1. The best time complexity for this function would be O(1) because the best case is when (*ptr == target) at first iteration. So, loop runs only one time and memory allocated for this pointer is also one, so the space complexity for this case is also O(1).

   For worst case, the target value would be at last or exceeding the limit. So, the loop would run n times so time complexity for worst-case would-be O(n). For space complexity for this case would be O(n) because it allocates a new memory every n time until target value is not equal to the *ptr. So, space complexity for this case is also O(n).

2. a) O(nlogn); The outer loop runs n times, and it incremented by i*2, so it doubled the value each time. The time complexity for outer loop is O(logn). The inner loop depends on the value of i. The value of I doubled each time, so the sum is 1+2+4+….+n which is n times. So, total complexity of these nested loops is O(nlogn).

   b) O(n^2) because outer loop runs n times and inner loop runs depends on I so when i is n inner loop runs n times, so the total sum should be [n(n+1)]/2. Time complexity is O(n^2).

   c) O(nlogn); The outer loop runs n times, and it incremented by i*2, so it doubled the value each time. The time complexity for outer loop is O(logn). The inner loop runs n times every time i runs.So, time complexity for inner loop is O(n). So, total complexity of these nested loops is O(nlogn).

3.

	1 Second	1 Hour	1 Month	1 Century
Log n	10^{300000}	$10^{1000000000}$	$10^{782000000000}$	$10^{933000000000000}$
$N^{1/2}$	10^{12}	12.96×10^{18}	6.76×10^{24}	9.6×10^{30}
n	10^{6}	3.6×10^{9}	2.6 x 10^12	3.1×10^{15}
N log n	4.34×10^{19}	7.6×10^{9}	1.1×10^{6}	
N^2	10^{3}	6×10^{4}	1.6×10^{6}	5.6×10^{7}
N^3	10^{2}	1.4×10^{3}	1.4×10^{4}	1.4×10^{5}
2^n	20	2^5	40	51
N!	9	12	19	39

4. Linear time algorithm for rearranging keys using swapping and with best space complexity is:

```
rearranging(keys, size)
        indexRed = 0;
        index = size-1;
        indexWhite = size-1;

        while(indexRed <= index)
                if(keys[index = red])
                        swap(keys[index], keys[indexRed])
                        increment indexRed by 1
                else if(keys[index] = blue])
                        decrement index by 1
                else if(keys[index] = white)
                        swap(keys[index], keys[indexWhite])
                        decrement index and indexWhite by 1
```

In this algorithm if the key is white, swap the index with white so that white is in its right position. If the key is red, swap the index with red so that red is in its right position. So, blue is automatically rearranged in the middle of red and white.