Table of Contents

Question 1:

Question 2:

Question 1.

Algorithm
Min_Stage2 (`jobs`: array of tuples $(j_i,\ s_i)$)

```
1: sort jobs by s_i decreasing
2: return jobs
```

As we can see, `Min_Stage2()` is a greedy algorith with a choice of longest stage 2. `Min_Stage2()` gives a feasible output since we just return a reordered list of the original list of jobs.

Claim: Swapping an earlier job with a later job that has a longer stage 2 will not make the end time worse.

Proof:

Let $f_i,\ s_i$ be the earlier job with a shorter stage 2

Let $f_j,\ s_j$ be the later job with a longer stage 2

$\implies s_j \geq s_i$

Let end time be the time it takes for both stages of the job to finish

Before swapping, we have

$$\text{end time}(s_i) = f_i + s_i$$

$$\text{end time}(s_j) = f_i + f_j + s_j$$

$$\begin{aligned}\text{latest end time} &= \max\big(f_i + s_i,\ f_i + f_j + s_j\big) \\ &= f_i + f_j + s_j \qquad [\text{since } s_j \geq s_i]\end{aligned}$$

After swapping, we have

$$\text{end time}(s_i)' = f_j + f_i + s_i$$

$$\text{end time}(s_j)' = f_j + s_j$$

$$\text{latest end time}' = \max\big(f_j + s_j,\ f_j + f_i + s_i\big)$$

in all cases, latest end time $\geq$ latest end time$'$ ■

Theorem: Greedy schedule S given by `Min_Stage2()` is optimal

Proof (Exchange argument):

Let O be an optimal ordering of jobs O-job$_i$

Let S be the ordering of jobs S-job$_i$ given by `Min_Stage2()`

Suppose
$$\begin{aligned} O\text{-job}_1 &= S\text{-job}_1 \\ O\text{-job}_2 &= S\text{-job}_2 \\ &\vdots \\ O\text{-job}_{i-2} &= S\text{-job}_{i-2} \\ O\text{-job}_{i-1} &= S\text{-job}_{i-1} \end{aligned}$$

but $O\text{-job}_i \neq S\text{-job}_i$

WTS: We can change order of O to make it more similar to S and the end time will not be any worse

Let the remaining different jobs be
O-Remaining $= [O\text{-job}_i,\ O\text{-job}_{i+1},\ \cdots,\ O\text{-job}_n]$ for O, and
S-Remaining $= [S\text{-job}_i,\ S\text{-job}_{i+1},\ \cdots,\ S\text{-job}_n]$ for S

since S-job$_i \neq O$-job$_i$, and S-Remaining are sorted decreasing by longest stage 2's, then we can swap O-job$_i$ with a job that has the longest stage 2. Namely, continously swap O-job$_i$ with the left job until it reaches the S-job$_i$'s position in S

We can repeat this until O-Remaining $= S$-Remaining
$\implies O$ becomes more similar to S.
and by the claim above, the end time does not get worse. ■

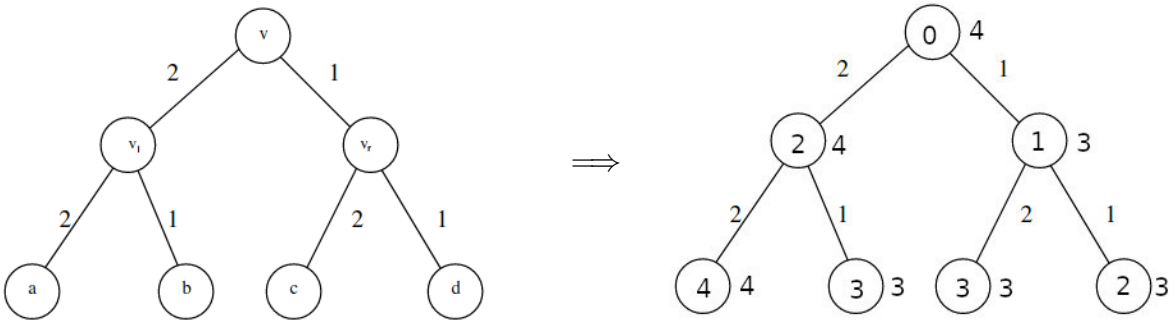
`Min_Stage2()`'s complexity is $\mathcal{O}(n \log n)$ since it's simply sorting

Question 2.

Our objective is to minimize the number of edges to add to get zero skew.

To attain this it's important to notice that increasing the length of an edge closer to the root increases the edge of 2^d leaves where d is the depth of that node at the end of the edge. Therefore, we should attempt to increase edge lengths greedily starting from the root, ensuring that the increase does not push one of the distances above the current maximum. This is because if we increase the maximum distance, we would at least not change the skew, or even risk the chance of increasing it.

This algorithm is done with an augmented binary tree, where each node stores the distance between it and the root and the maximum distance from the root to any of its subtree leaves: i.e.



Algorithm:

```
"""
BTree has shape
{
  left: {
          length: int
          node: BTree
        }
  right: {
    length: int
    node: BTree
  }
}

AugmentedBTree has shape
{
  max: int
  distance: int
  left:  AugmentedBTree
  right: AugmentedBTree
}
"""
def zero_skew(orig: BTree):
    def augment(root, distance):
        if (root.left == None and root.right == None):
            return Node (
                dist=distance,
                max=distance,
                left=None,
                right=None
            )
        left = augment(root.left.node, root.left.length + distance)
        right = augment(root.right.node, root.right.length + distance)
        return Node(
            dist=distance,
            max=max(left.max,right.max),
            left=left,
            right=right
        )

    AugmentedTree = augment(orig, 0)
    max_d = AugmentedTree.max

    def increase_length(root, inc):
        root.dist += inc
        root.max += inc
        if (root.left == None and root.right == None):
            root.dist = max_d
        elif root.max == max_d:
            root.left = increase_length(root.left, inc)
            root.right = increase_length(root.right, inc)
        else:
            i = max_d - root.max
            root.dist += i
            root.right = increase_length(root.right, i)
            root.left = increase_length(root.left, i)
        return root

    return increase_length(AugmentedTree, 0)
```

`zero_skew()` gives a feasible solution since the algorithm copies the tree with more info, then starts increasing the length.
The greedy choice is the first node to increase the length.
In terms of complexity, it is clear that all operations done in `zero_skew()` are inorder traversals of the input tree and therefore the total complexity is $O(n)$.

Proof of correctness (Exchange argument):

Since the operations of this algorithm are done in an inorder traversal, I will argue without loss of generality about one path.

Let A be the solution given by the greedy algorithm and A^{opt} be an optimal solution.
Let b be a path between the root and a leaf node in A, and b^{opt} be the path in A^{opt}
Let n be the first node differing between b and b^{opt}.

It cannot be that the node is larger in b^{opt} than b because then the greedy algorithm did not increase that node because increasing it would cause the the maximum distance to increase, which cannot happen in an optimal solution. So n is larger in b than b^{opt}. But this means that both left and right paths of the node in b^{opt} have an increase, as some nodes in both paths are below the maximum distance (which we know because the node was increased in b). Therefore, we can remove 1 from each of the subsequent nodes in the left and right branches and increase n in b^{opt}. But then we just reduced the amount of additions without going over the maximum distance in b^{opt} which goes against the assumption that b^{opt} is an optimal solution. So there cannot be an optimal solution that differs from the greedy solution.

$\therefore$ `zero_skew()` gives optimal solutions.