EE 367 Spring 2018
Note: Problem A has been updated

Homework 7 Total Points = 11

Problem 15.1-2 (page 370, 1 pt)

Problem 15.1-3 (page 370, 1 pt)

Problem 15.3-5 (page 370, 1 pt)

Problem 15.3-6 (page 370, 1 pt)

Problem 15.5-4 (page 404, 1 pt)

Problem 15.2 (page 405, 1 pt)

Problem 15-6 (page 408, 1 pt)

Problem 15-11 (page 411, 1 pt)

Problem A (1 pt). Given character strings s[ ] and t[ ], count the number of distinct subsequences of t in s[ ]. A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. For example, "ACE" is a subsequence of "ABCDE" while "AEC" is not.

Here is an example:

s[ ] = "rabbbit", t[ ] = "rabbit".

Return 3

Write a function

int subseq(char s[ ], char t[ ])

which returns the number of distinct subsequences of t in s. Your function must be an implementation of top-down dynamic programming with memoization. Attached is a code file subseq.c that has the function subseq(). Currently, subseq doesn't work. Modify it so it runs properly. You may add new functions. Also, attached are two data files 'target', which contains 'rabbbit', and 'key', which contains 'rabbit'. Turn in your subseq.c file by uploading it into laulima.

Problem B (1 pt). Consider the following function cycle_check that checks if a linked list has a cycle in it as shown in Figure 1. The input to the program is a pointer to the first node in the linked list. It returns -1 if there is no cycle, and returns a positive integer otherwise. For the example in Figure 1, what is the return value of cycle_check if the return value is positive. Your answer may be a function of D and C.

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

int cycle_check(struct node * p)
{
struct node *p1;
struct node *p2;

if (p == NULL) return(0);
p1 = p;
p2 = p;

k = 0;
while (1) {
        /* Increment p1 by 1 node*/
        if (p1 == NULL) return(-1);
        else p1 = p1->next;

        /* Increment p2 by 2 nodes */
        if (p2 == NULL) return(-1);
        else p2 = p2->next;
        if (p2 == NULL) return(-1);
        else p2 = p2->next;

        k++;
        if (p1 == p2) return(k);  /* Stop if both pointers are at the same node */
}
}
```

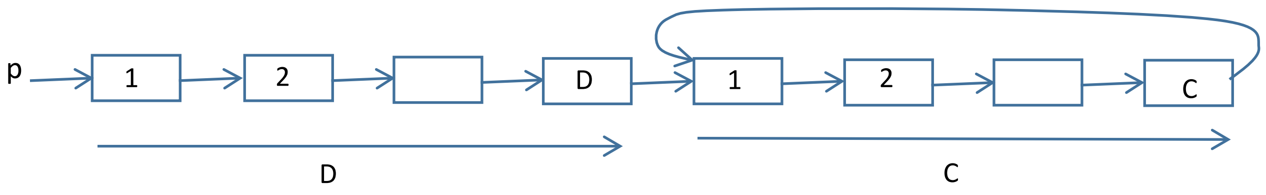

Figure 1. Linked list with D+C nodes, and a cycle with C nodes.

Hint: Note that when p1 reaches the cycle, p2 is already in it. Then subsequently p2 will “catch up” with p1.

Problem C (1 pt). [Bloom Filter] This problem is easy but there's a lot of reading. So bear with me.

The Internet is a distributed file system where you can download files from all over the world such as web pages, movie files, audio files, etc. Files are transferred by having them traverse one or more communication links in the communication network. For example, when a client in Honolulu downloads a file F in London, the file may traverse communication links connecting (London, New York), (New-York, Chicago), (Chicago, Denver), (Denver, Los Angeles), (Los Angeles, Honolulu). If the file F is popular, it is likely that multiple clients in Honolulu will download it. Each of these downloads would use the same set of links.

In order to efficiently use communication links, we could install file servers called *web caches* at each of the cities. Commonly downloaded files are stored at these web caches. When file F is first downloaded by a client in Honolulu, the file is stored in the web cache in Honolulu. Thereafter whenever the file F is requested, it is accessed through the web cache in Honolulu rather than from London. This saves communication link resources, and in particular the communication links from London to Honolulu.

However, it turns out that around three-quarters of downloaded files are never accessed again. These are known as "one-hit-wonders". Obviously, storing these files in a web cache is a waste. To avoid this waste, we could instead cache a file only when it is accessed a second time. To keep track of this, we can use a hash function as follows.

In the web cache in Honolulu, let there be a bit array b[1..m], where m is its size, and initially b is cleared to zero. Let h be a hash function that has range from 1 to m. Whenever a file F is requested by some client in Honolulu, the web cache will compute h(F) and determine if b[h(F)] = 0.

- If b[h(F)] = 0 then file F has not been accessed earlier, and could potentially be a "one-hit-wonder". So F is not cached.
- If b[h(F)] = 1 then file F is assumed to have been accessed earlier and so it is cached.

Then b[h(F)] is set to 1 to indicate that the file F has now been accessed at least once. The hash table is referred to as a *web cache filter*.

Note that b[h(F)] = 1 does not guarantee that the file F has been accessed earlier since the hash function can have conflicts, i.e., there are different keys key1 and key2 such that h(key1) = h(key2). Thus, there is a potential for a *false positive*, i.e., the hash gives a positive indication even though it is false. However, there is no false negative because b[h(key)] = 0 means that the key definitely has not been accessed yet.

Each false positive of a one-hit-wonder increases the cache occupancy, so we should keep this probability reasonably small. The following is some analysis.

First, let there be a sequence of client downloads from Honolulu, where n denotes the number of distinct downloads. Of these n distinct downloads, let 0.75n be one-hit-wonders. Thus, the amount of storage in the Honolulu web cache should be for 0.25n downloads. Of the 0.75n one-hit-wonders, a fraction p of them will result in false positives. They will be stored in web cache, unfortunately. This will be 0.75np. We want this to be small, and so we will design the cache so that this will be 5% of 0.25n. So 0.75np = (0.25)(0.05)n, which `p = 0.0167 (or 1.67%).

Second, we will compute the number of false positives of the n distinct downloads. We will denote these downloads by key(1), key(2), ..., key(n) in the order of their first appearance. Note that key(r) will not have a false positive if none of the previous r-1 keys are hashed into the same slot as key(r). Assuming *simple uniform* *hashing* (see page 259 of the textbook), the probability that key(r) will not have a false positive is $(1-1/m)^{r-1}$. Thus, the probability of a false positive is $1-(1-1/m)^{r-1}$. Then the expected number of false positives is

$$\sum_{r=1}^{n-1}\left[1-(1-\frac{1}{m})^r\right]$$

To simplify this expression, we will assume that m is large, and therefore 1/m is small. Then we can use the approximation $(1-1/m)^r \approx e^{-r/m}$. Also, if r/m is small then $e^{-r/m} \approx 1-r/m$ (from the Taylor series expansion). Note that this will be true if n/m is small, which we will assume. Then

$$\sum_{r=1}^{n-1}[1-(1-1/m)^r] \approx \sum_{r=1}^{n-1}\left[1-e^{-r/m}\right] \approx \sum_{r=1}^{n-1} r/m \approx \frac{n^2}{2m}$$

Since the fraction of these downloads that are false positives is (n/2m), then p = n/2m. Thus, the size of the bit array should be m = n/2p = 30n, where p = 0.0167.

We can generalize the design of the web cache filter by implementing a *bloom filter* rather than a simple hash table. The following is a description of a bloom filter.

A *bloom filter* is composed of a bit array b[1..m], where m is its size; this is the same as an ordinary hash table. However, the filter uses a collection of k hash functions h_1(key), h_2(key, 2),..., h_k(key) that have range from 1 to m. One application of a bloom filter is to identify items in a set S. The following are two useful operations of this data structure: *add* an item x to the set S, and *query* whether there is an item x in the set S.

Initially, the set S is assumed to be empty, and the bit array b[1..m] is cleared to 0. To add a key x to the set S, we set
b[h_1(x)] = 1, b[h_2(x)] = 1, ... , b[h_k(x)] = 1.

The following is an example of operations to a bloom filter. Suppose k = 3, and m = 10. Let the following be the contents of the bit array.

	1	2	3	4	5	6	7	8	9	10
b	0	0	0	0	0	0	0	0	0	0

Add key(1) to S: h_1(key(1)) = 2, h_2(key(1)) = 5, h_3(key(1)) = 8.
Then

	1	2	3	4	5	6	7	8	9	10
b	0	1	0	0	1	0	0	1	0	0

Add key(2) to S: h_1(key(2)) = 3, h_2(key(2)) = 5, h_3(key(2)) = 10.
Then

	1	2	3	4	5	6	7	8	9	10
b	0	1	1	0	1	0	0	1	0	1

Query key = x: $h_1(x) = 2$, $h_2(x) = 5$, $h_3(x) = 9$. Since b[$h_3(x)$] = 0, key = x is not in S.

Query key = y: $h_1(y) = 2$, $h_2(y) = 5$, $h_3(y) = 8$. This is a false positive since all the hash functions lead to bit 1 but y is not in S.

The following is an analysis of false positives. Consider key(r). Assume that the k hash functions will map key(r) to different bit locations (this is a reasonable approximation when k is small compared to the size of the bit array).

Consider one of the bit locations. The probability that none of the previous r-1 keys will hash to the location is $(1 - 1/m)^{k(r-1)} \approx e^{-k(r-1)/m}$. We will assume that k(r-1)/m is small, and so that $e^{-k(r-1)/m} \approx 1 - k(r - 1)/m$. (This will be true if k(n-1)/m is small.) Then the probability that at least one of the previous r-1 keys will hash to the location is approximately $(r - 1)/m$.

We next want to compute the probability that all the k hash values of key(r) have bit value equal to 1, which is the probability of a false positive for key(r). Since the probability that one of hash values leads to 1 is approximately $k(r - 1)/m$, we will approximate the probability that all k hash values equal 1 is $(k(r - 1)/m)^k$. This is the probability of a false positive for key(r).

Then the expected number of false positives is

$$\sum_{r=1}^{n-1} (kr/m)^k \approx \left(\frac{k}{m}\right)^k \sum_{r=1}^{n-1} r^k \approx \left(\frac{k}{m}\right)^k \frac{n^{k+1}}{k+1}$$

where the last approximation comes from approximating the summation with an integral. Let t denote m/n. Then the expected number of false positives is $\left(\left(\frac{k}{t}\right)^k \frac{1}{k+1}\right)n$, and the fraction of false positives is approximately $(k/t)^k \frac{1}{k+1}$. We want this fraction to be at most p = 0.0167 so that the bloom filter performs at least as well as a simple hash table.

Let's set $(k/t)^k \frac{1}{k+1}$ equal to p. Then $t = k/\sqrt[k]{p(k+1)}$, and $m = \left(k/\sqrt[k]{p(k+1)}\right)n$.

Okay, finally we come to the problem.

Using the formula $m = \left(k/\sqrt[k]{p(k+1)}\right)n$ and p = 0.0167, what are the values for m for k = 1 and 2?