(a) Group 1: \( f_3, f_4, f_1, f_2 \)
(b) Group 2: \( f_1, f_2, f_4, f_3 \)
(c) Group 3: \( f_3, f_4, f_2, f_1 \)
Note that we can simply solve this problem using the 1D-peak-finding algorithm. Here’s our pseudocode:

```python
def findMaxOfUnimodal(A):
    lowerBound = 0
    upperBound = len(A)

    while (true):
        index = (upperBound - lowerBound) / 2
            lowerBound = index
            upperBound = index
        else:
            return (index, A[index])
```
1. This doesn’t change either the dot product of $D_1$ (the vector for last year’s speech) and $D_2$ (the vector for this year’s) nor the sizes of them. Thus this doesn’t help Banach at all and $\Theta = 0$.

2. This in fact helps Banach. The dot product doesn’t change although the product of their lengths does.

3. This also helps Banach a little bit. This time it’s not quite as obvious however. Both the dot product and product of lengths are reduced. If $D_1$ is the vector of last year’s speech while $D_2$ is the vector of this year’s, we’ll have that $D_1 \circ D_2 = D_1 \circ D_1 = \|D_1\|^2$. Thus:

$$\frac{D_1 \circ D_2}{\|D_1\|\|D_2\|} = \frac{\|D_1\|}{\|D_2\|}$$

which is less than 1 and thus $\Theta$ is closer to $\pi/2$. 
(a) Save the tree into a sorted list using an inorder-walk. From here, you can iterate through the sorted list, counting elements once you reach \(a\) and returning the count after you reach \(b\). Another option is to binary search for \(a\) and \(b\), getting their indices in the list, and taking the difference to find the answer. Both methods are bound by the time it takes to do an inorder-walk through the tree and have a runtime of \(O(n)\). Note that we can do no better than \(O(n)\) since we must visit every node within the range \(a\) to \(b\) and there may be \(n\) nodes in that range.

Another (slower) option is to find \(a\) in the tree and call next-larger until you reach a node with \(b\) as the key. Since next-larger takes \(O(\log n)\) time in an AVL tree and we could potentially call next-larger \(O(n)\) times, the runtime of this option is \(O(n \log n)\).

(b) Say each node contains the field \texttt{size}, indicating the size of the subtree rooted at that node. Consider the operation \texttt{num-prices-below} as described below, which returns the number of keys found in the entire tree smaller than \(k\):

\begin{verbatim}
num-prices-below(k):
    count = 0
    n = root
    while n != NIL:
        if n = k:
            return count + n.left.size
        elif n < k:
            count = count + 1 + n.left.size
            n = n.right
        else:
            n = n.left
    return count
\end{verbatim}

Similarly, we can create the operation \texttt{num-prices-above} as described below, which returns the number of keys found in the entire tree larger than \(k\):

\begin{verbatim}
num-prices-above(k):
    count = 0
    n = root
    while n != NIL:
        if n = k:
            return count + n.right.size
        elif n > k:
            count = count + 1 + n.right.size
            n = n.left
\end{verbatim}
else:
    n = n.right
return count

Using \texttt{num-prices-below}, we can find out how many keys are smaller than \(a\) and using \texttt{num-prices-above}, we can find out how many keys are larger than \(b\). The sum of the two gives you the number of prices outside of the range \(a\) to \(b\). Simply subtract this from the total size of the tree to get the number of prices within the range \(a\) to \(b\).

\texttt{num-textbooks-in-range}(a,b) \texttt{returns}
root.size - \texttt{num-prices-below}(a) - \texttt{num-prices-above}(b).