CMPT125, Fall 2022

Homework Assignment 3

Due date: Friday, November 4, 2022, 23:59

You need to solve the first three problems in ***assignment3.c***.
For the fourth problem, you need to declare the struct in ***set_t.h***,
and solve the fourth problem in ***set_t.c***.

Submit all three files to CourSys.

Solve all 4 problems in the assignment.

Grading: The assignment will be graded automatically.
Make sure that your code compiles without warnings/errors, and returns the required output.

Compilation: Your code MUST compile in CSIL with the Makefile provided.
If the code does not compile in CSIL, the grade on the assignment is 0 (zero).
Even if you can't solve a problem, make sure the file compiles properly.

Warnings: Warnings during compilation will reduce points.
More importantly, they indicate that something is probably wrong with the code.

Dynamically allocated arrays: Do not use variable length arrays! Never!
If you need an array of unknown length, you need to use malloc.

Memory leaks: Memory leaks during execution of your code will reduce points.
Make sure all memory used for intermediate calculations are freed properly.

Readability: Your code must be readable, and have reasonable documentation, but not too much. No need to explain i+=2 with // increase i by 2.
Write helper functions if that makes the code more readable.

Testing: An example of a test file is included.
Your code will be tested using the provided tests as well as additional tests.
Do not hard-code any results produced by the functions as we will have additional tests.
You are strongly encouraged to write more tests to check your solution is correct, but you don't have to submit them.

1. You should not add main() to your solution, because it will interfere with main() in the test file.
2. Submit the files ***assignment3.c***, ***set_t.h***, ***set_t.c***, to CourSys.

Problem 1 [15 points]

Write a function that gets an array of strings of length n, and sorts the strings in the array. In the sorted array the strings are arranged as follows:

- *Shorter strings need to come before the longer strings*
- *For two strings of equal length, use the function strcmp() to decide which one needs to come before the other. Specifically, if strcmp(str1, str2)<0, then str1 should be before str2 in the sorted array. if strcmp(str1, str2)>0, then str1 should be after str2 in the sorted array.*

** strcmp(str1, str2) compares two strings according to the first non-matching character.*

- ***strcmp(str1, str2)>0** if the first non-matching character in str1 is **greater** (in ASCII) than that of str2.*
- ***strcmp(str1, str2)<0** if the first non-matching character in str1 is **smaller** (in ASCII) than that of str2.*
- ***strcmp(str1, str2)==0** if the two strings are equal.*

One way to solve it is to implement the comparison function, and then apply qsort() on the array with this comparison function.

```
void sort_strings(const char* A[], int n)
```

<u>For example</u>:

- Suppose arr = [`"Wx"`, `"ab"`, `"Zde"`, `"6_@7h"`, `"7"`, `"hij"`, `"hhh"`, `"b"`]. After running sort_strings() arr should become [`"7"`, `"b"`, `"Wx"`, `"ab"`, `"Zde"`, `"hhh"`, `"hij"`, `"6787h"`]. This is because the ascii value of `'7'` is smaller than the ascii value of `'b'`, and the ascii value of `'Z'` is smaller than the ascii value of `'h'`.

Problem 2 [20 points]

Write the following variant of insertion sort.

```
// The function gets an array of length n of ints,
// and applies the Insertion Sort algorithm on the array.
// The function returns the array of ints of length n
// where output[i] contains the number of swaps
// made by the algorithm in the i'th iteration of the outer loop
int* insertion_sort(int* array, int n);
```

<u>For example</u>: If the input is the array A=[4,3,6,1,2,5], Insertion Sort on A works as follows:

- Insert 4: [4,3,6,1,2,5]: 0 swaps
- Insert 3: [3,4,6,1,2,5]: 1 swaps
- Insert 6: [3,4,6,1,2,5]: 0 swaps
- Insert 1: [1,3,4,6,2,5]: 3 swaps
- Insert 2: [1,2,3,4,6,5]: 3 swaps
- Insert 5: [1,2,3,4,5,6]: 1 swaps

After the execution of the function

(1) A will become [1,2,3,4,5,6]

(2) the function returns [0,1,0,3,3,1].

Problem 3 [20 points]
Write the following four functions:

a) The function gets an array A of length n of ints, and a boolean predicate pred.
It returns the smallest index i such that pred(A[i])==true.
If no such element is not found, the function returns -1.

```
int find(int* A, int n, bool (*pred)(int));
```

b) The function gets an array A of length n of ints, and a boolean predicate pred.
It returns the number of indices i such that pred(A[i])==true.

```
int count(int* A, int n, bool (*pred)(int));
```

c) The function gets an array A of length n of ints, and a function f.
It applies f to each element of A.

```
void map(int* A, int n, int (*f)(int));
```

d) The function gets an array A of length n of ints, and a function f. The function f gets 2 ints and works as follows:

1. Start with accumulator = A[0]
2. For i=1...length-1 compute accumulator=f(accumulator, A[i])
3. Return accumulator

For example, if f computes the sum of the two inputs, then reduce() will compute the sum of the entire array.

```
int reduce(int* A, int n, int (*f)(int,int));
```

Problem 4 [45 points] - **solve the problem in set_t.h and set_t.c**
Define in **set_t.h** *the struct* `set_t` *representing a set of ints, i.e.,* *an unordered collection of ints without duplicates*. You may assume that the size of the set will always be at most 100.
You will need to implement the following functions in **set_t.c***:*

- `set_t* set_create_empty();`
  - *The function returns a pointer to an empty set.*
- `int set_size(set_t* A);`
  - *The function gets a pointer to the set A, and returns the size of the set.*
- `void set_insert(set_t* A, int x);`
  - *The function gets a pointer to the set A and a number x, and adds x to the set. If x was already in the set, the function has no effect on A.*
- `void set_remove(set_t* A, int x);`
  - *The function gets a pointer to the set A and a number x. The function removes x from the set.* Assumption*: x belongs to A.*
- `bool set_contains(set_t* A, int x);`
  - *The function gets a pointer to the set A and a number x. The function returns true if x belongs to A, and returns false otherwise.*
- `int set_map(set_t* A, int (*f)(int));`
  - *The function gets a set A and a function f, and appies f to all elements of A. Note that after applying f some values might become duplicates, in which case you need to remove the duplicates. The function returns the size of the obtained set A.*
- `void set_free(set_t* A);`
  - *The function gets a pointer to the set A, and frees the memory.*

For example:

```
set_t* A = create_empty_set(); // A = {}
for (int i=10;i<15;i++)
    set_insert(A, i);
// here A = {10,11,12,13,14}
set_map(A, plus1); // A becomes {11,12,13,14,15}
set_insert(A, 9);  // A becomes {9,11,12,13,14,15}
set_remove(A, 13); // A becomes {9,11,12,14,15}
set_insert(A, 1);  // A becomes {1,9,11,12,14,15}
set_insert(A, 12); // A doesn't change, 12 was already in A
set_map(A, mod5);  // A becomes {0,1,2,4}
set_free(A);
```

Here `mod5` is defined as: `int mod5(int x) {return x%5;}`
and `plus1` is defined as: `int plus1(int x) { return x+1;}`

* You need to decide how to represent the set using the struct in set_t.h. You may add any data fields you think are needed. The test cases will test that your functions work with each other as specified. The test cases will not check the implementation details.