Indian Statistical Institute

Semester-I 2018-2019

M.Tech.(CS) - First Year

Lab Test IVb (30 November, 2018)

Subject: Data and File Structures Laboratory

Total: 60 marks Duration: 4 hrs.

SUBMISSION INSTRUCTIONS

1. Naming convention for your programs: `cs18xx-test4b-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. When you have finished, copy all your files to `~dfslab/2018/labtest4b/cs18xx/`.

1. **(25 marks)** Let us consider a head-and-belly view of constructing hash tables in an efficient implementation for Robin Hood hashing. We start by placing all elements at their first choices $X_{i,0}, 1 \leq i \leq m$ (where m is the number of bins in the hash table) following the Robin Hood hashing. Note that a bin can be viewed as an array of data items. Initially (call this stage 1), some bins in the table may have many elements, but that is acceptable. Within a bin, data items are stored in the order of their insertion in the bin, with newer data items getting lower indices, and the older ones getting higher indices. At the kth stage in our construction, picture a hash table (the *head*) containing elements of age k, possibly many per bin, and a second hash table (the *belly*) containing at most one element per bin, and that element is of age less than k. Furthermore, if bin i in the *head* is occupied, then bin i is empty in the *belly*. This head-and-belly view allows us to proceed, by letting k grow until finally the *head* is empty, and all elements are in the *belly*.

   The *belly* is initially empty, and all elements are in the *head*, in stage one. Given the situation of $(k-1)$th stage, we construct the kth stage as follows (see the subsequent figure).

   A. All elements in the $(k-1)$-stage *head* that are in position one in their bins (i.e, the most recently inserted item in each bin) move to the corresponding bin in the $(k-1)$-belly.

   B. Each of the remaining elements in the $(k-1)$-stage *head* (i.e., all but the most recently inserted item in that bin) move to a randomly selected bin in the k-head (recall that we start from $k = 1$). Conflicts are resolved within the k-head as they arise. However, this may create some conflicts with the $(k-1)$-belly just created.

   C. While there is a conflict between the k-head and $(k-1)$-belly, take a conflicting element in the *belly* (i.e., an element in bin i, such that the k-head also has an element in bin i), and let it start hopping uniformly and randomly (and aging by one with each hop), according to the rules of Robin Hood hashing, until it, or the element it causes to move, finds a position in a bin by itself in the *belly*, without conflict with the k-head. If in this process it reaches age k, it must move to corresponding bin in the k-head. In the latter case, a new conflict may be triggered within the k-head. At the end of this, there is no further conflict between head and belly, and the resulting tables are called the k-head and the k-belly.

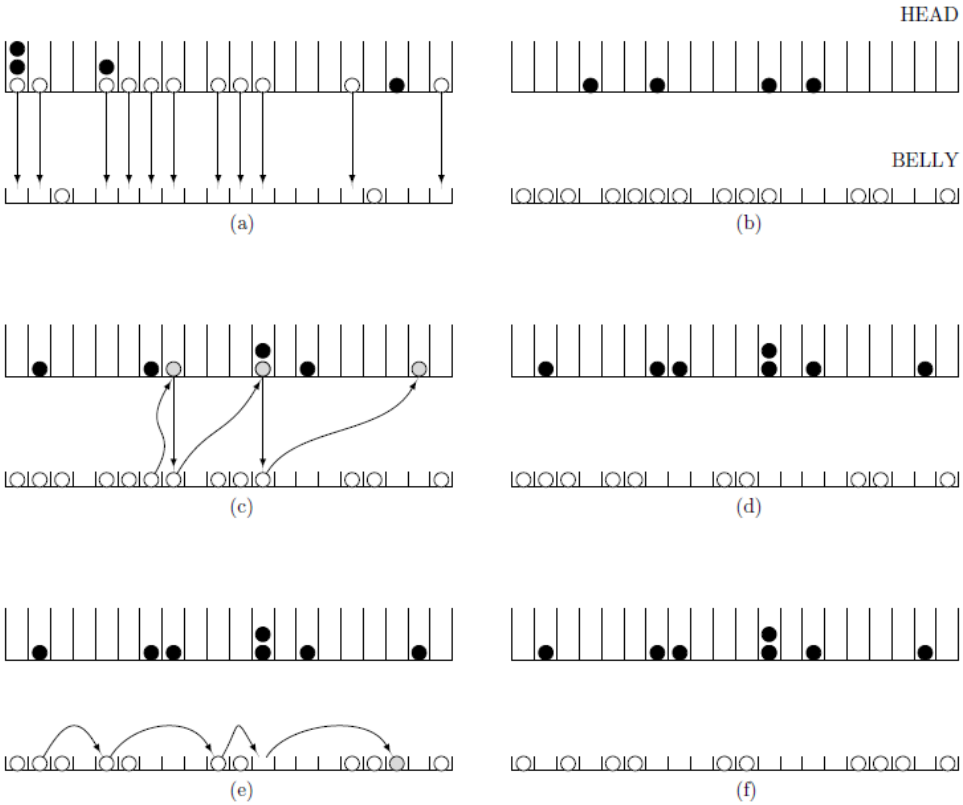


Figure 1: (a) A $(k-1)$-head that is not empty. The elements in position one (shown in white) move to the belly (step B). The elements in position other than one in their bins (shown in black) move to a random position in the k-head (step A). (b) The k-head and k-belly after the steps A and B. Clearly, there are some conflicts between head and belly. (c) For each conflict (resolved using step C), an element in the belly is taken and is moved to a random position in the belly. We show the moves of an element, as it first ages to age k (so that its randomly picked position lands it in the head), which triggers a new conflict in the belly, which is immediately taken care of by letting that element move to a random position, which again happens to be in the head (gray element), and finally, the last conflict generated leads to yet another element in the head, causing no further conflicts. (d) The resulting configuration (after applying step C). (e) The last remaining conflict is taken care of by random hops, resulting in the final configuration (f) of the k-belly and k-head. In example (e), all hops remain in the belly, and result finally in a bin in the belly being filled with a new element.

Write a program that takes the size of hash tables (number of bins in the *head* and *belly*), maximum number of elements that can be accommodated in each bin of the *head*, and a set of data items as user inputs, and returns the index of corresponding bins of the *head* and *belly*, in which the data items are finally placed.

To simplify the problem, let us assume whenever there is a necessity to move a data item to a randomly selected position (as per the algorithm), we will choose the next available slot following the circular right shift (hence both the algorithm and the corresponding figure get simplified). Moreover, consider that the hash function which is applied, to obtain the first stage of head-and-belly view, is given by $h(k) = k\%m$, where, k and m denote the data item and size of the hash table, respectively.

Input Format Input will be provided via standard input in the following format. The first line of input consists of two integers, namely the size of hash tables and the number of bins in each bin of the *head*. It follows by a set of integers to be hashed into.

Output Format Output is to be printed on the standard output in the following format. The output will print the index (starts from 0) of corresponding bins in the *head* and *belly* in which the data items get hashed into. If a data item is finally placed into the *head*, then print the indices preceding with a '+', otherwise if it is placed into the *belly* print the indices preceding with a '-'. If there is a conflict that cannot be resolved, then print '*'. In case of bin overflow, print an error message and terminate the program.

Sample Input 0

```
10 2
14 700 21 37 52 68 999 6 83 25
```

Sample Output 0

```
-4 -0 -1 -7 -2 -8 -9 -6 -3 -5
```

Sample Input 1

```
10 2
77 11 43 89 15 13 51 27 35 79
```

Sample Output 1

```
-8 -2 -4 -0 -6 -3 -1 -7 -5 -9
```

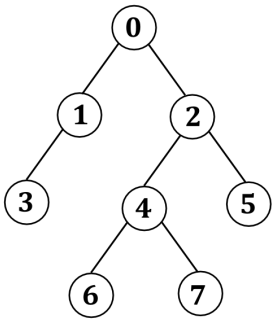
Sample Input 2

```
10 3
22 904 28 7 18 788 355 51 73 26
```

Sample Output 2

```
-2 -4 -0 -7 -9 -8 -5 -1 -3 -6
```

2. **(20 marks)** Suppose a first-n binary tree (FnBT) is defined as a binary tree having n nodes that contains the first n non-negative integers as its data items. Consider a special representation of an FnBT that uses an array A of length n. If the value stored in $A[i]$ is j, this indicates that the node with value j is the parent of the node with value i. For the root node, the parent is denoted by '-1'. Thus, the FnBT corresponding to the array representation $A = \{-1, 0, 0, 1, 2, 2, 4, 4\}$ looks like the following figure.



Suppose we want that each of the data items in an FnBT must be painted with one of three available colors. However, we need to obey the following rules:

- No two successive data items (e.g., i and $i+1$) can have the same color.
- The parent and its children data items cannot have the same color.

Write a program to find out the minimal and maximal number of data items that are painted with a single color, if the tree is colored according to the given rules.

Input Format Input will be provided via standard input in the following format. The input consists of a set of integers corresponding to the elements of the array A in the order of their appearance.

Output Format Output is to be printed on the standard output in the following format. The output prints the minimal and maximal number of data items of the FnBT that may be painted with a single color. If the input is not an FnBT then it simply prints a '-1'.

Sample Input 0

```
-1 0 0 1 2 2 4 4
```

Sample Output 0

```
2 4
```

Sample Input 1

```
-1 0 0 1 1 2 2 3 3 4 4 5 5 6 6
```

Sample Output 1

```
5 5
```

Sample Input 2

```
-1 0 0 1 2 2 2 4
```

Sample Output 2

```
-1
```

3. **(15 marks)** In computational biology, fragment assembly is an approach that helps to assemble the overlapping fragment sequences (of DNA, RNA, protein, etc.) from a sequencing project. Suppose, there was an error in a protein sequencing experiment that lead to the generation of non-overlapping protein fragments. We have to find out whether a set of fragments belong to a given protein sequence or not, and if so, how many minimal fragments does this involve. To simplify the problem, we mark the protein fragments with integers and pose it as an integer construction problem. This simplified version is stated below.

   Let us assume that we have a set of distinct integers S (that corresponds to the protein fragments). Given a sufficiently large integer (say I, corresponding to a protein sequence) as user input, we have to figure out the minimum number of elements from S that can form I by concantenation. Write a program that will take the elements of S and I as user inputs and efficiently return the minimum number of elements required from S to constitute I.

   Input Format Input will be provided via standard input in the following format. The first line of input consists of two integers, namely the number of elements in S and the integer I. It follows by one more input line. This line comprises the elements of S.

   Output Format Output is to be printed on the standard output in the following format. The output simply prints the minimum number number of elements required from S to constitute I. Note that, if $I \in S$, then it prints 1. If no combination of elements in S can constitute I then it prints 0.

   Sample Input 0

   ```
   12 1234567
   84 1234 214 567 123 234 3456 45 268 456 67 12
   ```

   Sample Output 0

   ```
   2
   ```

   Explanation: The integer 1234567 can be obtained by combining the two elements in {1234, 567} and the three elements in {123, 45, 67}, taken from the given integers. So, the minimum count is 2.

   Sample Input 1

   ```
   10 22456789
   47 22 13 56 45 24 67 58 89 79
   ```

   Sample Output 1

```
4
```

Explanation: The integer 22456789 can be obtained by combining the four elements in {22, 45, 67, 89}, taken from the given integers. So, the count is 4.

Sample Input 2

```
15 14159265358979323846
1 4259 2654 5 897 932 384 1 4159 28 53 8 898 933 38 46
```

Sample Output 2

```
0
```

Explanation: The integer 14159265358979323846 can be obtained in no way from the given integers. So, the count is 0.