Conor Farrell

Professor Chen

CSC 382

21 March 2023

Experiments of Sorting Algorithms

In this experiment, we will be comparing sorting algorithms such as merge sort, quick sort, and counting sort. By comparing these sorting algorithms we can gain a deeper understanding of how they work, what are there strengths and weaknesses, and how do they operate depending on input.

These sorting algorithms will be tested using Python 3.10 using an anaconda3 base environment and will be ran using VSCode's default Python debugger. All three sorting algorithms will be asked to sort an array of random numbers of a given range varying in size. Data such as the runtime in seconds, the number of swaps, and the memory usage. The first sorting algorithm tested is merge sort. Merge sort is a divide and conquer algorithm that splits a problem into many smaller problems and solves them individually then merging the merged parts back together and sorting them as it becomes one finished problem. Its master theory runtime is $2T(n/2) + O(n)$ which makes it an $O(\log(n))$ algorithm. Merge Sort's best, average, and worst time complexity is $O(\log(n))$ with $O(n)$ space complexity. The second algorithm Quick Sort is also a divide and conquer algorithm. It has a master theory runtime of $2T(n/2)+O(n)$ which is the same as merge sort but its best and average case are the only runtimes that are $O(\log(n))$

just like merge sort. Despite merge sort and quick sorts similarities, quick sort is actually a lot more different than merge than you think. For example, quick sorts worst run time is O(n^2) which only happens when a sorted array given as input, but in practice quick sort on average performs better than merge sort since it requires less comparisons to sort than merge sort and it uses less memory with an O(log(n)) space complexity. Quick sort is also not a stable algorithm so it will have trouble with equal values and objects. Although there are stable quick sort algorithms out there to fix this issue but in this experiment we will be using default quick sort. The third and final algorithm in this experiment is counting sort. Counting sort unlike merge sort and quick sort is not a divide and conquer algorithm but instead uses the idea of hashing to sort values. The reason counting sort uses hashing instead of comparisons like the previous sorts is that hashing takes less time than comparing making counting sort much faster than merge sort and quick sort. Counting sorts best, average, and worst case time complexity is always O(n+k) with k being the range of values that the algorithm must create buckets for. These buckets are used to hash the values that are being sorted with each bucket storing one specific value. This limits counting sort in some ways because it can only be used with integer numbers. Depending on implementation it can be used with only positive integers, negative integers, or both. This specific implementation of Counting Sort works will both positive and negative integers. Another limitation is that if the range of numbers is very large it will require a large amount of memory in order to store all the buckets that are being created which makes counting sorts space complexity O(k).

First we will test how the sorts perform with different array sizes. Each array with have randomized numbers from 0-1000 while the array size goes from 10,000 to 100,000 to 500,000.

Array Size 10,000 Range 0-10,000	Merge Sort	Quick Sort	Counting Sort
Time	0.18 s	0.12 s	0.02s
Comparisons/ Buckets	133,616	86,453	1001
Memory	38.37 MB	38.08 MB	38.63 MB

Array Size 100,000 Range 0-10,000	Merge Sort	Quick Sort	Counting Sort
Time	2.18 s	1.74 s	0.15s
Comparisons/ Buckets	1,668,928	1,457,191	1,001
Memory	41.60 MB	43.55 MB	42.46 MB

Array Size 500,000 Range 0-10,000	Merge Sort	Quick Sort	Counting Sort
Time	12.01 s	16.80 s	0.80s
Comparisons/ Buckets	9,475,712	17,472,848	1,001
Memory	57.49 MB	62.19 MB	58.06MB

With our results we can see that with a smaller array quick sort is faster but as the array gets larger merge sort ends up doing being faster. In terms of memory all sorts are similar besides quicksort with an array of size 500,000 taking much longer. We can see how much faster the O(n) counting sort is compared to the O(nlog(n)) merge and quick sort.

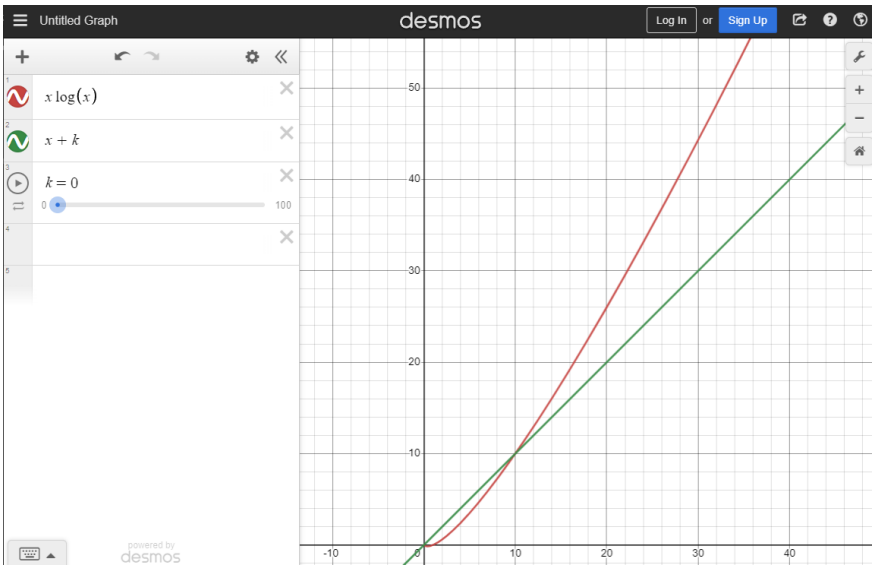
Now we will first test each sorting algorithm with the range of numbers inside the array from 0-10,000, 0-100,000, 0-1,000,000 all with an array of size 10,000.

Array Size 10,000 Range 0-10,000	Merge Sort	Quick Sort	Counting Sort
Time	0.18 s	0.11 s	0.0155s
Comparisons/ Buckets	133,616	80,494	1,001
Memory	38.42 MB	38.24 MB	38.63 MB

Array Size 10,000 Range 0-100,000	Merge Sort	Quick Sort	Counting Sort
Time	0.19	0.12	0.15
Comparisons/Buckets	133,616	83,517	100,001
Memory	38.27 MB	38.15 MB	42.43 MB

Array Size 10,000 Range 0-1,000,000	Merge Sort	Quick Sort	Counting Sort
Time	0.18 s	0.12 s	0.79 s
Comparisons	133,616	87,567	1,000,001
Memory	38.38 MB	38.34 MB	54.30 MB

After experimenting with bucket sizes we can see that the higher range of randomized numbers we include in our array increases the amount of time it takes counting sort to sort them. The reason for this is because counting sort has to loop through larger loops since there are more buckets that it has to look through. This also affected memory usage since a higher range means that the algorithm has to allocate more memory for the buckets. On the other hand, merge sort and quick sort remain mostly unchanged by the change in the range of numbers.



Comparing how the algorithms perform on a graph we can see that the red line being merge sort and quick sort perform better initially but overtime gets worse as the input gets larger. Counting sort on the other hand is always going to be linear and much faster than merge sort and quick sort.

Even when we increase k, the number of buckets the time for counting sort will always catch up and end up being faster despite the size of the array given. The increase in k can also be seen as the amount of memory that will always be allocated by k buckets.

In conclusion, merge sort, quick sort, and counting sort all have their uses in the world of computer science, but some are better at some things than others. For example, counting sort is much faster than merge sort and quick sort at sorting an array of numbers but it is also limited in the amount of memory it produces and that it can only sort numbers. While merge sort and quick sort may be slower it is much more flexible in the world of computer science with its ability to compare strings and objects.

Code:

Psutil library is not included with Python standard library but within the anaconda environment that I have installed. It allows me to easily see how much memory a process is taking.

Merge Sort:

```
import time
import random
import os
import psutil
# Python program for implementation of MergeSort
def mergeSort(arr,swaps):
    if len(arr) > 1:

         # Finding the mid of the array
        mid = len(arr)//2
        #print("Middle is here at index: ", mid, ' ', arr, end=' ')
        # Dividing the array elements
        L = arr[:mid]
        #print("Left starts at 1 and goes to ", len(L), " ", L, end =' ')
        # into 2 halves
        R = arr[mid:]
        #print("Right starts at ", mid + 1, " and goes until ", len(arr), ' ', R,
sep='')

        # Sorting the first half
        swaps = mergeSort(L,swaps) + mergeSort(R,swaps)
        #print("Left Merge ", swaps)
        # Sorting the second half

        #print("Right Merge ",swaps)
        i = j = k = 0

        # Copy data to temp arrays L[] and R[]
        while i < len(L) and j < len(R):
            if L[i] <= R[j]:
                arr[k] = L[i]
                swaps += 1
                i += 1
            else:
                arr[k] = R[j]
                swaps += 1
                j += 1
            #print("While and ",swaps)
            k += 1

        # Checking if any element was left
        while i < len(L):
            arr[k] = L[i]
            swaps += 1
            i += 1
            k += 1
        #print("while i ",swaps)
        while j < len(R):
```

```
            arr[k] = R[j]
            swaps += 1
            j += 1
            k += 1
        #print("while j ",swaps)
        return swaps
    #print("arr < 1" ,swaps)
    return 0


# generate random numbers for array
def randomArray(size):
    arr = []

    for _ in range(size):
        arr.append(random.randint(0,10000))

    #print(arr)
    return arr

# Driver Code

# code for looping from array sizes 0 to 10000
# for i in range(0,10001,1000):
#     arr = randomArray(i)
#     start = time.time()
#     swaps = 0
#     swaps = mergeSort(arr,swaps)
#     end = time.time()
#     print("Time taken for merge sort: ", round(end-start,2), " seconds", "with
array size ", i, " and this many swaps " , swaps)

# code used for comparisons
process = psutil.Process(os.getpid())
arr = randomArray(10000)
start = time.time()
swaps = 0
swaps = mergeSort(arr,swaps=0)
end = time.time()
print("Time taken for merge sort: ", round(end-start,2), " seconds ", "with array
size ", len(arr), " and this many swaps " , swaps)
print("Memory usage: ", process.memory_full_info().rss / 1000000, " MB")
```

Quick Sort:

```
import time
import random
```

```python
import os
import psutil

# Function to find the partition position
def partition(array, low, high, swaps):

    # Choose the rightmost element as pivot
    pivot = array[high]

    # Pointer for greater element
    i = low - 1

    swap_count = 0
    # Traverse through all elements
    # compare each element with pivot
    for j in range(low, high):
        if array[j] <= pivot:
            # If element smaller than pivot is found
            # swap it with the greater element pointed by i
            i = i + 1

            # Swapping element at i with element at j
            (array[i], array[j]) = (array[j], array[i])
            swap_count += 1

    #print("arr after pivot ", arr)
    # Swap the pivot element with
    # e greater element specified by i
    (array[i + 1], array[high]) = (array[high], array[i + 1])
    swap_count += 1
    swaps.append(swap_count)
    #print("after partition ", arr)
    # Return the position from where partition is done
    #print(swaps)
    return i + 1

# Function to perform quicksort


def quick_sort(array, low, high, swaps):
    if low < high:

        # Find pivot element such that
        # element smaller than pivot are on the left
        # element greater than pivot are on the right
        pi = partition(array, low, high, swaps)

        # Recursive call on the left of pivot
        quick_sort(array, low, pi - 1,swaps)

        # Recursive call on the right of pivot
        quick_sort(array, pi + 1, high,swaps)
```

```
def randomArray(size):
    arr = []

    for _ in range(size):
        arr.append(random.randint(0,10000))

    #print(arr)
    return arr



# Driver Code
if __name__ == '__main__':

    # for i in range(0,10001,1000):
    #     arr = randomArray(i)
    #     start = time.time()
    #     swaps = []
    #     quick_sort(arr,0,len(arr)-1,swaps)
    #     end = time.time()
    #     print("Time taken for merge sort: ", round(end-start,2), " seconds ", "with
array size ", i, " and this many swaps " , sum(swaps))

    process = psutil.Process(os.getpid())
    arr = randomArray(10000)
    start = time.time()
    swaps = []
    quick_sort(arr,0,len(arr)-1,swaps)
    end = time.time()
    print("Time taken for quick sort: ", round(end-start,2), " seconds ", "with array
size ", len(arr), " and this many swaps " , sum(swaps))
    print("Memory usage: ", process.memory_full_info().rss / 1000000, " MB")
```

Counting Sort:

```
import time
import random
import os
import psutil


def count_sort(arr):
    max_element = int(max(arr))
    min_element = int(min(arr))
    range_of_elements = max_element - min_element + 1
    # Create a count array to store count of individual
    # elements and initialize count array as 0
    count_arr = [0 for _ in range(range_of_elements)]
    output_arr = [0 for _ in range(len(arr))]
    #print("Create buckets with count array: ", count_arr)
    #print("Output arr: ", output_arr)
```

```
    # Store count of each character
    for i in range(0, len(arr)):
        count_arr[arr[i]-min_element] += 1
    #print("Count number of times each number appears buckets ", count_arr)
    # Change count_arr[i] so that count_arr[i] now contains actual
    # position of this element in output array
    for i in range(1, len(count_arr)):
        count_arr[i] += count_arr[i-1]
    #print("Add elements together elements: ", count_arr)
    # Build the output character array
    for i in range(len(arr)-1, -1, -1):
        output_arr[count_arr[arr[i] - min_element] - 1] = arr[i]
        count_arr[arr[i] - min_element] -= 1
    #print("Create output array: ", output_arr)
    #print("Count used: ", count_arr)
    # Copy the output array to arr, so that arr now
    # contains sorted characters
    for i in range(0, len(arr)):
        arr[i] = output_arr[i]
    #print("Return: ", arr)
    return arr

def randomArray(size):
    arr = []

    for _ in range(size):
        arr.append(random.randint(0,10000))

    #print(arr)
    return arr

# Driver code
if __name__ == '__main__':
    process = psutil.Process(os.getpid())
    arr = randomArray(10000)
    start = time.time()
    ans = count_sort(arr)
    end = time.time()
    print("Time taken for Counting Sort: ", round(end-start,7), " seconds with array
of size ", len(arr))
    print("Memory usage: ", process.memory_full_info().rss / 1000000, " MB")
    #print(str(ans))
```