字串 I

1

1.1 前言

沒錯沒錯，今天要介紹的就是字串啦！

不知道大家知不知道字串和普通的序列差別是甚麼？其實字串本身就是一個序列，但通常在題目跟連續性有關的時候，我們才會稱它為字串題。

那就讓我們趕快進入此次的主題－字串吧！

1.2 基礎名詞簡介

1. 字元 (character)：構成字串的單位。以下「字串 A 的第 i 個字元」簡寫為 A_i。
2. 字元集：所有可能的字元的集合。
3. 長度：一個字串的字元個數。以下將「字串 A 的長度」簡寫為 $|A|$。
4. 子字串 (substring)：一個字串的一段連續的字元所構成的字串稱作子字串。以下將「字串 A 的第 $[i,j]$ 個字串所構成的子字串」簡寫為 $A_{i...j}$。(注意本篇講義的字串為左閉右閉)
5. 前綴、後綴 (prefix,suffix)：一個字串只取最前面一些字元所構成的子字串是前綴，只取最後面的則是後綴。前綴可寫成 $A_{0...i}$、後綴可寫成 $A_{i...|A|-1}$。

1.3 字串匹配問題

字串匹配是字串的經典老題。因為它太老，而不常出現在競賽中。不過字串匹配的眾多解法時常出現許多富有巧思的變化，用以解決較複雜的字串問題。現在讓我們來看看這個原始的問題吧！

習題 1.3.1: 字串匹配

給你一個字串 T，以及字串 P。求 P 是否為 T 的其中一個子字串。

最基本的想法是枚舉起點，然後再一一往後配對，當匹配不上則將起點向右移動一格。這樣的複雜度會是 $O(|P||T|)$。

```
bool matching(string t,string p){
    if(t.size() < p.size()) return false;
    for(int i = 0; i < t.size() - p.size() + 1; i++){
        bool found = true;
        for(int j = 0; j < p.size(); j++)
            if(t[j+i] != p[j]) found = false;
        if(found) return true;
    }
    return false;
}
```

當然，這樣的複雜度對於每日處理龐大資料的電腦算是一場災難，所以我們需要更加快速的解決方案。

雜湊 (Hash)

在處理字串問題中所提到的 hash 通常都是指將字串轉換成為一個數字，這樣比較的時候就可以達到 $O(1)$ 的比較時間了呢！做法就是將字串看長一個 p 進位制的數字，具體一點來說，假設雜湊函數為 $h(A)(A$ 是一個字串$)$，則 $h(A) = A_0p^{|A|-1} + A_1p^{|A|-2} + ... + A_{|A|-1}$ ($|A|$ 為字串長度)。照理說，只要 p 比字元集大小還要大，那就可以將一個字串唯一的轉成一個數字了喔！好，接下來問題就是，要如何快速地找出一個字串中某個子字串的 hash 值呢？首先我們先預處理字串每個前綴的 hash 值，也就是找出 $h(A_{1...|A|-1}), h(A_{1...|A|-2}), ..., h(A_{1...1})$。根據定義，應該不難看出，$h(A_{1...i}) = h(A_{1...i-1}) \times p + A_i$。因此，只需要花 $O(|A|)$ 的時間就可以預處理完了。接著假設想要知道 $A_{i...j}$ 這段子字串的湊湊值，一樣根據定義，這段的雜湊值就是 $h(A_{1...j}) - h(A_{1...i}) \times p^{j-i}$，而這個值可以 $O(1)$ 算出 (畢竟 p 的冪次也可以 $O(|A|)$ 預處理然後 $O(1)$ 知道。

這樣說是不是感覺非常簡單易懂呢？但或許大家也發現了，就是雜湊值會非常大，long long也存不下嘛！而應對方式也非常簡單，就是在計算的時候模一個數字，就所有問題都解決了，因為上述的計算都可以在一個模數之下好好地做。但這樣也出現了新的問題，就是因為雜湊值模了一個數，有可能造成兩個不同的字串卻有相同的雜湊值，這就是碰撞 (collision) 囉。解決方式也很簡單，就是開很多 hash，如果那麼多個 hash 值都顯示兩個字串相同，那我們或許就可以合理相信兩個字串真的一樣了呢。那到底要寫多少 hash 才合理呢？我聽說一般是要寫 5 個 hash 喔！

最後的問題就是 p 和模數的選取了，我不是很確定怎麼選才好，不過一般認為模數應該要夠大 (要不然更容易碰撞)，而且 p 和模數應該不互質的時候會有某些數不被用到，所以我會選把兩個都選成質數 (或至少模數選質數)。

唉呀，講了這麼多，都還沒講到要怎麼用 hash 來匹配字串呢！

假設要在字串 A 中找到字串 B，則我們一樣花 $O(|A|)$ 的時間將 A 的每一個前綴的 hash 找出來，也順便算出 B 的 hash 值。接著我們可以知道，A 中有 $|A| - |B| + 1$ 個位子有可能找到 B，因此，我們就可以在線性時間內去看每個可能符合條件的位子的 hash 值跟 B 一不一樣就做完囉！整體來說，用 hash 來做字串匹配的複雜度是 $O(|A| + |B|)$。

Hash 固然是字串匹配的好幫手，不過千萬不要覺得 Hash 只能拿來做字串匹配，有許多其他字串題也有它出場的機會！

以下附上我醜醜的 code。

string::find

這個東西就是 C++ STL 裡面的函式，可以直接達到字串匹配的目的。直接給出範例程式囉！

```
#include<iostream>
#include<string>
using namespace std;
int main()
{
  string A = "ABCABCABC";
  string B = "CAB";
  cout << A.find(B) << endl;//在A中找出B第一個出現的位子
}
```

感覺非常實用呢，是不是？但相信大家或許跟我一開始會有一樣的問題，就是它的複雜度到底是多少呢？我大概上網查了一下，C++ 沒有規定它的複雜度，但我實測起來感覺速度頗快，大家好好斟酌一下比賽的時候要不要用吧。

1.4 古斯菲爾德演算法

這個字串匹配的演算法由 Dan Gusfield 提出，又稱作 Z-algorithm。這個演算法能夠實現，主要是建立在「對於一個字串 s，能夠在線性時間內計算其對應的 Z-陣列」這個基礎上。

Z-陣列

Z-陣列是一個與字串長度相同的陣列，每個元素 $Z[k]$ 代表的是以位置 k 為始的最長子字串長度，使得這個子字串也是整個字串的前綴。

以字串ACBACDACBACBACDA 為例，依此建構出的 Z-陣列就如以下所示：

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
16	0	0	2	0	0	5	0	0	7	0	0	2	0	0	1

其中 $Z[6] = 5$，因為以 $s[6]$ 開頭的子字串ACBACB剛好是整個字串的一個前綴。

如何計算 Z-陣列

計算 Z-陣列的方法正是 Gusfield's Algorithm 的精神所在。為了有效率的完成 Z-陣列的建構，這個演算法會維護一個區間 $s[x...y]$，使得這個區間是一個原字串的前綴，而且 y 要盡量越大越好。

維護這個區間的目的是：當你要計算一個未知的 $Z[i]$ 時，能夠確保可以利用先前儲存的值快速計算。如果將要計算 $Z[i]$ 的 i 在 $s[x...y]$ 的區間之外，我們就沒有儲存到任何有關 $Z[i]$ 的資訊，只能從頭計算；但是如果這個 i 在這個區間內，就可以很快知道 $Z[i]$ 至少有 $\min(Z[i-x], y-i+1)$，所以從這個值開始暴力搜就行了。

詳細作法如下：

1. 如果 $i > y$，代表還沒有已知的、與前綴相同的子字串包含 i，這時候重新暴力比對 $s[0...$ 與 $s[i...$，並更新 $x, y, Z[i]$。
2. 如果 $i \leq y$ 我們知道 $s[0...y-x]$ 與 $s[x...y]$ 相等，因此 i 的位置在 $s[x...y]$ 的地位就類似 $i-x$ 在 $s[0...y-x]$ 的地位。因此我們可以用 $Z[i-x]$ 來推算 $Z[i]$。
3. 如果 $Z[i-x] < y-i+1$，已經確定 $Z[i]$ 不能再多了，所以 $Z[i]$ 就是 $Z[i-x]$。
4. 如果 $Z[i-x] \geq y-i+1$，那麼 $Z[i]$ 只能確定不小於 $y-i+1$，所以一樣要暴力比對 $s[y...$ 與 $s[y-x...$，並更新 $x, y, Z[i]$。

```
vector<int> Z(string &s){
    vector<int> Z(s.size());
    int x = 0, y = 0;
    for(int i = 0; i < s.size(); i++){
        Z[i] = max(0, min(y-i+1, Z[i-x]));
        while(i+Z[i] < s.size() && s[Z[i]] == s[i+Z[i]])
            x = i, y = i+Z[i], Z[i]++;
    }
    return Z;
}
```

隨著越來越多的 $Z[i]$ 被算出來，y 值也會不斷的增加，每次暴力往後搜一格，y 值就會增加或至少不變，所以需要暴力搜的個數不會超過 $O(n)$。

也就是說，對於每個 i，當while迴圈的條件成立時頂多使 y 值不變一次，第二次開始 y 值一定會增加 1。因為 y 值最多只有字串長度，所以時間複雜度 $O(n)$。

Z-陣列與字串匹配

有了 Z-陣列之後，那我們要怎麼進行字串匹配呢？這邊我們需要一些巧思，將 P 與 T 兩個字串用特殊字元接在一起，再做一次 Z-陣列就行了。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	C	B	#	C	D	A	C	B	A	C	B	A	C	D	A
16	0	0	0	0	0	3	0	0	3	0	0	2	0	0	1

在這裡，Z-陣列中值恰好等於字串 P 長度的位置，就是找到了的 P 的一個匹配。

1.5 克努斯-莫里斯-普拉特演算法

除了 Gusfield's Algorithm 之外，還有一個類似的作法叫 KMP 算法，這個做法也是對於原字串建立一個陣列，再利用這個陣列的值進行字串匹配。

失配函數

想像一個情境：你嘗試用暴力法在 aaabaaaab 找 aaaa ，當你從第一個字元當開頭匹配到第四個字元時，你明明已經知道因為有個b卡在那裡，導致前四個字元都不可能作為開頭。但是你還是必須把開頭對到第二個字元，再慢慢掃，對吧。

這時候你就需要一個東西，先對字串 P 計算好一個陣列，指引配對失敗時開頭位置要怎麼往後跳，就可以不必每次都只往後一格。

次長共同前後綴

失配函數事實上就是次長共同前後綴。我們定義

$$Fail(i) = s[0...i]\textit{的次長共同前後綴的長度}$$

這裡用次長不用最長的原因很簡單，因為最長共同前後綴就是 $s[0...i]$ 本身，根本沒必要算。

那我們對於一個字串 P，我們要怎麼建立失配函數陣列呢？

與 Z-陣列的建構方式有些類似，對於每個 i，我們可以確定 $s[0...F[i-1]] = s[i-1-F[i-1]...i-1]$，因此可以假設 $q = F[i-1]$。如果 $s[q]$ 與 $s[i]$ 相同，則 $F[i]$ 的值就等於 $q+1$。

如果這兩個字元不相同，我們可以試著縮小 q 的值，來使 $s[q]$ 與 $s[i]$ 相同。因為失配函數本身的性質，我們可以知道，$s[i-F[q-1]-1...i-1]$ 與 $s[0...F[q-1]]$ 是一樣的，所以可以將新的 q 換成 $F[q-1]$，繼續比對 $s[q]$ 與 $s[i]$。

```
vector<int> build_failure(string &s){
    vector<int> fail = {0};
    for(int i = 1, q = 0; i < s.size(); i++){
        // q = fail[i-1];
        while(q && s[i] != s[q]) q = fail[q - 1];
        fail.push_back(q += (s[i] == s[q]));
    }
    return fail;
}
```

KMP 與字串匹配

相信有了失配函數之後，大家都很快就知道怎麼進行字串匹配了。我們先對欲尋找的字串 P，建立它的失配函數陣列，然後用兩個變數 i, j 在字串 T 與 P 上爬行。

如果 $T[i] = P[j]$ 表示配對正確，則繼續對 $T[i+1], P[j+1]$ 進行比對，直到字串完全匹配或失配；如果 $T[i] \neq P[j]$ 代表失配，將 j 的值跳到失配函數 $(F[j-1])$ 的位置就行了！$j = |P|$ 時就代表 P 字串已經被完全找到，達成字串匹配的目的。

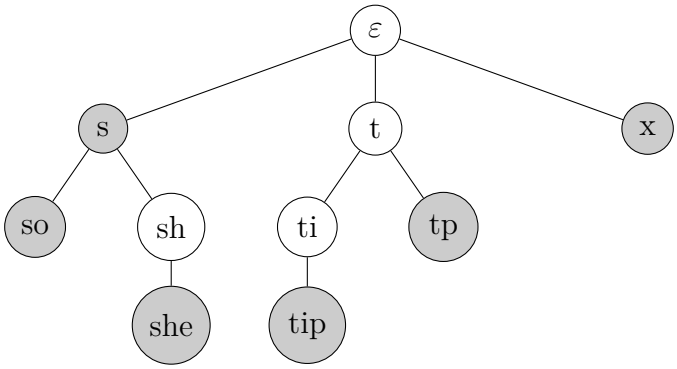
```
void KMPSearch(string t, string p){
    vector<int> fail = build_failure(p);
    int i = 0; // index for t[]
    int j = 0; // index for p[]
    while (i < t.size()){
        if(p[j] == t[i]){ // 匹配
            j++, i++;
        }
        if(j == p.size()){ // 找到了！
            printf("Found pattern at index %d \n", i-j);
            j = fail[j-1];
        }
        else if(i < t.size() && p[j] != t[i]){ // 失配
            if(j != 0) j = fail[j-1];
            else i++;
        }
    }
}
```

另一種比較短的寫法是

```cpp
void KMPsearch(string t, string p) {
    vector<int> fail = build_failure(p);
    for(int i = 0, j = 0; i < t.size(); i++) {
        while(j && t[i] != p[j]) j = fail[j-1];
        if(t[i] == p[j]) ++j;
        if(j == p.size()) {
            printf("Found pattern at index %d \n", i-j);
            j = fail[j-1];
        }
    }
}
```

1.6 Trie

首先考慮一個這樣的問題，你需要處理兩種操作：一、在資料結構中加入一個字串；二、給你一個字串，問你此字串是否曾經被插入該資料結構。大家或許會很直覺地認為直接開一個set，然後把字串全部丟進去就解決了！那這麼做的複雜度會是多少呢？假設目前有 N 個字串在set中，而當前要查詢一個長度為 L 的字串是否在set中，這樣複雜度就是 $O(L \log N)$。有沒有可能有更好的複雜度呢？這時候就要派我們的 Trie 出場了呢！看看它的英文字首，就知道他絕對是跟樹有關了呢！它的的確確是一棵樹，它的每一顆節點都包含了字元集大小的那麼多個指標，舉例來說，若全部的字串都是由英文小寫字母所組成的話，那每顆節點都有 26 個指標。好，接下來就直接說要如何插入一個字串吧，直接來看例子比較直接：
下圖插入了”s”, ”so”, ”tp”, ”tip”, ”she”, ”x”，灰色節點表示該節點是一個字串的結尾



相信大家看完了上面的例子，都大概了解 Trie 的運作方式了。而它的時間複雜度大家應該也知道，就是插入和查詢都是 $O(N)$，而空間複雜度就是 $O($ 字元集大小 $\times \sum L_i)$。其實 gcc 中的 pbds 也有一個能直接拿來用的 Trie，詳情就參考 pbds 的講義吧！

在字元集越小的時候，Trie 越容易派上用場，尤其是只有 0 跟 1 的時候，例如跟某些位元運算有關的題目常常得把 Trie 派出場。

習題 1.6.1: 最大區間 XOR（經典問題）

給定一個序列，求最大的區間 XOR 值。

對序列做一次前綴可以發現問題變成找到兩個數字 XOR 起來最大，我們可以一個一個把前綴的二進位表示法丟進 Trie 裡面（最高位放在前面）並順便拿去 Trie 裡面查，能夠往相反的地方走就盡量往相反的地方走便能找到最大 XOR 值。

```
int ch[N*30][2], tot; // 0同時代表根節點和空節點:P
void ins(int x) {
    int now = 0;
    for(int c = 29; c >= 0; c--) {
        bool d = x>>c & 1;
        if(!ch[now][d])
            ch[now][d] = ++tot;
        now = ch[now][d];
    }
}
int qry(int x) {
    int now = 0, res = 0;
    for(int c = 29; c >= 0; c--) {
        bool d = x>>c & 1;
        if(ch[now][!d])
            now = ch[now][!d], res ^= 1<<c;
        else
            now = ch[now][d];
    }
    return res;
}
```

1.7 習題

排版跑掉了 郭死

習題 1.7.1: Massacre at Camp Happy (TIOJ 1725)

給定兩個長度相同的字串 A、B，請你找出所有的 k，使得將 A 的前 k 個字元移到尾端時會跟 B 只差一個字元，或回答不存在符合的 k。($|A|, |B| \leq 10^6$)

習題 1.7.2: 似曾相識 (TIOJ 1515)

給定一字串 s，請問在此字串中重複出現兩次以上的最長字串長度為何 (若無則輸出 0)？($|s| \leq 2 \times 10^5$)(其實這題可以用一個較複雜的結構 suffix array 做完，但你能想到比較簡單的方法嗎？)

習題 1.7.3: 字串中的字串 (TIOJ 1306)

給你一個字串 T，以及很多字串 P。對於每個 P 請輸出 P 在 T 中出現過幾次。(T、P 都是由小寫字母所組成，長度 $\leq 10^4$)(你可以想到幾種方法來解這題呢？)

習題 1.7.4: k-口吃子字串 (TIOJ 1735)

給你一個字串 s，和一個非負整數 k，問你 s 中有多少組長度為 k 的兩個子字串相同且相連？($|s| \leq 10^5$)

習題 1.7.5: kukukey（TIOJ 1531）

給你一個字串 S 和 k，對於所有「恰好可以被分成 k 個相同的小字串」的前綴，請輸出可以分成的小字串的最大長度。$|S| \leq 5 \cdot 10^6$

字串 II

2

2.1 前言

前面已經學過了關於字串的一些知識，譬如 KMP 演算法，字典樹 (Trie) 等，這裡會介紹一些更進階的資料結構與演算法來更有效率的處理字串的各種問題，其中非常精妙，值得細嚼慢嚥的讀。

2.2 後綴序列 Suffix Array

什麼是後綴序列？

要看後綴序列之前，以防混淆，先定義一下字串的後綴是什麼：

定義 2.2.1: 後綴

首先，來定義後綴：對於一個字串 $S[0 \dots n-1]$，定義其第 i 個後綴為 $S[i \dots n-1]$。可以知道，第 0 個後綴為字串本身，而第 $n-1$ 個字串就是 S 的最後一個字元。

知道後綴是什麼了，就可以定義後綴序列了：

定義 2.2.2: 後綴序列

對於一個字串 S，我們將其後綴照字典序排序之後，將後綴序列 $SA[i]$ 定義為「S 的字典序第 i 小的後綴是第幾個後綴」

或許有一點抽象，所以我們就來跑一次例子試試看吧！令 $S = abaab$，我們可以得到其後綴們：

$$\begin{array}{cc} 0 & abaab \\ 1 & baab \\ 2 & aab \\ 3 & ab \\ 4 & b \end{array}$$

大家一定都想得到一個很簡單的方法來建這個序列：把東西全部都放進去一個陣列之後sort，但是因為兩個字串的比較是 $O(N)$（N 為字串長度），所以這樣複

雜度是 $O(N^2 \log N)$，如果稍微大一點的字串就會超時。在這裡，我們介紹一個 $O(N \log^2 N)$ 的演算法，雖然有 $O(N \log N)$ 甚至 $O(N)$ 的演算法，但是都沒那麼好寫，比賽通常也不會壓這個常數，所以先介紹這個簡單一點的版本。

建立後綴序列

注意：這裡要排序的一堆字串，不是隨便抓來的隨機字串，而是同一個字串的後綴！我們可以利用這個性質，來對演算法優化。我們的中心概念是「倍增法」：若可以依照前 2^k 個字元排序，能否用這個資訊得到前 2^{k+1} 個字元的排序？答案是：可以的。假設對於每一個位置 i，都已經知道對於目前前 2^k 個字元來說，i 是第幾位（當然如果相同的話就並列），那要怎麼得到下一個是第幾位？

其實很簡單：注意到第 i 個前綴的前 2^k 個字元其實就是第 i 個前綴的前 2^{k-1} 個字元，後面加上第 $i+2^{k-1}$ 個前綴的前 2^{k-1} 個字元。所以，要比較第 i 個前綴的前 2^k 個字元和第 j 個前綴的前 2^k 個字元那一個比較大就可以用前面所計算的來比較了。

定義 2.2.3: Rank Array

有些人會叫做 SA^{-1}，在此稱作 $rank$，也就是 SA 的「反函數」，SA 是給定一個「第幾名」，它跟你說第幾名是誰；而我們這個 $rank$ 陣列則是：給定第 i 個後綴，它會告訴我們它是第幾名。以數學方式來說，對於 $0 \le i < N$，$rank$ 會滿足：

$$rank[SA[i]] = i$$

具體來說，若 $L[i][k]$ 代表第 i 個前綴的前 2^k 個字元（若沒有則取代為最小字元）是第幾小的。我們想要由 $L[i][k]$ 獲得 $L[i][k+1]$。則因為第 i 個前綴的前 2^k 個字元其實就是第 i 個前綴的前 2^{k-1} 個字元，我們如果要看第 i 個前綴和第 j 個前綴的前 2^{k+1} 個字元哪一個比較小，先比較前面的：$L[i][k]$ 和 $L[j][k]$。若分不出高下，再比較 $L[i+2^k][k]$ 和 $L[j+2^k][k]$，若超出範圍則看 i 和 j 哪一個比較大，大的代表字串長度比較小，放前面。以上會用到 $O(N \log N)$ 的空間，可以滾動掉。時間的部分，可以知道有 $O(\log N)$ 次的計算，每次都需要 $O(N \log N)$ 來排序，所以總時間複雜度 $O(N \log^2 N)$。程式如下：

```
vector<int> SA, nextrk, rk; //目前的後綴序列，下一個Rank，目前的Rank（第幾個）
int gap, N;

bool cmp(int i, int j){
    if(rk[i] != rk[j])
        return rk[i] < rk[j]; //若前面就能比較

    i += gap;
    j += gap;

    if(i < N && j < N)
        return rk[i] < rk[j]; //比較後面
    else
        return i > j; //字串小的放前面
}
```

```
16
17 void getSA(string s){
18     N = s.length();
19     SA.resize(N);
20     nextrk.resize(N);
21     rk.resize(N);
22
23     nextrk[0] = 0;
24
25     for(int i = 0; i < N; i++){ //初始化，待會就會sort掉了
26         SA[i] = i;
27         rk[i] = s[i] - 'a';
28     }
29
30     for(gap = 1; ; gap *= 2){ //gap就是一次看幾個字串
31         sort(SA.begin(), SA.end(), cmp);
32         for(int i = 1; i < N; i++)
33             nextrk[i] = nextrk[i-1] + cmp(SA[i - 1], SA[i]);
                //若不同了，就要加一
34         for(int i = 0; i < N; i++)
35             rk[SA[i]] = nextrk[i]; //更新Rank
36         if(nextrk[N - 1] == N - 1) break; //結束條件
37     }
38 }
```

$O(N \log N)$ 建立後綴序列

嗯，剛剛的 $quicksort$ 還真的不錯，但畢竟時間複雜度很難唸，而且複雜度為 $O(N \log N)$ 的解也沒有難 (長) 太多，所以就不學白不學囉！其實做法跟剛剛所敘述的基本上一樣，也是使用倍增法，每次排序一次，只是真的有必要每次都做一次快速排序嗎？

等等，快速排序應該是比較行排序中數一數二快的了吧，還有甚麼排序法比快速排序法更快呢？

大家可能會想到所謂的計數排序 (counting sort) 可以 $O(N + C(\text{值域}))$ 做完排序。那大家不妨想想，這裡每次要排序時的值域多大呢？因為總共要比較兩次 rk，而所有後綴的 rk 最多有 N 種可能，所以這裡的值域是 $O(N^2)$，空間和時間都不允許。

那還有甚麼排序方式呢？如果要排序值域界在 1 到 100 之間的整數，真的只能開 100 格來做 counting sort 嗎？有沒有可能只開十格或二十格？其實有一種作法叫做 radix sort，不知道大家知不知道？假設要排序的數值都是以十進位制表示的二位整數，我們就開兩個十個格子的陣列 A、B，先做個位數再十位數。第一次把每個整數 a_i 丟到 $A[a_i\%10]$。第二次的時候從 $A[0]$ 的最底部 (最早被丟進去的) 開始往上看，再依序看 $A[1]$ 到 $A[9]$，也都是由底部往上看。這次，依照被看到的順序將每個整數 a_i 丟到 $B[a_i/10]$。做完之後，從 $B[0]$ 的最底部開始往上、往右看，將看到的數字依序記錄下來，就是排序好的序列了。大家可能不太懂，可以參考這個動畫：https://www.youtube.com/watch?v=xuU-DS_5Z4g。這真的是一個很神奇的方法，我一開始聽到的時候也沒有完全理解他為甚麼會

對，但是自己動手畫畫看，就應該可以理解其中的奧妙了。它的複雜度應該是 $O(($ 開的格字數 $+$ 序列長度 $)\times$ 進行輪數 $)$。

好，說了這麼多，不知道大家有沒有了解為什麼在這裡上 radix sort 會使複雜度變好。那我們就將目前的狀況代入它的複雜度中吧！好，首先序列長度不用說，就是字串長度 N 嘛，而進行輪數就是 2(比較前半段和後半段)，那我們要開幾格呢？因為每一輪都只要依照前半段或後半段的 rk 將其塞到對應的格子中，rk 右最多只有 N 種，所以只要開 N 個格子就好了！因此複雜度是 $O((N+N)\times 2) = O(N)$，就這樣，一個 log 就被壓掉了！

但實做上還是有一些技巧可以使用，就先煩請大家先稍微看一下 code 有那裡不懂，在 code 後我會再詳細解釋。

```
#define dictsz 200000
//這裡假設有200000種字元
int gap, N;
vector<int> SA, nextrk, rk, temp, cnt;
bool cmp(int i, int j){
    if(rk[i] != rk[j])
        return rk[i] < rk[j]; //若前面就能比較

    i += gap;
    j += gap;

    if(i < N && j < N)
        return rk[i] < rk[j]; //比較後面
    else
        return i > j; //字串小的放前面
}

void getSA(string s){
    N = s.length();
    SA.resize(N);
    nextrk.resize(N);
    rk.resize(N);
    cnt.resize(dictsz);//這就是桶子啦
    nextrk[0] = 0;
    int i;
    for(i = 0; i < N; i++) //初始化，待會就會sort掉了
        rk[i] = s[i];
    for(i = 0;i < dictsz;i++)
        cnt[i] = 0;//初始化
    for(i = 0;i < N;i++)
        cnt[rk[i]]++;//丟到桶子
    for(i = 1;i < dictsz;i++)
        cnt[i] += cnt[i-1];//待會解釋
    for(i = N-1;i >= 0;i--)
        SA[--cnt[rk[i]]] = i;//待會解釋
    for(gap = 1;; gap *= 2){ //gap就是一次看幾個字元
        temp.clear();
        for(i = N-gap;i < N;i++)
```

```
39          temp.push_back(i);//後半段會超出的後綴後半段一定最小字典序
40          for(i = 0;i < N;i++){
41              if(SA[i] >= gap)
42              temp.push_back(SA[i]-gap);//將後綴沒有超出的後綴按照字典序丟進去
43          }
44          //注意，這裡的temp就會依照所有後綴後半段的字典序由小到大排列了
45          for(i = 0;i < dictsz;i++)
46              cnt[i] = 0;//把桶子清空
47          for(i = 0;i < N;i++)
48              cnt[rk[temp[i]]]++;//照順序將每個後綴丟到代表其前半段字典序大小的桶子中
49          for(i = 1;i < dictsz;i++)
50              cnt[i] += cnt[i-1];//做前綴，為甚麼？待會解釋
51          for(i = N-1;i >= 0;i--)
52              SA[--cnt[rk[temp[i]]]] = temp[i];//這啥？待會解釋
53          nextrk[SA[0]] = 0;
54          for(i = 1; i < N; i++)
55              nextrk[SA[i]] = nextrk[SA[i-1]] + cmp(SA[i - 1], SA[i]); //若不同了，就要加一
56          for(i = 0; i < N; i++)
57              rk[i] = nextrk[i]; //更新Rank
58          if(rk[SA[N - 1]] == N - 1) break; //結束條件
59      }
60  }
```

大家應該可以發現，code 跟之前長得差不多。我覺得大家最不能理解的應該就是為甚麼要做前綴，還有這句`SA[--cnt[rk[temp[i]]]] = temp[i]`到底是什麼意思吧？其實這兩件事是一體的。簡單來說，做前綴的目的就是讓要放進去的時候知道要放到 SA 中的哪個位子。再仔細看看上面那行，它的`for`迴圈其實是倒過來寫的，它就是由後往前掃過一遍已經按照後半段排好的後綴，然後 cnt 的前綴值減一不就是它應該放在 SA 中的位置嗎？

2.3 最長共同前綴序列 LCP Array

LCP Array 又是什麼？

LCP Array 是有了後綴序列之後的一個好幫手，可以再次加速演算法！這一次也要先定義所需要的函數：對於兩個字串 A 和 B，我們定義 $lcp(A, B)$ 為他們兩個的最長共同前綴的長度。

定義 2.3.1: LCP Array

對於一個字串 S 和其前綴序列 SA，我們定義其 LCP 陣列 $LCP[i] = lcp(T_{SA[i]}, T_{SA[i-1]})$，且 $LCP[0] = 0$。此處 T_i 代表 S 的第 i 個前綴。

當然，不難想到一個 $O(N^2)$ 的計算方法，但是這樣一定太慢！有一個 $O(N)$ 的計算方法，讓我們繼續看下去！

Kasai's Algorithm

這個神奇的方法稱為 Kasai's Algorithm，能夠在給定字串 S 和其後綴序列 SA 的情況下在 $O(N)$ 算出 LCP 陣列。首先，我們先定義一個東東：

接下來，這個部分可能你會需要重讀很多遍，因為真的蠻細的；不過搞懂之後，實作不複雜。

Kasai's Algorithm 的中心思想在於思考第 i 個後綴與第 $i+1$ 個後綴的關係：因為只是拿掉前面一個字元，所以會有好性質可以用！

首先，考慮後綴序列裡面的連續兩個位置，稱為 i_1 與 i'_1。若 $lcp(i_1, i'_1) = 0$ 就算了；否則我們可以知道，T_{i_1} 和 $T_{i'_1}$ 的第一個字元一定相同。所以，如果拔掉第一個字元會怎麼樣？你會發現，T_{i_1} 拔掉第一個字元不就是 T_{i_1+1} 嗎？首先，我們可以知道 $lcp(i_1+1, i'_1+1) = lcp(i_1, i'_1) - 1$，因為前者只是後者拿掉第一個字元而已，所以顯然是對的。

因為 $lcp(i,j) = \min(lcp_i, lcp_{i+1}, \ldots, lcp_j)$（左邊的 $lcp(i,j)$ 代表 T_i 和 T_j 的最長共同前綴的長度），我們可以知道 $lcp(i_1+1) \geq lcp(i_1+1, i'_1+1) = lcp(i_1, i'_1) - 1$，所以只要從上一個值減一（若不是零的話）計算就好了。注意：這裡有兩個序列：第一個是後綴序列的順序，第二個是原本後綴的順序，此處 T_k 都是代表第 k 個後綴。而因為計算第 T_i 的時候會用到 T_{i-1}，所以跑的順序是依照後綴的順序跑的，而不是依照後綴序列的順序跑。程式：

```
vector<int> SA, lcp;
void getlcp(string s){
    lcp.resize(N);
    vector<int> SAinv;
    SAinv.resize(N);
    for(int i = 0; i < N; i++){
        SAinv[SA[i]] = i;
    }

    int k = 0;
    for(int i = 0; i < N; i++){
        if(!rk[i]){ //若其為第一個，則沒有lcp
            k = 0;
            continue;
        }
        int j = SA[rk[i] - 1]; //拿到後綴序列的上一個
        while(i + k < N && j + k < N && s[i + k] == s[j +k])

            k++;  //利用上面的性質硬跑
        lcp[SAinv[i]] = k; //設定，不是lcp[i]！
        if(k) k--; //界是 -1
    }
}
```

而這個為什麼是 $O(N)$ 呢？因為k 除了一次之外，每次都最多減少 1，所以while 迴圈不可能跑太多次，故時間複雜度是 $O(N)$。

2.4 AC 自動機（AC Automaton）

AC 自動機顧名思義就是一種可以讓你自動 AC 的程式啊 (X

好，絕對不要聽別人這樣唬爛，事實上 AC 自動機是 KMP 字串匹配演算法的進階版，可以一次尋找很多目標字串各個在主字串中出現的地方。不知道大家還記不記得，KMP 演算法是在進行字串匹配前，計算當匹配失敗時，可以直接從哪個位置開始重新匹配，也就是 failure function。而在 AC 自動機中也是利用相同的手法，在目標字串們建構出的 trie 上計算 failure pointer，同樣代表匹配失敗時應從哪裡重新開始。

節點

AC 自動機的節點大致上與 trie 都一樣，只是會多新增幾個資料，詳細的struct如下：

```
struct node{ //K是字元集的大小
    node *ch[K],*fail; //記錄子節點們和failure pointer
    vector<int> endof; //紀錄這個節點表示了那些字典集中的字串
    node(): endof(){
        for(int i=0; i<K; i++) ch[i]=nullptr;
        fail=nullptr;
    }
};
```

build AC automaton

最重要的當然是先建構出 trie 才能做其他事吧！當然，建構方式和正常的 trie 也差不多，因此就直接看 code 吧：

```
node *build(const vector<string> &dict) {
    node *root=new node();
    for(int i=0; i<dict.size(); i++) {
        node *now = root;
        for(char c: dict[i]) {
            if(!now->ch[c-'a'])
                now->ch[c-'a'] = new node();
            now = now->ch[c-'a'];
        }
        now->endof.push_back(i);
    }
    return root;
}
```

failure pointer

原本 failure function 代表的是某個前綴的「次長共同前後綴」，在 trie 上沿用這個概念的 failure pointer 指向的是「次長的存在於這個 trie 中的後綴」，我們同樣能知道 failure pointer 肯定離根比較近，因此移到 trie 時我們選用 BFS 的遍歷順序依序計算，如此便可保證計算某一節點時，長度更短的都已經算完了。說了這麼多，不如直接看程式碼吧：

```
vector<node *> buildFail(node *root) {
    vector<node *> BFS; // 儲存BFS的遍歷順序，待會用到
    queue<node *> Q;
    for(int c=0; c<K; c++) {
        if(root->ch[c]) {
            root->ch[c]->fail = root;
            Q.push(root->ch[c]);
        }
    }
    while(!Q.empty()) {
        node *p = Q.front(); Q.pop();
        BFS.push_back(p);
        for(int c=0; c<K; c++) if(p->ch[c]) {
            node *f = p->fail;
            while(f != root && !f->ch[c]) f = f->fail;
            if(f->ch[c]) f = f->ch[c];
            p->ch[c]->fail = f;
            Q.push(p->ch[c]);
        }
    }
    return BFS;
}
```

開始匹配

不外乎就是跟 KMP 的方法一樣，從第一個字開始看，若遇到失配的狀況就走 failure pointer。程式碼應該也相當易懂：

```
void match(node *root, const string &S, const vector<string>
   &dict){
    node *p = root;
    for(int i=0; i<S.size(); i++) {
        while(p != root && !p->ch[S[i]-'a']) p = p->fail;
        if(p->ch[S[i]-'a']) p = p->ch[S[i]-'a'];
        for(int id: p->endof) {
            cout << "Match pattern #" << id << " at position"
                    << i - dict[id].size() << '\n';
            }
    }
}
```

你以為結束了？

多玩幾次之後就會發現上面 AC 自動機在目標字串互相包含時會出問題！看下面範例：

dict={”ab”,”cababc”}
S=”cababc”

上面程式的輸出結果”ab” 消失了！為什麼會這樣？你畫出 AC 自動機後自己走一次就會發現，整個匹配過程都是走”cababc” 的那條路，然而經過”ab” 時，完全沒有標記讓 AC 自動機知道他也是一個單字。也就是說，只要某個字被包含在另一個字的中間時，就會被忽略。怎麼辦呢？

假設從節點 $S_{0,1\ldots k-1}$ 走到 $S_{0,1\ldots k}$ 時，會使 $S_{1,2\ldots k-2}$ 的所有後綴字串沒有被考慮到。因此我們應該要窮舉這些字串，檢查他們是不是也在字典中。然而，我們可以發現 $S_{1,2\ldots k-2}$ 後綴字串中所有在字典裡的字串，事實上就是不斷走 failure pointer 路上經過的那些字串！想一想就知道了！

```
void match(node *root, const string &S, const vector<string>
   &dict){
    node *p = root;
    for(int i=0; i<S.size(); i++) {
        while(p != root && !p->ch[S[i]-'a']) p = p->fail;
        if(p->ch[S[i]-'a']) p = p->ch[S[i]-'a'];
        for(node *now=p; now!=root; now=now->fail) { //跳fail
            for(int id: now->endof) {
                cout << "Match pattern #" << id << " at
                   position" << i - dict[id].size() << '\n';
            }
        }
    }
}
```

如果只想知道各個字串被匹配到的次數，我們可以記錄所有點被走過幾次，然後在 Match 跑完之後將每個節點被經過的次數不斷向他的 failure pointer 加，類似在樹或 DAG 上 DP 的感覺。那應該用什麼順序處理這件事呢？因為要不斷向上推，不如就從最深的開始處理，由於 failure pointer 的深度一定比該節點的深度小，直接用 BFS 遍歷順序的逆序正好可以保證正確性！直接來看一下 code 吧！假設 node 裡面已經有宣告了cnt，並且假如大字串在 match 中每一步走到的每個節點cnt都增加 1，經過 DP 之後最後的值將會表示大字串和這個節點所代表的字串的匹配數量。

```
void match(node *root,const string &S){
    node *p = root;
    for(char c: S) {
        while(p != root && !p->ch[c-'a']) p = p->fail;
        if(p->ch[c-'a']) p = p->ch[c-'a'];
        p->cnt += 1;
```

```
    }
    reverse(BFS.begin(), BFS.end());
    for(node *p: BFS) {
        for(int id: p->endof) {
                cout << "Match pattern #" << id << " " << p->
                    cnt << " times\n";
        }
        p->fail->cnt += p->cnt;
    }
}
```

一個小優化

看看我們的match函式可以發現，如果我們要 match 很多個（或很多次）大字串，每次都得跑 failure，感覺很浪費時間呢！是不是可以預先處理好如果以某個字元失配會跳到哪個節點呢？答案是肯定的！這樣做的優點還有在 AC 自動機上來回走的時候保證是 $O(1)$，而跳 fail 的方法僅保證一次匹配的均攤是好的，不能保證來回走的時候不會跳太多次（後面有習題似乎需要這個優化，嘻嘻）。下面的 code 直接用ch來儲存一個節點遇到一個字元應該跳到哪個節點，若會需要用到這個 trie 原來的結構可以另外開一個空間儲存。

```
void buildFail(node *root) {
    queue<node *> Q;
    for(int c=0; c<K; c++) {
        if(root->ch[c]) {
            root->ch[c]->fail = root;
            Q.push(root->ch[c]);
        } else root->ch[c] = root; // base case很重要
    }
    while(!Q.empty()) {
        node *p = Q.front(); Q.pop();
        for(int c=0; c<K; c++) {
            node *f = p->fail;
            if(p->ch[c]) { // 沒失配繼續往下走
                p->ch[c]->fail = f->ch[c];
                Q.push(p->ch[c]);
            } else {
                p->ch[c] = f->ch[c]; // 失配跳fail
            }
        }
    }
}
void match(node *root, const string &S) {
    node *p = root;
    for(int i = 0; i < S.size(); i++) {
        p = p->ch[s[i]-'a'];
        // do something on node p
    }
}
```

更多活用

這邊有一個引理

引理 2.4.1: 樹狀結構

所有 failure pointer 會形成一棵樹，因為恰好有（節點數-1）條指向別人的 failure pointer，並且每個節點的 failure pointer 唯一且比自己淺，所以不會有環。

有些題目會要求在 AC 自動機上動態一些的做事，因為一個節點可能對他的祖先或是整個子樹產生影響，我們可以用樹壓平解決。

AC 自動機另一個優點是，他把所有可能跟小字串匹配的狀態從 $\prod |P|$ 壓到 $\sum |P|$（原本紀錄每個字串 match 到第幾個，現在只要知道 match 到哪個節點），因此也常常出現和 DP 配合的題目，例如筆者寫這份講義左右的入營考最後一題。

2.5 習題

以下題目有可能不一定要用到 SA 或 AC 自動機，請自行發揮創意找出更多不同解法！

習題 2.5.1: 似曾相識 (TIOJ 1515)

給定一字串 s，請問在此字串中重複出現兩次以上的最長字串長度為何 (若無則輸出 0)？

習題 2.5.2: 字串中的字串 (TIOJ 1306)

給你一個字串 T，以及很多字串 P。對於每個 P 請輸出 P 在 T 中出現過幾次。(T、P 都是由小寫字母所組成，長度 $\leq 10^4$)

習題 2.5.3: 猜謎遊戲 (Guess)（TIOJ 1091）

給定 $S, P(|S| > |P|)$，請問至少要在 S 中刪除幾個字元才能使得其不包含 P 作為子字串？
註：字元僅有 AB 兩種。$|P| \leq 100, |S| \leq 1000$

習題 2.5.4: Letters and Question Marks（CF 1327G）

給一個字串 S，還有一些小字串 $[t_1, t_2, \ldots, t_k]$，每個小字串 t_i 有價值 c_i，同時 S 和 t 除了問號以外都只包含前 14 個英文字母（a 到 n）。
S 包含了不超過 14 個問號，這些問號必須被填入兩兩相異的字母，請找出所有可能的填入方法中，總價值最大可以是多少。總價值的計算方法，是對每個小字串計算在該字串中的出現次數乘上該小字串的價值，最後直接相加。
$k \leq 1000, \sum |t_i| \leq 1000, |c_i| \leq 10^6, |S| \leq 4 \cdot 10^5$

習題 2.5.5: Indie Album（CF 1207G）

Mishka 最愛的樂團發行了新專輯！專輯中包含了 N 首歌，每首歌的歌名 s_i 可能是以下兩種：

1. 1 c 表示 s_i 是一個單獨的字母 c
2. 2 j c 表示 s_i 是 s_j 再加上一個字母 c（$j < i$）

Vova 對這張專輯很感興趣，但他沒有足夠的時間去聆聽所有歌曲，因此他詢問了 Mishka 一共 M 個問題，每個問題包含了 i 和一個字串 t，表示他想知道 s_i 裡面出現了幾次 t。Mishka 不知道他為何要問這些問題，但他盡力提供訊息。你可以幫助 Mishka 回答 Vova 所有的問題嗎？
所有字串僅包含小寫拉丁字母，$N, M \leq 4 \cdot 10^5, \sum |t| \leq 4 \cdot 10^5$

習題 2.5.6: 同步 (Sync)（TIOJ 1927）

在一個多人單向卷軸遊戲中，有 $N \leq 10^5$ 個格子，每個格子都有一個不超過 10^9+6 的正整數，代表該格的狀況。有時遊戲中的兩人會產生「同步」的現象。產生同步的條件是兩人所在的格子的數字 a, b 分別滿足 $(ab)^{\frac{p-1}{2}} \equiv 1 \pmod{p}$，其中 $p = 10^9 + 7$。產生同步後，兩人會瞬移至下一格。如果在下一格又產生「同步」，則會繼續往下走，直到其中一人超出格子範圍 (到了終點了) 或者兩人不再同步。
現在有 $Q \leq 10^6$ 個詢問，每次給定 N 個格子中的正整數以及兩人的起始位子，請計算他們兩個將會「同步」幾次。