

Analysis of the Runtime of the Mergesort Algorithm Compared to the Quicksort Algorithm on Large Data Sets

Daigan Berger and Taha Alnasser

1 Objective

The objective of the research was to compare the runtimes of the **Mergesort** and **Quicksort** sorting algorithms and discern if there is any significant difference in the runtimes of the algorithms on large datasets. Both of these algorithms operate in $O(n \log_2 n)$ where n represents the size of the input array. It is commonly assumed for large data sets that **Quicksort** is faster, and this assumption will be put to the test in this experiment.

2 Experiment Setup and Assumptions

1. Python 3.10.11 (64 bit) is used
2. The same computer will be used throughout the entire experiment
3. For each trial, the same randomly generated array of size 100,000 will be used
4. There will be $n = 1000$ trials performed for each algorithm
5. We are assuming in this experiment that $\sigma_Q = \sigma_M$ because sorting algorithms tend to contain very little and similar amounts of variance between different data sets
6. All timings are measured in seconds

3 Hypothesis

Define μ_Q as the average time in seconds of sorting the datasets using **Quicksort** Define μ_M as the average time in seconds of sorting the datasets using **Mergesort**

$H_0 : \mu_Q - \mu_M = 0$ (**Mergesort** is faster than or equal to **Quicksort** in speed)

$H_A : \mu_Q - \mu_M < 0$ (**Quicksort** is faster than **Mergesort** in speed)

```

1 # Function to find the partition position
2 def partition(array, low, high):
3
4     # choose the rightmost element as pivot
5     pivot = array[high]
6
7     # pointer for greater element
8     i = low - 1
9
10    # traverse through all elements
11    # compare each element with pivot
12    for j in range(low, high):
13        if array[j] <= pivot:
14
15            # If element smaller than pivot is found
16            # swap it with the greater element pointed by i
17            i = i + 1
18
19            # Swapping element at i with element at j
20            (array[i], array[j]) = (array[j], array[i])
21
22    # Swap the pivot element with the greater element specified by i
23    (array[i + 1], array[high]) = (array[high], array[i + 1])
24
25    # Return the position from where partition is done
26    return i + 1
27
28 # function to perform quicksort
29
30
31 def quickSort(array, low, high):
32     if low < high:
33
34         # Find pivot element such that
35         # element smaller than pivot are on the left
36         # element greater than pivot are on the right
37         pi = partition(array, low, high)
38
39         # Recursive call on the left of pivot
40         quickSort(array, low, pi - 1)
41
42         # Recursive call on the right of pivot
43         quickSort(array, pi + 1, high)
44
45 def sort_q(array, size):
46     quickSort(array, 0, size - 1)

```

Listing 1: Quicksort Code

```

1 def merge(arr, l, m, r):
2     n1 = m - l + 1
3     n2 = r - m
4
5     # create temp arrays
6     L = [0] * (n1)
7     R = [0] * (n2)
8
9     # Copy data to temp arrays L[] and R[]
10    for i in range(0, n1):
11        L[i] = arr[l + i]
12
13    for j in range(0, n2):
14        R[j] = arr[m + 1 + j]
15
16    # Merge the temp arrays back into arr[l..r]
17    i = 0      # Initial index of first subarray
18    j = 0      # Initial index of second subarray
19    k = l      # Initial index of merged subarray
20
21    while i < n1 and j < n2:
22        if L[i] <= R[j]:
23            arr[k] = L[i]
24            i += 1
25        else:
26            arr[k] = R[j]
27            j += 1
28        k += 1
29
30    # Copy the remaining elements of L[], if there
31    # are any
32    while i < n1:
33        arr[k] = L[i]
34        i += 1
35        k += 1
36
37    # Copy the remaining elements of R[], if there
38    # are any
39    while j < n2:
40        arr[k] = R[j]
41        j += 1
42        k += 1
43
44    # l is for left index and r is right index of the
45    # sub-array of arr to be sorted
46
47
48 def mergeSort(arr, l, r):
49     if l < r:
50
51         # Same as (l+r)//2, but avoids overflow for

```

```

52         # large l and h
53         m = l+(r-1)//2
54
55         # Sort first and second halves
56         mergeSort(arr, l, m)
57         mergeSort(arr, m+1, r)
58         merge(arr, l, m, r)
59
60 def sort_m(array, size):
61     mergeSort(array, 0, size-1)

```

Listing 2: Mergesort Code

```

1  import time
2  import random
3  import copy
4  import math
5  # Function to find the partition position
6  def partition(array, low, high):
7
8      # choose the rightmost element as pivot
9      pivot = array[high]
10
11     # pointer for greater element
12     i = low - 1
13
14     # traverse through all elements
15     # compare each element with pivot
16     for j in range(low, high):
17         if array[j] <= pivot:
18
19             # If element smaller than pivot is found
20             # swap it with the greater element pointed by i
21             i = i + 1
22
23             # Swapping element at i with element at j
24             (array[i], array[j]) = (array[j], array[i])
25
26     # Swap the pivot element with the greater element specified by i
27     (array[i + 1], array[high]) = (array[high], array[i + 1])
28
29     # Return the position from where partition is done
30     return i + 1
31
32 # function to perform quicksort
33
34
35 def quickSort(array, low, high):
36     if low < high:
37
38         # Find pivot element such that
39         # element smaller than pivot are on the left

```

```

40         # element greater than pivot are on the right
41         pi = partition(array, low, high)
42
43         # Recursive call on the left of pivot
44         quickSort(array, low, pi - 1)
45
46         # Recursive call on the right of pivot
47         quickSort(array, pi + 1, high)
48
49 def sort_q(array, size):
50     quickSort(array, 0, size - 1)
51
52
53 def merge(arr, l, m, r):
54     n1 = m - l + 1
55     n2 = r - m
56
57     # create temp arrays
58     L = [0] * (n1)
59     R = [0] * (n2)
60
61     # Copy data to temp arrays L[] and R[]
62     for i in range(0, n1):
63         L[i] = arr[l + i]
64
65     for j in range(0, n2):
66         R[j] = arr[m + 1 + j]
67
68     # Merge the temp arrays back into arr[l..r]
69     i = 0      # Initial index of first subarray
70     j = 0      # Initial index of second subarray
71     k = l      # Initial index of merged subarray
72
73     while i < n1 and j < n2:
74         if L[i] <= R[j]:
75             arr[k] = L[i]
76             i += 1
77         else:
78             arr[k] = R[j]
79             j += 1
80         k += 1
81
82     # Copy the remaining elements of L[], if there
83     # are any
84     while i < n1:
85         arr[k] = L[i]
86         i += 1
87         k += 1
88
89     # Copy the remaining elements of R[], if there
90     # are any

```

```

91     while j < n2:
92         arr[k] = R[j]
93         j += 1
94         k += 1
95
96 # l is for left index and r is right index of the
97 # sub-array of arr to be sorted
98
99
100 def mergeSort(arr, l, r):
101     if l < r:
102
103         # Same as (l+r)//2, but avoids overflow for
104         # large l and h
105         m = l+(r-l)//2
106
107         # Sort first and second halves
108         mergeSort(arr, l, m)
109         mergeSort(arr, m+1, r)
110         merge(arr, l, m, r)
111
112 def sort_m(array, size):
113     mergeSort(array, 0, size-1)
114
115 def timing():
116     experiment_size = 1000
117     array_size = 100000
118     quick_sort_times = []
119     merge_sort_times = []
120     for i in range(experiment_size):
121         arr_merge_sort = generate_random(array_size)
122         arr_quick_sort = copy.deepcopy(arr_merge_sort)
123         timer = Timer()
124
125         timer.start_timer()
126         sort_m(arr_merge_sort, array_size)
127         timer.end_timer()
128         merge_sort_times.append(timer.get_elapsed_time())
129
130         timer.start_timer()
131         sort_q(arr_quick_sort, array_size)
132         timer.end_timer()
133         quick_sort_times.append(timer.get_elapsed_time())
134     mean_quick_sort = get_mean(quick_sort_times, experiment_size)
135     mean_merge_sort = get_mean(merge_sort_times, experiment_size)
136
137     var_quick_sort = get_var(mean_quick_sort, quick_sort_times, experiment_size)
138     var_merge_sort = get_var(mean_merge_sort, merge_sort_times, experiment_size)
139     print("Results: -----")
140     print("Mean of quick sort: ", mean_quick_sort)
141     print("Mean of merge sort: ", mean_merge_sort)

```

```

142     print("Variance of quick sort: ", var_quick_sort)
143     print("Variance of merge sort: ", var_merge_sort)
144     f = open("results.csv" ,"w")
145     f.write("Trial,Quick sort data,Merge Sort Data,,Quick sort mean,Merge sort
mean,Quick sort variance, Merge sort variance\n")
146     f.write(str(1) +"," +str(quick_sort_times[0]) + "," + str(merge_sort_times[0])
+ ",," + str(mean_quick_sort) + "," + str(mean_merge_sort) + "," + str(
var_quick_sort) + "," + str(var_merge_sort) + "\n")
147     for i in range(1, experiment_size):
148         f.write(str(i+1) +"," +str(quick_sort_times[i]) + "," + str(
merge_sort_times[i]) + "\n")
149     f.close()
150
151
152 def get_mean(arr, experiment_size):
153     sum = 0
154     for element in arr:
155         sum = sum + element
156     return sum / experiment_size
157 def get_var(mean, arr, experiment_size):
158     sum = 0
159     for element in arr:
160         sum += pow((element-mean),2)
161     return sum/(experiment_size -1)
162
163 class Timer():
164     def __init__(self):
165         self.delta_time = 0
166         self.start_time = 0
167     def start_timer(self):
168         self.start_time = time.time()
169     def end_timer(self):
170         if (self.start_time) == 0:
171             return "timer not started"
172         self.delta_time = (time.time() - self.start_time)
173         self.start_time = 0
174     def get_elapsed_time(self):
175         return self.delta_time
176
177
178 def generate_random(size):
179     return [random.randint(0, 100000) for i in range(size)]
180 timing()

```

Listing 3: Timing Code

4 Results

$$s_p^2 = \frac{(s_Q^2)(n_Q-1) + (s_M^2)(n_M-1)}{n_Q + n_M - 2} = \frac{(3.37 \cdot 10^{-5})(1000-1) + (7.53 \cdot 10^{-5})(1000-1)}{1000+1000-2} \approx 5.45 \cdot 10^{-5}$$

$$t = \frac{\bar{x}_Q - \bar{x}_M - d_0}{s_p \sqrt{\frac{1}{n_Q} + \frac{1}{n_M}}} = \frac{0.147 - 0.302 - 0}{\sqrt{5.45 \times 10^{-5}} \sqrt{\frac{1}{1000} + \frac{1}{1000}}} = -469.48$$

$$v = n_Q + n_M - 2 = 1998$$

$$t_{0.05,1998} \approx t_{0.05,\infty} = 1.645$$

5 Conclusions

Because our test statistic t fell into the rejection region, we can reject H_0 , the null hypothesis, and conclude that at the 0.05 significance level that **Quicksort** is faster than **Mergesort** in speed using the assumptions and experiment setup presented previously

6 References

1. <https://www.geeksforgeeks.org/python-program-for-merge-sort/>
2. <https://www.geeksforgeeks.org/python-program-for-quicksort/>

Rejection Region:

$$t < t_{\alpha, n_Q + n_M - 2}$$

$$-469.48 < 1.645$$

Confidence Interval:

$$error = t_{\alpha/2} s_p \sqrt{\frac{1}{n_Q} + \frac{1}{n_M}}$$

$$t_{0.025,1998} \approx t_{0.025,\infty} = 1.645$$

$$error = 1.645 \cdot 5.45 \cdot 10^{-5} \cdot \sqrt{\frac{1}{1000} + \frac{1}{1000}} = 4.01 \cdot 10^{-6}$$

$$(\bar{x}_Q - \bar{x}_M) - error < \mu_Q - \mu_M < (\bar{x}_Q - \bar{x}_M) + error$$

$$(0.147 - 0.302) - 4.01 \cdot 10^{-6} < \mu_Q - \mu_M < (0.147 - 0.302) + 4.01 \cdot 10^{-6}$$

$$-0.15500 < \mu_Q - \mu_M < -0.15499$$

7 Conclusions

Because our test statistic t fell into the rejection region, we can reject H_0 , the null hypothesis, and conclude that at the 0.05 significance level that **Quicksort** is faster than **Mergesort** in speed using the assumptions and experiment setup presented previously

8 References

1. <https://www.geeksforgeeks.org/python-program-for-merge-sort/>
2. <https://www.geeksforgeeks.org/python-program-for-quicksort/>