DNA strand S is an ordered sequence $D = (d_1, \dots, d_m)$ of DNA strands, where the ordered concatenation of D results in S . The **value**

of a division D is the number of DNA strands in D that appear as protein markers in P . Given a DNA strand S and set of protein markers P , describe an $O(k(|P| + k|S|))$ -time algorithm to determine the maximum value of any division of S .

Solution:

1. Subproblems

- First construct a hash table containing the protein markers in P as keys
- This hash table takes (expected) $O(k|P|)$ time to construct (it could take $O(k)$ time to compute the hash of each marker)
- $x(i)$: the maximum value of any division of the DNA strand suffix $S[i :]$

2. Relate

- Either the first nucleotide starts a protein marker or it does not
 - If it does not, recurse on remainder
 - If it does, guess its length (from 1 to k , or until the end of S)
- Let $m(i, j)$ be 1 if substring $S[i : j]$ is a protein marker and 0 otherwise
- We can evaluate $m(i, j)$ in expected $O(k)$ time by hash table look up
- $x(i) = \max\{m(i, j) + x(i + j) \mid j \in \{1, \dots, \min(k, |S| - i)\}\}$

3. Topo. Order

- Subproblem $x(i)$ only depends on subproblems with strictly larger i , so acyclic

4. Base

- If no nucleotides in suffix, no division can have value
- $x(|S|) = 0$

5. Original

- $x(0)$ is the maximum value of any division of S , as desired

6. Time

- # subproblems: $\leq |S| + 1 = O(|S|)$, $x(i)$ for $i \in \{0, \dots, |S|\}$
- work per subproblem: expected $O(k)$ to look up each of the $k m(i, j)$
- $O(k(|P| + k|S|))$ running time, which is **polynomial** in the size of the input
- Note that it is possible to achieve $O(k(|P| + |S|))$ if each of the $\Theta(k|S|)$ lookups can be checked in $O(1)$ time, i.e., via a rolling hash or suffix tree (not covered in this course).

Problem 9-4. Lazy Egg Drop

The classic egg drop problem asks for the minimum number of drops needed to determine the breaking floor of a building with n floors using at most k eggs, where the **breaking floor** is the lowest floor from which an egg could be dropped and break. This problem has a closed form solution, but can also be solved with dynamic programming (Exercise!). However, if the building does not have an elevator, one might instead want to minimize the **total drop height**: the sum of heights from which eggs are dropped. Suppose each of the n floors of the building has a known positive integer height h_i , where floor heights strictly increase with i . Given these heights, describe an $O(n^3k)$ -time algorithm to return the minimum total drop height required to determine the breaking floor of the building using at most k eggs.

Solution:

1. Subproblems

- $x(i, j, e)$ minimum total drop height needed to determine the breaking floor with e eggs, with floors i to $j \geq i$ remaining to check

2. Relate

- Next egg must be dropped from some remaining floor f (guess!)
- If egg breaks, have one fewer egg and need to recurse on floors below
- Otherwise, have same number of eggs and need to recurse on floors above
- $x(i, j, e) = \min\{h_f + \max\{x(i, f - 1, e - 1), x(f + 1, j, e)\} \mid f \in \{i, \dots, j\}\}$

3. Topo. Order

- Subproblem $x(i, j, e)$ only depends on subproblems with strictly smaller $j - i$, so acyclic

4. Base

- Impossible if no more eggs but floors left to check
- $x(i, j, 0) = \infty$ for $i, j \in \{0, \dots, n\}$ with $i \leq j$
- No more drops needed if no more floors left to check
- $x(i, i - 1, e) = 0$ for $i \in \{0, \dots, n\}$ and $e \in \{0, \dots, k\}$

5. Original

- $x(1, n, k)$ is the minimum total drop height needed to determine breaking floor with k eggs needing to check all floors, as desired

6. Time

- # subproblems: $\leq \binom{n}{2}(k + 1) = O(n^2k)$, $x(i, j, e)$ for $i, j \in \{1, \dots, n\}$ with $i < j$ and $e \in \{0, \dots, k\}$
- work per subproblem: $O(\max(1, j - i)) = O(n)$ time
- $O(n^3k)$ running time, which is **psuedo-polynomial** in k . However, since you will never need to drop more than n eggs to check all n floors, it suffices to only compute subproblems for $k \leq n$, so we could answer the problem in $O(n^3)$ -time by limiting the subproblems checked, which is polynomial in n .

Problem 9-5. Building a Wall

The pigs in Porkland from Problem Session 2, have decided to build a stone wall along their southern border for protection against the menacing wolf. The wall will be one meter thick, n meters long, and at most k meters tall. The wall will be built from a large supply of identical **long stones**: each a $1 \times 1 \times 2$ meter rectangular prism. Long stones may be placed either vertically or horizontally in the wall. With much difficulty, a single long stone can be broken into two 1-meter **cube stones**, but the pigs prefer not using cube stones when possible.

The ground along the southern border of Porkland is uneven, but the pigs have leveled each square meter along the border to an integer meter elevation. Let a **border plan** be an $n \times k$ array B correspond to what the border looks like before a wall has been built. $B[j][i]$ corresponds to the cubic meter whose top is at elevation $k - j$, located at meter i along the border. $B[j][i]$ is ' $.$ ' if that cubic meter is **empty** and must be covered by a stone, and ' $#$ ' if that cubic meter is **dirt**, so should not be covered. B has the property that if $B[j][i]$ is covered by dirt, so is every cubic meter $B[t][i]$ beneath it (for $t \in \{j, \dots, k - 1\}$), where

the top-most cubic meter $B[0][i]$ in each column is initially empty. Below is an example B for $n = 10$ and $k = 5$.

A **placement** of stones into border plan B is a set of placement triples:

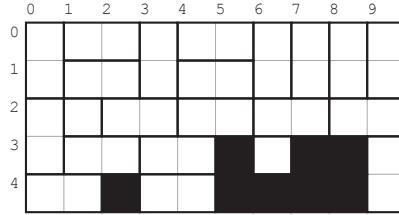
- $(i, j, '1')$ places a cube stone to cover $B[j][i]$;
- $(i, j, 'D')$ places a long stone oriented down to cover $B[j][i]$ and $B[j + 1][i]$; and
- $(i, j, 'R')$ places a long stone oriented right to cover $B[j][i]$ and $B[j][i + 1]$.

A placement is **complete** if every empty cubic meter in B is covered by some stone; and is **non-overlapping** if no cubic meter is covered by more than one stone and no stone overlaps dirt. Below is a complete non-overlapping placement for B that uses 2 cube stones, and a pictorial depiction.

```

1 B = [
2   '.....',          P = [
3   '.....',          (0,0,'D'), (0,2,'D'), (0,4,'R'), (1,0,'R'), (1,1,'R'),
4   '.....',          (1,2,'1'), (1,3,'R'), (2,2,'R'), (3,0,'D'), (3,3,'R'),
5   '....#.##.',    (3,4,'R'), (4,0,'R'), (4,1,'R'), (4,2,'R'), (6,0,'D'),
6   '...#...##.#.',  (6,2,'R'), (6,3,'1'), (7,0,'D'), (8,0,'D'), (8,2,'R'),
7   ]                  (9,0,'D'), (9,3,'D'),
]

```



- (a) Given $n \times k$ border plan B , describe an $O(2^{2k}kn)$ -time algorithm to return a complete non-overlapping placement for B using the fewest cube stones possible.

Solution:

1. Subproblems

- The approach will be to repeatedly cover the highest uncovered cube in the left-most column with some stone
- Placing a single stone will effect at most the first two columns, so our subproblems will remember the current covered state of the two left-most columns
- Represent a **partially-filled column** c of the border as a length- k array of Boolean values, where Boolean $c[j]$ is True if row j in the column still needs to be covered, and False otherwise
- Let $C_0(i)$ correspond to the partially-filled column i in the input border plan B , where $C_0(i)[j]$ is True if $B[j][i] = '.'$ and False if $B[j][i] = '#'$
- Let $C_0(n) = C_0(n+1) = c_F$ correspond to a column of all False values
- $x(i, c_1, c_2)$: the minimum number of cube stones needed to cover partially-covered columns c_1 and c_2 concatenated with original columns $i+2$ to $n-1$

2. Relate

- If all rows of the left-most column are covered, we can move on to cover uncovered cubes in the next column

- Otherwise, place a stone to cover the highest uncovered cube in the left-most column with either:
 - a cube stone;
 - a long stone vertically down (if the cube below is also uncovered); or
 - a long stone horizontally right (if the cube to the right is also uncovered).
- Let $f(c, k)$ correspond to the partially filled column resulting from changing element $c[k]$ in column c to False
- Let $t(c)$ correspond to the index of the first (top-most) True in column c
- $x(i, c_1, c_2) = x(i + 1, c_2, C_0(i + 2))$ if c_1 is all False
- Otherwise, $t(c_1)$ is defined:

$$x(i, c_1, c_2) = \min \left\{ \begin{array}{ll} 1 + x(i, f(c_1, t(c_1)), c_2) & \text{always} \\ x(i, f(f(c_1, t(c_1)), t(c_1) + 1), c_2) & \text{if } c_1[t(c_1) + 1] \text{ is True} \\ x(i, f(c_1, t(c_1)), f(c_2, t(c_1))) & \text{if } c_2[t(c_1)] \text{ is True} \end{array} \right\}$$

3. Topo. Order

- Subproblem $x(i, c_1, c_2)$ either strictly increases i or strictly decreases the number of True values appearing in column c_1 for the same i , so acyclic

4. Base

- No cube stones are required when nothing remains to be covered
- $x(n, c_F, c_F) = 0$

5. Original

- $x(0, C_0(0), C_0(1))$ corresponds to the minimum number of cube stones to fill the entire boarder plan, as desired
- Store parent pointers to reconstruct an optimizing placement

6. Time

- There are 2^k possible columns for each of c_1 and c_2
- # subproblems: $\leq n2^k2^k = O(n2^{2k})$ subproblems
- work per subproblem: $O(k)$ time to lookup in memo and/or to compute a new column
- $O(2^{2k}kn)$ running time, which is **exponential** in k (but polynomial in n)

- (b) Write a Python function `build_wall(B)` that implements your algorithm from (a) **for border plans with $k = 5$.**

Solution:

```

1 def get_col(B, i):
2     if i < len(B[0]):
3         return tuple(B[j][i] == '.' for j in range(5))
4     return tuple(False for j in range(5))
5
6 def build_wall(B):
7     memo = {}
8     def x(i, c1, c2):                                # top-down dynamic program
9         key = (i, c1, c2)
10        if key not in memo:                          # compute if not in memo
11            j = 0
12            while j < 5 and (not c1[j]):                j += 1
13            if j == 5:                                  # column c1 is full
14                if i == len(B[0]) - 1:                  # base case, last line
15                    memo[key] = (0, None, None)
16                else:                                 # shift to next pair of cols
17                    memo[key] = x(i + 1, c2, get_col(B, i + 2))
18                else:                                 # need to cover (i, j)
19                    new_c1 = [i for i in c1]
20                    new_c1[j] = False
21                    test, _, _ = x(i, tuple(new_c1), c2)  # place 1x1
22                    best = 1 + test
23                    move = (i, j, '1')
24                    parent = (i, tuple(new_c1), c2)
25                    if j < 4 and c1[j + 1]:                 # place 1x2 down
26                        new_c1[j + 1] = False
27                        test, _, _ = x(i, tuple(new_c1), c2)
28                        if test < best:
29                            best = test
30                            move = (i, j, 'D')
31                            parent = (i, tuple(new_c1), c2)
32                            new_c1[j + 1] = True
33                    if c2[j]:                                # place 1x2 right
34                        new_c2 = [i for i in c2]
35                        new_c2[j] = False
36                        test, _, _ = x(i, tuple(new_c1), tuple(new_c2))
37                        if test < best:
38                            best = test
39                            move = (i, j, 'R')
40                            parent = (i, tuple(new_c1), tuple(new_c2))
41                            memo[key] = (best, move, parent)
42                    return memo[key]
43    best, move, parent = x(0, get_col(B, 0), get_col(B, 1))
44    P = []                                         # compute placement from parent pointers
45    while parent:
46        P.append(move)
47        best, move, parent = memo[parent]
48    return P

```

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

Quiz 2 Review

Scope

- Quiz 1 material fair game but explicitly **not emphasized**
- 6 lectures on graphs, L09-L14, 2 Problem Sets, PS5-PS6

Graph Problems

- Graph reachability by BFS or DFS in $O(|E|)$ time
- Graph exploration/connected components via Full-BFS or Full-DFS
- Topological sort / Cycle detection via DFS
- Negative-weight cycle detection via Bellman-Ford
- Single Source Shortest Paths (SSSP)

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

- All Pairs Shortest Paths (APSP)
 - Run a SSSP algorithm $|V|$ times
 - Johnson's solves APSP with negative weights in $O(|V|^2 \log |V| + |V||E|)$

Graph Problem Strategies

- Be sure to **explicitly describe a graph** in terms of problem parameters
- Convert problem into finding a shortest path, cycle, topo. sort, conn. comps., etc.
- May help to duplicate graph vertices to encode additional information
- May help to add auxiliary vertices/edges to graph
- May help to pre-process the graph (e.g., to remove part of the graph)

Graph Problem Common Mistakes

- Define your graphs! Specify vertices, edges, and weights clearly (and count them!)
 - (e.g., construct graph $G = (V, E)$ with a vertex for each... and a directed edge (u, v) with weight w for each...)
- State the problem you are solving, not just the algorithm you use to solve it
 - (e.g., solve SSSP from s by running DAG Relaxation...)
- Connect the graph problem you solve back to the original problem
 - (e.g., the weight of a path from s to t in G corresponds to the sum of tolls paid along a driving route, so a path of minimum weight corresponds to a route minimizing tolls)

Problem 1. Counting Blobs (S18 Quiz 2)

An **image** is a 2D grid of black and white square pixels where each white pixel is contained in a **blob**. Two white pixels are in the same blob if they share an edge of the grid. Black pixels are not contained in blobs. Given an $n \times m$ array representing an image, describe an $O(nm)$ -time algorithm to count the number of blobs in the image.

Solution: Construct a graph G with a vertex per white pixel, with an undirected edge between two vertices if the pixels associated with them are both white and share an edge of the grid. This graph has size at most $O(nm)$ vertices and at most $O(nm)$ edges (as pixels share edges with at most four other pixels), so can be constructed in $O(nm)$ time. Each connected component of this graph corresponds to a blob, so run Full-BFS or Full-DFS to count the number of connected components in G in $O(nm)$ time.

Problem 2. Unicycles (S18 Quiz 2)

Given a **connected** undirected graph $G = (V, E)$ with strictly positive weights $w : E \rightarrow \mathbb{Z}^+$ where $|E| = |V|$, describe an $O(|V|)$ -time algorithm to determine a path from vertex s to vertex t of minimum weight.

Solution: Given two vertices in a weighted tree containing only positive weight edges, there is a unique simple path between them which is also the minimum weight path. A depth-first search from any source vertex s in the tree results in a directed DFS tree in $O(|V|)$ time (since $|E| = |V| - 1$). Then relaxing edges in topological sort order of the directed DFS tree computes minimum weight paths from s in $O(|V|)$ time. Since G has one cycle, our strategy will be to break the cycle by removing an edge, and then compute the minimum weight path from s to t in the resultant tree.

First, we find the vertex v closest to s on the cycle by running depth-first search from s in $O(|V|)$ time (since $|E| = |V|$). One edge e_1 of the cycle will not be in the tree returned by DFS (a back edge to v), with the other edge of the cycle incident to v being a single outgoing DFS tree edge e_2 . If s is on the cycle, $v = s$; otherwise the unique path from s to v does not contain e_1 or e_2 .

A shortest path from s to t cannot traverse both edges e_1 and e_2 , or else the path would visit v at least twice, traversing a cycle of positive weight. Removing either e_1 or e_2 results in a tree, at least one of which contains the minimum weight path from s to t . Thus, find the minimum weight path from s to t in each tree using the algorithm described above, returning the minimum of the two in $O(|V|)$ time.

Problem 3. Doh!-nut (S18 Quiz 2)

Momer has just finished work at the FingSpield power plant at location p , and needs to drive to his home at location h . But along the way, if his driving route ever comes within driving distance k of a doughnut shop, he will stop and eat doughnuts, and his wife, Harge, will be angry. Momer knows the layout of FingSpield, which can be modeled as a set of n locations, with two-way roads of known driving distance connecting some pairs of locations (you may assume that no location is incident to more than five roads), as well as the locations of the d doughnut shops in the city. Describe an $O(n \log n)$ -time algorithm to find the shortest driving route from the power plant back home that avoids driving within driving distance k of a doughnut shop (or determine no such route exists).

Solution: Construct a graph G with a vertex for each of the n city locations, and an undirected edge between two locations if there is a road connecting them, with each edge weighted by the positive length of its corresponding road. The degree of each vertex is bounded by a constant (i.e., 5), so the number of edges in G is $O(n)$. First, we identify vertices that are within driving distance k of a doughnut shop location: create an auxiliary vertex x with a 0-weight outgoing edge from x to every doughnut shop location, and run Dijkstra from x . Remove every vertex from the graph whose shortest path from x is less than or equal to k , resulting in graph $G' \subset G$. If either p or h are not in G' , then no route exists. Otherwise, run Dijkstra from p in G' . If no path exists to h , then no valid route exists. Otherwise, Dijkstra finds a shortest path from p to h , so return it (via parent pointers). This algorithm runs Dijkstra twice. Since the size of either graph is $O(|V|)$, Dijkstra runs in $O(|V| \log |V|) = O(n \log n)$ time (e.g. using a binary heap to implement a priority queue).

Problem 4. Long Shortest Paths

Given directed graph $G = (V, E)$ having arbitrary edge weights $w : E \rightarrow \mathbb{Z}$ and two vertices $s, t \in V$, describe an $O(|V|^3)$ -time algorithm to find the minimum weight of any path from s to t containing **at least** $|V|$ edges.

Solution: Our strategy will compute intermediate values for each vertex $v \in V$:

1. the minimum weight $w_1(v)$ of any path from s to v using **exactly** $|V|$ edges, and then
2. the minimum weight $w_2(v)$ of any path from v to t using any number of edges.

First, to compute (1), we make a duplicated graph similar to Bellman-Ford, but without edges corresponding to remaining at a vertex. Specifically, construct a graph G_1 with

- $|V| + 1$ vertices for each vertex $v \in V$: vertex v_k for $k \in \{0, \dots, |V|\}$ representing reaching v from s along a path containing k edges; and
- $|V|$ edges for each edge $(u, v) \in E$: edge (u_{k-1}, v_k) of the same weight for $k \in \{1, \dots, |V|\}$.

Now a path in G_1 from s_0 to $v_{|V|}$ for any $v \in V$ corresponds to a path from s to v in G through exactly $|V|$ edges. So solve SSSPs in G_1 from s_0 to compute the minimum weight of paths to each vertex traversing exactly $|V|$ edges. This graph is acyclic, and has size $O(|V|(|V| + |E|)) = O(|V|^3)$, so we can solve SSSP on G_1 via DAG relaxation in $O(|V|^3)$ time.

Second, to compute (2), we make a new graph G_2 from G where every edge is reversed. Then every path to t in G corresponds to a path in G_2 from t , so compute SSSPs from t in G_2 to find the minimum weight of any path from v to t in G using any number of edges, which can be done in $O(|V||E|) = O(|V|^3)$ time using Bellman-Ford.

Once computed, finding the minimum sum of $w_1(v) + w_2(v)$ over all vertices $v \in V$ will provide the minimum weight of any path from s to t containing at least $|V|$ edges, since every such path can be decomposed into its first $|V|$ edges and then the remainder. This loop takes $O(|V|)$ time, so the algorithm runs in $O(|V|^3)$ time in total.

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

Solution: Quiz 3

- Do not open this quiz booklet until directed to do so. Read all the instructions on this page.
- When the quiz begins, write your name on the top of every page of this quiz booklet.
- You have 60 minutes to earn a maximum of 60 points. Do not spend too much time on any one problem. Skim them all first, and work on them in an order that allows you to make the most progress.
- **You are allowed three double-sided letter-sized sheet with your own notes.** No calculators, cell phones, or other programmable or communication devices are permitted.
- Write your solutions in the space provided. Pages will be scanned and separated for grading. If you need more space, write “Continued on S1” (or S2) and continue your solution on the referenced scratch page at the end of the exam.
- Do not waste time and paper rederiving facts that we have studied in lecture, recitation, or problem sets. Simply cite them.
- When writing an algorithm, a **clear** description in English will suffice. Pseudo-code is not required. Be sure to argue that your **algorithm is correct**, and analyze the **asymptotic running time of your algorithm**. Even if your algorithm does not meet a requested bound, you **may** receive partial credit for inefficient solutions that are correct.
- **Pay close attention to the instructions for each problem.** Depending on the problem, partial credit may be awarded for incomplete answers.

Problem	Parts	Points
0: Information	2	2
1: Pseudo-Circling	3	3
2: Number Scrabble	1	17
3: Limited-Unlimited	1	19
4: Office Hour Optimization	1	19
Total		60

Name: _____

School Email: _____

Problem 1. [2 points] **Information** (2 parts)

- (a) [1 point] Write your name and email address on the cover page.

Solution: OK!

- (b) [1 point] Write your name at the top of each page.

Solution: OK!

Please solve problems (3), (4), and (5) using **dynamic programming**. Be sure to define a set of subproblems, relate the subproblems recursively, argue the relation is acyclic, provide base cases, construct a solution from the subproblems, and analyze running time. Correct but inefficient dynamic programs will be awarded significant partial credit.

Problem 2. [3 points] **Pseudo-Circling**

Indicate whether the given running times of each of problems (3), (4), and (5) are polynomial or pseudopolynomial by circling the appropriate word below. **One can answer this question without actually solving problems (3), (4), and (5).** (1 point each)

- (a) Problem 3: Number Scrabble Polynomial Pseudopolynomial

Solution: Polynomial

- (b) Problem 4: Limited-Unlimited Polynomial Pseudopolynomial

Solution: Pseudopolynomial

- (c) Problem 5: Office Hour Optimization Polynomial Pseudopolynomial

Solution: Polynomial

Problem 3. [17 points] **Number Scrabble**

Number Scrabble is a one-player game played on an array $T = [t_0, \dots, t_{n-1}]$ of n positive integers. There is a list $P = \{(p_0, v(p_0)), \dots, (p_{m-1}, v(p_{m-1}))\}$ of m unique **playable words**, where playable word p_i is a non-empty array of at most 10 positive integers and $v(p_i)$ is the positive integer value of p_i . The objective of the game is to find a **gameplay** S — a list of **non-overlapping** subarrays (i.e., substrings) of T , each a playable word — where S has maximum total value, $\sum_{s \in S} v(s)$. For example, if

$$T = [1, 5, 2, 4, 1] \text{ and } P = \{([2], 3), ([1], 1), ([5, 2], 8), ([1, 2], 12), ([1, 5], 2)\},$$

then $S_1 = ([1, 5], [2], [1])$, $S_2 = ([1], [5, 2], [1])$, and $S_3 = ([1], [2], [1])$ are all valid gameplays, with total values 6, 10, and 5 respectively. Note playable word $[1, 2]$ cannot exist in any gameplay of T , since $[1, 2]$ is not a contiguous subarray of T . Given T and P , describe an $O(n + m)$ -time algorithm to return a gameplay of maximum total value.

Solution: To solve this problem, it would be useful to be able to check whether a particular array of at most 10 positive integers is a playable word. Construct an empty hash table D and insert each p_i for $i \in \{0, \dots, m - 1\}$ into D , mapping to its value $v(p_i)$. Each hash table insertion takes expected constant time (as each p_i has constant size), so constructing D takes expected $O(m)$ time. Now we solve the problem via dynamic programming.

1. Subproblems

- $x(i)$: the maximum total value of any gameplay on suffix $T[i :]$ for $i \in \{0, \dots, n\}$

2. Relate

- Left-most playable word either starts with t_i or it does not
- If playable word starts with t_i , word may have any length in $\{1, \dots, 10\}$ (Guess!)
- $$x(i) = \max \{x(i+1)\} \cup \begin{cases} D[T[i : i+j]] + x(i+j) & j \in \{0, \dots, 10\} \quad \text{and} \\ & i+j \leq n \quad \text{and} \\ & T[i : i+j] \in D \end{cases}$$

3. Topo

- $x(i)$ only depends on subproblems with strictly larger i , so acyclic

4. Base

- $x(n) = 0$ (empty gameplay admits no value)

5. Original

- Solve subproblems via recursive top down or iterative bottom up
- $x(0)$ is the maximum value of any gameplay on T
- Store parent pointers to reconstruct an optimal gameplay

6. Time

- # subproblems: $n + 1 = O(n)$
- Work per subproblem: expected $O(1)$
- Together with hash table construction, yields expected $O(n + m)$ time
- (See scratch S2 for common mistakes)

Problem 4. [19 points] **Limited-Unlimited**

Given two sets of integers A and B , a **limited-unlimited sequence** of A and B is any sequence S of integers such that each integer $s \in S$ appears in either A or B , and if s appears in A then s appears at most once in S . Given a target sum m and two disjoint sets A and B , each containing exactly n **distinct positive** integers, describe an $O(nm)$ -time algorithm to determine whether m is the sum of any limited-unlimited sequence S of A and B , i.e., $m = \sum_{s \in S} s$.

Solution:**1. Subproblems**

- Fix an arbitrary order on $A = (a_0, \dots, a_{n-1})$ and $B = (b_0, \dots, b_{n-1})$
- $x_A(i, k)$: Boolean whether k is sum of any subset of suffix of $A[i :]$ (without repeats)
- $x_B(i, k)$: Boolean whether k is sum of any subset of suffix of $B[i :]$ (allowing repeats)
- for $i \in \{0, \dots, n\}$, $k \in \{0, \dots, m\}$

2. Relate

- Either use a_i once or not (cannot use again)
- $x_A(i, k) = \text{OR} \left\{ \begin{array}{ll} x_A(i+1, k - a_i) & \text{if } a_i \leq k \\ x_A(i+1, k) & \text{always} \end{array} \right\}$
- Either use b_j once or not (but may use again)
- $x_B(i, k) = \text{OR} \left\{ \begin{array}{ll} x_B(i, k - b_j) & \text{if } b_j \leq k \\ x_B(i+1, k) & \text{always} \end{array} \right\}$

3. Topo

- Subproblems $x_A(i, k)$ and $x_B(i, k)$ each depend only on subproblems with strictly smaller $k - i$, so acyclic

4. Base

- $x_s(i, 0) = \text{True}$ for $s \in \{A, B\}$, $i \in \{0, \dots, n\}$ (can always make zero sum)
- $x_s(n, k) = \text{False}$ for $s \in \{A, B\}$, $k \in \{1, \dots, m\}$ (cannot make positive sum)

5. Original

- Solve subproblems via recursive top down or iterative bottom up
- Original is whether a subset of A and a possibly-repeating subset of B sum to m
- i.e., $\text{OR}\{\text{AND}\{x_A(0, k), x_B(0, m - k)\} \mid k \in \{0, \dots, m\}\}$

6. Time

- # subproblems: $(n + 1)(m + 1) = O(nm)$
- Work per subproblem: $O(1)$
- Original takes $O(m)$ time
- $O(nm)$ running time in total
- (See scratch S2 for common mistakes)

Problem 5. [19 points] **Office Hour Optimization**

Class 0.660 (Algorithms for Introductions) is holding online office hours to help students on three problems — a , b , and c — in three corresponding breakout rooms. The TAs want to develop a ‘Sort Bot’ to effectively assign each student to a single room at the start of office hours. Assume there are $3n$ students, where each student i has known nonnegative integer **benefit** a_i , b_i , and c_i for being assigned to the room for problem a , b , and c , respectively.

Describe an $O(n^3)$ -time algorithm to determine whether it is possible to assign the students **equally** to the three breakout rooms (i.e., n students to each room) while providing **strictly positive** help to every student, and if possible, return the maximum total benefit to students of any such assignment. Note that the assignment must not assign a student to a room for which they would get zero benefit.

Solution:**1. Subproblems**

- Let the students be $i \in \{1, \dots, 3n\}$
- $x(i, j, k)$: the maximum benefit assigning students $\{1, \dots, i + j + k\}$ to breakout rooms, with i students to breakout a , j students to breakout b , and k students to breakout c , where each student is assigned to a breakout room with strictly positive benefit (or equals $-\infty$ if no such assignment is possible)
- for $i, j, k \in \{0, \dots, n\}$

2. Relate

- Must assign student $i + j + k$ to some room (Guess!)

$$\bullet \quad x(i, j, k) = \max\{-\infty\} \cup \begin{cases} a_{i+j+k} + x(i-1, j, k) & \text{if } i > 0 \text{ and } a_{i+j+k} > 0, \\ b_{i+j+k} + x(i, j-1, k) & \text{if } j > 0 \text{ and } b_{i+j+k} > 0, \\ c_{i+j+k} + x(i, j, k-1) & \text{if } k > 0 \text{ and } c_{i+j+k} > 0 \end{cases}$$

3. Topo

- Subproblem $x(i, j, k)$ depends only on strictly smaller $i + j + k$, so acyclic

4. Base

- $x(0, 0, 0) = 0$ (no benefit to assigning zero students)

5. Original

- Solve subproblems via recursive top down or iterative bottom up
- $x(n, n, n)$ is the maximum benefit to assign all students evenly to rooms

6. Time

- # subproblems: $(n + 1)^3 = O(n^3)$
- Work per subproblem: $O(1)$
- $O(n^3)$ running time in total
- (See scratch S2 for common mistakes)

SCRATCH PAPER 1. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write “Continued on S1” on the problem statement’s page.

Common Mistakes: (Problem 3: Number Scrabble)

- Not checking whether substrings of T are playable
- A relation that assumes a gameplay must include every number in T
- Not accounting for time to read P in running time
- Not using parent pointers to return a gameplay

Common Mistakes: (Problem 4: Limited-Unlimited)

- Not allowing integers in B to be used more than once
- Trying to store or maintain an arbitrary subset (can have exponential state)

Common Mistakes: (Problem 5: Office Hour Optimization)

- Poor communication of subproblems (not specifying suffix/prefix, etc.)
- Defining $O(n^4)$ subproblems without arguing that only $O(n^3)$ are needed
- Allowing more than n students in a room, or assigning students to rooms with no benefit
- Pseudopolynomially many subproblems with Boolean output (must be polynomial in n)
- Taking the maximum over an OR

SCRATCH PAPER 2. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write “Continued on S2” on the problem statement’s page.

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

Lecture 20: Course Review

6.006: Introduction to Algorithms

- Goals:
 1. Solve hard computational problems (with **non-constant-sized inputs**)
 2. Argue an algorithm is **correct** (Induction, Recursion)
 3. Argue an algorithm is “**good**” (Asymptotics, Model of Computation)
 - (effectively communicate all three above, to human or computer)
- Do there always exist “good” algorithms?
 - Most problems are not solvable efficiently, but many we think of are!
 - **Polynomial** means polynomial in size of input
 - **Pseudopolynomial** means polynomial in size of input AND size of numbers in input
 - NP: **Nondeterministic Polynomial** time, polynomially checkable certificates
 - NP-hard: set of problems that can be used to solve any problem in NP in poly-time
 - NP-complete: intersection of NP-hard and NP

How to solve an algorithms problem?

- Reduce to a **problem** you know how to solve
 - Search/Sort (Q1)
 - * Search: Extrinsic (Sequence) and Intrinsic (Set) Data Structures
 - * Sort: Comparison Model, Stability, In-place
 - Graphs (Q2)
 - * Reachability, Connected Components, Cycle Detection, Topological Sort
 - * Single-Source / All-Pairs Shortest Paths
- Design a new recursive algorithm
 - Brute Force
 - Divide & Conquer
 - Dynamic Programming (Q3)
 - Greedy/Incremental

Next Steps

- (U) 6.046: Design & Analysis of Algorithms
- (G) 6.851: Advanced Data Structures
- (G) 6.854: Advanced Algorithms

6.046

- Extension of 6.006
 - **Data Structures:** Union-Find, Amortization via potential analysis
 - **Graphs:** Minimum Spanning Trees, Network Flows/Cuts
 - **Algorithm Design (Paradigms):** Divide & Conquer, Dynamic Programming, Greedy
 - **Complexity:** Reductions
- Relax Problem (change definition of correct/efficient)
 - **Randomized Algorithms**
 - * 6.006 mostly deterministic (hashing)
 - * Las Vegas: always correct, probably fast (like hashing)
 - * Monte Carlo: always fast, probably correct
 - * Can generally get faster randomized algorithms on structured data
 - **Numerical Algorithms/Continuous Optimization**
 - * 6.006 only deals with integers
 - * Approximate real numbers! Pay time for precision
 - **Approximation Algorithms**
 - * Input optimization problem (min/max over weighted outputs)
 - * Many optimization problems NP-hard
 - * How close can we get to an optimal solution in polynomial time?
- Change Model of Computation
 - Cache Models (memory hierarchy cost model)
 - Quantum Computer (exploiting quantum properties)
 - Parallel Processors (use multiple CPUs instead of just one)
 - * Multicore, large shared memory
 - * Distributed cores, message passing

Future Courses

Model

- Computation / Complexity (6.045, 6.840, 6.841)
- Randomness (6.842)
- Quantum (6.845)
- Distributed / message passing (6.852)
- Multicore / shared memory (6.816, 6.846)
- Graph and Matrix (6.890)
- Constant Factors / Performance (6.172)

Application

- Biology (6.047)
- Game Theory (6.853)
- Cryptography (6.875)
- Vision (6.819)
- Graphics (6.837)
- Geometry (6.850)
- Folding (6.849)

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

Solution: Final

- Do not open this quiz booklet until directed to do so. Read all the instructions on this page.
- When the quiz begins, write your name on the top of every page of this quiz booklet.
- You have 180 minutes to earn a maximum of 180 points. Do not spend too much time on any one problem. Skim them all first, and attack them in the order that allows you to make the most progress.
- **You are allowed three double-sided letter-sized sheet with your own notes.** No calculators, cell phones, or other programmable or communication devices are permitted.
- Write your solutions in the space provided. Pages will be scanned and separated for grading. If you need more space, write “Continued on S1” (or S2, S3, S4, S5, S6, S7) and continue your solution on the referenced scratch page at the end of the exam.
- Do not waste time and paper rederiving facts that we have studied in lecture, recitation, or problem sets. Simply cite them.
- When writing an algorithm, a **clear** description in English will suffice. Pseudo-code is not required. Be sure to argue that your **algorithm is correct**, and analyze the **asymptotic running time of your algorithm**. Even if your algorithm does not meet a requested bound, you **may** receive partial credit for inefficient solutions that are correct.
- **Pay close attention to the instructions for each problem.** Depending on the problem, partial credit may be awarded for incomplete answers.

Problem	Parts	Points
1: Information	2	2
2: Decision Problems	10	40
3: Sorting Sorts	2	24
4: Pythagorean Quad	1	14
5: Animal Counting	1	20
6: Limited Connections	1	14
7: On the Street	1	14
8: RGB Graph	1	18
9: Separated Subsets	1	16
10: A Feast for Crowns	1	18
Total		180

Name: _____

School Email: _____

Problem 1. [2 points] **Information** (2 parts)

- (a) [1 point] Write your name and email address on the cover page.

Solution: OK!

- (b) [1 point] Write your name at the top of each page.

Solution: OK!

Problem 2. [40 points] **Decision Problems** (10 parts)

For each of the following questions, circle either **T** (True) or **F** (False), and **briefly** justify your answer in the box provided (a single sentence or picture should be sufficient). Each problem is worth 4 points: 2 points for your answer and 2 points for your justification. **If you leave both answer and justification blank, you will receive 1 point.**

- (a) **T F** $2^{2n} \in \Theta(2^n)$.

Solution: False. This statement is equivalent to saying $k^2 \in O(k)$ for $k = 2^n$. Constants in exponents matter asymptotically!

- (b) **T F** If $T(n) = \frac{9}{4}T\left(\frac{2}{3}n\right) + n^2$ and $T(1) = \Theta(1)$, then $T(n) = O(n^2)$.

Solution: False. This is an example of Case II of Master Theorem, since $a = \frac{9}{4}$, $b = \frac{3}{2}$, $f(n) = n^2$ and $n^2 = \Theta(n^{\log_{3/2} 9/4} \log^0 n)$. Thus, the recurrence evaluates to $T(n) = \Theta(n^2 \log n)$, which is not $O(n^2)$.

- (c) **T F** Performing an $O(1)$ amortized operation n times on an initially empty data structure takes worst-case $O(n)$ time.

Solution: True. This is the definition of amortization.

- (d) **T F** Given an array A containing n comparable items, sort A using merge sort. While sorting, each item in A is compared with $O(\log n)$ other items of A .

Solution: False. As a counter example, during the final merge step between two sorted halves of the array, each of size $\Theta(n)$, a single item from one array may get compared to all the items from the other list.

- (e) **T F** Given a binary min-heap storing n items with comparable keys, one can build a Set AVL Tree containing the same items using $O(n)$ comparisons.

Solution: False. If such an algorithm A existed, we would be able to sort an array of comparable items in $O(n)$ time, which would contradict the $\Omega(n \log n)$ comparison sort lower bound. Specifically, we could build a binary min-heap from the array using $O(n)$ comparisons, use A to construct a Set AVL Tree in $O(n)$ comparisons, and then return its traversal order.

- (f) **T F** Given a directed graph $G = (V, E)$, run breadth-first search from a vertex $s \in V$. While processing a vertex u , if some $v \in \text{Adj}^+(u)$ has already been processed, then G contains a directed cycle.

Solution: False. BFS can't be used to find directed cycles. A counterexample is $V = \{s, a, b, t\}$ and $E = \{(s, t), (s, a), (a, b), (b, t)\}$. Running BFS from s will first process vertices in levels $\{s\}$, then $\{a, t\}$, then $\{b\}$. When processing vertex b , vertex $t \in \text{Adj}^+(b)$ has already been processed, yet G is a DAG.

- (g) **T F** Run Bellman-Ford on a weighted graph $G = (V, E, w)$ from a vertex $s \in V$. If there is a **witness** $v \in V$, i.e., $\delta_{|V|}(s, v) < \delta_{|V|-1}(s, v)$, then v is on a negative-weight cycle of G .

Solution: False. A witness is only guaranteed to be **reachable** from a negative-weight cycle; it may not actually be **on** a negative-weight cycle.

- (h) **T F** Floyd–Warshall and Johnson’s Algorithm solve all-pairs shortest paths in the same asymptotic running time when applied to weighted **complete** graphs, i.e., graphs where every vertex has an edge to every other vertex.

Solution: True. A complete graph is dense, i.e., $|E| = \Theta(|V|^2)$, so Johnson’s algorithm runs in $O(|V|^2 \log |V| + |V||E|) = O(|V|^3)$ time, which is the same as Floyd–Warshall.

- (i) **T F** If there is an algorithm to solve 0-1 Knapsack in polynomial time, then there is also an algorithm to solve Subset Sum in polynomial time.

Solution: True. Subset Sum is the special case of 0-1 Knapsack. Specifically, one can (in linear time) convert an instance (A, T) of Subset Sum into an equivalent instance of 0-1 Knapsack, with an item i for each integer $a_i \in A$ having size $s_i = a_i$ and value $v_i = a_i$, needing to fill a knapsack of size T ; and then solve the instance via the polynomial-time algorithm for 0-1 Knapsack.

- (j) **T F** Suppose a decision problem A has a pseudopolynomial-time algorithm to solve A . If $P \neq NP$, then A is not solvable in polynomial time.

Solution: False. A problem could have a pseudopolynomial-time algorithm and a polynomial-time algorithm. In fact, any polynomial-time algorithm is also a pseudopolynomial-time algorithm!

Problem 3. [24 points] **Sorting Sorts**

- (a) [12 points] An integer array A is **k -even-mixed** if there are exactly k even integers in A , and the odd integers in A appear in sorted order. Given a k -even-mixed array A containing n distinct integers for $k = \lceil n/\lg n \rceil$, describe an $O(n)$ -time algorithm to sort A .

Solution: Scan through A and put all even integers in order into an array A_E and all odd integers in order into an array A_O (where $|A_E| = k$ and $|A_O| = n - k$). A_O is sorted by definition, and we can sort A_E in $O(k \log k) = O((n/\lg n) \log(n/\lg n)) = O(n)$ time, e.g., via merge sort. Then we can merge sorted A_E and A_O back into A in $O(n)$ time using the merge step of merge sort, using $O(n)$ time in total.

Common Mistakes:

- Using insertion sort
- Splitting into a non-constant number of subarrays and trying to merge
- Using binary search to insert evens into odds (but with linear shifts)
- Using radix or counting sort

- (b) [12 points] Let A be an array of n pairs of positive integers (x_i, y_i) with $x_i, y_i < n^2$ for all $i \in \{0, \dots, n-1\}$. The **power** of pair (x, y) is the integer $x + n^y$. Describe an $O(n)$ -time algorithm to sort the pairs in A increasing by power.

Solution: First note that $x < n^y$ for any integer $y > 1$ and for any $x \in \{0, \dots, n^2-1\}$. Scan through A and put all pairs having $y = 1$ into array A_1 , and all other pairs into array A_2 . Sort A_1 directly by computing and comparing their respective powers $x + n$. Since these values are bounded above by $O(n^2)$, sort A_1 in $O(n)$ time using Radix sort. To sort A_2 , use tuple sort, sorting first by x values and then by y values (since power is more sensitive to changes in y). Since the x and y values are both bounded above by $O(n^2)$, we can use Radix sort for tuple sort's stable sorting algorithm to sort A_2 in $O(n)$ time. Then merge A_1 and A_2 back into A in $O(n)$ time using the merge step of merge sort, using $O(n)$ time in total.

Common Mistakes:

- Not handling $y = 1$ (x may be larger or smaller than n^1)
- Explicitly computing powers (which may have exponential size)
- Concatenating instead of merging two overlapping lists

Problem 4. [14 points] **Pythagorean Quad**

A **Pythagorean Quad** consists of four integers (a, b, c, d) such that $d = \sqrt{a^2 + b^2 + c^2}$. Given an array A containing n distinct positive integers, describe an $O(n^2)$ -time algorithm to determine whether four integers from A form a Pythagorean Quad, where integers from A may appear more than once in the Quad. State whether your running time is worst-case, expected, and/or amortized.

Solution: First, we observe that it suffices to find (a, b, c, d) such that $a^2 + b^2 = d^2 - c^2$. Let P be the set of n^2 ordered pairs of integers from A , where integers in A may be repeated in a pair. Construct an empty hash table H , and for each pair $(a, b) \in P$, compute and insert value $a^2 + b^2$ into H . Then for each pair $(c, d) \in P$, compute and lookup value $d^2 - c^2$ in H . If the value is in H , then some $a^2 + b^2$ equals some $d^2 - c^2$, so return that a Pythagorean Quad exists. Otherwise, if no $d^2 - c^2$ exists in H , then return that a Pythagorean Quad does not exist. Each $a^2 + b^2$ or $d^2 - c^2$ value takes constant time to compute, so computing them all takes worst-case $O(n^2)$ time, while inserting them into or looking them up in the hash table takes expected $O(n^2)$ time, so this algorithm runs expected in $O(n^2)$ time in total.

Common Mistakes:

- Computing non-integers involving logarithms or square roots
- Saying a worst-case running time is amortized
- Trying to write an incorrect dynamic program
- Trying to write an incorrect four-finger algorithm

Problem 5. [20 points] Animal Counting

PurpleRock Park is a wildlife reserve, divided into zones, where each zone has a park ranger who records current **sightings** of animals of different **species** over time. Old animal sightings are periodically removed from the database. A species s is **common** if current park records contain at least 100 sightings of species s within any single zone of the park.

Describe a database to store animal sightings, supporting the following four operations, where n is the number of sightings stored in the database at the time of the operation. State whether your running times are worst-case, expected, and/or amortized.

<code>initialize()</code>	Initialize an empty database in $O(1)$ time
<code>add_sighting(s, i)</code>	Record a newest sighting of species s in zone i in $O(\log n)$ time
<code>remove_oldest()</code>	Remove the oldest sighting stored in the database in $O(\log n)$ time
<code>is_common(s)</code>	Return whether species s is common based on sightings that have not yet been removed from the database in $O(1)$ time

Solution: To implement the database, maintain the following data structures:

- A hash table H mapping each species s to a Set AVL tree T_s
- Each Set AVL Tree T_s stores pairs (i, c_i) of zone numbers i and the count c_i representing the number of sightings of species s in zone i , keyed by zone number.
- Augment each node x in each T_s by the maximum number of sightings $x.m$ of any zone in the subtree of x . $x.m$ can be maintained in $O(1)$ time from the augmentations of x 's children, specifically $x.m = \max\{x.left.m, x.key, x.right.m\}$.
- A doubly-linked list L of all current sightings (s, i) in the order in which they were added to the database (oldest at the front).

To implement `initialize()`, initialize an empty H and empty L in worst-case $O(1)$ time.

To implement `add_sighting(s , i)`, lookup s in H to find T_s in expected $O(1)$ time (if s does not exist in H , insert s mapping to an empty T_s in expected amortized $O(1)$ time). Then find zone i in T_s . If zone i is not in T_s , insert $(i, 1)$ into T_s . Otherwise, i is in T_s , so remove (i, c_i) from T_s and reinsert $(i, c_i + 1)$. It takes worst-case $O(\log n)$ time to remove or insert items from T_s while maintaining augmentations (since at most n sightings could exist for species s). Lastly, Insert (s, i) to the back of L in worst-case $O(1)$ time. Thus this operation takes $O(\log n)$ expected amortized time, and maintains the invariants of the database directly.

To implement `remove_oldest()`, remove the oldest pair (s, i) from the front of L in worst-case $O(1)$ time. Lookup s in H to find T_s in expected $O(1)$ time; then lookup i in T_s and decrease c_i by one. If c_i is decreased to zero, remove i from T_s . If T_s becomes empty, remove s from H in expected amortized $O(1)$ time. This operation takes $O(\log n)$ expected amortized time, and maintains the invariants of the database directly.

To implement `is_common(s)`, simply lookup s in H and return whether s is in H and the stored max at the root of T_s is 100 or greater in expected $O(1)$ time. This operation is correct based on the invariants of the data structure.

Common Mistakes: Continued on S1

Problem 6. [14 points] **Limited Connections**

For any weighted graph $G = (V, E, w)$ and integer k , define G_k to be the graph that results from removing every edge in G having weight k or larger.

Given a connected undirected weighted graph $G = (V, E, w)$, where every edge has a unique integer weight, describe an $O(|E| \log |E|)$ -time algorithm to determine the largest value of k such that G_k is not connected.

Solution: Construct an array A containing the $|E|$ distinct edge weights in G , and sort it in $O(|E| \log |E|)$ time, e.g., using merge sort. We will binary search to find k . Specifically, consider an edge weight k' in A (initially the median edge weight), and run a reachability algorithm (e.g., Full-BFS or Full-DFS) to compute the reachability of an arbitrary vertex $x \in V$ in $O(|E|)$ time. If exactly $|V|$ vertices are reachable from x , then $G_{k'}$ is connected and $k > k'$; recurse on strictly larger values for k' . Otherwise, $G_{k'}$ is not connected, so $k \leq k'$; recurse on non-strictly smaller values for k' . By dividing the search range by a constant fraction at each step (i.e., by always choosing the median index weight of the unsearched space), binary search will terminate after $O(\log |E|)$ steps, identifying the largest value of k such that G_k is not connected. This algorithm takes $O(|E| \log |E|)$ time to sort, and computes reachability of a vertex in $O(|E|)$ time, $O(\log |E|)$ times, so this algorithm runs in $O(|E| \log |E|)$ time in total.

Common Mistakes:

- Filtering linearly on all edge weights, rather than using binary search
- Using Dijkstra/Bellman-Ford to inefficiently solve reachability
- Simply stating ‘Use Dijkstra’

Problem 7. [14 points] **On the Street**

Friends Dal and Sean want to take a car trip across the country from Yew Nork to Fan Sancrisco by driving between cities during the day, and staying at a hotel in some city each night. There are n cities across the country. For each city c_i , Dal and Sean have compiled:

- the positive integer expense $h(c_i)$ of **staying at a hotel** in city c_i for one night; and
- a list L_i of the at most 10 other cities they could drive to **in a single day** starting from city c_i , along with the positive integer expense $g(c_i, c_j)$ required to drive directly from c_i to c_j for each $c_j \in L_i$.

Describe an $O(nd)$ -time algorithm to determine whether it is possible for Dal and Sean to drive from Yew Nork to Fan Sancrisco in at most d days, spending at most b on expenses along the way.

Solution: Let $C = \{c_0, \dots, c_{n-1}\}$, and let c_s denote Yew Nork and let c_t denote Fan Sancrisco. Construct a graph G with:

- a vertex (c_i, d') for each city $c_i \in C$ and day $d' \in \{0, \dots, d\}$, representing staying the night in city c_i on the night before day d' ; and
- a directed weighted edge $((c_i, d'), (c_j, d' + 1))$ with weight $g(c_i, c_j) + h(c_j)$ for each city $c_i \in C$, $c_j \in L_i$ and $d' \in \{0, \dots, d - 1\}$.

Then the weight of any path in G from vertex $(c_s, 0)$ to any vertex (c_t, d') for $d' \in \{0, \dots, d\}$ corresponds to the expenses incurred along a driving route from Yew Nork to Fan Sancrisco in at most d days (assuming they stay the night upon reaching c_t ; other assumptions are also okay). G is acyclic, since each edge always connects a vertex from a smaller day to a larger day, so run DAG Relaxation to compute single-source shortest paths from $(c_s, 0)$ in G . Then return whether $\delta((c_s, 0), (c_t, d')) \leq b$ for any $d' \in \{0, \dots, d\}$. G has $O(nd)$ vertices and $O(nd)$ edges (since $|L_i| \leq 10$ for all $i \in \{0, \dots, n - 1\}$), so DAG relaxation runs in $O(nd)$ time and checking all destination values takes $O(d)$ time, leading to $O(nd)$ time in total.

Common Mistakes:

- Not considering paths on fewer than d days (or considering paths on more than d days)
- Not enforcing staying at a hotel every night
- Not including hotel costs

Problem 8. [18 points] **RGB Graph**

Let $G = (V, E, w)$ be a weighted directed graph. Let $c : V \rightarrow \{r, g, b\}$ be an assignment of each vertex v to a color $c(v)$, representing red, green, or blue respectively. For $x \in \{r, g, b\}$,

- let V_x be the set of vertices with color x , i.e., $V_x = \{v \in V \mid c(v) = x\}$; and
- let E_x be the set of edges outgoing from vertices in V_x , i.e., $E_x = \{(u, v) \in E \mid u \in V_x\}$.

Suppose graph G and coloring c have the following properties:

1. Every edge in E either connects two vertices of the same color, goes from a red vertex to a green vertex, or goes from a green vertex to a blue vertex.
2. $|V_r| = |E_r| = O(|V|)$, and edges in E_r have identical positive integer weight w_r .
3. $|V_g| = |E_g| = O(|V|^{0.99})$, and edges in E_g have nonnegative integer weights.
4. $|V_b| = |E_b| = O(\sqrt{|V|})$, and edges in E_b can have positive or negative integer weights.

Given G , c , a red vertex $s \in V_r$, and a blue vertex $t \in V_b$, describe an $O(|V|)$ -time algorithm to compute $\delta(s, t)$, the minimum weight of any path from s to t .

Solution: Any path from s to t is a path through edges in E_r , followed by a path through edges in E_g , followed by a (possibly empty) path through edges in E_b . So we compute minimum weight distances in G incrementally, first using edges in E_r , then using edges in E_g , then edges in E_b .

Step 1: Construct unweighted graph $G' = (V', E')$ composed of the edges $E' = E_r$ and the vertices appearing in those edges, specifically $V' = \bigcup_{(u,v) \in E_r} \{u, v\}$ (which contains vertices from V_r and V_g). Run breadth-first search from s in G' to compute unweighted distances. Then the minimum weight distance in G from s to any green vertex in $V' \cap V_g$ is w_r times the unweighted distance computed. G' has size $O(|V|)$, so this step takes $O(|V|)$ time.

Step 2: Now construct weighted graph $G'' = (V'', E'')$ composed of vertex s with a new directed edge to each green vertex in $V' \cap V_g$ weighted by its distance found in Step 1 (i.e., the minimum weight of any path from s to that vertex), along with weighted edges E_g and the vertices appearing in those edges. All the weights in G'' are positive, so run Dijkstra from s in G'' to compute minimum weight distances. Then the computed distance to any blue vertex v in $V'' \cap V_b$ is the minimum weight of any path from s to v in G that traverses only red or green edges. G'' has size $O(1 + |V_g| + |E_g|) = O(|V|^{0.99})$, so this step takes $O(|V|^{0.99} \log |V|^{0.99}) = O(|V|)$ time.

Step 3: Now construct a weighted graph $G''' = (V''', E''')$ composed of vertex s with a new directed edge to each blue vertex in $V'' \cap V_b$ weighted by its distance found in Step 2 (i.e., the minimum weight of any path from s to that vertex), along with weighted edges E_b and the vertices appearing in those edges. Weights in G''' may be positive or negative, so run Bellman-Ford from s in G''' to compute weighted minimum weight distances. Then the computed distance to t is the minimum weight of any path from s to t in G , as desired. G''' has size $O(1 + |V_b| + |E_b|) = O(\sqrt{|V|})$, so this step takes $O(\sqrt{|V|} \sqrt{|V|}) = O(|V|)$ time, leading to $O(|V|)$ time in total.

Common Mistakes: Continued on S1.

Problem 9. [16 points] Separated Subsets

For any set S of integers and for any positive integers m and k , an (m, k) -separated subset of S is any subset $S' \subseteq S$ such that S' sums to m and every pair of distinct integers $a, b \in S'$ satisfies $|a - b| \geq k$. Given positive integers m and k , and a set S containing n distinct positive integers, describe an $O(n^2m)$ -time algorithm to **count the number** of (m, k) -separated subsets of S .

(When solving this problem, you may assume that a single machine word is large enough to hold any integer computed during your algorithm.)

Solution:**1. Subproblems**

- First sort the integers in S increasing into array A in $O(n \log n)$ time, e.g., via merge sort
- where $A = (a_0, \dots, a_{n-1})$
- $x(i, j)$: the number of (j, k) -separated subsets of suffix $A[i :]$
- for $i \in \{0, \dots, n\}$ and $j \in \{0, \dots, m\}$

2. Relate

- Sum the number of (j, k) -separated subsets using $A[i]$ with the ones that do not use $A[i]$
- If $A[i] \leq j$ is used:
 - Then no integer in $A[i :]$ smaller than $A[i] + k$ may be used
 - Let $f(i)$ be the smallest index greater than i such that $A[f(i)] - A[i] \geq k$
 - Then recursively count $x(f(i), j - A[i])$
- Otherwise, $A[i]$ is not used and we can recursively count $x(i + 1, j)$
- $$x(i, j) = \sum \left\{ \begin{array}{ll} x(f(i), j - A[i]) & \text{if } A[i] \leq j, \\ x(i + 1, j) & \text{always} \end{array} \right\}$$

3. Topo

- Subproblem $x(i, j)$ only depends on strictly larger i , so acyclic

4. Base

- $x(n, 0) = 1$, the empty subset can always be achieved
- $x(n, j) = 0$ for $j > 0$, empty sets cannot sum to a positive number

5. Original

- $x(0, m)$, the number of (m, k) -separated subsets of A

6. Time

- # subproblems: $(n + 1)(m + 1) = O(nm)$
- Work per subproblem: $O(n)$ to find $f(i)$ by linear scan
- $O(n^2m)$ time in total
- (Note that it is possible to compute $f(i)$ in $O(\log n)$ time via binary search, or in amortized $O(1)$ time from $f(i - 1)$, but these optimizations are not necessarily for full points.)

Common Mistakes: Continued on S1.

Problem 10. [18 points] A Feast for Crowns

Ted Snark is arranging a feast for the Queen of Southeros and her guests, and has been tasked with seating them along one side of a long banquet table.

- Ted has a list of the $2n$ guests, where each guest i has a known distinct positive integer f_i denoting the guest's **favor** with the Queen.
- Ted must seat the guests **respectfully**: the Queen must be seated in the center with n guests on either side so that guests' favor monotonically decreases away from the Queen, i.e., any guest seated between a guest i and the Queen must have favor larger than f_i .
- Additionally, every guest hates every other guest: for every two guests i, j , Ted knows the positive integer **mutual hatred** $d(i, j) = d(j, i)$ between them.

Given Ted's guest information, describe an $O(n^3)$ -time algorithm to determine a respectful seating order that minimizes the sum of mutual hatred between pairs of guests seated next to each other. **Significant partial credit** will be awarded to correct $O(n^4)$ -time algorithms.

Solution:**1. Subproblems**

- Sort the guests increasing by favor in $O(n \log n)$ time into $F = (f_0, \dots, f_{2n-1})$
- Any partition of F into two length- n subsequences corresponds to a respectful seating
- $x(i, j_L, j_R, n_L)$: minimum total hatred of adjacent guests possible by respectfully seating the $n - i$ guests from suffix $F[i :]$ next to the Queen, with n_L guests to the left and $n_R = (n - i) - n_L$ guests to the right, where guest $j_L < i$ has already been seated $n_L + 1$ places to the left, and guest $j_R < i$ has already been seated $n_R + 1$ places to the right.
- for $i \in \{0, \dots, 2n\}$, $j_L, j_R \in \{-1, \dots, 2n - 1\}$ and $n_L \in \{0, \dots, n\}$
where either $j_L = i - 1$ or $j_R = i - 1$
- Let $d(-1, i) = d(i, -1) = 0$ for all $i \in \{0, \dots, 2n - 1\}$ (no hatred at the end of table)

2. Relate

- Guess whether guest i is seated on the right or left
- Sitting next to j_L costs hatred $d(i, j_L)$; sitting next to j_R costs hatred $d(i, j_R)$
- $$x(i, j_L, j_R, n_L) = \min \begin{cases} d(i, j_L) + x(i+1, i, j_R, n_L - 1) & \text{if } n_L > 0, \\ d(i, j_R) + x(i+1, j_L, i, n_L) & \text{if } (n - i) - n_L > 0 \end{cases}$$

3. **Topo:** Subproblem $x(i, j_L, j_R, n_L)$ only depends on strictly larger i , so acyclic

4. **Base:** $x(2n, j_L, j_R, 0) = 0$ for all $j_L, j_R \in \{0, \dots, 2n\}$ (no hatred if no guests)

5. **Original:** $x(0, -1, -1, n)$, min hatred of adjacent guests by respectfully seating all guests

6. Time

- # subproblems: though there are four parameters, there are only $O(n^3)$ subproblems for which either $j_L = i - 1$ or $j_R = i - 1$
- Work per subproblem: $O(1)$, so $O(n^3)$ time in total

Common Mistakes: Continued on S1.

SCRATCH PAPER 1. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write “Continued on S1” on the problem statement’s page.

Common Mistakes: (for Problem 5)

- Not removing records for sightings or zones containing zero sightings
- Not accounting for existence or non-existence of a key in a data structure
- Algorithm is linear in number of zones or does account for zones at all
- Forgetting to say amortized for dynamic hash table operations
- Confusing Sequence/Set AVL Trees, or dynamic/direct-access arrays
- Not maintaining or updating number of sightings per zone per species correctly

Common Mistakes: (for Problem 8)

- Correctly finding shortest paths within each color, but not connecting them properly
- Assuming an optimal path from s to t goes through the green vertex closest to s
- Solving APSP on the red or green graph (inefficient)
- Trying to apply DAG relaxation to a possibly cyclic graph
- Assuming that if $|V| = |E|$, graph has at most one cycle (only true if connected)

Common Mistakes: (for Problem 9)

- Maintaining a subset of used items (yields exponential state)
- Unconstrained subproblem for relation that relies on a constraint
- Solving the decision problem rather than the counting problem (or max instead of sum)
- Missing a base case

Common Mistakes: (for Problem 10)

- Not ensuring that n guests are seated to either side of Queen
- Not keeping track of guests on either end
- Defining $\Omega(n^3)$ subproblems whose parameters are not independent
- Maintaining a subset of used items (yields exponential state)

SCRATCH PAPER 2. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write “Continued on S2” on the problem statement’s page.

SCRATCH PAPER 3. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write “Continued on S3” on the problem statement’s page.

SCRATCH PAPER 4. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write “Continued on S4” on the problem statement’s page.

SCRATCH PAPER 5. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write “Continued on S5” on the problem statement’s page.

SCRATCH PAPER 6. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write “Continued on S6” on the problem statement’s page.

SCRATCH PAPER 7. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write “Continued on S7” on the problem statement’s page.

MIT OpenCourseWare

<https://ocw.mit.edu>

6.006 Introduction to Algorithms

Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>