Name-Surname: ……………………………………………………………………………………………… ID: ………………………………………

13016239 ALGORITHM DESIGN AND ANALYSIS
MIDTERM EXAMINATION 2018/2

Sample Solutions

International College
King Mongkut's Institute of Technology Ladkrabang

DIRECTIONS:

- No books, documents, or electronic devices are allowed.
- Please write your answers in the accompanied answer book. There is no need to write the question statements in the answer book, but please clearly indicate in front of each of your answers which question the answer is for.
- Anything written in this exam paper will not be considered part of your answer.
- Before submission, please check that you have written you name and student ID clearly on all your answer booklets and also on this exam paper.

PROBLEM 1 (16 PTS)

Prove or disprove each of the following statements.

1.1 $3n + 10 \in O(n)$

Ans. To prove this, we need to show that there exist positive real c and non-negative integer N such that

$$3n + 10 \le cn \text{ for all } n \ge N \qquad (1)$$

Clearly, this inequality could be true only if $c > 3$. Let's set c to 4 and try to find N that makes the above inequality holds. We have

$$3n + 10 \le 4n \text{ if}$$
$$10 \le 4n - 3n \text{ if}$$
$$10 \le n.$$

This implies that $3n + 10 \le 4n$ for all $n \ge 10$. So (1) holds when $c = 4$ and $N = 10$. Therefore, $3n + 10 \in O(n)$.

1.2 $3n + 10 \in O(n^2)$

Ans 1. To prove this, we need to show that there exist positive real c and non-negative integer N such that

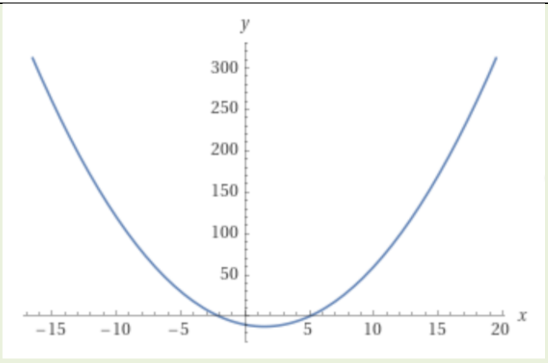
$$3n + 10 \le cn^2 \text{ for all } n \ge N \qquad (1)$$

It's quite clear that the RHS, which is a polynomial of degree 2, grows faster the LHS, which is a linear function. So, no matter what c is, as long as it is positive, we should be able to find N that makes the inequality holds.

Let us set c to be 1 and try to find N that makes (1) holds. We have

$$3n + 10 \le n^2 \text{ if}$$
$$0 \le n^2 - 3n - 10 \text{ if}$$
$$0 \le (n - 5)(n + 2).$$

When we plot the polynomial $(n - 5)(n + 2)$, we can see that it is a parabola which opens upward and intersects the x-axis at $x = -2$ and $x = 5$.



It can be seen from the graph that $(n-5)(n+2) \geq 0$ if $n \geq 5$. Hence, we can conclude that

$$3n + 10 \leq n^2 \text{ for all } n \geq 5$$

which implies that $3n + 10 \in O(n^2)$.

Ans 2. Since, from Question 1.1, we have shown that $3n + 10 \in O(n)$. To show 1.2, we can instead show that $O(n) \subseteq O(n^2)$.
Let $f(n)$ be any function in $O(n)$. By the definition of O, there exist positive real c and non-negative integer N such that

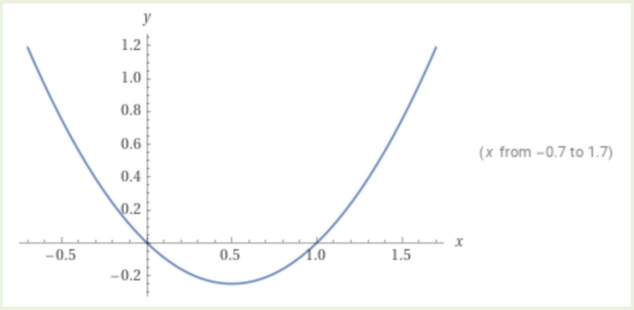
$$f(n) \leq cn \text{ for all } n \geq N \qquad (1)$$

Since it can be shown that $n \leq n^2$ when $n \geq 1$, we have $cn \leq cn^2$ when $n \geq 1$. Substituting this into (1), we have

$$f(n) \leq cn^2 \text{ for all } n \geq M,$$

where $M = \max(1, N)$. This implies that $f(n) \in O(n^2)$. Therefore, $O(n) \subseteq O(n^2)$.

Note: How do we show that $n \leq n^2$ when $n \geq 1$? One way is to show that, if $n \geq 1$, then $n^2 - n \geq 0$. If we plot the graph of the function $n^2 - n$, we can see that it is a parabola opening upward, intersecting the x-axis at $x = 0$ and $x = 1$.



It can be seen from the graph that if $n \geq 1$, then $n^2 - n \geq 0$, which is equivalent to $n \leq n^2$.

1.3 $\log_3(n^2) \in \theta(\log_2(n^3))$

Ans. To prove this, we need to show that there exist positive reals c and d and non-negative integer N such that

$$c\log_2(n^3) \leq \log_3(n^2) \leq d\log_2(n^3) \text{ for all } n \geq N \qquad (1)$$

First, let is change the base of the middle term to 2. We make use of the following rules of log: For any positive integers a and b and any positive real x

$$\log_a x = \frac{\log_b x}{\log_b a}$$

$$\log_b(x^a) = a\log_b x$$

Applying these rules on (1), we have

$$3c\log_2(n) \le \left(\frac{2}{\log_2 3}\right)\log_2 n \le 3d\log_2(n)$$

$$c\log_2(n) \le \left(\frac{2}{3\log_2 3}\right)\log_2 n \le d\log_2(n)$$

This inequality will hold for all positive integer n when

$$c \le \frac{2}{3\log_2 3} \text{ and } \frac{2}{3\log_2 3} \le d$$

We can thus let c and d be $\frac{2}{3\log_2 3}$ (which is a positive real) and the inequality (1) will hold for all $n \ge 1$ (i.e. $N = 1$).

1.4 $n\log^2 n \in O(n^2)$

Note:

- $\log^2 n$ means $(\log n)^2$, which is generally <u>not</u> equal to $\log(n^2)$.
- In this course,

$$\log n \text{ means } \log_{10} n$$
$$\lg n \text{ means } \log_2 n$$
$$\ln n \text{ means } \log_e n$$

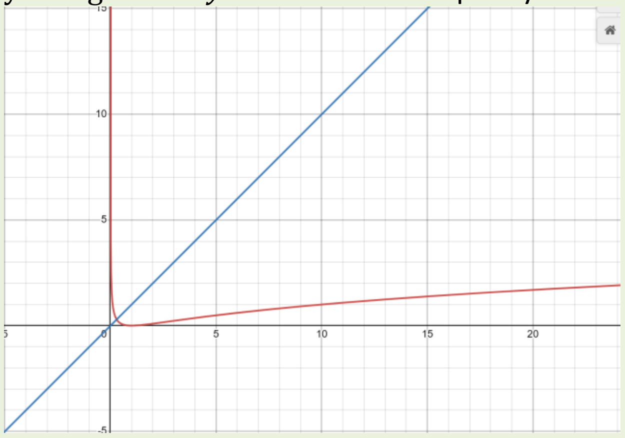
Ans. To prove this, we need to show that there exist positive real c and non-negative integer N such that

$$n\log^2 n \le cn^2 \quad \text{for all } n \ge N \qquad (1)$$

Let us try $c = 1$. We shall find a positive integer N that makes (1) holds. First, divide n on both sides, we obtain

$$\log^2 n \le n \qquad (2)$$

It is clear from the graphs of $y = \log^2 x$ and $y = x$ that this inequality holds when $n \ge 1$.



To prove this formally, we can use techniques from calculus. When $n \ge 1$, the inequality (2) holds if $\log n \le \sqrt{n}$. We shall show this latter inequality instead. Define

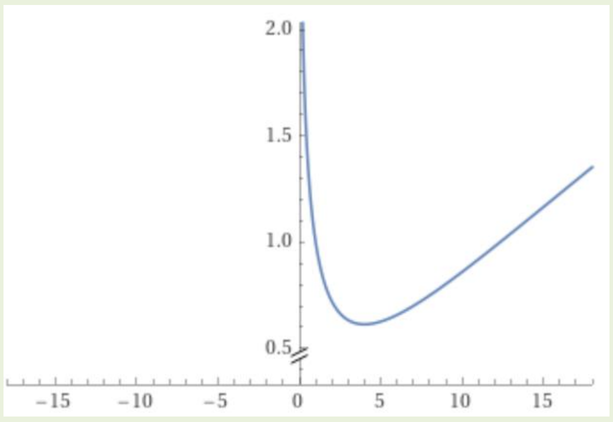
$$g(x) = \sqrt{x} - \log x$$

We shall show that $g(x)$ is positive for all sufficiently large value of x. We first find the positive values of x where $g(x)$ is increasing, which is the case when its derivative $g'(x)$ is non-negative, i.e.

$$0 \le g'(x)$$
$$0 \le \frac{1}{2\sqrt{x}} - \frac{1}{x}$$
$$\frac{1}{x} \le \frac{1}{2\sqrt{x}}$$
$$2 \le \sqrt{x}$$

$$4 \le x$$

This tells us that $g(x)$ is increasing when $x \ge 4$ and decreasing when $0 < x \le 4$. Hence, when x is positive, $g(x)$ has the minimum at $x = 4$. Since $g(4) = \sqrt{4} - \log 4$ is positive, it follows that $g(x)$ is positive for all $x > 0$.



Therefore, we can conclude that $\log n \le \sqrt{n}$ and, consequently, $\log^2 n \le n$, for all $n \ge 1$. In other words, the inequality (1) holds when $c = 1$ and $N = 1$.

Problem 2 (8 pts)

Find closed-form solutions to the following recurrence equations. You may express your solutions using Big-Θ notation. You may apply the Master Theorem.

2.1 Let *g* be the function defined for all non-negative integers as follows:

$$g(0) = 0, g(1) = 1$$
$$g(n) = 3g\left(\left\lfloor\frac{n}{3}\right\rfloor\right) + n, \quad n > 1$$

Ans. We can apply the Master Theorem. We can see that $a = 3$ and $b = 3$. Since $n = \Theta(n^1)$, we have $c = 1$. We find $\log_b a = \log_3 3 = 1$, which equals to c. This means we need to apply Case 2 of the Master Theorem. Therefore, we have $g(n) = \Theta(n^c \log n) = \Theta(n \log n)$.

2.2 Let *h* be the function defined for all non-negative integers as follows:

$$h(0) = 0, h(1) = 1$$
$$h(n) = 3h\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + n^2 + 2n, \quad n > 1$$

Ans. We can apply the Master Theorem. We can see that $a = 3$ and $b = 2$. Since $n^2 + 2n = \Theta(n^2)$, we have $c = 2$. We compute $\log_b a = \log_2 3$, which is smaller than $c = 2$. This means we need to apply Case 1 of the Master Theorem. Therefore, we have $g(n) = \Theta(n^c) = \Theta(n^2)$.

Problem 3 (4 pts)

Analyze the time complexity of the following pseudocode fragment, where m ≥ 0 is the size of the input. Describe its running time using Big-Θ notation.

```
s = 0
for(i = 0 to m)
    for(j = 0 to i)
        s = s + i*j
```

$\rightarrow \Theta(1)$

$\rightarrow \Theta(1)$

$$\sum_{j=0}^{i} \Theta(1) = \Theta\left(\sum_{j=0}^{i} 1\right) = \Theta(i+1) = \Theta(i)$$

$$\sum_{i=0}^{m} \Theta(i) = \Theta\left(\sum_{i=0}^{m} i\right) = \Theta\left(\frac{m(m+1)}{2}\right) = \Theta\left(\frac{m^2}{2}+\frac{m}{2}\right) = \Theta(m^2)$$

$$\Theta(1)+\Theta(m^2) = \Theta(1+m^2) = \Theta(m^2)$$

#

Problem 4 (4 pts)

Analyze the time complexity of the following pseudocode fragment, where m ≥ 1 is the size of the input, assuming that m is a power of 2. Describe its running time using Big-Θ notations.

```
j = m; s = 0;
while(j > 0) {
    for(i = 0 to j)
        s = s + 1
    j = ⌊j/2⌋
}
```

Ans. First, we find the number of iterations of the while loop. Let n be the number of iterations. Since n depends on the initial value of j, which equals to m, we need to find the relationship between n and m. To do so, let us consider the value of j before and after each iteration. To simplify the analysis, we assume that m is a power of 2.

Iteration k	Value of j **before** iteration k	Value of j **after** iteration k
1	m	$\frac{m}{2^1}$
2	$\frac{m}{2^1}$	$\frac{m}{2^2}$
3	$\frac{m}{2^2}$	$\frac{m}{2^3}$
…		…
$\lg m$	$\frac{m}{2^{\lg m-1}}$	$\frac{m}{2^{\lg m}} = 1$
$(\lg m)+1$	$\frac{m}{2^{\lg m}} = 1$	$\left\lfloor \frac{1}{2} \right\rfloor = 0$

We can see that the number n of iterations is $(\lg m) + 1$. We can then analyze the time complexity of the program. Note that the running time of each iteration of the while loop depends on the value of j before each iteration.

```
j = m; s = 0;
while(j > 0) {
    for(i = 0 to j)
        s = s + 1
    j = ⌊j/2⌋
}
```

$\Theta(1)$

$\sum_{i=0}^{j} \Theta(1) = \Theta(\sum_{i=0}^{j} 1)$

$= \Theta(j+1)$

$= \Theta(j)$

$\Theta(1)$

$\Theta(j) + \Theta(1)$

$= \Theta(j+1)$

$= \Theta(j)$

$\Theta(1)$

$\sum_{k=1}^{n} \Theta(\frac{m}{2^{k-1}})$

$= \Theta(m)$

(see below)

$\Theta(1) + \Theta(m)$

$= \Theta(1+m)$

$= \Theta(m)$

#

Here's the simplification of $\sum_{k=1}^{n}\Theta(\frac{m}{2^{k-1}})$.

$$
\begin{aligned}
\sum_{k=1}^{n}\Theta\left(\frac{m}{2^{k-1}}\right) &= \Theta\left(\sum_{k=1}^{n}\frac{m}{2^{k-1}}\right)\\
&= \Theta\left(\sum_{k=0}^{n-1}\frac{m}{2^{k}}\right) = \Theta\left(m\sum_{k=0}^{n-1}\frac{1}{2^{k}}\right)\\
&= \Theta\left(m\left(\frac{\frac{1}{2^{n}}-1}{\frac{1}{2}-1}\right)\right) = \Theta\left(2m-\frac{m}{2^{n-1}})\right)\\
&= \Theta\left(2m-\frac{m}{2^{\lg m}}\right) \qquad\qquad \text{since } n = (\lg m)+1\\
&= \Theta\left(2m-\frac{m}{m}\right) = \Theta(2m-1) = \Theta(m).
\end{aligned}
$$

Problem 5 (8 pts)

Consider the pseudocode description of the function FunA(A) given below.

5.1 Analyze the best-case time complexity of FunA(A) in terms of the size of the given array A. What are the input arrays A in the best-case scenario?

5.2 Analyze the worst-case time complexity of FunA(A) in terms of the size of the given array A. What are the input arrays A in the worst-case scenario?

```
FunA(A[1…n]) {
    return FunA(A, 1)
}

FunA(A[1…n], i) {
    if (i > n) return False
    if (A[i] == 0)
        return True
    else
        return FunA(A[1…n], i+1)
}
```

Ans. From the program, we can see that the function FunA(A,i) tries check whether 0 occurs in A at index i or later. Hence, the function FunA(A), which calls FunA(A,1), check whether 0 occurs in A or not.

5.1 The best-case scenario of FunA(A) happens when A[1] == 0, i.e. the first item in A is 0. In this case the function FunA(A, 1) which is called by FunA(A) will terminate on Line 8 and will not make a recursive call. In this case, the time that FunA(A, 1) (and, hence, FunA(A)) spends is constant (i.e. not depending on n). Therefore, the best-case time complexity of FunA(A) is $\Theta(1)$.

5.2 The worst-case scenario happens when the function FunA(A,1) terminates on Line 6, which happen when the array A does not contain 0. In this case,

- Fun(A) calls FunA(A,1)
- FunA(A,1) calls FunA(A,2)
- FunA(A,2) calls FunA(A,3)
- …
- FunA(A,n-1) calls FunA(A,n)
- FunA(A,n) calls FunA(A,n+1)
- FunA(A,n+1) terminates on Line 6.

Since each step above takes constant time and there are $n + 2$ steps. Therefore, the worst-case time complexity is $\Theta(n)$.

Problem 6 (8 pts)

Consider the pseudocode description of the function `FunB(A)` given below.

6.1 Analyze the worst-case time complexity of `FunB(A)` in terms of the size of the given array A. Describe its running time using Big-Θ notation. To simplify the analysis, assume that the array size, n, is a power of 2 (n = 2^m, for some non-negative integer m).

6.2 Give an alternative implementation of `FunB(A)` that does not use recursion.

```
FunB(A[1…n]) {
    return FunB(A, 1, n)
}

FunB(A[1…n], i, j) {
    if (i == j) return A[i]
    d = $\lfloor (j-i+1)/2 \rfloor$
    x = FunB(A, i, i+d-1)
    y = FunB(A, i+d, j)
    if(x > y)
        return x
    else
        return y
}
```

Ans.

6.1 Let $T(m)$ be the worst-case time complexity of `FunB(A,i,j)` where m = j-i+1.

m=1 when i=j. In this case, FunB takes constant time (Line 6), hence $T(1) = \Theta(1)$.

For m>1, FunB(A, i, j) makes two recursive calls to FunB(A, i, i+d-1) and FunB(A, i+d, j), where d = $\lfloor (j-i+1)/2 \rfloor = \lfloor m/2 \rfloor$.

- The worst-case time complexity of the first call FunB(A, i, i+d-1) is
$$T\big((i+d-1)-i+1\big) = T(d) = T(\lfloor m/2 \rfloor)$$
- The worst-case time complexity of the second call FunB(A, i+d, j) is
$$T(j-(i+d)+1) = T((j-i+1)-d) = T(m-d) = T(m-\lfloor m/2 \rfloor) = T(\lceil m/2 \rceil)$$

The other instructions in FunB(A, i, j) take constant time. Hence, we have

$$T(m) = T\left(\left\lfloor \frac{m}{2} \right\rfloor\right) + T\left(\left\lceil \frac{m}{2} \right\rceil\right) + \Theta(1).$$

In case m is a power of 2, this equation can be simplified to

$$T(m) = 2T\left(\frac{m}{2}\right) + \Theta(1).$$

Applying Case 3 the Master Theorem ($a = b = 2$ and $c = 0$ since $\Theta(1) = \Theta(n^0)$), we have $T(m) = \Theta\big(m^{\log_b a}\big) = \Theta\big(m^{\log_2 2}\big) = \Theta(m)$.

Since FunB(A) calls FunB(A, 1, n), where n is the size of array A, the worst-case time complexity of FunB(A) is $T(n-1+1) = T(n) = \Theta(n)$.

6.2 From the code of program, we can see that the function `FunB(A,i,j)` tries to find the greatest element between index i and index j (inclusively). Thus, FunB(A), which calls FunB(A,1,n), tries to find the greatest element in array A. We can rewrite this function using a loop to find the greatest element instead.

```
FunB(A[1…n]) {
    max_so_far = A[1]
    for i from 2 to n:
       if A[i]>max_so_far
         max_so_far = A[i]
    return max_so_far
}
```

Problem 7 (8 pts)

Suppose you are given an array A[1…n] (where n ≥ 3) containing n integers. For simplicity, assume all the integers in A are distinct, i.e. no number occurs twice in the array. You are told that the sequence of values A[1], A[2],...,A[n] is *unimodal*: There exists an index p, called the "*peak index*", where 1 < p < n, such that the values in the array entries increase up to position p in A and then decrease the remainder of the way until position n.

For example, the array A[1…7] = [3,15,23,24,48,35,29] is unimodal with the peak index p = 5.

7.1 Provide a pseudo-code description of an algorithm for the function `findPeak(A)` which, given a non-empty unimodal array A containing 3 or more distinct integers, returns the peak index of A. The function should have the worst-case running time of O(log n).

7.2 Give a worst-case time complexity analysis of your algorithm.

Ans.

7.1 We modify the binary-search algorithm slightly to find the peak index. Suppose `find_peak_index(A,lo,hi)` is supposed to find the peak index in array A between index lo to index hi (inclusively). The idea is, when considering A[mid], where mid is the middle index, i.e. $\lfloor (\text{lo} + \text{hi})/2 \rfloor$, we consider the three possibilities.

a) A[mid-1]<A[mid] and A[mid]>A[mid+1]. In this case, A[mid] is the peak. We can just return mid as the peak index.
b) A[mid-1]<A[mid]<A[mid+1]. The peak must be to the right of mid (i.e. the peak could be at mid+1 or further right). In this case, we should make a recursive call to find the peak index between mid and hi.
c) A[mid-1]>A[mid]>A[mid+1]. The peak must be to the left of mid (i.e. the peak could be at mid-1 or further left). In this case, we should make a recursive call to find the peak index between lo and mid.

Note that the case where A[mid-1]>A[mid] and A[mid]<A[mid+1], i.e. A[mid] is the bottom of a valley, will never happen because the given array A is assumed to be unimodal.

Here is the pseudo-code.

```
find_peak_index(A[1…n]) {
    return find_peak_index(A, 1, n)
}

find_peak_index(A[1…n], lo, hi) {
    if (lo>hi) return None
    mid = ⌊(lo + hi)/2⌋
    if A[mid-1]<A[mid] and A[mid]>A[mid+1]
        return mid
    else if A[mid-1]<A[mid]<A[mid+1]
        return find_peak_index(mid,hi)
    else
        return find_peak_index(lo,mid)
}
```

7.2 The worst-case time complexity is similar to that of the binary search. Let $T(m)$ be the worst-case time complexity of `find_peak_index(A, lo, hi)`, where m=hi-lo+1. From the program, we can write the recurrence equation describing $T(m)$ as follows.

$$T(0) = T(1) = \Theta(1)$$
$$T(m) = T\left(\left\lceil\frac{m}{2}\right\rceil\right) + \Theta(1) \qquad \text{where } m > 1$$

Applying Case 2 the Master Theorem ($a = 1, b = 2$ and $c = 0$ since $\Theta(1) = \Theta(n^0)$), we have $T(m) = \Theta(n^c \log n) = \Theta(n^0\ \log m) = \Theta(\log m)$.

Since `find_peak_index(A)` calls `find_peak_index(A, 1, n)`, where n is the size of array A, the worst-case time complexity of `find_peak_index(A)` is $T(n - 1 + 1) = T(n) = \Theta(\log n)$.

PROBLEM 8 (8 PTS)

You are asked to design an algorithm for a function

```
Boolean findSum(float[] A[1…n], float s)
```

which, given an array A of distinct numbers (possibly non-integers) and a number s, determines whether there are two distinct numbers in the array whose sum equals to s. If so, the function returns True; otherwise, it returns False.

For example, if A = [17,24,6,18,25,3] and s = 27, then `findSum` should return True because there are two distinct numbers in A, namely, 24 and 3, whose sum is 27. On the other hand, if A = [17,24,6,18,25,3] and s = 12, then `findSum` should return False.

8.1 Provide a pseudo-code description of your algorithm for `findSum` which has the worst-case running time of O(n log n), where n is the size of the given array.

8.2 Give a worst-case time complexity analysis of your algorithm.

Ans.

8.1 A brute-force algorithm is to try checking the sum of every pair of elements, A[i]+A[j], where $1 \le i < j \le n$, to find a pair whose sum exactly equals s. Clearly, this brute-force algorithm takes $\Theta(n^2)$ time, which is not in $O(n \log n)$.

A more efficient approach is to first sort the given array A in increasing order. Once sorted, we travese the array using two index variables, say i and j, where initially i is at leftmost end ($i = 1$) and j is at the rightmost end ($j = n$). The algorithm tries to find two elements between index i and index j (inclusively) whose sum equals to s. To do so, in each iteration, we compute A[i]+A[j] and consider the following possibilities.

- A[i]+A[j] = s. We found the pair whose sum equals s, so return True.
- A[i]+A[j] > s. This means that A[i]+A[j] is too large. And since every element to the right of j is larger than A[j], all the sums
  A[i]+A[j], A[i]+A[j+1], A[i]+A[j+2], …, A[i]+A[n]
  are too large. Moreover, for any k to the right of i, since A[k]>A[i], all the sums
  A[k]+A[j], A[k]+A[j+1], A[k]+A[j+2], …, A[k]+A[n]
  will also be too large too. Hence, we do not need to consider A[j], A[j+1], …A[n] anymore because they cannot be matched with any earlier element to get the sum s. So, we decrease the index variable j by one.
- A[i]+A[j] < s. This means that A[i]+A[j] is too small. And since every element to the left of i is smaller than A[j], all the sums
  A[1]+A[j], A[2]+A[j], …, A[i]+A[j]
  are too small. Moreover, for any k to the left of j, since A[k]<A[j], all the sums

A[1]+A[k], A[2]+A[k], …, A[i]+A[k]

will also be too small too. Hence, we do not need to consider A[1], A[2], …, A[i] anymore because they cannot be matched with any later element to get the sum s. So, we increase the index variable i by one.

Here is the pseudocode.

```
 1: findSum(A[1…n], s) {
 2:     merge-sort(A)
 3:     i = 1
 4:     j = n
 5:     while i<j
 6:        if A[i]+A[j]==s
 7:            return True
 8:        else if A[i]+A[j]>s
 9:            j = j-1
10:        else
11:            i = i+1
12:     return False
```

8.2 Let us analyze the worst-case time complexity of this algorithm. Suppose the given array A has n elements. Merge sorting array A takes $\Theta(n \log n)$ time in the worst case. Lines 3, 4, and 12 use constant time.

How many iterations does the while loop take? Observe that before the while loop starts, $j - i = n - 1$. After each iteration of the while loop before the last iteration, either j is decreased by 1 or i is increased by 1. Hence, $j - i$ is always reduced by 1. In the worst case, these repeats until i and j become equal, i.e. $j - i = 0$. Therefore, the maximum number of iterations in the while loop is $n - 1$. This happens when there are no two elements in the area whose sum equals to s. Since each iteration in the while loop also uses constant time, the while loop takes $(n-1)\Theta(1) = \Theta(n)$ time in the worst case.

Therefore, the worst-case time complexity of this algorithm is $\Theta(n \log n) + \Theta(n) + \Theta(1) = \Theta(n \log n)$.

This is the end of the exam paper.