C Programming, Exercises for the fifth week

(Obs! Last two exercises produce more points than the earlier ones.)

1. Represent a suit of a deck of cards as an enumeration (clubs, diamonds, hearts, spades). Define two arrays parallel to this definition, one for input and one for output. The input array should contain one letter codes: 'c', 'd', 'h', 's'. The output array should contain the suit names as strings.

Represent the card values as integers in the range 2...14. The numbers on the cards, called spot values, are entered and printed using the following one-letter codes: '2', '3', '4', '5', '6', '7', '8', '9', 'T', 'J', 'Q', 'K', 'A'. These should be translated to integers.

Represent a card as a structure containing a suit and a spot value. A poker hand will be an array of five cards.

Write the above type definitions. (Hint: Read also the next 3 tasks in order to see the motivation for the definitions.)

2. Write a function

```c
void get_hand(card*);
```

that reads and validates five cards from the keyboard, one card per line. Each card should consist of a two-letter code such as 3H or TS. Permit the user to enter either lower-case or upper-case letters. Return the data through the array of cards.

3. Write a function

```c
void sort_hand(card*);
```

that sorts the five cards in increasing order by spot value (ignore the suits when sorting). Use insertion sort and pointers.

4. Write a function

```c
void print_hand(card*);
```

that displays the five cards in a hand, one card per line. Each card should be printed in its full English form; that is 9 of Hearts or King of Spades.

5. The scoring combinations are listed in the following figure, in increasing order of value:
<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bust</td>
<td>Five cards that include none of the following combinations</td>
</tr>
<tr>
<td>One pair</td>
<td>The hand has two cards with the same spot value</td>
</tr>
<tr>
<td>Two pairs</td>
<td>The hand has two pairs of cards</td>
</tr>
<tr>
<td>Three of a kind</td>
<td>The hand has three cards with the same spot value</td>
</tr>
<tr>
<td>Straight</td>
<td>The spot values of all five cards are consecutive</td>
</tr>
<tr>
<td>Flush</td>
<td>All five cards have the same suit</td>
</tr>
<tr>
<td>Full house</td>
<td>The hand has one pair and three of a kind</td>
</tr>
<tr>
<td>Four of a kind</td>
<td>The hand has four cards with the same spot value</td>
</tr>
<tr>
<td>Straight flush</td>
<td>The suits all match and the spot values are consecutive</td>
</tr>
<tr>
<td>Royal flush</td>
<td>The suits all match and the spot values are T,J,Q,K,A</td>
</tr>
</tbody>
</table>

Write five functions for the following scoring combinations: Bust, One pair, Two pairs, Three of a kind, Straight.

These functions will take a hand as a parameter and return true if the hand has the particular scoring combination that this function is looking for.

6. Do the same for the rest of the scoring combinations.

7. Write a function

   ```c
   int value_hand(card*);
   ```

   that, given a sorted hand, evaluate it using the scoring functions. Return the highest value that applies.

8. Write a main program that reads two hands, calls the evaluation function twice, prints each hand and its value, then says which hand wins. If two hands have the same value, then one one wins. (This is a slight simplification of the rules.)

9. Write a function `maxi()` which takes three parameters, two double values `x` and `y` and a double function `f(double)`, and returns the larger of the two values: `f(x)` and `f(y)`.

10. Write a function `maxi1()` that has four parameters. This function is similar to `maxi()` except it returns the value through the fourth parameter. Test your functions carefully.

11. Write a function

   ```c
   int compareGen(const void *block1, const void *block2, size_t elemSize,
                   size_t block1Size, size_t block2Size,
                   int (*compareIt) (const void*, const void*));
   ```

   which performs a lexicographical comparison of the two blocks. Test your program using blocks of doubles and blocks of pointers to doubles.
12. Implement a generic module *Bags* that represents an unordered collection (duplicates allowed). There should be operations to add an element to the bag, and remove it from the bag. Test your module with a collection of double objects, and a collection with integer objects.

13. Continue with the automata from the exercise four. Write a function that checks if there are unreachable states in the automaton. (Hint: Use depth-first search.)

14. Write a function that checks if there is a closed path or cycle in the automaton. (A path is closed if it starts and ends in the same state.) Hint: Modify depth-first search.