B05611047 化學三 鍾宜樺

Problem 1 Handsome and his Mysterious Jets

1-1.

(0,2), (0,3), (1,2), (1,3), (2,3)

1-2. find the numbers of feasible hops in O(N+M) times

(都從西到東)

假設：

西山高：1,1,2,3,5,7,9,無限大 (N = 8)

東山高：1,2,4,5,6,8,10,12,無限大 (N = 9)

//無限大是自己加上去的。

第一輪：

兩個 array 都分別從 idx = 0 開始走

比較東西 ptr 所指山高的大小，發現 1 == 1(西山高 == 東山高)

第二輪：

Ptr_東山向後移動到 idx =1，發現 1<2(西山高<東山高)

可以 hop，hops+=東山總數-ptr 所指之 idx，因為山 sort 過，東山高度一定越來越大，因此 hops 都成立。

且接著 ptr_西山向後移動移動到 idx =1;

第三輪：(同第二輪)

第四輪：ptr_西山 = 3，ptr_東山 = 1，發現 2 == 2(西山高 == 東山高)

因此移動 ptr_東山，到 idx = 3，2 < 4 可以 hop，因此 hops+=東山總數-ptr 所指之 idx

第五輪：(同第二輪)

同理，一直移動到其中一 ptr 指到無限大的時候，就代表山走完了，hops 也算完了，跳出 while，印出 hops 數。

```c
#include <stdlib.h>
#include <stdio.h>
int main(){
    //N-> west, M-> east
    int N,M;
    scanf("%d %d", &N, &M);
    int arr_1[] = N+1;
    int arr_2[] = M+1;
    arr_1[N] = INT_MAX;
    arr_2[M] = INT_MAX;
    int count = 0;
    while ( count != N ){
```

```
        scanf("%d", &arr_1[count]);
        count++;
    }
    count = 0;
    while( count != M ){
        scanf("%d", & arr_2[count]);
        count ++;
    }

    int tmp = 0, ptr = 0, hops = 0;
    while (arr_1[tmp] != INT_MAX && arr_2[ptr] != INT_MAX ){
        if (arr_1[tmp] >= arr_2[ptr])
            ptr++;
        else{ //arr_2 > arr_1's value
            hops += (M-ptr);
            tmp++;
        }
    }
    printf("所求 = %d\n", hops );
    return 0;
}
```

1-3.Calculate the number of feasible flying hops in O(N lgN) time.

```
/*
因為題目要求須要在 nlgn 的時間內做完，因此選擇 merge sort 做為 sort 的方式。
想法：
在做 merge sort 的時候計算，對於每一個數值而言，有多少個值比他大的數值，從其左邊在 sort 的過程中被移到 (因為本數值較大，故被往後移動) 這個數字的右邊。
舉例來說，假設一個 seq 原本的長像是：3,8,1,6,10。
經過 sort 後長得像：1,3,6,8,10，就可以知道對於數值 1 而言，他最多可以有 4 個(因為在他後面有 4 個數字比他大)合法的 hops。
然而比對原本的 seq，會發現到在 1 前面有兩個數值比 1 的數字。因此對於 1 而言它真正的合法 hops 只有`4-2 = 2 ，也就是最多只有兩個合法 hops。
因此，為了獲得這個[ 有多少比他大的數字原本排在他前面 ] 這個值。可以先開一個用來計數數值的 int invalid_hops = 0，在做 merge sort 的時候就可以在每一次交換，去計算哪ˇ個數字被往前移了 x(正值)格。(只記往前移的就好，因為和往後移動(負值) sigma = 0)，
-
```

就對 invalid_hops 的值進行的 +x (因為一定對於一個比較小的數值來說，他減少了 x 個合法的 hops)。換言之就是在一次交換，有 X 個比他大的數字被往後移動了，因此合法的 hops 數要-X。
因此在做完整個 merge sort 之後，就可以獲得所有不合法的 hops 數 = invalid_hops。
又任取兩數必有一數值另一個小(因為題目說每座山的山高都是 unique 的)，因此合法的 hops 數 = C(所有山的數量)取 2 - invalid_hops，即為所求。
花費時間：
在做 merge sort 的時候會花 O(NlgN)，而做紀錄是 O(1)，
又最後計算 valid_hops 是一條家法算式也是 O(1)，
因此總花費時間是 = O(NlgN) + O(1) + O(1) = O(NlgN)。

*/

```
#include <stdlib.h>
#include <stdio.h>
#include  <limits.h>

int invalid_hops = 0;

void merge(int arr[], int l, int m, int r)
{
//    printf("in\n");
//    printf("l = %d, m = %d, r = %d\n",l,m,r );
    int i, j, k;
    int n1 = m - l + 1;
    int n2 =  r - m;

    /* create temp arrays */
    int L[n1], R[n2];

    /* Copy data to temp arrays L[] and R[] */
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1+ j];

    /* Merge the temp arrays back into arr[l..r]*/
    i = 0; // Initial index of first subarray
```

```
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    int con_in_right = 0;
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            con_in_right = 0;
//            printf("in left \n");
//            printf("arr[%d] = %d, L[%d] = %d\n",k,arr[k],i ,L[i]);
            arr[k] = L[i];
//            printf("l+i-k  = %d\n",l+i-k );
//            invalid_hops += ( l+i-k );
            i++;
        }
        else
        {
//            printf("con_in_right = %d\n",con_in_right);
//            printf("in right \n");
//            printf("arr[%d] = %d, R[%d] = %d\n",k,arr[k], j,R[j]);
            arr[k] = R[j];
//            printf("k-i+n1+j -k - con_in_right = %d\n",k-i+n1+j -k -
con_in_right);
            invalid_hops += ( k-i+n1+j -k - con_in_right);
            j++;
            con_in_right ++;
        }
        k++;
    }
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2)
    {
```

```
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        int m = l+(r-l)/2;

        // Sort first and second halves
//        printf("arr[l] =%d, arr[m] = %d\n",arr[l],arr[m]);
        mergeSort(arr, l, m);
//        printf("arr[m+1] =%d, arr[r] = %d\n",arr[m+1],arr[r]);
        mergeSort(arr, m+1, r);

        merge(arr, l, m, r);
    }
}


int main(){
    //先讀進來山的總數，和山高 array，且第一座山為 0
    //下面假設 N = 12, 且山高如下
    int N ;
    scanf("%d", &N);
    int MM_hts[N];
    //最多的時候是 C N 取 2
    int valid_hops = (N * (N-1))/2;

    for (int i = 0; i < N; i++){
        scanf("%d", &MM_hts[i] );
    }
    mergeSort(MM_hts, 0, N-1);
```

```
    printf("total_valid__hops = %d\n",valid_hops - invalid_hops);

return 0;

}
```

1-4. whether the number of feasible flying hops are odd number or not. Your algorithm should run in O(N lgN + Q) time.

```
/*
算法和前一題相同。
只是這一次要多一份 reverse 的時間，和最後需要 valid hops %2 去判斷是否為奇數。
時間複雜度
做一次：
reverse = O(R-L+1) worst case = O(N)
sort = O(NlgN)
判斷是否為奇數 = O(1)
做一次總時間 = O(N)+O(NlgN)+O(1) = O(NlgN)
又因為有 Q 比資料因此做 Q 次 O(QNlgN) = O(NlgN)//Q 是常數
*/
#include <stdlib.h>
#include <stdio.h>
#include  <limits.h>

int invalid_hops = 0;

void merge(int arr[], int l, int m, int r)
{
//    printf("in\n");
//    printf("l = %d, m = %d, r = %d\n",l,m,r );
    int i, j, k;
    int n1 = m - l + 1;
    int n2 =  r - m;

    /* create temp arrays */
    int L[n1], R[n2];

    /* Copy data to temp arrays L[] and R[] */
    for (i = 0; i < n1; i++)
```

```
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1+ j];

    /* Merge the temp arrays back into arr[l..r]*/
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    int con_in_right = 0;
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            con_in_right = 0;
//            printf("in left \n");
//            printf("arr[%d] = %d, L[%d] = %d\n",k,arr[k],i ,L[i]);
            arr[k] = L[i];
//            printf("l+i-k  = %d\n",l+i-k );
//            invalid_hops += ( l+i-k );
            i++;
        }
        else
        {
//            printf("con_in_right = %d\n",con_in_right);
//            printf("in right \n");
//            printf("arr[%d] = %d, R[%d] = %d\n",k,arr[k], j,R[j]);
            arr[k] = R[j];
//            printf("k-i+n1+j -k - con_in_right = %d\n",k-i+n1+j -k -
con_in_right);
            invalid_hops += ( k-i+n1+j -k - con_in_right);
            j++;
            con_in_right ++;
        }
        k++;
    }
    while (i < n1)
    {
        arr[k] = L[i];
```

```
        i++;
        k++;
    }

    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        int m = l+(r-l)/2;

        // Sort first and second halves
//        printf("arr[l] =%d, arr[m] = %d\n",arr[l],arr[m]);
        mergeSort(arr, l, m);
//        printf("arr[m+1] =%d, arr[r] = %d\n",arr[m+1],arr[r]);
        mergeSort(arr, m+1, r);

        merge(arr, l, m, r);
    }
}


int main(){
    //先讀進來山的總數，和山高 array，且第一座山為 0
    //下面假設 N = 12, 且山高如下
    int N ;
    scanf("%d", &N);
    int MM_hts[N];
    //最多的時候是 C N 取 2
    int valid_hops = (N * (N-1))/2;
```

```
    for (int i = 0; i < N; i++){
        scanf("%d", &MM_hts[i] );
    }
    int Q;
    scanf("%d", &Q);
    int Q_data[Q][2];
    int ct = 0;
    while (ct != Q){
        //先 reverse
        int left = Q_data[ct][0], right = Q_data[ct][1];
//        printf( "Q_data[%d][0] = %d,Q_data[%d][1] = %d\n
",ct,Q_data[ct][0],ct,Q_data[ct][1]);
        int chg_M_num = right - left + 1;
        for(int i = 0; i < chg_M_num; i++){
            chg_MM[i] = MM_hts[left + i];
        }
        for (int i = 0; i < chg_M_num; i++){
            MM_hts [left + i] = chg_MM [chg_M_num - i - 1];
        }
//        printf("Revs\n：");
//        for(int i = 0; i < N; i++)
//            printf("%d", MM_hts[i]);
//        printf("\n");
        mergeSort(MM_hts, 0, N-1);
        printf("total_hops = %d\n",total_hops);
        if( (valid_hops - invalid_hops) %2 != 0)
            printf("odd\n");
        else
            printf("not odd\n");

        ct++;
        invalid_hops = 0;    //歸零重算
    }

return 0;

}
```

Problem 2

2-1 Which of the following algorithms might be vulnerable to Algorithmic ComplexityAttackifalltheinputsarevalid? Explainyourreasonbyprovidingtheirworst-case and average-case time complexity.

	Worst case	Avg case
Quick Sort	O(N^2)	O(NlgN)
Merge Sort	O(NlgN)	O(NlgN)
Selection Sort	O(N^2)	O(N^2)
Insertion Sort	O(N^2)	O(N^2)

不想被 Dos 影響的話就是希望 Worst case 和 Avg case 的時間複雜度相同。他們不管 input 長怎樣所需要花的時間複雜度都一樣。因此 Merge Sort 、Selection Sort、 Insertion Sort 都可以。

2-2 please provide a valid input sequence to most effectively enforce Algorithmic Complexity Attack to the server, and briefly explain why your input sequence can achieve such attack.

Qs 是尋找一個 pivot，不斷地進行分堆達到 sort 的效果。然而若是取得的 privot 無法順利地分成兩堆大小差不多的分堆的時候，就會造成要一個一個比對造訪的狀況因此會到 O(N^2)。又題目說 qs 是取最右邊的值當作是 pivot，因此只要讓右邊的值無法將前面的數據進行分堆就可以達到 Dos 的效果。

又題目要求舉出一個例子 ：1,2,3,4,…,999999,1000000。

2-3 RANDOMIZED-QUICKSOR

<case1>

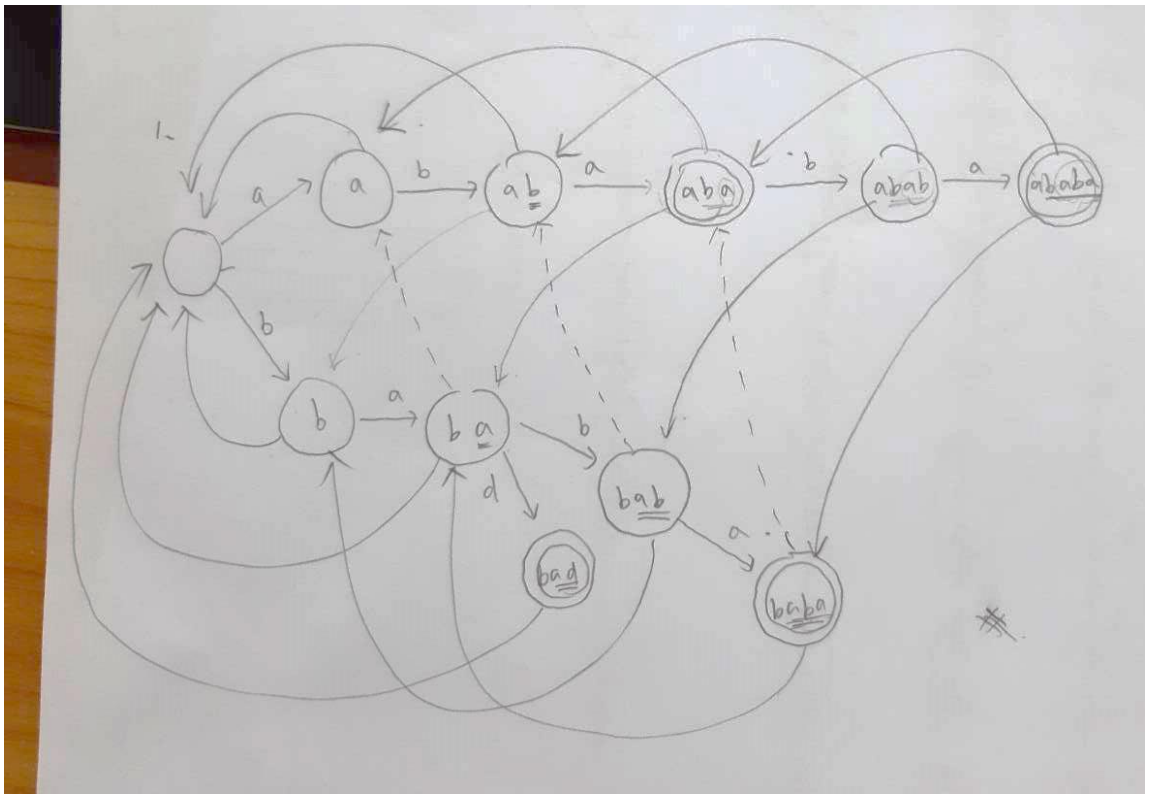
我不確定這題的 valid input 是不是也是要 unique positive integer，如果不是的話，則產生一個像是 1,1,…,1,1 這種整行全部都長一樣的數列即可，因為全部的值都一樣，即使使用 random 去求得 pivot 的值，仍然無法順利將其分堆成兩堆，因此可以產生 Dos 的狀況。

<case2>

假設 valid input 是要 unique positive integer，那麼就像前面一題所說的，只要 pivot 能夠產生無法分堆的狀況，那麼就可以讓時間達到 O(N^2)，因此所舉的例子也和前面相同：1,2,3,4,…,999999,1000000。因為這的 seq 已經 sort 好了，然而 server 並不會知道這件事情，因此還是會做 sort，然而就會發生和前一題一樣的狀況，就是不管 pivot 選到哪個數字都無法分堆，造成 O(N^2)的狀況。因此可達到 Dos 的效果。或者是像是：2,1,4,3,6,5,…,1000000,999999 這 seq 也可以達到類似的效果，無法進行順利的分堆，假設 Pivot 一開始取到 5，但最後一路跑到 6 的時候進行 change 後變成 2,1,4,3,5,6，同理跑接下來狀況，一次都只能換到一組，因此 pivot 無法進行好的分堆，造成 Dos。

Problem 3

3-1 Please draw a Aho-Corasick automaton with the vocabulary set {“aba”, “ababa”, “bad”, “baba”}.



3-2 Show that every fail link points to a node with smaller depth.

因為 failure link 的做法是退回去父親那格的 node 後，沿著父親的 failure link 走到另一個 node，接著看看這個 node 是否擁有一樣的兒子後將 failure link 補上去，如果沒有的話就再繼續向前面的 failure link 找去。又走到最後就會走回 root node，又 root node 是指向自己，且為第零層。因此可以知道 failure link 所指到的 node 深度一定低於本身的深度。又或者說因為 failure link 是在找當自己匹配失敗的時候，希望能找到自己的最長的 suffix 指到下一個有機會匹配成功的 node 上面，又 suffix 的定義會小於本身 string 的長度，因此可以知道，failure link 指到的 node 的深度一定會比原本 node 的小。

3-3 Briefly explain that you can construct an Aho-Corasick automaton…

建立 ACA 大概分成下列：

1.建立 trie

2.加上 failure function

第一部分：

建立 trie，因為需要把所有字串都走一遍，才可以將所有S_i都加到 trie 上，因此這部分的時間需要 O($\sum_{i=1}^{n}|S_i|$)

第二部份：

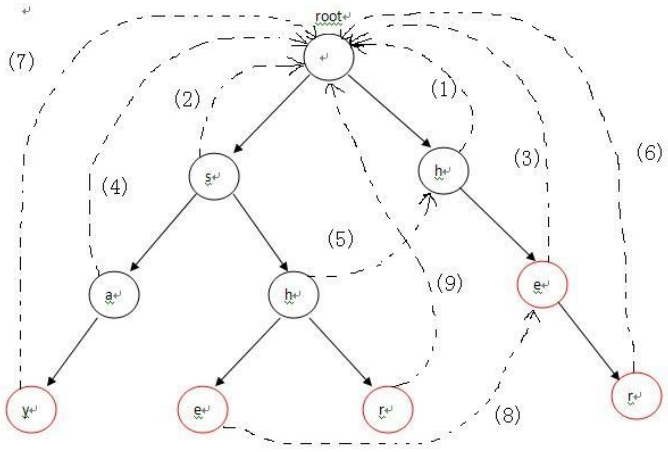
加上 failure link，每個 link 都會指回到當匹配失敗的時候，應該跳回到哪邊開始比較會比較洽當。又 failure link 是希望能指到：原本的 node 最長的 suffix = 其他 node 的整行文字的地方。

舉個例子來說就像是：假設現在走到一個 node_A 是 C，為了尋找他的 failure link 應該指回哪裡，就先走回去前一格，也就是 C 這格的父親的位置，接著沿著其父期的 failure link 走到所指的 node 後，再往後走一格，若此格的兒子也是 C 那麼就將 node_A 的 failure link 指過來。延續這種作法一直到最後如果都沒有找到的話，就將 failure link 指回 node。

具體操作起來只需要：先把 root 加入隊列(root 的失敗指針指向自己或者 NULL)，這以後我們每處理一個點，就把它的所有兒子加入隊列，隊列為空。之後欲將所有的 node 都加上 failure link，通常是用 BFS 將 node 都 traverse 過一次，而當一個點被 visit 過後便被加入隊列中，讓之後被 visit 到的 Node 可以指到他，因此所需要的時間和上面相同也是 O($\sum_{i=1}^{n}|S_i|$)。

(Ref: 具體的例子取自：
http://www.cppblog.com/mythit/archive/2009/04/21/80633.html)

具體例子：



對照圖-2 來看，首先 root 的 fail 指針指向 NULL，然後 root 入隊，進入循環。第 1 次循環的時候，我們需要處理 2 個節點：root->next[‘h’ - ‘a’](節點 h) 和 root->next[‘s’ - ‘a’](節點 s)。把這 2 個節點的失敗指針指向 root，並且先後進入隊列，失敗指針的指向對應圖-2 中的(1)，(2)兩條虛線；第 2 次進入循環後，從隊列中先彈出 h ，接下來 p 指向 h 節點的 fail 指針指向的節點，也就

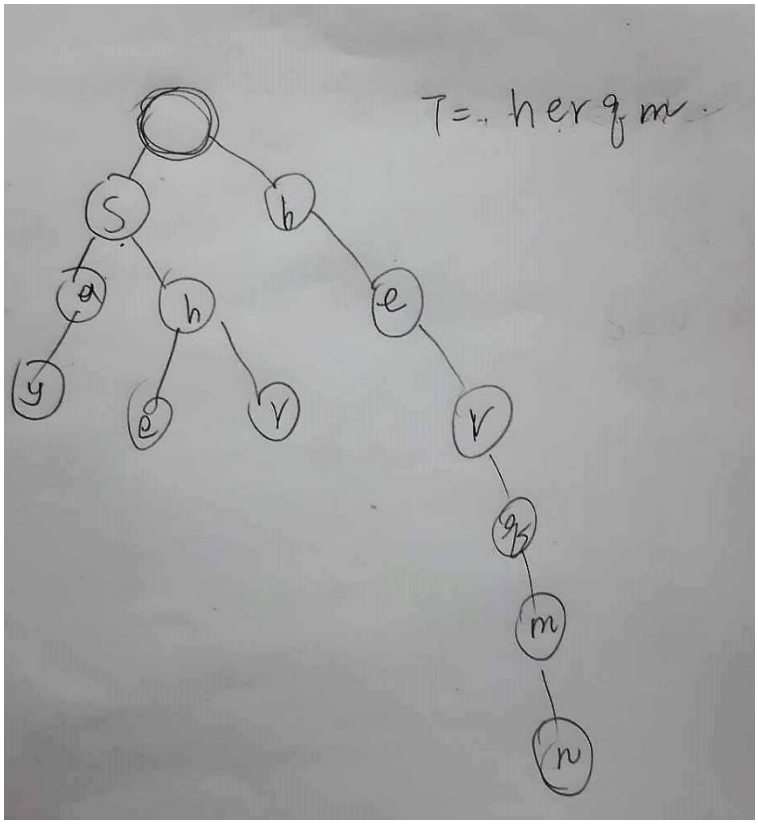
是 root；進入第 13 行的循環後，p=p->fail 也就是 p=NULL，這時退出循環，並把節點 e 的 fail 指針指向 root，對應圖-2 中的(3)，然後節點 e 進入隊列；第 3 次循環時，彈出的第一個節點 a 的操作與上一步操作的節點 e 相同，把 a 的 fail 指針指向 root ，對應圖-2 中的(4)，併入隊；第 4 次進入循環時，彈出節點 h(圖中左邊那個)，這時操作略有不同。**在程序運行到 14 行時，由於 p->next[i]!=NULL(root 有 h 這個兒子節點，圖中右邊那個)，這樣便把左邊那個 h 節點的失敗指針指向右邊那個 root 的兒子節點 h，對應圖-2 中的(5)，然後 h 入隊。以此類推：在循環結束後，所有的失敗指針就是圖-2 中的這種形式。**
(粗體字的地方所提到的就是前面說的，退回去 visit 其父親 node，如果父親擁有和自己相同值的兒子，則將其 failure link 指過去)。
因此綜合上述兩個部分，總共所花費的時間是 O($\sum_{i=1}^{n}|S_i|$)+ O($\sum_{i=1}^{n}|S_i|$) = O(2*$\sum_{i=1}^{n}|S_i|$)，又因為 2 是 const，因此可知建立 ACA 所花的時間是 O($\sum_{i=1}^{n}|S_i|$)

3-4 Given a string T and the vocabulary set{si|i = 1,...,n}, briefly explain that for each position i of T, you can find the longest s such that s ⊏ T[1..i]and s is a prefix of one of the vocabularies, in O(T + $\sum_{i=1}^{n}|S_i|$)
接下來再見完 ACA 之後就是匹配的部分。在這個建立好的 ACA 中尋找 string T 是否能夠匹配成功。如果不能匹配成功的話也一定會找到在每一行 Si 中階可以找到最長的 prefix 。
時間複雜度的 O($\sum_{i=1}^{n}|S_i|$)是用在建立 ACA 上。在建完 ACA 之後字串開始比對。一樣用上面的圖二做來解釋。
假設 T 是 shsera，且$S_1 = her, S_2 = shr, S_3 = she, S_4 =$ say。一開始，T[1]發現 root 有兒子一樣因此 ptr 指到 node s 上，且知道，此時的 T[1]是$S_2 = shr, S_3 = she$,的 prefix，長度為 1。接著繼續往下走 T[2] = h, 狀況和 T[1]時相同，又 T[2] 在這裡是$S_2 = shr, S_3 = she$,的 prefix，長度為 2。接下來走到 T[3]往下走發現不合，因此走 failure function ，走到了右邊的 h node，發現 h node 沒有兒子符合 = s 的 node，因此停下，且得知此時題目所求知 S(最常 prefix) 為$S_1 = her$,的 prefix funtion。因此可知，以此類推一直走，最後走到底走完 T 都可得到題目所求知 S。又 S 必定為 longest，因為如果一直都沒有走到 failure link 上，代表一直以來都有匹配到，因此可知 prefix 長度 = T 長。而若走到了 failure link 上，就可知道，產生了沒有匹配到的問題，因此重新進行接下來的匹配。就像是前面的例子，一開始走到左邊的 h 上，在往下走之後沒能匹配成功，因此走到了 failure link 上，也就是另一個 h 上希望能看看其他的 h 後面有沒有能夠匹配成功的兒子，如果有的話就可以延長 prefix 的長度。且 s ⊏ T[1..i]
/*
匹配過程分兩種情況：(1)當前字符匹配，表示從當前節點沿著樹邊有一條路徑可以到達目標字符，此時只需沿該路徑走向下一個節點繼續匹配即可，目標字符串指針移向下個字符繼續匹配；(2)當前字符不匹配，則去當前節點失敗指針

所指向的字符繼續匹配，匹配過程隨著指針指向 root 結束。重複這 2 個過程中的任意一個，直到模式串走到結尾為止。又因為不管怎麼 traverse 都是走在 trie 上面，如果此時停駐的 node 不是 root 就一定可以知道此時停在地 node 加上其前面的 node 一定是某一條S_i的 prefix。又如果是在 root 上的話，就可以知道，此時的 prefix length = 0，即完全無法匹配到。
*/

因此，將整行 T 走完需要 length 的時間，即 = O(T)，因此總共的時間是：
$O(\sum_{i=1}^{n}|S_i|) + O(T) = O(T + \sum_{i=1}^{n}|S_i|)$。

3-5 Given a string T and the vocabulary set{si|i = 1,...,n}, briefly explain that for each position i of T, you can find the longest s, if it exists, such that s ⊐ T[1..i] and s is one of the vocabularies, in $O(T + \sum_{i=1}^{n}|S_i|)$
題目和 3-4 類似，走法也一樣，不同於 3-4 的是，3-5 要求的是 s ⊐ T[1..i]，即此 S 若存在，他的長度將包含 T 的整個字串。而這種狀況要發生的時候就是從頭到尾都有對到，即在這所有的S_i當中，有一個S_x完整的包含了 T 這個字串。
像是下面的狀況：(下面的圖和圖二長的一樣 failure link 標示也都一樣，不一樣的地方是在S_1的下面多了幾個 node，且這些 node 的 failure link 都指回 root)：



因為 S 要完整的包含 T[1..i]，因此一定是完整的走完 T 且每一個都有對到。

此時 T = herqm，會發現到在 T 走完之後會完整的走在S_1上面，且S_1並尚未到最後一個 node。且 T 為S_1的 longest prefix，又 s ⊐ T[1..i]。然而這種狀況產生的機率感覺非常的低，因此題目寫說 if it exist。又如第四題所述。
建立 ACA 需要花 O($\sum_{i=1}^{n}|S_i|$)的時間，而欲將整條 T 走完則需要花 O(T)的時間。因此總時間複雜度是：O($\sum_{i=1}^{n}|S_i|$) + O(T) = O(T + $\sum_{i=1}^{n}|S_i|$)。