**B05902118 陳盈如**

**Problem2.4 b05902124**

**Problem3.2 b05902066**

**Problem4 助教 b05902041**

**Problem5 b05902066**

**Problem1.1**

**(a)**

*MAIN()* *//初始化*

**1** for (i = 1; i <= N; i++)

**2** i.color = white;

**3** i.weight = 0;

**4** **WEIGH**(G, 1);

*WEIGH(G, i)*

**1** i.color = gray;

**2** for (each j is node of i)

**3** if (j.color == white)

**4** **WEIGH**(G, j);

**5** i.weight += j.weight;

**6** i.weight += 1;

**7** i.color = black;

利用DFS先衝到最下面，再慢慢推回來，自己.weight就是所有子node.weight的總和，加完之後要再+1，也就是加上自己。

white代表從來沒碰過，gray代表已經碰過但不是所有子node都被碰過，black代表自己跟所有子node都被碰過了。

**(b)**

*CENTROID()*

**1** smallest\_max = N + 1;

**2** centroid = 0;

**3** for (i = 1; i <= N; i++)

**4** i.biggest = 0;

**5** for (each j is node of i)

**6** if (i.biggest < j.weight)

**7** i.biggest = j.weight;

**8** if (i.biggest < N - i.weight)

**9** i.biggest = N - i.weight;

**10** if (i.biggest < smallest\_max)

**11** smallest\_max = i.biggest;

**12** centroid = i;

**13** return centroid;

讀取每一個node，若是刪掉此node之後，剩下的tree的weight去比較，找出最大的值紀錄在biggest。

剩下的tree會是此node的所有子node.weight本身就已經有紀錄了，那parent那邊的weight就利用root.weight - 此node.weight得到。

最後把每一個node.biggest作比較，找出最小的biggest，他就是答案。

**Problem1.2**

**(a)**

*MAIN() //初始化*

**1** for (i = 1; i <= N; i++)

**2** i.color = white;

**3** i.level = 0;

**4** **LEVEL**(G, 1);

*LEVEL(G, i)*

**1** i.color = gray;

**2** for (each j is node of i)

**3** if (j.color == white)

**4** LEVEL(G, j);

**5** if (i.level < j.level)

**6** i.level = j.level;

**7** i.level += 1;

**8** i.color = black;

利用DFS先衝到最下面，再慢慢推回來，自己.level就是所有子node.level的最大值，找到之後要再+1，也就是加上自己。

white代表從來沒碰過，gray代表已經碰過但不是所有子node都被碰過，black代表自己跟所有子node都被碰過了。

**(b)**

**1** max = 0;

**2** highest = higher = 0;

**3** 1.color = gray;

**4** 1.length = 0;

**5** Q = [];

**6** **ENQUEUE**(Q, 1);

**7** while (Q != [])

**8** i = **DEQUEUE**(Q);

**9** for (each j is node of i)

**10** if (j.color == white)

**11** j.color = gray;

**12** j.length = i.length + 1;

**13** **ENQUEUE**(Q, j);

**14** if (j.level > highest)

**15** highest = j.level;

**16** temp = j;

**17** for (each j is node of i)

**18** if (j.level > higher && j != temp)

**19** higher = j.level;

**20** if (i.length > highest)

**21** higher = i.length;

**22** else if (i.length > higher)

**23** higher = i.length;

**24** i.diameter = highest + higher;

**25** i.color = black;

**26** for (i = 1; i <= N; i++)

**27** if (max < i.diameter)

**28** max = i.diameter;

**29** return max;

利用BFS知道每一個node距離root多遠，然後把每一個node身邊所有跟他接在一起的node那條路徑的長度作比較，找出最大的兩個加在一起紀錄在diameter，diameter值就是這個node的diameter path的長度，最後把每一個node.diameter比較找出最大值就是答案。

**Problem1.3**

**(a)**

利用同心圓，假設每一個node皆在同心圓的邊上，同心圓的正中心就是midpoint，在同心圓的任一個邊上挑一個點，距離此點最遠的點必定由此點出發朝midpoint前進，並且落在同心圓最外面的邊上。

而diameter path是距離midpoint最遠的兩個點之間的連線，因此我們剛剛找到的那個距離midpoint最遠的點就是diameter path的其中一個末端點。

**(b)**

*section A*

previous[u] = father;

*section B*

**1** temp = b;

**2** if ((distance[b] + 1) % 2 == 1)

**3** while (distance[temp] != (distance[b]/2))

**4** temp = previous[temp];

**5** return temp;

**6** else

**7** while (distance[temp] != ((distance[b] + 1)/2))

**8** temp = previous[temp];

**9** return (temp, previous[temp]);

**Problem2.1**

**(a)**

*Quick Sort*在worst case時的time complexity是O(n2)，執行*Quick Sort*時需要從未排列的element中選一個pivot，並將剩下的element依照一邊大於一邊小於的方式分類(等於的情況就自行決定要固定放哪一邊即可)，若每次挑選到的pivot在分類的過程中都只偏重其中一邊，也就是其他element可能全部都是大於或全部都是小於，因此此時的分類根本就在耍廢沒有甚麼效果，而time complexity是O(n2)。

**(b)**

在best case的情況下，也就是所有數字都已經排列好的情況，*Insertion Sort*只需要O(n)，而*Merge Sort*需要O(nlogn)，此時*Insertion Sort* can run faster than *Merge Sort*。

**Problem2.2**

**1** array[k + 1] = {0};

**2** for (i = 0; i < n; i++)

**3** array[ n[i] ]++;

**4** for (i = 1; i <= k; i++)

**5** array[i] = array[i] + array[i - 1];

**6** if (a == 0)

**7** return array[b] - array[0];

**8** else

**9** return array[b] - array[a - 1];

n[i]就是題目給的n個數字中的第i個數字，

**Problem2.3**

**(a)**

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)**

n:代表總element數( = 7)、k:代表最大可能到的element數( = 2345)

r:代表以什麼進位( = 10)

*Radix Sort:*O((n + r) \* (logrk / 1))

*Counting Sort:*O(n + k)

After comparing the computational cost between these two algorithms, we can know that *Radix Sort* runs much faster than *Counting Sort*.

**Problem2.4**

*Merge Sort:*

個位數：

20 29 57 / 37 36 50 59

20 / 29 57 / 37 36 / 50 59

20 / 57 29 / 36 37 / 50 59

20 57 29 / 50 36 37 59

20 50 36 57 37 29 59

十位數：

20 50 36 / 57 37 29 59

20 / 50 36 / 57 37 / 29 59

20 / 36 50 / 37 57 / 29 59

20 36 50 / 29 37 57 59

result:20 29 36 37 50 57 59

*Heap Sort:*

result:29 20 36 37 59 57 50

No, they are different. Because *Heap Sort* is unstable.

**Problem2.5**

*Bucket Sort*

**Problem3.1**

(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)

**Problem3.2**

*MAIN()*

**1** count[N] = {1};

**2** for (i = 1; i < N; i++) *//因為總共只有N-1個edge*

**3** if (E[i].color == black)

**4** father\_A = **FIND\_SET**(E[i].node\_A);

**5** father\_B = **FIND\_SET**(E[i].node\_B);

**6** count[father\_A] += count[father\_B];

**7** father\_B.parent = father\_A;

**8** temp = 0;

**9** for (i = 1; i <= N; i++)

**10** if (FIND\_SET(i) == 0)

**11** root[temp] = i;

**12** temp++;

**13** ans = 0;

**14** for (i = 0; i < temp; i++)

**15** for (j = i; j < temp; j++)

**16** for (k = j; k < temp; k++)

**17** ans += count[ root[i] ] \* count[ root[j] ] \* count[ root[k] ];

*FIND\_SET(i)*

**1** if (i.father == 0) return i;

**2** else return **FIND\_SET**(i.father);