

## 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 ( $\Omega$ ), tight bounds ( $\Theta$ )  $\in, =, \text{is, order}$
  - Time estimate below based on one operation per cycle on a **1 GHz** single-core machine
  - Particles in universe estimated  $< 10^{100}$  1,000,000,000 Hz

$2^{10}=1024$

| 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           | $\approx 10$     | 1000        | $\approx 10,000$   | 1,000,000     | $1000^c$      | $2^{1000} \approx 10^{301}$ |
| Time  | 1 ns        | 10 ns            | 1 $\mu s$   | 10 $\mu s$         | 1 ms          | $10^{3c-9} s$ | $10^{281}$ millenia         |

nano                      micro

$/10^9$  ↗

## 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:  $(\&\&, ||, !, ==, <, >, <=, ==>)$
  - **(bitwise** arithmetic:  $(\&, |, <<, >>, \dots)$
  - 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  $\sim 4$  GB memory, 64-bit words  $\rightarrow$  max  $\sim 16$  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>

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

- Special case interfaces:

|              |   |
|--------------|---|
| <b>stack</b> | <code>insert_last(x)</code> and <code>delete_last()</code>  |
| <b>queue</b> | <code>insert_last(x)</code> and <code>delete_first()</code> |

## 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><br><code>len()</code>   | given an iterable $x$ , build sequence from items in $x$<br>return the number of stored items  |
| Static    | <code>find(k)</code>  | return the stored item with key $k$  |
| Dynamic   | <code>insert(x)</code><br><code>delete(k)</code>  | add $x$ to set (replace item with key $x.key$ if one already exists)<br>remove and return the stored item with key $k$   |
| Order     | <code>iter_ord()</code><br><code>find_min()</code><br><code>find_max()</code><br><code>find_next(k)</code><br><code>find_prev(k)</code> | return the stored items one-by-one in key order<br>return the stored item with smallest key<br>return the stored item with largest key<br>return the stored item with smallest key larger than $k$<br>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><br><code>set_at(i, x)</code> | <code>insert_first(x)</code><br><code>delete_first()</code> | <code>insert_last(x)</code><br><code>delete_last()</code> | <code>insert_at(i, x)</code><br><code>delete_at(i)</code> |
| Array          | $n$                              | 1   | $n$   | $n$   | $n$   |

## 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><br><code>set_at(i, x)</code> | <code>insert_first(x)</code><br><code>delete_first()</code> | <code>insert_last(x)</code><br><code>delete_last()</code> | <code>insert_at(i, x)</code><br><code>delete_at(i)</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                           |                                 |                                 |
|                | build(X)                         | get_at(i)<br>set_at(i, x) | insert_first(x)<br>delete_first() | insert_last(x)<br>delete_last() | insert_at(i, x)<br>delete_at(i) |
| 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>

## Lecture 3: Sorting

### Set Interface (L03-L08)

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

- 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<br>Data Structure | Operations $O(\cdot)$ |                      |  |  |  |
|-----------------------|-----------------------|----------------------|--|--|--|
|                       | Container             | Static               | Dynamic  | Order  |  |
|                       | <code>build(X)</code> | <code>find(k)</code> | <code>insert(x)</code><br><code>delete(k)</code> | <code>find_min()</code><br><code>find_max()</code> | <code>find_prev(k)</code><br><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:  $S(n) = \Theta(n), \quad cn = \Theta(1) + c(n - 1) \implies 1 = \Theta(1)$
    - \* 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:  $T(n) = \Theta(n^2), \quad cn^2 = \Theta(n) + c(n - 1)^2 \implies c(2n - 1) = \Theta(n)$
    - \* 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]'''
3      if b is None: b = len(A)                                     # O(1)
4      if 1 < b - a:                                               # O(1)
5          c = (a + b + 1) // 2                                     # O(1)
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]'''
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>



## Lecture 4: Hashing

### Review

| Data Structure | Operations $O(\cdot)$ |          |                        |                          |                              |
|----------------|-----------------------|----------|------------------------|--------------------------|------------------------------|
|                | Container             | Static   | Dynamic                | Order                    |                              |
|                | build(X)              | find(k)  | insert(x)<br>delete(k) | find_min()<br>find_max() | find_prev(k)<br>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  $h(a) = h(b) \rightarrow$  **Collision!** :(
  - 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  $\Theta(n)$  :(
  - 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)<br>delete(k) | find_min()<br>find_max() | find_prev(k)<br>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>

## 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)<br>delete(k) | find_min()<br>find_max() | find_prev(k)<br>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  $x.key = (x.k_1, x.k_2, x.k_2, \dots)$ .
- 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:**  $[32, 03, 44, 42, 22] \implies [42, 22, 32, 03, 44] \implies [03, 22, 32, 42, 44]_{(n=5)}$

- **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]
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

```

| 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>

## Lecture 6: Binary Trees I

### Previously and New Goal

| Sequence<br>Data Structure | Operations $O(\cdot)$ |                           |                                   |                                 |                                 |
|----------------------------|-----------------------|---------------------------|-----------------------------------|---------------------------------|---------------------------------|
|                            | Container             | Static                    | Dynamic                           |                                 |                                 |
|                            | build(X)              | get_at(i)<br>set_at(i, x) | insert_first(x)<br>delete_first() | insert_last(x)<br>delete_last() | insert_at(i, x)<br>delete_at(i) |
| Array                      | $n$                   | 1                         | $n$                               | $n$                             | $n$                             |
| Linked List                | $n$                   | $n$                       | 1                                 | $n$                             | $n$                             |
| Dynamic Array              | $n$                   | 1                         | $n$                               | $1_{(a)}$                       | $n$                             |
| <b>Goal</b>                | $n$                   | $\log n$                  | $\log n$                          | $\log n$                        | $\log n$                        |

| Set<br>Data Structure | Operations $O(\cdot)$ |           |                        |                          |                              |
|-----------------------|-----------------------|-----------|------------------------|--------------------------|------------------------------|
|                       | Container             | Static    | Dynamic                | Order                    |                              |
|                       | build(X)              | find(k)   | insert(x)<br>delete(k) | find_min()<br>find_max() | find_prev(k)<br>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            | $n_{(e)}$             | $1_{(e)}$ | $1_{(a)(e)}$           | $n$                      | $n$                          |
| <b>Goal</b>           | $n \log n$            | $\log n$  | $\log n$               | $\log n$                 | $\log n$                     |

### 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> |  | <B> |  | <C> |  | <D> |  | <E> |  | <F> |  |
| 2 |  | ___<B>_____<C> | item   |  | A   |  | B   |  | C   |  | D   |  | E   |  | F   |  |
| 3 |  | ___<D>_____<E> | parent |  | -   |  | <A> |  | <A> |  | <B> |  | <B> |  | <D> |  |
| 4 |  | <F>            | left   |  | <B> |  | <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
- 

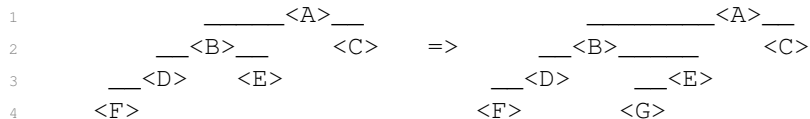
## Tree Navigation

- **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

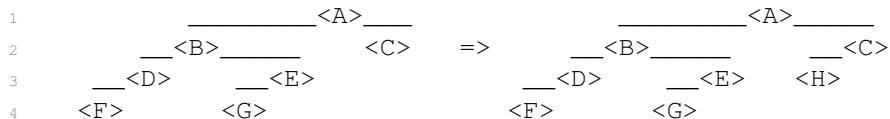
## Dynamic Operations

- 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

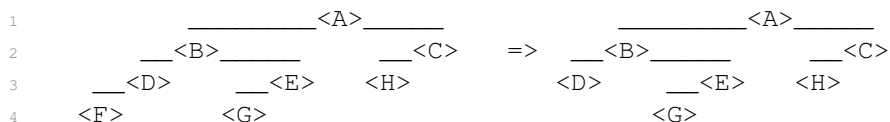
- **Example:** Insert node  $\langle G \rangle$  before  $\langle E \rangle$  in traversal order



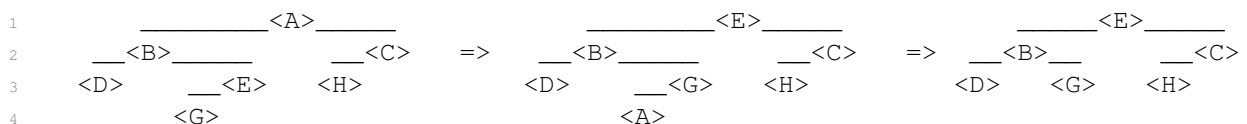
- **Example:** Insert node  $\langle H \rangle$  after  $\langle A \rangle$  in traversal order



- **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  $i^{\text{th}}$  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<br>Data Structure | Operations $O(\cdot)$ |          |                        |                          |                              |
|-----------------------|-----------------------|----------|------------------------|--------------------------|------------------------------|
|                       | Container             | Static   | Dynamic                | Order                    |                              |
|                       | build(X)              | find(k)  | insert(x)<br>delete(k) | find_min()<br>find_max() | find_prev(k)<br>find_next(k) |
| Binary Tree           | $n \log n$            | $h$      | $h$                    | $h$                      | $h$                          |
| <b>Goal</b>           | $n \log n$            | $\log n$ | $\log n$               | $\log n$                 | $\log n$                     |

| Sequence<br>Data Structure | Operations $O(\cdot)$ |                           |                                   |                                 |                                 |
|----------------------------|-----------------------|---------------------------|-----------------------------------|---------------------------------|---------------------------------|
|                            | Container             | Static                    | Dynamic                           |                                 |                                 |
|                            | build(X)              | get_at(i)<br>set_at(i, x) | insert_first(x)<br>delete_first() | insert_last(x)<br>delete_last() | insert_at(i, x)<br>delete_at(i) |
| Binary Tree                | $n$                   | $h$                       | $h$                               | $h$                             | $h$                             |
| <b>Goal</b>                | $n$                   | $\log n$                  | $\log n$                          | $\log n$                        | $\log n$                        |

## 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>



## Lecture 7: Binary Trees II: AVL

### Last Time and Today's Goal

| Sequence<br>Data Structure | Operations $O(\cdot)$ |                           |                                   |                                 |                                 |
|----------------------------|-----------------------|---------------------------|-----------------------------------|---------------------------------|---------------------------------|
|                            | Container             | Static                    | Dynamic                           |                                 |                                 |
|                            | build(X)              | get_at(i)<br>set_at(i, x) | insert_first(x)<br>delete_first() | insert_last(x)<br>delete_last() | insert_at(i, x)<br>delete_at(i) |
| Binary Tree                | $n$                   | $h$                       | $h$                               | $h$                             | $h$                             |
| AVL Tree                   | $n$                   | $\log n$                  | $\log n$                          | $\log n$                        | $\log n$                        |

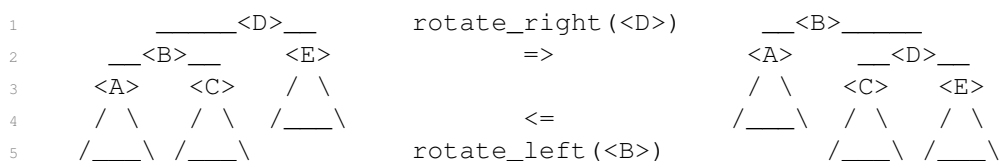
| Set<br>Data Structure | Operations $O(\cdot)$ |          |                        |                          |                              |
|-----------------------|-----------------------|----------|------------------------|--------------------------|------------------------------|
|                       | Container             | Static   | Dynamic                | Order                    |                              |
|                       | build(X)              | find(k)  | insert(x)<br>delete(k) | find_min()<br>find_max() | find_prev(k)<br>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  $\pm 1$ , so skews still have magnitude  $\leq 2$
  - **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

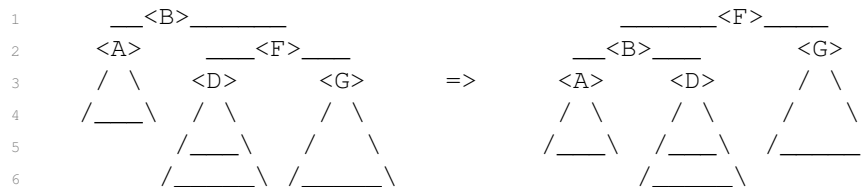
- **Local Rebalance:** Given binary tree node  $\langle B \rangle$ :

- 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  $\langle B \rangle$



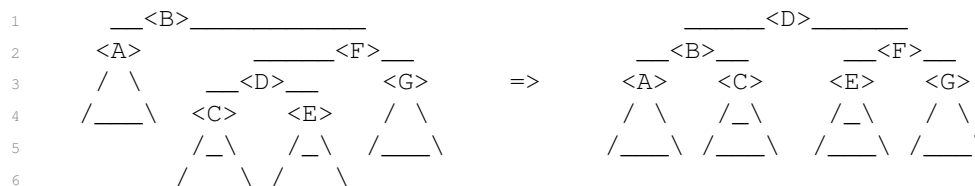
- \* Let  $h = \text{height}(\langle A \rangle)$ . Then  $\text{height}(\langle G \rangle) = h + 1$  and  $\text{height}(\langle D \rangle)$  is  $h + 1$  in Case 1,  $h$  in Case 2

- \* 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>

## 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:
 

|                    |  |
|--------------------|--|
| build( <i>X</i> )  | build priority queue from iterable <i>X</i>    |
| insert( <i>x</i> ) | add item <i>x</i> to data structure            |
| delete_max()       | remove and return stored item with largest key |
| find_max()         | 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<br>Data Structure | Operations $O(\cdot)$ |                    |                | Priority Queue Sort |           |                |
|----------------------------------|-----------------------|--------------------|----------------|---------------------|-----------|----------------|
|                                  | build( <i>A</i> )     | insert( <i>x</i> ) | delete_max()   | Time                | In-place? |                |
| Dynamic Array                    | $n$                   | $1_{(a)}$          | $n$            | $n^2$               | Y         | Selection Sort |
| Sorted Dynamic Array             | $n \log n$            | $n$                | $1_{(a)}$      | $n^2$               | Y         | Insertion Sort |
| Set AVL Tree                     | $n \log n$            | $\log n$           | $\log n$       | $n \log n$          | N         | AVL Sort       |
| <b>Goal</b>                      | $n$                   | $\log n_{(a)}$     | $\log n_{(a)}$ | $n \log n$          | Y         | 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)
- 

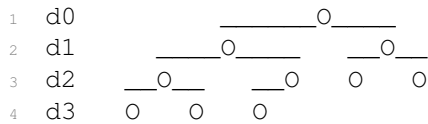
## Priority Queue: Sorted Array

- 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?

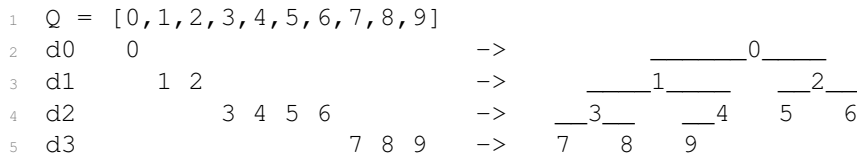


## Array as a Complete Binary Tree

- **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(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
  - `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 `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>

## Lecture 9: Breadth-First Search

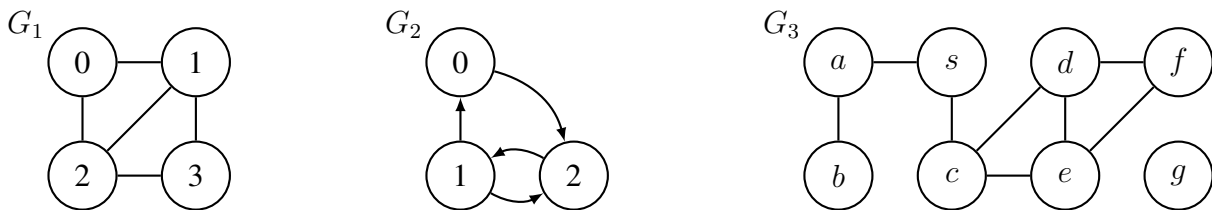
### New Unit: Graphs!

- 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  $\text{Adj}$  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  $\text{Adj}$ , since want lookup fast by vertex
- Common to use **array** or **linked list** for each  $\text{Adj}(u)$  since usually only iteration is needed<sup>1</sup>
- For the common representations,  $\text{Adj}$  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  $\text{Adj}$ , and store  $\text{Adj}(u)$  in an array. Example 3 uses a hash table for  $\text{Adj}$ .

|                   |  |                  |  |                              |
|-------------------|--|------------------|--|------------------------------|
| 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

<sup>1</sup>A 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 vertices<sup>2</sup>
- 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:
- 
- $\text{SINGLE\_PAIR\_REACHABILITY}(G, s, t)$ : is there a path in  $G$  from  $s \in V$  to  $t \in V$ ?
  - $\text{SINGLE\_PAIR\_SHORTEST\_PATH}(G, s, t)$ : return distance  $\delta(s, t)$ , and a shortest path in  $G = (V, E)$  from  $s \in V$  to  $t \in V$
  - $\text{SINGLE\_SOURCE\_SHORTEST\_PATHS}(G, s)$ : 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  $s$ )

---

<sup>2</sup>A 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  $\infty$ )
- **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  $\delta$  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  $\leq 1$  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

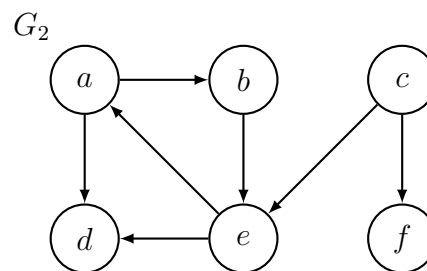
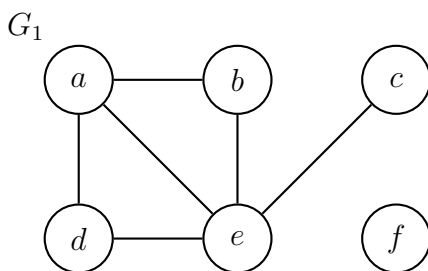
For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>

## 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
  - $\text{Single\_Pair\_Reachability}(G, s, t)$
  - $\text{Single\_Source\_Reachability}(G, s)$
  - $\text{Single\_Pair\_Shortest\_Path}(G, s, t)$
  - $\text{Single\_Source\_Shortest\_Paths}(G, s)$  (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

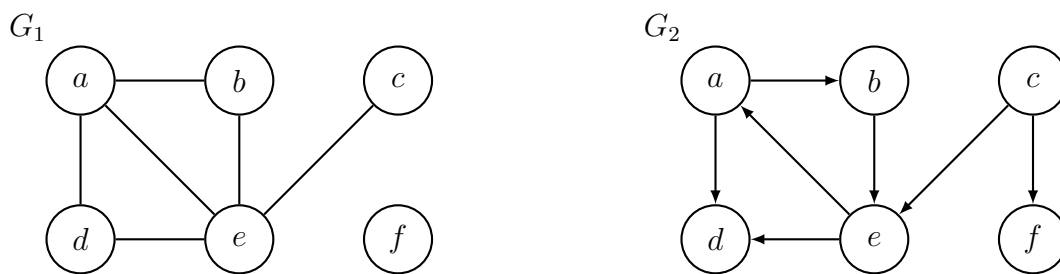
- **Claim:** DFS visits  $v$  and correctly sets  $P(v)$  for every vertex  $v$  reachable from  $s$
- **Proof:** induct on  $k$ , for claim on only vertices within distance  $k$  from  $s$ 
  - Base case ( $k = 0$ ):  $P(s)$  is set correctly for  $s$  and  $s$  is visited
  - Inductive step: Consider 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)
- $\text{Connectivity}(G)$ : is undirected graph  $G$  connected?
- $\text{Connected\_Components}(G)$ : 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>

## Lecture 11: Weighted Shortest Paths

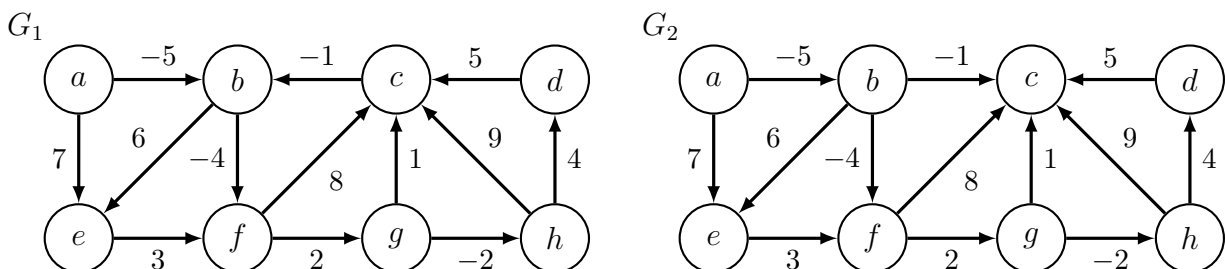
### Review

- 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  $\pi$  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  $\pi$  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  $\infty$ ) 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  $\infty$ )  $\forall v \in V$
- **Proof:** Assume  $d(s, v')$  is weight of a path (or  $\infty$ ) 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>

## 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  $\pi$  be a shortest path with fewest vertices
  - $\pi$  not simple, so exists cycle  $C$  in  $\pi$ ;  $C$  has non-negative weight (or else  $\delta(s, v) = -\infty$ )
  - Removing  $C$  from  $\pi$  forms path  $\pi'$  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

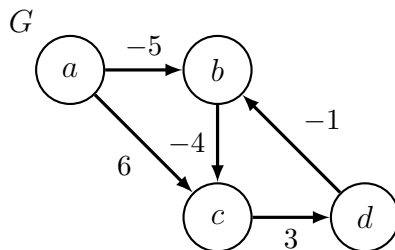
## Negative Cycle Witness

- **$k$ -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

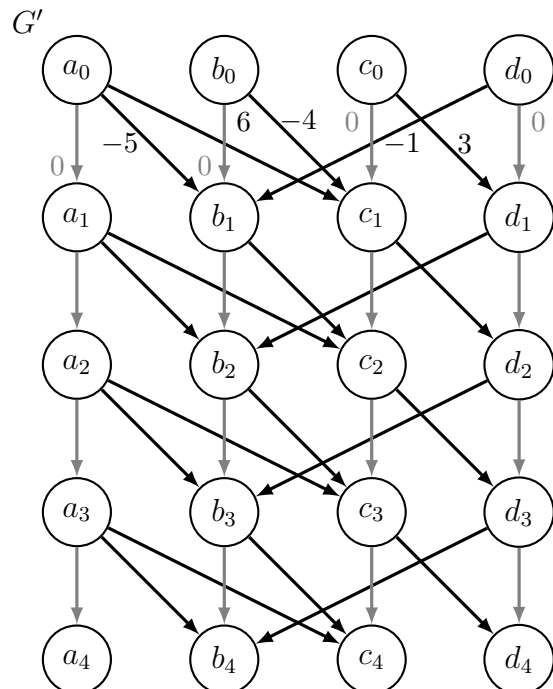
- **Idea!** Use **graph duplication**: make multiple copies (or levels) of the graph
  - $|V| + 1$  levels: vertex  $v_k$  in level  $k$  represents reaching vertex  $v$  from  $s$  using  $\leq k$  edges
  - If edges only increase in level, resulting graph is a DAG!
- 
- Construct new DAG  $G' = (V', E')$  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   | $\infty$  | $\infty$  | $\infty$  |
| 1               | 0   | -5        | 6         | $\infty$  |
| 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  $k = k'$ 

$$\begin{aligned} \delta(s_0, v_{k'}) &= \min\{\delta(s_0, u_{k'-1}) + w(u_{k'-1}, v_{k'}) \mid u_{k'-1} \in \text{Adj}^-(v_{k'})\} \\ &= \min\{\{\delta(s_0, u_{k'-1}) + w(u, v) \mid u \in \text{Adj}^-(v)\} \cup \{\delta(s_0, v_{k'-1})\}\} \\ &= \min\{\{\delta_{k'-1}(s, u) + w(u, v) \mid u \in \text{Adj}^-(v)\} \cup \{\delta_{k'-1}(s, v)\}\} \quad (\text{by induction}) \\ &= \delta_{k'}(s, v) \end{aligned} \quad \square$$
- **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  $\square$

## 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  $\pi$  for any witness  $v$  contains a negative-weight cycle in  $G$
- **Proof:** Since  $\pi$  contains  $|V| + 1$  vertices, must contain at least one cycle  $C$  in  $G$ 
  - $C$  has negative weight (otherwise, remove  $C$  to make path  $\pi'$  with fewer vertices and  $w(\pi') \leq w(\pi)$ , contradicting witness  $v$ )  $\square$
- 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>

## 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:

|                                    |   |
|------------------------------------|---|
| <code>Q.build(X)</code>            | initialize $Q$ with items in iterator $x$           |
| <code>Q.delete_min()</code>        | remove an item with minimum key                     |
| <code>Q.decrease_key(id, k)</code> | find stored item with ID $id$ and change key to $k$ |

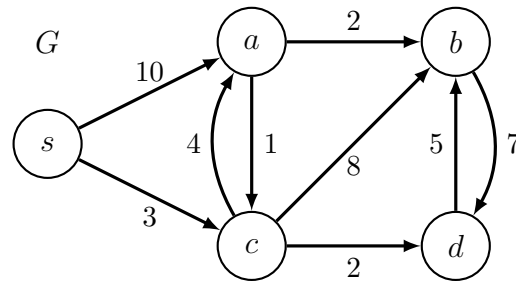
- Implement by **cross-linking** a Priority Queue  $Q'$  and a Dictionary  $D$  mapping IDs into  $Q'$
- Assume vertex IDs are integers from 0 to  $|V| - 1$  so can use a direct access array for  $D$
- For brevity, say item  $x$  is the tuple  $(x.id, x.key)$

- 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<br>$v$ from $Q$ | $d(s, v)$ |          |          |          |          |
|------------------------|-----------|----------|----------|----------|----------|
|                        | $s$       | $a$      | $b$      | $c$      | $d$      |
| $s$                    | <b>0</b>  | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| $c$                    |           | 10       | $\infty$ | <b>3</b> | $\infty$ |
| $d$                    |           | 7        | 11       |          | <b>5</b> |
| $a$                    |           | <b>7</b> | 10       |          |          |
| $b$                    |           |          | <b>9</b> |          |          |
| $\delta(s, v)$         | 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  $\infty$ )
  - 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  $\pi$  from  $s$  to  $v'$ , with  $w(\pi) = \delta(s, v')$
    - \* Let  $(x, y)$  be the first edge in  $\pi$  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:

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

- \* 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 $Q'$<br>on $n$ items | $Q$ Operations $O(\cdot)$ |                        |                               | Dijkstra $O(\cdot)$<br>$n =  V  = O( E )$ |
|-------------------------------------|---------------------------|------------------------|-------------------------------|---|
|                                     | $\text{build}(X)$         | $\text{delete\_min}()$ | $\text{decrease\_key}(id, k)$ |   |
| 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>

## 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  $\pi$  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>

## 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:  $S(i, j)$  = sorted array on elements of  $A[i : j]$  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:  $F(i)$  = the  $i$ th Fibonacci number  $F_i$  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 subproblems<sup>1</sup>
- 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

<sup>1</sup>This 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

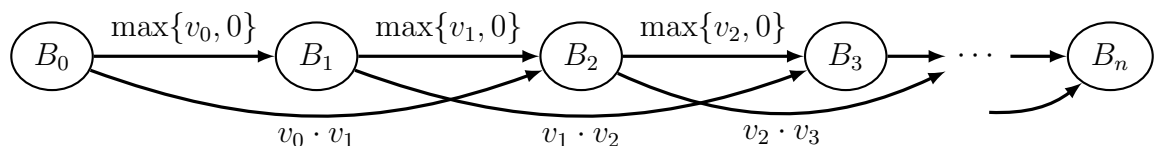
- Given  $n$  pins labeled  $0, 1, \dots, n-1$
- Pin  $i$  has **value**  $v_i$
- Ball of size similar to pin can hit either
  - 1 pin  $i$ , in which case we get  $v_i$  points
  - 2 adjacent pins  $i$  and  $i+1$ , in which case we get  $v_i \cdot v_{i+1}$  points
- Once a pin is hit, it can't be hit 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:  $B(i, j) = \text{maximum score starting with just pins } i, i + 1, \dots, j - 1,$   
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:  $B(i) = \text{maximum score starting with just pins } i, i + 1, \dots, n - 1,$   
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>

## 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 `heglo` or `iello` or `ieglo`, all length 5
- Maximization problem on length of subsequence

### 1. Subproblems

- $x(i, j)$  = length of longest common subsequence of suffixes  $A[i : ]$  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  $j$ ,  $0 \leq i \leq j < n$
- $x(i, j)$  = 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
- $x(i, j, p)$  = maximum total value I can take when player  $p \in \{\text{me}, \text{you}\}$  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>

## 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  $\delta_k(u, v)$  = minimum weight of path from  $u$  to  $v$  using at most  $k$  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

- $d(u, v, k)$  = minimum weight of a path from  $u$  to  $v$  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
- $x(i, j, \text{opt}) = \text{opt} \{x(i, k, \text{opt}') *_{\text{opt}} x(k, j, \text{opt}'') \mid i < k < j; \text{opt}', \text{opt}'' \in \{\min, \max\}\}$

### 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

- $x(i)$  = 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

- $x(i, f)$  = minimum total difficulty for playing notes  $t_i, t_{i+1}, \dots, t_{n-1}$  starting with finger  $f$  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  $S$  = 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>

## 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  $\ell$  for all  $\ell \in \{1, 2, \dots, L\}$
- Goal: Cut the rod to maximize the value of cut rod pieces
- Example:  $L = 7$ ,  $v = [0, 1, 10, 13, 18, 20, 31, 32]$   
 $\ell = \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{matrix}$
- 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  $\ell$
- 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  $\ell$ : Subproblems  $x(\ell)$  depend only on strictly smaller  $\ell$ , 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  $\ell$  and optimal choice is  $\ell'$ , 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     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     l, pieces = L, []
12     while parent[l] is not None:      # walk back through parents
13         piece = l - parent[l]
14         pieces.append(piece)
15         l = parent[l]
16     return pieces

```

## Subset Sum

- Input: Sequence of  $n$  positive integers  $A = \{a_0, a_1, \dots, a_n\}$
- 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

- $x(i, t)$  = does any subset of  $A[i : ]$  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  $t - a_i$  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  $i$

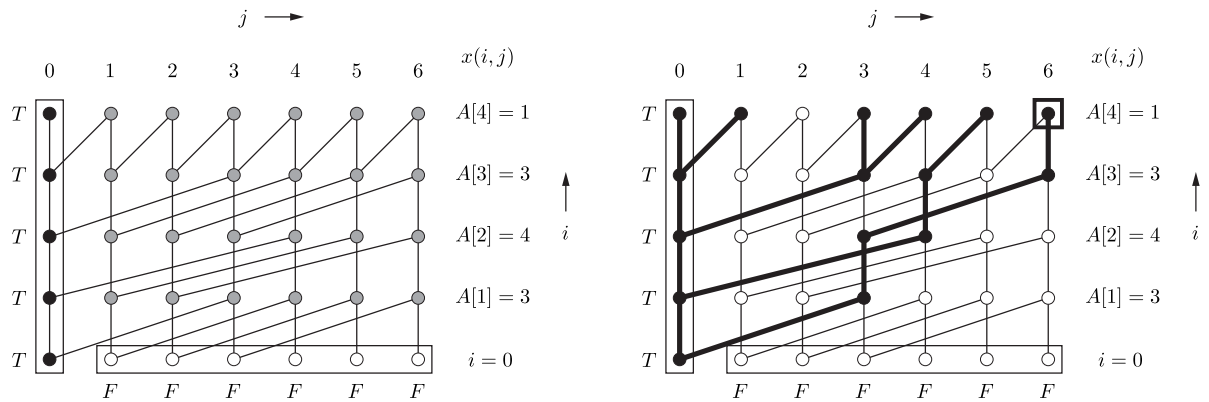
### 4. Base

- $x(i, 0) = \text{YES}$  for  $i \in \{0, \dots, n\}$  (space packed exactly!)
- $x(0, t) = \text{NO}$  for  $t \in \{1, \dots, T\}$  (no more items available to pack)

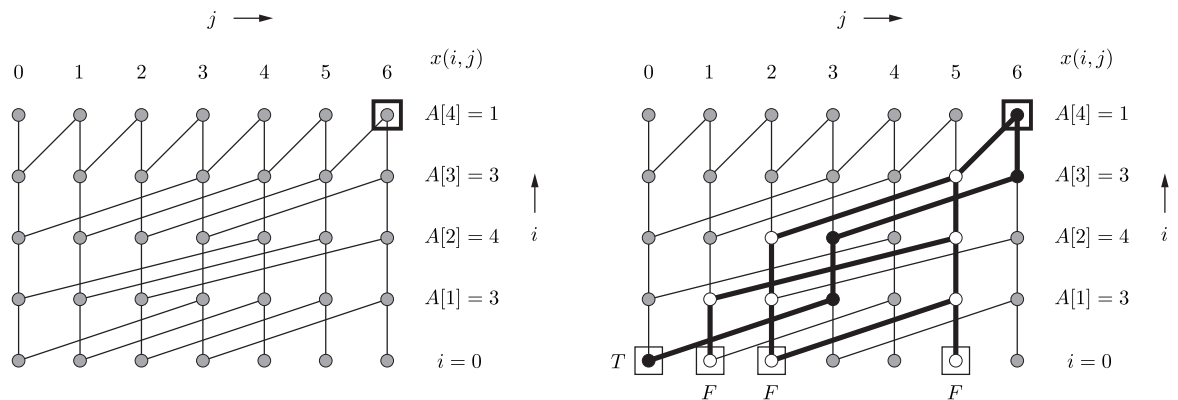
### 5. Original problem

- 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  $w = \lg(n + 1)$ , 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  $L$  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>

## 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 <b>simple</b> 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

|            |   |   |
|------------|---|---|
| <b>R</b>   | problems decidable in finite time                     | (‘R’ comes from recursive languages)            |
| <b>EXP</b> | problems decidable in exponential time $2^{n^{O(1)}}$ | (most problems we think of are here)            |
| <b>P</b>   | 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 <b>simple</b> 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                            |

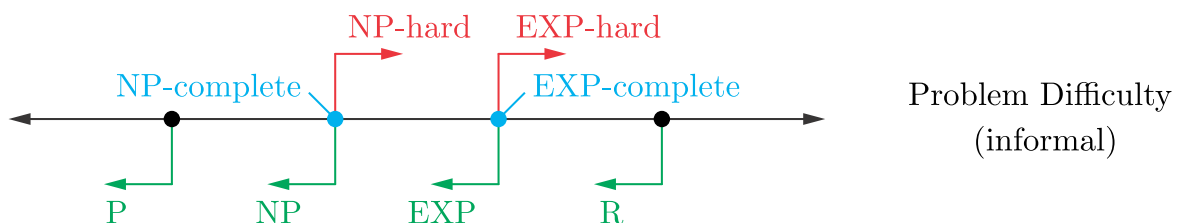
- **P**  $\subseteq$  **NP**: The verifier  $V$  just solves the instance ignoring any certificate
- **NP**  $\subseteq$  **EXP**: Try all possible certificates! At most  $2^{n^{O(1)}}$  of them, run verifier  $V$  on all
- **Open**: Does **P** = **NP**? **NP** = **EXP**?
- Most people think **P**  $\subsetneq$  **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**  $\neq$  **NP**
- How do we relate difficulty of problems? Reductions!

## Reductions

- Suppose you want to solve problem  $A$
- One way to solve is to convert  $A$  into a problem  $B$  you know how to solve
- Solve using an algorithm for  $B$  and use it to compute solution to  $A$
- This is called a **reduction** from problem  $A$  to problem  $B$  ( $A \rightarrow B$ )
- 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  $X \in \mathbf{NP}$ )
- **NP-complete** =  $\mathbf{NP} \cap \mathbf{NP-hard}$
- 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  $\mathbf{NP} \setminus \mathbf{P}$ , these are
- Chess is **EXP-complete**: in **EXP** and reducible from every problem in **EXP** (so  $\notin \mathbf{P}$ )



## 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>

## 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>