Midterm 1 : C Programming

Laurent Michel

September 30, 2021

Grading Structure

The midterm has 4 main questions (Q1..Q4) and one *extra-credit* question (Q5). The grade break down is given as follows:

Questions	Points	Category
Q1	25	core
Q2	10	core
Q3	15	core
Q4	50	core
Q5	10	extra-credit
Total	110	

Note that *each question* is covered by many grading scripts and that you can earn partial credit. For instance, Q1 is covered by 5 scripts each bestowing 5 points. There are 9 scripts for Q4 as well. Some are easier to pass then others.

Second, I want to emphasize once again that the *solutions are short.* If you find yourself writing a substantial amount of code, you are most likely making this too hard on yourself.

Finally, the extra-credit is worth 10% of the midterm total. A perfect grade is therefore a note of 100. If you earn more, that can offset whatever you might have lost in homeworks. A word of caution, **Q5** is also the hardest question on the test.

Q1: Read a String

Read a string of arbitrary length from the standard input. The string is linefeed (`\n`) terminated. Think carefully of where memory is coming from as there is no upper bound on the string length (it could be hundreds of kilobytes). Your program cannot allocate more memory than twice the string length.

- input: none
- ouput: a pointer to a buffer container the string you just read. It ought to be a valid string (`\0` terminated) and it does not contain the terminating `\n`. You should *never* allocate more space than twice the length of the actual string you just read.

It is advisable to correctly handle corner cases like reading an empty string.

Q2: Hamming Distance

Compute the hamming distance between two strings. The simple case in when both strings have exactly the same length. In that case, the hamming distance is the number of indices where the two strings differ. If the strings have different length, then number of differences you can have is upper-bounded by $L = \min(|s_1|, |s_2|) + 1$ and is at least 1 (since the lengths are different). Indeed, all the characters of the shortest string might be different and you pick up one extra difference because the other string is longer.

- inputs:
  - **s1** : pointer to string 1
  - **s2** : pointer to string 2
- output: an integer representing the number of indices where the string disagree
- Note:
  1. This should work for strings of any length.
  2. You can assume that **s1** and **s2** have the same length.
     For instance, the inputs

```
qwerty uiop asdf
qwerty ui0p a3df
```

should yield the output

```
2
```

since two characters disagree. The following example:

```
Univers
Universally
```

Are two strings with "Univers" as the shortest and a prefix of "Universally". Since "Univers" is an exact prefix, the difference between these two is

```
1
```

to account for one string being shorter than the other. Naturally, the other extreme case is, for instance:

```
Hell
Joyous
```

And the difference is 5 since all 4 characters are different and the strings have different length.

Q3: Testing it all...

Write a test program that reads two strings from the standard input (of arbitrary lengths) and compute their hamming distance. The user input is not guaranteed to give strings of the same length!

Your code cannot have memory leaks or memory corruptions. We will check with valgrind.

For the same input example as given above

```
qwerty uiop asdf
qwerty ui0p a3df
```

Your output ought to be

```
hd:2
```

Namely, the string `hd:` following by the calculated Hamming distance. You can't have any leaks or memory corruptions.

If the input is, instead, the following:

```
qwerty
qw3rty for3v3r
```

Then the output ought to be:

```
hd:2
```

to report that the two input strings do not have the same length and in the six characters of the shortest, the third one does not match.

Q4: Subset sum enumeration

You are to solve the classic subset sum problem. Your task is to get on the standard input a set of numbers (integers) and a target sum (integer) and produce on the standard output the list of all the subsets of these numbers that sum up exactly to the target sum.

Specifically, given the following on the standard input

$$n\ s\ a_0\ a_1\ \ldots\ a_{n-1}$$

produce on the standard output the list of subsets as:

$sss : [a_{0_0} \ldots a_{0_k}]$
$\vdots$
$sss : [a_{i_0} \ldots a_{i_j}]$

Note how each subset is prefixed by the string `sss:`. In the above, the output produces $i+1$ subsets with subset 0 containing k numbers from the list $[a_0, \ldots, a_n]$ and the i^{th} subset contains j numbers from the same list.

Assumptions

You can assume the following:

1. The value n is the number of values in the input numbers.
2. The value s is the target sum to reach
3. There are no value repetitions in $a_0 \ldots a_n$.
4. The values $a_0 \ldots a_n$ are given in increasing order.

Requirements

1. You *cannot* output the same subset more than once.
2. Your output should be in increasing lexicographic order
3. You *must* output all the valid subsets
4. You *must* handle *any value* for s, i.e., $s \in \mathbb{Z}$

Example

For the input:

```
4 10 1 2 7 9
```

You ought to be

```
sss:[1 2 7 ]
sss:[1 9 ]
```

You are to write the necessary subroutine to compute and print the subsets as well as the test program that calls it. We provide a bare minimum skeleton.

Q5: XOR list (extra-credit)

XOR List are an interesting variant of the list data structure. They have the same space usage as singly-linked list, yet they can be traversed both *forward* and *backward*. This is achieved with a simple idea. Recall that in a singly-linked list, one stores in each list node a pointer to the next node (`next`) in the list. Likewise, in in doubly-linked list, in each list node, one store a pointer to the previous node (`prev`) and a pointer to the next node (`next`)

In an **XOR list**, the idea is to store in each list node a *link* value which is none other that the exclusive OR between the `prev` and `next` pointers one would have in a doubly linked list. Since pointers are 64-bit values (just like the `long` type), you can easily compute the XOR of two pointers by converting the pointers to a `long` value.

Note how this makes it possible, given a node i to figure out the address of its successor in the list. Indeed

$$i \to link = i \to prev \text{ XOR } i \to next$$

Therefore if you are traversing *forward*, you know the value of $i \to prev$. It is precisely the node your were on before! You can therefore solve this XOR equation to determine the value of $i \to next$ since you know the values of $i \to link$ as well as $i \to prev$. With $i \to next$ in hand, you can *hop* to the next node and repeat the process.

Observe how this works equally well for doing a backward traversal!

Code handout

To make it easier for you, we do provide a startup code base. First we defined 3 data types (structures) to represent, respectively:

1. `Ptr` represents a value that one can see as either a pointer to a node or a `long`.

2. `XNode` is an XOR list node holding a value `val` and a link holding the XOR of the `prev` and `next` one would find in a doubly-linked list.

3. `XList` represents an XOR list with a pair of pointers to the head of the list (first node) and its tail (last node).

The code is shown in **Types**. Armed with these types, you can easily implement a function to

Types

```
typedef union Ptr {
   struct XNode* p;
   long          v;
} Ptr;

typedef struct XNode {
   int  val;
   long link;
} XNode;

typedef struct XList {
   XNode* head;
   XNode* tail;
} XList;
```

allocate and initialize a heap-based instance of an XOR list in Figure **Code**. A test program that

Code

```
XList* createList()
{
   XList* l = (XList*)malloc(sizeof(XList));
   l->head = l->tail = NULL;
   return l;
}
void printForward(XList* l)  { // Beautiful! It's the same code in both directions.
   printXList(l->head);
}
void printBackward(XList* l) {
   printXList(l->tail);
}
```

fills a list with numbers from the standard input and then traverses the list first *forward*, then *backward* is also given in Figure **Test**

Your task

You are to implement a handful of functions to complete this implementation. Specifically you are to implement

Test

```c
int main()
{
   int val;
   XList* l = createList();
   while(scanf("%d",&val) != -1)
          append(l,val);
   printForward(l);
   printBackward(l);
   freeList(l);
   return 0;
}
```

```c
void append(XList* l,int v);
void printXList(XNode* from);
void freeList(XList* l);
```

Together, these 3 functions allow you to:

- add a value at the end of the list
- print the content of list, when starting from the node `from` which is either the `head` or the `tail` of the list.
- release all the memory used by the list (all the nodes and the list itself).