Class Plan
Advanced Sorting Algorithm - Count Sort, Bucket Sort -Part -1

Relevel
by Unacademy

Educator Introduction (5 mins)

- Greet Students
- Enquire students if they have revised previous class content

Concept to be covered in class (2 mins)

- Introduction to Advance Sorting
  - Count Sort
  - Definition
  - Algorithm
  - Code
  - Properties
- Bucket Sort
  - Definition
  - Algorithm
  - Code
  - Properties

Advanced Sorting Algorithms (15mins)

A technique to interpret data in some organized manner with the help of a comparator is known as sorting. Here comparator means a function that defines the order in which data should be present. For example: Ascending or Descending.

In our previous lecture, we discussed many sorting algorithms and their complexities. For example, Insertion Sort, Selection Sort takes O(n^2). Also, we have optimized it further using Merge Sort, and Quick Sort, which takes O(nlogn).

But sometimes, given some favorable conditions, we can improve the complexities to just linear time O(n) or O(n + k) means only one iteration is enough to sort the data.

Advanced sorting algorithms don't work on a comparison basis like other sorting algorithms. Here we sort with the help of some properties of the data given.

This lecture will discuss some advanced sorting algorithms and their complexities. Also, when we can use it and when it is impossible to use it.

Count Sort

DEFINITION (10mins)

- It is an efficient sorting algorithm that works well in a range of data.
- Here we use an array to store the frequency/count of the elements then sort it accordingly using the algorithm explained below.
- It works based on keys having a specific range. Its name is Count srt since it is counting the number of objects having distinct keys.

ALGORITHM (10mins)

1.Given an unsorted array with n elements and range from 0 to k.
Example: Given array = [1,4,3,2,1,3]

2.Calculate k = (maximum element-minimum element+1).
Example: k = 4 - 1 + 1 = 4

3.Create the sparse array (say frequency) of size k + 1 to store the frequency of elements.

4.Iterate over the given array and to store frequency for element x, increment frequency[x - minElement].
Example: frequency = [0, 2, 1, 2, 1]

5.Now iterate over the frequency array, and update the frequency array in such a way that at each index stores the sum of all the previous frequency. The updated frequency is the index of each element in the given array.
Example: frequency = [0, 2, 3, 5, 6]

6.Now create a result array of the same size as the given array.

7.Start filling the result array with the help of indexes of each element.

8.Iterate over the elements in the given array, get the updated index from the frequency array and put the element in that position in the result array.
resultArray[frequency[givenArray[i]] - minElement - 1] = givenArray[i]

9.Decrement the frequency of the given element.

frequency[givenArray - min] -= 1

10.Repeat from steps 7 to 9 till the end of the given array.
Example: resultArray = [1, 1, 2, 3, 3, 4]

11.Now replace all the values in the givenArray from the result array.

CODE (30mins)

For example: If we start filling from front of givenArray.
(for let i = 0, i < givenArray.length, i++)
givenArray= [{1:a}, {1, b}, {1:c}]

Now frequency array will be [0, 3]

When we start to fill it.

1. resultArray = [null, null, null]
2. resultArray = [null, null, {1:a}], frequency = [0, 2]
3. resultArray = [null, {1:b}, {1:a}], frequency = [0, 1]
4. resultArray = [{1:c}, {1:b}, {1:a}], frequency = [0, 0]

But if we fill it from the back. (for let i = givenArray.length -1, i >= 0, i--)

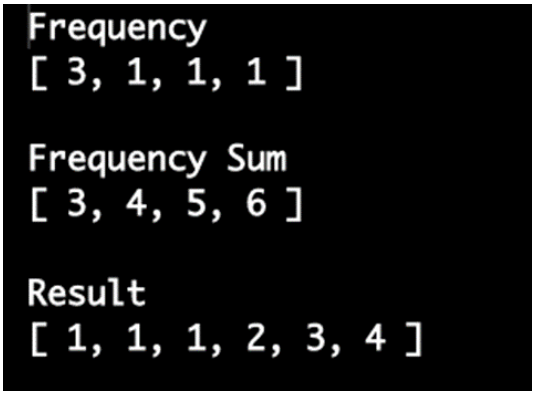
5.resultArray = [{1:a}, {1:b}, {1:c}]

Now let's see the code.

```
const countSort = (givenArray) => {
    const maxElement = Math.max(...givenArray)
   const minElement = Math.min(...givenArray)
   const range = maxElement - minElement + 1;

   let frequency = new Array(range).fill(0);

   for (let i = 0; i < givenArray.length; i++)
       frequency[givenArray[i] - minElement]++;

   console.log("Frequency");

    console.log(frequency);
    console.log();
    for (let i = 1; i < range; i++)
       frequency[i] += frequency[i - 1];

    console.log("Frequency Sum");

    console.log(frequency);
    console.log();

   resultArray = new Array(givenArray.length).fill(0);
    for (let i = givenArray.length - 1; i >= 0; i--) {
       resultArray[frequency[givenArray[i] - minElement] - 1] =
givenArray[i];
       frequency[givenArray[i] - minElement]--;
```

```
    }
     for (let i = 0; i < givenArray.length; i++)
        givenArray[i] = resultArray[i];

}
 let givenArray = [1, 4, 2, 3, 1, 1];
 countSort(givenArray);
 console.log("Result");
console.log(givenArray);
```

Output:



Properties of Count Sort (20mins)

1. **Time Complexity:** O(n + k) where n is the size of a given array and k is the range. So, keep in mind that k is not much greater than n.
2. **Space Complexity:** O(k)
3. It is a stable algorithm.
4. It is very fast.
5. It is an integer-based sorting algorithm, not a comparison-based like a merge, insertion, and others. If you see the algorithm. Here we have not used any comparator at all.
6. Not suitable for sorting large datasets.
7. Also able to sort negative integers.
8. Used in radix sort as a sub-routine. We will discuss it later.

Bucket Sort

DEFINITION (20mins)

Bucket sort is a sorting technique where we divide the given array into buckets of elements depending upon some property of the dataset. Then separately sort all the buckets using different sorting algorithms or recursively call the bucket Sort.

Bucket sort is also known as bin sort. We can interpret that bucket sort as a collective form of different sorting algorithms. Hence it is complex but also very versatile.

Bucket sort is a general form of counting sort. If there is only one element in each bucket, it will be similar to the counting sort. But counting sort can not be directly implemented on float data. Hence we can use bucket sort there.

Bucket sort will only work well when the data is uniformly distributed among the range. If so, multiple buckets will be created, and time complexity will go down to linear time O(n). But if it is not uniformly distributed, then O(n^2) because we have only one bucket in working condition.
There are many ways to put value in the bucket.
For example using a linked list, using a 2-D array. Here in this example, we will use a 2-D array only.

ALGORITHM (20mins)

Let's go through the algorithm to understand it better.

1.Given an array of n elements
Example: givenArray = [0.29, 0.34, 0.19, 0.39, 0.21, 0.41]

2.We have multiple options to create the bucket. By dividing the array using the least significant value or the most significant value. But if you can see here, if we divide using least significant we have only 3 buckets (1,4 and 9). But if we go with the most significant we have more buckets (1, 2, 3, and 4).

3.Create a bucket using a 2-d array of size 5.

4.Let's put the values in the bucket now, by multiplying values by 10 and floor the value.
Example: 0.34 * 10 = 3.4 => floor(3.4) = 3. So, it will be placed in 3rd index.
bucket = [[null, 0.19, 0.29, 0.34, 0.41],
[null, null, 0.21, 0.39, null]]
So, two values in the 2nd and 3rd index.
5.Now, use some sorting algorithm to sort it. For example, insertion sort or merge sort. You can use the same bucket sort again recursively. Just, you have to now multiply with a hundred to create buckets among them for the second most significant value.

6.Go to each bucket and sort the values.
Example: bucket = [[null, 0.19, 0.21, 0.34, 0.41],
[null, null, 0.29, 0.39, null]]
7.Iterate over the bucket and again put the values in the givenArray in sorted form.
Example: givenArray = [0.19, 0.21, 0.29, 0.34, 0.39, 0.41]

Why do we say that bucket sort is linear time complexity? (20mins)

As I have mentioned bucket sort can be used with various variations. So, generally, if the data is uniformly distributed then we have not many elements in any of the buckets.

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
Quicksort	$n \log n$	$n \log n$	n^2	$\log n$	No	Partitioning	Quicksort is usually done in-place with $O(\log n)$ stack space.[5][6]
Merge sort	$n \log n$	$n \log n$	$n \log n$	n	Yes	Merging	Highly parallelizable (up to $O(\log n)$ using the Three Hungarians' Algorithm).[7]
In-place merge sort	—	—	$n \log^2 n$	1	Yes	Merging	Can be implemented as a stable sort based on stable in-place merging.[8]
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	No	Partitioning & Selection	Used in several STL implementations.
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No	Selection	
Insertion sort	n	n^2	n^2	1	Yes	Insertion	$O(n + d)$, in the worst case over sequences that have d inversions.
Block sort	n	$n \log n$	$n \log n$	1	Yes	Insertion & Merging	Combine a block-based $O(n)$ in-place merge algorithm[9] with a bottom-up merge sort.
Timsort	n	$n \log n$	$n \log n$	n	Yes	Insertion & Merging	Makes n-1 comparisons when the data is already sorted or reverse sorted.
Selection sort	n^2	n^2	n^2	1	No	Selection	Stable with $O(n)$ extra space, when using linked lists, or when made as a variant of Insertion Sort instead of swapping the two items.[10]
Cubesort	n	$n \log n$	$n \log n$	n	Yes	Insertion	Makes n-1 comparisons when the data is already sorted or reverse sorted.
Shellsort	$n \log n$	$n^{4/3}$	$n^{3/2}$	1	No	Insertion	Small code size.
Bubble sort	n	n^2	n^2	1	Yes	Exchanging	Tiny code size.
Exchange sort	n^2	n^2	n^2	1	Yes	Exchanging	Tiny code size.
Tree sort	$n \log n$	$n \log n$	$n \log n$ (balanced)	n	Yes	Insertion	When using a self-balancing binary search tree.
Cycle sort	n^2	n^2	n^2	1	No	Selection	In-place with theoretically optimal number of writes.
Library sort	$n \log n$	$n \log n$	n^2	n	No	Insertion	Similar to a gapped insertion sort. It requires randomly permuting the input to warrant with-high-probability time bounds, which makes it not stable.
Patience sorting	n	$n \log n$	$n \log n$	n	No	Insertion & Selection	Finds all the longest increasing subsequences in $O(n \log n)$.
Smoothsort	n	$n \log n$	$n \log n$	1	No	Selection	An adaptive variant of heapsort based upon the Leonardo sequence rather than a traditional binary heap.
Strand sort	n	n^2	n^2	n	Yes	Selection	
Tournament sort	$n \log n$	$n \log n$	$n \log n$	n[11]	No	Selection	Variation of Heapsort.
Cocktail shaker sort	n	n^2	n^2	1	Yes	Exchanging	A variant of Bubblesort which deals well with small values at end of list
Comb sort	$n \log n$	n^2	n^2	1	No	Exchanging	Faster than bubble sort on average.
Gnome sort	n	n^2	n^2	1	Yes	Exchanging	Tiny code size.
Odd–even sort	n	n^2	n^2	1	Yes	Exchanging	Can be run on parallel processors easily.

Here this is just for a reference that some algorithms even sort in linear time at the best time. Also, if you use recursive bucket sort or counting sort even, you can achieve linear time complexity.

Now, let's suppose you are using insertion sort only. Then as well as the size of each bucket is very less then. Sorting each bucket using insertion sort will take O(k^2), where k is the size of the bucket or even with merge sort O(klogk).
Hence k << n. Then it means n have more significance here, and it will drive the overall complexity.
Hence the complexity will always be O(n + k).

In the worst condition, when there is only 1 bucket and data is not uniform, the worst-case complexity of bucket sort will be O(n^2).

So, choose wisely the algorithm before use.

CODE (30mins)

Now, let's go through the code:

```
const bucketSort = (givenArray, numberOfBucket) => {

    let bucket = new Array(numberOfBucket);
    for (let i = 0; i < numberOfBucket; i++){
        bucket[i] = [];
    }

    for (let i = 0; i < givenArray.length; i++) {
        let index = Math.floor(10 * givenArray[i]);
        bucket[index].push(givenArray[i]);
    }

    console.log("Before Sorting Each Bucket");
    console.log(bucket);

    for (let i = 0; i < numberOfBucket; i++)
        bucket[i].sort();

    console.log("After Sorting Each Bucket");
    console.log(bucket);

    let index = 0;
    for (let i = 0; i < numberOfBucket; i++)
        for (let j = 0; j < bucket[i].length; j++)
            givenArray[index++] = bucket[i][j];
}


let givenArray = [0.29, 0.34, 0.19, 0.39, 0.21, 0.41];
let numberOfBucket = 10;

bucketSort(givenArray, numberOfBucket);

console.log(givenArray);
```

Output:

```
Before Sorting Each Bucket
[
  [],
  [ 0.19 ],
  [ 0.29, 0.21 ],
  [ 0.34, 0.39 ],
  [ 0.41 ],
  [],
  [],
  [],
  [],
  []
]
After Sorting Each Bucket
[
  [],
  [ 0.19 ],
  [ 0.21, 0.29 ],
  [ 0.34, 0.39 ],
  [ 0.41 ],
  [],
  [],
  [],
  [],
  []
]
[ 0.19, 0.21, 0.29, 0.34, 0.39, 0.41 ]
```

Let's suppose you want to sort only integers. You just have to make a bucket from either the most significant value or least, but you have to change the function for putting values in the bucket by dividing by some base. Also, you can add the logic for how we can work in ranges using this.

We just have to calculate the number of buckets by dividing the max element by the base we want to use minus max element by base + 1.
And while calculating the index, just subtracts the index of minElement.

For example:

```
const bucketSort = (givenArray) => {

    const maxElement = Math.max(...givenArray);
    const minElement = Math.min(...givenArray);
    const base = 10;

    const numberOfBucket = Math.floor(maxElement/base) -
Math.floor(minElement/base) + 1;

    let bucket = new Array(numberOfBucket);
    for (let i = 0; i < numberOfBucket; i++){
        bucket[i] = [];
    }venArray);
```

```
    for (let i = 0; i < givenArray.length; i++) {
        let index = Math.floor(givenArray[i] / 10) -
Math.floor(minElement/base); // Starting from the minElement only not
from 0.
        bucket[index].push(givenArray[i]);
    }

    console.log("Before Sorting Each Bucket");
    console.log(bucket);

    for (let i = 0; i < numberOfBucket; i++)
        bucket[i].sort();

    console.log("After Sorting Each Bucket");
    console.log(bucket);

    let index = 0;
    for (let i = 0; i < numberOfBucket; i++)
        for (let j = 0; j < bucket[i].length; j++)
            givenArray[index++] = bucket[i][j];
}


let givenArray = [29, 34, 19, 39, 21, 41];

bucketSort(givenArray);

console.log(gi
```

Output:

```
Before Sorting Each Bucket
[ [ 19 ], [ 29, 21 ], [ 34, 39 ], [ 41 ] ]
After Sorting Each Bucket
[ [ 19 ], [ 21, 29 ], [ 34, 39 ], [ 41 ] ]
[ 19, 21, 29, 34, 39, 41 ]
```

Properties of Bucket Sort (10mins)

1. **Time Complexity:** O(n + k) where n is the number of buckets and k is the complexity of sorting a bucket.
2. **Space Complexity:** O(n) where n is the number of elements.
3. It is based on the assumption that data is uniformly distributed among the range.
4. It is also a non-comparison-based algorithm as we are using some property to divide the bucket. However, if we use a comparison-based algorithm to sort a bucket, it will still be a partial comparison-based algorithm.

MCQ

1.Average Time Complexity of counting sort.
a) O(n^2)
b) O(nlogn)
c) O(n + k)
d) O(n)
Ans: O(n + k)

2. Worst Case Time Complexity for bucket sort.
a) O(n^2)
b) O(nlogn)
c) O(2^n)
d) O(n + k)
Ans: O(n^2)

3. If we increase the base in radix sort. The time complexity will
a) Increase
b) Decrease
c) Remains Constant
d) None of the above.
Ans: Decrease

4. Bucket sort can be used with
a) Insertion Sort
b) Counting Sort
c) Bucket Sort
d) All of the above
Ans: All of the above

5. Radix sort uses which of these as a subroutine
a) Count sort
b) Bucket sort
c) Shell sort
d) Quicksort
Ans: Count Sort

Assignment Problem:

1) Write Bucket sort for sorting negative floating numbers as well using linked lists.

2) Given an array. You need to sort array using iterative quick sort algorithm
Input - [3,6,5,2,10]
Output - [2,3,5,6,10]