Listing 1: SearchNSort.hpp

```
#pragma once

#include <algorithm>
#include <ctime>
#include <doctest.h>
#include <random>
#include <utility>

/*-----------------------------------------------------------------------------
 * class definition
 *---------------------------------------------------------------------------*/

/**
 * @brief CMP 246 Module 8 templated class with searching and sorting methods.
 *
 * This class provides a series of static methods for the following operations:
 *
 * - searches: linearSearch and binarySearch
 *
 * - quadratic sorts: bubbleSort, insertionSort, selectionSort
 *
 * - n lg n sorts: mergeSort, quickSort
 */
template <class T> class SearchNSort {
public:
  /**
   * @brief Perform a binary search on an array.
   *
   * @param pArr Pointer to the first element of the array to search. The array
   * must be sorted in ascending order.
   * @param n Number of elements in the array.
   * @param key Key value to search for
   * @param compare Pointer to a function used to compare two elements. This
   * must return a negative value if x < y, zero if x == y, or positive if
   * x > y.
   * @return int Index of an occurrence of key in pArr, or -1 if the key
   * was not found in the array.
   */
  static int binarySearch(const T *pArr, size_t n, const T &key,
                          int (*compare)(const T &x, const T &y));

  /**
   * @brief Sort an array using the bubble sort algorithm.
   *
   * @param pArr Pointer to the first element of the array to search.
   * @param n Size of the array.
   * @param compare Pointer to a function used to compare two elements. This
   * must return a negative value if x < y, zero if x == y, or positive if
   * x > y.
   */
  static void bubbleSort(T *pArr, size_t n,
                         int (*compare)(const T &x, const T &y));

  /**
   * @brief Sort an array using the insertion sort algorithm.
   *
   * @param pArr Pointer to the first element of the array to search.
   * @param n Size of the array.
   * @param compare Pointer to a function used to compare two elements. This
   * must return a negative value if x < y, zero if x == y, or positive if
   * x > y.
   */
  static void insertionSort(T *pArr, size_t n,
                            int (*compare)(const T &x, const T &y));

  /**
   * @brief Perform a linear search on an array.
   *
   * @param pArr Pointer to the first element of the array to search.
   * @param n Number of elements in the array.
   * @param key Key value to search for
   * @param compare Pointer to a function used to compare two elements. This
   * must return a negative value if x < y, zero if x == y, or positive if
   * x > y.
```

```
75      * @return int Index of the first occurrence of key in pArr, or -1 if the key
76      * was not found in the array.
77      */
78     static int linearSearch(const T *pArr, size_t n, const T &key,
79                             int (*compare)(const T &x, const T &y));
80
81     /**
82      * @brief Sort an array using the merge sort algorithm.
83      *
84      * @param pArr Pointer to the first element of the array to search.
85      * @param n Size of the array.
86      * @param compare Pointer to a function used to compare two elements. This
87      * must return a negative value if x < y, zero if x == y, or positive if
88      * x > y.
89      */
90     static void mergeSort(T *pArr, size_t n,
91                           int (*compare)(const T &x, const T &y));
92
93     /**
94      * @brief Sort an array using the quicksort algorithm.
95      *
96      * @param pArr Pointer to the first element of the array to search.
97      * @param n Size of the array.
98      * @param compare Pointer to a function used to compare two elements. This
99      * must return a negative value if x < y, zero if x == y, or positive if
100     * x > y.
101     */
102    static void quickSort(T *pArr, size_t n,
103                          int (*compare)(const T &x, const T &y)) {
104
105      quickSort(pArr, 0, n - 1, compare);
106    }
107
108    /**
109     * @brief Sort an array using the selection sort algorithm.
110     *
111     * @param pArr Pointer to the first element of the array to search.
112     * @param n Size of the array.
113     * @param compare Pointer to a function used to compare two elements. This
114     * must return a negative value if x < y, zero if x == y, or positive if
115     * x > y.
116     */
117    static void selectionSort(T *pArr, size_t n,
118                              int (*compare)(const T &x, const T &y));
119
120 private:
121    /**
122     * @brief Merge two sorted portions of an array together.
123     *
124     * @param pA Array containing two sorted portions
125     * @param pB Scratch array destination
126     * @param left left index of left sorted portion
127     * @param right one past right index of right sorted portion
128     * @param mid start of right sorted portion
129     * @param compare Pointer to a function used to compare two elements. This
130     * must return a negative value if x < y, zero if x == y, or positive if
131     * x > y.
132     */
133    static void merge(T *pA, T *pB, size_t left, size_t right, size_t mid,
134                      int (*compare)(const T &x, const T &y));
135
136    /**
137     * @brief Recursive helper function for mergeSort
138     *
139     * @param pA Pointer to first element of the array to sort
140     * @param pB Pointer to first element of scratch array
141     * @param left Index of leftmost element in the section to be sorted
142     * @param right One past index of rightmost element in the section to be
143     * sorted
144     * @param compare Pointer to a function used to compare two elements. This
145     * must return a negative value if x < y, zero if x == y, or positive if
146     * x > y.
147     */
148    static void mergeSort(T *pA, T *pB, size_t left, size_t size_t,
149                          int (*compare)(const T &x, const T &y));
150
```

```
151   /**
152    * @brief Partitioning helper function for quickSort
153    *
154    * @param pArr Pointer to first element of the array to sort.
155    * @param lo Leftmost index in range being sorted.
156    * @param hi Rightmost index in range being sorted.
157    * @param compare Pointer to a function used to compare two elements. This
158    * must return a negative value if x < y, zero if x == y, or positive if
159    * x > y.
160    * @return int index in [lo, hi] such that everything to the left is less
161    * than or equal to everything in the right.
162    */
163   static size_t partition(T *pArr, size_t lo, size_t hi,
164                           int (*compare)(const T &x, const T &y));
165
166   /**
167    * @brief Recursive helper function for quickSort.
168    *
169    * @param pArr Pointer to first element of the array to sort.
170    * @param lo Index of the leftmost element in range being sorted.
171    * @param hi Index of the rightmost element in range being sorted.
172    * @param compare Pointer to a function used to compare two elements. This
173    * must return a negative value if x < y, zero if x == y, or positive if
174    * x > y.
175    */
176   static void quickSort(T *pArr, size_t lo, size_t hi,
177                         int (*compare)(const T &x, const T &y));
178 };
179
180 //-----------------------------------------------------------------------------
181 // function implementations
182 //-----------------------------------------------------------------------------
183
184 /*
185  * Implementation of iterative binarySearch() function.
186  */
187 template <class T>
188 int SearchNSort<T>::binarySearch(const T *pArr, size_t n, const T &key,
189                                  int (*comp)(const T &x, const T &y)) {
190   size_t i = 0, j = n - 1, mid;
191   while (i <= j) {
192     mid = (i + j) / 2;
193     int res = comp(pArr[mid], key);
194     if (res == 0) {
195       return mid;
196     } else if (res > 0) {
197       j = mid - 1;
198     } else {
199       i = mid + 1;
200     }
201   }
202   return -1;
203 }
204
205 // doctest unit test for binarySearch
206 TEST_CASE("testing SearchNSort::binarySearch") {
207   // make array for searching; make sure it does not have 0, but does have 1
208   int pA[100];
209   std::mt19937_64 prng(time(0));
210   std::uniform_int_distribution<int> dist(-100, 100);
211   for(size_t i = 0; i < 100; i++) {
212     int v = dist(prng);
213     pA[i] = v == 0 ? 1 : v;
214   }
215   pA[17] = 1;
216
217   // sort the array
218   std::sort(pA, pA + 100);
219
220   // lambda function for comparing elements for the search
221   auto cmp = [](const int &a, const int &b) { return a - b; };
222
223   // search for 1 should work
224   int idx = SearchNSort<int>::binarySearch(pA, 100, 1, cmp);
225   CHECK(-1 != idx);
226   // search for 0 should not work
```

```
227   idx = SearchNSort<int>::binarySearch(pA, 100, 0, cmp);
228   CHECK(-1 == idx);
229 }
230
231 /*
232  * Implementation of bubbleSort() function.
233  */
234 template <class T>
235 void SearchNSort<T>::bubbleSort(T *pArr, size_t n,
236                                 int (*comp)(const T &x, const T &y)) {
237
238   do {
239     size_t newN = 0u;
240     for (size_t i = 1u; i < n; i++) {
241       if (comp(pArr[i - 1], pArr[i]) > 0) {
242         std::swap(pArr[i - 1], pArr[i]);
243         newN = i;
244       }
245     }
246     n = newN;
247   } while (n != 0u);
248 }
249
250 // doctest unit test for bubbleSort
251 TEST_CASE("testing␣SearchNSort::bubbleSort") {
252   // make two arrays for sorting
253   int pA[100], pB[100];
254   std::mt19937_64 prng(time(0));
255   std::uniform_int_distribution<int> dist(-100, 100);
256   for(size_t i = 0; i < 100; i++) {
257     pA[i] = pB[i] = dist(prng);
258   }
259
260   // lambda function for comparing elements for the sort
261   auto cmp = [](const int &a, const int &b) { return a - b; };
262
263   // sort using bubbleSort
264   SearchNSort<int>::bubbleSort(pA, 100, cmp);
265
266   // sort using std::sort
267   std::sort(pB, pB + 100);
268
269   // check elements are the same
270   for(size_t i = 0; i < 100; i++) {
271     CHECK(pA[i] == pB[i]);
272   }
273 }
274
275 /*
276  * Implementation of insertionSort() function.
277  */
278 template <class T>
279 void SearchNSort<T>::insertionSort(T *pArr, size_t n,
280                                    int (*comp)(const T &x, const T &y)) {
281
282   for (size_t i = 1u; i < n; i++) {
283     size_t j = i;
284     while (j > 0u && comp(pArr[j - 1], pArr[j]) > 0) {
285       std::swap(pArr[j], pArr[j - 1]);
286       j--;
287     }
288   }
289 }
290
291 // doctest unit test for insertionSort
292 TEST_CASE("testing␣SearchNSort::insertionSort") {
293   // make two arrays for sorting
294   int pA[100], pB[100];
295   std::mt19937_64 prng(time(0));
296   std::uniform_int_distribution<int> dist(-100, 100);
297   for(size_t i = 0; i < 100; i++) {
298     pA[i] = pB[i] = dist(prng);
299   }
300
301   // lambda function for comparing elements for the sort
302   auto cmp = [](const int &a, const int &b) { return a - b; };
```

```
303
304   // sort using insertionSort
305   SearchNSort<int>::insertionSort(pA, 100, cmp);
306
307   // sort using std::sort
308   std::sort(pB, pB + 100);
309
310   // check elements are the same
311   for(size_t i = 0; i < 100; i++) {
312     CHECK(pA[i] == pB[i]);
313   }
314 }
315
316 /*
317  * Implementation of linearSearch() function.
318  */
319 template <class T>
320 int SearchNSort<T>::linearSearch(const T *pArr, size_t n, const T &key,
321                                  int (*comp)(const T &x, const T &y)) {
322
323   for (size_t i = 0u; i < n; i++) {
324     if (comp(pArr[i], key) == 0) {
325       return i;
326     }
327   }
328
329   // not found? Return -1 flag value
330   return -1;
331 }
332
333 // doctest unit test for linearSearch
334 TEST_CASE("testing SearchNSort::linearSearch") {
335   // make array for searching; make sure it does not have 0, but does have 1
336   int pA[100];
337   std::mt19937_64 prng(time(0));
338   std::uniform_int_distribution<int> dist(-100, 100);
339   for(size_t i = 0; i < 100; i++) {
340     int v = dist(prng);
341     pA[i] = v == 0 ? 1 : v;
342   }
343   pA[17] = 1;
344
345   // lambda function for comparing elements for the search
346   auto cmp = [](const int &a, const int &b) { return a - b; };
347
348   // search for 1 should work
349   int idx = SearchNSort<int>::linearSearch(pA, 100, 1, cmp);
350   CHECK(-1 != idx);
351   // search for 0 should not work
352   idx = SearchNSort<int>::linearSearch(pA, 100, 0, cmp);
353   CHECK(-1 == idx);
354 }
355
356 /*
357  * Implementation of private merge() function.
358  */
359 template <class T>
360 void SearchNSort<T>::merge(T *pA, T *pB, size_t left, size_t right, size_t mid,
361                            int (*comp)(const T &x, const T &y)) {
362
363   size_t i = left, j = mid;
364
365   for (size_t k = left; k < right; k++) {
366     if (i < mid && (j >= right || comp(pA[i], pA[j]) <= 0)) {
367       pB[k] = pA[i++];
368     } else {
369       pB[k] = pA[j++];
370     }
371   }
372 }
373
374 /*
375  * Implementation of public mergeSort() function.
376  */
377 template <class T>
378 void SearchNSort<T>::mergeSort(T *pArr, size_t n,
```

```
                                int (*comp)(const T &x, const T &y)) {
  // create temporary array
  T *pB = new T[n];

  // do the sorting
  mergeSort(pArr, pB, 0, n, comp);

  // free temporary array
  delete[] pB;
}

// doctest unit test for mergeSort
TEST_CASE("testing SearchNSort::mergeSort") {
  // make two arrays for sorting
  int pA[100], pB[100];
  std::mt19937_64 prng(time(0));
  std::uniform_int_distribution<int> dist(-100, 100);
  for(size_t i = 0; i < 100; i++) {
    pA[i] = pB[i] = dist(prng);
  }

  // lambda function for comparing elements for the sort
  auto cmp = [](const int &a, const int &b) { return a - b; };

  // sort using mergeSort
  SearchNSort<int>::mergeSort(pA, 100, cmp);

  // sort using std::sort
  std::sort(pB, pB + 100);

  // check elements are the same
  for(size_t i = 0; i < 100; i++) {
    CHECK(pA[i] == pB[i]);
  }
}

/*
 * Implementation of private mergeSort() helper function.
 */
template <class T>
void SearchNSort<T>::mergeSort(T *pA, T *pB, size_t left, size_t right,
                               int (*comp)(const T &x, const T &y)) {

  // array of size one or less is already sorted!
  if ((right - left) < 2) {
    return;
  }

  // otherwise, split, sort, and merge
  size_t mid = (left + right) / 2;
  mergeSort(pA, pB, left, mid, comp);
  mergeSort(pA, pB, mid, right, comp);
  merge(pA, pB, left, right, mid, comp);

  // copy from scratch back to original array
  for (size_t i = left; i < right; i++) {
    pA[i] = pB[i];
  }
}

/*
 * Implementation of partition() helper function.
 */
template <class T>
size_t SearchNSort<T>::partition(T *pArr, size_t lo, size_t hi,
                                 int (*compare)(const T &x, const T &y)) {

  // arbitrarily choose first value in range as pivot value
  const T &pivot = pArr[lo];

  // indices to slide right and left
  size_t i = lo - 1;
  size_t j = hi + 1;

  while (true) {
    // slide i right until we find value >= pivot
```

```
455       while (compare(pArr[++i], pivot) < 0)
456         ; // empty loop body
457
458       // slide j left until we find value <= pivot
459       while (compare(pArr[--j], pivot) > 0)
460         ; // empty loop body
461
462       // if the indices have crossed, j is the pivot index
463       if (i >= j) {
464         return j;
465       }
466
467       // if not, swap the out of place elements and continue
468       // sliding i and j
469       std::swap(pArr[i], pArr[j]);
470     }
471   }
472
473   /*
474    * Implementation of recursive quickSort() helper function.
475    */
476   template <class T>
477   void SearchNSort<T>::quickSort(T *pArr, size_t lo, size_t hi,
478                                  int (*compare)(const T &x, const T &y)) {
479
480     // portion of size 0 or 1 is already sorted!
481     if (lo < hi) {
482
483       // array portion of array so pArr[lo, p - 1] is <= pArr[p],
484       // and pArr[p + 1, hi) is >= pArr[p]
485       size_t p = partition(pArr, lo, hi, compare);
486
487       // recursively sort left and right halves
488       quickSort(pArr, lo, p, compare);
489       quickSort(pArr, p + 1, hi, compare);
490     }
491   }
492
493   // doctest unit test for quickSort
494   TEST_CASE("testing␣SearchNSort::quickSort") {
495     // make two arrays for sorting
496     int pA[100], pB[100];
497     std::mt19937_64 prng(time(0));
498     std::uniform_int_distribution<int> dist(-100, 100);
499     for(size_t i = 0; i < 100; i++) {
500       pA[i] = pB[i] = dist(prng);
501     }
502
503     // lambda function for comparing elements for the sort
504     auto cmp = [](const int &a, const int &b) { return a - b; };
505
506     // sort using quickSort
507     SearchNSort<int>::quickSort(pA, 100, cmp);
508
509     // sort using std::sort
510     std::sort(pB, pB + 100);
511
512     // check elements are the same
513     for(size_t i = 0; i < 100; i++) {
514       CHECK(pA[i] == pB[i]);
515     }
516   }
517
518   /*
519    * Implementation of selectionSort() function.
520    */
521   template <class T>
522   void SearchNSort<T>::selectionSort(T *pArr, size_t n,
523                                      int (*comp)(const T &x, const T &y)) {
524     size_t i, j, minIndex;
525
526     for (i = 0u; i < n - 1u; i++) {
527
528       minIndex = i;
529       for (j = i + 1u; j < n; j++) {
530         if (comp(pArr[j], pArr[minIndex]) < 0) {
```

```
531           minIndex = j;
532         }
533       }
534
535       if (minIndex != i) {
536         std::swap(pArr[i], pArr[minIndex]);
537       }
538     }
539   }
540
541   // doctest unit test for selectionSort
542   TEST_CASE("testing␣SearchNSort::selectionSort") {
543     // make two arrays for sorting
544     int pA[100], pB[100];
545     std::mt19937_64 prng(time(0));
546     std::uniform_int_distribution<int> dist(-100, 100);
547     for(size_t i = 0; i < 100; i++) {
548       pA[i] = pB[i] = dist(prng);
549     }
550
551     // lambda function for comparing elements for the sort
552     auto cmp = [](const int &a, const int &b) { return a - b; };
553
554     // sort using selectionSort
555     SearchNSort<int>::selectionSort(pA, 100, cmp);
556
557     // sort using std::sort
558     std::sort(pB, pB + 100);
559
560     // check elements are the same
561     for(size_t i = 0; i < 100; i++) {
562       CHECK(pA[i] == pB[i]);
563     }
564   }
```

Listing 2: Scheduling.cpp

```
1   #include <cstdlib>
2   #include <fstream>
3   #include <iostream>
4   #include <string>
5   #include "../1-SearchNSort/SearchNSort.hpp"
6
7   /**
8    * @brief Class representing a task that takes two phases.
9    *
10   */
11  class Task {
12  public:
13      /**
14       * @brief Construct a new Task object with empty values.
15       *
16       */
17      Task() : name(""), elapsed1(0), elapsed2(0) { }
18
19      /**
20       * @brief Name of the task.
21       *
22       */
23      std::string name;
24      /**
25       * @brief Elapsed time for phase 1.
26       *
27       */
28      size_t elapsed1;
29      /**
30       * @brief Elapsed time for phase 2.
31       *
32       */
33      size_t elapsed2;
34
35      /**
36      * @brief Override of < for Task objects.
37      *
38      * @param t Task to compare with this one.
39      * @return If elapsed1 fields are equal, return elapsed2 < t.elapsed 2;
```

```
40      * otherwise, return elapsed1 < t.elapsed1;
41      */
42     bool operator<(const Task &t) {
43         if(elapsed1 == t.elapsed1) {
44             return elapsed2 < t.elapsed2;
45         } else {
46             return elapsed1 < t.elapsed1;
47         }
48     }
49 
50     /**
51      * @brief Override of > for Task objects.
52      *
53      * @param t Task to compare with this one.
54      * @return If elapsed1 fields are equal, return elapsed2 > t.elapsed 2;
55      * otherwise, return elapsed1 > t.elapsed1;
56      */
57     bool operator>(const Task &t) {
58         if(elapsed1 == t.elapsed1) {
59             return elapsed2 > t.elapsed2;
60         } else {
61             return elapsed1 > t.elapsed1;
62         }
63     }
64 
65     /**
66      * @brief Override of == for Task objects.
67      *
68      * @param t Task to compare with this one.
69      * @return If elapsed1 fields are equal, return elapsed2 == t.elapsed 2;
70      * otherwise, return elapsed1 == t.elapsed1;
71      */
72     bool operator==(const Task &t) {
73         if(elapsed1 == t.elapsed1) {
74             return elapsed2 == t.elapsed2;
75         } else {
76             return elapsed1 == t.elapsed1;
77         }
78     }
79 };
80 
81 /**
82  * @brief Stream insertion override for Task objects.
83  *
84  * @param out std::ostream to write to
85  * @param t Task object to write
86  * @return std::ostream& out for chaining
87  */
88 std::ostream& operator<<(std::ostream &out, const Task &t) {
89     out << t.name << ":␣(" << t.elapsed1 << ",␣" << t.elapsed2 << ")";
90     return out;
91 }
92 
93 /**
94  * @brief Comparison function for two Task objects.
95  *
96  * Compares based on elapsed time of second phase of the task.
97  *
98  * @param a First Task object
99  * @param b Second Task object
100  * @return int negative if a < b, 0 if a == b, positive if a > b
101  */
102 int compare(const Task &a, const Task &b) {
103     return a.elapsed2 - b.elapsed2;
104 }
105 
106 /**
107  * @brief Application entry point
108  */
109 int main() {
110 
111     // open data file an find out how many tasks there are
112     std::ifstream inFile("times.txt");
113     size_t n;
114     inFile >> n;
115 
```

```
116     // fill an array of Task objects with data from the file
117     Task tasks[n];
118     std::string name;
119     size_t s1, e1, s2, e2, idx = 0;
120     while(inFile >> name >> s1 >> e1 >> s2 >> e2) {
121         tasks[idx].name = name;
122         tasks[idx].elapsed1 = (e1 - s1);
123         tasks[idx].elapsed2 = (e2 - s2);
124         idx++;
125     }
126     inFile.close();
127 
128     // print initial list of tasks
129     std::cout << "There␣are␣" << n << "␣jobs␣to␣complete:" << std::endl;
130     for(idx = 0; idx < n; idx++) {
131         std::cout << tasks[idx] << std::endl;
132     }
133 
134     // sort by phase 2 times
135     SearchNSort<Task>::quickSort(tasks, n, compare);
136 
137     // print out suggested order of tasks
138     std::cout << "\nTasks␣ordered␣by␣phase␣2␣time:␣" << std::endl;
139     for(idx = 0; idx < n; idx++) {
140         std::cout << tasks[idx] << std::endl;
141     }
142 
143     return EXIT_SUCCESS;
144 }
```