(1.)

```
// Boolean definition
#define TRUE 1
#define FALSE 0

// Recursive version
int chech_recursive(A, left, right)
{
	if(left >= right)
		return TRUE;

	if(A[left] != A[right])
		return FALSE;
	else
		return check_recursive(A, left+1, right-1);
}

// Iterative version
int check_iterative(A, left, right)
{
	while(left < eight)
	{
		if(A[left] != A[right])
			return FALSE;
		left++;
		right--;
	}
	return TRUE;
}
```

(2.)

```
void printback(A, n)
{
	// notice that in a C-like program, the last element is A[n - 1], not A[n]
	if(n > 0)
	{
		printf(“%c”, A[n - 1]);
		printback(A, n - 1);
```

```
    }
    return;
}
```
Actually, we do not care your “Index-Error” if you have made some.

(3.)
```
void merge(X, Y, X, m, n)
{
    int index_x = 0;      // index of X
    int index_y = 0;      // index of Y
    int index_z = 0;      // index of Z

    // the core of “merge”
    // this part usually left some numbers which need to be attached to the end of Z
    while(index_x < m && index_y < n)
    {
        if(X[index_x] <= Y[index_y])
        {
            Z[index_z] = X[index_x];
            index_x++;
        }
        else
        {
            Z[index_z] = Y[index_y];
            index_y++;
        }
        index_z++;
    }

    // after the merge step, X or Y still should have some numbers
    // attach these numbers to the end of Z
    if(index_x < m)      // rest part of X
        while(index_x < m)
        {
            Z[index_z] = X[index_x];
            index_x++;
            index_z++;
        }
```

```
        else        //rest part of Y
            while(index_y < n)
            {
                Z[index_z] = Y[index_y];
                index_y++;
                index_z++;
            }

        return;
}
```

(4.)

```
#define MAX_STACK_SIZE 1000

char stack[MAX_STACK_SIZE];  // the character stack
int top;    // the top pointer of the stack

// the “clear stack” operation
void clear_stack(void)
{
        top = 0;
        return;
}

// the “push” operation
void push(char ch)
{
        stack[top] = ch;
        top++;
        return;
}

// the “pop” operation
char pop(void)
{
        if(top <= 0)
            ERROR(“Stack Empty!”);
        else
```

```
    {
        top--;
        return stack[top];
    }
}

// it's OK if you didn't define the Stack operation as above

// the TextEditor core
// notice that this function process a line at a time
// it returns the parsed line as a string. (that is, a character string)
char *TextEditor(void)
{
    char ch;

    scanf("%c", &ch);  // read first character
    while(ch != '\n')     // terminate if the end of the line is reached
    {
        if(ch == '&')
            clear_stack();
        else if(ch == ''#)
            pop();
        else
            push(ch);

        scanf("%c", &ch);  // read next character
    }
    stack[top] = '\0';
    return stack;    // stack should contain the processed text
}
```

(5.)

a.

outer loop => O(log(n))

inner loop => O(m)

overall => O(m * log(n))

b.

O(log(n))

(6.)
Here, let the right-most element in “stack” be the “Top”.
a.
push(1), stack = {1}, output = “”
push(2), stack = {1, 2}, ouppur = “”
pop(), stack = {1}, ouput = “2”
push(3), stack = {1, 3}, output = “2”
push(4), stack = {1, 3, 4}, output = “2”
pop(), stack = {1, 3}, output = “24”
push(5), stack = {1, 3, 5}, output = “24”
push(6), stack = {1, 3, 5, 6}, output = “24”
pop(), stack = {1, 3, 5}, output = “246”
pop(), stack = {1, 3}, output = “2465”
pop(), stack = {1}, output = “24653”
pop(), stack = {}, output = “246531”
Done, possible.

b.
push(1), stack = {1}, output = “”
pop(), stack = {}, output = “1”
push(2), stack = {2}, output = “2”
push(3), stack = {2, 3}, output = “2”
pop(), stack = {2}, output = “13”
push(4), stack = {2, 4}, output = “13”
push(5), stack = {2, 4, 5}, output = “13”
push(6), stack = {2, 4, 5, 6}, output = “13”
pop(), stack = {2, 4, 5}, output = “136”
Now, we expect the next popped element to be “4”, but the top element is 5.
Thus, it’s impossible.

(7.)
a.
the most expected answer should be ((rear + n - front) mod n))
if you write “(rear > front)? (rear - front): n + rear – front” or a short if-else construct, it’s acceptable.

b.
n + n^2 => O(n^2)
log(n) + n => O(n)
log(n!) is bounded by log(n^n) = n*log(n) => O(n*log(n))
2^n => O(2^n)
since O(2^n) > O(n^2) > O(n*log(n)) > O(n),
2^n > n + n^2 > log(n!) > log(n) + n

c.
A[1, 2] at 4143 => A[1, 0] at 4141
A[2, 6] at 4154 => A[2, 0] at 4148
A[1, 0] ~ A[2, 0] => 4148 – 4141 = 7 element
thus, we know c = 7, which is critical for computing the address of A[6, 3]
we can first compute the address of A[0, 0] and then compute the address of A[6, 3]
A[0, 0] should at (4141 - 7) = 4134
A[6, 3] at (4134 + 6*7 + 3) = 4179

d.

	a * b + c + d * e – f * g	
1.	------	ab*
2.	----------	ab*c+
3.	------	ab*c+de*
4.	----------	ab*c+de*+
5.	-----	ab*c+de*+fg*
6.	----------	ab*c+de*+fg*-

(8.)
Here, let the right-most element in “stack” be the “Top”.
bcde*+a-*, a = 2, b = 2, c = 5, d = 3, e = 7
push(b), stack = {b}
push(c), stack = {b, c}
push(d), stack = {b, c, d}
push(e), stack = {b, c, d, e}
operation * and {b, c, d, e} => pop d and e, d * e = 3 * 7 = 21, push(21),
stack = {b, c, 21}
operation + and {b, c, 21} => pop c and 21, c + 21 = 5 + 21 = 26, push(26),
stack = {b, 26}
push a, stack = {b, 26, a}

operation - and {b, 26, a} => pop 26 and a, 26 - a = 26 - 2 = 24, push(24)

=> stack = {b, 24}

operation * and {b, 24} => pop b and 24, b * 24 = 2 * 24 = 48, push(48)

=> stack = {48}

The result is 48.