Dynamic Programming: simplifying a complicated problem by breaking it down into simpler sub-problems in a [recursive](https://en.wikipedia.org/wiki/Recursion) manner.

Difference in Interface and Abstract class.

Difference in Implement and extend

Interface: A class with methods that have no implementations.

You implement the Interface with the keyword “Implement” in a class.

The key with the interface is that when you implement the interface, you must include all functions inside the class that implements the interface.

Abstract: It can be a normal class, but must be named abstract, and if abstract then it must contain AT LEAST 1 abstract method with no implementation inside that method. You use the abstract class with the keyword “extend”. And now you can use any function from the abstract class without having to create the object of that class.

Difference between Array and ArrayList.

* **Array:**
  + Simple fixed sized arrays that we create in Java.
  + Is a fixed size data structure.
  + Can contain both primitive and data types.
* **ArrayList<>:**
  + Dynamic sized arrays that implement list interface in Java.
  + Part of a collection framework in Java.
  + Has a set of methods to access elements and modify them.
  + Not a fixed sized data structure.
  + Only supports Object entry’s, not primitive types.

Data Structures:

**Performance:**

* **Time:** Number of processes, operations.
* **Space:** Memory needed by code to store info, disk space needed for persistent storage.
* **Network:** Bandwidth code uses to pass info to clients or machines.
* **Complexity:** 
  + A measure of how resource requirements change as the input size gets larger.
  + Higher complexity effects performance.
  + Measured by worst case.
* **Arithmetic Operations:**
  + Read, assign, write, test.

**Big-O Notation:**

* **O(1) :** When complexity doesn’t change no matter what input size.
* **O(N):** If time taken when the Algorithm increases linearly when N increases.
* **O(N^2):** If time taken in the algorithm increases quadratically when N increases.
* **O(logN):** If it takes half the time to complete.

**LinkedList:**

* **Complexity:**
  + Insert-end-of-List: O(N) Traverse through entire list to insert element.
  + Insert-beginning-of-list: O(1) Insert right at the head.
  + Find element: O(N) Traverse each node and compare for element in search.
  + Delete 1st element: O(1) Delete head.
  + Delete random element O(N): Traverse through elements to find the one to delete.
* Using import java.util.\*
* Elements are linked to, or references to the next element, they chained together.
* LinkedList<T> Standard Java library.
  + Starts: ->Head, Ends: Tail -> null.
  + Each node contains data, nextNode
  + Node class. getNext(), setNext(), getData()
  + Pop(): Returns the first (head) value in the list.
* **Use LinkedList for:**
  + When having a large number of insert/delete operations.
  + When you have no idea how large the list might be.
* **Use Arrays for:**
  + When read operations need to be extremely fast.
  + When needing random access to elements.

**Stacks:**

* **Complexity:**
  + Push() : O(1)
  + Pop(): O(1)
  + isEmpty(): O(1) (With the helper of getSize() else is O(N))
  + isFull(): O(1)
* Best data structure for a stack is a LinkedList.
* **Stack:**
  + Last element added is the first one accessed. (LIFO)
  + Top(): Focused on one end of the stack .
  + Push(): Adds new element to the top of stack.
  + Pop():Remove element from top of stack.
  + Peek():View top element of stack.
  + isEmpty():Helper method that checks if its empty.
  + isFull():Helper method that checks if its full.
  + getSize(): The number of elements present in stack.
* **Use Stacks for:**
  + Undo, redo operations.
  + Back button in browser.
  + Holds memory for recursive calls.
  + Translating infix notation for expressions to postfix.
* **Ex:**
* **Find min value in a stack.** 
  + You need 2 stacks. minStack, regStack.

**Queues:**

* **Complexity:** 
  + Enqueue: **O(1)**
  + Dequeue: **O(1)**
  + isEmpty: **O(1)**
  + isFull: **O(1)**
* Elements added to end of queue, removed from beginning of queue. (FIFO)
* Enqueue: adds the element to the queue.
* Dequeue: removes element from beginning of the queue.
* You can peek() for the first element in the queue.
* Offer(): Offers element to queue if there’s space.

* **Circular Queue:** 
  + Can be used as an array, last element wraps around to the first element.
  + Contains special value: head = -1, to denote emptyList.
  + Only one element when head and tail are in the same index.
* **Can be used in:**
  + Customer service hotline.
  + Jobs to be printed.
  + In order processing systems. (transactions in atm, deposit first then you can withdraw.)
* **Ex.**
* **Implement queue using 2 stacks.** 
  + **Complexity:**
    - O(M): number of operations performed in the queue.
  + Create **Stack<T> forwardSta**ck to push enqueues.
  + Create **Stack<T> reverseStack**. Holds elements in reverse from f **Stack<T> forwardStack.**
  + Pop all elements from **Stack<T> forwardStack** then push into reverseStack, then dequeue.
  + We operate the full enqueue process via the **Stack<T> forwardStack**.

**Sorting Algorithms:**

* **SelectionSort:**
  + **Complexity:** 
    - **Comparisons: O(N^2)**
    - **Swaps: O(N)**
  + Selects one element for every iteration, compare with every element in list, find the smallest, then swap with first in the list, continue.
  + Move to the next index, repeat.
  + Once first element is in place, we don’t go back to compare it again.
  + Not stabled, if 2 equal values are in list, the order can get rearranged.
  + Not adaptive, can’t break out of loop even if the list is sorted already.
* **BubbleSort:**
  + **Complexity:**
    - **O(N^2)**: Swaps
    - **O(N^2)**: Comparisons
  + Each iteration, every element is compared to its neighbor.
  + Stable, doesn’t require iteration if the list is sorted already, break out of the loop if no swaps have been done.
  + Horrible, processes additional passes over all elements.
* **InsertionSort:**
  + **Complexity:**
    - **O(N^2):** Worst case if in descending and we want ascending.
  + Start with sorted sub-list, size 1.
  + Insert new element, compare to its neighbor, swap until it’s in place.
  + Stable sort, if no swap, you can break out of inner loop.
* **ShellSort:**
  + **Complexity:**
    - **O(N^2):**
  + Takes in InsertionSort and processes it differently.
  + Partitions original list into sublists.