For question 1, the goal of this problem is to have a custom class Vector and a Linked List and test which of the two data structures is faster by filling them up with random integers, random strings (using ascii values) and move semantics. I created a templated Vector class which contains three constructors: a default constructor, and two constructors that take in values. I had also implemented the big three, a copy constructor, an assignment operator and destructor. I wrote the functions for size (this is for the vectors length), front (this will be the first index of the vector), back (this will be the last index of the vector), clear (this will clear the vector), pop_back (this will delete the last element in the vector) and push_back (this will add an element to the vector). I also created an at and operator [] function so when we push_back or do any of the other functions, we can use [] to get the index, [] uses the "at()" function. I also have private functions and variables, allocate(), release(), _capacity, _size and _vec.

I used Professors Fried's code for the Linked List and the iterative function. I didn't create an iterative function for my vector class since I can use a simple for loop and push back on it. Outside of the classes, I created five functions. A function that randomly generates a number, a function that will randomly fill a vector with ints, a function that will randomly fill a linked list with ints, a function that will randomly fill a vector with chars and a function that will randomly fill a Linked List with chars. Although the instructions did say to fill it up with strings using ascii values, I later realized that we couldn't use strings and had to use chars for the ascii values. One idea to make strings is to concatenate the chars. When running it in the main, we can see that depending on our size we may have the same timing or in some cases, the vector will be faster. If we have an array that is 10, then the vector and Linked List will both be timed at 0. This is because 10 is a small size, so they both are very quick, but as we increase our size then we can see that there are actual numbers. I had set my array size to 100,000 and when it came to filling them up with random ints, the Linked List was faster but when it came to filling them up with random strings (ascii values) and using move semantics, the vector was much faster. I did run it a few times because for some reason some of the numbers were very big, like 127 for the Linked List which was weird but when running it again it would go back to a much smaller number.

For question 2, the goal of this problem is to time which binary search (recursive, iterative, and linked list) is faster with a million and 10 million as a size in the array. I had used Professor Fried’s code when using recursive and iterative binary search and geeksforgeeks as a reference when it came to the linked list binary search. I had an idea of where to start for binary search and when coming across geeksforgeeks code, it started to make sense on how to structure it using Linked List. When creating a Linked List binary search, I had to create a node class which I named ListNode. Within the ListNode class, I have our public values which are the value (or data) and next. I created functions that will create a new node, find the middle element (which is crucial for binary search) and the actual binary search function.

The middleElement function simply looks for the middle element in our list. In our binary search function, we check to see if the middle value is the value we are looking for. If so, we return our middle element but if not then we continue on to next only if our mid value is less than the target value we are looking for. If it isn’t then we can say our last element is equal to our middle element. The while loop continues this until we find our value. For the recursive and iterative binary search, we have a similar ideology. Here we didn’t have to build another function to find the middle element and instead can find it by adding the first and last element together and dividing by two. In the recursive binary search we call the function again in the function and it repeats the process until we have our middle value while in our iterative binary search it's a much shorter block of code and is very straightforward. When running this in our main I wasn’t able to find a way to insert a million or ten million nodes in a linked list. I was able to fill the recursive and iterative functions with random numbers, but for the linked list one I had to hard code some of them. Unlike the first example that was given, when timing these three I got 0 for all of them even with a very large size. I got 0 for a million elements in an array and the same goes for 10 million. If we go any higher we get an overload, and that is something we don’t want. If I had to guess I would say iterative binary search is a lot more effective and would be a lot faster and the code is a lot more simplistic.

Our goal in question 3 is to sort 2d matrices using Bubble Sort, Insertion Sort and Selection Sort. The implementations in the code are very similar to each other since we have to sort a nested array for all of them. In Bubble Sort, Insertion Sort and Selection Sort I used three for loops to go through the rows and columns. The first for loop is made specifically for iterating the rows. The second for loop iterates 1 index at a time for the specific row we are currently in. The third for loop iterates another index at a time at the specific row we're currently in and does some piece of code to swap or sort the columns first.

The way Insertion Sort works in a matrix is to iterate the rows and in that specific row we iterate the index of each column, compare the current element to the previous element and if the current element is smaller than the previous element, we compare the current element to the elements before. Once we know where we wish to insert the element, we want to create space by shifting the other elements one position to the right and inserting the element at the right position. At the end of this, the columns would be sorted and we would need to sort the rows. The way Selection Sort works is we want to search for the minimum element in the list, swap the value at the location of the minimum variable (which is at the first position), increment the minimum variable to point to the next element. We want to establish a sorted subarray and unsorted subarray throughout the iterations and doing Selection Sort as we move on in the iterations, the first part will be sorted while the second part will be unsorted until we reach the end of the iterations. At the end of this, the columns would be sorted and we would need to sort the rows. The way Bubble Sort works is that it looks at one index, and the adjacent index, sees which of the two is the minimum and swaps them. At the end of this, the columns would be sorted and we would need to sort the rows. To sort the rows for all 3 sorting algorithms, it follows the same structure where the outer loop is the columns and the inner for loops is the row.

In the main, I plug in the values for my matrix. I find my rows by finding the size of the matrix/first row, the first index of the matrix's rows. I do the same for my columns where it's the matrix row/matrix row and matrix column. I then call the functions built and print them in a for loop.

For question 8, our goal is to create a templated class that will ask the user for a size and a target number. We fill the size of a vector with random integers and find all the combinations to add to the target number. I created four functions in this templated class. The first function which is ask() simply asks the user for a size value and a target sum. I create a vector and in a for loop i push back random numbers until it reaches the size given by the user. I call the function sortArray which sorts the elements in the randomNums vector so it's easier to go through it and add the numbers together for our targeted sum. I call the function possibleUniqueComb which will find all the different types of combinations iteratively. In this function we check to see if our temp sum (currently 0) is the same as our target sum. If so, it will call a new function uniqueCombinations. The uniqueCombinations function simply displays a unique combination. We come back to the possibleUniqueComb if the temp sum isn't the target sum. It goes through a for loop to find other combinations. We check to see if the targeted sum is less than the temp sum created or if there are any repeating numbers to ensure we're not pushing that combination in a new vector if it doesn't satisfy the criteria of unique combinations or the sum of those numbers equals the target sum. After that, if we did satisfy the criteria, we push back the values that are found fit for the combination in the new array. We will keep determining if the values fit the criteria by calling this function recursively. If some values don't work, we popback those values and try again. We also popback the values to have a clean start with the vector and combinations again.

In the main you are prompted to give a size and targeted sum and you'll get the answer of all combinations that were found

For question 10, our goal is to sort the linked list from odd numbers to even. I am assuming that the linked list is already sorted and I don't need to sort it again. To achieve this, I created a ListNode class to create the nodes with values. I create a newNode function which will create new nodes for us and input data. I create an oddEvenList function which will take in a number (our data). I have the base case where if our head node is a null pointer we need to return null pointer. I have my three variables to keep track of the odds and evens. In the for loop I check to see if the evenNode and the next value are a null pointer. If they aren't then we switch some values around. The odd value starts off with the head. I equal that value to the value next to our even value and rename that value as odd. I do the same thing with the even value. I iterate through the loop until even or next to even is null pointer. In this case, given what I worte in the main, we iterate through the loop twice. In this iteration, the odd values contains 1, 3, and 5 while our even and temp are 2 and 4. After the while loop is done, I set the next value next to odd be temp, which is the null pointer. I return the head which contains our sorted list. The last function made prints out the list.

In the main I hard code the values and we can see how the list looks before and after the sort.

For question 11, our goal is to have a restaurant class with a virtual function name menu and we can override that function so that way a new class can write their own menu. The reason to have a pure virtual function is so that when we override it and rewrite that function in our derived class anyway we want. I have three protected variables, the name, address and tables (this was never used but it can determine how big the restaurant is). I have three dervived classes, Italian, Greek and Chinese. In each class, the parameters taken are name and address. I use the operator equal function to insert the parameters taken in so that way the restaurant is customized. I created the menu as a vector since it's dynamic and it's easier to list everything out. Each menu function contains popular items from the culture of each restaurant. I created a templated reader robot that reads the menu and prints out each menu item.

In the main, you can put any restaurant you want (name and address) and it will print out the menu using the reader robot template class and then print out the menu. As seen I put specific nams and their addresses, and it prints out a menu made for that restaurant.