CS 584 Project Report

Empirical Analysis of Linear Time Sorting Algorithms

By Neha Gadge

Description:

This project implements and compares three linear time sorting algorithms – Radix sort, Bucket sort, and Counting sort[1]. These algorithms are compared with input size of 100K, 1M, 10M, and 20M elements. Various parameters are considered while comparing each algorithm individually. Base for radix sort, distribution for bucket, and range for counting sort are varied and their individual performance is studied by changing the above parameters.

Implementation Details:

The following programs are implemented in this project.

Main Module: Main.cpp

The program begins execution in main.cpp. This program accepts two or three command line arguments. The first argument specifies the name of the sorting algorithm to be implemented, the second argument takes the name of the input set on which the algorithm is to be tested. The third argument takes the value of parameter for radix and bucket sort. There is no third argument for counting sort. The user will provide algorithm name, input filename, and parameter value (if applicable) to the program via command line. Then based on the inputs provided by the user, the program first reads dataset from the input file into a vector, then runs the sorting algorithm on the dataset. After sorting the input, the program checks if the array is sorted or not and then generates an output file with the sorted numbers. It will also display the required running time taken by each algorithm. This way we can find out how much time each algorithm has taken to run, on that particular dataset.

This program can be compiled using the command: **g++ -O3 -g main.cpp –o sorting.exe**
Run the program by command : **./sorting.exe <algorithm name> <input dataset name> <parameter value if applicable>**

E.g., ./sorting.exe counting input6.txt
./sorting.exe radix input6.txt 100
./sorting.exe bucket input6.txt 10

Where 100 is base for radix sort and 10 is the number of buckets.

Input Sets: input_generate.cpp

This is a separate file which is specifically written to generate inputs. This program generates ten different input files:

- **input2 to input8** are files with random numbers of varying ranges: {100, 1K, 10K, 100K, 1M, 10M, 20M} respectively.
- input9.txt is a file with 10M numbers of range 0-1K.
- input10.txt is a file with 10M numbers of range 0-10K.
- input11.txt is a file with 10M constant numbers.
- input12.txt is a file with 10M numbers in descending order.

While input 2 to 8 are varying the ranges, input 9, 10 and 11 are varying the distribution.

Any of these input files are fed to the main program via command line arguments where the data inside them is sorted and output is returned in a separate output file.

Input_generate.cpp can be compiled by : **g++ -O3 –g input_generate.cpp –o gen.exe**

Input set can be generated by command: **./gen.exe**

Common Functions: common.h

All the common functions: converting file to vector, displaying sorted vector on command line, writing sorted vector in output file, checking if the input is sorted or not – are written in this separate header file. This header is included in all other cpp files, including main.cpp.

readFileToVector() : function to read input file and convert it to vector

writeFileToVector() : function to write sorted vector to output file

displayVector() : function to display sorted vector on standard output.

check() : function to check if the array is sorted.

Sorting Algorithms: radixsort.cpp, bucketsort.cpp, coutingsort.cpp

Three separate files for each algorithm are made respectively. The respective algorithm is called from main function depending on what is the second command line argument provided by the user. Input to the algorithm is an unsorted vector and output will be a sorted vector, written to a file. We will go through each algorithm in detail below:

radixsort.cpp

In radix sort, we first sort the elements based on last digit (least significant digit). Then the result is again sorted by second-last digit. Repeat this process for all digits until we reach most significant digit.

We are using LSD radix sort here, which sorts each digit by counting sort algorithm. User inputs the base which is then used to do the sorting.

Algorithm:

1. Calculate the dynamic range of array by subtracting min element from max element of array.
2. Calculate the number of digits by ceil(log10(range))/ log10(base).
3. Call counting sort to sort the elements of every digit iteratively from 0 to number of digits.
4. After sorting, combine the elements and return the sorted array.

Radix Sort Time Complexity		
Worst Case	Best Case	Average Case
O(nk)	O(nk)	O(nk)

where, n is the number of elements and k is the number of digits in the largest element.

bucketsort.cpp

In bucket sort, we create buckets and put elements in them. Then we apply some sorting algorithm to sort the elements in each bucket. Finally, we take the elements out and join them to get the sorted result. The number of buckets to be created is taken from the user.

Algorithm

1. Input the number of buckets n from the user and make an array of n buckets, B[n].
2. Find the value of divider element which is used to put the elements in corresponding bucket. Divider = ceil((max+1) / bucket).
3. Put the input element, a[i] in the correct bucket by using the formula B[j] = a[i], where j = floor(a[i] / divider), where B[j] is bucket array.
4. Since any method can be used to do internal bucket sorting, I have used sort() method used to sort vectors in C++.
5. Combine the sorted elements from each bucket and return the sorted array.

Bucket Sort Time Complexity		
Worst Case	Best Case	Average Case
$O(n^2)$	O(n+k)	O(n+k)

where, n is the number of elements and k is the number of buckets.

countingsort.cpp

In counting sort, we count the number of elements having distinct key values. Then we again loop over the elements to determine the final position of each key value of elements.

Algorithm

1. Create an empty map to store the frequency of each element in array.
2. Store the distinct elements in array as key and their respective counts as values.
3. Traverse the map and get output with sorted elements. Map will iterate based on the sorted order of the keys.
4. Return the sorted elements.

Counting Sort Time Complexity		
Worst Case	Best Case	Average Case
O(n+k)	O(n+k)	O(n+k)

where, n is the number of elements and k is the range of inputs.

Evaluation:

This project is evaluated on Intel Core i7 8700K processor with 16GB RAM and 512GB storage. Each sorting algorithm is tested on unsorted vector and the time required to sort the vector is reported. This time is measured in milliseconds. Each combination is fed to algorithm 5 times and final time is calculated by taking average of the 5 different running times. Input sets with smaller ranges are not taken into consideration since the time taken was negligible. Only larger datasets are included.

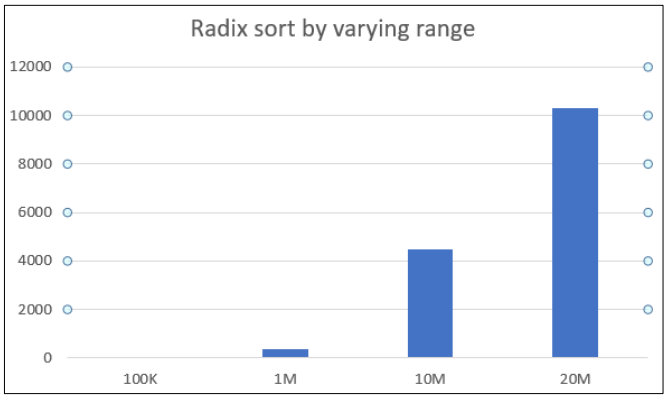
Y-axis in all the graphs shown hereafter represent algorithm runtimes in milliseconds.

Radix Sort

Radix sort performance is checked by:

Changing the number of digits in input, while keeping base constant:

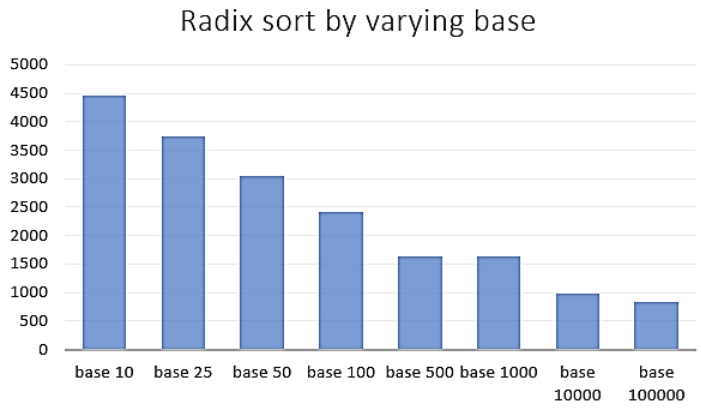
Radix Sort			
Base	**Digits**	**Range**	**Time in millisecs**
10	6	100,000	30
	7	1,000,000	373.6
	8	10,000,000	4479.2
	8	20,000,000	10287



The time increases with increase in range of numbers. As the range increases, the number of digits of each element also increases. Since number of digits plays an important part in analysis of radix sort, we can see that increase in number of digits in the numbers increases the time required to sort them, which is evident from ranges of 100K. 1M and 10M.

Changing the base, while keeping range constant:

Radix Sort		
Range	**Base**	**Time in millisecs**
10,000,000	10	4451.4
	25	3743
	50	3043
	100	2405
	500	1625
	1000	1639
	10,000	980
	100,000	840

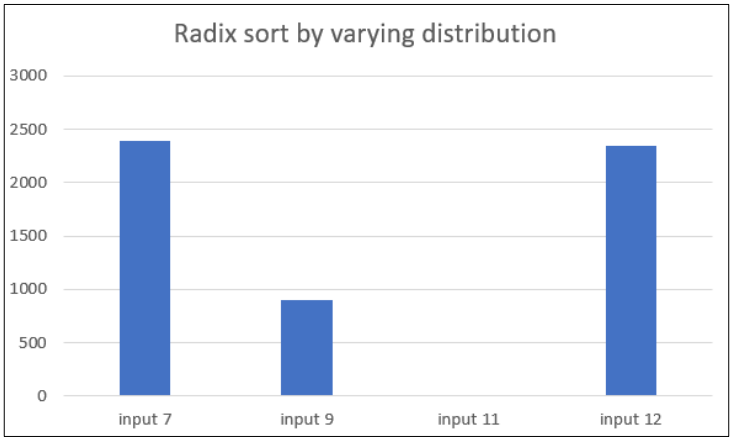


We can see that as the base increases, time required to sort the elements decreases, for a constant range of 10M. This happens because as the split size to sort the number increases, the time required to sort the elements decreases.

Changing the distribution of numbers, while keeping base constant:

Here we consider the following datasets distribution varies, keeping the base as 100:

Base	**Dataset**	**Time in millisecs**
100	Input7	2395.4
	Input9	899.4
	Input11	8.4
	Input12	2345.75



Input7 is the base set with 10M random numbers which is compared with other datasets with different distribution and same number of elements.

While checking the effect of range on radix sort, we notice that radix sort performs better when range lies between 0-1K. This is because of the range of the numbers used, which makes k=3 in as compared to base input7 where k=9, where k is the number of digits in the largest element.

The time required is always less than 10ms when all elements are constant, irrespective of the number of digits.

Radix sort doesn't has any effect when the same numbers are arranged in descending order. This happens because when elements are reversed, the number of digits in highest element are still 9 which is same as case of base input7.

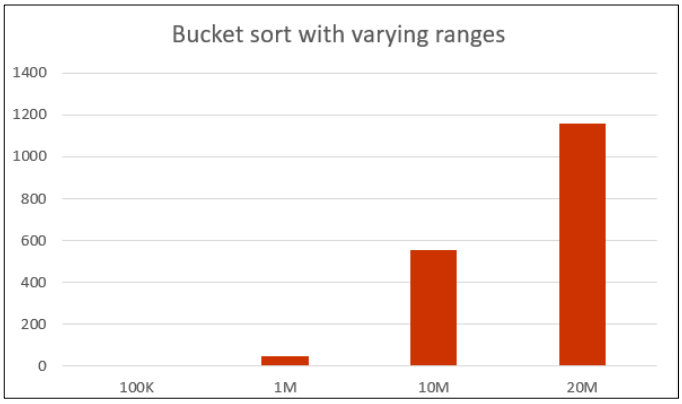
The complexity of radix sort is O(nk) where n is the number of elements and k is the largest number of digits. Time complexity increases with increase in number of elements along with number of digits. This is not preferable when using arrays with higher ranges. The running time will increase 10 times with each increase in digit, which is not preferable. Therefore, algorithms like quicksort are preferred over radix sort for higher ranges, because k is replaced by log(n) in case of quick sort. Thus radix sort can be really quick for smaller values of k.[7]

Bucket Sort

Bucket sort analysis is done by:

Changing the range, keeping the number of buckets constant:

Bucket Sort		
Number of buckets	**Range**	**Time in millisecs**
10	100,000	3
	1,000,000	46.8
	10,000,000	552.6
	20,000,000	1157.5

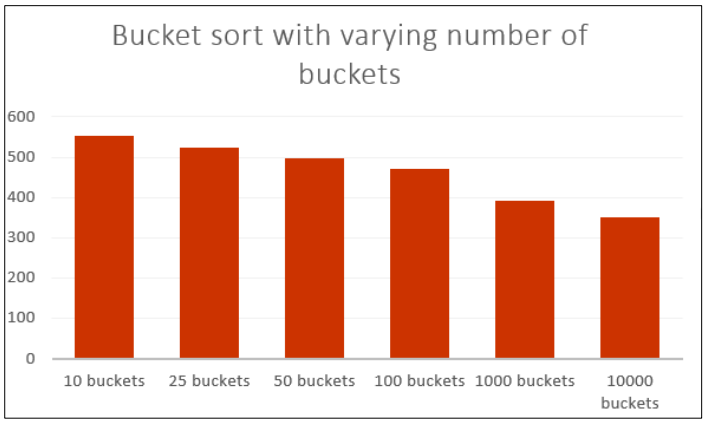


It is evident from the graph that performance is inversely proportional to range for bucket sort.

Changing the number of buckets, keeping the range constant:

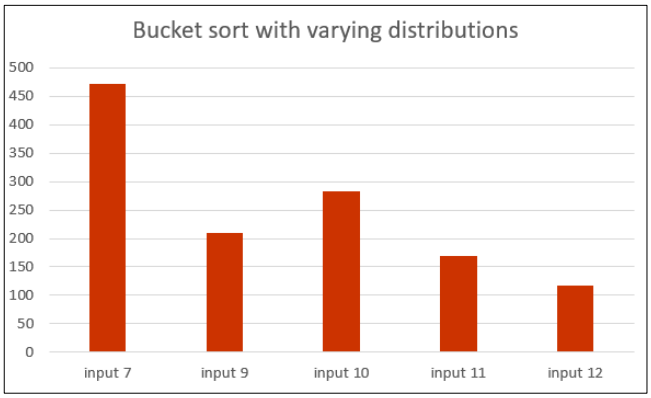
When the number of buckets is increased, time taken to sort algorithms is decreased. This is because data is sorted by dividing it into more number of buckets. Fewer number of buckets degrades performance of bucket sort.

Bucket Sort		
Range	**No. of buckets**	**Time in millisecs**
10,000,000	10	551.8
	25	524
	50	497.2
	100	470.2
	1000	392
	10000	351.8



Changing the distribution, while keeping the number of buckets constant:

No. of buckets	**Dataset**	**Time in millisecs.**
100	Input7	471
	Input9	209.4
	Input10	282.2
	Input11	168.6
	Input12	117



Here, we can see that bucket sort performs better when the range of elements is reduced to 1K and 10K for 1M elements. It performs even better when the element in the array is constant number.

Bucket sort performs best the input is sorted in descending order. In order for bucket sort to perform better, array elements must be uniformly sorted. Since the uniformity is highest in reverse sorted numbers, the efficiency of bucket sort is highest

Bucket sort is mainly useful when input is uniformly distributed over a range. When the input contains clusters, i.e. closed to each other, those elements are likely to be placed in the same bucket, which results in some buckets containing more elements than average.

The worst-case scenario occurs when all the elements are placed in a single bucket. The overall performance would then be dominated by the algorithm used to sort each bucket making bucket sort less optimal. [6]

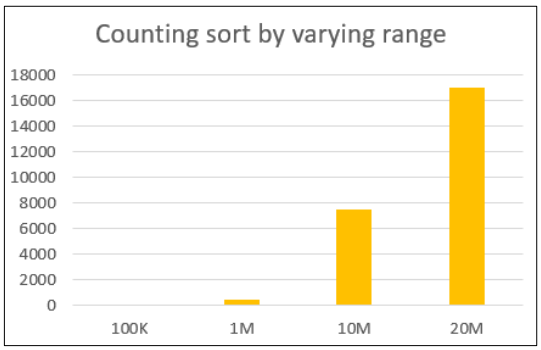
The distribution of elements into buckets is the main advantage of bucket sort, where each bucket is sorted independently. If this distribution is not done correctly, we may end up in doing huge amount of extra work with minimal benefit. As a result, bucket sort works best on uniformly distributed data and worst when data is clustered.

Counting Sort

Counting sort analysis is done by:

Changing the range:

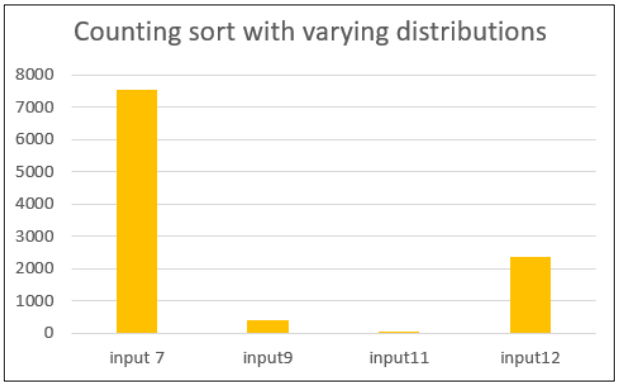
Counting Sort	
Range	**Time in millisecs**
100,000	16
1,000,000	415.4
10,000,000	7524
20,000,000	17031



The time required to sort the elements is directly proportional to the range.

Changing the distribution:

Dataset	Time in millisecs
Input7	7524
Input9	391.2
Input11	21.4
Input12	2370.2



Here we observe that range has a significant effect on counting sort also. When range is changed from 0-1M to 0-1K, time required decreases drastically from 7524 microseconds to 391.2 microseconds. This is because of increase in the repetition of numbers. Because counting sort works on the frequency of numbers, repeated numbers require less time to sort, than non-repeated numbers. This is evident from input11 also which is a single element array. Input12, which is descending array of elements also shows a better performance than input7. This proves that an already sorted array takes lesser time in counting sort than a non-sorted array.

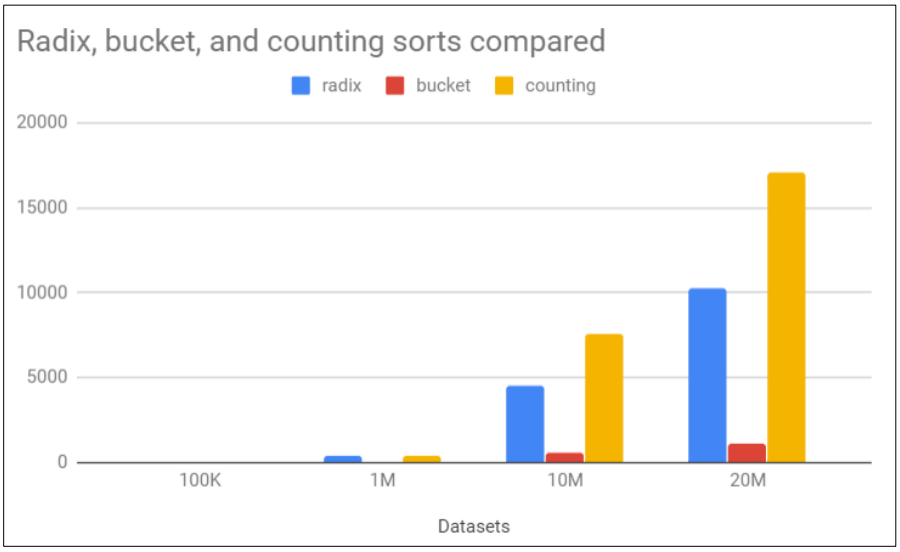
The complexity of counting sort depends on (n+k) where n is number of elements and k is range, which is difference between minimum and maximum element in array. Therefore, although this algorithm seemingly works in linear time, it is not always preferable to use, in cases when the maximum element is array is exponentially higher than actual number of elements. [8]

E.g., In an array of 10 elements, if the highest number is 100, this makes complexity $O(n+n^2)$, which essentially reduces to $O(n^2)$. Thus, the complexity deteriorates and it can get even worse.

Therefore, counting sort works in linear time if and only if k ≈ n. Based on the analysis, we can say that counting sort works well for smaller ranges. As the range increases, sorting gets slower.

Analysis of all three sorting algorithms with respect to datasets:

Algorithm	**100K**	**1M**	**10M**	**20M**
Radix	30	373.6	4479.2	10287
Bucket	3	46.8	552.6	1157.5
Counting	16	415.4	7524	17031.5



This comparison is done by keeping 10 as number of buckets and 10 as base for radix. It is evident from the comparison that bucket sort has the best performance out of all the three algorithms. Bucket sort performed better because of uniformity of data, however, increase in range and increase in number of digits affected the performances of counting sort and radix sort, respectively. Thus, bucket sort is useful when the dataset is huge and has uniform distribution, while counting and radix sorts are useful for datasets with smaller ranges.

References:

1. CH 8 in Introduction to Algorithms (Third Edition) by Thomas H.Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein
2. https://stackoverflow.com/
3. https://www.quora.com
4. https://www.geeksforgeeks.org
5. https://www.growingwiththeweb.com
6. https://en.wikipedia.org/wiki/Bucket_sort
7. https://en.wikipedia.org/wiki/Radix_sort
8. https://en.wikipedia.org/wiki/Counting_sort