Assignment 2

May 6, 2020 4:30 PM

Problem 1

QuickSort(A, 1, 12)

1) A=[13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21]

i=0

always points to the index of the last number smaller than pivot

[13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21] pivot

cur; 13 < 21, swap with number at i+1 which is itself
i++, cur++

i

[13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21]

cur: 19 < 21, swap with number at i+1 which is itself, i++, cur++

[13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21]

cur

i

[13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21]

cur

i

[13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21]

cur

i

[13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21]

cur

i

[13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21]

cur

i

[13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21]

cur

i

[13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21]

cur

i
[13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21]
cur

i
[13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21]
cur

i
[13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21]
cur
returns 12

swap Array[i+1] with Array[end]
A[12] A[12]

↓

[13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21]

Quicksort (A, 1, 11) ~~Quicksort (A, 13, 12)~~

i [13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6]
cur

i [13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6]
cur

i [13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6]
cur

smaller than pivot; swap it with the number at index i+1.

i
[5, 19, 9, 13, 12, 8, 7, 4, 11, 2, 6]
cur

i
[5, 19, 9, 13, 12, 8, 7, 4, 11, 2, 6]
cur

i
[5, 19, 9, 13, 12, 8, 7, 4, 11, 2, 6]
cur

i
[5, 4, 9, 13, 12, 8, 7, 19, 11, 2, 6]
cur

i
[5, 4, 2, 13, 12, 8, 7, 19, 11, 9, 6]
cur

i
[5, 4, 2, 6, 12, 8, 7, 19, 11, 9, 13]
returns 4

Quicksort (A, 1, 3)

i [5, 4, 2]
cur

i [5, 4, 2]
cur

[2, 4, 5]
returns 1

Quicksort (A, 5, 11)

i [12, 8, 7, 19, 11, 9, 13]
cur

i
[12, 8, 7, 19, 11, 9, 13]
cur

i
[12, 8, 7, 19, 11, 9, 13]
cur

i
[12, 8, 7, 19, 11, 9, 13]
cur

i
[12, 8, 7, 19, 11, 9, 13]
cur

i
[12, 8, 7, 11, 19, 9, 13]
cur

i
[12, 8, 7, 11, 9, 19, 13]

[12, 8, 7, 11, 9, 13, 19]

returns 10

~~Quicksort(A, 1, 0)~~ Quicksort(A, 2, 3) Quicksort(A, 5, 9) ~~Quicksort(A, 11, 11)~~

i [4, 5]
cur

i
[4, 5]
swapped with itself
returns 3

i [12, 8, 7, 11, 9]
cur

i [12, 8, 7, 11, 9]
cur

i
[8, 12, 7, 11, 9]
cur

i
[8, 7, 12, 11, 9]
cur

i
[8, 7, 9, 11, 12]
returns 7

~~Quicksort(A, 2, 2)~~ ~~Quicksort(A, 4, 3)~~

Quicksort(A, 5, 6)

i [8, 7]
cur

i [7, 8]
returns 5

Quicksort(A, 8, 9)

i [11, 12]
cur

i
[11, 12]
returns 9

~~Quicksort(A, 5, 4)~~ ~~Quicksort(A, 6, 6)~~ ~~Quicksort(A, 8, 8)~~
~~Quicksort(A, 10, 9)~~

Done. The ...

Done. The whole array is now sorted:

[2, 4, 5, 6, 7, 8, 9, 11, 12, 13, 19, 21]

2) Quicksort runs $O(n^2)$ when the whole array is filled with the same number, because every time we choose the last number as the pivot, and all other numbers are "greater than or equal to" the pivot, therefore the array doesn't shuffle at all until finally we finish the iteration and repeat with pivot now set at A[A.length-1], A[A.length-2] ... etc. For every iteration, only one element becomes "sorted", which is the pivot itself. We reduce the problem size by 1 every time and traversing the whole array requires time $O(n)$, therefore the recurrence relation is:

$$T(n) = T(n-1) + n$$

$T(n)$
$\downarrow n$
$T(n-1)$
$\downarrow$
$T(n-2)$
$\vdots$
$T(1)$

n levels * n

$\rightarrow O(n^2)$

3) $O(n^2)$. When the array is sorted in descending order and we always choose the last element as pivot, this is equivalent to always choosing the smallest number as the pivot. Every iteration, only one element becomes "sorted" (the pivot) and we reduced the problem size by one only:

$$T(n) = T(n-1) + n$$

$T(n)$
$\downarrow n$
$T(n-1)$
$\downarrow n$
$T(n-2)$
$\vdots$
$T(1)$

n levels * n

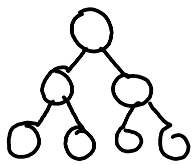
$\rightarrow O(n^2)$

Problem 2

1) A heap contains the maximum number of elements when it is represented by a full binary tree In that case, the number of nodes is:

$$2^0 + 2^1 + 2^2 + \ldots + 2^h = 2^{(height+1)} - 1 \text{ nodes}$$

e.g. height = 2,

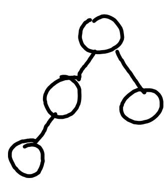


$$2^{2+1} - 1 = 2^3 - 1 = 7 \text{ nodes}$$

A heap contains the minimum number of elements when its last level contains only one node. It is like having a full binary tree with height $(h-1)$, and then adding an extra node:

$$2^0 + 2^1 + 2^2 + \ldots + 2^{h-1} = 2^{height} - 1 + 1 = 2^{height} \text{ nodes}$$

e.g. height = 2,



$$2^2 = 4 \text{ nodes}$$

2) The smallest element reside in one of the leaf nodes. According to the heap property, in a max heap, all nodes in a level have values smaller / equal to a node in its ancestor levels. Therefore, the smallest element must be within one of the leaf nodes Furthermore, the max heap property specifies that parent(i) $\geq$ i but specifies no relationship between the left and right children of the same node. Therefore, we can only say with certainty that it's one of the leaf nodes but we can't tell which one exactly

for which one exactly.

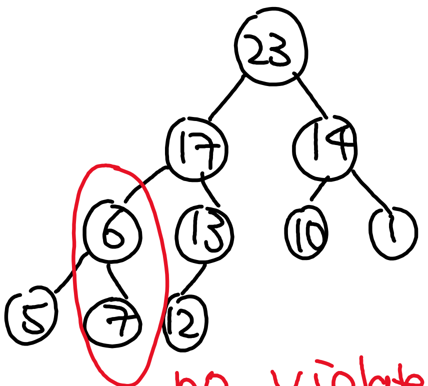
Specific indexes:

Index of last element : n

index of parent of last element : $n/2$

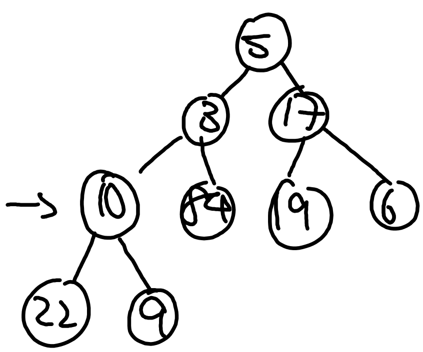
$\therefore$ indexes of leaf nodes : $A[\lfloor n/2 \rfloor + 1, n]$

3) [23, 17, 14, 6, 13, 10, 1, 5, 7, 12]



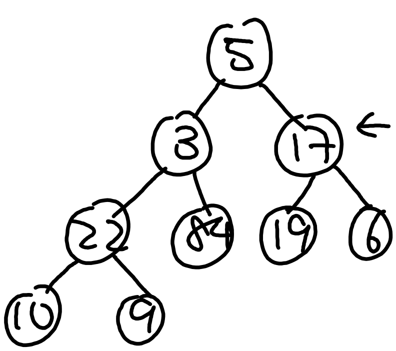
no, violates max-heap property.

4) [5, 3, 17, 10, 84, 19, 6, 22, 9]

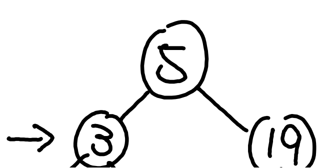


Direction : bottom → up, right → left

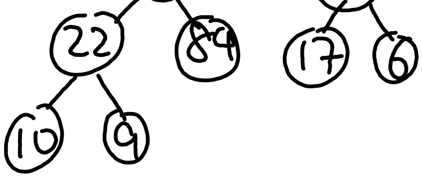
Starting with the first non-leaf node : 10
Left(10) is greater than 10, $\therefore$ we swap 22 with 10.



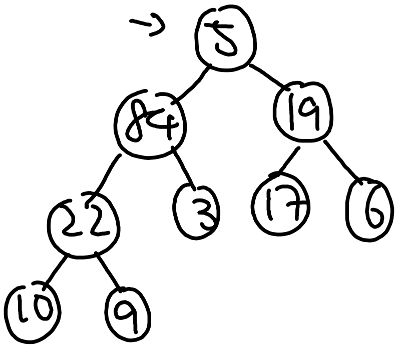
Left(17) is greater than 17, $\therefore$ we swap 19 with 17.



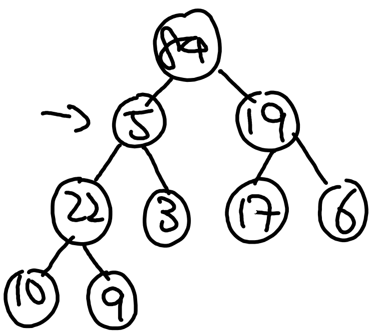
Both Left(3) and Right(3) are larger than 3, we swap 3 with the larger one 84



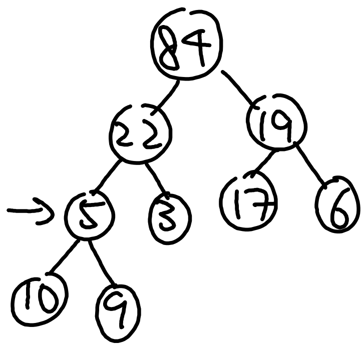
larger one, 84.



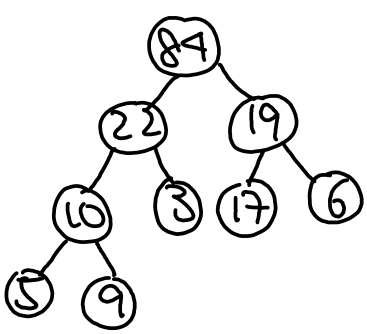
Both Left(5) and Right(5) are larger than 5, swap 5 with the larger one, 84.



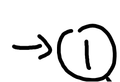
Left(5) is still larger; swap 5 with 22.



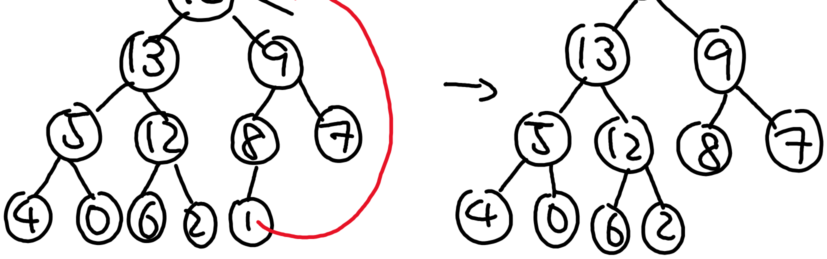
Both left(5) and right(5) are larger; swap 5 with the large one, 10.



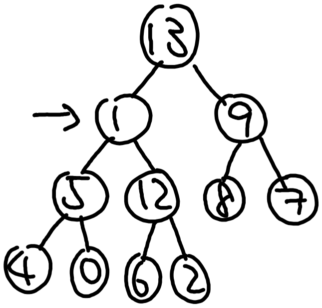
5) [15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1]



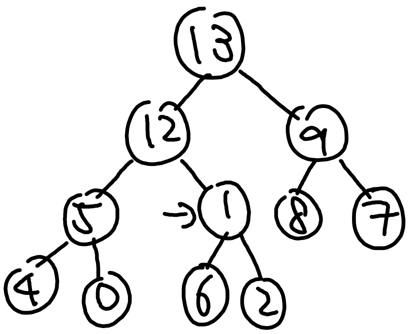
Both Left(1) and Right(1)



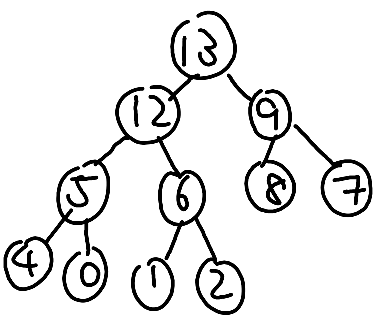
Both Left(1) & Right(1) are large; swap 1 with the large one, 13.



Both Left(1) and Right(1) are large; swap 1 with the large one, 12.



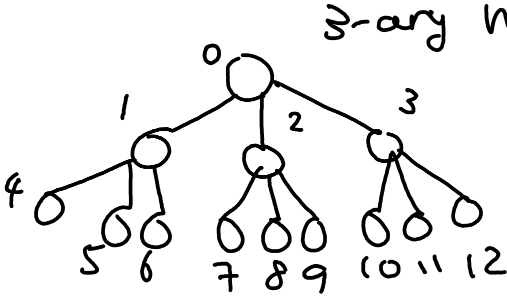
Both Left(1) and Right(1) are large; swap 1 with the large one, 6.



Problem 3

1) Here, let us assume that the corresponding position of the root node in the array is 0.

3-ary heap:



children of 0th node: 1 2 3

.. 1th node: 4 5 6

.. 2nd node: 7 8 9

...

To find the index of the i^{th} child assuming the parent's index is p: $dp+i$

To find the index of the parent node assuming the child's node is c: $\lfloor (c-1)/d \rfloor$

2) Height of a d-ary heap: $h = \lfloor \log_d n \rfloor$

3) Intuition: After deleting the maximal value from a d-ary max-heap (the root), we first replace the root with the last element in the underlying array, and it percolates down until it finds it right position and the max-heap property is maintained.

Pseudocode & Explanation:

```
DELETE_D-ARY_MAX (A[ ])
if A.heap-size < 1
    error "heap underflow"   // we cannot delete from empty heap
max = A[0]
 A[0] = A[A.heap-size-1] // swap the current maximum element
                         with the last element of the underlying
                         array
 A.heap-size = A.heap-size - 1  // we do not consider the last element
D-ARY_MAX_HEAPIFY (A[ ],0)  (maximum value to be returned)
return max;                    anymore
```

After we swap the maximum element with the last element in the array, all the subtrees of the new n-ary heap are still valid n-ary max heaps, but the new A[0] might be smaller than its children, thus violating the max-heap property. D-ARY_MAX_HEAPIFY function lets the value at

A[0] percolate down the max-heap so the whole n-ary heap obeys the max-heap property again.

```
D-ARY_MAX_HEAPIFY (A[], i, d)   ~> here, i = 0.
largest = i     // assume that the heap property is still maintained
for(int j = 1, j ≤ d, j++){
    if A[di + j] > A[i]{
        largest = di + j;    // set largest to the maximum value
    }                        // among the parent & its children
}

if largest ≠ i
    swap A[i] with A[largest]
    D_ARY-MAX-HEAPIFY(A[], largest)   // The node indexed by
                                      // largest now has the
                                      // original value at
                                      // A[i], and thus the
                                      // subtree rooted at
                                      // "largest" might violate
                                      // the max-heap property.
                                      // Consequently, we call
                                      // D_ARY-MAX-HEAP recursively
                                      // on that subtree.
```

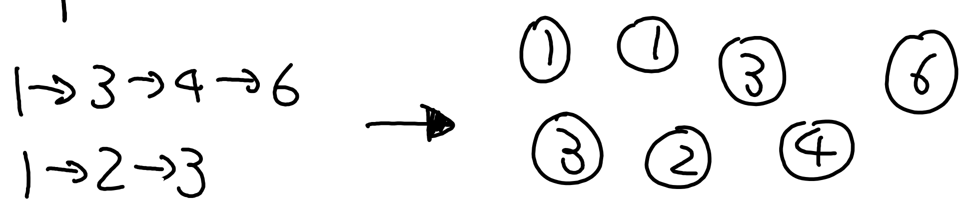
Runtime analysis:

- The n-ary heap has depth $\log_d n$, therefore we will do a maximum of $\log_d n$ swaps. For each swap, we have to do a for-loop with d values, therefore the total runtime is:

$O(d \log_d n)$.

<u>Problem 4</u>

- Intuition: My initial thought was to break down each linked list to individual nodes and put all values inside a min heap:



Then, we initialize a dummy node and pop from the heap and append to the end of the resulting list until the heap becomes empty. This methods runs O(nlogn), and then I realized that instead of putting individual nodes inside the heap, we can put whole linked lists instead; and every time when a linklist has its head popped out but its next field is pointing to another node (not null), we repush it to the heap. → This will take O(nlogk).

· Pseudocode:

```
MERGE_K_SORTED_LISTS (ListNode[] lists){
    //base cases
    if (lists == null || (lists.length == 0) return null;
    PriorityQueue <ListNode> minHeap = new PriorityQueue <>
        ( (a,b) -> a.val - b.val );
    // Linked lists will always be placed in the heap
    // according to the value of their head:
    // i.e.  1->3->4->2 comes before 3->1->4->9.
    for(int i=0 ; i< lists.length ; i++){
        if(lists[i] != null){
            minHeap.offer(lists[i]);
        }
    }

    ListNode res = new ListNode(0);
    ListNode cur = res;
    while (!minHeap.isEmpty()){
        ListNode n = minHeap.poll();
        cur.next =n;
```

```
        cur = cur.next;
        if (n.next != null) {
            minHeap.offer(n.next);
        }
    return res.next;
}
```

- Time Complexity Analysis:

Building the initial heap takes O(k); after, we need to pop n times from the heap since there are n elements, and at any given moment, there are a maximum of k linked lists inside the heap, so each REMOVE_MIN operation takes O(log k)

→ total O(n log k); since O(n log k) > O(k) we state the running time as O(n log k).

Problem 5

[6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2]

Input Array (A)

1										n
6	0	2	0	1	3	4	6	1	3	2

Working Array (B)

0	1	2	3	4	5	6
0	0	0	0	0	0	0

Output Array (C)

1										n

```
for (i=1...n)
    B[A[i]]++;
```

↳ B

0	1	2	3	4	5	6
2	2	2	2	1	0	2

```
for (i=1 ... k)
    B[i] = B[i] + B[i-1];
```

	0	1	2	3	4	5	6
B	2	4	6	8	9	9	11

```
for (i=n ... 1)
    C [B [A[i]]] = A[i];
        B [A[i]] -= 1;
```

	1	2	3	4	5	6	7	8	9	10	11
A	6	0	2	0	1	3	4	6	1	3	2

	0	1	2	3	4	5	6
B	2	4	6	8	9	9	11

	1	2	3	4	5	6	7	8	9	10	11
C						2					

	0	1	2	3	4	5	6
B	2	4	5	8	9	9	11

C						2		3			

B	2	4	5	7	9	9	11

C				1		2		3			

B	2	3	5	7	9	9	11

C				1		2		3			6

B	2	3	5	7	9	9	10				
C				1		2		3	4		6
B	2	3	5	7	8	9	10				
C				1		2	3	3	4		6
B	2	3	5	6	8	9	10				
C			1	1		2	3	3	4		6
B	2	2	5	6	8	9	10				
C		0	1	1		2	3	3	4		6
B	1	2	5	6	8	9	10				
C		0	1	1	2	2	3	3	4		6
B	1	2	4	6	8	9	10				
C	0	0	1	1	2	2	3	3	4		6
B	0	2	4	6	8	9	10				
C	0	0	1	1	2	2	3	3	4	6	6

B	0	2	4	6	8	9	9

Problem 5

- Intuition: We can make use of counting sort here. First, since we have n integers from 0 to k, we need a working array B of length $k+1$:

	0	1	2	3	4	...	k	(index = i)
B						...		

  Then, we go over array B to calculate the index of the last appearance of number i.

  Then, given a range $[a..b]$, we take $B[a]$ and $B[b]$, and the number of the n integers that fall into this range is $B[b] - B[a-1]$.

- Pseudocode:

```
NUMBER_QUERY(n numbers in range 0-k, a, b){
    int[] B = new int[k+1];
    for (int num : n integers){
        B[num] += 1;
    }

    for (int = 1 ... k){
        B[i] = B[i] + B[i-1];
    }

    return B[b] - B[a-1];   constant-time operation; O(1) query time.
}
```

- Runtime analysis:
  - ↳ Going through n integers and putting them to corresponding places in array B: $O(n)$
  - ↳ Going through array B and updating it to store the last index of each number i: $O(k)$

  Total time: $O(n+k)$

Problem 7

- all 3-letter words; maximum number of "digits" = 3.

	Corresponding	ASCII	values:
cow	99	111	119
dog	100	111	103
sea	115	101	97
rug	114	117	103
row	114	111	119
mob	109	111	98
box	98	111	120
tab	116	97	98
bar	98	97	114
ear	101	97	114
tar	116	97	114
dig	100	105	103
big	98	105	103
tea	116	101	97
now	110	111	119
fox	102	111	120

- soln: run counting-sort 3 times with a working array B of size 26.

1			2			3
sea	101		tab	116		bar
tea	101		bar	98		big
mob	111		ear	101		box

tab	97	tar	116	cow
dog	111	sea	115	dig
rug	117	tea	116	dog
dig	105	dig	100	ear
big	105	big	98	fox
bar	97	mob	109	mob
ear	97	dog	100	now
tar	97	cow	99	row
cow	111	row	114	rug
row	111	now	110	sea
now	111	box	98	tab
box	111	fox	102	tar
fox	111	rug	114	tea