CI583: Data Structures and Operating Systems
Recursion, and some recursive problems

MergeSort

MergeSort is a recursive algorithm that is much more efficient that the simple sorting methods we studied earlier: it is loglinear, or $O(n \log n)$, rather than $O(n^2)$.

To sort 10,000 items, this means that MergeSort is proportional to 40,000 steps, while SelectionSort is proportional to 100,000,000 steps.

If the MergeSort took 40 seconds, the SelectionSort would take in the region of 28 hours.

The downside is that MergeSort uses extra memory, making it a good choice only if we have enough space.

MergeSort

At the heart of MergeSort is the process of merging two sorted arrays, A and B, into a third array, C.

We start with pointers to the front of A and B and compare the first elements. We move the smallest into C and increment the pointer for the array it came from.

So, if the first element is moved from $B[0]$ into C then the next step will be to compare $A[0]$ and $B[1]$, and so on.

A and B need not be the same size. When we get to the end of either A or B we can move elements from the remaining array without any further comparisons.

MergeSort

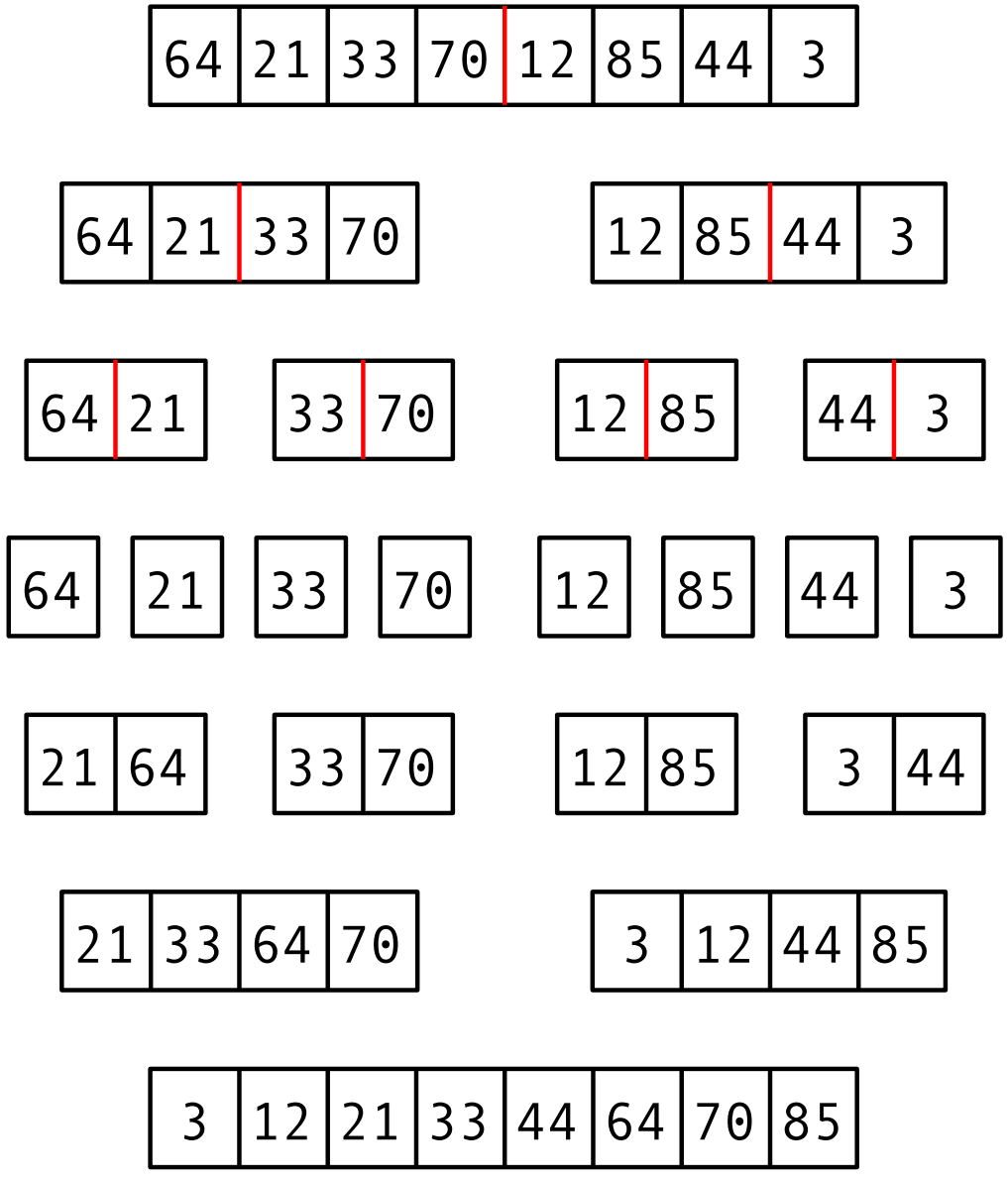
That explains how to merge two sorted arrays, but how did they become sorted in the first place?

The idea behind MergeSort is to divide an array into two halves, sort the halves then merge them.

We do this recursively, dividing the subarrays until we reach the base case: an array with one element.

A one element array is already sorted, so when we have two of those, we just need to merge them.

MergeSort


64 21 33 70 12 85 44 3
64 21 33 70
12 85 44 3
64 21
33 70
12 85
44 3
64
21
33
70
12
85
44
3
21 64
33 70
12 85
3 44
21 33 64 70
3 12 44 85
3 12 21 33 44 64 70 85

MergeSort

Although we are hiding the complexity in the Merge algorithm, the recursive MergeSort algorithm is extremely elegant.

The storage for all of these smaller arrays comes from allocating a temporary array with the same size as the input.

```
procedure MergeSort(array, first, last)
    if first < last then
        mid ← (first + last)/2
        MergeSort(array, first, mid)
        MergeSort(array, mid + 1, last)
        Merge(array, first, mid, mid + 1, last)
    end if
end procedure
```

Complexity of MergeSort

As an informal consideration of the growth rate of MergeSort, consider the example from a few slides ago:

24 swaps were needed to sort 8 items. $\lg 8$ is 3, and $8 \times \lg 8$ is 24.

The swapping required, which is a similar process to a binary search, halving the problem each time, accounts for the logarithmic component in $O(n \log n)$.

As for the number of comparisons, in the worst case, merging n items could take $n - 1$ comparisons, and this is where the n in $O(n \log n)$ comes from.

QuickSort

The last sorting algorithm we will look at is QuickSort.

Like MergeSort, it is highly efficient, but in this case only the average and best performance are $O(n \log n)$.

The worst case performance is much worse, $O(n^2)$. Unlike MergeSort, however, QuickSort does not require extra memory.

QuickSort

Because of the combination of low memory usage and very good average performance, QuickSort is one of the most popular sorting algorithms.

You may find that sorting a collection with the standard library method in your favourite language uses it (Java does, for example).

QuickSort

The idea is that we pick a "pivot element", p, then move every element which is less than p to its left, and every element greater than p to its right.

The two halves are not sorted, just less than or greater than p.

At this point, p is in its final position. Then we call QuickSort recursively on the left and right.

The actual sorting is done recursively by the method that moves elements before or after the pivot, so the real work is down on the way down.

This is the opposite of MergeSort, where the sorting is done on the way up, merging smaller lists into bigger ones.

QuickSort

```
procedure QuickSort(array, first, last)
    if first < last then
        p ← PivotList(array, first, last)
        QuickSort(array, first, p − 1)
        QuickSort(array, p + 1, last)
    end if
end procedure
```

QuickSort

There are several ways that we might choose the value of p.

We would like a method that avoids picking a very large or small value, relative to the rest of the list, but obviously we are unable to examine every element.

An optimisation that has been found to be quite effective is to examine the first, last and middle elements and pick the median – this is the median-of-three approach.

At the same time as choosing the pivot, we sort the three elements, so that the pivot is in the mid point.

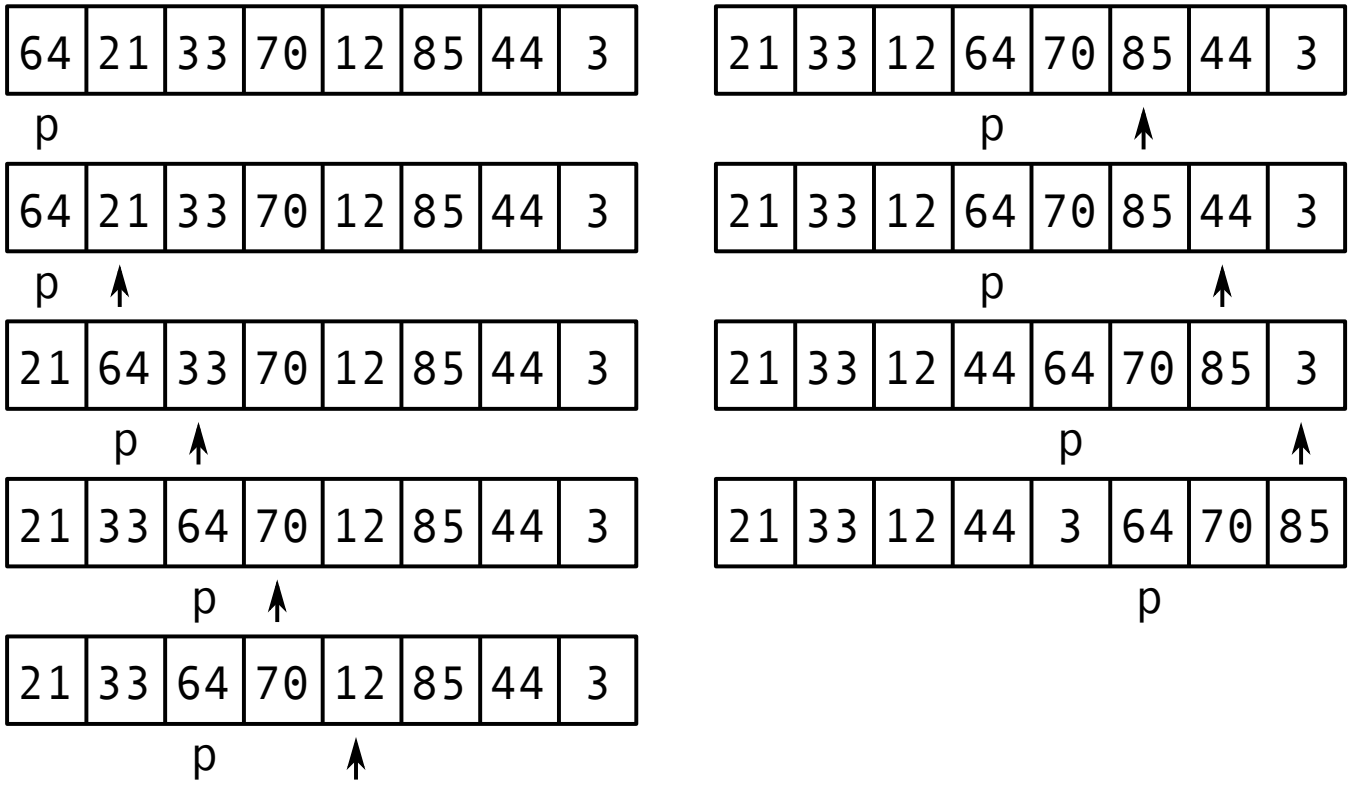
QuickSort

In the example coming up we use the simplest way of picking the pivot:

set p to the first element then move any element less than p before it and leave everything else where it is.

QuickSort

One run of PivotList:



QuickSort

When PivotList is called on a list with n elements it needs to do $n - 1$ comparisons.

The worst case is one in which every element is larger (or smaller) than p.

In this case we end up with $O(n^2)$ performance, no better than BubbleSort.

One situation that will cause this is if the list is already sorted or inversely sorted.

QuickSort

In the best case, p will end up in the middle of the partition being sorted, and this halving-every-time gives us loglinear performance.

Analysing the average case performance results in the same order: $O(n \log n)$.

See, for example, the Cormen book for full details.

Next week

Hashtables!