1. C programming function calls (4pts)

1) There are two ways to the arguments can be passed during the function calls in C, call by value and call by reference. In your own words, describe the differences between the two (in 2- 3 sentences).

A) Call by value doesn't pass the actual value but the copy of it to the function whereas Call by reference passes the memory address to the function. In result, Call by reference changes the original data but Call by value doesn't.

2) Read the follow program. Using your response in 1), explain which variable is passed by value, and which one is passed by reference.

Note: In 2), you MUST use your response in 1) to explain the answer.

```
#include
// Function 1
void fun1(int a) {
        a = a + 1;
}
// Function 2
void fun2(int *b) {
        *b = *b + 1;
}
```

```
int main() {
   int x = 10;
   int y = 10;
   printf("Before the function call: x = %d₩n", x);
   fun1(x);
   printf("After the function call: x = %d₩n₩n", x);
   printf("Before the function call: y = %d₩n", y);
   fun2(&y);
   printf("After the function call: y = %d₩n", y);

   return 0;
}
```

A) fun1(x) is passing by value of the variable x so the copy of x is passed to the function. In result, original x value isn't changed because the copy of the value gets changed in the parameter not the original value of x.

fun2(&y), on the other hand, the y variable is passed by reference. So the function passes the memory address of the variable y. In result, the original value of y gets changed to 11.

2. C programming pointers (12pts)

There are some errors in the following program. Please identify them and provide the correct statements.

Note: Some lines of code with errors may be compiled and executed but will not do what it is expected to do.

```
int main() {
   int A[5]={1,3,7,4,0};
   int *P[5], *pA;
   int i, x, y;
   //Set pointer to the base address of array
   *pA = &A; -> Error number 1 -> It should be changed to pA = A;
   // Assign A[1] to x and y
   x = *pA+1; -> Error number 2 -> To access A[1], it should be changed to *(pA+1);
   A++; -> Error number 3 -> Increment of array name is illegal. It should be changed to
                                pA++;
   y=*A;   -> Error number 4 -> It should be also changed to y = *(pA+1);
   printf("x = ₩t%d",x);
   printf("y = ₩t%d",y);
    //Set every pointer to one array element
    for(i=0;i<5;i++)
        *P[i]=&A[i];   -> Error number 5 -> Correct statement would be P[i]=&A[i]; since
        P is a pointer array.
    //print array elements using pointers
    for(i=0;i<5;i++)
        printf("₩t%d",P[i]);     -> Error number 6 -> In order to print the values you need
        to put an asterisk. -> printf("₩t%d",*P[i]);
```

```
    return 0;
}
```

3. Big-O, big Omega, and Theta (9pts)

Referring to the definitions and examples of Big-O, big Omega, and Theta introduced in the

course slides, show that

1) 3n^2 = O(n^3)

-> By definition, we need to find a constant c such that f(n) <= cg(n) and f is a big O of g(n)

So if we define f(n) and g(n)

- **➔ f(n) = 3n^2**
- **➔ g(n) = n^3**

3n^2 <= c*n^3

- **➔ Dividing both sides by n^3**

3/n <= c

C = 3; n_0 = 1

Thus 3n^2 = O(n^3)

2) n^2+2n ≠ Ω(n^3)

-> By definition, we need to find a constant c such that f(n) >= cg(n)

So if we define f(n) and g(n)

- **➔ f(n) = n^2 +2n**
- **➔ g(n) = n^3**

f(n) >= c*g(n)

- **➔ n^2 +2n >= c*n^3**

Divide both sides by n^2

- **➔ 1 + 2/n >= c*n**

lim $n \to \infty$ 2 /n = 0.

1>= c*n

There are no positive constants c and n_0 in that condition thus n^2 +2n ≠ Ω(n^3)

3)n^2-2n= $\boldsymbol{\Theta}$ (n^2)

If we define f(n) and g(n)

f(n) = n^2 -2n

g(n) = n^2

We say that f(n) ∈ (g(n)) if f(n) ∈ Ω(g) and f(n) ∈ O(g)

Showing f(n) ∈ Ω(g)

f(n) >= c_1*g(n)

n^2 -2n >= c_1*n^2

Dividing by n^2 both sides

1 – 2/n >= c_1

Thus when c_1 = 1/2, n_0 = 4

Showing f(n) ∈ O(g)

f(n) <= c_2*g(n)

n^2 -2n <= c_2*n^2

Dividing by n^2 both sides

1 – 2/n <= c_2

lim $n \to \infty$ 2 /n = 0.

c_2 = 1, n_1 = 4

-→ n_0 = 4, c_1 = 1/2, c_2 = 1

Thus 1/2(n^2) <= n^2-2n <= n^2

So it proves.

Note: you MUST use the definitions of the different notions introduced in class to solve the questions.

4. Bubble sort and revised bubble sort total operations (12pts)

Bubble sort is a simple sorting algorithm. It compares each adjacent pair and check if the

elements are in order. If they aren't, the elements are swapped. This process continues until all elements are sorted.

If there is an integer array of n elements, and the algorithm output the array in an increasing

order (from smallest to the largest), the bubble sort can be implemented as

```
for(int i = 0;i < n; i++){
 for(int j=0;j < n - 1; j++){
 if(arr[j] > arr[j+1]){
 int temp = arr[j];
 arr[j] = arr[j+1];
 arr[j+1] = temp;
 }
 }
}
```

You have an array X=[1,2,3,…,n] and an array Y=[n,n-1,n-2,…,1]. So, X represents the best case

(the array is already sorted) and Y represents the worst case (the array is sorted reversely).

1) Analyze the above bubble sort and calculate the total number of operations with input X and Y, respectively.

   So X is the best case of a sample array. To analyze the total operations of it, the outer loop operates n times and the inner loop operates n-1 times. Which means the total operations is n(n-1) times. And we need to multiply

the executed lines in the code.

for(int i = 0;i < n; i++){ -→n times

for(int j=0;j < n - 1; j++){ -> n-1 times

if(arr[j] > arr[j+1]){ -> n-1 times (It only checks the condition and doesn't swap)

int temp = arr[j];

arr[j] = arr[j+1];

arr[j+1] = temp;

Thus 1*n*(2*(n-1))

In terms of Y, which is the worst case, the total operations is exactly the same with X which is n(n-1) times but it contains the swapping operations as well whereas X only does the comparisons. And we need to multiply the executed lines in the code as well. If I skip the for(int ~~~) condition lines as an execution, we have to multiply 4 which are the if condition and the swapping algorithm lines

for(int i = 0;i < n; i++){ -→n times

for(int j=0;j < n - 1; j++){ -> n-1 times

if(arr[j] > arr[j+1]){ -> n-1 times

int temp = arr[j]; -> n-1 times

arr[j] = arr[j+1]; -> n-1 times

arr[j+1] = temp; -> n-1 times

. So in result, n(n-1)*(1+1+1+1+1) times.

There is another way of implementing bubble sort where the algorithm also checks if the array has been sorted (to improve the efficiency). We create a flag that checks if a swap has occurred between any adjacent pairs. If there is no swap while traversing the entire array, the array is completely sorted, and the algorithm can break out of the loop. This is called a revised bubble sort. It can be implemented as

```
for(int i = 0;i < n; i++){
  boolean isSwapped = false;
  for(int j=0;j < n - 1; j++){
       if(arr[j] > arr[j+1]){
            int temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
            isSwapped = true;
     }
     if(!isSwapped){
            break;
     }
  }
}
```

2) Analyze the above revised bubble sort and calculate the total number of operations with input X and Y, respectively.

So for the X array , which is the best case, the outer loop is supposed to operate n times and the inner loop operates n-1 times. However, in the first loop, the bool flag would be false so the outer loop breaks in the first iteration. Which means the total operation times of X array is n-1 which is the operation times of inner loop. But also we need to add and multiply the operations(executed lines).

So we add four lines that are executed only once. Which are

```
for(int i = 0;i < n; i++){
  boolean isSwapped = false;
if(!isSwapped){
        break;
    }
```

And we multiply two executed lines in the inner loop which is

```
for(int j=0;j < n - 1; j++){
if(arr[j] > arr[j+1]){
```

In result, the answer is 4 + (n-1)*2 = 2n+2

As for the Y array, which is the worst case, in every iteration of j for loop, isSwapped is true at the end of all of the iterations including outer loop and the inner loop .

So X is O(n) in terms of big-O expression and Y is O(n^2)

But also we need multiply the operations(executed lines).

We multiply three lines by n which are

for(int i = 0;i < n; i++){ **= Executed n times**

boolean isSwapped = false; **= Executed n times**

if(!isSwapped){ **= Executed n times**

and add break only one since when it's executed, it means everything is sorted

break; **= Executed 1 time**

and we multiply five lines in the inner loop by (n(n-1))/2 which are

for(int j=0;j < n - 1; j++){ **=Executed (n(n-1))/2 times**

if(arr[j] > arr[j+1]){ **=Executed (n(n-1))/2 times**

int temp = arr[j]; **=Executed (n(n-1))/2 times**

arr[j] = arr[j+1]; **=Executed (n(n-1))/2 times**

arr[j+1] = temp; **=Executed (n(n-1))/2 times**

isSwapped = true; **=Executed (n(n-1))/2 times**

In result 3*n + 6*(n(n-1))/2 +1

=3n + (6n^2-6n)/2 +1

=3n^2 +3n -3n +1

=3n^2 +1

Note: you MUST include the calculation steps and explain how you get the answers.

5. Complexity (3pts)

Suppose you have a computer that requires 1 minute to solve problem instances of size n = 100. What instance sizes can be run in 1 minute if you buy a new computer that runs 64 times faster than the old one, assuming the Time complexities T (n) ∈ $\boldsymbol{\theta}$ (n^2) for the algorithm?

T (n) ∈ $\boldsymbol{\theta}$ (n^2)

c_1*g(n) <= f(n) <=c_2*g(n)

➔ T(n) = c*n^2

The amount of time spent of the old computer with the size of 100

-> 100^2*c = 10000*c

The amount of time spent of the new faster with the size of 100

-> (10000*c)/64 = 156.25c

To solve this question, we need to make an equation out of this information.

10000*c = a * (10000 / 64) * c (a is a variable that makes the time 1 minute for the new computer)

a = 64

64*10000 = 640,000

Since 640,000 = n^2

n = sqrt(640,000)

n = 800

Thus the size of 800 can be run in 1 minute on the new computer.

Note: you MUST include the calculation steps in your answer