CS 599: Algorithms **(Due: October 15th, 2020)**

Submission Homework 3

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

1. Implementation of Heap Sort

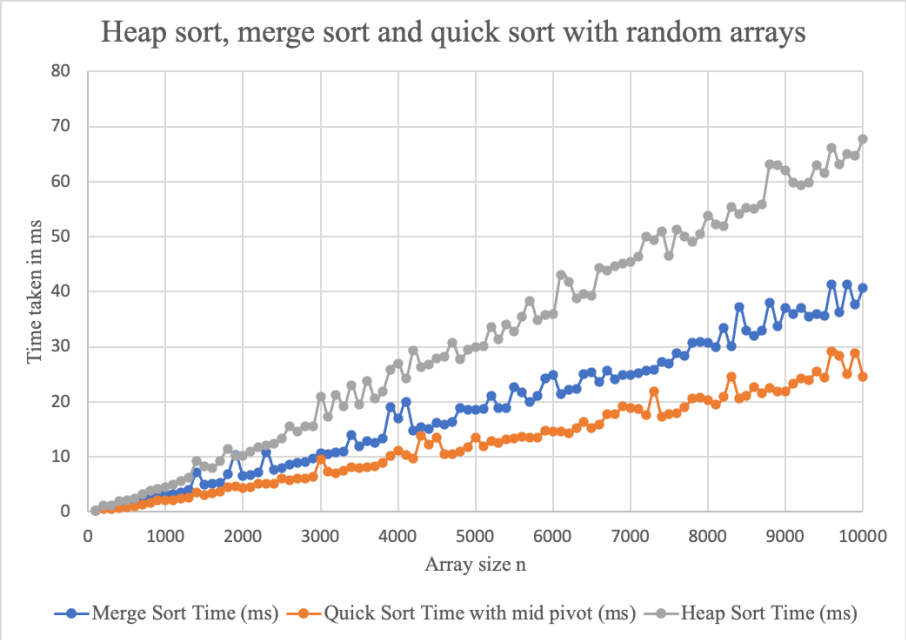
Heap Sort:

```
# Time Complexity: O(n log n) + O(n)
# Space Complexity: O(log n) for recursive stack space of max_heapify
def heap_sort(arr):
    n = len(arr)
    build_max_heap(arr, n)  # O(n)
    for i in range(n - 1, 0, -1):  # delete n elements one by one from top, heapify
        arr[i], arr[0] = arr[0], arr[i]
        max_heapify(arr, i, 0)


# assumes part of heap is sorted: log(n)
def max_heapify(arr, n, i):  # log n
    maximum = i
    left = (2 * i) + 1
    right = (2 * i) + 2
    if left < n and arr[i] < arr[left]:
        maximum = left
    if right < n and arr[maximum] < arr[right]:
        maximum = right
    if maximum != i:
        arr[i], arr[maximum] = arr[maximum], arr[i]
        max_heapify(arr, n, maximum)


# build max heap from unsorted array: O(n)
def build_max_heap(arr, n):
    for i in range(n // 2 - 1, -1, -1):
        max_heapify(arr, n, i)
```

2: Merge Sort vs Quick Sort vs Heap Sort

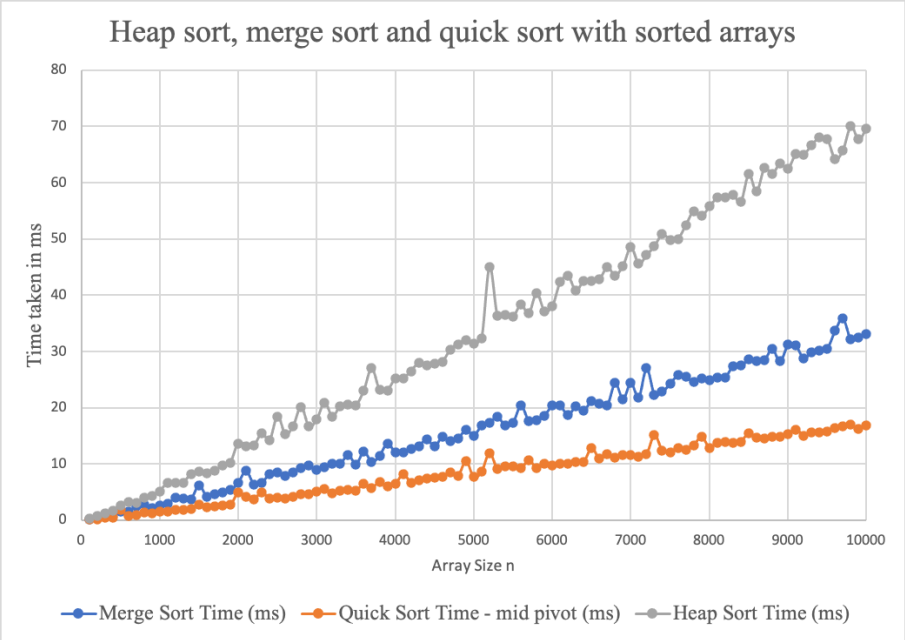
Theoretically, all three algorithms merge sort, quick sort and heap sort exhibit similar time complexity of $O(n\ logn)$. However, there are constant factor differences in the run-times of these algorithms and some of them perform worse than others in the worst case. Of course there can be multiple modifications to a particular algorithm to overcome its drawbacks. Here, we are comparing:
1. traditional recursive merge sort algorithm
2. quick sort that chooses middle element as pivot (a modification to handle worst case $O(n^2)$ on sorted array input)
3. heap sort that builds max heap using heapify() rather than inserting element one by one to construct max heap (a modification to build max heap with $O(n)$ rather than $O(n\ logn)$).

1. Case 1: Performance when input array is random (unsorted)



In the average case, heap sort seems to be the slowest of all three. The reason this happens is because in this algorithm, we build the max-heap first and then go on deleting max element one by one followed by adjusting max-heap property. In short, for the same input array, we perform the most number of comparisons and swaps in heap sort than that of quick and merge.
Quick sort version that chooses middle element as pivot seems to be the fastest of all three. It is better than merge sort as it is in-place and it doesn't need creation of new array for sorted output. This makes sure not to add to the time for creating new array and appending to it.

2. Case 2: Performance when input array is sorted in ascending order:



Even in the worst case, the three algorithms seem to exhibit similar pattern for timing. Heap sort being the slowest, merge sort in the middle and quick sort being the fastest. The timing difference factor between these three however increases because as the input size increases, all three algorithms have to deal with more elements. Heap sort needs to do lot many comparisons and swaps than the other two. Merge sort requires sufficient space to create an extra output array as it isn't in place sorting algorithm.

3: k'th element in the merged array

Since the input arrays are sorted in ascending order, we can exploit this property to optimize over the brute force of having to merge both arrays until the kth element is reached. This brute force would become $O(m+n)$ in the worst case where m is the length of $array1$ and n is the length of $array2$.

```
def find_kth(list1, list2, start1, end1, start2, end2, k):
    # Edge case 1: k greater than size of list1 + list2
    if k > len(list1) + len(list2):
        print("Invalid k: not in range of combined lists")
        return -1

    # Edge case 2: If list1 is exhausted, kth element is the kth in list2
    if start1 > end1:
        return list2[start2 + k - 1]

    # Edge case 3: If list2 is exhausted, kth element is the kth in list1
    if start2 > end2:
        return list1[start1 + k - 1]

    # Core Logic: calculate mid elements of list1 and list2
    mid1 = (start1 + end1) // 2
    mid2 = (start2 + end2) // 2

    # if k is less than number of elements in list1[start1...mid1] and list2[start2...mid2] combined,
    # the kth element lies in range and so we halve search space by manipulating mids, moving to right halves
    if (mid1 - start1 + 1 + mid2 - start2 + 1) > k:
        if list1[mid1] > list2[mid2]:  # get rid of right half of list1
            return find_kth(list1, list2, start1, mid1 - 1, start2, end2, k)
        else:  # get rid of right half of list2
            return find_kth(list1, list2, start1, end1, start2, mid2 - 1, k)

    # k is not within the range of combined list1[start1...mid1] and list2[start2...mid2],
    # we shift the search space by halving lists and manipulating k
    else:
        if list1[mid1] > list2[mid2]:  # move to right half of list2, change k by factor of shift
            return find_kth(list1, list2, start1, end1, mid2 + 1, end2, k - (mid2 - start2 + 1))
        else:  # move to right half of list1, change k by factor of shift
            return find_kth(list1, list2, mid1 + 1, end1, start2, end2, k - (mid1 - start1 + 1))
```

(a) 1. the algorithm can be designed to run recursively so as to reduce the search spaces into halves with each recursive call. To avoid creation of new subarrays during recursive calls, we handle entire execution with start, mid, and end indexes.
2. start by taking midpoints of the two arrays, mid1 and mid2.
3. if the sum of number of elements up to the corresponding mids is greater than k, then we can discard right half of one of the arrays as k is smaller. We choose to safely discard the right half of array which has larger mid element as our kth element won't go higher than that.
4. we are left with the condition where k is less than or equal to the sum of number of elements up to the corresponding mids. This means that k may lie towards the right halves now. We compare the elements at mid and whichever array has smaller element at mid index, we safely discard the left half of that array. Since this array's left elements will be small to be considered for kth position. We need to update k by the number of elements we discarded in the process since we moved to the right.
5. whenever one of the arrays is exhausted while discarding its half parts, we know that the kth element lies at kth index in the other array.
6. in short, we try to exploit the sorted array property to eliminate one half of the two arrays during every recursive call. To decide which half to discard, we use conditions to safely say that kth element won't lie in

this half.

(b) During each recursive call, we are reducing one of the two arrays in half. This means that an array of length x will be reduced $log\ x$ times until it's completely exhausted.
In the best case scenario, we may keep reducing search space of one of the arrays until it is completely exhausted. $\mathrm{m} \rightarrow \mathrm{m}/2 \rightarrow \mathrm{m}/4 \rightarrow \mathrm{m}/8...$ or $\mathrm{n} \rightarrow \mathrm{n}/2 \rightarrow \mathrm{n}/4 \rightarrow \mathrm{n}/8...$ This would make the time complexity of $O(log\ m)$ *or* $O(log\ n)$ depending on whichever was completely reduced.
In the worst case, we may have to reduce both arrays entirely until kth element is found. These search space reductions would occur one after the other in any order; making it $O(log_2\ m) + O(log_2\ n) \rightarrow O(log_2\ m + log_2\ n)$.
In the designed divide and conquer algorithm, we do not modify the input arrays and our base cases take care of the program's termination (too vague an assertion but I found it cumbersome to prove the whole thing).