I don’t know.

1.Describe a $\Theta(n \lg n)$-time algorithm that, given a set S of n integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x.

(SOL):
There is more than one algorithm that satisfies this problem. You should have described an algorithm and done the necessary proof for correctness, as well as proved the asymptotic running time. If your algorithm utilizes an algorithm covered in the text, you may assume it is correct and use the results of its run-time analysis.
First run merge sort on S to get the sorted array $S'=s1', s2', s3', \ldots\ldots, sn'$, where $si' \leq si+1'$ for all i. As proved in Cormen, this runs in $\Theta(n \lg n)$-time

```
SUM-X(S, x)
1 MERGE − SORT(S)
2 i ←1
3 j ←n
4 while i < j
5         do if S[i] + S[j] < x
6                   then i ←i + 1
7                   else if S[i] + S[j] > x
8                             then j ←j − 1
9                             else return S[i], S[j]
10 if i = j
11         then  return  NIL
```

Proof of correctness: Assume that the set S is sorted, so that $s1<s2<\ldots<sn$. For $1\leq p<q\leq n$, call (p, q) an x pair if $sp+sq=x$. There may be many such x pairs. Initially i = 1 and j = n. SUM-X maintains a pair (i, j) and either increases i or decreases j until it finds an x pair, or i = j. If $si+sj>x$ then j is decremented, and if $si+sj<x$ then i is incremented. If there are no x pairs in S then SUM-X terminates without finding such a pair. Assume there is at least one x pair. Let (i, j) be the first loop at which either the left index i or the right index j agrees with some x pair. Without loss of generality suppose that j = q and $i\leq p$, thus $si+sq<x$.
This means that $si+sq=x$ and i = p, or $si+sq<x$ and i is increased until i = p. Hence, SUM-X finds an x pair.
The worst-case run time of SUM-X is $O(n \lg n)$. After the set is sorted, the procedure to find the two elements that sum to x has a worst-case time of $O(n)$.
A second solution is to run merge sort to get the sorted arrays S'. Then for each element in S' compute $di=si'-x$ and do a binary search for di in S' . In the worst-case we search for n elements using binary search. The worst-case run time analysis of binary search is O(lg n). Therefore, the algorithm is $\Theta(n \lg n)$.

2-2 Correctness of bubblesort

Bubblesort is a popular, but inefficient, sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order.

```
BUBBLESORT.
1 for i <- 1 to A:length - 1
2 for j <- A:length downto i + 1
3 if A[ j ] < A[ j-1 ]
4 exchange A[ j ] with A[ j-1 ]
```

a. Let A' denote the output of BUBBLESORT(A) To prove that BUBBLESORT is correct, we need to prove that it terminates and that

$A'1\leq A'2\leq\ldots\leq A'[n]$

where n = A.*length*. In order to show that BUBBLESORT actually sorts, what else do we need to prove?

The next two parts will prove inequality (2.3).

b. State precisely a loop invariant for the **for** loop in lines 2–4, and prove that this loop invariant holds. Your proof should use the structure of the loop invariant proof presented in this chapter.

c. Using the termination condition of the loop invariant proved in part (b), state a loop invariant for the **for** loop in lines 1–4 that will allow you to prove inequality (2.3). Your proof should use the structure of the loop invariant proof presented in this chapter.

d. What is the worst-case running time of bubblesort? How does it compare to therunning time of insertion sort?

(SOL):

a. We need to show that the elements A' form a permutation of the elements of A

b. **Loop invariant:** At the start of each iteration of the for loop of lines 2-4, A[j] = min { A[k] : $j \le k \le n$} and the subarray A[j .. n] is a permutation of the values that were in A[j .. n] at the time that the loop started.
Initialization: Initially, j = n, and the subarray A[j .. n] consists of single element A[n]. The loop invariant trivially holds.
Maintenance: Consider an iteration for a given value of j. By the loop invariant, A[j] is the smallest value in A[j .. n]. Lines 3-4 exchange A[j] and A[j-1] if A[j] is less thanA[j-1], and so A[j-1] will be the smallest value in A[j–1 .. n] afterward. Since the only change to the subaaray A[j-1 .. n] is this possible exchange, and the subarray A[j..n] is a permutation of the values that were in A[j .. n] at the time that the loop started, we see that A[j-1 .. n] is a permutation of the values that were in A[j-1 .. n] at the time that the loop started. Decrementing j for the next iteration maintains the invariant.
Termination: The loop terminates when j reaches i. By the statement of the loop invariant, A[i] = min {A[k] : $I \le k \le n$} and A[i .. n] is a permutation of the values that were in A[i .. n] at the time that loop started.

c. Using the termination condition of the loop invariant proved in part (b), state a loop invariant for the **for** loop in lines 1-4 that will allow you to prove inequality (1). Your proof should use the structure of the loop invariant proof presented in this chapter.
Loop invariant: At the start of each iteration of the **for** loop of lines 1-4, the subarray A[1 .. i-1] consists of the i-1 smallest values originally in A[1..n], in sorted order, and A[i..n] consists of the n-i+1 remaining values originally in A[1..n].
Initialization: Before the first iteration of the loop, I = 1. The subarray A[1..i-1] is empty, and so the loop invariant vacuously holds.
Maintenance: Consider an iteration for a given value of i. By the loop invariant, A[1.. i-1] consists of the i smallest values in A[1..n], in sorted order. Part(b) showed that after executing the **for** loop of lines 2-4, A[i] is the smallest value in A[i..n], and so A[1..i] is now the I smallest values originally in A[1..n], in sorted order. Moreover, since the **for** loop of lines 2-4 permutes A[i..n] , the subarray A[i+1 .. n] consists of the n – I remaining values originally in A[1.. n]
Termination: The **for** loop of lines 1-4 terminates when i = n + 1, so that i – 1 = n. By the statement of the loop invariant, A[1.. i-1] is the entire array A[1.. n], and it consists of the original array A[1..n], in sorted order.

d. The running time depends on the number of iterations of the for loop of lines 2-4. For a given value of I, this loop makes n – i iterations, and i takes on the values 1,2,…,n. The total number of iterations, therefore, is

$\sum_{i=1}^{n} n-i = \sum_{i=1}^{n} n-i = \sum_{i=1}^{n} n-i = \sum_{i=1}^{n} i = n^2-\frac{n}{2}+\frac{1}{2} = \frac{n^2}{2}-\frac{n}{2}+\frac{1}{2} = \frac{n^2}{2}-\frac{n}{2}$

Thus, the running time of bubblesort is $\Theta(n^2)$ in all cases. The worst-case running time is the same as that of insertion sort

3.1-1.

Let f(n) and g(n) be asymptotically nonnegative functions. Using the basic definition of Θ-notation, prove that $\max(f(n),g(n)) = \Theta(f(n)+g(n))$

(SOL):

Show that $\max(f(n),g(n)) = \Theta(f(n)+g(n))$.

This is true iff $\exists\, c1,c2,n0>0$ such that

$$0 \le c1(f(n)+g(n)) \le \max(f(n),g(n)) \le c2(f(n)+g(n))$$

For all $n \ge n0$

Choose for instance $c1=12$ two situations:

1.$f(n) > g(n)$ $\Rightarrow \frac{1}{2}(f(n)+g(n)) < \frac{1}{2}(f(n)+f(n)) = f(n) = \max(f(n), g(n))$

2.$f(n) < g(n)$ $\Rightarrow \frac{1}{2}(f(n)+g(n)) < \frac{1}{2}(g(n)+g(n)) = f(n) = \max(f(n), g(n))$

Choose for instance $c2=1 \Rightarrow \max(f(n), g(n)) \le f(n) + g(n)$.

So, choose $c1=12$, $c2=1$ and the statement holds.

其餘:參考課本解法