Sajad Gholamzadehrizi
CSC 22000
Sorting Algorithms Running Time Analysis Report

Arrays of sizes 10, 100, 1000, 10000, 100000, 1000000 are used for the analysis. They are randomly initialized by integers ranging from 0 to 100000.

Analyzing the results of sample run below it can be noticed that even being an $O(n^2)$ algorithm, insertion sort works reasonably fast when the size of the array is less than 100 on an average compared to other sorting algorithms.

However, as the size grows, insertion sort running time rises very fast. Quick sorts seem to work the best in these cases especially as the size of the array increases. Heap sort would be next in terms of the best running time (slower growth as the input size increases) followed by merge sort.

A sample run:

Size of array	10	100	1000	10000	100000	1000000
Insertion Sort	0.03982 milliseconds	1.27482 milliseconds	50.38810 milliseconds	4.21849 seconds	Exceeds a minute	Exceeds an hour
Merge Sort	0.36621 milliseconds	1.33610 milliseconds	9.06396 milliseconds	76.84779 milliseconds	779.29401 milliseconds	8.99332 seconds
Heap Sort	0.06127 milliseconds	0.30804 milliseconds	1.62292 milliseconds	19.30404 milliseconds	160.04086 milliseconds	1.59725 seconds
Quick Sort w/ last element as pivot	0.06294 milliseconds	0.42129 milliseconds	9.01198 milliseconds	36.22198 milliseconds	416.26620 milliseconds	5.77156 seconds
Quick Sort w/ random pivot	0.09179 milliseconds	0.63682 milliseconds	3.87907 milliseconds	45.00413 milliseconds	534.48129 milliseconds	6.97310 seconds

Insertion sort for the case of 100000 input size would take minutes and for the case of 1000000, would take hours to complete so I bounded that by 30 seconds time when running the program by raising an exception at 30 second so the program could move forward with other sorting algorithms.

Proper exception handling is also applied on quicksort as the worst case of quicksort in the case that the randomly generated array is nearly sorted or reversely sorted can be $\Theta(n^2)$ as well, because of the choice of pivot element.

A more detailed analysis of each sorting algorithm is explained as follows.

Insertion Sort:

In insertion sort, the algorithm works by iteratively going from array second element to length of array and finding the minimum in that array and then bringing it to its correct place one by one. So, it includes two loops, one for iterating from 1 to array length and other to shift down the minimum element. That is the reasons insertion sort is $O(n^2)$ algorithm in the worst case. The best case could happen when the array is nearly sorted, in which the second loop would not run since the second loop which does that swapping, would only run if the elements are not sorted, in this case the running time would be $\Omega(n)$.

```
def insertion_sort(array):
    s = time.time()         # saving start time to avoid a long running time by throwing an exception
    i = 1
    while i < len(array):   # loop untill i is less than length of array
        val = array[i]      # key is selected at index i
        j = i-1
        while j >=0 and val < array[j] :  # looping from index i-1 to 0
                array[j+1] = array[j]     # bring the lowest element to its place
                j -= 1
        array[j+1] = val    # settig the element at its correct place after findig actual j index
        if time.time() - s > 30:    # raising exception if execution time exceeded 30 seconds
            raise Exception('Time taken by insertion sort more than 30 seconds so aborted')
        i+=1
```

Merge Sort:

In Merge sort, we aim at breaking our problem of sorting the big array to sorting multiple smaller arrays and then merging them to finally get the big sorted array. This approach is called divide and conquer. The array is split into left and right subarrays until the size reaches 1, and then merges the arrays such that the merged array is sorted. Merge is done in Θ(n) time. And the breaking of the array into two equal halves every time result in two recursive calls or in a binary tree depth of Logarithm based 2 of n, where n is the array size. Solving the recurrence would give us a running time of Θ(n Logn).

```
def merge_sort(array,left,right):
    if left < right:
        m = (left+(right-1))/2          # finding the index about which array is split
        merge_sort(array, left, m)    # solving the split array from l to m first
        merge_sort(array, m+1, right) # solving the split arrayy from m+1 to r secondly
        merge(array, left, m, right)     # merging the sorted left and right subarray efficiently
```

Finding the index m where the split is going to happen which is the middle element. And then solving the left and right subarrays recursively.
Merge method is as follows which merges two sorted arrays into one big sorted array in Θ(n) time.

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

    Left = [0] * (n1)          # creatig empty array of size n1
    Right = [0] * (n2)          # creatig empty array of size n2

    # copying elements on left side to left array
    for i in range(0 , n1):
        Left[i] = arr[l + i]

    # copying elements on left side to right arry
    for j in range(0 , n2):
        Right[j] = arr[m + 1 + j]


    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 Left[i] <= Right[j]:
            arr[k] = Left[i]
            i += 1
        else:
            arr[k] = Right[j]
            j += 1
        k += 1

    while i < n1:
        arr[k] = Left[i]
        i += 1
        k += 1

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

Heap Sort:

In heap sort, we aim at making a binary tree which has a unique property that every child of a parent is smaller/larger than the parent. This way when we retrieve the smallest/largest element, we can build a sorted array. In this case, we are building a Max heap.

In heap sort, we first heapify the array by continuously and recursively swapping the largest element in the children with the parent. This builds a nearly but not completely heapified tree. This would take Θ(Logn) time.

Next, we need to build a complete heap by iteration over the tree stating from the last parent to the root of the tree. This would take Θ(n Logn) time.

After that, we will loop over the entire tree elements and keep swapping every element with the first element and heapifying the tree. Finally, the entire sorting process would take Θ(n Logn) + Θ(n Logn) time, or just simply Θ(n Logn) time.

```
def heapify(arr, size, i):
    largest = i  # Initialize largest as root
    left = 2 * i + 1     # left child of parent i
    right = 2 * i + 2     # right child of parent i


    if left < size and arr[i] < arr[left]:  # if left child exists and is less than parent i
        largest = left                      # set largest as left

    if right < size and arr[largest] < arr[right]:  # if right child exists and is less than parent i
        largest = right                             # set largest as right

    if largest != i:    # swap the largest with i and recursively call heapify at largest down the heap
        arr[i],arr[largest] = arr[largest],arr[i]  # swap

        # Heapify the root.
        heapify(arr, size, largest)

def heap_sort(arr):
    i = len(arr)
    while i >=0:
        heapify(arr, len(arr) , i)      # building the heap
        i-=1

    i=len(arr) -1       # starting loop with i as len(arr) -1
    while i>=1:         # while i is greater than equal to 1
        swap(arr,i,0)               # swap i and 0 indexed elements of array named arr
        arr[i], arr[0] = arr[0], arr[i]   # swap
        heapify(arr, i, 0)
        i-=1
```

Quick Sort – Last element as pivot and Random pivot:

Like merge sort, Quick Sort is a divide and conquer algorithm. In quicksort, the algorithm aims at partitioning an array by a computed pivot such that all elements greater than pivot come on the right side of pivot and all elements smaller than pivot come on the left side of pivot.
If we keep doing this for all the partitioned subarrays , we can be sure that the array is finally sorted at the end.
The algorithm mainly depends on the process of finding the location of pivot element.
Choosing the pivot as the last element or a random element in the array are two ways of different ways to select the pivot.

The for both is similar and quite simple. The difference between the random pivot and the last element as pivot, is in the way we choose the pivot element. Our partition method works with the last element as the pivot. In order to make it work with a random element as well, after we pick a random element in array, we swap that with the last element in the array and keep going with the same logic we have in the partition method.
We start from the first element and loop over array, if we find an element smaller than the pivot, we swap that with the element at index i. i is initially set to the first index minus one and gets incremented every time that we want to swap an element as we loop over array and keep comparing with the pivot. By keep doing this we would find the partitioning index to be returned.
Because of this loop, partitioning takes $\Theta(n)$ time. Now, what determines the best or worst case in this algorithm is whether or not the array is sorted/reversely sorted. When the array is not sorted, pivot would fall somewhere in the middle of the array, and partitioning builds up two recursive calls with size of nearly n/2. In this case, the algorithm would take $\Omega(n \log n)$ time to finish which is called the best case.
Whereas, if the array is nearly sorted, pivot will be at the end of the array, it will leave a recursive call with size of n and a recursive call with size zero. This case would take $O(n^2)$ time to finish which is much worse than the first case.
Compared to the last element as pivot, by choosing a random pivot, we are increasing the chance of falling the partitioning point somewhere closer to the middle of the array on average. In theory we can increase the chance even further by choosing 3 or more random elements and picking the median as the pivot. But the time of spending doing so may not always be to our benefit.

In general, picking a random pivot would be better than choosing the last element all the time as the pivot.

```
def partition(array,start,end):    # selecting last element as pivot
    i = ( start-1 )         # index to keep track of partition index
    pivot = array[end]     # pivot selected as last element
    j = start
    while j < end:          # loop from low to high
        if   array[j] <= pivot:   # partition by keeping elements lower than pivot to its left
            i = i+1
            array[i],array[j] = array[j],array[i]   # swapping
        j+=1
    array[i+1],array[end] = array[end],array[i+1]
    return ( i+1 )      # return the index about which partition happens finally

def quick_sort_pivot_last(arr,start,end):
    s = time.time()
    if start < end:                                 # solve until start is less than end
        partition_index = partition(arr,start,end)  # find the partiton element index
        quick_sort_pivot_last(arr, start, partition_index-1)    # recurse with left subarray
        quick_sort_pivot_last(arr, partition_index+1, end)      # recurse with right subarray
    if time.time() - s > 30:    # raising exception if execution time exceeded 30 seconds
        raise Exception('Time taken by quick sort (last pivot) more than 30 seconds so aborted')

def partition_random(array, low, high):
    r = random.randrange(low,high)      # selecting a random element as pivot
    array[r],array[high] = array[high],array[r] # swapping with the last element
    return partition(array, low, high)  # moving forward with our partition() method for last pivot

def quick_sort_random_pivot(arr,start,end):
    s = time.time()
    if start < end:                             #solve until start is less than end
        pi = partition_random(arr,start,end)    #partition the array
        quick_sort_random_pivot(arr, start, pi-1)   #recursively solve left subarray of pivot
        quick_sort_random_pivot(arr, pi+1, end)  #recursively solve right subarray of pivot
    if time.time() - s > 30:    # raising exception if execution time exceeded 30 seconds
        raise Exception('Time taken by quick sort (last pivot) more than 30 seconds so aborted')
```

SAMPLE OUTPUT:

Enter the input size of array 10
Running time of insertion sort 0.03982 milliseconds
Running time of merge sort 0.36621 milliseconds
Running time of heap sort 0.06127 milliseconds
Running time of quickSort_last 0.06294 milliseconds
Running time of quickSort_random 0.09179 milliseconds

[18018, 23550, 23698, 50354, 56695, 62658, 74183, 84965, 95007, 98780]

Enter the input size of array 100
Running time of insertion sort 1.27482 milliseconds
Running time of merge sort 1.33610 milliseconds
Running time of heap sort 0.30804 milliseconds
Running time of quickSort_last 0.42129 milliseconds
Running time of quickSort_random 0.63682 milliseconds

[714, 2342, 2494, 2610, 2925, 3410, 4314, 4695, 5328, 6442, 6867, 7001, 7705, 8251, 9061, 14552, 15933, 16170, 17547, 17691, 19304, 19638, 20606, 20795, 23114, 23310, 23413, 23447, 23599, 24639, 26247, 28304, 28652, 29654, 30084, 30878, 31538, 32780, 34271, 36249, 38179, 40017, 40897, 42579, 42981, 43395, 43801, 44722, 45560, 46698, 47638, 48671, 50078, 50503, 50726, 50813, 52862, 53394, 53638, 55150, 55462, 55721, 57525, 57797, 58820, 59422, 63173, 64650, 64944, 64957, 65331, 65411, 66863, 67796, 68487, 70654, 70916, 72442, 73070, 73985, 75238, 76432, 77942, 80648, 81745, 82047, 82509, 82688, 84909, 85886, 87433, 89018, 90988, 92022, 93788, 94604, 95721, 96481, 96837, 99339]

Enter the input size of array 1000
Running time of insertion sort 50.38810 milliseconds
Running time of merge sort 9.06396 milliseconds
Running time of heap sort 1.62292 milliseconds
Running time of quickSort_last 9.01198 milliseconds
Running time of quickSort_random 3.87907 milliseconds

Enter the input size of array 10000
Running time of insertion sort 4.21849 seconds
Running time of merge sort 76.84779 milliseconds
Running time of heap sort 19.30404 milliseconds
Running time of quickSort_last 36.22198 milliseconds
Running time of quickSort_random 45.00413 milliseconds

Enter the input size of array 100000
Running time of insertion sort exceeds 30 seconds so aborted
Running time of merge sort 779.29401 milliseconds
Running time of heap sort 160.04086 milliseconds
Running time of quickSort_last 416.26620 milliseconds
Running time of quickSort_random 534.48129 milliseconds

Enter the input size of array 1000000
Running time of insertion sort exceeds 30 seconds so aborted
Running time of merge sort 8.99332 seconds
Running time of heap sort 1.59725 seconds
Running time of quickSort_last 5.77156 seconds
Running time of quickSort_random 6.97310 seconds