Data Structures - HW2

November 20, 2021

Question 1:

1.a

Assuming 0 based indexing to the array, it can represented as follows:

- parent of i = $(i-1)/d$
- children k of parent i where $k = \{1, 2...d\}$= d*i+k

1.b

Solution: The height of a d-heap with n nodes is: $h = log_d n$

1.c

The EXTRACT-MAX algorithm for d-ary heap is similar to the binary one. the difference is the number of children needed to be compared when bubbling down. In the case of d-ary it's d children.

The implementation is as follows:

1. extract the root element which is the max value in the heap and exchange it with the rightmost leaf in the heap.
2. bubble down the root element to its right place by comparing it to its d children. switch between the element and the largest child from the d children.
3. keep doing so until the element is larger than all of his children or when it becomes a leaf.

The time complexity is d comparison * the height of the tree $d * log_d(n) \rightarrow O(dlog_d(n))$

1.d

Once again the INSERT algorithm is similar to the binary one, the implementation is as follows:

1. insert the element into the next available slot

2. bubble up the element, if the element is bigger that its parent switch between the two.

3. do so until the element is smaller than its parent or until you reach the root.

The time complexity is the height of the tree, as the number of children the node has is irrelevant, thus: $log_d(n) \rightarrow \theta(log(n))$

Question 2:

2.a

Basically we'll do a linear search:

1. we iterate over the array

2. if we find x or x^2 will save its location and continue to search x^2or x respectively.

3. if we find x^2or x, depending on what we found in section 1, will save its location, stop and return both locations.

4. if we reached the end of the array we return False or indexes -1,-1 as some indication that x, x^2 weren't found.

The worst-case time complexity is O(n), when the pair couldn't be found or one of them is in the end of the array.

2.b

the algorithm is as follows:

1. first sort the array using MergeSort.

2. declare a counter = 1.

3. ran over each element in the sorted array, if two consecutive element are different raise the counter by 1.

4. return the counter.

the time complexities for section 1 and section 2 are $O(nlog(n))$, $O(n)$ respectively, thus the total time complexity is $O(nlog(n))$, this will be the worst and best case run time.

2.c

The algorithm is as follows:

1. first sort the array using MergeSort.

2. declare max_counter, counter = 1.

3. ran over each element in the sorted array,
   (a) raise the counter by 1 while two consecutive elements are equal.
   (b) if the two consecutive elements are different and counter > max_counter then max_counter = counter and reset counter = 1
4. return max_counter.

The sorting part requires $O(nlog(n))$ time in the worst case and running over the array will take $O(n)$, resulting in a total complexity time of $O(nlog(n))+O(n) = O(nlog(n))$.

2.d

The algorithm is as follows:

1. first sort the array using MergeSort.
2. iterate over the sorted array and for each element x do binary search for x^2
3. if the element is found, we stop the iterations and return True, otherwise we continues until the end of the array.
4. if we reached to the end of the array, we return False.

The time complexity comprises of sorting the array + iterating all elements * binary search for each elements = $O(nlog(n))+O(n)*O(log(n)) = O(nlog(n))$.

Question 3:

3.a

2	3	4	12
5	8	9	14
16	∞	∞	∞
∞	∞	∞	∞

3.b

Assume $Y[1,1] = \infty$. consider $Y[i,j]$ for any i and j. By the property of a Young tableau, $Y[i,j] \geq Y[1,j] \geq Y[1,1]$. Therefore,$Y[i,j] = \infty$. This means that the Young tableau is empty because the above is true for every i and j.

Now assume $Y[m,n] < \infty$. Consider $Y[i,j]$ for any i and j. By the property of a Young tableau,$Y[i,j] \leq Y[m,j] \leq Y[m,n]$. Therefore,$Y[i,j] < \infty$. This means that the Young tableau is full because the above is true for every i and j.

3.c

The main idea behind the algorithm is that it removes the first element and replace it with ∞. Then it percolate down the ∞ until the tableau is sort correctly. In each step of the percolation we need switch the ∞ with the smaller of the adjacent elements, the bottom or the right element.

let's define a function to do the percolation:

```
def fixY(Y, i=1, j=1): {
# get the values present at the bottom and right cell of the current cell.
# in case we've reached Y's bounds,∞ will be assigned.
    if (i < M):
        bottom = Y[i + 1][j]
    else:
        bottom =∞
    if (j < N)
        right = Y[i][j + 1]
    else:
        bottom = ∞
    if bottom == ∞ and right == ∞:
    return. # we have reached to the end of Y
    if bottom < right: # go down
    # swap ‘Y[i][j]‘ with ‘Y[i + 1][j]‘
        temp = Y[i][j]
        Y[i][j] = Y[i + 1][j]
        Y[i + 1][j] = temp
        fixY(Y, i + 1, j)
    else:
    # go right # swap ‘Y[i][j]‘ with ‘Y[i][j + 1]‘
        temp = Y[i][j]
        Y[i][j] = Y[i][j + 1]
        Y[i][j + 1] = temp
        fixY(Y, i, j + 1)
    }
```

Now let's define a extract the next minimum element from Y

```
def EXTRACT-MIN(Y):

    # the first cell of the tableau stores the minimum element.
        min = Y[1][1]
        # make the first element as infinity
        Y[1][1] = ∞
        # fix Y
        fixY(Y)
        return min
```

In order to prove correctness of the algorithm we will show that in each percolation down fixY(i, j) every element in the tableau uphold $Y[i', j'] \leq Y[i'+1, j']$ and $Y[i', j'] \leq Y[i', j'+1]$ expect $Y[i, j]$.

we'll prove it by induction using z=i+j.

Base case: i=1, j=1, z=2:

In this case Y was Young tableau before the element change and calling Y(1,1) thus it's true.

Inductive hypothesis: Let's assume the claim is true for z=i+j , Y[i, j] and prove for z=z+1

z= z+1 may result from one of the following cases:

1. going down:
   (a) z= (i+1)+j, this case occurs when Y[i+1, j] < Y[i, j+1]
2. going right:
   (a) z= i+(j +1), this case occurs when Y[i+1, j] ≥ Y[i, j+1]

From the inductive hypothesis, in both cases, when Y[i, j] was called, the only element that breached the tableau rule was Y[i, j]. However, when the next percolation step occurs (z=z+1), Y[i, j] is switched with either Y[i+1, j] in case of case 1 or with Y[i, j+1] in case of case 2. we need to insure that after the switch $Y[i, j] \leq Y[i+1, j]$ and $Y[i, j] \leq Y[i, j+1]$, but this is true because of the switches and the definition of each case.

Thus only element that is violating the tableau is the 'z+1' element.

Running time:

Starting with Y of size MxN, in each percolation we either go down or go right and continue in solving either an $(m-1)\times n$ or an $m\times(n-1)$ problem. we continue until Y[i, j] is smaller than its neighbors or we've reached the end of the tableau.

Let's z = m+n, in each percolation we do a switch between cells and "reduce" the problem to a m+n-1 problem, thus we do it z times as max:

T(z)=T(z−1)+O(1)=T(z−2)+O(1)+O(1)=....=O(z) = O(n+m)

3.d

The main idea behind the algorithm is that we place the element we want to insert in the bottom-right corner of the tableau. then we percolate up the element by swapping it with largest of Y[i-1, j] and Y[i, j-1]. we continue to do so until the adjacent left or above cell are smaller than the element or we've reach the beginning of the tableau

```
INSERT(Y, element)

    Y[m,n]=element
    PERCOLATE_UP(Y, m, n)
```

```
PERCOLATE_UP(Y, i, j)
    max_i=i
    max_j=j
    if i-1 ≥ 1 and Y[i, j] < Y[i-1, j]
        max_i=i-1
        max_j=j
    if j-1 ≥ 1 and Y[max_i, max_j] < Y[i,j-1]
        max_i=i
        max_j=j-1
    if max_i ≠ i or max_j ≠ j
        temp = Y[i, j]
        Y[i, j] = Y[max_i, max_j]
        Y[max_i, max_j] = temp
    PERCOLATE_UP(Y, max_i, max_j)
```

Similar to explanation for the run time of EXTRACT-MIN in section 3.c we remove at each percolation step one column or one row so at most we have O(n+m) iterations.

3.e

Given n^2 numbers, we can insert them using INSERT. this will result in n^2 inserts which will take $n^2 * (n + n) = O(n^3)$. After that, we can perform EXTRACT-MIN n^2 times with a time complexity of O(n+n) which will result with sorted elements with a total time complexity of $n^2 * (n + n) = O(n^3)$.

Question 4:

4.a

we can address this question by looking on the exponent of the computers memory. we do $log_2(2^k)$ for each of the elements. we will end up with an array of size n elements, ranging for 1 to k where k $=\{1, 2...n\}$. Changing of n elements will have a run time of $O(n)$. Then, we can use counting algorithm with time complexity of O(n+k) = O(2n) = O(n). Then we need to iterate the array once again to restore the original values, this will have a run time if O(n). The total running time will be $O(n) + O(n) + O(n) = O(n)$.

4.b

1. we create two array, one for the integers and one for the non-integers.
2. we copy the integers to the first array which will be the size of a least $n - 3\frac{n}{log(n^2)} = O(n)$
3. we copy the non-integers to the second array which will be the size of $3\frac{n}{log(n^2)} = O(n)$

4. we sort the integer array with count sort with time complexity of $O(n - 3\frac{n}{log(n^2)}) = O(n)$, for all n>1
5. we merge sort the non-integers array with time complexity of $mlog(m)$,with $m = 3\frac{n}{log(n^2)}$
6. we ran on both arrays and do linear sort between the two, back to the original array $= O(n)$

now let's show that $mlog(m) = O(n)$

$$mlog(m) = \frac{3n}{log(n^2)} log(3\frac{n}{log(n^2)})$$

$$\frac{3n}{log(n^2)} log(3\frac{n}{log(n^2)}) = \frac{3n}{log(n^2)} log(3n) - \frac{3n}{log(n^2)} log(log(n^2)) =$$

$$\frac{3n}{2log(n)} log(3) + \frac{3n}{2log(n)} log(n) - \frac{3n}{log(n^2)} log(log(n^2)) =$$

$$\frac{3n}{2log(n)} log(3) + \frac{3n}{2} - \frac{3n}{log(n^2)} log(log(n^2)) \leq 3n + 3n - \frac{3n}{log(n^2)} log(n^2) =$$

$$O(3n) + O(3n) - O(3n) = O(n)$$

So the sorting time complexity is $O(n)$.
Thus, the total time of the algorithm is $5 * O(n) = O(n)$.

Question 5:

$T(n) = T(\frac{4n}{5}) + T(\frac{n}{5}) + O(n)$.

each partition step will split the array into two parts, one will have roughly $\frac{1}{5}$ of the elements and the larger part which will have $\frac{4}{5}$. so if we considered the number of splits of the larger one to determine the running time, we will have $log_{\frac{5}{4}} n$ splits. In each split will have $O(n)$ sorting operations, resulting of a total running time of $O(nlog_{\frac{5}{4}} n) = O(nlogn)$

Question 6:

The algorithm is not valid because radix-sort is based on a stable sorting. each digits sorting needs to rely on the original positions of the previous digits. quick-sort is not a stable sorting algorithm as the comparison relates to the pivot only. here is an example:

If we choose 260 to be the pivot in the second iteration, the sorting will produce wrong result

$$
\begin{array}{ccccc}
100 & & 100 & & 100 \\
910 & & 910 & & 140* \\
530 & & 120 & & 120* \\
120 & \rightarrow & 530 & \rightarrow & 260 \\
140 & & 140 & & 530 \\
260 & & [260] & & 910
\end{array}
$$