Question 1. [10 MARKS]

Recall that the abstract data type Queue has operations `enter(data)` (add data to the tail of the queue), and `exit()` (remove data from the head of the queue and return it). With a queue that holds characters, the input sequence $A\,B$ (ordered from left to right), plus an operation `output(character)` (print next character from either the input sequence or the queue on standard output) you could generate (processing the input in left-to-right order):

$A\,B$	`output(A) output(B)`
$B\,A$	`enter(A) output(B) output(exit())`

Suppose your input were $A\,B\,C$. Which of the six permutations of $A\,B\,C$ can you generate using `enter(data)`, `exit()`, and `output(data)`? Show the operations that yield the permutations that are possible. Explain why any impossible permutations are impossible (it's not enough to show you've tried lots of combinations of operations). To get you started, you can generate:

$A\,B\,C$ `output(A) output(B) output(C)`.

If a student assumes they can perform `enter(exit())`, then they can produce all permutations (for full marks). Otherwise, the situation is as below (2 marks for each permutation/operations pair. 2 marks if they explain the impossible permutation in terms of the queue ordering).

$A\,C\,B$	`output(A) enter(B) output(C) output(exit())`.
$B\,A\,C$	`enter(A) output(B) output(exit()) output(C)`.
$B\,C\,A$	`enter(A) output(B) output(C) output(exit())`.
$C\,A\,B$	texttttenter(A) enter(B) output(C) output(exit()) output(exit()).

It's impossible to produce $C\,B\,A$, since if C is the first character output, the other two must have been `entered()`ed, and hence be in same order when they are `exit()`ed.

Question 2. [6 MARKS]

Which complexity class best describes the function `gloop(n)`? Choose your answer from $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$. Justify your answer (no marks without justification).

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

When `i` is 1, the inner loop executes `n` times, when `i` is 2 the inner loop executes `2n` times ..., leading to:

$$T(n) = n(1 + 2 + \cdots + (n-1)) = n\frac{n(n-1)}{2}, \in O(n^3).$$

Question 3. [10 MARKS]

Complete the function `gcd(int a, int b)` (Greatest Common Divisor of `a` and `b`) below, using recursion. You may **not** use any loop (`for`, `while`, etcetera).

```
/*
 * precondition:  n and m are non-negative integers
 *
 * postcondition: return the largest integer that divides both n and m with a remainder of 0.  If n is 0, return m.
 *
 * Note:          You may use the fact that if m >= n, then gcd(m, n) is the same as gcd(n, m%n).
 */
```

4 marks for the base case, 6 marks for the recursive call. The recursive call may test whether m ¿ n, although the alternative given here guarantees this is true after one more call.

```
int gcd(int m, int n)
{
    if (n == 0)
        return m;
    else
        return gcd(n, m%n);
}
```

Question 4. [10 MARKS]

Suppose the file `lists.h` contains declarations for `extern push(int n)`, `extern pop()` (to manipulate an integer stack), `extern enter(int n)`, and `extern exit()` (to manipulate an integer queue), while the file `lists.c` contains implementations (definitions) for the functions declared in `lists.h`, and these are compiled into module `lists.o`. You decide to write a program contained in `permute.c` that links to the functions `push(int n)`, `pop()`, `enter(int n)`, and `exit()` to generate permutations of integer sequences. After each of the following statements, write either `Yes` or `No`, and a brief explanation, in the space provided.

1. `permute.c` should `#include lists.c`.

2. `lists.c` should `#include permute.c`.

3. `lists.c` should `#include lists.h`.

4. `lists.h` should `#include lists.c`.

5. `permute.c` should `#include lists.h`.

`permute.c` should include `lists.h` so that it can see declarations of functions it calls. `lists.c` should include `lists.h` to keep the declarations consistent. None of the others should include the suggested files. 2 marks per item.

Question 5. [8 MARKS]

Recall that $T(n)$ is $O(g(n))$ if there is some constant $c \in \mathbb{R}^+$, and some integer n_0 such that

$$T(n) \le c \times g(n) \qquad \text{for all } n > n_0.$$

In each of the following cases set $n_0 = 1$, and find the smallest c that shows that $T(n)$ is $O(g(n))$:

1. $T(n) = 3n$, $g(n) = n$.

   Let $c = 3$, for $n > 1$:

   $$3n \le cn \Longrightarrow c \ge 3.$$

2. $T(n) = 5n^2 + 60n + 10^8$, $g(n) = n^2/2$.

   Let $c = 70 + 10^8/2$, for $n > 1$:

   $$5n^2 + 60n + 10^8 \le cn^2/2 \Longrightarrow c \ge 10 + 60 + 10^8/2.$$

3. $T(n) = n\log_2(n) + 70n$, $g(n) = (n\log_2 n)/17$.

   Let $c = 1207$, for $n > 1$:

   $$n\log_2 n + 70n \le c(n\log_2 n)/17 \Longrightarrow c \ge 17(1+70) = 1207.$$

4. $T(n) = 5n^2$, $g(n) = n^3$.

   Let $c = 5/2$, for $n > 1$:

   $$5n^2 \le cn^3 \Longrightarrow c \ge 5/2.$$

Question 6. [20 MARKS]

Read the declarations below from `int_queue.h`. Then, **on the next page** write the implementation of functions `enterq()`, `exitq()`, `empty()`, `full()`, `init_queue()`, and `destroy_queue()`. Implement `enterq()` and `exitq()` so that they have time complexity $O(1)$.

```
/*
 * int_queue.h --- public interface for an integer queue
 */

/*
 * data is an array of capacity integers  count is the number of ints currently stored,
 * head is the index of the next item to exitq(),  tail is the index of the last item to enterq()
 */
typedef struct {
  int *data;
  int head, tail;
  int capacity, count;
} int_queue_t;

/*
 * precondition:  n is a positive integer.
 *
 * postcondition: a queue with capacity for n ints is created, with head and tail set to -1,
 *                count 0 is created, and its address is returned.
 *
 */
extern int_queue_t *init_queue(int n);

/*
 * precondition:  n is an int, queue is the address of an initialized non-full queue.
 *
 * postcondition: n is the data at the tail of the queue.
 */
extern void enterq(int n, int_queue_t *queue);

/*
 * precondition:  queue is the address of an initialized non-empty queue.
 *
 * postcondition: element at the head of *queue is removed and returned.
 */
extern int exitq(int_queue_t *queue);

/*
 * return 1 if *queue is empty, 0 otherwise.  An empty queue has fewer than 1 elements.
 */
extern int empty(int_queue_t *queue);

/*
 * return 1 if *queue is full, 0 otherwise.  A full queue has >= capacity elements.
 */
extern int full(int_queue_t *queue);

/*
 * precondition:  queue is the address of a valid int_queue_t
 *
 * postcondition: resources allocated to *queue are released.
 */
extern void destroy_queue(int_queue_t *queue);
```

Use this page to write your code implementing the functions declared in `queue.h`.

```
int_queue_t *init_queue(int n)
{
    int_queue_t *new_queue = (int_queue_t *)
                             malloc(sizeof(int_queue_t));
    new_queue->head = new_queue->tail = -1;
    new_queue->count = 0;
    new_queue->capacity = n;
    new_queue->data = (int *)malloc(n * sizeof(int));
    return new_queue;
}


void enterq(int n, int_queue_t *queue)
{
    /* tail wraps around (if necessary) to 0 */
    queue->tail = (queue->tail + 1) % queue->capacity;
    if (queue->count == 0)
        queue->head = queue->tail;
    queue->count++;
    queue->data[queue->tail] = n;
}


int exitq(int_queue_t *queue)
{
    int value = queue->data[queue->head];
    /* head wraps around to 0 */
    queue->head = (queue->head + 1) % queue->capacity;
    queue->count--;
    return value;
}


int empty(int_queue_t *queue)
{
    return queue->count < 1;
}


int full(int_queue_t *queue)
{
    return queue->count >= queue->capacity;
}

void destroy_queue(int_queue_t *queue)
{
    free (queue->data);
    free(queue);
    queue = NULL;
}
```

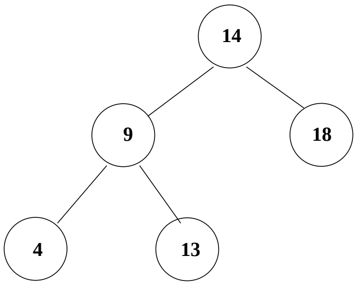
Question 7. [12 MARKS]

Implement a radix sort for a list of strings, where each string has exactly N uppercase alphabetic characters (i.e. each character is between 'A' and 'Z'). Each character position should be treated as a digit.

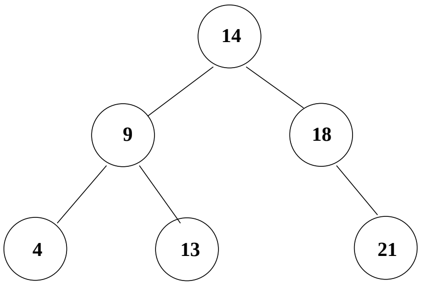
```
void stringRadixSort( char *array[], int array_size )
{
   int count[26],i,j;
   char **aux = (char **)
     malloc(sizeof(char *)*array_size);

   /* do all digits */
   for (i = N-1; i >= 0; --i) {

     /* clear the count array */
     for (j = 0; j < 26; ++j)
       count[j] = 0;

     /* count occurences of characters */
     for (j = 0; j < array_size; ++j) {
       count[array[j][i] - 'A']++;
     }

     /* find the last slot of each bin */
     for (j = 1; j < 26; ++j)
       count[j] += count[j-1];

     /* put the records into the bins */
     for (j = array_size-1; j >= 0; --j)
       aux[--count[array[j][i]-'A']] = array[j];

     /* copy the aux back to the array */
     for (j = 0; j < array_size; ++j)
       array[j] = aux[j];
  }
}
```

Question 8. [8 MARKS]

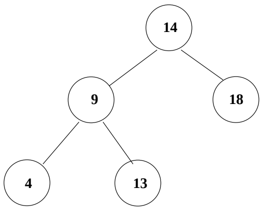
Assume that the trees below are AVL trees. First insert a new node with a key of 12 into (a). Next, insert a new node with a key of 3 into (b). For both parts, show the trees before and after each rotation you perform.



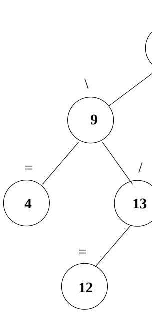
(a)



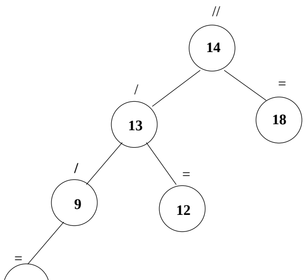
(b)



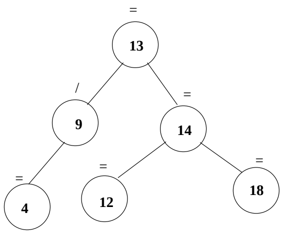
(1) original tree



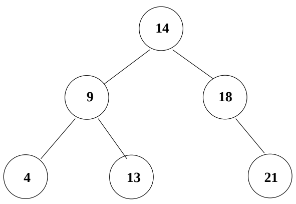
(2) after insertion



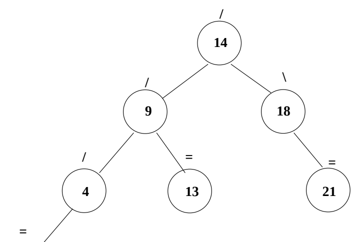
(3) after left rotation



(4) after right rotation



(1) original tree



(2) after insertion (no need to rotate)

Question 9. [8 marks]

The following code operates on a binary search tree of outer_node_t nodes, in which each outer_node_t node contains a binary search tree of inner_node_t nodes. Determine the computational complexity of the `sort_forest` function. Assume that the `sort_forest` function is passed the root of an outer tree which is a **full** binary search tree with N nodes. Also assume that each of the outer tree nodes contains an inner tree that is also a **full** binary search tree also with N nodes. Justify your answer (no marks without justification).

```
typedef struct inner_node_struct {
  int value;
  struct node_struct *left;
  struct node_struct *right;
} inner_node_t;

typedef struct outer_node_struct {
  int value;

  inner_node_t *inner_tree;

  struct outer_node_struct *left;
  struct outer_node_struct *right;
} outer_node_t;

void sort_inner_bst(inner_node_t *root) {
  if (root->left) sort_inner_bst(root->left);
  print("%d\n",root->value);
  if (root->right) sort_inner_bst(root->right);
}

void sort_forest(outer_node_t *root) {
    printf("Sorted elements for outer node %d:\n",
           root->value);
    sort_inner_bst(root->inner_tree);
    printf("\n");
    if (root->left) sort_forest(root->left);
    if (root->right) sort_forest(root->right);
}
```

(solution on next page...)

The sort forest function visits each node of the outer tree once, calling sort_inner_bst on its inner tree. Therefore there are N calls to sort_inner_bst. At each call to sort_inner_bst, each of the N inner tree nodes are visited once, their value printed as they are visited. There are N nodes in each inner tree. So, each call to sort_inner_bst takes N steps. Therefore in total with have a complexity of O(NxN) or O(N^2).

More formally, let's say that the time to perform sort_forest on a tree of size N is T(N) and the time to perform sort_inner_bst on a tree of size M is S(M). Since M is the same for all of the inner trees, we see that the the time for sort_forest is:

$$T(1) = c$$

$$T(N) = S(M) + 2 \cdot T(\frac{N}{2}) = (2^i - 1) \cdot S(M) + 2^i \cdot T(\frac{N}{2^i}) = (N-1) \cdot S(M) + N \cdot c$$

and

$$S(1) = k$$

$$S(M) = k + 2 \cdot T(\frac{M}{2}) = (2^i - 1) \cdot k + 2^i \cdot T(\frac{N}{2^i}) = (N-1) \cdot k + N \cdot k$$

$$S(M) = O(M)$$

and therefore, knowing that N == M,

$$T(N) = O(N \cdot M) = O(N^2)$$

Question 10. [8 MARKS]

After each of the following statements, write either `Yes` or `No`, and a brief explanation, in the space provided.

1. Radix sort can only be performed on sequential lists, not on linked lists.

2. When applied to an already sorted list, quick sort shows its worst-case complexity. When applied to a reverse-sorted list, quick sort shows its best-case complexity.

3. A complete binary tree is also an AVL tree.

4. For a complete binary tree represented in memory as an array, if there is a node at index $4 \times i + 3$ it must be a child of a child (grandchild) of the node at i.

1. No. Radix sort can be performed on both sequential and linked lists.
2. No. Both a sorted and reverse sorted list cause worst-case performance.
3. No. A complete binary *search* tree would be an AVL tree, but a binary tree has no constraints on ordering and so may violate the binary search tree property.
4. Yes. The right child of node i is $2 \cdot i + 1$ and therefore its right child would be $2(2 \cdot i + 1) + 1 = 4 \cdot i + 3$.