# DAA Assignment 2

**2. Practice Solving Recurrences**

(a) T(n) = 4T(n/2) + $n^{2.1}$

From the above recurrence equation we can say that a = 4, b = 2 and f(n) = $n^{2.1}$ and n^log$_b$$^a$ = n^log$_2$$^4$ = $n^2$. Choosing ε= 0.1 we get f(n) = $\Omega(n^{2+\varepsilon})$. Hence Masters theorem case 3 applies if we can show that the regularity condition holds for f(n).
i.e. af(n/b) <= cf(n) for some constant c < 1 and $n_0$>= 0 for all n >= $n_0$.

=> 4*$(n/2)^{2.1}$<= c * $n^{2.1}$

=> (4/2$^{2.1}$) * $n^{2.1}$<= c * $n^{2.1}$

Therefore the above statement is true for 1 > c >= (4/2$^{2.1}$).

Hence T(n) = $\Theta(n^{2.1})$.

(b) T(n) = 3T(n/3) + 4T(n/3) + $n^2$

We can rewrite the above recurrence equation as T(n) = 7T(n/3) + $n^2$.
From the above, we can say that a = 4, b = 3 and f(n) = $n^2$and n^log$_b$$^a$ = n^log$_3$$^7$ = $n^{1.77}$ which is less than $n^2$. Choosing ε= 0.23 we get f(n) = $\Omega(n^{1.77+\varepsilon})$. Hence Masters theorem case 3 applies if we can show that the regularity condition holds for f(n).
i.e. af(n/b) <= cf(n) for some constant c < 1 and $n_0$>= 0 for all n >= $n_0$.

=> 7*$(n/3)^2$<= c * $n^2$

=> (7/3$^2$) * $n^2$<= c * $n^2$

=> (7/9) * $n^2$<= c * $n^2$

Therefore the above statement is true for 1 > c >= (7/9).

Hence T(n) = $\Theta(n^2)$.

(c) T(n) = 3T(n/2) + $n^{7/3}$

From the above recurrence equation, we can say that a = 3, b = 2, and f(n) = $n^{7/3}$ and n^log$_b$$^a$ = n^log$_2$$^3$ = $n^{1.58}$which is less than $n^{7/3}$. Choosing ε= 0.75 we get f(n) = $\Omega(n^{1.58+\varepsilon}$ ). Hence Masters theorem case 3 applies if we can show the regularity condition holds for f(n).

i.e. af(n/b) <= cf(n) for some constant c < 1 and $n_0$>= 0 for all n >= $n_0$.

=> 3*$(n/2)^{7/3}$<= c * $n^{7/3}$

=> (3/$2^{7/3}$) * $n^{7/3}$ <= c * $n^{7/3}$

=> (3/$2^{2.33}$) * $n^{7/3}$ <= c * $n^{7/3}$

Therefore the above statement is true for 1 > c >= (3/$2^{2.33}$).

Hence T(n) = $\Theta(n^{7/3})$.

**3. Stack depth of quicksort**

(a) Argue that TAIL-RECURSIVE-QUICKSORT(A, 1, A.length) correctly sorts the array A.

Let us prove this by induction.

**Base Case:** If array A contains only one element, then p=r, and the algorithm terminates considering the only single element present is sorted.

**Inductive Step:** Let us assume that for 1 ≤ *k* ≤ *n*, TAIL-RECURSIVE QUICKSORT correctly sorts an input array A containing k elements. Therefore, for an array having *n* + 1 elements and *q* as the pivot, TAIL-RECURSIVE-QUICKSORT correctly first sorts the left subarray A[1…q] by the induction hypothesis, since the left subarray has definitely fewer elements than *n* + 1. After that, *p* is updated in Line 5: *p* = *q* + 1, and the exact same sequence of steps are followed as a call is made to TAIL-RECURSIVE-QUICKSORT(A, q+1, n). Since the right side of the array, A[q+1...n] is of a strictly smaller size, the algorithm correctly sorts it by the induction hypothesis.

From the above two steps in the induction process, we can say that TAIL-RECURSIVE-QUICKSORT(A, 1, A.length) correctly sorts the array A.

(b) Describe a scenario in which TAIL-RECURSIVE-QUICKSORT's stack depth is Θ(n) on an n-element input array.

The stack depth will be Θ(n) if the input array is already sorted. The right subarray will always have size 0 so there will be *n* − 1 recursive calls before the while-condition p < r is violated.

(c) Modify the code for TAIL-RECURSIVE-QUICKSORT so that the worst-case stack depth is Θ(log(n)). Maintain the O(nlogn) expected running time of the algorithm.

Modifying the algorithm to make the recursive call on the smaller subarray to avoid pushing too much on the stack. In every recursion, the range of the array will be reduced at least by half, limiting the number of recursive calls to O(log(n)). The following variation of TAIL-RECURSIVE-QUICKSORT checks which of the two subarrays returned from PARTITION is smaller and recursively executes on this subarray. Since the array size is reduced by at least half in each recursive call, the stack depth is in the worst-case O(log(n)). The expected running time is not affected, because exactly the same work is done as before, i.e. the same partitions are produced, and the same subarrays are sorted.

```
MODIFIED-QUICKSORT(A, p, r)
1: while p < r do
2:        q = PARTITION(A, p, r)
3:        if q ≤ ⌊(r – p)/2⌋ then
4:            MODIFIED-QUICKSORT(A, p, q-1)
5:            p = q + 1
6:        else
7:            MODIFIED-QUICKSORT(A, q+1, r)
8:            r = q − 1
9:        end if
10: end while
```

**4. Practice algorithm design and the use of data structures.**

**Approach and Textual Description**

Here in the above problem, we have few important points to make note of.

- The output timestamps should be in sorted order.
- The input is a continuous stream and does follow an order i.e earlier quotes may arrive after the later quotes because of the differences in server loads and network traffic routes. So in order to calculate a minimum or maximum out of a continuously changing input heap data structure is the best option.
- Every time-stamp (integer) in the stream is at most a hundred positions away from its correctly sorted position. So a heap of size 101 should be able to solve our problem.

Based on the above considerations we can come up with an algorithm that makes use of heap data structure. The heap should be a min-heap, as we have to output the timestamp in the increasing order.

Since every timestamp is located at most a hundred steps far from its correct position, a min-heap of size 101 should contain all the sorted elements in heapified order with the minimum element at the root node. Initially, we keep on inserting the input stream elements until it reaches above 100. After that, we can start extracting minimum to the output all the time when the size > 100 is satisfied. When the size <= 100, we stop extracting elements and wait until the heap is full above 100 for the next time and we repeat the same process. Once the streaming has stopped in the end we pull out all the elements in the heap one by one emptying it, so that we get all the elements in sorted order. Thus, The memory is independent of the number of timestamps being processed.

**Algorithm Pseudocode**:

```
// Method to min heapify the MinHeap.
minHeapify(stream, currentIdx, endIdx)
    while(currentIdx <= endIdx) {
        leftChild = currentIdx * 2 + 1
        rightChild = currentIdx * 2 + 2
        Idx = currentIdx
        if(leftChild <= endIdx and stream[leftChild] < stream[currentIdx]) {
            idx = leftChild
        }
        if(rightChild <= endIdx and stream[rightChild] < stream[idx]) {
            idx = rightChild
        }
        if(idx != currentIdx) {
            exchange stream[idx], stream[currentIdx]
            currentIdx = idx
        }
        else {
            return
        }
    }

// Method to shitUp the element inserted into the MinHeap.
shiftUp(stream)
    currentIdx = stream.length - 1
    parentIdx = (currentIdx - 1) / 2
    while(currentIdx > 0 and stream[parentIdx] > stream[currentIdx]) {
        exchange stream[parentIdx], stream[currentIdx]
        currentIdx = parentIdx
        parentIdx = (currentIdx - 1) / 2

// Method to extract minimum element from the MinHeap.
```

```
extract_min(stream)
        exchange stream[0], stream[stream.length - 1]
        valueToRemove = stream.pop()
        minHeapify(stream, 0, stream.length)
        return valueToRemove

// Method to insert an element into MinHeap.
min-heap-insert(stream, timestamp)
        stream.add(timestamp)
        shiftUp(stream)

// Main algorithm that handles the entire logic of getting the timestamps in sorted order.
count=0
while(True){
        // logic to handle the timestamp arriving case.
        if(timestamp is arriving){
                min-heap-insert (stream, timestamp)
                count += 1
                if(count > 100) {
                        extract_min(stream)
                        count -= 1
                }
        }
        // logic to handle once all the timestamps arriving is finished.
        if(timestamp arrival is finished){
                while(count > 0){
                        extract_min(stream)
                        count -= 1
                }
                break
        }
}
```

**Correctness of the Algorithm:**

**Loop invariant condition:** An element A which is in the output is always the smallest element among all other 100 elements in the stream and any future element.

**Initialization**: Before the execution of min-heap-insert, there is no element in the heap to output.

**Maintenance**: At the time an element A is in output, it is the smallest among all 101 in min-heap. Assume, at any point in time in the future an element B < A arrives. Because B must be at most 100 steps behind from its correct position, this violates the first sentence: A is the smallest among all 101 elements in the stream. Thus, whenever A is in output, it is smaller than all other 100 elements in the stream and any future element. Maintenance of the loop invariant guarantees that time stamps are outputed in the correct order which is ascending in nature.

**Termination**: After the termination, all the elements are in the output are in sorted order.

**Time Complexity:**
Since heap size remains constant (i.e <=101 all the time) all insert and extract operations use O(1) time per operation.
**Space Complexity:**
Since we maintain a heap of constant size (i.e <=101 all the time), the space complexity of this implementation is also O(1).

**Example:**

An example of maximum element size = 3.
Input stream: 15, 16, 23, 0, 45, 95

| Stream Input | Min-Heap | Output |
|---|---|---|
| 15 | 15 | - |
| 15, 16 | 15, 16 | - |
| 15, 16, 23 | 15, 16, 23 | - |
| 15, 16, 23, 0 | 0, 15, 16, 23 | 0 |
| 15, 16, 23, 0, 45 | 15, 16, 23, 45 | 0, 15 |
| 15, 16, 23, 0, 45, 95 | 16, 23, 45, 95 | 0, 15, 16 |
| - | 23, 45, 95 | 0, 15, 16, 23 |
| - | 45, 95 | 0, 15, 16, 23, 45 |

| | | |
|---|---|---|
| - | 95 | 0, 15, 16, 23, 45, 95 |