**CS 514**

# **Quiz 2 - Divide and Conquer, Heap and Priority Queue**

**Q1. Perform Quick Sort on the following array, taking the pivot as the middle element every time. Show the intermediate output after each timestep**

**arr = [2, 1, 5, 3, 4, 6]**

**Solution -**

**Code –**

def partition(arr,left,right):  
 i = left  
 j = right  
 print("Array indexes considered = ",i,j)  
 pivot = arr[(left + right)//2]  
 print("Pivot Element = " ,pivot)  
 while (i<=j):  
 while arr[i] < pivot:  
 i += 1  
 while arr[j] > pivot:  
 j -= 1  
 print("Intermediate Array(before sorting) =", arr)  
 arr[i], arr[j] = arr[j], arr[i]  
 print("Intermediate Array(after sorting) =",arr)  
 if (i>=j):  
 return j  
  
def qsort(input\_array, low, high):  
 if low < high:  
 p\_index = partition(input\_array, low, high)  
 if (low < p\_index - 1):  
 qsort(input\_array,low,p\_index -1)  
 if p\_index < high:  
 qsort(input\_array,p\_index+1,high)  
 return input\_array  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 arr = [2,1,5,3,4,6]  
 n = len(arr)  
 print(qsort(arr,0,n-1))

**Output of the code -**

Array indexes considered = 0 5

Pivot Element = 5

Intermediate Array(before sorting) = [2, 1, 5, 3, 4, 6]

Intermediate Array(after sorting) = [2, 1, 4, 3, 5, 6]

Intermediate Array(before sorting) = [2, 1, 4, 3, 5, 6]

Intermediate Array(after sorting) = [2, 1, 4, 3, 5, 6]

Array indexes considered = 0 3

Pivot Element = 1

Intermediate Array(before sorting) = [2, 1, 4, 3, 5, 6]

Intermediate Array(after sorting) = [1, 2, 4, 3, 5, 6]

Intermediate Array(before sorting) = [1, 2, 4, 3, 5, 6]

Intermediate Array(after sorting) = [1, 2, 4, 3, 5, 6]

Array indexes considered = 1 3

Pivot Element = 4

Intermediate Array(before sorting) = [1, 2, 4, 3, 5, 6]

Intermediate Array(after sorting) = [1, 2, 3, 4, 5, 6]

Intermediate Array(before sorting) = [1, 2, 3, 4, 5, 6]

Intermediate Array(after sorting) = [1, 2, 3, 4, 5, 6]

Array indexes considered = 1 2

Pivot Element = 2

Intermediate Array(before sorting) = [1, 2, 3, 4, 5, 6]

Intermediate Array(after sorting) = [1, 2, 3, 4, 5, 6]

[1, 2, 3, 4, 5, 6]

Process finished with exit code 0

**Explanation –**

The quicksort code given considers the pivot as the middle element of the subset of the array always. In the first iteration, the whole array , i.e. [2,1,5,3,4,6] is considered, which is from index 0 to 5. The pivot here will be thee middle element - 5(index – 2) and the list is sorted wrt the pivot, that is all the elements lesser than 5 are moved to left side of the 5 and all the elements larger than 5 will go on the right of 5.

The we recursively sort the left and the right sub lists taking the middle element as pivot. The intermediate arrays will be –

Input array - [2, 1, 5, 3, 4, 6]

Pivot = 5, After sorting - [2, 1, 4, 3, 5, 6]

Array considered – left sublist of previous array – [2,1,4,3]

Pivot – 1, After sorting - [1, 2, 4, 3, 5, 6] where the sorted part has all elements greater than 1 on the right side of 1.

Array considered – right sublist of the pivot 1 subarray – [2,4,3]

Pivot – 4, After sorting - [1, 2, 3, 4, 5, 6] where the sorted part has all elements greater than 1 on the right side of 1.

Array considered – right sublist of pivot 4 subarray – [2,3]

Pivot – 2, After sorting - [1, 2, 3, 4, 5, 6] where these 2 elements are sorted.

Final Arrray – [1, 2, 3, 4, 5, 6]

**Q2. Consider a Divide and Conquer algorithm that divides each problem into 2 sub-problems of size 3n/4 each. Write the recurrence relation for this algorithm, and compute the time complexity using Master Theorem.**

**Answer –**

As the algorithm divides each problem into 2 sub problems of size 3n/4 each always, the recurrence relation T(n) of the algorithm will be of the form

T(n) = 2T(3n/4) + f(n)

Here as we are not sure what is f(n), we derive time complexity of the recurrence relation assuming 3 cases of f(n) – O(1), O(n) and O(n2)

1. **f(n) = O(1)**

Now, T(n) = 2T(3n/4) + O(1)

We know the Master theorem general form is

T(n) = aT(n/b) + f(n)

where,

a is the number of subproblems,

b is the factor by which the problem size is reduced, and

f(n) is the time complexity outside the recursive calls.

Here, a = 2, b = 4/3 and f(n) = O(1)

For deriving the time complexity, we compare f(n) to n^(logb (a)).

As O(1) is present,

We get – n^0 = n^(logb (a) + ε)

So, logb (a) + ε = 0 => log4/3 2 + ε = 0

On solving this equation we get ε = log34 which is > 0 ( Case 1 of Master theorem)

So T(n) =. Θ(n^( logb (a)).

**So, T(n) = Θ(n^( log4/3 (2)).**

1. **f(n) = O(n)**

Now, T(n) = 2T(3n/4) + O(n)

We know the Master theorem general form is

T(n) = aT(n/b) + f(n)

where,

a is the number of subproblems,

b is the factor by which the problem size is reduced, and

f(n) is the time complexity outside the recursive calls.

Here, a = 2, b = 4/3 and f(n) = O(n)

Now, we compute n^ (logb (a)).

n^ (log4/3 (2)) = n^(log(2)/log(4/3))

Compare f(n) with n^ (logb (a)):

As n^(logb (a)) grows with n, this case falls into case 2 of the Master theorem.

Therefore, the time complexity is

**T(n) = Θ((n^( logb (a))\* log n) = Θ((n^( log4/3 (2))\* log n)**

1. **f(n) = O(n2)**

Now, T(n) = 2T(3n/4) + O(n2)

We know the Master theorem general form is

T(n) = aT(n/b) + f(n)

where,

a is the number of subproblems,

b is the factor by which the problem size is reduced, and

f(n) is the time complexity outside the recursive calls.

Here, a = 2, b = 4/3 and f(n) = O(n2)

Now, we compute n^ (logb (a)).

n^ (log4/3 (2)) = n^(log(2)/log(4/3))

Compare f(n) with n^ (logb (a)):

As f(n) = O(n2) and n^(logb (a)) grows faster than n2, this case falls into case 3 of the Master theorem.

Therefore, the time complexity is

**T(n) = Θ(f(n)) = Θ(n2)**

**Q3. Given a sequence of numbers: 19, 6, 8, 11, 4, 5**

1. **Draw a binary min-heap (in a tree form) by inserting the above numbers and reading them from left to right.**

**b) Show a tree that can be the result after the call to deleteMin() on the above heap.**

**c) Show a tree after another call to deleteMin().**

Solution – given in the pdf attached

**Q4. What is the big-Oh time complexity of getting a sorted array out of a max heap? Justify your answer.**

The removal of the maximum element from a max heap is a key operation in transforming it into a sorted array. This operation involves two main steps: locating and removing the maximum element, followed by restoring the heap property.

a. **Removal of Maximum Element (O(log(n)))** - The process of finding and removing the maximum element is a logarithmic operation, denoted as O(log(n)). This is because the maximum element is located at the root of the heap, and to remove it while maintaining the heap structure, the algorithm needs to traverse the height of the heap, which is log(n) in the worst case.

b. **Re-balancing the Heap (O(log(n))** - After removing the maximum element, the heap needs to be re-balanced to maintain its properties. The re-balancing process, which involves moving elements to their correct positions, also has a time complexity of O(log(n)).

c. **Total Time Complexity (O(nlog(n)))** - Since you repeat this removal process for all n elements in the heap, the overall time complexity becomes O(n \* log(n)), where n is the number of elements in the heap.