Array Rotation

Given an array, `A`, and index, `n`, rotate `A` by `n`.
For example, `n = 3`, `[1,2,3,4,5,6,7]` is rotated to `[5,6,7,1,2,3,4]`.

1) Reverse left partition (`L.length = A.length - n`)
   - `[4,3,2,1,5,6,7]`
3) Reverse right partition (`R.length = n`)
   - `[4,3,2,1,7,6,5]`
5) Reverse whole array
   - `[5,6,7,1,2,3,4]`

Time: $O(n)$
Space: $O(1)$

K Largest Element

Given an unsorted array A, find the k^{th} largest element.

Max Heap: heapify to max heap, then poll heap k times
Time: $O(n + k \log n)$, worst case $k = n$
Space: $O(1)$

Quick-select
Time: $O(n^2)$ worst, $O(n)$ average
Space: $O(1)$

Next Largest Permutation/Number

eg. given `[5,3,1,7,6,4,2]`, output `[5,3,2,1,4,6,7]`

1) Starting from right, find first number smaller than right neighbour (`1` in this case)
2) Find smallest number bigger, and to the right, of the number found in step 1 (`2` in this case)
3) Swap the numbers found in step 1 and 2 (`[5,3,2,7,6,4,1]`)
4) Reverse array for index more than the number found in step 1 (`[5,3,2,1,4,6,7]`)

Time: $O(n)$
Space: $O(1)$

Reverse Stack/Queue Recursively

Given only a stack or only a queue, reverse them *without* the use of any additional auxiliary data structure.

> Hint: make use of the call stack to store data

```
void reverse(A) {
    if (A.isEmpty()) {
        return; // stop when A is empty
    }

    int current = A.poll();

    reverse(A); // recursively call reverse and add to call stack

    A.add(current);
}
```

Time: $O(n)$
Space: $O(n)$

Reverse Singly Linked List

```
Node reversed_list = null
Node current = head
while current is not null
    Node next = current.next
    current.next = reversed_list
    reversed_list = current
    current = next
head = reversed_list
```

Time: $O(n)$
Space: $O(1)$

Height of a Binary Tree

```
function height(tree)
    if tree is null
        return 0

    heightLeft = height(tree.left)
    heightRight = height(tree.left)

    return max(heightLeft, heightRight) + 1
```

Time: $O(n)$
Space: $O(n)$

Diameter of a Tree

1. Run BFS to find the furthest node, A, from given source node
2. While keeping a counter for distance travelled, run BFS from A to find furthest node, B, from A
3. Diameter is distance from A to B

BFS: $O(V + E)$

Assuming Binary Tree, where $edges = vertices - 1$,
Time: $O(n)$
Space: $O(n)$

Least Common Ancestor (Binary Tree)

Intuition, given 2 nodes, x and y:

1. If current node is x or y, then LCA is current node
2. If x is found on left subtree and y is found on right subtree, then LCA is current node
3. If a node x is found, and y isn't, it means y is child of x and y is LCA
4. Else, recursively call this algorithm on left and right subtrees

```
# This assumes x and y can be found in the tree
function findLCA(root, x, y)
    if root is null
        return null

    # Case 1: current node is x or y
    if root == x or root == y
        return root

    # Look for keys in left and right subtrees
    leftLCA = findLCA(root.left, x, y)
    rightLCA = findLCA(root.right, x, y)

    # Case 2: nodes found on either side of tree:
    if leftLCA != null and rightLCA != null
        return root

    # Case 3: one node is child of the other:
    if leftLCA != null
        return leftLCA
    else
        return rightLCA
```

Time: $O(n)$
Space: $O(n)$

Flatten BST

1) In-order traversal
2) Perform right rotation on node if there exists left child on node

Time: $O(n)$
Space: $O(1)$ (nodes are simply moved about, not created)

2 Array Median

Given two arrays of sorted integers A and B of size N and M respectively, give an efficient algorithm to find the combined median of all the values in A and B.

- Mergesort: $O(n \log n)$
- Merge: $O(n)$

```
[ {COUNT1}, TL, TR, {COUNT2} ]
[ {COUNT1}, BL, BR, {COUNT2} ]
```

Invariants:
1) COUNT1 == COUNT2
2) TL < BR
3) BL < TR

Time: $O(\log n)$

K-way Merge

Given k sorted list, with the total number of elements in all the list being n, give an efficient algorithm to obtain a sorted list, S, of size n with all the elements from the k sorted list.

```
public ListNode mergeKLists(ListNode[] lists) {
    PriorityQueue<ListNode> pq = new PriorityQueue<>((x, y) -> x.val - y.val);

    // insert first element of each list to min heap
    for (int i = 0; i < lists.length; i++) {
        pq.add(lists[i]);
    }

    // create new empty linkedlist (note: nodes are moved here, not created)
    ListNode head = new ListNode(0);
    ListNode toReturn = head;

    while (pq.size() > 0) {
        // poll smallest element from min heap
        ListNode current = pq.poll();
        // add next element of smallest element if exist
        if (current.next != null) {
            pq.add(current.next);
        }
        // shift head pointer
        head.next = current;
        head = current;
    }
    return toReturn.next;
}
```

- Max heap size = $O(k)$ at any point of time
- Each Insert/ExtractMin operation is $O(\log k)$
- This is done n times

Time: $O(n \log k)$
Space: $O(k)$

Line Intersection

Suppose you are given two sets of 2-dimensional points $P = \{p_1, p_2, \ldots, p_N\}$ and $Q = \{q_1, q_2, \ldots, q_N\}$. Connect each point p_i to the corresponding q_i. Give an efficient algorithm for determining how many pairs of these line segments intersect.

Similar to counting total number of inversions in an array.
Time: $O(n \log n)$

Bipartite Graph

Check if a given graph, G, is a bipartite graph (graph whose vertices can be divided into two disjoint and independent sets U and V such that every edge connects a vertex in U to one in V)

1. Start from any node, colour it red
2. While there are still uncoloured nodes,
    1. Colour all neighbours of red nodes blue
    2. Colour all neighbours of blue nodes red
3. If there are any 2 adjacent nodes with the same colour, bipartite graph does not exist
4. Else, the 2 coloured sets represent the 2 required set of nodes

Longest Path in a DAG

1. Use Topological Sort to get topological ordering of nodes
2. Init all dist estimate as $-\infty$
3. For every vertex u in topological order,
    - For every adjacent vertex v of u,
        - If $dist[v] < dist[u] + weight(u, v)$,
            - $dist[v] = dist[u] + weight(u, v)$

Time: $O(E + V)$

> Note: relax condition is swapped to $<$ instead of $>$

Shortest Path, Edges Have Exactly 2 Weights: 0 and K

1. Use deque (doubly-ended queue) instead of queue
2. Push 0-weight edge to front, K-weight edge to back of deque
3. Do BFS, but pop from front of deque (ie. keep visiting 0 first)

Time: $O(V + E)$

> Only works if one of the edge is 0 weight

Subset Sum Problem

Given an array, A, check if any subset of A sums up to a given number

```
public boolean bfs(int[] arr, int target) {
    Arrays.sort(arr);

    LinkedList<Node> queue = new LinkedList<>();
    queue.add(new Node(0, -1));

    while (!queue.isEmpty()) {
        int numLoops = queue.size();
        for (int i = 0; i < numLoops; i++) {
            Node curr = queue.poll();
            for (int j = curr.arrIdx + 1; j < arr.length; j++) {
                int sum = arr[j] + curr.sum;
                if (sum == target) {
                    return true;
                } else if (sum < target) {
                    queue.add(new Node(sum, j));
                }
            }
        }
    }
    return false;
}
```

Brute force BFS approach
In worst case, all possible subsets (all nodes in the tree) have to be visited
Number of nodes $= 2^n$
Time: $O(2^n)$