When working on this quiz, recall the rules stated on the Academic Integrity statement that you signed. You can download/unzip the **q6helper** project folder (available for Friday, on the **Weekly Schedule** link) in which to write/test/debug your code. Submit your completed **q6solution.hpp** file online by Thursday, at 11:30pm. I will post my solutions to EEE reachable via the **Solutions** link on Friday.

---

1. (6 pts) In the download, fill in the **selection_sort** method to sort a linked list. Below is the array code for this method, whose functionality you should translate for linked lists. Use no arrays in your solution. Hint: instead of **int** cursors use **LN<T>*** cursors; note you change **values** in the linked list but do not change the order of its **nodes**. There are 5 tests of calling **selection_sort** in the driver on random arrays.

```
template<class T>
void selection_sort(LN<T>* l) {
  for (LN<T>* c = l; c!=nullptr; c=c->next) {
    LN<T>* smallest = c;
    for (LN<T>* s=smallest->next; s!=nullptr; s=s->next)
      if (s->value < smallest->value)
        smallest = s;
    std::swap(c->value,smallest->value);
  }
}
```

2. (6 pts) In the download, fill in the **merge** method (it has many parameters) to merge two **contiguous** parts of an array (i.e., **left_high + 1 = right_low**) back into the array. See the pre- and post-conditions in the comments in the download for more context; they are not repeated here. There is a single test of a call to **merge** (use this mostly for debugging purposes) and then 5 tests of calling **merge_sort** (which calls **merge** many times) in the driver on random arrays. Below is pseudo-code for this method

```
void merge(int a[], int left_low,  int left_high, int right_low, int right_high) {
  int length = right_high-left_low+1;
  int temp[length];
  int left  = left_low;
  int right = right_low;

  for (int i = 0; i< length; ++i) {
    if (left > left_high)
      temp[i] = a[right++];
    else if (right > right_high)
      temp[i] = a[left++];
    else if (a[left] <= a[right])
      temp[i] = a[left++];
    else
      temp[i] = a[right++];
  }

  for (int i=0; i< length; ++i)
    a[left_low++] = temp[i];
}
```

3. (8 pts) In the download, fill in the **radix_sort** method to sort arrays of 6 digit non-negative numbers. There are 5 tests of calling **radix_sort** in the driver on random arrays. Below is pseudo-code for this method

```
void radix_sort(int a[], int length) {
  ics::ArrayQueue<int> buckets[10];
  for (int place = 1; place<=100000; place=place*10) {
    for (int i=0; i<length; i++)
      buckets[select_digit(a[i],place)].enqueue(a[i]);
    int i=0;
    for (int b=0; b<10; b++)
      while (!buckets[b].empty())
        a[i++] = buckets[b].dequeue();
  }
}
```

Note that when **place** is **1**, calling **select_digit (9416, place)** returns **6** (which is the digit in the ones place); when **place** is **10**, calling **select_digit (9416, place)** returns **1** (which is the digit in the tens place); when **place** is **100**, calling **select_digit (9416, place)** returns **4** (which is the digit in the hundreds place); when **place** is **1,000**, calling **select_digit (9416, place)** returns **9** (which is the digit in the thousands place); when **place** is **10,000** or bigger, calling **select_digit (9416, place)** returns **0** (think of each number "padded" on the left with zeros). Use the **ArrayQueue** implementation –all its operations are **O(1)**.

4. (5 pts) In the download, fill in the **test_partition** function to test how well two different methods that choose a pivot index for the partition algorithm (used in quicksort) work empirically. Fill in the function so that it tests (a) any specified function (**choose_pivot_index**) to choose the index of the pivot, (b) tests this function a specified (**num_tests**) number of times, (c) on a specified (**length**) length array, randomly filled with integer values. The result it returns is the average length of the longest subarray (divided by the length of the original array) that must be recursively sorted after the partitioning. If the pivot index is optimally chosen, the longest subarray would always be half the size of the array, so your function should always return values >= .5

Your function should create an array of size **length** and fill it with the integers **0** to **length-1**. For each test, randomize the values in the array (use the **shuffle** function in **q6utility.hpp**). Compute the pivot index for this array using the supplied function argument (**choose_pivot_index**). Then partition (see the **partition** function in **q6utility.hpp**) the array using the pivot at that index; **partition** returns the index where that pivot value appears in the partitioned array. Next compute the longest subarray to sort recursively; ultimately the **test_partition** function returns the average size of these longest subarrays (as a percent of the array size).

For example, if the array size is 8, it might store the following values in the following order

0	1	2	3	4	5	6	7
7	2	1	0	6	4	3	5

If we choose the rightmost index (**7**) as the index of the pivot, the pivot will be **5**. When the **partition** function returns, the array will appear as follows, with **partition** returning the index where **5** moves to.

0	1	2	3	4	5	6	7
3	2	1	0	4	5	7	6

Note the values to the left of index 5 are all smaller than the value 5 (and unsorted); the values to the right are all bigger (and unsorted).

For this example, the size of the left subarray is **5** and the right subarray is **2**. Since **5** is bigger, that value is added into some variable accumulating the sums of all the bigger sizes. Eventually the sum is divided

by the number of tests (`num_tests`), and before reporting the value we further divide it by the length (`length`) of the array, to return the average as a percentage of the length of the entire array.

```
double test_partition(int length, int num_tests,
                      int (*choose_pivot_index) (int a[], int low, int high)) {
  int test_array[length];
  for (int i=0; i <length; ++i)
    test_array[i] = i;

  int sum_sizes = 0;

  for (int i=0; i<num_tests; ++i) {
    shuffle(test_array,length);
    int pivot_index = choose_pivot_index(test_array,0,length-1);
    int pivot = partition(test_array, 0, length-1, pivot_index);
    int best = std::max(pivot,length-pivot-1);
    sum_sizes += best;
  }
  return float(sum_sizes)/num_tests/length;
}
```