Take away from last two lectures (warm up lectures for the course) [1]

1. To find maximum (or minimum) of n numbers in an array using comparison, we need at least $n-1$ comparisons. *Find the reason out. It was discussed in the previous class.*

2. To find maximum and minimum of given n numbers in array using comparison, we need at least $(3n/2)-2$ comparisons. An algorithm was discussed in the class which finds maximum and minimum using the $(3n/2)-2$ comparisons.

3. To find maximum and second maximum of given n numbers in array using comparison, we need at least $n+o(n)$ comparisons. An algorithm was discussed in the class which finds maximum and second maximum using the $n-1+\log n-1$ comparisons. The algorithm was based on Tournament Data Structure. I highly encourage you to study this data structure. It has many applications. Try to implement it to get more insights.

4. Given two arrays A and B, both containing n numbers. Find common elements of both arrays and store in C. This is a straight forward n application of Binary Search. Please look in to more applications of Binary search.

5. A naive paradigm was discussed in class- Inductive programming. The idea of using this paradigm is two stage-“Think Backward, Solve forward”. During the first stage, one should ask question to him(her)self that given a solution of problem of size $n-1$, can we solve it for the problem of size n. Here, recursive thinking will help you. Then instead of solving backward, i.e., recursively, the paradigm solves a problem forward, i.e., sequentially or iteratively.

6. To prove the correctness of the algorithm, try to find loop invariant(s). Use induction on loop invariant(s) to show the correctness of the algorithm.

   Example of proving loop invariant using induction

7. Finding max of numbers: Use the above logic. If given solution for the problem of size $n-1$, we can solve it for the problem of size n. Use recursion to think backward.

   Solving forward (Algo. for max):

   $max = A[1]$; for $(i = 1$ to $n)$\{if $(max < A[i])$ $max = A[i]$; \} return max;

   Loop invariant: ”After iteration i, the variable *max* contains the maximum element seen so far in the subarray A[1...i]”.

   Proof:

   Base case: At the beginning of the loop, when i = 1, *max* is initialized to A[1]. Since A[1...1] is the subarray containing only the first element of the array, and max is initialized to that element, the loop invariant holds.

   Induction Hyp: Assume that the loop invariant holds for some i = k, where $k > 1$. That is, assume that after the k-th iteration, *max* contains the maximum element seen so far in the subarray A[1...k].

   To prove ind step, use 1. Logic of your algorithm, 2. Induction Hypothesis.

   Inductive step: We need to prove that the loop invariant holds for $i = k + 1$. To do this, we examine the logic of the loop for the (k+1)-th iteration.

   In the $(k + 1)$-th iteration, we compare the current element of the array, A[k+1], with *max*. If A[k+1] is greater than *max*, we update *max* to be A[k+1]. If A[k+1] is less than or equal to *max*, we leave *max* unchanged.

   Now, after the (k+1)-th iteration, *max* contains the maximum element seen so far in the subarray A[1...k+1]. This is because if A[k+1] is greater than *max*, *max* is updated to be A[k+1], which is the largest element in the subarray A[1...k+1]. Otherwise, if A[k+1] is less than or equal to *max*, *max* remains unchanged, which still contains the maximum element seen so far in the subarray A[1...k]. Therefore, the loop invariant holds for i = k+1.

[1]Prepared by Pawan K. Mishra

8. Time complexity of above pseudo code is $O(n)$. Note that we can also use $T(n) = T(n-1) + 1$ to justify the time complexity. This is another way to see the time complexity. “Thinking backward” logic will help you to see this recurrence relation. Think hard, yes you are close !!

9. Similar approach can be applied for linear search, insertion sort, bubble sort, selection sort. Note that finding a good loop invariant(s) is a challenging task. Proving via induction is straight forward.