COMP1511 18s2

Week-13 Laboratory Exercises

Topics

- Exercise-01: Count Odds (individual, warmup)
- **Exercise-02: Print Evens (individual)**
- **Exercise-03: Frequency of Last (individual)**
- **Exercise-04: List Increasing (individual)**
- Exercise-05: Second Last in List (individual, optional)

Preparation

Before the lab you should re-read the relevant lecture slides and their accompanying examples.

We will only assess lab Exercises 02, 03 and 04 to derive marks for this lab. However, you will learn a lot by attempting and solving two additional exercises (02 and 05).

Please also check out **Practice Questions for the Final Exam** . In particular, can you solve the following questions?

- 02: `int identical(struct node *head1, struct node *head2);` // returns 1 if the contents of the two linked lists are identical
- 03: `struct node *copy(struct node *head);` // returns a copy of a linked list
- 04: `struct node *set_intersection(struct node *set1, struct node *set2);`

If not, you may want to get some **hints** from your tutor in your lab.

Getting Started

The first 10 minutes of the lab is set aside for you to complete the myExperience survey for COMP1511. Your tutors will leave the room to ensure your answers stay confidential.

Create a new directory for this lab called `lab13` by typing:

```
$ mkdir lab13
```

Change to this directory by typing:

```
$ cd lab13
```

Exercise-01: Count Odds (individual, warmup)

Download **array_count_odd.c**, the starting code for this exercise.

Your task is to add code to this function:

```
// return the number of odd values in an array.
int count_odd(int length, int array[]) {
    // PUT YOUR CODE HERE (you must change the next line!)
    return 42;
}
```

Add code so that **count_odd** returns the number of odd values in the array.
For example if the array contains these 8 elements:

```
16, 7, 8, 12, 13, 19, 13, 12
```

Your function should return **4**, because these 4 elements are odd:

```
7, 13, 19, 13
```

Testing

array_count_odd.c also contains a simple **main** function which allows you to test your **count_odd** function.

Your **count_odd** function will be called directly in marking. The main function is only to let you test your **count_odd** function

Assumptions/Restrictions/Clarifications.

An odd number is not divisible by 2.
count_odd should return a single integer.

count_odd should not change the array it is given.

count_odd should not call scanf (or getchar or fgets).

count_odd can assume the array only contains positive integers.

count_odd can assume the array contains at least one integer.

count_odd function should not print anything. It should not call printf.

Your submitted file may contain a main function. It will not be tested or marked.

When you think your program is working you can use `autotest` to run some simple automated tests:

```
$ 1511 autotest array_count_odd
```

When you are finished working on this exercise you must submit your work by running **give**:

```
$ give cs1511 wk13_array_count_odd array_count_odd.c
```

Exercise-02: Print Evens (individual)

Write a C program **eof_even.c** which reads integers from standard input until it reaches end-of-input.
It should then print the *even* integers on a single line, in the order they occurred.

Match the the example below EXACTLY.

```
$ dcc eof_even.c -o ./eof_even
$ ./eof_even
1
4
1
5
6
2
6
9
Ctrl-d
4 6 6
$
```

Explanation: given the input **1 4 1 5 6 2 6 9**, your program should print **4 6 6**, as those are the *even* values.

Assumptions/Restrictions/Clarifications.

Your program must read until the end of-input. End of input is signalled on a Linux terminal by typing the **Ctrl** and **d** keys together. This is what `Ctrl-d` indicates in the above examples.
You can assume the input will only only contain positive integers, one per line.

You can assume each line will contain one and only one integer.

You can assume your input contains at least one integer.

You can assume your input contains no more than 10000 integers.

You can assume no integer will be smaller than 1.

You are free to write this program in any way you wish: there is no specific function that you need to implement. Note that your program will need to have a $ma \in$ function.

When you think your program is working you can use `autotest` to run some simple automated tests:

```
$ 1511 autotest eof_even
```

When you are finished working on this exercise you must submit your work by running **give**:

```
$ give  cs1511    wk13_eof_even  eof_even.c
```

Exercise-03: Frequency of Last (individual)

Write a C program **eof_count_last.c** which reads integers from standard input until it reaches end-of-input.
It should then print the number of times the last integer read occurred in in its input.

Match the the examples below EXACTLY.

```
$ dcc eof_count_last.c -o ./eof_count_last
$ ./eof_count_last
1
4
1
5
6
2
6
1
Ctrl-d
3
$
```

Explanation: given the input **1 4 1 5 6 2 6 1**, your program should print **3**, as the last number read was **1** and it occurred 3 times in its input.

```
$ ./eof_count_last
10
20
30
40

Ctrl-d
1
$ ./eof_count_last
10
20
10
20
10
Ctrl-d
3
$ ./eof_count_last
42
42
42
42
42
42
42
Ctrl-d
7
$
```

Assumptions/Restrictions/Clarifications.

Your program must read until the end of-input. End of input is signalled on a Linux terminal by typing the **Ctrl** and **d** keys together. This is what `Ctrl-d` indicates in the above examples.
You can assume the input will only only contain positive integers, one per line.

You can assume each line will contain one and only one integer.

You can assume your input contains at least one integer.

You can assume your input contains no more than 10000 integers.

You can assume no integer will be smaller than 1.

You are free to write this program in any way you wish: there is no specific function that you need to implement. Note that your program will need to have a $ma\in$ function.

When you think your program is working you can use `autotest` to run some simple automated tests:

```
$ 1511 autotest eof_count_last
```

When you are finished working on this exercise you must submit your work by running **give**:

```
$ give cs1511 wk13_eof_count_last eof_count_last.c
```

Exercise-04: List Increasing (individual)

Download **list_increasing.c** here .
Your task is to add code to this function in **list_increasing.c**:

```
int increasing(struct node *head) {

    // PUT YOUR CODE HERE (change the next line!)
    return 42;

}
```

increasing is given one argument, **head**, which is the pointer to the first node in a linked list.
Add code to **increasing** so that its returns 1 if the list is in increasing order - the value of each list element is larger than the element before.

For example if the linked list contains these 8 elements:

```
1, 7, 8, 9, 13, 19, 21, 42
```

increasing should return **1** because is is increasing order

Testing

list_increasing.c also contains a **main** function which allows you to test your **increasing** function.
This main function:

- converts the command-line arguments to a linked list
- assigns a pointer to the first node in the linked list to **head**
- calls **list_increasing(head)**
- prints the result.

Do not change this main function. If you want to change it, you have misread the question.

Your **list_increasing** function will be called directly in marking. The main function is only to let you test your **list_increasing** function

Here is how you use main function allows you to test **list_increasing**:

```
$ dcc list_increasing.c -o list_increasing
$ ./list_increasing 1 2 4 8 16 32 64 128 256
1
$ ./list_increasing 2 4 6 5 8 9
0
$ ./list_increasing 13 15 17 17 18 19
0
$ ./list_increasing 2 4
1
$ ./list_increasing 42
1
$ ./list_increasing
1
```

Assumptions/Restrictions/Clarifications.

increasing should return a single integer.
increasing should not change the linked list it is given. Your function should not change the next or data fields of list nodes.

increasing should not use arrays.

increasing should not call malloc.

increasing should not call scanf (or getchar or fgets).

You can assume the linked list only contains positive integers.

increasing should not print anything. It should not call printf.

Do not change the supplied **main** function. It will not be tested or marked.

When you think your program is working you can use `autotest` to run some simple automated tests:

```
$ 1511 autotest list_increasing
```

When you are finished working on this exercise you must submit your work by running **give**:

```
$ give cs1511 wk13_list_increasing list_increasing.c
```

Exercise-05: Second Last in List (individual, optional)

Download **list_second_last.c** here, the starting code for this exercise.
Note **list_second_last.c** uses the following familiar data type:

```
struct node {
    struct node *next;
    int          data;
};
```

Your task is to add code to this function:

```
int second_last(struct node *head) {

    // PUT YOUR CODE HERE (change the next line!)
    return 42;

}
```

second_last is given one argument, **head**, which is the pointer to the first node in a linked list.
Add code to **second_last** so that its returns the second last element of the list.

For example if the linked list contains these 8 elements:

```
1, 7, 8, 9, 13, 19, 21, 42
```

second_last should return **21** because this is the second last element.

second_last can assume the list has at least two elements.

Testing

list_second_last.c also contains a **main** function which allows you to test your **second_last** function.
This main function:

- converts the command-line arguments to a linked list
- assigns a pointer to the first node in the linked list to **head**
- calls **list_second_last(head)**
- prints the result.

Do not change this main function. If you want to change it, you have misread the question.

Your **list_second_last** function will be called directly in marking. The main function is only to let you test your **list_second_last** function

Here is how you use main function allows you to test **list_second_last**:

```
$ cp -n /web/cs1511/18s1/activities/list_second_last/list_second_last.c .
$ dcc list_second_last.c -o list_second_last
$ ./list_second_last 1 2 4 8 16 32 64 128 256
128
$ ./list_second_last 2 4 6 5 8 9
8
$ ./list_second_last 13 15 17 17 18 19
18
$ ./list_second_last 2 4
2
```

Assumptions/Restrictions/Clarifications.

second_last will be given a list containing at least two elements.
second_last should return a single integer.

second_last should not change the linked list it is given. Your function should not change the next or data fields of list nodes.

second_last should not use arrays.

second_last should not call malloc.

second_last should not call scanf (or getchar or fgets).

You can assume the linked list only contains positive integers.

second_last should not print anything. It should not call printf.

Do not change the supplied **main** function. It will not be tested or marked.

When you think your program is working you can use `autotest` to run some simple automated tests:

```
$ 1511 autotest list_second_last
```

When you are finished working on this exercise you must submit your work by running **give**:

```
$ give cs1511 wk13_list_second_last list_second_last.c
```