CS2040 Summary Notes

Author: Lim Wei Liang

Lecturer/Semester Taken: Dr Chong Ket Fah, AY20/21 Special Term 2

Main References:

- Class Lecture Slides & Lab Slides

VisuAlgo - visualising data structures and algorithms through animation

VisuAlgo was conceptualised in 2011 by Dr Steven Halim as a tool to help his students better understand data structures and algorithms, by allowing them to learn the basics on their own and at their own pace.

https://visualgo.net/en

0. Prerequisites

Powers

- $a^x \times a^y = a^{x+y}$
- $a^x \div a^y = a^{x-y}$
- $(a^x)^y = a^{xy}$
- $(ab)^x = a^x b^x$

Logarithms (inverse of taking power)

- $a^b = y \Leftrightarrow log_a y = b$
- $log_a(a^x) = x$
- $a^{log_a(x)} = x$
- $log_a(m \times n) = log_a m + log_a n$
- $log_a(m/n) = log_a m - log_a n$

- $(\frac{a}{b})^x = \frac{a^x}{b^x}$

- $log_a(m^r) = rlog_a m$
- Change of Base: $log_a x = \frac{log_b x}{log_b a}$
- $a^{log_c b} = b^{log_c a}$ (derived using above)
- $log_b a = 1/log_a b$
- $log_b(1) = 0$
- $log_b b = 1$

Useful Series

- **AP Summation Formula:** $\frac{n}{2}(2a + (n-1)d)$ **or** $\frac{n}{2}(a_1 + a_n)$
- **GP Summation Formula:** $S_n = \frac{a(1-r^n)}{1-r}$ or $S_n = \frac{a(r^n-1)}{r-1}$
  - **GP Infinite Series Formula (for |r| < 1):** $S_\infty = \frac{a}{1-r}$
- **Harmonic Series:** $\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n} \approx ln(i+1)$
- **Sum of Squares:** $\sum_{i=1}^{n} i^2 = 1 + 4 + 9 + ... + n^2 = \frac{n(n+1)(2n+1)}{6}$

Describing Algorithms

- Declare (and if necessary describe) all important data structures and variables at the start
- Be concise
- No need to describe details of data structures/algorithms discussed in class, unless they are modified
- No "black boxes"
- Use the correct terminology
- Be clear in your descriptions

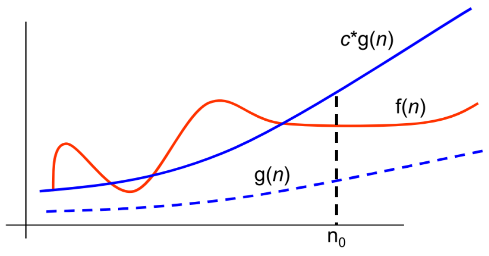
1. Introduction - Analysis of Algorithms

Theoretical basis for measuring and comparing the efficiency of algorithms

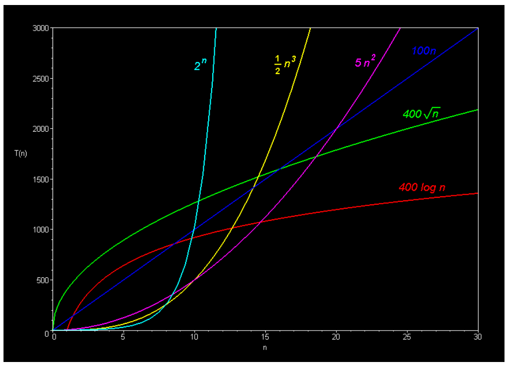
- **An algorithm** is a step-by-step procedure for solving a problem. Properties:
  - Every step must be **exact**
  - It must **terminate**
  - It must be **effective**
  - As much as possible, it should also be **general** (exceptions - maybe in very hard problems)
- Algorithms are **abstract** & implemented using code
- When solving problems, we can write in pseudocode first and ignore language-specific details
- **Analysis of Algorithms** involves comparing the efficiency/complexity of solution methods (i.e. abstract algorithms, and not implemented programs)
  - We want to represent the resources required (e.g. time, space, bandwidth) mathematically

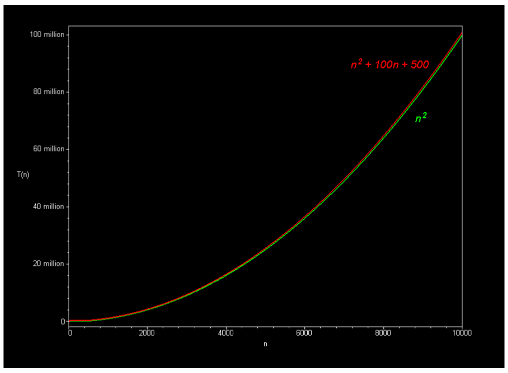
  - This module focuses more on the time complexity of algorithms
  - We also want our efficiency analysis to be **independent** of programming languages, specific implementations, computer used, data (not always possible)
    - thus, cannot use actual run-time
  - Instead, we can count the number of primitive operations the algorithm makes (directly related to runtime)
    - Further simplify by ignoring differences between operation types, and number of operations in a statement (so long as it is constant) → just count number of statements executed
- **Further Simplification - Asymptotic Analysis**
  - Focuses on the algorithm's **growth rate** (how number of operations executed grows as input increases in size) for large input sizes
    - because, modern computers generally won't have issues for small input sizes regardless of algorithm
    - enables comparisons between algorithms
  - Consider only the dominating term, and ignore its coefficients

- **Big-O notation (e.g. O(n))**
  - **Definition:** Given a function f(n), we say that g(n) is an asymptotic upper bound of f(n), denoted f(n) = O(g(n)), if there exists a constant **c** > 0, and a positive integer n_0 such that f(n) ≤ **c***g(n) for all n ≥ n_0
    - f(n) is said to be bounded from above by g(n)
    - note that c and n_0 are not unique



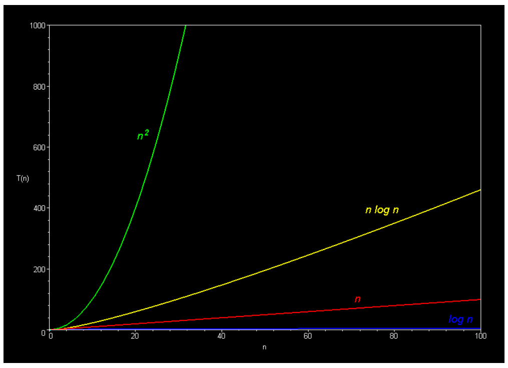
  - Complexity of an algorithm can be bounded by many functions, but we want to find the **tightest bound**.
  - From the definition, we derive that we can omit all coefficients of all terms in a formula, as well as non-dominating terms → simplify formula to **single term with coefficient 1.**
    - This is called the **growth term/rate of growth/order of growth/order-of-magnitude (or even function families?)**
    - Note that O(f(n)) + O(g(n)) = O(f(n) + g(n))





  - **Common Growth Terms:**
    - Constant O(1) ← "fastest"/slowest growth rate
    - Logarithmic O(logn) ←doesn't matter which log (derive from change of base formula), this represents base 2.

        - Square-Root O(n**0.5)
        - Linear O(n)
        - Linearithmic, Loglinear, or quasilinear O(nlogn)
        - Quadratic O(n**2)
        - Cubic O(n**3) ← this and smaller terms are polynomial
        - Exponential O(k**n)
        - Factorial O(n!) ← "slowest"/fastest growth rate



- **Finding Complexity of Programs**
    - Basically count the number of statements executed
    - Things to take note of: Sequencing (e.g. Lists, Loops), Nesting (remember to multiply!), Composition (e.g. f(g(x)), add up the Big-Os)
    - E.g. In a loop with a range of values n, and each iteration reduces the range by a fixed constant fraction (e.g. 1/2) → O(log n)
    - O(log(n!)) is equal to O(n log(n)), (proof shows that they are upperbounds for each other)
- **Example 1A - Sequential Search** (unsorted data)
    - Worst Case/Average Case/Unsuccessful Search: O(n)
    - Best Case: O(1)
- **Example 1B - Non-recursive Binary Search** (sorted data)
    - **Pseudocode:**
        - Given an array sorted in **ascending** order
        - Maintain a subarray where x (the search key) might be located
        - Repeatedly compare x with m, the middle element of the current subarray
            - If x = m, found it!
            - If x > m, continue search in subarray after m
            - if x < m, continue search in subarray before m
    - Worst Case will be O(log n), because we are halving the size of the list each time.
- **Example 2 - Recursive nth Fibonacci Number**
    - Form a recurrence relation and simplify → $O(2^n)$
- **Analysis of Different Cases**
    - **Worst-Case** Analysis (i.e. maximum amount of time required), **the focus of CS2040**
    - **Best-Case** Analysis ← not so useful
    - **Average-Case** Analysis (need to know probability distribution of inputs to know the 'average' input)

        - **Expected-Case** Analysis, for randomized algorithms
        - **Amortized Analysis** (when worst-case behaviour not possible for every run, hence we determine time complexity required for a sequence of runs and calculate the 'amortized' cost per run)
- **The Big Idea (Summary)**
    - Each function family grows much faster than the one before it!
    - And: on modern computers, any function family is usually efficient enough on small n, so we only care about large n (as in order of magnitude analysis)
    - So... Constants do not matter nearly as much as function families
    - Practically...
        - Do not prematurely or overly optimize your code
        - Instead: **think algorithmically!!!**
    - Also, consider trade-offs between time and memory requirements
- **Java API Tips**
    - Use **Buffered I/O** when required because **Scanner** and **System.out** can be quite **slow**!
    - Be mindful when using Wrapper classes
    - Use the mutable StringBuilder/StringBuffer (synchronized) classes instead of the immutable String class.

```
import java.io.*;

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

// Remember to flush() or close() this before exiting the program!
PrintWriter pw = new PrintWriter(new BufferedWriter(new OutputStreamWriter(System.out)));
```

2. Sorting

Means of organizing data that simplifies many problems, such as searching, checking for duplicates, min, max, set intersection & union etc.

- A wide variety of interesting algorithmic solutions exist, and these embody many ideas:
    - Internal vs External sort
    - Iterative vs Recursive
    - Comparison vs non-Comparison based
    - Divide-and-conquer
    - Best/worst/average case time complexity bounds
    - Randomized algorithms
- **Sort Key -** an attribute that you sort by (relevant when sorting objects with multiple attributes)

2.1. Basic Sorting Algorithms

*The following three sorting algorithms are **comparison-based** and **iterative**:*

Selection Sort

- **General Idea (given an array of n items)**
    1. Find the largest item (could be smallest as well)
    2. Swap it with the item at the end of the array
    3. Repeat steps 1 and 2 with a sub-array excluding the last item (that is guaranteed to be in order), until the entire array is 'processed'
- **Analysis**
    - n - 1 passes are required, while within each pass the searching for the maximum element takes n-1, n-2, ... 1 comparisons as the 'live' subarray shrinks.
    - No means of early termination
    - Hence, run-time is **always** $O(n^2)$

Bubble Sort

- **General Idea**
    1. In each pass through the array, swap values that are not in the correct order. (think a sliding window of size two that slides down the array) Largest element will definitely be "bubbled" to the very end in each pass.
    2. Shrink the subarray by one (taking out the largest element at the end) and repeat.
- **Analysis**
    - n-1 passes are required, within each pass number of comparisons is n-1, n-2, ... , 1
    - Hence, always takes $O(n^2)$ time complexity.
    - Can be improved by adding a **flag,** checking if the array is sorted on each pass.
        - **Worst-case** time complexity remains $O(n^2)$, when smallest element is on the rightmost position
        - **Best-case** time complexity becomes $O(n)$, taking only one pass through the array. (input must be sorted already)

Insertion Sort

- **General Idea**
    - Maintain a sorted subarray starting with the leftmost element.
    - In each iteration, pick the next element from the adjacent unsorted subarray and insert it in it's correct place in the sorted subarray by comparing and shifting elements aside.
    - Terminates once all elements are in the sorted subarray.
- **Analysis**
    - Outer loop always executes n-1 times
    - Best-case (sorted data) - inner loop doesn't need to execute, so O(n)
    - Worst-case (reversely sorted data) - always need to shift elements all the way to insert next element at the very front of the array (1 + 2 + 3 + ... + n-1) - so $O(n^2)$

2.2. Divide-and-Conquer Sorting Algorithms

*Merge Sort and Quick Sort are also **comparison-based** and **recursive** (although iterative versions exist!)*

Divide-and-Conquer Algorithms (break a big problem down into smaller, solvable subproblems)

- **Divide Step:** divide the larger problem into smaller problems
- (recursively) solve the smaller problems
- **Conquer Step:** combine the results of the smaller problems to produce the result of the larger problem

Merge Sort

- **General Idea**
  - **Divide Step:** Divide the array into two equal halves
  - (recursively) **Merge Sort** the two halves, where the base case is an array of length 1
  - **Conquer Step: Merge** the two sorted halves to form a sorted array
- **Analysis**
  - **Merge Step -** takes O(k) time, where k is the length of the subarray (roughly 3k - 1 comparisons and copies). Bulk of computation happens here!
  - Each level in the recursive tree takes O(n) time, and there are logn levels ⇒ run-time is **O(nlogn) always**
  - Requires **additional space complexity of O(n)** for temp array for the merge step

Quick Sort

- **General Idea**
  - **Divide Step:** Choose a pivot item p and partition the array into two parts where one part contains all items < p, and the other part contains everything ≥ p. (s1, p, s2)
    - note: our version chooses the first element as the pivot, which might not be ideal
  - Recursively sort the 2 parts, where base case is once again an array of length 1
  - **Conquer Step:** nothing!
- **Analysis**
  - **Partition Step -** bulk of computation is done here! Takes O(k) time, where k is the length of the subarray
  - **Worst-case -** when the array is already sorted (either way), or has entirely identical elements, our divide step is very lopsided - s1 is empty and s2 has n - 1 elements. Recursive tree thus has n levels and the algorithm takes $O(n^2)$ time. (n + n-1 + ... 1) time for each level
    - one possible way to deal with many identical elements: a 3rd partition for ==?
  - **Best-case -** when partition always splits the array into two equal halves ⇒ depth of recursion is logn + each level takes ≤ n time, so time complexity is O(nlogn)
    - In practice, average time is O(nlogn), especially if using **randomized pivoting**, or even the median as pivot.

    - Technically**,** the space complexity is O(logn) because of the nested recursive calls, but this is miniscule so we still consider it more or less in-place.
- Variation of this for **selecting the kth smallest element** is called **Quickselect** - runs in O(n) average time

2.3. Non Comparison-Based Sorting Algorithms

All methods before this were **comparison-based,** so they had a best-possible theoretical worst-case time complexity of **O(nlogn)**. However, if we can make certain assumptions and avoid using **comparisons** when sorting, we can achieve a better time-complexity.

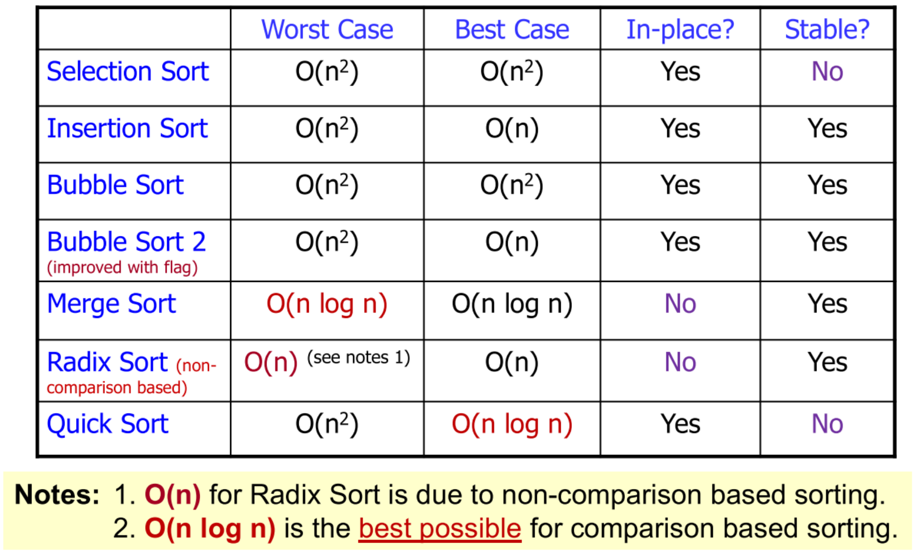
- e.g. **Counting Sort**, where given data within a very limited range, we could simply count the frequency of each piece of data appearing (O(n)), then print out the output in sorted order (O(n+k) for integers, if a range of k integers appear). Only viable if k is small.

Radix Sort

- Treats data to be sorted as a character string.
- Can only be used if input can be converted to **finite character strings of equal length** (e.g. no irrationals)
- **General Idea**: in each iteration, group the data using ordered buckets based on a character in each item (going from least to most significant character)
    - bucket is FIFO, to maintain relative order from previous groupings
- **Analysis:** O(d x (n+k)) where d is the maximum length of the character string & k is the number of buckets. **If** d & k are fixed/bounded, complexity is O(n).
    - **Warning:** If size of n can increase, this takes O(lgn x n) = O(nlogn) time, since # of digits increases at a rate of lgn.

2.4. Summary

- **In-place sort -** only requires a constant O(1) extra space while sorting
- **Stable sort -** relative order of elements with the same key value is preserved by the algorithm
    - to make quick sort stable, might have to sacrifice time complexity.
- For sorting, sort from characteristic of **least priority**→ **most priority** (e.g. ones to thousands place)



	Worst Case	Best Case	In-place?	Stable?
Selection Sort	O(n^2)	O(n^2)	Yes	No
Insertion Sort	O(n^2)	O(n)	Yes	Yes
Bubble Sort	O(n^2)	O(n^2)	Yes	Yes
Bubble Sort 2 (improved with flag)	O(n^2)	O(n)	Yes	Yes
Merge Sort	O(n log n)	O(n log n)	No	Yes
Radix Sort (non-comparison based)	O(n) (see notes 1)	O(n)	No	Yes
Quick Sort	O(n^2)	O(n log n)	Yes	No

Notes: 1. **O(n)** for Radix Sort is due to non-comparison based sorting.
2. **O(n log n)** is the best possible for comparison based sorting.

- **Java API tips:**
  - Observe that for **utility classes** in Java -> ends with 's'. e.g. Arrays, Objects, Collections contain contain helpful static utility methods we can use
  - Arrays.sort(arr) uses **double-pivot quicksort** for primitive arrays, and **TimSort** for arrays with reference types. (TimSort is stable but not in-place, with best-case O(n) and worst-case O(nlogn))
  - Collections.sort(list) also uses TimSort
  - Objects can implement the Comparable<T> interface for a natural ordering (Strings, Wrappers and Primitives already do so)
  - Use and pass Comparator<T> functional interfaces to define custom sort functions

3. Linear Abstract Data Types (ADTs)

- An **Abstract Data Type** represents a collection of data + a specification of a set of operations on the data, and does not involve any implementation details (e.g. how to store the data, how to implement the operations)
- **Data Structures** are constructs defined within a programming language to **store and organize** a collection of data (e.g. Arrays, Fraction, Integer) - a **concretized/implemented ADT**
  - hide implementation details from the client
  - generally, they are meant to support **efficient** insertions, searches, deletions, queries and/or updates
  - usually have certain **properties** which must be **maintained** by all operations of the data structure (e.g. Binary Heap property)
- In Java, we define an ADT using interfaces. (specifies methods + (possibly) constants)

3.1. Lists (via Arrays and Linked Lists)

Data type for storing and managing a list of objects. Generally, we try to only store items of the same type in one list.

- A dynamic (size not fixed), linear ADT where data is stored one after the other.
- Usually zero-indexed, i.e. items are given indexes starting from 0 all the way to n-1, where n is the number of items in the list (i.e. the length of the list)
- Two possible implementations with varying advantages and disadvantages:

1. **Using Fixed-Size (static) Arrays**
    - Occupies a contiguous block of memory
    - **Retrieval -** trivial because we are using an array, simply do arr[i]
    - **Insertion -** shift objects from i+1 onward to the right to make a gap, and insert your item into the gap
        - if the array is already full, make a new one of double the size, copy the elements over and proceed as per normal
    - **Deletion -** shift items from i+1 onward to the left to cover up the item
    - Also other operations like empty(), size(), contains(), indexOf() etc. (pretty obvious how to implement)
    - **Analysis**
        - **Retrieval - O(1) time** since entire array is loaded into memory and items can be retrieved using their index
        - **Insertion**
            - **Best-Case: O(1)** if adding to back and no need to enlarge array
            - **Worst-Case: O(n)** if adding to the front since you have to shift everything to the right OR if the array needs to be enlarged (due to copying and creation of new array)
                - Note: **Amortized Time Complexity of enlarging array is O(1)** since it does not always take place **(doubling strategy)**
            - **Average-Case: O(n)** if inserting at the midpoint
        - **Deletion**
            - **Best-case: O(1)** if deleting from the back
            - **Worst-case: O(n**) if removing from the front since everything has to be shifted to the left
            - **Average-Case: O(n)** if deleting from the middle
        - **Space Complexity -** O(n). worst-case we use 2n space for n+1 items
2. **Using Linked Lists**
    - Idea: each item is stored in a **node,** which contains a 'next' reference to the node on its right. We then only need to store a reference to the **head** node and the number of nodes.
    - The elements will thus occupy non-contiguous memory
    - More book keeping needed! Also, need to be careful when updating references to avoid accidental garbage collection.
    - Special cases also need to be considered.

- Drawings especially useful for understanding and visualizing a Linked List's (and its variants) operations
- **Retrieval -** start at the head and iterate through the 'next' nodes
- **Insertion at index i**
    - make a new node containing the item, access the node at i-1, update references and increment number of nodes
    - special case if adding to the front of the list (adjust head reference)
- **Deletion**
    - update reference of previous node to the neighbour of neighbour of the node you want to delete → the node has no more reference pointing to it and gets automatically garbage collected, then decrement number of nodes
    - special case if removing from the front of the list (adjust head reference)
- **Analysis**
    - **Retrieval**
        - best case **O(1)** for first node
        - average and worst cases **O(n)** because you need to iterate over the nodes
    - **Insertion**
        - **Best Case O(1)** when adding to the front
        - **Worst Case & Average Case - O(n)** because you need to get to the node at index i-1, insertion itself takes O(1) time (just need to adjust references)
    - **Deletion**
        - **Best Case O(1)** when removing from the front
        - **Worst Case & Average Case - O(n)** because you need to get to the node at index i-1, deletion itself takes O(1) time (just need to adjust references)
    - **Space Complexity is O(n)** - just constant overhead needed to store the n nodes

3. **When to use which implementation?**
    - Add/remove from the front → use Linked List
    - Add/remove from the back → use Array (can even make a big enough array from the start if we know what we need)
    - If add/remove from random indexes → both are the same
    - If add/remove from a particular index → use Linked Lists since we can maintain a reference to the node at i-1 to speed up insertion/deletion to O(1) time.
    - Lots of accessing but little modifying → use an Array
4. **More Variants of Linked List**
    - **Note**: Extra attributes = **extra housekeeping required** (more things to consider in the code)
    - **Tailed Linked List**

    - adds a tail reference to the last node in the list so that we can access + add to the last node quickly in O(1) time
        - note: removing from the tail still takes O(n) time because the 2nd last node is needed
  - **Circular Linked List**
    - same as tailed linked list, but last node is also linked back to the first node, which allows for repeated cycling through the list
  - **Doubly Linked List**
    - same as tailed linked list, except that nodes have both a 'next' pointer as well as a 'prev' pointer
    - reduces time taken for list operations (start at the closer end of the list), but doesn't change big-O

5. **Java API Tips**
  - **ArrayList<T>** (and it's synchronized counterpart Vector<T>) & **LinkedList<T>** (a doubly linked list with tail reference) both implement the **List<T>** interface
  - Using List::of and List::copyOf both create an immutable List

3.2. Stacks

*Collection of data accessed in a **last-in-first-out manner (LIFO)***

- **Operations**: empty(), push(item), peek(), pop()
- **Implementation**
  - using an array, where the back of the array is the top of the stack (for faster insertion/deletion). Maintain a variable 'top' to store the index of the top of the stack.
    - **Analysis -** O(1) worst-case for peek() and pop(), O(1) amortized for push()
  - using a basic linked list, where the front of the list functions as the top of the stack
    - O(1) worst-case for all operations
- **In Java:** java.util.Stack<E>
- **Possible Applications**:
  - storing a function call stack (e.g. recursion's nested function calls)
  - matching parentheses
  - evaluating arithmetic expressions (e.g. evaluating unique postfix expressions, converting infix → postfix)
  - maze traversal etc.
  - **Note:** reverses the order of whatever is passed into it

3.3. Queues

*Collection of data accessed in a **first-in-first-out manner (FIFO)***

- **Operations:** empty(), offer(item)/enqueue(item), peek(), poll()/dequeue()

- **Possible Applications:**
  - print queues, simulations (event queue), breadth-first graph traversal
  - **note:** maintains the order of whatever is passed into it
- **Implementation**
  - using an array with front and back indices and 'circular' indexing
    - O(1) worst-case for empty(), peek(), and poll(), O(1) amortized for offer()
  - using a tailed linked list
    - O(1) worst-case for all operations
- **In Java:**
  - only a Queue<T> interface exists, but LinkedList<T> implements Queue<T>
  - Java also has a Deque<T> interface (Double-ended Queue), allows for queue operations on both ends.

3.4. Maps (via Hash Tables)

Map is a data type for storing a collection of <key, value> pairs.

- Keys must be unique, and can map to values in a one-to-one or many-to-one relation
- **Basic Operations**
  - **Retrieve** a value using a key
  - **Inserting**/**Replacing** a value using a key
  - **Deleting** a <Key, Value> pair using a key

	Sorted List (Array impl. By sorting key)	**Balanced BST**	**HashTable**
Insertion	O(n)	O(log n)	O(1) avg
Deletion	O(n)	O(log n)	O(1) avg
Retrieval	O(log n)	O(log n)	O(1) avg

- Implement using a hash table for maximum efficiency

Hash Tables

- A **hash table** is a data structure that stores <key, value> pairs, by using a hash function to determine where to store the data. This allows for efficient retrieval and modification later on.
- Hashing the key using our **hash function** efficiently gives the location in the array where the value is stored.
  - **The hash function** converts large integers to smaller integers, and non-integer keys to integers that can be used as indexes of the array
- Since a hash function can be many-to-one, **collisions** can occur when two keys have the same hash value and are assigned the same slot in the hash table.
  - this is also why we must store the key alongside the value internally
- Simplified analogue would be a **direct addressing table**, where the keys are used directly as the indexes for the array storing the keys and values
  - **Drawbacks:** keys must be non-negative integer values, with a small and dense range of keys due to memory concerns

Hash Functions

- **What makes a good hash function?**
  - Simple and fast to compute
  - Evenly distributes keys throughout the hash table → less collisions → space used more efficiently
  - Doesn't require a lot of table slots to work (i.e. has a large output range)
- Generally, we want to make use of all the data in the key to avoid uneven distributions and patterns
  - e.g. for Strings, make use of both character position AND all characters
- **Perfect Hash Functions** are a one-to-one mapping between keys and hash values, resulting in zero collisions
  - only possible if all keys are known beforehand e.g. reserved keywords in a language, built-in commands
  - considered **minimal** if the table size is the same as the number of keywords supplied
- **Uniform Hash Functions** distribute keys evenly in the hash table
- **Common Hash Functions**
  - Modulo operator
    - $hash(k) = k \% m$ **. Output range is from 0 to m-1 (m slots total)**
    - Warning: Java % is not a true modulo operator like in Python, be careful when using on negative numbers (e.g. -1 % 3 = -1 in Java, but -1 % 3 = 2 in Python)
    - The most popular!
    - Values in key range evenly distributed among hash values + all the key values mapped to each hash value are also evenly distributed in the key range
  - Multiplication method
    - $hash(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$, where 0 < A < 1, m is the hash table size
    - Reciprocal of the golden ratio is a recommended choice for A $(\sqrt{5} - 1)/2 \approx 0.618033$
    - Not that popular because more complicated and more primitive operations are needed
- **Choosing m, the hash table size**
  - Don't choose 2^n or 10^n, same as extracting last n bits/digits of the keys
  - Rule of thumb: Pick a prime number close to a power of two
    - minimal factors → almost always uses most indices in the hash table e.g. if we had 2n % 8 we would only use the even integers
    - because we generally double when resizing

Resolving Collisions

- Collision is **more likely** than one might think - when inserting 23 keys into a table with 365 slots, collisions occur more than half of the time (analogous to the Birthday Paradox)
- $Load\,Factor = \alpha = n/m$, where n is the number of keys in the hash table, and m is the size of the hash table. A measure of how full the hash table is.

- **Separate Chaining Technique (Closed Addressing)**
    - commonly used to resolve collisions!
    - Use a (doubly) linked list to store collided keys (a (sorted) array is possible too)
    - not cache friendly as it doesn't occupy a contiguous block in memory (computer does something called memory caching for faster retrieval of adjacent data)
    - α is the average length of the linked lists, map operations take $O(\alpha)$ time
    - To keep α bounded at a constant → reconstruct the entire hash table when it gets too big
        - **Rehash** all keys into a bigger table about double the size (amortized cost O(1), worst-case O(m))

- **Open-Addressing Techniques**
    - **Linear Probing + Modified Linear Probing**
        - Probe with **step-size of 1**
        - **find():** scan through slots until an empty slot/the element is encountered
        - **insert():** first check that the element is not already there by using find(). Then, if a collision occurs during insertion scan through the table for the next earliest empty slot (with wraparound)
        - **delete():** use lazy deletion and only mark the item as deleted without actually removing it to avoid affecting find(). This slot can be replaced by future items inserted.
        - **Issues:**
            - **Primary Clustering,** where many consecutive occupied slots are created, increasing the runtime of map operations.
            - Can be reduced by using **modified linear probing,** where the probing sequence follows **step-size of d**
            - integer d is the **new step size**, and should ideally be **co-prime to m** (i.e. gcd of m & d are 1) so that all slots will be covered
    - **Quadratic Probing**
        - Probing sequence has **step size of k^2**
        - **Theorem of Quadratic Probing:** If $\alpha < 0.5$, and m is prime, **then we can always find an empty slot**
            - Implication: we should always **rehash** to a bigger table once $\alpha \geq 0.5$
        - **Issues: Secondary Clustering** - when two keys have the same initial position their probe sequences are always the

Linear Probe Sequence

- $hash(key)$
- $(hash(key) + 1)\,\%\,m$
- $(hash(key) + 2)\,\%\,m$
- $(hash(key) + 3)\,\%\,m$
- ...

Modified Linear Probing Sequence

- $hash(key)$
- $(hash(key) + d)\,\%\,m$
- $(hash(key) + 2d)\,\%\,m$
- $(hash(key) + 3d)\,\%\,m$
- ...

Quadratic Probing Sequence

- $hash(key)$
- $(hash(key) + 1^2)\,\%\,m$
- $(hash(key) + 2^2)\,\%\,m$
- $(hash(key) + 3^2)\,\%\,m$
- ...
- $(hash(key) + k^2)\,\%\,m$
- **Note:** k^2 is relative to the original index. If step-size between indexes, follows the trend 1,3,5,7,9,...

Double Hashing Probe Sequence

same (not as bad as primary clustering, also occurs for both modified and normal linear probing)

- **Double Hashing**
  - Generally the best open-addressing technique.
  - Uses a **secondary hash function** to give the step size to use if a collision occurs - now, different keys at the same initial position can have different probe steps
    - $hash(key)$
    - $(hash(key) + hash_2(key))\,\%\,m$
    - $(hash(key) + 2hash_2(key))\,\%\,m$
    - $(hash(key) + 3hash_2(key))\,\%\,m$
    - ...
  - **Conditions:**
    - Secondary hash function must not evaluate to zero!!
      - e.g. 5 - (key % 5) is fine, but not key % 5
    - All secondary hash values generated must be co-prime with the size of the hash table
      - so, using a prime number m' < m is okay

- **Criteria for a good collision resolution method**
  - Minimize primary and secondary clustering
  - always find an empty slot if it exists
  - fast
- **Java API**
  - HashMap<K, V> & HashTable<K,V> (is synchronized but slightly slower)
  - Practically, don't *have* to use a prime as the capacity
  - **Warning: only use immutable objects as keys in a HashMap - hash code has to always be the same for this to work properly**
  - When defining your own custom classes, you have to come up with your own hash function. (override the default hashCode() inherited from Object)
    - make use of the Objects::hash helper method
    - Remember to adhere to the general contract of hashCode in objects, e.g. override equals() in a certain way

4. Tree-Based ADTs (non linear)

4.1. Tree Definitions (primarily from CS1231)

- An undirected **graph G = (V,E) consists of:**
  - a set of vertices V = $\{v_1, v_2, ..., v_n\}$ and
  - a set of (undirected) edges E = $\{e_1, e_2, ..., e_k\}$,
  - where (undirected) edge e connecting v_i and v_j is denoted as $e = \{v_i, v_j\}$
- A **tree** is an (undirected)* graph that is **circuit-free** and **connected.** ***at least in the Graph Theory context.

    - E = V - 1 and only one unique path exists between any pair of vertices.

- Let T be a tree. If T has only one or two vertices, then each is called a **terminal vertex (or leaf)**. If T has at least three vertices, then a vertex of degree 1 in T is called a **terminal vertex (or leaf)**, and a vertex of degree greater than 1 in T is called an **internal vertex. (for CS2040/VisuAlgo** → the root is **not an internal vertex)**
- A **rooted tree** is a tree in which there is one vertex that is distinguished from the others and is called the root.
    - The **level of a vertex** is the number of edges along the unique path between it and the root.
    - The **height of a rooted tree** is the maximum level of any vertex of the tree.
    - **Children, Parents, Ancestors, Descendants** (definitions omitted, quite self-explanatory)
- A **binary tree** is a rooted tree in which every parent has at most two children. Each child is designated either a left child or a right child (but not both), and every parent has **at most** one left child and one right child.
    - A **complete binary tree** is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.
    - A **full/perfect binary tree** is a binary tree in which each parent has exactly two children. (i.e. all nodes filled)
    - Given any parent v in a binary tree T, if v has a left child, then the **left subtree** of v is the binary tree whose root is the left child of v, whose vertices consist of the left child of v and all its descendants, and whose edges consist of all those edges of T that connect the vertices of the left subtree.
      The right subtree of v is defined analogously.
- A **spanning tree** for a graph G is a subgraph of G that contains every vertex of G and is a tree. Every connected graph will have a spanning tree.

4.2. Priority Queue (via Binary Heaps)

A data type that takes in elements, and always returns the element with the highest priority. (different from a FIFO queue!)

- **Operations:** enqueue(), dequeue(), empty() and peek()
- If there are >1 elements with the same highest priority, return the first inserted element
- We can define what counts as 'highest priority'
- **Note**: Can't create a valid implementation using Hash Tables because they have no ordering.

Implementation #1 - Circular Arrays

- **Property:** the contents of the array are always in correct order
- Hence enqueue(x) takes O(N) to find and insert at the correct insertion point, while dequeue() takes O(1) to return the front item.

Implementation #2 - Circular Arrays

- **Property:** dequeue() will return the correct item

- Hence enqueue(x) takes O(1) to insert at the back of the queue, while dequeue() takes O(N) to scan through the array and return the first item with the highest priority. (and to close the gap)

Implementation #3 - Binary Heaps (preferred)

- A binary heap is a **complete binary tree**, and is stored as a **1-based array** of size ≥ heapsize.
- **Key Properties:** Complete Binary Tree, (max/min) heap property
- **Navigation Options: (easier for a 1-based array)**
  - parent(i) = floor(i/2), except for i = 1 (root) (integer division)
  - left(i) = 2*i, no left child when left(i) > heapsize
  - right(i) = 2*i + 1, no right child when right(i) > heapsize
- **(Max/Min) Binary Heap Property:** (applies to all nodes except root)
  - A[parent(i)] ≥ A[i] (for max heaps)
  - A[parent(i)] ≤ A[i] (for min heaps)
  - **Implication:** The largest element of the binary max heap will be at the root
- Every subtree within the binary heap will also be a valid binary heap.
- Does not impose a total ordering on its constituent elements, but only a partial ordering.
- **Priority Queue Operations:**
  - **insert(v) (i.e. enqueue(v))**- insert at the back of the array, then **shiftUp** (i.e. repeatedly swap with its parent) the element until the binary heap property is not violated
    - **Analysis:** Time complexity depends on **shiftUp,** takes worst-case **O(logn)** time.
  - **extractMax()/extractMin() (i.e. dequeue()) -** take out and return the root element, swap in the last element in the array to fill in the gap at the front, then **shiftDown** (i.e. repeatedly swap with its child nodes) the last element so that the binary heap property is satisfied.
    - for shiftDown, be careful to pick the correct child node to swap with! (e.g. for max heap swap down with the larger child node, if applicable)
    - **Analysis:** Time complexity depends on **shiftDown,** takes worst-case **O(logn)** time.
  - Hence, the binary heap allows us to efficiently implement a priority queue, where the 'key' used to organize the binary heap represents the priority of each item.
  - Probably doesn't maintain FIFO for items with the **same key/priority** at least for the default binary heap, will need a modified implementation if FIFO is important.
- **Other Operations:**
  - **createHeapSlow(arr)** - for each element in the array, call insert(v) on the binary heap.
    - **Analysis:** $log1 + log2 + log3 + ... + logN = logN! = O(NlogN)$
  - **createHeap(arr)** - use the given array directly as the heap's internal array, then for each node starting from the last internal node to the root node, call **shiftDown(node)**
    - This approach builds bigger and bigger valid binary heaps from the bottom up.
    - **Analysis:**
      - Height of complete binary tree of size N is $\lfloor logN \rfloor = h$.

- Cost to run shiftDown(i) = h_i
- # of nodes at height h of a perfect binary tree (bottommost layer is 0) = $\lceil (\frac{N}{2^{h+1}}) \rceil$
- Cost of createHeap(arr):
    - **note**: final step uses an Arithmetic-Geometric Progression (AGP) summation formula. This occurs when each term is the result of both an arithmetic and geometric progression.

$$\underbrace{\sum_{h=0}^{\lfloor \lg(n) \rfloor}}_{\text{Sum over all levels}} \underbrace{\overbrace{\left\lceil \frac{n}{2^{h+1}} \right\rceil}^{\text{\# of nodes at height h}} \overbrace{O(h)}^{\text{Cost to Heapify a node at height h}}}_{\text{Cost for a level}} = \sum_{h=0}^{\lfloor \lg(n)\rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil c*h = O\left(n\sum_{h=0}^{\lfloor \lg(n)\rfloor} \frac{h}{2^h}\right) = O\left(n\sum_{h=0}^{\infty}\frac{h}{2^h}\right) = O(2\mathbf{N}) = O(\mathbf{N})$$

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} = 2 \qquad \boxed{x = 1/2}$$

- **heapSort(arr) - First, create a binary (max) heap, and then repeatedly call extractMax() N times in order to fill out the sorted array.**
    - Analysis: N x O(logN) runs of extractMax(), so the worst-case time complexity is **O(NlogN)**.
    - Can be made **in-place** if the input array is reused as the binary heap.
    - **Not 'cache-friendly'** because of large jumps in indices (x2) that may not be part of regions of cached memory. (unlike algorithms like QuickSort which increase the indices referenced incrementally)
    - **Not stable** using the regular implementation of a binary heap.
- Final note, faster than using the more complicated bBSTs for this purpose due to the simpler nature of binary heaps.

Java API

- Java has a PriorityQueue<T> class which internally uses a Binary Heap, and is a min heap by default.
- The generic type must either implement Comparable, or a custom Comparator can also be provided.

4.3. Sets (via HashTables and Union-Find Disjoint Sets)

An unordered collection of items with no duplicates (with duplicates it's called a multi-set)

- (Possible) **Operations:** find, insert, remove, union, intersect
- Can be implemented by a hash table if we only need the basic operations (i.e. find, insert and remove), just make the keys and values identical.
- **Java API:** HashSet

Union-Find Disjoint Sets

*An alternative implementation if we need to model **multiple disjoint sets** and also **union** them efficiently.*

- A collection of disjoint sets, with the following **operations:** unionSet(i, j), findSet(item), isSameSet(i, j)

- Each set is modeled as a **tree** (not necessarily binary)**,** and the entire collection of disjoint sets form a **forest.**
- Each set is represented by a single **representative item**, which is the root of the tree of that set.
- **Storing a UFDS**
  - Use an array p (parent), where p[i] records the parent of item i. If p[i] = i, then i is a root and the representative item of the set itis in.
  - At the start, set p[i] = [i] for all elements (represents n disjoint sets of size 1 at the start)
  - If dealing with objects and not just integers, use a hashMap to map the integers to objects.
- **findSet(i)**
  - Recursively visit p[i] until p[i] = i. Then, use **path compression** to speed up future find operations.
  - **Heuristic #1 - Path Compression**
    - **Intuition**: we want to flatten the tree as much as possible because time taken for tree operations is usually proportional to the tree height.
    - set the parent of any nodes passed through along the way to be the **representative item** so that future findSet operations on those nodes only take O(1) time.
- **isSameSet(i, j)**
  - To check if two items belong in the same set, simply check if they have the **same representative item**/same output from findSet(i).
- **unionSet(i, j)**
  - If same set, do nothing. Otherwise, set the representative item of the taller tree to be the new representative item of the combined set. (hence, findSet is needed)
  - **Heuristic #2 - Union-by-Rank:**
    - Maintain an integer array **rank** together with the UFDS**,** where rank[i] stores the **upper bound** of the height of the subtree rooted at i. (maintained only for the representative element, not 100% accurate due to path compression but good enough).
      - Initialize rank[i] = 0 for all elements at the start.
    - This action helps to keep the height of trees small, because if one tree is taller than the other then the combined height will not change when the shorter tree is put under the taller tree. Rank now only increments if two equal-rank trees are unioned.
    - If both trees are equally tall, then the heuristic is not used. Convention in this module is to put the set of i under set of j.
- All operations run in $O(\alpha(N))$ amortized time-complexity if **both heuristics** are used to keep tree heights from growing quickly.
  - $\alpha(N)$ is the **inverse Ackermann** function which grows extremely slowly, and can be assumed to be **constant** or O(1).
- Without both heuristics, operations take about **O(n)** time, using only one they will take about **O(logn)** time.
- That's it for UFDS in this module, as the other operations and actual analysis are fairly complex.

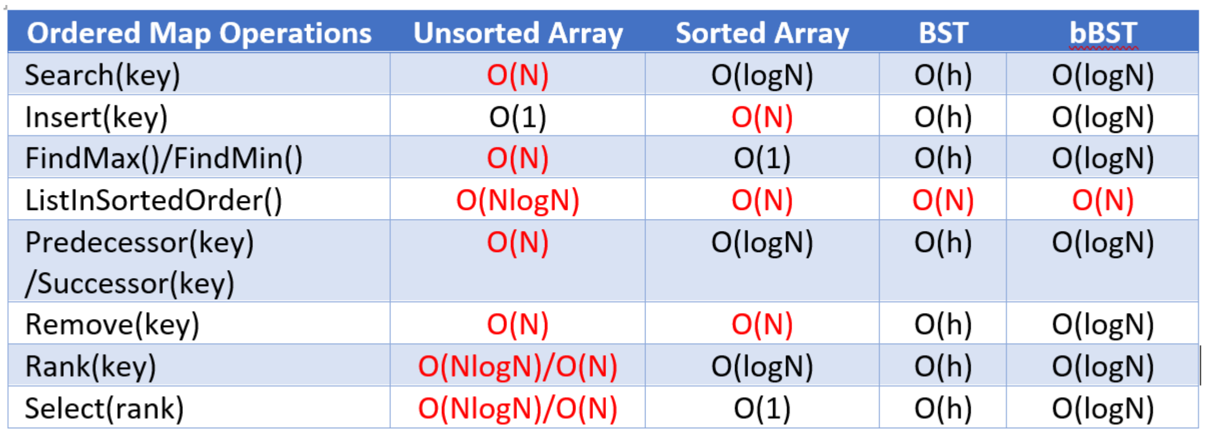
- Java API does not have this data structure, so you'll have to use your own (fairly simple anyways, <100 lines)

4.4. Ordered Maps/Sets (via balanced Binary Search Trees)

where items are given a certain ordering so that more operations are supported.

Introduction

- Map ADT does search, insert and remove well in O(1) time, but does not really support operations like retrieving the data in sorted order or finding an object's rank. (no sense of ordering)
- Hence, introduce **ordering** to Map ADT to give an Ordered Map ADT, where items are still accessed/manipulated through a key.



Ordered Map Operations	Unsorted Array	Sorted Array	BST	bBST
Search(key)	O(N)	O(logN)	O(h)	O(logN)
Insert(key)	O(1)	O(N)	O(h)	O(logN)
FindMax()/FindMin()	O(N)	O(1)	O(h)	O(logN)
ListInSortedOrder()	O(NlogN)	O(N)	O(N)	O(N)
Predecessor(key) /Successor(key)	O(N)	O(logN)	O(h)	O(logN)
Remove(key)	O(N)	O(N)	O(h)	O(logN)
Rank(key)	O(NlogN)/O(N)	O(logN)	O(h)	O(logN)
Select(rank)	O(NlogN)/O(N)	O(1)	O(h)	O(logN)

Operations for Ordered Map ADT and their respective time complexity given different implementations.

- Note: for Unsorted Array, either sort it first or use QuickSelect (for Select(int))/linear scan (for rank(key))
- Big difference between O(N) and O(logN) - e.g. If N = 1 million, $log_2 N \approx 20$
- **Static data structures** are those that are efficient if there is no (or rare) updates (e.g. the insert/remove operations)
- **Dynamic data structures** are efficient even if there are many update operations. Binary Search Trees/AVL trees fall into this category.

Binary Search Trees

- Assume that all keys are unique. We can easily modify our BST to deal with duplicate keys by adding in a 'count' variable at each vertex.
- **BST Property:** For every vertex x and y:
  - y.key < x.key if y is in left subtree of x
  - y.key > x.key if y is in right subtree of x
- **Other Attributes:**
  - Height: #edges on the path from this vertex to deepest leaf
  - Size: #vertices of the subtree rooted at this vertex

- These have to be **updated** when doing operations that modify the data structure (i.e. insert/delete) → only have to update all nodes along the taken path (O(logn))

- **Operations:**
  - **search(value)** - find the given value within the BST.
    - keep going left/right depending if the search value is more or less than the current node's key, until you either run out of nodes, or reach a node with key == search value.
    - a similar approach to regular **Binary Search** in a sorted array.
    - **Analysis: O(h)**, where h is the height of the BST
  - **insert(value)**
    - similar to search, except that upon running out of nodes to traverse we insert the value at that point in a new node
    - **Analysis: O(h)**
  - **findMin()/findMax()**
    - keep going to the left/right subtree until you reach the end, which will be the node with either the minimum or maximum key in the BST.
    - **Analysis: O(h)**
  - **listInSortedOrder()**
    - simply do an **inorder traversal** of the BST:
    - **inorder()**
      1. Traverse the left subtree by recursively calling the in-order function
      2. Print the data of the current vertex
      3. Traverse the right subtree by recursively calling the in-order function
    - Hence, using the BST to sort an array is possible, but the other sorting algorithms are probably faster/easier to use.
    - **Note:** also have preorder() (print data at current vertex first) and postorder() traversal (print data at current vertex last)
    - **Analysis:** Each vertex is only visited once → **O(N)**
  - **successor(value)** - find the next value in sorted order from the given value (opposite: **predecessor(value))**
    1. First, look for the value within the BST.
    2. If the value has a right subtree, simply return the result of findMin in the right subtree
    3. Otherwise, keep going up the parents until a value larger than the current one is encountered. If you go all the way to the root without finding such a value, it means that the given value has no successor/is the max element.

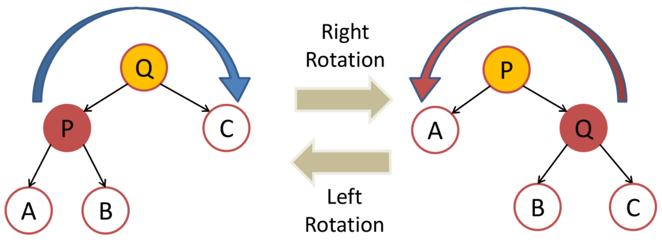
    - **Analysis: O(h)** for search(value) **+ O(h)** worst-case for getting the successor/predecessor (when it doesn't exist)
  - **remove(value)**

        - First, find the location of the node using search (or return that the node is not found)
        - We then have three possible cases:
            1. Node to be removed is a **leaf node** - simply get rid of it by disconnecting it from the tree
            2. Node to be removed has **one child** - just bypass this node and connect its child directly to its parent
            3. Node to be removed has **two children** - replace the node with its successor amongst its children (since the right subtree is guaranteed to exist), then call remove() on the successor node
                - **Rationale** for replacing the node with its successor: (predecessor is also fine)
                - successor node can only have ≤ 1 child in this case, so it will be easier to delete
                - BST property will not be violated
        - **Analysis:**
            - O(h) time needed for search(value)
            - Case 1: O(1) time to delete the node
            - Case 2: O(1) time for bypassing the node
            - Case 3: ~O(h) time to find the successor and delete it (abit simplistic)
            - Total time-complexity: **O(h) time**
    - **rank(value)** - return the rank k of the element with the given value, rank starts at 1.
        - Hence, numSmaller(value) = rank(value) - 1
        - Recursively determined starting from the root node using the size attribute at each node. Basically, only count the number of elements to the left of it.
        - Time-complexity: **O(h) time**
    - **select(integer) -** return the value of the kth smallest element
        - Similar to the QuickSelect operation - start at the node and recursively go down the left and right subtrees until the rank of the current node = the desired rank
        - Time-complexity: **O(h) time**
- **Caveat:** The worst-case height of a BST is O(N), e.g. if we added the nodes in sorted order (our tree structure depends on the **order of insertion)** → need a way to **balance the binary tree** so that the height is kept at O(logN).
    - Observe that if we have a perfectly balanced binary tree, then searching for something would work almost exactly like binary search on a sorted array.

<u>Adelson-Velskii & Landis (AVL) Trees</u>

- We say that a BST is balanced if h = O(logN) i.e. c*logN. A perfectly balanced BST is hard to achieve, but we can maintain a fairly well-balanced one.
- **Strategy:**
    - Define a good property of a tree
    - Show that if the good property holds, then the tree is balanced

- After every insert/delete, make sure the good property still holds.
    - If not, fix it!
- **General Idea:**
    - **Step 1:** Augment the tree with the height attribute
    - **Step 2:** Define the invariant property:
        - The tree must be **height-balanced**, where **height-balanced** means that all vertices in the tree are height balanced
        - A **vertex is height-balanced** if $|x.left.height - x.right.height| \leq 1$
        - **Lemma**: A height-balanced tree with N vertices has height $h < 2 * log_2 N = O(logN)$
    - **Step 3:** Show how to maintain height-balance
        - Define **balance factor bf(x) = x.left.height - x.right.height**
        - Every time we modify the tree, check the balance factor of every vertex along the path we took and do a rebalancing operation immediately if encountering bf(x) = 2 or -2
        - **Basic Rebalancing Operations**



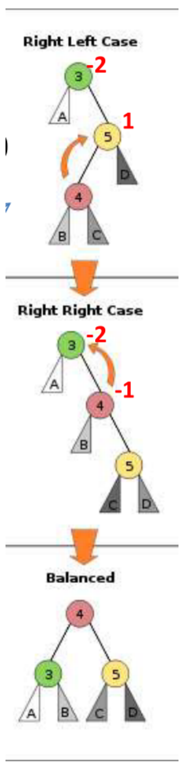
Rotations maintain ordering of keys
$\Rightarrow$ Maintains BST property *(see vertex B where P $\leq$ B $\leq$ Q)*

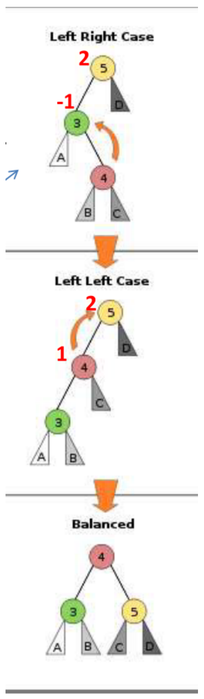
- Remember to update the height and sizes of P & Q after rebalancing
- Need to be careful to avoid vertices getting garbage collected!

Four Possible Cases:

- **Left Right Case:** bf(x) = +2 and bf(x.left) = -1
    - leftRotate(x.left), then rightRotate(x)
- **Left Left Case:** bf(x) = +2 and 0 ≤ bf(x.left) ≤ 1
    - rightRotate(x)
- **Right Left Case:** bf(x) = -2 and bf(x.right) = 1
    - rightRotate(x.right), then leftRotate(x)
- **Right Right Case:** bf(x) = -2 and -1 ≤ bf(x.right) ≤ 0
    - leftRotate(x)
- **A note on rebalancing:**

- Insertion can only trigger **at most one** of the four rebalancing cases
- Deletion can trigger one of the four rebalancing cases **up to h = logn times.**





- Many different types of balanced search trees exist, such as B-trees, red-black trees, splay trees, skip lists etc.
- **Java API:**
  - TreeSet/TreeMap which are an ordered version of HashSet/HashMap and support more operations, albeit mostly in O(logn) time. Keys need to either implement the Comparable interface, or a Comparator must be provided
    - Internally, uses Red-Black trees
    - **Note**: getting a subset of elements takes O(1) time, but only because the same tree is being re-used with the new boundaries defined. calling size() subsequently on the subtree would take O(n) time
    - **Warning:** Does not support duplicate keys!

5. Graphs

5.1. An Introduction to Graphs (mostly from CS1231)

Graph Types

- An **undirected graph G** consists of 2 finite sets: a nonempty set V of **vertices** and a set E of **edges**, where each (undirected) edge is associated with a set consisting of either one or two vertices called its **endpoints**.
  - An edge is said to **connect** its endpoints; two vertices that are connected by an edge are called **adjacent vertices**; and a vertex that is an endpoint of a loop is said to be **adjacent to itself.**
  - An edge is said to be **incident on** each of its endpoints, and two edges incident on the same endpoint are called **adjacent edges.**

    - We write e = {v, w} for an undirected edge e incident on vertices v and w.
- A **directed graph** G, consists of 2 finite sets: a nonempty set V of **vertices** and a set E of **directed edges**, where each (directed) edge is associated with an **ordered pair** of vertices called its **endpoints**.
    - We write e = (v, w) for a directed edge e from vertex v to vertex w.
- A **weighted graph** is a graph for which each edge has an associated positive real number weight . The sum of the weights of all the edges is the total weight of the graph.
- **Other edge types:** Undirected, Directed, Bidirected, Weighted/Unweighted
- A **simple graph** is an undirected graph that does **not** have any loops or parallel edges. (That is, there is at most one edge between each pair of distinct vertices.)
- A **complete graph** on n vertices, n > 0, denoted K_n, is a simple graph with n vertices and exactly one edge connecting each pair of distinct vertices. (will have nC2 = $O(V^2)$ edges)
- A **bipartite graph** is a simple graph whose vertices can be divided into two disjoint sets U and V such that every edge connects a vertex in U to one in V. (also exists **complete bipartite graph** $K_{m,n}$)
    - **Properties:** The graph is **2-colourable** and there are **no odd length cycles.**
- A graph H is said to be a **subgraph** of graph G iff every vertex in H is also a vertex in H, every edge in H is also an edge in G, and every edge in H has the same endpoints as it has in G.
- **Sparse Graphs** - # of edges E ≤ O(V)
- **Dense Graphs** - # of edges E = $O(V^2)$

<u>**Graph Properties**</u>

- Let G be a graph and v a vertex of G. The **degree of v**, denoted deg(v), equals the number of edges that are incident on v, with an edge that is a loop counted twice.
    - Also, we have indegree and outdegree for directed graphs
- Let G be a graph, and let v and w be vertices of G. A **walk** from v to w is a finite alternating sequence of adjacent vertices and edges of G. Thus a walk has the form $v_0 e_1 v_1 e_2 \ldots v_{n-1} e_n v_n$, where the v's represent vertices, the e's represent edges, v_0=v, v_n=w, and for all i ∈ {1, 2, …, n}, v_{i-1} and v_i are the endpoints of e_i. The number of edges, n, is the **length** of the walk.
- The **trivial walk** from v to v consists of the single vertex v.
- A **(simple) path** from v to w is a walk that does not contain a repeated vertex or repeated edges.
    - **Path Length/Weight** is given either by # of edges in the path (for unweighted graphs), or sum of edge weights in the path (for weighted graphs)
- A **simple cycle** is a path that does not have any other repeated vertex except the first and last, and starts and ends with the same vertex. (≥3 vertices for a non-trivial cycle)
    - An **undirected graph** is cyclic if it contains a loop or a cycle; otherwise, it is acyclic.
- Two vertices v and w of a graph G are **connected** iff there is a walk from v to w.
- The graph G is **connected** iff given any two vertices v and w in G, there is a walk from v to w.
- A graph H is a **connected component** of a graph G iff
    1. The graph H is a subgraph of G;

2. The graph H is connected; and
3. No connected subgraph of G has H as a subgraph and contains vertices or edges that are not in H.

Graph Applications

- Graphs are an extremely useful tool for problem-solving. The general strategy would be to **model the problem as a graph**, consisting of **vertices, edges (and maybe weights).** Following which, we can apply or modify one of the already-known graph algorithms to tackle the problem at hand.
  - e.g. Map Colouring Problem → Convert to a Graph so this becomes a **Vertex Colouring Problem**
  - R**eal-world examples:** Social Network Analysis, Transportation Networks, Optimization Problems, Communication Networks etc.

5.2. Graph Data Structures

In order of how **compressed** the information is:

1. **Adjacency Matrix**
   - Edge information is stored within a 2D array. AdjMatrix[i][j] contains 1 if there exists an edge i → j in G, otherwise it contains 0. For weighted graphs, it would store the weight of the edge instead. (with a way to indicate if the edge is not present)
   - The matrix is hence symmetrical for an undirected graph.
   - **Space Complexity:** $O(V^2)$
2. **Adjacency List**
   - An array of V lists, where AdjList[i] stores the (possibly sorted) list of i's neighbours. For weighted graphs, store the pair (neighbour, weight)
     - Implication: for undirected graphs we basically store each edge twice
   - **Space Complexity:** O(V+E) $\approx O(max(V, E))$
3. **Edge List**
   - an array of (possibly sorted) E edges, where each edge is represented by (u, v, w(u, v))
   - **Space Complexity:** O(E)

- Note: in the worst-case, $O(E) = O(V^2)$
- **Simple Applications:**
  1. **Count the number of vertices V** → trivial O(1) for AdjMatrix and AdjList, cannot be done correctly for EdgeList due to the potential of vertices with degree 0
  2. **Count the number of edges E**
     - AdjMatrix - $O(V^2)$, need to iterate over at least half of the AdjMatrix
     - AdjList - O(V), sum the length of the V lists.
     - Trivial O(1) for edgeList
     - remember not to double-count for undirected edges!

    - If we track this with a separate variable, can be done in O(1) time for all.

3. **Enumerating neighbours of a vertex v (important for many algorithms!!)**
    - AdjMatrix - O(V), scan the entire row AdjMatrix[v]
    - AdjList - O(k), where k is the number of neighbours of v, output-sensitive algorithm because we can over AdjList[v]
    - EdgeList - binary search for required vertex, then scan up and down? At least O(logE + k)
4. **Checking existence of an edge (u, v)**
    - AdjMatrix → O(1), AdjMatrix[u][v]
    - AdjList - O(k) or O(logk) to check through AdjList[u]
    - EdgeList - troublesome, some kind of binary search as well if sorted, O(logE)

- **Summary**: Pros & Cons exist for each of the data structures, depends on the application. Generally, AdjMatrix is better for dense graphs while AdjList is better for sparse graphs.

5.3. Graph Traversal (and some applications)

Graph Traversal

- What's needed
    - **Start -** need to know which is the **source vertex**. For rooted trees this is usually the **root**.
    - **Movement -** we need to define **how** we want to travel across the graph using edges. For binary trees we have preorder/inorder/postorder traversal, while for general graphs we can use **breadth-first search/depth-first search.** Also, we need to consider how we might deal with cycles.
- **General Idea:** If a vertex v is reachable from s, then all neighbours of v will also be reachable from s (recursive definition)
- Both algorithms make use of a **visited** array to avoid revisiting the same vertices again, and a **predecessor** array to memorize the paths taken, which will form a **spanning tree** of the connected component the source is in. The different algorithms may form **different** spanning trees.
- **Breadth-first Search (BFS)**

  Analogy: *ripples from a water droplet spreading evenly outwards.*

```
Breadth-First Search Pseudocode (basic)
- Initialize the visited and predecessor arrays // O(V)
- Initialize a queue to track the vertices to visit. Initially it contains only the source vertex.
- Set visited[s] = true

while Q is not empty: // O(E), each reachable vertex and its neighbours scanned exactly once
    int currVertex = Q.dequeue()
    for all neighbours adjacent to currVertex:
        if not visited[neighbour]:
            visited[neighbour] = true // avoid enqueuing same vertex multiple times
            p[neighbour] = currVertex
            Q.enqueue(neighbour)
```

  - **Analysis:**
      - Initialization is **O(V)**

    - while-loop always takes O(E) time to scan all edges. Not O(V+E) because if E < V then number of vertices visited still depends on E. Assumes we use an **adjacency list.** If adjacency matrix is used then it will take $O(V^2)$ time.
    - Total: **O(V+E)** time. For a sparse graph this means O(V) time, for a dense graph $O(V^2)$ time.
- **Depth-first Search (DFS)**

  Analogy: how we solve a maze by travelling all the way along a potential route, and backtracking to the previous intersection if that doesn't work.

```
Depth-First Search Pseudocode (basic)
- initialize the visited and predecessor arrays // O(V)
- call the recursive DFS function, DFS(source)

DFS(node):
    visited[node] = true
    for all neighbours adjacent to node:
        if !visited[neighbour]:
            p[neighbour] = node
            DFS(neighbour)
```

- Implicitly uses a **stack** through recursion to keep track of the order of vertices to visit. The recursive stack will 'unwind' once it reaches the end.
- Might use less memory than BFS (if the graph is wider than it is deep e.g. in a full binary tree)
- **Analysis:**
    - Similar to BFS, O(V+E) time-complexity if adjacency list is used. For a sparse graph - O(V), dense graph - $O(V^2)$
- We can **reconstruct the path taken** from the source to a specific vertex quite easily by iterating\recursing through the p array. Takes O(k) time where k is the path length.

<u>**Applications (refer to lecture slides for detailed pseudocode)**</u>

1. **Reachability Test** - test whether a vertex v is reachable from vertex u.
    - Do BFS/DFS from s = u. After that, check if visited[v] is true.
    - Takes **O(V+E)** time
2. **Shortest Path** between two vertices in an **unweighted/equally weighted graph** using BFS (see SSSP section)
3. **Identifying/Counting Components**
    - keep a CC counter and the visited array separate from BFS/DFS
    - for all v in V: if v is not visited yet mark all connected vertices as visited using bfs/dfs, and increment the CC counter.
    - **O(V+E)** time complexity, because each edge is still visited either once or twice (for undirected graphs)
4. **Topological Sort (i.e. linearization of partial orders in CS1231)**
    - **Definition:** A **linear ordering** of the vertices of a **Directed Acyclic Graph (DAG)** in which each vertex comes before all vertices to which it has outbound edges.
    - Every DAG has ≥ 1 topological sorts.

- **Proof:** Basically, it can be proven that we can always find a node with no incoming edges in a DAG, so we can repeatedly take the node with no incoming edges to get a valid topological sort. (since removing it from the DAG will still result in a DAG)
    - This is the idea behind Kahn's algorithm which uses BFS
- **Kahn's Algorithm (BFS-based)**
    - Modify BFS using an indegree array and a topoSort array to store the results.
    - Initialize the data structures, and enqueue all vertices with indegree = 0. (to account for multiple connected components)
    - In the while-loop, we will also append our dequeued vertex to topoSort, and decrement the indeg counter for all its neighbours. If any neighbour now has indeg = 0, add it to the queue.
    - return the topoSort array
    - **Analysis: still O(V+E),** since we will pass through all vertices and process all their edges once.
- **DFS-based Topological Sort**
    - Do DFS except that the vertices are appended to the back of the topoSort array in **postorder** fashion. (after all neighbours have been visited)
    - **Result**: always put all reachable vertices **before** the current vertex → reversing the topoSort array will result in a valid topological sort
    - Have to also account for the possibility of multiple components by having a loop that calls the DFS function for a vertex if it has not been visited.
    - Even if we start from a vertex with indegree ≠ 0, algorithm still works because its predecessors will definitely be **behind** it in the topoSort array.
    - **Analysis:** O(V+E), similar to analysis for identifying CCs.

5. **Identifying Strongly Connected Components (SCCs)**
    - **Strongly connected component** is the equivalent of a connected component for **directed graphs.**
        - Harder to identify due to the directionality of the edges
        - An SCC **must have a cycle** if it has >1 vertex
            - Implication: DAG → always has N SCCs, where N is the number of vertices
        - Treating each SCC in a directed graph as a node, we get a DAG
    - **Kosaraju's Algorithm**
        1. Perform DFS topological sort on the given directed graph G → get a post-order ordering of vertices in an array K. (not a DAG, so not a topological ordering per se) Remember to reverse K as in DFS topoSort. Takes **O(V+E)**
        2. Create transpose graph G' of G, where the direction of all edges are reversed. **O(V+E)**
        3. Count SCCs in G' similar to how we counted CCs, except that we visit all vertices based on the order given by the reversed K. If the vertex we visit is not marked as visited yet, SCC += 1 and do DFS/BFS from that vertex to mark all connected vertices as visited. This also takes O(V+E)

- **Why does this work?**
    - Reversing all the edges in the SCC will still result in the same SCC, except with the edges between SCCs flipped.
    - the DFS TopoSort algorithm ensures that:
        - for any SCC x, all reachable SCCs from x will have their first-reached vertex placed in K before the first-reached vertex of x
        - all vertices in x will be placed before the first-reached vertex in x
    - After that, when we visit SCCs in topological ordering of G, the first vertex we encounter for each SCC will be exactly the first-reached vertex during the topoSort, and the reversed edges prevent us from visiting **unvisited** vertices in the other SCCs. Hence, we can count the number of SCCs correctly.

Others:

- Detecting Bipartite Graphs (see tutorial)
- Cycle Detection (both directed & undirected graphs) (see tutorial)
- Checking for cut vertex/articulation points or bridges in the graph (see cp3 book)
    - A **Cut Vertex, or an Articulation Point**, is a vertex of an undirected graph which removal disconnects the graph. Similarly, a **bridge** is an edge of an undirected graph which removal disconnects the graph.

5.4. Minimum Spanning Trees

*A **minimum spanning tree** for a connected, undirected and weighted graph is a spanning tree that has the **least possible total weight** compared to all other spanning trees for the graph.*

- **Properties:**
    - **Cycle Property -** For any cycle C in graph G(V, E), if the weight of an edge e is larger than every other edge in C, e **cannot** be included in the MST of G(V,E)
    - **Cut Property -** For any cut of the graph (partition of vertices into 2 disjoint sets), if the weight of an edge e in the cut-set (set of edges crossing the cut) is the smallest out of all the edges in the cut-set, then e belongs to all MSTs of the graph.
        - Note: doesn't say anything about whether the other edges in the cut-set belong in the graph
    - MSTs may not be unique if there exist edges with the same weights
- **Strategies:**
    1. **Brute Force**
        - find all the cycles in the graph and break them by removing the largest edge (makes use of the cycle property)
        - problem: how do we get all the cycles in the graph efficiently? up to $O(2^N)$ possible cycles.
    2. **Prim's Greedy Algorithm**
        - **General Idea:** Steadily grow the spanning tree from the source by greedily choosing the smallest edge to an unconnected vertex amongst all the edges currently connected to the spanning tree.

1. First, decide on a starting vertex s and set-up a vertexSet/vertexTaken array.
2. Enqueue all edges connected to s into a priority queue that orders them by increasing weight
3. while the priorityQueue is not empty/less than V vertices have been connected:
   - poll() an edge from the priorityQueue
   - if the vertex v this edge connects to is not yet taken (i.e. v not in vertexSet):
     - vertexSet.add(v)
     - enqueue each edge adjacent to v into the priorityQueue if they connect to a vertex not in vertexSet
4. Return the MST. Depending on what we want, we could keep an edgeSet or just track the weight of the MST.

- **Analysis: O(E log V),** because each edge is only processed at most once, and the priorityQueue could have E edges, so enq and deq takes O(E) = 2O(V)
- **Correctness:** Prim's is correct because it exploits the cut property at every greedy step (always chooses the smallest edge in the current cut-set)
- Note that we could replace the Binary Heap with faster and more advanced data structures like Fibonacci Heaps to achieve better time-complexity!

3. **Prim's Variant for Dense Graphs**
   - Motivation: If a graph is dense then regular Prim's will take $O(V^2 logV)$, but we can achieve $O(V^2)$ by replacing the priority queue with an array.
   - General Idea: The array will keep track of the smallest weighted edge to v amongst all edges to v that have been explored thus far. If a vertex is already visited, set its weight in the array to +inf so it doesn't get selected.
     - I guess the intuition is that while dequeuing the minimum edge now takes O(V) time, enqueuing/considering all the neighbours now takes O(1) time per edge → ultimately faster since there are so many edges. Normal Prim's is O(logV) for both operations.
   - Not useful for sparse graphs as it **always** takes $O(V^2)$ time. We also need prior knowledge about the input charateristics, so by default we would just stick to regular Prim's.
4. **Kruskal's Greedy Algorithm**
   - **Idea:** Sort the edgeList in increasing weight, and while we haven't picked V-1 edges we consider the next smallest unprocessed one one and add it to the MST if it wouldn't form a cycle.
   - To test for cycles, we use a UFDS (check if the two ends are already in the same set), as Kruskal's may result in a forest of trees in intermediate steps, so we cannot simply keep track of which vertices have been connected.
     - Every time we add in a new edge, we call unionSet on both ends of the edge
   - **Time Complexity: O(ElogV)**, where the sorting step is the chokepoint.
   - **Correctness:** Again, exploits the cut property at every greedy step by always choosing the smallest possible edge, so no wrong edge can possibly be chosen - see lecture notes for the full proof

    - Apparently, Kruskal's tends to be a bit faster for sparse graphs, while Prim's tends to be slightly faster for denser graphs.
- **Applications:**
    - Finding the **MiniMax path**, which minimizes the maximum edge encountered along the path between two vertices (and its opposite, MaxiMin path)
        - First, find the Minimum Spanning Tree, then we can do DFS/BFS to find the exact path to take.
        - While reconstructing the path, we can then keep track of what is the maximum edge we encounter along the path.

5.5. Single-Source Shortest Paths (SSSP)

Given a weighted graph and a source vertex s, find the ***minimum path weight*** *from s to every other vertex in the graph.*

Introduction

- A very well-known and important problem in Computer Science, used in applications like GPS routing.
- Maintain **distance array D** of size V to track the current shortest path estimate from s to v.
    - D[v] should start off larger than $\delta(s, v)$ but converge to that value once the SSP algorithm terminates
    - If v = s, D[v] = 0.
    - Note that the minimum path weight from a to b, $\delta(a, b)$, can be infinity if b is unreachable from a
- Maintain **predecessor array p** (same as in BFS/DFS). p[v] = -1 if not defined, otherwise contains the **predecessor on the best path from source s to v.**
- Graphs can sometimes contain **negative weights,** and even **negative cycles** (where the total cycle weight is negative). In negative cycle situations, the shortest path to the vertices involved will be **undefined** as we can endlessly traverse the cycle to get a smaller path weight converging on negative infinity.
- **Relaxation Operation** (important for many of the SSSP algorithms!)

```
relax(u, v, w(u, v)):
    if D[v] > D[u] + w(u, v): // if the SP can be shortened
        D[v] = D[u] + w(u, v) // relax the edge
        p[v] = u
        // update some other data structures if necessary
```

- **O(VE) Bellman Ford's SSSP Algorithm**
    - **Idea**: An algorithm has solved the SSSP problem when all edges can no longer be relaxed further.
    - Hence, repeatedly relax edges until we no longer need to do so. The below algorithm will always solve the SSSP for all vertices so long as no **negative weight cycles** exist. (negative weights are okay)
    - **Algorithm:**
        1. initialize the data structures (D and P arrays)

2. repeat the following V-1 times: \\ O(V)
    - for each edge in the edgeList, relax() that edge. \\ O(E) (adjacencyList also possible)

- **Possible improvement:** Early termination if no edges are relaxed in a single pass (similar to improved bubble sort)
- **Proof of Correctness:**
    - Theorem 1: If G = (V, E) contains no negative weight cycle, then the shortest path p from s to v is a **simple path**. (proof by contradiction)
        - **Implication**: any shortest path p has at most V-1 edges from the source s to the "furthest possible" vertex v in G.
    - Theorem 2: If G = (V, E) contains no negative weight cycle, then after Bellman Ford's terminates D[v] = $\delta(s, v), \forall v \in V$ (proof by induction)
        - Because after each pass, we minimally solve the shortest path for vertices that have a shortest path one edge further from s, regardless of what order the edges are in!
        - Hence after V-1 passes, even the vertex with the longest possible shortest path (of length V-1) will be solved.
- That said, if the edges are in a 'good' order it is possible that we require **far less passes.**
- **Corollary**: If a value D[v] fails to converge after V-1 passes, then there exists a **negative weight cycle** reachable from s. (but the algorithm will still terminate in O(VE) always)

<u>Cases</u>

All the cases can be solved by the O(VE) Bellman Ford's algorithm, but this is rather slow! In many situations, we can try to simplify the problem by making certain assumptions, leading to better algorithms.

1. The graph is a **tree**
    - Every path in the tree is now the shortest path, as there is only one unique path between vertices.
    - Doesn't matter if there are negative weights
    - Hence, using simple **BFS/DFS** can solve our problem in O(V + V - 1) = **O(V) time.**
2. The graph is **unweighted/all edges have identical weights**
    - simplify problem into finding the **least number of edges** to get from the source to the other vertices
    - Getting the **BFS spanning tree** does this precisely. We can then reconstruct the path to v using the predecessor array.
    - If we want the cost to each vertex, modify BFS to update a Distance array as it goes so we no longer need to do a path reconstruction.
    - **O(V+E) time complexity**
3. The graph is a **DAG (-ve edge weights are okay)**
    - Cycles are problematic for finding the shortest paths as there's no good way to get the optimal ordering beforehand
    - Hence, it may result in us relaxing the same edge multiple times before we get the correct shortest path as multiple candidate paths can use the same edge.

- For acyclic graphs, we can do a **one-pass Bellman Ford** and only do the relaxation once for each edge, **so long as it is done in topological order.**
    - Relax the outgoing edges of vertices listed in topological order
    - Idea behind this is based on **dynamic programming**, where we always find the shortest path for vertices before moving on to their neighbours (solve the subproblems and build up)
- **Time-Complexity: O(V+E)** for both getting the topological sort and for the single pass of Bellman Ford

4. The graph has **no negative weight edges (i.e. all w(u, v) ≥ 0)**
    - **General Idea for (original) Dijkstra's Algorithm**
        - Maintain a **solved** set of vertices whose final shortest path weights have been determined, and originally solved = {S}
        - Repeatedly select vertex u in {V - solved} with the min shortest path estimate D[u], add u to solved, and relax all edges out of u.
            - Choice of relaxation order is '**greedy**', and vertices are **never reselected**, but this strategy does work.
            - Vertices are added to solved with **non-decreasing** shortest path cost
    - **Detailed Pseudocode**
        1. We initialize both a priority queue and D[] array for the shortest path estimates.
            - In the PQ, we store the current shortest path estimate for a vertex v as the pair (d, v), where d = D[v], and the PQ is ordered by smallest d first.
            - Initially, enqueue (infinity, v) for all vertices except the source, which will be (0, s)
        2. **Main Loop**: keep removing vertex u with minimum d from the PQ, add u to solved, and relax all edges until the PQ is empty
            - If an edge (u, v) is relaxed find the vertex v it is pointing to in the PQ and update the shortest path estimate as well as it's value in the D[] array. This is only done if v remains in the PQ.
            - Rather than making a custom binary heap with a decreaseKey operation, we can simply use a bBST and search/delete/insert elements with the same time complexity of O(logV) (although initialization will be a tad slower)
    - **Why does this work?**
        - At every iteration, the vertices in the solved set have the correct shortest path estimate
        - **Lemma 1**: Subpaths of a shortest path are shortest paths (proof by contradiction)
        - **Lemma 2**: After a vertex v is added to solved, the shortest path from s to v has been found
            - We can consider all the cases and prove this can never be false (proof by contradiction) because of how we greedily choose vertices with minimum shortest path estimate + the fact that there are no negative edge weights
    - **Time Complexity:**
        - **O(VlogV)** because each vertex is only inserted and extracted from the PQ once

        - **O(ElogV)** for the relaxing of every single edge once. In the worst case we always get to relax an edge, which results in us deleting and re-inserting a new (d, v) entry into the bBST.
        - Total: **O((V+E) log V)** run-time
        - For a dense graph, we may want to make a modification similar to the dense variant of Prim's, where we replace the priority queue with an array → achieve $O(V^2)$ time complexity instead

5. The graph has **no negative weight cycle**
    - **Modified Dijkstra's Algorithm**
        - Original Dijkstra does not work when there are negative weight edges.
        - **Solution:**
            - Allow vertices to be processed multiple times, by always enqueuing a (D[v], v) pair into the priority queue so long as D[v] decreases during edge relaxation. This ensures that an updated shortest path estimate is propagated to all it's neighbours.
            - There can now be multiple (D[v], v) pairs within the priority queue, so we have to check that d = D[v] (most updated pair) before relaxing all edges out of v.
            - Hence, a normal priority queue will suffice.
        - **Pseudocode:**

```
Modified Dijkstra's Algorithm, in many ways simpler than the original

initSSSP(s) \\ O(V)

PQ.enqueue(0, s)
while PQ is not empty:
    (d, u) = PQ.dequeue()
    if d == D[u]:
        for each vertex v adjacent to u:
            if D[v] > D[u] + w(u, v):
                D[v] = D[u] + w(u, v)
                PQ.enqueue((D[v], v))
```

        - **Analysis:**
            - If there's no negative weight edge, there will never be another path that can decrease D[u] once u is dequeued from the PQ.
                - Each vertex will hence still be dequeued from the PQ and processed once (and outdated copies will be ignored)
                - In the worst-case, we always relax all of our neighbours so the PQ can have a size of O(E)
                - Hence, our runtime will be **O(ElogE)** in the worst-case.
                - Same as O((V+E)logV) except when E < O(V) in very sparse graphs
            - In certain extreme cases, there could be exponential time-complexity where certain edges are re-processed a large number of times.
            - But in practice this usually runs much faster than Bellman Ford's algorithm.
            - If the graph has a high chance of having a negative weight cycle, Bellman Ford might be a better choice because it will not get trapped in an infinite loop.

5.6. All-Pairs Shortest Paths (APSP)

Given a weighted graph, find the ***minimum path weight (shortest path)*** *between* ***every pair of vertices*** *in the graph.*

- **Application:** Finding the **diameter** of a graph, defined as the greatest shortest path distance between any pair of vertices. (e.g. finding the # of clicks to traverse between any pair of webpages)

Naive Approaches using SSSP Algorithms

- **Unweighted Graph** - Call BFS V times, once from each vertex → O(V*(V+E)) = $O(V^3)$ for a dense graph
- **Weighted Graph**
  - Call Bellman Ford's V times → O(V*VE) = $O(V^4)$ for a dense graph
  - Call original/modified Dijkstra's V times → $O(V^3 logV)$ for a dense graph

Floyd Warshall's Algorithm

- We can do better than the naive approaches and attain a time-complexity of $O(V^3)$
- If we expect many shortest path queries, it can be used as a **preprocessing step** on the data
- **Pseudocode: (pretty simple!)**
  - Use a 2D matrix to keep track of the shortest path cost - D[][]
  - Initialize D[i][i] = 0, D[i][j] = weight of edge(i, j) if it exists, otherwise infinity
  - Main Loop:
    - for (int k = 0; k < V; k++)
      - for (int i = 0; i <V; i++)
        - for (int j = 0; j < V; j++)
          - **D[i][j] = Math.min(D[i][j], D[i][k] + D[k][j]); // relax D[i][j]**
- **Rough Intuition:**
  - Makes use of Dynamic Programming, and builds up the solution by finding the shortest paths between i and j passing through only [0, ... , k] and incrementing k. Upon initiation k = -1, and we only consider direct paths.
  - When tracing the algorithm, it should be clear that all possible cases will be accounted for.
- **Variants:**
  1. **Printing the actual shortest path**
     - Create a 2D predecessor matrix p, where **p[i][j]** is the predecessor of **j** on a shortest path from **i** to **j** i.e. i → ... → p[i][j] → j
     - Initially, set p[i][j] = i for all pairs of i and j (regardless if the edge exists)
     - While running Floyd Warshall, if edge relaxation takes place, set p[i][j] = p[k][j]
     - Afterwards, when doing path reconstruction, if p[i][j] == i also need to check that D[i][j] ≠ infinity
  2. **Transitive Closure**

    - **The Transitive Closure Problem:** Given a graph, we want to determine if vertex i is connected to vertex j either directly or indirectly.
    - Modify D to contain only 0s and 1s, and D{i][i] = 0 initially, while D[i][j] = 1 if an edge exists
    - During Floyd Warshall, change the relaxation step to:
        - **D[i][j] = D[i][j] | (D[i][k] & D[k][j])**, where | and & are bitwise logical operators
    - Resultant D matrix can be used to determine SCCs in the graph.

3. **MiniMax/MaxiMin**
    - D[i][j] now stores smallest possible maximum edge weight instead (for the minimax problem)
    - Initially, D[i][i] = 0, D[i][j] = weight of edge if it exists, otherwise infinity
    - During Floyd Warshall's, relaxation step should be changed to:
        - **D[i][j] = Math.min(D[i][j], Math.max(D[i][k], D[k][j])**
4. **Detecting positive/negative cycles**
    - Set the main diagonal to positive infinity
    - Run Floyd Warshall's
    - Recheck the main diagonal:
        - If the value is ≥0 but < infinity → positive cycle
        - if the value is less than 0 → negative cycle

- **Final Note**: Be careful of integer overflow when implementing, make sure a proper infinity constant is used that supports operations like infinity + infinity = infinity, infinity + x = infinity etc.