```cpp
//Name: Ghazi Najeeb Al-Abbar
//ID: 2181148914
//Heap.cpp

//Suppose that the size of the heap is n
#include <iostream>
#include <cmath>
using namespace std;

//Swap function for heapify
void Swap(int& x, int& y){

    int temp = x;
    x = y;
    y = temp;
}

//Power function for finding the minimum
int power(int base, int pwr){

    int result = 1;

    for(int i = 0; i < pwr; i++)
        result *= base;

    return result;
}

//Heap Data structure
class Heap{

public:

    //Heap Constructor
    Heap()
    {
        //Sets all the heap array elements to zero
        for (int i = 0; i < 100; i++)
            heapArr[i] = 0;

        //Heap size counter. Starts at -1, and it is incremented each time a new element is a
        Last_inserted = -1;
    }

    //Inserts elements into the heap
    //Time Complexity: O(log(n)) because the heap is traversed by jumping in powers of 2
    void Insert(int data){

        //The case where the heap has reached its capacity
        if (Last_inserted == 99){

            cout << "The heap has reached its capacity!\n";
            return;
        }

```

```
56:         //Checks if the Last_inserted is -1 (meaning the heap is empty)
57:         if (Last_inserted == -1){
58:
59:             Last_inserted++; //increments Last_inserted by 1
60:             heapArr[Last_inserted] = data; //Sets the first element of the heap as the user i
61:
62:             cout << data <<" Has been inserted to the heap!\n";
63:
64:             return; //Leaves function
65:         }
66:
67:         //The case where the heap has not reached its capacity, and is not empty
68:
69:         Last_inserted++; //Last_inserted is incremented by 1
70:         heapArr[Last_inserted] = data; //Sets the first element of the heap as the user input
71:         cout << data <<" Has been inserted to the heap!\n";
72:
73:         int i = Last_inserted; //i is the size of the heap, and it will be used to traverse t
74:
75:         //Heapifies the heap after inserting data.
76:         //Checks if the newly inserted element is larger than its parent. If it is, they swap
77:         while (i >= 0){
78:
79:             //checks if the newly inserted element is larger than its parent
80:             if (heapArr[i] > heapArr[i/2]){
81:
82:                 //If that's the case, then they swap
83:                 Swap(heapArr[i], heapArr[i/2]);
84:                 i /= 2; //i is halved
85:             }
86:
87:             //if the newly inserted element is not larger than its parent, then the program l
88:             else
89:                 break;
90:         }
91:     }
92:
93:     //Removes an the largest element from the heap
94:     //Time Complexity: O(log(n)) because the heap is traversed by jumping in powers of 2
95:     int Delete(){
96:
97:         //Checks if the heap is empty
98:         if (Last_inserted == -1){
99:
100:            cout << "The heap is empty!\n";
101:            return 0;
102:        }
103:
104:        //Checks if there is only one element in the heap
105:        if (Last_inserted == 0){
106:
107:            Last_inserted--; //decrement Last_inserted by 1
108:            cout << heapArr[0] << " Has been removed.\n";
109:            return heapArr[0]; //returns the removed element
110:        }
```

```
111:
112:         cout << heapArr[0] << " Has been removed.\n";
113:
114:         int i = 0; //index i starts at 0
115:         int Removed = heapArr[0]; //variable to store the removed element
116:
117:         //////////////Process of heapify//////////////
118:         //Determines whether the first child is greater than the 2nd child
119:         //The case where the first child is greater than the second child
120:         if (heapArr[1] > heapArr[2]){
121:
122:             heapArr[i] = heapArr[1]; //The root of the heap is now equal to the first child
123:             i = 1; //index is changed to the first child
124:         }
125:         //The case where the first child is less than the second child
126:         else{
127:
128:             heapArr[i] = heapArr[2]; //The root of the heap is now equal to the second child
129:             i = 2; //index is changed to the second child
130:         }
131:
132:         //Heapifies the heap past the first or second child
133:         while (i <= Last_inserted){
134:
135:             //Determines whether the first child is greater than the 2nd child
136:             //The case where the first child is less than the second child
137:             if (heapArr[2*i] < heapArr[2*i + 1]){
138:
139:     for(int i = 0; i < pwr; i++)pArr[2*i + 1];
140:                 i = 2*i + 1; //index is changed to the second child
141:             }
142:
143:             //The case where the first child is greater than the second child
144:             else{
145:
146:                 heapArr[i] = heapArr[2*i]; //The root of the heap is now equal to the first c
147:                 i = 2*i; //index is changed to the first child
148:             }
149:         }
150:
151:         Last_inserted--; //Decrements the heap size after removing the the root and heapifyin
152:
153:         return Removed; //Returns the removed element
154:     }
155:
156:     //Displays the heap in array form
157:     //Time Complexity: O(n) because the heap size is variable
158:     void display(){
159:
160:         //Checks if the heap is empty
161:         if (Last_inserted == -1){
162:
163:             cout << "The heap is empty!\n";
164:             return;
165:         }
```

```
166:
167:         //Goes through the heap and prints its contents
168:         for (int i = 0; i <= Last_inserted; i++)
169:             cout << heapArr[i] << " "; //Prints the heap element
170:
171:         cout << endl;
172:     }
173:
174:     //Finds the minimum value in the heap and returns it
175:     //Time Complexity: O(log(n)) because n is variable
176:     int getMin(){
177:
178:         int Log_size = (int)log(Last_inserted); //Log_size takes only the integer part of log
179:         int Min = heapArr[Last_inserted]; //Min is initialised
180:
181:         //Goes through the lower parts of the heap and finds the minimum value
182:         for (int i = Last_inserted; i >= Last_inserted - power(2,Log_size); i--)
183:
184:             //Checks if the heap element of index i is less than Min.
185:             if (heapArr[i] < Min)
186:                 Min = heapArr[i];
187:
188:         return Min; //Returns Min
189:     }
190:
191:     //Returns the Maximum element of the heap
192:     //Time Complexity: O(1) since there is only one instruction
193:     int getMax(){
194:
195:         //Returns the root of the heap
196:         return heapArr[0];
197:     }
198:
199:     //Returns the size of the heap
200:     //Time Complexity: O(1) Since there is one instruction
201:     int Size(){
202:
203:         //Returns Last_inserted + 1 since the first heap index is 1
204:         return Last_inserted + 1;
205:     }
206:
207:     //Checks if the heap is empty
208:     //Time Complexity: O(1) Since there is one instruction
209:     bool isEmpty(){
210:
211:         //Checks if Last_inserted is -1. If it is, then it returns True. Otherwise, returns F
212:         return Last_inserted == -1;
213:     }
214:
215: private: //Encapsulated members
216:
217:     int heapArr[100]; //Main heap array
218:     int Last_inserted; //Counter to keep track of the size of the heap
219: };
220:
```

```cpp
//Beginning of main
int main(){

    Heap h;

    cout << "Is the heap empty? The answer is: ";
    if (h.isEmpty())
        cout << "True!\n";
    else
        cout << "False!\n";

    h.Insert(5);
    h.Insert(33);
    h.Insert(100000);
    h.Insert(12);
    h.Insert(106);
    h.Insert(394);
    h.Insert(507);

    h.display();

    h.Delete();

    h.display();

    h.Insert(942);

    h.display();

    cout << "The largest element is: " << h.getMax() << " and the smallest element is: " << h
    cout << "The size is: " << h.Size() << endl;

    cout << "Is the heap empty? The answer is: ";
    if (h.isEmpty())
        cout << "True!\n";
    else
        cout << "False!\n";

    return 0;
}
//End of main
```

```cpp
//Name: Ghazi Najeeb Al-Abbar
//ID: 2181148914
//Sorting_Algorithms.cpp

//Assume the size of all arrays are n
//Quick sort and merge sort don't work on my pc's compiler, but they work just fine o
#include <iostream>
using namespace std;

//Swap function
void Swap(int& x, int& y){

int temp = x;
x = y;
y = temp;
}

/////////Bubble Sort/////////////////

//Time Complexity: O(n^2) because the size of the array is variable, also there is a
void bubbleSort(int arr[], int size){

    //Quick test to check if the array is already sorted. It does not affect the overall

    //Boolean flag to check if the array is sorted
    bool isSorted = false;

    //Goes through the array and checks if it is sorted
    for (int i = 0; i < size - 1; i++){

        //The case where the element of index i is greater than the element of index i + 1
        if (arr[i] > arr[i + 1]){

            isSorted = false; //Boolean flag is set to false
            break; //Breaks the loop
        }

        //The case where the element of index i is less than the element of index i + 1
        else
            isSorted = true; //Boolean flag is true
    }

    //If the flag is true, then the array is sorted and the function ends with time compl
    //If not, then the array will continue to be sorted

    if (isSorted){

        cout << "Your array is already sorted!\n";

        return;
    }

    //The case where the array is not sorted
    //Goes through the array and sorts its elements
    for (int i = 0; i < size - 1; i++){
```

```
56:
57:         for (int j = 0; j < size - (i + 1); j++){
58:
59:             //Checks if array element of index j is greater than the element of index j + 1
60:             if (arr[j] > arr[j + 1]){
61:
62:                 int temp = arr[j]; //temporary variable holds element of index j
63:
64:                 arr[j] = arr[j + 1]; //assign element of index j + 1 to the element of index
65:
66:                 arr[j + 1] = temp; //assigns the temporary variable to the element of index j
67:             }
68:         }
69:     }
70: }
71: ////////////////////////////////////////////////////////////
72:
73:
74: ///////////////Insert Sort//////////////////////////////////
75:
76: //Time Complexity: O(n^2) because the size of the array is variable, also there is a
77: void insertSort(int arr[], int size){
78:
79:     //Goes through the array and sorts its elements
80:     for (int i = 1; i < size; i++){
81:
82:         int key = arr[i];
83:         int j = i - 1;
84:
85:         while (j >= 0 && arr[j] > key){
86:
87:             arr[j + 1] = arr[j];
88:             j--;
89:         }
90:
91:         arr[j + 1] = key;
92:     }
93: }
94: ////////////////////////////////////////////////////////////
95:
96: /////////////////Merge Sort/////////////////////////////////
97:
98: //Takes two arrays and merges them into one array in sorted order
99: //Time Complexity: O(n) Since the size of the array is variable
100: void Merge(int arr[], int Start, int Middle, int End){
101:
102:     //Declare two new arrays
103:     int *subArrOne = new int[Middle - Start + 1], *subArrTwo = new int[End - Middle];
104:
105:     //Assigns the first array with the elements from the original array from start to mid
106:     for (int i = 0; i < Middle - Start + 1; i++)
107:         subArrOne[i] = arr[Start + i];
108:
109:     //Assigns the second array with the elements from the original array from Middle to E
110:     for (int i = 0; i < End - Middle; i++)
```

```
111:         subArrTwo[i] = arr[Middle + 1 + i];
112:
113:     int i = 0, j = 0, Current_index = Start;
114:
115:     //Goes through the two arrays and merges them in the original array in sorted order
116:     //Stops if at least one of the arrays has reached its end
117:     while (i < Middle - Start + 1 && j < End - Middle){
118:
119:         //Checks if the index i of the first array is less than index j from the second array
120:         if (subArrOne[i] < subArrTwo[j]){
121:
122:             arr[Current_index] = subArrOne[i]; //assigns index i of the first array to curren
123:             i++; //i is incremented by 1
124:         }
125:
126:         //Checks if the index i of the first array is greater than or equal to index j from t
127:         else{
128:             arr[Current_index] = subArrTwo[j]; //assigns index j of the second array to curre
129:                j++; //j is incremented by 1
130:         }
131:
132:         Current_index++; //The current index is incremented by 1
133:     }
134:
135:     //If the second array has reached its end and the first array did not
136:     while(i < Middle - Start + 1){
137:
138:         arr[Current_index] = subArrOne[i]; //assigns index i of the first array to current in
139:         i++; //i is incremented by 1
140:         Current_index++; //The current index is incremented by 1
141:     }
142:
143:     //If the first array has reached its end and the second array did not
144:     while (j < End - Middle){
145:
146:         arr[Current_index] = subArrTwo[j]; //assigns index j of the second array to current i
147:         j++; //j is incremented by 1
148:         Current_index++; //The current index is incremented by 1
149:     }
150: }
151:
152: //Splits the arrays into two halves recursivley and merges all halves using the merge
153: //Time Complexity: O(nlog(n)) Since the recursive movement is dependent on halving th
154: void MergeSort(int arr[], int Start, int End){
155:
156:     //Stops if the Starting index is greater than or equal to the Stopping index
157:     if (Start >= End)
158:         return;
159:
160:     //Finds the midpoint of the two arrays
161:     int Middle = (Start + End)/2;
162:
163:     //Recursivley call itself to split the array and merge them
164:     MergeSort(arr, Start, Middle);
165:     MergeSort(arr, Middle + 1, End);
```

```
166:
167:     Merge(arr, Start, Middle, End);
168: }
169: ////////////////////////////////////////////////////////
170:
171: ////////////////////Quick Sort//////////////////////////////
172:
173: //Sorts an array from a starting point to an ending point
174: //Time Complexity: O(n) since the array size is variable
175: int Partition(int arr[], int Start, int End){
176:
177:     //i is the index of the array
178:     //j is the index of where the swapping happens, and it determines where the new pivot
179:     int i, j;
180:     i = Start; j = Start - 1;
181:
182:     //Goes through the sub-array and sorts its elements
183:     for (i; i < End; i++){
184:
185:            //Checks if current index of the array is less than the last element
186:            if (arr[i] < arr[End]){
187:
188:             j++; //increment j by 1
189:             Swap(arr[i], arr[j]); //swap element of index i with element of index j
190:         }
191:     }
192:
193:     //swap element of index j + 1 with the last element
194:     Swap(arr[j + 1], arr[End]);
195:
196:     return j + 1; //Return j + 1
197: }
198:
199: //Sorts an array recursivley by splitting it between and sorting after a pivot point
200: //Time Complexity: O(nLog(n)) Because n is variable
201: void QuickSort(int arr[], int Start, int End){
202:
203:     //Stops if the Starting index is greater than or equal to the Stopping index
204:     if (Start >= End)
205:         return;
206:
207:     //Gets the pivot point from the partition function
208:     int Pivot = Partition(arr, Start, End);
209:
210:     //Recursivley split the array using the pivot
211:     QuickSort(arr, Start, Pivot - 1);
212:     QuickSort(arr, Pivot + 1, End);
213: }
214: ////////////////////////////////////////////////////////
215:
216: //Display function
217: void Display(int arr[], int Size){
218:
219:     for (int i = 0; i < Size; i++)
220:         cout << arr[i] << " ";
```

```
221:
222:     cout << endl;
223: }
224:
225: //Beginning of main
226: int main(){
227:
228:     int arr1[5] = {6, 7, 4, 8, 1};
229:     int arr2[5] = {2, 9, 20, 1, 8};
230:     int arr3[5] = {5, 4, 3, 2, 1};
231:     int arr4[5] = {5, 4, 3, 2, 1};
232:
233:     cout << "arr1 before bubble sort: ";
234:     Display(arr1, 5);
235:
236:     cout << "arr1 after bubble sort: ";
237:     bubbleSort(arr1, 5);
238:     Display(arr1, 5);
239:
240:     cout << "\n\n";
241:
242:     cout << "arr2 before insert sort: ";
243:     Display(arr2, 5);
244:
245:     cout << "arr2 after insert sort: ";
246:     insertSort(arr2, 5);
247:     Display(arr2, 5);
248:
249:     cout << "\n\n";
250:
251:     cout << "arr3 before Merge sort: ";
252:     Display(arr3, 5);
253:
254:     cout << "arr3 after Merge sort: ";
255:     MergeSort(arr3,0,5);
256:     Display(arr3, 5);
257:
258:     cout << "\n\n";
259:
260:     cout << "arr4 before Quick sort: ";
261:     Display(arr4, 5);
262:
263:     cout << "arr4 after Quick sort: ";
264:     QuickSort(arr4,0,5);
265:     Display(arr4, 5);
266:
267:     return 0;
268: }
269: //End of Main
```

PI not submitted

-30

0/30

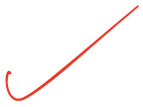
10/10

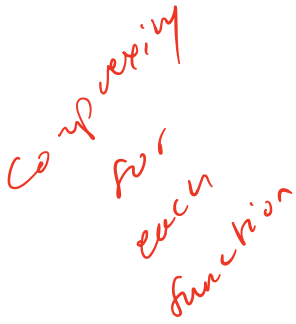
```cpp
//Name: Ghazi Najeeb Al-Abbar
//ID: 2181148914
//problem-2.cpp

//Assume n is the size of the array
#include <iostream>
using namespace std;

void Merge(int arr[], int Start, int Middle, int End){

    int *subArrOne = new int[Middle - Start + 1], *subArrTwo = new int[End - Middle];

    for (int i = 0; i < Middle - Start + 1; i++)
        subArrOne[i] = arr[Start + i];

    for (int i = 0; i < End - Middle; i++)
        subArrTwo[i] = arr[Middle + 1 + i];

    int i = 0, j = 0, Current_index = Start;
    while (i < Middle - Start + 1 && j < End - Middle){

        if (subArrOne[i] < subArrTwo[j]){

            arr[Current_index] = subArrOne[i];
            i++;
        }
        else{
            arr[Current_index] = subArrTwo[j];
               j++;
        }

        Current_index++;
    }

    while(i < Middle - Start + 1){

        arr[Current_index] = subArrOne[i];
        i++;
        Current_index++;
    }
    while (j < End - Middle){

        arr[Current_index] = subArrTwo[j];
        j++;
        Current_index++;
    }
}

void MergeSort(int arr[], int Start, int End){

    if (Start >= End)
        return;

    int Middle = (Start + End)/2;

```

```
56:     MergeSort(arr, Start, Middle);
57:     MergeSort(arr, Middle + 1, End);
58:     Merge(arr, Start, Middle, End);
59: }
60:
61: //Time Complexity: O(nlog(n)) Since merge sort is used
62: int KthSmallest(int arr[], int size, int k){
63:
64:     //Checks if k is larger than the array size
65:     if (k > size){
66:
67:         cout << k << "is larger than the array size!\n";
68:         return 0;
69:     }
70:
71:     //Declare a new array so the original array stays the way it is
72:     int *newArr = new int [size];
73:
74:     //Assign the original array to the old array
75:     for (int i = 0; i < size; i++)
76:         newArr[i] = arr[i];
77:
78:     MergeSort(newArr, 0, size); //Use merge sort on the new array
79:
80:     return newArr[k - 1]; //Return element k - 1 from the new array
81: }
82:
83: void Display(int arr[], int Size){
84:
85:     for (int i = 0; i < Size; i++)
86:         cout << arr[i] << " ";
87:
88:     cout << endl;
89: }
90:
91: //Beginning of main
92: int main(){
93:
94:     int arr[10] = {99, 104, 1000, 10, 303, 70, 195, 603, 817, 1023};
95:
96:     Display(arr, 10);
97:
98:     cout << "\nThe 3rd smallest element in the array is: " << KthSmallest(arr, 10, 7);
99:
100:    return 0;
101: }
102: //End of main
```

```cpp
//Name: Ghazi Najeeb Al-Abbar
//ID: 2181148914
//problem-3.cpp

//Let n be the size of the array
#include <iostream>
#include <cmath>
#include <stack>
using namespace std;

void Swap(int& x, int& y){

    int temp = x;
    x = y;
    y = temp;
}

int power(int base, int pwr){

    int result = 1;

    for(int i = 0; i < pwr; i++)
        result *= base;

    return result;
}

class Heap{

public:

    Heap()
    {
        for (int i = 0; i < 100; i++)
            heapArr[i] = 0;

        Last_inserted = -1;
    }

    void Insert(int data){

        if (Last_inserted == 99){

            cout << "The heap has reached its capacity!\n";
            return;
        }

        if (Last_inserted == -1){

            Last_inserted++;
            heapArr[Last_inserted] = data;

            //cout << data <<" Has been inserted to the heap!\n";

            return;
```

```
56:         }
57:
58:         Last_inserted++;
59:         heapArr[Last_inserted] = data;
60:         //cout << data <<" Has been inserted to the heap!\n";
61:
62:         int i = Last_inserted;
63:
64:         while (i >= 0){
65:
66:             if (heapArr[i] > heapArr[i/2]){
67:
68:                 Swap(heapArr[i], heapArr[i/2]);
69:                 i /= 2;
70:             }
71:
72:             else
73:                 break;
74:         }
75:     }
76:
77:     int Delete(){
78:
79:         if (Last_inserted == -1){
80:
81:             //cout << "The heap is empty!\n";
82:             return 0;
83:         }
84:
85:         if (Last_inserted == 0){
86:
87:             Last_inserted--;
88:             //cout << heapArr[0] << " Has been removed.\n";
89:             return heapArr[0];
90:         }
91:
92:         //cout << heapArr[0] << " Has been removed.\n";
93:
94:         int i = 0;
95:         int Removed_Data = heapArr[0];
96:
97:         if (heapArr[1] > heapArr[2]){
98:
99:             heapArr[i] = heapArr[1];
100:            i = 1;
101:        }
102:        else{
103:
104:            heapArr[i] = heapArr[2];
105:            i = 2;
106:        }
107:
108:        while (i <= Last_inserted){
109:
110:            if (heapArr[2*i] < heapArr[2*i + 1]){
```





```
111:
112:                 heapArr[i] = heapArr[2*i + 1];
113:                 i = 2*i + 1;
114:             }
115:
116:             else{
117:
118:                 heapArr[i] = heapArr[2*i];
119:                 i = 2*i;
120:             }
121:         }
122:
123:         Last_inserted--;
124:
125:         return Removed_Data;
126:     }
127:
128:     void display(){
129:
130:         if (Last_inserted == -1){
131:
132:             cout << "The heap is empty!\n";
133:             return;
134:         }
135:
136:         for (int i = 0; i <= Last_inserted; i++)
137:             cout << heapArr[i] << " ";
138:
139:         cout << endl;
140:     }
141:
142:     int getMin(){
143:
144:         int Log_size = (int)log(Last_inserted);
145:         int Min = heapArr[Last_inserted];
146:
147:         for (int i = Last_inserted; i >= Last_inserted - power(2,Log_size); i--)
148:             if (heapArr[i] < Min)
149:                 Min = heapArr[i];
150:
151:         return Min;
152:     }
153:
154:     int getMax(){
155:
156:         if (Last_inserted == -1){
157:
158:             cout << "The list is Empty!";
159:         }
160:         else
161:             return heapArr[0];
162:     }
163:
164:     int Size(){ return Last_inserted + 1;}
165:
```

```
166:     bool isEmpty(){ return Last_inserted == -1;}
167:
168: private:
169:
170:     int heapArr[100];
171:     int Last_inserted;
172: };
173:
174:
175: //Time Complexity: O(nlog(n)) since heap insert and delete were used inside a loop th
176: void HeapSort(int arr[], int size){
177:
178:     //Declare a heap to sort the array
179:     Heap
180:     //Declare a stack to use its FILO feature
181:     stack<int> S;
182:
183:     //Insert all the array elements in the heap
184:     for (int i = 0; i < size; i++)
185:         H.Insert(arr[i]);
186:
187:     //insert all heap elements in the stack whilst emptying the heap
188:     while (!H.isEmpty())
189:         S.push(H.Delete());
190:
191:     //Insert all stack elements into the array
192:     for (int i = 0; i < size; i++){
193:
194:         arr[i] = S.top();
195:         S.pop();
196:     }
197: }
198:
199: void Display(int arr[], int Size){
200:
201:     for (int i = 0; i < Size; i++)
202:         cout << arr[i] << " ";
203:
204:     cout << endl;
205: }
206:
207:
208: int main(){
209:
210:
211:     int arr[5] = {3, 1, 5, 2, 4};
212:
213:     cout << "The array before being sorted: ";
214:     Display(arr, 5);
215:
216:     cout << "\n\n The array after being sorted: " ;
217:     HeapSort(arr, 5);
218:     Display(arr, 5);
219:
220:     return 0;
```