DFS LAB – ASSIGNMENT 4

MTech(CS) I year 2018–2019

Deadline: 12 December, 2018

Total: 60 marks

SUBMISSION INSTRUCTIONS

1. Naming convention for your programs: `cs18xx-assign3-progy.c`

   IMPORTANT: Insert a single alpha-numeric string of your choice, 6-8 characters long, in the name given above as shown in the examples below. Think of this string as something like a security password, except that you are not required to remember the string. Examples: `cs1840-assign3-x19jdh4-prog1.c`, `ppo03wws-cs1840-assign3-prog2.c`, `cs1840-assign3-prog2-jsiwm7de.c`

2. To submit a file, go to the directory containing the file and run the following command from your account on the server (IP address: 192.168.64.35)

   ```
   cp -p name-of-your-file ~dfslab/2018/assign3/cs18xx/
   ```

   If you want to submit your files from a computer with a different IP address, run

   ```
   scp -p name-of-your-file \
   mtc18xx@www.isical.ac.in:/user1/perm/pdslab/2018/assign3/cs18xx/
   ```
   (enter your password when prompted).

   To submit all `.c` and `.h` files at one go, use
   `cp -p *.c *.h ~dfslab/2018/assign3/cs18xx/` or similar.

NOTE: Unless otherwise specified, all programs should take the required input from stdin, and print the desired output to stdout.

Q1. Let $S_1, S_2, \ldots, S_M$ be M stacks, with capacities $k_1, k_2, \ldots, k_M$, respectively. Suppose that, initially, all stacks are full, and no further PUSH operations can be done. The objective is to obtain a non-empty subset (not necessarily proper) of the stacks such that all stacks in the subset have the same number of elements. In order to achieve this objective, you may use as many POP operations as you like. Write a program that selects a subset such that the total number of elements in this subset is the maximum possible. If the total number of elements in multiple such subsets is the same, you should choose the subset that has fewer stacks. [10]

Input format: The capacities $k_1, k_2, \ldots, k_M$ will be provided as command-line arguments in order.

Output format: The indices of the stacks that are included in the selected subset, and the number of elements per stack in this subset. Note that stacks are indexed starting from 1.

Sample input 0: `./prog1 1 1 1 1`

Sample output 0: `1 2 3 4 1`

Sample input 1: `./prog1 8 1 4 2`

Sample output 1: `1 8`

A subset containing 8 elements in all can also be obtained by selecting stacks 1 and 3, and popping 4 elements from stack 1. However, this subset contains two stacks, rather than only one.

Sample input 2: `./prog1 100 80 150 120`

Sample output 2: `1 2 3 4 80`

Q2. Write an efficient program that takes a text file, and a list of strings, and for each string, prints the number of words in the text file that start with the given string. You may assume that the two files are small enough to fit in memory simultaneously, but do not make any other assumptions about the maximum length of words or input strings. [25]

Input format: Two file names will be provided as command-line arguments. The first file will contain plain English text (letters, digits, whitespace and punctuation marks). The second file will contain a list of alphanumeric strings, one per line. For this problem, you may assume that a word is defined as any non-empty sequence of letters and digits (not containing any other symbols). Note that case is **not significant** for this problem.

Output format: Your program should print on standard output one line corresponding to each alphanumeric string in the second input file. Each output line should contain two columns: the given string, and the number of words in the first file starting with that given string.

Sample input 0: `./prog2 input.txt strings.txt`

Contents of `input.txt`

```
All work and no play make Anand a dull man.
```

Contents of `strings.txt`

```
a
an
ma
e
```

Sample output 0:

```
a 4
an 2
ma 2
e 0
```

Q3. Your task in this problem is to implement and measure the performance of a *Bloom filter*, a data structure used for inexact searching in sets. Suppose S is a set stored using a Bloom filter. In response to a search for an element x, the Bloom filter returns one of two answers: x is not in S, and I am sure of that, or I think x is in S, but I could be mistaken (this is called a *false positive* error when x is not actually in S). At the end of this question, there is an example from `https://en.wikipedia.org/wiki/Bloom_filter` of a situation where inexact searching using Bloom filters is useful.

How a Bloom filter works[1]**.** An empty Bloom filter is a bit array A of m bits, all set to 0. There are also k different hash functions, say h_1, h_2, …, h_k, each of which maps an element of S to one of the m array positions in a uniformly random manner.

To add an element x to the Bloom filter, the bits $A[h_1(x)], A[h_2(x)], \ldots, A[h_k(x)]$ are each set to 1. To query for an element, say y, i.e., to test whether $y \in S$, we check $A[h_1(y)], A[h_2(y)], \ldots, A[h_k(y)]$. If any of these bits is 0, then $y \notin S$ (if $y \in S$, then all the bits would have been set to 1 when y was inserted). If all are 1, then the Bloom filter returns "y may be in S". If these bits were set by chance to 1 during the insertion of other elements, this would be a false positive result. Intuitively, if m and k are large, the chances of a false positive are small; as more and more elements are inserted into the Bloom filter, the chance of a false positive increases.

Note that, the space required by a Bloom filter to store a set of n items remains fixed at m, while the space required by exact search structures such as balanced search trees (BSTs) usually grows linearly. On the other hand, the probability of a false positive error for a Bloom filter grows with n, whereas it is always zero for BSTs.

Problem statement. The objective of this question is to study the false positive rate vs. space-efficiency tradeoff for Bloom filters.

(a) Given a sequence of non-negative integers (as command-line arguments), possibly with repetitions, insert them in an AVL tree.[2] For each element, report whether it was actually inserted (print INSERTED), or if it was already contained in the tree (print DUPLICATE). Measure the time taken to process the sequence; also, compute the total number of nodes in your final balanced search tree, and thus, estimate the total storage space required (in bytes) for the tree.

(b) Repeat the above exercise with the same sequence, but use the Bloom filter defined below. Note that your output would sometimes be wrong, i.e., your program would print DUPLICATE for some integers that are actually appearing for the first time. As before, measure the time taken to process the sequence; also, compute the false positive error rate per cent.

Bloom filter definition. For a fixed m and k, to insert a non-negative integer x in the Bloom filter, call `srand(x)`; then compute `rand() % m` k times to get k values. Set the bits in these k positions to 1 in the Bloom filter.

[1] `https://en.wikipedia.org/wiki/Bloom_filter`

[2] You may use libraries such as GDSL for this purpose.

NOTE: You may use a byte instead of a bit to store 0 and 1, but you will get full credit only if you store the 0 and 1 at the bit level.

Input format: The values of m and k will be provided as the first two command-line arguments. The remaining command-line arguments comprise the sequence of input numbers.

Output format: Your program should print to stdout a sequence of INSERTED and DUPLICATE that corresponds to the given input. It should also print to stderr the number of bytes and the time taken to process the sequence using an AVL tree, m, k, and the time taken to process the sequence using a Bloom filter.

Example: A Web server can use a Bloom filter to determine whether a Web object (a page, an image, etc.) has been requested earlier. If it has been requested at least once earlier, only then it is stored in a cache.[3] If a cached object is requested again (i.e., thrice or more in all), it can be served quickly from the cache; otherwise, the object is served more slowly from the hard disk where it is stored. The reason for this strategy is that a very large proportion of Web objects are requested only once; there is no benefit to storing such objects in the cache.

One way to implement this strategy would involve storing the identifiers of the requested objects in an exact search structure, such as a balanced search tree (BST). Since Web servers service requests for a very large number of objects, such a BST would be large. In response to the question: "Has object x been requested before?", a BST would always correctly reply YES or NO.

Instead, if the identifiers were stored in a Bloom filter, the amount of space needed would be *much* less. Of course, this space saving would come at a cost. In response to the above question, a NO would be guaranteed to be a correct answer, but a YES would sometimes be mistaken, i.e., occasionally, an object being requested for the first time would appear to have been requested earlier, and would end up being cached unnecessarily. [25]

[3] A *cache* is a faster, but much smaller, storage space that is intended to store a frequently accessed subset of the complete data stored in a larger, but much slower, storage medium.