# Lecture C10

## Response to 'Muddiest Part of the Lecture Cards'

(7 respondents)

1) *Is insertion sort just bubble sort*?
No. Both algorithms have a worst case complexity of $O(n^2)$, but the insertions sort algorithm has a better best case complexity than the bubble sort algorithm.

The insertion sort algorithm inserts each item directly into its proper place in the list. It does so by comparing the value to each item in the already sorted list by swapping place with the preceding element until it finds its proper place in the list.

```
InsertionSort(A, n)
for j in 2..n loop
  key:= A[j]
  i  := j-1
  while i > 0 and A[i] > key
     A[i+1]:= A[i]
     i:= i-1
  A[i+1]:= key
end loop
```

The bubble sort algorithm compares each item to the neighboring item and the items are swapped if needed. This process is repeated until a pass through the list does no longer generate any swaps between items.

```
last := length;
for I in 1 .. Last –1 loop
     for J in I+1 .. Last loop
               if List(I) < List(J) then
                     swap list(i) and list(j)
          end if
     end loop
end loop
```

Note that the complexity in the best case is also $O(n^2)$. You have to compute it in your pset.

2) *Lost as to what Big-O is?*
Big-O notation allows us to perform an asymptotic analysis on resource usage (the resource could be memory used or processor time). This allows us to compare two algorithms in terms of best case, worst case and average case behavior. If T(n) is the computation time of the algorithm, then its asymptotic behavior can be expressed as

$T(n) = O(f(n))$ such that $T(n) \leq O(c\ f(n))$, for a constant c and all $n \geq n_0$

The two constants c and $n_0$ are defined such that the algorithm always executes in less that c*f(n) time.

3) *Can you write up / revisit the question about searching a sorted array?*
Can and will do shortly in lecture C11.

If the array is sorted in ascending order, then checking the element in the middle of the array allows you to determine which half of the array to search. The binary search algorithm does exactly that.

- The algorithm stops either if the element is found or if the lower bound is greater than the upper bound.
- If the element in the middle of the array is less than the element you are searching for, set the lower bound to middle+1
- Else, set the upper bound to middle -1

How do we end up with $T(N) = O(\log_2 N)$?

```
The Binary Search Algorithm
Input:   Array to search, element to search for
Output: Index if element found, -1 otherwise
Algorithm:

Set Return_Index to -1;                          -- c1
Loop                                             -- c2 (log n +1)
       Set Current_Index to (UB + LB) /2         -- c3 (log n +1)
       if the LB > UB                            -- c4 (log n +1)
              Exit;                              -- c5

       if Input_Array(Current_Index) = element -- c6 log n
              Return_Index := Current_Index      -- c7
              Exit;                              -- c8

       if Input_Array(Current_Index) < element -- c9 log n
              LB := Current_Index +1             -- c10 log n
       else                                      -- c11 log n
              UB := Current_Index – 1            -- c12 log n

Return Return_Index                              -- c13
```

$T(n) = (c1 + c2 + c3 + c4 + c5 + c7 + c8 + c13) +$
$(c2 + c3 + c4 + c6 + c9 + c10 + c11 + c12) \log n$
$= c' + c''\log(n)$
$= O(\log (n))$

4) *What's the answer to life?*
The answer is: **42** – The meaning of life, the universe and everything
"The hitchhiker's guide to the galaxy" by Douglas Adams

5) *No mud, seeing the mathematical side of computers is perfect* (3 students)