Solutions to Workshop Test 1: Algorithm Analysis (Total Marks: 10)

(for reference only)

Q1.(1 mark)

Solution:

Option 1 – (use of inequality):

Since $1024<1050<2048$

we have $\log_2(1024) < \log_2(1050) < \log_2(2048)$,

then $10 = \log_2(2^{10}) = \log_2(1024) < \log_2(1050) < \log_2(2048) = \log_2(2^{11}) = 11$

or $10 < \log_2(1050) < 11$

therefore we have floor($\log_2(1050)$) =10 and ceiling($\log_2(1050)$) = 11.

Option 2: (by halving the value – the number of times of halving is floor/ceiling(x)).

Since 1050/2=525,
525/2 = 262,
262/2 = 131,
131/2 = 65
65/2 =32
32/2 = 16,
16/2 = 8,
8/2 = 4,
4/2 = 2,
2/2 = 1,
1 /2 = 0,

Then floor($\log_2(1050)$) =10 as it takes 10 times to halve 1050 (divisions by 2) to reach 1,

and ceiling($\log_2(1050)$) = 11 as it takes 11 divisions by 2 to reach 0.

Q2. (1 mark)

Solution (for reference only):

by running the algorithm:

(initial: p = 1050, q = 588)
(p%q =>1050%588= > 462 (as 1050÷588 =1 with remainder 462) so, r = 462, p = 588, q = 462
(now p%q=> 588%462 = 126 (as 588÷462 =1 with remainder126) so, r = 126, p = 462, q = 126
(now p%q=> 462 %126 = 84 (as 462÷126 =3 with remainder 84) so, r = 84, p = 126, q = 84
(now p%q=>126%84 = 42 so, r = 42, p = 84, q = 42
(now p%q=>84%42 = 0
Therefore, GCD = q = 42

Q3 (2 mark)

Solution (for reference only):

$O(776\times n^2 \times log_2(n) + 3.1 \times n^3 + 8\times n^2 + 30\times n^{2/3} + 850)$

$\Rightarrow \max \{O(776\times n^2 \times \log_2(n)), O(3.1 \times n^3), O(8 \times n^2), O(30 \times n^{2/3}), O(850)\}$,

$\Rightarrow \max \{O(n^2 \times \log_2(n)), O(n^3), O(n^2), O(n^{2/3}), O(1)\}$,

$= O(n^3)$

Therefore, the time complexity of the algorithm is $O(n^3)$.

(Notes: $n^{2/3} = \sqrt[3]{n^2} \neq n^2 / n^3 = n^{-1} = \frac{1}{n}$)

Q4 (2 marks)

Solutions (for reference only):

Method 1: (step-by-step analysis to the whole algorithm)

`int example(int[] array)            //line 01`	O(1)
`{ if (array==null) return 0;        //line 02`	O(1)
`  int n=array.length;               //line 03`	O(1)
`  if (n==0) return 0;               //line 04`	O(1)
`  int maximum=array[0];             //line 05`	O(1)
`  int minimum=array[0];             //line 06`	O(1)
`  for (int i=1; i<n; i++)           //line 07`	O(1) * #e
`  { if (array[i]>maximum)           //line 08`	O(1)
`         maximum=array[i];          //line 09`	O(1)
`     if (array[i]<minimum)          //line 10`	O(1)
`         minimum=array[i];          //line 11`	O(1)
`  }                                 //line 12`	
`  return n*maximum*minimum;         //line 13`	O(1)
`}                                   //line 14`	
Loop control: i=1, , …, n-1 **#e=Number of executions = O(n)** **maximum cost: O(n)**	

Method 2: By counting the number of characteristic operations, e.g., comparisons, additions, multiplications, or copying etc. of the algorithm

Let n be the number of the elements of the array, i.e., n = array.length.
Lines 02~06 and line 13: each can be done in constant time, or $O(1)$ time.
The *for* loop in line 07 executed for n times. Inside the body of the look, lines 08~09 conducted one comparison, and lines 10 ~11 did the same too. Thus, the total number of comparisons of the for loop (in lines 07~12) is of $2*n$.
Therefore, the time complexity of the algorithm is $O(1) + O(2*n) = O(n)$.

Q5 (4 marks)

Solution (for reference only):

(1) The best search algorithm is the *binary search algorithm* because it has the best time complexity for sorted arrays and array A is already sorted. Detailed steps can be

To find which (if any) component of the sorted (sub)array $a[0\ldots n^2]$ equals **T**

1. Set $l = 0$, and set $r = n^2$.
2. While $l \leq r$, repeat:
   - 2.1. Let *index* be an integer about midway between l and r.
   - 2.2. If *target* equals *A*[*index*], terminate with answer ***index***.
   - 2.3. If *target* is less than *A*[*index*], set r = *index* –1.
   - 2.4. If *target* is greater than *A*[*index*], set l = *index* +1.
3. Terminate with answer ***–1***.

(2) **Analyse** (the time complexity of the **entire process**):

There are two parts to be considered: The first part is to read values from the file **F** and store them in array **A**. The second part is to search target value **T** from the array **A**.

1) Suppose that reading one value sequentially from **F** and then storing it in an appropriate cell of array **A** needs constant time, i.e., O(1) time. Then the total time used for reading and storing all values would be $O(n^2)$ because there are n^2 values to be read and stored.
2) The analysis to the second part of the process can be similar to the algorithm analysis to the *binary search algorithm*. We assume that steps 2.2–4 perform a single comparison.

- If the search is **unsuccessful**, these steps are repeated as often as we must halve n^2 to reach 0:
  Number of comparisons = $\text{floor}(\log_2 n^2) + 1 = \text{floor}(2 \times \log_2 n) + 1$
- If the search is **successful**, these steps are repeated at most that many times:
  Maxi number of comparisons = $\text{floor}(\log_2 n^2) + 1 = \text{floor}(2 \times \log_2 n) + 1$

In either case, the time complexity is

$$O(\text{floor}(2 \text{ x } \log_2 n) + 1) \Rightarrow \max\{O(2 \text{ x } \log_2 n), O(1)\} = O(\log_2 n)$$

Combining the above two parts, we have the total time complexity of

$$O(n^2) + O(\log_2 n) \Rightarrow \max\{O(n^2), O(\log_2 n)\} = O(n^2)$$