CS2500 Project 2
Run Time Complexity of Merge Sort, Quicksort, and Heapsort

Evan Wilcox

Due February 12, 2019

1 Motivation

Knowing which sorting algorithm to choose when needing to sort something can be difficult. Because certain algorithms work more efficiently on certain types of data sets being able to analyse these algorithms to find the best one for a given situation can be very useful.

This report will implement three algorithms in C++: merge sort, heap, sort and quick sort. The algorithms will then be tested and analysed to find the inputs that give the worst, best, and average runtimes for each algorithm.

2 Background

Each algorithm being analysed in this report uses a technique called recursion. Recursion is a method of solving a problem where you break the problem into smaller instances of the same problem. You can then solve the main problem by combining the solutions of the smaller problems into the final solution.

For example, merge sort starts by splitting the original array in half then calling merge sort on each half. Merge sort continually splits the array in half until the array only has one element. It then merges the smaller arrays back in to larger arrays until it is a sorted array.

Similarly, quick sort splits the array into two subarrays. One that contains elements that are less than some pivot element p. And one that contains elements greater than p. It then puts the pivot element in between the two subarrays then calls quicksort on each subarray resulting in a sorted array.

Heap sort works a little differently. It starts by turning the array in to a max heap. A max heap is a binary tree structure that keeps the heap sorted by keeping the highest element at the root of the tree and the smaller elements as leafs. Heap sort then removes the max element and reheapifies the remaining array. What is left is a sorted array.

3 Procedures

1. Develop a precondition, postcondition, and loop invariants for merge sort, quick sort, and heapsort.
2. Show that the previous preconditions, postconditions, and loop invariants are correct.
3. Express merge sort, quick sort, and heap sort using pseudocode.
4. Implement merge sort, quick sort, and heap sort in python.
5. Implement preconditions, postconditions, and invariants using python assert statements to validate correctness.
6. Measure run times of the three algorithms to experimentally determine their run time complexity and compare to their expected run time complexity.
7. List problems encountered during development.
8. Develop and implement a testing plan.
9. Produce a conclusion addressing the efficacy of the methods used.

4 Pseudocode

Heapsort Pseudocode

Max-Heapify(A, i)
$l = 2i$
$r = 2i + 1$
if $l \leq A.heap\text{-}size$ and $A[l] > A[i]$
    $largest = l$
else $largest = i$
if $4 \leq A.heap\text{-}size$ and $A[r] > A[largest]$
    $largest = r$
if $largest \neq i$
    exchange $A[i]$ with $A[largest]$
    Max-Heapify$(A, largest)$

Build-Max-Heap(A)
$A.heap\text{-}size = A.length$
for $i = \lfloor A.length/2 \rfloor$ **downto** 1
    Max-Heapify(A, i)

Heap-Sort(A)
Build-Max-Heap(A)
for $i = A.length$ **downto** 2
    exchange $A[1]$ with $A[i]$
    $A.heap\text{-}size = A.heap\text{-}size - 1$
    Max-Heapify$(A, 1)$

Merge Sort Pseudocode

```
Merge(A, p, q, r)
n_1 = q - p + 1
n_2 = r - q
let L[1..n1 + 1] and R[1..n_2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i - 1]
for j = 1 to n2
    R[i] = A[q + j]
L[n_1 + 1] = ∞
R[n_2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k]DR[j]
        j = j + 1
```

```
Merge-Sort(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    Merge-Sort(A, p, r)
    Merge-Sort(A, p + 1, r)
    Merge(A, p, q, r)
```

Quicksort Pseudocode

```
Quick-Sort(A, p, r)
if p < r
    q =Partition(A, p, r)
    Quick-Sort(A, p, q - 1)
    Quick-Sort(A, q + 1, r)
```

```
Partition(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

5 Problems Encountered

Originally the three algorithms were planned to be written, tested, and timed using Python but because of Python's overhead the expected results could not be achieved. Specifically, the QuickSort algorithm when implemented in Python ran out of memory, hit the max recursion depth, or took an absurd amount of time even to sort small data sets. These problems were solved by implementing the three algorithms in C++ instead.

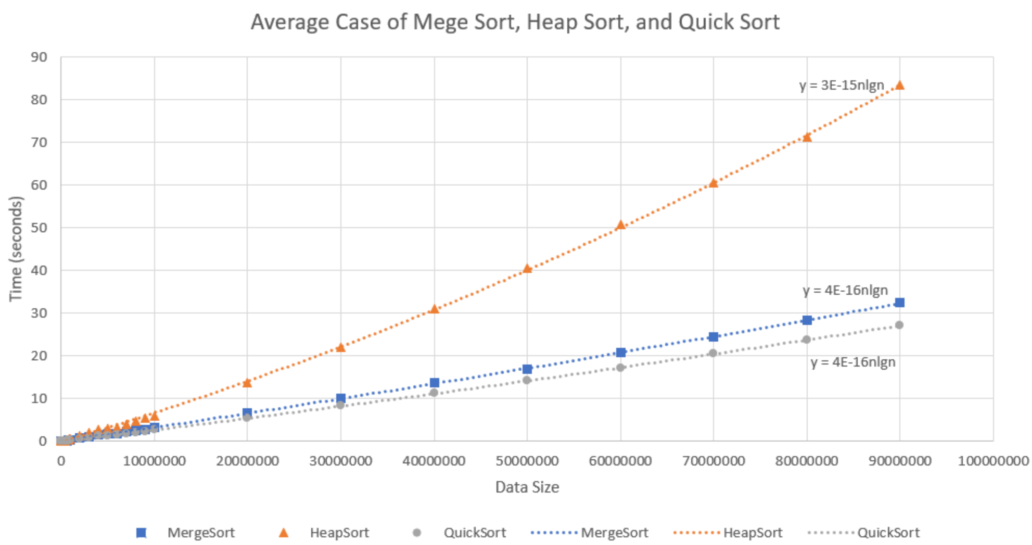
6 Testing Plan

All three algorithms were tested with the following testing plan:

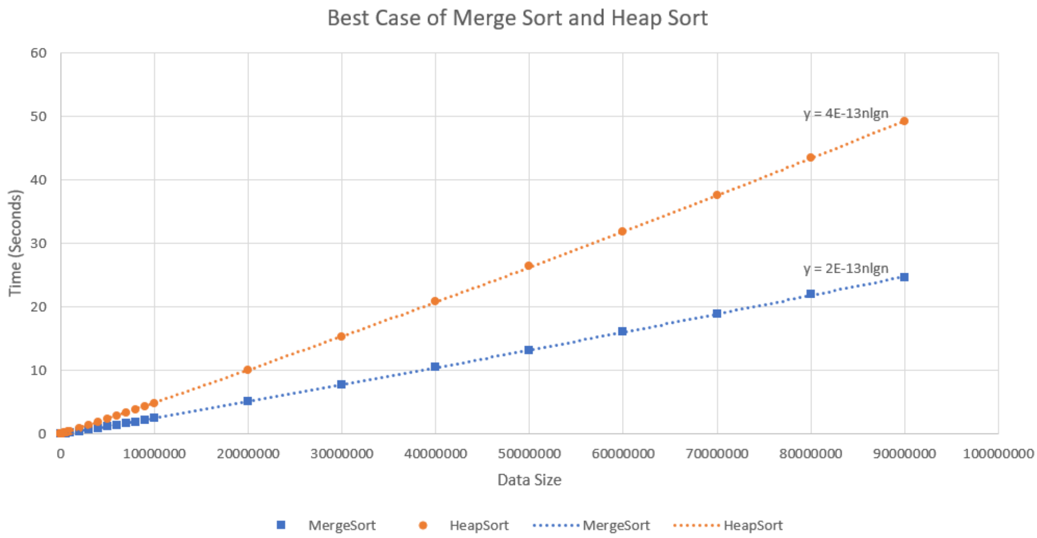
Input	**Expected Output**	**Output Received**
Empty Array	Empty Array	Empty Array
$n < 0$	Precondition Failed	Assert Error
Unsorted Array, n = length of array	Sorted Array	Sorted Array
Sorted Increasing Array, n = length of array	Sorted Array	Sorted Array
Sorted Decreasing Array, n = length of array	Sorted Array	Sorted Array
Unsorted Array, $n <$ length of array	Sorted Array up to n	Sorted Array up to n

7 Performance Results

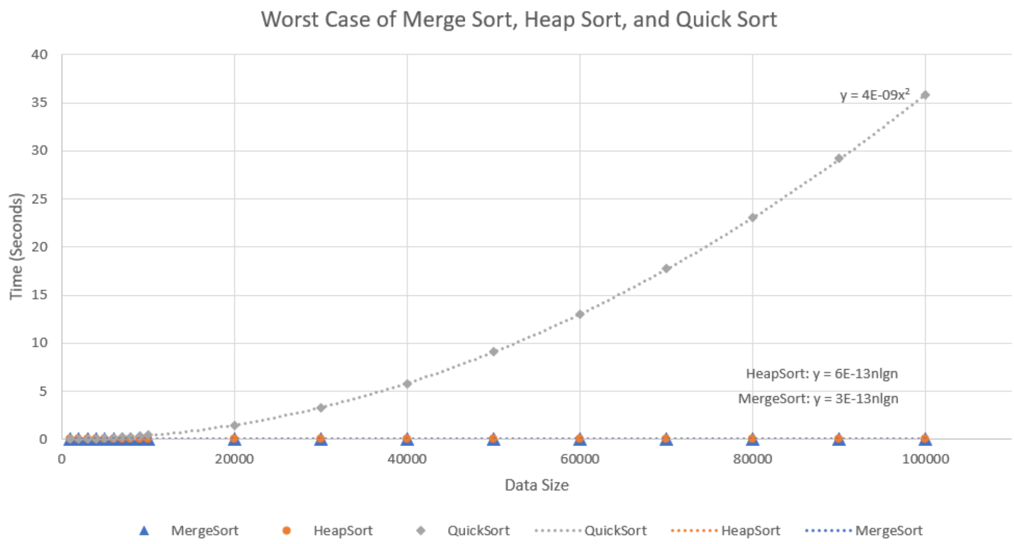
The average case runtime for each algorithm is $n \lg n$. This can be seen by sorting an array of random integers. As shown in the graph below, all three algorithms follow their expected average case runtime of $n \lg n$. Heap sort had the largest coefficient and quick sort had the smallest.



The input that gives the best case runtime for each algorithm is different. For merge sort it would be an array sorted in increasing order. This is because it would render the least amount of comparisons when merging. For heap sort it would be an array sorted in decreasing order because an array in decreasing order is already a max heap so this reduces the amount of calls to heapify. For quick sort the input would be an array where every pivot element would split the array exactly in half. Below is a graph comparing merge sort and heap sort's best case run time. Quick sort was not included because generating the input for quick sorts best case run time is significantly more difficult than merge sort and quick sort's.



Quick sort is the only one of the three algorithms whose expected worst case runtime(n^2) is different from its average case. The input that produces quick sort's worst case runtime would be an array whose pivot splits the array into a zero length array and a $n-1$ length array, aka, the largest or smallest element in the array. This array would look like a sorted array in increasing order. Both merge sort and heap sort's worst case input would be an array that maximizes the amount of comparisons. These would look similar to arrays of random integers.



8 Conclusion

In this report we showed that merge sort, heap sort, and quick sort meet their expected runtime complexity for best, worst, and average case. We also showed the required inputs for each algorithm to produce their best, worst, and average case runtimes. Quick sort was found to have the lowest leading coefficient, which means it is more efficient. Except for the worst case where it has a runtime of n^2 instead of $n\lg n$. Heap sort was found to have the greatest leading coefficient which means it was the least efficient of the three algorithms.

Appendix A - Source Code

```
////////////////////////////////////////
// @file   Sort.h
// @author Evan Wilcox
// @brief  Header file for Sort.cpp
////////////////////////////////////////

#ifndef SORT_H
#define SORT_H
#include <iostream>
#include <assert.h>
using namespace std;

////////////////////////////////////////
// @class Sort
// @brief Wrapper class for MergeSort,
//        HeapSort, and QuickSort
////////////////////////////////////////
class Sort
{
  private:
    bool debug;  // indictaes if debug mode is on

    ////////////////////////////////////////
    // @fn     swap
    // @brief  swaps the values at *a and *b
    // @pre    a and b are pointers
    // @post   the values of a* and b* are swapped
    // @param  a pointer to an int
    // @param  b pointer to an int
    // @return none
    ////////////////////////////////////////
    void swap(int *a, int *b);

    ////////////////////////////////////////
    // @fn     checkSort
    // @brief  checks if the passed array is in increasing order
    // @pre    0 <= l <= r
    // @post   true is returned if the subarry arr[l..r] is in increasing order
    // @param  arr the array containing the subarry to be checked
    // @param  l is the first index of the array to be checked
    // @param  r is the last index of the array to be cecked
    // @return bool b is true if the array is in increasing order
    ////////////////////////////////////////
    bool checkSort(int arr[], int l, int r);

    ////////////////////////////////////////
    // @fn     checkHeap
    // @brief  checks if the node i is a max heap
```

```
  // @pre    heapSize and i >= 0
  // @post   true is returned if the node i is a max heap
  // @param  arr the array containing the the node i
  // @param  heapSize the size of the heap in the the array arr
  // @praam  i the node that is being checked
  // @return returns true if the node i is a max heap
  /////////////////////////////////////////
  bool checkHeap(int arr[], int heapSize, int i);

  /////////////////////////////////////////
  // @fn     merge
  // @brief  merges two sorted subarrays
  // @pre    the subarrays arr[l..m] and arr[m+1..r] are in increasing order
  //         0 <= l <= m <= r
  // @post   the two subarrays are merged
  // @param  arr the array containing the subarrays to be sorted
  // @param  l left index of the first array
  // @param  m the middle index between the two subarrays
  // @param  r right index of the second subarray
  // @return none
  /////////////////////////////////////////
  void merge(int arr[], int l, int m, int r);

  /////////////////////////////////////////
  // @fn     heapify
  // @brief  turns the node i into a max heap
  // @pre    n and i >= 0
  // @post   the node i is a max heap
  // @param  arr the array containing the node i
  // @param  n the heapsize of the heap in the array arr
  // @param  i the node to be turned into a max heap
  // @return none
  /////////////////////////////////////////
  void heapify(int arr[], int n, int i);

  /////////////////////////////////////////
  // @fn     partition
  // @brief  partitions the subarry arr[l..h]
  // @pre    0 <= l <= h
  // @post   the subarray arr[l..h] is partitioned
  // @param  arr the array containing the subarry to be pratitioned
  // @param  l the left index of the subarry
  // @param  h the right index of the subarray
  // @return the index of the element used as the partition
  /////////////////////////////////////////
  int partition(int arr[], int l, int h);

public:
  /////////////////////////////////////////
  // @fn     Sort
```

```
// @brief  default constructor
// @pre    none
// @post   Sort object is created
// @return none
////////////////////////////////////////
Sort();

////////////////////////////////////////
// @fn     Sort
// @brief  constructor that sets debug
// @pre    none
// @post   Sort object is created
// @param  b the debug mode to be set
// @return none
////////////////////////////////////////
Sort(bool b);

////////////////////////////////////////
// @fn     setDebug
// @brief  changes the debug mode
// @pre    none
// @post   debug = b
// @param  b the new debug mode
// @return none
////////////////////////////////////////
void setDebug(bool b);

////////////////////////////////////////
// @fn     print
// @brief  prints out the given array
// @pre    n >= 0
// @post   arr is printed out using cout
// @param  arr and array of integers
// @param  n the length of arr
// @return none
////////////////////////////////////////
void print(int arr[], int n);

////////////////////////////////////////
// @fn     mergeSort
// @brief  recursive implementation of the quicksort algorithm
// @pre    0 <= l <= r
// @post   arr[l..r] is sorted in increasing order
// @param  arr the array containing the subarry to be sorted
// @param  l the left index of the subarry
// @param  r the right index of the subarray
// @return none
////////////////////////////////////////
void mergeSort(int arr[], int l, int r);
```

```
    ////////////////////////////////////////
    // @fn     heapSort
    // @brief  recursive implementation of heapsort
    // @pre    n is the length of arr,
    //         0 <= n
    // @post   arr is in increasing order
    // @param  arr the array to be sorted
    // @param  n the length of arr
    // @return none
    ////////////////////////////////////////
    void heapSort(int arr[], int n);

    ////////////////////////////////////////
    // @fn     quickSort
    // @brief  recursive implementation of quicksort
    // @pre    0 <= l <= h
    // @post   arr[l..h] is in increasing order
    // @param  arr the array containing the subarry to be sorted
    // @param  l the left index of the subarry to be sorted
    // @param  h the right index of the subarry to be sorted
    // @return none
    ////////////////////////////////////////
    void quickSort(int arr[], int l, int h);
};

#endif


///////////////////////////////////////////////////
// @file   Sort.cpp
// @author Evan Wilcox
// @brief  Function definitions for the Sort class
///////////////////////////////////////////////////

#include "Sort.h"

void Sort::swap(int *a, int *b)
{
  int temp = *a;
  *a = *b;
  *b = temp;
}


bool Sort::checkSort(int arr[], int l, int r)
{
  // Precondition
  if(debug)
  {
    assert(0 <= l);
```

```
    assert(l <= r);
  }

  for(int i = l; i < r-1; i++)
  {
    if(arr[i] > arr[i+1])
    {
      return false;
    }
  }

  return true;
}


bool Sort::checkHeap(int arr[], int heapSize, int i)
{
  // Precondition
  if(debug)
  {
    assert(0 <= heapSize);
    assert(0 <= i);
  }

  if((i * 2 + 1) < heapSize)
  {
    if(arr[i*2+1] > arr[i])
    {
      return false;
    }
  }

  if((i * 2 + 2) < heapSize)
  {
    if(arr[i*2+2] > arr[i])
    {
      return false;
    }
  }

  return true;
}


void Sort::merge(int arr[], int l, int m, int r)
{
  // Preconditions
  if(debug)
  {
    assert(0 <= l);
```

```
  assert(l <= m);
  assert(m <= r);
  assert(checkSort(arr, l, m));
  assert(checkSort(arr, m+1, r));
}


int i, j, k;
int n1 = m - l + 1;
int n2 = r - m;

int *L = new int[n1];
int *R = new int[n2];

for(i = 0; i < n1; i++)
{
  L[i] = arr[l+i];
}
for(j = 0; j < n2; j++)
{
  R[j] = arr[m+1+j];
}

i = 0;
j = 0;
k = l;

////////////////////////////////////////////
// merges the arrays L and R in to arr
// loop precondition:  arrays L and R are in sorted increasing order
// loop postconsition: arr[l..r] is in increasing order
// ivariant: arr[l..k] contains the smalest elements from
//           L and R in increasing order
// proof:
//    initialization: before the loop k=l so the subarry arr[l..k]
//                    has no elements so it is sorted
//
//    maintenance: either an element from L or R gets moved to arr
//                 and k increases each iteration so arr[l..k] still
//                 contains elements from L and R
//
//    termination: at termination k = r so the subarry arr[l..r] contains
//                 the smallest elements from L and R at termination
//
///////////////////////////////////////////
while(i < n1 && j < n2)
{
  // Invariant
  if(debug)
  {
```

```
      assert(checkSort(arr, l, k));
    }

    if(L[i] <= R[j])
    {
      arr[k] = L[i];
      i++;
    }
    else
    {
      arr[k] = R[j];
      j++;
    }
    k++;
  }

  while(i < n1)
  {
    arr[k] = L[i];
    i++;
    k++;
  }

  while(j < n2)
  {
    arr[k] = R[j];
    j++;
    k++;
  }

  delete L;
  delete R;

  // Postcondiion
  if(debug)
  {
    assert(checkSort(arr, l, r));
  }
}


void Sort::heapify(int arr[], int n, int i)
{
  // Preconditions
  if(debug)
  {
    assert(n >= 0);
    assert(i >= 0);
  }
```

```
  int largest = i;
  int l = 2*i + 1;
  int r = 2*i + 2;

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

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

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

  // Postcondition
  if(debug)
  {
    assert(checkHeap(arr, n, largest));
  }
}


int Sort::partition(int arr[], int l, int h)
{
  // Precondition
  if(debug)
  {
    assert(0 <= l);
    assert(l <= h);
  }

  int x = arr[h];
  int i = (l - 1);

  for(int j = l; j <= h-1; j++)
  {
    if(arr[j] <= x)
    {
      i++;
      swap(&arr[i], &arr[j]);
    }
  }

  swap(&arr[i+1], &arr[h]);
```

```
  return (i+1);
}


Sort::Sort()
{
  debug = false;
}


Sort::Sort(bool b)
{
  debug = b;
}


void Sort::setDebug(bool b)
{
  debug = b;
}


void Sort::print(int arr[], int n)
{
  // Precondition
  if(debug)
  {
    assert(0 <= n);
  }

  for(int i = 0; i < n-1; i++)
  {
    cout << arr[i] << ", ";
    if((i+1) % 10 == 0)
    {
      cout << endl;
    }
  }

  cout << arr[n-1] << endl << endl;
}


void Sort::mergeSort(int arr[], int l, int r)
{
  // Preconditions
  if(debug)
  {
    assert(0 <= l);
    assert(l <= r);
```

```
  }

  if(l < r)
  {
    int m = l+(r-l)/2;

    mergeSort(arr, l, m);
    mergeSort(arr, m+1, r);
    merge(arr, l, m, r);
  }

  // Postcondition
  if(debug)
  {
    assert(checkSort(arr, l, r));
  }
}


void Sort::heapSort(int arr[], int n)
{
  // Preconditions
  if(debug)
  {
    assert(0 <= n);
  }

  ////////////////////////////////////////////
  // this for loop builds a max heap
  // loop precondition:  n is the number of elements in arr
  // loop postconsition: arr is not a max heap
  // ivariant: the nodes i+1, i+2, .., n are roots of
  //           a max heap
  // proof:
  //    initialization: i = n/2 so the nodes n/2 +1, n/2 +2,
  //                    ..., n are leafs so are trivial roots of
  //                    max heaps
  //
  //    maintenance: the children of i are numbered higher than i,
  //                 by the loop invariant they are both roots of max heaps
  //                 decrementing i restablishes the invariant
  //
  //    termination: at termination i = -1, each node 1, 2, 3 are
  //                 roots of max heaps
  //
  ///////////////////////////////////////////
  for(int i = n/2 - 1;i >= 0; i--)
  {
    heapify(arr, n, i);
```

```
    // Invariant
    if(debug)
    {
      for(int j = n/2; j < n; j++)
      {
        assert(checkHeap(arr, n, j));
      }
    }
  }

  for(int i = n-1; i >= 0; i--)
  {
    swap(&arr[0], &arr[i]);
    heapify(arr, i, 0);
  }

  // Postcondition
  if(debug)
  {
    assert(checkSort(arr, 0, n));
  }
}


void Sort::quickSort(int arr[], int l, int h)
{
  // Preconditions
  if(debug)
  {
    assert(0 <= l);
    assert(l <= h);
  }

  if(l < h)
  {
    int p = partition(arr, l, h);
    quickSort(arr, l, p-1);
    quickSort(arr, p+1, h);
  }

  // Postcondition
  if(debug)
  {
    assert(checkSort(arr, l, h));
  }
}
```

```
//////////////////////////////////////////////
// @file   main.cpp
// @author Evan Wilcox
// @brief  Program used to test the Sort Class
//////////////////////////////////////////////

#include <iostream>
#include "Sort.h"
#include <random>
#include <time.h>
using namespace std;

int main()
{
  Sort S;
  S.setDebug(true);

  const unsigned long MAX_SIZE = 1000000000;

  int *A = new int[MAX_SIZE];
  int *B = new int[MAX_SIZE];
  int *C = new int[MAX_SIZE];

  int rInt;
  clock_t t;

  unsigned n = 10;
  unsigned k = n;

  // Testing the algorithms with arrays of random integers to find average runtime.
  for( ; n < 10000; n+=k)
  {
    cout << "n = " << n << endl;

    for(unsigned j = 0; j < n; j++)
    {
      rInt = rand() % (n*10);
      A[j] = rInt;
      B[j] = rInt;
      C[j] = rInt;
    }

    t = clock();
    S.mergeSort(A, 0, n-1);
    cout << "MergeSort: " << (float)(clock()-t)/CLOCKS_PER_SEC << endl;

    t = clock();
    S.heapSort(B, n);
    cout << "HeapSort:  " << (float)(clock()-t)/CLOCKS_PER_SEC << endl;
```

```
    t = clock();
    S.quickSort(C, 0, n-1);
    cout << "QuickSort: " << (float)(clock()-t)/CLOCKS_PER_SEC << endl << endl;

    if(n == k*10)
    {
      k = n;
    }
  }


  n = 10;
  k = n;
  // Testing the algorithms with sorted arrays to find best/worst runtime.
  for( ; n < MAX_SIZE; n+=k)
  {
    cout << "n = " << n << endl;

    for(unsigned j = 0; j < n; j++)
    {
      A[j] = j;
      B[j] = n-j;
      C[j] = j;
    }

    t = clock();
    S.mergeSort(A, 0, n-1);
    cout << "MergeSort: " << (float)(clock()-t)/CLOCKS_PER_SEC << endl;

    t = clock();
    S.heapSort(B, n);
    cout << "HeapSort:  " << (float)(clock()-t)/CLOCKS_PER_SEC << endl;

    t = clock();
    S.quickSort(C, 0, n-1);
    cout << "QuickSort: " << (float)(clock()-t)/CLOCKS_PER_SEC << endl << endl;

    if(n == k*10)
    {
      k = n;
    }
  }

  delete A;
  delete B;
  delete C;

  return 0;
}
```