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 100,0000 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)

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

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

    # pointer for greater element
    i = low - 1

    # 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 the pivot element with the greater element specified by i
    (array[i + 1], array[high]) = (array[high], array[i + 1])

    # Return the position from where partition is done
    return i + 1

# function to perform quicksort


def quickSort(array, low, high):
    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)

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

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

def sort_q(array, size):
    quickSort(array, 0, size - 1)
```

Listing 1: Quicksort Code

```
def merge(arr, l, m, r):
    n1 = m - l + 1
    n2 = r - m

    # create temp arrays
    L = [0] * (n1)
    R = [0] * (n2)

    # Copy data to temp arrays L[] and R[]
    for i in range(0, n1):
        L[i] = arr[l + i]

    for j in range(0, n2):
        R[j] = arr[m + 1 + j]

    # Merge the temp arrays back into arr[l..r]
    i = 0     # Initial index of first subarray
    j = 0     # Initial index of second subarray
    k = l     # Initial index of merged subarray

    while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1

    # Copy the remaining elements of L[], if there
    # are any
    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1

    # Copy the remaining elements of R[], if there
    # are any
    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1

# l is for left index and r is right index of the
# sub-array of arr to be sorted


def mergeSort(arr, l, r):
    if l < r:

        # Same as (l+r)//2, but avoids overflow for
```

```
        # large l and h
        m = l+(r-l)//2

        # Sort first and second halves
        mergeSort(arr, l, m)
        mergeSort(arr, m+1, r)
        merge(arr, l, m, r)

def sort_m(array, size):
    mergeSort(array, 0, size-1)
```

Listing 2: Mergesort Code

```
import time
import random
import copy
import math
# Function to find the partition position
def partition(array, low, high):

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

    # pointer for greater element
    i = low - 1

    # 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 the pivot element with the greater element specified by i
    (array[i + 1], array[high]) = (array[high], array[i + 1])

    # Return the position from where partition is done
    return i + 1

# function to perform quicksort


def quickSort(array, low, high):
    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)

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

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

def sort_q(array, size):
    quickSort(array, 0, size - 1)


def merge(arr, l, m, r):
    n1 = m - l + 1
    n2 = r - m

    # create temp arrays
    L = [0] * (n1)
    R = [0] * (n2)

    # Copy data to temp arrays L[] and R[]
    for i in range(0, n1):
        L[i] = arr[l + i]

    for j in range(0, n2):
        R[j] = arr[m + 1 + j]

    # Merge the temp arrays back into arr[l..r]
    i = 0     # Initial index of first subarray
    j = 0     # Initial index of second subarray
    k = l     # Initial index of merged subarray

    while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1

    # Copy the remaining elements of L[], if there
    # are any
    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1

    # Copy the remaining elements of R[], if there
    # are any
```

```python
    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1

# l is for left index and r is right index of the
# sub-array of arr to be sorted


def mergeSort(arr, l, r):
    if l < r:

        # Same as (l+r)//2, but avoids overflow for
        # large l and h
        m = l+(r-l)//2

        # Sort first and second halves
        mergeSort(arr, l, m)
        mergeSort(arr, m+1, r)
        merge(arr, l, m, r)

def sort_m(array, size):
    mergeSort(array, 0, size-1)
def timing():
    experiment_size = 1000
    array_size = 100000
    quick_sort_times = []
    merge_sort_times = []
    for i in range(experiment_size):
        arr_merge_sort = generate_random(array_size)
        arr_quick_sort = copy.deepcopy(arr_merge_sort)
        timer = Timer()

        timer.start_timer()
        sort_m(arr_merge_sort, array_size)
        timer.end_timer()
        merge_sort_times.append(timer.get_elapsed_time())


        timer.start_timer()
        sort_q(arr_quick_sort, array_size)
        timer.end_timer()
        quick_sort_times.append(timer.get_elapsed_time())
    mean_quick_sort = get_mean(quick_sort_times, experiment_size)
    mean_merge_sort = get_mean(merge_sort_times, experiment_size)

    var_quick_sort = get_var(mean_quick_sort,quick_sort_times, experiment_size)
    var_merge_sort = get_var(mean_merge_sort, merge_sort_times, experiment_size)
    print("Results: ----------")
    print("Mean of quick sort: ", mean_quick_sort)
    print("Mean of merge sort: ", mean_merge_sort)
```

```python
    print("Variance of quick sort: ", var_quick_sort)
    print("Variance of merge sort: ", var_merge_sort)
    f = open("results.csv" ,"w")
    f.write("Trial,Quick sort data,Merge Sort Data,,Quick sort mean,Merge sort mean,Quick sort variance, Merge sort variance\n")
    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")
    for i in range(1, experiment_size):
        f.write(str(i+1) +","+str(quick_sort_times[i]) + "," + str(merge_sort_times[i]) + "\n")
    f.close()


def get_mean(arr, experiment_size):
    sum = 0
    for element in arr:
        sum = sum + element
    return sum / experiment_size
def get_var(mean, arr, experiment_size):
    sum = 0
    for element in arr:
        sum += pow((element-mean),2)
    return sum/(experiment_size -1)

class Timer():
    def __init__(self):
        self.delta_time = 0
        self.start_time = 0
    def start_timer(self):
        self.start_time = time.time()
    def end_timer(self):
        if (self.start_time) == 0:
            return "timer not started"
        self.delta_time = (time.time() - self.start_time)
        self.start_time = 0
    def get_elapsed_time(self):
        return self.delta_time


def generate_random(size):
    return [random.randint(0, 100000) for i in range(size)]
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:

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

$-469.48 < 1.645$

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/**