1 Introduction

Some questions to ask before starting on a problem

- Extract out important keywords (what DS to use?)
- Edge cases? *e.g.* if `size==0` or `size==1`,
- Trivial cases? can just hardcode

Code styling

- CS2030 Code Styling Guide
- Google Java Styling Guide
- **Modularity**: use method to print answers inside main method

```
\\ print answer
ans = simulate(n,k,m);
printAns();
```

- **No global variables**

2 Java

How to throw exception?

```
public class MyException extends
    Exception {
  private int var;
  public MyException(int var) {
    this.var = var
  }
  public int getVar() {
    return this.var;
  }
}

public class Main {
  public static void main(String[] args
    ) {
    try {
      ...
      throw new MyException(errorVar);
    } catch (MyException e) {
      System.out.println(e.getVar());
    }
  }
}
```

3 Data Structures

$$O(1) < O(\log(n)) < O(n^c) \text{ where } c < 1$$
$$O(n) < O(\log(n!)) = O(n\log(n)) < O(n^2)$$
$$O(n^k)[\text{ where } k > 2] < O(k^n)[\text{ where } k \geq 1] < O(n!)$$

How to implement Data Structures?

- Composition: use well-known DS as an attribute of the implemented DS
- Inheritance: extends well-known DS

3.1 Linked List

- Motivation: implementation of list using array needs to occupy contiguous memory space (can result in memory error)
- Variants of linked list:
  - Tailed (need to maintain `head` and `tail`)
  - Circular
  - Doubly linked (`prev` and `next` attributes for `ListNode`)
- How to find cycle?
  Answer: use fast and slow pointers

```
   slow = slow.next;
   fast = fast.next.next;
```

- **[IMPT]** Drawing pictures is very important to visualize the program!
- Sometimes maintaining two pointers is good (especially for deletion)

```
Node head = new Node();
Node prev = head;
```

Java API: `ArrayList` or `LinkedList`

```
\\ constructor
ArrayList<Integer> list = new
 ArrayList<Integer>();
```

3.2 Stack

```
// to construct an array of generics
E[] arr = (E[]) new Object[size];
/*
// does not work
E[] arr = new E[size]
*/
```

Uses:

- **[IMPT]** Converting infix to postfix expression (Lecture 4 Slide 28)
- **[IMPT]** Evaluating postfix expression

3.3 Queue

Uses:

- **[IMPT]** Breadth-first traversal of trees
- Sliding Window (especially important for contiguous blocks of stuff)

Implementations:

- ArrayDeque: can remove from back and front
  - However Java API does not allow random access
  - Hence, need to implement our own (Lab 3C)

4 Recursion

[IMPT] Recipe for recursion (3 fingers)

1. General recursive case: identify simpler instances of the same problem
2. Base case: cases that we can solve without recursion
3. Be sure that we are able to reach the simplest instance so that we won't end up in infinite loop

Uses

- Insert item into sorted LinkedList
- Tower of Hanoi
- **[IMPT]** Combination (n choose k)
- Binary search

- Finding k-th smalles element (use pivot element p)
  move elements < p to the left of p
  move elements > p to the right of p
- Printing all permutations of a String

Overloading: same function name but with different parameters (useful in Java)

Backtracking

- Solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point in time
- *e.g.* Queens Lab 4B: board is fixed
  queens can be added or removed!

```
Pair pos = new Paid(i,j);
if (isValidBoard(pos, currentQueens)) {
  currentQueens.add(pos);
  solveQueens(n, queensLeft-1, currentQueens);
  currentQueens.remove(pos);
}
```

5 Sorting

Some definitions

- **Sort key**: use particular value of an object to do comparison and sort
- **In-place**: requires only a constant amount of extra space during the sorting process
- **Stable**: relative order of elements with the same key value is preserved by the algorithm

Some ideas used in sorting:

- Internal vs external sort
- Iterative vs recursive
- Comparison vs non-comparison based
  e.g. radix sort
- Divide and conquer

Applications

- Uniqueness testing
- Deleting duplicates
- Frequency counting
- Efficient searching

	Iterative	Recursive
Comparison	Bubble, Selection, Insertion	Quick, Merge
Non-comparison		Radix

5.1 Algorithms

5.1.1 Selection Sort

Time complexity: $O(n^2)$
Limitation: Not stable

5.1.2 Bubble Sort

Time complexity: $O(n^2)$

- Using flag: $O(n)$ `isSorted`, is the input already sorted?

5.1.3 Insertion Sort

Time complexity:

- Best case: input already sorted ($O(n)$)
- Worst case: input reversely sorted ($O(n^2)$)

5.1.4 Merge Sort

Time complexity:

- `merge(arr, left, mid, right)` is $O(right-left+1)$
- `merge` is called $\log n$ times
- Hence $O(n \log n)$

Limitations:

- Need temporary array to store values during the `merge` process (not in-place)

5.1.5 Quick Sort

Time complexity:

- `partition()`
- `quicksort(a, i, p)`
- Worst case is when it is already sorted, so the first group (elements $<$ p) is empty: $O(n^2)$
- Best case: occurs when array is divided into 2 equal halves
  - Depth is $\log n$
  - Each level takes n comparisons (including swaps)
  - Hence $O(n \log n)$ which is also the average case

Limitation: Not stable

5.1.6 Radix Sort

Treat each data as a character string: no comparison needed
Trick: sort by unit digit $\rightarrow$ tenth digit $\rightarrow$ hundredth and so on...
Time complexity:

- Initialize 10 groups (queues) to group the elements
- Complexity is $O(dn)$ where d is the maximum number of digits of the n numeric strings in the array

Limitation: Not in-place

5.1.7 Bucket Sort

How it works:

- There are b buckets, and each element `arr` is inserted into bucket according to a function *e.g.* `(int) arr[j]*10`
- Similar to radix sort but b can be any number (base?) *e.g.* Tut 5 Q 3(b) where $N \leq$ `arr[i]` $\leq 3N$, we can have $3N$ buckets $1, 2, 3, \ldots, 3N$ so that each bucket contains only 1 element
- So only 1 pass is needed a.k.a. $O(3N)$ time
- Possible problem: takes up alot of space?

	Worst Case	Best Case	In-place?	Stable?
Selection	$O(n^2)$	$O(n^2)$		No
Insertion	$O(n^2)$	$O(n)$		
Bubble	$O(n^2)$	$O(n^2)$		
Bubble (Flag)	$O(n^2)$	$O(n)$		
Merge	$O(n \log n)$	$O(n \log n)$	No	
Radix	$O(dn)$	$O(dn)$	No	
Quick	$O(n^2)$	$O(n \log n)$		No

5.2 Java Sorting

For list/arrays:

- To convert arrays to list use `Arrays.asList`
- `Arrays.sort` or `Collections.sort`

For others: use `Collections.sort(list, compObj)`

```
import java.util.Comparator;
class ObjComparator implements
    Comparator<Obj> {
  public int compare(Obj o1, Obj o2) {
    // if positive, o1 > o2
    // if negative, o1 < o2
    // if zero,     o1 = o2
  }
    public boolean equals(Object obj) {
      // check to see if we have the
    same comparator object
      return this = obj;
    }
}
```

6 Hashing

Map ADT: <key, value> pairs mapping with 3 basic operations

- **Retrieval**: retrieve value using the given key
- **Insertion**: insert/replace a value using the given key
- **Deletion**: delete the <key, value> pair using the given key

Hash Table: data structure that uses a **hash function** to efficiently map keys to values, for efficient search and retrieval

Types of Tables:

- Direct Addressing Table
  - Restrictions:
    - Keys must be non-negative integer values
    - Range of keys must be small
    - Keys must be dense
- Hash Tables
  - Map large integers to smaller integers (mod?)
  - Map non-integers to integers
  - **Collision**: hash function does not guarantee two different keys go to two different slots
    two different keys have the same **hash value**
  - Criteria of good hash functions
    - Fast to compute
    - Scatter keys evenly
    - Less collisions
    - Need less slots
  - Perfect hash functions: *one-to-one* mapping between keys and hash values *i.e.* no collision occurs
  - Minimal perfect hash functions: table size is the same as the number of keywords supplied
  - **Uniform Hash functions**: distributes keys evenly in the hash table (e.g. mod, floor function)

$$hash(k) = \left\lfloor \frac{km}{X} \right\rfloor \text{ where } k = 0, 1, 2, \ldots, X-1$$

    - Division method: map into a table with m slots
      $$hash(k) = k \bmod m$$
    - Multiplication method:
      1. Multiply by a constant real number A between 0 and 1
         **Knuth recommends $A = 1/\phi = 0.618033$ to minimize collisions*
      2. Extract the fractional part
      3. Multiply by m, the hash table size
         $$hash(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$$
  - How to choose m?
    * Pick a prime number close to a power of two
    * power of 10 or 2 are no good cos it's the same as extracting the last few digits of decimal/binary representation
  - **Hashing of Strings**
    - Summing all the characters is no good: because strings with same letters but different orders will collide
    - How to solve: take into account order of characters!

```
hash(s):
  sum=0
  for each character c in s:
    sum = sum*31+s
  return sum%m
```

6.1 Collision Resolution

$$\alpha(\text{load factor}) = \frac{n(\text{total number of keys})}{m(\text{number of slots})}$$

α measures how full the hash table is

Criteria of good collision resolution method

- Minimize clustering
- Always find an empty slot if it exists
- Give different probe sequences when 2 initial probes are the same (secondary hash function)
- Fast

6.1.1 Separate Chaining

- Use LinkedList to store keys with the same slot location
- Ideally can sort the LinkedList based on key to aid in searching

Some problems:

- find(key) and delete(key) takes $O(n)$ time
- α is the average length of the LinkedList and will increase when n increases
  - Hence it is good to keep α bounded $\Rightarrow$ reconstruct the whole table when α exceeds a bound
- Not cache friendly

6.1.2 Linear Probing

Probe sequence:

```
hash(key)
hash(key+1)%m
hash(key+2)%m ...
```

- **Insert:**
  When we get a collision, we scan linearly for the next available slot and put the key there
- **Find:**
  Probe sequence increases linearly from hash(k) until current key is equal to the key we want to find
- **Delete**
  - **[IMPT]** Cannot simply remove a value because find() only works when contiguous cells are occupied
  - So how? Use lazy deletion (three different states of a slot)
    * Occupied
    * Occupied but mark as deleted
    * Empty

Some problems:

- **Primary clustering**: Can create many consecutive slots, increasing running time of find/insert/delete $O(n)$

Modified linear probing: (d and m are co-primes) to avoid primary clustering

```
hash(key)
hash(key+1*d)%m
hash(key+2*d)%m ...
```

6.1.3 Quadratic Probing

```
hash(key)
hash(key+1^2)%m
hash(key+2^2)%m ...
```

Theorem of Quadratic Probing

If $\alpha < 0.5$(half-full) and m is prime, the we can always find an empty slot

Some problems

- When table is more than half-full, there can be endless looping!
  - To avoid table half-full, we can **resize** the table
  - Usually new $m = 2 \times m$
  - But also need to re-hash all existing keys (expensive operation)
- **Secondary clustering**: if two keys have the same initial position, their probe sequences are the same, but not as bad as linear probing

6.1.4 Double Hashing

Use a secondary hash function

```
hash(key)
hash(key+1*hash2(key))%m
hash(key+2*hash2(key))%m ...
```

Note that the secondary hash function must not evaluate to 0 (otherwise it's the same as separate chaining if not worse because of infinite loop)

$$\text{hash}_2(\text{key}) = p - (\text{key mod } p)$$

7 Trees

Some terminologies: ancestor, descendant, parent, sibling, child, root, leaf node

- Internal node: has one or more children, but root node is not an internal node
- Level of a node: level of root is 0, depends on how far it is from the root
- Height: maximum level of the nodes
- Size: number of nodes
- **Binary tree**: each node has at most 2 ordered children
- **Full binary tree**: all nodes at level $< h$ have two children, where h is the height of the tree
- **Complete binary tree**: full down to level $h-1$, with level h filled in from left to right

Implementations

- Reference based
- Array based

```
class BinaryTree {
  int root;
  int free; // free space
  TreeNode tree[];
}

```

  - What is free space?
    the last element where the slot is free, if there are multiple free slots, all the slots before the last will link towards the last one ⇒ last one is the pointer of the free
  - Representing complete tree using an array: use heap?

$$i_{left} = i * 2 + 1, i_{parent} = i//2$$

7.1 Traversals

7.1.1 Post-order traversal

Traverse the root after traversing the left and right subtrees

```
postorder(T) {
  if T is not empty then {
    postorder(T.left)
    postorder(T.right)
    print(T.item)
}}
```

7.1.2 Pre-order traversal

Traverse the root before traversing the left and right subtrees

```
preorder(T) {
  if T is not empty then {
    print(T.item)
    postorder(T.left)
```

```
    postorder(T.right)
}}
```

7.1.3 In-order traversal

Traverse the root in between traversing the left and right subtrees (**Do a sweep from left to right**)

```
inorder(T) {
  if T is not empty then {
    postorder(T.left)
    print(T.item)
    postorder(T.right)
}}
```

7.1.4 Level-order traversal

Traverse the tree level by level and from left to right (Queue is important)

```
levelorder(T) {
  if T is empty return
  Q = new Queue
  Q.enqueue(T)

  while Q is not empty {
    curr = Q.dequeue()
    print curr.item
    if curr.left is not empty {
      Q.enqueue(curr.left)
    } if curr.right is not empty {
        Q.enqueue(curr.right)
        }
}}
```

7.1.5 Evaluation of Expression Tree

Note that post-order, in-order, and pre-order of expression tree will produce postfix, infix, and prefix expressions **[IMPT]** Recursive procedure!

7.2 Binary Search Trees (BST)

Visualgo Hacks:

- Possible number of structurally different BSTs with n distinct elements

$$f(n) = \sum_{i=0}^{n-1} f(i) \times f(n-1-i)$$

  where $f(0) = 1$ and $f(1) = 1$
- Minimum number of vertices in an AVL Tree of a given height h

$$f(h) = f(h-1) + f(h-2) + 1$$

  where $f(1) = 2$ and $f(2) = 4$

Some operations: Usually $O(h)$, but note that it's possible that $h = n$ if it is skewed (hence need AVL Tree)

- Find min/max element
- Search for x
- Insertion
- Deletion (3 cases)
  - node to be deleted T has no children
  - T has only 1 child (left)
  - node to be deleted T has two children $\Rightarrow$ replace with **successor (smallest element in the right subtree)**
- Successor/Predecessor
  [IMPT] note that if this was to be implemented, we need a `.parent` attribute as an addition to the `.child` `attribute`
- Inorder traversal $\Rightarrow$ each node will be traversed not more than 3 times! wow

7.3 AVL Trees

Property:

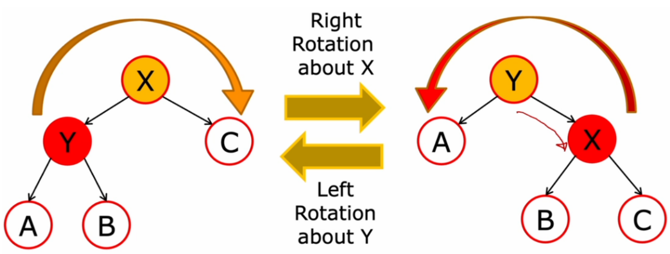
- At any node, the difference in height between left and right subtree is at most 1 (invariant)

$$|h_l - h_r| \leq 1$$

- A height balanced tree with N vertices has height $h < 2 \times \log_2(N) \Rightarrow h = O(\log(N))$
- Minimal AVL Tree of height h: having the height h and fewest possible number of nodes

Operations

- `rotateRight()/rotateLeft()`



- `insert()`: do BST to find the appropriate node to insert to, then have two cases, and also need to pass through parents to see if they are height-balanced
  - insert outside: single rotation
  - insert inside: double rotation
- Find k-th smallest item
  store `.size` attribute and do recursive stuff **[IMPT]**
  Quickselect and partition on unsorted array
  Running time:
  - BST: $O(h)$
  - Unsorted array: Best case is $O(N)$, Worst case is $O(N^2)$

7.4 TreeMap Java API

- `higherkey()`, `floorkey()`: similar to predecessor and successor
- Has sorted key in tree structure

8 Priority Queue

Property

- Insert item with a given key
- Remove the item with maximum key

Implementations

- Unsorted list: insertion takes $O(1)$ but deletion takes $O(n)$ to remove the maximum key
- Sorted list: insertion takes $O(n)$ time but deletion takes $O(n)$ time
- Heap!

8.1 Heap

VisuAlgo Hacks

- Maximum number of swaps between heap elements required to construct a max heap of n elements using the $O(n)$ `BuildHeap(arr)`: happens when the array is sorted in ascending order
- Maximum number of comparison between heap elements required to construct a max heap of n elements using the $O(n)$ `BuildHeap(arr)`

Properties:

- A **complete binary tree**
  - Either is empty,
  - , or satisfies the **heap property**: for every node v, the search key in v is greater or equal to those in the children of v
- Usually we talk about max heap
- **[IMPT]** Half of the items are leaves!
- Some access stuff

```
left(i)   = 2*i + 1
right(i)  = 2*i + 2
parent(i) = floor((i-1)/2)
```

Operations

- `heapRebuild(i)`: swap down from index i until it reaches a leaf and satisfies the heap property
- `heapify()`: build a heap from an unsorted array (utilises `heapRebuild`) and is used for heapSort
  - **Running time** $= O(n)$
  - Total number of nodes $= n = 2^{h+1} - 1$
  - Total number of bubbling down $= n - h - 1$ which is less than the total number of edges connecting the nodes
  - Worst case is when the array is sorted in ascending order (assuming max heap)

```
heapify(arr) {
  for (int i=size/2; i>=0; i--)
   {
    heapRebuild(i);
}}
```

- `heapSort`: partition the unsorted array into two parts, the heap and sorted portion; remove the max value from heap and put it into the sorted portion so that eventually the array is in ascending order
  - In-place
  - Not stable (because of bubbling/swapping operations)
  - Complexity $= O(n \log n)$

9 Graphs

Properties

- Loops are possible; trees don't have loops
- Multiple paths are possible; Trees only have 1 path from A to B

Types of graphs

- Weighted/Unweighted
- Directed/Undirected
- **Complete Graph**: Simple graph with n vertices and ${}_nC_2$ edges
- Sparse/Dense: not so many edges vs many edges (arbitrary definition)
- Disconnected/Connected

Some problems

- Shortest Path Problem: What is the shortest way to travel between A and B?
- Traveling Salesman Problem: How to minimize the cost of visiting n cities such that we visit each city exactly once, and finishing at the city where we start from?
- Topological Sort: Find a sequence of modules to take that satisfy the prerequisite requirements
- 4-Colours Problem

9.1 Implementation

Some terminologies

- In/Out Degree of a vertex: for directed graph
- Cycle: only possible if there are **[IMPT]** $geqn$ edges!!
- Path: number of edges in a path
- Adjacent vertices: $adj(v)$ is a set of vertices adjacent to v
  - For directed graphs

$$\sum_v |adj(v)| = |E|$$

  - For undirected graphs

$$\sum_v |adj(v)| = 2|E|$$

Implementations

- Adjacency matrix: space complexity is $O(V^2)$
- Edge List: list all the edges (an Edge class) in a list space complexity is $O(E)$ where $E = O(V^2)$
- Adjacency List: stores all the neighbours of vertex A; space complexity is $O(V + E)$
- Vertex Map: similar to adjacency list but uses HashMap

9.2 Traversal

9.2.1 BFS

- Use Queue for ordering
- How to differentiate visited vs univisited vertices?
  - 1D array `visited` of size V, `visited[v]=0` initially, and `visited[v]=1` when `v` is visited
- How to memorize the path?
  - 1D array `parent` of size V, where `p[v]` denotes the predecessor of `v`

- The edges will form a spanning tree
- Running time is $O(E)+O(V) = O(V+E)$ if adjacency list is used (Main loop (secondary loop) + initialization inside first loop)

$$O(\sum_v adj(curr)) = O(E)$$

- Running time is $O(V^2)$ if adjacency matrix is used

Uses:

- **Building BFS Tree**

```
    Q = new Queue
    Q.enq(v)
    mark v as visited
    while Q is not empty {
      curr = Q.deq()
      print curr
      for each w in curr.adj { // an array of adjacent nodes
        if w is not visited {
          Q.enq(w)
          w.parent = curr
          mark w as visited
    }}}
```

- **Building BFS Tree**

```
    Q = new Queue
    Q.enq(v)
    mark v as visited
    v.level=0
    while Q is not empty {
      curr = Q.deq()
      print curr
      for each w in curr.adj { // an array of adjacent nodes
        if w is not visited {
          Q.enq(w)
          w.level = curr.level+1
          mark w as visited
    }}}
```

9.2.2 DFS

- Go to each path as far as we can go
- Use a stack
- Edges used in DFS traversal will form a DFS spanning tree
- $O(V+E)$

```
S = new Stack
S.push(v)
print and mark v as visited
while S is not empty {
  curr = S.top() // top of stack
  if every vertex in adj(curr) is visited {
    S.pop()
  } else {
    let w be an unvisited vertex in adj(curr)
    S.push(w)
    print and mark w as visited
}}
```

9.3 Uses of BFS/DFS

- Searching for a vertex/node
- Reachability test: is v reachable from vertex u?
- Find shortest path between 2 vertices
- Identifying components (disconnected subgraphs)
  - Run BFS on one node, and then label the visited vertices if they can be reached from the first node *e.g.* 1,2,3,...
- **Cycle Detection**: DFS!
  - Undirected: need to make sure that the path goes back to parent node
    * Possible false positive: two-nodes cycle, need to protect against this
  - Directed: 3 states: unvisited, not completed, completed (Tut 9)
- **Topological sort**: Module selection
  - Should be a **Directed Acyclic Graph (DAG)**: a directed graph with no cycle since otherwise there will be a cycle where mods are prerequisites of each other
  - **Goal**: Order the vertices, such that if there is a path from u to v, u appears before v in the output
  - **Complexity**: $O(V+E)$

```
q = new Queue()
put all vertices with in-degree 0 into q
while q is not empty
  v = q.deq()
  print v
  remove v from G
  enqueue neighbours of v with in-degree 0
```

10 Java Tricks

- OOP is important (CardGame)
  - If it involves an array, OOP is useful, methods can just modify properties/attributes of the object class (*e.g.* `reversed=true; increment=4`)
    * Especially true if only need to print statement at the end
- Invariance: property that stays constant???
  Lab 5C: Pancakes: Use number of inversions if it's even or not
- Use `StringBuilder` for return statements
  - Java StringBuilder API
  - Zigzag conversion
- Tut 07 ⇒ Contiguous cells can be represented by start and end only
- **[IMPT] Iterator**

```
LinkedList<Integer> ll = new LinkedList<>();
Iterator<Integer> it = ll.iterator();
```

```
it.next();
// must call next first before
    removing;
it.remove();
```

```
it.next();
// must call next first
it.remove();
```