Lecture C11

Response to 'Muddiest Part of the Lecture Cards'

(9 respondents)

1) *It seems like merge sort is a very poor algorithm. Once you break it down to length-2 strings, they must often still be split to put other elements from say a different string in. Why not just pop them all to memory and insertion sort*?

I am not really sure how you came to this conclusion. If you analyze the two algorithms you will see that merge sort has a lower upper bound (**$n*\log_2 n$**) than the upper bound for insertion sort (**n^2**).

The basic idea for Insertion sort is to scan the array from the beginning to the end. Insert the current element into its proper position (which is found using a search algorithm). The running time of Insertion Sort is

Best case : Array is already sorted
- Using linear search → O(n)
- Using binary search → O(n log n)

Worst case : Array is already sorted, but in reverse order
- Using linear search → $O(n^2)$
- Using binary search → $O(n^2)$

Average case is same as worst case

Merge sort is a *divide and conquer algorithm* (break list in to two halves, sort each half recursively, merge the two sorted halves together). The base case is when arrays reach size 1, i.e., they are already sorted. The analysis of Merge Sort was shown in Lecture 11 slides: Time to merge sort N elements = twice the time to merge sort n/2 elements + time to merge the N numbers which eventually → $O(n \log_2 n)$

For large values of **n**, the function **n*log n** grows much slower than **n^2**

2) *Items that Build_Heap sorts using heapify, gets an array that's from highest to lowest, then heap_sort reverses the array by using heapify. (+ related questions – 5 students)*

```
Build_Heap(A);
for i in size downto 2
     Swap(A[1], A[i])
     heap_size:= heap_size -1;
     Heapify(A, 1);
```

The Build_Heap statement, ensures that the array satisfies the heap property i.e. the largest element in the array is in location 1. It does not however order its children i.e. left child could be greater than right child and vice versa.

`Swap(A[1], A[i])`, moves the largest element to the last unsorted location in the array.

```
Heapify (heap_array, i)
        lchild := Left(i)
        rchild := Right(i)
        if (lchild <= heap_size(A) and A[lchild] > A[i])
                largest= lchild
        else
                largest = i
        if (rchild <= heap_size(A) and A[rchild] > A[largest])
                largest = rchild
        if (largest != i)
                Swap(A, i, largest)
                Heapify(A, largest)
```

The heapify function uses the heap_size to determine when to stop the recursion. The `heap_size:= heap_size -1` statement in the heap sort program, ensures that elements in the array that are already sorted are not displaced by the heapify function.

The elements in A(1 .. size-1) are no longer a heap. This is remedied by the `Heapify(A, 1)` statement.

This process is repeated until the entire array is sorted.

3) *In Merge Sort, what is Merge(A, left, mid, right)*?
The procedure Merge takes 2 sorted sub-arrays and merges them into one array containing all the elements between left .. right. The left, mid, right, specify the bounds pf the sorted subarrays.

For example Merge(A,1,4,7) merges subarrays A(1..4) and A(5..7).

Array prior to merge:

1	2	3	4	5	6	7
1	3	7	9	-1	5	12

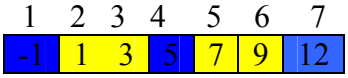
A(1..4)

1	2	3	4	5	6	7
1	3	7	9	-1	5	12

A(5..7)

1	2	3	4	5	6	7
1	3	7	9	-1	5	12

Array after merging



4) *No mud* (3 students)