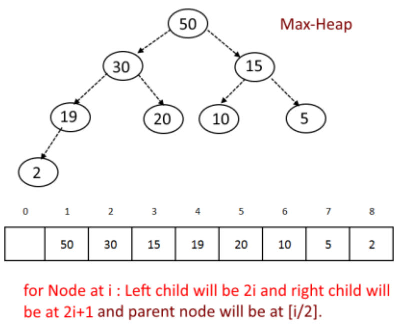
B05611047 化學三 鍾宜樺

Problem 1 Heaps! More Heaps!

1-1 Find-Greater(v) in O(k)

假設樹如下圖：



欲找到比 19 大的總 element 個數，則先檢查 root(必定為最大)，接著用遞迴檢查左子樹的值，30>19，繼續檢查其左子樹，19 == 19，找到一樣的數字了，則停下來，又因為為 max_heap ，因此 19 之子樹值比小於 19，因此不用往下檢查，於是開始檢查右子樹，30 的右子樹 20，比 19 大，繼續向下檢查，然而發現，20 是子葉，因此停下，再往上檢查 50 的右子樹，然而 15<19 不合，同前面說法，因為 max heap，因此 15 下面的子樹必小於 15，又 15<19，因此也不用繼續向下檢查。得比 15 大的數有 50, 30, 19, 20，return 4。而時間複雜度，最糟糕的情況最多也只會減達 2k 次(v 所在位置在樹的最後一格的情況)。因此可以得之時間複雜度為 O(2k) = O(k)。

Code:

```
int Find_Greater(int heap [],int v, int root_id, int heap_sz){
    //max heap
    int k = 0; // the num larger than v, v != id
    int l_id = root_id*2 + 1;
    int r_id = root_id*2 + 2;
    if ( heap[root_id] <= v)
        return 0;
    if(heap[root_id] > v){
        k++;
        //繼續找下去
        if (l_id < heap_sz )
            k += up_to_down_heapify(heap, v, l_id, heap_sz);
        if (r_id < heap_sz)
            k += up_to_down_heapify(heap, v, r_id, heap_sz);
    }
    return k;
}
```

1-2 delete(id)

Delete a node，做法先將 id(th) node 的值和整個 heap 的 Last node value 做交換。之後刪掉最後一個 node，時間為 const time。接著對第 id(th) 的 node 重新尋找他在 max heap 中正確的位置。又這個時間複雜度和第 id(th)的子樹高度成正比，因此 total time complexity = O(1) + O(h)，最糟的狀況，當 id = 0，也就是刪掉 root 的時候，h = logN 時，時間複雜度為 O(1)+O(logN) = O(logN) 。

Code:

```
void Delete (int heap[], int del_id, int heap_sz){
    heap [del_id] = heap[heap_sz -1];
    heap[heap_sz] = NULL;
    heap_sz --;
    up_to_down_heapify(heap, del_id,heap_sz);
}
void up_to_down_heapify (int heap [], int root_id ,int heap_sz){
    int max = root_id;
    int tmp;
    int l_id = root_id*2 + 1;
    int r_id = root_id*2 + 2;
    if (l_id < heap_sz && heap[l_id] > heap[max])
        max = l_id;
    if (r_id < heap_sz && heap[r_id] > heap[max])
        max = r_id;
    if (max != root_id) {
        tmp = heap[root_id];
        heap[root_id] = heap[max];
        heap[max] = tmp;
        up_to_down_heapify(heap, max, heap_sz);
    }
}
```

1- 3 median heap implementation

作法：

利用 max heap 和 min heap 兩個 heap 製作出 median heap。讓兩個 heap 都保持在擁有接近 n/2 個 data 的狀況下。

Median():

欲取得當下的 median，假若，min heap 的 data 數>max heap 的 data 數，則 median 為 min heap’s root。若個數差反之，則 median 為 max heap’s root。假若個數相等時，為根據題目定義為(n/2)th 也就是取小的，即比較兩 heap 的 root，取較小值為 median。因為整個 function 中最糟的狀況為 O(2)仍為 const time，因此時間複雜度仍為 O(1)。

Insert():

最一開始的 input data 我放在 min heap，又在這之後(min_sz != 0 && max_sz != 0 的時候)，每次要 input data = x 進去，若 x > median，則 x 放入 min heap，之後對 min heap 重新做 heapify 找到 x 在 min heap 中正確的位置後，如同最一開始所說，須讓兩個 heap 都維持在 data 個數都最接近 n/2 的狀態，因此這時候要做一個 rebalance 的動作，讓兩個 heap 的數字保持個數 delta 只差 1，且此時，欲移動的 data 是 min heap[0] 移到 max heap 裡面，之後再對 max heap 作處理，讓其仍為 max heap，整個結束才是完成一個 Insert。反之則放入 max heap(又因為 data 都是 unique 不可能等於)，後面同理。

時間複雜度：

假若兩個 heap 都為空，insert 則為 const time。假若兩個 heap 都有 data。討論 Worst case，假設 min_sz = k, max_sz = k-1, insert_data > median，則 insert_data 須被加入 min_heap 中，Insert 一開始加在最下面開始向上換，最糟的狀況可能被換到 min_heap 的 root (即 median = max_heap’s root 此時)，所花費的時間和 min heap 的高度成正比為 O(log(k))。然而，這時候 min_sz 會變成 k+1，造成 min_sz – maz_sz >1，因此需要作 rebalance，將 min heap 的 root 移動給 max heap，這個動作會再花 O(logk)的時間，最後，被 insert 到 heap 的這個 data，已知此 data > max_heap_root 如前所述，因此他一定會被換到 max heap 的 root，又所花的時間和 max heap 的高度成正比為 O(logk)，因此可之總花費時間為 O(logk)+ O(logk)+ O(logk) = O(3logk) = O(logk)。

Extract-Median()：

這個 function 其實在 Insert 的時候就用到了，就是在發現兩個 heap 的 size 相差超過 1 時，要將擁有較多 data 的 heap 的 root 移到另一個 heap 時會使用到。

做法：

先保留 root data 後，將 heap 中最後一個 data 的值壓過去 root data，最後刪掉 heap 的最後一個 node。然而這個時候的 min (max) heap 不一定是 min (max) heap，因此需要重新作 heapify，worst case 就是從第一層換到最後一層，因此所花費的時間和 heap 的高度成正比，最糟也是 O(logN)。

Code:

```
/*
Ref：https://stackoverflow.com/questions/15319561/how-to-implement-a-median-heap/15319593
*/

#include <stdio.h>
#include <stdlib.h>
int min_sz = 0;
int max_sz = 0;

//O(1)
int Median(int min_heap[], int max_heap[]) {
    if ( min_sz > max_sz){
        return min_heap[0];
    }
    else if (min_sz < max_sz){
        return max_heap[0];
    }
    else { // ==，取(n/2)th，取值小的
        if (min_heap[0] < max_heap [0]){
            return min_heap[0];
        }
        else {
            return max_heap[0];
        }
    }
}
// Insert 部分開始
void Insert_min_heap (int heap [], int value){
    heap[min_sz] = value;//置入
    min_down_to_up_heapify(heap, min_sz);
    min_sz ++;//new heap_sz
}
void min_down_to_up_heapify (int heap [],int min_sz){

    int parent = ( min_sz -1 ) /2;
    int tmp;
```

```
    if (min_sz == 0){
        return;
    }
    if ( heap[min_sz] > heap[parent]){
        return;
    }
    else{ //(heap[heap_sz] < heap [parent])
        tmp = heap[min_sz];
        heap[min_sz] = heap[parent];
        heap[parent] = tmp;
        min_down_to_up_heapify(heap, parent);
    }
}
void Insert_Max_heap (int heap [], int value){
    heap[max_sz] = value;
    max_down_to_up_heapify(heap, max_sz);
    max_sz ++;
}
void max_down_to_up_heapify (int heap [], int max_sz){
    int parent = ( max_sz -1 ) /2;
    int tmp;
    if (max_sz == 0){
        return;
    }
    if ( heap[max_sz] < heap[parent]){
        return;
    }
    else{ //(heap[heap_sz] > heap [parent])
        tmp = heap[max_sz];
        heap[max_sz] = heap[parent];
        heap[parent] = tmp;
        max_down_to_up_heapify(heap, parent);
    }
}
void Delete_min_root (int heap [], int heap_sz){
   heap[0] = heap[heap_sz];
   heap[heap_sz]= NULL;
   min_up_to_down_heapify(heap, 0, heap_sz);
```

```
}
void min_up_to_down_heapify (int heap [], int root_id ,int heap_sz){
    int min = root_id;
    int tmp;
    int l_id = root_id*2 + 1;
    int r_id = root_id*2 + 2;

    if (l_id < heap_sz && heap[l_id] < heap[min])
        min = l_id;
    if (r_id < heap_sz && heap[r_id] < heap[min])
        min = r_id;
    if (min != root_id) {
        tmp = heap[root_id];
        heap[root_id] = heap[min];
        heap[min] = tmp;
        min_up_to_down_heapify(heap, min, heap_sz);
    }
}
void Delete_max_root (int heap [], int heap_sz){
   heap[0] = heap[heap_sz];
   heap[heap_sz]= NULL;
   max_up_to_down_heapify(heap, 0, heap_sz);
}
void max_up_to_down_heapify (int heap [], int root_id ,int heap_sz){
    int max = root_id;
    int tmp;
    int l_id = root_id*2 + 1;
    int r_id = root_id*2 + 2;

    if (l_id < heap_sz && heap[l_id] > heap[max])
        max = l_id;
    if (r_id < heap_sz && heap[r_id] > heap[max])
        max = r_id;
    if (max != root_id) {
        tmp = heap[root_id];
        heap[root_id] = heap[max];
        heap[max] = tmp;
        max_up_to_down_heapify(heap, max, heap_sz);
```

```c
    }
}
void Rebalance(int min_heap[], int Max_heap[], int type ){
    //type1 : min_heap 多 2
    if (type == 1){
        Insert_Max_heap(Max_heap, min_heap[0]);
        min_sz --;
        Delete_min_root(min_heap, min_sz);
    }
    if (type == 2){
        Insert_min_heap(min_heap, Max_heap[0]);
        max_sz --;
        Delete_max_root(Max_heap, max_sz);
        }
}
//O(logN)
void Insert(int min_heap[], int max_heap[], int x){
    printf("into insert\n");
    int med;
    //init
    if (max_sz == 0 && min_sz == 0){
        min_heap[0] = x;
        med = x;
        min_sz ++;
    }
    else{
        med = Median(min_heap,max_heap);
        printf("Median = %d\n",med);
    }
    //不會有 == 因為都是 unique
    if ( med > x ){ // 若 x 比較小，x 放進 max heap
        Insert_Max_heap( max_heap , x );
    }
    if (med < x){
        Insert_min_heap( min_heap, x );
    }
    int type;
    if (min_sz - max_sz > 1){
```

```
        type = 1;
        Rebalance( min_heap, max_heap,  type);
    }
    if (max_sz - min_sz > 1){
        type = 2;
        Rebalance( min_heap, max_heap,  type);
    }
}
// Insert 部分結束
//O(logN)
void Extract_Median(int min_heap[], int max_heap[], int min_sz, int
max_sz) {
    //median 在 min_heap 中
    if (min_sz > max_sz){
        Delete_min_root(min_heap,min_sz);
    }
    //在 max 中
    else if( max_sz > min_sz){
        Delete_min_root(max_heap, max_sz);
    }
    else {
        //在 min 中
        if (min_heap[0] < max_heap [0]){
            Delete_min_root(min_heap,min_sz);
        }
        else {
            Delete_min_root(max_heap, max_sz);
        }
    }
}
//以下的 main  function 中的 printf 純粹為在本機中測試上面的 function 是否能
work 用的 QQ

int main()
{
    int tot_num,data;
    int min_heap[tot_num];
    int max_heap[tot_num];
    scanf("%d", &tot_num);
```

```
    printf("%d\n", tot_num);
    for (int i = 0; i < tot_num; i++){
        scanf("%d", &data);
        printf("%d\n", data);
        Insert(min_heap,max_heap,data);
        printf("no. %d times \n", i);
        printf("min_heap_sz = %d: \n", min_sz);
        for (int i = 0; i < min_sz; i++){
            printf("%d = %d\n",i,min_heap[i]);
        }
        printf("max_heap_sz = %d : \n", max_sz);
        for (int i = 0; i < max_sz; i++){
            printf("%d = %d\n",i,max_heap[i]);
        }
        printf("---------------------\n");
    }
    return 0;
}
```

Problem 2 Algorithmic Complexity Attack (2) - Hash Table and Function

2-1-a

做法：如題目已知 hash(x) = xmodk，因此在作業上給的網址，開始 try & error

hash(9000000000000000000) = 9000000000000000000(不夠大，要增大)

hash(10000000000000000000) = 7766279631452241920 (爆了，要縮小)

hash(3000000000000000000) = 694156990786306049

(一路一路慢慢試...)

hash(2305843009213693951) = 0

2-1-b 欲發生 collision，就須要讓所有的不同 input key 卻得到一樣的 value，如前一題已知道 hash 所使用的 k 是 2305843009213693951。

因此要給出 10^6 個 valid key-value pairs to be inserted into the dictionary，第一個 data 取 x(key = 1)，則第二個 data 則取 kx(key = 2)，則這兩個不同的 key 在 dictionary 中取 mode k 後，就可以達成不同的 key 卻得到相同的 value 的值，就代表發生了 collision。然而因為須要有 10^6 個數字，電腦數字會存爆，因此取到 x*k^m (此時的 key 為 m+1)

的電腦能容忍最大值後，開始取(x+1)最為第 m+2，而 key = m+2 之 value 便取(x+1)*k，同理下一個 value 一樣在乘以一個 k…，最後一路同理取到 10^6 個。就會發生很多很多次的 collision。

2-2

2-2-a (Ref : http://users.ece.utexas.edu/~adnan/360C/hash.pdf)

Theorem 2 In a hash table in which collisions are resolved by chaining, a successful search takes $\Theta(1+\alpha)$ time on average, assuming simple uniform hashing.

Proof: Assume that the search is equally likely to be any of the n keys, and that inserts are done at the end of the list.
Expected # of elements examined = 1 + # elements examined when sought after element was inserted.
Take average over the n elements of 1 + expected length of list to which the i-th element was added.
The expected length of list to which i-th element is added is $(i-1)/m$

$$\begin{aligned}(1/n)\cdot\Big(\sum_{i=1}^{n}(1+(i-1)/m)\Big) &= 1+\frac{1}{m\cdot n}\cdot\Big(\sum_{i=1}^{n}(i-1)\Big)\\ &= 1+\frac{1}{m\cdot n}\cdot\Big(\frac{n\cdot(n-1)}{2}\Big)\\ &= 1+\frac{\alpha}{2}-\frac{1}{2\cdot m}\\ &= \Theta\Big(1+\frac{\alpha}{2}-\frac{1}{2\cdot m}\Big)\end{aligned}$$

Hence overall complexity is $\Theta(1+\frac{\alpha}{2}-\frac{1}{2\cdot m})=\Theta(1+\alpha)$. ■
Think about the case where $\alpha=1$, when $\alpha \ll 1$, and when $\alpha \gg 1$.

，又$\alpha = 0.8$，所求 = 1.8 次。

2-2-b (Ref:

https://www.cs.oberlin.edu/~bob/cs151.spring17/Class%20Examples%20and%20Notes/April/April%205/HashMapAnalysis.pdf)

If you assume that the data in a hash table is randomly distributed, then the probability that any particular cell is occupied is λ and the probability it is unoccupied is 1-λ. The probability that the first location a linear probe tests is unoccupied is 1-λ. The probability that the first is occupied and the second is free is λ*(1-λ). The probability that the first two are occupied and the third is free is λ*λ*(1-λ). Altogether the expected number of probes we need under the assumption of complete randomness is

ENP = 1*(1-λ) + 2*λ*(1-λ) + 3*λ*λ*(1-λ) + ...

ENP = 1*(1-λ) + 2*λ*(1-λ) + 3*λ*λ*(1-λ) + ...
= (1-λ) *[1 + 2*λ + 3*$λ^2$+ 4*$λ^3$ + ...]

Let S be the portion of this in square brackets:
S = 1 + 2*λ + 3*$λ^2$+ 4*$λ^3$ + ...
Then λ*S = λ + 2*$λ^2$ + 3*$λ^3$+ 4*$λ^4$ + ...

If we subtract these we get
S - λ*S = 1 + λ + $λ^2$+ $λ^3$ + ...
This is a geometric series; it sums to 1/(1-λ)
So S - λ*S = 1/(1-λ)
S(1-λ) = 1/(1-λ)
S = 1/$(1-λ)^2$
ENP = (1-λ) *S = 1/(1-λ)

因此所求 = 1/(1-0.6) = 1/0.4 = 2.5 次

2-3

(a) even part

11BE 3AEF | 951C A6B4 | 4400 6FC0 | 478D 9205

① 11BE xor 3AEF :

0001 0001 1011 1110
0011 1010 1110 1111
0010 1011 0101 0001
2 B 5 1

② 951C ^ A6B4

1001 0101 0001 1100
1010 0110 1011 0100
0011 0011 1010 1000
3 3 A 8

③ 4400 ^ 6FC0

0100 0100 0000 0000
0110 1111 1100 0000
0010 1011 1100 0000
5 B C 0

④ 478D ^ 9205

0100 0111 1000 1101
1001 0010 0000 0101
1101 0101 1000 1000
D 5 8 8

∴ YAG(y) = (2B51 33A8 5BC0 D588) hex ※

(b) YAG(z) = YAG(y) = 同上，將YAG(y)分成 4 parts 討論，

2B51 → First part，

從 (a) 可知 2B51 轉成二進位会是16碼 & 16碼 XOR，∴簡化討論，
再將 2B51 分成 4 個 parts，∴首先討論，欲經YAG transfer後得"2"
的机率：

假設 abcd xoR efgh = $\boxed{ijkl}$ (二進位) = (2) hex = 0010

(abcd)的可能性：可知有 $2^4=16$ 种，efgh同理也有 16种，
又欲產生 xoR 後得 0010，則可知，若 a=0 時則 e=0，a=1時，e=1，
xoR 後才可產生(得) 0，同理b亦相关，…… 因此只須討論 abcd，
則 efgh 会連同改變，∴ abcd 可能產生的組合有 $2^4=16$ 种，即
總計 abcd xoR efgh = 0010 的方式，有 16 种 ∴ collision 的机率 $= \frac{16}{256}\left(=\frac{1}{16}\right)$，
∴可知欲得 YAG(z) = YAG(Y) 的机率 $= \left(\left(\frac{1}{16}\right)^4\right)^4 = \left(\frac{1}{2^4}\right)^{16} = \frac{1}{2^{64}}$ ※

(c) YAG(a) = YAG(b)，從 (b) 可知任一 YAG(y) collision 的机率 $= \frac{1}{2^{64}}$，
∴無論 a 長怎樣，欲得 YAG(a) = YAG(b) 之率和 (a) 相同，∵每個
16-byte-long file 發生机率是一樣的 (Randomly generate)。

(d) ∵ collision 的机率很小，∴為好的 hash function，

Problem 3 Disjoint Set

3-1

```
/*
假設所有 node 都是 unique
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#define MAX_N (某個數字，隨著每次 node data 的最大值而改變)
struct Node {
    struct Node * parent;
    int data;
    int set_min;
    int rank;
};
struct Node * create_Node (void){
    struct Node * New_node = (struct Node*)malloc(sizeof(struct Node));
    if(New_node == NULL)
        exit(1);
    return (New_node);
}
struct Node * Node_Array [MAX_N];
void Make_Set (int new_data){
    struct Node * tmp = create_Node();
    tmp->data = new_data;
    tmp->parent = tmp; //parent 連到自己
    tmp->set_min = new_data;
    tmp->rank = 0;
    Node_Array[new_data] = tmp;//存入 node array 這樣等等比較好 call
}
void Union(int x, int y){
    struct Node * n_x = Node_Array[x];
    struct Node * n_y = Node_Array[y];
    Link(Find_Set(n_x),Find_Set(n_y));
};
void Link(struct Node * x, struct Node * y){
```

```
    if (x->rank > y->rank){
        y->parent = x;
    }
    else {
        x->parent = y;
        if (x->rank == y->rank){
            y->rank = y->rank + 1;
        }
    }
    // min_element 只有 head 會對
    if (x->set_min > y->set_min)
        x->set_min = y->set_min;
    else
        y->set_min = x->set_min;
}
Struct  Node *  Find_Set(struct Node * x){
    if (x->parent != x)
        x->parent = Find_Set(x->parent);
    return x->parent;
};
int Min_element(int k){
    struct Node * tmp = Node_Array[k];
    tmp = Find_Set(tmp);// 回傳每個 set 的頭
    return tmp->set_min;
};
int main()
{
}
```

3-2 Isolate, same_set

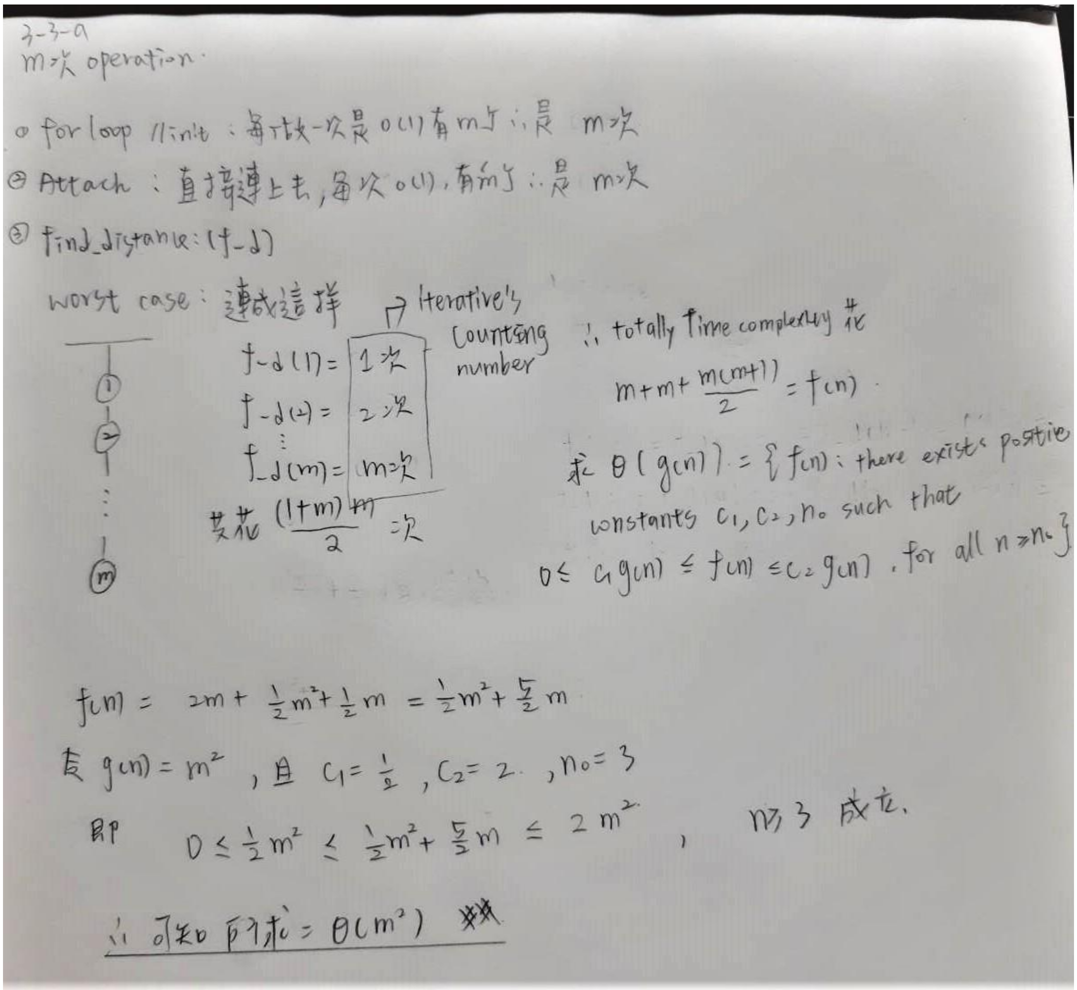
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#define MAX_N (某個數字，隨著每次 node data 的最大值而改變)

struct Node {
```

```c
    struct Node * parent;
    int data;
    int rank;
};
struct Node * create_Node (void){
    struct Node * New_node = (struct Node*)malloc(sizeof(struct Node));
    if(New_node == NULL)
        exit(1);
    return (New_node);
};

struct Node * Node_Array [MAX_N];
void Make_Set (int new_data){
    struct Node * tmp = create_Node();
    tmp->data = new_data;
    tmp->parent = tmp; //parent 連到自己
    tmp->rank = 0;
    Node_Array[new_data] = tmp;//存入 node array 這樣等等比較好 call
}
void Union(int x, int y){
    struct Node * n_x = Node_Array[x];
    struct Node * n_y = Node_Array[y];
    Link(Find_Set(n_x),Find_Set(n_y));
};
void Link(struct Node * x, struct Node * y){
    if (x->rank > y->rank){
        y->parent = x;
    }
    else {
        x->parent = y;
        if (x->rank == y->rank){
            y->rank = y->rank + 1;
        }
    }
}
bool jdg;
int dat;
struct Node * Find_Set(struct Node * x){
```

```
    dat = 0;
    if (x->data != -1)
        dat = x->data;
    jdg = false;
    //若讀到 -1 == -1 則要回傳前一個
    if (x->parent != x){
        x->parent = Find_Set(x->parent);
    }
    if(x->parent->data == -1 && x->data == -1){
        jdg = true;
    }
    if (jdg == false)
        return x->parent;
    else // 讀到 -1 == -1，代表頭之前被刪掉了
        return Node_Array[dat];
};

//跟上面不一樣的是要更新 rank
struct Node *Find_Set_for_delete(struct Node * x){
    dat = 0;
    if (x->data != -1)
        dat = x->data;
    jdg = false;

    //去除之前被刪掉的點
    if (x->parent->data == -1){
        //遇到之前刪中間的狀況，重新接頭
        x->parent = x->parent->parent;
        //遇到之前刪掉頭的狀況
        if (x->parent->parent->data == -1){
            x->parent = x;
        }
    }
    //更新 rank
    x->rank --;
    if (x->parent != x){
        x->parent = Find_Set_for_delete(x->parent);
    }
```

```
    if(x->parent->data == -1 && x->data == -1){
        jdg = true;
    }
    if (jdg == false)
        return x->parent;
    else
        return Node_Array[dat];
};

void Isolate(int k){
    struct Node * del_node = Node_Array[k];
    del_node->data = -1; //將 data 設為-1，之後找 parent 遇到-1 時，就繼續往
上找

    // 純粹更新 rank 用
    struct Node * head = Find_Set_for_delete(del_node);

    //Node_Array 裡面的 Node 也會被新的 Node 蓋掉
    Make_Set(k);//重新 new 這個 data 成一個新的 Node
}

bool Same_Set (int x, int y){
    struct Node * x = Node_Array[x];
    struct Node * y = Node_Array[y];
    if (Find_Set(x) == Find_Set(y))
        return true;
    else
        return false;
}
```

3-3-a



3-3-a

m次 operation.

① for loop //init：每做一次是 O(1)，有 m 个 ∴ 是 m 次

② Attach：直接連上去，每次 O(1)，有 m 个 ∴ 是 m 次

③ find_distance：(f-d)

worst case：連成這樣 → Iterative's counting number

f-d(1) = 1次

f-d(2) = 2次

⋮

f-d(m) = m次

共花 $\frac{(1+m)m}{2}$ 次

∴ totally time complexity 花 $m+m+\frac{m(m+1)}{2} = f(n)$

求 $\Theta(g(n)) = \{ f(n)$: there exists positive constants c_1, c_2, n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$, for all $n \ge n_0 \}$

$f(m) = 2m + \frac{1}{2}m^2 + \frac{1}{2}m = \frac{1}{2}m^2 + \frac{5}{2}m$

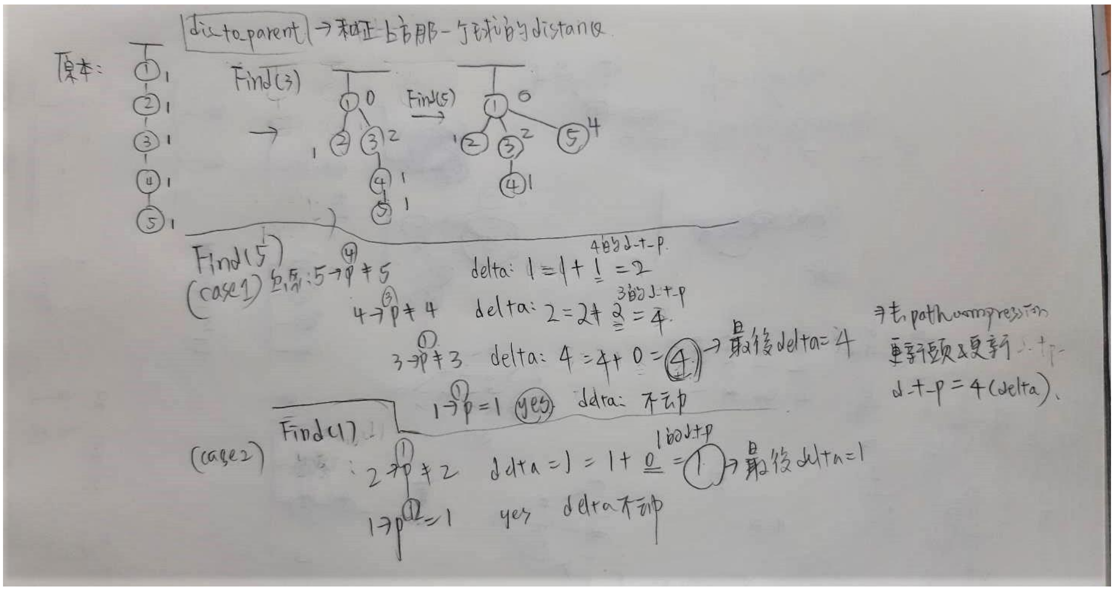
令 $g(n) = m^2$，且 $c_1 = \frac{1}{2}$，$c_2 = 2$，$n_0 = 3$

即 $0 \le \frac{1}{2}m^2 \le \frac{1}{2}m^2 + \frac{5}{2}m \le 2m^2$，$m \ge 3$ 成立.

∴ 可知所求 $= \Theta(m^2)$

3-3-b implement path compression

假設圖：



dis_to_parent → 和正上方那一个點的distance

原本：

Find(3) →

Find(5) →

Find(5)

(case1) 跑 5→p≠5 delta: $1 = 1 + \underset{4\text{的d-t-p}}{1} = 2$

4→p≠4 delta: $2 = 2 + \underset{3\text{的d-t-p}}{2} = 4$

3→p≠3 delta: 4 = 4 + 0 = 4 → 最後delta=4

1→p=1 yes delta: 不動

→ 去 path compression 重新頭&更新 d-t-p = 4 (delta).

Find(1)

(case2) 2→p≠2 delta=1 = $1 + \underset{1\text{的d-t-p}}{0}$ = 1 → 最後delta=1

1→p=1 yes delta不動

```c
/*
假設所有 node 都是 unique
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#define MAX_N (某個數字，隨著每次 node data 的最大值而改變)
struct Node {
    struct Node * parent;
    int data;
    int dis_to_par;
};
struct Node * create_Node (void){
    struct Node * New_node = (struct Node*)malloc(sizeof(struct Node));
    if(New_node == NULL)
        exit(1);
    return (New_node);
};
struct Node * Node_Array [MAX_N];
void Make_Set (int new_data){
    struct Node * tmp = create_Node();
    tmp->data = new_data;
    tmp->parent = tmp; //parent 連到自己
    tmp->dis_to_par = 0;
    Node_Array[new_data] = tmp;//存入 node array 這樣等等比較好 call
}
void Attah(int x, int y){
    struct Node * dn = Node_Array[x]; // 連在下面
    struct Node * up = Node_Array[y]; //連在上面
    dn->parent = up;
    dn->dis_to_par = 1;
}
int Find_dis(int x){
    struct Node * tmp = Node_Array[x];
    struct Node * head = Find_Set(x);
    path_com(tmp, head); // 更新 parent, 更新 d_t_p distance to parent
    return tmp->dis_to_par + 1;
```

```
}
int del;
Struct  Node *  Find_Set(struct Node * x){
    // x->dis_to_par 此時 == 1
    del = 1; // 因為原本 = 1
    if (x->parent != x){
        del += x->parent->dis_to_par;
        x->parent = Find_Set(x->parent);
    }
    return x->parent;
};
void path_com (struct Node * x, struct Node *head){
    //更新距離 //更新頭
    x->parent = head;
    x->dis_to_par = del;
}

int main(){
    return 0;
}
```