Note to Students: This file contains sample solutions to the term test. Please read the solutions carefully. Make sure that you understand why the solutions given here are correct, that you understand the mistakes that you made (if any), and that you understand *why* your mistakes were mistakes. Remember that although you may not agree completely with the marking scheme used to grade your paper, it was followed the same way for all students. We will remark your test only if you clearly demonstrate that there is a need for this. For all remark requests:

- Please submit your request directly to Peter Sun (peteryiping.sun@mail.utoronto.ca)
- Your request must be a document which contains a clear description of why there is a need for remark and a scanned copy of the solution you had provided to the question in doubt
- Please remember that your question will be completely remarked and there is a chance that more marks maybe deducted

Question 1. [20 marks]

Please provide short answers to each one of the following sub-questions.

Part (a) [2 marks]

What are the three steps that convert a C program into executable code. Which function in the C program does execution begin from?

Sample Solution:

> Pre-processing (modifies original program), compiling (converts to machine code) and linking (combines additional libraries if necessary). Execution starts from the main function.

Part (b) [2 marks]

Will this function definition result in any accidental loss of information and if so why? If not, why not?

```
float f(char a, float b)
{
  short int d=a;
  return b+d;
}
```

Sample Solution:

> There will be no undefined behaviour as `short int` uses more bits than char to store a value and char can be treated as numbers as integers occupying 8 bits. In the return statement, d is type cast into float and this value is added with b which is a float. Hence, no information is lost.

Part (c) [3 marks]

Re-write the following function so that the same behaviour of `f` is retained but with only one `if` statement and without any `else` statements (i.e. replace the following four `if` statement blocks and the `else` block with one `if` statement).

```
void f(int a)
{
  if (a<3)
  {
     if(a>=-1)
     {
        printf("%d\n",a);
     }
     if(a < -2)
     {
        printf("%d\n",a);
     }
     if(a<-1 && a>=-2)
     {
        printf("%d\n",a+1);
     }
  }
  else
  {
     printf("%d\n",a+1);
  }
}
```

SAMPLE SOLUTION:

```
void f(int a)
{
  int i=1;
  if (a<3 && ((a>= -1) || (a<-2) ))
  {
     i=0;
  }
  printf("%d\n",a+i);
}
```

Part (d) [1 MARK]

Re-write the equivalent code snippet of the following `do-while` loop in terms of a `for` loop.

```
i=n;
do{
   printf("%d\n",i);
   i=i+1;
}while(i<n);
```

SAMPLE SOLUTION:

```
i=0;
for(i=n;i<n+1;i++)
{
   printf("%d\n",i);
}
```

Part (e) [2 MARKS]

Assume that in the initial function call to `division`, `a>1`. Do you foresee any possible issues with this code? If so and if not, why? Are there other restrictions on the possible values `a` can take?

```
float division(float a)
{
   if(a==1.0)
   {
      return a;
   }
   else
   {
      division(a/2);
   }
}
```

Sample Solution:

> There is a possibility of infinite regression if `a` is not 2^n.

Part (f) [2 marks]
Are there any issues with this code snippet? Why or why not?

```
int * a, ** b, c=1;
b=&a;
a=&c;
**b=2;
```

Sample Solution:

> No issues. a and b store valid addresses.

Part (g) [2 marks]
What problem(s) may result from the following code sequence?

```
int * i = (int *)malloc(sizeof(int));
int * j = (int *)malloc(sizeof(int));
int * k=j;
i=j;
free(j);
*k=3;
```

Sample Solution:

> Memory leak with dynamically allocated space assigned to `i`. Dangling pointer error when indirection operator is used.

Part (h) [2 marks]
Are there any issues with the following code snippet? If so, why? If not, what will be printed to the console?

```
int * a = (int *)malloc(sizeof(int)*3);
int i;
for (i=0;i<3;i++){
   a[i]=i;
}
printf("%d \n", *(a+2));
```

Sample Solution:

> No issues and 2 will be printed to the screen

Part (i) [2 marks]
What is the issue in this function implementation?

```
int ** func()
{
    int * i = (int*)malloc(sizeof(int));
    *i=1;
    return &i;
}
```

Sample Solution:

This is an issue with scope. `i` is a pointer variable and will cease to exist once the scope of `f` is over. Returning the memory address of a variable that no longer exists will result in undefined behaviour.

Part (j) [2 marks]

Consider the following code snippet. After statement 2, will applying the following operations on `a` be valid: `*&**&a;`? Why or why not?

```
int *a, b=1, c;
a=&b;
```

Sample Solution:

Yes, this is a valid sequence of operations.

Question 2. [15 marks]

In the following, you will be requested to implement a set of functions related to the *Fibonacci* sequence.

Part (a) [5 marks]

A fibonacci sequence is a sequence of numbers in which every term, with the exception of the first two terms, is the sum of the previous two terms. The first two terms are 0 and 1 respectively. The indexing of the terms start from 0. For instance the 6^{th} term of the Fibonacci sequence $\{0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8\}$ is 8. In the function `int fib(int n)`, you will implement via ***recursion*** the logic that computes the n^{th} term of the Fibonacci sequence and returns the result (i.e. a call to `fib(6)` should return 8).

Sample Solution:

```
int fib(int n){
    if (n==0) {
        return 0;
    }
    else if(n==1){
        return 1;
    }
    return fib(n-1)+fib(n-2);
}
```

Part (b) [4 marks]

Draw the call tree resulting from the recursive implementation of the `fib` function for the specific function call `fib(4)`.

Sample Solution:

```
                             fib(4)
                                3
                          /             \
                    fib(3)              fib(2)
                     2                      1
                /        \              /      \
        fib(2)         fib(1)     fib(1)    fib(0)
         1               1          1          0
    /         \
fib(1)    fib(0)
1            0
```

Part (c) [6 MARKS]

In this question, you will implement the function `int sumFib(int n, int s)` ***recursively*** which computes the sum of the Fibonacci sequence up to the n^{th} term. For instance, the sum up to the 6^{th} term of the sequence {0, 1, 1, 2, 3, 5, 8} is 20. Two arguments are passed to this function and these are `n` (the term up to which the Fibonacci sequence is summed) and `s`. You can use the integer argument `s` as a helper variable. If you are using this variable, specify clearly how this variable is aiding with your implementation. This function will return the summation of the sequence up to the n^{th} term (i.e. a call to `sumFib` where $n = 6$ should return 20).

SAMPLE SOLUTION: **Solution 1:**

```
int sumFib1(int n, int s)
{
    int sum=0;
    if (n<1) {
        return 0;
    }
    else if(n==1){
        return 1;
    }
    if (s!=0) {
        while (n>0) {
            sum=sum+sumFib1(n,0);
            n=n-1;
        }
        return sum;
    }
    return sumFib1(n-1,0)+sumFib1(n-2,0);
}
```

Solution 2:

```
int sumFib2(int n, int s){
    if (n<1) {
        return 0;
    }
    else if(n==1){
        return 1;
    }

    if (s==0) {
        return sumFib2(n-1,0)+sumFib2(n-2,0);
    }
    else{
         return sumFib2(n-1,1)+sumFib2(n,0);
    }
}
```

Question 3. [15 MARKS]

The following set of function implementations will be related to creating and accessing dynamically allocated space.

Part (a) [7 MARKS]

Implement the function `int ** initStructure(int r, int * a)` which dynamically allocates a matrix-like space with `r` 'rows' and each row consists of certain number of integer elements. For example, suppose the space to be allocated consists of 3 rows and each row has the following number of integer elements:

- Row 1 contains 3 integer elements
- Row 2 contains 4 integer elements
- Row 3 contains 2 integer elements

then `r` is 3 and `a` is the array {3,4,2} containing the number of integer elements in each row. `r` and `a` are both passed as arguments to the function. After dynamically allocating this space, each integer element within this space is to be set to 0. The function returns a double integer pointer which is a reference to the dynamically allocated space. You can implement this function using regular array notation or pointer arithmetic.

SAMPLE SOLUTION:

```
int ** initStructure(int r, int * a)
{
    int i,j;
    int ** s=(int **)malloc(sizeof(int*)*r);
    for (i=0; i<r; i++) {
        s[i]=(int*)malloc(sizeof(int)*a[i]);
        for (j=0; j<a[i]; j++) {
            s[i][j]=0;
        }
    }
    return s;
}
```

Part (b) [8 MARKS]

Complete the `main` function below to call `initStructure` which you have already defined above in order to dynamically allocate space for the example provided earlier (i.e. `r` is 3 and `a` is {3,4,2}). Then, the initialized content of this space shall be printed to the console where each row is separated with a return carriage. You can create the array `a` in `main` however you wish. However, use ***only pointer arithmetic*** to access the content of the array and allocated matrix-like space. Also, ensure that there are no memory leaks in your implementation.

SAMPLE SOLUTION:

```
int main()
{
    int c[]={3,4,2};
    int ** s=initStructure(c,3);
    int i,j;
    for (i=0; i<3; i++) {
        for (j=0; j<c[i]; j++) {
            printf("%d ",*(*(s+i)+j));
        }
        printf("\n");
    }
    for (i=0; i<3; i++) {
        free(s[i]);
    }
    free(s);
}
```

Question 4. [10 marks]

In the following, you will implement the function that identifies the sub-array in an array of integers containing the largest sum. Given an array filled with `n` integer elements (consisting of both positive and negative values), implement the function `int findLargestSum(int * array, int n, int * a, int * b)` which computes the largest sum of a set of continuous elements in the array. The start index and end index of this sub-array are passed indirectly back to the function call via `a` and `b` which are pointers to valid regions of memory. For instance, suppose the array $\{-2,\ -3,\ 4,\ -1,\ -2,\ 1,\ 5,\ -3\}$ is passed to the function `findLargestSum`. The sub-array in this array containing the largest sum is $\{\ 4,\ -1,\ -2,\ 1,\ 5\}$ which begins from index 2 and ends at index 6 of the original array. If you implement this function in a more efficient manner whereby each element is examined only once, you will receive **1 bonus** mark (hint: keep track of positive sums of continuous elements in the array).

Sample Solution:

Inefficient Solution:

```
int findLargestSum1(int * array, int n, int * a, int * b){
    int currSum=array[0],sum=0;
    int i,j;
    *a=0;
    *b=0;
    for (i=0; i<n; i++) {
        for (j=i; j<n; j++) {
            sum=sum+array[j];
            if(sum>currSum){
                currSum=sum;
                *a=i;
                *b=j;
            }
            //printf("%d \n", currSum);
        }
        sum=0;
    }
    return currSum;
}
```

Efficient Solution:

```
int findLargestSum2(int * array, int n, int * a, int * b){
    int maxSoFar=0,maxTrack=0, i, lastUpdateIndex=0;
    for (i=0; i<n; i++) {
        maxTrack=maxTrack+array[i];
        if (maxTrack<0) {
            maxTrack=0;
            lastUpdateIndex=-1;
        }
        else{
            if (lastUpdateIndex==-1) {
                lastUpdateIndex=i;
            }
        }
        if (maxTrack>maxSoFar) {
            maxSoFar=maxTrack;
            *a=lastUpdateIndex;
            *b=i;
        }
    }
    return maxSoFar;
}
```