Question 1.

A. Merge sort	————	4. Stable and fast
B. Quick sort	————	6. Fast general-purpose sort
C. Shell sort	————	3. Not analyzed
D. Insertion sort	————	2. Works well with order
E. Selection sort	————	5. Optimal data movement
F. 3-way quicksort	————	1. Works well with duplicates

Question 2.

Assume sort() is O(NlogN). (because it might be Mergesort or Quicksort)
The complexity of binary search is O(logN). As there are two for loops, and the binary search is in the inner loop. So, for this code, the complexity is
$O(N\log N) + N^2 * O(\log N) = O(N^2\log N)$
Replace N with value 3500 and 35000, we get the ratio is
$(35000^2 \log 35000) / (3500^2 \log 3500) \approx 103$
$103 * 1$second $\approx$ 2 minutes
So, this code will take 2 minutes to execute when N = 35000.

Question 3.

We know that the time complexity of merge sort is NlogN, which means the number of compares of merge sort is NlogN.
$C(N) = N\log N$, $C(N+1) = (N+1)\log(N+1)$
$C(N+1) / C(N) = (N+1)\log(N+1) / N\log N$
Since N is integer and $N>0$, we can get $\log(N+1)>\log N$, and also $(N+1)>N$.
So $(N+1)\log(N+1)/N\log N > 1$, which means $C(N+1)>C(N)$. In conclusion, the number of compares used by merge sort is monotonically increasing for all $N>0$.

Question 4.

Set three arrays named A, B, C separately. They have the same length N.
Use merge sort or quick sort sort B and C, which will take the total of (NlogN + NlogN) time. Then try to find every element from A in B and C use binarySearch. Each of them will cost NlogN time. So, the total will be NlogN + NlogN + 2* NlogN = 4* NlogN, which is O(NlogN). Once we find an element that both exist in B and C, return that element.

Question 5.

(a) 4 3 2 1 0 9 8 7 6 5 This is possible.

Push 0 1 2 3 4 into stack. Pop 5 times we get 4 3 2 1 0.

Push 5 6 7 8 9 into stack. Pop 5 times we get the output 4 3 2 1 0 9 8 7 6.

(b) 4 6 8 7 5 3 2 9 0 1

Push	Pop	Output	Stack Remain
01234	4	4	0123
56	6	4 6	01235
78	87	4 6 8 7	01235
	532	4 6 8 7 5 3 2	01
9	910	4 6 8 7 5 3 2 9 1 0	

So, there’s no way the last two outputs are 0 1. Therefore, this output order is impossible.

(c) 2 5 6 7 4 8 9 3 1 0

Push	Pop	Output	Stack Remain
012	2	2	01
345	5	2 5	0134
6	6	2 5 6	0134
7	7	2 5 6 7	0134
	4	2 5 6 7 4	013
8	8	2 5 6 7 8	013
9	9	2 5 6 7 8 9	013
	310	2 5 6 7 8 9 3 1 0	

In conclusion, this output order is possible.

(d) 4 3 2 1 0 5 6 7 8 9

Push	Pop	Output	Stack Remain
01234	43210	4 3 2 1 0	
5	5	4 3 2 1 0 5	
6	6	4 3 2 1 0 5 6	
7	7	4 3 2 1 0 5 6 7	
8	8	4 3 2 1 0 5 6 7 8	
9	9	4 3 2 1 0 5 6 7 8 9	

This output order is also possible.

Question 6.

In order to find the number we wanted.

Because this is a bitonic array, we need find the changing point. (The position that the monotonic changes) The partial array before (or include) this number is increasing, and after (or include) this number is decreasing. We assume this array's name is A. And the changing point is A[a].

Assume the number we want find is x.

```
a =0; b = N-1;
while(a!=b){
    int m = (a+b)/2;                    //get the mid of the array
    if(m && a[m-1]<a[m])                //if the array is increasing at this point
        a = m;
    else
        b = m;
}
return binarySearch(A, 0, a, x) || binarySearch(A, a+1, N-1, x);
```

The while loop that we used to find the changing point will take logN time. In worst case, we need do binarySearch for both part of the array. Because we divided the array into two parts, the first part contains a elements and other part has (N-a) elements. So the binarySearch will take log(a) and log(N-a) separately.

Therefore, log(a) is O(logN) and log(N-a) also O(logN).

In conclusion, the time complexity of worst case is logN + O(logN) + O(logN) ~ 3logN.