DSA Homework 4

B00401062 羅文斌

Problem 1.1

a. The weight of any given subtree is 1 (the root itself) plus the sum of the weight of its children. The weight of each node is stored and returned to its calling parent. Every node in the tree will be traversed exactly once. Therefore, this is an $O(N)$ algorithm.

```
def weight(root):
    if root is None:
        return 0
    root.weight = 1
    for child in root.children:
        root.weight += weight(child)
    return root.weight
```

b. We first arbitrarily choose a node as the root, which takes constant time. Next, we setup the weight of each subtree with respect to that root, which takes $O(N)$ as stated in the last subproblem. Then, we perform DFS from the same root to traverse the tree. A node is called the centroid if and only if it satisfies the following conditions: (1) all of its children have weight at most $\frac{N}{2}$, and (2) the number of nodes excluding its children and itself is at most $\frac{N}{2}$. DFS would always achieve a centroid and take time less than $O(|V| + |E|) = O(N + (N - 1)) = O(N)$. Therefore, the whole process takes $O(1) + O(N) + O(N) = O(N)$.

Problem 1.2

a. The height of any given subtree is 1 (the root itself) plus the maximum height of its children. The height of each node is stored and returned to its calling parent. Every node in the tree will be traversed exactly once. Therefore, this is an $O(N)$ algorithm.

```
def height(root):
    if root is None:
        return 0
    root.height = 0
    for child in root.children:
        root.height = max(root.height, height(child))
    root.height += 1
    return root.height
```

b. We start with an arbitrarily chosen node as the root can choose the maximum value from the following values:

- 1 plus the sum of two maximum heights of its children.
- The maximum diameter among its children subtree.

This algorithm involves recursion. The pseudocode is as follows. The `kmax(array,k)` function in line 9 returns the maximum k values from the input array.

```
1  def diameter(root):
2     if root is None:
3         return 0
4     heights = []
5     diameters = []
6     for child in root.children:
7         heights.append(height(child))
8         diameters.append(diameter(child))
9     return max(1 + sum(kmax(heights, 2)), max(diameters))
```

Problem 1.3

a. Suppose the arbitrarily-picked root is x, the farthest node from the root is y, and the two ends of the diameter path is u and v. Let $d(a, b)$ be the distance between any two given node a and b. There are two conditions to be considered if y is not on (u, v):

- (x, y) intersects with (or touches) (u, v) at a node j other than y. Since (u, v) is the diameter path, we can write down $d(u, v) = d(u, j) + d(v, j) \geq d(u, j) + d(y, j)$ and $d(u, v) = d(u, j) + d(v, j) \geq d(y, j) + d(v, j)$. Therefore, $d(u, j) \geq d(y, j)$ and $d(v, j) \geq d(y, j)$. Since y is the farthest node from x, we can write down $d(x, y) = d(x, j) + d(y, j) \geq d(x, j) + d(u, j)$ and $d(x, y) = d(x, j) + d(y, j) \geq d(x, j) + d(v, j)$. Therefore, $d(u, j) \leq d(y, j)$ and $d(v, j) \leq d(y, j)$. Finally, we can conclude, $d(y, j) = d(u, j) = d(v, j)$. Likewise, $d(x, j) = d(u, j) = d(v, j)$. To sum up, if (x, y) intersects with (or touches) (u, v) at a node j other than y, then (x, y) itself is a diameter paths. y is therefore a valid node on a diameter path.
- (x, y) does not intersect with (u, v). There is always exactly a path (j_1, j_2) connecting (x, y) and (u, v) where j_1 is a node on (x, y) and j_2 is a node on (u, v); otherwise, this is not a valid tree. Also, $d(j_1, j_2) > 0$ since (x, y) does not intersect with (u, v). Since (u, v) is a diameter path, we can write down $d(u, v) = d(u, j_2) + d(v, j_2) \geq d(u, j_2) + d(j_1, j_2) + d(y, j_1)$. Therefore, $d(v, j_2) \geq d(j_1, j_2) + d(y, j_1)$. Since y is the farthest node from x, we can write down $d(x, y) = d(x, j_1) + d(y, j_1) \geq d(x, j_1) + d(j_1, j_2) + d(v, j_2)$. Therefore, $d(y, j_1) \geq d(j_1, j_2) + d(v, j_2)$. The inequality $\begin{cases} d(v, j_2) \geq d(j_1, j_2) + d(y, j_1) \\ d(y, j_1) \geq d(j_1, j_2) + d(v, j_2) \end{cases}$ gives us

$d(j_1, j_2) = 0$. However, this is a contradiction to $d(j_1, j_2) > 0$.

To sum up, if y is the farthest node from x and not on a diameter path (u, v), then (x, y) must intersect with (or touch) (u, v), and if (x, y) intersects with (or touches) (u, v), y itself is still a valid node on a diameter path.

b. Section A:

```
previous[u] = father
```

Section B:

```
m = b
while distance[m] is not distance[b]/2:
   m = previous[m]
return m
```

Problem 2.1

a. Suppose we arbitrarily choose the last element of the array as the pivot in each partition, one of the worst cases is that the array is sorted already, e.g. $[1, 2, \ldots, n-1, n]$

b. Insertion sort takes linear time to sort a sorted array, e.g. $[1, 2, \ldots, n-1, n]$, and is faster than merge sort, which always takes $O(N \lg(N))$ to sort any array regardless of the initial pattern.

Problem 2.2

Since the range of integers is known, we can create a list of buckets of size $k+1$ to count the number of integers having showed up. An array of input size n would take $O(n)$ to record the counts. Next, we calculate the accumulative sum at each index. The resulting values imply the starting position of any integers in the sorted array. This step takes only $O(k)$. On query, we just substract the starting positions to get the number of integers falling within the query range. The pseudocode is as follows:

```
k = len(array)
buckets = [0] * (k+1) # initialize buckets of size k+1 with 0
for e in array: # O(n)
   buckets[e] += 1
accusum = [0] * (k+2)
for i in range(1, k+2): # O(k)
   accusum[i] = accusum[i-1] + buckets[i-1]
def query(a, b):
   return accusum[b+1] - accusum[a]
```

Problem 2.3

a. [501, 939, 1137, 2345, 666, 34, 218] → [501, 34, 2345, 666, 1137, 218, 939] →
[501, 218, 34, 1137, 939, 2345, 666] → [34, 1137, 218, 2345, 501, 666, 939] →
[34, 218, 501, 666, 939, 1137, 2345]

b. Time complexity of radix sort is $O(nd) = O(n(\lfloor \log_r k \rfloor + 1))$, and that of pure counting sort is $O(k)$. In the case when data are densely-distributed in the range of k, pure conuting sort sort is favored. In contrast, if the data are sparsely-distributed in the range of k, radix sort would be of the better choice. In the example [501, 939, 1137, 2345, 666, 34, 218], radix sort is favored, since $n = 7$ is much smaller than $k = 2345$.

Problem 2.4

Sorting result of LSD RadixSort + MergeSort: [20, 29, 57, 37, 36, 50, 59] →
[20, 50, 36, 57, 37, 29, 59] → [20, 29, 36, 37, 50, 57, 59]
Sorting result of LSD RadixSort + HeapSort: [20, 29, 57, 37, 36, 50, 59] →
[20, 50, 36, 37, 57, 29, 59] → [20, 29, 36, 37, 50, 57, 59]
The result of the LSD RadixSort + HeapSort greatly depends on the implementation of the heap. Heap sort is known as an unstable sorting algorithm. Therefore, the ordering of the less significant digits may be disrupted when we are sorting the more significant digits with heap sort. The result of LSD RadixSort + HeapSort may also be incorrect. In contrast, merge sort is a stable sort, the result is guaranteed to be correct.

Problem 2.5

Bucket sort.

Problem 3.1

The tree can be partitioned into three trees: {1,2,3}, {4,5}, {6,7}. To find a three tuple, we choose one element from each of the three sets. All valid three tuples are (1,4,6), (1,4,7), (1,5,6), (1,5,7), (2,4,6), (2,4,7), (2,5,6), (2,5,7), (3,4,6), (3,4,7), (3,5,6), (3,5,7)

Problem 3.2

A group of vertices conneted by black edges forms a disjoint set. The number of tuples is the product of the number of vertices in each disjoint set. A tree can be seen as a graph. During the construction of the adjacency list, the red edges can be dropped since each red edge defines the boundary of a disjoint set. Next, we use either BFS or DFS to traverse all vertices connected by black edges until the entire graph is traversed. BFS is used in the pseudocode given below.

```
def tuples(G):
    # initialization
    for v in G.V:
        v.visited = False
    ans = 1
    Q = Queue()
    # traversal
    for s in G.V:
        if s.visited:
            continue
        tmp = 0
        Q.enqueue(s)
        while not Q.empty():
            u = Q.dequeue()
            for v in G.adj[u]:
                if not v.visited:
                    v.visited = True
                    Q.enqueue(v)
                    tmp += 1
        ans *= tmp
    return ans
```