1. Question 9.3-3 from CLR (second Addition). **Answer:** We use median-selection to find the pivot of merge sort. Since finding the median of $n$ elements takes $\leq cn$ time (for a constant $c$), the running time of the whole algorithm obeys the formula $T(n) \leq cn + c'n + 2T(n/2)$ (where $c'n$ is the time needed for the partitioning). The solution of this formula is $O(n \log n)$.

2. Question 9.3-9 from CLR (second Addition).

**Answer:** The pipe must pass above exactly $n/2$ of the wells. If it passes above more than $n/2$, then by shifting the line down we decrease its vertical distance to these ones, but decrease the distance to $< n/2$ lines, so altogether the total distance decreases. An analogous argument holds if it passes above less than $n/2$ wells.

3. Problem 9–1 from CLR (second Addition).

**Answer:** (a) $O(n \log n)$. (b) Takes $O(n) + O(i \log n)$. (c) Takes $O(n) + O(i \log i)$.

4. You are given a set $L$ of $n$ lines in the plane, in a sorted order order of slopes. Show, using a potentials function that the running time of the algorithm studied in class for computing the lower envelop of $L$ is $O(n)$.

**Answer:** Assume $L = \{\ell_1 \ldots \ell_n\}$ in sorted order of slopes.

Let $F_i$ denote the lower envelope of the lines $\{\ell_1 \ldots \ell_i\}$. Let $\phi_i$ denote the number of lines on the lower envelope after inserting $\ell_i$. If in the $i$th stage $k$ segments of $F_{i-1}$ need to be scanned, than all but the last one can also be deleted (as argues in class) so $c_i$, the actual work at this stage is $k$, and $\phi_i - \phi_{i-1} = 1 - k$. Hence the amortized time $\hat{c}_i$ is

$$\hat{c}_i = c_i + \phi_i - \phi_{i-1} = k + (1 - k) = 1$$

5. The standard operations defined on a stack $S$ are pop($S$) that returns the element in the top of the tact and remove it from the stack, and push($S$, $x$) that pushes $x$ into $S$. 
The operation on a queue \( Q \) are \( \text{EnQueue}(Q, x) \) that insert the element \( x \) into the tail of \( Q \), and the operation \( \text{DeQue}(Q) \) that returns the element at the head of \( Q \), and remove it from \( Q \).

Assume that you are given two stacks \( S_1, S_2 \), and \( O(1) \) memory in addition. Explain how you can support \( O(n) \) operations on a queue, where the only operations done on the stacks are of the type \( \text{push}(S_1, x) \), \( \text{pop}(S_1) \), \( \text{push}(S_2, x) \), and \( \text{pop}(S_2) \). So that a sequence of \( m \) \( \text{EnQueue} \) and \( \text{DeQueue} \) operations would require \( O(m) \) operations on the stack.

**Answer:**

```plaintext
Function \text{EnQueue}(x, Q)
   Push(S_1, x)

Function \text{DeQueue}(Q)
   While \( S_1 \) is Not empty Do
      push(S_2, pop(S_1))
   Return pop(S_2)
```

Each element is inserted into each of the stacks exactly once, so the total time is \( O(m) \).

6. Problem 17-2 from CLR (Second edition) a,b. Section c is more challenging.

**Answer:**

(a) Since the number of arrays is \( O(\log n) \), and search is done by performing a binary search in each, of sizes \( 1, 2, 2^2 \ldots 2^{[\log_2 n]} \), and it takes \( \Theta(\log_2 2^i) = \Theta(i) \) time to perform a binary search in each, the query time is (in the worst case)

\[
\Theta \left( \sum_{i=[\log_2 n]}^1 i \right) = \Theta(\log_2^2 n)
\]

(b) To perform insert of a new element \( x \), create an array of size 1 for \( x \). Next, we repeat: As long as there are two arrays of the same size, we merge them into an array of double size. We need to merge an array of size \( m = 2^k \) only after \( k \) insertions, and the merge process takes \( cm \) time (for a constant \( k \)). Hence the time needed for \( n \) insertions is

\[
 cn + 2c \frac{n}{2} + 4c \frac{n}{4} + \ldots + c 2^i \frac{n}{2^i} + \ldots + nc = cn \log_2 n
\]

(where we assume for simplicity that \( n \) is a power of 2. Thus the amortized time for an insertion is \( O(\log n) \).

A slightly different way to obtain the same time bound, is to note that an element can be moved from an array of size \( m \) to an array of size \( 2m \).
(during a merging process) only once, and so it can be moved at most \( \log_2 n \) times, and each time that an element is transferred to a new array we spend \( c \) time.

One can obtain the same running time you obtained for this question, but in the worst case setting (i.e. not amortized). The idea is to keep a few copies of the data structure. Once merging of two arrays of size \( m \) is required as a result of inserting a new element, the merging process is divided into small tasks, so that each is accomplished during a sequence of \( m \) operations. Can you show the details here, and prove that the running time is not changed?

7. Question 17.4-3 from CLR. You can prove the result in any way you choose.

8. Question 27 a,b from the handout on Splay trees. See how you feel about parts c,d.