Order of Complexity

Best Case: $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(c^n) < O(n!)$:Worst Case

For forLoops, count the amount of times the loop is called. Multiple it with the amount of operation in the loop.

Note: This is $\frac{n(n+1)}{2}$

```
for(int i=1; i<=n; i++){
        for(int j=0; j<i; j++){
                some operation;
        }
}
```

Given the time to process an operation, it would take $\frac{time * new\ operation}{original\ operation}$ to process the new operation.

New operation is calculated by plugging in the number of operations to the original number of operations. If original number of operations is a and new number of operation is b.

F(a) = a-1 then F(b) = b-1

ArrayList

An array which decreases and increases the size according to the amount of items.

When adding an item into an arrayList, the size of the array is increased by 1.
Copy all items from the old array to the new array.

When removing an item from an arrayList, the size is decreased by one.
Copy all items but that one item which is removed into the new array.

The complexity of removing and adding an array is O(n) which n is the amount of items.
The reason is because the program has to copy everything from the old array to the new array.

To find the size of the arrayList, you can have a variable that increments or decrements as you add/remove item. You can also get the length of the arrayList.

Linked List

There are 3 types of linked list.

1. Singly-Linked list: Only points to the next node.
    - Use this to save memory if the program works.
    - Use getLink() to get the next node.
    - Must have a reference to the head.
    - May or may not have tail reference
    - Finding a node: O(n)
    - Wanting the last node: O(n)
        - Must go through the whole list to get to end
    - Removing the first node: O(1)

```
head = head.getLink();
```

    - Removing the last node without tail reference: O(n)

```
// This is only an example. You still have to check if tail = head.
Node cursor = head;
// The node before the tail. Going through the whole list.
while(cursor.getLink().getLink() != null){
        cursor = cursor.getLink();
}
cursor.setLink(null);
```

    - Removing the last node with tail reference: O(n)
2. Circular-Linked list: Same as singly-linked list but the tail is linked to the head.
    - Usually have reference to head or tail but not both
    - Adding node before head with tail reference no head reference: O(1)

```
// tail.getLink() is the head.
newNode.setLink(tail.getLink());
tail.setLink(newNode);
```

    - Adding node after tail with tail reference no head reference: O(1)

```
// Next of newNode is head
newNode.setLink(tail.getLink());
Tail.setLink(newNode);
tail = newNode;
```

    - Removing head with tail reference no head reference: O(1)

```
Tail.setLink(tail.getLink().getLink());
```

    - Removing tail with tail reference no head reference: O(n)

```
// Code is not complete. Check if tail is the only node.
Node cursor = tail;
// Go to the node before the cursor.
while(cursor.getLink()!=tail){
        cursor = cursor.getLink();
}
// If only two nodes, the node before the cursor will point to itself
cursor.setLink(tail.getLink());
tail = cursor;
```

3. Doubly-Linked list: Points to the next node and previous node
    - The node before the head is null
    - The node after tail is null
    - Use getNext() to get the next node.
    - Use getPrev() to get the previous node.
    - Finding the first/last node is: O(1)
        - Only if there is a head/tail reference
    - Adding new node at head with head reference: O(1)
    - Adding new node at tail with tail reference: O(1)
    - Adding new node at tail without tail reference: O(n)
    - Removing the first node with head reference: O(1)

```
head = head.getNext();
head.setPrev(null);
```

    - Removing the last node with tail reference: O(1)

```
tail = tail.getPrev();
tail.setNext(null);
```

- Removing the last node without tail reference: O(n)

Has a node (object) which refers to the next or previous of the list. Unlike an array which all the addresses are right next to each other, a linked list is stored randomly and needs a reference to get to the next node.

Finding a node is: O(1) if it is the head
O(n) if it is somewhere in the middle of the list or the tail

Inserting a node somewhere in the middle of the list is O(n)
No reference other than head and tail. Find the location to place the newNode.
Counting the amount of nodes without any additional variable is O(n) b/c go through whole array
Declare a variable and set equal to zero. When adding a node, increment the variable by 1. When removing a node, decrement the variable by 1.

Use getData() to get the data of that node.

ArrayList vs. Linked List

Initial size is unknown: Linked List
Elements are accessed randomly or frequently(not at same position): ArrayList

Stacks (First in Last Out)
Use push (add) and pop (remove) in an array or linked-list
When popping, the last element which was put in will be taken out.
The remove method returns the element which was removed.

Array stack:

- Bottom is 0 index. Top is the last index which has an element.
    - Have a reference to the top of the index so it is easier to access.
    - Pushing and popping is O(1)
- If you use arrayList, complexity of pushing and popping is O(n)

Linked-list stack:

- Use singly-linked saves memory works the same.
- Reference to the head which is called top.
    - Push: O(1)
      newNode.setLink(head);
      head = newNode;
    - Pop: O(1)
      head = head.getLink();

Queues (First in First Out)
Uses enqueue (add) and dequeuer (remove). The first element in is the first element out.

Array Queue:

- Have a reference to the front and rear. Remove from front and add from rear.
  When rear is full, go to the front of array and place.
- Enqueue & Dequeue is O(1)
- Note: (rear+1)%CAPACITY == front lets you know if the whole array is full.
  In such case, you cannot add anymore elements

Linked-list Queue:

- Use singly-linked saves memory works the same.
- Head = front. Tail = rear.
- Enqueue: O(1)
  rear.setLink(newNode);
  rear = newNode;
- Dequeue: O(1)
  front = front.getLink();

Priority Queue:
Enqueue and dequeuer equals O(1) or O(n) but cannot be both.
Depending on how you want to add and remove.
For linked-list, it is better if enqueuer by priority

Recursion:

- When a method calls itself over and over.
- Must have base case. The stopping case
- All recursion methods can be written using a loop.

Tail Recursion:

- When the method calls the previous determined value
- n = 4 + (n-1) but what is (n-1)?

Binary Tree:

Has a root which is the 0 index in an array.
Depth of the tree is the longest path from root to all leaves (no repetition allowed)
Note: The root is at depth 0!

If it is a 2-ary tree, the formulas are as followed (i = index of element)

Left child = 2i+1

Right child = 2i+2

Parent = $\frac{i-1}{2}$

Full Binary Tree:

$n = 2^{d+1}-1$

- All leaves are the same depth
- All nodes have 2 children except the leaves
- If tree is full, it is complete!

Complete Binary Tree:

- All leaves are only 1 depth index apart. And the greater depths are towards the left of the tree.

Preorder: (Going around the tree and pass left)
	Root -> left -> right
Inorder: (Going around tree and pass under)
	Left -> Root -> Right
Postorder: (Going around tree and pass right)
	Left -> Right -> Root

```
Public void order(){ // Place System.out.println(data) To print the tree in such order
	// Pre Order
	if(left != null)
		left.order();
	// In Order
	if(right != null)
		right.order():
	// Post Order
}
```

Binary Search Tree

All nodes towards the left of that node is less than that node.
All nodes towards the right of that node is greater than that node.

The smallest value is the most bottom left value. The largest value is the most bottom right value.

When inserting into a binary search tree, order matters. The new Node will always end up as a leaf.
When removing a root from a binary search tree, look for most right node in the left subtree to replace the root. If not possible, try the right subtree most left node.
Note: The most left or most Right nodes MUST be a leaf.

Different amount of forms a binary tree can take is
$b_n = b_0 * b_{n-i-1} + b_1 * b_{n-i-2} + \ldots + b_{n-i-1} * b_0$
so 0 1 2 3 … n
and n n-1 … 2 1 0

Heap

- The maximum value is the root.
- All children of that node has a smaller or equal value.
- Follows the complete binary tree format.
- Inserting a node
  - Place node in the most right position such that it would still be a complete binary tree.
  - Check if the parent has a smaller value.
    - If yes, swap.
      - Is new Node at root? If yes, stop.
    - If no, stop.
- Removing a node
  - Replace the root with the deepest, most right position.
  - Check which of the children is the larger value. Swap. Continue until node is a leaf.

B-Tree

- The minimum value is the minimum element each node in the tree can have.
  - The root does not follow this rule. It can have as few as one element.
- The maximum is automatically decided. Minimum * 2.
- The children one node MUST have is the amount of element in node + 1.
- The first element in the parent node is ALWAYS larger than its first children node.
- It's better to look at a drawing of a B-tree and determine where each element goes.

2-3-4 Tree

- If the node is not a leaf, it must have 2/3/4 children.
- The placement of each element follows the B-tree rules.

Red-Black Tree

- Root is always black
- Red node = black children
- Red to black always has same amount of black

AVL Tree

- Is a BST such that the depths of the leaf are at most 1 depth apart.
- Rotation is used when above rule is broken.

Search Note: Dealing with arrays

- Sequential/linear: Searches an array one by one until target is found
  - Best Case O(1) target is at the first index
  - Worst case O(n) target is not at the first index

- Binary:
  - Note: must be sorted.
  - Looks at middle. If target == middle return
    Otherwise, split array in half and look at middle
    Continue until either only 1 element in array left or target found.
  - O(log n)

- Hashing:
  Load Factor: Elements in the array / Size of array.
  In this case, arrays are called tables.
  - Linear Probing: $\frac{1+\frac{1}{1-a}}{2}$ avg search time
    - Uses H(k) to figure out a position num % usually prime num
    - If position of H(k) is taken, continue the array from that position until an empty space is found.
    - Have another array with the same length that stores boolean values. When inserting a value, change value from false to true. This way, when searching for the target, program will not stop at an empty space but rather continue searching.
    - Without collision, insertion will take O(1) Otherwise, O(n)
    - Searching/Removal: Best Case: O(1) Worst Case: O(n)

  - Quadratic Probing: $\frac{-\ln(1-a)}{a}$
    - Similar to Linear Probing but instead, when collision occur,
      - Add 0^2, 1^2, 2^2, 3^2 to the result and check if that space is available

  - Double Hashing:
    - Has 2 functions $H_1(k)$ and $H_2(k)$ such that if $H_1(k)$ has been used, try $H_1(k) + H_2(k) = m$ and try $H_1(m)$ if occupied, try the last 2 ans with the first formula.
    - Choosing what to mod. Use prime numbers.

  - Chained Hashing: $a + \frac{a}{2}$
    - Load factor can be over 1.0
    - In an array, if there is a collision, add to it using linked list.

Sorting These are all sorted in increasing order.

- Selection Sort: Search for the smallest value and move it to the first position
  - $O(n^2)$
  - Have a variable which keeps track of the min index.
  - Have a for loop which loops through the whole array. Call this loop i.
  - Within the for loop, have another for loop (call this loop j) which loops from i to the end of the array. In this loop, look for the minimum value in the array and sets min to that index (i).
  - Within i and under j, swap the values at position min and i.

```
// arr is the array we have to sort. n is the size of arr.
public static int[] selection Sort(int[] arr, int n){
        int minLocation;
        // Go through the whole array.
        for(int i=0; i<n-1; i++){
                minLocation = i;
                for(int j=i; j<n; j++){
                        if(arr[j] < arr[minLocation])
                                minLocation = j;
                }
                if(minLocation != i){
                        // swap
                        int temp = arr[i];
                        arr[i] = arr[minLocation];
                        arr[minLocation] = temp;
                }
        }
        return arr;
}
```

- Insertion Sort: Goes through the array and if the value is less than the previous value, swap the positions and keep going.
  - $O(n^2)$

```
// powerpoint code.
public static int[] insertionSort(int[] arr, int n){
        int item;
        for(int i=1; i<n; i++){
                item = data[i];
                int j = i;
                while(j>0 && data[j-1] > item)
                        data[j] = data[j--];
        }
        return arr;
}
```

- Bubble Sort: If the data of the current position is greater than the position after it, swap. Do this the array size amount of times.
  - $O(n^2)$

```
// This is the easy to memorize but longer way.
// The Fanny way of doing a bubble sort
public static int[] bubbleSort(int[] arr, int n){
        for(int i=0; i<n; i++)
                for(int j=0; j<n-1; j++)
                        if(arr[j]>arr[j+1]){
                                int temp = arr[j];
                                arr[j] = arr[j+1];
                                arr[j+1] = temp;
                        }
        return arr;
}

// This is the power point code which is harder to understand but less
// comparisons
public static int[] bubbleSort(int[] arr, int n){
        for(int i=0; i<n-1; i++)
                for(int j=n-1; j>I; j==)
                        if(data[j] < data[j-1]){
                                int temp = arr[j];
                                arr[j] = arr[j-1];
                                arr[j-1] = temp;
                        }
        return arr;
}
```

// Too much work to post codes here. Google or look at code FolderGUI (Email HW)

- Merge Sort: Break array into halves. Keep breaking. Then merge them. While merging, sort them in order such that comparing the first digits in each array and inserting them in the array.
  - O(n log n)
- Quick Sort: Choose a pivot. All values smaller to the left all values larger to the right. Choose another pivot. Do the same until pivot is the only value left.
  - O(n log n)
- Heap Sort: Finding the largest value and putting at root.

Graphs

Edge the line that connect two nodes.
Vertex the node.
Adjacent = neighbors
Path = way of getting from one point to another.
Cycle = Starting at one vertex and ending on the same vertex. If it's a cycle, it's a path.
Degree = The amount edges one vertex has.

Not everything in these two charts are needed. In fact, only know the definition of simple graph.

Type of graphs	Edges	Multi-edges	Loop
Simple	Undirected	No	No
Multigraph	Undirected	Yes	No
Pseudo	Undirected	Yes	Yes
Simple Directed	Directed	No	No
Directed multi	Directed	Yes	Yes
Mixed	Both	Yes	Yes

Graphs		vertices	edges	Bipartite when	Degree Seq
Complete	K_n	n	$\frac{n(n-1)}{2}$	$n \geq 3$	n-1,n-1…
Cycle	C_n	n	n	$n \geq 3$	2,2,2…..
Wheel	W_n	$n+1$	2n	$n \geq 3$	n,3,3,3….
n-dimensional cube	Q_n	2^n	2^{n-1}	$n \geq 2$	
Star	S_n	$n+1$	n	Always	n,1,1,1….
Complete bipartite	$K_{m,n}$	m+n	m*n	Always	

Depth first: Pick the root. Pick an adjacent node. Make a path. Collect all the nodes you missing in depth first order. Go to root pick another path. The end

Breadth first: The root. All the adjacent nodes of root. All the adjacent nodes of those. Keep going.