INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR	
	Stamp / Signature of the Invigilator

EXAMINATION (End Semester)	SEMESTER (Spring)

Roll Number									Section		Name	
Subject Number	C	S	1	0	0	0	3	Subject Name	*Programming and Data Structures*			
Department / Center of the Student									Additional sheets			

Important Instructions and Guidelines for Students

1. You must occupy your seat as per the Examination Schedule/Sitting Plan.
2. Do not keep mobile phones or any similar electronic gadgets with you even in the switched off mode.
3. Loose papers, class notes, books or any such materials must not be in your possession, even if they are irrelevant to the subject you are taking examination.
4. Data book, codes, graph papers, relevant standard tables/charts or any other materials are allowed only when instructed by the paper-setter.
5. Use of instrument box, pencil box and non-programmable calculator is allowed during the examination. However, exchange of these items or any other papers (including question papers) is not permitted.
6. Write on both sides of the answer script and do not tear off any page. **Use last page(s) of the answer script for rough work.** Report to the invigilator if the answer script has torn or distorted page(s).
7. It is your responsibility to ensure that you have signed the Attendance Sheet. Keep your Admit Card/Identity Card on the desk for checking by the invigilator.
8. You may leave the examination hall for wash room or for drinking water for a very short period. Record your absence from the Examination Hall in the register provided. Smoking and the consumption of any kind of beverages are strictly prohibited inside the Examination Hall.
9. Do not leave the Examination Hall without submitting your answer script to the invigilator. **In any case, you are not allowed to take away the answer script with you.** After the completion of the examination, do not leave the seat until the invigilators collect all the answer scripts.
10. During the examination, either inside or outside the Examination Hall, gathering information from any kind of sources or exchanging information with others or any such attempt will be treated as **'unfair means'**. Do not adopt unfair means and do not indulge in unseemly behavior.

Violation of any of the above instructions may lead to severe punishment.

Signature of the Student

To be filled in by the examiner

Question Number	1	2	3	4	5	6	7	8	9	10	Total
Marks Obtained											

Marks obtained (in words)	Signature of the Examiner	Signature of the Scrutineer

CS10003 Programming and Data Structures, Spring 2022–2023

13-June-2023 **End-Semester Test** Maximum marks: 75

Instructions to students

- Write your answers in the question paper itself. Be brief and precise. Answer all questions.
- Write the answers only in the respective spaces provided.
- The questions appear on the next **eight** pages.
- Space is provided for doing rough work. Please do not write the answers in the rough-work spaces. Ask for supplementary sheets during the test if you need more space for rough work.
- All the questions use the programming language C.
- Not all blanks carry equal marks. Evaluation will depend on overall correctness.
- In the fill-in-the-blanks type questions, do not change the logical structures of the codes. Do not use any variables other than those already supplied to you.

Do not write anything on this page.

1. **(a)** Suppose that we want to generate a list of all prime powers up to some $n \leqslant 100$ supplied by the user. The prime powers are to be stored in a 10×100 array A declared in the main function as:

```
int A[10][100];
```

The main function initializes all the entries of A to 0. The primes $\leqslant n$ are themselves stored in the first row (index 0) of A in the ascending order. The columns store the powers $\leqslant n$ of each prime in the first row. For example, if $n = 40$, then **only** the following entries of A are to be assigned values (the row and the column indices are shown as row and column headers). All other entries of A will continue to remain 0.

	0	1	2	3	4	5	6	7	8	9	10	11
0	2	3	5	7	11	13	17	19	23	29	31	37
1	4	9	25									
2	8	27										
3	16											
4	32											

To do this, we write three functions. Fill in the blanks in the code of each function so that the function works properly. Assume that $2 \leqslant n \leqslant 100$. Do **not** use any math library function anywhere in this exercise.

We first write a recursive function `isprime(A,n,i)` which checks whether or not `n` is divisible by any of the primes stored in the first row (index 0) of A. In the function, `i` is the **index** of a prime stored in $A[0]$. Assume that n is larger than all the primes stored in $A[0]$. If n is not divisible by any of these primes, then n itself is a prime, and 1 is returned. Otherwise, 0 is returned. **(3)**

```
int isprime ( int A[][100], int n, int i )
{
   /* Base case: No more primes to divide n with */
   if (i < 0) return 1;

   if (n % A[0][i] == 0) return _____________0_____________ ;

   return isprime ( A, _____________n_____________ , _____________i - 1_____________ );
}
```

Next, we write a function `genprimes(A,n)` to populate the first row `A[0]` with primes $\leqslant n$. This function returns the **index** of the last prime (not the number of primes) stored in `A[0]`. Assume that $2 \leqslant n \leqslant 100$. **(7)**

```
int genprimes ( int A[][100], int n )
{
   /* Base case */
   if (n == 2) {

      A[0][0] = _____________2_____________ ;

      return _____________0_____________ ;
   }
   /* Recursively generate all primes < n */

   int i = genprimes( A, _____________n - 1_____________ );
   /* Now check for the primality of n and proceed accordingly */

   if ( isprime ( A, n, _____________i_____________ ) ) {

      A[0][ _____________i + 1_____________ ] = n;

      return _____________i + 1_____________ ;
   } else {

      return _____________i_____________ ;
   }
}
```

Finally, we write the desired function **`genprimepowers(A,n)`** to populate the two-dimensional array **`A`**, as described in the first paragraph. The function returns the **index** of the last column of **`A`** with at least one non-zero element (recall that all the elements of **`A`** were initialized to 0 before this function is called). **(2)**

```
int genprimepowers ( int A[][100], int n )
{
   int i, j, k, ppwr;
   i = genprimes(A,n);     /* First generate all the primes <= n */
   for (j=0; j<=i; ++j) {  /* For the prime at index j */
      k = 1;
      while (1) {

         ppwr = ____________A[k-1][j] * A[0][j]____________ ;
         if (ppwr > n) break;
         A[k][j] = ppwr;

         ________++k;________     /* Write a single statement */
      }
   }
   return i;
}
```

(b) Consider the following program.

```
#include <stdio.h>

int main ()
{
   char A[4][4] = { "tis", "hie", "shn", "ted" };
   char B[25];
   int i,j,m;

   for (m = 0; m <= 24; m++) B[m] = '\0';

   for (i = 0; i <= 3; i++)
      for (j = 0; j <= 3; j++)
         B[(i+j)*(i+j+1)/2 + j] = A[i][j];

   for (m = 0; m <= 24; m++)
      if (B[m] != '\0') printf("%c", B[m]);

   printf("\n%s\n", B);

   return 0;
}
```

What will be the output of this program? **(3)**

thisistheend
thisisthe

2. **(a)** Fill in the blanks in the code below. The code should work as per the comments provided. Also write the output of the program.

```
#include <stdio.h>

typedef struct {
   int x;
   float *y;
} myStruct;

/* The function fun takes two parameters t1 and t2 of the above structure type */

void fun ( ______myStruct______ t1 , ______myStruct______ t2 )                 (1)
{
   int temp;
   float tempf;

   temp = t1.x; t1.x = t2.x; t2.x = temp;
   /* The code below swaps the values at locations pointed to by t1.y and t2.y */

   __________tempf = *t1.y; *t1.y = *t2.y; *t2.y = tempf;__________            (2)
}

int main()
{
   float a = 2.34, b = -9.39;
   /* Declare two variables s1 and s2 of the above structure type */

   ______myStruct______ s1 = {10, &a}, s2 = {20, &b};                         (1)

   fun(s1,s2);

   /* Print the value of x in structure s1, the value at location y in s1,
      the value of x in s2, and the value at location y in s2, in that order. */

   printf("%d, %.1f, %d, %.1f", ______s1.x, *s1.y, s2.x, *s2.y______ );      (4)

   return 0;
}
```

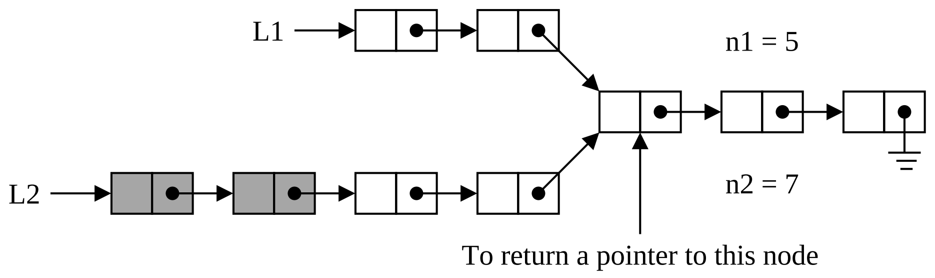
Output of the program: ______`10, -9.4, 20, 2.3`______ **(4)**

(b) What will be the output of the following program?

```
#include <stdio.h>

struct myStruct {
   int x;
   char y[5];
   struct myStruct *ptr;
};

int main()
{
   struct myStruct s1 = {10, {'a','b','\0','c','\0'}, NULL}, s2;
   s2 = s1;
   s2.x = 20;
   s1.ptr = &s2; s2.ptr = &s1;
   s1.y[0] = 'd'; s2.y[1] = 'e'; s2.y[2] = 'z'; s1.ptr->x = 30;
   printf("%d, %s, %s", s2.x, s1.ptr->y, s2.ptr->y);
   return 0;
}
```

Output of the program: ______`30, aezc, db`______ **(3)**

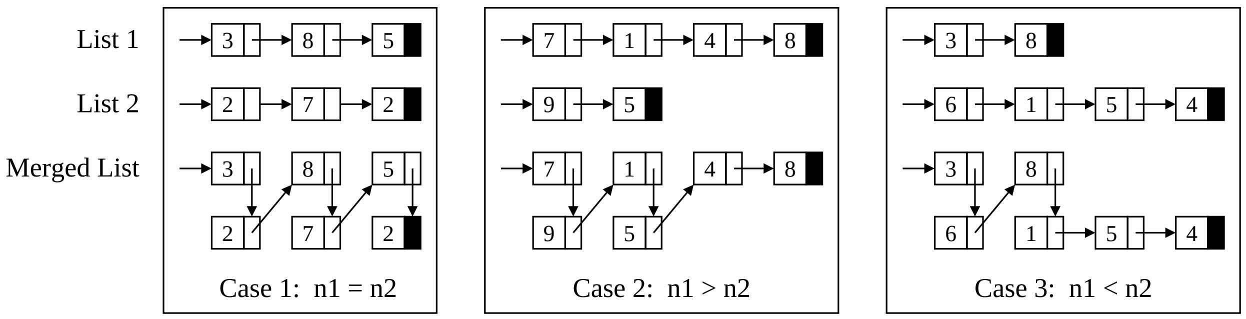
3. **(a)** You have two linked lists which (may) intersect at some node. The head pointers of the lists are supplied. You need to find the intersecting node. If the lists do not intersect, you return NULL. The numbers of nodes in the lists may be different, and are not known beforehand. The following picture illustrates this concept. The data values stored in the nodes are irrelevant for this exercise, and are not shown.

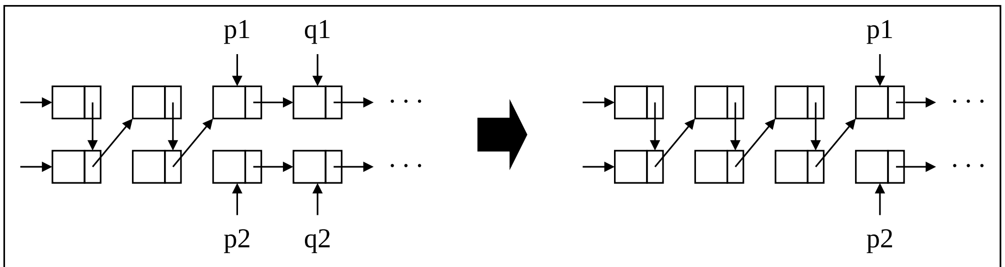


The function `findIntersection()` returns a pointer to the intersecting node. Fill in the blanks to complete the code of the function. You must use the algorithm specified as comments inside the code. **(7)**

```
typedef struct _node {
   int data;
   struct _node *next;
} node;

node* findIntersection ( node *L1, node *L2 )
{
   int n1, n2, diff, i;
   node *p1, *p2;

   /* Compute the number of nodes (n1) in L1 */
   n1 = 0; p1 = L1;

   while (p1 != NULL) { ++n1; p1 = ______________ p1 -> next ______________ ; }

   /* Compute the number of nodes (n2) in L2 */
   n2 = 0; p2 = L2;

   while (p2 != NULL) { ++n2; p2 = ______________ p2 -> next ______________ ; }

   /* Let p1 point to the larger list, and p2 to the smaller list */
   /* Also compute the absolute difference (diff) of the list sizes */
   if (n1 >= n2) {
      p1 = L1;
      p2 = L2;
      diff = n1 - n2;
   } else {

      p1 = ___________________ L2 ___________________;

      p2 = ___________________ L1 ___________________ ;
      diff = n2 - n1;
   }

   /* Skip first diff nodes from larger list (shaded nodes in the above figure) */

   for (i=0; i<diff; i++) __________________ p1 = p1 -> next __________________ ;

   /* Now the intersecting node is at the same distance from p1 and p2 */
   /* Locate the intersecting node by advancing p1 and p2 simultaneously */

   while ( _________________________ p1 != p2 _________________________ ) {
      p1 = p1 -> next; p2 = p2 -> next;
   }

   return ____________ p1 ____________ ;
}
```

(b) The function **`mergelists()`** given below merges two linked lists by alternately taking nodes from the two lists, with the first node taken from the first list (this has nothing to do with merge sort). If both the lists are not of the same size, the extra elements from the longer list are added to the end of the merged list. Examples of three cases are given in the top half of the figure below. Here, n1 and n2 are the sizes of the two lists (we do **<u>not</u>** need to compute them), and the NULL pointers are shown as solid black rectangles. The algorithm used in **`mergelists()`** is indicated in the bottom half of the figure. The pointers **`p1`** and **`p2`** run over the two lists. The pointers **`q1`** and **`q2`** point to the next nodes in the respective lists. Fill in the blanks properly as per the comments specified in the code, to complete the function **`mergelists()`**. **(8)**





Explaining an iteration of the loop

```
struct listNode {
   int data;
   struct listNode *next;
};

struct listNode *mergelists ( struct listNode *list1, struct listNode *list2 )
{
   struct listNode *p1, *p2, *q1, *q2;

   /* First handle the pathological cases */

   if (list1 == NULL) return __________list2__________ ;

   if (list2 == NULL) return __________list1__________ ;

   p1 = list1; p2 = list2;     /* Initialize the loop */

   /* Repeat so long as there are nodes remaining in both the lists */

   while ( ( _______p1 != NULL________ ) && ( _______p2 != NULL________ ) ) {
      q1 = p1 -> next; q2 = p2 -> next;     /* Save the next pointers */

      /* Include a node from the second list */

      _________________p1 -> next = p2;_________________

      /* Conditionally include a node from the first list */

      if ( q1 != NULL ) __________p2 -> next = q1;__________ ;

      /* Prepare for the next iteration: Write two statements */

      _______________________p1 = q1; p2 = q2;_______________________
   }

   return __________list1__________ ;
}
```

4. An implementation of the stack ADT supplies **only** the following functions. The `stack` data type is designed to store stacks of `int` variables. Write below what the return types of the ADT calls should be. **(5)**

`stack`	`newstack ( int maxsize );`	Create a new stack (empty) capable of storing a maximum of `maxsize int` variables. You are **not** allowed to change the stack capacity afterwards.
`int`	`isempty ( stack S );`	Return whether the stack `S` is empty or not.
`int`	`top ( stack S );`	Return the top element stored in the stack `S`. You should **not** call this function if `S` is empty.
`stack`	`push ( stack S, int x );`	Insert a new element `x` to the stack `S`. It is your duty to ensure that you do **not** exceed the stack capacity that you specified during the `newstack()` call.
`stack`	`pop ( stack S );`	Remove the top of the stack. You must make sure that you do **not** try to pop from an empty stack.

Do not write/change the codes of these ADT calls. Use the ADT implementation given to you for solving the following problem. You have no idea how the stack is implemented (using static arrays, or dynamic arrays, or linked lists, or whatever). You work with your `int` stack(s) using only the above five ADT functions.

Let $R[\,]$ be a `float` array storing the rainfall amounts $r_0, r_1, r_2, \ldots, r_{n-1}$ for n consecutive days. Take i in the range $0 \leqslant i \leqslant n-1$. The rainfall r_i is said to *dominate* for d_i days if $r_i, r_{i+1}, r_{i+2}, \ldots, r_{i+d_i-1} \leqslant r_i$, whereas $r_{i+d_i} > r_i$. If we take $r_n = \infty$, then d_i is defined for all i in the range $0 \leqslant i \leqslant n-1$. An example of the rainfall amounts and the domination-day-counts is given below for $n = 12$.

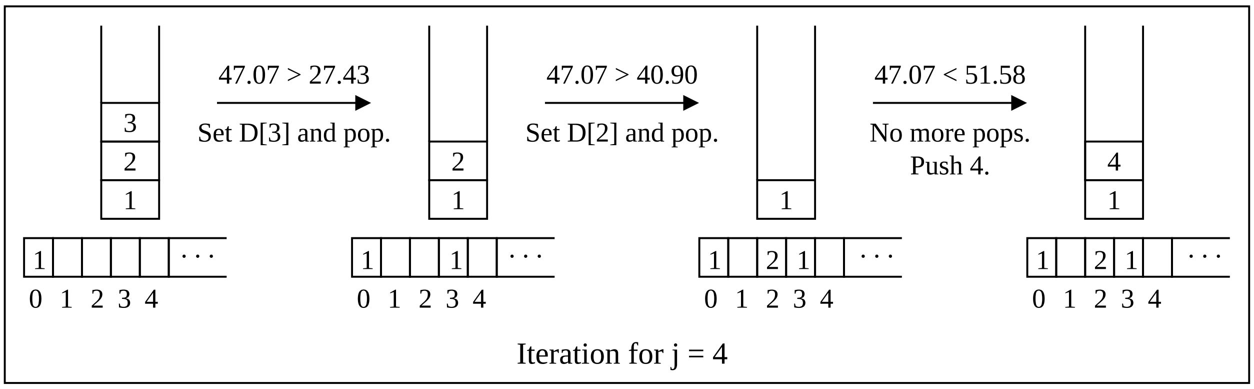
i	0	1	2	3	4	5	6	7	8	9	10	11	12
r_i	31.39	51.58	40.90	27.43	47.07	19.82	20.26	53.92	43.32	33.19	20.42	37.40	∞
d_i	1	6	2	1	3	1	1	5	4	2	1	1	–

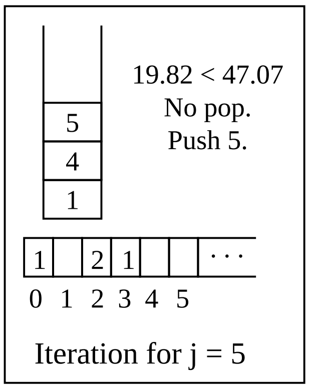
The basic task is to determine, for each i, the **first** $j > i$ for which $r_j > r_i$, and set $d_i = j - i$. We can start the search at every index i to find the index j and subsequently compute d_i. But this requires multiple passes through the array. We use a stack S in order to compute d_i for all i, using a single pass of the array $R[\,]$. Inside a loop over j (not over i), the stack stores all $i < j$ for which d_i is not yet determined. The stack is sorted in the decreasing order of i from top to bottom. Since S is a stack of `int` variables, we cannot store r_i in the stack. We store i, and get r_i from the array R as $R[i]$. Because the domination-day-counts are not determined at every index stored in S, the r_i values corresponding to the indices i stored in S must be in the non-decreasing order from top to bottom of S.

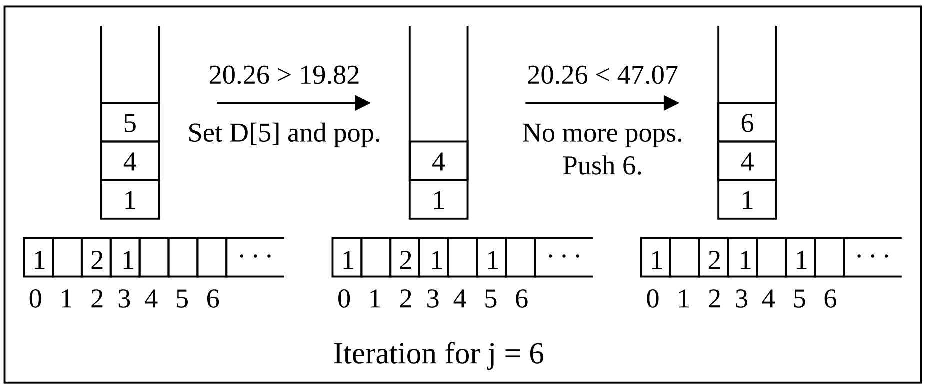
We start with an empty stack. Then, we read r_j for $j = 0, 1, 2, \ldots, n$ in that sequence. For each j, zero or more indices $i < j$ reside in the stack. We determine which of these r_i values are smaller than r_j. For each such i, the nearest larger member of r_i is r_j. We set their d_i values, and pop them from the stack (one by one). The process stops when the stack becomes empty or the top stores an i for which $r_j \leqslant r_i$. In the second case, if there are one or more indices i' in the stack below the top, then $r_{i'} \geqslant r_i \geqslant r_j$, so $d_{i'}$ cannot be determined by this j. Finally, for the current j, the value d_j cannot be determined without looking into the elements that follow r_j in R, so we push j to S, and proceed to the next iteration.

As an example, consider $j = 4$ in the above table (also see the figure on the next page, where the stack S is shown vertically, and the d_i values are shown in the horizontal array; the uncomputed d_i values are shown as blank entries in the array). Before the beginning of the iteration, only d_0 is computed. The stack stores $3, 2, 1$ from top to bottom. Now, the element $r_j = r_4 = 47.07$ is considered. Since $r_4 > r_3$, we set $d_3 = 4 - 3 = 1$, and pop 3 from S. We then observe that $r_4 > r_2$ as well, so we set $d_2 = 4 - 2 = 2$, and pop 2 from the stack. After this, we see that $r_4 \leqslant r_1$, so d_1 cannot be determined now. We push 4 (so the stack now contains $4, 1$ from top to bottom), and the iteration for $j = 4$ finishes.

In the next iteration with $j = 5$, no d_i value can be set for $i = 4, 1$. Moreover, 5 is pushed to S. Then, for $j = 6$, only $d_5 = 6 - 5 = 1$ is set, and 6 is pushed to S. The next iteration with $j = 7$ sets $d_6 = 7 - 6 = 1$,







$d_4 = 7 - 4 = 3$, and $d_1 = 7 - 1 = 6$, and the stack becomes empty. After this, 7 is pushed to S. Eventually, the loop encounters $r_{12} = \infty$, and sets $d_i = 12 - i$ for all i that are still residing in the stack.

Fill in the blanks in the following code snippet that implements this algorithm. The d_i values are to be stored in an `int` array $D[\,]$. Assume that $R[\,]$ and $D[\,]$ (declared earlier) have sufficient space to handle n days. The number n of days and the daily rainfall amounts $r_0, r_1, r_2, \ldots, r_{n-1}$ (the entries of $R[\,]$) are read from the user (or from a weather-data file) before the snippet. The snippet uses the additional variables `j` and `S`. Use no other variables. Whenever needed, use the stack ADT calls from the given implementation.

```
int j;
stack S;

/* Get ready for the algorithm */
R[n] = INFINITY;     /* INFINITY is defined in math.h */

S = newstack( ______n______ ) ;     /* S should not overflow */          (1)

for (j = 0; j <= ______n______ ; ++j) {                                  (1)
   /* Consider r_j */

   /* Repeat the loop so long as some d_i value can be set */

   while ( ______(!isempty(S)) && (R[j] > R[top(S)])______ ) {           (3)

      D[ ______top(S)______ ] = ______j - top(S)______ ;                 (2)

      ______S = pop(S);______                    /* Single statement */  (1)
   }

   ______S = push(S,j);______                    /* Single statement */  (1)
}

for (j = 0; j <= ___n-1___ ; ++j)                                        (1)
   printf("Domination-day-count for %f is %d\n", R[j], D[j]);
```

5. (a) The following C function takes an integer array `A[]` and its number of elements `n` as parameters, and rearranges the elements such that all negative numbers in `A[]` precede the non-negative ones. Fill in the blanks appropriately. **(5)**

```
void rearrange ( int A[], int n )
{
    int k, i, p, t;

    k = ______________-1______________ ;

    p ______________= 0______________ ;
    for (i=0; i<=n-1; ++i) {

        if (A[i] ______________< p______________ ) {

            ______________++k;______________

            t = A[k]; A[k] = A[i]; ______________A[i] = t;______________
        }
    }
}
```

(b) Answer the following questions. No need to show your calculations. Write only your final answers in the blanks/boxes provided. **(2 × 5)**

(i) What is the hexadecimal equivalent of the decimal number 6752.953125?

1A60.F4

(ii) What are the smallest and largest numbers that can be represented in 32-bit 2's-complement notation?

Smallest: -2^{31}. Largest: $+(2^{31}-1)$.

(iii) What will be the result of performing 65 – 113 using 8-bit 2's-complement arithmetic? Write the bits of the result.

1	1	0	1	0	0	0	0

(iv) Consider the following 32-bit single-precision floating-point number representation in the IEEE-754 format.

0 00100111 0100110 00000000 00000000

What is the value of this number in decimal?

$1.296875 \times 2^{-88} \approx 4.190429 \times 10^{-27}$

(v) Write the 32 bits of the single-precision floating-point representation of the decimal number –2.75 in the IEEE-754 format.

sign	exponent	mantissa
1	10000000	0110000 00000000 00000000

For rough work only

For rough work only