Tutorial 1

Harjot Singh Oberai (15114032)

Merge Sort

Algorithm:

```
procedure mergesort(var a as array)
   if (n == 1) return a

   var l1 as array = a[0] ... a[n/2]
   var l2 as array = a[n/2+1] ... a[n]

   l1 = mergesort(l1)
   l2 = mergesort(l2)

   return merge(l1, l2)
end procedure

procedure merge(var a as array, var b as array)

   var c as array

   while (a and b have elements)
      if (a[0] > b[0])
         add b[0] to the end of c
         remove b[0] from b
      else
         add a[0] to the end of c
         remove a[0] from a
      end if
   end while

   while (a has elements)
      add a[0] to the end of c
      remove a[0] from a
   end while

   while (b has elements)
      add b[0] to the end of c
      remove b[0] from b
   end while

   return c

end procedure
```

Correctness:

Correctness of MergeSort
- Correct on one element -- returns with no further sorting
- More than one element -- sort each half (closer to base case)
- Assuming Merge works correctly, merged array will be sorted
- We are really passing off all the work of sorting into Merge

Correctness of Merge
- Left and right array halves are sorted (assume!)
- Minimum of whole array either lies on left or right
- Will be minimum of its half, so first in its half
- Whichever minimum is smaller is minimum of whole array
- Copy into first element of sorted array, then skip over it next time
- Now second-minimum of whole array either lies on left or right, etc.
- Continue in this fashion until left or right side is completely used up
- Then just copy remaining elements from other half onto end of sorted array

Analysis:
The recurrence relation is T(n) = 2T(n/2) + O(n)
The first term is because of the 'divide' operation, i.e. dividing the array into two parts and recursively sorting them. The second term O(n) is because of the linear mergeing operation of two sorted arrays.
Solving this recurrence relation gives T(n) = O(nlogn)
The best case, average case as well as the worst case complexity of merge sort is O(n logn)

Quick Sort

Algorithm:

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p – 1)
        quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
    pivot := A[hi]
    i := lo     // place for swapping
    for j := lo to hi – 1 do
        if A[j] ≤ pivot then
            swap A[i] with A[j]
            i := i + 1
    swap A[i] with A[hi]
    return i
```

Correctness:
For n=1, the array is already sorted.
For n>1 consider array divided into three parts,first subarray say of elements k then a pivot element and then remaining n-k-1 elements. After choosing the pivot we divide the array into two parts one with all elements smaller than pivot and other with all elements greater than pivot and then put the pivot element in between these two parts.Now the pivot is at its place where it should be after sorting the whole array.We now again repeat these steps to the left and right parts of the array.

Analysis:
Worst Case:
In the worst case, the pivot element is supposed to be the first element, in this case:
$T(n) = O(n)+T(0)+T(n-1)$
Which gives $T(n) = O(n^2)$
Best Case:
In the best case, the pivot element is supposed to be the middle element in every recursive step. In this case, $T(n) = O(n) + 2*T(n/2)$ which is the same relation as merge sort. Hence,
$T(n) = O(nlog(n))$
Average Case:
In average case, the algorithm roughly takes O(nlog(n)) time.