```
 1  #ifndef SORT_H
 2  #define SORT_H
 3
 4  #define NULL 0
 5  #define TRUE 1
 6  #define FALSE 0
 7  #define MAXNUMS 500
 8  #define BASE 10
 9  #define MAXDIGITS 5
10  #define MAXSIZE 256
11
12  /* ==============UTILITIES==============*/
13
14  void swap(a, b)
15  int *a;
16  int *b;
17  {
18    int temp;
19    temp = *a;
20    *a = *b;
21    *b = temp;
22  }
23
24  /* =============BUBBLE SORT=============*/
25
26  bubble_sort(arr, n)
27  int arr[];
28  int n;
29  {
30    int i, j;
31    for (i = 0; i < n - 1; i++) {
32      for (j = 0; j < n - i - 1; j++) {
33        if (arr[j] > arr[j+1]) {
34          swap(&arr[j], &arr[j+1]);
35        }
36      }
37    }
38  }
39
40  /* ===========INSERTION SORT============*/
41
42  insertion_sort(arr, n)
43  int arr[];
44  int n;
45  {
46    int i, j, temp;
47    for (i = 1; i < n; i++) {
48      temp = arr[i];
49      j = i - 1;
50      while((j >= 0) && (temp < arr[j])) {
51        arr[j+1] = arr[j--];
52      }
53      arr[j+1] = temp;
54    }
55  }
56
57  /* ===========SELECTION SORT============*/
58
59  int smallest(arr, k, n)
60  int arr[];
61  int k, n;
62  {
63    int pos, small, i;
64    pos = k;
65    small = arr[k];
66    for (i = k + 1; i < n; i++) {
67      if (arr[i] < small) {
68        small = arr[i];
69        pos = i;
70      }
71    }
72    return pos;
73  }
```

```
 74
 75  selection_sort(arr, n)
 76  int arr[];
 77  int n;
 78  {
 79    int k, pos;
 80    for (k = 0; k < n; k++) {
 81      pos = smallest(arr, k, n);
 82      swap(&arr[k], &arr[pos]);
 83    }
 84  }
 85
 86  /* ==============MERGE SORT==============*/
 87
 88  merge(arr, b, m, e)
 89  int arr[];
 90  int b, m, e;
 91  {
 92    int i, j, k, s;
 93    int temp[MAXSIZE];
 94
 95    i = b;
 96    j = m + 1;
 97    k = 0;
 98    s = e - b;
 99
100    while (i <= m && j <= e)
101      temp[k++] = (arr[i] < arr[j]) ? arr[i++] : arr[j++];
102    if (i > m)
103      while (j <= e)
104        temp[k++] = arr[j++];
105    else
106      while (i <= m)
107        temp[k++] = arr[i++];
108    for (k = 0; k <= s; k++)
109      arr[k+b] = temp[k];
110  }
111
112  msort(arr, left, right)
113  int arr[];
114  int left, right;
115  {
116    int mid;
117    if (left < right)
118    {
119      mid = left + (right - left) / 2;
120      msort(arr, left, mid);
121      msort(arr, mid+1, right);
122      merge(arr, left, mid, right);
123    }
124  }
125
126  merge_sort(arr, n)
127  int arr[];
128  int n;
129  {
130    if (n <= 0) return;
131    msort(arr, 0, n-1);
132  }
133
134  /* ==============QUICK SORT==============*/
135
136  int partition(arr, b, e)
137  int arr[];
138  int b, e;
139  {
140    int l, r, temp, loc, flag;
141    loc = l = b;
142    r = e;
143    flag = FALSE;
144
145    while (flag != TRUE) {
146      while ((arr[loc] <= arr[r]) && (loc != r)) r--;
```

```
147       if (loc == r)
148         flag = TRUE;
149       else
150         if (arr[loc] > arr[r]) {
151           swap(&arr[loc], &arr[r]);
152           loc = r;
153         }
154       if (flag != TRUE) {
155         while ((arr[loc] >= arr[l]) && (loc != l)) l++;
156         if (loc == l)
157           flag = TRUE;
158         else
159           if (arr[loc] < arr[l]) {
160             swap(&arr[loc], &arr[l]);
161             loc = l;
162           }
163       }
164     }
165     return loc;
166   }
167
168   qsort(arr, b, e)
169   int arr[];
170   int b, e;
171   {
172     int loc;
173     if (b < e) {
174       loc = partition(arr, b, e);
175       qsort(arr, b, loc-1);
176       qsort(arr, loc+1, e);
177     }
178   }
179
180   void quick_sort(arr, n)
181   int arr[];
182   int n;
183   {
184     qsort(arr, 0, n-1);
185   }
186
187   /* ==============RADIX SORT==============*/
188
189   counting_sort(arr, tmp, cnt, n, pos)
190   int arr[];
191   int tmp[];
192   int cnt[];
193   int n;
194   int pos;
195   {
196     int i;
197     int d;
198     int base;
199     base = 1;
200     for (i = 0; i < pos; i++)
201       base *= BASE;
202     for (i = 0; i < BASE; i++)
203       cnt[i] = 0;
204     for (i = 0; i < n; i++) {
205       d = (arr[i] / base) % BASE;
206       cnt[d]++;
207     }
208     for (i = 1; i < BASE; i++)
209       cnt[i] += cnt[i - 1];
210     for (i = n - 1; i >= 0; i--) {
211       d = (arr[i] / base) % BASE;
212       cnt[d]--;
213       tmp[cnt[d]] = arr[i];
214     }
215     for (i = 0; i < n; i++)
216       arr[i] = tmp[i];
217   }
218
219   radix_sort(arr, n)
```

```
220  int arr[];
221  int n;
222  {
223    int tmp[MAXNUMS];
224    int cnt[BASE];
225    int pass;
226    for (pass = 0; pass < MAXDIGITS; pass++)
227      counting_sort(arr, tmp, cnt, n, pass);
228  }
229
230  /* ==============HEAP SORT==============*/
231
232  reheap_up(heap, n)
233  int heap[];
234  int n;
235  {
236    int i, val;
237    i = n - 1;
238    val = heap[i];
239    while ((i > 0) && (heap[i/2] < val)) {
240      heap[i] = heap[i/2];
241      i /= 2;
242    }
243    heap[i] = val;
244  }
245
246  reheap_down(heap, n)
247  int heap[];
248  int n;
249  {
250    int i, child, left, right;
251    i = 0;
252    while (TRUE) {
253      left = 2 * i + 1;
254      if (left >= n) return;              /* heap[i] is a leaf node */
255      right = left + 1;
256      child = left;
257      if (right < n && heap[right] > heap[left])
258        child = right;
259      if (heap[i] >= heap[child]) return; /* It's now a heap */
260      swap(&heap[i], &heap[child]);
261      i = child;
262    }
263  }
264
265  heapify(arr, n)
266  int arr[];
267  int n;
268  {
269    int i;
270    for (i = 1; i <= n; i++) {
271      reheap_up(arr, i);
272    }
273  }
274
275  heap_sort(arr, n)
276  int arr[];
277  int n;
278  {
279    int e, i;
280    heapify(arr, n);
281    for (e = n - 1; e > 1; e--) {
282      swap(&arr[0], &arr[e]);
283      reheap_down(arr, e-1);
284    }
285  }
286
287  /* ==============TREE SORT==============*/
288
289  struct tree {
290    struct tree *left;
291    int num;
292    struct tree *right;
```

```
293  };
294
295  insert(tpp, n)
296  struct tree **tpp;
297  int n;
298  {
299    struct tree *tree_node;
300    tree_node = *tpp;
301    if (tree_node == NULL) {
302      tree_node = alloc(8);
303      tree_node->num = n;
304      tree_node->left  = NULL;
305      tree_node->right = NULL;
306      *tpp = tree_node;
307      return 0;
308    }
309    if (n < tree_node->num)
310      insert(&(tree_node->left),  n);
311    else
312      insert(&(tree_node->right), n);
313    return 0;
314  }
315
316  inorder(tree_node, arr, ip)
317  struct tree *tree_node;
318  int arr[];
319  int *ip;
320  {
321    if (tree_node == NULL)
322      return 0;
323    inorder(tree_node->left,  arr, ip);
324    arr[(*ip)++] = tree_node->num;
325    inorder(tree_node->right, arr, ip);
326    return 0;
327  }
328
329  free_tree(tree_node)
330  struct tree *tree_node;
331  {
332    if (tree_node == NULL)
333      return 0;
334    free_tree(tree_node->left);
335    free_tree(tree_node->right);
336    free(tree_node);
337    return 0;
338  }
339
340  tree_sort(arr, n)
341  int arr[];
342  int n;
343  {
344    int i, index;
345    struct tree *root;
346
347    root = NULL;
348    for (i = 0; i < n; i++)
349      insert(&root, arr[i]);
350
351    index = 0;
352    inorder(root, arr, &index);
353    free_tree(root);
354  }
355
356  #endif //SORT_H
```