(3%) Give the best Big-Oh characterization for each of the following running time estimates (where n is the size of the input problem).

(a) $\log(n) + 10000$ **Answer:** $O(\log(n))$

(b) $n\log(n) + 15n + 0.002n^2$ **Answer:** $O(n^2)$

(c) $37\,n + n\log(n^2) + 5000\log(n)$ **Answer:** $O(n \log n)$

(d) $1000n^2 + 16n + 2^n$ **Answer:** $O(2^n)$

(e) $n + (n-1) + (n-2) + \cdots + 3 + 2 + 1$ **Answer:** $O(n^2)$

(f) $2^{10} + 3^5$ **Answer:** $O(1)$

(10%) For each of the following **three** algorithms, give its time complexity (in Big-Oh notation).

Algorithm `Algo1(A)`
Input: An array A storing $n \geq 1$ integers
Output: The sum of the elements in A
$s \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n-1$ **do**
  $s \leftarrow s + A[i]$
return s

$O(n)$

Algorithm `Algo2(A)`
Input: An array A storing $n \geq 1$ integers
Output: The sum of the prefix sums in A
$s \leftarrow 0$
for $i \leftarrow 1$ **to** $n-1$ **do**
  $s \leftarrow s + A[0]$
  for $j \leftarrow 1$ **to** i **do**
    $s \leftarrow s + A[j]$
return s

$O(n^2)$

Algorithm `Algo3(A, B)`
Input: Arrays A and B, each of them storing $n \geq 1$ integers
Output: The number of elements in B equal to the sum of the prefix sums in A
$c \leftarrow 0$
for $i \leftarrow 1$ **to** $n-1$ **do**
  $s \leftarrow 0$
  for $j \leftarrow 1$ **to** $n-1$ **do**
    $s \leftarrow s + A[0]$
    for $k \leftarrow 1$ **to** j **do**
      $s \leftarrow s + A[k]$
  if $B[i] = s$ **then**
    $c \leftarrow c + 1$
return c

$O(n^3)$

(8%) Use the definition of **Big-Oh** to prove that $0.01n\log(n) - 2000n + 6$ is $O(n\log(n))$.

$0.01 n \log n - 2000n + 6 \leq c n \log n$

$cn\log n - 0.01 n \log n + 2000n - 6 \geq 0$

$cn\log n - 0.01 n \log n - 6n \log n \geq 0$

$(c - 0.01 - 6) n \log n \geq 0$

$c = 7$

$n_0 = 0$

.....

True or False

$20n^3 + 10n\log n + 5$	is $O(n\log n)$ FALSE
2^{100}	is $\Theta(1)$ TRUE
$\log(n^x)$	is $O(\log n)$ *where* $x > 0$ *is const* TRUE
$500\log^5 n + n + 10$	is $O(n)$ TRUE
$0.5\log n$	is $\Theta(n)$ FALSE

Order the following list of functions by the big-Oh asymptotic growth rate

$$6\log n,\ \log\log n,\ 2^{\log n},\ n\sqrt{n},\ n^2$$

The functions are in increasing complexity order (i.e., lowest to highest):

$$\log\log n,\ 6\log n,\ \ n\sqrt{n},\ n^2,\ 2^{\log n}.$$

Express as a function of the input size n the worst-case running time $T(n)$ of the following algorithm

```
int Me(int n, int A[], int k)
{
    int tot = 0;

    for (int i=n; i>=0; i--) {
       if (i==k) {
          cout << i << endl;
          tot++;
       }
    }
    return tot;
 }
```

Solution:

```
int Me(int n, int A[], int k)
{
    int tot = 0;                 //   1
    for (int i=n; i>=0; i--) { //3*(n+1)
       if (i==k)  {               // 3 , counting the test condition
          cout << i << endl;      // 1
          tot++;                  // 1
       }
    }
    return tot;                  // 1
 }
```

The worst case run time and complexity are:

$$T(n) = 3 * (n + 1) + 2 = \Theta(n)$$

Suppose you have three algorithms to solve a problem (with an input size of n). You are given that The first has asymptotic time complexity $\Theta(n^2)$, the second has asymptotic time complexity $O(n\sqrt{n})$, and the third has asymptotic time complexity $\Omega(n \log, n)$. Which algorithm would you choose? Briefly explain.

Solution:

Have to compare algorithms with worst-case time complexities estimated as follows: $\Theta(n^2)$, $O(n\sqrt{n}), \Omega(n \log n)$.

• An algorithm that is bounded from above by $O(n\sqrt{n})$ is definitely better than the $\Theta(n^2)$ algorithm, so we have to choose between the last two algorithms.

It is true that the function $n \log n$ has lower asymptotic growth rate than $n\sqrt{n}$. On the other hand the last algorithm is bounded from below by $n \log n$ which does not guarantee that it is better than the $O(n\sqrt{n})$ algorithm. In an extreme case, as an example, a $\Theta(2^n)$ algorithm is $\Omega(n \log n)$. Thus, with the information we have, the second algorithm, $O(n\sqrt{n})$, should be selected.

Give the worst-case and the best-case recurrences expressing the running time of the following algorithm manipulating a BST rooted at p.

BST:Binary Search Tree

```
int You(int &p, int a, int b)
{
    int tot = 0;
    if (p==0) return 0;
    if ( p > a && p < b) {
        cout << p << endl;
        tot++;
    }
    tot += You( Left(p), a, b);
    tot += You( Right(p), a, b);
    return tot;
}
```

Solution:

```
int You( int *p, int a, int b)
{
    int tot = 0;                //  1
    if (p==0) return 0;         //  base case T(0)=2

    if ( *p > a && *p < b) {   // 3 , counting the test condition
        cout << *p << endl;     // 1
        tot++;                  // 1
    }
    tot += You( Left(p), a, b);  // T(left subtree)
    tot += You( Right(p), a, b); // T(right subtree)
    return tot;                  // 1
}
```

What are the best- and the worst- case time complexities of this algorithm?

The running time of the recursive procedure on a tree with n nodes can be expressed as:

$$\begin{aligned} T(0) &= 3 \\ T(n) &= T(left\ subtree) + T(right\ subtree) + 5 \end{aligned}$$

The analysis is similar to that for the tree traversal algorithms.

The number of statements executed at each call (excluding the recursive calls themselves) is constant (5 to be precise). The procedure will be called for each node (for a subtree rooted in each node) exactly once. Since there are n nodes, and for each one the work done is constant, the total runt time is $5n$. Independent of the exact tree structure, all nodes are visited, i.e. the best-and worst- case complexities are the same $\Theta(n)$.

In a directed graph, the *in-degree* of a vertex is the number of edges entering it. Let n be the number of vertices, and m the number of edges in the graph, and let $d_I(v)$ be the in-degree of vertex v.

Given an **adjacency matrix** representation of a directed graph, and a specified vertex v, how would you best compute the in-degree of v? Write your algorithm in pseudo code, analyze the time complexity of your algorithm.

Here you need just to count the number of 1's in the column of the adjacency matrix corresponding to v. Since there are n entries in the column, clearly the time complexity is $\Theta(n)$.

```
// M is the adjacency matrix, of size nxn, M[u][v]=1 if (u,v) is an edge
InDegree(M, v)
   int deg=0;                  // 1
   for (int u=0; u<n; u++)     // n
      deg += M[u][v];              // 1
   return deg;                 // 1
```

Let $G = (V, E)$ be graph with n vertices and m edges.

$$T(n, m) = n + 3 = \Theta(n)$$

For the following program fragment compute the worst-case asymptotic time complexity (as a function of n). Whenever it says "loop body" you can assume that a constant number of statements are there. Show your work. your answer.

```
for (i=0; i<=n-1; i++){
   for (j=0; j< 5; j++) {
       for (k=0; k<n; k++) {
      loop body
}
for (i=0; i<=n-1; i++){
   for (j=0; j<=i; j++){
      loop body
   }
}
```

Solution:

```
for (i=0; i<=n-1; i++){          // Sum n times 5.n.c ==> 5.n.n.c
   for (j=0; j< 5; j++) {        // Sum 5 times n.c  ==> 5.n.c
       for (k=0; k<n; k++) {     // Sum n times c ==> n.c
      loop body                  // c
}
for (i=0; i<=n-1; i++){          // Sum (i+1)c for all i from 0 to
n-1
   for (j=0; j<=i; j++){         // Sum i+1 times c ==> (i+1)c
      loop body                  // c
   }
}
```

- $$\begin{aligned} T(n) &= 5n^2c + \sum_{i=0}^{n-1}(i+1)c = \\ &= 5n^2c + c\sum_{i=0}^{n-1} i - nc = \\ &= 5cn^2 + \frac{(n-1)n}{2} - nc = \\ &= (5c+0.5)n^2 - (c+0.5)n = \Theta(n^2) \end{aligned}$$

Answer each of the following questions AND justify your answer.

(a) Assume that f and g take only positive values. Is $f(n)+g(n) = O(\max\{f(n), g(n)\})$? Justify your answer.

(b) If $f(n) = O(g(n))$ then does it follow that $g(n) = O(f(n))$?

Solution:

(a) Assume that f and g take only positive values. Is $f(n)+g(n) = O(\max\{f(n), g(n)\})$? Justify your answer.

Note that you cannot do a prove, by "example" here, i.e. it is not enough to show that for some pair of functions the statement is true. You must show that for every pair, it is true.

• For any n, $f(n) + g(n) \leq 2\max\{f(n), g(n)\}$.

Thus if we choose $n_0 = 1$, and $C = 2$, the definition of Big-Oh is satisfied. Thus

$$f(n) + g(n) = O(\max\{f(n), g(n)\})$$

The answer is "yes".

(b) If $f(n) = O(g(n))$ then does it follow that $g(n) = O(f(n))$?

• "No", since there are functions for which $f(n) = O(g(n))$, and $g(n) \neq O(f(n))$.

For example, take $f(n) = 1$ and $g(n) = n$.

Note that since we found one example for which

$$f(n) = O(g(n)) \text{ does not imply } g(n) = O(f(n)).$$

The answer is "No". Proof by counterexample is ok here.

What is the worst-case asymptotic time complexity of the following divide-and-conquer algorithm. You may assume that n is a power of 2. (NOTE: It doesn't matter what this algorithm does.)

```
foo(n,A){
  let B be an array of size n
  if (n==1){
    B[0] = 1;
    return B;
  }

  let AL be an array of size n/2
  let AR be an array of size n/2
  for (i=0; i <= (n/2)-1; i++)
    AL[i] = A[i];
  for (i=n/2; i <= n-1; i++)
    AR[i- (n/2)] = A[i];

  BL = foo(n/2,AL);
  BR = foo(n/2,AR);

  for (i=0; i<= n-1 ; i++)
     B[i] = 1;
  for (i=n/2; i <= n-1; i++)       //for each element on right half
     for (j=0; j <= (n/2)-1; j++)  //  for each element on the left half
        if (A[i-n/2] > A[j])
           B[i] = max(B[i],BR[i]+BL[j]);
  return B;
}
```

Solution:

Let `T(n)` be the run time of `foo()` on an input of size `n`.

```
foo(n,A){
  let B be an array of size n
  if (n==1){                          //  2, base case
    B[0] = 1;
    return B;
  }
  let AL be an array of size n/2
  let AR be an array of size n/2
  for (i=0; i <= (n/2)-1; i++)       //  Theta(n/2)=Theta(n)
    AL[i] = A[i];
  for (i=n/2; i <= n-1; i++)         //  Theta(n/2)=Theta(n)
    AR[i- (n/2)] = A[i];

  BL = foo(n/2,AL);                  // T(n/2)
  BR = foo(n/2,AR);                  // T(n/2)

  for (i=0; i<= n-1 ; i++)           // Theta(n)
     B[i] = 1;

  for (i=n/2; i <= n-1; i++)             // Theta(n^2)
     for (j=0; j <= (n/2)-1; j++)        // Theta(n/2) = Theta(n)
        if (A[i-n/2] > A[j])             // c
           B[i] = max(B[i],BR[i]+BL[j]);
  return B;                              // 1
}
```

• Note that when we add up Big-Thetas, the complexity of the sum is the same as the highest order Big-Theta.

$$\begin{aligned} T(1) &= \Theta(1) \\ T(n) &= 2T(n/2) + \Theta(n^2) \end{aligned}$$

We use the Master method, $a = 2 \geq 1, b = 2 > 1, k = 2 \geq 0, p = 0 \geq 0$, since $a < b^k$, we have case 3, and thus

$$T(n) = \Theta(n^2)$$