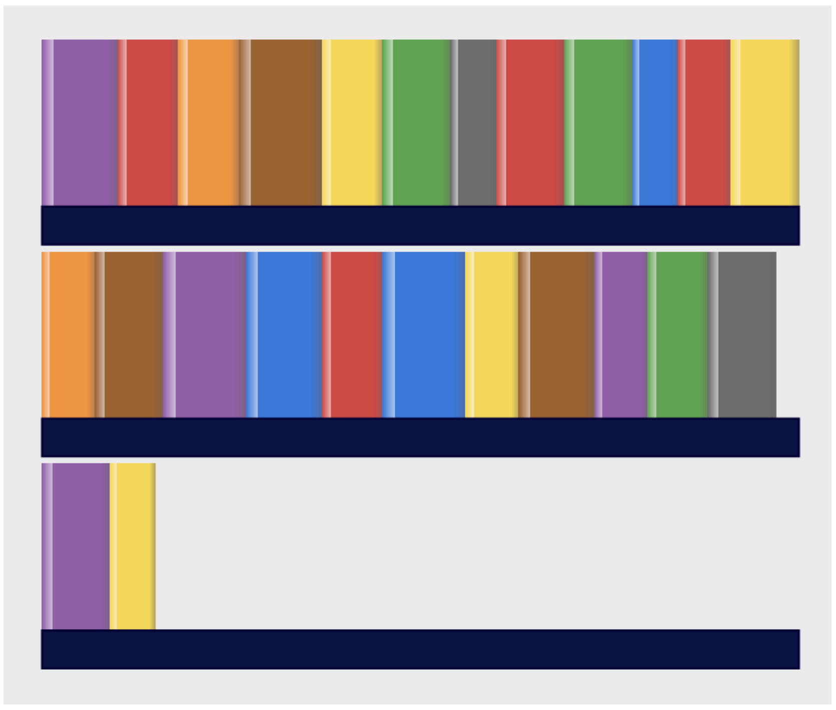
Library Sorting

Imagine that you have a library in which you place books in sorted order.

If you placed your books tightly next to each other, your library might look like the following:

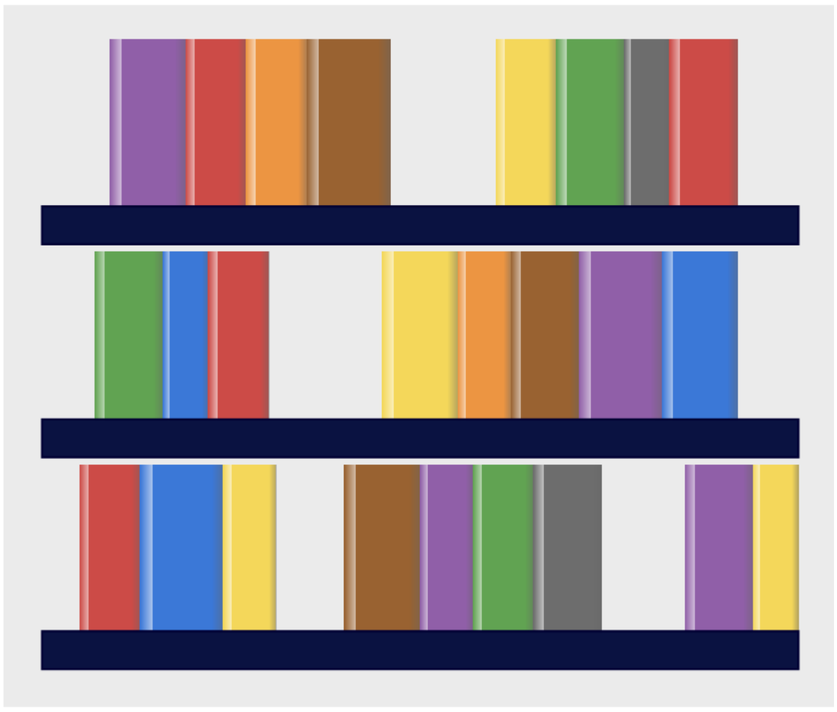


But think about what would happen if you acquired a book whose correct position is in the middle of the top shelf.

In that case, you would need to move one or more books from the middle shelf to another shelf, and accordingly, books from the top shelf to the middle shelf, in order to make room to place the book in its correct position.

Therefore, inserting just one book might require many movements — in the worst-case scenario, if the book must be placed at the far left end of the top shelf, then all the books in the library would need to be moved.

To avoid this, we could be more proactive and place the books in the library in sorted order, but leave some gaps between them, as shown in the diagram below:



This situation doesn't occur only in libraries. If we want to store data in sorted order, and also want to be able to insert additional data into the correct position while minimizing data movement each time, we can adopt a similar approach by designing an appropriate data structure: a sparse table.

The sparse table will have capacity m, and we will be able to store n genuine keys in it, where $m > n$. The remaining $m - n$ positions in the table will store dummy keys. We describe the table as:

$y = [y_0, y_1, \ldots, y_{m-1}]$, where $y_0 \leq y_1 \leq \ldots \leq y_{m-1}$

Thus, the genuine key or the dummy key y_i is at position i in the table, with $0 \leq i < m$.

If the genuine keys of the table are:

$y_{i_0} < y_{i_1} < \ldots < y_{i_{n-1}}$, with $0 \leq i_0 < \ldots < i_{n-1} = m - 1$

then for each genuine key y_{i_t}, we store dummy keys in the successive positions that span from the previous genuine key $y_{i_{t-1}}$ up to y_{i_t}, as follows:

$i_{t-1} + 1, i_{t-1} + 2, \ldots, i_t - 1$, for $0 \leq t < n$, with $i_{-1} = -1$

That is:

$y_0 = \ldots = y_{i_0} < y_{i_0+1} = \ldots = y_{i_1} < \ldots < y_{i_{n-1}+1} = \ldots = y_{i_{n-1}}$

For example, in the table below, the genuine keys are stored at positions 2, 5, 6, and 8:

$y = [2, 2, 2, 3, 3, 3, 4, 5, 5]$, $n = 4$, $m = 9$

From here on, we will consider the table as circular, meaning that after position $m - 1$ we loop back to position 0, and all operations are performed modulo m. The first key in the table is not necessarily at position 0; we must maintain a pointer that always shows the position of the first key. We will call this pointer the head.

The sparse table is constructed as keys are inserted into it. The table size is defined by two sequences of numbers:

n_k, which contains natural numbers, and

m_k, which contains real numbers:

$\{n_k : 0 \le k < \infty\}$ and $\{m_k : 1 \le k < \infty\}$

where:

$1 = n_0 < n_1 < \ldots < n_k < \ldots$, and $n_k < n_{k-1} \times m_k$ for $k = 1, 2, \ldots$

Specifically, the size m will be:

$m = n_{k-1} \times m_k$ when the number n of genuine keys in the table satisfies:

$n_{k-1} < n \le n_k$ for $k > 1$, or

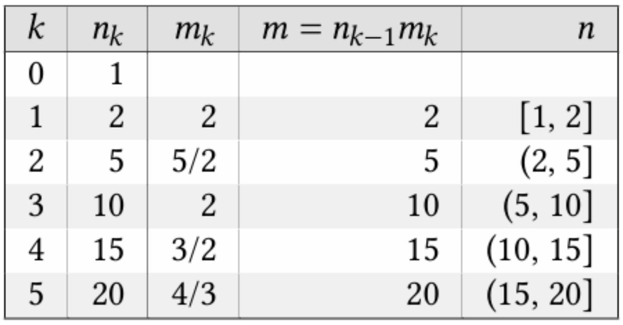
$n_{k-1} \le n \le n_k$ for $k = 1$

For example, suppose:

$n_k = \{1, 2, 5, 10, 15, 20\}$

$m_k = \{2, 5/2, 2, 3/2, 4/3\}$

Then, the table size and the keys that can be stored might be:



k	n_k	m_k	$m = n_{k-1}m_k$	n
0	1			
1	2	2	2	[1, 2]
2	5	5/2	5	(2, 5]
3	10	2	10	(5, 10]
4	15	3/2	15	(10, 15]
5	20	4/3	20	(15, 20]

To insert a key x into a table of size $m = n_{k-1} \times m_k$, which currently contains

n genuine keys where $n_{k-1} < n < n_k$, we proceed as follows:

Using binary search, we find the position s such that $y_{s-1} < x < y_s$.

If y_s is a dummy key, we simply replace it with x, and the table becomes:

$[y_0, \ldots, y_{s-1}, x, y_{s+1}, \ldots, y_{m-1}]$

If y_s is a genuine key, and the next t keys are also genuine:

$y_s \neq y_{s+1} \neq \ldots \neq y_{s+t-1} \neq y_{s+t} = y_{s+t+1}$

Then these t genuine keys are shifted one position to the right (circularly), we update the head of the table accordingly, and x is inserted at position s, resulting in:

$[y_0, \ldots, y_{s-1}, x, y_s, \ldots, y_{s+t-1}, y_{s+t+1}, \ldots, y_{m-1}]$

If we want to insert a key x into a table $m = n_{k-1} \times m_k$, which already contains n_k genuine keys, then we rebuild the table. We increase the size of the table to:

$m' = n_k \times m_{k+1}$

and insert the existing n_k genuine keys:

$y_{i_0} < y_{i_1} < \ldots < y_{i_{nk-1}}$

evenly across the enlarged table. Then, we perform a binary search in the new table and insert x as before.

To distribute the n_k genuine keys evenly across the table, we calculate the quotient and remainder:

$q = \lfloor m' / n_k \rfloor$

$r = m' \bmod n_k$

This means we can distribute the keys across n_k intervals in the table, where each interval will initially have size q. However, we’ll have r leftover slots, which we want to distribute as evenly as possible among the intervals.

If we are at interval i, then by the start of the interval we should have distributed:

$\lfloor (i + 1) \times r / n_k \rfloor$

So, if:

$\lfloor i \times r / n_k \rfloor < \lfloor (i + 1) \times r / n_k \rfloor$

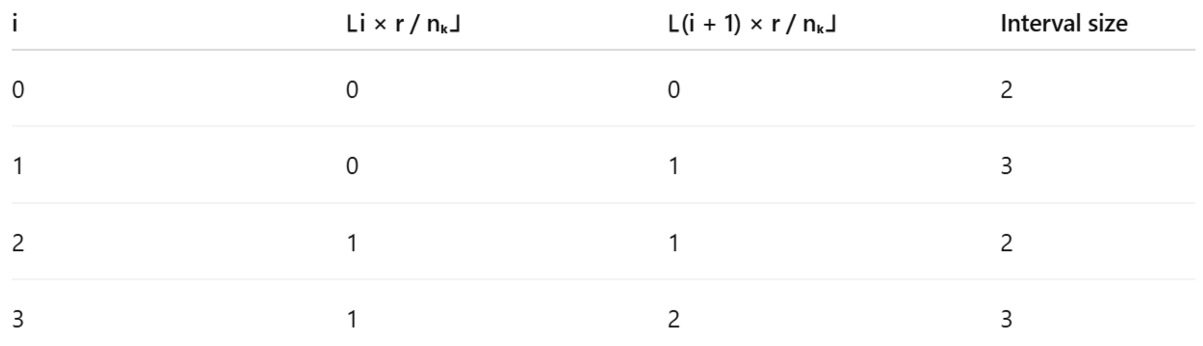
then interval i will have size q + 1 instead of q.

Example:

If we have $m' = 10$ and $n_k = 4$, then:

$q = 2$,

$r = 2$,



i	$\lfloor i \times r / n_k \rfloor$	$\lfloor (i + 1) \times r / n_k \rfloor$	Interval size
0	0	0	2
1	0	1	3
2	1	1	2
3	1	2	3

Deleting a Key

To delete a key from the table, we must again take care of the dummy keys and rebuild the table if necessary. Suppose we want to delete key x from the table:

$y = [y_0, y_1, \ldots, y_{nk-1} \times m_{k-1}]$

We perform a binary search and find the position s of the first appearance of key x in the table (since it may be followed by dummy keys):

$y_{s-1} < x = y_s$

Suppose that dummy keys follow, such that the next genuine key is L positions ahead (circularly):

$y_s = y_{s+1} = \ldots = y_{s+L-1} \neq y_{s+L}$

Then we replace those dummy values:

$[y_s, y_{s+1}, \ldots, y_{s+L-1}]$

with the value of the next genuine key:

$[y_{s+L}, y_{s+L}, \ldots, y_{s+L}]$ (L−1 times)

This gives the new table:

$y' = [y_0, y_1, \ldots, y_{s-1}, y_{s+L}, y_{s+L}, \ldots, y_{s+L}, y_{s+L+1}, \ldots, y_{n_{k-1} \times m_{k-1}}]$

Now, if after deleting a key, the number of genuine keys drops to n_{k-2}, then we must rebuild the table again. We reduce the table's size to $n_{k-2} \times m_{k-1}$, and insert the n_{k-2} genuine keys evenly into the new smaller table

Purpose of the Assignment

The purpose of this assignment is to implement sparse tables according to the description above.

Program Requirements

The use of external graph libraries or implementations of algorithms is not allowed,unless explicitly permitted.

The use of built-in Python data structures such as stacks, dictionaries, sets, etc., is allowed.

The following libraries or their components may be used:

The program must be written in Python 3.

The program output must match exactly what is shown in the provided examples. Verbose output will not be rewarded.

Final Program

The program will be executed as follows (using the appropriate command depending on your system):

```
python library_sorting.py test
```

The **test** argument refers to the name of a JSON file that contains instructions for building and using the sparse table.

This JSON file follows a specific schema (see JSON Schema documentation for details). Note that the array **nn** corresponds to $\mathbf{n_k}$ and the array **mm** corresponds to $\mathbf{m_{k-1}}$, so that the table does not begin with a gap.

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "description": "Schema for describing sparse table test runs.",
  "properties": {
    "nn": {
      "description": "Array of n_k integers.",
      "type": "array",
      "items": {
        "type": "integer"


      }
    },
    "mm": {
      "description": "Array of m_{k-1} real numbers.",
      "type": "array",
      "items": {
        "type": "number"
      }
    },
    "k": {
      "description": "Integer indexing nn and mm.",
      "type": "integer"
    },
    "x": {
      "description": "Initial key to insert in the table.",
      "type": "integer"
    },
    "actions": {
      "type": "array",
      "description": "List of actions to perform on the sparse table.",
      "items": {
        "type": "object",
        "properties": {
          "action": {
            "type": "string",
            "enum": ["insert", "lookup", "delete"]
          },
          "key": {
            "type": "number"
          }
        },
        "required": ["action", "key"],
        "additionalProperties": false
      }
    }
  },
  "required": ["nn", "mm", "k", "x", "actions"],
  "additionalProperties": false
}
```

Examples

<u>Example 1</u>

If the user runs the program with the following command:

python library_sorting.py example_1.json

then your program should read the file example_1.json and **must display exactly** the following output.
Note that the table displays its **head** between the characters ><.

```
  CREATE with k=1, n_k=[1, 2, 5, 10, 15], m_k=[2, 2.5, 2, 1.5,
   ↪  1.33], key=3
  [>3<, 3]
  INSERT 5
  [5, >3<]
  INSERT 6
  [6, >3<, 5, 5, 5]
  INSERT 3
  [6, >3<, 5, 5, 5]
  INSERT 4
  [6, >3<, 4, 5, 5]
  INSERT 10
  [6, 10, >3<, 4, 5]
  INSERT 8
  [>3<, 3, 4, 4, 5, 5, 6, 6, 8, 10]
  INSERT 7
  [10, >3<, 4, 4, 5, 5, 6, 6, 7, 8]
  LOOKUP 10
  Key 10 found at position 0.
  [10, >3<, 4, 4, 5, 5, 6, 6, 7, 8]
  LOOKUP 15
  Key 15 not found. It should be at position 1.
  [10, >3<, 4, 4, 5, 5, 6, 6, 7, 8]
  LOOKUP 5
  Key 5 found at position 5.
  [10, >3<, 4, 4, 5, 5, 6, 6, 7, 8]
  LOOKUP 0
  Key 0 not found. It should be at position 1.
  [10, >3<, 4, 4, 5, 5, 6, 6, 7, 8]
  DELETE 3
  [10, >4<, 4, 4, 5, 5, 6, 6, 7, 8]
  DELETE 10
  [>4<, 4, 4, 4, 5, 5, 6, 6, 7, 8]
  DELETE 10
  [>4<, 4, 4, 4, 5, 5, 6, 6, 7, 8]
  DELETE 4
  [>5<, 5, 5, 5, 5, 5, 6, 6, 7, 8]
  DELETE 5
  [>6<, 6, 6, 6, 6, 6, 6, 6, 7, 8]
  LOOKUP 6.5
  Key 6.5 not found. It should be at position 8.
  [>6<, 6, 6, 6, 6, 6, 6, 6, 7, 8]
  LOOKUP 6
```

```
Key 6 found at position 4.
[>6<, 6, 6, 6, 6, 6, 6, 6, 7, 8]
DELETE 6
[>7<, 7, 8, 8, 8]
DELETE 7
[>8<, 8]
```

Good luck!