David Ma / e8y0b

1. a) $T(2) = c$

$T(3) = T(2) + d = c + d$

$T(4) = T(2) + d = c + d$

$T(5) = T(3) + d = c + 2d$

$T(9) = T(3) + d = c + 2d$

$T(10) = T(4) + d = c + 2d$

$T(17) = T(5) + d = c + 3d$

...

IH: $T(n) = c + \lceil \lg \lceil \lg n \rceil \rceil d \quad (n \geq 2)$

b) When $k = n^2$

Is: $T(n^2) = T(\lceil n^2 \rceil) + d \quad (n^2 > 2)$

$= c + (\lceil \lg \lceil \lg n \rceil \rceil + 1) d$

(from IH)

$= c + (\lceil \lg \lceil \lg n \rceil \rceil + \lg 2) d$

$= c + (\lceil \lg 2 \cdot \lceil \lg n \rceil \rceil) d$

$= c + \lceil \lg \lceil \lg n^2 \rceil \rceil d$

$= T(n^2)$ QED

c) $T(n) \in \Theta(\lg \lg n)$

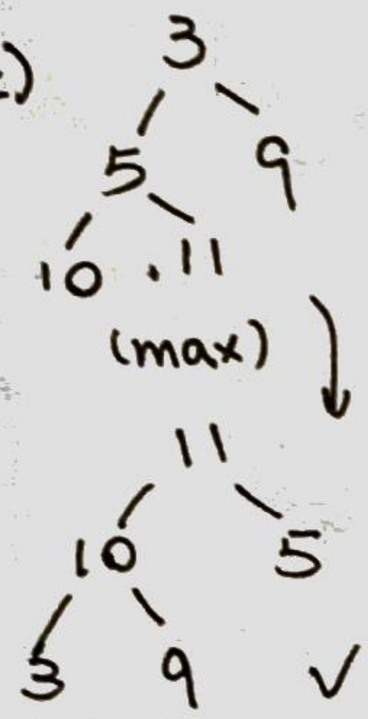
2. a)

The algorithm fails because if it makes no chages on the root node, the rest of the nodes would stay the same.

b)

It only takes care of the node that needs to be swapped down, so it fails to consider the new parent node.

c)



It starts from the children nodes of the ~~parents~~ root node and swap them up. By doing so, it maintains a heap of increasing size (from 1 to n)

If it encounters a node — it can swap it to the appropriate position.

↑ up

3.

```
/* Yes the algorithm is tail recursive */
void swapDown(int* heap, int i, int size) {
    int min = i;
    if (2 * i < size - 1 &&
          heap[2 * i + 1] < heap[min])
        min = 2 * i + 1;
    if (2 * i < size - 2 &&
          heap[2 * i + 2] < heap[min])
        min = 2 * i + 2;

    while (min != i) {
        int leftChild = 2 * min + 1;
        int rightChild = 2 * min + 2;
        swap(heap[i], heap[min]);
        i = min;
        if (leftChild < size &&
              heap[leftChild] < heap[min])
            min = leftChild;
        if (rightChild < size &&
              heap[rightChild] < heap[min])
            min = rightChild;
    }
}
```

```c
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <assert.h>

struct TreeNode {
        char name[1000];
        TreeNode *left;
        TreeNode *right;
};
typedef struct TreeNode TreeNode;
```

```c
// 4(a)
const char *treeToString(TreeNode * root) {
        char *repr = NULL;
        if (root == NULL) return "";
        if (root->left == NULL && root->right == NULL)
                return (const char *) root->name;

        repr = (char *) calloc(1000, sizeof(char));

        // return '(' + root->l + ',' + root->r + ')'
        repr[0] = '(';
        strcpy(&repr[strlen(repr)], treeToString(root->left));
        strcpy(&repr[strlen(repr)], ",");
        strcpy(&repr[strlen(repr)], treeToString(root->right));
        strcpy(&repr[strlen(repr)], ")");

        return repr;
}
```

```c
// 4(b)
int height(const char *str, int lbs=0, int max=0) {
        if (str[0] == '\0') return max + 1;

        if (str[0] == '(')
                lbs += 1;
        else if (str[0] == ')')
                lbs -= 1;
        if (lbs > max) max = lbs;
        return height(str+1, lbs, max);
}
```

5. (a) Base case:

tmp[.] does not contain any element, so it is in sorted order. (we cannot compare the value of x[a] and x[b] for now)

Induction Step:

(i) a has elements but b does not ($a \leq mid$ and $b > hi$)

The fact that b does not have element infers that it's last element (and the previous ones) is smaller than x[a], and we set the next value of tmp to be x[a]. Since x[lo..mid] is sorted, x[a] is smaller than the latter elements, but greater than the previous ones. Therefore, the new tmp[] array is sorted. Since $b = hi+1$, we only need to consider x[a]. We have proved that x[a] is both greater than its previous element and x[b], so the invariant holds.

(ii) a and ~~be~~ b are both in the range but $x[a] < x[b]$.

$a \leq mid$ and $x[a] < x[b]$

in this case, x[a] is the smaller element, so we assign it to temp[k] and increase the index of a by 1. By doing so, we can compare x[a+1... mid] with x[b] later. Therefore, every elements added later is ≥ larger than x[a]. ~~Since we it hasn't reached the end of a and b,~~ and the current element x[a] is greater than tmp[k-1].

(iii) b has elements but a does not

this case is very similar to i) and the same conclusion holds.

(b) When the loop terminates, the tmp[] array is filled with elements from x[] in ascending order.

(b) variables lo and hi do not change throughout the loop, while the counter k increases by 1 each time. There are no recursive calls or external calls inside the loop. The loop terminates when $k > hi$, so it iterates hi-lo

times.

(c) The sorted array is stored in tmp[] from index lo to hi (inclusive) and it contains all the elements in x[lo..mid] and x[mid+1..hi].

6. Denote the ~~length~~ original parameter n as n_0

invariant:

$$fib_{old} + fib_{n_0 - n} = fib$$