You can form a project group of 2 to 3 persons to solve the problem in project.

Project 2 - Set

(Credit: The following questions come from the School of Computing)

The descriptions are based on the compiler gcc and the program runs on the UNIX system. You can run your program on the PC system.

Problem Statement:

Set operations are prevalent in many Science and engineering problem. A set is a collection of **distinct** items. In programming, it is common to represent sets using arrays.

In this lab, write a program that represents an integer set by a one-dimensional array, and performs three common integer set operations: membership test, set intersection operation, and top-k-sum operation.

In this program, we restrict the type of elements that a set can contain to be integers, and the cardinality

(ie., the size) of a set cannot exceed 50.

This task is divided into several levels. Read through all the levels (from first to last, then from last to first) to see how the different levels are related. **You may start from any level.**

♦ ♦ ♦ ♦ ♦

Level 1

Name your program `setOp1.c`.

Write a program that reads in a non-negative integer, *n*, as the size of a set, followed by reading in *n* integers to form a set of size *n*. (In case *n* is 0, there is no need to read in any set element.) Then, print out the content of the set. Since a set is represented by an array, the ordering of elements in the array becomes important (even though this is not necessarily the case for sets in general). In this program, the order of the elements in the array is the same as the order of the integers read in from the user.

For reading in elements of a set from the user, your program should call a procedure named `scan_set` to do the job. The prototype of `scan_set` is as follows:

```
void scan_set(int set[ ], int size) ;
```

The task of printing out a set should be performed by another procedure named `prt_array`, with the following prototype. The formatting detail should be derived from the sample run.

```
void prt_set(int set[ ], int size) ;
```

The following is a sample run of the program. User input is underlined. Ensure that the last line of output is followed by a `n` newline character.

```
$ ./a.out
4
-10 1 56 -87
{-10, 1, 56, -87}.
```

```
$ ./a.out
0
{}.
```

To proceed to the next level (say level 2), copy your program by typing the Unix command

```
cp setOp1.c setOp2.c
```

♦ ♦ ♦ ♦ ♦

Level 2

Name your program `setOp2.c`.

Write a program that reads in an integer set by frist accepting a non-negative integer, *n*, as the size of a set, followed by reading in *n* integers to form a set of size *n*. (In case *n* is 0, there is no need to read in any set element.) Then, print out the content of the set. Since a set is represented by an array, the ordering of elements in the array becomes important (even though this is not necessarily the case for sets in general). In this program, the order of the elements in the array is the same as the order of the integers read in from the user.

After printing out the integer set, your program will read in an integer and check if this integer is a member of the set.

For reading in elements of a set from the user, your program should call a procedure named `scan_set` to do the job. The prototype of `scan_set` is as follows:

```
void scan_set(int set[ ], int size) ;
```

The task of printing out a set should be performed by another procedure named `prt_array`, with the following prototype. The formatting detail should be derived from the sample run.

```
void prt_set(int set[ ], int size) ;
```

To handle membership test, you write a function `is_member` that checks if the integer is a member of the set. Function `is_member` has the following prototype:

```
int is_member(int item, int set[ ], int size) ;
```

The following are some sample runs of the program. User input is underlined. Ensure that the last line of output is followed by a `n` newline character.

Sample 1:

```
$ ./a.out
5
10 20 -30 -40 30
{10, 20, -30, -40, 30}.
30
Item 30 is a member of the set.
```

Sample 2:

```
$ ./a.out
3
1 2 3
{1, 2, 3}.
4
Item 4 is NOT a member of the set.
```

To proceed to the next level (say level 3), copy your program by typing the Unix command

```
cp setOp2.c setOp3.c
```

♦ ♦ ♦ ♦ ♦

Level 3

Name your program `setOp3.c`.

Write a program to perform two set operations: set-membership check and set-intersection operation.

The program will begin by asking the user to enter either a value `1` or a value `2`. If the input provided is 1, the program will perform membership testing; if the input provided is 2, the program will perform set intersection operation.

1. For membership test operation, the program will read an integer set by frist accepting a non-negative integer, *n*, as the size of a set, followed by reading in *n* integers to form a set of size *n*. (In case *n* is 0, there is no need to read in any set element.) Then, print out the content of the set. Since a set is represented by an array, the ordering of elements in the array becomes important (even though this is not necessarily the case for sets in general). In this program, the order of the elements in the array is the same as the order of the integers read in from the user.

   After printing out the integer set, your program will read in an integer and check if this integer is a member of the set.

2. For set intersection operation, the program will read in two integer sets and determine the intersection of these two sets. Reading in each of the sets will require first reading in an integer *n* indicating the size of the set, followed by reading in *n* integers to form the set. Immediately after forming the set, your program should print out the set.

   After reading in two sets, your program should perform set-intersection operation on them, and return and display the intersected set.

   Since the elements of the intersected set occur in both two original sets, and these elements may appear in different orders in these two original sets (which are represented by arrays). In order to eliminate any ambiguity, it is important that *elements in the intersected set are arranged according to their ordering in the* **first** *set*. For instance, if the first set is represented in an array as `{10, 20, 30, 40}` (in that order), and the second set is `{40, 20, 0, -20}`, then the intersect set is represented in an array as `{20, 40}`, and not `{40, 20}`.

For reading in elements of a set from the user, your program should call a procedure named `scan_set` to do the job. The prototype of `scan_set` is as follows:

```
void scan_set(int set[ ], int size) ;
```

The task of printing out a set should be performed by another procedure named `prt_array`, with the following prototype. The formatting detail should be derived from the sample run.

```
void prt_set(int set[ ], int size) ;
```

To handle membership test, you write a function `is_member` that checks if the integer is a member of the set. Function `is_member` has the following prototype:

```
int is_member(int item, int set[ ], int size) ;
```

To perform set-intersection operation, you write a function `intersect` that returns the intersection. Function `intersect` has the following prototype:

```
int intersect(int setA[], int sizeA, int setB[ ], int sizeB, int
setC[]) ;
```

Here, it is important to note that:

1. The intersection set is stored in the parameter `setC`.
2. The size of `setC` is the returned value (of type integer) of this function.

You are encouraged to **reuse `is_member` function** defined earlier in your computation of set intersection.

The following are some sample runs of the program. User input is underlined. Ensure that the last line of output is followed by a `n` newline character.

Sample 1: The sample run chooses 1 to take in a set of 5 integers, and checks if number 30 is a member of the set.

```
$ ./a.out

Enter 1 (membership checking) or 2 (intersection): 1

5

10 20 -30 -40 30

{10, 20, -30, -40, 30}.

30

Item 30 is a member of the set.
```

Sample 2: The sample run chooses 1 to take in a set of 5 integers, and checks if number 4 is a member of the set.

```
$ ./a.out

Enter 1 (membership checking) or 2 (intersection): 1

5

10 20 30 -40 -50

{10, 20, 30, -40, -50}.

4

Item 4 is NOT a member of the set.
```

Sample 3: This sample run chooses 2 to take in two sets and computes the intersection. The last two lines of output comprises the size and the content of the intersect set, respectively.

```
$ ./a.out

Enter 1 (membership checking) or 2 (intersection): 2

5

10 20 -30 40 -50
```

```
{10, 20, -30, 40, -50}.

3

10 20 30

{10, 20, 30}.

2

{10, 20}.
```

To proceed to the next level (say level 4), copy your program by typing the Unix command

```
cp setOp3.c setOp4.c
```

♦ ♦ ♦ ♦ ♦

Level 4

Name your program `setOp4.c`.

Write a program to perform three set operations: set-membership check, set-intersection operation and top-k-sum operation.

The program will begin by asking the user to enter either a value: `1`, `2` or `3`. If the input provided is 1, the program will perform membership testing; if the input provided is 2, the program will perform set intersection operation; if the input provided is 3, the program will perform top-k-sum operation.

1. For membership test operation, the program will read an integer set by frist accepting a non-negative integer, *n*, as the size of a set, followed by reading in *n* integers to form a set of size *n* (In case *n* is 0, there is no need to read in any set element.) Then, print out the content of the set. Since a set is represented by an array, the ordering of elements in the array becomes important (even though this is not necessarily the case for sets in general). In this program, the order of the elements in the array is the same as the order of the integers read in from the user.

   After printing out the integer set, your program will read in an integer and check if this integer is a member of the set.

2. For set intersection operation, the program will read in two integer sets and determine the intersection of these two sets. Reading in each of the sets will require first reading in an integer *n* indicating the size of the set, followed by reading in *n* integers to form the set. Immediately after forming the set, your program should print out the set.

   After reading in two sets, your program should perform set-intersection operation on them, and return and display the intersected set.

   Since the elements of the intersected set occur in both two original sets, and these elements may appear in different orders in these two original sets (which are represented by arrays). In order to eliminate any ambiguity, it is important that *elements in the intersected set are arranged according to their ordering in the* **first** *set*. For instance, if the first set is represented in an array as `{10, 20, 30, 40}` (in that order), and the second set is `{40, 20, 0, -20}`, then the intersect set is represented in an array as `{20, 40}`, and not `{40, 20}`.

3. For top-k-sum operation, the program will read in an integer set and a non-negative integer *k*. It will then find the top *k* biggest integers from the set, and return their sum. If *k* is bigger than the size of the set, the program will display a message "`The set is too small.`".

For reading in elements of a set from the user, your program should call a procedure named `scan_set` to do the job. The prototype of `scan_set` is as follows:

```
void scan_set(int set[ ], int size) ;
```

The task of printing out a set should be performed by another procedure named `prt_array`, with the following prototype. The formatting detail should be derived from the sample run.

```
void prt_set(int set[ ], int size) ;
```

To handle membership test, you write a function `is_member` that checks if the integer is a member of the set. Function `is_member` has the following prototype:

```
int is_member(int item, int set[ ], int size) ;
```

To perform set-intersection operation, you write a function `intersect` that returns the intersection. Function `intersect` has the following prototype:

```
int intersect(int setA[], int sizeA, int setB[ ], int sizeB, int
setC[]) ;
```

Here, it is important to note that:

1. The intersection set is stored in the parameter `setC`.
2. The size of `setC` is the returned value (of type integer) of this function.

You are encouraged to **reuse `is_member` function** defined earlier in your computation of set intersection.

The top-k-sum operation is performed by a function `top_k_sum`. It has the following prototype:

```
int top_k_sum(int k, int set[], int size) ;
```

Here, the first parameter is the non-negative integer *k*. The sum, if found, will be returned by the function. (What if the sum cannot be found? We shall leave the decision to you.)

The following are some sample runs of the program. User input is underlined. Ensure that the last line of output is followed by a `n` newline character.

Sample 1: The sample run chooses 1 to take in a set of 5 integers, and checks if number 30 is a member of the set.

```
$ ./a.out

Enter 1 (membership checking) or 2 (intersection) or 3 (top-k-sum): 1

5

10 20 -30 -40 30

{10, 20, -30, -40, 30}.
```

```
30

Item 30 is a member of the set.
```

Sample 2: The sample run chooses 1 to take in a set of 5 integers, and checks if number 4 is a member of the set.

```
$ ./a.out

Enter 1 (membership checking) or 2 (intersection) or 3 (top-k-sum): 1

5

10 20 30 -40 -50

{10, 20, 30, -40, -50}.

4

Item 4 is NOT a member of the set.
```

Sample 3: This sample run chooses 2 to take in two sets and computes the intersection. The last two lines of output comprises the size and the content of the intersect set, respectively.

```
$ ./a.out

Enter 1 (membership checking) or 2 (intersection) or 3 (top-k-sum): 2

5

10 20 -30 40 -50

{10, 20, -30, 40, -50}.

3

10 20 30

{10, 20, 30}.

2

{10, 20}.
```

Sample 4: The sample run chooses option 3. It then reads in a set of 5 integers, and receive the value of *k* to be 3. Here, the sum of the top 3 integers is 70.

```
$ ./a.out

Enter 1 (membership checking) or 2 (intersection) or 3 (top-k-sum): 3

5

10 20 -30 40 -50

{10, 20, -30, 40, -50}.

3

70
```

Sample 5: This sample run chooses option 3. It then receives a singleton set. It then asks for the value of *k*. Since the value of *k* is 2, which is bigger than the size of the singleton set, it displays the sentence "`The set is too small.`"

```
$ ./a.out

Enter 1 (membership checking) or 2 (intersection) or 3 (top-k-sum): 3

1

100

{100}.

2

The set is too small.
```

THE END