```
#include <iostream>
#include <time.h>
#include <vector>
#include <limits>
#include <thread>
#include <cstring>

#define LIST_SIZE 50000 //Macro defining the size of the list to be sorted
#define HIGH LIST_SIZE / 2 //Macro defining the lower-bound of list integer range
#define LOW -LIST_SIZE / 2 //Macro defining the upper-bound of list integer range

/*Structure called Algo which contains information
about the performance of the sorting method as well as
general metadata used by other functionality*/
class Algo{
   public:
      Algo() = default;
      explicit Algo(int id){ //Constructor to create an Algo based on an id
         switch (id){
            case 0: //If id is 0, Algo is called compare
               this->name = "compare";
            case 1: //If id is 1, Algo is called selection sort
               this->name = "selection sort";
               break;
            case 2: //If id is 2, Algo is called bubble sort
               this->name = "bubble sort";
               break;
```

```
        case 3: //If id is 3, Algo is called insertion sort
            this->name = "insertion sort";
            break;
        case 4: //If id is 4, Algo is called quick sort
            this->name = "quick sort";
            break;
        case 5: //If id is 5, Algo is called merge sort
            this->name = "merge sort";
            break;
        default: //Else, Algo is called invalid
            this->name = "invalid";
    }

    this->id = id;
} //Constructor to create an Algo based on an id

const Algo& operator=(int id){ //Operator to allow assignment of Algos to integers
    return *this = Algo(id); //Creates a new Algo given an id and assigns it to the caller
}

void reset(){ //Reset's the algorithm's statistics
    this->access_count = 0;
    this->move_count = 0;
}

//Declare sorting algorithms
void selection_sort(int arr[], int n){
```

```
    //Create a start and stop time variable to time the sorting algorithms
    std::chrono::time_point<std::chrono::system_clock> start_time, end_time;
    start_time = std::chrono::system_clock::now(); //Get algorithm start time

    for (int i = 0; i < n - 1; ++i){
        int cur = i;

        /*Each iteration, complete another iteration,
        keeping track of the current smallest (or largest)
        element and comparing it to the jth.*/
        for (int j = i + 1; j < n; ++j){
            //Update cur if more significant index is found
            if (arr[j] < arr[cur])
                cur = j;
            this->access_count += 2;
        }

        //Swap the ith element with cur
        swap(arr[i], arr[cur]);
        this->access_count += 2;
        ++this->move_count;
    }

    end_time = std::chrono::system_clock::now(); //Get algorithm finish time
    this->time = end_time - start_time; //Set algorithm's time to the end time - start time
}
```

```
void bubble_sort(int arr[], int n){
   //Create a start and stop time variable to time the sorting algorithms
   std::chrono::time_point<std::chrono::system_clock> start_time, end_time;
   start_time = std::chrono::system_clock::now(); //Get algorithm start time

   int iter_num{}, swap_num{};

   for (int i = 0; i < n - 1; ++i){
      /*With each iteration, perform another internal
      iteration, decreasing the magnitude each time
      i increments for efficiency.*/
      for (int j = 0; j < n - i - 1; ++j){
         /*Swap two adjacent indecies if the greater
         index is more significant*/
         if (arr[j + 1] < arr[j]){
            swap(arr[j], arr[j + 1]);
            this->access_count += 2;
            ++this->move_count;
         }

         this->access_count += 2;
      }
   }

   end_time = std::chrono::system_clock::now(); //Get algorithm finish time
   this->time = end_time - start_time; //Set algorithm's time to the end time - start time
}
```

```
void insertion_sort(int arr[], int n){
   //Create a start and stop time variable to time the sorting algorithms
   std::chrono::time_point<std::chrono::system_clock> start_time, end_time;
   start_time = std::chrono::system_clock::now(); //Get algorithm start time

   for (int i = 1; i < n; ++i){
      int j = i;

      /*With each iteration, iterate from right to left
      until the index fits in place or reaches the end*/
      while (j > 0 && arr[j] < arr[j - 1]){
         //Swap with left adjacent index if out of place
         swap(arr[j], arr[j - 1]);
         --j;

         this->access_count += 4;
         ++this->move_count;
      }

      this->access_count += 2;
   }

   end_time = std::chrono::system_clock::now(); //Get algorithm finish time
   this->time = end_time - start_time; //Set algorithm's time to the end time - start time
}
```

```
void quick_sort(int arr[], int begin, int end){
   //Create a start and stop time variable to time the sorting algorithms
   std::chrono::time_point<std::chrono::system_clock> start_time, end_time;

   if (begin == 0 && end == LIST_SIZE - 1){
      start_time = std::chrono::system_clock::now(); //Get algorithm start time
   }

   //Sorts if the segment is greatet than one element
   if (begin < end)
   {
      //Performs recursion on several pivots to sort the list
      int pivot_index = partition(arr, begin, end);

      //Recursively sort each side of a pivot
      quick_sort(arr, begin, pivot_index - 1);
      quick_sort(arr, pivot_index + 1, end);
   }

   if (begin == 0 && end == LIST_SIZE - 1){
      end_time = std::chrono::system_clock::now(); //Get algorithm finish time
      this->time = end_time - start_time; //Set algorithm's time to the end time - start time
   }
}

void merge_sort(int arr[], int begin, int end){
   //Create a start and stop time variable to time the sorting algorithms
```

```
    std::chrono::time_point<std::chrono::system_clock> start_time, end_time;

    if (begin == 0 && end == LIST_SIZE - 1){
        start_time = std::chrono::system_clock::now(); //Get algorithm start time
    }

    //Split up array until segment of maximum length 2
    if (begin < end){
        //Variable to store the middle index of the array segment
        int mid = (begin + end) / 2;

        //Split up the left side of the segment into smaller segments
        merge_sort(arr, begin, mid);
        //Split up the right  side of the segment into smaller segments
        merge_sort(arr, mid + 1, end);
        //Sort and merge the two segments split up
        merge(arr, begin, mid, end);
    }

    if (begin == 0 && end == LIST_SIZE - 1){
        end_time = std::chrono::system_clock::now(); //Get algorithm finish time
        this->time = end_time - start_time; //Set algorithm's time to the end time - start time
    }
}

//Get functions, do not allow for modifications due to const specifier
std::string algo_name() const{ return this->name; }
```

```
    int algo_id() const{ return this->id; }
    uint64_t algo_accessc() const{ return this->access_count; }
    uint64_t algo_movec() const{ return this->move_count; }
    std::chrono::duration<double> algo_time() const{ return this->time; }

    //Number of algorithms available to choose from
    static constexpr int num_options{5};

  private:
    std::string name; //Indicates the name of the sorting method
    int id; //Associates an id with the sorting method
    //Counters for the number of array accesses and data moves performed by the sorting method
    uint64_t access_count{}, move_count{};
    std::chrono::duration<double> time{}; //Variable to store the time taken to sort the list using a method

    //Swap the data in two elements passed to swap()
    void swap(int& x, int& y){
      int temp = x;
      x = y;
      y = temp;
    }

    //Create pivots in an array of data to sort the array
    int partition(int arr[], int begin, int end){
      //Use first index in array as pivot
      int pivot_data = arr[begin];
```

```
//Iterate list an determine pivot absolute point in sorted list
int count{};
for (int i = begin + 1; i <= end; i++) {
   if (arr[i] <= pivot_data)
      ++count, ++this->access_count;
}

//Update the posititon of the pivot in its absolute point
int pivot_index = begin + count;
swap(arr[pivot_index], arr[begin]);
this->access_count += 2;
++this->move_count;

int b = begin, a = end;

/*Loops until all the smaller elements are before the pivot
and all the larger elements are after the pivot*/
while (b < pivot_index && a > pivot_index) {
   while (arr[b] <= pivot_data)
      ++b, ++this->access_count;

   while (arr[a] > pivot_data)
      --a, ++this->access_count;

   /*Swaps elements on each side of pivot
   element i is before pivot and element j is after*/
```

```
            if (b < pivot_index && a > pivot_index) {
                swap(arr[b++], arr[a--]);
                this->access_count += 2;
                ++this->move_count;
            }
        }

        /*Returns the pivot index for quick_sort() to perform
        recursion and sort both sides of the pivot*/
        return pivot_index;
    }

    //Sort and merge two subarrays into one larger sorted array
    void merge(int arr[], int const left, int const mid, int const right){
        int left_index = left, //First index of the left array segment
            right_index = mid + 1, //First index of the right array segment
            merge_index = left; //First index of the merged array segment
        int* temp = new int[right - left + 1]; //Created the container for the merged array
segment

        /*Loop until either the end of the left segment
        or right segment has been reached*/
        while (left_index <= mid && right_index <= right){
            /*If the value in the left segment index
            is smaller, sub it into the merge array segment
            and increment each counter*/
            if (arr[left_index] < arr[right_index]){
```

```
        temp[merge_index - left] = arr[left_index];
        ++left_index, ++this->access_count, ++this->move_count;
    }
    /*Else if the value in the right segment index
    is smaller, sub it into the merge array segment
    and increment each counter*/
    else{
        temp[merge_index - left] = arr[right_index];
        ++right_index, ++this->access_count, ++this->move_count;
    }

    this->access_count += 2;
    ++merge_index;
}

/*Since the end of one side of the array segment may
be reached before the other, fill in the rest of
the merged array with the leftover contents
of the unfinished array*/
while (left_index <= mid){ //For the left array segment
    temp[merge_index - left] = arr[left_index];
    ++left_index, ++merge_index, ++this->access_count, ++this->move_count;
}
while (right_index <= right){ //For the right array segment
    temp[merge_index - left] = arr[right_index];
    ++right_index, ++merge_index, ++this->access_count, ++this->move_count;
}
```

```
        /*Once done sorting the merged array segment,
        copy each element into the actual original array*/
        for (int i = 0; i < right - left + 1; ++i)
        {
            arr[left + i] = temp[i];
            ++this->move_count;
        }
    }
};

//Abstraction to receive input from the user and format an option accordingly
void receive_input(Algo& input){
   //Extract characters from the standard input stream and store them in the integer variable id
   int id;
   std::cin >> id;

   input = id; //Calls integer to option assignment operator

   //Error handling if the input stream fails to extract characters
   if (std::cin.fail())
   {
      std::cin.clear();
      std::cin.ignore(std::numeric_limits<int>::max(), '\n');

      input = -1;
   }
```

```
}

//Initialize an array of integers with random integers between low and high
void list_init(int low, int high, int (&list)[LIST_SIZE]){
   for (auto& x : list)
      x = (rand() % (high - low)) + low; //Generates a random number between low (inclusive) and high (exclusive)
}

//Using a vector of Algos, print a table containing information about the sorting methods compared.
void get_results(const std::vector<Algo>& options){
   Algo stats[6]; //Array of six Algo objects
   /*stats[0] for fastest time, stats[1] for lowest array access count, stats[2] for lowest data move count
   stats[3] for slowest time, stats[4] for highest array access count, stats[5] for highest data move count*/
   for (int i = 0; i < 6; ++i)
      stats[i] = options[0]; //Initialize array of six Algos

   //Cycle through list of Algos and compare to current stats stored
   for (int i = 1; i < options.size(); ++i){
      if (options[i].algo_time() < stats[0].algo_time())
         stats[0] = options[i]; //If lower time is found, update stats[0]

      if (options[i].algo_accessc() < stats[1].algo_accessc())
         stats[1] = options[i]; //If lower access count found, update stats[1]
```

```
        if (options[i].algo_movec() < stats[2].algo_movec())
            stats[2] = options[i]; //If lower move count found, update stats[2]

        if (options[i].algo_time() > stats[3].algo_time())
            stats[3] = options[i]; //If higher time found, update stats[3]

        if (options[i].algo_accessc() > stats[4].algo_accessc())
            stats[4] = options[i]; //If higher access count found, update stats[4]

        if (options[i].algo_movec() > stats[5].algo_movec())
            stats[5] = options[i]; //If higher move count found, updates stats[5]
    }

    //Display table of statistics calculated
    std::cout << "\n__RESULTS__\n";
    std::cout << "FASTEST TIME: {type: " << stats[0].algo_name() << "}, {time: " <<
stats[0].algo_time().count() << "s}\n";
    std::cout << "LOWEST # OF ARRAY ACCESSES: {type: " << stats[1].algo_name() << "},
{accesses: " << stats[1].algo_accessc() << "}\n";
    std::cout << "LOWEST # OF DATA MOVES: {type: " << stats[2].algo_name() << "}, {moves: " <<
stats[2].algo_movec() << "}\n\n";

    std::cout << "SLOWEST TIME: {type: " << stats[3].algo_name() << "}, {time: " <<
stats[3].algo_time().count() << "s}\n";
    std::cout << "HIGHEST # OF ARRAY ACCESS: {type: " << stats[4].algo_name() << "}, {accesses: "
<< stats[4].algo_accessc() << "}\n";
    std::cout << "HIGHEST # OF DATA MOVES: {type: " << stats[5].algo_name() << "}, {moves: " <<
stats[5].algo_movec() << "}\n\n";
}
```

```
//Initialize the array to be sorted, run the sorting methods, and print metadata about their execution
void compare_options(std::vector<Algo>& options){
   int unsorted_list[LIST_SIZE], copy_list[LIST_SIZE]; //Create two lists of size LIST_SIZE to hold the data


   list_init(LOW, HIGH, unsorted_list); //Initialize the first list created with random numbers between -25,000 and 25,000
   memcpy(copy_list, unsorted_list, sizeof(unsorted_list)); //Perform a byte-wise copy on the first list into the copy list


   std::endl(std::cout);
   std::cout << "__SORTING DATA (1s delay)__\n";
   //Loop through list of Algos and execute each one
   for (auto& option : options){
      //Switch statement to choose which algorithm to execute based on the current option in the option list
      switch (option.algo_id()){
         case 1:
            option.selection_sort(unsorted_list, LIST_SIZE); //Execute selection sort
            break;
         case 2:
            option.bubble_sort(unsorted_list, LIST_SIZE); //Execute bubble sort
            break;
         case 3:
            option.insertion_sort(unsorted_list, LIST_SIZE); //Execute insertion sort
            break;
         case 4:
```

```
            option.quick_sort(unsorted_list, 0, LIST_SIZE - 1); //Execute quick sort
            break;
        case 5:
            option.merge_sort(unsorted_list, 0, LIST_SIZE - 1); //Execute merge sort
            break;
    }

    //Print metadata about sorting algorithm
    std::cout << "Sorting " << LIST_SIZE << " integers (" << LOW << " <=> " << HIGH << ") {type: "
<< option.algo_name() <<
        "}, {id: " << option.algo_id() << "} took " << option.algo_time().count() << "s to sort the
list, "
        << option.algo_accessc() << " array accesses, " << option.algo_movec() << " data
moves..." << std::endl;

    memcpy(unsorted_list, copy_list, sizeof(unsorted_list)); /*Perform a byte-wise copy on the
copy list into the original list ,
    so the unsorted data can remain constant throughout each test*/

    std::this_thread::sleep_for(std::chrono::seconds(1)); //Pause the thread of execution for
one second, so the metadata can be viewed more easily
  }
  std::endl(std::cout);

  get_results(options); //Call get_results() and print a table of notable statistics about all the
trials
}

//Interface/menu for the user to choose the sorting methods to be compared
```

```
int get_options(std::vector<Algo>& options){
    Algo option(-1); //Create a new Algo object


    //Requests the user to enter a number 0 through 5 so it may pass the execution to the comparing segment
    std::cout << "\nPlease add a sorting algorithm (1-" << Algo::num_options << "), press (0) to compare, or anything else to exit...\n";
    std::cout << "Currently selected:\n";
    if (options.size() > 0){
        for (auto& x : options){
            std::cout << "=> {type: " << x.algo_name() << "}, {id: " << x.algo_id() << "}\n";
        }
    }
    else
        std::cout << "=> None\n";


    std::cout << "\n__Menu__\n";
    std::cout << "0 - Compare\n";
    std::cout << "1 - Selection sort\n";
    std::cout << "2 - Bubble sort\n";
    std::cout << "3 - Insertion sort\n";
    std::cout << "4 - Quick sort\n";
    std::cout << "5 - Merge sort\n";


    //Loops until the user enters a number not on the menu, such as -5 or the letter 'k'
    do{
        bool removed = false;
```

```
    receive_input(option); //Call the input function to extract characters from the input stream and configure the option


    //If the option's id is greater than 0 and less than 5, either add the option to the option list or remove it if already present
    if (option.algo_id() > 0 && option.algo_id() <= Algo::num_options){
        int count{};


        //Loops through the options and removes the current chosen option if in the list
        for (auto& x : options){
            if (x.algo_id() == option.algo_id()){
                options.erase(options.begin() + count);
                std::cout << "[ Removed option {type: " << option.algo_name() << "}, {id: " << option.algo_id() << "} from comparison ]\n";
                removed = true;
            }


            ++count;
        }


        //If nothing was removed from the list, add the option to the options list
        if (!removed)
        {
            options.push_back(option);
            std::cout << "[ Added option {type: " << option.algo_name() << "}, {id: " << option.algo_id() << "} to comparison ]\n";
        }
    }
```

```
    //Else if the option's id is equal to 0, exit the function and return control to the main function if there are more than two options selected
    else if (option.algo_id() == 0){
      if (options.size() < 2){ //If less than two options are selected, alert the user and set option to 1 so the loop continues
        std::cout << "Must have at least two options selected to compare!\n";
        option = 1;
      }
    }
    //Else if the option's id is greater than 0 as well as 5 in this case, alert the user that the option number is invalid
    else if (option.algo_id() > 0){
      std::cout << "Algo number must be in the range of (1-" << Algo::num_options << ")!\n";
    }
    else return 0; //Return 0 if invalid input and return control to main function
  } while(option.algo_id() > 0);

  return 1; //If everything suceeds, return 1 and return control to the main function
}

int main(){
  srand(time(0)); //Initialize random class with the current system time seed for realistic random number generation

  std::vector<Algo> options; //Vector of Algos to store the list of options
  char input; //Input variable to store a character for yes/no questions

  //Loop comparing sorting algorithms until the user says they don't want to (N)
```

```
   do{
      if (get_options(options)){ //Get sorting algorithms and push them to the options list. Execute the list if anything but zero is returned, meaning success
         do{
            compare_options(options); //Run the chosen options/sorting algorithms and compare them


            for (auto& x : options) //Resets the statistics recorded of each algorithm
               x.reset();


            //Request if the user would like to run the test again with the same sorting styles
            std::cout << "Would you like to compare these sorting styles again? (y/N)\n";
            std::cin >> input;
         } while(tolower(input) == 'y');
      }


      //Request if the user would like to quit the application
      std::cout << "Would you like to quit the application? (Y/n)\n";
      std::cin >> input;
   } while (tolower(input) == 'n');


   //If the user wants to quit the application, continue to the end of main() where the program will exit naturally
   std::cout << "Quitting program in... ";
   for (int i = 3; i > 0; --i){
      std::cout << i << ' ' << std::flush;
      std::this_thread::sleep_for(std::chrono::seconds(1)); //Pause the current thread of execution for three seconds, while printing a countdown in between each pause
```

```
    }
    std::endl(std::cout);
}
```