```
/*
Author: Don Spickler
Creation Date: 1/15/2023
Last Update: 1/25/2024
Description: Set of standard sorting routines.
Notes: Algorithms and code were taken from either
Data Structures and Algorithms in C++ by Adam Drozdek or
Introduction to Algorithms Fourth Edition by Cormen, Leiserson, Rivest, and
Stein
*/

#ifndef SORTS_H_
#define SORTS_H_

#include <algorithm>
#include <deque>
#include <iostream>
#include <vector>

using namespace std;

template <class T> bool sorted(T A[], int size);

template <class T> void bubble(T A[], int size);
template <class T> void insertion(T A[], int size);
template <class T> void selection(T A[], int size);
template <class T> void merge(T A[], T Temp[], int startA, int startB, int end
    );
template <class T> void mergeSort(T A[], T Temp[], int start, int end);
template <class T> void mergeSort(T A[], int size);
template <class T> void quickSort(T A[], int left, int right);
template <class T> void quickSort(T A[], int size);
template <class T> void combsort(T data[], const int n);
template <class T> void Shellsort(T data[], int n);
template <class T> void moveDown(T data[], int first, int last);
template <class T> void heapsort(T data[], const int n);
// T needs to be an integer type for radix and count.
template <class T> void radixsort(T data[], const int n, const int radix);
template <class T> void countsort(T *A, long sz);
// T needs to be a float type for bucket.
template <class T> void BucketSort(T *A, long sz, long buckets);

/*
Description: Determines if the array is sorted.
Parameters: Array A and size of the array.
Return: Boolean of the array being sorted or not.
*/
template <class T> bool sorted(T A[], int size) {
  for (int i = 0; i < size - 1; i++)
    if (A[i] > A[i + 1])
      return false;
```

```
51
52    return true;
53  }
54
55  /*
56  Description: Sorts the array using the standard bubble sort.
57  Parameters: Array A and size of the array.
58  Return: None
59  */
60  template <class T> void bubble(T A[], int size) {
61    for (int i = 0; i < size - 1; i++) {
62      for (int j = 0; j < size - i - 1; j++) {
63        if (A[j] > A[j + 1]) {
64          T temp = A[j];
65          A[j] = A[j + 1];
66          A[j + 1] = temp;
67        }
68      }
69    }
70  }
71
72  /*
73  Description: Sorts the array using the standard insertion sort.
74  Parameters: Array A and size of the array.
75  Return: None
76  */
77  template <class T> void insertion(T A[], int size) {
78    for (int i = 0; i < size; i++) {
79      int j = 0;
80      T val = A[i];
81      for (j = i; j > 0 && A[j - 1] > val; j--)
82        A[j] = A[j - 1];
83
84      A[j] = val;
85    }
86  }
87
88  /*
89  Description: Sorts the array using the standard selection sort.
90  Parameters: Array A and size of the array.
91  Return: None
92  */
93  template <class T> void selection(T A[], int size) {
94    int minindex;
95
96    for (int i = 0; i < size; i++) {
97      minindex = i;
98      for (int j = i; j < size; j++)
99        if (A[j] < A[minindex])
100         minindex = j;
101
```

```
102     T val = A[i];
103     A[i] = A[minindex];
104     A[minindex] = val;
105   }
106 }
107
108 ////////////////////////////////////////////////////////
109 //  Merge Sort
110 ////////////////////////////////////////////////////////
111
112 /*
113 Description: Merging routine to merge two sorted subarrays into a sorted
114 array.
115 Parameters: Array A, temp array, starting positions for sections and
116 the end position of the second section.
117 Return: None
118 */
119 template <class T>
120 void merge(T A[], T Temp[], int startA, int startB, int end) {
121   int aptr = startA;
122   int bptr = startB;
123   int i = startA;
124
125   while (aptr < startB && bptr <= end)
126     if (A[aptr] < A[bptr])
127       Temp[i++] = A[aptr++];
128     else
129       Temp[i++] = A[bptr++];
130
131   while (aptr < startB)
132     Temp[i++] = A[aptr++];
133
134   while (bptr <= end)
135     Temp[i++] = A[bptr++];
136
137   for (i = startA; i <= end; i++)
138     A[i] = Temp[i];
139 }
140
141 /*
142 Description: Recursive subdividing portion of the merge sort algorithm.
143 Parameters: Array A, temp array, starting and ending positions of the
144 portion being subdivided.
145 Return: None
146 */
147 template <class T> void mergeSort(T A[], T Temp[], int start, int end) {
148   if (start < end) {
149     int mid = (start + end) / 2;
150     mergeSort(A, Temp, start, mid);
151     mergeSort(A, Temp, mid + 1, end);
152     merge(A, Temp, start, mid + 1, end);
```

```
153   }
154 }
155
156 /*
157 Description: Non-recursive starting function for the merge sort.
158 Parameters: Array A and size of the array.
159 Return: None
160 */
161 template <class T> void mergeSort(T A[], int size) {
162   T *Temp = new T[size];
163   mergeSort(A, Temp, 0, size - 1);
164   delete[] Temp;
165 }
166
167 ////////////////////////////////////////////////////////////
168 //  Quick Sort
169 ////////////////////////////////////////////////////////////
170
171 /*
172 Description: Recursive subdividing portion of the quick sort algorithm.
173 Parameters: Array A, starting and ending positions of the portion being
174 subdivided.
175 Return: None
176 */
177 template <class T> void quickSort(T A[], int left, int right) {
178   int i = left;
179   int j = right;
180   int mid = (left + right) / 2;
181
182   T pivot = A[mid];
183
184   while (i <= j) {
185     while (A[i] < pivot)
186       i++;
187
188     while (A[j] > pivot)
189       j--;
190
191     if (i <= j) {
192       T tmp = A[i];
193       A[i] = A[j];
194       A[j] = tmp;
195       i++;
196       j--;
197     }
198   }
199
200   if (left < j)
201     quickSort(A, left, j);
202
203   if (i < right)
```

```
204     quickSort(A, i, right);
205 }
206
207 /*
208 Description: Non-recursive starting function for the quick sort.
209 Parameters: Array A and size of the array.
210 Return: None
211 */
212 template <class T> void quickSort(T A[], int size) {
213   quickSort(A, 0, size - 1);
214 }
215
216 ////////////////////////////////////////////////////////
217 //  Comb Sort
218 ////////////////////////////////////////////////////////
219
220 /*
221 Description: Sorts the array using the standard comb sort.
222 Parameters: Array A and size of the array.
223 Return: None
224 */
225 template <class T> void combsort(T data[], const int n) {
226   int step = n, j, k;
227   // phase 1
228   while ((step = int(step / 1.3)) > 1)
229     for (j = n - 1; j >= step; j--) {
230       k = j - step;
231       if (data[j] < data[k])
232         swap(data[j], data[k]);
233     }
234
235   // phase 2
236   bool again = true;
237   for (int i = 0; i < n - 1 && again; i++)
238     for (j = n - 1, again = false; j > i; --j)
239       if (data[j] < data[j - 1]) {
240         swap(data[j], data[j - 1]);
241         again = true;
242       }
243 }
244
245 ////////////////////////////////////////////////////////
246 //  Shell Sort
247 ////////////////////////////////////////////////////////
248
249 /*
250 Description: Sorts the array using the standard Shell sort.
251 Parameters: Array A and size of the array.
252 Return: None
253 */
254 template <class T> void Shellsort(T data[], int n) {
```

```
255   int i, j, hCnt, h;
256   int increments[20], k;
257   // create an appropriate number of increments h
258   for (h = 1, i = 0; h < n; i++) {
259     increments[i] = h;
260     h = 3 * h + 1;
261   }
262   // loop on the number of different increments h
263   for (i--; i >= 0; i--) {
264     h = increments[i];
265     // loop on the number of subarrays h-sorted in ith pass
266     for (hCnt = h; hCnt < 2 * h; hCnt++) {
267       // insertion sort for subarray containing every hth element of
268       for (j = hCnt; j < n;) { // array data
269         T tmp = data[j];
270         k = j;
271         while (k - h >= 0 && tmp < data[k - h]) {
272           data[k] = data[k - h];
273           k -= h;
274         }
275         data[k] = tmp;
276         j += h;
277       }
278     }
279   }
280 }
281
282 ////////////////////////////////////////////////////////
283 //  Heap Sort
284 ////////////////////////////////////////////////////////
285
286 /*
287 Description: Restores the heap property on the array segment.
288 Parameters: Array A and the first and last index to restore.
289 Return: None
290 */
291 template <class T> void moveDown(T data[], int first, int last) {
292   int largest = 2 * first + 1;
293   while (largest <= last) {
294     if (largest < last && // first has two children (at 2*first+1 and
295         data[largest] < data[largest + 1]) // 2*first+2) and the second
296       largest++;                           // is larger than the first;
297
298     if (data[first] < data[largest]) {  // if necessary,
299       swap(data[first], data[largest]); // swap child and parent,
300       first = largest;                  // and move down;
301       largest = 2 * first + 1;
302     } else
303       largest = last + 1; // to exit the loop: the heap property
304   }                       // isn't violated by data[first];
305 }
```

```
306
307 /*
308 Description: Sorts the array using the standard heap sort.
309 Parameters: Array A and size of the array.
310 Return: None
311 */
312 template <class T> void heapsort(T data[], const int n) {
313   int i;
314   for (i = n / 2 - 1; i >= 0; --i) // create the heap;
315     moveDown(data, i, n - 1);
316   for (i = n - 1; i >= 1; --i) {
317     swap(data[0], data[i]);   // move the largest item to data[i];
318     moveDown(data, 0, i - 1); // restore the heap property;
319   }
320 }
321
322 ////////////////////////////////////////////////////////
323 //  Radix Sort: implementation for integer data.
324 ////////////////////////////////////////////////////////
325
326 /*
327 Description: Sorts the array using the standard radix sort.
328 Parameters: Array A, size of the array, and radix to use.
329 Return: None
330 Notes: This is for integer data only.
331 */
332 template <class T> void radixsort(T data[], const int n, const int radix) {
333   T d, j, k, factor;
334   T max = data[0];
335   for (int i = 0; i < n; i++)
336     if (max < data[i])
337       max = data[i];
338
339   deque<T> queues[radix];
340   for (d = 0, factor = 1; max / factor > 0; factor *= radix, d++) {
341     for (j = 0; j < n; j++)
342       queues[(data[j] / factor) % radix].push_back(data[j]);
343     for (j = k = 0; j < radix; j++)
344       while (!queues[j].empty()) {
345         data[k++] = queues[j].front();
346         queues[j].pop_front();
347       }
348   }
349 }
350
351 ////////////////////////////////////////////////////////
352 //  Count Sort: implementation for positive integer data.
353 ////////////////////////////////////////////////////////
354
355 /*
356 Description: Sorts the array using the standard count sort.
```

```
Parameters: Array A and the size of the array.
Return: None
Notes: This is for positive integer data only.
*/
template <class T> void countsort(T *A, long sz) {
  T maxval = A[0];
  for (int i = 0; i < sz; i++)
    maxval = max(maxval, A[i]);

  int *counts = new int[maxval + 1]();
  int *temp = new int[sz];

  for (int i = 0; i < sz; i++)
    counts[A[i]]++;

  for (int i = 1; i < maxval + 1; i++)
    counts[i] += counts[i - 1];

  for (int i = 0; i < sz; i++)
    temp[--counts[A[i]]] = A[i];

  copy(temp, temp + sz, A);
  delete[] temp;
  delete[] counts;
}

////////////////////////////////////////////////////////////
//  Bucket Sort: implementation for floating point data in [0, 1).
////////////////////////////////////////////////////////////

/*
Description: Sorts the array using the standard bucket sort.
Parameters: Array A and the size of the array.
Return: None
Notes: This is for floating point data only in the range [0, 1).
*/
template <class T> void BucketSort(T *A, long sz) {
  //    deque<T> *Buckets = new deque<T> [sz];
  vector<T> *Buckets = new vector<T>[sz];

  for (long i = 0; i < sz; i++)
    Buckets[static_cast<long>(sz * A[i])].push_back(A[i]);

  //    for (long i = 0; i < sz; i++)
  //        sort(Buckets[i].begin(), Buckets[i].end());

  for (long i = 0; i < sz; i++)
    insertion(Buckets[i].data(), Buckets[i].size());

  int pos = 0;
  for (long i = 0; i < sz; i++)
```

```
408       for (unsigned long j = 0; j < Buckets[i].size(); j++)
409         A[pos++] = Buckets[i][j];
410 
411     delete[] Buckets;
412 }
413 
414 #endif /* SORTS_H_ */
```