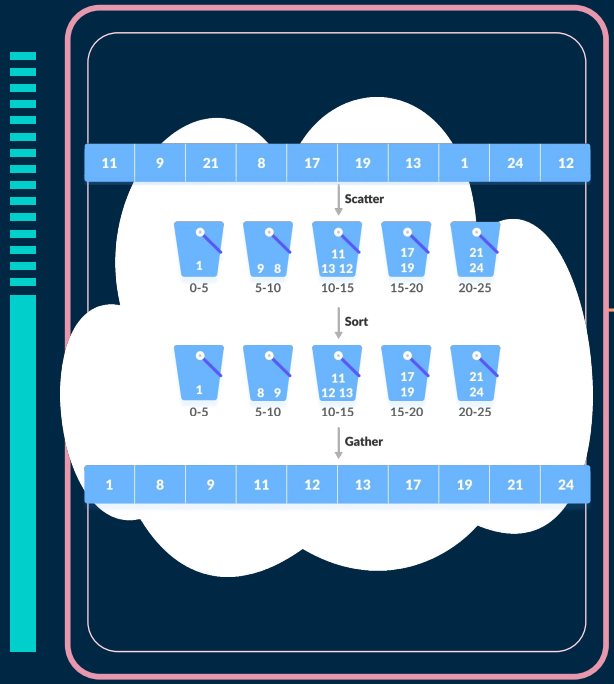
RADIX & BUCKET SORTING ALGORITHM

BAGUIO | GO | JUMALON | PEROL

Bucket Sorting

- Bucket Sort is a sorting algorithm that divides the unsorted array elements into several groups called buckets.
- Scatter-Gather approach
- Out-of-place algorithm
- Both stable and unstable sort



In-place

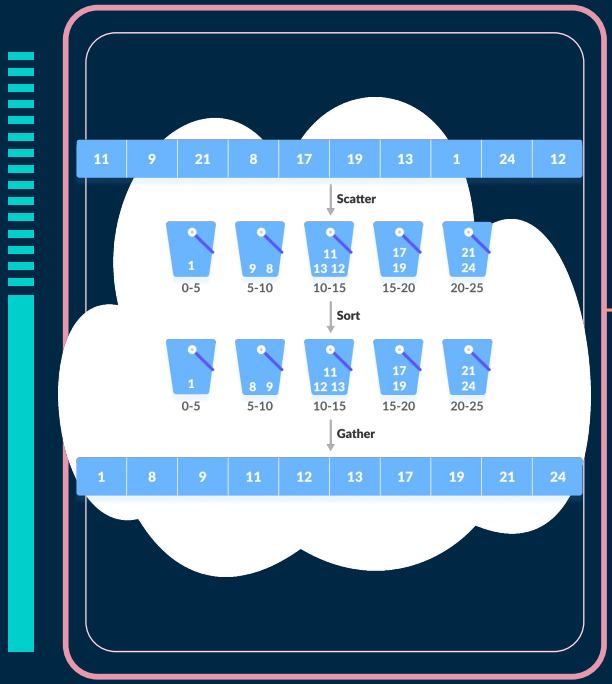
- Transforms the input without using any extra memory.
- Amount of memory required must not be dependent on the input size and should be constant.

Out-of-place

- the extra space used by an out-of-place algorithm depends on the input size.

Bucket Sorting

- Bucket Sort is a sorting algorithm that divides the unsorted array elements into several groups called buckets.
- Scatter-Gather approach
- Out-of-place algorithm
- Both stable and unstable sort

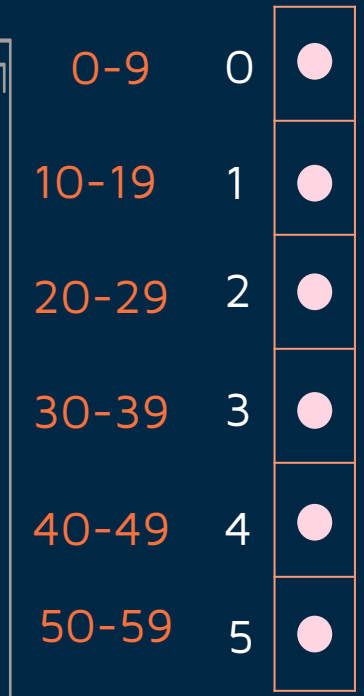


BUCKET SORT

Algorithm

1. Create an empty array of size n (empty buckets)

2	53	25	47	58	34	23	59



Determining the range:

⌈ Max - Min / N ⌉

Where:
Max - maximum element (array)
Min - minimum element (array)
N - number of buckets

Stable

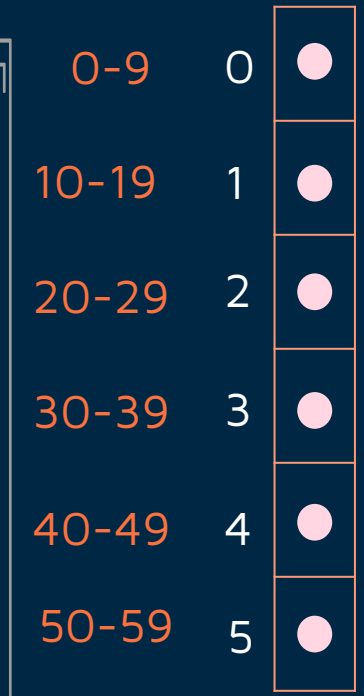
Unstable

BUCKET SORT

Algorithm

1. Create an empty array of size n (empty buckets)

2	53	25	47	58	34	23	59



Determining the range:

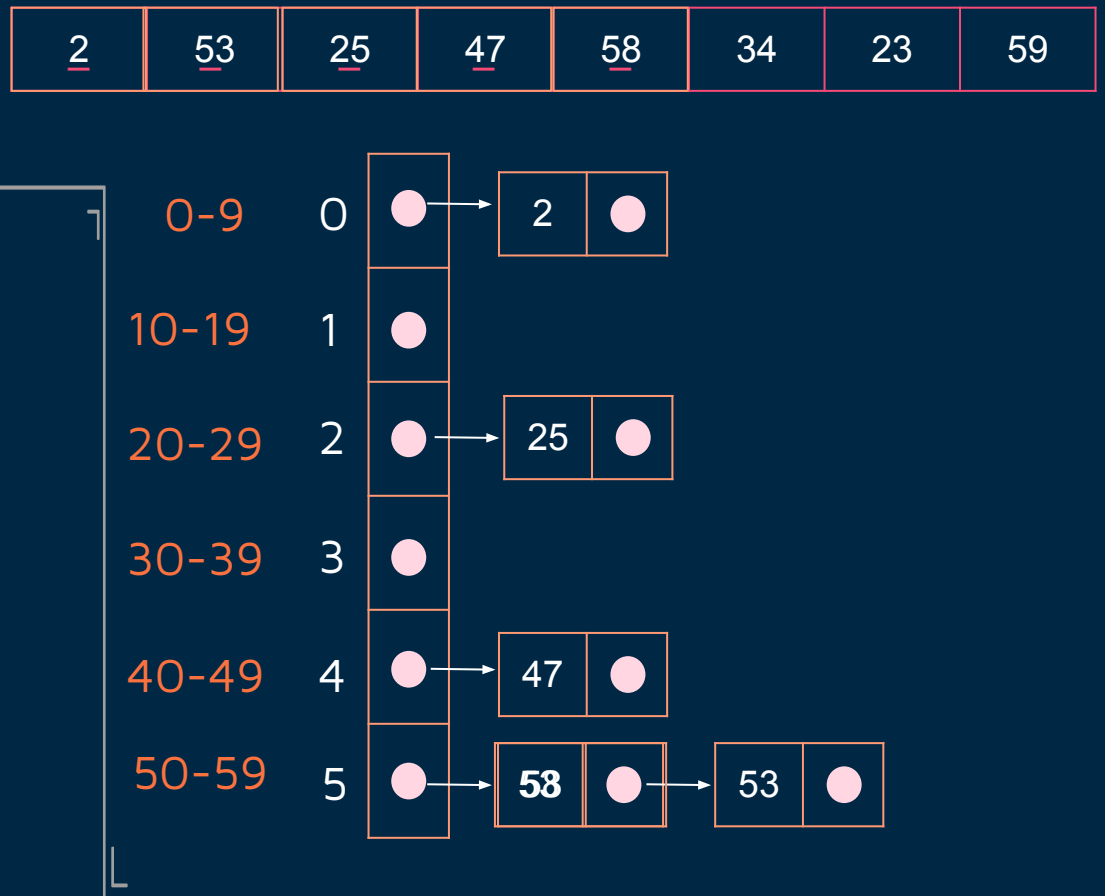
⌈ Max - Min / N ⌉

Where:
Max - maximum element (array)
Min - minimum element (array)
N - number of buckets

BUCKET SORT

Algorithm

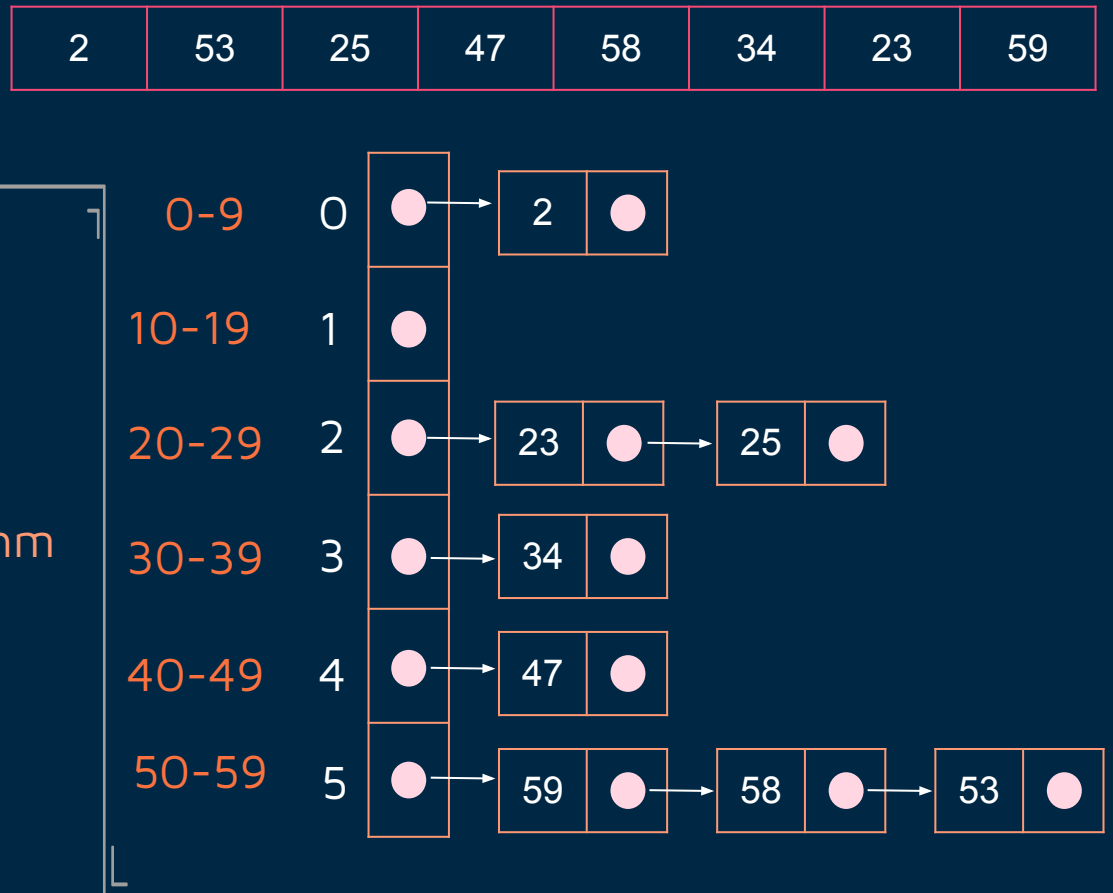
1. Create an empty array of size n (empty buckets)
2. Loop through the original array and put each array element in a “bucket”



BUCKET SORT

Algorithm

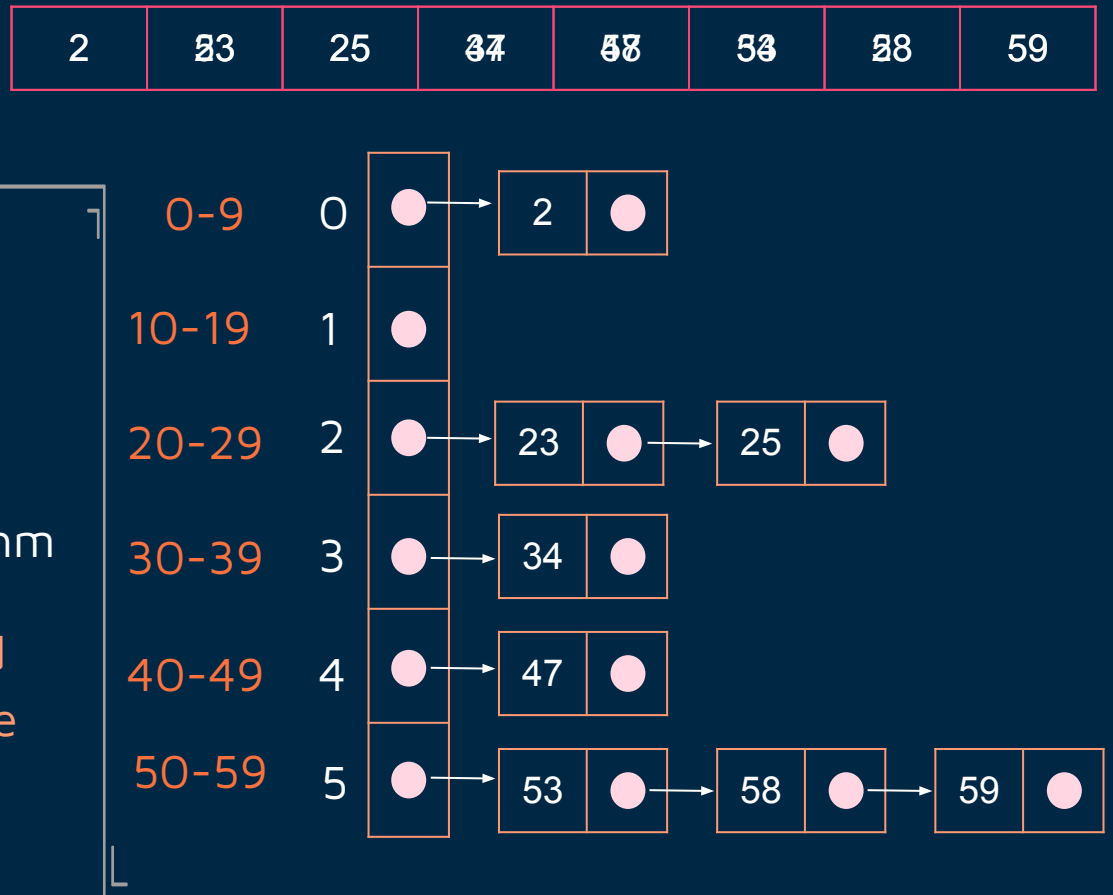
1. Create an empty array of size n (empty buckets)
2. Loop through the original array and put each array element in a “bucket”
3. Sort each of the non-empty buckets using sorting algorithm



BUCKET SORT

Algorithm

1. Create an empty array of size n (empty buckets)
2. Loop through the original array and put each array element in a “bucket”
3. Sort each of the non-empty buckets using sorting algorithm
4. Visit the buckets in order and put all elements back into the original array



Internet Code vs Group Code

Internet Code

Group Code

1. Create an empty array of size n (empty buckets)

```
#define NARRAY 7    // Array size
#define NBUCKET 6   // Number of buckets
#define INTERVAL 10  // Each bucket capacity

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

```
// Create buckets and allocate memory size
buckets = (struct Node **)malloc(sizeof(struct Node *) * NBUCKET);

// Initialize empty buckets
for (i = 0; i < NBUCKET; ++i) {
  buckets[i] = NULL;
}
```

```
int arr[] = {2,53,25,47,58,34,23,59};
```

```
#include <stdio.h>
#include <stdlib.h>
#define BUCKET_SIZE 10
#define rangePerBucket 10
#define MAX 10


typedef struct node{
    int data;
    struct node* link;
} List , *Listptr;

void bucketSort(int A[], int n);
```

```
Created Buckets
Bucket[0]:
Bucket[1]:
Bucket[2]:
Bucket[3]:
Bucket[4]:
Bucket[5]:
Bucket[6]:
Bucket[7]:
Bucket[8]:
Bucket[9]:
```

```
// Create an empty array of size  BUCKET_SIZE (empty buckets)
bucket =  (Listptr*)malloc(sizeof(Listptr) * BUCKET_SIZE );

for(i=0; i<BUCKET_SIZE; i++){
    bucket[i]=NULL;
}
```

Internet Code

Group Code

2. Loop through the original array and put each array element in a “bucket”

```
// Fill the buckets with respective elements
for (i = 0; i < NARRAY; ++i) {
  struct Node *current;
  int pos = getBucketIndex(arr[i]);
  current = (struct Node *)malloc(sizeof(struct Node));
  current->data = arr[i];
  current->next = buckets[pos];
  buckets[pos] = current;
}
```

```
int getBucketIndex(int value) {
  return value / INTERVAL;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#define BUCKET_SIZE 10
#define rangePerBucket 10
#define MAX 10


typedef struct node{
    int data;
    struct node* link;
} List , *Listptr;

void bucketSort(int A[], int n);
```

```
Bucktets before sorting
Bucket[0]: 2
Bucket[1]:
Bucket[2]: 23 25
Bucket[3]: 34
Bucket[4]: 47
Bucket[5]: 59 58 53
Bucket[6]:
Bucket[7]:
Bucket[8]:
Bucket[9]:
```

```
// Loop through the original array and put each array element in a “bucket”

for(i=0; i < n ; i++){

    pos = arr[i] / rangePerBucket;

    // Insert First Linklist
    temp = (Listptr)malloc(sizeof(List));
    temp->data = arr[i];
    temp->link = bucket[pos];
    bucket[pos] = temp;

}
```

Internet Code

Group Code

3. Sort each of the non-empty buckets using sorting algorithm

```
// Sort the elements of each bucket
for (i = 0; i < NBUCKET; ++i) {
  buckets[i] = InsertionSort(buckets[i]);
}
```

```
#include <stdio.h>
#include <stdlib.h>
#define BUCKET_SIZE 10
#define rangePerBucket 10
#define MAX 10


typedef struct node{
    int data;
    struct node* link;
} List , *Listptr;

void bucketSort(int A[], int n);
```

```
Bucktets after sorting
Bucket[0]: 2
Bucket[1]:
Bucket[2]: 23 25
Bucket[3]: 34
Bucket[4]: 47
Bucket[5]: 53 58 59
Bucket[6]:
Bucket[7]:
Bucket[8]:
Bucket[9]:
```

```
//  Sort each of the non-empty buckets using sorting algorithm

for(i=0; i<BUCKET_SIZE;i++){
    bucket[i] = InsertionSort(bucket[i]);
}
```

Internet Code

Group Code

4. Visit the buckets in order and put all elements back into the original array

```
// Put sorted elements on arr
for (j = 0, i = 0; i < NBUCKET; ++i) {
  struct Node *node;
  node = buckets[i];
  while (node) {
    arr[j++] = node->data;
    node = node->next;
  }
}
```

```
int arr[] = {2,53,25,47,58,34,23,59};
```

```
// Visit the buckets in order and put all elements back into the original array

for(i=0, j=0; i<BUCKET_SIZE ; i++){
    trav = &bucket[i];
    while(*trav != NULL){
        temp = *trav;
        arr[j++] = temp->data;
        *trav = temp->link;
        free(temp);
    }

}
```

```
Sorted Array
Array[0]: 2
Array[1]: 23
Array[2]: 25
Array[3]: 34
Array[4]: 47
Array[5]: 53
Array[6]: 58
Array[7]: 59
```

Time and Space Complexity

	Average	Best	Worst	Space
Bucket Sorting	O(n+k)	O(n+k)	O(n^2)	O(n)

BUCKET SORT VARIATIONS

BUCKET SORT VARIATIONS

- Postman's Sort
  - An algorithm that takes advantage of hierarchical structures of elements
  - Similar to radix sort

- Histogram Sort
  - Checks the number of elements that will be in each bucket using a count array
  - Also known as counting sort

BUCKET SORT VARIATIONS

- Proxmap Sort
  - Uses a "map key" function that preserves a partial ordering on the keys

- Shuffle Sort
  - Begins by removing the first 1/8 of the n items to be sorted, sorts them recursively, and puts them in an array

PROXMAP SORT

GIVEN

7	5	1	9	3	9	4	9	2	6	8	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12

Determine how many keys will map to the same subarray, using an array of "hit counts,", H

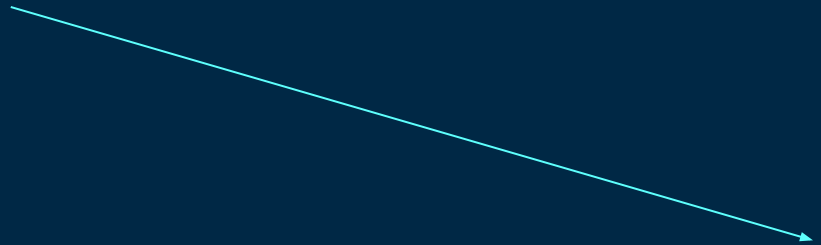
H

0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12

PROXMAP SORT

GIVEN

7	5	1	9	3	9	4	9	2	6	8	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12

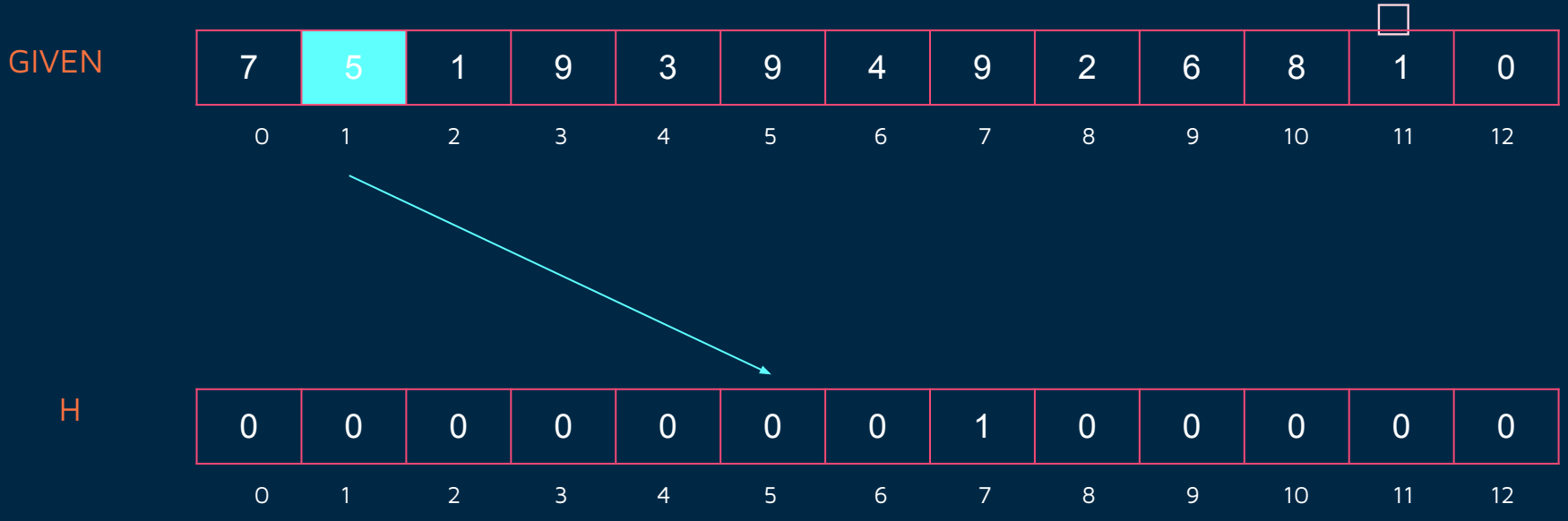


H

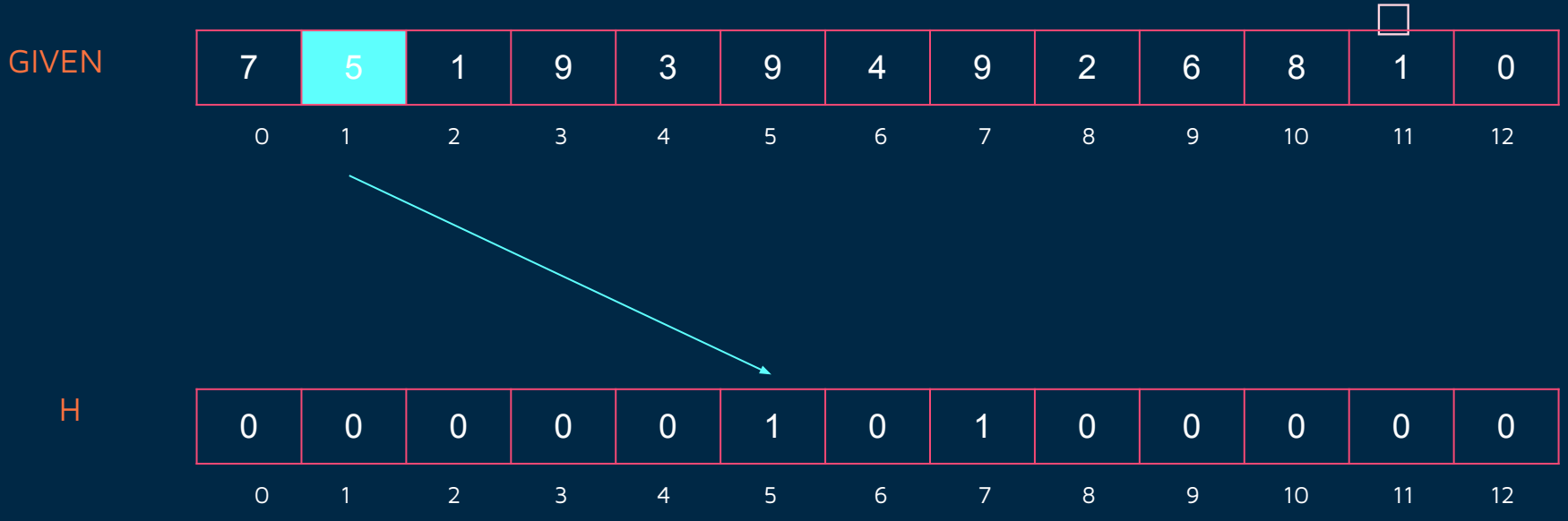
0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12

PROXMAP SORT

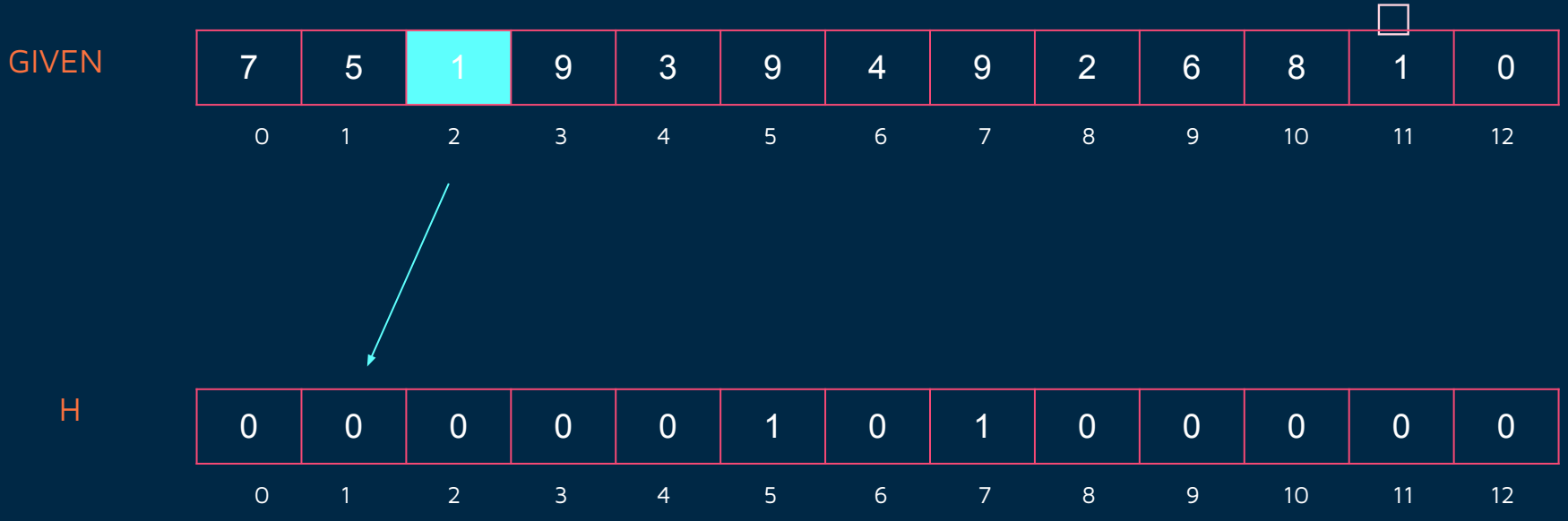
PROXMAP SORT



PROXMAP SORT



PROXMAP SORT



PROXMAP SORT

GIVEN

7	5	1	9	3	9	4	9	2	6	8	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12

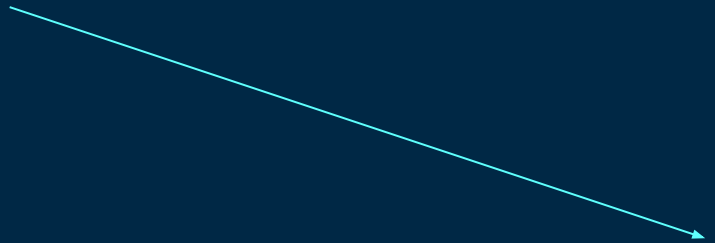
H

0	1	0	0	0	1	0	1	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12

PROXMAP SORT

GIVEN

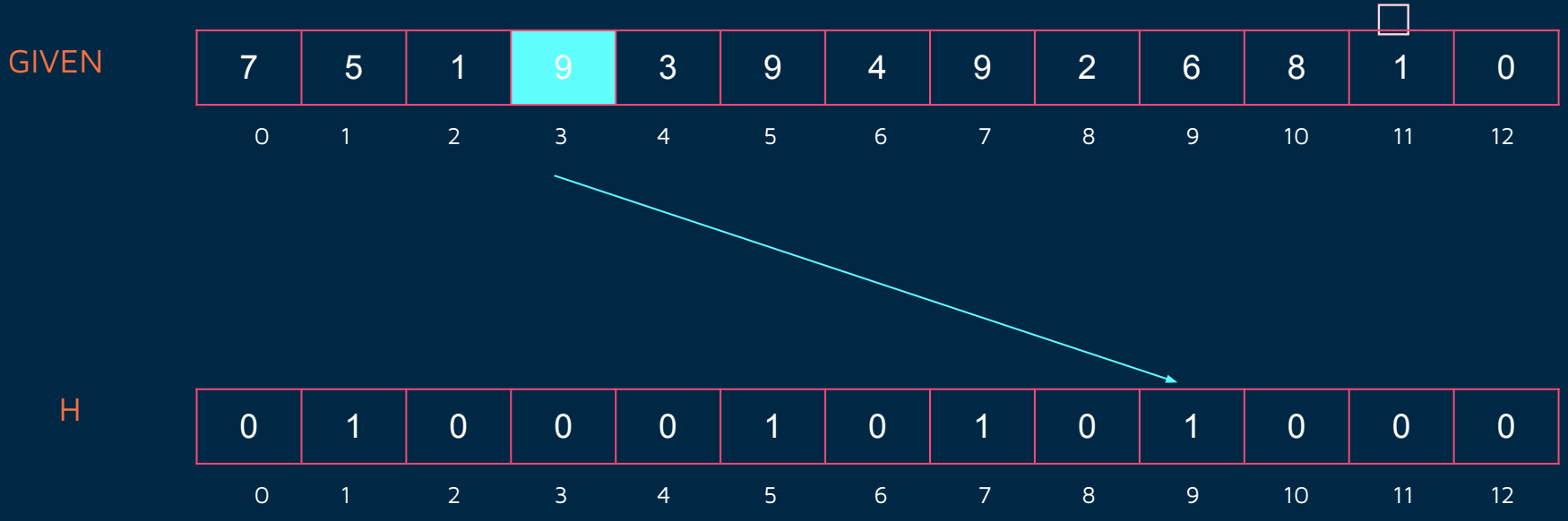
7	5	1	9	3	9	4	9	2	6	8	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12



H

0	1	0	0	0	1	0	1	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12

PROXMAP SORT



PROXMAP SORT

GIVEN

7	5	1	9	3	9	4	9	2	6	8	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12

H

0	1	0	0	0	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12

PROXMAP SORT

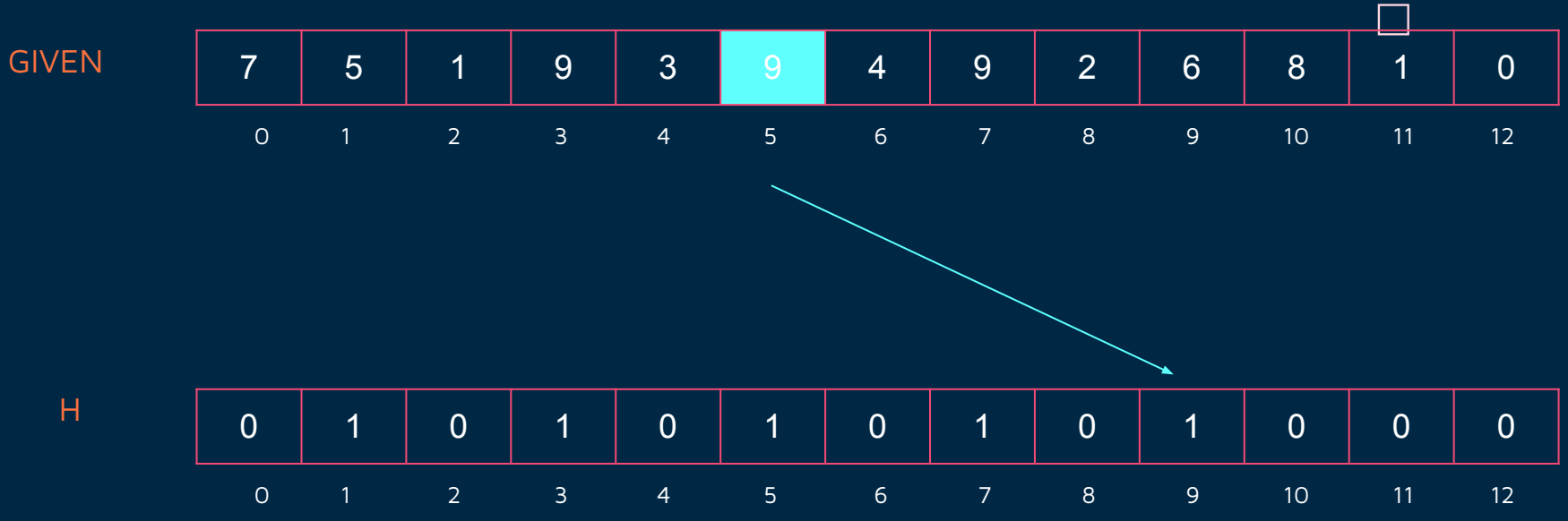
GIVEN

7	5	1	9	3	9	4	9	2	6	8	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12

H

0	1	0	1	0	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12

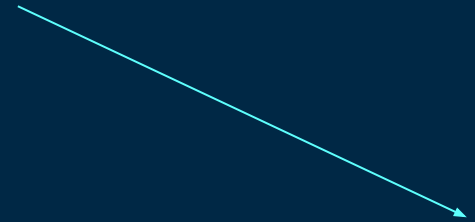
PROXMAP SORT



PROXMAP SORT

GIVEN

7	5	1	9	3	9	4	9	2	6	8	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12



H

0	1	0	1	0	1	0	1	0	2	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12

PROXMAP SORT

GIVEN

7	5	1	9	3	9	4	9	2	6	8	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12

Determine where each subarray will begin in the destination array so that each bucket is exactly the right size to hold all the keys that will map to it, using an array of "proxmaps," P

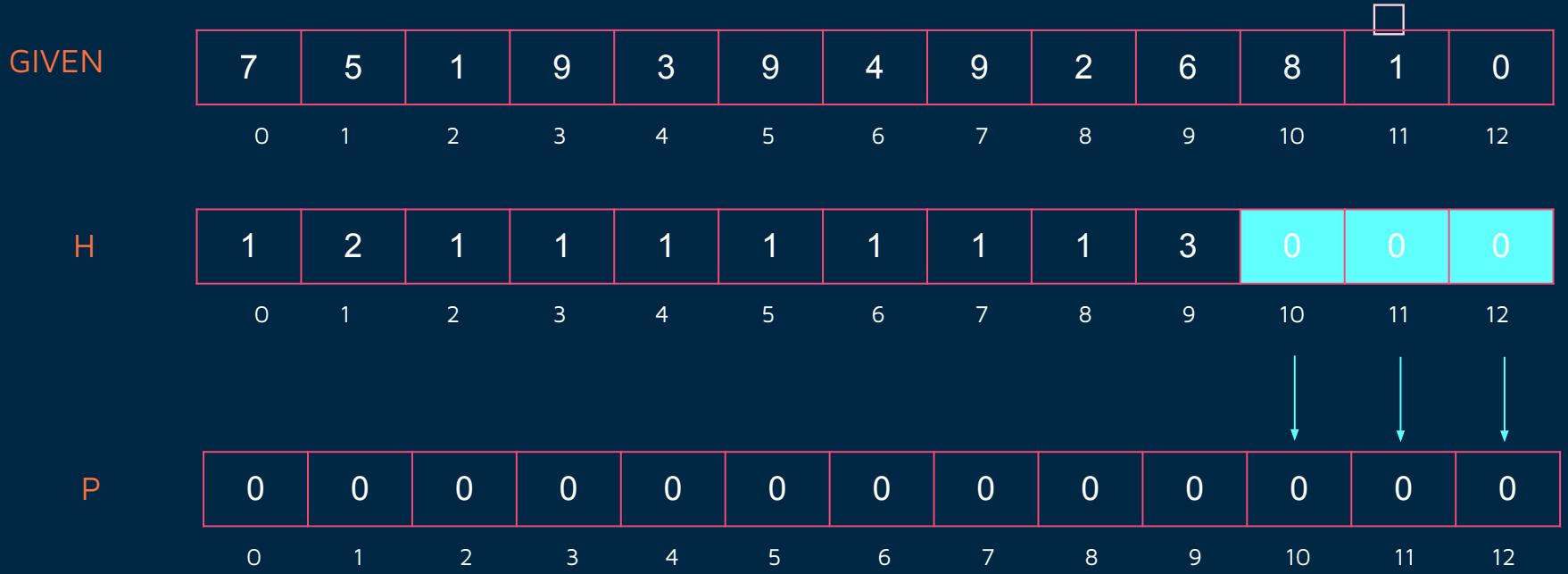
H

1	2	1	1	1	1	1	1	1	3	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12

P

0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12

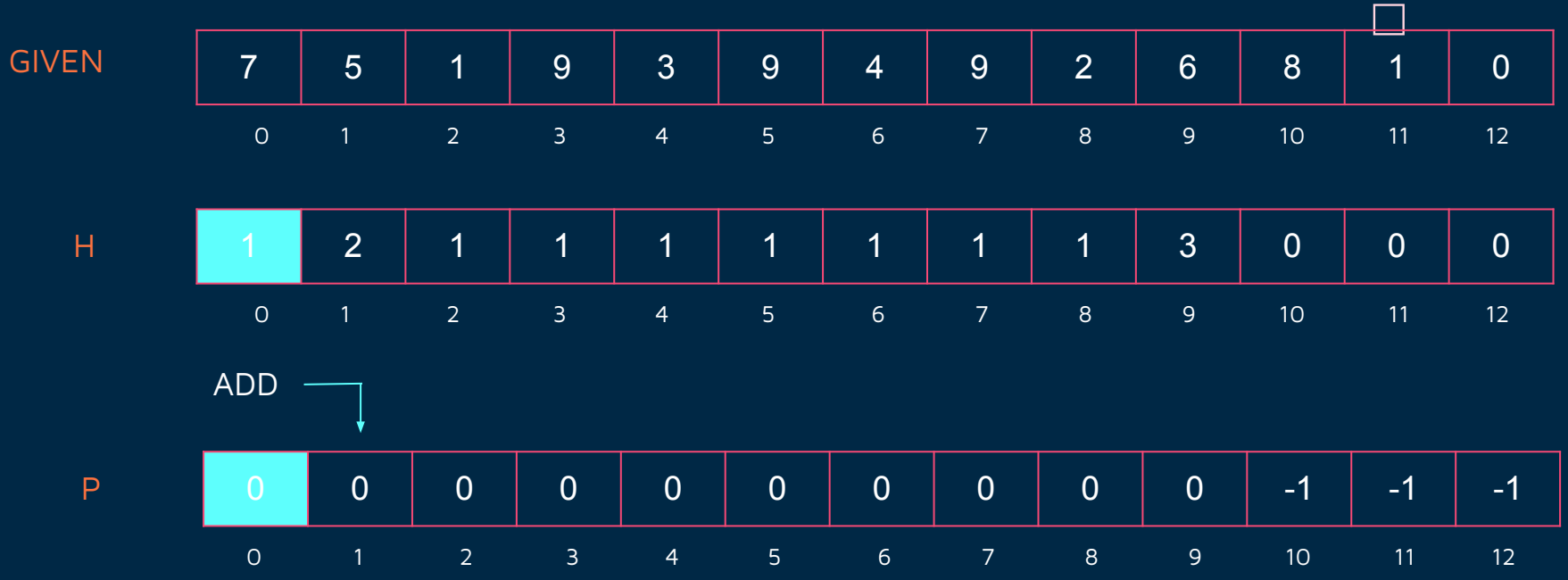
PROXMAP SORT



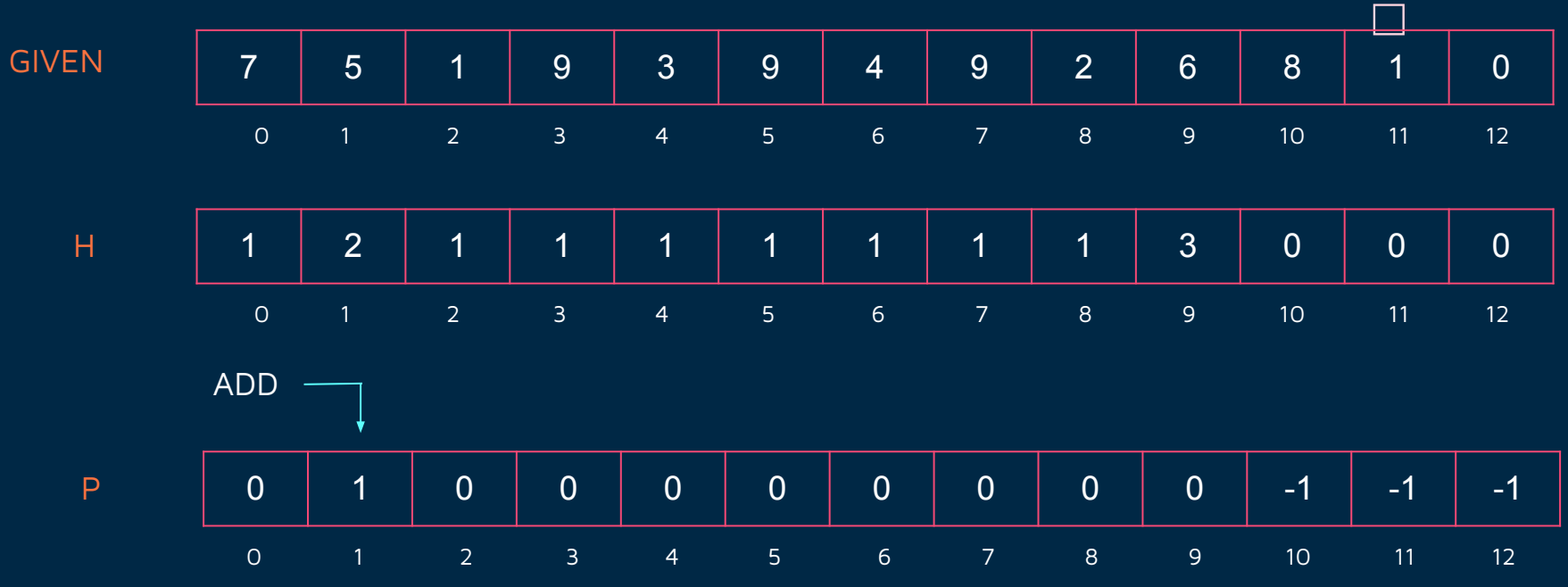
PROXMAP SORT

	0	1	2	3	4	5	6	7	8	9	10	11	12
GIVEN	7	5	1	9	3	9	4	9	2	6	8	1	0
H	1	2	1	1	1	1	1	1	1	3	0	0	0
P	0	0	0	0	0	0	0	0	0	0	-1	-1	-1

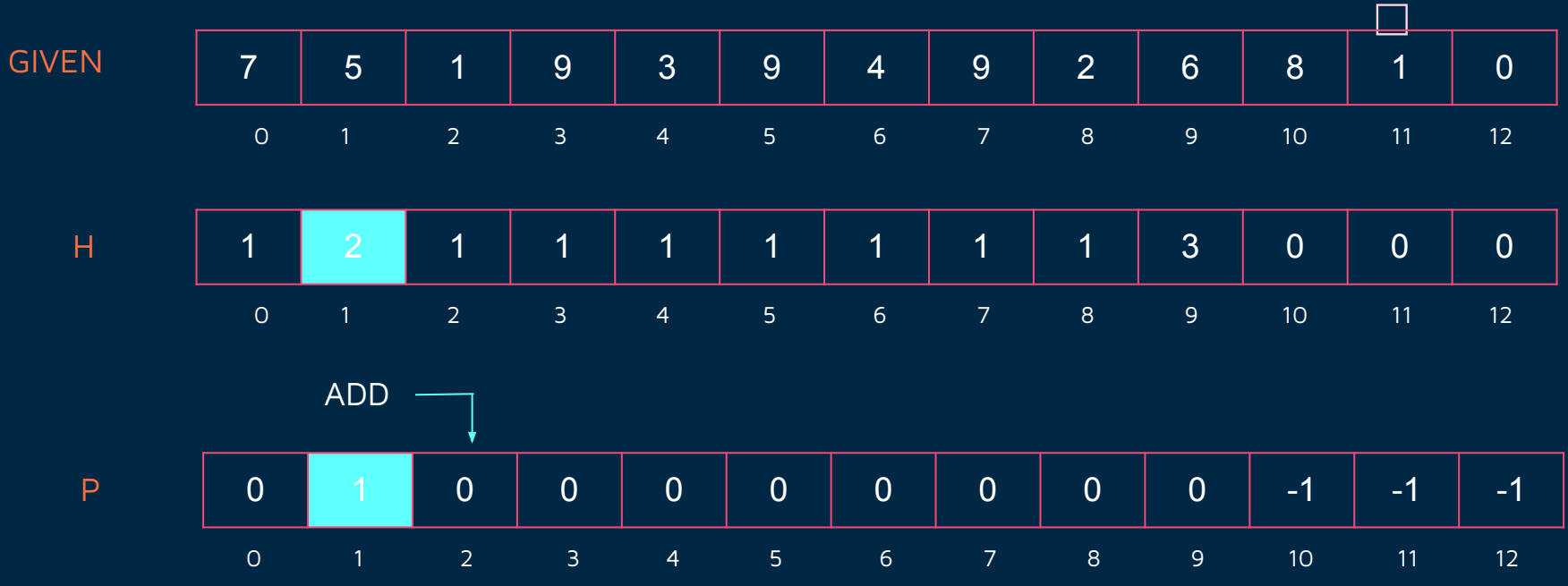
PROXMAP SORT



PROXMAP SORT



PROXMAP SORT



PROXMAP SORT

GIVEN

7	5	1	9	3	9	4	9	2	6	8	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12

H

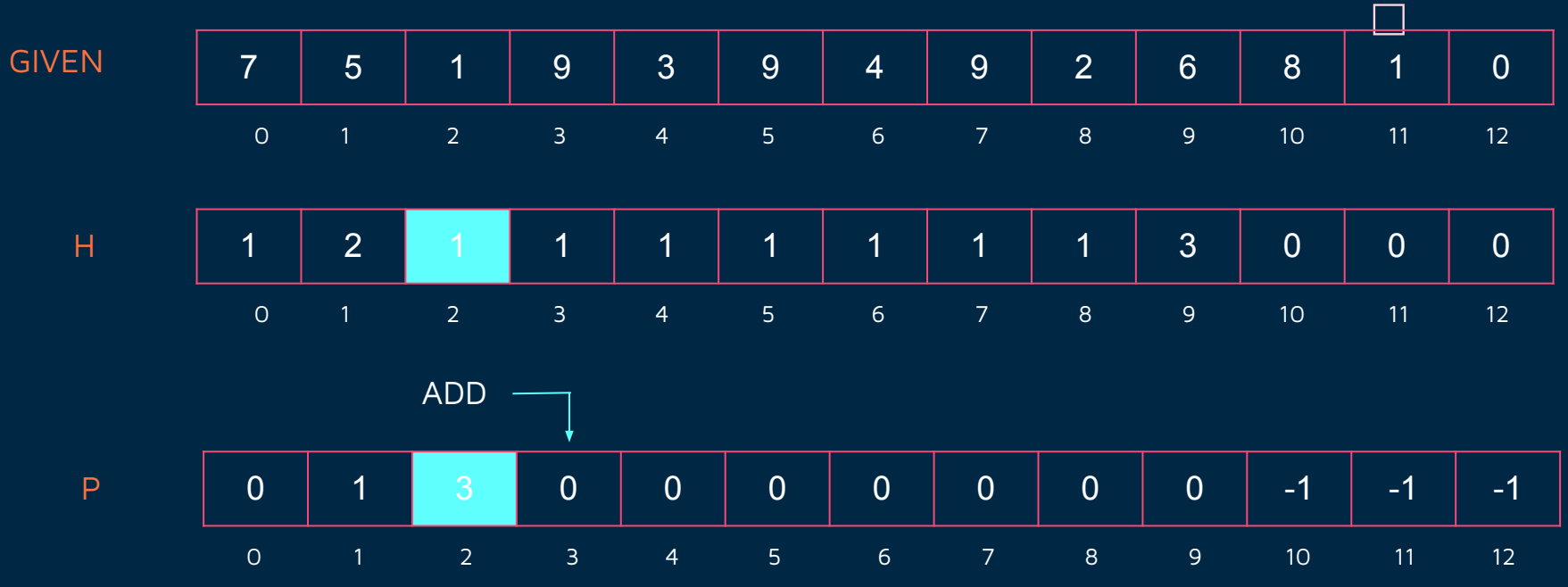
1	2	1	1	1	1	1	1	1	3	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12

ADD →

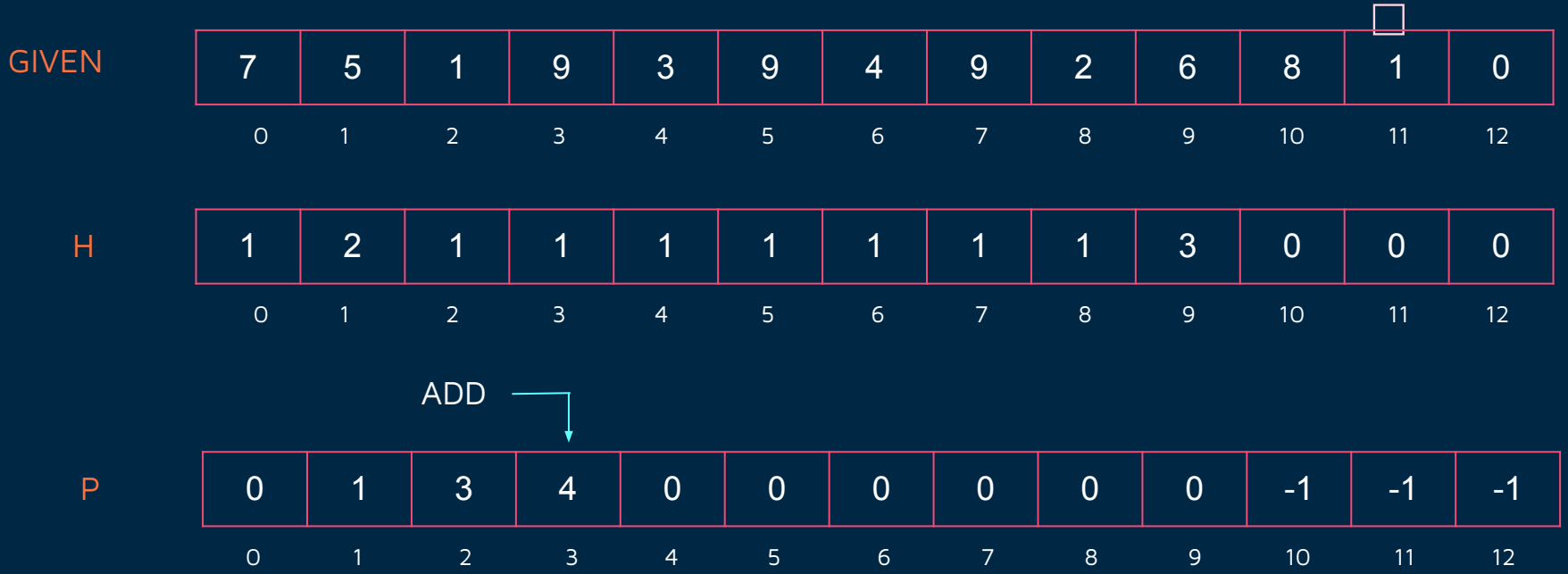
P

0	1	3	0	0	0	0	0	0	0	-1	-1	-1
0	1	2	3	4	5	6	7	8	9	10	11	12

PROXMAP SORT



PROXMAP SORT



PROXMAP SORT

GIVEN	7	5	1	9	3	9	4	9	2	6	8	1	0
	0	1	2	3	4	5	6	7	8	9	10	11	12
H	1	2	1	1	1	1	1	1	1	3	0	0	0
	0	1	2	3	4	5	6	7	8	9	10	11	12
P	0	1	3	4	5	6	7	8	9	10	-1	-1	-1
	0	1	2	3	4	5	6	7	8	9	10	11	12

PROXMAP SORT

GIVEN	7	5	1	9	3	9	4	9	2	6	8	1	0
	0	1	2	3	4	5	6	7	8	9	10	11	12

H	1	2	1	1	1	1	1	1	1	3	0	0	0
	0	1	2	3	4	5	6	7	8	9	10	11	12

For each key, compute the subarray it will map to, using an array of "locations," L

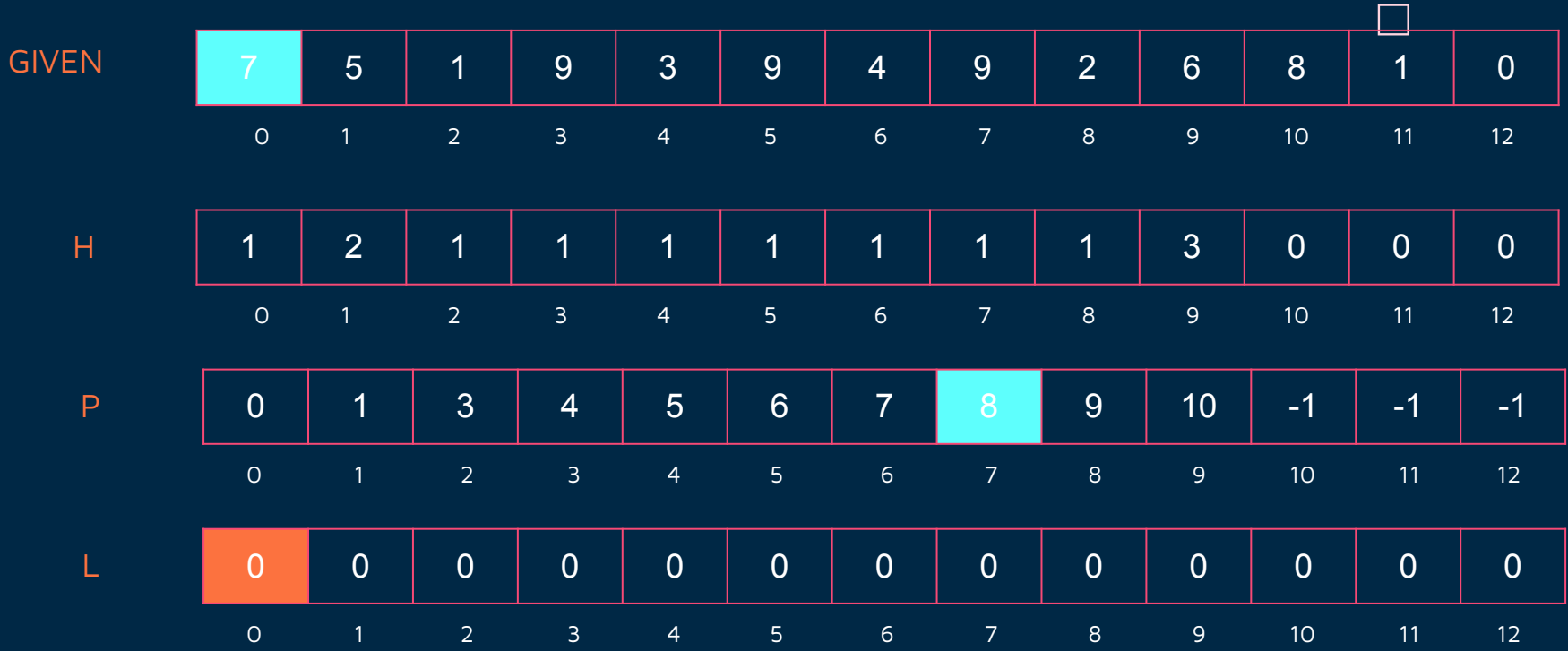
P	0	1	3	4	5	6	7	8	9	10	-1	-1	-1
	0	1	2	3	4	5	6	7	8	9	10	11	12

L	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6	7	8	9	10	11	12

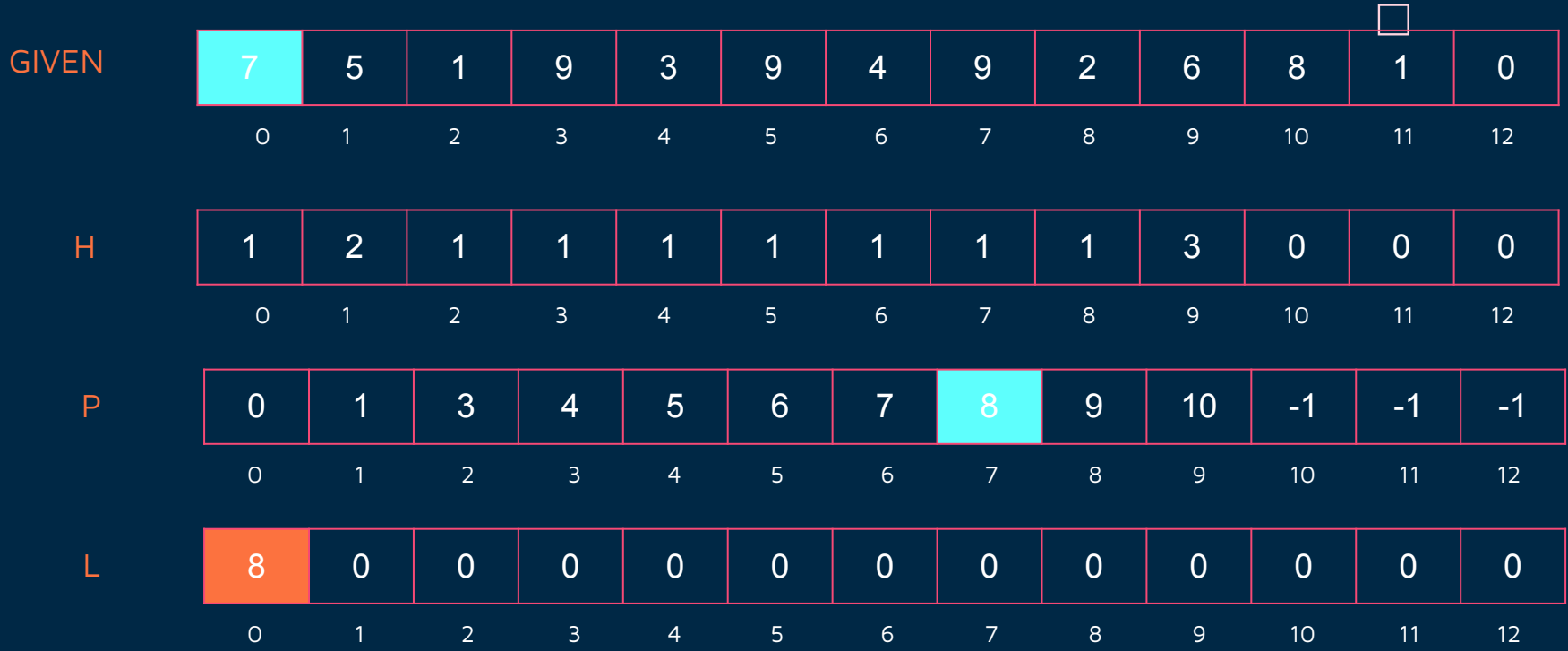
PROXMAP SORT

	0	1	2	3	4	5	6	7	8	9	10	11	12
GIVEN	7	5	1	9	3	9	4	9	2	6	8	1	0
H	1	2	1	1	1	1	1	1	1	3	0	0	0
P	0	1	3	4	5	6	7	8	9	10	-1	-1	-1
L	0	0	0	0	0	0	0	0	0	0	0	0	0

PROXMAP SORT



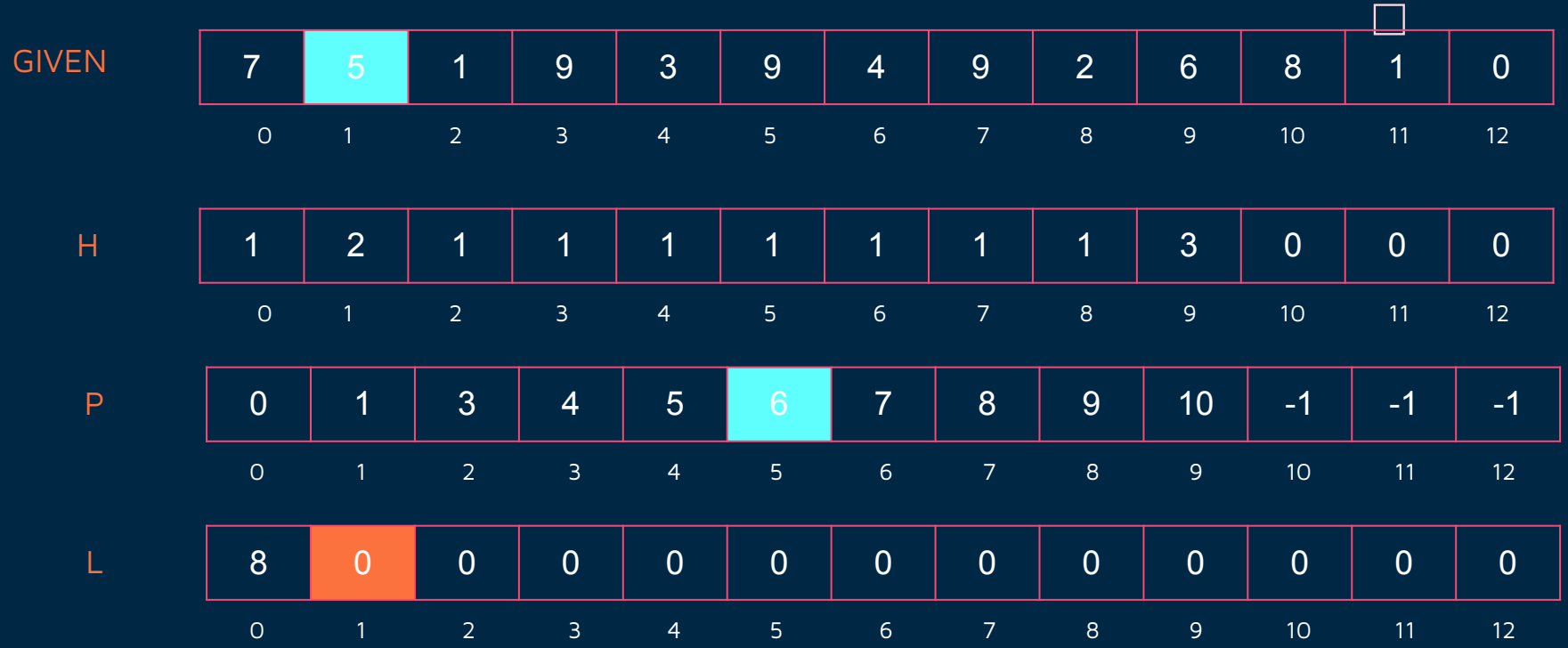
PROXMAP SORT



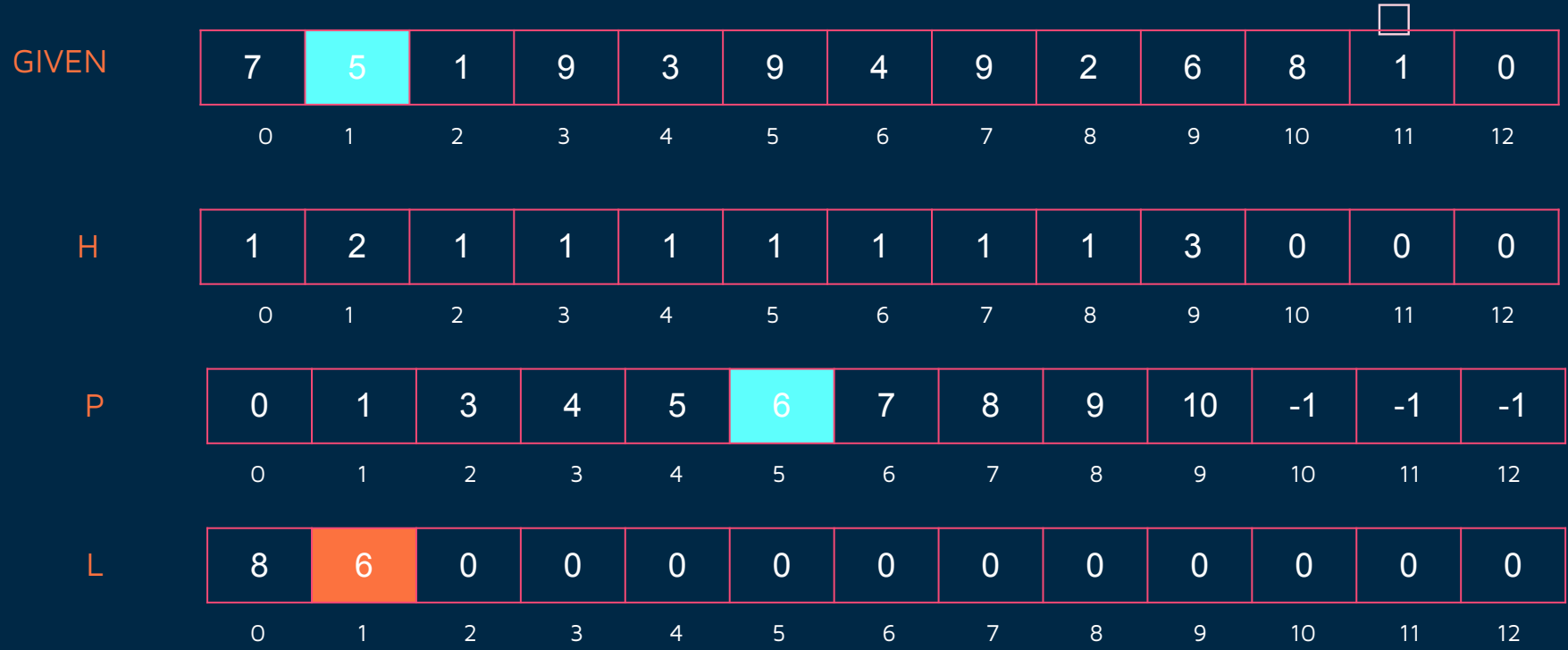
PROXMAP SORT

	0	1	2	3	4	5	6	7	8	9	10	11	12
GIVEN	7	5	1	9	3	9	4	9	2	6	8	1	0
H	1	2	1	1	1	1	1	1	1	3	0	0	0
P	0	1	3	4	5	6	7	8	9	10	-1	-1	-1
L	8	0	0	0	0	0	0	0	0	0	0	0	0

PROXMAP SORT



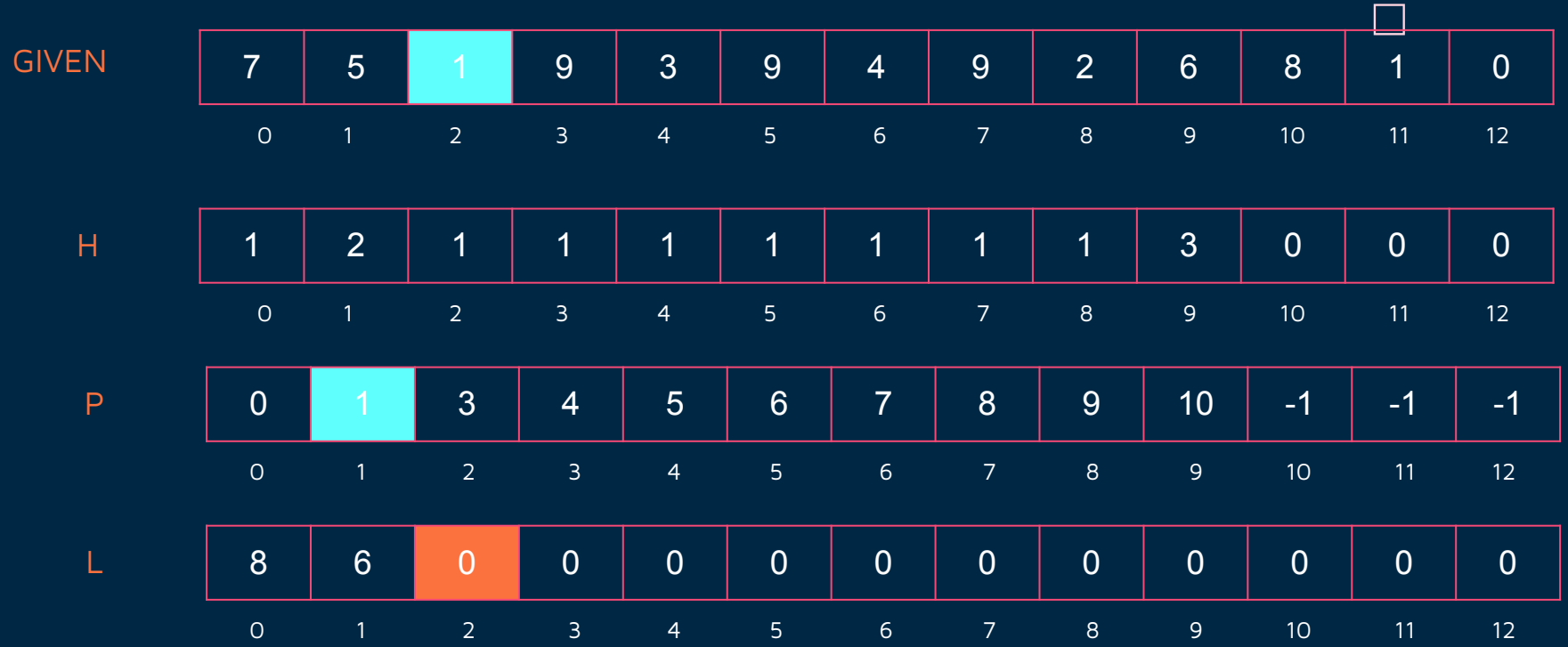
PROXMAP SORT



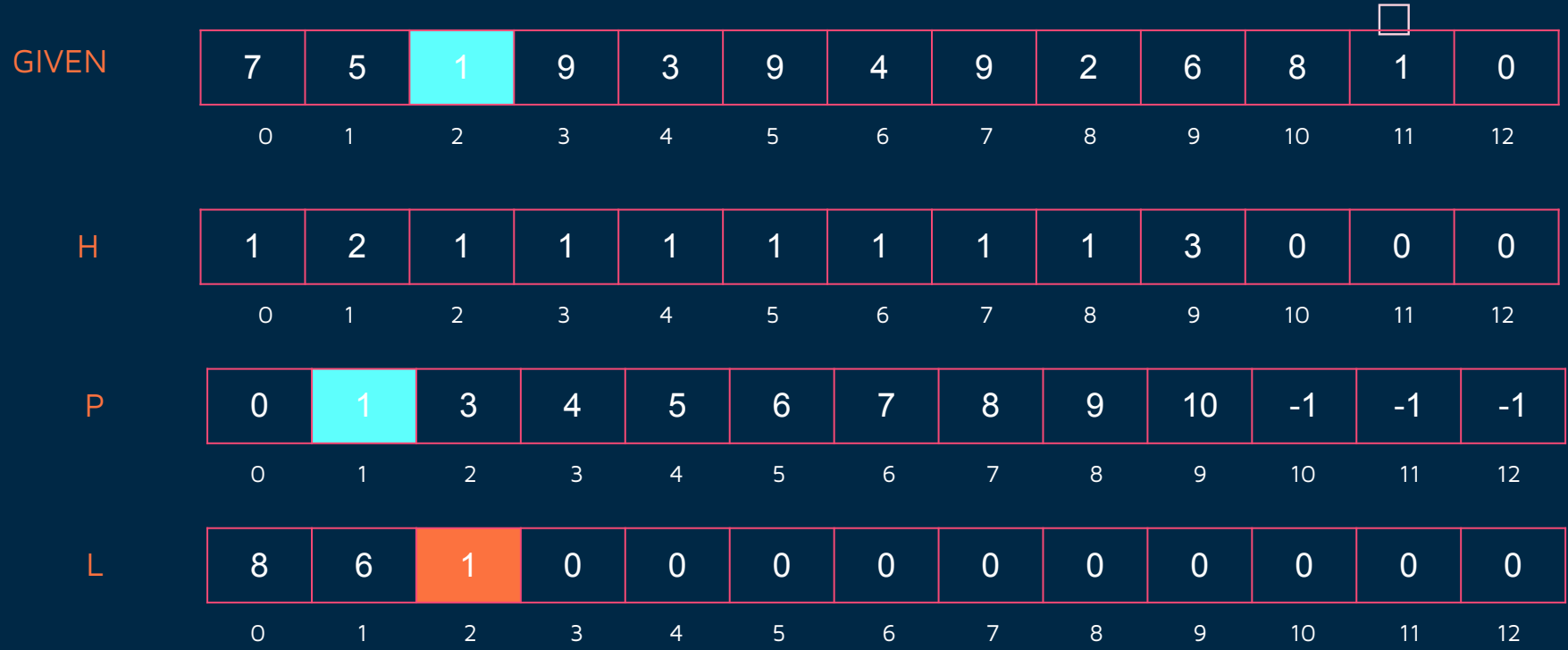
PROXMAP SORT

	0	1	2	3	4	5	6	7	8	9	10	11	12
GIVEN	7	5	1	9	3	9	4	9	2	6	8	1	0
H	1	2	1	1	1	1	1	1	1	3	0	0	0
P	0	1	3	4	5	6	7	8	9	10	-1	-1	-1
L	8	6	0	0	0	0	0	0	0	0	0	0	0

PROXMAP SORT



PROXMAP SORT



PROXMAP SORT

	0	1	2	3	4	5	6	7	8	9	10	11	12
GIVEN	7	5	1	9	3	9	4	9	2	6	8	1	0
H	1	2	1	1	1	1	1	1	1	3	0	0	0
P	0	1	3	4	5	6	7	8	9	10	-1	-1	-1
L	8	6	1	10	4	10	5	10	3	7	9	1	0

GIVEN

7	5	1	9	3	9	4	9	2	6	8	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12

H

1	2	1	1	1	1	1	1	1	3	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12

For each key, look up its location, place it into that cell of A2; if it collides with a key already in that position, insertion sort the key into place, moving keys greater than this key to the right by one to make a space for this key

P

0	1	3	4	5	6	7	8	9	10	-1	-1	-1
0	1	2	3	4	5	6	7	8	9	10	11	12

L

8	6	1	10	4	10	5	10	3	7	9	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12

GIVEN

7	5	1	9	3	9	4	9	2	6	8	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12

For each key, look up its location, place it into that cell of A2; if it collides with a key already in that position, insertion sort the key into place, moving keys greater than this key to the right by one to make a space for this key

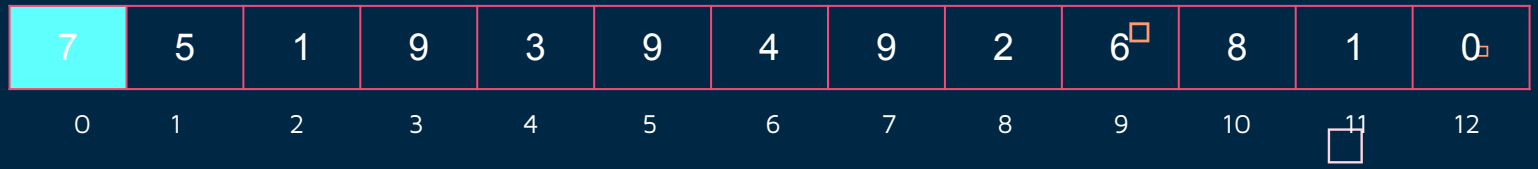
L

8	6	1	10	4	10	5	10	3	7	9	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12

A2

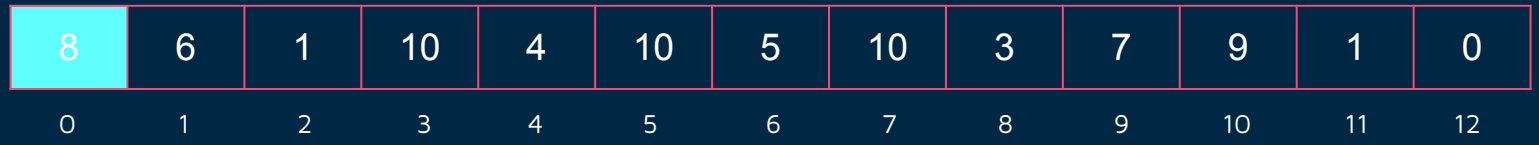
0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12

GIVEN

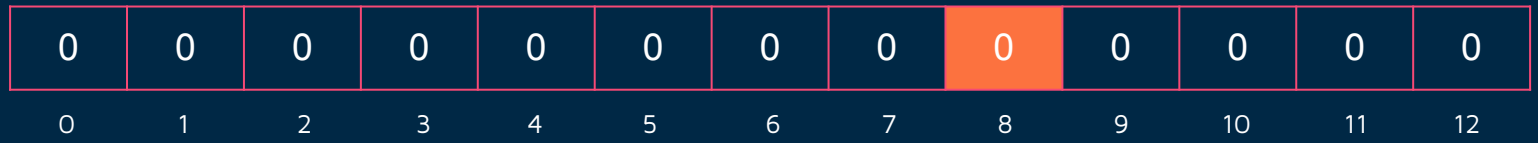


For each key, look up its location, place it into that cell of A2; if it collides with a key already in that position, insertion sort the key into place, moving keys greater than this key to the right by one to make a space for this key

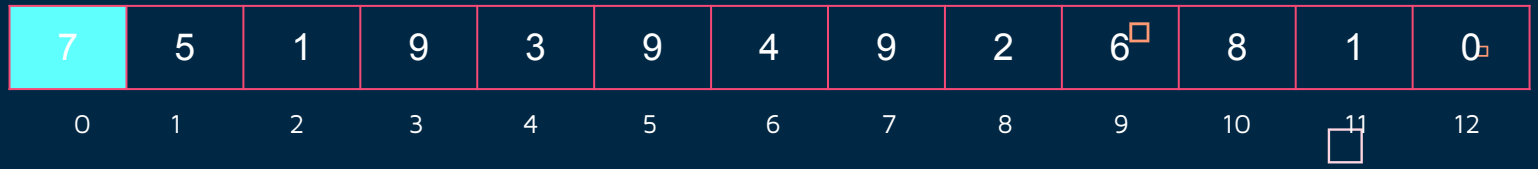
L



A2

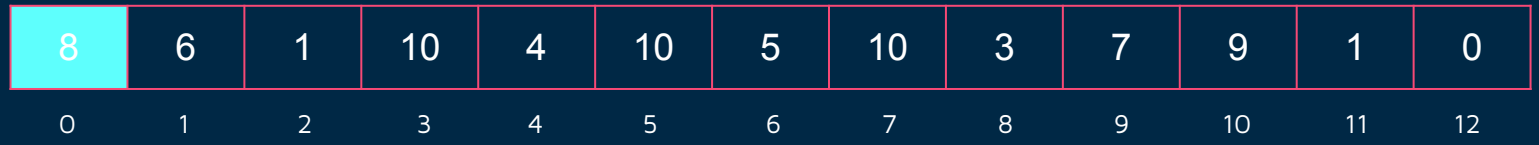


GIVEN

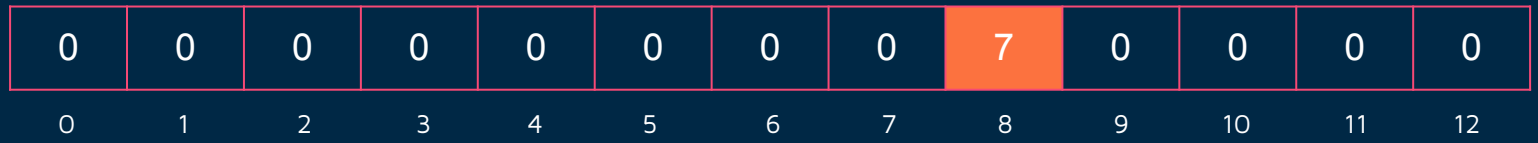


For each key, look up its location, place it into that cell of A2; if it collides with a key already in that position, insertion sort the key into place, moving keys greater than this key to the right by one to make a space for this key

L



A2



GIVEN

7	5	1	9	3	9	4	9	2	6	8	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12

For each key, look up its location, place it into that cell of A2; if it collides with a key already in that position, insertion sort the key into place, moving keys greater than this key to the right by one to make a space for this key

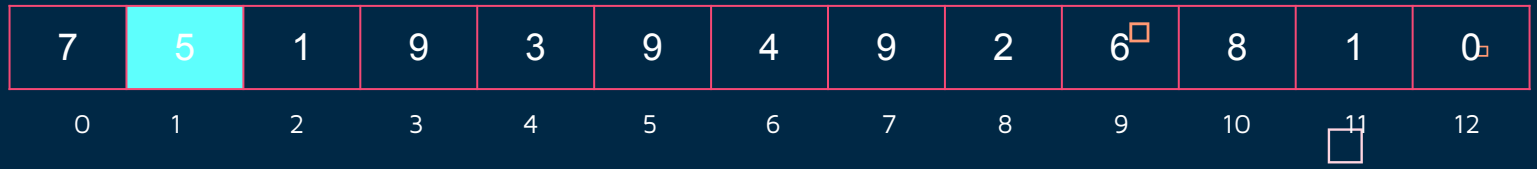
L

8	6	1	10	4	10	5	10	3	7	9	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12

A2

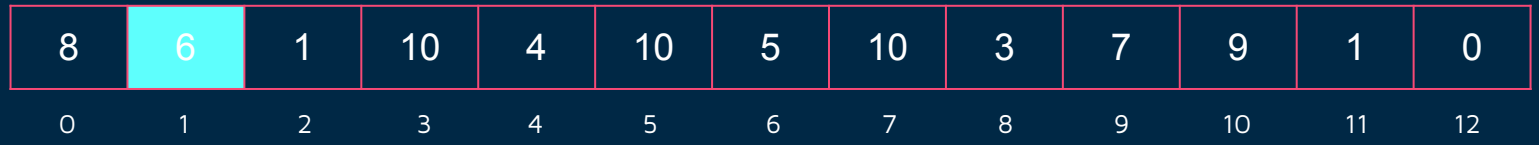
0	0	0	0	0	0	0	0	7	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12

GIVEN

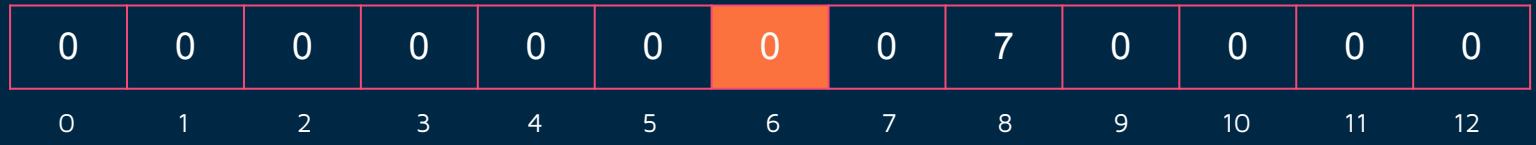


For each key, look up its location, place it into that cell of A2; if it collides with a key already in that position, insertion sort the key into place, moving keys greater than this key to the right by one to make a space for this key

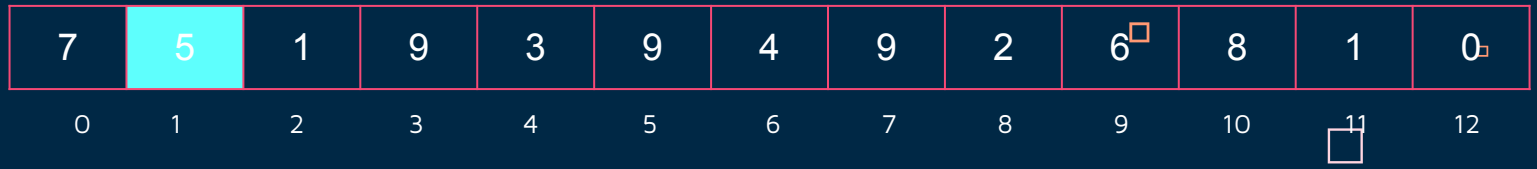
L



A2

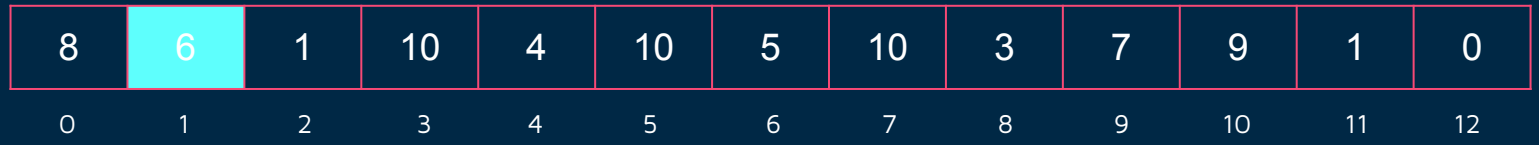


GIVEN

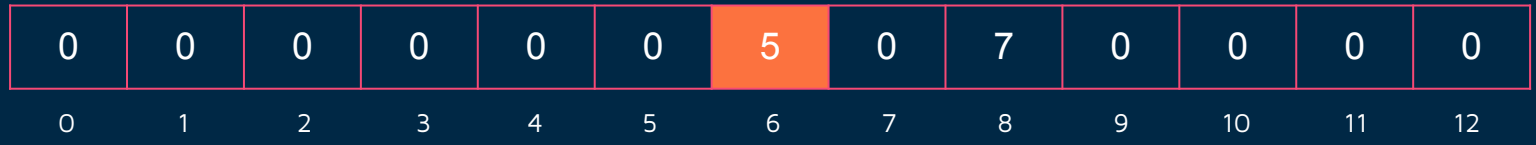


For each key, look up its location, place it into that cell of A2; if it collides with a key already in that position, insertion sort the key into place, moving keys greater than this key to the right by one to make a space for this key

L



A2



GIVEN

7	5	1	9	3	9	4	9	2	6	8	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12

For each key, look up its location, place it into that cell of A2; if it collides with a key already in that position, insertion sort the key into place, moving keys greater than this key to the right by one to make a space for this key

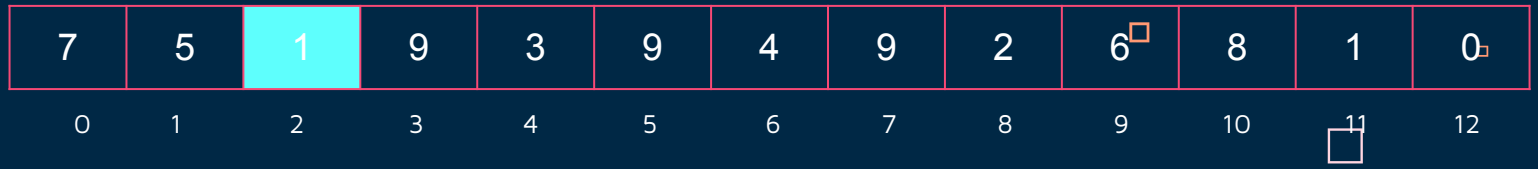
L

8	6	1	10	4	10	5	10	3	7	9	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12

A2

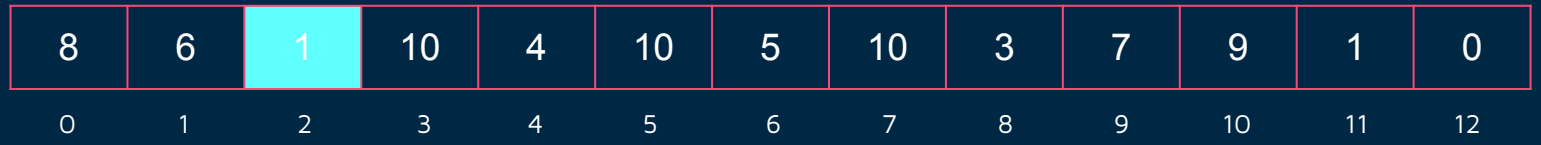
0	0	0	0	0	0	5	0	7	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12

GIVEN

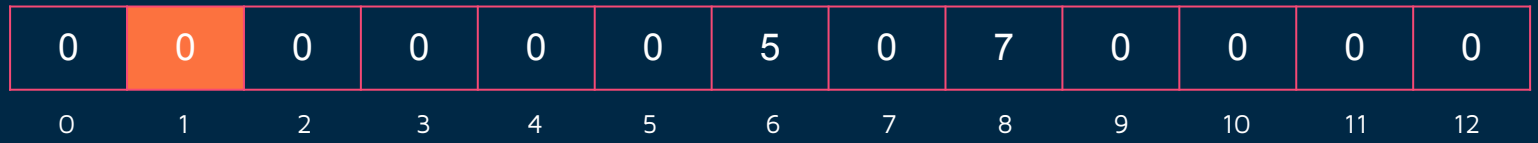


For each key, look up its location, place it into that cell of A2; if it collides with a key already in that position, insertion sort the key into place, moving keys greater than this key to the right by one to make a space for this key

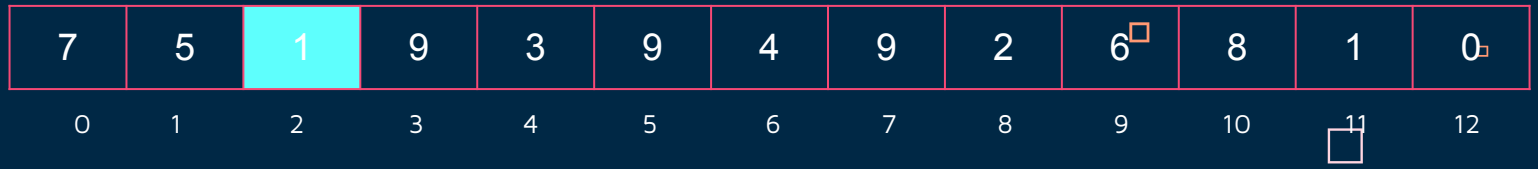
L



A2

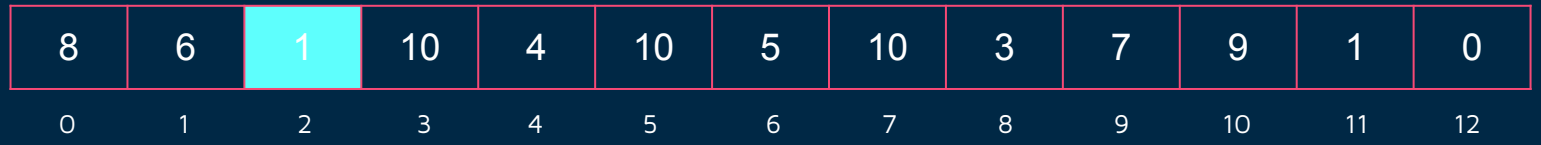


GIVEN

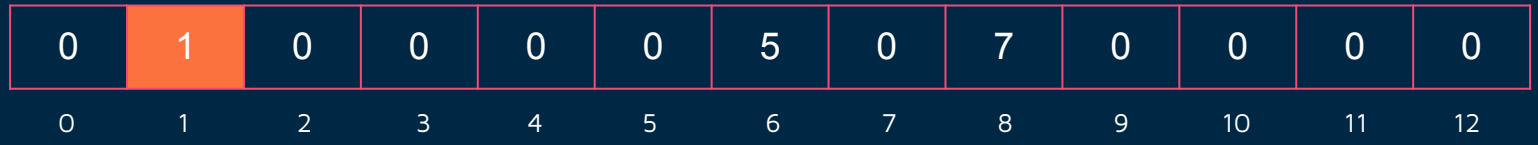


For each key, look up its location, place it into that cell of A2; if it collides with a key already in that position, insertion sort the key into place, moving keys greater than this key to the right by one to make a space for this key

L



A2



GIVEN

7	5	1	9	3	9	4	9	2	6	8	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12

For each key, look up its location, place it into that cell of A2; if it collides with a key already in that position, insertion sort the key into place, moving keys greater than this key to the right by one to make a space for this key

L

8	6	1	10	4	10	5	10	3	7	9	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12

A2

0	1	0	0	0	0	5	0	7	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12

GIVEN

7	5	1	9	3	9	4	9	2	6	8	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12

For each key, look up its location, place it into that cell of A2; if it collides with a key already in that position, insertion sort the key into place, moving keys greater than this key to the right by one to make a space for this key

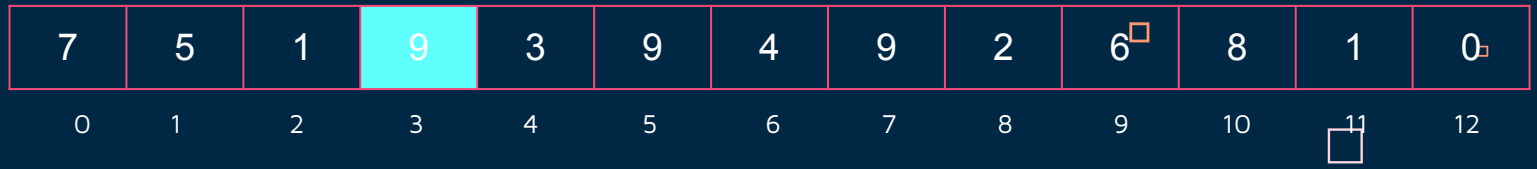
L

8	6	1	10	4	10	5	10	3	7	9	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12

A2

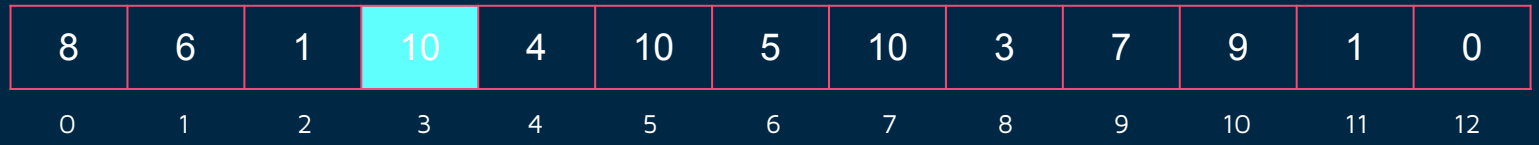
0	1	0	0	0	0	5	0	7	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12

GIVEN

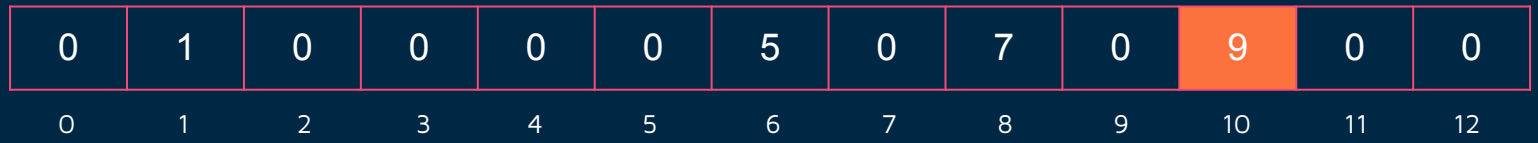


For each key, look up its location, place it into that cell of A2; if it collides with a key already in that position, insertion sort the key into place, moving keys greater than this key to the right by one to make a space for this key

L



A2



GIVEN

7	5	1	9	3	9	4	9	2	6	8	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12

For each key, look up its location, place it into that cell of A2; if it collides with a key already in that position, insertion sort the key into place, moving keys greater than this key to the right by one to make a space for this key

L

8	6	1	10	4	10	5	10	3	7	9	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12

A2

0	1	0	0	0	0	5	0	7	0	9	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12

GIVEN

7	5	1	9	3	9	4	9	2	6	8	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12

For each key, look up its location, place it into that cell of A2; if it collides with a key already in that position, insertion sort the key into place, moving keys greater than this key to the right by one to make a space for this key

L

8	6	1	10	4	10	5	10	3	7	9	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12

A2

0	1	0	0	0	0	5	0	7	0	9	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12

GIVEN

7	5	1	9	3	9	4	9	2	6	8	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12

For each key, look up its location, place it into that cell of A2; if it collides with a key already in that position, insertion sort the key into place, moving keys greater than this key to the right by one to make a space for this key

L

8	6	1	10	4	10	5	10	3	7	9	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12

A2

0	1	0	0	3	0	5	0	7	0	9	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12

GIVEN

7	5	1	9	3	9	4	9	2	6	8	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12

For each key, look up its location, place it into that cell of A2; if it collides with a key already in that position, insertion sort the key into place, moving keys greater than this key to the right by one to make a space for this key

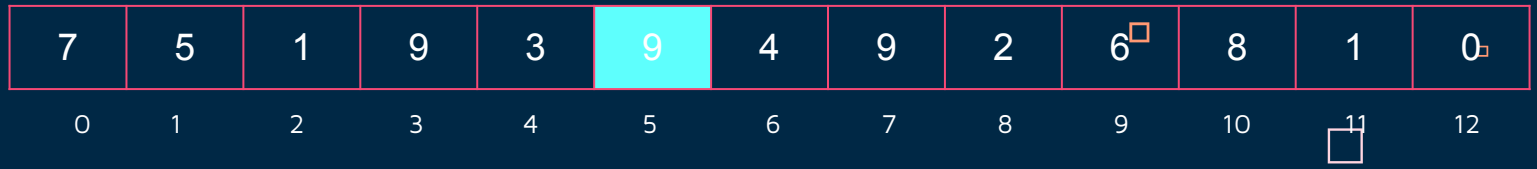
L

8	6	1	10	4	10	5	10	3	7	9	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12

A2

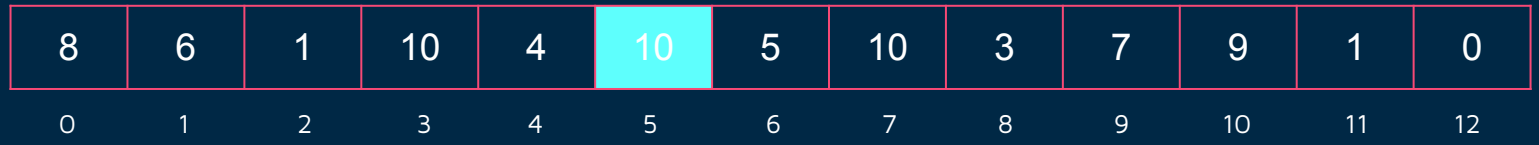
0	1	0	0	3	0	5	0	7	0	9	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12

GIVEN

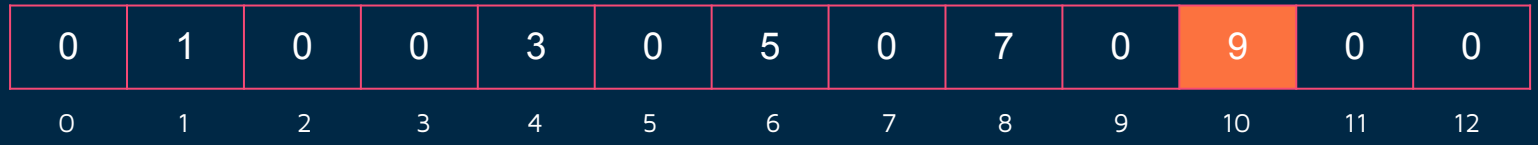


For each key, look up its location, place it into that cell of A2; if it collides with a key already in that position, insertion sort the key into place, moving keys greater than this key to the right by one to make a space for this key

L



A2



GIVEN

7	5	1	9	3	9	4	9	2	6	8	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12

For each key, look up its location, place it into that cell of A2; if it collides with a key already in that position, insertion sort the key into place, moving keys greater than this key to the right by one to make a space for this key

L

8	6	1	10	4	10	5	10	3	7	9	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12

A2

0	1	0	0	3	0	5	0	7	0	9	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12

GIVEN

7	5	1	9	3	9	4	9	2	6	8	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12

For each key, look up its location, place it into that cell of A2; if it collides with a key already in that position, insertion sort the key into place, moving keys greater than this key to the right by one to make a space for this key

L

8	6	1	10	4	10	5	10	3	7	9	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12

A2

0	1	0	0	3	0	5	0	7	0	9	9	0
0	1	2	3	4	5	6	7	8	9	10	11	12

GIVEN

7	5	1	9	3	9	4	9	2	6	8	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12

For each key, look up its location, place it into that cell of A2; if it collides with a key already in that position, insertion sort the key into place, moving keys greater than this key to the right by one to make a space for this key

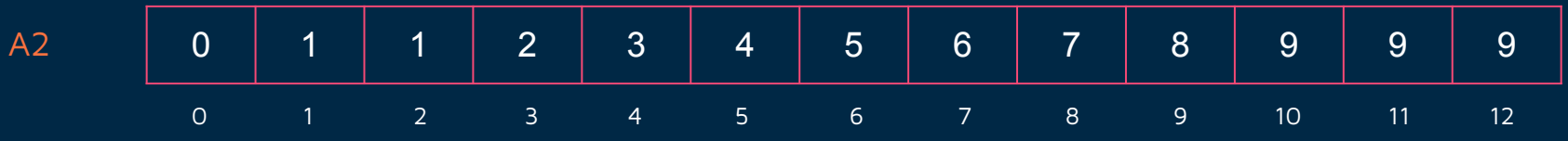
L

8	6	1	10	4	10	5	10	3	7	9	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12

A2

0	1	1	2	3	4	5	6	7	8	9	9	9
0	1	2	3	4	5	6	7	8	9	10	11	12

SORTED ARRAY



BUCKET SORT (SHUFFLE)

L

71	61	53	26	72	64	35	70	87	20	83	65	63	67	86	90
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

n = 16

K = n/8 = 16/8 = 2

A = n/8 = 16/8 = 2

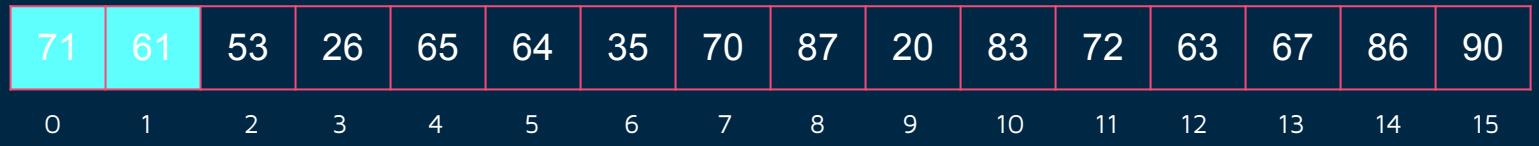
B = A+1 = 2+ 1 = 3

Let K be a list of the first n/8 elements of L (remove from L)

K

71	61

BUCKET SORT (SHUFFLE)



n = 16

K = n/8 = 16/8 = 2

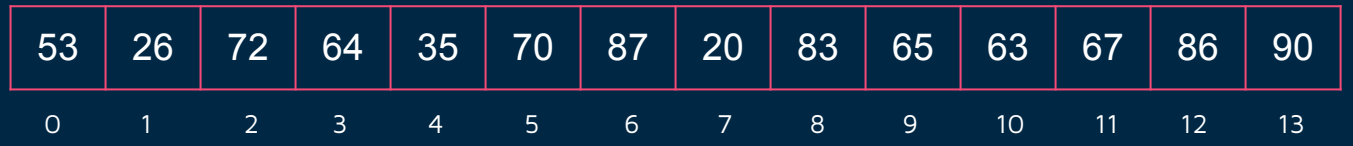
A = n/8 = 16/8 = 2

B = A+1 = 2+ 1 = 3

Let K be a list of the first n/8 elements of L (remove from L)

BUCKET SORT (SHUFFLE)

L



n = 16

K = n/8 = 16/8 = 2

A = n/8 = 16/8 = 2

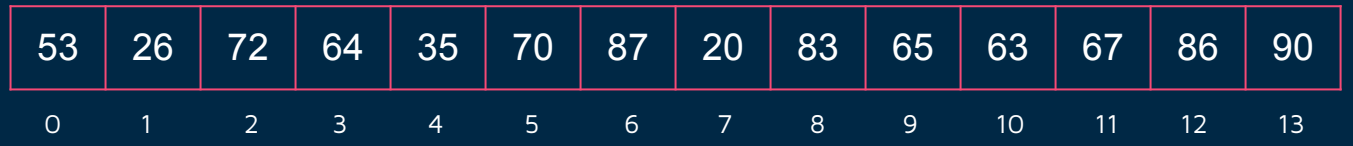
B = A+1 = 2+ 1 = 3

Let K be a list of the first n/8 elements of L (remove from L)

K

BUCKET SORT (SHUFFLE)

L



n = 16

K = n/8 = 16/8 = 2

A = n/8 = 16/8 = 2

B = A+1 = 2+ 1 = 3

Let K be a list of the first n/8 elements of L (remove from L)

Sort K recursively

K

BUCKET SORT (SHUFFLE)

L

53	26	72	64	35	70	87	20	83	65	63	67	86	90
0	1	2	3	4	5	6	7	8	9	10	11	12	13

n = 16

K = n/8 = 16/8 = 2

A = n/8 = 16/8 = 2

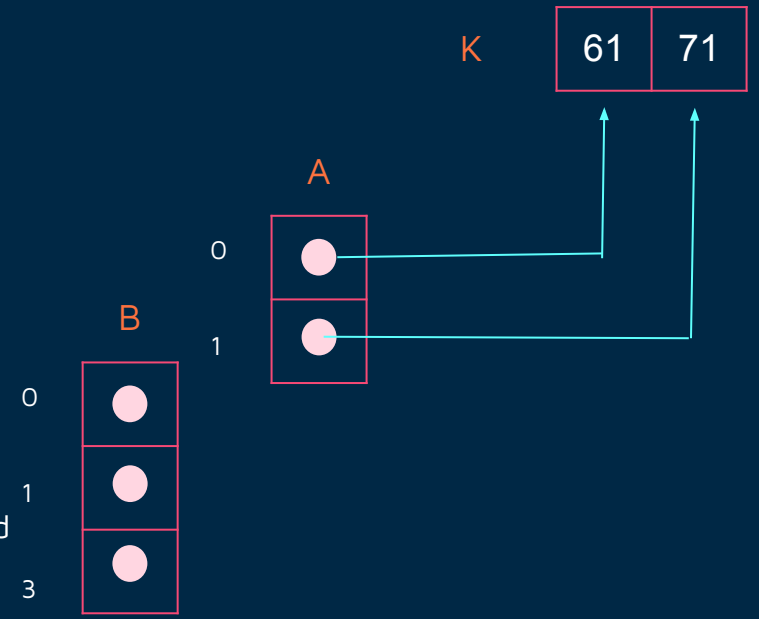
B = A+1 = 2+ 1 = 3

Let K be a list of the first n/8 elements of L (remove from L)

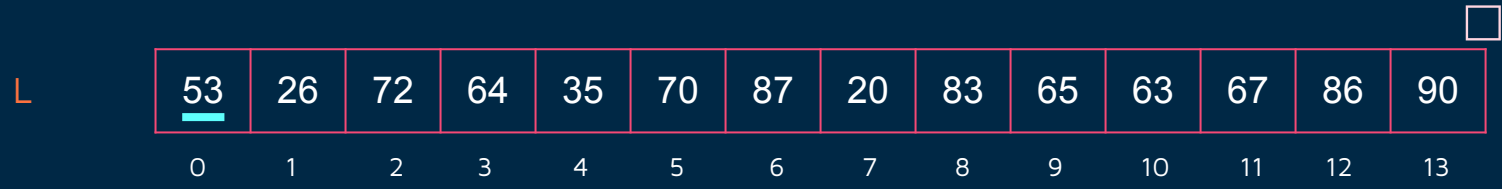
Sort K recursively

Let A be an array of n/8 pointers to elements, and set them to the elements of K

Let B be an array of n/8+1 empty lists. These correspond to the lists which are inbetween, proceed, and proceed the elements in K

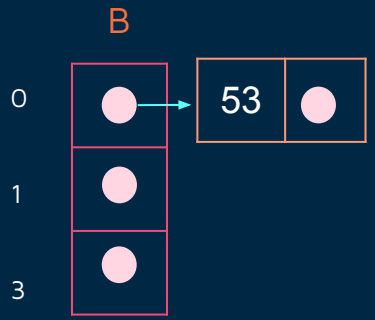


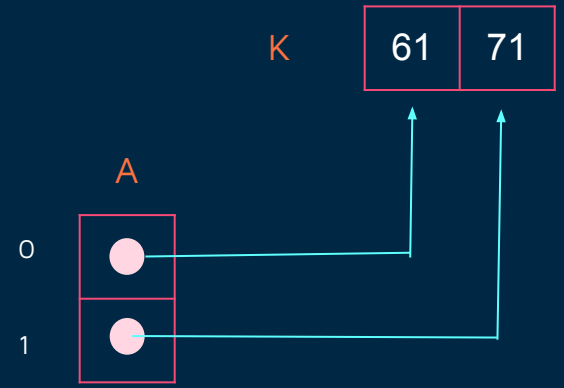
BUCKET SORT (SHUFFLE)



For the remaining elements in L, append each to the appropriate list in B as determined by a binary search in A.

Recursively sort elements listed in B





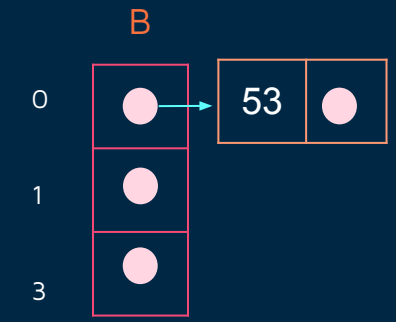
BUCKET SORT (SHUFFLE)

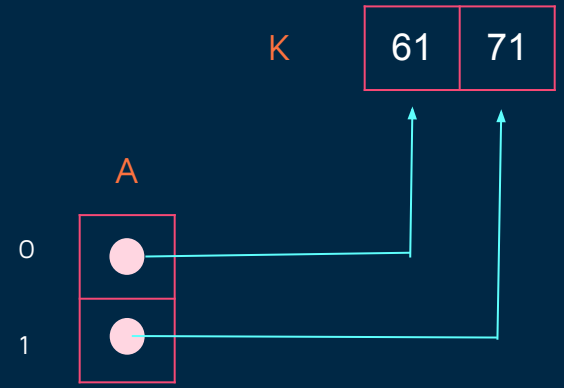
L

53	26	72	64	35	70	87	20	83	65	63	67	86	90
0	1	2	3	4	5	6	7	8	9	10	11	12	13

For the remaining elements in L, append each to the appropriate list in B as determined by a binary search in A.

Recursively sort elements listed in B





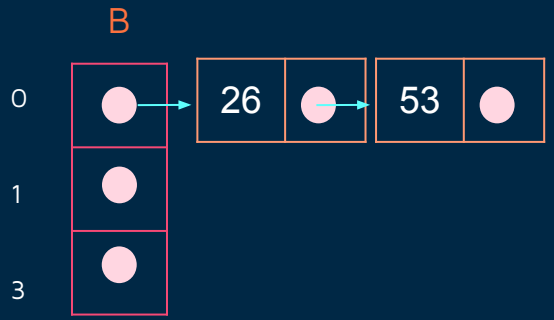
BUCKET SORT (SHUFFLE)

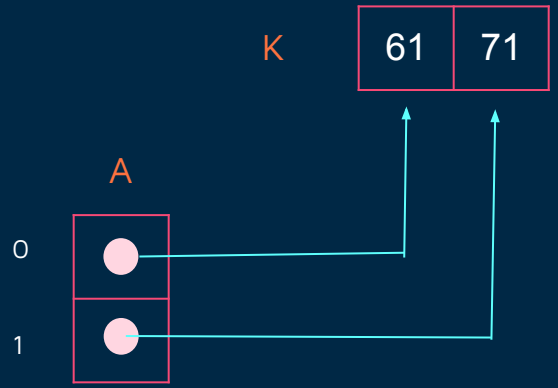
L

53	26	72	64	35	70	87	20	83	65	63	67	86	90
0	1	2	3	4	5	6	7	8	9	10	11	12	13

For the remaining elements in L, append each to the appropriate list in B as determined by a binary search in A.

Recursively sort elements listed in B





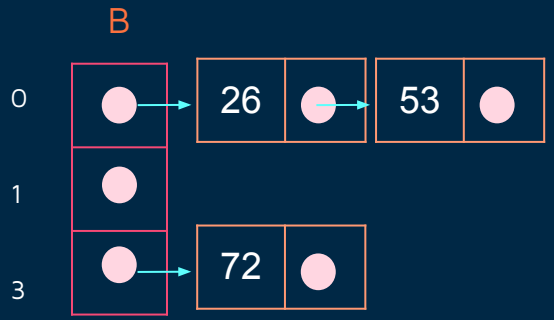
BUCKET SORT (SHUFFLE)

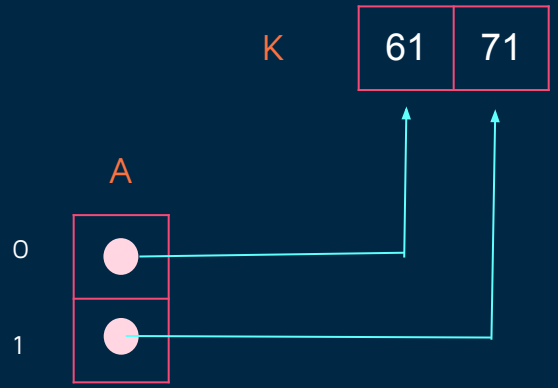
L

53	26	72	64	35	70	87	20	83	65	63	67	86	90
0	1	2	3	4	5	6	7	8	9	10	11	12	13

For the remaining elements in L, append each to the appropriate list in B as determined by a binary search in A.

Recursively sort elements listed in B





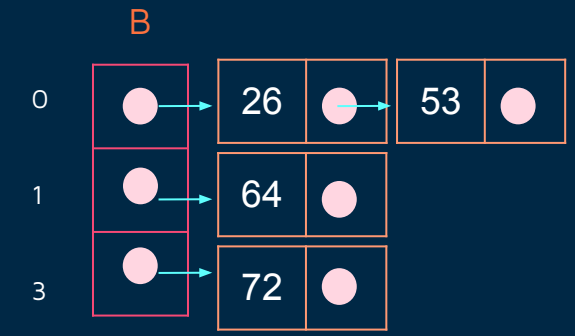
BUCKET SORT (SHUFFLE)

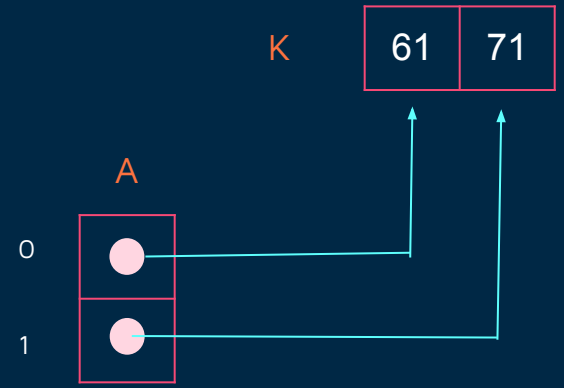
L

53	26	72	64	35	70	87	20	83	65	63	67	86	90
0	1	2	3	4	5	6	7	8	9	10	11	12	13

For the remaining elements in L, append each to the appropriate list in B as determined by a binary search in A.

Recursively sort elements listed in B





BUCKET SORT (SHUFFLE)

L

53	26	72	64	35	70	87	20	83	65	63	67	86	90
0	1	2	3	4	5	6	7	8	9	10	11	12	13

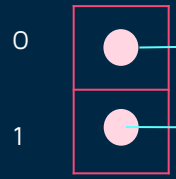
For the remaining elements in L, append each to the appropriate list in B as determined by a binary search in A.

Recursively sort elements listed in B

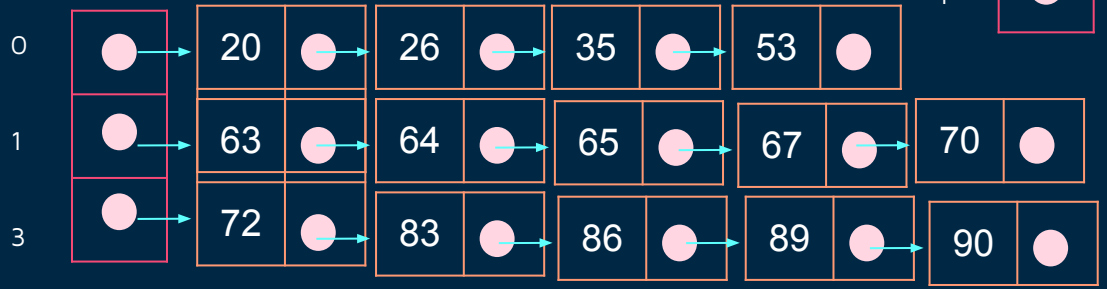
K

61	71

A



B



BUCKET SORT (SHUFFLE)

L

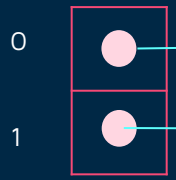
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
20	26	35	53	61	63	64	65	67	70	71	72	83	86	89	90

Construct the original list by appending
the elements of the first
list in B, then the first element of K,
then the second list in B,
then the second element of K, ..., then
the nth element of K,
then the n+1th element of B, and return

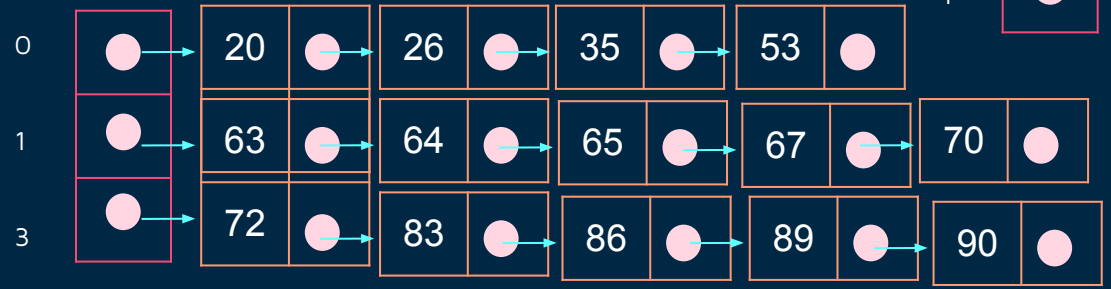
K

61	71

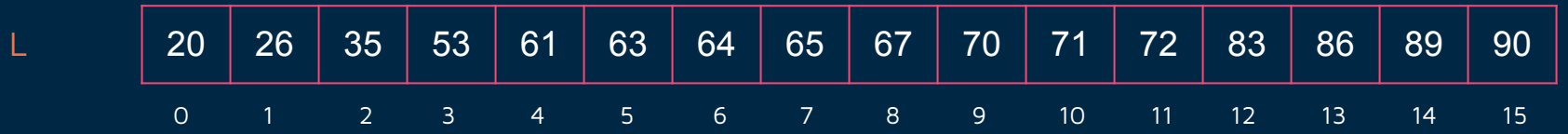
A



B

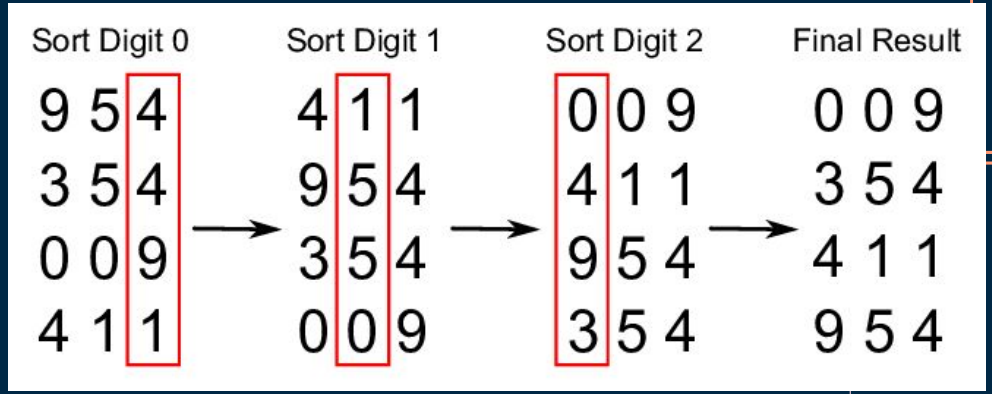


SORTED ARRAY



Radix Sort

- Radix sort can be dated back in 1887 by Herman Hollerith with the tabulating machines.
- Radix sorting algorithm can be traced to Harold H. Seward in 1954
- Considered as a **stable** sort
- An **out-of-place algorithm**



Radix Sort

Input Array

121	432	564	23	1	45	788
3	3	3	2	1	2	3

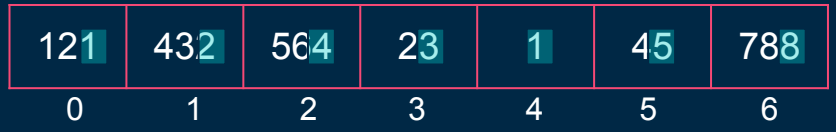
1. Find the maximum digit (d)

d

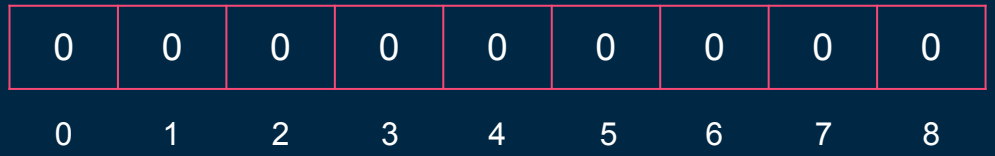
3

Radix Sort

Input Array

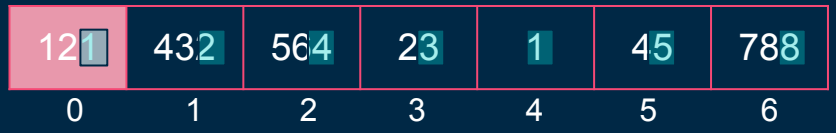


Count Array

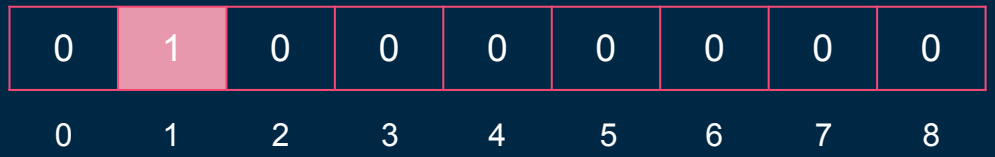


Radix Sort

Input Array

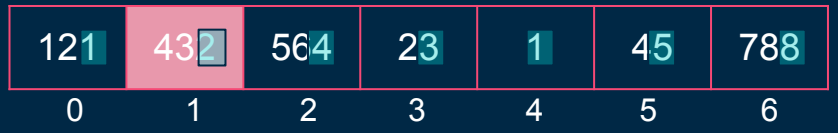


Count Array

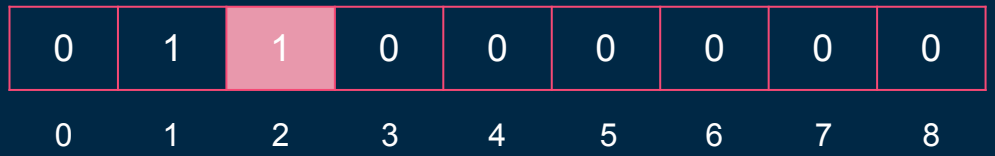


Radix Sort

Input Array

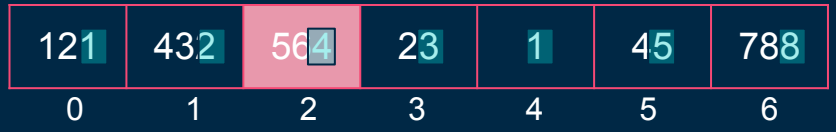


Count Array

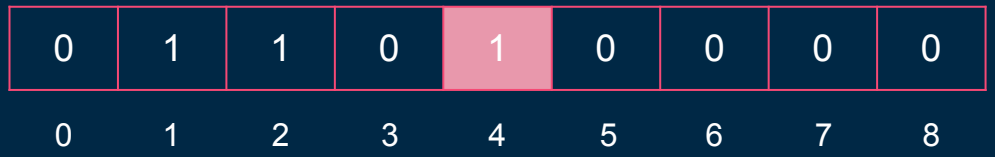


Radix Sort

Input Array

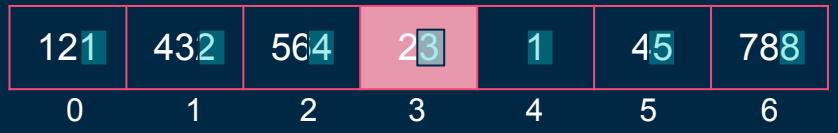


Count Array

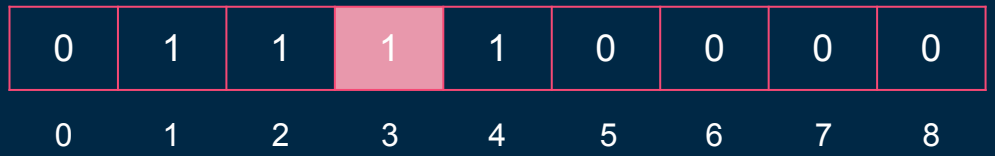


Radix Sort

Input Array

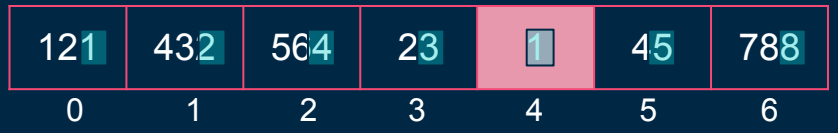


Count Array

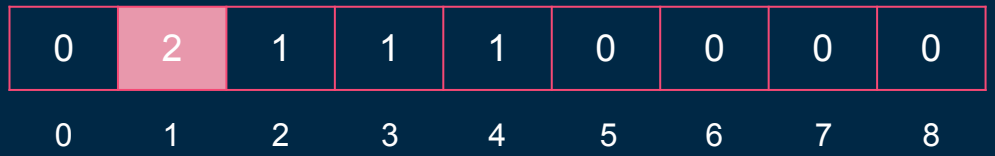


Radix Sort

Input Array

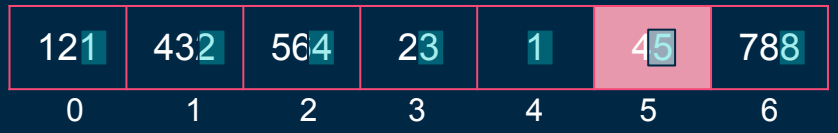


Count Array

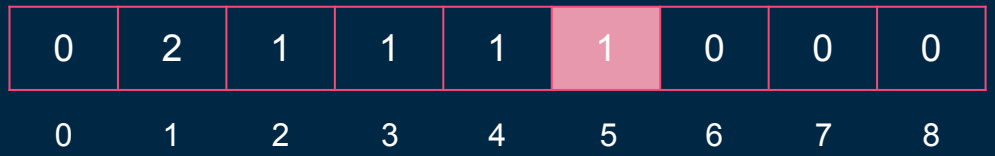


Radix Sort

Input Array

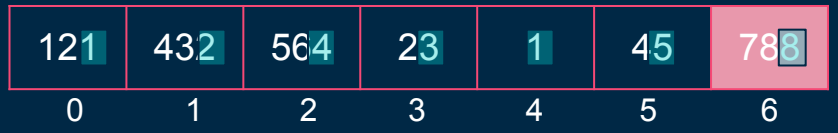


Count Array

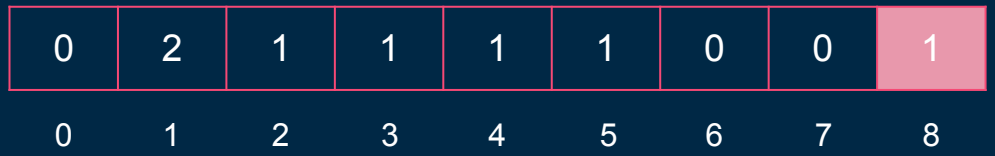


Radix Sort

Input Array



Count Array



Radix Sort

Input Array

121	432	564	23	1	45	788
0	1	2	3	4	5	6

Count Array

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

Subtract 1 from the 0 position so that the resulting sums yield correct positions in the Auxilliary array.

Radix Sort

Input Array

121	432	564	23	1	45	788
0	1	2	3	4	5	6

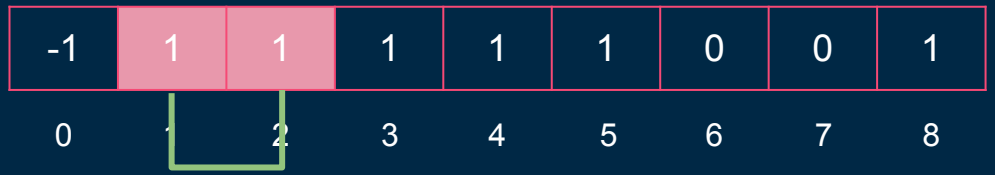
Count Array

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

Radix Sort

Input Array

Count Array



Radix Sort

Input Array

121	432	564	23	1	45	788
0	1	2	3	4	5	6

Count Array

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

Radix Sort

Input Array

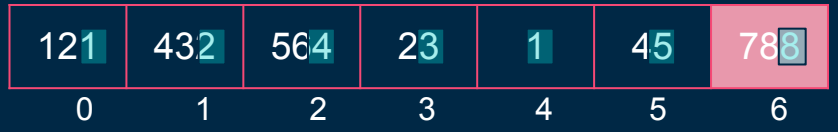
121	432	564	23	1	45	788
0	1	2	3	4	5	6

Count Array

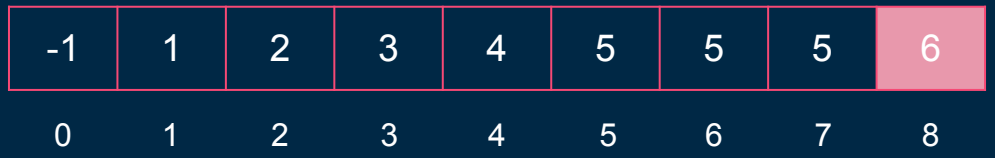
-1	1	2	3	1	1	0	0	1
0	1	2	3	4	5	6	7	8

Radix Sort

Input Array

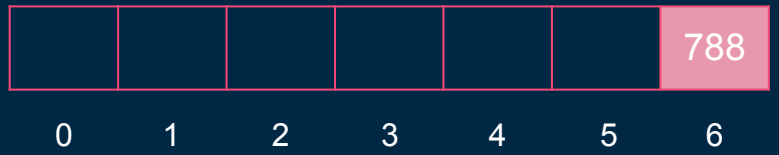


Count Array



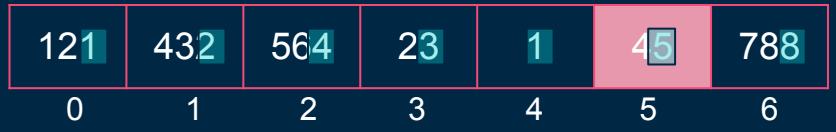
Decrement
the value by 1

Auxilliary Array

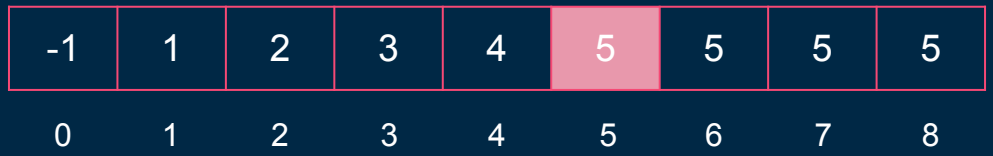


Radix Sort

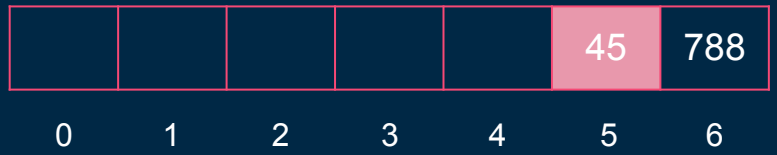
Input Array



Count Array

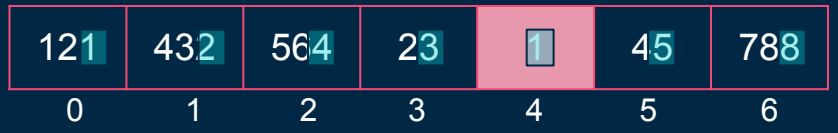


Auxilliary Array

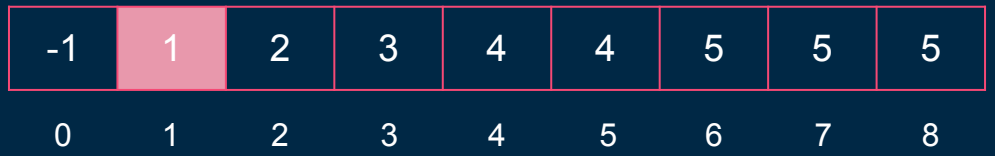


Radix Sort

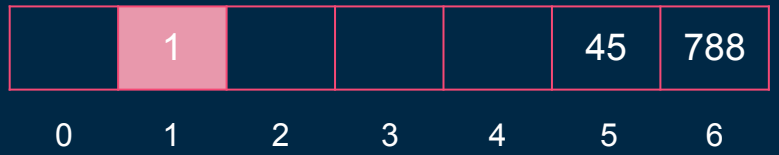
Input Array



Count Array

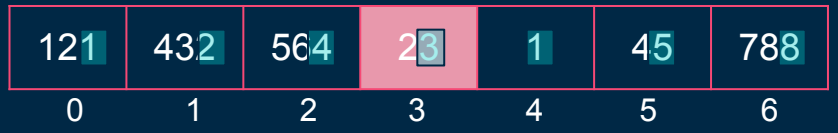


Auxilliary Array

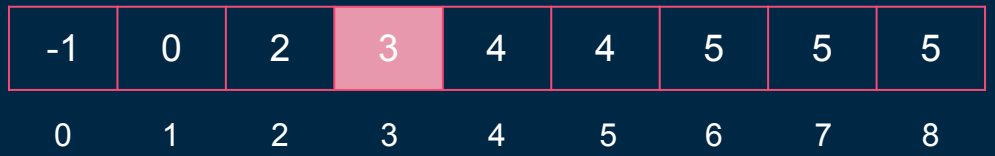


Radix Sort

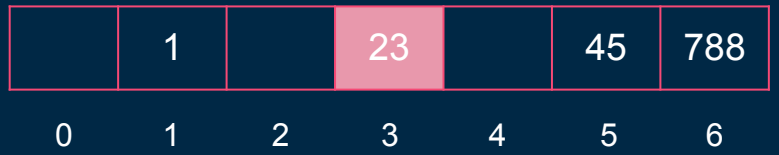
Input Array



Count Array

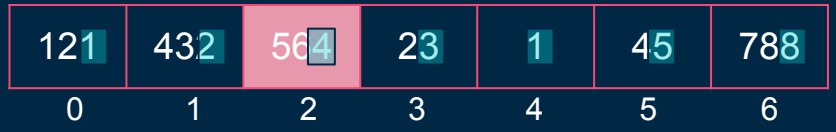


Auxilliary Array

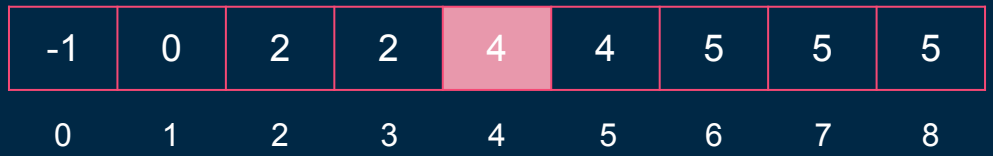


Radix Sort

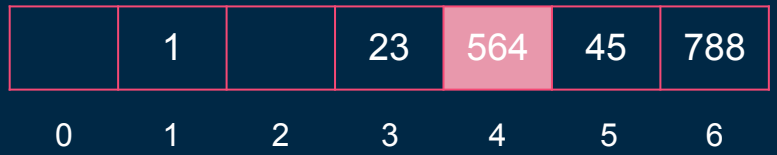
Input Array



Count Array

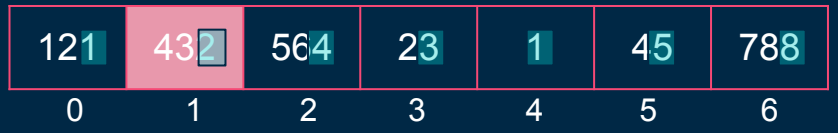


Auxilliary Array

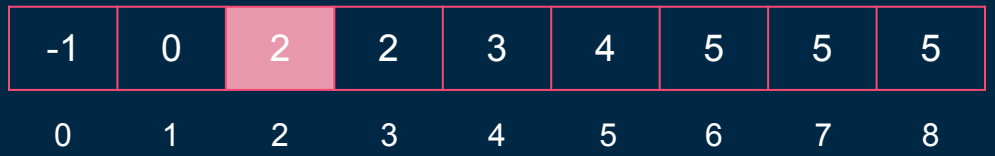


Radix Sort

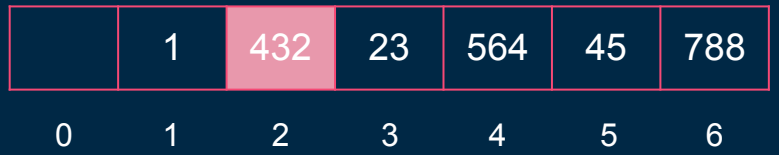
Input Array



Count Array

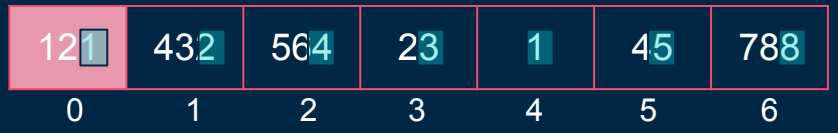


Auxilliary Array

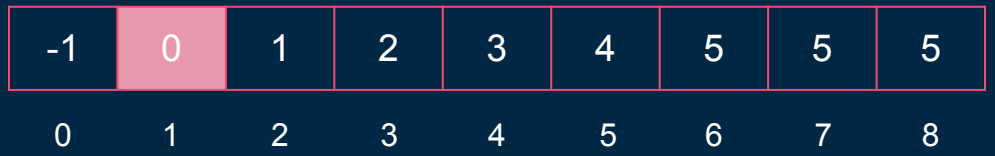


Radix Sort

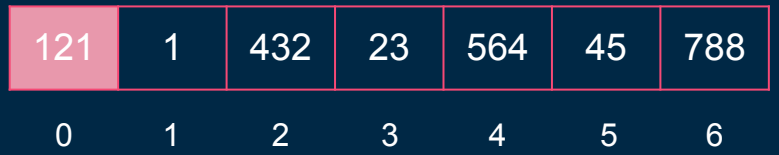
Input Array



Count Array

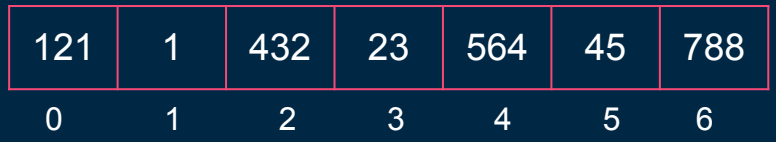


Auxilliary Array

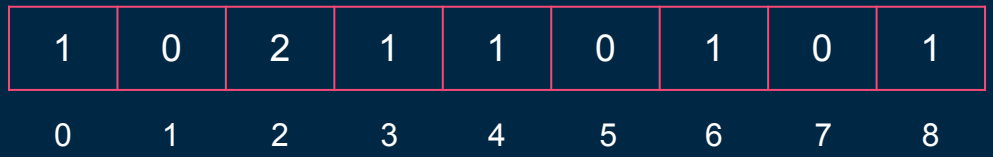


Radix Sort

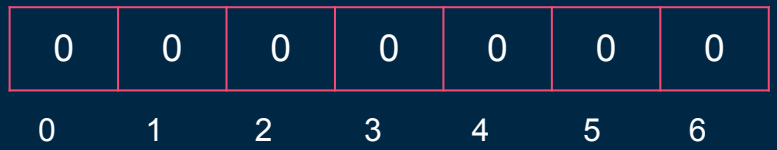
Input Array



Count Array



Auxilliary Array



Radix Sort

Input Array

121	1	432	23	564	45	788
0	1	2	3	4	5	6

Count Array

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

Auxilliary Array

1	121	23	432	45	564	788
0	1	2	3	4	5	6

Radix Sort

Input Array

1	121	23	432	45	564	788
0	1	2	3	4	5	6

Count Array

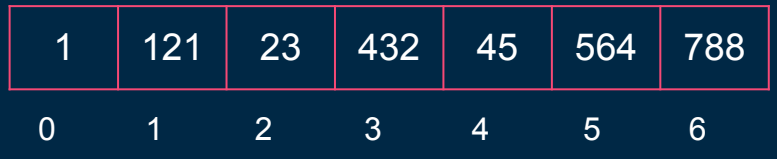
3	1	0	0	1	1	0	1	0
0	1	2	3	4	5	6	7	8

Auxilliary Array

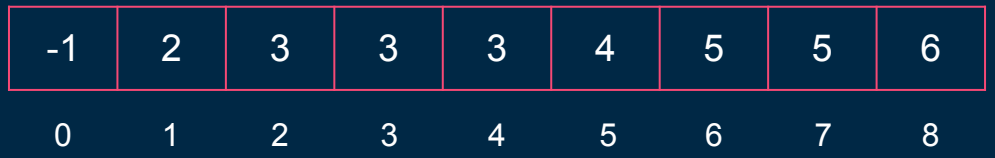
0	0	0	0	0	0	0
0	1	2	3	4	5	6

Radix Sort

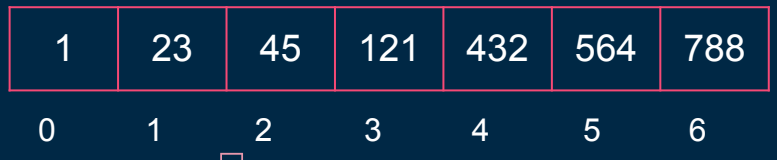
Input Array



Count Array



Auxilliary Array



Internet Code vs Group Code

Internet Code

Group Code

1. Find biggest number

```c
#include<stdio.h>
int get_max (int a[], int n){
   int max = a[0];
   for (int i = 1; i < n; i++)
      if (a[i] > max)
         max = a[i];
   return max;
}
```

```c
int getMax (int a[], int n){
   int max, i;
   for (max = a[0], i = 1; i < n; i++){
        if (a[i] > max)
         max = a[i];
   }

   return max;
}
```

Internet Code

Group Code

2. Apply sorting algorithm to sort elements based on place value

```
void radix_sort (int a[], int n){
   int bucket[10][10], bucket_cnt[10];
   int i, j, k, r, NOP = 0, divisor = 1, lar, pass;
   lar = get_max (a, n);
```

```
   while (lar > 0){
      NOP++;
      lar /= 10;
   }
   for (pass = 0; pass < NOP; pass++){
      for (i = 0; i < 10; i++){
         bucket_cnt[i] = 0;
      }
      for (i = 0; i < n; i++){
         r = (a[i] / divisor) % 10;
         bucket[r][bucket_cnt[r]] = a[i];
         bucket_cnt[r] += 1;
      }
      i = 0;
      for (k = 0; k < 10; k++){
         for (j = 0; j < bucket_cnt[k]; j++){
            a[i] = bucket[k][j];
            i++;
         }
      }
      divisor *= 10;
```

```
m= getMax(arr, n);

bucket =  (Listptr*)malloc(sizeof(Listptr) * BUCKET_SIZE );

for(i=0; i<BUCKET_SIZE; i++){
    bucket[i]=NULL;
}
```

```
while (m > 0){

    for(i=0; i < n ; i++){

        pos = (arr[i] / extract) % 10;

        // Insert Last Linklist
        temp = (Listptr)malloc(sizeof(List));
        temp->data = arr[i];
        temp->link = NULL;

        for(trav = &bucket[pos]; *trav != NULL; trav = &(*trav)->link){}
        *trav = temp;
    }

    for(i=0, j=0; i<BUCKET_SIZE ; i++){
        trav = &bucket[i];
        while(*trav != NULL){
            temp = *trav;
            arr[j++] = temp->data;
            *trav = temp->link;
            free(temp);
        }
    }
    m /= 10;
    extract *= 10;
```

RADIX SORT VARIATIONS

- LSD Radix Sort
  - LSD = least significant digit
  - Sorting of digits starting from the least significant to most significant digit.

- MSD Radix Sort
  - MSD = most significant digit
  - Sorting of digits starting from the most significant digit

Time and Space Complexity

	Average	Best	Worst	Space
Radix Sorting	O(nk)	O(nk)	O(nk)	O(n+k)

THANKS!

- TEAM TIGI -