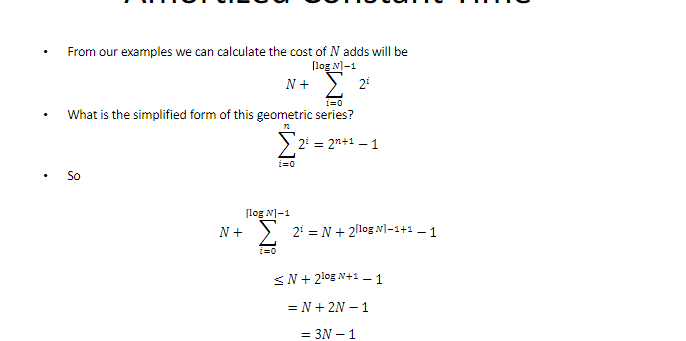
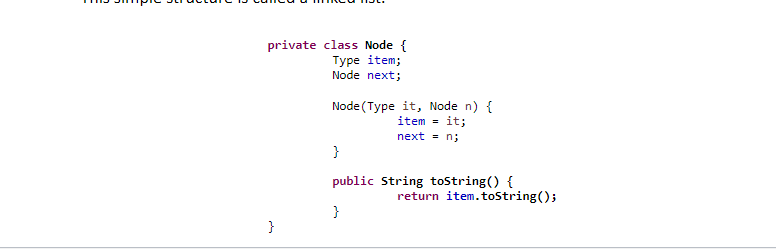
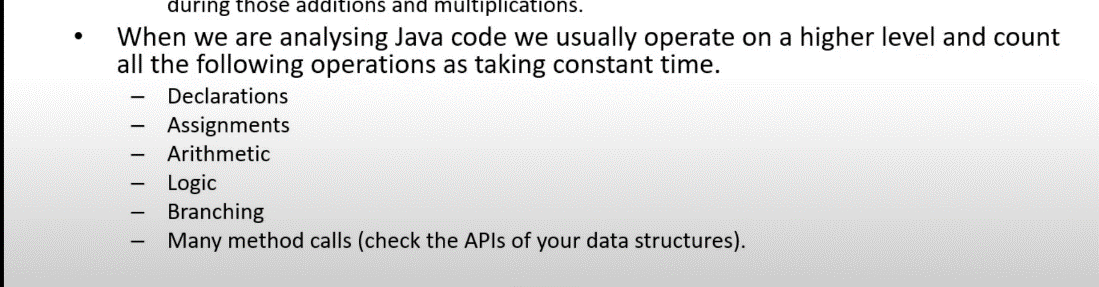
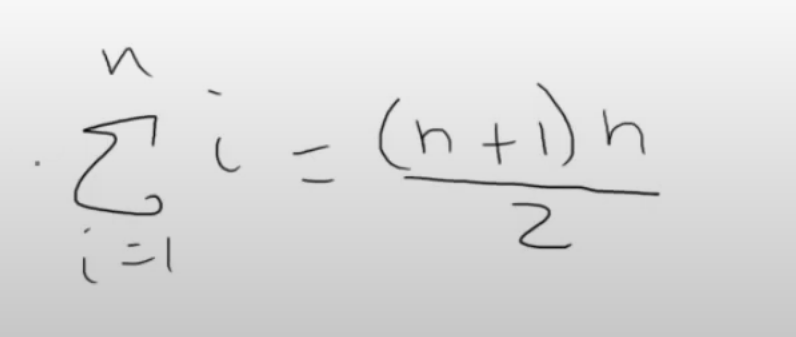
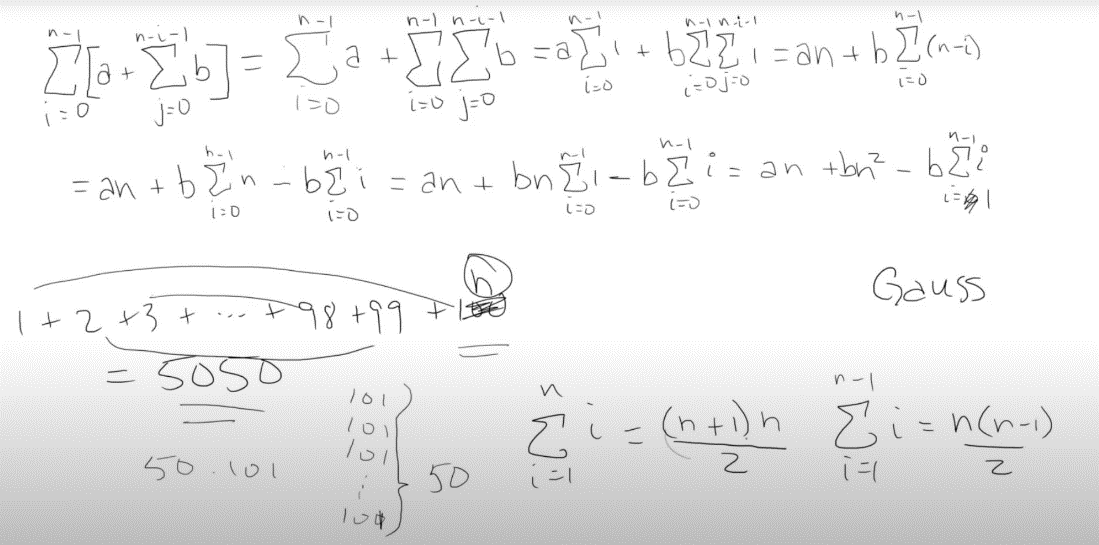
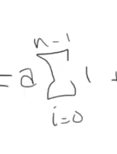
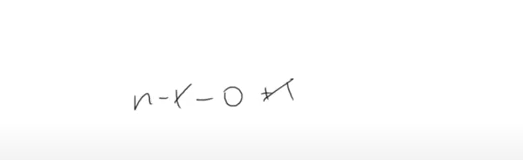
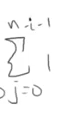
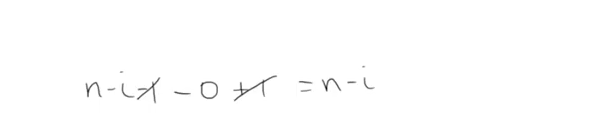
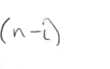
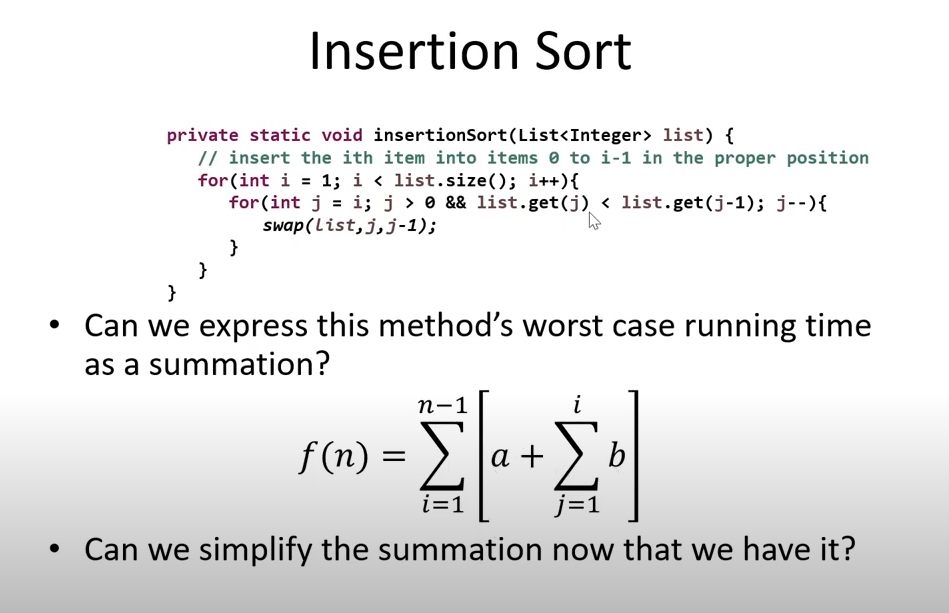
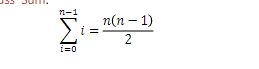
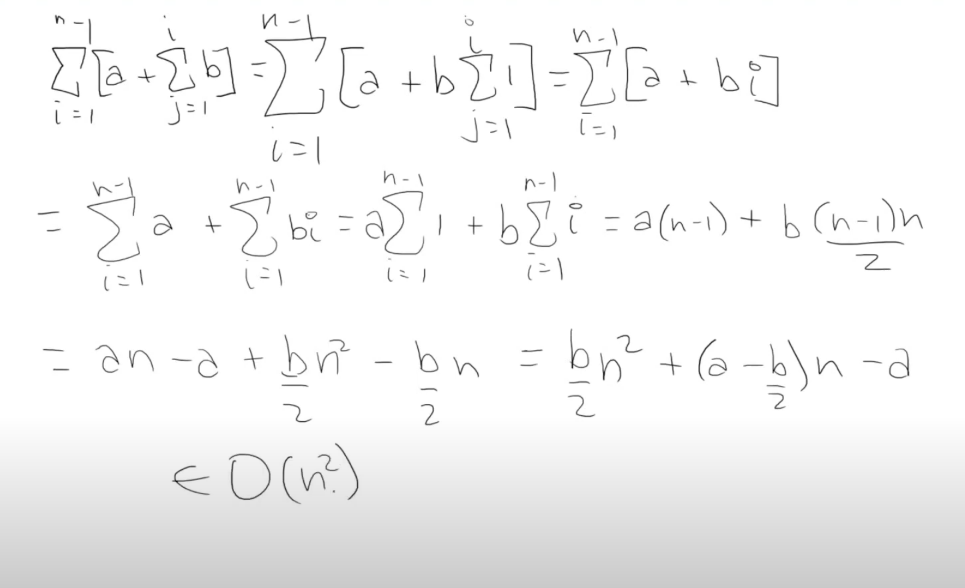
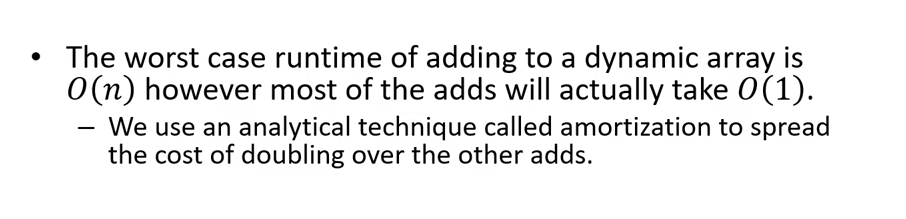
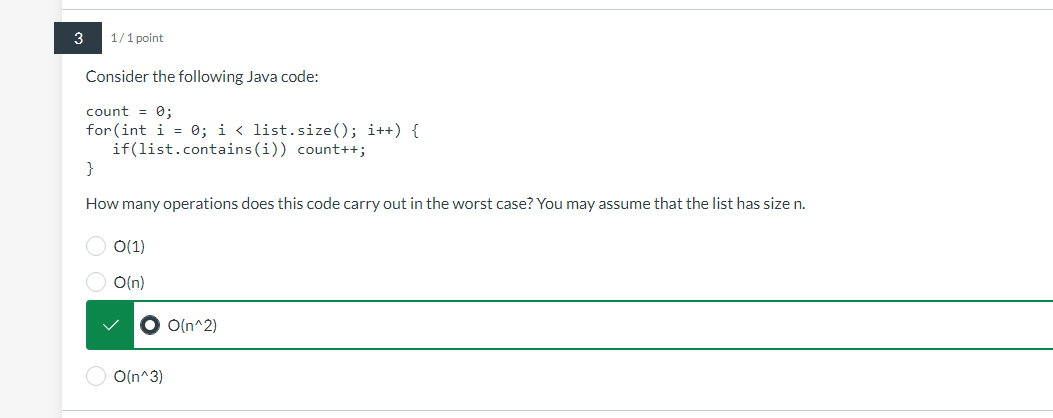
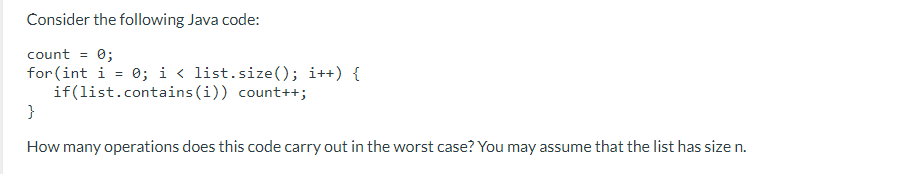
Unit 1

* The benefit of using an array is random access (uses constant time access, in other words, uses the same unit of resources to access memory address in the array.)
* To access memory address of elements would do
* Address of array + 4 \*index of element.
* Since each integer of in a int[] = new int[size ]
* Holds 32 bits or 4 bytes of memory.
* Address of array us address of first element.
* If our array runs out of space we might want to extend memory region, but that space of memory after n-1 is probably occupied by some other program data.
* Static Arrays can’t grow in size
* So we would make a brand-new array and copy the contents of array over.
* Wouldn’t want to do this to often because it takes too much data and memory, it would take O(n) operations.
* So would then switch over to an ArrayList in java, which is a dynamic array.
* Dynamic Arrays like Java’s array list will begin with an Initial Capacity
* Can be set to have a capacity of 1 but that’s uncommon.
* Normally a small power of 2 like 2^4 = 16.
* Dynamic Array keeps track of capacity and size of array.
* Elements can be added simply when size of is less than capacity. Takes O(1) or constant time.
* Need to grow array when size reaches capacity.
* First we declare new array with twice the capacity,
* Then we copy all elements from old array to new array.
* This takes O(n) or linear time.
* In the worst case it will take O(n) operations, but normally it would cost O(1).
* Will analyze using amortization to spread cost of doubling over the other adds.
* During Amoritization we stop doubling when the current value we are adding is just before the size of the adds or previous size
* Cost is not always going to be exactly double the old capacity.
* 
* 
* 
* 
* 
* Simplifying expressions
* 
* Subtract upper bound by lower bound and add 1
* 
* To get
* 
* do
* 
* to
* 
* Insertion sort:
* Remember for Gauss sum
* Text, letter

  Description automatically generated
* N is the upperbound, and n+1 is the first term and the last term added together.
* Changed the 0 into a 1 in the work above, since it means were just adding 0.
* N-1 -1 =N times n-1 pairs and divide it by 2
* 
* A is the outer constant where the for-loop tests and updates and b is the swap operation and nested for loop updates.
* 







Calculating run time of code as a summation.

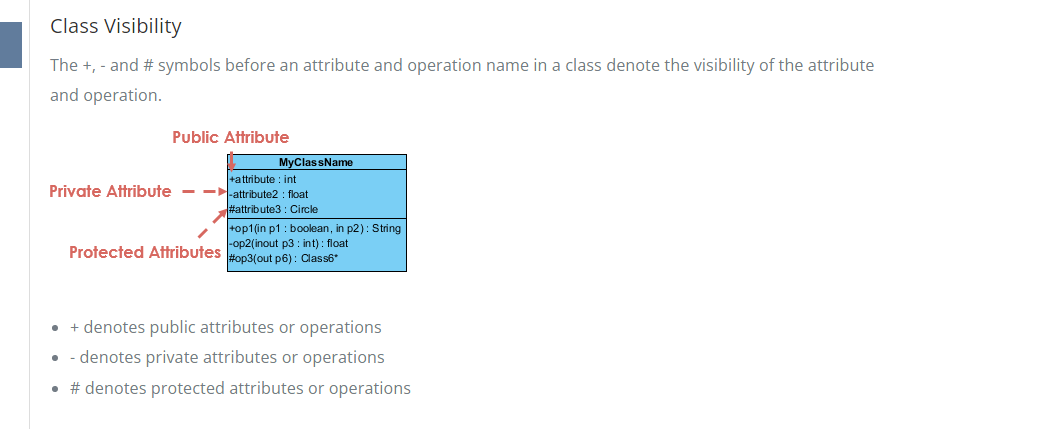
A+

A+B

A+ B(N)

A+ B(N) \*N

A+ B(N2)



Text

Description automatically generated

M = index value

N= size of list.

A + [**B** + ]

A + [**B** + ]

A + [**B +**

A + [**B +**  (m-1) –(n) +1]

A + [**B +**  (n-m) ]