Lab # 4

Collaboration is encouraged. You may discuss the problems with other students, but you must **write up your own solutions**, *including all your C programs, by yourself. If you submit identical or nearly identical solutions to someone else, this will be considered a violation of the code on academic honesty.*

A sorting algorithm is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the use of other algorithms (such as search and merge algorithms) which require input data to be in sorted lists; it is also often useful for organizing data and for producing human-readable output.

Since the dawn of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. For example, bubble sort was analyzed as early as 1956.

In this lab, we'll use `quicksort`, a sorting algorithm developed by C.A.R. Hoare in 1962. Given an array, one element is chosen and the others partitioned in two subsets – those less than the partition element and those greater than or equal to it. The same process is then applied recursively to the two subsets. When a subset has fewer than two elements, it doesn't need any sorting; this stops the recursion.

Consider the following code

```
# include <stdio.h>
# include <stdlib.h>
# include <time.h>

/* swap: interchange v[i] and v[j] */
void swap(int v[], int i, int j)
{
    int temp;
    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

```c
/* quicksort: sort v[left]...v[right] into increasing order */
void sort(int v[], int left, int right)
{
   int i, last;

   /* do nothing if array contains fewer than two elements */
   if (left >= right)
      return;

   /* move "pivot" element to v[0] */
   swap(v, left, (left + right)/2);
   last = left;


   /* partition */
   for (i = left + 1; i <= right; i++)
      if (v[i] < v[left])
         swap(v, ++last, i);

   /* restore "pivot" element */
   swap(v, left, last);
   sort(v, left, last-1);
   sort(v, last+1, right);
}

int main(int argc, char **argv) {

   int i;
   int v[52];
   srand(time(NULL));
   for (i = 0; i < 52; i++) {
      v[i] = 52*((double)rand())/RAND_MAX;
      printf("%d ", v[i]);
   }
   printf("\n");
   sort(v, 0, 51);
   for (i = 0; i < 52; i++) {
      printf("%d ", v[i]);
   }
   printf("\n");
}
```

The main function first constructs a list of 52 pseudo-random numbers. It then calls quicksort to sort them. To generate the pseudo-random numbers, it calls a library function `rand()`. This function must first be initialized with a seed, by calling the function `srand()`. For the seed, the code uses the time of day, obtained with library function `time()`.

When I run it, this code prints out:

```
11  31  10  51  45  24  36  31  21  42  33  13  5
    2  29  32  1  8  51  10  10  36  18  0  11  18
        33  24  32  25  31  21  3  48  46  4  5  40
            51  28  23  41  50  18  33  19  48  31  48  43  21  21
0  1  2  3  4  5  5  8  10  10  10  11  11  13  18
    18  18  19  21  21  21  21  23  24  24  25  28
        29  31  31  31  31  32  32  33  33  33  36
            36  40  41  42  43  45  46  48  48  48  50  51  51  51
```

When you run it, you'll likely see a different random sequence.

Problem

Modify the code so that it produces the following output.

1. A random shuffle of a deck of 52 cards.
2. A sorted deck of 52 cards.

Of course, there is only one card of each type in a deck of cards. Note that the code above produces duplicates; you have to change that. Also, you can't cheat here and just print out a sorted deck directly for Part (2); your code must call the sorting algorithm.

Here's what your code might print out:

```
Random
Q spades
3 hearts
2 diamonds
J hearts
4 clubs
9 diamonds
K hearts
9 spades
10 hearts
```

```
3 diamonds
3 spades
Q clubs
5 diamonds
8 clubs
5 clubs
6 diamonds
J clubs
A hearts
4 hearts
2 clubs
8 diamonds
7 hearts
10 diamonds
7 clubs
7 diamonds
6 hearts
4 diamonds
4 spades
9 clubs
8 hearts
2 hearts
J spades
J diamonds
2 spades
K diamonds
10 spades
K clubs
3 clubs
A clubs
Q hearts
8 spades
A diamonds
6 clubs
A spades
9 hearts
5 spades
Q diamonds
10 clubs
5 hearts
7 spades
```

```
K spades
6 spades

Sorted
A clubs
2 diamonds
3 hearts
4 spades
5 clubs
6 diamonds
7 hearts
8 spades
9 clubs
10 diamonds
J hearts
Q spades
K clubs
A diamonds
2 hearts
3 spades
4 clubs
5 diamonds
6 hearts
7 spades
8 clubs
9 diamonds
10 hearts
J spades
Q clubs
K diamonds
A hearts
2 spades
3 clubs
4 diamonds
5 hearts
6 spades
7 clubs
8 diamonds
9 hearts
10 spades
J clubs
```

```
Q diamonds
K hearts
A spades
2 clubs
3 diamonds
4 hearts
5 spades
6 clubs
7 diamonds
8 hearts
9 spades
10 clubs
J diamonds
Q hearts
K spades
```

For printing out the cards, consider using the switch statement. It is a multi-way decision that tests whether an expression matches one of a number of constant integer values, and branches accordingly.

```
switch (expression) {
   case const-expr: statements
   case const-expr: statements
   default: statements
}
```

Each case is labeled by one or more integer-valued constants or constant expressions. If a case matches the expression value, execution starts at that case. All case expressions must be different. The case labeled default is executed if none of the other cases are satisfied. A default is optional; if it isn't there and if none of the cases match, no action at all takes place. Cases and the default clause can occur in any order.

The break statement causes an immediate exit from the switch. Because cases serve just as labels, after the code for one case is done, execution falls through to the next unless you take explicit action to escape. break and return are the most common ways to leave a switch. Falling through cases is a mixed blessing. On the positive side, it allows several cases to be attached to a single action, as with the digits in this example. But it also implies that normally each case must end with a break to prevent falling through to the next. Falling through from one case to another is not robust, being prone to disintegration when the program is modified. With the exception of multiple labels for a single computation, fall-throughs should be used sparingly, and commented. As a matter of good form, put a break after the last case (the default here) even though it's logically unnecessary. Some day when

another case gets added at the end, this bit of defensive programming will save you.

Here's an example.

```
switch(i) {
   case 0: case 1: case 2: case 3: case 4;
   case 5: case 6: case 7: case 8: case 9;
      printf("number\n);
      break;
   case 10:
      printf("jack\n");
      break;
   case 11:
      printf("queen\n");
      break;
   case 12:
      printf("king\n");
      break;
   default:
      printf("dumbass, there are only 13 cards of each suit\n");
      break;
}
```