Counting Sort

A Presentation and Analysis

12/4/2017

Author: Patrick Ryan

CS 566 - Analysis of Algorithms

Boston University - Metropolitan College

Table of Contents:

Introduction:

Counting sort (or count sort) is an algorithm for sorting an array of elements according to their numerical values. It operates by counting the number of distinct occurrences of each element, and then uses non-comparison arithmetic to determine the correct position of each element in the output sequence.

Counting sort works by instantiating a helper array of size (k+1). The helper array is then populated using the numerical values an index. This effectively creates a histogram within the helper array, where the value of each array element contains the value of each number that will ultimately be output in the final sorted array. This histogram array is then used to repopulate the final array by looping through each element and repopulating its index (numerical value) onto the final array a number of times equal to its value.[1]

Counting sort can be exceptionally fast because it sorts an array's values using indexing, instead of comparison operations. This means that slightly more memory is required for the extra array at the cost of running time. Because counting sort uses key values as indexes into an array, it is not a comparison sort, and the Ω (n log n) lower bound for comparison sorting does not apply.[1]

Historical Background:

Counting sort is believed to have been pioneered in 1954 by the computer scientist, Harold H, Seward during his residency at the MIT Instrumentation Lab. Seward initially developed counting sort as part of a subroutine in radix sort, an application where itis still commonly used today.

After developing these two algorithms, Seward went on to work in the MIT Servomechanisms Laboratory for the United States Navy, where he aided in the development of the Whirlwind 1 vacuum tube computer, one of the first digital computers to calculate in parallel (rather than serial). The whirlwind system became a key computational system used in powering guidance systems onboard the Apollo spacecraft and the Polaris missile.

General Algorithm:

Although counting sort can be designed in a variety of ways, it is commonly implemented in the following five steps:

i) Take an unsorted array of length (n), and identify the largest numbered contained within it (k).
ii) Instantiate a helper array of length (k+1), populated with zeroes.
iii) Loop through each element in the input array, and increment the helper array (+1) for each occurrence of each number.
iv) Use the helper array to repopulate the original input array (in place).
v) Return the now-sorted array.

A summary walkthrough of counting sort performing these steps on the array: [1, 2, 3, 3, 4, 5] is provided below in [Figure 1]. For additional reference, a complete visual walkthrough of counting sort being performed on the same array can be found in [Figure 13].

[Figure 1: Visual Summary of the Counting Sort Algorithm]

Step	Notes	Arrays
1	Start with an unsorted array of length (n), and identify the largest numbered contained within it (k). If the maximum number is unknown, a simple loop comparing each number to a running maximum should be implemented.	Input Array: 5 | 4 | 3 | 3 | 2 | 1
2	Instantiate a helper array of length (k+1), populated with zeroes. Because the largest number in the original array is 5, we must have a helper array of length 6. Each space in the array represents a different number, (e.g. the 0^{th} (1^{st}) space holds the number of 0's in the input array, the 1^{st} (2^{nd}) holds the number of 1's, etc.)	Input Array: 5 | 4 | 3 | 3 | 2 | 1 Helper Array: 0 | 0 | 0 | 0 | 0 | 0
3	Loop through each element in the input array, and increment the helper array (+1) for each occurrence of each number. (Note that the 4^{th} slot has been incremented twice due to the duplicate pair of 3's present in the initial input array.)	Input Array: 5 | 4 | 3 | 3 | 2 | 1 Helper Array: 0 | 1 | 1 | 2 | 1 | 1
4	Using the helper array, repopulate the original input array (in place). (All information on the count of each number is now present in the helper ("histogram") array, so no relevant information is lost in this copying step)	Input Array: 1 | 2 | 3 | 3 | 4 | 5 Helper Array: 0 | 1 | 1 | 2 | 1 | 1
5	Return the now-sorted array.	Output Array: 1 | 2 | 3 | 3 | 4 | 5

Implementation:

Counting sort can be implemented in a variety of ways, but generally speaking, the algorithm loops over the input array and compiles a histogram, which represents the count of each unique element. Counting sort then performs a prefix sum computation (a second loop, over the range of possible keys) to determine, for each key, the starting position in the output array of the items having that key. Finally, it loops over the helper array again, copying each element (once for each occurrence) into the correct position on the output array.

A general implementation of counting sort in Python 3.6 is shown below in [Figure 2].

[Figure 2: Counting Sort Algorithm Implemented in Python 3.6]

```
# Inputs are 1) the array to be sorted and 2) the maximum value in the array.

def sort(A, max_val):

# Create a counting array, 1 greater in length than the max value

    counting_array = [0] * (max_val + 1)
    index = 0

# For each value in the input array, increment the slot in the counting array

    for i in range(0, len(A)):
        counting_array[A[i]] += 1

# Iterate through the input array, and re-populate using the helping array

    for j in range(0, len(counting_array)):
        while (counting_array[j] > 0):
            A[index] = j
            index += 1
            counting_array[j] -= 1

# Finally, return the now-sorted array

    return A
```

Note: Text preceded with a hashtag denotes non-executed script

Design Variations:

As with most algorithms, counting sort can designed and implemented in a variety of ways, while still accomplishing the same sorting objectives. Some possible design variations for counting sort are:

i) Max value: Instead of requiring the user to include the max value (k) as a parameter, a simple loop and comparison can be used to determine the maximum value in the input array. Implementing such a loop will add another instruction of Θ(n) complexity to the total runtime.

ii) New Array: Instead of populating the original array in-place with information from the helper array, a new array can be instantiated and populated, however doing so will increase overall space complexity, for the total algorithm.

iii) Direction: The original input array can be populated in either ascending or descending order (i.e. reversed order). To achieve reverse order sorting, the primary (j) for-loop must be reversed, going from k:0. This variation will have no impact on the algorithm's complexity. [4]

iv) Lower limit: Although an upper limit must always be identified (by way of the maximum value), the lower limit can be restricted to further shorten the algorithm's computing time. By default the range (represented by the helper array) is [0 : max_val].

v) Populating Mechanism: Different indexing and population mechanisms can be implemented for populating the final output array (whether it is the original array or newly created output array.)

vi) Number Sets: Counting sort is often implemented to sort integers, (1, 3, 250, etc.) however it can also be applied to rational numbers (1.25, 3.333, 250.122, etc.) as long as the maximum number of post-decimal significant figures in the array is identified prior to the first population step. (i.e. if sorting the array [7.20, 3.33, 2.30], the "buckets" in the helper array would need to account for all rational numbers from 0 to the maximum value with two trailing post-decimal digits.

Time Complexity:

Counting sort is one of the few known algorithms that is able to sort an array in a near-Θ(n) time complexity. The time complexities for counting sort are shown below, with (n) representing the size of the input array, and (k) representing the maximum number in the input array.

[Figure 3: Time and Space Complexity of Counting Sort]

Time Complexity			Space Complexity
Worst case	Best case	Average case	Worst case
$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(k)$ auxiliary

As shown in [Figure 3] the time complexity for the worst, best and average case of counting sort is O(n+k), however when k = O(n), sorting runs in Θ(n) time.[2]

Sorting Stability

Counting sort is a stable sorting algorithm. Although the information regarding order is destroyed when mapping to the "histogram" helper array, counting sort does in fact preserve the relative order of equal elements. (I.e. if two elements a[i] and a[j] have the same value, and i< j then a[i] will appear before a[j].) Sort stability is an important for counting sort to maintain, especially when used as a subroutine in radix sort.

In the context of radix sort, f x<y are two elements of a, and the most significant bit at which x differs from y has index r, then x will be placed before y during pass [r/d] and subsequent passes will not change the relative order of x and y. As it runs, radix-sort will perform w/d passes of counting-sort. Each pass requires $O(n+2^d)$ time. [3]

Ideal Usage Scenarios:

Counting sort is an ideal choice for sorting scenarios where:

i) The list is made up of integers or can be mapped to integers.
ii) The range and max element is known of elements is known.
iii) All or most of the elements in the range are present. [1]
iv) There are no constraints or issues related to additional memory usage.

If the list is known to be nearly sorted then another option that takes advantage of this, such as insertion sort may result in a quicker time complexity. [1]

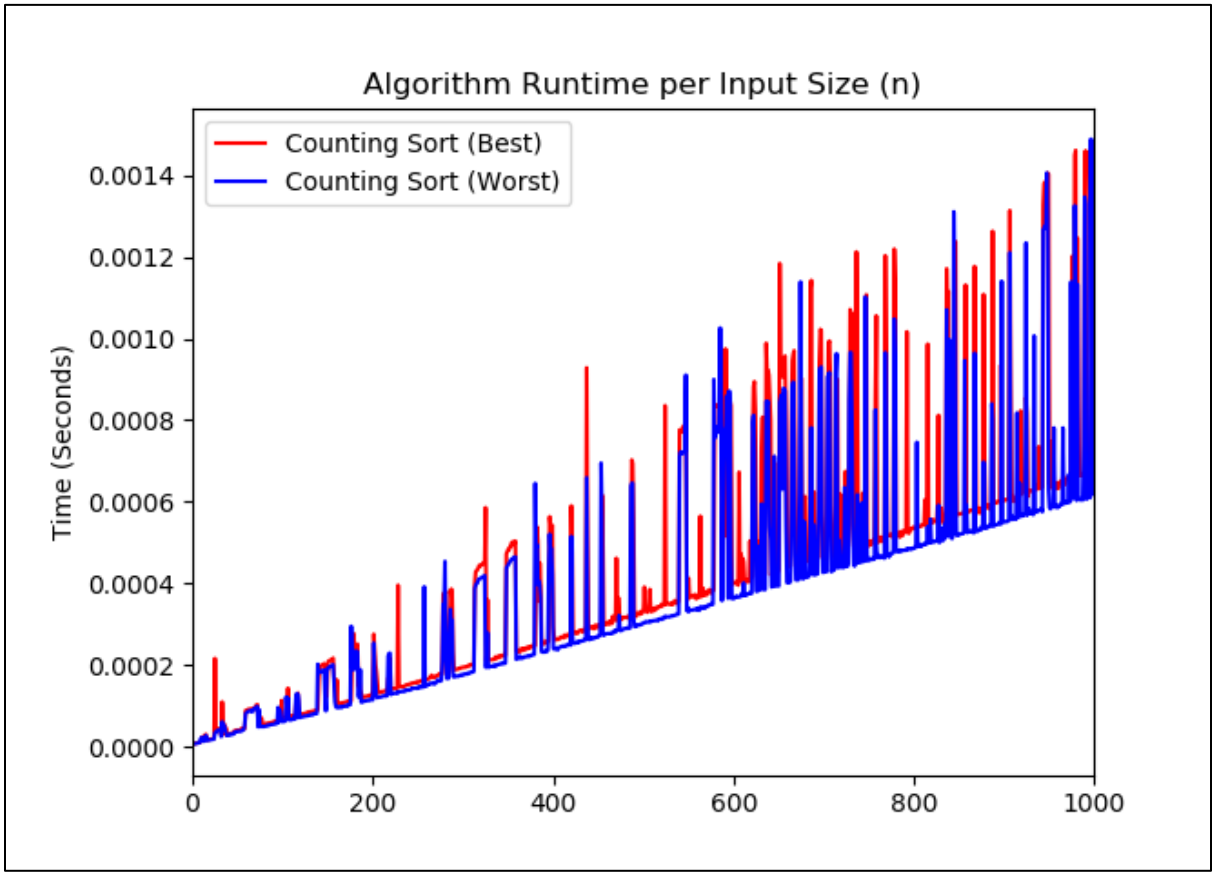
In the most general case, the input to counting sort consists of a collection of (n) items, each of which has a non-negative integer key whose maximum value is at most (k). In some descriptions of counting sort, the input to be sorted is assumed to be more simply a sequence of integers itself, but this simplification does not accommodate many applications of counting sort. For instance, when used as a subroutine in radix sort, the keys for each call to counting sort are individual digits of larger item keys; it would not suffice to return only a sorted list of the key digits, separated from the items.[1]

Runtime Comparison A: (Counting Sort Best and Worst Case):

A common way of verifying the conclusion of an asymptotic analysis is through standalone or comparative runtime trials. To gauge the rough time complexity of a counting sort algorithm one can iterate through a range of input sizes. [Figure 4] and [Figure 5] show a timeit trial implemented in Python 3.6. The parameters of the algorithms and inputs analyzed in this trial are as follows:

- A max value parameter is not supplied as input and is identified using a comparison loop.
- The input array is populated directly back in-place from the helper array.
- The input array is re-populated in ascending order, before being returned.
- Best case is an ascending ordered list (i.e. [1, 2, 3, 4...1000]) with no duplicated elements.
- Worst case is a descending ordered list (i.e. [1000, 999, 998…1]) with no duplicate elements.
- Follow-up visualization has been produced using the Seaborn module in Python 3.6.

[Figure 4: Algorithm Runtime: Counting Sort Best and Worst Case Scenarios (n) = 1000, ylim = 0.0014]

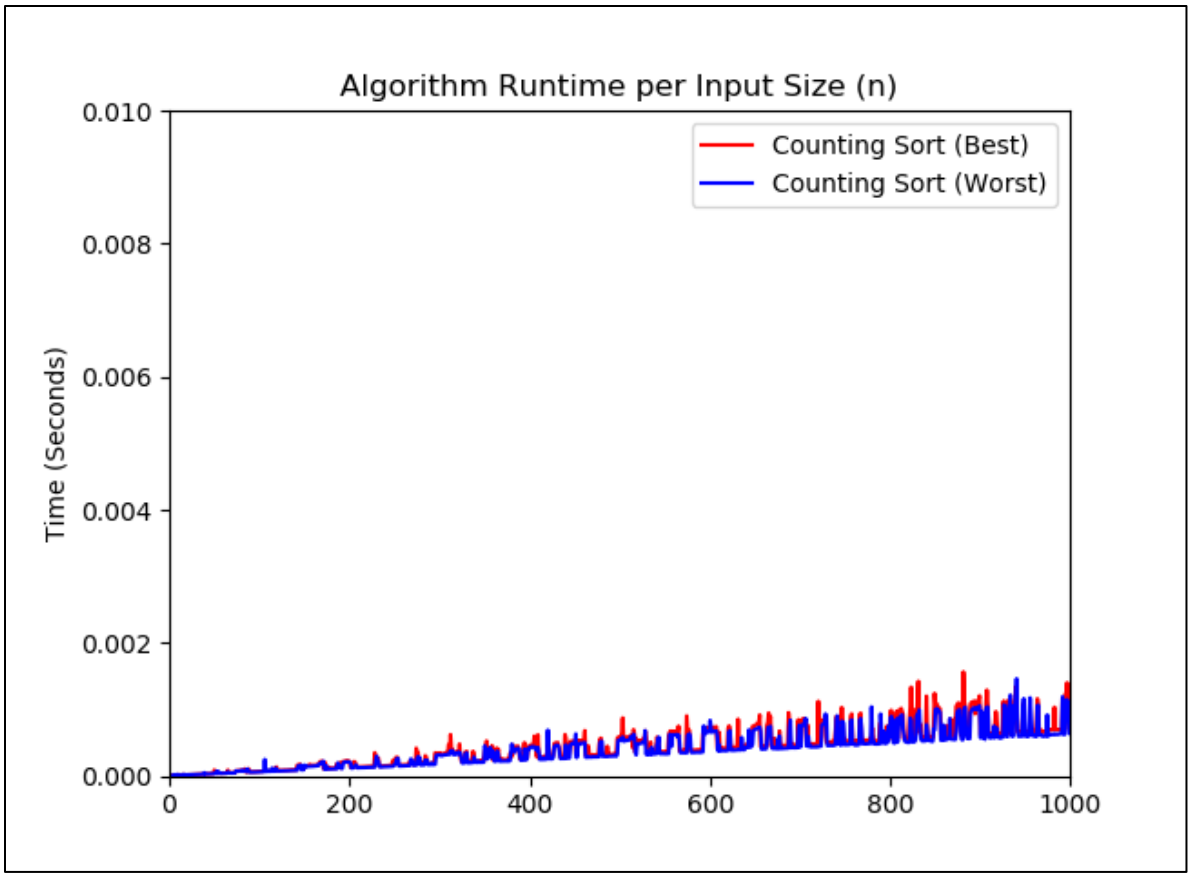


As expected, the best and worst case scenarios execute with a near identical runtime relative to the input array size.

With such a short processing time per (n), significant background processing noise is observed through Python's built-in timeit function. However, as the runtime asymptotically grows to longer runtimes (e.g. greater than 0.01 seconds), the processing noise will become relatively negligible. A separate runtime analysis with a larger y axis limit is shown in [Figure 5].

A quick look at the first 1,000 iterations clearly show that the lower bound of the algorithm run time are very similar. Oddly, the worst scenario run time consistently runs marginally quicker than the best time scenario, whereas one should expect the opposite given the additional reverse array operation performed (see [Figure 8]), however this discrepancy is most likely caused by a technical feature within the python compiler or the timeit module.

[Figure 5: Algorithm Runtime: Counting Sort Best and Worst Case Scenarios (n) = 1000, ylim = 0.01]



As expected, expanding the y axis to include larger runtime lengths, we see that difference in the time complexity for each case is fairly trivial. As with before, we can see the linear lower bounds on each case, thus suggesting a linear time complexity for the overall algorithm.

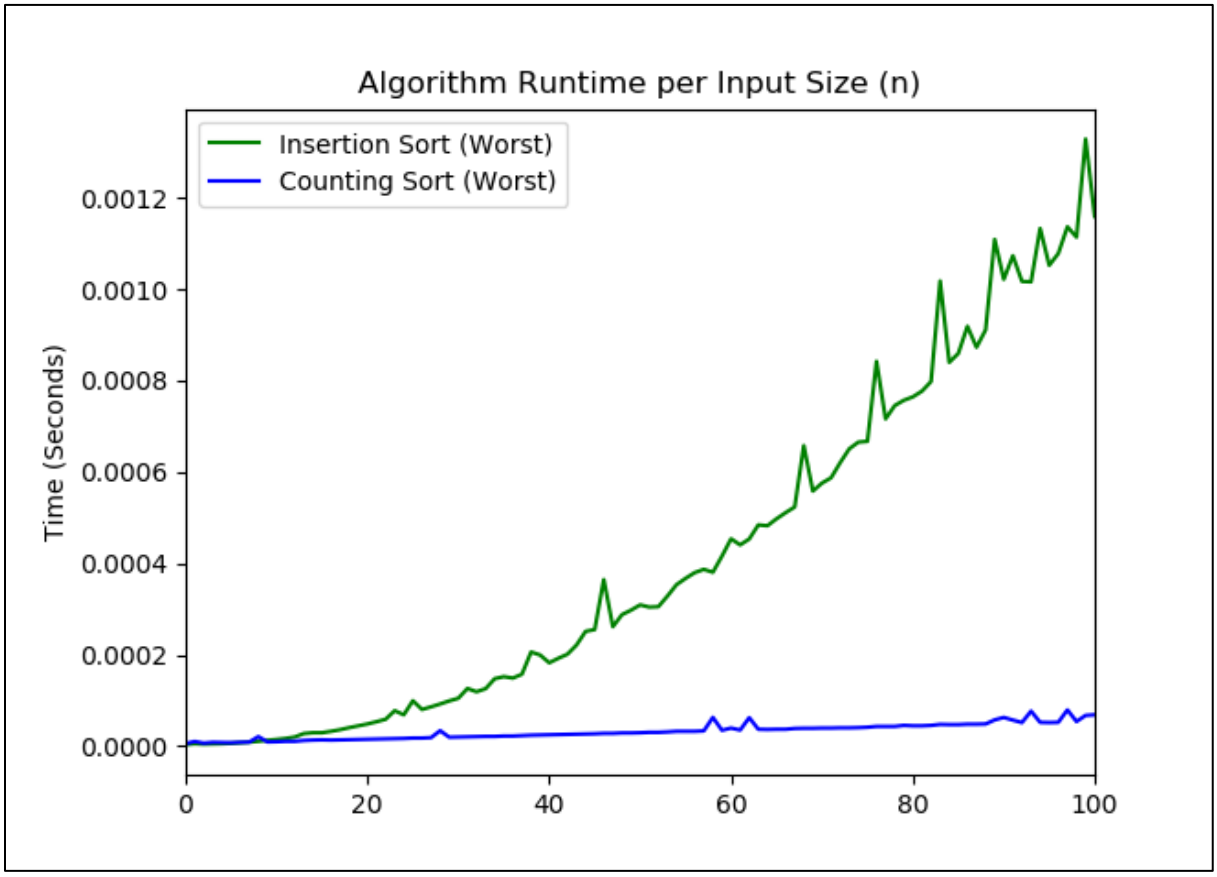
Runtime Comparison B: (Insertion Sort Case and Counting Sort Worst Case):

Comparing the best and worst case scenarios of a single algorithm is insightful (especially when assessing the runtime impact of different implementations), but this type of analysis provides little information on how this algorithm will perform with respect to other algorithms. To provide a more asymptotically focused analysis, we will compare counting sort with insertion sort, a common sorting algorithm features a worst case time complexity of $\Theta(n^2)$.

The parameters of the algorithms and inputs analyzed in this trial are as follows:

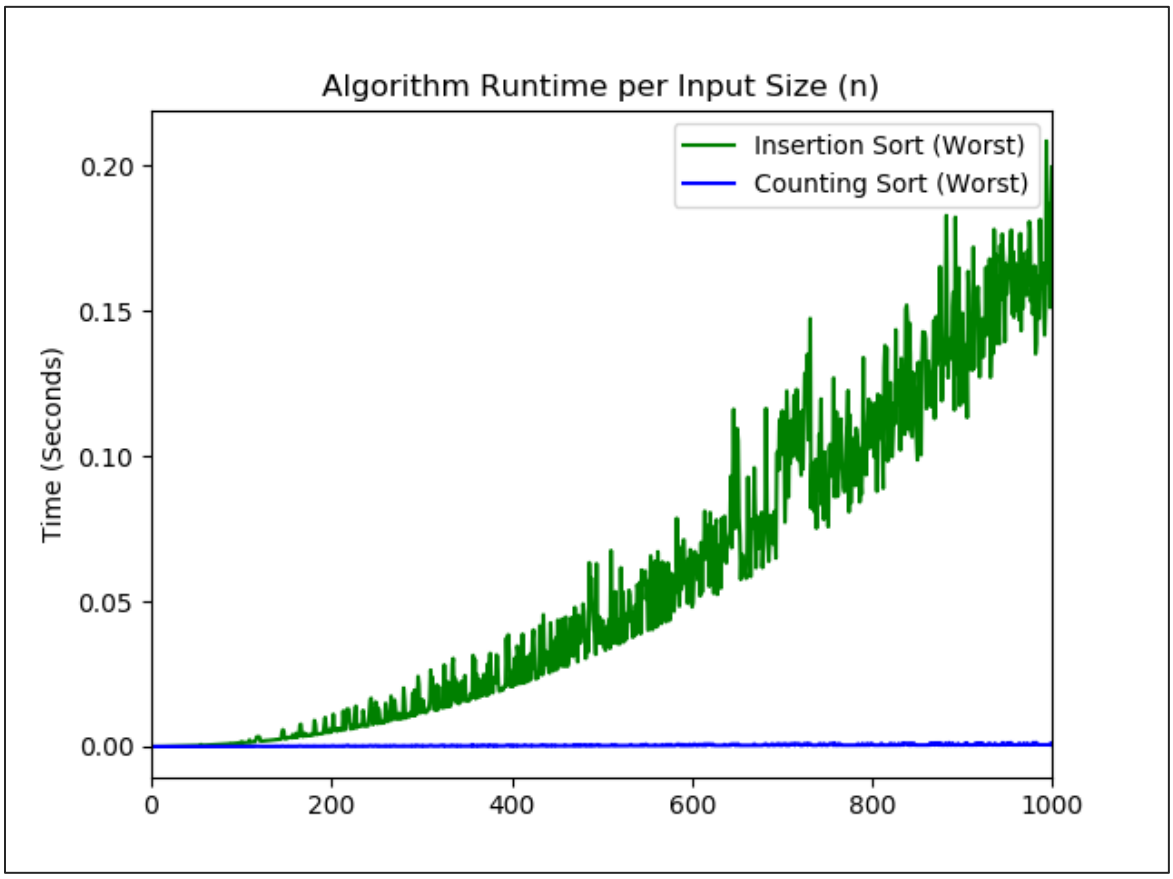
- Max value parameter is not supplied as input and is identified using a comparison loop.
- The input array is populated directly back in-place from the helper array.
- The input array is re-populated in ascending order, before being returned.
- Worst case scenarios for both counting sort and insertion sort are input as identical descending ordered lists (i.e. [1000, 999, 998, 997…1]) with no duplicate elements.
- Follow-up visualization has been produced using the Seaborn module in Python 3.6.

[Figure 6: Algorithm Runtime Comparison of Counting Sort and Insertion Sort: (n) = 100, ylim = 0.0014]



As expected, a significant difference in each algorithm's growth rate can be observed. The worst case scenario of insertion sort (as indicated by the green line) shows clear exponential growth, whereas the worst case scenario of counting sort (as indicated by the blue line) suggests time-linear growth. [Figure 7] shows the same analysis with an input size of length 1,000.

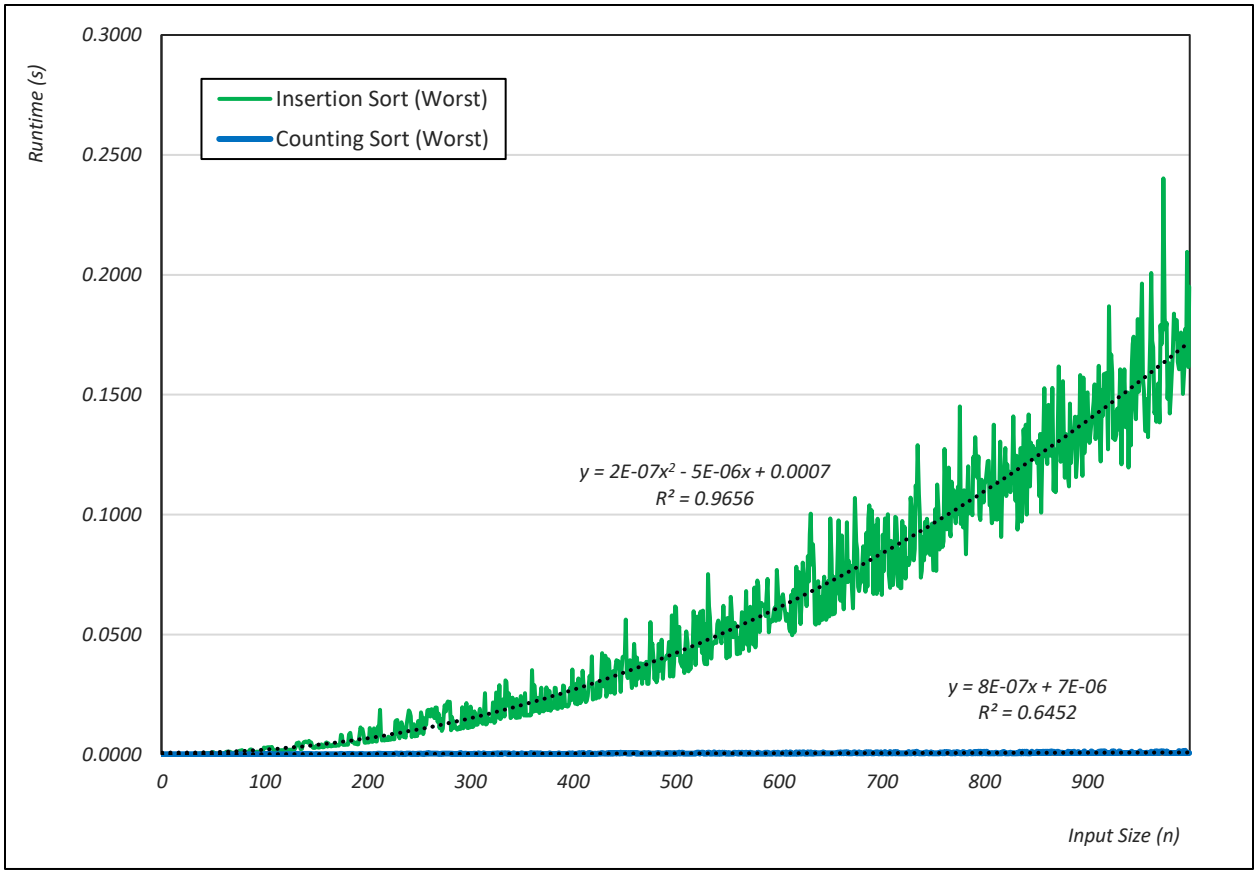
[Figure 7: Algorithm Runtime Comparison of Counting Sort and Insertion Sort: (n) = 1000]



Again, we see the exponential growth rate of insertion sort contrasted against the linear growth rate of counting sort. This comparison provides an excellent example of the importance of conducting asymptotic analysis. As we can see, as (n) grows exponentially the time complexity of insertion sort becomes dominant, while the complexity of counting sort becomes trivial. In fact, the complexity of counting sort almost appears to be constant (Θ(n)) when compared against a second order growth rate.

Lastly, in [Figure 8], we import the data frame in .csv format, and conduct polynomial regression in excel to confirm the time complexity of each algorithm.

[Figure 8: Excel Based Regression Analysis, (Verifying Slope of Insertion Sort and Counting Sort)]

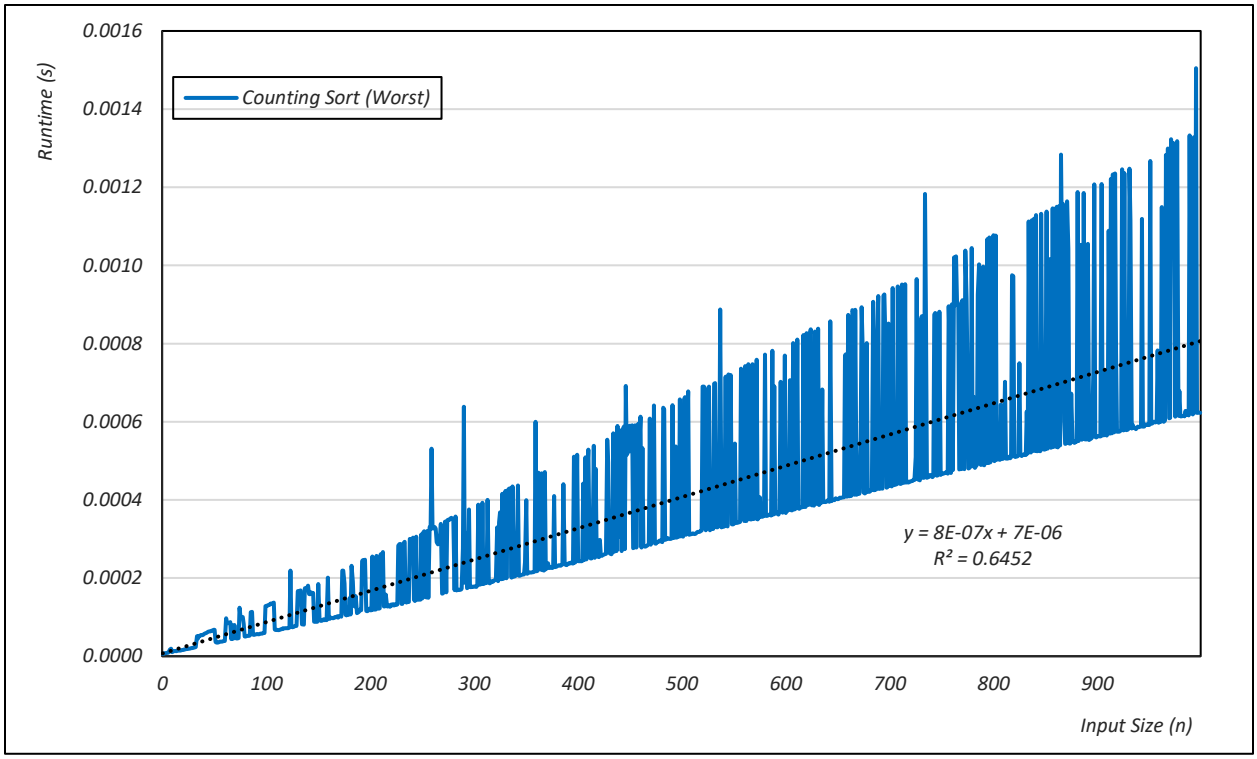


As expected, polynomial regression and fitting confirms the exponential time complexity of insertion sort and the linear time complexity of counting sort.

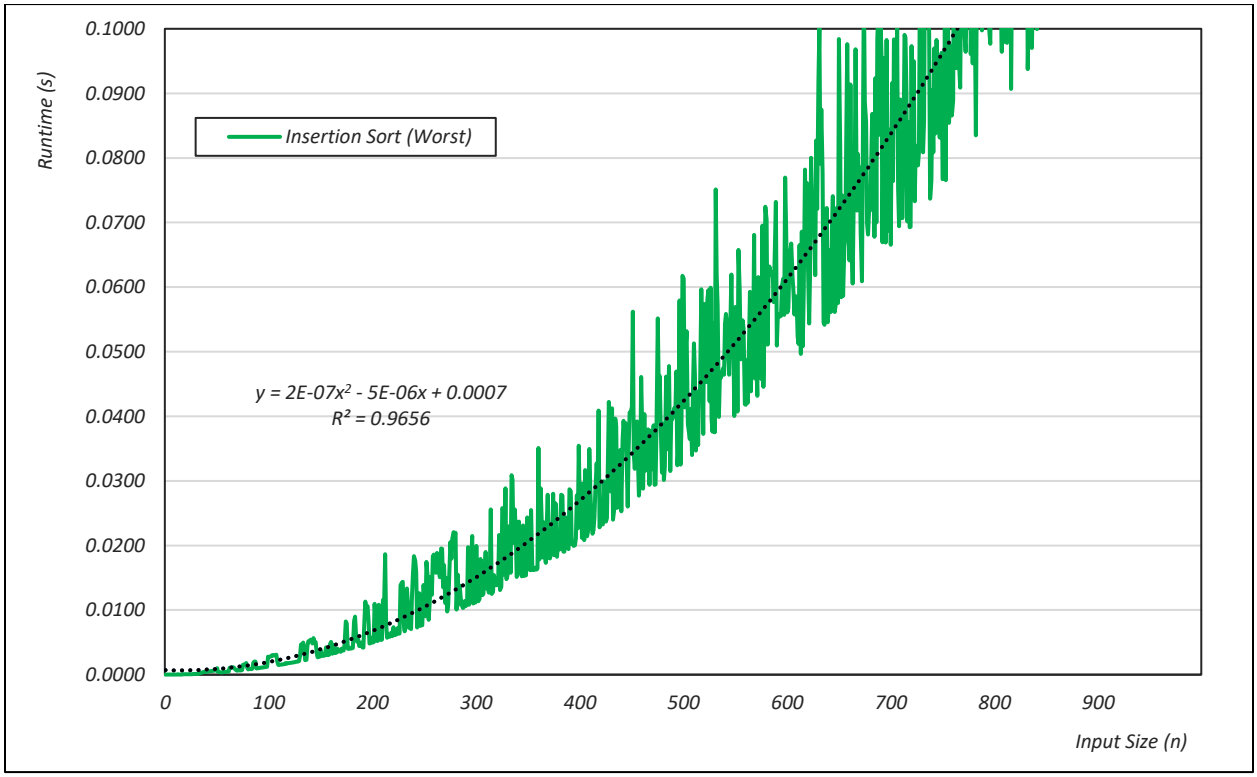
Insertion Sort: $(2E\text{-}07x^2 - 5E\text{-}06x + 0.0007) \in \Theta(n^2)$
Counting Sort: $8E\text{-}07x + 7E\text{-}06 \in \Theta(n)$

The observed R^2 value (coefficient of determination) of the insertion sort (0.9656) regression shows a clear fit between the calculated trend line and the actual series. However the R^2 value of counting sort (0.6452) seems to imply a surprisingly poor fit. Although unexpected, this likely mismatch stems from the high level of background noise, as highlighted in [Figure 4]. (Caused by a relatively high degree of background delays at short runtime lengths). Focusing in on the counting sort regression in [Figure 9], we can see that the regression trend line does in fact, fit within the upper and lower limits generated by the runtime projections. A focused view on insertion sort's time complexity function also confirms the fit implied by its R^2 [Figure 10].

[Figure 9: Excel Based Regression Analysis of Counting Sort]



[Figure 10: Excel Based Regression Analysis of Insertion Sort]



Conclusions and Findings

Counting sort is a clever algorithm which can be exceptionally fast and effective when implemented under the right conditions. Counting sort circumvents the traditional Ω(n log n) lower bound on sorting algorithms because it facilitates sorting through a non-comparison based approach. Instead, counting sort uses the actual values of the elements to index into an array.

Counting sort is a stable sorting algorithm and preserves the order of elements as they are found in the original input array. (A property which allows it to be an effective subroutine in radix sort). Time complexity for the worst, best and average case of counting sort is O(n+k), however when k = O(n), sorting can perform in Θ(n) time.[2]

When compared against higher order growth-rate algorithms, such as insertion sort or merge sort, counting sort can offer a comparatively superior runtime, even if the maximum array value is not explicitly identified beforehand. Counting sort is a great example of how creatively engineered algorithms can produce drastically different time outcomes than conventional alternatives, while still accomplishing the same end-objectives.

Annex A: (Comparison of Counting Sort)(Python 3.6)

[Figure 11: Python Script Comparing the Best and Worst Case Runtime of Counting Sort]

```
import timeit, functools, numpy as np
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns


def counting_sort(A, max_val=None):
    if max_val is None:
        max_val = 0

        for i in range(0, len(A)):
            if A[i] > max_val:
                max_val = A[i]

    counting_array = [0] * (max_val + 1)
    index = 0

    for i in range(0, len(A)):
        counting_array[A[i]] += 1

    for k in range(0, len(counting_array)):
        while (counting_array[k] > 0):
            A[index] = k
            index += 1
            counting_array[k] -= 1

    return A


# Create two empty lists to hold the best and worst runtimes of counting sort
cs_best_time_array = []
cs_worst_time_array=[]

for x in range (1000):   # Outside for-loop dictates the number of iterations in runtime

    print("Current Iteration: " + str(x))

    # List comprehension to generate an ascending array of [1:x]
    k = [n for n in range (x)]

    t=timeit.Timer(functools.partial(counting_sort, k))
    cs_best_time_array.append(t.timeit(1))

    # List comprehension to generate an ascending array of [1:x]
    k = [p for p in range (x)]
    rev_k = k[::-1]   # Reverse list
    t=timeit.Timer(functools.partial(counting_sort, rev_k))
    cs_worst_time_array.append(t.timeit(1))

#  Plot runtime of counting sort's best case and worst case using the seaborn module
ax = (sns.tsplot(data=cs_best_time_array, color ='r', condition="Counting Sort (Best)"))
ax = (sns.tsplot(cs_worst_time_array, color = 'b', value = "Time (Seconds)",
condition="Counting Sort (Worst)"))
ax.set_title('Algorithm Runtime per Input Size (n)')
plt.show()
```

Annex B: (Comparison of Counting Sort and Insertion Sort)(Python 3.6)

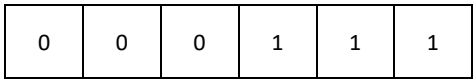
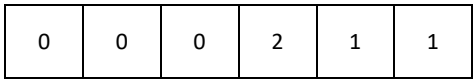
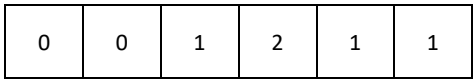
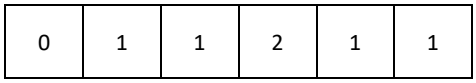
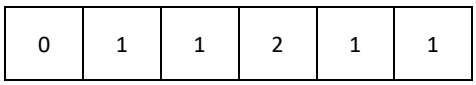
[Figure 12: Python Script Comparing the Runtime of Counting Sort and Insertion Sort Algorithms]

```
import timeit, functools
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns


def counting_sort(A, max_val=None):
    if max_val is None:
        max_val = 0
        for i in range(0, len(A)):
            if A[i] > max_val:
                max_val = A[i]
    counting_array = [0] * (max_val + 1)
    index = 0
    for i in range(0, len(A)):
        counting array[A[i]] += 1
    for k in range(0, len(counting_array)):
        while (counting_array[k] > 0):
            A[index] = k
            index += 1
            counting_array[k] -= 1
    return A

def insertion_sort(A):
    set_length = (len(A))
    for j in range(1, set_length):
        key = A[j]
        i = j - 1
        while i > -1 and A[i] > key:
            A[i+1] = A[i]
            i = i - 1
        A[i+1] = key
    return A

is_worst_time_array=[]
cs_worst_time_array=[]

for x in range (100):

    print("Current Iteration: " + str(x))

    k = [p for p in range (x)]
    rev_k = k[::-1]
    t=timeit.Timer(functools.partial(insertion_sort, rev_k))
    is worst time array.append(t.timeit(1))

    k = [p for p in range (x)]
    rev_k = k[::-1]
    t=timeit.Timer(functools.partial(counting_sort, rev_k))
    cs_worst_time_array.append(t.timeit(1))


ax = (sns.tsplot(data=is_worst_time_array, color ='g', condition="Insertion Sort (Worst)"))
ax = (sns.tsplot(cs_worst_time_array, color = 'b', value = "Time (Seconds)",
condition="Counting Sort (Worst)"))
ax.set_title('Algorithm Runtime per Input Size (n)')
plt.show()
```

Annex C: (Comparison of Counting Sort and Insertion Sort)(Python 3.6)

[Figure 13: Complete Visual Summary of Counting Sort]

Step	Notes	Arrays
1	Take an unsorted array of length (n), and identify the largest numbered contained within it (k). If the maximum number is unknown, a simple loop comparing each number to a running maximum should be implemented.	Input Array: 5 | 4 | 3 | 3 | 2 | 1
2	Instantiate a helper array of length (k+1), populated with zeroes. Because the largest number in the original array is 5, we must have a helper array of length 6. Each space in the array represents a different number, (e.g. the 0^{th} (1^{st}) space holds the number of 1's in the input array, the 1^{st} (2^{nd}) holds the number of 1's, etc.)	Input Array: 5 | 4 | 3 | 3 | 2 | 1 Helper Array: 0 | 0 | 0 | 0 | 0 | 0
3	Begin populating the helper array (similar to a histogram) by incrementing each space (+1) for each occurrence in the input array. First, we change the 6^{th} element in the helper array to 1, because the first element in the input array is 5. (i = 0)	Input Array: 5 | 4 | 3 | 3 | 2 | 1 Helper Array: 0 | 0 | 0 | 0 | 0 | 1
4	Next we change the 5^{th} element in the helper array to 1, because the second element in the input array is 4. (i = 1)	Input Array: 5 | 4 | 3 | 3 | 2 | 1 Helper Array: 0 | 0 | 0 | 0 | 1 | 1

5	Next, we change the 4th element in the helper array to 1, because the third element in the helper array is 3. (i = 2)	Input Array:  Helper Array: 
6	Next, we change the 3rd element in the helper array to 2, because the fourth element in the helper array is 3. (i = 3)	Input Array:  Helper Array: 
7	Next, we change the 2nd element in the helper array to 1, because the fifth element in the helper array is 2. (i = 4)	Input Array:  Helper Array: 
8	Next, we change the 1st element in the helper array to 1, because the fifth element in the helper array is 1. (i = 5)	Input Array:  Helper Array: 
9	Now the helper array is fully formed, and because it contains all pertinent information about the original input array, we can now populate the information from the helper array directly back onto the input array without penalty.	Input Array:  Helper Array: 

10	First look at the 0^{th} element in the helper array. Because its zero, we know that we don't have any zeroes in the original input array, and therefore we end the inner while loop. (index = 0, k = 0)	Input Array: 5 | 4 | 3 | 3 | 2 | 1 Helper Array: 0 | 1 | 1 | 2 | 1 | 1
11	Next, we look at the 1^{st} element in the helper array. Because k is 1, and index is 0, we populate the 0^{th} slot in our input array with a 1, and decrement (-1) the 1^{st} slot in the helper array. (index = 0, k = 1)	Input Array: 1 | 4 | 3 | 3 | 2 | 1 Helper Array: 0 | 0 | 1 | 2 | 1 | 1
12	Next, we look at the 2^{nd} element in the helper array. Because k is 2, and index is 1, we populate the 1^{st} slot in our input array with a 2, and decrement (-1) the 2^{nd} slot in the helper array. (index = 1, k = 2)	Input Array: 1 | 2 | 3 | 3 | 2 | 1 Helper Array: 0 | 0 | 0 | 2 | 1 | 1
13	Next, we look at the 3^{rd} element in the helper array. Because k is 3, and index is 2, we populate the 2^{nd} slot in our input array with a 3, and decrement (-1) the 3^{rd} slot in the helper array. (index = 2, k = 3)	Input Array: 1 | 2 | 3 | 3 | 2 | 1 Helper Array: 0 | 0 | 0 | 1 | 1 | 1
14	Next, we look keep looking at the 3^{rd} element in the helper array. Because k is 3, and index is 3, we populate the 3^{rd} slot in our input array with a 3, and decrement (-1) the 3^{rd} slot in the helper array. (index = 3, k = 3)	Input Array: 1 | 2 | 3 | 3 | 2 | 1 Helper Array: 0 | 0 | 0 | 0 | 1 | 1

15	Next, we look keep looking at the 4th element in the helper array. Because k is 4, and index is 4, we populate the 4th slot in our input array with a 4, and decrement (-1) the 4th slot in the helper array. (index = 4, k = 4)	Input Array: 1 | 2 | 3 | 3 | 4 | 1 Helper Array: 0 | 0 | 0 | 0 | 0 | 1
16	Next, we look keep looking at the 5th element in the helper array. Because k is 5, and index is 5, we populate the 5th slot in our input array with a 5, and decrement (-1) the 5th slot in the helper array. (index = 5, k = 5)	Input Array: 1 | 2 | 3 | 3 | 4 | 5 Helper Array: 0 | 0 | 0 | 0 | 0 | 0
17	Finally, we return the fully sorted array.	Output Array: 1 | 2 | 3 | 3 | 4 | 5

Bibliography:

[1] Imms, Daniel. Growing With the Web, 25 July 2016, www.linkedin.com/in/danielimms.

[2] Cormen, Thomas H., et al. Introduction to algorithms. The MIT Press, 2014.

[3] Counting Sort and Radix Sort, opendatastructures.org/ods-ava/11_2_Counting_Sort_Radix_So.html.

[4] Edmonds, Jeff (2008), "5.2 Counting Sort (a Stable Sort)", How to Think about Algorithms, Cambridge University Press, pp. 72–75, ISBN 978-0-521-84931-9.