CS 599: Algorithms **(Due: October 8th, 2020)**

Submission Homework 2

Instructor: Prof. Prasad Tadepalli *Name:* Rashmi Jadhav, *Student ID:* 934-069-574

1. Implementation of Merge Sort and Quick Sort

1. Merge Sort:

```python
# Time Complexity: O(n log n); worst case O(n^2) when list is already sorted
# Space Complexity: O(log n) stack space for recursion; O(n) stack in worst case
def quick_sort_recursive(numbers, low, high):
    if low >= high:
        return
    partition_index = partition(numbers, low, high)
    quick_sort_recursive(numbers, low, partition_index - 1)
    quick_sort_recursive(numbers, partition_index, high)


def partition(numbers, low, high):
    pivot = numbers[(low + high) // 2]  # Take middle element as pivot
    while low <= high:
        while numbers[low] < pivot:
            low += 1
        while numbers[high] > pivot:
            high -= 1
        if low <= high:
            swap(numbers, low, high)
            low += 1
            high -= 1
    return low  # new partition position, every left is smaller than pivot, every right is bigger than pivot


def swap(numbers, i, j):
    temp = numbers[i]
    numbers[i] = numbers[j]
    numbers[j] = temp
```

1. This algorithm is **not in-place** and it requires a separate output list to form a merged sorted list. To avoid this extra space, we can use **linked list** instead of array in which case we simply need to change pointers without initializing extra space whilst also making it **in-place**.
2. Merge sort is friendly with sorting a huge number of elements even when we can't sustain this entire list in main memory. This external sorting way of merge sort is used in relational database's merge join algorithm.
3. Merge sort is **stable** in the sense that even if there are duplicates in a list, their order will be maintained in the sorted output.

2. Quick Sort:

```python
# Time Complexity: O(n log n)
# Space Complexity: O(log n) stack space for recursion + O(n) array after merge
def merge_sort(numbers, low, high):
    if low < high:
        middle = (low + high) // 2
        merge_sort(numbers, low, middle)
        merge_sort(numbers, middle + 1, high)
        merge(numbers, low, middle, high)


def merge(numbers, low, mid, high):
    i, j = low, mid + 1
    sorted_nums = []
    while i <= mid and j <= high:
        if numbers[i] < numbers[j]:
            sorted_nums.append(numbers[i])
            i += 1
        else:
            sorted_nums.append(numbers[j])
            j += 1

    while i <= mid:
        sorted_nums.append(numbers[i])
        i += 1
    while j <= high:
        sorted_nums.append(numbers[j])
        j += 1
    numbers[low:high + 1] = sorted_nums
```

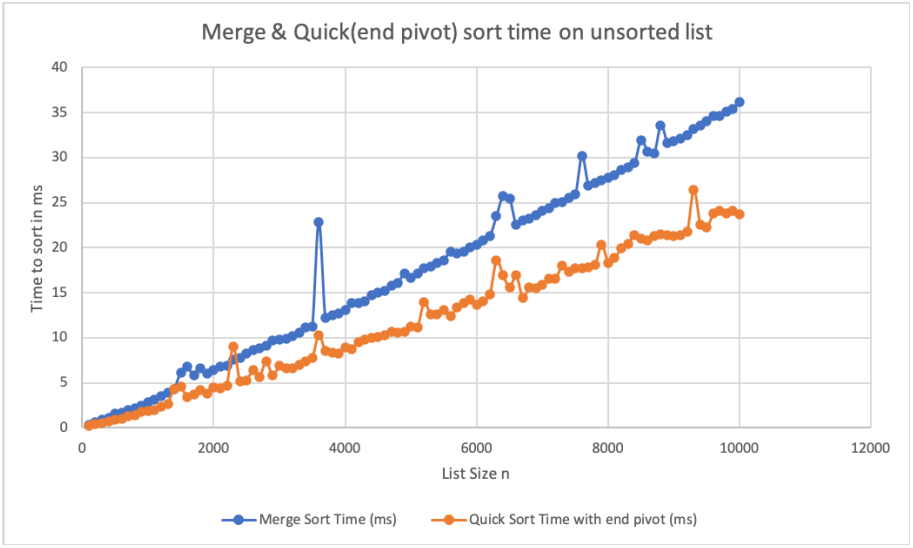
1. Quick sort is **in-place sorting algorithm** and it doesn't need any extra space other than recursion stack.
2. This algorithm is **not stable** but it can be made stable with some extra handling.
3. If pivoting doesn't divide the list in two halves in each iteration, the recursion tree becomes skewed and the time complexity grows from $O(n * log_2 n)$ to $O(n^2)$ in the worst case. This can be somewhat mitigated by choosing a random pivot in each iteration.

2: Run Time and Time Complexity Analysis of Merge Sort and Quick sort

Some really interesting insights:

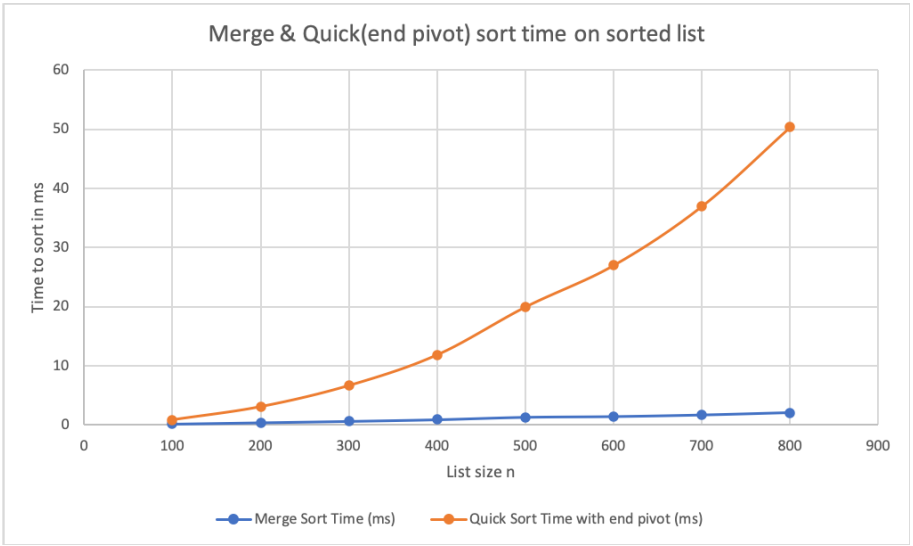
1. When input array is random (unsorted), quick sort seems to be doing consistently better than merge sort; although both exhibit $n \log n$ curves. Merge sort's extra array used is probably causing this difference. The difference would probably not be there had we merged without an extra array.

Size of list (n)	Merge Sort Time (ms)	Quick Sort Time with end pivot (ms)
100	0.298023224	0.190973282
200	0.636100769	0.406980515
300	0.905990601	0.488042831
400	1.14774704	0.761032104
500	1.588106155	0.946998596
600	1.724243164	1.035928726
700	1.931905746	1.33562088
800	2.176761627	1.409053802
900	2.486944199	1.789093018
1000	2.802848816	1.860141754
1100	3.127098083	1.991987228
1200	3.560066223	2.402067184
1300	3.918170929	2.678871155
1400	4.291772842	4.328966141
1500	6.16812706	4.562139511
1600	6.786108017	3.443956375
1700	5.807876587	3.710031509
1800	6.618738174	4.152059555
1900	6.013154984	3.765106201
2000	6.433010101	4.457950592
2100	6.756067276	4.372119904
2200	6.91485405	4.711866379
2300	7.601976395	9.016752243
2400	7.812976837	5.176067352
2500	8.252859116	5.224943161
2600	8.629798889	6.399154663
2700	8.838176727	5.67483902
2800	9.133815765	7.39812851
2900	9.747028351	5.796194077
3000	9.782075882	6.856203079
3100	9.921073914	6.603956223
3200	10.20908356	6.573200226
3300	10.58506966	7.021903992
3400	11.18063927	7.393121719
3500	11.28697395	7.786989212
3600	22.8228569	10.30087471

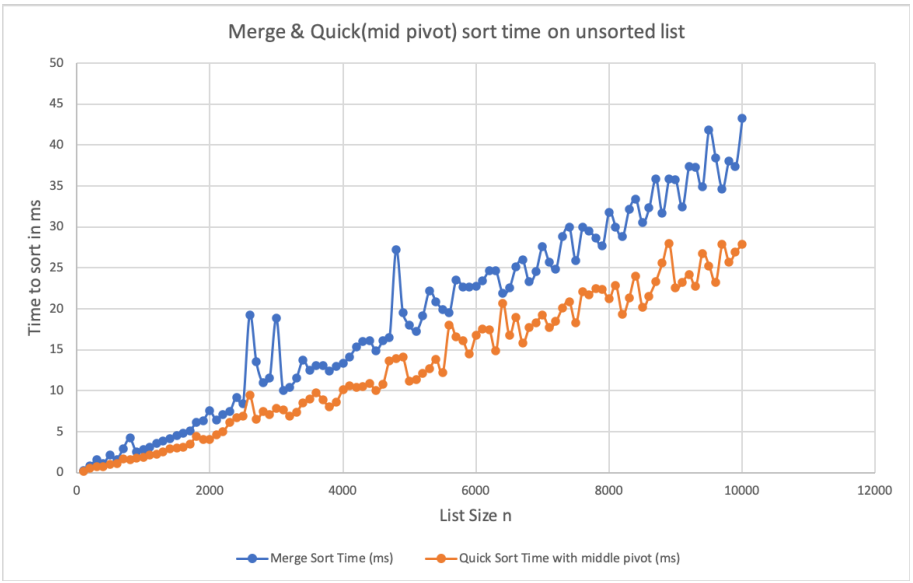


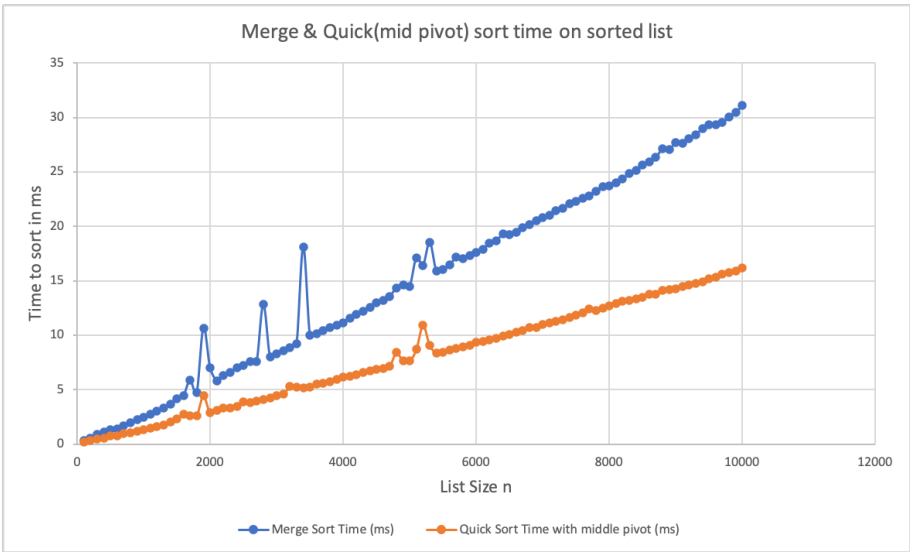
2. In the worst case when the array is already sorted and we try to run quick and merge sort on it:
   Quick sort ends up forming a skewed recursive tree due to using end element as pivot always. In each iteration, we compare nth element with every other element in the array as it is already sorted. This causes the graph to exhibit $O(n^2)$ behavior.
   At one point, where recursion stack has 997 recursive calls stacked up in it, the default recursion limit is reached and the program gives an error.
   Merge sort on the other hand always picks up middle element for dividing list into two halves. This guarantees *(n log n)* in worst case.

Size of list (n)	Merge Sort Time (ms)	Quick Sort Time with end pivot (ms)
100	0.195264816	0.832557678
200	0.412940979	3.067970276
300	0.634908676	6.64305687
400	0.901937485	11.83795929
500	1.291036606	19.92607117
600	1.410961151	26.9780159
700	1.669883728	36.96990013
800	2.048015594	50.3718853



3. To avoid quick sort's worst case behavior of $O(n^2)$, we try to pick middle element as pivot in each iteration. This way quick sort's behavior is better than merge.





3: Bubble sort correctness

```
def bubble_sort(a):
    swap, j = 1, len(a)
    # Assertion 0: a[0], ..., a[n - 1] is an array of integers with at least 1 element
    while swap == 1:
        # Assertion 1: swap = 1 && a'[j] <= ... <= a'[n - 1] && {a'[j], ...,a'[n - 1]} = {a[j], ..., a[n - 1]}
        swap = 0
        j = j - 1
        for i in range(j):
            # Assertion 2: a[i] is pointing to max a[0]...a[i]
            print(i, j)
            print(a)
            if a[i] > a[i + 1]:
                a[i], a[i + 1], swap = a[i + 1], a[i], 1
    # Assertion 3: a'[0] <= ...<= a'[n - 1] & {a'[0], ..., a'[n - 1]} = {a[0], ..., a[n - 1]}
```

Consider indexes as standard array indexes 0 *to* $n-1 => a[0, ...n-1]$

1. Assertion 0: $a[0], ..., a[n-1]$ is an array of integers

2. Assertion 1: At the start of each iteration of the while loop, the subarray a[j: n - 1] is sorted.
   $swap = 1$ && $a'[j] <= ... <= a'[n-1]$ && $\{a'[j], ..., a'[n-1]\} = \{a[j], ..., a[n-1]\}$

   During to the first iteration, the array a[n : n] is empty (j = n). That empty subarray is sorted by definition. At the end of first iteration, $j = j - 1$ array $a[n-1]$ has last element which is sorted by default as it is the single element in the array.

   Due to the inner for loop, at the end of each iteration of the outer while loop, the value at $a[j]$ is the largest value in the range $a[0 : j-1]$. Since the values in $a[j+1 : n-1]$ were sorted and were greater than the value in $a[j]$, the values in the range $a[j : n-1]$ are sorted.

   The while loop would terminate when swap is 0 or j has reached 0th index. When j = 0, we can say a[0...n-1] is sorted. If swap was 0, inner for loop would have taken care of a[0,..j - 1] to be sorted

3. Assertion 2: Every iteration in for loop, i is pointing to $max\{a[0]...a[i]\}$

   In the first iteration when $i = 0$, the first element is the maximum from array $a[0...0]$.

   In the consequent iterations, the if condition compares i^{th} element with $(i+1)^{th}$. If i^{th} element is greater than $(i+1)^{th}$, the two are swapped which makes sure that i points to the maximum element in range $a[0]...a[i]$. Swap operation makes sure that we are not deleting any elements nor adding any new elements.

   The for loop terminates when $i > j$ or $i = j + 1$ to be precise. This means that $a[i] = a[j-1]$ is the largest in range $a[0]...a[i]$.

4. Assertion 3: $a'[0] <= ... <= a'[n-1] \& a'[0], ..., a'[n-1] = a[0], ..., a[n-1]$

   As both assertion 1 & 2 hold true, at the end of outer while loop, we would have a sorted array from 0 to n - 1 which would contain no elements other than the one's it contained.

4: Bubble sort time complexity

1. In the best case when the input list is pre-sorted, above algorithm would have to do only n comparisons as it won't need to bubble up any elements nor would it have to do any swap operations. $=> O(n)$ in best case.
   In the worst case when the input list is reverse-sorted, it would have to do comparisons as $n, n-1, n-2, ..., 1$ per iteration. This would make it $=> O(n^2)$ in worst case.
   Thus the above algorithm's time complexity is $O(n^2)$.