Assignment No.2

1 Aim:

Using Divide and Conquer Strategies design a class for Concurrent Quick Sort using C++.

2 Theory:

2.1 Features of Quick Sort:

- Similar to mergesort - divide-and-conquer recursive algorithm
- One of the fastest sorting algorithms
- Average running time O(NlogN)
- Worst-case running time O(N2)

2.2 Basic idea of Quick Sort:

1. Pick one element in the array, which will be the pivot.
2. Make one pass through the array, called a partition step, re-arranging the entries so that:
   - the pivot is in its proper place.
   - entries smaller than the pivot are to the left of the pivot.
   - entries larger than the pivot are to its right.
3. Recursively apply quicksort to the part of the array that is to the left of the pivot, and to the right part of the array.

2.3 Algorithm:

STEP 1. Choosing the pivot

Choosing the pivot is an essential step.
Depending on the pivot the algorithm may run very fast, or in quadratic time.:

(a) Some fixed element: e.g. the first, the last, the one in the middle This is a bad choice - the pivot may turn to be the smallest or the largest element, then one of the partitions will be empty.

(b) Randomly chosen (by random generator) - still a bad choice.

(c) The median of the array (if the array has N numbers, the median is the [N/2] largest number. This is difficult to compute - increases the complexity.

(d) The median-of-three choice: take the first, the last and the middle element. Choose the median of these three elements.

Example: 8, 3, 25, 6, 10, 17, 1, 2, 18, 5
The first element is 8, the middle - 10, the last - 5.
The median of [8, 10, 5] is 8

STEP 2. Partitioning
Partitioning is illustrated on the above example.

(a) The first action is to get the pivot out of the way - swap it with the last element
5, 3, 25, 6, 10, 17, 1, 2, 18, 8

(b) We want larger elements to go to the right and smaller elements to go to the left.
Two "fingers" are used to scan the elements from left to right and from right to left:

[5, 3, 25, 6, 10, 17, 1, 2, 18, 8]
ˆ ˆ
i j

i. While i is to the left of j, we move i right, skipping all the elements less than the pivot. If an element is found greater then the pivot, i stops

ii. While j is to the right of i, we move j left, skipping all the elements greater than the pivot. If an element is found less then the pivot, j stops

iii. When both i and j have stopped, the elements are swapped.

iv. When i and j have crossed, no swap is performed, scanning stops, and the element pointed to by i is swapped with the pivot . In the example the first swapping will be between 25 and 2, the second between 10 and 1.

3. Restore the pivot. After restoring the pivot we obtain the following partitioning into three groups: [5, 3, 2, 6, 1] [8] [10, 25, 18, 17]

STEP 3. Recursively quick sort the left and the right parts

2.4 Code:

Here is the code, that implements the partitioning.
left points to the first element in the array currently processed, right points to the last element.

```
if( left + 10 <= right)
{
int i = left, j = right - 1;
for ( ; ; )
{
while (a[++i] < pivot )  // move the left finger
while (pivot < a[–j] )  // move the right finger
if (i < j) swap (a[i],a[j]); // swap
else break; // break if fingers have crossed
}
swap (a[I], a[right-1); // restore the pivot
quicksort ( a, left, i-1); // call quicksort for the left part
quicksort (a, i+1, right); // call quicksort for the left part
}
else insertionsort (a, left, right);
```

If the elements are less than 10, quicksort is not very efficient.
Instead insertion sort is used at the last phase of sorting.

2.5 Implementation notes:

Compare the two versions:

A. while (a[++i] < pivot) while (pivot < a[–j])
   if (i < j) swap (a[i], a[j]); else break;

B.
   while (a[i] < pivot) i++; while (pivot < a[j]) j–;
   if (i < j) swap (a[i], a[j]); else break;

If we have an array of equal elements, the second code will never increment i or decrements j, and will do infinite swaps. i and j will never cross.

2.6 Concurrency in Quick sort:

Concurrent Quicksort

Simple concurrent implementation uses a collection of worker threads and a coordinator thread.The coordinator sends a message to an idle worker telling it to sort the array and waits to receive messages from the workers about the progress of the algorithm.

A worker partitions a sub-array, and every time that worker gets ready to call the partition routine on a smaller array, it checks to see if there is an idle worker to assign the work to. If so, it sends a message to the worker to start working on the sub-problem; if not the current worker makes calls the partition routine itself.

After each partitioning, two recursive calls are (usually) made, so there are plenty of chances to start other workers.The diagram below shows two workers sorting the same 5-element array. Each blue line represents the flow of control of a worker thread, and the red arrow represents the message sent from one worker to start the other. (Since the workers proceed working concurrently, it is no longer guaranteed that the smaller elements in the array will be ordered before the larger; what is certain is that the two workers will never try to manipulate the same elements.)

A worker can complete working either because it has directly completed all the work sorting the subarray it was initially called on, or because it has ordered a subset of that array but has passed some or all of the remaining work to other workers.In either case, it reports the number of elements it has ordered back to the coordinator.(The number of elements a worker has ordered is the number of partitions of sub-arrays that have 1 or more members).

When the coordinator hears that all the elements in the array have been ordered, it tells the workers that there is nothing left to do, and the workers exit. That's the basic idea.

Multithreading in C++ :
Demo Example:
Create a function that you want the thread to execute. I'll demonstrate with a trivial example:

```
void task1(std::string msg)
{
std::cout << "task1 says: " << msg;
}
```

Now create the thread object that will ultimately invoke the function above like so: std::thread t1(task1, "Hello");
(You need to #include <thread> to access the std::thread class)
As you can see, the constructor's arguments are the function the thread will execute, followed by the function's parameters.
Finally, join it to your main thread of execution like so:
t1.join();
(Joining means that the thread who invoked the new thread will wait for the new thread to finish execution, before it will continue it's own execution).

2.7 Complexity of Quick sort:

Worst-case: O(N2) This happens when the pivot is the smallest (or the largest) element. Then one of the partitions is empty, and we repeat recursively

the procedure for N-1 elements.
Best-case O(NlogN) The best case is when the pivot is the median of the array, and then the left and the right part will have same size. There are **logN** partitions, and to obtain each partitions we do **N** comparisons (and not more than **N/2** swaps). Hence the complexity is **O(NlogN)**
Average-case - O(NlogN)

2.8 Advantages:

One of the fastest algorithms on average.
Does not need additional memory (the sorting takes place in the array - this is called in-place processing). Compare with merge sort: merge sort needs additional memory for merging.

2.9 Disadvantages:

The worst-case complexity is **O(N2)**

2.10 Applications:

- Commercial applications use Quick sort - generally it runs fast, no additional memory,
- this compensates for the rare occasions when it runs with O(N2) Never use in applications which require guaranteed response time: Life-critical (medical monitoring, life support in aircraft and space craft)
- Mission-critical (monitoring and control in industrial and research plants handling dangerous materials, control for aircraft, defense, etc) unless you assume the worst-case response time.

2.11 Comparison with merge sort:

- merge sort guarantees O(NlogN) time, however it requires additional memory with size N.
- quick sort does not require additional memory, however the speed is not guaranteed
- usually merge sort is not used for main memory sorting, only for external memory sorting.
  So far, our best sorting algorithm has **O(nlog n)** performance: can we do any better? In general, the answer is no.

3 Mathematical Model:

Consider a set S consisting of all the elements related to a program.The mathematical model is given as below,

S={s,e,X,Y,Fme}

Where,
s = Initial State
e = End State
X = Input.

- x={Unsorted Elements}

Y = Output.

- y={Sorted Elements}

Fme = Algorithm

```
int createPartition(int elements[], int begin, int end)
{
int temp, temp1;
int x = elements[end];
int i = begin - 1;
for(int j = begin; j¡= end - 1; j++)
if(elements[j] ¡= x)
i = i+1;
temp = elements[i];
elements[i] = elements[j];
elements[j] = temp;
}
}
temp1 = elements[i + 1];
elements[i + 1] = elements[end];
elements[end] = temp1;
return i + 1;
}
```

4 Testing:

4.1 Black Box Testing:

Input: Unsorted Elements

Enter the Number of Elements:
9

Enter the Unsorted Elements:
19 23 15 1 8 3 6 25 12

Output:Sorted Elements
1 3 6 8 12 15 19 23 25

4.2 White Box Testing:

Test case	Event	Expected Output	Actual Result
1.	Pivot	Last Element should be selected as pivot	last element selected as pivot
2.	pivot position at each iteration	Comparing Elements with pivot and set the position of the pivot	position of the pivot is set at each iteration.and right elements are greater than pivot, and left elements are smaller
3.	Partition	Divide the array into two parts	array is divided into two parts at each iteration
4.	Parallel	each part should be performed parallel	each part is executed parallel.

4.3 Positive and Negative Testing:

Test case	Event	Positive/Negative Test	Input	Actual Result
1.	Enter Unsorted Element	NegativeTestCase#1	No input	Error : at least one element should be enter
		PositiveTestCase#1	Unsorted Element	Sorted Element as output

5 Conclusion:

Concurrent Implementation of Quick Sort with multithreading approach is implemented.

Roll No.	Name of Student	Date of Performance	Date of Submission	Sign.
BECOC357	Sunny Shah	29 / 06 / 2017	13 / 07 / 2017	