Name: Lesego Nzimande

Student ID: Computer Science

Semester/Year: 2020/2021

Date: 14/01/2021

Subject: Algorithms & data structures

Introduction

Sorting algorithms are a set of instructions that take an array or list as an input and arrange the items into a particular order; these algorithms are most commonly in numerical or a form of alphabetical order, and can be in ascending (A-Z, 0-9) or descending (Z-A, 9-0) order. Since sorting can often reduce the complexity of a problem, it is an important algorithm in Computer Science. These algorithms have direct applications in searching algorithms, database algorithms, divide and conquer methods, data structure algorithms, and many more.

Counting sort

Counting sort is a sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values (kind of hashing). Then doing some arithmetic to calculate the position of each object in the output sequence.

Example:

For simplicity, consider the data in the range 0 to 9.

Input data: 1, 4, 1, 2, 7, 5, 2

1) Take a count array to store the count of each unique object.

Index: 0 1 2 3 4 5 6 7 8 9

Count: 0 2 2 0 1 1 0 1 0 0

2) Modify the count array such that each element at each index

stores the sum of previous counts.

Index: 0 1 2 3 4 5 6 7 8 9

Count: 0 2 4 4 5 6 6 7 7 7

The modified count array indicates the position of each object in

the output sequence.

3) Output each object from the input sequence followed by

decreasing its count by 1.

Process the input data: 1, 4, 1, 2, 7, 5, 2. Position of 1 is 2.

Put data 1 at index 2 in output. Decrease count by 1 to place

next data 1 at an index 1 smaller than this index.

Points to be noted:

1. Counting sort is efficient if the range of input data is not significantly greater than the number of objects to be sorted. Consider the situation where the input sequence is between range 1 to 10K and the data is 10, 5, 10K, 5K.

2. It is not a comparison based sorting. It running time complexity is O(n) with space proportional to the range of data.

3. It is often used as a sub-routine to another sorting algorithm like radix sort.

4. Counting sort uses a partial hashing to count the occurrence of the data object in O(1).

5. Counting sort can be extended to work for negative inputs also.

Bucket sort

Bucket sort is mainly useful when input is uniformly distributed over a range. For example, consider the following problem.

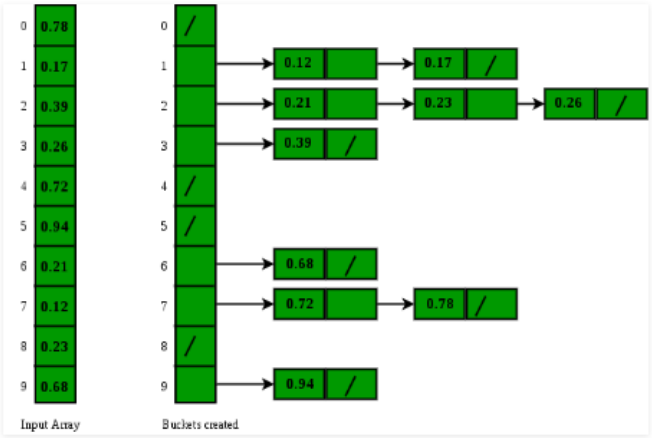
Sort a large set of floating point numbers which are in range from 0.0 to 1.0 and are uniformly distributed across the range. How do we sort the numbers efficiently?

A simple way is to apply a comparison based sorting algorithm. The lower bound for Comparison based sorting algorithm (Merge Sort, Heap Sort, Quick-Sort .. etc) is Ω(n Log n), i.e., they cannot do better than nLogn.

Can we sort the array in linear time? Counting sort can not be applied here as we use keys as index in counting sort. Here keys are floating point numbers.

The idea is to use bucket sort. Following is bucket algorithm:

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



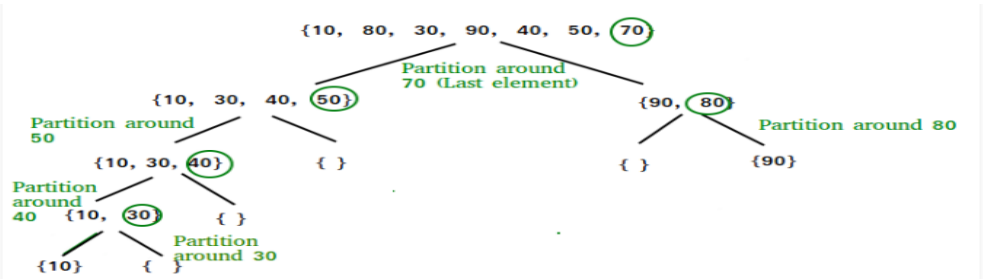
Time Complexity: If we assume that insertion in a bucket takes O(1) time then steps 1 and 2 of the above algorithm clearly take O(n) time. The O(1) is easily possible if we use a linked list to represent a bucket (In the following code, C++ vector is used for simplicity). Step 4 also takes O(n) time 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.

Quick / Index sort

QuickSort is a Divide and Conquer algorithm. It picks an element as 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 first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in 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.



Partition Algorithm

There can be many ways to do partition, following pseudo code adopts the method given in CLRS book. The logic is simple, we start from the leftmost element and keep track of index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap current element with arr[i]. Otherwise we ignore current element.

Analysis of QuickSort

Time taken by QuickSort in general can be written as following.

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

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

The time taken by QuickSort depends upon the input array and partition strategy. Following are three cases.

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

T(n) = T(0) + T(n-1) + \theta(n)

which is equivalent to

T(n) = T(n-1) + \theta(n)

The solution of above recurrence is \theta(n2).

Best Case: The best case occurs when the partition process always picks the middle element as pivot. Following is recurrence for best case.

T(n) = 2T(n/2) + \theta(n)

The solution of above recurrence is \theta(nLogn). It can be solved using case 2 of Master Theorem.

Average Case:

To do average case analysis, we need to consider all possible permutation of array and calculate time taken by every permutation which doesn't look easy.

We can get an idea of average case by considering the case when partition puts O(n/9) elements in one set and O(9n/10) elements in other set. Following is recurrence for this case.

T(n) = T(n/9) + T(9n/10) + \theta(n)

Solution of above recurrence is also O(nLogn)

Although the worst case time complexity of QuickSort is O(n2) which is more than many other sorting algorithms like Merge Sort and Heap Sort, QuickSort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data. QuickSort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data. However, merge sort is generally considered better when data is huge and stored in external storage.

Is QuickSort stable?

The default implementation is not stable. However any sorting algorithm can be made stable by considering indexes as comparison parameter.

Is QuickSort In-place?

As per the broad definition of 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.

What is 3-Way QuickSort?

In simple QuickSort algorithm, we select an element as pivot, partition the array around pivot and recur for subarrays on left and right of pivot.

Consider an array which has many redundant elements. For example, {1, 4, 2, 4, 2, 4, 1, 2, 4, 1, 2, 2, 2, 2, 4, 1, 4, 4, 4}. If 4 is picked as pivot in Simple QuickSort, we fix only one 4 and recursively process remaining occurrences. In 3 Way QuickSort, an array arr[l..r] is divided in 3 parts:

a) arr[l..i] elements less than pivot.

b) arr[i+1..j-1] elements equal to pivot.

c) arr[j..r] elements greater than pivot.

How to implement QuickSort for Linked Lists?

QuickSort on Singly Linked List

QuickSort on Doubly Linked List

Can we implement QuickSort Iteratively?

Yes, please refer Iterative Quick Sort.

Why Quick Sort is preferred over MergeSort for sorting Arrays?

Quick Sort in its general form is an in-place sort (i.e. it doesn't require any extra storage) whereas merge sort requires O(N) extra storage, N denoting the array size which may be quite expensive. Allocating and de-allocating the extra space used for merge sort increases the running time of the algorithm. Comparing average complexity we find that both type of sorts have O(NlogN) average complexity but the constants differ. For arrays, merge sort loses due to the use of extra O(N) storage space.

Most practical implementations of Quick Sort use randomized version. The randomized version has expected time complexity of O(nLogn). The worst case is possible in randomized version also, but worst case doesn't occur for a particular pattern (like sorted array) and randomized Quick Sort works well in practice.

Quick Sort is also a cache friendly sorting algorithm as it has good locality of reference when used for arrays.

Quick Sort is also tail recursive, therefore tail call optimizations is done.

Why MergeSort is preferred over QuickSort for Linked Lists?

In case of linked lists the case is different mainly due to difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike array, in linked list, we can insert items in the middle in O(1) extra space and O(1) time. Therefore merge operation of merge sort can be implemented without extra space for linked lists.

In arrays, we can do random access as elements are continuous in memory. Let us say we have an integer (4-byte) array A and let the address of A[0] be x then to access A[i], we can directly access the memory at (x + i*4). Unlike arrays, we can not do random access in linked list. Quick Sort requires a lot of this kind of access. In linked list to access i'th index, we have to travel each and every node from the head to i'th node as we don't have continuous block of memory. Therefore, the

overhead increases for quick sort. Merge sort accesses data sequentially and the need of random access is low.

References:

https://www.interviewbit.com/tutorial/sorting-algorithms/

https://medium.com/@george.seif94/a-tour-of-the-top-5-sorting-algorithms-with-python-code-43ea9aa02889

https://www.studytonight.com/data-structures/introduction-to-sorting

https://codeburst.io/the-o-n-sorting-algorithm-of-your-dreams-d2dd31b06848

https://www.youtube.com/watch?v=Y0-9xN36MrM

https://randerson112358.medium.com/sorting-algorithms-6005e9ddd8c0