1: [20 points] Using the quicksort algorithm in the textbook (pages 60 – 66), show how to sort the following list of integers.

1, 2, 3, 4, 5, 6, 7

(a) [10 points] Use the first element in a (sub)list as the pivot item. Show the sorting actions step by step.

[n]: n is the pivot.
Step:
1: [1] | 2 3 4 5 6 7
2: [1] | [2] | 3 4 5 6 7
3: [1] | [2] | [3] | 4 5 6 7
4: [1] | [2] | [3] | 4 | 5 6 7
7: return {6, 7}
8: return {5, 6, 7}
9: return {4, 5, 6, 7}
10: return {3, 4, 5, 6, 7}
11: return {2, 3, 4, 5, 6, 7}
12: return {1, 2, 3, 4, 5, 6, 7}.

(b) [10 points] Randomly choose an element in a (sub)list as the pivot item. Show the sorting actions step by step.

The following is a sample execution sequence where [n] is the pivot.
Step
1: 1 2 3 | 4 | 5 6 7 //Partition the list to 2 sublists: {1, 2, 3} and {5, 6, 7}.
2: 1 [2] 3 | [4] | 5 6 | [7] //Use 2 as the pivot in {1, 2, 3} and use {7} as the pivot in {5, 6, 7}.
4: Return {5, 6}.
5: Return {5, 6, 7}.
6: Return {1, 2, 3}.
7: Return {1, 2, 3, 4, 5, 6, 7}.
2: [20 points] Do Exercise 2.6.

ternarySearch(x, A, left, right) //x: item to search for, A: array
{
    if (right < left)
        return "x not found"
    else if (x == A[(left+right)/3])
        return (left+right)/3;
    else if (x == A[2(left+right)/3])
        return 2(left+right)/3;
    else if (x < A[(left+right)/3])
        ternarySearch(x, a, 1, (left+right)/3);
    else if (A[(left+right)/3] ≤ x ≤ A[2(left+right)/3])
        ternarySearch(x, a, (left+right)/3, 2(left+right)/3);
    else if (x > A[2(left+right)/3])
        ternarySearch(x, a, 2(left+right)/3, right);
}

Thus, the time complexity of this algorithm is expressed as follows:

\[ T(n) = 1 \text{ when } n = 1 \]
\[ T(n) = T(n/3) + 1 \text{ when } n > 1 \]

Assume that n is a power of 3. (Dealing with a case in which n is not a power of 3 is trivial, similar to binary search.) A complete time complexity analysis follows:

\[ T(n) = T(n/3) + 1 \]
\[ = T(n/9) + 2 \]
\[ = T(n/27) + 3 \]
\[ \vdots \]
\[ = T(1) + \log_3 n \]
\[ = \Theta(\log_3 n) \]

3: [10 points] Do Exercise 2.11.

**Key Idea:** Non-recursive variant of mergesort: mergesort in a bottom-up way.

/*
The array is sorted by a sequence of passes. In each pass, the array consists of blocks of size m. Initially, m=1; that is, we have n blocks where each block has a single array element. In one pass, every two adjacent blocks are merged. As a result, the next pass has to merge blocks where each block is of size 2m. Repeat this until there is only a single block of size n.
*/
Input: array a[0..n-1];
m = 1;
while m ≤ n do
{
i = 0;
while i < n-m do
{
    merge subarrays a[i...i+m-1] and a[i+m...min(i+2 × m-1,n-1)];
    i = i + 2 × m;
}
m = m × 2;
}

4: [10 points] The behavior of a recursive algorithm is expressed by the following recurrence relations.

\[
T(n) = 7T(n/5) + 10n \text{ for } n > 1 \\
T(1) = 1
\]

(a) [5 points] Find T(125).

\[
T(125) = 7 \times T(25) + 10 \times 125 \\
= 7 \times T(25) + 1250 \\
T(25) = 7 \times T(5) + 10 \times 25 \\
= 7 \times T(5) + 250 \\
T(5) = 7 \times T(1) + 10 \times 5 \\
= 57
\]

Therefore,

\[
T(125) = 7 \times T(25) + 1250 \\
= 7 \times 649 + 1250 \\
= 5793
\]

(b) [5 points] What is the time complexity of this algorithm? [Hint: Refer to Theorem B.5 in Appendix B.]

By Theorem B.5, we get a=7, b=5, and k=1. Because \(7 > 5^1\), \(T(n) \in \Theta(n^{\log_5 7})\).

5: [10 points] Do Exercise 2.7.

\[
\text{max(left, right, A)} \\
\{
\]
If the size of array A is 1, simply return the only element in the array.

Else, return the maximum of \(\{\max(\text{left}, (\text{left+right})/2), \max((\text{left+right})/2, \text{right})\}\}

This algorithm makes recursive calls until it gets \(n\) arrays where each array has a single element. Given \(n\) input data, it takes the algorithm \(\Theta(\log n)\) time to reach this stage. (Assume that \(n\) is a power of 2.) The algorithm then returns the maximum of the two nearby subsequences until it finds the maximum for all \(n\) numbers in the original array. Thus, the total number of comparisons for the merge is:

\[
n/2 + n/4 + n/8 + \ldots + 1 = n(1/2 + 1/4 + \ldots + 1/n) = n \cdot \frac{1/2 \cdot (1 - (1/2)^{\log n})}{1 - 2} = n \cdot (1 - 1/n) = \Theta(n).
\]

Note: \(r + r^2 + \ldots + r^n = \frac{r(1-r^n)}{1-r}\).

6: [10 points] You need to write a program to compute the \(n^{th}\) Fibonacci term for input \(n\), which is a positive integer greater than 1. Answer the following questions.

(a) [2 points] Will you use Algorithms 1.6 or 1.7?
   Algorithm 1.7.

(b) [8 points] Justify your choice in part (a).
   Algorithm 1.6 has exponential time complexity in terms of \(n\), while Algorithm 1.7 is \(\Theta(n)\).

7: [10 points] Do Exercise 1.28.

\[
T(n) = O(k^2) = O(\log^2 n).
\]

8: [10 points] Do Exercise 2.37.

```c
fact(n)
{
    if (n == 0 or n == 1) return 1;
    else return n * fact(n-1)
}
```

The time complexity is:

\[
T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n-1) + 1 & \text{if } n > 1. \end{cases}
\]
\[ T(n) = T(n - 1) + 1 \]
\[ = T(n - 2) + 2 \]
\[ = T(n - 3) + 3 \]
\[ \vdots \]
\[ = n - 1 \]

Thus, it does not have exponential time complexity.