Hw4 b05902022 張雅信

Problem1

1.1.(a)

```
struct Node
{
	int data;
	struct Node* children[]
};
weight[N]={}; // indexed from 1 to N

//record weight in array
int find_weight(Node* root){
	if(root==NULL) return 0;
	weight[root->data]= 1;
	if(is_leaf(root)) return 1;
	for each_child in children:
		weight[root->data]+=find_weight(each_child);
	return weight[root->data];
}
```

1.1.(b)

```
weight[N];	// indexed from 1 to N

int find_centroid(Node* root){
	int all_weight=find_weight(root);
	Node* centroid=root;
	Node* temp=root;
	Node* largest_child;
	int temp_max = all_weight;				// finding max tree weight
	while(all_weight-weight[temp] >= weight[temp]){//離開 while的前個temp即為centroid
		centroid = temp;
		temp_max=0;

		// find and goto largest subtree
		for i in temp->children:
			if(weight[i->data]>temp_max){
				temp_max = weight[i->data];
				largest_child = i;
			}
			//有兩個同樣重的子樹->不會再往下走
			else if(weight[i->data]==temp_max) break;
		temp = largest_child;
	}
	return centroid->data;
}
```

1.2.(a)

```
struct Node
{
	int data;
```

```
	struct Node* children[]
};
height_array[N]={};  // indexed from 1 to N

//record weight in array
int find_height(Node* root){
	if(root==NULL) return -1;
	height_array[root->data]= 0;
	if(is_leaf(root)) return 0;
	int max_height=0;
	for each_child in children:
		if(find_height(each_child)>=max_height)
			max_height=1+find_height(each_child);
	height_array[root->data]= max_height;
	return max_height;
}
```

1.2.(b)

```
//filling height_array first
int only_to_call_function = find_height(root);
int find_diameter(Node* root){
	if(root==NULL) return -1;
	if(is_leaf(root)) return 0;
	int diameter=0;
	int first_max=0;
	int second_max=0;

	//find largest two height
	for i in children:
		int h=height_array[i];
		if(h>=first_max)
			second_max=first_max;
			first_max=h;
		else if(h>=second_max):
			second_max=h;
	//not necessarily go through root, so go recursion
	//但只需往最長的那條走就可以了
	//assume max=a,second_max=b
	// -> a+b > 2b-2
	// -> 保證其他路不會更長（等同1.3.(a)的證明）
		int child_path=find_diameter(first_max_child);
		if(child_path>diameter)
			diameter=child_path;
	diameter = (first_max+second_max>diameter)? (first_max+second_max):diameter;
	return diameter;
}
```

1.3.(a)

we can give recursive definition,start from root:

case(diameter path pass through root):

choose largest height from two different subtree(diameter path=倒V型)

```
                -> include the farthest node from the root
case(diameter path does not pass through root):
        choose the subtree which has the largest height
                -> (called recursively) include the farthest node from the root
        <if not>
                let the largest height of the subtree you choose = h,
                        the distance between root and the farthest root = h0
                -> h0>h => h0+h > 2h
                -> diameter will be h+h0+1
                -> diameter path will pass through root,which lead to a contradiction
```

1.3.(b)

```
section A:
        previous[u]=father
section B:
        midpoint=b
        for(i=0;i<distance[b]/2;i++)
                midpoint=previous[midpoint]
        return midpointø
```

Problem 2

2.1.(a)

assume that the quicksort is not modified,
in other words, always choose the leftmost one as pivot
instead of pick one randomly. And it aims to sort from small to large.

```
QUICKSORT(A, p, r)
if p<r
        q=PARTITION(A, p, r)
        QUICKSORT(A, p, q-1)
        QUICKSORT(A, q+1, r)
```

example of size N:
[N N-1 N-2 N-3 ..... 3 2 1]
each time it calls QUICKSORT, q=r
so QUICKSORT will be called N times,
each time cost O(N) to PARTITION (just throwing all elements to the left side)
->running time = O(N^2)
< actually: N+(N-1)+(N-2)+....+1 = N*(N+1)/2 -> O(N^2) >

2.1.(b)

```
Insertion-Sort(A)
for j = 2 to A.length
        key = A[j] // Insert A[j] into the sorted sequence A[1 . . j – 1].
        i=j−1
        while(i>0 and A[i]>key)
                A[i + 1] = A[i]
                i=i−1
        A[i+1] = key
```

Merge sort is non-adaptive and has running time O(NlogN)
no matter what kind of input is given.

However, Insertion Sort is adaptive, so if we give a input
that is already sorted, that is, A[i] always < key
it will lead to O(N) running time, since it never enters the while loop.
example of size N:[1 2 3 4 ... N-1 N]

2.2
use Counting Sort:

```
int C[];
void CountingSort (int A[], int n, int B[], int k) {
	int i, j;
	for (i=0;i<=k;i++) C[i]=0;			//O(k)
	for (j=1;j<=n;j++) C[A[j]]++;			//O(n)
	for (i=1;i<=k;i++) C[i]+=C[i-1];		//O(k)
	for (j=n;j>=1;j--) {				//O(n)
		B[C[A[j]]]=A[j];
		C[A[j]]--;
	}						// running time: theta(n+k)

int answer(int a,int b){
	if(a==0)	return C[b];
	return C[b]-C[a-1];	//O(1)
}
```

2.3.(a)
list (501, 939, 1137, 2345, 666, 34, 218)
C[10] // from 0 to 9
個位數：(501, 34, 2345, 666, 1137, 218, 939)
C=[0,1,0,0,1,1,1,1,1,1]
十位數：(501, 218, 34, 1137, 939, 2345, 666)
C=[1,1,0,3,1,0,1,0,0,0]
百位數：(34, 1137, 218, 2345, 501, 666, 939)
C=[1,1,1,1,0,1,1,0,0,1]
千位數：(34, 218, 501, 666, 939, 1137, 2345)
C=[5,1,1,0,0,0,0,0,0,0]

2.3.(b)
implement RadixSort with CountingSort: O(d*(n+k)) ,where (d=log k / log r)
only using CountingSort: O(n+k)
in this case,
(n,k,r)=(7,10,10) for implement RadixSort with CountingSort, where d=4,k from 0 to 9
(7,10000,(dont care)) for using CountingSort only, where k from 0 to 9999
-> 4*(7+10) = 68 < 10007 = 7+10000

2.4
sequence [20, 29, 57, 37, 36, 50, 59]

LSD RadixSort using MergeSort
個位數：[20, 50, 26, 57, 37, 29, 59]
十位數：[20, 26, 29, 37, 50, 57, 59] (done)

LSD RadixSort using HeapSort
個位數：[20, 36, 50, 37, 29, 57, 59]
十位數：[20, 29, 50, 37, 36, 57, 59] (done)

結果不一樣，因為 heap sort 對每個subtree，不會比較children之間的大小，只取最大的放到root。而 merge sort 是 non-adaptive，所以兩者的結果不一樣，也可預期與使用 counting sort 來實作的結果會不同。

2.5
bogosort

Problem 3
3.1
(1,4,6), (1,4,7), (1,5,6), (1,4,7),
(2,4,6), (2,4,7), (2,5,6), (2,4,7),
(3,4,6), (3,4,7), (3,5,6), (3,4,7)

3.2
let C(a,b)= a!/(b!*(a-b)!) (combination)
use disjoint set.

```
find_tuple_number(E[],N){
        //initialize:make N set, each contain a node
        int number_of_set=N;
        int size[N];
        for(i=1;i<=N;i++){
                size[i]=1;
                Make_Set(i);            //make set initialization: head(group_num)=self
        }
        //let the nodes connected by black edge be one set
        for e in E:
                if(e.color=='b') {
                        Union(nodeA, nodeB);
                        size[nodeA->head]+=size[nodeB->head];
                        size[nodeB]=0;
                        number_of_set--;
                        delete(e);
                }
        //build another graph, vertex = set, edge = red edge
        // 新作的圖也會是樹，沒有重複連接的問題(各個set之間不會有兩條路)
        // 所以只要任選三個vertice，每個vertex選一個node，即是一組tuple
        rename each group:
                new group_num start from 1 to number_of_set
        int sum=0;
        for(i=3;i<=number_of_set;i++)
                for (j=2;j<i;j++)
                        for (k=1;k<j;k++)
                                sum+=size[i]*size[j]*size[k];
        return sum;
}
```

in the given example,new graph:
(group1)--(group2)--(group3)
group1:(1,2,3),size=3
group2:(4,5),size=2
group3:(6,7),size=2
sum=3*2*2=12.