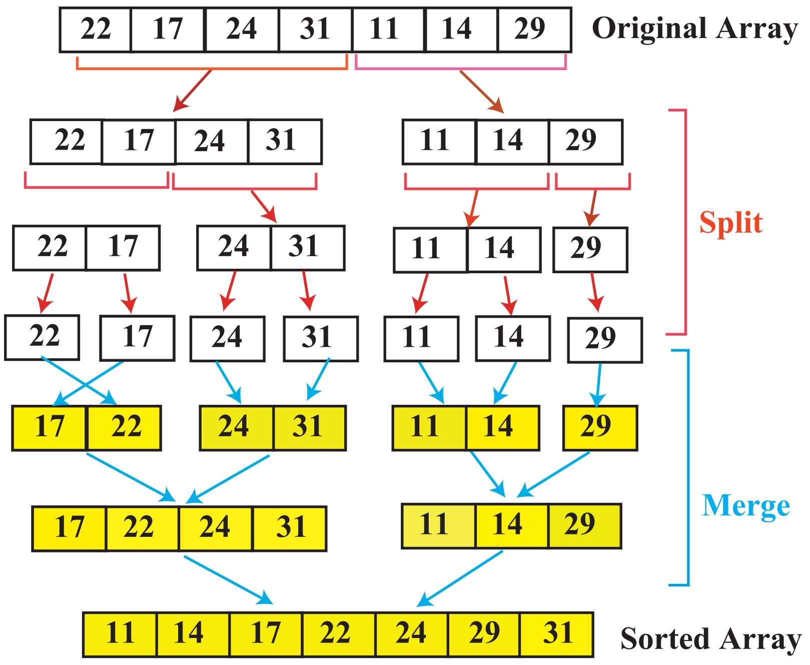
Week 8: Sorting (Part 2)

We discussed some sorting algorithms in Week 7. We discuss some advanced sorting algorithms in this week.

MERGE SORT

The ***merge sort*** is a recursive algorithm. It is also called *splitting* sort because it first splits the array into one-elements sub-arrays. It then merges them into an *n*-element array (See Figure 1).

Figure 1 *Merge Sort*



Merge-Sort Algorithm

Merge sort is a recursive algorithm. We give the algorithm below and leave the C++ program as the assignment.

Merge Sort

```
MergeSort (Array, size)
{
    Allocate an array called leftArray of size1 = size / 2 ;
    Allocate an array called rightArray of size2 = size - size1
    Copy elements from Array into leftArray
    Copy element from Array into rightArray
    Recursively apply the merge sort to leftArray.
    Recursively apply the merge sort to rightArray.
    Combine the one-element arrays into one single array.
}
```

Analysis of Merge Sort

We can use the following argument to analyze the *merge sort*.

1. In the first split we have in average n / 2 exchange.
2. In the second split we have in average n / 4 exchange,
3. In the third split, we have in average n / 8 exchange.
4. And so on.

This means that we need $n/2 + n/4 + n/8 + \ldots$ for the total of $n \log_2 n$ for splitting. We also need the same $n \log_2 n$ for merging. This means the total operations can be defined as $2n \log_2 n$.

The complexity of Merge Sort is $2n \log_2 n$.

Example 1

In a totally unsorted array with 10 elements, we need to do 20 $\log_2$ (10) or approximately 65 operations.

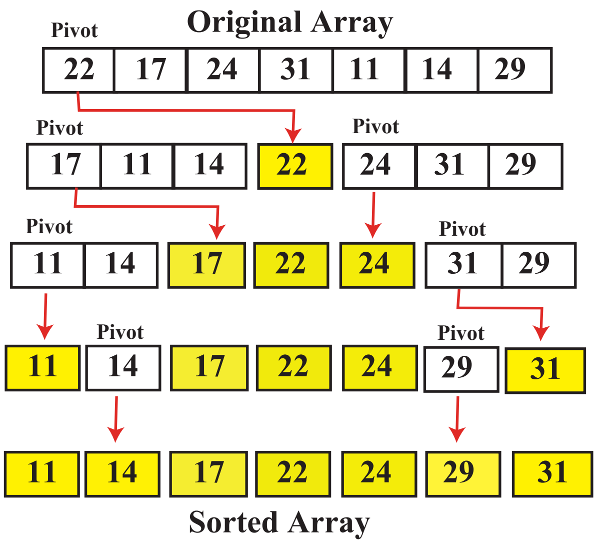
Example 2

Inan unsorted array with 100 elements, we need to do approximately 200 $\log_2$ (100) or approximately 1340 operations to sort the array.

QUICK SORT

The ***quick sort*** like the merge sort is a recursive algorithm. It partitioned the array into two pieces separated by a single element (normally the first one), called the **pivot**, in which all elements in the left piece are smaller than the *pivot* and all elements in the right piece are greater than the *pivot*. This means that the pivot is in the right position. Then the algorithm applies the same strategy on each piece. In each round normally the first element in the corresponding section is chosen as the *pivot*. Figure 2 shows a quick sort

Figure 2 *Quick Sort*



Quick Sort Algorithm

Quick sort is a recursive algorithm. We give the algorithm below and leave the C++ program as the assignment.

Quick Sort

```
QuickSort (Array, first, last)
{
   pivot = first;
   partition (array with pivot at the right place);
   QuickSort (Array, first, pivot - 1)
   QuickSort (Array, pivot + 1, last)
}
```

Analysis of Quick Sort

It can be proved that the complexity of the Quick Sort is O($n \log_2 n$).

Example 3

In a totally unsorted array with 10 elements, we need 10 $\log_2$ (10) or approximately 33 operations.

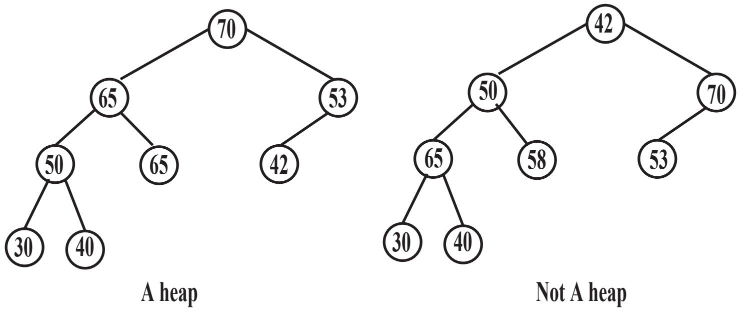
Example 4

In a totally unsorted array with 100 elements, we need 100 $\log_2$ (100) or approximately 669 operations to sort the array.

HEAP SORT

The *heap sort* is based on the idea of a *heap*. A binary tree is a tree in which each element has maximum two children (zero, one, or two). A heap is a binary tree if elements from the root to any leaf are all non-increasing. The following figure shows two binary trees. The left one is a heap; the right one is not a heap.

Figure 3 *A heap*



Note that we can have two or more equal values in a heap. In this case, the first one becomes a parent and the reset the children.

Heap Sort Algorithm

To be able to use the heap sort, we need to make a *heap* from the array and change the heap to an array.

Heap Sort

```
HeapSort (Array, first, last)
{
   Make a heap from original array;
   Copy the root as the last element of the new array
   HeapSort (Array, first, pivot - 1)
   HeapSort (Array, pivot + 1, last)
}
```

Analysis of Heap Sort

It can be proved that the complexity of the Heap Sort is $O(n \log_2 n)$.

Example 3

In a totally unsorted array with 10 elements, we need 10 $\log_2$ (10) or approximately 33 operations.

Example 4

In a totally unsorted array with 100 elements, we need100 log2 (100) or approximately 670 operations to sort the array.

PRACTICE SETS

Homework Assignments

PR-1. Manually use the merge sort to sort the following sequence. Show all of the steps.

```
10, 17, 9, 44, 22, 8, 14,71
```

PR-2. Manually use the quick sort to sort the following sequence. Show all of the steps.

```
15, 10, 21, 8, 14, 18, 14, 12
```

PR-3. Manually use the heap sort to sort the following sequence. Show all of the steps.

```
20, 33, 10, 6, 54, 36, 21, 21
```

Lab Assignments

PRG-1. Write a program in C++ using the merge sort and test it with the following data. You need to give the program and show the result.
14, 22, 41, 56, 34, 12, 19, 52

PRG-2. Write a program in C++ using the quick sort and test it with the following data. You need to give the program and show the result.
10, 17, 9, 44, 22, 8, 14, 71

PRG-3. Write a program in C++ using a heap sort and test it with the following data. You need to give the program and show the result.
11, 25, 5, 14, 7, 78, 44, 44