Sorting Algorithms

Content

Heap Sort

About

Heap sort is a comparison-based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the minimum element and place the minimum element at the beginning. We repeat the same process for the remaining elements.

How to “heapify” a tree?

The process of reshaping a binary tree into a Heap data structure is known as ‘heapify’. A binary tree is a tree data structure that has two child nodes at max. If a node’s children nodes are ‘heapified’, then only the ‘heapify’ process can be applied over that node. A heap should always be a complete binary tree.

Starting from a complete binary tree, we can modify it to become a Max-Heap by running a function called ‘heapify’ on all the non-leaf elements of the heap. i.e. ‘heapify’ uses recursion.

Heap Sort Algorithm for sorting in increasing order:

1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of the heap by 1. Finally, heapify the root of the tree.
3. Repeat step 2 while the size of the heap is greater than 1.

How to build the heap?

Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom-up order.

```
Input data: 4, 10, 3, 5, 1
         4(0)
        /   \
     10(1)   3(2)
    /   \
 5(3)    1(4)

The numbers in bracket represent the indices in the array
representation of data.

Applying heapify procedure to index 1:
         4(0)
        /   \
    10(1)    3(2)
    /   \
5(3)    1(4)

Applying heapify procedure to index 0:
        10(0)
        /  \
     5(1)  3(2)
    /   \
 4(3)    1(4)
The heapify procedure calls itself recursively to build heap
 in top down manner.
```

Implementation

https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Sorting/Codes/HeapSort.cpp

Notes

1. Heap sort is an in-place algorithm.
2. Its typical implementation is not stable, but can be made stable.
3. Time Complexity: The time complexity of heapify is O(Logn). The time complexity of createAndBuildHeap() is O(n) and the overall time complexity of Heap Sort is O(nLogn).

Advantages of heapsort

1. Efficiency – The time required to perform Heap sort increases logarithmically while other algorithms may grow exponentially slower as the number of items to sort increases. This sorting algorithm is very efficient.
2. Memory Usage – Memory usage is minimal because apart from what is necessary to hold the initial list of items to be sorted, it needs no additional memory space to work.

Applications of HeapSort

1. Sort a nearly sorted (or K sorted) array
2. k largest(or smallest) elements in an array

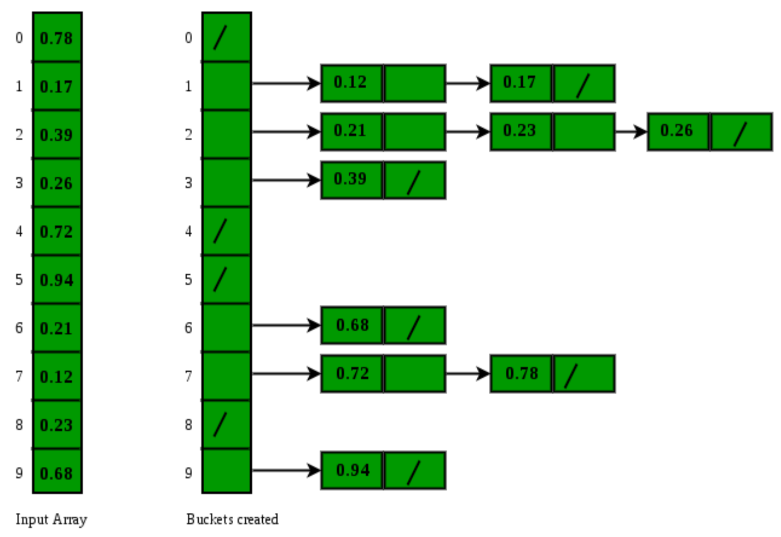
Heap sort algorithms have limited uses because Quicksort and Mergesort are better in practice.

Bucket Sort

The idea is to use bucket sort is the below algorithm:-

bucketSort(arr[], n)

1. Create n empty buckets (Or lists).
2. Do following for every array element arr[i].
    a. a) Insert arr[i] into bucket[n*array[i]]
3. Sort individual buckets using insertion sort.
4. Concatenate all sorted buckets.



Bucket Sort for numbers having integer part

Algorithm

1. Find maximum element and minimum of the array
2. Calculate the range of each bucket

   range = (max - min) / n   [n is the number of buckets]

3. Create n buckets of calculated range
4. Scatter the array elements to these buckets

   BucketIndex = (arr[i] - min) / range

5. Now sort each bucket individually
6. Gather the sorted elements from buckets to original array

Examples

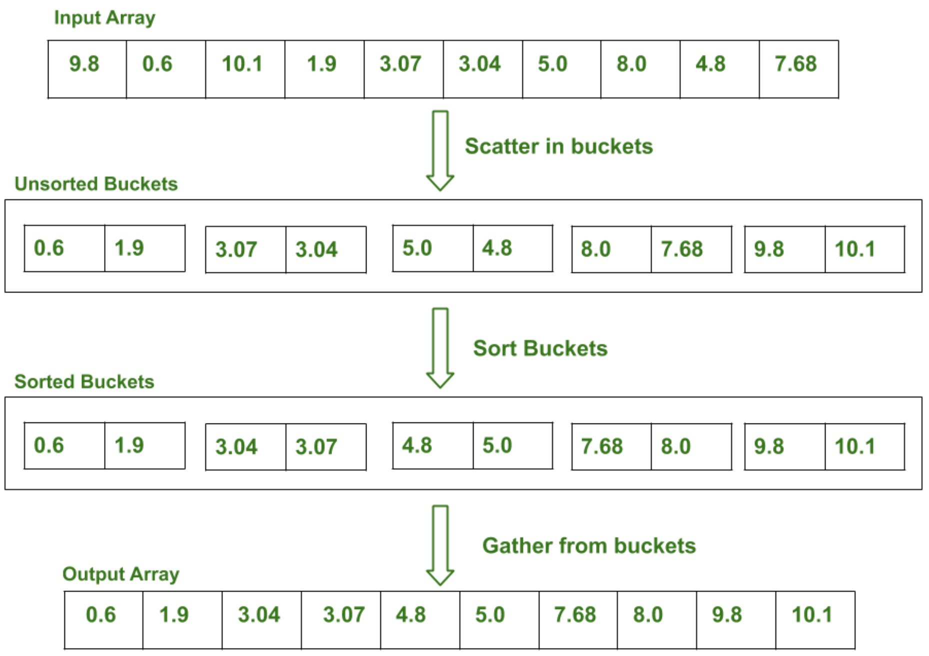
Input :

Unsorted array: [9.8 , 0.6 , 10.1 , 1.9 , 3.07 , 3.04 , 5.0 , 8.0 , 4.8 , 7.68]

No of buckets: 5

Output :

Sorted array: [0.6 , 1.9 , 3.04 , 3.07 , 4.8 , 5.0 , 7.68 , 8.0 , 9.8 , 10.1]



Implementation

https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Sorting/Codes/BucketSort.cpp

Time Complexity

If we assume that insertion in a bucket takes O(1) time then steps 1 and 2 of the above algorithm clearly take O(n) time. The O(1) is easily possible if we use a linked list to represent a bucket (In the following code, C++ vector is used for simplicity).

Step 4 also takes O(n) time as there will be n items in all buckets.

The main step to analyze is step 3. This step also takes O(n) time on average if all numbers are uniformly distributed.

QuickSort

About

Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

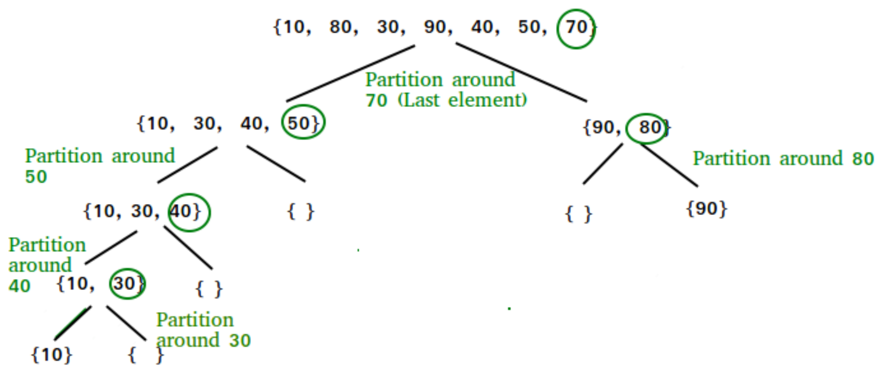
1. Always pick the first element as a pivot.
2. Always pick the last element as the pivot (implemented below)
3. Pick a random element as a pivot.
4. Pick median as a pivot.

The key process in quickSort is a partition(). The target of partitions is, given an array and an element x of the array as a pivot, put x at its correct position in a sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
   if (low < high)
   {
      /* pi is partitioning index, arr[pi] is now
         at right place */
      pi = partition(arr, low, high);

      quickSort(arr, low, pi - 1);  // Before pi
      quickSort(arr, pi + 1, high); // After pi
   }
```

}



Partition Algorithm

The logic is simple, we start from the leftmost element and keep track of the index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap the current element with arr[i]. Otherwise, we ignore the current element.

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

Implementation

https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Sorting/Codes/QuickSort.cpp

Analysis of QuickSort

Time taken by QuickSort, in general, can be written as follows.

T(n) = T(k) + T(n-k-1) + (n)

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements that are smaller than the pivot.

Worst Case: The worst case occurs when the partition process always picks the greatest or smallest element as a pivot. If we consider the above partition strategy where the last element is always picked as a pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is the recurrence for the worst case.

T(n) = T(0) + T(n-1) + O(n)
which is equivalent to
T(n) = T(n-1) + O(n)

The solution of the above recurrence is O(n2).

Is QuickSort stable?
The default implementation is not stable. However, any sorting algorithm can be made stable by considering indexes as comparison parameters.
Is QuickSort In-place?
As per the broad definition of an in-place algorithm, it qualifies as an in-place sorting algorithm as it uses extra space only for storing recursive function calls but not for manipulating the input.