W.P.Hwang
CS-CCS-NCTU

MACHINE TEST

INTRO. TO COMPUTERS & PROGRAMMING

FALL 2010

General information

1 Time 2010/1/6 6:30~10:30 pm

2 Score 4 problems: A (25%), B (25%), C (25%), D (25%)

3 General requirements

- Use the stipulated algorithm, if any.
- Comments are not required.
- Each problem has a downloadable sample test and a sample output. Suffice it to run the sample test. However, you shall present a general solution for each problem – any solution tailored for the sample test data will come to nought.
- Be honorable!
  Any activity unrelated to the test such as making friends on Facebook, Google stalking, playing game, etc., is strictly prohibited.

4 Grading policy

Correctness, efficiency, and beauty are the three factors that will influence your scores.

5 Source file name

Problem A	<PID>A.cpp	Note: <PID> is your personal identifier.
Problem B	<PID>B.cpp	
Problem C	<PID>C.cpp	
Problem D	<PID>D.cpp	

The first two lines of each source file shall contain the following information:

// <name> <student ID>

// <compiler used, i.e. VC++ or Dev C++>

Problem A

Good sequence

An unsigned integer n may be split into a sequence of numbers, each containing k adjacent digits of n, for $k = 1,2,\ldots,\text{length}(n)$, where $\text{length}(n)$ denotes the number of digits contained in n.

As an example, for $n = 23232$, we have the following sequences

$k = 1$	$2, 3, 2, 3, 2$	
$k = 2$	$23, 32, 23, 32$	
$k = 3$	$232, 323, 232$	
$k = 4$	$2323, 3232$	← good sequence
$k = 5$	23232	← good sequence

A sequence of numbers is *good* if all numbers in the sequence are distinct. For the preceding example, the sequence for $k = 4$ is good; so is the sequence for $k = 5$, since a single number always constitutes a good sequence.

For this problem, you are asked to write the following function

```
void good(unsigned n);
```

to generate the good sequence that is formed by the *least* number of adjacent digits of n. For examples,

`good(23232);`	yields	`2323 3232`
`good(1234567890);`	yields	`1 2 3 4 5 6 7 8 9 0`
`good(3333333333u);`	yields	`3333333333`

Hints

1 Modify the function **`good`** of HW4 by adding an outermost loop to split the unsigned integer in different ways.
2 Beware of integer overflow for the test case **`good(3333333333u)`**.

Requirement

See the sample output on the next page for the requied output format.

Sample test: See file A.cpp

Sample output

```
2323 3232
1 2 3 4 5 6 7 8 9 0
10935 9350 93500 35000 50000 0
345656 456565 565656 656565 565657
3333333333
```

Problem B

Cycles in permutation

DEFINITION
A *cyclic permutation* is a permutation that contains only one cycle.
A *derangement* is a permutation that contains no cycle of length 1.

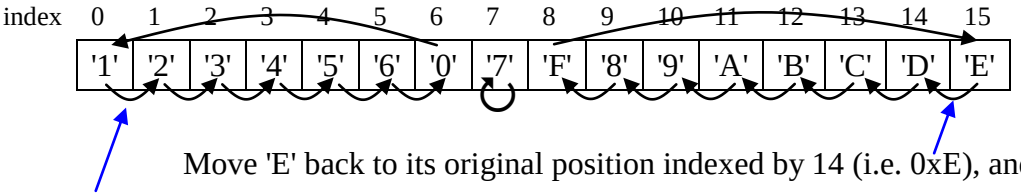
For this problem, you are asked to write the following function

```
void cycle(const char (*a)[16]);
```

Input An array pointed to by **a** that contains a permutation of the 16 hexadecimal digits '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', and 'F'.

Output The cycle notation of the input permutation. Also, indicate whether the input is a cyclic permutation or a derangement or neither.

For example, consider the following permutation



For this permutation, we have
Cycle notation: (1 2 3 4 5 6 0)(7)(F E D C B A 9 8)
Derangement: No, since (7) is a cycle of length 1
Cyclic permutation: No, since there are three cycles

Hint

Modify the function **cycle** of HW6 by mapping the hexadecimal digits '0' .. '9', 'A' .. 'F' to the array indices 0 .. 15. (It is a good idea to define a function for this mapping.)

Requirement

See the sample output on the next page for the requied output format.
In particular, if the input is a cyclic permutation, you need only report so. DO NOT report that it is also a derangement.
(N.B. A cyclic permutation is also a derangement.)

Sample test: See file B.cpp

Sample output

```
(1 2 3 4 5 6 0)(7)(F E D C B A 9 8)
(0)(1)(2)(3)(4)(5)(6)(7)(8)(9)(A)(B)(C)(D)(E)(F)
(1 2 3 4 A B C 0)(6 7 8 9 D E F 5) --- Derangement
(9 0)(A 1)(B 2)(C 3)(D 4)(E 5)(F 6)(8 7) --- Derangement
(B A D 1 3 5 7 9 E C F 2 4 6 8 0) --- Cyclic permutation
```

Problem C

Quaternary strings

A quaternary string is a string of 0's, 1's, 2's and 3's.

Given an integer $n \geq 0$, generate all the quaternary strings of length n in which the number of 0's is even, and count the number of such strings.

For examples, for $n = 2$, there are 10 such quaternary strings, namely,

2 0's: 00

0 0's: 11 12 13 21 22 23 31 32 33

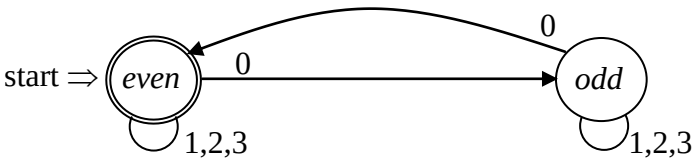
and, for $n = 3$, there are 36 such quaternary strings, namely,

2 0's: 001 002 003 010 020 030 100 200 300

0 0's: 111 112 113 121 122 123 131 132 133
211 212 213 221 222 223 231 232 233
311 312 313 321 322 323 331 332 333

Requirement

Your generator shall base on the following DFA.



The states *even* and *odd* recognize those quaternary strings in which the number of 0's is even and odd, respectively.

You shall write the following two functions:

`int even(int n);`

Staring with state *even*, this function generates all the quaternary strings of length n in which the number of 0's is even, and returns the number of such strings as the function value.

`int odd(int n);`

This function does similar things to function **`even`**, except that it starts with state *odd*.

Since the number of quaternary strings with even number of 0's grows rapidly as n grows larger, we shall only test on $n = 1, 2, 3, 4$. So, you may declare

```
struct stack {
    int top;
    char stk[4];  // assume that the length of quaternary string ≤ 4
};
```

Comment
There are $(2^n + 4^n)/2$ quaternary strings of length n in which the number of 0's is even, as we shall learn in discrete mathematics next semester.

Sample test: See file C.cpp

Sample output

```
1 2 3
3 quaternary strings in total

00 11 12 13 21 22 23 31 32 33
10 quaternary strings in total

001 002 003 010 020 030 100 111 112 113 121 122 123 131 132 133 200 211 212 213
221 222 223 231 232 233 300 311 312 313 321 322 323 331 332 333
36 quaternary strings in total

0000 0011 0012 0013 0021 0022 0023 0031 0032 0033 0101 0102 0103 0110 0120 0130
0201 0202 0203 0210 0220 0230 0301 0302 0303 0310 0320 0330 1001 1002 1003 1010
1020 1030 1100 1111 1112 1113 1121 1122 1123 1131 1132 1133 1200 1211 1212 1213
1221 1222 1223 1231 1232 1233 1300 1311 1312 1313 1321 1322 1323 1331 1332 1333
2001 2002 2003 2010 2020 2030 2100 2111 2112 2113 2121 2122 2123 2131 2132 2133
2200 2211 2212 2213 2221 2222 2223 2231 2232 2233 2300 2311 2312 2313 2321 2322
2323 2331 2332 2333 3001 3002 3003 3010 3020 3030 3100 3111 3112 3113 3121 3122
3123 3131 3132 3133 3200 3211 3212 3213 3221 3222 3223 3231 3232 3233 3300 3311
3312 3313 3321 3322 3323 3331 3332 3333
136 quaternary strings in total
```

Problem D

Singly-linked lists

Enlarge the singly linked list example given in lecture with two functions:

`node* maximum(node* p);` **(10%)**
Given a list pointed to by **`p`**, this function either returns
NULL, if the list is empty, or
a pointer pointing to the node containing the maximum integer of the list.

`node* remove(node* p);` **(15%)**
This function returns a pointer pointing to the list obtained by removing *every other* node in the list pointed to by **`p`**. Put differently, it removes the 1^{st}, 3^{rd}, 5^{th},… elements from the list.
For examples,

$$1\,2\,3\,4\,5\,6 \xrightarrow{\textbf{remove}} 2\,4\,6 \xrightarrow{\textbf{remove}} 4 \xrightarrow{\textbf{remove}} \text{empty list} \xrightarrow{\textbf{remove}} \text{empty list}$$

You need only write the preceding two functions. To activate them, two new commands have already been added to the toy user interface (see file D.cpp):

`m` invokes **`maximum`**
`r` invokes **`remove`**

Suggestion

You may feel free to code the two functions iteratively or recursively. However, it is strongly suggested that the function **`remove`** be coded *recursively*, as it is much easier to code it recursively than iteratively. To this end, you may define two mutually recursive functions:

`node* remove(node* p);`
This function removes the 1^{st}, 3^{rd}, 5^{th},… elements from the list pointed to by **`p`**.

`node* remove2(node* p);`
This function removes the 2^{nd}, 4^{th}, 6^{th},… elements from the list pointed to by **`p`**.

Sample test: See file D.cpp

Sample output

```
Command: i6
Command: i5
Command: i4
Command: i3
Command: i2
Command: i1
Command: p
1 2 3 4 5 6
Command: m
Maximum = 6
Command: r
Command: p
2 4 6
Command: m
Maximum = 6
Command: r
Command: p
4
Command: m
Maximum = 4
Command: r
Command: p

Command: m
No maximum - list is empty
Command:
```