problem 1 (r-3.3)

temporary matrix를 사용하지 않고 transpose를 구해야 한다. 그래서 temporary matrix 대신에 행렬의 어떤 값 하나만을 따로 저장해둘 변수를 설정해놓고 transpose를 구한다.

```
int i, j, n; // 몇 행 몇 열의 행렬인지 n으로 결정한다.
float temp;
float arr[MAX][MAX]; // MAX는 n의 최댓값

for (i=0;i<n;i++) {
        for (j=0;j<i;j++) {
                temp = arr[i][j];
                arr[i][j] = arr[j][i];
                arr[j][i] = temp;
        }
}
```

problem 2 (c-3.11)

```
NODE *current, *next, *previous;

current = head;
next = NULL;
previous = NULL;

while(current != NULL) {
        next = current->next;
        current->next = previous;
        previous = current;
        current = next;
}
head = previous;
```

next, previous, current라는 3종류의 포인터가 필요하다. current가 바로 reverse되는 대상이다. next와 previous는 reverse를 도와주는 일종의 장치이다.

head를 이용해서 가장 첫 번째 node의 주소를 current에 저장한다.
next와 previous에는 NULL을 저장한다.

current가 NULL이 아닐 때에만 while문을 돌린다. 왜냐하면 current가 NULL일 때는 이미 list의 마지막까지 지나 왔다는 뜻이기 때문이다.

next에 current->next를 저장한다. 왜냐하면 reverse할 것이므로 current->next가 이전 node의 주소로 변경될 것이기 때문이다. 즉, next에는 다음에 reverse할 node를 저장해두는 것이다.

reverse를 해주는 과정이 current->next = previous; 이다.

previous = current; current = next; 라는 과정을 통해서 다음 node를 reverse할 수 있게 만든다.

current == NULL이 되어 while문이 전부 돌아가고 난 뒤에는 마지막으로 head에 previous의 값을 넣어서 방향을 완전히 reverse하게 된다.

problem 3 (c-3.12)

하노이탑을 재귀함수로 구현하는 문제이다.

하노이탑은 더 큰 원판이 더 작은 원판보다 위에 있을 수 없다.
따라서 한 기둥(기둥 a)에 꽂혀있는 원판들을 다른 기둥(기둥 c)으로 옮기기 위해서는 또 다른 기둥(기둥 b)을 거치는 과정이 필요하다.

n개의 disk를 a에서 b를 거쳐 c로 옮기는 함수를 hanoi라고 하자.

맨 아래에 있는 disk 1개와 그 위에 있는 n-1개의 disk를 분리하여 생각한다.

1개를 옮기는 과정 : 1개를 a에서 c로 옮긴다.
n-1개를 옮기는 과정 :
(1) n-1개를 a에서 c를 거쳐 b로 옮긴다.
(2) 1개를 a에서 c로 옮긴다.
(3) n-1개를 b에서 a를 거쳐 c로 옮긴다.

코드로 구현하면,

```
void hanoi(int n, char from, char temp, char to) {

if (n==1)
        printf("disk %d : %c에서 %c로 이동\n", n, from, to);
else {
        hanoi(n-1, from, to, temp);
        printf("disk %d : %c에서 %c로 이동\n", n, from, to);
        hanoi(n-1, temp, from, to);
}
}

int main() {
int n;
scanf("%d", &n);
hanoi(n, 'a', 'b', 'c');

return 0;
}
```

problem 4 (c-3.16)

```
int max(int x, int y) {
 if (x >= y)
        return x;
 else
        return y;
}

int max(int x, int y) {
 if (x <= y)
        return x;
 else
        return y;
}

int toMax(int arr[], int n) {
if (n == 1)
 return arr[0];
else
 return max(arr[n-1], toMax(arr, n-1));
}

int toMin(int arr[], int n) {
if (n == 1)
 return arr[0];
else
 return min(arr[n-1], toMin(arr, n-1));
}
```

총 4개의 함수를 사용한다. max는 x와 y를 비교해서 큰 값을 리턴하고, min은 x와 y를 비교해서 작은 값을 리턴한다.
toMax함수는 배열에 저장되어 있는 값들 중에서 최대의 값을 리턴하는 함수이다.
toMin함수는 배열에 저장되어 있는 값들 중에서 최소의 값을 리턴하는 함수이다.

최대값을 구하는 recursive의 구조를 설명하자면,
arr[0]부터 arr[n-1]까지 있다고 하였을 때 재귀의 과정을 통해
max함수가 쌓이게 되고 arr[0]과 arr[1]의 비교 -> 그 중 큰 값을 arr[2]와 비교 -> 그 중 큰 값을 arr[3]과 비교 -> ..... -> 그 중 큰 값을 arr[n-1]과 비교해서 최대값을 찾는 구조이다. 최소값에 대한 recursive도 작은 값을 찾아가며 마찬가지의 과정을 거친다.

problem 5 (c-3.22)

두 circularly linked list L과 M에 대해서 확인해야할 것은 크게 2가지이다.
1. 둘은 같은 circularly linked list인가?
2. 둘의 cursor는 다른가?

둘이 서로 같은 circularly linked list라는 말의 의미는 사실 하나의 circularly linked list에서 cursor가 가리키는 것만 다르다는 뜻일 것이다.

먼저 각 list의 cursor가 다른지 비교한다. 다르다면 L의 cursor가 가리키고 있는 node와 같은 node가 있는지 M의 cursor를 이용하여 하나씩 다음 node로 넘어가면서 원형으로 연결된 구조를 쭉 탐색한다. 그리고 L의 cursor의 다음 node로 옮겨서 같은 탐색을 진행한다.

이 과정을 통해 L에 있는 모든 node가 M에도 전부 있다면, cursor가 가리키는 node만 다르고 서로 같은 circularly linked list라는 것을 알 수 있을 것이다.