```cpp
/*
* @author Cole Van Verth
* @pengo cverth
* @email colevanverth@gmail.com
* @file main.cpp
* @assignment 3: Quicksort
*/

/*
* This program implements Lomuto and Hoare partition style quicksorts.
* The containers used by both quicksort methods are ‘std::vector‘. Upon EOF,
* ‘lomutoQuicksort‘ will be executed on the vector (default) or ‘hoareQuicksort‘
* (if "-h" flag is present). Upon completion, the sorted data is printed to
* standard out in ascending order. Leading zeros are added to
* each output entry until there are 9 total digits.
*
* ‘lomutoPartition‘ and ‘hoarePartition‘ apply ‘medianOfThree‘ if the subarray
* size is large enough (11 and 12 respectively). These constants were determined
* experiementally using a binary search algorithim (see "hoare.txt" and
* "lomuto.txt").
*
* The program compiles with no warnings using the command "g++ main.cpp". Both
* ‘hoareQuicksort‘ and ‘lomutoQuicksort’ were tested by running the program and
* sort on the same inputs. The following inputs were tested and generated the
* same output for both programs:
*    i. 1,000,000 random numbers -999,999,999 to 999,999,999 (unpadded output)
*    ii. 1,000,000 random numbers 0 to 999,999,999
*    iii. 1,000,000 numbers in ascending order from 0
*    iv. 1,000,000 numbers in descending order from 1,000,000
*    v. 1,000,000 random numers from 0 to 9999 (high number of duplicates)
*    vi. 0 numbers (edge case)
*/

#include <iostream>
#include <vector>
#include <string>
#include <iomanip>

const int lomutoConstant = 11;
const int hoareConstant = 12;

/**
* Overload for ‘hoareQuicksort‘ that provides a better user interface.
* @param ‘A‘ vector containing data to sort
* @param ‘size‘ size of vector ‘A‘
*/
void hoareQuicksort(std::vector<int> & A);

/**
* Overload for ‘lomutoQuicksort‘ that provides a better user interface.
* @param ‘A‘ vector containing data to sort
* @param ‘size‘ size of vector ‘A‘
*/
void lomutoQuicksort(std::vector<int> & A);

/**
* Quicksort algorithim that utilizes Hoare style partioning.
* @param ‘A‘ vector containing data to sort
* @param ‘p‘ leftmost indeice in ‘A‘
* @param ‘r‘ rightmost indice in ‘A‘
*/
void hoareQuicksort(std::vector<int> & A, int p, int r);

/**
* Partitions an array using a Hoare partioning style that starts from right
* and left side and iterates towards the center.
```



```cpp
* @param ‘A‘ vector containing vector to partition
* @param ‘p‘ leftmost indice of ‘A‘
* @param ‘r‘ rightmost indice of ‘A‘
* @return index of new pivot
*/
int hoarePartition(std::vector<int> & A, int p, int r);

/**
* Quicksort algorithim that utilizes Lomuto style partitioning.
* @param ‘A‘ vector containing data to sort
* @param ‘p‘ leftmost indice in ‘A‘
* @param ‘r‘ rightmost indice in ‘A‘
*/
void lomutoQuicksort(std::vector<int> & A, int p, int r);

/**
* Partitions an array using a Lomuto partioning style that starts from the
* left and iterates towards the pivot on the right.
* @param ‘A‘ vector containing data to partition
* @param ‘p‘ leftmost indice in ‘A‘
* @param ‘r‘ rightmost indice in ‘A‘
* @return index of new pivot
*/
int lomutoPartition(std::vector<int> & A, int p, int r);

/**
* Finds the median of three values in a vector.
* @param ‘A‘ vector that contains ‘indexOne‘, ‘indexTwo‘, and ‘indexThree‘
* @param ‘indexOne‘ first index to evaluate in ‘A‘
* @param ‘indexTwo‘ second index to evaluate in ‘A‘
* @param ‘indexThree‘ third index to evaluate in ‘A‘
* @return index ‘indexOne‘, ‘indexTwo‘, or ‘indexThree‘ depending on which
* index references an element with a median value of the three
*/
int medianOfThree(const std::vector<int> & A, int indexOne, int indexTwo,
                  int indexThree);

/**
* Swaps two values in a vector.
* @param ‘A‘ vector to swap values in
* @param ‘indexOne‘ index of first element to swap
* @param ‘indexTwo‘ index of second element to swap
*/
void swap(std::vector<int> & A, int indexOne, int indexTwo);

int medianOfThree(const std::vector<int> & A, int indexOne, int indexTwo,
                  int indexThree) {
   // Loads values from indexes
   int valOne = A[indexOne];
   int valTwo = A[indexTwo];
   int valThree = A[indexThree];

   // Finds median value, returns index of that element
   if (valTwo <= valOne && valOne <= valThree) {
      return indexOne;
   }
   else if (valThree <= valOne && valOne <= valTwo) {
      return indexOne;
   }
   else if (valOne <= valTwo && valTwo <= valThree) {
      return indexTwo;
   }
   else if (valThree <= valTwo && valTwo <= valOne) {
      return indexTwo;
   }
   else if (valOne <= valThree && valThree <= valTwo) {
```



```cpp
      return indexThree;
   }
   else {
      return indexThree;
   }
}

void lomutoQuicksort(std::vector<int> & A, int p, int r) {
   if (p < r) {
      int q = lomutoPartition(A, p, r);
      lomutoQuicksort(A, p, q - 1);
      lomutoQuicksort(A, q + 1, r);
   }
}

int lomutoPartition(std::vector<int> & A, int p, int r) {
   // Applies medianOfThree if size is large enough
   int arraySize = r - p;
   if (arraySize >= lomutoConstant) {
      int mid = p + ( ( r - p ) / 2 );
      int pivotIndex = medianOfThree(A, p, r, mid);
      swap(A, pivotIndex, r);
   }

   int pivotValue = A[r];
   int position = p - 1;

   for (int i = p; i < r; ++i) {
      if (A[i] <= pivotValue) {
         position++;
         swap(A, position, i);
      }
   }

   swap(A, position + 1, r); // Moves pivot to right of numbers less than it
   return position + 1; // Index of the new pivot
}

void swap(std::vector<int> & A, int indexOne, int indexTwo) {
   int temp = A[indexOne];
   A[indexOne] = A[indexTwo];
   A[indexTwo] = temp;
}

int hoarePartition(std::vector<int> & A, int p, int r) {
   int pivotIndex = p;

   // Applies medianOfThree if size is large enough
   int arraySize = r - p;
   if (arraySize >= hoareConstant) {
      int mid = p + ( ( r - p ) / 2);
      pivotIndex = medianOfThree(A, p, r, mid);
   }

   int pivotValue = A[pivotIndex];
   int leftPosition = p - 1;
   int rightPosition = r + 1;

   while (true) {
      do {
         rightPosition--;
      } while (A[rightPosition] > pivotValue);
      do {
         leftPosition++;
      } while (A[leftPosition] < pivotValue);
      if (leftPosition < rightPosition) {
```



```cpp
         swap(A, leftPosition, rightPosition);
      }
      else {
         return rightPosition;
      }
   }
}

void hoareQuicksort(std::vector<int> & A, int p, int r) {
   if (p < r) {
      int q = hoarePartition(A, p, r);
      hoareQuicksort(A, p, q);
      hoareQuicksort(A, q + 1, r);
   }
   return;
}

void hoareQuicksort(std::vector<int> & A) {
   hoareQuicksort(A, 0, A.size() - 1);
}

void lomutoQuicksort(std::vector<int> & A) {
   lomutoQuicksort(A, 0, A.size() - 1);
}

int main(int argc, char** argv) {
   int bufferNum;
   std::vector<int> array;

   // Loads values into ‘array‘
   while (std::cin >> bufferNum) {
      array.push_back(bufferNum);
   }

   // ‘hoareQuicksort‘ called if flag present
   if (argc > 1) {
      if (std::string(argv[1]) == "-h") {
         hoareQuicksort(array);
      }
   }
   else {
      lomutoQuicksort(array); // ‘lomutoQuicksort‘ called if no flag
   }

   // Prints sorted vector to standard out
   for (int i = 0; i < array.size(); i++) {
      std::cout << std::setfill('0') << std::setw(9) << array[i] << std::endl;
   }
}
```

```
p3: main.o
   g++ -o p3 main.o

main.o: main.cpp
   g++ -c main.cpp

clean:
   rm -f p3 *.o *~
```