Algorithms I

Tutorial 1 Solution Hints

August 12, 2016

Problem 1

Try it yourself.

Problem 2

$f_4(n), f_2(n), f_5(n), f_1(n), f_3(n)$

Problem 3

$T(n) = n + \frac{n}{2} + \frac{n}{3} + \ldots + 1$

$T(n) = n\left(1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}\right)$

$T(n) = n \cdot H_n, H_n$ is n^{th} harmonic number $= O(n \log n)$

Problem 4

$f(i) = a[i] - i$ is a non-decreasing function. So, we can binary search for i to check if $f(i) = 0$ for some i.

Problem 5

A cyclically sorted array can be represented as a concatenation of two sorted arrays. We just need to identify the position where the first array ends (let's call this position *pivot*). e.g. in $[3, 1, 2]$, the arrays are $[3]$ and $[1, 2]$. Once we find *pivot*, we can binary search in the two arrays individually. *pivot* is the maximum index i such that $array[0] < array[i]$. So, we can find *pivot* by doing a binary search in the original array.

Problem 6

We'll use a stack that can store a node of the binary tree.
Pseudo-code:

```
PUSH(stack, root)
while stack is not empty do
    p ← POP(stack)
    print p → key
    if p has right child then
        PUSH(stack, p → right)
```

```
    end if
    if p has left child then
        PUSH(stack, p → left)
    end if
end while
```

Problem 7

We'll use a stack that can store a $\{node, 0/1\}$ pair of the binary tree.
Pseudo-code:

```
PUSH(stack, {root, 0})
while stack is not empty do
    {p, x} ← POP(stack)
    if x = 1 then
        print p → key
    else
        if p has right child then
            PUSH(stack, {p → right, 0})
        end if
        PUSH(stack, {p, 1})
        if p has left child then
            PUSH(stack, {p → left, 0})
        end if
    end if
end while
```

Problem 8

```
function ARESAME(p, q)
    if p = NULL ∧ q = NULL then
        return true
    else if p = NULL ∨ q = NULL then
        return false
    end if
    if p → key = q → key then
        return ARESAME(p → left, q → left) ∧ ARESAME(p → right, q → right)
    else
        return false
    end if
end function
```

Problem 9

At each node in the BST, we also store size of the sub-tree rooted at that node. Now, we create a balanced BST (e.g. Red-Black tree) and update the size accordingly while insertion and deletion. Now, we need to find $\lceil \frac{n}{2} \rceil^{th}$ element. Let's take $k = \lceil \frac{n}{2} \rceil$. We start at root node. If size of sub-tree of current node is equal to k, we return the key of current node as median. If left child has more than or equal to k elements, we continue searching in the left child. Otherwise, we update $k := k - size$(left child sub-tree) $- 1$ and continue searching in the right child.

Problem 10

Let's first find the node with minimum key. Let's call this node x. If x has a right child (let's call it y), then the second minimum will be the minimum key in the sub-tree rooted at y. So, we start at y and keep following the left pointer as long there's a left child. We'll end up at the required node.

If x has no right child, then the parent of x has the second minimum key.

Problem 11

We can do this using in-order traversal. For a BST, in-order traversal must give a sorted sequence. So, we just maintain the key of the previously visited node during traversal. At each node, it's key must be greater than the key of the previously visited node.

Problem 12

No, the deletion operation is not commutative.

Problem 13

Largest: $2^{2k} - 1$, Smallest: $2^k - 1$

Problem 14

We'll use a max heap of size k. For the first k values, we just insert them into the heap. For the next values, if the new value is smaller than the top value of the heap then we delete the top value from heap and insert the new value. If the new value is larger than the max value, we just ignore. The top value of the heap will be the k^th smallest value.

Problem 15

We maintain one min heap (H_L) for values smaller than or equal to median and one max heap (H_R) for values greater than or equal to median. We also maintain the condition that size of H_L and H_R differ by at most 1 i.e. $|\,|H_L| - |H_R|\,| \leq 1$.

At each step we maintain the current median of the stream of numbers. Now, there are 3 cases when a new element x arrives:

- $|H_L| = |H_R| - 1$, If x is less than the current median then insert x into H_L, otherwise delete the top element from H_R and insert it into H_L, also insert x into H_R.
- $|H_L| = |H_R|$, If x is less than the current median then insert x into H_L, otherwise insert x into H_R.
- $|H_L| = |H_R| + 1$, If x is greater than the current median then insert x into H_R, otherwise delete the top element from H_L and insert it into H_R, also insert x into H_L.

If $|H_L| = |H_R|$, we take median as the average of the top elements from H_L and H_R. Otherwise, we take the top element from the heap having more number of elements.