Assignment - 1

2.1) Finding the maximum element is O(1) in both a max-heap and a sorted array. so, both are equally good.

2) Deletion in heap takes O(lg n) time to go from top to bottom in order to heapify again (if required). However, a sorted array requires O(n) for all elements to get updated after deletion. Thus, heaps are more optimal than sorted array.

3). Heap formation requires O(n) time, while sorted array requires O(n lg n) time. Thus, heaps are more optimal.

4) For finding the minimum element in a max-heap, we have to go to each of the leaf node i.e. O(n) operation in worst case, however, it is O(1) for a sorted array. So, arrays perform better than heaps.

3) We know that the lower bound of searching is O(n lg n). so, according to Mr Dull, one could construct a linear time sorting algorithm, by inserting all elements and extracting max elements one by one. So, this is not possible. Hence, his claim is incorrect.

5) Iterate over the list of integers and convert each one to base n. Converting to base n will take lesser time than base 10. This is usually useful when values of n is large. Each number will have at most $\lg_n n^3$ = 3 digits, so 3 loops are required. For each loop, there are n possible values, hence we can use counting sort to sort each digit in O(n) time.

1) We have to perform two passes of the partition operation from QuickSort. First, we have to treat red and white elements as indistinguishable, and separate them from blue. Then, we have to separate the elements within the red / white sub-array.

Let (i_1, i_2) be indices of Red, (j_1, j_2) be indices of White and (k_1, k_2) be indices of blue. First initialise all of them as 0. Now, the algorithm will be - (consider a as iterable object). We are iterating over the array. So,

```
if ( Examine (A, a) == Blue)
    { a++;
      k2++;
    }.
if ( Examine (A, a) == White)
    { ~~swap (Elements [k1], Elements~~
      swap ( A, k1, a);.
      j2++;
      k1++;
      k2++;
    }.
if (Examine (A, a) == Red)
    { swap (A, k1, a);.
      swap (A, k2, j1);
      i2++;
      j1++;
      j2++;
      k1++;
      k2++;
    }.
```

This O(n) complexity algorithm.

Note: i_1, i_2 are starting and ending indices of A when it is sorted.

4) The main concept used behind creating a min-heap (if we have to do in O(n), is to ~~compare~~ iterate from the end of the array and compare it with its parent node. If the child is less than its parent, swap min of two child with its parent and so on.

Pseudocode:

```
for ( int i = n/2 - 1 ; i >= 0 ; i-- )
{   Heapify ( arr, n, i) {
    initialised i as the smallest as root = smallest ;
    l = 2*i + 1 ;
    r = 2*i + 2 ;
    if ( left child < root)
    {
        smallest = l ;
    }
    if (smallst != root)
    {
        swap arr[i] and arr[smallst];
        heapfy (arr, n, smallest);
    }
}
}
```

Proof for O(n):

Height of heap = $\lg n$.

For each operation, complexity is O(1).

So, recursive relation:

$T(n) = T(n-1) + O(1)$ (∵ height will be $\log(n-1)$ in next recursive call).

$\Rightarrow T(n) = O(n)$