City University of New York (CUNY)
CUNY Academic Works

Publications and Research | New York City College of Technology

2022

An Analysis of Non-Comparison Based Sorting Algorithms

Jacob M. Gomez
CUNY New York City College of Technology

Edgar Aponte
CUNY New York City College of Technology

Brad Isaacson
CUNY New York City College of Technology

How does access to this work benefit you? Let us know!

More information about this work at: https://academicworks.cuny.edu/ny_pubs/891
Discover additional works at: https://academicworks.cuny.edu

This work is made publicly available by the City University of New York (CUNY).
Contact: AcademicWorks@cuny.edu

“An Analysis of Non-Comparison Based Sorting Algorithms”

By: Jacob Gomez , Edgar Aponte, Prof. Brad Isaacson

Abstract

Sorting algorithms put elements of a list into an order (e.g., numerical, alphabetical). Sorting is an important problem because a nontrivial percentage of all computing resources are devoted to sorting all kinds of lists. For our project, we implemented two non-comparison based sorting algorithms from pseudocode and compared them to various comparison based sorting algorithms. The two algorithms have their advantages and disadvantages as well as their unique features. We found that Radix Sort (which is a modified version of Counting Sort) was the most efficient of them all.

Introduction

Many sorting algorithms have been invented for sorting. They are either comparison based or non-comparison based. Their relative advantages and disadvantages have been studied extensively.

A comparison based sorting algorithm runs the elements through a single abstract comparison operation (usually a "less than or equal to" operator) that determines which of two elements should occur first in the final sorted list. Such examples include Bubble Sort, Insertion Sort, Mergesort, Heapsort, Timsort, and Intro Sort.

Non-comparison based sorting algorithms, on the other hand, are sorting algorithms which are not comparison based. Examples include Counting Sort and Radix Sort.

In this poster, we chose to discuss the non-comparison based sorting algorithms Counting Sort and Radix Sort. These are straightforward algorithms in which all Computer Science and Applied Mathematics majors should be able to learn. Counting Sort is a linear sorting algorithm that sorts in $O(n+k)$ time when elements are in the range from 1 to k. Radix Sort uses Counting Sort as a subroutine to sort.

For our project, we created an application which included these two sorting algorithms. We analyzed the running times of these sorting algorithms with various sets of unsorted data. Through this process, we gained a better understanding of these two sorting algorithms and their worst-case time complexities. In conclusion, Radix Sort was the fastest and most efficient sorting algorithm we tested.

Process

The application we used to compile the algorithms was JDoodle. This is an online site where you can run and write out different languages of code like Java, C++, Python and many more. The only limitation on this site is the amount of memory you’re given, so printing large arrays causes run-time errors. Nonetheless, we were able to calculate the running time for the sorting algorithms we implemented. Although we tested various inputs, we preferred to use randomly generated lists of 10,000 integers from 0 to 9, 10,000 integers from 0-999,999, and 50,000 integers from 0 to 999,999. The pseudocodes for the 2 non-comparison based sorting algorithms we implemented, Counting Sort and Radix Sort, can be found on the right side of this poster.

Discussion

Based on the results, we found that Radix Sort was the fastest and most efficient sorting algorithm. We found that Counting Sort was one of the slowest and least efficient for lists containing a large range of values but when dealing with integers 0 to 9, Counting Sort was 2x faster than radix sort. Since Radix Sort was the fastest algorithm we tested in all of our experiments, our findings are consistent with Radix Sort being used as the non-comparison based sorting algorithm industry standard.

Counting Sort

```
function CountingSort(input, k)

    count ← array of k + 1 zeros
    output ← array of same length as input

    for i = 0 to length(input) - 1 do
        j = key(input[i])
        count[j] += 1

    for i = 1 to k do
        count[i] += count[i - 1]

    for i = length(input) - 1 downto 0 do
        j = key(input[i])
        count[j] -= 1
        output[count[j]] = input[i]

    return output
```

- Counting Sort is a non-comparison based algorithm that collects objects according to keys (like hashing). The way it operates is by determining the position for each key value in the output sequence by counting the number of objects with distinct key values. When using counting sort, the input of the soon-to-be sorted array is a simple sequence of integers itself. It's efficient if the range of input data is not significantly greater than the number of objects to be sorted. It's very fast when dealing with arrays containing a small range of values and has a worst-case running time that is linear to the input size.

Radix Sort Using Counting Sort

```
Radix-Sort(A, d)
//It works same as counting sort for d number of passes.
//Each key in A[1..n] is a d-digit integer.
//(Digits are numbered 1 to d from right to left.)
    for j = 1 to d do
        //A[]-- Initial Array to Sort
        int count[10] = {0};
        //Store the count of "keys" in count[]
        //key- it is number at digit place j
        for i = 0 to n do
         count[key of(A[i]) in pass j]++
        for k = 1 to 10 do
         count[k] = count[k] + count[k-1]
        //Build the resulting array by checking
        //new position of A[i] from count[k]
        for i = n-1 downto 0 do
         result[ count[key of(A[i])] ] = A[j]
         count[key of(A[i])]--
        //Now main array A[] contains sorted numbers
        //according to current digit place
        for i=0 to n do
          A[i] = result[i]
    end for(j)
end func
```

- The idea of Radix Sort is to do a digit-by-digit sort starting from least significant digit to most significant digit. Radix Sort uses Counting Sort as a subroutine to sort (although not apparent from the pseudocode provided). This algorithm takes advantage of the fact that any number in the decimal system can be represented by digits from 0 to 9. So in this algorithm, we perform a Counting Sort with 10 keys in every pass for a total of $ceil(log_{10}(max\_element))$ number of passes. For example, if we were sorting a list whose elements are in the range of 0 through 999,999, Radix Sort would be executing a Counting Sort with 10 keys five times: once for the ones digit, tens digit, hundreds digit, thousands digit, and ten-thousands digit. This algorithm employs the linked list data structure which is a linear collection of data elements not given by their physical placement in memory. The issue with linked lists is that it requires a lot of memory allocation and deallocation which slows the algorithm down. This algorithm is slower than comparison based algorithms for smaller sized lists due to the aforementioned overhead, but superior for larger sized lists. The worst-case running time of Radix Sort is linear to the input size of the list.

Results

We tested 4 sorting algorithms:

(i) Comparison based:

(1) Timsort, (2) Intro Sort,

(ii) Non-Comparison based:

(3) Counting Sort, (4) Radix Sort using Counting Sort.

Below are the running times of the 4 sorting algorithms using randomly generated unsorted lists of (1) 10,000 integers whose values are between 0 and 9, (2) 10,000 integers whose values are between 0 and 999,999, and (3) 50,000 integers whose values are between 0 and 999,999. Each box corresponds to the same input.

10,000 Elements (Integers 0-9)

```
Time Shown 0.04924464225769043 seconds (Tim sort)
Time Shown 0.03343319892883301 seconds (Intro sort)
Time Shown 0.006261348724365234 seconds (Radix sort)
Time Shown 0.0034852027893066406 seconds (Counting sort)
```

```
Time Shown 0.04634857177734375 seconds (Tim sort)
Time Shown 0.033632755279541016 seconds (Intro sort)
Time Shown 0.006282806396484375 seconds (Radix sort)
Time Shown 0.0035185813903808594 seconds (Counting sort)
```

10,000 Elements (Integers 0-999,999)

```
Time Shown 0.19681119918823242 seconds (Counting sort)
Time Shown 0.051963090896606445 seconds (Tim sort)
Time Shown 0.042730093002319336 seconds (Intro sort)
Time Shown 0.03761935234069824 seconds (Radix sort)
```

```
Time Shown 0.20296931266784668 seconds (Counting sort)
Time Shown 0.05223226547241211 seconds (Tim sort)
Time Shown 0.0399172306060791 seconds (Intro sort)
Time Shown 0.03919267654418945 seconds (Radix sort)
```

50,000 Elements (Integers 0 – 999,999)

```
Time Shown 0.34042906761169434 seconds (Tim sort)
Time Shown 0.26348209381103516 seconds (Counting sort)
Time Shown 0.24551963806152344 seconds (Intro sort)
Time Shown 0.22077488899230957 seconds (Radix sort)
```

```
Time Shown 0.40522193908691406 seconds (Tim sort)
Time Shown 0.2821061611175537 seconds (Counting sort)
Time Shown 0.2418227195739746 seconds (Intro sort)
Time Shown 0.23436379432678223 seconds (Radix sort)
```