Yida Zou
Peer Reviewer: Joel Herzog

Assignment 2 Pseudocode

1)

```
1. Declare a pointer with enough bytes for 300 integer elements.
2. Declare an int vector.
3. Using a doubly linked list class with an int variable and pointers to next and prev
4. int j = 0          //for keeping track of position in pointer
5. For(int i=150; i<=448; i+=2)     //Use a loop to get even numbers 150-448.
    a. *(numbersPointer + j) = i;      //For pointer: assign each number into the memory
       pointed to by the pointer while incrementing the memory address
    b. pushback i to vector
    c. insertTail(i)      //insert new node at the tail of the linked list and assign it the
       value of i
    d. j++      //increment position for pointer
6. For(int i=449; i>=151; i-=2)     //Use another loop to append odd numbers 449-151 using
   similar method to above.
    a. j++      //Keep using j to keep track of position of pointer (not reset to 0)
    b. All other code should be almost the same
```

2a) <u>Insertion Sort (ascending)</u>

```
1. Input: an unsorted pointer.
2. Select each element in the pointer, starting from the 2nd element.
3. Compare and swap the selected element with each and every element in the sorted
   portion of the list starting from the rightmost(highest) element in the sorted portion
    a. At first, this portion will only be the 1st element
4. Stop comparing and swapping when the selected element is less than the element it's
   compared to.
5. Repeat previous steps until you reach the end of the pointer.
```

2b) <u>Exchange Sort (ascending)</u>

```
1. Input: an unsorted pointer.
2. Starting with the first position, assign this element to a variable X.
3. Compare the value of X, with each and every element in the list one by one.
    a. if (X > element)
4. If X's value is bigger, swap the elements.          //use a temp integer variable
5. Repeat previous steps with the unsorted elements
```

3a) <u>Selection Sort (descending)</u>

```
1. Input: an unsorted vector.
2. Starting with the first index, assign this element to a variable X.
3. Compare the value of X, with each element in the list one by one.
    a. if (X < element)
4. If X's value is smaller, assign the new element as X.
5. move element X to the front of the vector
6. Repeat previous steps with the unsorted elements
```

3b) <u>Bubble Sort (descending)</u>

1. Input: an unsorted vector.
2. Compare adjacent pairs of elements and swap them so that the bigger element comes first, starting from the first element pair.
    a. if (element1 < element2)
        i. int temp = element1;
        ii. element1 = element2;
        iii. element2 = temp;
3. Repeat step 2 for the next pair in the vector.
4. Once one run of comparisons is over, do another run of comparisons.
5. Repeat until no more comparisons are needed
    a. Use a bool variable to detect if a swap is made for each run.
        i. (reset to false at the beginning of each run)
        ii. (switch to true if a swap is made)

4a) <u>Insertion Sort (ascending)</u>

1. Input: an unsorted doubly linked list.
2. Set the first(head) node as a sorted element
3. Select the next unsorted element and compare it to the sorted elements.
    a. if (unsortedElement < sortedElement)
4. Starting from the first sorted element, check to see if the selected element is less than the sorted element.
    a. If it is, then insert it before the node it was compared to.
    b. If not, then go to the next sorted element and compare them and repeat this step.
5. Repeat steps 3 and 4.

4b) <u>Exchange Sort (ascending)</u>

1. Input: an unsorted doubly linked list.
2. Starting with the first node, assign this element to a variable **X**.
3. Compare the value of **X**, with each element in the list one by one.
    a. if (**X** > element)
4. If **X**'s value is bigger, swap the elements**.**
5. Repeat previous steps with the unsorted elements

6)

The average timings of the insertion sorts were faster than the exchange sorts for both algorithms. This seems to prove that insertion sort is faster than exchange sort for a data structure of 300 elements. The sorting algorithms for the pointer were faster than their respective sorts for the linked list. This is because it takes less time to iterate through a pointer than a doubly linked list.

7) <u>Linear Search</u>

1. Input: (an integer to find, and a sorted data structure)
2. Output: address of the integer (and index if the data structure is a vector or pointer)

3. Iterate through the elements starting from the first element, checking if the element searched for is equal to an element in the list
    a. For pointer: Iterate by incrementing the memory address; print the address
    b. For vector: Iterate by incrementing the index starting from zero; print the address and index
    c. For doubly linked list: Iterate using the next pointer; print the address of the node.

8) <u>Binary Search</u>

1. Input: (an integer to find, and a sorted data structure)
2. Output: address of the integer (and index if the data structure is a vector or pointer)
3. Find the middle element
4. middleIndex = leftIndex + (rightIndex - leftIndex)/2
    a. For pointer: The address of the middle element would be (pointerAddress + (numberOfElements/2 + middleIndex)
    b. For vector: The middle index would be (middleIndex)
5. For doubly linked list: Use 2 pointers to iterate through the nodes with one iterating through twice as fast.
    a. Once the faster one finishes iterating, the slower one would have found the middle
6. Compare the value of the searched-for element with the value of the middle element
    a. If equal, print the middle element's address (and index for vector)
    b. If less than or greater than, then binary search(repeat steps 3-6) the elements before or after the middle value respectively. Update rightIndex and leftIndex respectively as well.
        i. Assuming an Ascending list
           (If descending, swap less than and greater than)
            1. If less than: rightIndex = middleIndex-1
            2. If greater than: leftIndex = middleIndex+1