```cpp
#include <iostream>
#include <fstream>

using namespace std;

ifstream dataIn("infile.txt");
ofstream dataOut("outfile.txt");

void compare(int, int, int);
void quickSort(int a[], int first, int last,int& swaps, int &compare);
int pivot(int a[], int first, int last, int& swaps , int &compare);
void swap(int& a, int& b, int& swaps);


void heapify(int[], int , int);
void heapSort(int[], int, int& , int& );

int main()
{
    int arr[100], bubble[100], Quick[100], heap[100], amount;
    string type;

    while (!dataIn.eof()){
    dataIn>>amount;
    dataIn>>type;
    dataOut<<endl<<amount<<" "<<type<<" in order."<<endl;
    for(int i = 0 ; i < amount; i++){

        dataIn>>arr[i];
        bubble[i] = Quick[i] = heap[i] = arr[i];
        dataOut<<bubble[i]<<" ";
    }

    dataOut<<"\n\n";
      // Bubble Sort Starts Here
     int temp, compareBubble= 0, interchangeBubble = 0 ;
     for(int i=0; i<amount; i++){
        for(int j=0; j<amount-1; j++){

            compareBubble ++;
            if(bubble[j]>bubble[j+1]){

                interchangeBubble++;
                temp=bubble[j];
                bubble[j]=bubble[j+1];
                bubble[j+1]=temp;
            }
        }
    }
    dataOut<<"Bubble Sort Used  :: "<<interchangeBubble<<" interchange : " << compareBubble<<" comparisons"<<endl;
    for(int i = 0 ; i < amount; i++){
      dataOut<<bubble[i]<<" ";

     }
     int interchangeQuick = 0, compareQuick = 0;
    quickSort(Quick, 0, amount, interchangeQuick, compareQuick);
    dataOut<<"\n\nQuick Sort Used  :: "<<interchangeQuick<<" interchange : " << compareQuick<<" comparisons"<<endl;
    for(int i = 0 ; i < amount; i++){
      dataOut<<Quick[i]<<" ";

     }
    int interchangeHeap= 0, compareHeap = 0;
    heapSort(heap, amount, interchangeHeap , compareHeap);
    dataOut<<"\n\nHeap Sort Used  :: "<<interchangeHeap<<" interchange : " <<
```

```cpp
compareHeap<<" comparisons"<<endl;
          for(int i = 0 ; i < amount; i++){
           dataOut<<heap[i]<<" ";

          }
          dataOut<<"\n\nINTERCHANGES :: ";
         compare(interchangeBubble, interchangeQuick, interchangeHeap);
         dataOut<<"\nCOMPARISONS  :: ";
         compare(compareBubble, compareQuick, compareHeap);
          dataOut<<"\n\n\n\n\n";
     }

         return 0;
     }
     void compare(int bubble, int quick, int heap){

         if(bubble>quick && bubble > heap){

             if(quick > heap){

                 dataOut<<"Bubble had the most, then Quick , then heap";
             }
             else if(quick < heap){
                 dataOut<<"Bubble had the most,then heap, then Quick";
             }
             else{
                 dataOut<<"Bubble had the most,then heap and Quick as equal";
             }

         }
         else if(quick>bubble && quick > heap){

             if(bubble > heap){

                 dataOut<<"Quick had the most, then Bubble , then heap.";
             }
             else if(bubble < heap){
                 dataOut<<"Quick had the most, then heap, then Bubble.";
             }
             else{
                 dataOut<<"Quick had the most, then heap and Bubble as equal.";
             }

         }
         else{

             if(bubble > quick){

                 dataOut<<"Heap had the most, then Bubble , then Quick.";
             }
             else if(bubble < quick){
                 dataOut<<"Heap had the most, then Quick, then Bubble.";
             }
             else {
                 dataOut<<"Heap had the most, then Quick and Bubble as equal.";
             }
         }
     }


     void quickSort( int a[], int first, int last, int& swaps, int& compare )
     {
         int pivotElement;
         int counter = 0;
         if(first < last)
         {
```

```
         pivotElement = pivot(a, first, last, swaps,compare);
        quickSort(a, first, pivotElement-1,swaps, compare);
        quickSort(a, pivotElement+1, last,swaps,compare);

    }

}

/**
 * Find and return the index of pivot element.
 * @param a - The array.
 * @param first - The start of the sequence.
 * @param last - The end of the sequence.
 * @return - the pivot element
*/
int pivot(int a[], int first, int last, int& swaps, int& compare)
{
    int  p = first;
    int pivotElement = a[first];

    for(int i = first+1 ; i <= last ; i++)
    {
        /* If you want to sort the list in the other order, change "<=" to ">" */
        compare++;
        if(a[i] <= pivotElement)
        {
            p++;
            swap(a[i], a[p],swaps);
        }
    }

    swap(a[p], a[first],swaps);

    return p;
}


/**
 * Swap the parameters.
 * @param a - The first parameter.
 * @param b - The second parameter.
*/
void swap(int& a, int& b, int& swaps)
{
    swaps++;
    int temp = a;
    a = b;
    b = temp;
}

/**
 * Swap the parameters without a temp variable.
 * Warning! Prone to overflow/underflow.
 * @param a - The first parameter.
 * @param b - The second parameter.
*/
void swapNoTemp(int& a, int& b)
{
    a -= b;
    b += a;// b gets the original value of a
    a = (b - a);// a gets the original value of b
}


//*******************************************************

```

```
void heapify(int arr[], int n, int i, int& swaps, int & compare)
{
    int largest = i;  // Initialize largest as root
    int l = 2*i + 1;  // left = 2*i + 1
    int r = 2*i + 2;  // right = 2*i + 2

    // If left child is larger than root

    if (l < n && arr[l] > arr[largest])
        largest = l;

    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    compare++;
    if (largest != i)
    {
        swap(arr[i], arr[largest],swaps);

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest, swaps , compare);
    }
}

// main function to do heap sort
void heapSort(int arr[], int n, int& swaps , int& compare)
{
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i, swaps, compare);

    // One by one extract an element from heap
    for (int i=n-1; i>=0; i--)
    {
        // Move current root to end
        swap(arr[0], arr[i], swaps);

        // call max heapify on the reduced heap
        heapify(arr, i, 0 , swaps , compare);
    }
}
```