Rohan Behera

10/27/20

UCR - CS 010C

Assignment 2

Deliverables:

Create a single PDF file that contains your answers to the questions. Then create a zip file that contains this PDF file along with all your code source files. Submit this zip file in iLearn.

Deadline: 10/27/2020 11:59 pm.

Exercise 1

If the elements of a list are sorted, is an array-based or a linked-list-based implementation of the list more efficient for binary search? Explain. Put your answer in the PDF file.

An array-based implementation of a sorted list is more efficient than a linked-list implementation because with an array, you can access any element in constant time while in a linked list you have to traverse the list sequentially. The time complexity for a binary search of a linked list goes from O(nlogn) and for an array the notation would be O(1).

Exercise 2

Write a C++ class that implement two stacks using a single C++ array. That is, it should have functions pop_first(), pop_second(), push_first(…), push_second(…), size_first(), size_second(), …. When out of space, double the size of the array (similarly to what vector is doing).

Notes:

- Complete all the functions *in exercise_2.cpp*, then submit this cpp file.
- pop_first() and pop_second() should throw *std::out_of_range* exception when stack is empty.

```
#include <cstdio>

#include <stdexcept>

#include <iostream>

#include <exception>

using namespace std;
```

```
template <class T>
class TwoStacks {
 public:
   int top1=-1, top2, s2_ctr;
 // Constructor, initialize the array to size_hint
 TwoStacks(int size_hint = 16) : size_(size_hint), array_(new T(size_hint)) {
    //array_ = new T[size_hint];
    top1 = -1;
    top2 = size_;
    s2_ctr = 0;
 }
 // Destructor
 ~TwoStacks() {
    delete[] array_;
 }
 // Push a value to the first stack
 void push_first(const T& val) {
    if (top1 < top2 -1) {
       top1++;
       array_[top1] = val;
    } else {
          cout << "Stack 1 is full" << endl;
          reinitializeArray();
    }
```

```
}
// Push a value to the second stack
void push_second(const T& val) {
    if (top1 < top2 -1) {
       top2--;
       s2_ctr++;
      array_[top2] = val;
      cout << "push2 val: " << array_[top2] << endl;
    } else {
            cout << "Stack 2 is full" << endl;
            reinitializeArray();
    }
}
// Pop from the first stack and return the value
T pop_first() {
     if(top1 >= 0) {
        int x = array_[top1];
        top1--;
        return x;
     } else {
          throw std::out_of_range("Out of range");
     }
}
// Pop from the second stack and return the value
T pop_second() {
```

```
    if (top2 < size_){
        int x = array_[top2];
        top2++;
        s2_ctr--;
        return x;
    } else {
         throw std::out_of_range("Out of range");
    }
}
int getMaxArraySize() {
   return (int)sizeof(array_);
}
// Return the size of the first stack
size_t size_first() const {
          size_t s = top1+1;
   if (s > 0) return s;
   else return 0;
}

// Return the size of the second stack
size_t size_second() const {
/*     size_t s = top2+1-size_;
   if (s > 0)return s;
   else return 0;*/
   return s2_ctr;
```

```
  }


  // Return true if the first stack is empty
  bool empty_first() const {
     return size_first() == 0;
  }
  // Return true if the second stack is empty
  bool empty_second() const {
     return size_second() == 0;
  }
  /**
   * Divide the array into two half
   * array[0...(hint/2)-1)] for stack1
   * array[[hint/2)... hint-1] for stack2
   * when it overflows in either stack,
   * copy the elements into two arrays
   * re-initialize the original array to size 2*hint
   * copy stack1 elements from first half of old array to 1st half of new array
   * copy stack2 elements from last half of old array to 2nd half of new array
   */
 void reinitializeArray() {
    size_t old_size_ = size_;
    size_ = size_ * 2;
    T* array2 = new T[size_];
    size_t st1_size = size_first();
```

```
    for (size_t i = 0; i < st1_size; i++) {
        array2[st1_size-i] = array_[i];
    }
    size_t st2_size = size_second();
    for (size_t j = 0; j < st2_size; j++) {
        array2[size_-j] = array_[old_size_-j];
    }
    *array_ = *array2;
            cout << "Overflow: Stack ReinitializeArray() called: New Array Size : " << size_ << endl;
}
 private:
  int size_;
  T *array_;
};
int main(int argc, char** argv) {
            try {
                        TwoStacks<int> ts(20);
                        ts.push_first(31);
                        ts.push_second(90);
                        ts.push_first(38);
                        ts.push_first(23);
                        cout << "Size of stack1 is " << ts.size_first() << endl;
                        cout << "Size of stack 2 is " << ts.size_second() << endl << endl;
                        ts.push_second(46);
                        cout << "Size of stack1 is " << ts.size_first() << endl;
```

```
cout << "Size of stack 2 is " << ts.size_second() << endl << endl;
ts.push_first(45);
cout << "Size of stack1 is " << ts.size_first() << endl;
cout << "Size of stack 2 is " << ts.size_second() << endl << endl;;
ts.push_first(64);
cout << "Size of stack1 is " << ts.size_first() << endl;
cout << "Size of stack 2 is " << ts.size_second() << endl << endl;
ts.push_first(74);
cout << "Size of stack1 is " << ts.size_first() << endl;
cout << "Size of stack 2 is " << ts.size_second() << endl << endl;
ts.push_first(36);
cout << "Size of stack1 is " << ts.size_first() << endl;
cout << "Size of stack 2 is " << ts.size_second() << endl << endl;
ts.push_second(53);
cout << "Size of stack1 is " << ts.size_first() << endl;
cout << "Size of stack 2 is " << ts.size_second() << endl << endl;
ts.push_second(12);
cout << "Size of stack1 is " << ts.size_first() << endl;
cout << "Size of stack 2 is " << ts.size_second() << endl << endl;
ts.push_first(21);
cout << "Size of stack1 is " << ts.size_first() << endl;
cout << "Size of stack 2 is " << ts.size_second() << endl << endl;
ts.push_second(47);
cout << "Size of stack1 is " << ts.size_first() << endl;
cout << "Size of stack 2 is " << ts.size_second() << endl << endl;
```

```
            ts.push_second(65);
            cout << "Size of stack1 is " << ts.size_first() << endl;
            cout << "Size of stack 2 is " << ts.size_second() << endl << endl;
            ts.push_second(21);
            cout << "Size of stack1 is " << ts.size_first() << endl;
            cout << "Size of stack 2 is " << ts.size_second() << endl << endl;
             cout << "Popped element from stack1 is " << ts.pop_first() << endl;
            cout << "Popped element from stack2 is " << ts.pop_second() << endl;
            cout << "Size of stack1 is " << ts.size_first() << endl;
            cout << "Size of stack 2 is " << ts.size_second() << endl;
        } catch (exception& e) {
            cout << "Exception : " << e.what() << endl;
        }
  return 0;
}
```

Exercise 3

a. Implement functions for insertion sort, quicksort, heapsort and merge sort that input an array of integers and sort it in-place.

```
void insertion_sort(int array[], size_t size) {
 // Implement here
   int n, j;
   for (int i = 1; i < size; i++) {
      n = array[i];
      j = i-1;
      while (j >= 0 && array[j] > n) {
         array[j+1] = array[j];
         j = j-1;
      }
      array[j+1] = n;
   }
```

```
}

//Finds pivot element and divides array into two parts so
//elements on the left are smaller than the ones on the right
int partition(int array[], int low, int high) {
   int pivot = array[high];
   int partIndex = low;
   for (int i = low; i < high; i++) {
      if (array[i] <= pivot) {
         swap(array[i], array[partIndex]);
         partIndex++;
      }
   }
   swap(array[partIndex], array[high]);
   return partIndex;
}

void quicksortCopy(int array[], int low, int high) {
 // Implement here
   if (low < high) {
      int partIndex = partition(array, low, high);
      quicksortCopy(array, low, partIndex-1);
      quicksortCopy(array, partIndex+1, high);
   }
}

void quick_sort(int array[], size_t size) {
   int low = 0;
   int high = size-1;
   quicksortCopy(array, low, high);
}

//build heap with given elements
//Create max heap to sort elements in ascending order
//Swap root node with last node and delete last node from heap
void heapify(int arr[], int n, int i) {
   int largest = i;
   int left = 2*i+1;
   int right = 2*i+2;

   if (left < n && arr[left] > arr[largest]) {
      largest = left;
```

```
    }

    if (right < n && arr[right] > arr[largest]) {
        largest = right;
    }

    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

void heap_sort(int array[], size_t size) {
  // Implement here
    for (int i = size/2-1; i >= 0; i--) {
        heapify(array, size, i);
    }

    for (int i = size-1; i >= 0; i--) {
        swap(array[0], array[i]);
        heapify(array, i, 0);
    }
}

// divides the input array into two halves, calls itself for the two halves,
// and then merges the two sorted halves.
void merge(int array[], int low, int middle, int high) {
    int num1 = middle-low+1;
    int num2 = high-middle;

    int leftArray[num1], rightArray[num2];

    for (int i = 0; i < num1; i++) {
        leftArray[i] = array[low+i];
    }
    for (int j = 0; j < num2; j++) {
        rightArray[j] = array[middle+j+1];
    }

    int i = 0;
    int j = 0;
    int k = low;
```

```
   while (i < num1 && j < num2) {
      if (leftArray[i] <= rightArray[j]) {
         array[k] = leftArray[i];
         i++;
      }
      else {
         array[k] = rightArray[j];
         j++;
      }
      k++;
   }

   while (i < num1) {
      array[k] = leftArray[i];
      i++;
      k++;
   }
   while (j < num2) {
      array[k] = rightArray[j];
      j++;
      k++;
   }
}
void mergesortCopy(int array[], int left, int right) {
 // Implement here
   if (left < right) {
      int middle;
      middle = left + (right-left)/2;
      mergesortCopy(array, left, middle);
      mergesortCopy(array, middle+1, right);
      merge(array, left, middle, right);
   }
}

void merge_sort(int array[], size_t size) {
   int low = 0;
   int high = size-1;
   mergesortCopy(array, low, high);
}
```

b. Write a program that generates random integer arrays (hint: use seed appropriately to avoid generating same sequences) of lengths 10, 100, 1000, 10,000, 100,000, 1000,000, and then sorts each using each of the sorting functions from (a), and measures the time in nanoseconds. The program will repeat this process 30 times and will compute the average execution time for each (arraysize,sorting-function) pair, over these 30 iterations. Finally, the program will output all these numbers in a readable format, e.g., as a table, in the PDF file.

```
// Assignment 2, exercise 3
// To compile, run:
//    g++ -std=c++11 exercise_3.cpp
#include <chrono>
#include <cstdio>
#include <algorithm>
#include <iostream>
using namespace std;
/*
 * Implement the 4 in-place sort functions for exercise 3.a.
 */

/* a sorting algorithm that treats the input as two parts,
 * a sorted part and an unsorted part, and repeatedly inserts
 *  the next value from the unsorted part into the correct location in the sorted part.
 */

//Setting global variables for each sort for each array
double firstInsertion, firstQuicksort, firstHeap, firstMerge;
double secondInsertion, secondQuicksort, secondHeap, secondMerge;
double thirdInsertion, thirdQuicksort, thirdHeap, thirdMerge;
double fourthInsertion, fourthQuicksort, fourthHeap, fourthMerge;
double fifthInsertion, fifthQuicksort, fifthHeap, fifthMerge;
double sixthInsertion, sixthQuicksort, sixthHeap, sixthMerge;

void insertion_sort(int array[], size_t size) {
 // Implement here
   int n, j;
   for (int i = 1; i < size; i++) {
      n = array[i];
      j = i-1;
      while (j >= 0 && array[j] > n) {
         array[j+1] = array[j];
         j = j-1;
      }
      array[j+1] = n;
```

```
   }
}

//Finds pivot element and divides array into two parts so
//elements on the left are smaller than the ones on the right
int partition(int array[], int low, int high) {
   int pivot = array[high];
   int partIndex = low;
   for (int i = low; i < high; i++) {
      if (array[i] <= pivot) {
         swap(array[i], array[partIndex]);
         partIndex++;
      }
   }
   swap(array[partIndex], array[high]);
   return partIndex;
}

void quicksortCopy(int array[], int low, int high) {
 // Implement here
   if (low < high) {
      int partIndex = partition(array, low, high);
      quicksortCopy(array, low, partIndex-1);
      quicksortCopy(array, partIndex+1, high);
   }
}

void quick_sort(int array[], size_t size) {
   int low = 0;
   int high = size-1;
   quicksortCopy(array, low, high);
}

//build heap with given elements
//Create max heap to sort elements in ascending order
//Swap root node with last node and delete last node from heap
void heapify(int arr[], int n, int i) {
   int largest = i;
   int left = 2*i+1;
   int right = 2*i+2;

   if (left < n && arr[left] > arr[largest]) {
```

```
        largest = left;
    }

    if (right < n && arr[right] > arr[largest]) {
        largest = right;
    }

    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

void heap_sort(int array[], size_t size) {
  // Implement here
    for (int i = size/2-1; i >= 0; i--) {
        heapify(array, size, i);
    }

    for (int i = size-1; i >= 0; i--) {
        swap(array[0], array[i]);
        heapify(array, i, 0);
    }
}

// divides the input array into two halves, calls itself for the two halves,
// and then merges the two sorted halves.
void merge(int array[], int low, int middle, int high) {
    int num1 = middle-low+1;
    int num2 = high-middle;

    int leftArray[num1], rightArray[num2];

    for (int i = 0; i < num1; i++) {
        leftArray[i] = array[low+i];
    }
    for (int j = 0; j < num2; j++) {
        rightArray[j] = array[middle+j+1];
    }

    int i = 0;
    int j = 0;
```

```
   int k = low;

   while (i < num1 && j < num2) {
      if (leftArray[i] <= rightArray[j]) {
         array[k] = leftArray[i];
         i++;
      }
      else {
         array[k] = rightArray[j];
         j++;
      }
      k++;
   }

   while (i < num1) {
      array[k] = leftArray[i];
      i++;
      k++;
   }
   while (j < num2) {
      array[k] = rightArray[j];
      j++;
      k++;
   }
}
void mergesortCopy(int array[], int left, int right) {
 // Implement here
   if (left < right) {
      int middle;
      middle = left + (right-left)/2;
      mergesortCopy(array, left, middle);
      mergesortCopy(array, middle+1, right);
      merge(array, left, middle, right);
   }
}

void merge_sort(int array[], size_t size) {
   int low = 0;
   int high = size-1;
   mergesortCopy(array, low, high);
}
```

```
/*
 * Generate random integers for exercise 3.b
 */
int *random_ints(size_t size) {
 int *numbers = new int[size];
 // Generate random numbers here
 srand(time(0));
 for (int i = 0; i < size; i++) {
     numbers[i] = (rand() % (100)+1); //setting range of random numbers from 0-100
 }
 return numbers;
}


int randarrayten() {
   for (int i = 0; i < 30; i++) {
      int *firstarray = random_ints(10);

       //insertion sort time output
      auto start_time = chrono::high_resolution_clock::now();
      insertion_sort(firstarray, 10);
      auto end_time = chrono::high_resolution_clock::now();
      auto duration = chrono::duration_cast<chrono::nanoseconds>(end_time - start_time);
        cout << "Length 10 Array Insertion sort elapsed time in nanoseconds: " << duration.count()
<< " ns" << endl;
        firstInsertion = duration.count();

      //quicksort time output
      start_time = chrono::high_resolution_clock::now();
      quick_sort(firstarray, 10);
      end_time = chrono::high_resolution_clock::now();
      duration = chrono::duration_cast<chrono::nanoseconds>(end_time - start_time);
      cout << "Length 10 Array Quicksort elapsed time in nanoseconds: " << duration.count() <<
" ns" << endl;
      firstQuicksort = duration.count();

      //heapsort time output
      start_time = chrono::high_resolution_clock::now();
      heap_sort(firstarray, 10);
      end_time = chrono::high_resolution_clock::now();
      duration = chrono::duration_cast<chrono::nanoseconds>(end_time - start_time);
```

```
    cout << "Length 10 Array Heap sort elapsed time in nanoseconds: " << duration.count() <<
" ns" << endl;
    firstHeap = duration.count();

    //merge sort time output
    start_time = chrono::high_resolution_clock::now();
    merge_sort(firstarray, 10);
    end_time = chrono::high_resolution_clock::now();
    duration = chrono::duration_cast<chrono::nanoseconds>(end_time - start_time);
    cout << "Length 10 Array Merge sort elapsed time in nanoseconds: " << duration.count() <<
" ns" << endl;
    firstMerge = duration.count();
  }
    cout << endl;
}

int randarrayhundred() {
    for (int i = 0; i < 30; i++) {
      int *secondarray = random_ints(100);
       //insertion sort time output
      auto start_time = chrono::high_resolution_clock::now();
      insertion_sort(secondarray, 100);
      auto end_time = chrono::high_resolution_clock::now();
      auto duration = chrono::duration_cast<chrono::nanoseconds>(end_time - start_time);
       cout << "Length 100 array Insertion sort elapsed time in nanoseconds: " <<
duration.count() << " ns" << endl;
       secondInsertion = duration.count();

      //quicksort time output
      start_time = chrono::high_resolution_clock::now();
      quick_sort(secondarray, 100);
      end_time = chrono::high_resolution_clock::now();
      duration = chrono::duration_cast<chrono::nanoseconds>(end_time - start_time);
      cout << "Length 100 Array Quicksort elapsed time in nanoseconds: " << duration.count()
<< " ns" << endl;
      secondQuicksort = duration.count();

      //heapsort time output
      start_time = chrono::high_resolution_clock::now();
      heap_sort(secondarray, 100);
      end_time = chrono::high_resolution_clock::now();
      duration = chrono::duration_cast<chrono::nanoseconds>(end_time - start_time);
```

```
        cout << "Length 100 Array Heap sort elapsed time in nanoseconds: " << duration.count()
<< " ns" << endl;
        secondHeap = duration.count();

        //merge sort time output
        start_time = chrono::high_resolution_clock::now();
        merge_sort(secondarray, 100);
        end_time = chrono::high_resolution_clock::now();
        duration = chrono::duration_cast<chrono::nanoseconds>(end_time - start_time);
        cout << "Length 100 Array Merge sort elapsed time in nanoseconds: " << duration.count()
<< " ns" << endl;
        secondMerge = duration.count();
      }
        cout << endl;
}

int randarraythousand() {
      for (int i = 0; i < 30; i++) {
        int *thirdarray = random_ints(1000);

        //insertion sort time output
        auto start_time = chrono::high_resolution_clock::now();
        insertion_sort(thirdarray, 1000);
        auto end_time = chrono::high_resolution_clock::now();
        auto duration = chrono::duration_cast<chrono::nanoseconds>(end_time - start_time);
         cout << "Length 1000 Array Insertion sort elapsed time in nanoseconds: " <<
duration.count() << " ns" << endl;
         thirdInsertion = duration.count();

        //quicksort time output
        start_time = chrono::high_resolution_clock::now();
        quick_sort(thirdarray, 1000);
        end_time = chrono::high_resolution_clock::now();
        duration = chrono::duration_cast<chrono::nanoseconds>(end_time - start_time);
        cout << "Length 1000 Array Quicksort elapsed time in nanoseconds: " << duration.count()
<< " ns" << endl;
        thirdQuicksort = duration.count();

        //heapsort time output
        start_time = chrono::high_resolution_clock::now();
        heap_sort(thirdarray, 1000);
        end_time = chrono::high_resolution_clock::now();
```

```
        duration = chrono::duration_cast<chrono::nanoseconds>(end_time - start_time);
        cout << "Length 1000 Array Heap sort elapsed time in nanoseconds: " << duration.count()
<< " ns" << endl;
        thirdHeap = duration.count();

        //merge sort time output
        start_time = chrono::high_resolution_clock::now();
        merge_sort(thirdarray, 1000);
        end_time = chrono::high_resolution_clock::now();
        duration = chrono::duration_cast<chrono::nanoseconds>(end_time - start_time);
        cout << "Length 1000 Array Merge sort elapsed time in nanoseconds: " <<
duration.count() << " ns" << endl;
        thirdMerge = duration.count();
    }
        cout << endl;
}

int randarraytenthousand() {
    for (int i = 0; i < 30; i++) {
        int *fourtharray = random_ints(10000);

        //insertion sort time output
        auto start_time = chrono::high_resolution_clock::now();
        insertion_sort(fourtharray, 10000);
        auto end_time = chrono::high_resolution_clock::now();
        auto duration = chrono::duration_cast<chrono::nanoseconds>(end_time - start_time);
         cout << "Length 10000 Array Insertion sort elapsed time in nanoseconds: " <<
duration.count() << " ns" << endl;
         fourthInsertion = duration.count();

        //quicksort time output
        start_time = chrono::high_resolution_clock::now();
        quick_sort(fourtharray, 10000);
        end_time = chrono::high_resolution_clock::now();
        duration = chrono::duration_cast<chrono::nanoseconds>(end_time - start_time);
        cout << "Length 10000 Array Quicksort elapsed time in nanoseconds: " <<
duration.count() << " ns" << endl;
        fourthQuicksort = duration.count();

        //heapsort time output
        start_time = chrono::high_resolution_clock::now();
        heap_sort(fourtharray, 10000);
```

```
        end_time = chrono::high_resolution_clock::now();
        duration = chrono::duration_cast<chrono::nanoseconds>(end_time - start_time);
        cout << "Length 10000 Array Heap sort elapsed time in nanoseconds: " <<
duration.count() << " ns" << endl;
        fourthHeap = duration.count();

        //merge sort time output
        start_time = chrono::high_resolution_clock::now();
        merge_sort(fourtharray, 10000);
        end_time = chrono::high_resolution_clock::now();
        duration = chrono::duration_cast<chrono::nanoseconds>(end_time - start_time);
        cout << "Length 10000 Array Merge sort elapsed time in nanoseconds: " <<
duration.count() << " ns" << endl;
        fourthMerge = duration.count();

    }
        cout << endl;
}

int randarrayhundredthousand() {
    for (int i = 0; i < 30; i++) {
        int *fiftharray = random_ints(100000);
        //insertion sort time output
        auto start_time = chrono::high_resolution_clock::now();
        insertion_sort(fiftharray, 100000);
        auto end_time = chrono::high_resolution_clock::now();
        auto duration = chrono::duration_cast<chrono::nanoseconds>(end_time - start_time);
         cout << "Length 100000 Array Insertion sort elapsed time in nanoseconds: " <<
duration.count() << " ns" << endl;
         fifthInsertion = duration.count();

        //quicksort time output
        start_time = chrono::high_resolution_clock::now();
        quick_sort(fiftharray, 100000);
        end_time = chrono::high_resolution_clock::now();
        duration = chrono::duration_cast<chrono::nanoseconds>(end_time - start_time);
        cout << "Length 100000 Array Quicksort elapsed time in nanoseconds: " <<
duration.count() << " ns" << endl;
        fifthQuicksort = duration.count();

        //heapsort time output
        start_time = chrono::high_resolution_clock::now();
```

```
        heap_sort(fiftharray, 100000);
        end_time = chrono::high_resolution_clock::now();
        duration = chrono::duration_cast<chrono::nanoseconds>(end_time - start_time);
        cout << "Length 100000 Array Heap sort elapsed time in nanoseconds: " <<
duration.count() << " ns" << endl;
        fifthHeap = duration.count();

        //merge sort time output
        start_time = chrono::high_resolution_clock::now();
        merge_sort(fiftharray, 100000);
        end_time = chrono::high_resolution_clock::now();
        duration = chrono::duration_cast<chrono::nanoseconds>(end_time - start_time);
        cout << "Length 100000 Array Merge sort elapsed time in nanoseconds: " <<
duration.count() << " ns" << endl;
        fifthMerge = duration.count();
      }
        cout << endl;
}

int randarraymillion() {
   for (int i = 0; i < 30; i++) {
      int *sixtharray = random_ints(1000000);

       //insertion sort time output
      auto start_time = chrono::high_resolution_clock::now();
      insertion_sort(sixtharray, 1000000);
      auto end_time = chrono::high_resolution_clock::now();
      auto duration = chrono::duration_cast<chrono::nanoseconds>(end_time - start_time);
       cout << "Length of a million Array Insertion sort elapsed time in nanoseconds: " <<
duration.count() << " ns" << endl;
       sixthInsertion = duration.count();

      //quick sort time output
      start_time = chrono::high_resolution_clock::now();
      quick_sort(sixtharray, 1000000);
      end_time = chrono::high_resolution_clock::now();
      duration = chrono::duration_cast<chrono::nanoseconds>(end_time - start_time);
      cout << "Length of a million Array Quicksort elapsed time in nanoseconds: " <<
duration.count() << " ns" << endl;
      sixthQuicksort = duration.count();

      //heap sort time output
```

```
    start_time = chrono::high_resolution_clock::now();
    heap_sort(sixtharray, 1000000);
    end_time = chrono::high_resolution_clock::now();
    duration = chrono::duration_cast<chrono::nanoseconds>(end_time - start_time);
    cout << "Length of a million Array Heap sort elapsed time in nanoseconds: " <<
duration.count() << " ns" << endl;
    sixthHeap = duration.count();

    //merge sort time output
    start_time = chrono::high_resolution_clock::now();
    merge_sort(sixtharray, 1000000);
    end_time = chrono::high_resolution_clock::now();
    duration = chrono::duration_cast<chrono::nanoseconds>(end_time - start_time);
    cout << "Length of a million Array Merge sort elapsed time in nanoseconds: " <<
duration.count() << " ns" << endl;
    sixthMerge = duration.count();
  }
    cout << endl;
}

double averageSortingTime() {
  int i;
  double avgtimeInsertion[], avgtimeQuicksort[], avgtimeHeap[], avgtimeMerge[];
    avgtimeInsertion[i] = (firstInsertion + secondInsertion + thirdInsertion + fourthInsertion +
fifthInsertion + sixthInsertion)/6;
    avgtimeQuicksort[i] = (firstQuicksort + secondQuicksort + thirdQuicksort + fourthQuicksort
+ fifthQuicksort + sixthQuicksort)/6;
    avgtimeHeap[i] = (firstHeap + secondHeap + thirdHeap + fourthHeap + fifthHeap +
sixthHeap)/6;
    avgtimeMerge[i] = (firstMerge + secondMerge + thirdMerge + fourthMerge + fifthMerge +
sixthMerge)/6;
    cout << "Average insertion sort time: " << avgtimeInsertion[i] << endl;
    cout << "Average quicksort time: " << avgtimeQuicksort[i] << endl;
    cout << "Average heapsort time: " << avgtimeHeap[i] << endl;
    cout << "Average mergesort time: " << avgtimeMerge[i] << endl;
}

int main() {

    int generateArray1 = randarrayten();
    cout << generateArray1 << endl;
```

```
    int generateArray2 = randarrayhundred();
    cout << generateArray2 << endl;

    int generateArray3 = randarraythousand();
    cout << generateArray3 << endl;

    int generateArray4 = randarraytenthousand();
    cout << generateArray4 << endl;

    int generateArray5 = randarrayhundredthousand();
    cout << generateArray5 << endl;

//      int generateArray6 = randarraymillion();
//      cout << generateArray6 << endl;

    double copyavgsortTime = averageSortingTime();
    cout << copyavgsortTime << endl;

  return 0;
}
```

Average Insertion sort Time	Average Quicksort time	Average heapsort time	Average mergesort time
1.34021e+09	8.49185e+09	5.31802e+06	3.39576e+06

Average doesn't include values for 1 million array size.

c. Are your computed numbers reasonable given your knowledge of the asymptotic complexity of each sorting algorithm? Explain. Put your answer in the PDF file.

Yes the computed numbers are reasonable given my knowledge of the asymptotic complexity for each sorting algorithm. For array size 10, insertion sort has the fastest time but the rest of the arrays except for the last one, the merge sort algorithm was the fastest. These results make sense because the best case for insertion sort is O(n) and for merge, heap, and quicksort the best case is O(nlogn). I didn't calculate the times for each sort for the 1 million size array because it takes a lot longer time to sort through compared to the other arrays but I can predict that merge sort likely is sorts the array the quickest.

Note:

- Complete all the functions for a and b *in exercise_3.cpp*, then submit this cpp file.

Throughout the exercises, make any assumptions necessary.