解題思路 1

1.1 前言

在程式競賽中，有許多的題目是多方面的——搭配了資料結構、演算法、還有許多的巧思：這些巧思或許真的是通靈，但是也有許多是有跡可循的，或者可以讓算法變得更快，要不然能提供有用的思考方法。這些方法是比較常用的，在解題若想不大到可以往這些方面想；也有可能你認為這些概念非常籠統，沒辦法實際應用。沒錯！這些就是比較抽象的思考方向，提供可能的路，而要如何走在於你。

1.2 二分搜

猜數字的例子

這是最基礎的思考路徑，很多時候可以將一個演算法的複雜度從線型硬壓成對數複雜度！來看一個簡單的例子，就知道是什麼意思了：

習題 1.2.1: Guess My Number（TIOJ 1044）

一開始電腦會產生從 1 到 N 的一個正整數 K，而你每次可以問一個數字 Q，而電腦會回傳 K 和 Q 的大小關係；請猜出 K 為何？($1 \le N \le 10^{18}$)

第一個想法一定會是從 1 猜到 N，一路猜到 N，一定會中！然而，這樣盲目的猜只能得到殘忍的 TLE——$N = 10^{18}$ 的話，實在跑太久了！而且，沒有充分利用題目給的資訊：大小關係一定很重要！就來想，如果跟你說：「Q 這個數字太大了！」，代表什麼？代表，對於所有的 $X \le Q$，都不用猜了，一定不會是答案；如果跟你說：「Q 這個數字太小了」，當然也是一樣，比 Q 小的數字都不用猜了。利用這個資訊，我們可以每次都紀錄目前「答案可能在的區間」$[L, R]$，一開始 $[L, R] = [1, N]$，而每次我們都問 $M = \lfloor \frac{L+R}{2} \rfloor$，所以區間就可以變成 $[L, M-1]$ 或 $(M+1, R)$ 了。顯然地，每次新的區間都會比舊的區間小一半，所以想要區間長度變成 1（也就是答案只有一個可能）需要 $\log_2(N) = O(\log N)$ 次猜測。所以呀，透過了這樣的「砍半」，可以有效將搜尋從 $O(n)$ 變成非常好的 $O(\log N)$！這種技巧稱為**對答案二分搜**。

更難的例子

當然，二分搜的題目沒有都那麼仁慈；來看一個稍微難一點的題目吧！

習題 1.2.2: Solve It（UVa 10341）

給定整數 $0 \le p, q \le 20$ 和 $-20 \le q, s, t \le 0$，請解方程

$$f(x) = pe^{-x} + q\sin(x) + r\cos(x) + s\tan(x) + tx^2 + u = 0$$

且 $0 \le x \le 1$。若無解，請輸出`-1`。

我知道這裡有許多數學好的人，會想要用數學方法直接推解答——別這樣，不需要！只需要知道「一個數字和解答的大小關係」就可以了，而顯然可以 $O(1)$（假設運算皆為 $O(1)$ 算出來：

```
double eps = 1e-9;

double f(double x){
    return (p*exp(x) + q*sin(x) + r*cos(x) + s*tan(x) + t*x*x + u;
}
```

也就是是否 $|f(x)| < 10^{-9}$，如果是，就說是解了。

精度問題：因為電腦存浮點數的方法是近似，所以不能直接判斷是否為 0，只能判斷是否夠靠近。

那要怎麼二分搜呢？就用上次的 L，R 吧，一開始答案的值域是 $(L, R) = (0, 1)$：

```
double l = 0, r = 1, m = (l + r)/2;
while(r - l > eps){
    if(f(m) > 0){
        l = m;
    } else {
        r = m
    }
    m = (l + r)/2;
}
```

而這裡的陷阱就是：$f(x)$ 在 $[0, 1]$ 上遞減（驗證便得），所以寫的時候要小心。

二分搜什麼時候會用？

二分搜也不是萬能的，也會有限制；而知己知彼，才能百戰百勝，所以需要來認識二分搜的短處。首先，如果是這樣的題目：

習題 1.2.3: 這是微分嗎（經典問題）

電腦生成了一個整數係數二次函式

$$f(x) = ax^2 + bx + c \ (a > 0)$$

（$a > 0$）每次可以猜一個數字 X，而我會給你 $f(X)$ 的值。請猜出 $f(x)$ 的最小值發生在哪裡？

你可能想要對 $f(K) - f(Q)$ 二分搜到 0，但是有一個問題：三次方程中間有凹下去的地方，如果搜到那邊的話，會停在凹槽的最底端，而因為兩邊都比較大，而回傳錯誤的答案。所以：得到了結論：**對答案二分搜的時候，被二分搜的變數必須有單調性，或是有明確的分界（也就是說，在一個臨界值之前都不符合，而在一個臨界值之後都符合，找最小符合的答案的時候會需要）**

三分搜

那要如何解決上一題呢？這裡要介紹一個相對比較少用（但不代表不重要）的資訊！三分搜不同於二分搜在於它可以處理一個「轉折」，也就是不單調的地方可以有一個。這裡，還是假設目前答案區間在 $[L, R]$，則可以猜兩個數字 $A = \lfloor \frac{2L+R}{3} \rfloor$，$B = \lfloor \frac{L+2R}{3} \rfloor$，將區間分為三等份（以後稱為左、中、右）。根據三一律，可以分為三個 case：

1. $f(A) > f(B)$，則可以知道 $[L, A]$ 一定沒有最小值，新的答案區間為 $[A, R]$
2. $f(A) = f(B)$，可以知道 $[L, A]$，$[B, R]$ 都不存在最小值，所以新的答案區間為 $[A, B]$
3. $f(A) < f(B)$，同（1）可以知道 $[B, R]$ 一定沒有最小值，新的答案區間為 $[L, B]$

在最差情況下，每次更新區間，長度都會變成原本的 $\frac{2}{3}$，所以複雜度 $\log_{\frac{3}{2}} N = O(\log N)$

STL 中的二分搜

相信大家都覺得寫二分搜很麻煩吧！幸好，STL 也有幫忙二分搜的函數：lower_bound和upper_bound！給定一個**排序好**的序列，lower_bound(iter a, iter b, T t)回傳在a和b這兩個指標之間，第一個回傳大於或等於t的元素。另外一個函式upper_bound(iter a, iter b, T t)回傳在a和b這兩個指標之間，第一個回傳大於t的元素。請看以下的範例：

```
1 int arr[5] = {4, 5, 13, 71, 92};
2 cout << *lower_bound(arr, arr + 5, 5) << endl;
3 cout << *lower_bound(arr, arr + 5, 6) << endl;
4 cout << *upper_bound(arr, arr + 5, 5) << endl;
5 cout << *upper_bound(arr, arr + 5, 6) << endl;
```

輸出：

```
5
13
13
13
```

習題

來練習一下題目吧！

習題 1.2.4: Wifi （UVa 11516）

一條街上有編號為 1 到 N 的房子，編號為 i 的房子都和相鄰的房子距離 1 個長度單位。其中 M 個居民（都住在不同房子）想要買最多 K 個發射台（每一個發射台都有固定的發射半徑 R），使得這 M 個居民的房子都有至少一個接收台在其發射半徑內。請問，如果可以任意放發射台（與房子同位置也可以），最小的半徑 R 為何？

習題 1.2.5: Sagheer and Nubian Market（CF 812C）

小沙來到了一個市集，而這個市集賣 n 個東西，編號為 1 到 n。而每一樣東西都有一個「基本價」a_i。但是，這個市集的計價方式很特別：若小沙總共買了 k 樣東西，編號分別為 $x_1, x_2, x_3, \ldots x_k$，則第 i 項東西的價格是 $a_{x_i} + kx_i$。小沙只有 S 元，而他想要買最多樣東西。請輸出小沙最多可以買幾樣東西和買那麼多樣東西所花費的最小價格。（$1 \leq n, a_i \leq 10^5$，$1 \leq S \leq 10^9$）

1.3 動態規劃 (Dynamic Programming)

動態規劃（簡稱 DP），根據維基百科的定義，是一種通過把原問題分解為相對簡單的子問題的方式求解複雜問題的方法。當我們碰到一個複雜的問題時，有時直接求得此問題的答案是非常耗時或甚至難以完成的，因此我們會想到，或許可以利用原問題的子問題經過一些轉換後得到原問題的答案，而這就是動態規劃最開始的思維。讓我們先以一個大家耳熟能詳的數列來作為例子吧！

習題 1.3.1: no judge

輸入 n，求費氏數列第 n 項。$n \leq 10^7$

相信大家都知道什麼是費氏數列，也很輕鬆可以完成這樣一份程式碼：

```
    int f(int n) {
        if (n <= 2) return 1;
        return f(n-1) + f(n-2);
    }
```

經過一些數學推導後我們會發現上述程式碼的複雜度為 $O(\varphi^n)$ ，此處 $\phi = \frac{1+\sqrt{5}}{2}$，倘若要計算 $f(10^7)$，我們大約需要經過 10^{2*10^6} 筆操作，現行的超級電腦約可以達

到每秒 10^12 左右的運算次數，一世紀大約有 $3*10^9$ 秒，我們大約需要經過 10^{2*10^6} 世紀後才有機會跑完這份 code。這時候，我們可以考慮如下的程式碼：

```
int f(int n) {
    int arr[10000005];
    arr[1] = 1;
    for (int i = 2; i <= n; i++) {
        arr[i] = arr[i-1] + arr[i-2];
    }
    return arr[n];
}
```

上述程式碼的複雜度為 $O(n)$ ，倘若要計算 $f(10^7)$，以目前大家手邊的電腦，也僅需要不到一秒即可輸出結果。
上述的例子是 DP 最基礎的應用，欲先算好每個子問題的答案並紀錄之，便可以迅速求得母問題的答案，達到「以空間換取時間」的效果，這個概念十分抽象，但應用卻非常廣泛，講義後面會舉出更多例題供大家參考。

滾動 DP

有時候我們會發現，有些子問題的答案在某個時間後就再也不會被用到了，這時候我們就可以將新的答案覆蓋上去，進而達到節省空間的效果。一樣用費氏數列來舉例，請大家看看如下程式碼：

```
int f(int n) {
    int a, b, c;
    a = b = 1;
    for (int i = 3; i <= n; i++) {
        c = a + b;
        a = b;
        b = c;
    }
    return c;
}
```

這樣便完成了時間 $O(n)$、空間 $O(1)$ 的費氏數列演算法。而事實上，已知費氏數列的最佳演算法複雜度是 $O(\log n)$ ，詳情請參考矩陣快速冪的部分。

例題討論

接下來會講解一些最基本的 DP 問題

習題 1.3.2: 最大連續和 (No Judge)

給定一個長度為 n 的序列 $a_1, a_2, ..., a_n$ ，求出這個序列中最大的區間和。

建立一個陣列 arr，$\text{arr}[i]$ 代表以 a_i 為右界所能產生出的最大連續和，可以列出轉移式 $\text{arr}[i] = \max(\text{arr}[i-1]+a_i, a_i)$，而由於第 i 項的答案最多只需用到第 $i-1$ 項的答案，因此可以利用滾動法來實作。

```
int f(int n, int a[]) {
    int now; // 表示以當前指標為右界的答案
    int ans; // 表示最終答案
    for (int i = 1; i <= n; i++) {
        now = max(now + a[i], a[i]);
        ans = max(ans, now);
    }
    return ans;
}
```

習題 1.3.3: 背包問題 (No Judge)

你有一個背包，裡頭最多能裝重量總和為 W 的物品，而你有 N 個物品，第 i 項物品分別有價值 v_i 與重量 w_i，請求出在不超過背包重量限制的情況下，最多能夠拿到多少價值的物品。

建立一個陣列 arr，$\text{arr}[i]$ 代表當重量為 i 時所能裝下的最大價值，接著分別利用每個物品來進行轉移，$\text{arr}[i] = \max(\text{arr}[i - w_i]+v_i, \text{arr}[i])$，而過程中為大的 $\text{arr}[i]$ 就是我們所要的答案。

```
int f(int n, int W, item C[]) {
    int arr[MAXN+5], ans;
    for (int i = 0; i < n; i++) {
        for (int j = W; j >= C[i].w; j--) {
            arr[j] = max(arr[j], arr[j-C[i].w] + C[i].v);
            ans = max(ans, arr[j]);
        }
    }
    return ans;
}
```

習題

來寫習題囉

習題 1.3.4: 無限背包問題 (No Judge)

你有一個背包，裡頭最多能裝重量總和為 W 的物品，而你有 N 種物品，每種物品分別有價值 v 與重量 w，而數量則沒有限制，請求出在不超過背包重量限制的情況下，最多能夠拿到多少價值的物品。

習題 1.3.5: 進化版最大連續和 (CF 1155D)

一樣是求最大連續和，只是你多了一個操作，可以將某個區間 $[l, r]$ 的值都同乘以 k，且只能做一次。($n \leq 3 \cdot 10^5$)

一些常用的東東?

2

2.1 min/max

其實就是一個很簡單的取最大值和取最小值，$min(a,b)$ 回傳 a 和 b 的最小值（constant reference type），其中比較的方法是用小於（$<$）運算子，所以 a,b 若是自己寫的 struct，那就必須重載小於運算子或者寫比較函式（以下提及）。而 $\min(a,b,\text{cmp})$ 會回傳 a 和 b 在 cmp 比較函式下的最小值。值得一提的是，參數 a 和 b 必須是相同型別，否會造成編譯錯誤（例如 $\max(\text{int},\text{long long})$ 就不行）。

2.2 離散化

離散化是利用「k 是第幾大的」來代表 k，這樣可以使數字範圍變小且保持大小順序。這個技巧在以後會經常用到，因為有時候數字太大，要開值域陣列會超過記憶體限制。

unique

這是一個神奇的 std 函式，參數傳入首尾指標或迭代器（左閉右開），可以將序列相鄰重複出現的元素丟掉，剩下的往前移（陣列大小不變，因向前移動而空出之位置的值是不確定的 undefined behavior）
利用這樣的工具，若我們先將陣列排序好，重複的數字一定會被放到最後面，剩下的就是不重複、排序好的數字，其 index 就會代表他是第幾大的數。上述複雜度為排序的 $O(N \log N)$，以下範例程式碼：

```
1 void process(vector<int> &v){//notice '&'
2     ///{1,4,2,5,3,5,4,8,5,3}
3     sort(v.begin(),v.end());
4     ///{1,2,3,3,4,4,5,5,5,8}
5     v.erase(unique(v.begin(),v.end()),v.end());
6     ///unique->{1,2,3,4,5,8,x,x,x,x}
7     ///where 'x' is indeterminate
8     ///erase->{1,2,3,4,5,8}
9 }
```

set

另一種做法是將所有數字丟進set中，再從頭掃一遍，一一賦值。這個想法比較直觀，複雜度同為 $O(N \log N)$ 但常數較unique的方法大。以下為範例程式碼：

```
void process(vector<int> &v,map<int,int> &m){
      ///v is the original,m is converting map
    m.clear();
    for(auto &i:v) m[i]=0;
    v.clear();
    int t=0;
    for(auto &i:m) v.push_back(i.first),i.second=t++;
}
```

2.3 單調隊列

單調隊列顧名思義就是要保持一個隊列的單調性。直接講實在太抽象了，不如直接來看例題吧！

習題 2.3.1: 簡單易懂的現代都市 (TIOJ 1566)

輸入 N、M、K，以及 N 棟大樓的高度 H_i。輸出每組 l, r，使得 $r - l = m - 1$ 且第 l 棟到第 r 棟的高度最大值減最小值恰為 K（題目邊界有些特例，不過不重要）。$(2 \leq N \leq 10^7, 2 \leq M \leq 10^6, 1 \leq K \leq 2^31)$

直覺的想法就是每次找每個區間最大值及最小值相減若為 K 就輸出，為了方便思考，我們先將問題簡化：

習題 2.3.2: 固定區間的最大值

輸入 N、M，以及 N 棟大樓的高度 H_i。輸出每相鄰 M 棟大樓的高度最大值。$(2 \leq N \leq 10^7, 2 \leq M \leq 10^6)$

假如暴力做複雜度 $O(NM)$ 一定會 TLE，另一個較聰明的想法是每次將範圍內的高度丟入二元搜尋樹中，最大值便可以 $O(\log M)$ 查到，總複雜度為 $O(N \log M)$，相當危險。不過遵循著這個作法，我們知道從區間 $[1, m]$ 到 $[2, m + 1]$ 只需要將 H_1 刪除並插入 H_{m+1} 就可以了。
然而我們又發現一個性質：假如新插入一個 H_i，那原本的這 M 個高度當中比 H_i 小的全部都不會再被用到了！要怎麼維護這些性質呢？我們拿一個deque一一插入最前面的 M 個高度，假設要插入的高度為 H_i，那就先將dequeue尾端小於 H_i 的全部pop掉，最後再將自己插入尾端。用這樣的方式不但只保留可能被用到的數字，還能維持dequeue內數字嚴格遞減，使插入時小於 H_i 的數字一定都在最後面。而這樣查詢時只要看最前面的元素就會是最大值了。然而離 AC 還有一小步，那就是該怎麼刪除呢 ??? 事實上很簡單，我們只要知道每個元素是第幾棟，查詢前先不斷pop掉最前面不在考慮範圍內的那些數字，就可以取到最大值了！

利用上面的方法，同時處理最大值和最小值就可以快速找到答案了。詳細的程式碼如下：

```
signed main() {
  int n,m,k,i;
  vector<pair<int,int>> ans;
  deque<pair<int,int>> maxx,minn;
  cin>>n>>m>>k;
  for(i=0;i<n;i++)
    cin>>h[i];
  for(i=0;i<m-1;i++){
    while(maxx.size()&&maxx.rbegin()->first<h[i])
      maxx.pop_back();
    maxx.push_back(make_pair(h[i],i));
    while(minn.size()&&minn.rbegin()->first>h[i])
      minn.pop_back();
    minn.push_back(make_pair(h[i],i));
    if(maxx.begin()->first-minn.begin()->first==k)
      ans.push_back(make_pair(1,i+1));
  }
  for(;i<n;i++){
    while(maxx.size()&&maxx.rbegin()->first<h[i])
      maxx.pop_back();
    maxx.push_back(make_pair(h[i],i));
    while(maxx.size()&&maxx.begin()->second<=i-m)
      maxx.pop_front();
    while(minn.size()&&minn.rbegin()->first>h[i])
      minn.pop_back();
    minn.push_back(make_pair(h[i],i));
    while(minn.size()&&minn.begin()->second<=i-m)
      minn.pop_front();
    if(maxx.begin()->first-minn.begin()->first==k)
      ans.push_back(make_pair(i-m+2,i+1));
  }
  for(i=0;i<m-1;i++){
    while(maxx.size()&&maxx.begin()->second<=i+n-m)
      maxx.pop_front();
    while(minn.size()&&minn.begin()->second<=i+n-m)
      minn.pop_front();
    if(maxx.begin()->first-minn.begin()->first==k)
      ans.push_back(make_pair(i+n-m+2,n));
  }
  cout<<ans.size()<<endl;
  for(i=0;i<ans.size();i++)
    cout<<ans[i].first<<" "<<ans[i].second<<endl;
}
```

基礎資料結構與 STL 3

資料結構是在程式裡儲存資料以方便處理的方法，不同的資料結構有各自的優點與缺點，選擇適當的資料結構可以使得資料的查找、添增或刪除更有效率。

C++ 的**標準模板庫** (Standard template library, STL) 提供許多十分強大的內建資料結構可以使用，熟悉這些工具既可以減少許多賽場上不必要的精力花費，又可以降低出錯的可能性。

這裏會先介紹最基本的資料結構，包含`stack`、`queue`、陣列的延伸`vector`、Linked List 和一些比較進階的：`priority_queue`、`map`、`set`等資料結構。前四個基本資料結構建議自己要實作看看，之後大部分都是用 STL 的東西，知道概念之後就可以直接用。

這裏重要的是要熟悉用法和變化題，其中前四個資料結構一定要自己寫出來一遍，知道內部運作。

3.1 標頭檔與 std 名稱空間

STL 的東西都是寫在`std`名稱空間之下的，因此在使用這些工具之前，都必須加入這兩行程式碼：

```
1 #include <bits/stdc++.h>
2 using namespace std;
```

其中的`<bits/stdc++.h>`是一個對於競程選手友善的標頭檔，它涵蓋了幾乎所有演算法競賽所需要的工具，以下所有工具都包含在這個標頭檔下，當然你也可以一個一個`include`。

3.2 動態陣列 (Dynamic arrays)

最基礎的資料結構大概就是陣列了，相信大家也都會使用。

STL 的方便之處是它提供了一個動態型的陣列`vector`，它可以像一般的陣列一樣操作，並且可以隨時更動陣列大小以配合內部元素數量的增減。

vector

我們可以用以下方式來宣告vector

```
vector<int> vecA;
// 宣告一個元素型別為int的空vector
vector<int> vecB(10);
// 宣告一個長度為10的vector，初始值為0
vector<int> vecC(10,2);
// 宣告一個長度為10的vector，初始值為2
vector<int> vecD = {1, 2, 3, 4, 5}; // 賦予初始值
vector<int> vecE {1, 2, 3, 4, 5}; // 也可以這樣賦予初值
```

< >中的資料型別可以替換，代表陣列中元素的型別。

增添與查找

vector的push_back函數可以將元素新增到vector的最尾端；pop_back函數則刪除最尾端的元素。

```
vector<int> vec;
vec.push_back(1); // [1]
vec.push_back(4); // [1,4]
vec.pop_back(); // [1];
```

vector與一般陣列一樣，可以用operator[]進行查找元素。另外也可以透過 STL 內建的at函數與back函數來進行元素的查找。

```
cout << v[0] << endl;
// 印出vector的第0個元素
cout << v.at(0) << endl;
// 也可以這樣
cout << v.back() << endl;
// 印出vector尾端的元素值
```

尋訪vector裡的每個元素可以用更簡短的寫法：

```
for(auto &x:vec){
    cout << x << endl;
}
```

size函數回傳有多少元素在這個vector裡面

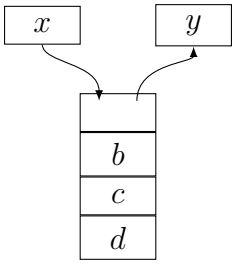
```
for(int i=0;i<vec.size();i++)
    cout << vec[i] << endl;
```

3.3 堆疊 (Stack)

堆疊，顧名思義就是一堆資料疊成一堆，每次都只能從最上面存取資料，否則整個堆疊就會倒塌，這正是所謂先進後出 (first in last out, FILO) 原則。

實作原理

假設我想要疊盤子，將盤子疊在一起，**每次拿到的就是最上面的盤子，放盤子也是最上面**，那就輪到stack出來了！正好就是stack的用武之地。



以上就是stack的示意圖：只能從最上面拿和放（此處原本最上面是 y，可以拿走；或者可以從最上面放 x 進去。）那要怎麼實作呢？只需要用一個陣列即可，那還需要什麼資訊呢？需要維護這個stack的最上方的位置，隨著東西的push、pop而移動。

習題 3.3.1: stack 練習

請實作一個stack，需要支援兩種操作：

1. PUSH x，代表stack內要放入一個值為 x 的物體
2. POP，代表stack內要將最上層的東西拿出來，並輸出其值。如果stack是空的，請輸出stack is empty!

```cpp
int Stack[maxn], tot = 0;
void PUSH(int x) {
    Stack[tot++] = x;
}
void POP(){
    if(tot == 0)
        cout << "stack is empty!\n";
    else
        cout << Stack[--tot] << endl;
}
```

stack

STL 也有內建的stack容器適配器，有新增、刪除、查詢最頂部元素的功能。基本上 STL 中stack能做到的事情vector都能做到，事實上stack就是用vector實作完成的。

容器適配器（container adapter）是利用 STL 原有的容器 (如vector) 去實作的介面。利用容器適配器可以使得程式碼更加簡潔，增加程式可讀性。STL 的stack、queue與priority_queue都是容器適配器的一種。

以下是宣告一個stack的方法。

```
stack<int> sta; // 宣告一個儲存int型別的stack
```

stack是放在<stack>標頭檔內，使用前記得引入。

堆疊操作

push() 可以將資料放入堆疊的頂部；
pop() 將頂端元素刪除；
top() 查詢堆疊頂端元素；
size() 查詢目前還位於堆疊中的資料數；
empty() 回傳堆疊是否為空。

```
sta.push(3); // [3]
sta.push(2); // [3,2]
sta.push(5); // [3,2,5]
sta.pop(); // [3,2]
cout << sta.size() << endl; // 2
cout << sta.empty() << endl; // 0 (表示非空)
cout << sta.top() << endl; // 2
```

常犯錯誤：在使用top()函數時記得要加括號，不要 CE 在哪裡都不知道。

習題

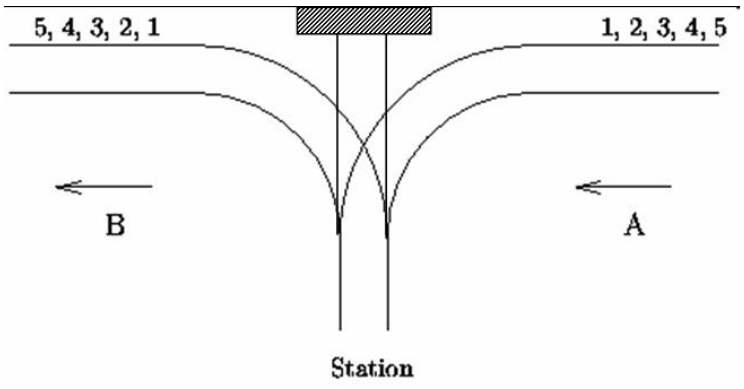
來做題目吧！

習題 3.3.2: Parentheses Balance (ZJ b304, UVa 673)

給你一個由四個字元（'('、')'、'{'、'}'）所組成的字串 S，請問 S 是否為一個合法的字串（我們定義字串為合法，若且唯若每一個括弧都可以被匹配到，"()()"合法，而"{ (})"則不合法。），且 $|S| \leq 10^5$。

習題 3.3.3: Rails (TIOJ 1012, UVa 514)

現在有一個火車站，和編號為 $1, 2, 3, \cdots n$ 的火車，身為火車長的 03t 想要將之重新排列。火車的排列如下：



(此處，想要的重新排列是 $5, 4, 3, 2, 1$，而火車可以做的事就是進站，如果 a 號火車進站了，之後又有一個 b 火車進站，則 b 火車得先出站，a 火車才能出站（太窄了 !)，請問 03t 所想要的重新排列是否可能？($n \leq 10^5$)

習題 3.3.4: Cows (TIOJ 1176)

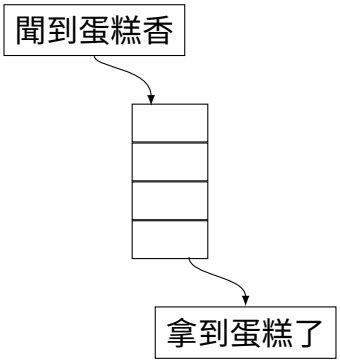
這一題比較難！
有 N 隻牛排成一條線，每一隻牛都有高度 a_i，這些牛很喜歡看來看去，而第 i 隻牛看得到第 j 隻牛的條件就是中間沒有牛擋住了視線，也就是不存在 $i < k < j$ 使得 $a_k > a_i$。給定 N 和 $a_1, a_2, a_3, \cdots, a_N$，請問每一隻牛可以看得到幾隻牛？($N \leq 10^6$)

3.4 佇列 (Queue)

佇列，顧名思義就是一堆人在排隊，先進去排隊的人先享有辛苦排隊的成果，這是所謂先進先出 (first in first out, FIFO) 原則。

實作原理

想像一家熱門的蛋糕店，有很多人在排隊，而每一個人都不會插隊，那要怎麼模擬呢？當最前面的人拿到了夢寐以求的蛋糕的時候，就會從**最前面離開隊伍**，而當有一個人聞香而來，就會從**最後面加入隊伍**。這就是queue的精神！以下為蛋糕店排隊的模擬圖：



和stack不同的是：一個是拿、放同側，而這個是異側，是 FIFO（First In Last Out）資料結構。實作方法比較難一點，得在陣列維持兩個變數（也就是頭和尾），插入的時候動尾，而離開的時候動頭。請注意：這裏的 Queue 實作方法是循環的，不再和操作數有關，而和元素數量有關。

習題 3.4.1: queue 練習

請實作出一個queue吧！要支援兩個操作：

1. PUSH x，代表queue的最後方要插入一個值為 x 的物體。
2. POP，代表queue內要將最前面的東西拿出來，並輸出其值。如果queue是空的，請輸出"queue is empty!"

```
int Queue[maxn], Front = 0, Back = 0;
void PUSH(int x) {
    Queue[Back] = x;
    if(++Back >= maxn) Back -= maxn;
}
void POP(){
    if(Front == Back)
        cout << "queue is empty!\n";
    else{
        cout << Queue[Front] << endl;
        if(++Front >= maxn) Front -= maxn;
    }
}
```

假設這個佇列最多只能存 N 個元素，當下一個元素要被放進第 $N+1$ 格時，因為有 pop 操作的存在，所以這個佇列不一定是滿的狀態。通常為了有效節省實作佇列時所需的空間，我們都會用循環的方式來實作 queue，意即假設原本存在第 1 格的元素已經被 pop 掉，我們就可以將原本要儲存在 $N+1$ 格的元素儲存在第 1 格，重複利用能用的空間。

queue

STL 也為佇列設計了一個容器適配器，預設的容器跟stack一樣是vector；它支援從後面插入資料，但是資料是從前面取出。

以下是queue的宣告方法。

```cpp
queue<int> que; // 宣告一個儲存int型別的queue
```

queue是放在<queue>標頭檔內，使用前必須引入。

佇列操作

push() 可以將資料排入佇列的尾端；
pop() 將前端元素刪除；
front() 查詢佇列前端元素；
size() 查詢目前還位於佇列內的資料數；
empty() 回傳佇列是否為空。

```cpp
que.push(3); // [3]
que.push(2); // [3,2]
que.push(5); // [3,2,5]
que.pop(); // [2,5]
cout << que.front() << endl; // 2
```

新手常犯錯誤：是front()，不是top()，不要搞錯了。

習題

做習題很重要！

習題 3.4.2: Team queue (UVa 540)

一堆人要排隊去買東西，但是還有一個限制：每一個人可能會是 M 個組中的成員之一（每一個人最多只會加入一個組），所以排隊會有一個新的規則：如果一個人要去排隊了，但是隊伍中有同組的人的話，那這個（有點缺德）的人就會插隊插到自己組的末端。會跟你說隊伍有誰，和一堆進入隊伍／最前面的人離開的指令，請模擬這個情況。(指令數 $\leq 2 \times 10^5$，$M \leq 10^3$)

習題 3.4.3: Throwing cards away I（UVa 10935）

我現在有 n 張牌寫上了 1 到 n 的正整數，一開始由上而下是 $1, 2, 3, \cdots, n$，而我每次會進行這個操作：只要有兩張牌以上，就把第一張牌拿掉，並且將最後一張牌放到最下面。請輸出牌被拿掉的順序？($n \leq 10^5$)

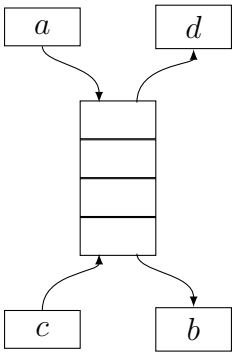
3.5 雙端佇列 (Deque)

在介紹這個資料結構之前，要先決定這個資料結構的唸法。

定理 3.5.1: Deque 的正確唸法

根據 Knuth 的 *The Art of Computer Programming*，Volume 1, Section 2.2.1 "Stacks, Queues, and Deques"：「A deque ("double-ended queue") is a linear list for which all insertions and deletions (and usually all accesses) are made at the ends of the list. A deque is therefore more general than a stack or a queue; **it has some properties in common with a deck of cards, and it is pronounced the same way.**」正確唸法為 [dɛk]，如果念為 [di:kju:] 可能會誤認為 *dequeue*，意思是從queue中移除，造成誤會。

這個東西感覺像是queue和stack的進化版，**可以從頭尾拿，也可以從頭尾放東西**，見以下示意圖：



所以會支援四個操作：兩邊各兩個，放和拿：push_back、push_front、pop_back、pop_front。那為什麼要用stack和queue，而不直接用deque呢？因為不需要那麼多的功能，還會影響程式的可讀性：如果每次都用deque，但是stack比較好懂，不要每次都註解這個是stack，queue之類的。那要怎麼實作呢？需要維護頭尾在哪裡，所以從這個方面實作。

習題 3.5.1: deque 練習

來實作deque吧！這一次，需要支援四個操作：

1. POP_FRONT 和 POP_BACK，分別代表要從前面和後面拿出東西出來並輸出拿出來的東西的值，如果沒有東西可以拿的話，那就輸出"deque is empty!"
2. PUSH_FRONT x 和 PUSH_BACK x，分別代表要從前面和後面插入一個值為 x 的物體。

   並且保證總共不會超過 N 個操作。

```
int DeQueue[maxn], Front = 0, Back = 0;
void PUSH_FRONT(int x) {
    if(--Front < 0) Front += maxn;
    DeQueue[Front] = x;
}
void PUSH_BACK(int x) {
    if(++Back >= maxn) Back -= maxn;
```

```
8       DeQueue[Back] = x;
9   }
10  void POP_FRONT(){
11      if(Front == Back)
12          cout << "deque is empty!\n";
13      else{
14          if(++Front >= maxn) Front -= maxn;
15          cout << DeQueue[Front] << endl;
16      }
17  }
18  void POP_BACK(){
19      if(Front == Back)
20          cout << "deque is empty!\n";
21      else{
22          if(--Back < 0) Back += maxn;
23          cout << DeQueue[Back] << endl;
24      }
25  }
```

deque

STL 裡面的deque也有雙端佇列的功能，可以在首端或尾端存取資料，宣告時要引入<deque>標頭檔。以下是deque的宣告。

```
1  deque<int> dq; // 宣告一個空的deque
```

雙端佇列操作

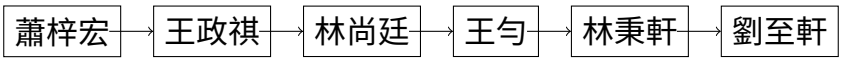
push_back() 可以將資料排入雙端佇列的尾端；
pop_back() 將雙端佇列尾端元素刪除；
push_front() 可以將資料排入雙端佇列的首端；
pop_front() 將雙端佇列首端元素刪除；
front() 查詢雙端佇列首端元素；
back() 查詢雙端佇列尾端元素；
size() 查詢目前還位於佇列內的資料數；
empty() 回傳佇列是否為空。

```
1  dq.push_back(3); // [3]
2  dq.push_back(1); // [3,1]
3  dq.push_front(2); // [2,3,1]
4  dq.pop_back(); // [2,3]
5  dq.pop_front(); // [3]
6  dq.push_front(1); // [1,3]
7  cout << que.front() << endl; // 1
8  cout << que.back() << endl; // 3
```

能用stack、queue、一般陣列、或vector解決的問題就盡量避免用deque。

3.6 鏈結串列 (Linked List)

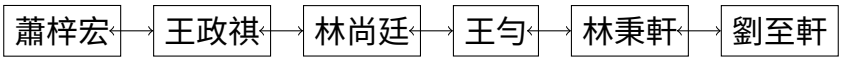
現在假設我們在玩一個遊戲：每一個人都要指一個人，那要怎麼紀錄這種關係呢？將每一個人想成一個東西，裡面包著自己的值和指向的人！這就是 Linked List 的精神。



這裏，蕭梓宏指向王政祺，王政祺指向林尚廷..... 直到劉至軒停止。

雙向串列（Doubly Linked List）

有時候會有人想要知道是誰指向他，就會再維護一個指標指向指向我的那個人，圖形大概長這樣：



作法就是呢：當 A 連到 B 的時候，將 A 指向的人設為 B，而 B 的被指的人設為 A。

查詢和修改和複雜度

如果想要查詢「一個東東指到誰？」的話，顯然這個可以在 $O(1)$ 內做完。那如果想要做的是查詢「我這個人一直指，指到第 k 個人會是誰呢？」那就得花 $O(k)$ 的時間，所以**串列不適合隨機存取**，也就是在只給串列頭的狀況下，查詢第 n 個的值，會變得很慢。

若要隨機存取，通常會搭配陣列來存（也就是呢，維持一個陣列來存）、陣列第 i 值存第 i 個人的值、指向下一個人的陣列值、和被指的人的陣列值。

實作時間

先來個經典問題。

習題 3.6.1: 實作噩夢之 linked list 篇

請實作一個linked list，需要支援四種操作：

1. INSERT x，在 id $= x$ 的節點之後插入一個新節點 (給予流水號 id)
2. DELETE x，刪除 id $= x$ 的節點
3. PREV/NEXT x，查詢 id $= x$ 的節點的前一項/下一項的 id
4. TRAVEL，依序輸出 linked list 內部的內容

對於前三種操作，若找不到 id = x 的節點，則輸出"The node does not exist." 於一行。注意，在此題中，id $= 0$ 的節點為鏈結串列的開頭。

模板題。在實作 linked list 時，我們通常會在最前面加上一個空節點，以利插入操作的實作方便以及一致性。

```
1  struct linked_list {
2      struct node {
3          node *prev, *next;
4          int id;
5          node(): prev(NULL), next(NULL){}
6      };
7      vector<node*> List;
8      // list[x] 儲存指向 id = x 的指標，這就是 linked list 通常搭配陣列的地方
9      linked_list(int n): List(n+1,NULL) {
10         List[0] = new node; // 起始空節點
11         List[0]->id = 0;
12     }
13     int prev(int x){ // 查詢前一項的 id
14         if(List[x] && List[x]->prev)
15             return List[x]->prev->id;
16         return -1;
17     }
18     int next(int x){ // 查詢下一項的 id
19         if(List[x] && List[x]->next)
20             return List[x]->next->id;
21         return -1;
22     }
23     bool Insert(int x, int id) {
24         // 在 id = x 的節點之後插入 id = id 的節點
25         if(!List[x]) return false;
26         node *n = new node;
27         List[id] = n;
28         n->next = List[x]->next;
29         if(n->next) n->next->prev = n;
30         List[x]->next = n;
31         n->prev = List[x];
32         n->id = id;
33         return true;
```

```
    }
    bool Delete(int x){
        // 刪除 id = x 的節點
        if(!List[x]) return false;
        if(List[x]->next)
            List[x]->next->prev = List[x]->prev;
        List[x]->prev->next = List[x]->next;
        delete List[x];
        List[x] = NULL;
        return true;
    }
    void travel(){ // 依序尋訪所有節點
        node *s = List[0]->next; // start
        while(s){
            printf("%d ",s->id);
            s = s->next;
        }
        puts("");
    }
};
```

list 的使用時機：當題目要考慮某個 id 的前後項的時候，便可以考慮使用 linked list 解題

習題 3.6.2: 陸行鳥大賽車（NeOJ 21）

輸入說明
第一行包含一個整數 $N(N \leq 10^5)$，代表現在有 N 個玩家，編號 1～N 的玩家目前分別為第 1～N 名 (編號 1 第 1 名、編號 2 第 2 名...)。
第二行包含一個整數 $M(M \leq 50,000)$，代表接下來會依序發生 M 個事件。
接下來的 M 行，每行包含兩個整數 $T_i, X_i(0 \leq T_i \leq 1, 1 \leq X_i \leq N)$，$T_i$ 為 0 的時候，代表編號 X_i 的玩家遭受攻擊，然後離開遊戲；T_i 為 1 的時候，代表編號 X_i 的玩家使用衝刺，無條件超越當前名次比他高一名的玩家。
測試資料保證玩家被淘汰之後不會再出現任何紀錄。

輸出說明
輸出一行，包含 Y 個整數 (Y 是剩餘玩家的數量)，由名次小到大依序輸出，整數兩兩間以空白隔開。

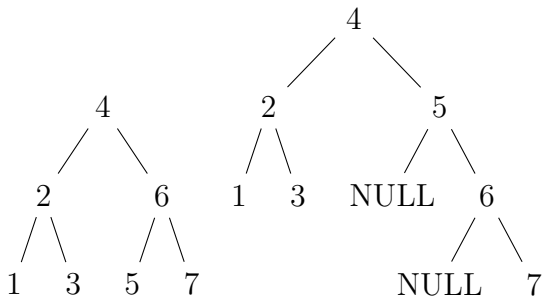
對於初始序列的建構，轉化成按照名次順序插入節點就行了。刪除節點的操作，模板就有直接支援。超越的部分比較麻煩，對於每一筆超越的操作，可以想像成先將前一名的節點刪除，再插入到自己的後面，因為 linked list 對於 id 的插入、刪除操作都是 $O(1)$ ，所以總複雜度 $O(N+M)$。

```
int main(){
    int n, m, operation, id;
    cin >> n >> m;
    link_list List(n);
```

```
5
6       // 初始化
7       for(int i=0;i<n;i++)
8           List.Insert(i,i+1);
9
10      while(m--){
11          cin >> operation;
12          if(operation == 0) {
13              // 刪除節點 id
14              cin >> id;
15              List.Delete(id);
16          } else {
17              // 超越前一名
18              cin >> id;
19              int p = List.prev(id);
20              if(p!=0){
21                  int pp = List.prev(p);
22                  List.Delete(p);
23                  List.Delete(id);
24                  List.Insert(pp,id);
25                  List.Insert(id,p);
26              }
27          }
28      }
29      List.travel();
30  }
```

3.7 二元搜尋樹 (Binary Search Tree)

這是一個很重要的觀念，要好好學！**這裏會用到一些基本圖論的用語，如果還是不熟的話可以去複習一下**。對於一個**二元樹**（也就是每一個節點都有少於或等於兩個子節點的樹）賦值，使得對於所有節點，**其右節點的子樹中的數字都比這個節點大，而左節點的子樹中的數字都比這個節點小**（稱為二元搜尋樹性質）。什麼意思？且看圖中分解。以下是對於集合 $\{1,2,3,4,5,6,7\}$ 所建的兩棵二元搜尋樹：



對於一棵二元尋樹上的所有節點，其右子樹中的數字都比這個節點大；左子樹中的數字都比這個節點小

為什麼這個重要呢？來看下一個段落！

二元搜尋樹所能做的有趣的好玩的事

這是一個二元樹的節點，每個節點有三個指標分別指向自己的父節點以及左右子節點。每個指向node型別的指標都可以視為一棵二元樹，因此在這個節點內部實際上是存了三棵樹。

```
struct node {
    node *parent,*lson,*rson;
    int val;
    node(int data):
        parent(NULL), lson(NULL), rson(NULL), val(data){}
};
```

一個指標需要占 8 個 bytes，所以用指標實作的東西都要花蠻多的記憶體，在時間上常數也會比較大。

插入一個元素

假設已經有一個二元搜尋樹了，那要怎麼插入一個新元素？假設這個元素的值為 x，且目前走到的節點的值為 y（一開始當然是從根節點開始走），那該放這個節點的哪裡呢？根據二元搜尋樹的定義，如果 $x < y$，則往左走；反之，則往右走。那如果沒得走了（也就是要走的節點為 NULL），那就將本來應該要走的節點設為 x，就好了。

```
void Insert(node *&x, int val) {
    if(!x) {
        x = new node(val);
        return x->parent = NULL, void();
    }
    if(x->val > val){
        if(x->lson) Insert(x->lson, val);
        else x->lson =
            new node(val), x->lson->parent = x;
    }
    if(x->val < val){
        if(x->rson) Insert(x->rson, val);
        else x->rson =
            new node(val), x->rson->parent = x;
    }
}
```

如何建立二元搜尋樹

為什麼要先講插入再講感覺比較基本的建立？因為呢，對於一個序列 $[a_1, a_2, \cdots a_n]$，將 a_1 設定為根，再對 a_2、a_3、a_n 等，依序插入，插入完了就建立完成！

拜訪這棵樹

中序尋訪是二元樹的拜訪方式之一，利用中序尋訪一個二元搜尋樹可以將資料結構內部的值按照大小排序輸出。因此二元搜尋樹是一個有序 (ordered) 的資料結構。

```
void travel(node *x){
    if(!x) return;
    travel(x->lson);
    printf("%d ", x->val);
    travel(x->rson);
}
```

另外還有前序與後序尋訪方式，在其他領域上用得到。

在樹上查詢

在這個二元搜尋樹上，根據值找到節點（或者是查看是否存在，找在樹上最靠近一個值的點之類的），都可以在樹上跑來跑去來搜尋。假設要搜尋的值是 x，然後目前節點的值是 y，那也可以重複以上的動作：$x < y$ 則往左走，反之則往右走，結束條件就是沒得走或找到了。

```
node *Find(node *x, int val) {
    if(!x) return NULL;
    if(x->val == val) return x;
    else if(x->val > val) return Find(x->lson, val);
    else return Find(x->rson, val);
}
```

對一個節點 Say Goodbye

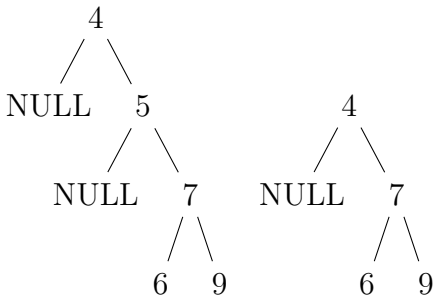
如果想要刪除一個節點，那得分三個 case：依照是不是葉節點（也就是有沒有子節點），可以看出需不需要找替換來遞補原本的節點的位置。如果是葉節點就直接刪掉即可，否則需要找一個替身。那作法：

```
bool Delete(node *&root, node *x) {
```

1. 如果是葉節點，則直接刪掉。

```
if(!x) return false;
if(!x->lson && !x->rson){
    if(x->parent) (x->parent->val > x->val)?
        x->parent->lson = NULL:
        x->parent->rson = NULL;
    delete x;
}
```

2. 如果其中一個子節點為空，則可以直接拉上來



假設要刪除 5 發現他只有一個子樹，那就可以直接把 4 連到 7 就好了（從左圖變成右圖）。

```
else if(!x->lson){
    if(x->parent){
        (x->parent->val > x->val)?
        x->parent->lson = x->rson:
        x->parent->rson = x->rson;
    } else root = x->rson;
    x->rson->parent = x->parent;
    delete x;
}
else if(!x->rson){
    if(x->parent){
        (x->parent->val > x->val)?
        x->parent->lson = x->lson:
        x->parent->rson = x->lson;
    } else root = x->lson;
    x->lson->parent = x->parent;
    delete x;
}
```

3. 如果不是，則**需要找一個節點當替身**。

引理 3.7.1: 替身的條件

找到的替身要符合：如果將要刪除的節點設為替身的值，並且將替身刪掉之後，還是會符合二分搜尋樹性質。

看左邊子樹，然後開始往右找到底。例如：如果選了左子樹，那走了之後，開始一直往右子樹走，直到不能再走了，那就將原本的節點的值換成那個點，然後遞迴刪除那個替身。

```
else{
    node *exchange = x->lson;
    while(exchange->rson) exchange = exchange->rson;
    x->val = exchange->val; // copy the data
    Delete(root, exchange);
}
```

最後再加上一行就可以了。

```
    return true;
}
```

刪除一個值

當我們要在二元搜尋樹中刪除一個值時，我們可以先找到它，再把節點刪除。

```
bool Delete_Val(node *&root, int val){
    return Delete(root, Find(root, val));
}
```

體驗二元搜尋樹的美好

整個二元搜尋樹的程式碼在此：https://pastebin.com/ED77CYHV

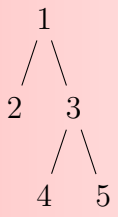
習題 3.7.1: 二元搜尋樹 (TRVBST) (TIOJ 1609)

一個序列依照插入的順序可以排成許多不同的二元搜尋樹，而給你 N 個不同的整數 a_i，依序為插入的順序，請問所構成的二元搜尋樹的中序遍歷為何？($N \leq 10^6$，$-2^{30} \leq a_i \leq 2^{30}$)

二元樹的表示方法：想要輸出一個二元樹，有三種比較常用的方法：

1. 前序：先輸出自己，再照著前序輸出左子樹，再照著前序輸出右子樹
2. 中序：先照著中序輸出左子樹，再輸出自己，再照著中序輸出右子樹
3. 後序：先照著後序輸出左子樹，再照著後序輸出右子樹，再輸出自己

假設用這個樹來當例子：



那前中後序分別為：1 2 3 4 5、2 1 4 3 5、2 4 5 3 1。

做事的代價

現在我們要分析一下以上介紹的運算的複雜度：可以知道，如果這棵樹的高度為 h（最深的節點深度），則最壞的情況就是在插入或搜尋的時候，每次都遇到最下面的節點，則複雜度 $O(h)$。那 h 大概會是多少呢？可以觀察：第一層最多可

以有 1 個節點，第二層最多可以有 2 個節點，第三層最多可以有 4 個節點..... 第 k 層最多可以有 2^{k-1} 個節點，總共可以有 $\sum_{k=1}^{h} 2^{k-1} = 2^h - 1$ 個節點，所以

$$n \approx 2^h \implies h \approx \log(n)$$

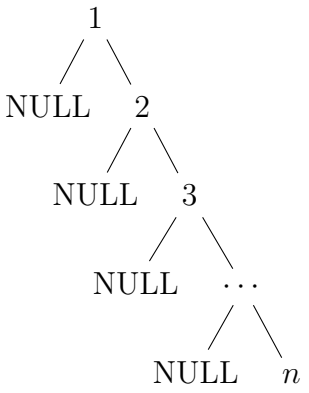
所以如果是一個夠平衡（待會會看不平衡）的二元搜尋樹，則搜尋和插入的複雜度都是 $O(\log n)$。

定理 3.7.1: 搜尋樹的複雜度

給定一個 n 個節點的隨機生成的二元搜尋樹，其搜尋和插入的平均複雜度為 $O(\log n)$

整人搜尋樹

現在我們要對這個序列建立搜尋樹：$[1, 2, 3, 4 \cdots n]$，那結果將會是：



這種情況稱為**退化**。這樣子呢，$h = n$ 而搜尋和插入都變很可悲的 $O(n)$ 了。以後會學到如何避免這個狀況，現在遇到就祈禱題目是隨機產生不會搞你就好了！

平時要多做好事，才不會比賽的時候被測資雷 (X
測資一定會想辦法卡掉這個，所以才需要平衡二元樹 (O

3.8 集合 (Set)

這是 STL 提供的 Binary Search Tree，內部實作是噁心的紅黑樹（Red-Black Tree），是一個實作複雜（所以直接用）但是可以自動平衡避免退化的資料結構。STL 已經幫你實作出來了，用就好了！set 裡面的東西不能有重複，插入、查詢、刪除的複雜度都是 $O(\log n)$。看以下程式片段：

```
1 set<int> se; // 宣告一個元素型別為 int 的集合
2 se.insert(1); // 插入元素 1
3 se.insert(2);
4 se.insert(5);
5 cout << se.count(1) << endl; // 查找元素，true
```

```
cout << se.count(3) << endl; // false
se.erase(1); // 刪除元素，若集合內不含此元素則回傳 false
cout << *se.find(3) << endl;
// find() 回傳 iterator，若找不到則回傳 se.end()
cout << *se.lower_bound(3) << endl;
cout << *se.upper_bound(3) << endl;
// 因為 set 有序，所以也支援二分搜操作
```

multiset

STL 也支援可以重複元素的集合，用法與set差不多：

```
multiset<int> se; // 宣告一個元素型別為 int 的 multiset
se.insert(1); // 插入元素 1
se.insert(1); // 可以重複插入
se.insert(5);
cout << se.count(1) << endl; // 計算元素個數，2
cout << se.count(3) << endl; // 0
se.erase(1);
// 一次刪除所有元素 1，若集合內不含此元素則回傳 false
se.erase(find(5)) // 一次刪除一個元素 5
```

當然，set與multiset也支援size()與empty()，在此不做贅述了。

set的遍歷

set是一棵二元搜尋樹，當然可以進行遍歷的操作。STL 的二元搜尋樹沒辦法存取一個子樹的左右節點，不過 STL 提供了迭代器 (iterator)，可以用不同的方式進行遍歷。

```
for(set<int>::iterator it = se.begin(); it != se.end(); it++)
    cout << *it << ' ';
```

或者用 C++11 後提供的 auto 關鍵字進行尋訪

```
for(auto &val: se)
    cout << val << ' ';
```

習題

不要被梗！

習題 3.8.1: 今晚打老虎 (ZJ a091)

奕辰找到了一個神奇的機器，可以做三件事：

1. INSERT x，也就是把一個數字輸入進去這個機器
2. QUERY MIN，也就是查詢目前的最小值
3. QUERY MAX，同上，查詢目前的最大值

而且，**只要一個數字被查詢過了，機器就會把他吐出來，代表不在機器中了！**同一時間內不會超過 10^6 個數字，且數字可以被int存下。

習題 3.8.2: 好多燈泡 (TIOJ 1513)

很會找東西的奕辰找到了一張紙條：上面寫了 N 個數字！他在一個很多燈泡和開關的房間裡，每一個燈泡和開關都有編號，被相同編號的開關所控制。一開始每一盞燈都是關的。現在，紙張上寫了數字代表操作順序，第 i 個數字代表要去操作第 i 盞燈（關的變成開的，開的變成關的），已知最後操作了之後只有一盞燈是亮著的，並且直視那盞燈可以獲得神奇魔法！但是奕辰最近忙於看片，沒辦法以一操作，所以請你幫他寫程式，讀入 N 和 N 個指令，輸出哪一個燈泡最後亮著。($N \leq 10^5$)

註：也可以想想 $O(n)$ 解喔 $><$

3.9 映射 (Map)

基本上跟上面的一樣，可是不是存數字而是存兩個值，key和value（也就是存一個pair，第一個存key ，第二個存value。紅黑樹依照key排列，所以是依照key 搜尋。用處類似一個可以存任何東西，複雜度稍微高一點（同上）的陣列。跟set一樣，map的key值不能有重複。

```cpp
map<string,int> mapp;
// 宣告一個由 string 映射到 int 的 map
mapp.insert({"apple", 1});
// 插入一個 key 值為 "apple" 的節點
mapp["book"] = 2; // 也可以這樣插入
mapp["apple"] = 3; // 改值
cout << mapp.count("apple") << endl; // true
cout << mapp.count("my girlfriend") << endl; // false

for(map<string,int>::iterator it = mapp.begin(); it != mapp.end(); it++)
    cout << it->first << ' ' << it->second << endl;

for(auto &val: mapp)
    cout << val.first << ' ' << val.second << endl;

// apple 3
```

```
17  // book 2
```

如果要看一個map裡面的東西，但是不確定它存不存在（像是以下程式）
`if(mapp["orange"] == 2) //...`
那mapp就會先開一個 key 為orange的點，佔空間！所以要先判斷存不存在：
`if(mapp.count("orange") && mapp["orange"] == 2) //...`
才不會浪費許多不必要的記憶體。

multimap

同樣的 STL 也有multimap這種東西，目前我還想不到實際應用，想知道詳細的可以看 cpp reference。

跟著蕭電這樣做

：知道 multimap 的應用的人麻煩私訊我，筆者感激不盡 OwO

習題

map 是個好用的東西喔！

習題 3.9.1: Hardwood Species（UVa 10226）

恭喜成為森林管理者！現在你的第一個工作就是計算森林中樹的比例。有 N 棵樹，給你他們的英文名稱（由大小寫英文字母表示），請輸出他們所佔的比例為何。（比例定義為：$\frac{\text{樹出現的次數}}{N}$），（至多 10^4 種樹、10^6 棵樹，請輸出至小數點後四位。）

習題 3.9.2: 連通塊數量 (OJDL)

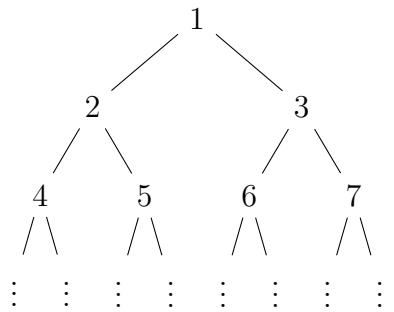
給平面上 N 個點 $(N \leq 10^6)$ 的 x, y 坐標 $(x, y \leq 10^9)$。若兩個點的 x 坐標或 y 坐標只差 1 可以視為同一個連通塊，求連通塊數量。

3.10 堆積 (Heap)

堆積的定義比較不能望文生義：它也是一個二元樹，可是不滿足以上的二元搜尋樹性質，而是符合堆積性質：對於一個 Min-Heap，一個節點的值一定比其子樹的每一個節點的值都小，對於一個 Max-Heap，則是都大。還有一個重要的性質就是：通常實作的時候會把這個樹當成一個**完全二元樹**（也就是除了最後一階可能不填滿其他都滿）的樹，這個好處就是可以用陣列存這個樹。

定理 3.10.1: 用陣列存二元樹

給定一個陣列，可以用陣列來存二元樹，編號就是 BFS 順序（或者白話一點就是橫著編號）：



經觀察可以發現：編號為 i 的節點左節點為 $2i$，右節點為 $2i+1$，而其父節點為 $\lfloor \frac{i}{2} \rfloor$。如果是完全二元樹，就可以發現只需要 $O(n)$ 的空間來存；但是如果退化成鍊可就糟糕了：需要 $O(2^n)$ 的空間存，所以要取捨。這個概念不只在這個 Heap 中會用到，所以得獨立出來講！

顯然，對於一個數字的集合，也會有許多的 Min(Max) Heap 可能。以下所講的 Heap 皆指 Min-Heap，至於 Max-Heap 的實作就只是把不等號反過來就好了。要支援的操作比較多，以下列出來之後我們慢慢揭曉。最後的節點指的是最後一層最右邊的節點。

1. `int getMin()`，代表回傳堆疊根節點的值（也就是最小值），複雜度 $O(1)$。

```
int getMin(vector<int> &heap) {
    if(heap.empty())
        printf("Are you kidding?\n");
    else return heap[0];
}
```

2. `void insert(int x)` 要加入一個值為 x 的數字進去 Heap 裡面。作法就是先將 x 先丟進去最下面，然後一直往上，只要發現 x 現在的節點比其父節點的值小，就交換他們兩個，並繼續看。

```
void Insert(vector<int> &heap, int x) {
    heap.push_back(x);
    int loc = heap.size()-1;
    while(loc){
        if(heap[loc] < heap[loc>>1])
            swap(heap[loc], heap[loc>>1]);
        loc >>= 1;
    }
}
```

3. `void deleteRoot(int x)` 刪除根節點。將根換成最後的節點（最後的節點也刪掉），然後開始遞迴往下：如果目前的值比左右子樹的最小值大，則與左右子樹的最小值交換，並且往那個子樹遞迴下去。

```cpp
void deleteRoot(vector<int> &heap) {
    swap(heap[0], heap[heap.size()-1]);
    heap.pop_back();
    int pos = 0;
    while(pos < heap.size()) {
        int l, r, Min;
        l = (2*pos >= heap.size()) ?
            INF : heap[2*pos];
        r = (2*pos+1 >= heap.size()) ?
            INF : heap[2*pos+1];
        if(l == INF && r == INF) break;
        Min = (l < r) ? 2*pos : 2*pos+1;
        if(heap[Min] < heap[pos])
            swap(heap[pos], heap[Min]), pos = Min;
        else break;
    }
}
```

不難發現，insert() 和deleteRoot() 的複雜度都是 $O(\log n)$，因為最差就是一直遞迴到底，而因為是完全二元樹，就有 $O(\log n)$ 層。

在轉角與堆積邂逅

這是 code: https://pastebin.com/zkaMe93Z

STL 內的 Heap

當然，超棒的 STL 有提供 Heap 可以用，稱為priority_queue，但是不一定是依照以上的實作，可能有出入。如果想要看另外一種複雜的 Heap，可以去查 Fibonacci Heap。

宣告

我們可以用下列方法宣告一個priority_queue

```cpp
priority_queue<int> pq; // 宣告一個整數的 max-heap
```

若要自行定義比較函數 (也就是定義" 小於")，可以下列方式進行：

```cpp
priority_queue<int, vector<int>, greater<int> > min_pq;
// 把"小於"定義成"大於"，也就是 min-heap
priority_queue<int, vector<int>, comp> custom_pq;
// 自訂義小於比較
```

priority_queue的比較函數稍微複雜一點，要用 struct 的 operator() 來實作，以下是實作方法：

```cpp
struct comp{
    bool operator()(int a,int b){
```

```
3           return a%10 < b%10;
4       }
5   };
```

三個操作

STL 內建的`priority_queue`一樣支援了 heap 該有的操作：
`push()` 可以將資料丟入堆積的內部；
`pop()` 將最大元素刪除；
`top()` 查詢堆積內的最大元素；

```
1   pq.push(3);
2   pq.push(2);
3   cout << pq.top() << endl; // 3
4   pq.pop();
5   cout << pq.top() << endl; // 2
```

當然也支援查詢資料結構的大小：
`size` 函數查詢目前還位於堆疊中的資料數；
`empty` 函數回傳堆疊是否為空。

千萬不要對一個空的`priority_queue`呼叫 top() 函數，不然你會吃個 RE

習題

不，第一題不是 DP 題！

習題 3.10.1: Add All (UVa 10954、ZJ d221)

給你 N 個數字 $a_1, a_2, \cdots, a_N$，請把這些數字加起來。很簡單？但是但是，要加兩個數字 x 和 y 有一個條件：必須付出 $x + y$ 的代價。請問要把這 N 個數字加起來的最小代價為何？($2 \leq N \leq 5000$)

(例：如果 $N = 3$ 且數列為 $[1, 2, 3]$，則最好的作法就是先加 1 和 2，花 3，再把兩個 3 加起來，花費 6，總共花費 $3 + 6 = 9$。

習題 3.10.2: 排隊買飲料 (TIOJ 1999)

有一天，你經過了一家飲料店，發現有 N 個人排隊要買飲料。不過，這家飲料店人手充足，一共有 M 個店員可以服務這些排隊的人潮。為了公平起見，雖然店員很多，但是排隊只排成一列，避免在排很多列的情況下，每列前進的速度會不一樣。
另外，為了服務品質起見，這家店的店員必須遵守兩個規則：必須按照客人排隊順序服務客人，不能先服務排在後面的顧客，而且，如果某個店員服務了某個客人，則該客人點的所有飲料都要由該店員製作，製作完成之後才能服務下一位客人。
你做了一下市場調查，詢問每位排隊的客人要買幾杯飲料。假如所有的店員製作飲料的速度都是每分鐘 1 杯，在遵守這些規則的前提下，請問服務完這些客人至少需要多久？
($N \leq 10^6$、$M \leq 10^4$，輸入完了之後會有 N 個數字，為每個排隊顧客要買多少飲料（最多 1000 杯）

3.11 位元集 (Bitset)

很多時候，會想要存布林陣列，但是你可知道：一個布林值居然佔了八個位元?! 如果題目出機車一點壓記憶體可能就要用到這一招：bitset。就是一個加強版布林陣列，除了終於讓每一個布林值佔一個位元（也就是空間佔八分之一）之外還加了許多的功能，如反轉，變成字串或數字（unsigned long long int之類的）的功能！請看以下程式片段。

建構元

```
bitset<10> bs; // 宣告 10-bit 的位元集，初始化為 0
bitset<10> bsA(71); // 將 71 表示為二進制存入
bitset<10> bsB("1010111"); // 也可以這樣初始化
```

運算子

```
cout << bsA << endl; // 輸出
cout << (bsA & bsB) << endl; // AND
cout << (bsA | bsB) << endl; // OR
cout << (bsA ^ bsB) << endl; // XOR
cout << ~bsA << endl; // NOT
cout << bsA[0] << endl; // 下標運算子
cout << (bsA<<1) << endl; // 左移右移
```

計數技術

```
cout << bsA.size() << endl; // 一開始宣告的bit數
cout << bsA.count() << endl; // 輸出總共有幾個 1
```

型別轉換

```
string s = bsA.to_string();
unsigned int x = bsA.to_ulong();
```

3.12 雜湊表 (Hash table)

先講一下何謂雜湊（Hash）：這個是 Hash Brown，中文叫做薯餅：

可是不是我們要的主題。我們要的是**雜湊（Hash）**！先來看維基百科的定義：

> 「雜湊函式（英語：Hash function）又稱雜湊演算法，是一種從任何一種資料中建立小的數字「指紋」的方法。雜湊函式把訊息或資料壓縮成摘要，使得資料量變小，將資料的格式固定下來。該函式將資料打亂混合，重新建立一個叫做雜湊值（hash values，hash codes，hash sums，或 hashes）的指紋。雜湊值通常用一個短的隨機字母和數字組成的字串來代表。」

簡單來說，就是讓一個很難表示的東西透過某個函數變成一個比較好表示的東西！假設想要對生日來 Hash，那一個生日可能是 2002/11/05，不好處理，那我可能會壓縮成一個字串"2002/11/05"或表示成一個 b 進位的數字 $\pmod p$，b，p 為相異質數之類的，就是為了方便存，而這個函數就叫做 Hash Function。顯然，如果 Hash 出來的東西不一樣，則兩個東西一定不一樣；但是如果 Hash 出來的東西一樣，不代表是一樣的。雖然如此，但是如果好好選 Hash Function，就可以避免碰撞（Collision）了。現在來分析這個可能性：

定理 3.12.1: 雜湊碰撞

假設一個 Hash Function 有 N 個可能值，然後有 M 個東西要被雜湊。每一個都相異的機率為 $\binom{N}{M} \cdot \frac{1}{N^M}$，故有出現碰撞的機率為 $1 - \binom{N}{M} \cdot \frac{1}{N^M}$。這個會隨著 M 而增加，所以在用 Hash 前得先稍微估一下。

這個的實作方法通常是用一個陣列來做，這樣均攤複雜度可以到 $O(1)$ 查詢，$O(1)$ 修改。維持一個長度為 n 的陣列 A，每次要加入一個東西 k 的時候，就把 k 存在 $A[\text{hash}(k) \pmod n]$ 的地方即可。通常也會搭配 Linked List，將 A 的每一個元素視為一個 Linked List 的頭，插入的時候如果遇到已經有東西了就會將 k 放在 $A[\text{hash}(k) \pmod n]$ 的最後頭。刪除也差不多，刪除 k 就到 $A[\text{hash}(k) \pmod n]$ 去看，找到了之後刪除。這樣的好處是平均上來說複雜度非常好，但是如果遇到雷測資就會爆複雜度。

如何選擇雜湊函數

一個好的雜湊函數需要那些特質呢？當然，需要有相當的平均性：不可以輸入一堆東西都出來是同樣的！第二個重要的特質是簡單計算：如果一個雜湊函數需要 $O(n^2)$ 的時間計算，反而會將演算法變慢。通常，喜歡把一個字串表示成一個 p 進位的數字，$a = 1, b = 2 \cdots$ 類推，例如 $\text{hash}(\text{"}abc\text{"}) = 26^2 \times 1 + 26^1 \times 2 + 26^0 \times 3 = 731$，然後為了防止值太大會模一個大質數，如 10003457 之類的。好一點（或是機車一點）的測資會卡常見的雜湊函數哦！所以可以自己想一想屬於自己的雜湊函數。

unordered_set

STL 的 hash table 叫做 unordered_set，它與 set 一樣，都支援了插入、查詢、刪除的功能，唯一有差別的是因為它的實作原理是 hash ，對於每次操作的複雜度都會是 $O(1)$，但是這樣一來就會沒有二元搜尋樹的良好性質，所以不能按照順序遍歷輸出。

```
unordered_set<int> use; // 宣告一個元素型別為 int 的集合
use.reserve(N); //reserve N個記憶體可以加速程式
use.insert(1); // 插入元素 1
use.insert(2);
use.insert(5);
cout << use.count(1) << endl; // 查找元素，true
cout << use.count(3) << endl; // false
use.erase(1); // 刪除元素，若雜湊表內不含此元素則回傳 false
// 不支援 upper_bound, lower_bound, find 等二分查找功能
for(auto &val: use)
    cout << val << ' ';
// 可以遍歷，但不會照順序
```

unordered_map

unordered_map 的用法與一般的 map 也差不多，實作原理也是個雜湊表。大部分的成員函數以及運算子都與 map 一樣，複雜度 $O(1)$。

```
unordered_map<string,int> umapp;
// 宣告一個由 string 雜湊到 int 的 unordered_map
umapp.insert({"apple", 1});
umapp["book"] = 2;
umapp["apple"] = 3;
```

```
6  cout << umapp.count("apple") << endl; // true
7  cout << umapp.count("my girlfriend") << endl; // false
8
9  for(auto &val: umapp)
10     cout << val.first << ' ' << val.second << endl;
11 // 支援遍歷，一樣是無序的
```

也可以自己寫雜湊！這是unordered_map的定義範例（假設是把string透過雜湊函數變成unsigned long）：

```
unordered_map<
std::string, //key
unsigned long, //value
std::function<unsigned long(std::string)>,
//hash: string 變成 unsigned long
std::function<bool(std::string, std::string)> //key 的比較函數
> mymap(n, hashing_func, key_equal_fn<std::string>);
//mymap 初始大小，雜湊函數，key 的比較函數
```

雜湊表雖然 $O(1)$，但是也有比平衡二元樹慢的時候。當你的鍵值為非常龐大的資料結構時 (例如：超長 string)，在測資隨機生成的條件下，二元搜尋樹的比較函數通常只要比對前幾個字元就能快速查找；雜湊則要花字串長度的線性時間才有辦法計算出一個雜湊值。所以要看輸入資料的類型進行取捨，不要看到無關順序的東西就用unordered_map砸。

實戰演練

這題是 TOI 入營考的題目，時限卡得非常緊，用了常數比較大的 map 一定會 TLE，並且拿到跟暴力 $O(n^4)$ 一樣的分數。對於本題來說，必須考慮使用常數較小的方法 (如排序)，或者直接使用 $O(n^2)$ 的演算法。

習題 3.12.1: 四點共線 (TOI 2019 初選 pA)

給你 n 個點的 x, y 坐標 ($n \leq 3000$，$x_i, y_i \leq 10^4$)，每個點的編號為它們的輸入順序。求編號字典序最小的共線四點，若不存在四點共線則輸出 -1。

並查集 Disjoint Sets 4

並查集 (Disjoint Sets) 是一個用來處理集合的資料結構，他只支援兩種操作：

1. 將兩個集合合併 (Union)
2. 查詢一個元素所在的集合 (Find)

這是一個基礎的資料結構，一定要學會，後面的演算法常常會用到這個概念!

實作

一般我們維護一個 DSU 的方式都是以一棵樹來記錄一個集合，而利用 root 來代表該集合，事實上，要維護這些樹只要用一個 array 就行了，array 第 i 項的值就是 i 的 parent，而 root 的 parent 則是自己。前面說到一個正常的 DSU 應該要支援兩種功能，現在就來講講這兩種功能的實作概念吧～

合併 (Union)

合併兩個集合的方式其實異常簡單，就是分別找到那兩個集合的 root，將其中一個 root 指向另外一個，便大功告成了。

查詢 (Find)

要查詢一個元素所在集合的方式也十分直觀：不斷往自己的 parent 找，直到找到一個元素的 parent 指向自己，那麼他就是 root 了。而在往上尋找時還有一個小技巧，就是在過程中記錄經過的節點，之後一併將這些點的 parent 改為 root，便可以節省下次查詢這些點的時間。

習題 4.0.1: DSU 練習

請實作一個DSU，需要支援兩種操作：

1. `Union`(a, b)，將 a 所在的集合與 b 所在的集合合併
2. `Find`(a)，回傳 a 所在的集合

```cpp
int v[MAXN];

void Find(int a) {
    if (v[a] == a) return a;
    v[a] = Find(v[a]);
    return v[a];
}

void Union(int a, int b) {
    v[Find(a)] = Find(b);
}
```

維護 size

倘若想同時記錄每個集合的大小，其實也沒那麼困難，只要多用一個 size 變數紀錄每個集合的大小，合併時將兩個集合的大小相加，就能成功維護集合大小了。

一點小優化

對於每個集合多給一個變數 rank，記錄每個集合中 tree 的最大深度，合併時選擇 rank 較小的合併到 rank 較大的集合中，而倘若兩個集合的 rank 相同，則要在兩個集合合併後把 rank 值 $+1$，這樣可以再降低之後 Find 的複雜度，使 DSU 操作的時間複雜度答案近 $O(\alpha N)$ 的程度。
(可以想想為什麼兩集合 rank 相同時，合併後要 $+1$？

排序演算法 5

5.1 排序問題

將一堆資料排序，一直是電腦科學中一個十分重要且基本的問題；在日常生活中，也少不了需要排序的工作。**排序問題**是大部分演算法教科書第一道經典例題，它的作法有非常多種，接下來我將會探討各種作法的優缺利弊。

習題 5.1.1: sorting problem

給定一個正整數 n，接下來有 n 個整數，目標是將這 n 個整數由小到大按照順序輸出。

排序好的數列通常有許多好性質可以執行許多如二分搜等方便的操作，也有許多題目 (例如在一個序列中求眾數的問題) 是利用排序的衍生概念，能夠熟悉實作原理對競賽一定有極大幫助。

5.2 選擇排序 (selection sort)

樸素做法

回憶一下你怎麼排序東西的。不斷找到最小的，放到旁邊去，然後就排好了，對吧。詳細流程如下：

1. 從未排序的數列中找到最小的元素。
2. 將此元素取出並加入到已排序數列最後。
3. 重複以上動作直到未排序數列全部處理完成。

```
1 void selectionSort(int* arr, int len){
2     for(int i = 0, minIdx; i < len-1; i++){
3         minIdx = i;
4         for(int j = i+1; j < len; ++j)
5             if(arr[j] < arr[minIdx])
6                 minIdx = j;
7         if(minIdx != i)
```

```
8               swap(arr[minIdx], arr[i]);
9       }
10  }
```

效率？

我們知道找到最小值要花 $O(n)$ 的時間，而每找一次僅能將未排序陣列減少一個元素，需要執行 $n-1$ 次才能排序完畢。因此整個選擇排序需要花 $O(n^2)$ 的時間才能完成。

5.3 插入排序 (insertion sort)

也有人是這樣排序的 (本人就是)，不斷找到下一個數在已排序陣列中該去的位置，並將其插入，當最後一個元素也排到該去的位置即排序完畢。

1. 從未排序數列取出一元素。
2. 由後往前和已排序數列元素比較，直到遇到不大於自己的元素並插入此元素之後；若都沒有則插入在最前面。
3. 重複以上動作直到未排序數列全部處理完成。

```
1   void insertionSort(int *arr, int len){
2       for(int i = 1; i < len; i++){
3           int key = arr[i];
4           int j = i-1;
5           while(key < arr[j] && j >= 0){
6               arr[j+1] = arr[j];
7               j--;
8           }
9           arr[j+1] = key;
10      }
11  }
```

它要跑多久

相似地，插入操作需要 $O(n)$ 時間，需要插入 $n-1$ 次，因此插入排序一樣是十分慢的 $O(n^2)$。

5.4 氣泡排序 (bubble sort) 和 3t 的國中回憶

再來是比較不直觀但實作非常方便的氣泡排序，它讓數字像氣泡一樣慢慢浮上來，第 k 次上浮都會使第 k 小值跑到前面去，最終完成排序的目的。氣泡排序有許多實作方法，一般來說都是找到所有相鄰的元素比較一次，不斷交換上來，上浮 $n-1$ 次即完成排序。

```
void bubbleSort(int* arr, int len){
    for(int i = 0; i < len-1; i++)
        for(int j = 0; j < len-1; j++)
            if(arr[j] > arr[j+1])
                swap(arr[j], arr[j+1]);
}
```

有的人 (迷之音：明明只有你在用 $==$) 是每次都跟未排序陣列的第一個元素比較，這個比較像選擇排序，一樣可以獲得相同的效果。

```
void bubbleSort(int* arr, int len){
    for(int i = 0; i < len; i++)
        for(int j = i+1; j < len; j++)
            if(arr[i] > arr[j])
                swap(arr[i], arr[j]);
}
```

我與 TLE 的故事

那這個效率多高呢？很抱歉，一樣是 $O(n^2)$。看到 code 裡面的兩層迴圈就很明顯了，內外層各要跑 $O(n)$ 次。

(NPSC 2016 國中組初賽 pC)
千萬不要拿 $O(n^2)$ 的排序演算法做題，會 TLE 不瞑目

5.5 合併排序 (merge sort)

WTF! TLE，怎麼辦？
其實在 $O(n \log n)$ 時間內有效率的對一個序列排序是可能的，但是想法就會不夠顯然，實作也複雜許多，但是在競賽題的應用層面上相對的比較廣。接下來要介紹的就是其中一種–合併排序。

divide & conquer

我們先思考一個比較簡單的問題：

習題 5.5.1: 合併遞增序列

給兩個長度分別為 n, m 的**遞增**正整數序列 ($m, n \leq 5 \times 10^6$，值域 $\leq 10^9$)，將兩序列出現過的數字合併成一個序列，並**遞增**排序輸出。

對於這個問題，我們可以用爬行法的方式解決。先假設這兩個陣列叫做 A 與 B；我們用兩個變數 ptrA 與 ptrB 分別指著兩個陣列的開頭。接著選擇 ptrA 與 ptrB 兩變數指向的值較小的那一個 (若一方已指向最尾端，則直接選擇另外一方)，將其值輸出，然後再把對應的 ptr 右移一格，直到 ptrA 與 ptrB 皆指向陣列的最尾端。

因為輸入保證兩數列已經做好遞增排序，因此 A 陣列中未被輸出的數字必定比 ptrA 指向的值還要大，B 陣列亦然。因為有這個性質，我們可以確定的是 ptrA 與 ptrB 指向的值較小的那個就是所有未輸出數字的最小值，所以以上的演算法可以保證正確性。

```
vector<int> Merge(vector<int> &A, vector<int> &B){
    vector<int> result;
    int ptrA = 0, ptrB = 0;
    while(ptrA != A.size() || ptrB != B.size()){
        if(ptrA == A.size() || (ptrB != B.size() && B[ptrB] < A[ptrA]))
            result.push_back(B[ptrB++]);
        else result.push_back(A[ptrA++]);
    }
    return result;
}
```

那我們要怎麼用這個來解決排序問題呢？首先你會發現假設你的序列能夠分成兩個遞增的區間，套用上面的合併演算法就做完了 OAO，但是一般的序列切成兩半後幾乎不可能有這種性質，那該怎麼辦呢？**排序 !!!**

分治法 (divide and conquer) 是一種將原問題分割成相同但輸入量變小的問題，並且再以一定合併程序求解原問題的方法。合併排序就是分治的一道經典問題！我們將左右兩邊的序列都先用合併排序本身排好，然後再合併！

```
void mergeSort(vector<int> &arr, int len){
    if(len <= 1) return;
    int mid = len/2;
    vector<int> arrL, arrR;
    for(int i = 0; i < len; i++)
        (i < mid)? arrL.push_back(arr[i]):
                   arrR.push_back(arr[i]);
    mergeSort(arrL, mid);
    mergeSort(arrR, len-mid);
    vector<int> result = Merge(arrL, arrR);
    arr.clear();
    for(auto i: result) arr.push_back(i);
}
```

5.6 堆積排序 (heap sort)

另外一種有效率的排序方法是利用堆積資料結構。仔細觀察插入排序、氣泡排序以及選擇排序，你會發現他們在做的事就是不斷找到未排序的最小值。heap 資料結構恰好提供了高效找出最小值的方法，大大提升執行效率。

將所有數字放進 heap，再一個一個拔出來，自然是排序完畢的樣子。

```
void heapSort(int *arr, int len){
    priority_queue<int, vector<int>, greater<int> > pq;
    for(int i = 0; i < len; i++)
```

```
4           pq.push(arr[i]);
5       for(int i = 0; i < len; i++)
6           arr[i] = pq.top(), pq.pop();
7   }
```

heap 的插入與刪除只需要 $O(\log n)$ 的時間，因此堆積排序也能在 $O(n \log n)$ 時間內完成。

5.7 快速排序 (quick sort)

快速排序，顧名思義就是比較快速的排序 (廢話 X。因為它的常數小，所以它通常執行的比合併排序與堆積排序來的快。快速排序使用分治法（Divide and conquer）策略來把一個序列分為**數值較小**和**數值較大**的兩個子序列，然後遞迴地排序兩個子序列。

1. 挑選基準值：從數列中挑出一個元素，稱為「基準」（pivot），
2. 分割：重新排序數列，所有比基準值小的元素擺放在基準前面，所有比基準值大的元素擺在基準後面（與基準值相等的數可以到任何一邊）。在這個分割結束之後，對基準值的排序就已經完成，
3. 遞迴排序子序列：遞迴地將小於基準值元素的子序列和大於基準值元素的子序列排序。

遞迴到最底部的判斷條件是數列的大小是零或一，此時該數列顯然已經有序。

```
1  void quickSort(vector<int> &arr, int len){
2      if(len < 2) return;
3      int RND = rand()%len, pivot = arr[RND];
4      vector<int> Less, Greater;
5      for(int i = 0; i < len; i++){
6          if(i == RND) continue;
7          if(arr[i] < pivot)
8              Less.push_back(arr[i]);
9          else Greater.push_back(arr[i]);
10     }
11     quickSort(Less, Less.size());
12     quickSort(Greater, Greater.size());
13     arr.clear();
14     arr.insert(arr.end(),Less.begin(),Less.end());
15     arr.push_back(pivot);
16     arr.insert(arr.end(),Greater.begin(),Greater.end());
17 }
```

原地分割

上述的版本有個缺點，它一樣需要 $O(n)$ 的額外空間，在電腦中實際執行時也會影響運行速度。因此我們有個辦法可以直接在原序列做分割的動作，不僅降

低空間複雜度，也大幅提升演算法效率。
以下是快速排序的泛型 (template) 實作範例。

```
template<typename iter>
void quickSort(iter l, iter r){
    if(r-l < 2) return;
    iter head = l, pivot = r-1;
    swap(*(l+rand()%(r-l)), *pivot);
    for(iter it = l; it != pivot; it++){
        if(*it < *pivot){
            swap(*it, *head);
            head++;
        }
    }
    swap(*head, *pivot);
    quickSort(l, head);
    quickSort(head+1, r);
}
```

定理 5.7.1: 泛型函數

C++ 提供了泛型函數的撰寫功能，當你需要實作相同或相似的功能，但是引數型別可能不盡相同的時候，使用**泛型**是一個不錯的選擇。另外，C++ 也提供了**多型**的功能，可以對於不同型別或不同數量的引數實作不同的函數。

快速排序，快速嗎？

原地分割大大降低的程式執行的常數因子，但快速排序真的夠快嗎？想像一個最差的情況，每次的 pivot 都恰好選擇到最小的數值，這樣數值比 pivot 大的序列就只比原序列少一個數，因此最差需要 $O(n^2)$ 的時間複雜度。但我們可以期望這種最差情況幾乎不可能發生，因此我們可以用平均 $O(n \log n)$ 的複雜度來看待它。

5.8 std::sort – introsort

當然，我們 namespace std 的強大函數庫也少不了排序！一行搞定！
這裡的sort()的原理是 introsort，一開始與 quick sort 一樣，一旦出現遞迴過深的情況，就會改而採取 insertion sort 或者 heap sort，因此是經過極佳常數優化，也是最有效率的排序演算法。記得使用前要先 include <algorithm> 唷！

```
int arr[5] = {3, 2, 4, 1, 5};
vector<int> vec = {3, 2, 4, 1, 5};

sort(arr, arr+5);
// 傳入兩端點指標 (注意是左閉右開的)
sort(vec.begin(), vec.end());
```

```
// 排序 vector 要傳 iterator
sort(arr, arr+5, greater<int>());
// greater<T>() 可以由大到小排序
sort(arr, arr+5, cmp);
// 也可以自己寫比較函數
```

以下是自訂比較函數的寫法

```
bool cmp(int a, int b){
    return a%10 < b%10;
}
// 定義"小於"，return true 代表 a 要排前面
```

有了這個強大工具，排序再也不是難事了，也沒有在比賽時 TLE 的問題了，恭喜！

5.9 $O(n)$ sorting algorithm

還可以更快嗎？我們看看以下定理。

定理 5.9.1: 比較排序的複雜度限制

試想一棵樹，每一個非葉節點有兩個子節點，分別代表在排序中比較某兩個元素後會出現的兩種情況 (是否小於)。而每一個葉節點會是所有元素的一種排列，並且唯一符合它的所有祖先比較的情況。如此時間複雜度會是這棵樹的高度 h。
葉子的數量至多是 2^h，注意到所有葉節點必須包含元素的全排列 $(n!$ 種$)$，否則沒出現的排列就無法被排序。故 $2^h \geq n!, h \geq \log n! = O(n \log n)$

真的不行了嗎？計數排序！

計數排序 (Counting Sort) 演算法是不需進行比較的排序演算法，顧名思義，它會去數元素的數量來進行排序。這種排序法只需要線性時間和空間的複雜度就可以完成排序。雖然如此，計數排序法是並不算是常見的排序演算法，因為它只能用來排序已知數值範圍的序列。也就是說，用個陣列紀錄**每個數字出現了幾次**。如果數字 3 出現了 5 次，那麼cnt[3]=5。這樣一來只要讓i從 0 開始跑，然後把i輸出cnt[i]次，就排序完成了。

```
vector<int> CountingSort(vector<int> vec, int len){
    int cnt[MAXQ] = {};
    vector<int> ans;
    for(auto i: vec)
        cnt[i]++;
    for(int i = 0; i < MAXQ; i++)
        for(int j = 0; j < cnt[i]; j++)
            ans.push_back(i);
```

```
9       return ans;
10  }
```

你會發現第 6 行的for迴圈有個MAXQ，因此 counting sort 也不是真正的 $O(n)$ 排序演算法。它的時間複雜度是 $O(n+q)$，空間複雜度是 $O(q)$，因此當值域太大時這個演算法完全不可行，因此我們需要另外一種排序。

基數排序

這邊要先介紹什麼是 stable sort。

定義 5.9.1: stable sort

一個 stable 的排序必須符合: 若兩個元素比較的結果是相等，則排序後他們的先後順序必須和排序前的相同，這種情況通常發生在排序 struct 或以自定義比較函式排序時。前面提到的方法有好幾個已是 stable 的排序方法 (如 merge sort 等)，而其他的排序法都可以稍加修改成 stable sort。namespace std裡面也有stable_sort()可以用，用法與sort()一樣，前面多加個 stable 就行了。

有了 stable sort，那 radix sort 的概念就十分容易了。我們可以把所有數字依照個位數字做一次 stable sort，然後十位，然後百位...，因為每次都使用 stable sort，所以當最高位排序完之後，整個序列都排序完畢了。

```
1   int base = 10;
2   void RadixSort(vector<int> &vec){
3       vector<int> bucket[base];
4       int p = 1;
5       for(int i = 0; i < DIGIT; i++){
6           for(auto x:vec)
7               bucket[(x/p)%base].push_back(x);
8           vec.clear();
9           for(int b = 0; b < base; b++){
10              for(auto x: bucket[b])
11                  vec.push_back(x);
12              bucket[b].clear();
13          }
14          p *= base;
15      }
16  }
```

假設最大的數有 d 位數，那整個流程就要進行 d 次的排序，我們可以拿 counting sort 來排序，所以總複雜度只有 $O(nd)$，如果要排序整數，這是一個有效率又不怕值域太大的問題。

5.10 習題的啦

大家都會排序了嗎？這裡是一些排序的應用。

習題 5.10.1: Sort 五部曲 (TIOJ 1287,1328,1682 + ZJ a233,d190)

sort! sort! sort! sort! sort!

習題 5.10.2: 保羅的寶貝 (NPSC 2016 jun-pre pC)

保羅已經知道 N 個寶貝的重量與 M 個存放寶貝的櫃子離起點的距離，所謂疲勞程度就是每個寶貝重量乘上搬運該寶貝的距離的總和，他想知道他搬運所有寶貝疲勞程度最小是多少？($N, M \leq 10^6$，距離, 重量 $\leq 10^4$)

習題 5.10.3: 蛋糕內的信物 (TIOJ 1364)

給 N 個正整數的序列，求第 k 大 ($k, N \leq 10^6$)
提示：這題可以 $O(n)$ 解呦 ><

習題 5.10.4: 公車司機排班 (NTPC TOI-camp 2018 pC)

題敘我忘了，我只記得故事跟做法 XD

當年我的 NPSC 隊友在順利拿下亞軍之後，在新北市的 TOI 養成營隊裡因為不會 sort 而與 TOI 初選資格錯身而過 (備取超過十，後來還是補進去了 XD)，當場在座位上爆哭，有人還為了這個改編了很多首歌並且寫了一首詩來紀念他隆重的競賽退休儀式：
萬 兔 咧滴 夠
嗯又嗯又嗯又嗯又
嗯又嗯又嗯又嗯又
早上大排到小
下午前後顛倒
兩兩相加做好
最長工時扣掉
負則不要加到
乘加班費等跑
ＡＣ就沒煩惱
嗯又嗯又嗯又嗯又
嗯又嗯又嗯又嗯又
備註：他參加了 2018, 2019 的 TOI 初選，全部輸光光。
備註 2：編這首詩的人現在進資奧一階了
備註 3：兩個隊員都不用會排序照樣拿 NPSC 亞軍

習題 5.10.5: ZJ c431

一開始有個數字 $n(1 \leq n \leq 1048576)$，接下來一行 $a_1a_2a_3.....a_n$ 共 n 個數字 $(1 \leq a_i \leq 100)$，請將這 n 個數字由小到大排序

習題 5.10.6: ZJ c531

有 10 至 20 個用逗號分隔的數字。請將數字中偶數的數字加以排序，請勿更動奇數數字的位置。並將結果輸出。

習題 5.10.7: TIOJ 1609

給一個二元搜尋樹的前序尋訪結果，求中序輸出結果。

習題 5.10.8: TIOJ 1585

定義 $\mathrm{a} < \mathrm{z} < \mathrm{b} < \mathrm{y} < \mathrm{c} < \mathrm{x} < \ldots\ldots$，排序長度 < 1000 的字串

習題 5.10.9: 逆序數對 (TIOJ 1080)

給一個序列 $< a_n >$，求有幾對 (i, j) 使得 $i < j$ 且 $a_i > a_j$

區間資料結構 6

在這個章節當中，我們將會提到一些處理區間問題的常用資料結構，包括線段樹以及樹狀樹組。在一個被稱為區間問題的題目當中，我們要處理的是計算一個序列中一段連續區間內某些特別的值。比方說區間總和、乘積、最大公因數、最大值或是最小值都是十分典型的區間問題。

6.1 靜態區間問題

先來看看簡單題目：

習題 6.1.1: 簡單區間問題

給一個長度為 n 的序列以及 l, r 兩個整數 $(l, r \leq n)$，求區間 $[l, r]$ 中所有數字的總和。

這個問題最簡單的方法就是一個迴圈掃過去，事實上，它也是一個有效率的做法。

```
int rangeSum(int l, int r, int arr[]){
    int ans = 0;
    for(int i = l; i <= r; i++)
        ans += arr[i];
    return ans;
}
```

這個方法的確能有效率的達到解題的目的，但是當一次出現很多筆詢問時，每次都暴力掃過去就稍嫌太慢了，讓我們看看以下例題。

習題 6.1.2: 吞食天地 (ZJ a693)

給一個長度為 n 的序列，接下來會有 m 筆詢問，每筆詢問會輸入兩個數 l, r，對於每筆詢問，必須輸出區間 $[l, r]$ 中的數字總和。$(n, m, l, r \leq 10^5)$

這題可以用簡單的前綴和完成，讓我們再看一個難一點的題目。

習題 6.1.3: 直升機 (TOI 2018 初選, TIOJ 2055, ZJ d539)

給一個長度為 n 的序列 $(a_i \leq 10^6)$，接下來會有 n 筆詢問，每筆詢問會輸入兩個數 l, r，對於每筆詢問，輸出區間 $[l, r]$ 中的數字最小值再加上 1。$(n, l, r \leq 10^5)$

這題的原題敘比較難懂 (~~事實上超難懂，還錯誤百出~~)，事實上就是區間最小值，直接開個 Sparse Table 就過了。當然，最大值或最大公因數也可以這樣做。

6.2 讓序列動起來

於是我們就解決好多靜態區間問題了，恭喜！但是當問題開始變動態，一切就麻煩了，以下問題是動態區間處理的經典題，接下來整個章節都會以這個問題為核心進行推廣。

習題 6.2.1: 單點修改區間和 (奇怪的是竟然找不到 judge OAO)

給一個長度為 n 的序列，接下來會有 q 筆操作，每筆操作可能為下列兩種：

1. `1 p v`：將序列中第 p 個數加上 v
2. `2 l r`：輸出區間 $[l, r]$ 的數字總和

$n, q, p, l, r, v, a_i \leq 10^6$，保證過程中所有數字以及答案皆小於 10^{18}。

這個問題也常以各種形式出現或隱藏在各大演算法競賽的題目中，熟悉這種問題的模式以及精髓絕對是百利而無一害的，那就讓我們開始吧！

6.3 線段樹 (Segment Tree)

線段樹是一個可以支援下列兩種操作的資料結構：計算區間問題以及修改序列中的值。這裡的區間問題可以是區間和、區間乘積、區間最大/最小值等等，這些操作都可以在 $O(\log n)$ 時間內完成。

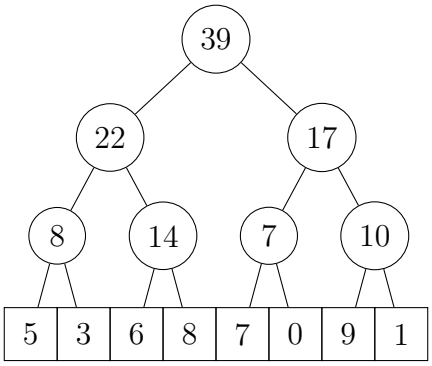
原理及結構

線段樹的構想是來自於分塊 (塊狀數組) 的進化版本，不過這次不是 $\sqrt{n}$ 個元素分一塊，而是兩兩一組；也不是只有一層，而是分塊中還有分塊，一層一層的疊上去，好像一棵樹一樣。

讓我們回到區間求和的問題。考慮以下陣列：

0	1	2	3	4	5	6	7
5	3	6	8	7	0	9	1

我們可以將這個陣列轉換成以下的線段樹：



其中每個節點都對應著一個區間，並且每個區間都會被分為兩個大小差不多的區間，也就是左右兩個子節點代表的區間。在區間求和的問題中，我們會把對應區間的和儲存在節點內，因為線段樹有分塊的性質，所以這個值剛好會等於左右兩個子節點的和。

我們的節點長怎樣

為了方便處理區間問題，在一棵線段樹的節點中，我們除了會記錄這個區間的答案（總和）之外，基於實作上方便，還會額外儲存指向兩個子節點的指標，以便操作使用。有的人會在節點裡儲存對應區間的左界與右界，這樣雖然直觀，但是事實上區間的左右界可以在操作過程中計算出，所以如果是競賽程式目的，就不需要浪費空間額外存取左右界。

```
1 struct node(){
2     int v;
3     node *l, *r;
4     node(int v): v(v), l(nullptr), r(nullptr){}
5     node(node *l, node* r): l(l), r(r){pull();}
6     void pull(){if(l) v = l->v + r->v;}
7 };
```

這裡的 `pull()` 函數類似計算整個分塊的答案，線段樹的每個分塊只有兩個子節點，所以只要對兩個子節點進行處理。這個函數可以根據你所要解決的區間詢問而進行更改，以區間和的詢問為例，`pull()` 函數就是兩子節點的值的和。

第 4 行的 constructor 是葉節點的建構元，因為葉節點沒有子節點，所以 `l`, `r` 兩個指標都指向 `NULL`，也因為沒有子節點的緣故，所以必須傳入 `v` 來手動為節點賦值。

第 5 行是一般節點的建構元。在樹狀結構中，每個指向節點的指標都可以視為一棵樹，線段樹是一層一層蓋上去的，事實上這個動作能看成在兩棵子樹上新增一個分塊節點，其代表的區間就是兩個子區間合併之後的區間。

線段樹的構建

前面提到過陣列與線段樹的轉換，那麼給定一個陣列，我們要怎麼建構對應的線段樹呢？

我們知道一個節點代表的是一段區間，對於一個長度為 n 的序列來說，對應的線段樹的根節點代表的區間就是 $[0, n)$，如果這個節點還不是葉節點，無法確定這個節點該有的值，所以要將區間分成兩半，對左右兩半的序列以遞迴的方式分別建立一棵線段樹，直到左右兩節點的值都已經確定，再做一次 `pull()` 操作連結父節點以及兩個子樹並確定父節點的值；如果這個節點已經是葉節點了 (也就是區間長度只有 1)，就代表這個節點的值就是原序列上對應區間 (只有一個數) 的值，因此只要將對應的值填入即可。

```
node *build(int l, int r, int arr[]){
    int mid = l+r >> 1;
    if(l+1 == r) return new node(arr[l]);
    return new node(build(l,mid,arr),build(mid,r,arr));
}
```

因為 `mid` 在這裡面十分常用，所以在實作線段樹時，都會習慣性地加上：
`#define mid l+r>>1`

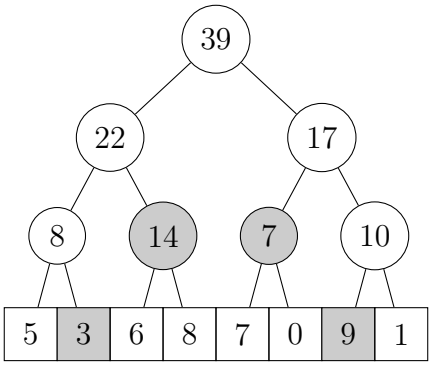
宣告一個長度為 n 的線段樹的方式如下：

```
node *segTree = build(0, n, arr);
```

其中的 `arr` 是一個已經填好值的陣列，建構完的線段樹即可以對這個序列進行區間詢問。

查詢區間問題

我們有了線段樹之後，要怎麼查詢一個區間的總和呢？與分塊一樣，我們可以將詢問區間拆成很多個小區間，這邊有點類似 sparse table 的感覺，但這裡區間必需清楚分割，不允許疊合。比如說要查詢區間 $[1, 6]$，那就必須查詢以下四個節點。



給定一組 l, r 以及一個序列對應的線段樹，我們能在眾多的子節點中選出足夠的節點，使得合併起來的答案就是所求 $[l, r]$ 的解，但總不能全部選擇葉節點吧！一般來說查詢的方法會用到遞迴的概念，從根節點開始根據規則對特定的節點做尋訪，每次遇到一個節點時可以分成三種 case：

1. 這個節點對應的區間與詢問區間完全互斥：

   這個節點的值並不會影響整個詢問的答案，故只要回傳一個無關緊要的答案（也就是合併運算的單位元）就行了。在區間和的問題中，這個值是 0；在區間最小值的問題中，這個值是INF。

2. 這個節點對應的區間完全包含於詢問區間之內：

   這個值將完全影響詢問的答案，因此直接回傳節點的數值即可。

3. 兩個區間互相交錯，只有部分重疊（即非上述兩種情況）：

   這個節點無法為答案直接提供任何訊息，只能將區間分成兩半 (也就是左右子節點) 分別遞迴計算答案，然後再進行一次合併運算決定這個節點的答案。

在實作細節上，我們會實作一個函數，傳入一個節點以及它對應的區間左右界，一開始傳入根節點以及 $[0, n)$，再根據情況分成三種 case，不斷遞迴下去。

```
int query(node *a, int l, int r, int ql, int qr){
      if(r <= ql || qr <= l) return 0; // 互斥
      if(ql <= l && r <= qr) return a->v; // 完全包含
      return query(a->l, l, mid, ql, qr) +
                query(a->r, mid, r, ql, qr); // 只有部分重疊
}
```

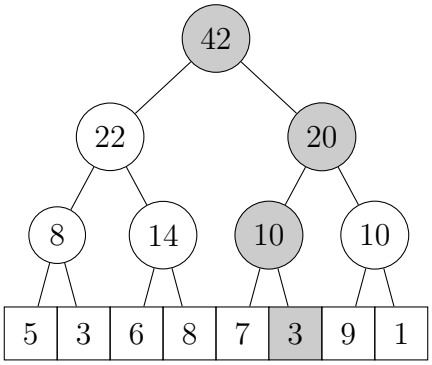
我們多傳了 ql, qr 兩個參數，代表詢問區間；每次要詢問時只要呼叫以下函數就完成區間詢問的操作了：

```
// 這裡的 l, r 是要詢問的區間，左閉右開。
query(segTree, 0, n, l, r);
```

修改一個值

修改是線段樹的靈魂。沒有修改，也不需要線段樹了，前綴就辦的到。那麼當序列的值被更動了之後，我們要怎麼對線段樹修改，才能維持詢問操作的正確答案呢？

你會發現當修改序列中的一個值時，只有一些節點會受到影響，如下圖所示：



更新這些節點的方式十分簡單，一樣從根節點遞迴下去，遇到每個節點就分成三個情況：

1. 此節點為葉節點：

   直接將此節點的值改掉，不需要做任何多餘操作。

2. pos 在節點對應區間的左半邊，即 pos < mid：

   對左子節點遞迴下去，更新完左子樹記得 pull() 一下來更新這個節點的值。

3. pos 在節點對應區間的右半邊，即 pos >= mid：

   對右子節點遞迴下去，更新完右子樹記得 pull() 一下來更新這個節點的值。

```
void modify(node *a, int l, int r, int pos, int val){
    if(l+1 == r) a->v += val;
    else if(pos < mid) modify(a->l, l, mid, pos, val);
    else modify(a->r, mid, r, pos, val);
    a->pull();
}
```

更新序列中的一個值只要簡單的呼叫以下函數就可以達成了：

```
modify(segTree, 0, n, p, v);
```

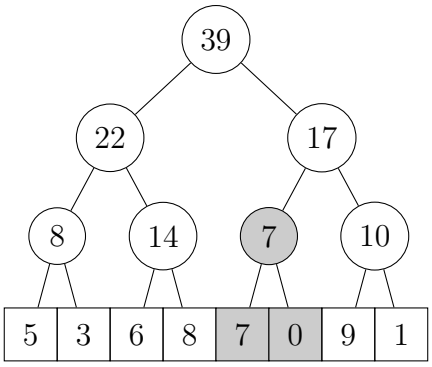
這裏的 p, v 就是題目輸入的位置與需要增加的值。

線段樹的複雜度

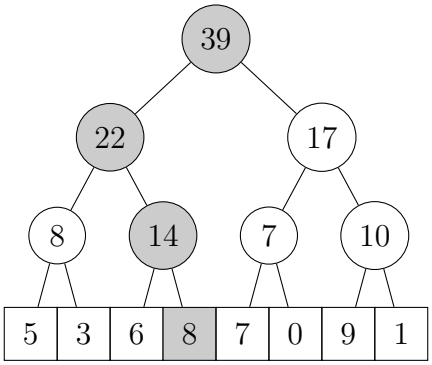
再來我們來討論線段樹的複雜度。我們先回憶一下我們需要線段樹幹嘛：線段樹提供了區間查詢以及單點修改的操作，這兩個操作可以暴力 $\langle O(n), O(1)\rangle$ 或者前綴 $\langle O(1), O(n)\rangle$ 完成，對於要求比較嚴苛的題目來說，這樣的複雜度還是嫌太慢了，那麼線段樹的複雜度又是如何呢？

操作的代價

先來看詢問操作：觀察以下的圖示，因為每次查詢的是一個連續區間，所以每層頂多用到左右兩邊各一個節點。如果同一層中有兩個節點需要被尋訪到，則這兩個節點至少有一個必在上一層已經完全涵蓋，可以合併到上一層去。根據這個性質，我們可以知道被尋訪的節點數不會超過**兩倍樹高**，因此詢問的複雜度是 $O($ 樹高 $)$。



再來看修改的部分：當尋訪到一個節點時，如果不是葉節點，只會對左子樹或者右子樹遞迴下去，直到碰到需要修改的葉節點為止。因此對於一次修改操作，頂多會尋訪到**樹高**個節點，因此修改的複雜度也是 $O($ 樹高 $)$。



定理 6.3.1: 線段樹的樹高

因為線段樹是一棵 Complete Binary Tree，也是一棵平衡二元樹，所以一棵有 n 個節點的線段樹的樹高是 $O(\log n)$。

既然線段樹是一棵平衡二元樹，那麼修改與查詢操作的複雜度也是 $O(\log n)$。

建構的花費

既然查詢與修改的執行時間都如此迅速，想必是另外多花時間與空間進行預處理的。在這裡我們將討論建構線段樹所需的花費，這邊一定要保持腦袋清晰並銘記在心。

首先我們先討論空間複雜度的部分，在這裡先假設序列的長度是二的冪次，如果不是，我們可以在序列後面補 0。

我們知道線段樹是多層的分塊，每層是由下面的兩個節點組成。因此對於線段樹的每一層，節點數都會是下面一層的一半。將每層的所有節點數加起來，整棵樹的節點數就是 $n + \frac{n}{2} + \frac{n}{4} + ... + 1 = 2n$。

重要！線段樹的空間複雜度是 $O(n)$，不是 $O(n \log n)$。

再來是時間複雜度。這部分比較簡單，因為建構一個節點只是單純的呼叫一次建構元以及 `pull()` 函數，因此時間複雜度與空間複雜度相同，皆是 $O(n)$。

定理 6.3.2: 線段樹的複雜度

建構一個序列長度為 n 的線段樹需要 $O(n)$ 的時間以及 $O(n)$ 的空間。
對於一個序列長度為 n 的線段樹，查詢以及修改的時間複雜度皆為 $O(\log n)$。

一定要指標嗎

其實線段樹不一定要用上述的指標型來實作，因為線段樹是一棵 Complete Binary Tree，所以其實陣列是一個不錯的選擇，陣列型線段樹的優點將會在接下來的內容提到。

快速改造–陣列線段樹

指標型的線段樹要改成陣列其實十分的簡單，通常一個熟悉線段樹實作的人能在十分鐘內改造完畢。

在之前的章節中，我們有提到過二元樹的陣列表示法–利用編號 $2n$ 與 $2n + 1$ 的節點為編號 n 的節點的左右子樹，其中編號 1 是樹根。以下是節點與編號的對應：

如果 `a` $\to i$，則

- `a->l` $\to 2i$
- `a->r` $\to 2i + 1$

我們用個陣列來儲存節點的值，也就是說

- `a->v` $\to$ `segTree[i]`

最後注意一下實作細節，將 `idx` (即節點編號) 一起傳入函數，方便做陣列處理即可。

以下是陣列線段樹的程式碼：

```cpp
int segTree[4*MAX_N];

void pull(int idx){
    segTree[idx] = segTree[2*idx] + segTree[2*idx+1];
}

void build(int l, int r, int arr[], int idx = 1){
    if(l+1 == r) segTree[idx] = arr[l];
    else build(l, mid, arr, 2*idx),
        build(mid, r, arr, 2*idx+1), pull(idx);
}

int query(int l, int r, int ql, int qr, int idx = 1){
    if(r <= ql || qr <= l) return 0;
    if(ql <= l && r <= qr) return segTree[idx];
    return query(l, mid, ql, qr, 2*idx) +
           query(mid, r, ql, qr, 2*idx+1);
}

void modify(int l, int r, int pos, int val, int idx=1){
    if(l+1 == r) segTree[idx] += val;
    else if(pos < mid) modify(l, mid, pos, val, 2*idx);
    else modify(mid, r, pos, val, 2*idx+1);
    pull(idx);
}
```

注意這裡的陣列要開四倍，不然在修改或者查詢的操作中可能會超出陣列外面，造成 SF。

線段樹要開四倍，不是兩倍，更不是 $\log n$ 倍。

從這邊可以看出陣列型與指標型線段樹的微小差別：
https://drive.google.com/file/d/1l6fLCyxqCZS4TgBStCUoUMArfcg5Qo5f/view?usp=sharing

陣列線段樹有什麼好處

首先，最直接的就是記憶體用量。一個指標要占 8bytes ，一個節點就至少要 20bytes，而陣列只要區區四個位元組。有些題目故意卡記憶體就會卡到指標。再者，指標的存取速度比較慢，遇到卡常數的題目就會 TLE。有時候線段樹的初始值要全部都是 0，這時候指標的 build() 函數就會執行過慢導致逾時；而陣列一宣告就可以自動填上 0，甚至不需要呼叫 build() 函數。

另外還有一種偽指標的方法，也就是在一開始開好一個拿來當記憶體池的陣列，並且以陣列的索引值代替指標，如此能夠省下部分的空間複雜度，實作上只需要把指標型的程式碼中new的部分改為回傳一個空的陣列索引即可。

6.4 BIT 樹狀數組

BIT，是 Binary Indexed Tree 的簡稱，最早由 Peter Fenwick 所發明，因此又稱作 Fenwick Tree，中文譯作樹狀數組，是一個拿來處理動態前綴和的資料結構，以下我們將直接簡稱 BIT。
讓我們先看看題目～

習題 6.4.1: 經典題 - 動態前綴和 (No judge)

給定一個序列 $a_1, a_2, \ldots, a_n$，之後有 q 筆操作，操作有以下兩種：查詢前 i 項的和、修改 a_i 的值。$n, q \leq 10^5$

對於這樣一個看起來很簡單的問題，大家可能會想到兩種解法：

- 每次詢問時便計算一次 $a_1 + a_2 + \cdots + a_i$，修改時直接修改該項的值即可。詢問複雜度 $O(N)$、修改複雜度 $O(1)$
- 用一個陣列 pre 來維護前 i 項的和，詢問時直接輸出答案，修改時則需要改變 pre[i] ～pre[n] 的值。詢問複雜度 $O(1)$、修改複雜度 $O(N)$

這兩種方法分別將複雜度花在了詢問或修改上，但總複雜度一樣都是 $O(NQ)$，這時候便出現了一種想法，有沒有可能讓詢問及修改的複雜度盡可能平均，而降低總複雜度呢？

如果有認真看前面講義的人，可能就會想到可以利用線段樹來解決這個問題，而在刻線段樹的同時，大家會發現對於每個 node，左子節點的值加上右子節點的值其實就是這個 node 自己本身的值。我們便利用這樣的性質對這棵線段樹進行一點優化，試著將右子節點全部刪掉，然後用左子節點和節點本身來表示右子節點的值，這就是 BIT 最重要的精神！

那麼，這樣一顆奇形怪狀的樹到底該怎麼維護呢？經過一番整理與歸納後我們會發現，在這棵樹上的每個節點的範圍 $[l, r]$ 中，每個 r 會對應到唯一的 l，這表示我們只要用一個陣列來記錄這些節點的值就行了，我們且將這個陣列稱作 BIT。

接著我們要討論到底該如何完成 BIT 陣列，由前面討論出我們所要記錄的節點，可以發現這些節點所記錄的區間其實就相當於 i 以二進位表示後最低位的 1 的位置（之後稱之為 lowbit(i)），例如 $26_{10} = 11010_2$，BIT[26] = [26−lowbit(26) + 1, 26] = [25, 26]，而預處理這樣的陣列也有個簡單的作法，我們維護當前的前綴 pre[i]，那麼 BIT[i] 就會等於 pre[i] - pre[i−lowbit(i)]，便能以 $O(N)$ 預處理建出 BIT 陣列了。

完成了 BIT 陣列的建置後，便可以開始進行操作了，我們先以詢問的角度來看，對於每次的詢問 $[1, k]$，將詢問分為 $[1, k-\text{lowbit}(k)]$ 與 $[k-\text{lowbit}(k)+1, k]$ 兩部分，根據上段所述，我們能以 $O(1)$ 求得後項的答案，之後再對前項遞迴下去，最多只需 $\log N$ 次，便能得到 $[1, k]$ 的值了。

接下來是修改的部分，當我們修改 a_i 的值時，考慮到被影響的內容是 BIT$[i]$、BIT$[i+\text{lowbit}(i)]\cdots$（有點通靈，請大家認真思考一下），於是我們便遞迴下去，僅需 $O(\log_2 N)$ 即可完次成對 BIT 陣列的修改。

順帶一提，計算 lowbit(k) 的方式也不難，lowbit$(k) = k\&(\sim k+1)$（& 是 bitwise-and、~是 bitwise-not），而利用目前大部分的處理器架構中，負數其實就是 2 補數（反轉再加一）的特性，又可以直接寫作 $k\& -k$。

```
int a[N+1], pre[N+1], bit[N+1];
inline int lowbit(int n) { return n&-n; }
void build() {
  for (int i = 1; i <= N; i++)
    bit[i] = pre[i] - pre[i-lowbit(i)];
}
int sum(int n) {
    int ret = 0;
    for(;n;n-=lowbit(n)) ret += bit[n];
    return ret;
    //類似將n以二進位分解
}
void add(int n, int v) {
    for(;n<=N;n+=lowbit(n)) bit[n] += v;
    //想像更新編號為n的節點的所有父節點
}
```

初始化、詢問、修改三個函式各自的程式碼都不超過 3 行，這應該是各位除了 STL 外會接觸到最親民的資料結構了，因此一定要好好把握。

最後來統整一下複雜度吧！按照前面所講的作法，預處理 $O(N)$、每次查詢或修改 $O(\log N)$，總複雜度為 $O(Q\log N)$，複雜度不輸線段樹，coding 複雜度又低，無怪它這麼受人歡迎呢！

值域 BIT

話不多說，請大家看看例題吧

習題 6.4.2: 經典題 - 逆序數對 (TIOJ 1080)

給定一個序列 $a_1, a_2, \ldots, a_n$，詢問此序列中有多少對 (i, j) ，使得 $i < j$ 且 $a_i > a_j$。$n \le 10^5$

看到這個題目很容易發現，我們只要統計每個 a_i 前面比它大的數字並加總，便能得到答案了，現在的問題只剩下，要怎麼快速知道每個 a_i 前面有多少個比它大的數呢？這時候我們的 BIT 就派上用場了，首先定義 ans 表示答案，然後我們讓 BIT$[a_i]$ 表示目前有多少個 a_i，接著我們從 a_0 開始，每次讓 ans 加上 $(i-[1, a_i])$ 表示當前比 a_i 大的數的數量，接著將 a_i 加入 BIT 陣列中，完成後便能得到我們的答案，而這就是所謂值域 BIT 的概念。

```
int num[N+1], bit[N+1], ans;
inline int lowbit(int n) {return n & -n;}
```

```
int sum(int n) {
  return n ? bit[n] + sum(n-lowbit(n)) : 0;
}
void add(int n, int v) {
  bit[n] += v;
  if (n <= N) add(n+lowbit(n),v);
}
int main() {
  for (int i = 0; i < N; i++) {
    ans += i-sum[num[i]];
    add(num[i], 1);
  }
  return 0;
}
```

區間加值 & 單點查詢

接下來開始是一些進化版的 BIT

習題 6.4.3: 區間加值 & 單點查詢 (No judge)

給定一個序列 $a_1, a_2, \ldots, a_n$，之後有 q 筆操作，操作有以下兩種：查詢第 i 項的值、使區間 $[l, r]$ 同時 $+k$。$n, q \leq 10^5$

這問題感覺與普通的 BIT 有種似曾相識的感覺（?，差別只在於這次我們是一次修改一個區間，而查詢也縮成了一次只查詢一個點，對於這樣的問題，我們可以想想該如何變化我們的 BIT 。首先，一次必須修改一個區間，我們勢必不能對這個區間內的值一個一個修改，因此，我們會需要用到一個與前綴恰恰相反的方法－差分，我們先建立原數列的差分陣列 d，而第 i 項的值就是 $[1, i]$ 的前綴和，接著來看看修改的部分，當我們將區間 $[l, r]$ 同時 $+k$，也就表示我們讓 $\mathrm{d}[l]$ 的值 $+k$、$\mathrm{d}[r+1]$ 的值 $-k$，如此一來便能成功用 BIT 達成區間加值 & 單點查詢了。

區間加值 & 區間查詢

再更進階一點...

習題 6.4.4: 區間加值動態前綴和 (No judge)

給定一個序列 $a_1, a_2, \ldots, a_n$，之後有 q 筆操作，操作有以下兩種：查詢前 i 項的和、使區間 $[l, r]$ 同時 $+k$。$n, q \leq 10^5$

情況似乎又變得更複雜了...，這次似乎無法單靠將原本的陣列轉成差分便求得答案，這時候我們決定先將詢問轉化看看：

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} \sum_{j=1}^{i} d[j]$$

考慮每個 $d[j]$ 被加到的次數

$$\sum_{i=1}^{n}\sum_{j=1}^{i} d[j] = \sum_{i=1}^{n}(n-i+1)\cdot d[i] = (n+1)\cdot\sum_{i=1}^{n} d[i] - \sum_{i=1}^{n} i\cdot d[i]$$

因此我們可以利用兩個 BIT 來做到區間加值 & 查詢！，一個 BIT 維護 $\mathrm{d}[i]$，另一個維護 $i\cdot\,\mathrm{d}[i]$，然後再用上述的方法合併，修改的部分如前一小節所述。

更高維的版本

BIT 動態前綴和推廣成高維版本的方式，就是使 BIT 的每一項都成為一個 BIT ，這樣便是一個二維 BIT 了，至於複雜度的話，每多增加一維，便會變為 $\log N$ 倍，因此複雜度為 $O(\log N^D Q)$，但倘若是要查詢任意區間的話，就必須要使用排容原理，複雜度 $O(2^D \log N^D Q)$ 。

6.5 Sparse Table

sparse table 主要是處理 RMQ（區間最小值）的問題，為倍增法的一種實現。考慮到暴力建表需要 $O(N^2)$ 的複雜度，我們發現以下的性質：

$$min(a,b) = min(a,b,b)$$

也就是說，一段區間詢問的拆解是可以拆成有交集的區間（因為相同數字重複取 min 並不會影響答案）。如此一來，我們只要想辦法用某些長度的聯集湊出任意區間，就能加快詢問速度了。但我們該用哪些長度湊出這些區間呢？想起來我們的前言了嗎？沒錯，實際上用長度為 2 的冪次的區間來處理會是最有效的。那對於某段區間 $[l,r)$，我們該如何拆解呢？我們只要找到滿足下列式子的 d 即可：

$$\tfrac{r-l}{2} \le d = 2^k \le r-l$$

不難發現 $k = [\log_2(r-l)]$，$d = 2^k = 2^{[}\log_2(r-l)]$。因此 $min([l,r)) = min([l,l+d),[r-d,r))$。於是我們建立一個 st 表格，其中 $\mathrm{st}[i][j]$ 代表編號在 $[j,j+2^i)$ 範圍內的最小值。這樣一來只要將詢問轉換一下就可以 $O(1)$ 查詢了。

預處理

建表時不難發現以下式子：

$$\begin{cases}\mathrm{st}[i][0] = a[i]\\ \mathrm{st}[i][j] = min(\mathrm{st}[i-1][j],\mathrm{st}[i-1][j+2^{i-1}])\end{cases}$$

表格總共 $\log_2 N$ 行 N 列，可以 $O(N\log N)$ dp 建完。

查詢

查詢時根據上述原理將輸入範圍 $[l,r)$ 轉換為 $[l,l+2^{[\log_2(r-l)]})$ 和 $[r-2^{[\log_2(r-l)]},r)$ 兩段，於是 $[l,r)$ 的答案就會是 $min(\text{st}[[\log_2(r-l)]][l],\text{st}[[\log_2(r-l)]][r-2^{[\log_2(r-l)]}])$，達成 $O(1)$ 查詢。

以下是程式碼：

```
int a[maxn],st[(int)log2(maxn)+1][maxn];
void build(int n){
    for(int i=0;i<n;i++) st[0][i]=a[i];
    for(int l=1,i=1;i<=(int)log2(n);i++,l<<=1)
        for(int j=0;j+l<n;j++)
            st[i][j]=min(st[i-1][j],st[i-1][j+l]);
}
int query(int l,int r){
    return min(st[(int)log2(r-l)][l],st[(int)log2(r-l)][r-1<<((int)log2(r-l
}
```

小小的推廣

上述 sparse table 的 min 運算可替換為「具交換律、結合律且滿足 $f(x,x)=x$」的任意二元運算 f。

習題 6.5.1: 胖胖殺蚯事件 (TIOJ1603)

給定 n 棟大樓高度 $h[i]$ 和 m 筆詢問 $[l,r]$，輸出每筆詢問區間的最大值 $-$ 最小值。

基礎圖論 7

圖論在演算法這門學科裡佔了十分重要的地位，在競賽中有大約一半的題目會用到圖論的算法與觀念，學著怎麼處理圖，與學習語法一樣重要。

7.1 名詞解釋

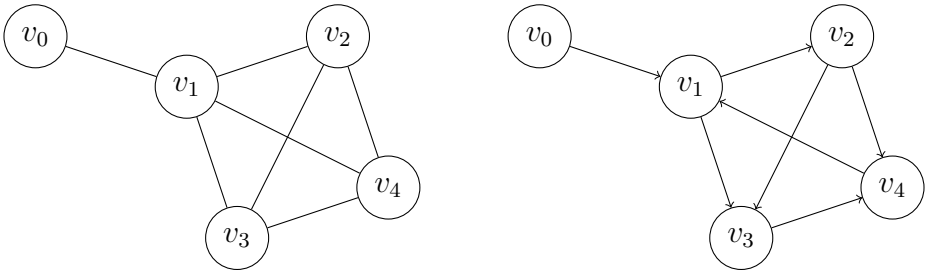
一堆怪東東

1. 圖 (Graph)：許多頂點與邊的集合，常用 $G(V, E)$ 表示。
2. 頂點 (Vertex)：就是頂點。常用 v 表示。V 就是頂點的集合。
3. 邊 (Edge)：連接兩個頂點的東西，可表示成 $e = (u, v)$。E 就是邊的集合。
4. 有向/無向 (Directed/Undirected) 邊：如果 (u, v) 與 (v, u) 代表的是同一條邊，則稱這條邊是無向邊，反之則其中任一條邊為有向邊。
5. 度數 (Degree)：一個頂點連接的邊數，即為這個頂點的度數。
6. 入度/出度 (Indegree/Outdegree)：若為有向邊，則度數分為入度與出度，分別代表以此頂點為終點與起點的邊數。
7. 鄰接 (Adjacent)：如果兩個頂點間有邊連接，則稱這兩個頂點鄰接。
8. 自環：連接相同頂點的邊，即 (v, v)。
9. 重邊：兩條或以上連接相同兩個點的邊。
10. 路徑 (Path)：一個頂點與邊交錯的序列，滿足每個邊都要連接兩個頂點，且從頂點開始、頂點結束。以 $(v_s, e_1, v_1, e_2, v_2, ..., v_e)$ 表示。
11. 行跡 (Trace)：不包含重複邊的路徑。
12. 簡單路徑 (Track)：不包含重複頂點的路徑。
13. 迴路 (Circuit)：起點與終點為相同頂點的路徑。
14. 環 (Cycle)：起點與終點為唯一相同頂點的簡單路徑。

各種圖

1. 有向圖/無向圖：每條邊都是無向邊的圖稱為無向圖，反之為有向圖。
2. 帶權圖/不帶權圖：有時點上或邊上會有權重，稱為帶權圖。
3. 連通圖：把所有邊變成無向邊後，對於圖上的所有頂點對 (u, v)，都存在一個起點為 u，終點為 v 的路徑，則稱這個圖為連通圖。
4. 強連通圖：若有向圖上的所有頂點對 (u, v)，都存在一個起點為 u，終點為 v 的路徑，則稱這個圖為強連通圖。
5. 簡單圖：沒有重邊以及自環的圖，不一定連通。
6. 完全圖：每個頂點都與圖上其他所有頂點鄰接的圖，稱為完全圖。
7. 子圖：今有兩圖 $G(V, E)$ 與 H，若對於所有屬於 H 的頂點 v_i 與邊 e_i 皆有 $v_i \in V$ 且 $e_i \in E$ (即 H 內所有頂點與邊都屬於 G)，則稱 H 是 G 的子圖。
8. 補圖：若圖 G 與圖 H 的頂點集合相同，且兩圖的邊集合聯集為完全圖、交集為空集合，則稱圖 G 與圖 H 互為補圖。
9. 樹：沒有環的連通無向圖稱為樹。
10. 森林：很多樹 (包括一棵) 的聯集稱為森林。
11. 二分圖：如果可以將一張圖的點集分為兩部分，同一部分的任兩點不鄰接，則稱為二分圖。
12. 有向無環圖：簡稱 DAG，就是沒有環的有向圖。
13. 稀疏圖/稠密圖：如果邊數十分多（如完全圖），也就是 $|E| = O(|V^2|)$，則稱這個圖是稠密圖。若邊數不多（$|E| = O(|V|)$ 或 $O(|E|) = O(|V| \log |V|)$），則稱為稀疏圖。

以上是定義，讓我們看看例圖：



我們常用這種方法表示無向圖與有向圖。圖論中的圖上每條邊都是連接兩個頂點，不會交叉。

7.2 圖的儲存

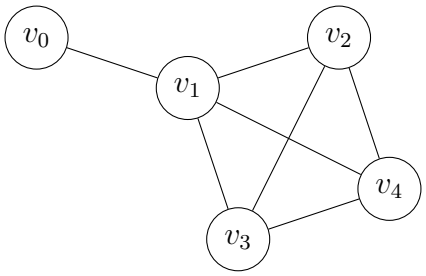
在學習圖論的時候，看到各式各樣的演算法，一定要先確定這個演算法要用什麼方式把圖存下來會比較好處理。這裡提供了三種把圖存在記憶體裡的方式，各自有各自的優缺利弊，在不同圖論算法上有不同應用。

通常測資在輸入一張圖時，會先給定兩個數 n, m，分別代表頂點數以及邊數，接下來會有 m 行，每行輸入兩個數字 u_i, v_i，代表有一條邊從頂點 u_i 連至頂點 v_i，如果是帶權圖，那麼每行會輸入三個數，分別代表兩端點以及權重。

讓我們把這種輸入格式轉成方便處理的形式吧！

鄰接矩陣 (Adjacency Matrix)

鄰接矩陣是把圖存下來最直覺的想法。考慮一張無向圖：



我們可以將這張圖轉成以下矩陣：

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

若 A 是一張圖的鄰接矩陣，表示頂點 i 與頂點 j 之間有邊時 $A[i][j] = A[j][i] = 1$，反之 $A[i][j] = 0$。

定義 7.2.1: 鄰接矩陣

對於一張圖 G，若矩陣 A 滿足：

$$A[i][j] = 1, \text{若邊 } (i, j) \in G$$

$$A[i][j] = 0, \text{若邊 } (i, j) \notin G$$

則稱矩陣 A 是圖 G 的鄰接矩陣。

鄰接矩陣也可以處理有向圖以及帶權圖。帶權圖的處理方式十分簡單，就是直接把權重存入鄰接矩陣內；對於一個有向邊 (i, j)，則可以用 $A[i][j] = 1, A[i][j] = 0$ 來表示。

將輸入轉為鄰接矩陣的方式非常容易，只要將輸入的邊一一填入矩陣即可。

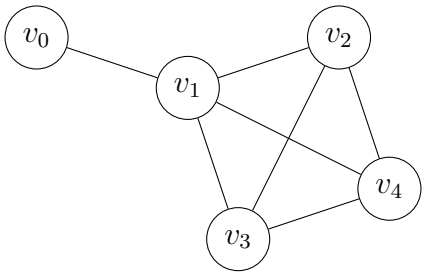
```
int A[MAX_N][MAX_N] = {}; // 初始化為 0
main(){
    int n, m, a, b;
    cin >> n >> m;
    for(int i = 0 ; i < m; i++){
        cin >> a >> b;
        A[a][b] = A[b][a] = 1; // 有邊則改為 1
    }
}
```

鄰接矩陣雖然可以在 $O(1)$ 時間檢查兩個頂點之間是否有邊，但是需要用到 $O(|V|^2)$ 的記憶體，所以鄰接矩陣只適合儲存稠密圖。雖然大部分演算法競賽都是以稀疏圖的應用為主，但是鄰接矩陣也有 Floyd Warshall 演算法以及矩陣樹定理等應用。

7.3 鄰接串列 (Adjacency List)

至於稀疏圖，用鄰接串列來儲存是一個比較好的選擇。

我們一樣用同一張圖：



可以轉換成下列鄰接串列：

$$\begin{array}{lcl} 0 & : & 1 \\ 1 & : & 0 \quad 2 \quad 3 \quad 4 \\ 2 & : & 1 \quad 4 \quad 3 \\ 3 & : & 1 \quad 4 \quad 2 \\ 4 & : & 2 \quad 3 \quad 1 \end{array}$$

每個數字 i 後面接的一長串數字就是頂點 i 有連接到的所有頂點的編號。注意串列串的一串數字是無序的，所以檢查兩個點是否有鄰接的最差時間複雜度是 $O(|E|)$。但如果將鄰接串列先排序過，就可以用 $O(\log|E|)$ 的時間二分搜檢查鄰接性了！

我們常用一個 `vector<int> L[MAX_N]` 來實作鄰接串列。`L[i]` 是一個 `vector`，這裡儲存所有與頂點 i 鄰接的所有頂點的編號，如果是帶權圖，就改儲存一個 `pair`，表示鄰接頂點的編號與這條邊的權重。以下是將輸入轉成鄰接串列的方法。

```cpp
vector<int> L[MAX_N];
int main(){
    int n, m, a, b;
    cin >> n >> m;
    for(int i = 0 ; i < m; i++){
        cin >> a >> b;
        L[a].push_back(b);
        L[b].push_back(a); // 如果是無向圖必須加上反向邊
    }
}
```

鄰接串列應該是演算法競賽裡最常出現的圖儲存方式了，超過半數的問題都是用鄰接串列實現。接下來要提到的 DFS、BFS、最短路徑、二分圖色問題等等，都可以用鄰接串列來完成。

7.4 很多邊的東東（?）

這個東西的英文叫 Edge List，就是一堆邊的集合，這個儲存方式比較簡單也比較接近輸入格式，就是用個 vector 將所有邊的兩端點儲存下來，維持原本的輸入格式也可以應用在一些好用的演算法。以下是範例程式碼。

```cpp
vector<pair<int,int>> E;
signed main(){
    int n, m, a, b;
    cin >> n >> m;
    for(int i = 0; i < m; i++){
        cin >> a >> b;
        E.push_back({a, b});
    }
}
```

這種儲存方式可以解決最小生成樹的問題，或者實作一些只需要枚舉邊的演算法。

其他存圖的方式

如果讀者有接觸過樹狀資料結構的話，就會知道可以用指標或陣列儲存一棵樹，而樹也是圖的一種，因此如果圖是一棵樹的話，也可以嘗試用樹狀結構的儲存方法。

另外，圖還有一種叫做前向星（Forward Star）的儲存方式，不過前向星能做到的事都可以被前面三種所取代，而且實作沒有比較簡單，所以就逐漸被程式競賽淘汰了。

7.5 戶口調查時間！

有了圖之後，我們就到各個頂點看看吧！Let's go！

深度優先搜尋 (Depth-First-Search, DFS)

讓我們發揮冒險精神，進入鄰接串列迷宮，往深處探險去吧！

終於到了頂點 s 了！這裡還沒被插上探險的標記，讓我們跟居民聊聊天吧。

冒險者：叩叩叩，請問一下你們家有多少人？

s 屋主：寒舍只有我與妻子二人，要入屋內坐坐嗎？我這就去殺雞設酒。

冒險者：不用了，謝謝。我還要繼續冒險呢！

s 屋主：這是我家私藏的冒險地圖，寫著與這裡鄰接的節點們，祝你好運！

冒險者插上了旗子，代表來過這裡的標記，就收起行囊離開前往頂點 t 了。

```
vector<int> G[MAX_N];
bool visited[MAX_N] = {};
void dfs(int s){
    // process vertex s
    visited[s] = true;
    for(auto t: G[s]){
        if(!visited[t]) dfs(t);
    }
}
```

DFS 是圖論演算法的基礎，通常會使用遞迴來實作，當所有節點都已經被遍歷過時，就達到遞迴的中止條件，可以停止演算法了。在這個程式碼當中 `if(!visited[t])` 的部分在所有節點都已經被遍歷時就不會被呼叫，所以這個遞迴呼叫就一定會被中止。

值得注意的是：一次 DFS 只能拜訪過與起點連通的連通塊的所有節點，如果你想遍歷整張圖的所有節點，必須對所有未被拜訪過的頂點 DFS 一次，才能確保所有節點都被計算到。這樣雖然最多可能會進行 DFS $O(|V|)$ 次，但是 `visited` 陣列每格最多只會被改成 `true` 一次，所以均攤複雜度仍然是 $O(|V|)$

```
int main(){
    // 在這裡輸入鄰接串列
    for(int i = 0; i < n; i++){
        if(!visited[i]) dfs(i); // 拜訪所有節點
    }
}
```

對於非連通圖的遍歷，一定要寫個 for 迴圈對所有節點 DFS 一次，不然吃 WA 不瞑目。

廣度優先搜尋 (Breadth-First-Search,BFS)

相較於深度優先搜尋一路衝到底的精神，廣度優先搜尋比較接近一層一層的探索。

出了鄰接串列迷宮之後，冒險者得到了迷宮的全圖。

冒險者：這迷宮如此錯綜複雜，我要怎麼知道從起點走到每個頂點要多少時間呢？

紫紅神：你可以嘗試遍歷一次地圖，然後帶著一種具有距離標示的旗子。每次幫所有加過標記的點周圍都放上距離多 1 的旗子，這樣就可以卻保距離較近的點先被走到嘍！

冒險者：那我要怎麼一次找出所有已標記地點附近的未標記地呢？

紫紅神：你應該拿回迷宮的地圖了吧！你可以在標記一個地方的時候，將其附近的頂點全部都放入隊列當中，等到前面還沒標記完的節點被標記後就可以一次找出這個地點的所有附近的未標記地了呢！

冒險者：哇！好聰明！我來試試看！

```
vector<int> G[MAX_N];
bool visited[MAX_N] = {};
queue<int> que;
void bfs(int s){
    que.push(s);
    while(!que.empty()){
        int v = que.front(), que.pop();
        // process vertex v
        visited[v] = true;
        for(auto t: G[v]){
            if(!visited[t]) que.push(t);
        }
    }
}
```

BFS 一樣可以對整張圖進行遍歷，不過 BFS 有一個性質，就是先被處理到的點與起點的距離會比較近，也就是說這種演算法可以計算出起點到圖上任一點的最短路徑長度。讓我們看看以下例題：

習題 7.5.1: 最短路徑

給一張無向連通圖與起終點的編號，求從起點至少需要走過多少邊才能抵達終點。

對於這個問題，可以記錄一個陣列 `dist[i]` 代表從起點到頂點 i 的最短路徑，顯然，起點到起點的最短距離是 0。然後從起點開始 BFS，每次抵達一個節點 v 時，就將自己附近沒被標記過的節點設成 `dist[v]+1`，直到整個圖都被計算過之後，再求取終點的 `dist` 值即可。此外，因為 BFS 可以對整張圖進行遍歷，所以假設起點不動，一次 BFS 就可以算出起點到圖上任意終點的最短路徑。

```
int dist[MAX_N];
int visted[MAX_N];
vector<int> G[MAX_N];
queue<int> que;
void bfs(int s){
    dist[s] = 0;
    que.push(s);
    while(!que.empty()){
        int v = que.front(), que.pop();
        visited[v] = true;
        for(auto t: G[v]){
            if(!visited[t]){
                dist[t] = dist[v] + 1;
                que.push(t);
            }
        }
    }
}
```

7.6 完全搜尋可以幹嘛

以下用一些例題來舉例說明完全搜尋的用處吧！

習題 7.6.1: 新手訓練系列 ~ 圖論 (ZJ a290)

給一張有向圖與頂點 A, B，求是否可以從 A 通過圖上的邊抵達頂點 B。($|V| \leq 800, |E| \leq 10000$)

這題是個暖身題，十分容易。就是從頂點 A 開始 DFS，最後檢查 `visited[B]` 是否為真就行了，輕輕鬆鬆。

習題 7.6.2: 最佳路徑 (99 北基區資訊能競 p4, ZJ d908)

給一張有向且帶邊權的圖，以及起點邊號，求從起點開始的所有路徑裡面的最大權重和。(頂點編號 $\in$ 大寫英文字母，邊權 ≤ 100)

這題稍微麻煩了一點，這次不是求最短路徑，反而求起最長路徑來了。但其實想法是一樣的，用個 dist 陣列儲存到這個節點的最長路徑

習題 7.6.3: 空拍圖 (TIOJ 1336)

給一個 $H \times W$ 的照片，- 代表空地，G 代表綠地，W 代表河流或湖泊，B 代表建築物。如果有兩格八方位相鄰的綠地，那麼兩格綠地會被計算為同一塊綠地。同樣地，八方位相鄰的空地也會被視為同一塊空地。現在需要知道城市中究竟有多少塊綠地和空地。

這題一樣是 DFS 的應用。我們可以把照片想像成圖，在周圍八格的字元想像成鄰接。DFS 一次可以把一整個連通塊的地方都搜尋過一遍，所以我們可以掃過所有點，如果遇到綠地或空地就從那裡開始 DFS，每次 DFS 都將整個連通塊的綠地或空地破壞掉（改成）其他標記，再將答案加上 1。這樣枚舉完所有點後，計算完的答案就是正確的連通塊數量了！

```
char city[110][110];
int w, h, greenland = 0, emptyspace = 0;
void dfs(int x, int y, char ch){
    city[x][y] = 'V'; // visited
    int dx[] = {1,  1,  1,  0, -1, -1, -1,  0};
    int dy[] = {1,  0, -1, -1, -1,  0,  1,  1};
    for(int i = 0; i < 8; i++){
        if(x+dx[i] >= w || x+dx[i] < 0) continue;
        if(y+dy[i] >= h || y+dy[i] < 0) continue;
        if(city[x+dx[i]][y+dy[i]] == ch){
            dfs(x+dx[i], y+dy[i], ch);
        }
    }
}
int main(){
    cin >> h >> w;
    for(int i = 0; i < w; i++){
        for(int j = 0; j < h; j++)
            cin >> city[i][j];
    }
    for(int i = 0; i < w; i++){
        for(int j = 0; j < h; j++){
            if(city[i][j] == '-')
                dfs(i, j, '-'), emptyspace++;
            if(city[i][j] == 'G')
                dfs(i, j, 'G'), greenland++;
        }
    }
    cout << greenland << ' ' << emptyspace << endl;
}
```

在處理這種方格型的題目時，記得邊界的處理很重要！不然很容易因為戳到陣列外面造成各種無法預期的結果。上面的範例程式碼是用 continue 指令將超出邊界的點忽略掉。另外，還有一種方法，就是先在外圍都先預留一行代表 visited 的標誌，這樣就可以不用特別判邊界了。

習題 7.6.4: 迷宮問題 #1 (ZJ a982)

給你一個 $N \times N$ 格的迷宮，迷宮中以 # 代表障礙物，以 . 代表路，你固定在 $(2, 2)$ 出發，目的是抵達 $(n-1, n-1)$，求從起點走到終點的最短路徑長。

這是 BFS 的經典題。有了上一題處理方格的方法，相信這題一點都不難做。

習題 7.6.5: 三維迷宮問題 (TIOJ 1085)

給定一個立體 $(x \times y \times z)$ 的迷宮，某人自 $(1, 1, 1)$ 走至 (x, y, z)，請求出一條最短路徑，若有多組解，任一組都可。

這題是一個麻煩題，不只是要找到最短路徑長，還要把整個最短路徑輸出。

最短路徑問題的處理大家都已經不陌生了，一樣是用一個三維陣列 dist[][][] 儲存著從起點到那個點的最短路徑長。因為最後要將整條路徑輸出，所以除了記錄最短路徑長之外，還要順便記錄前一步是從哪裡來。轉移來源的儲存方式有很多種，可以記錄走來的方位（上下左右等等），也可以直接紀錄座標，兩種方式都能達到一樣的效果。有了轉移來源之後，就可以從終點沿路走回去，最後再倒轉輸出即可。

```
#include<bits/stdc++.h>
using namespace std;

int G[51][51][51];
int dist[51][51][51];
int pre[51][51][51];

// 六種轉移方式
int dx[] = {-1, 1, 0, 0, 0, 0};
int dy[] = { 0, 0,-1, 1, 0, 0};
int dz[] = { 0, 0, 0, 0,-1, 1};

struct point{ // 儲存點坐標
    int x, y, z;
    point(int x, int y, int z): x(x), y(y), z(z){}
};

int main(){
    // input
    int x, y, z;
```

```
21      cin >> z >> y >> x;
22      for(int i = 0; i < x; i++){
23          for(int j = 0; j < y; j++){
24              for(int k = 0; k < z; k++)
25                  cin >> G[i][j][k];
26          }
27      }
28      // BFS
29      queue<point> que;
30      que.push(point(0, 0, 0));
31      dist[0][0][0] = 1;
32      while(!que.empty()){
33          point now = que.front();
34          que.pop();
35          for(int i = 0; i < 6; i++){
36              point nxt(now.x+dx[i], now.y+dy[i], now.z+dz[i]);
37              if(nxt.x >= x || nxt.x < 0) continue; // 超出邊界
38              if(nxt.y >= y || nxt.y < 0) continue;
39              if(nxt.z >= z || nxt.z < 0) continue;
40              if(dist[nxt.x][nxt.y][nxt.z] != 0) continue; // visited
41              if(G[nxt.x][nxt.y][nxt.z] == 0){
42                  dist[nxt.x][nxt.y][nxt.z] = dist[now.x][now.y][now.z] +
    1;
43                  pre[nxt.x][nxt.y][nxt.z] = i; // 紀錄轉移來源是第 i 種
44                  que.push(nxt);
45              }
46          }
47      }
48      // back tracking
49      if(dist[x-1][y-1][z-1] == 0 || G[0][0][0] == 1){
50          // 注意 no route 的條件，容易漏判
51          cout << "no route" << endl;
52          return 0;
53      }
54      stack<point> ans; // 反向輸出，用 stack
55      x--, y--, z--; // 從終點 (x-1,y-1,z-1) 開始走
56      while(x || y || z){ // 直到走回起點 (0,0,0)
57          ans.push(point(x, y, z));
58          int i = pre[x][y][z];
59          x -= dx[i], y -= dy[i], z -= dz[i];
60      }
61      // output
62      cout << "(1,1,1)";
63      while(!ans.empty()){
64          point now = ans.top();
65          cout << "->(" << now.z+1 << ',' << now.y+1 << ',' << now.x+1 <<
    ")";
66          ans.pop();
67      }
```

```
68 }
```

跟著蕭電這樣做

還有一種方法可以不用紀錄轉移來源的方式可以找到最短路徑。當 `dist` 陣列被建好之後，你會發現對於每個點的所有相鄰點，至少有一個相鄰點的 `dist` 值與自己本身差 1，這個差 1 的點恰好就是轉移來源。所以可以在反向走回去時檢查哪一個 `dist` 值剛好與自己差 1，一樣可以走回起點。

再來是本章的最後一個習題，有時候我們在乎遍歷順序。

習題 7.6.6: 最短路線問題 (TIOJ 1572)

給一張無向不帶權圖以及固定的起點終點，求從起點走到終點的最短路徑長以及路徑本身。在本題中，輸出的最短路徑必須是字典序最小的那條，否則可能吃 WA。(頂點數 $\leq 10^6$)

這題要稍微想一下做法，有時候從起點 BFS 不是那麼有用。

圖！都是圖！ 8

8.1 有向無環圖

看到有向無環圖，你會想到什麼呢？沒錯，就是 dp！

聽說你喜歡 dp

複習一下 dp 是什麼。

跟著蕭電這樣做

dp 包含三個重要的部分：邊界條件、狀態以及轉移。如果能定義出良好的狀態，而且能夠找到一個合適的轉移順序，那就可以試著用 dp 解決這個問題。

dp 順序的問題在日常生活中或工作上也有一些應用。比如說你有一堆工作要做，可是做 A 工作之前要先完成 B 和 D，做 B 之前要完成 C，C 之前要解決 D 和 E 等等。這時候你就需要採用一個最佳的順序來完成這些工作項目。

我們可以將 dp 視覺化成為一張有向無環圖。每個節點是一個狀態（工作），而每條有向邊都是一個合理的轉移方式（需要先解決什麼才能開始做什麼），我們必須找到一個良好的順序，使得每一種狀態被計算前所有指向它的狀態都已經被計算過。

拓樸排序

為了解決上述的規劃問題，我們可以先將有向無環圖做「拓樸排序」，再按照順序做即可。這邊來看一個例題：

習題 8.1.1: 拓樸排序

給一張有向無環圖，所有點都是黑色，現在你需要把所有頂點塗白。不幸的，圖上所有的邊都是一條抹黑通道，只要被任何黑色點指向的點就無法塗白。你的任務是要輸出一個塗白的順序，使得所有頂點都有辦法被塗白。

要找出合理的排列順序的方法很像貪心法，首要目的就是得決定第一個被塗白的點！知道如何找出第一點，那麼就可以循序漸進的再找出第二點、第三點了。

可以作為第一點的點，想必它不必排在其他點後方塗色。也就是說，沒有被任何邊連向的點（也就是入度為零的點），就可以作為第一點。如果有很多個入度為零的點，那麼找哪一點都行。

因為第一點已經被塗白，不會對整張圖造成任何影響了，因此我們可以將其拔掉（直接移出圖外），所有它指向別人的邊也再也沒有效用，所以可以一併拔掉。拔掉之後剩下的圖又是一張所有點都是黑色的有向無環圖了，而且與先前被拔掉的所有點無關！所以我們可以遞迴求解，找到第二、第三、...，一直到只剩下一個點為止。

實作上可以用一個queue來記錄所有入度是 0 的點，每次都從queue的最前面取出一個點將其與它連出去的邊拔掉，萬一在拔掉的過程中有任何的頂點入度因此變成 0 了，那就將入度不幸歸零的點加進queue裡面。依照這個演算法做下去，最後拔點的順序就會是塗色的順序了！

```
int n = 頂點數;
vector<int> adj[MAX_N]; // adjacency lists
int inDegree[MAX_N];     // 記錄圖上每一個點目前的入度

void topological_sorting(){
    // 累計圖上每一個點的入度
    for(int i = 0; i < n; i++) inDegree[i] = 0;
    for(int i = 0; i < n; i++)
        for(auto j: adj[i])
            inDegree[j]++;

    // 宣告一個queue來記錄已經沒有被任何邊連向的點
    queue<int> que;
    for(int i = 0; i < n; i++)
        if(!inDegree[i]) que.push(i);

    // 開始找出一個合理的排列順序
    for(int i = 0; i < n; i++){
        // 尋找沒有被任何邊連向的點
        if(que.empty()) break; // 找不到，目前殘存的圖是個環
        int s = que.front(); que.pop();
        inDegree[s] = -1;       // 設為已找過（刪去s點）
        cout << s;              // 印出合理的排列順序的第i點

        // 更新inDegree的值（刪去由s點連出去的邊）
        for(auto j: adj[s]){
            inDegree[j]--;
            // 記錄已經沒有被任何邊連向的點
            if(!inDegree[j]) que.push(j);
```

```
30          }
31      }
32  }
```

8.2 二分圖

最近（筆者正在打這行字的大約一個月之前）的 CF div3 很喜歡出二分圖的問題。有時候，把一個複雜的問題轉換成二分圖之後就可以輕鬆解決了。

先來個定義

定義 8.2.1: 二分圖

如果圖 G 可以分成兩個互斥的點集 S 與 T，使得 S 與 T 內部都不存在兩點有邊連接（即圖 G 的所有邊都連接 S 上的頂點與 T 上的頂點），則稱圖 G 為二分圖。

那我們可以拿二分圖來幹嘛呢？答案是「塗色」。塗色是一個判斷一張圖是否為二分圖的一種方法，說穿了它就只是 DFS 而已。

習題 8.2.1: 二分塗色問題 (TIOJ 1209)

給定多張無向圖，對於每張圖，若該圖是二分圖，請輸出 Yes，否則輸出 No。$(1 \leq |V| \leq 40,000; 0 \leq |E| \leq 500,000)$

DFS 可以做很多事，除了遍歷之外，還可以順便計算關於每個節點的許多性質。以二分圖來說，這個性質就是要與鄰接的頂點不同「顏色」。為了滿足這個性質，我們可以在走到一個頂點時，都將其周圍的點都塗上與之相反的顏色，直到出現矛盾或者整張塗皆被完全上色。要注意的是輸入的圖不一定是連通圖，所以要對每個連通塊嘗試塗色才行。

```cpp
vector<int> adj[40010];
int color[40010];
int isbipartite;

void dfs(int s){
    for(auto i: adj[s]){
        // 將所有鄰接的點塗上與自己不同的顏色
        if(!color[i]) color[i] = -color[s], dfs(i);
        // 如果兩個相同顏色點鄰接，則不是二分圖
        if(color[i] == color[s]) isbipartite = 0;
    }
}

int main(){
    int n, m;
    while(cin >> n >> m, n){
```

```
17          // 初始化
18          isbipartite = 1;
19          for(int i = 1; i <= n; i++){
20              adj[i].clear(), color[i] = 0;
21          }
22          // 輸入
23          for(int i = 0; i < m; i++){
24              int from, to;
25              cin >> from >> to;
26              adj[from].push_back(to);
27              adj[to].push_back(from);
28          }
29          // dfs （二分圖不一定是連通圖！）
30          for(int i = 1; i <= n; i++){
31              if(!color[i]) color[i] = 1, dfs(i);
32          }
33          cout << (isbipartite? "Yes": "No") << endl;
34      }
35  }
```

習題

這是建中校內賽的簽到題，有了這題再加八分就有校隊。

習題 8.2.2: 分點問題（一）(TIOJ 2062)

給一張無向圖，若此圖為二分圖，則輸出此圖分成的兩個部分的頂點數及編號；若不是，則輸出 -1。($|V| \leq 10^6, |E| \leq 2 \times 10^6$)

習題 8.2.3: 二分圖測試 (TIOJ 1209)

給你一個圖 (Graph)，請問這個圖是否為一個二分圖 (bipartite graph)？

解法就是塗色看看會不會出先矛盾（例如將紅色點再塗藍）。

習題 8.2.4: CF 1176E, CF 1114F

就是節首提到的 CF 題，都是二分圖塗色問題。

8.3 樹

什麼是樹，可以種嗎？

樹是一種特別的圖，它長的就像一顆樹。樹必須沒有環，而且要連通。樹有順序性（也就是它可以拓樸排序），所以可以在樹上做一些能在序列上做的事（例如 dp、開線段樹等等）。

定義 8.3.1: 樹 (tree)

一棵樹是一個無向圖，必須滿足以下其中之一：

1. 任意兩個頂點間存在唯一一條路徑。
2. 沒有迴路且連通。
3. 邊數比頂點數少一的簡單圖。

上述三點可以被證明是等價的。

接下來是一些有關樹的名詞：

1. 樹 (tree)：沒有迴路的連通圖。
2. 根 (root)：有些題目會指定一個點當根，變成有根樹 (rooted tree)，根節點只有子節點沒有父節點。
3. 父節點 (parent)：根節點必定是其鄰接節點的父節點，若 v 是 u 的子節點，則 u 是 v 的父節點。
4. 子節點 (children)：除了父節點之外的其他鄰接節點。
5. 兄弟節點 (sibling)：父節點是同一節點的節點互為兄弟節點。
6. 子樹 (subtree)：子節點以及其子樹構成的樹。
7. 森林 (forest)：許多不連通的樹構成的集合。
8. 生成樹 (spanning tree)：包含一張連通圖的所有頂點以及某些邊的樹，稱為此圖的生成樹。

來爬樹 XD

尋訪一棵樹可以用 DFS 來實現，跟普通的 DFS 沒什麼兩樣。不過樹上的每個點都只有一個父節點，所以也可以用下列實作方法：

```
void dfs(int s, int par){
    // process node s
    for(auto chi: adj[s]){
        if(chi != par)   dfs(chi, s);
    }
}
```

利用變數 `par` 紀錄父節點的方式，比免回頭走就行了，不須紀錄 `visited` 陣列。根節點沒有父節點，因此一開始呼叫這個函數的時候 `par` 參數要傳入非任何節點的編號（例如 -1）。

距離

有時候我們需要這個節點距離樹根多遠，這時候可以慢慢往上爬到樹根（對，圖論的樹根在上面，不知道為什麼），但是這樣最差可能會需要用到 $O(n)$ 時間。所以可以進行一次 DFS 預處理，後來就只要 $O(1)$ 查詢。

```
void dfs(int s, int par){
    for(auto chi: adj[s]){
        if(chi != par) dist[chi] = dist[s]+1, dfs(chi, s);
    }
}
```

如果先將 dist[根] 初始化成 0，再對根節點 DFS 一次，dist 陣列就會儲存著每個節點到根的距離。

樹 dp

尋訪樹的時候當然就是要在上面偷偷做事嘍，最常見的就是 dp。

```
void dfs(int s, int par){
    siz[s] = 1;
    for(auto chi: adj[s]){
        if(chi != par) dfs(chi, s), siz[s] += siz[chi];
    }
}
```

這個程式可以計算出以每個點為根的子樹大小，紀錄在 size[] 陣列裡。

樹直徑

回顧一下樹的性質，樹上任一個頂點對（兩相異頂點）都只存在唯一路徑。有時候我們在乎這些路徑的最大距離，這個距離稱為直徑 (diameter)。

那要怎麼找到這個直徑呢？首先隨便找一個點當根，做一次 DFS 找到距離這個根最遠的點。這個點必定是直徑的一個端點，所以就從這個最遠點再做一次 DFS，再找到距離最遠點最遠的點，連接這兩個點的路徑就是直徑。

```
int diameter(){
    int root = 0, max_dist = 0;
    int point1 = 0, point2 = 0;
    dfs(root, -1);
    for(int i = 0; i < N; i++)
        if(dist[i] > max_dist){
            max_dist = dist[i], point1 = i;
        }
    max_dist = 0;
    dfs(point1, -1);
    for(int i = 0; i < N)
        if(dist[i] > max_dist){
            max_dist = dist[i], point2 = i;
```

```
14          }
15      // point1, point2 是直徑兩端點
16      return max_dist;
17  }
```

這裡省略了 DFS 的部分以及 dist 陣列，這兩個部分詳見找距離的段落。

樹重心

一棵樹的重心是樹上的一個點，如果將這點當作整棵樹的根，就可以使根節點最大的子樹最小。我們一樣可以用 DFS 找出重心。

```
1  int centroid;
2  void find_centroid(int s, int par){
3      bool fnd = false;
4      for(auto chi: adj[s]){
5          if(chi != par && siz[chi] > N/2)
6              find_centroid(chi, s), fnd = true;
7      }
8      if(!fnd) centroid = s;
9  }
```

這邊要用到siz[]陣列，可以用前面提到的 DFS 方法找出。

8.4 其他不同的圖

以假亂真

很多看起來不是圖論的題目事實上是圖論題目呦 XD

習題 8.4.1: 塗色問題

給一個 $n \times n$ 的方格，裡面有許多黑色與白色的格子，你每次可以將一直欄或一橫列塗成白色，問最少要塗幾次才能將所有格子塗成白色。$(n \leq 50)$

故弄玄虛

很多看起來是圖論的題目事實上不是圖論題目呦 XD

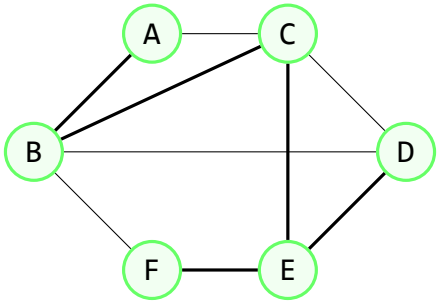
習題 8.4.2: 氣泡排序圖 (CF 340D)

給一個 1~n 的排列，Iahub 要將這個序列按照題序給定的方法做氣泡排序(跟大部分人的氣泡排序幾乎一樣)。每次兩個數字交換時，他會將編號為這兩個數字的頂點連上一條邊。求氣泡排序結束之後，最大點獨立集合的點數。$(n \leq 10^5)$

最小生成樹 (Minimum Spanning Tree) 9

9.1 生成樹（Spanning Tree）

給定一張連通圖 G，若 G 的子圖 T 是一棵樹，並且包含 G 的所有頂點，我們說 T 是 G 的生成樹。T 同時也是 G 最少邊數的子圖，使得所有頂點之間連通。T 理所當然會有所有一棵樹該有的性質，由於通常維護樹上資料比維護圖上資料結構簡單，處理一張圖的問題時，我們可能會以一棵生成樹代表之，如下圖所示。



Spanning Tree

注意若所有點不連通，則生成樹不會存在。

9.2 最小生成樹（MST, Minimum Spanning Tree）

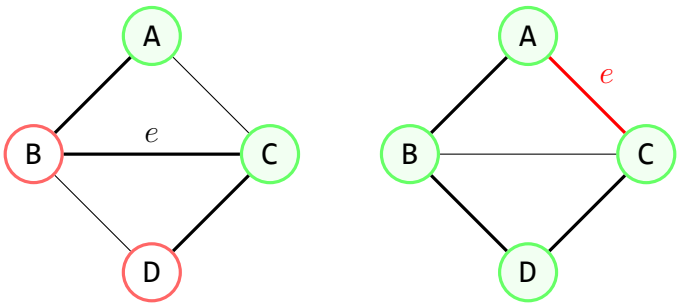
最小生成樹是最小權重生成樹的簡稱，也就是所有生成樹中邊權總和最小的。最小生成樹的形狀不一定唯一，但其邊權和是固定的。最小生成樹有以下一些性質，我們可以利用貪心法求出最小生成樹。

定理 9.2.1: Cut Property

將 G 的頂點集合分成兩個頂點集合 S、$V-S$，設連結兩個頂點集合的邊集為 E_{cut}，其中最小的邊為 e，則必定存在一個包含 e 的最小生成樹。

證明：設所有 MST 均不包含 e，在任一最小生成樹 T 中加上 e 後必會形成一環，除了 e 之外該環上至少有一條屬於 E_{cut} 的邊（否則 S、$V-S$ 不連通），我

們以 e 替換這條邊能夠得到權重不小於 T 且包含 e 的生成樹，假設矛盾。



Cut Property　　Cycle Property

定理 9.2.2: Cycle Property

對於每一個環 C 其上最大的邊 e，必定有不包含它的最小生成樹（也就是不選擇它不會影響 MST 的解答）。

證明：設所有 MST 都包含 e，去除了 e 之後它們都會變為兩棵子樹，而環上有另一不比 e 大的邊可以用來連接兩棵子樹，便構成了權重不小於原本的 MST，並且不包含 e 的生成樹，假設矛盾。

本章介紹的所有找最小生成樹的演算法都是屬於利用了 Cut Property 的 Greedy method。

9.3 Prim's Algorithm

概念

我們可以將 Cut Property 中的 S 視為執行到目前已經確定的 MST 點集，而不斷的以 E_{cut} 中最小的邊擴增 S 的大小，這是 Prim 的主要思想。Prim 和 Dijkstra 演算法的架構相當類似，我們以 V 代表原本的點集，E 代表原本的邊集 V_{new} 代表 MST 中的點集，E_{new} 代表 MST 中的邊集；以下是 Prim 的執行步驟：

1. 初始化：$V_{new} = \{x\}$，其中 x 為任一起始點，$E_{new} = \{\}$。
2. 重複下列操作，直到 $V_{new} = V$：
   a) 選取權值最小的邊 (u, v) 使得 $u \in V_{new}$，而 $v \notin V_{new}$（如果存在多條，則可任選），v 同時也可以說是距離目前的 MST 最近的頂點。
   b) 將 v 加入集合 V_{new} 中，將 (u, v) 加入集合 E_{new} 中。
3. V_{new} 和 E_{new} 即是最後的 MST！

證明

考慮 Cut Property，對於有 n 個頂點的圖 G_n，某個起點 x 其最近的鄰居若是 y，則邊 (x, y) 必定會屬於 MST，之後我們可以將 x, y 看成同一點 z，並以圖 G_{n-1} 代表此剩下 $n-1$ 個點的新圖（原圖連向 x 或 y 的邊均連到 z），我們利用數學歸納法能夠好好的確認 Prim 的正確性。

實作

從上面的流程中需要不斷的選取邊權最小的邊，在 V_{new} 加入新節點時又要不斷插入新邊權。這樣有效支援插入數字以及取最小值的資料結構，不難想到可以用priority_queue來幫助我們。由於概念都很簡單，最難的就是證明，所以我們先看程式碼吧！

```
#include "bits/stdc++.h"
typedef pair<int,int> pii;
vector<pii> g[MAXV]; // adjacency list {weight, to}
int prim(int n){
    int sum=0, v=0LL; // 權重和、已選取頂點數
    bool inMST[MAXV]={};
    // heap 中的 pair 表示
    // {該頂點與MST的最短距離, 不在目前MST中的頂點編號}
    priority_queue<pii,vector<pii>,greater<pii> > pq;
    pq.push({0,0});
    while(v<n && pq.size()){
        pii cur=pq.top(); pq.pop();
        // 如果拿出來的最近頂點已經在MST中則跳過
        if(inMST[t.second]) continue;
        inMST[t.second]=true;
        sum+=t.first, v++;
        for(auto &e:g[t.second]) {
            if(!inMST[e.second]) pq.push(e);
        }
    }
    return sum;
}
```

由於我們至多存取 heap $|V|+|E|$ 次，Prim 演算法的總時間複雜度將會是 $O((|V|+|E|)\log|V|)$，如果用費波納契堆還能進一步優化到 $O(|E|+|V|\log|V|)$。

9.4 Kruskal's Algorithm

這個演算法較不複雜，應該是最常被使用的 MST 算法，可以好好看一下。

概念

Kruskal 演算法是以邊為主角，以下為 Kruskal 的流程：

1. 將所有邊 (u_i, v_i) 按照邊權sort
2. 初始化，將所有點視為獨立的連通塊
3. 由小到大檢查所有邊 (u, v)，若 u 與 v 互不連通，則將這條邊加入 MST 中，並合併 u, v 所在的連通塊，若相連通則略過。
4. 重複前一步驟，直到所有點都相連通。

證明

和 Prim 類似，Kruskal 每次會選取 G_n 中權重最小且連接不同連通塊的邊 (x, y)，此時能夠將 x 及 y 看成同一點，得到 G_{n-1}，利用數學歸納法同樣可以得到證明。

實作

上面的流程中提到要在新建的 MST 中，檢查任兩個點有沒有相連。當然最直覺的做法是每次都 DFS 看兩個點有沒有相連，但這個方法很明顯太慢了。然而你會發現，事實上我們在意的其實就是兩個點所屬的連通塊是否相同。我們可以想到用 Disjoint Sets 的資料結構維護，畢竟程式碼結構真的很簡單，所以直接看 code 吧！

```cpp
struct edge{
    int u,v,w;
};
bool operator<(edge a, edge b){return a.w<b.w;}
vector<edge> edges;
int pa[MAXV],sz[MAXV]; // 大家還記得 dsu 怎麼寫嗎？
void init(int n) {
    for(int i = 0; i < n; i++) pa[i] = i, sz[i] = 1;
}
int anc(int x){
    return x==pa[x] ? x : (pa[x]=anc(pa[x]));
}
bool same(int x,int y){
    return x=anc(x), y=anc(y), x==y;
}
void join(int x,int y){
    if((x=anc(x)) == (y=anc(y))) return;
    if(sz[x] < sz[y]) swap(x, y);
    pa[y] = x, sz[x] += sz[y];
}
int kruskal(int n){
    int CC = n, sum = 0; // 連通塊數、權重和
    init(n); // 初始化 dsu
    sort(edges.begin(),edges.end()); // 按邊權 sort
    for(auto &e:edges) { // 邊權由小到大檢查
        if(!same(e.u,e.v)) {//兩個點若不連通，則加入 MST
            join(e.u,e.v);
            sum += e.w;
        }
    }
    return sum;
}
```

Kruskal 主要的時間花費在排序邊 $O(|E|\log|E|)$，排序之後的合併只需要 $O(|E| \cdot \alpha(|E|, |V|)$ 即能完成，故 Kruskal 的總時間複雜度為 $O(|E|\log|E|)$。

9.5 Borůvka's Algorithm

又名 Sollin 演算法，它其實是最早被發明的 MST 多項式時間複雜度演算法，不過卻有點像是 Prim 和 Kruskal 的混合版，似乎很少人在競賽中使用這個演算法。

概念與實作

首先，一開始所有頂點都被設為是獨立的連通塊。對於每個連通塊，找出其連到其他連通塊的邊之中最短的邊（可以 $O(|E|)$ 掃過一遍），把所有連通塊對應的邊連上（$O(|V|)$），並對新的連通塊們重複執行這個步驟，直到只剩下一個連通塊。

證明

同樣由 Cut Property 可以知道每個連通塊 S 向外連的最短邊 e 一定屬於 MST，對點數強數歸能夠得知 Borůvka 的正確性。

```cpp
vector<edge> edges; // edge 同前面的宣告
int boruvka(int n){
    int CC = n, sum = 0; // 連通塊數、權重和
    edge cheapest[MAXV] = {}; // 連通塊向外連最短的邊
    init(n); // 使用並查集
    while(CC != 1){
        for(int i = 0; i < n; i++) cheapest[i].w = 1e9;
        for(auto &e:edges){
            int fu = anc(e.u), fv = anc(e.v);
            if(fu == fv) continue;
            // 找到每個連通塊往外最短的邊
            if(e < cheapest[fu]) cheapest[fu] = e;
            if(e < cheapest[fv]) cheapest[fv] = e;
        }
        for(int i = 0; i < n; i++) {
            if(i != anc(i)) continue;
            auto &e = cheapest[i];
            // 嘗試將每個連通塊以最短邊往外連
            if(!same(e.u,e.v)) {
                join(e.u,e.v);
                sum += e.w, --CC;
            }
        }
    }
    return sum;
}
```

可以注意到，每一輪操作中每一個連通塊都會和其他連通塊合併，也就是總連通塊數至少會減少為原先的一半，因此最多需要執行 $O(\log |V|)$ 輪合併操作，總複雜度 $O((|V|+|E|)\log |V|)$（森林結構維持在最佳狀態，DSU 操作的總時間複雜度，從 $O(|E|\cdot\alpha(|E|,|V|))$ 下降至 $O(|E|)$ —演算法筆記）。

據某些大陸人說，Borůvka 有時靈活性比 Prim 和 Kruskal 都好，因不須將頂點或邊直接比較，但筆者目前還沒找到必須要用 Borůvka 才能解決的題目；不過值得一提的是，利用 Borůvka 的想法能夠進一步得到隨機期望複雜度線性（$O(|V|+|E|)$）的最小生成樹做法，在此暫不贅述。

9.6 例題

習題的唷

習題 9.6.1: 圖論之最小生成樹（TIOJ 1211）

給你一個加權的無向圖 (weighted undirected graph)，請問這個圖的最小生成樹 (minimum spanning tree) 的權重和為多少？
($|V| \le 10^5, |E| \le 10^6, 1 \le w_i \le 1000$)

習題 9.6.2: 咕嚕咕嚕呱啦呱啦（TIOJ 1795）

給定 N 個點 M 條邊，以及所有邊的邊權重，是否有辦法建構出一顆生成樹之權重總和剛好為 K？另外，任意一條邊的權重只有可能為 0 or 1。
($N \le 10^5, M \le 3 \times 10^5$)

習題 9.6.3: 蓋捷運（OJ 71）

給定一張圖，每條邊上有兩個權值 X, Y，求所有生成樹 T 中下述比率的最大值。

$$\frac{\sum_{e \in T} e_X}{\sum_{e \in T} e_Y}$$

($n, m \le 2 \times 10^5$，$1 \le x, y \le 10^9$)

習題 9.6.4: 機器人組裝大賽（TIOJ 1445）

給定一張圖，請輸出其最小生成樹的權重以及所有生成樹中權重和不嚴格第二小的權重和。
($|V| \le 1000, |E| \le \frac{|V|(|V|-1)}{2}, w_i \ge 0$，保證答案在`long long`內)

最短路徑 (Shortest Path) 10

習題 10.0.1: 不定向邊 (TIOJ 1290)

給一個圖，單向邊，邊有 cost，一開始邊是未決定方向的，你可以對任一條邊決定一個方向。接著，問起點到終點最短路徑長。如果到不了的話請輸出"He is very hot"(不含雙引號)。
輸入包含多組測試資料。
$1 \leq |V| \leq 1000, 1 \leq |E| \leq |V|^2$

給定一張圖，邊上面有邊權，**最短路徑**是由起點到終點、經過邊權的和最小的路徑，可能有許多條，也可能不存在。最短路徑不見得是邊最少、點最少的路徑。以下將介紹數種最短路徑的演算法。

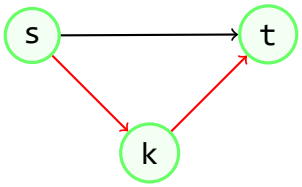
若圖上有負環，最短路徑無法定義；而求取最短簡單路徑 (不經過重複點的路徑) 也會變為 NP-complete 問題，因為一個無權圖的哈密頓路徑可以在多項式時間歸約成每條邊權都是 -1 的最短簡單路徑。(可以想一想！)

10.1 BFS

奇怪，不是感覺之前在圖上找最短路徑就是直接用 BFS $O(|V| + |E|)$ 解決嗎？不行不行！！要注意的是，BFS 只能套用在邊權都等於一的情況下，而普通的圖邊權不一定為一，BFS 就拿它沒轍了。不過，其實待會介紹的各種演算法有些也跟 BFS 的概念有些相近喔！

10.2 Relaxation

先介紹一個接下來每個演算法都會用到的概念－－鬆弛。若目前找到的路徑上有一部分可以用其他更短的路徑代替，我們當然可以用較短的那段代替較長的路徑，整個路徑也會因此而變得更短。我們把這個動作叫做**鬆弛 (Relax)**。



如上圖所示，由 s 經過 k 到 t 的路徑可以用直接從 s 到 t 的路徑代替。

10.3 單點源最短路徑

單點源最短路徑問題，即是求出一個圖當中，一個固定的起點到各點的最短路徑。

Bellman-Ford

直接依照鬆弛的定義來鬆弛看看吧！暴力地每一回合將所有邊都鬆弛一次！總共做 $|V|-1$ 次，時間複雜度為 $O(|V||E|)$。

為什麼是 $|V|-1$ 次呢？因為圖上沒有負環，所以最短路徑一定是簡單路徑，否則可以在經過同一點時從有環的路徑上移除環 (一定是正環，路徑會變短)，而這條路徑最多經過 $|V|$ 個點，所以經過的邊數最多是 $|V|-1$；*** 每次鬆弛所有邊實際上是嘗試用比前一輪多一條的邊代替最短路徑，有點類似 BFS，第 n 次鬆弛操作保證了所有使用 n 個邊的路徑最短，*** 因此鬆弛 $|V|-1$ 回合後，就可以找到從起點到所有點的最短距離。這同時也告訴我們如果在第 $|V|$ 回合仍有邊可以鬆弛，則代表圖上有負環。

大家或許覺得這個複雜度有點差，但這個演算法可以處理有負邊權 (無負環) 的最短簡單路徑，而且還可以找出圖中有沒有負環，所以不要小看它喔！

```
#define ff first // 小寫字母教 OwO
#define ss second
typedef pair<int,int> pii;
vector<pii> g[N+1]; // adjacency list
int dis[N+1], v, relax;
void BellmanFord(int src){
    dis[src]=0;
    // 重複 Relax V-1次，第V次仍有鬆弛則表示有負環
    for(int r = 0;r < v;r++){
        relax = false;
        for(int i = 0;i < v;i++) // O(E)
            for(auto e:g[i])
                if(dis[e.ss]>dis[i]+e.ff) // Relax
                    dis[e.ss]=dis[i]+e.ff, relax=true;
        if(!relax) return;
    }
    // 有負環
}
```

SPFA - 猜猜它的全名

這個算法其實就是剛剛那個大家覺得很暴力的 Bellman-Ford 演算法的改進，他就像從皮丘進化成了皮卡丘，電度上升了不只一倍。其實它也沒有那麼難啦！他的做法就是從 Bellman-Ford 的每次鬆弛所有邊，變成每次只鬆弛特定的邊。需要鬆弛的邊只有前一輪更新過端點的邊，我們可以用一個 queue 來記錄哪些節點旁邊的邊可以拿來鬆弛其他節點。

SPFA 有許多奇奇怪怪的優化方式，但在這邊先不贅述太多；有一個優化就是在要把點塞入 queue 裡面的時候，先看它是否已經在 queue 裡面，如果該點已經在 queue 裡面的話，就不用再把那個點放進去一次囉！

```
vector<pii> g[N+1]; // adjacency list
int dis[N+1], inQ[N+1], v, cur;
void SPFA(int src){
    queue<int> q;
    fill(dis,dis+v,INF), fill(inQ,inQ+v,0);
    q.push(src), dis[src] = 0, inQ[src] = 1;
    while(!q.empty()){ // Repeat Relaxing
        cur = q.front(), q.pop(), inQ[cur] = 0;
        for(auto e:g[cur])
            if(dis[e.ss] > dis[cur]+e.ff){
                dis[e.ss] = dis[cur]+e.ff;
                if(!inQ[e.ss])
                    inQ[e.ss]=1, q.push(e.ss);
            }
    }
}
```

那 SPFA 的複雜度到底是多少呢？不瞞各位所說，SPFA 的平均複雜度還沒有被提出嚴謹的證明，而它的最壞複雜度亦為 $O(|V||E|)$，與 Bellman-Ford 並無二致，不過在隨機圖的試驗中，它的平均運行時間是 $2|E|$ 左右。意想不到的快，對吧？

Dijkstra

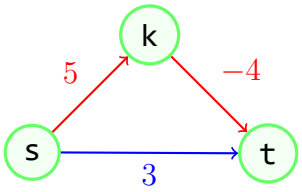
> Dijkstra 有人念 Dijkstra，有人則念 Dijkstra，而且還有念 Dijkstra 的人。

我們可以把 BFS 的搜尋順序想成是一層一層向外擴張的同心圓，更新距離為 1、距離為 $2\cdots$ 距離為 k 的點。Dijkstra 可以想成是加上權重的 BFS，由小到大以不同的半徑更新節點 (半徑最小的同心圓)，每次找到距起點最近的未拜訪節點，並嘗試更新其周圍的節點。

從另一觀點看，若圖上沒有負邊，最短路徑的前綴一定是最短路徑，因此每次選擇一最近節點加入**最短路徑樹**，並嘗試鬆弛周遭節點，是一種 Greedy 算法，和 Prim 演算法 (某種最小生成樹演算法) 有異曲同工之妙。

什麼是最短路徑樹呢？就是把從起點到每一個點的最短路徑畫出來，形成的一棵以起點為根的樹。因為起點到每個點的最短簡單路徑都只有一條，若有兩條以上就可以把一條拔掉，所以最短路徑們會形成一棵樹。

要記得注意的是，這個演算法只能套用在沒有負邊權的情況下，否則可能會發生以下這種狀況：



假設 s 是起點，t 是終點，一般的 Dijkstra 算法會先入為主地認為直接走到 t 比較快，但其實先走到看似比較遠的 k，到最後可能會有比較短的路徑！

```
vector<pii> graph[N+1];// adjacency list
int dis[N+1] = {}, v, e;
void Dijkstra(int src){
    //期望節點清單，依距起點距離排序 {dist,vertex}
    priority_queue<pii,vector<pii>,greater<pii> > pq;
    pii cur;
    fill(dis,dis+v,INF);
    dis[src] = 0;
    pq.push({0,src});
    //每個節點只會被更新一次
    for(int i = 0;i < v;i++){
        //將已更新的節點從清單移除(Lazy Deletion)
        do cur = pq.top(), pq.pop();
        while(cur.ff>dis[cur.ss]);
        for(auto e:graph[cur.ss])// Relax
            if(dis[e.ss] > cur.ff+e.ff)
                dis[e.ss] = cur.ff+e.ff,
            pq.push({dis[e.ss],e.ss});
    }
}
```

利用 STL 的 priority_queue 實作找出和起點距離最小節點的部分，時間複雜度 $O(|E|\log|V|)$。注意中間有 Lazy Deletion 的技巧，Dijkstra 中已經更新過的節點必須從 heap 裡面移除，不過 priority_queue 不支援刪除任意數字的操作，因此我們在 pop 出來的時候才跳過這些節點。如果用費波納契堆實作可以達到 $O(|E|+|V|\log|V|)$ 的時間複雜度，但幾乎不太可能手刻出來。

10.4 全點對最短路徑

有時候我們會想要求每一個點到每一個點間的最短距離，我們當然可以對每個頂點都做一次 Bellman-Ford 或 Dijkstra，但這樣複雜度分別是 $O(|V|^2|E|)$ 和 $O(|E||V|\log|V|)$，還有優化的空間。

Floyd Warshall

Floyd Warshall 是利用 DP 的想法，只需要 $O(|V|^3)$ 即能求出所有點對之間的距離。DP 式大概長這樣:

$$dp[i][j][1] = w[i][j]$$

$$dp[i][j][k] = \min(dp[i][j][k-1], dp[i][k][k-1] + dp[k][j][k-1])$$

其中 $dp[i][j][k]$ 表示從 i 到 j，且中途只用到前 k 個點當中繼點的最短距離，也就是說每次看 k 能不能替換 i 到 j 的路徑中一部份，逐步開放第 $1, 2 \cdots k$ 個點。順帶一提用鄰接矩陣就可以直接枚舉 i, j 就好，而且照著 k 遞增的順序 dp 可以滾動。Code 超短，常數也超低，所以如果測資很鬆可以偶爾拿來做單點源最短路徑。

```
// adjacency matrix
int dis[N+1][N+1], v;
void FloydWarshall{
    for(int k = 0;k < v;k++)
        for(int i = 0;i < v;i++)
            for(int j = 0;j < v;j++)
                if(dis[i][j] > dis[i][k]+dis[k][j])
                    dis[i][j] = dis[i][k]+dis[k][j];
}
```

習題們

習題的唷

習題 10.4.1: 乳酪的誘惑 (ZJ d273)

給定 $n \times m$ 方格上的障礙物，求起點走到終點的最短路徑長。$n, m \leq 1000$。

習題 10.4.2: 地道問題 (TIOJ 1509)

給定一張有向圖，問你編號一的點到所有其他點的最短距離和加上所有點到編號一的點的最短距離和。$(|V|, |E| \leq 10^6)$

習題 10.4.3: 貨物運送計畫 (TIOJ 1641)

邊權乘積最小路徑 $(|V| \leq 10^4, |E| \leq 2 \times 10^2)$。

習題 10.4.4: 阿思歐思 (TIOJ 1685)

路徑上點權 max 最小路徑 $(|V| \leq 200, |E| \leq 10^4)$ 最多有 10^4 次詢問。

習題 10.4.5: Wormholes(UVa 558)

判斷一張有向圖中是否有負環。$(|V| \leq 1000, |E| \leq 2000)$

習題 10.4.6: Going in Cycle!!(UVa 11090)

給定一張有向圖，找出一個環，使得環上邊權的平均最小。($|V| \leq 50$, 邊權 $\leq 10^7$)

習題 10.4.7: 圖論專家 (ZJ d243)

給定一張無向圖，求起點到終點的嚴格第 k 短路徑長。

習題 10.4.8: 成為神奇寶貝大師！(競程日記 MCC #6 pE)

https://oj.icpc.tw/c/36/E (題敘略)

LCA 和他們的產地 11

11.1 前言

LCA（Lowest Common Ancestor），最低共同祖先，到底是甚麼呢？當你把一堆生物的親緣關係樹畫出來的時候，你會很想要知道兩種生物的親緣關係接不接近，而最直接的想法就是找出他們的共同祖先嘛！但是他們可能有很多共同祖先，要找哪個好呢？如果找的是很早很早的祖先，那所有的生物都是他的後代，意義並不大，要找的話就是找出他們最低的共同祖先，也就是 LCA，才會最有意義。我們可以說，LCA 越低的兩個生物，親緣關係越近。

那 LCA 在 CP 中有甚麼用武之地呢 ?? 嗯，他跟上面所舉的例子一樣，專門用來處理有關於樹的題目，但處理樹的工具千奇百怪，包括樹鍊剖分啦、重心剖分啦、樹分治、LCT（Link Cut Tree）啦、ETT（Euler Tour Tree）啦、樹壓平啦，LCA 可以說是比較基礎的一個主題。而 LCA 的作法也是各式各樣，接下來都會介紹，而且每個都有每個的好處，千萬不要因為會了其中一個，就對其他的不屑一顧喔！

11.2 暴力做 LCA

如果要暴力去做 LCA，相信大家都會：每次詢問兩個節點 A 和 B 的 LCA 時，紀錄從根到 A、B 的路徑成兩個序列，然後找到最前面的節點使得序列不同，這個的前一個就是了（如果都一樣就是根了）。這樣子每次詢問 DFS 兩次 $O(N)$，然後找尋 LCA，$O(N)$ 掃過去即可。

另外一個想法就是：DFS 一次紀錄每一個點的父節點和深度，然後每次詢問的時候，不斷把深度小的（一樣就隨便）往上走一次，直到走到一樣為止，這樣複雜度進步為 $O(N)$。每次處理都紀錄所有算到的東東就可以大大大減少其複雜度，具體的實作方法：

$$\mathrm{LCA}(x,y)=\begin{cases} x, & \text{for } x=y \\ x, & \text{for } \mathrm{parent}[y]=x \\ y, & \text{for } \mathrm{parent}[x]=y \\ \mathrm{LCA}(\mathrm{parent}[y],\,x), & \text{for } \mathrm{depth}[x]\le\mathrm{depth}[y] \\ \mathrm{LCA}(\mathrm{parent}[x],\,y), & \text{for } \mathrm{depth}[y]<\mathrm{depth}[x] \end{cases}$$

當然，暴力永遠（大部分的時候）不是答案，所以在這裡要介紹各種常見找 LCA 的方法。

11.3 樹壓平取 LCA

RMQ 問題

在講 LCA 之前，首先看個區間的問題：

習題 11.3.1: RMQ（區間極值）

給定 N 個數字 $a_1, a_2, a_3, \cdots, a_N$，和 Q 個詢問 $[l_i, r_i]$，請對於每一個詢問輸出 $[l_i, r_i]$ 內的最大值。

相信大家都有辦法在不裝弱的前提下容易的寫出 $O(NQ)$ 的解答（寫不出來可能要去面壁思過了），所以通常題目也不會出那麼簡單。這裏具體實作方法不寫（請洽其他章節！），不過用 Sparse Table 或線段樹等資料結構可以變成 $O(N + Q \log N)$ 或 $O(N \log N + Q)$。

歐拉遍歷（Euler Tour）

要如何把一個樹變成一個序列呢？一個簡單的想法就是 DFS，然後遇到每一個點進去和離開的時候都push_back進去。這個序列會有 $1 + 2(N-1) = 2N - 1$（根先進去，然後每一個邊都加兩次）個元素，並且紀錄每一個點進來和出去的時間戳。這個有什麼性質呢？可以發現，如果 u 是 v 的祖先，則 u 會先放進去，然後 v 進來出來了，u 才會放第二次。除此之外，還需要紀錄一些額外的東西，待會一一介紹。所以呢，**對於一個點，其所有子節點的子樹在序列的位置會被包在那個點在序列中的位置兩邊。**如果感覺有一點抽象，那就看個例子吧！(Source: GeeksForGeeks)

會需要維護四個陣列：

1. 在序列裡面每一個節點的深度 $\mathrm{dep}[x]$
2. 原本的歐拉遍歷序列 $\mathrm{S}[x]$
3. 每一個節點在序列中第一次出現的位置 $\mathrm{first}[x]$（$= \arg\min_i(\mathrm{S}[i] = x)$）
4. 在遍歷序列中，每一個節點所對應到的深度序列 $\mathrm{SDep}[x]$（$\mathrm{SDep}[i] = [\mathrm{S}[i]]$）

當每次要詢問 a 和 b 的 LCA 時，找到 SDep 內的區間 $(\mathrm{first}[a], \mathrm{first}[b])$ 的最小深度所在，假設在 x，則 $\mathrm{S}[x]$ 為答案。用以上的圖解釋，如果想要找 4 和 9 的 LCA，則發現 $\mathrm{first}[3] = 2$（第一次出現在位置為 2 的地方，在 1，2 前面）、$\mathrm{first}[9] = 7$。則可以發現，在 SDep 中（圖中下面的序列），位置為 3 的有最小深度 1，所以回去看 S 中位置為 3 的是誰，發現是 2（1，2，4，**2**），所以回傳 2。

為什麼這樣會對？跑的夠快嗎？

當想要計算 a 和 b（$\text{first}[a] < \text{first}[b]$）的 LCA 時，從 $\text{S}[\text{first}[a], \cdots, \text{first}[b]]$ 的序列是進入 a 了，然後出去 b 了，然後往上遞迴，每到他的一個祖先就會遞迴到那個祖先的所有子樹，直到進入了 b 為止。顯然，一旦回溯到了 $\text{LCA}(a, b)$，即會遞迴到 b，而到了區間的結尾。所以呢，$\text{LCA}(a, b)$ 在區間裡面，而且它一定是深度最下面的那一個。

至於複雜度，第一次 DFS 的複雜度是 $O(V)$，而如果用線段樹或 Sparse Table 的話，複雜度差不多是 $O(V \log V)$。

11.4 Doubling 倍增法

Doubling，顧名思義，就是倍增法，聽到倍增法，就可以知道跟二的冪次有關。

Doubling 可以說是 LCA 的各種做法中最受高中競賽選手歡迎的一種做法。它的好處是 code 短好刻，又可以在求 LCA 的時候順便紀錄某一些性質，以便做到某些神奇的事。

話不多說，就直接講做法囉！首先介紹預處理的部分。預處理非常的簡單明瞭，只要記錄兩個簡單的東西：節點的深度，還有每個節點的第 2^k 輩祖先。點的深度相信大家都會，我也不多費唇舌了。至於每個節點的第 2^k 輩祖先，則可以用以下的 DP 做法簡單求得：

```
anc[root][0] = -1; //根的祖先不存在！
anc[x][0] = pa[0];//第一輩祖先就是父節點
for(int i = 0;i + 1 < MaxLog; i++)
{
    anc[x][i+1] = anc[anc[x][i]][i]; //anc[i][j]表示第i個節點的第$2^j $輩祖
}
```

這樣子看來，預處理的複雜度是 $O(N \log N)$（N 是節點數）。

那詢問要如何回答呢？

第一步，想辦法讓兩個要求 LCA 的節點等高。具體做法就是找出兩者的深度差（稱為 Δh），並讓較低的節點慢慢上升（每次提升 Δh 的二進位中為一的位數所代表的二的冪次，也就是每次都盡量移動多），複雜度 $O(\log N)$。Code 長這樣：

```
if(dep[a] > dep[b])
    swap(a,b);
int k  = 0;
for(int i = dep[b]-dep[a]; i > 0; i/=2)
{
    if(i % 2 == 1)
        b = anc[b][k];
    k++;
}
```

接下來，基本上就是二分搜了，我們可以知道，若兩個節點深度相同，要找出兩節點的 LCA 等於找出一個 t，使兩者的第 t 輩祖先恰好相同，我們二分搜的目標

就是這個 t。更詳細地說，就是從大的二的冪次開始，若兩者的第 2^i 祖先不同，就將兩者轉換為兩者原先的第 2^i 祖先然後繼續。

```
for(int i = MaxLog-1;i >= 0;i--)
{
    if(anc[a][i] != anc[b][i])
    {
        a = anc[a][i];
        b = anc[b][i];
    }
}
```

最後的 LCA 就是 anc[a][0]。

一開始有說到可以在倍增法的時候紀錄某些性質，是甚麼意思呢？讓我們直接來看看一個例題：

習題 11.4.1: 經典題目

給你一個有 N 個節點的樹，其中每條邊都有邊權，並有 Q 筆詢問，每次詢問問你兩個點之間的路徑上所有邊權的最大值。$(N, Q \leq 10^5)$

我看到題目的第一個想法就是紀錄從跟開始到每個節點的路徑上的邊權的大值。可是稍微想了一想之後，就會發現這樣做好像沒辦法解決問題（因為 max 沒辦法好好合併）！

接著，可能就要請出 LCA 了，能否用 doubling 呢？然後你就會發現真的可以喔！具體做法就是在預處理的時候順便紀錄每個節點走到他 2^k 輩祖先所經過的路徑中的邊權最大值，之後在處理詢問的時候，就可以在向上跳躍的過程中順便找出跳躍過的這一大堆路徑最大值中的最大值囉！

11.5 Tarjan 的離線 LCA 演算法

演算法敘述

可能聰明的讀者有聽說過 Tarjan 這個人的名字！他對演算法學的貢獻不只在這裏，他有更為人知的求割點的演算法，但是這裏要介紹的是他的離線 LCA 求法。顧名思義，他的演算法雖然很快（幾乎線性 !），但是必須離線，也就是需要事先知道需要查詢哪些點對。

他的作法運用到了並查集（Disjoint Sets），所以會用到其基本的操作（你們應該會並查集，放在這裡只是複習用）：

定理 11.5.1: 並查集所支持的操作

1. Find(x)：找到 x 的代表
2. Union(x, y) ：將 x、y 合併成為同一個集合

那假設想要詢問的 LCA 是 $Q = \{(u_1, v_1), (u_2, v_2), \cdots, (u_n, v_n)\}$，那進行以下演算法：從根開始先 DFS 一次，DFS 完一個子節點即將子節點的代表設定為自己，而當遞迴完所有的

對於目前節點 u 的每一個子節點 v 先遞迴下去，然後將 v 的代表變成 u。然後將 u 的顏色（自己設定的值，類似visited）設定為 1，也就是黑色（一開始全部都是 0，白色）。可以發現，**一個節點為黑色若且唯若其所有的子樹都被遞迴完了**。

兩個奇怪的操作

可是，為什麼要做設定代表和判斷顏色這兩個奇怪的操作呢？首先，設定 v 的代表的意思就是：每搜到一個 v 的祖先 x，則會將Find(v)設定為 x，代表搜到其他 x 的子樹內，如果有問到 v 而且沒有在之前就就回答到了，那個詢問的答案就是 x，而且 x 的子樹也都一樣，這個操作利用並查集剛好可以快速實作。

第二，顏色的問題就比較簡單了，就是想要比較晚的來問而已，晚來的會問早來的，而首次（也就是要回答的那一次）問的時候就是被他們的 LCA 遞迴的時候，此時 $dsu[$ 早來的 $]$ 就會是他們的 LCA。

程式碼

看了那麼多，可能你眼花撩亂；那就看簡短的程式碼理解吧（並查集的操作省略）！

```cpp
vector<vector<int> > children; //各個節點的子節點
vector<vector<int> > queries //各個詢問有那個節點的資料，Ex. 詢問(2, 3)會同
vector<int> dsu; //並查集
void tarjanLCA(int u){//u為目前的節點
    for(int v : children[u]){
        tarjanLCA(v);
        Union(u, v);
        anc[Find(u)] = u;
    }
    colour[u] = 1;
    for(int v : queries[u]){
        if(colour[v] == 1){
            Find(v);
            //LCA(u, v) = anc[Find(v)]
        }
    }
}
```

複雜度

這個在一次 DFS 即可完成，而且每一個 query 會被弄到兩次，所以會很接近線性（超線性）的複雜度（會多一個艾克曼的反函數，一般用途不會超過 4）的 $O((N+Q)\cdot\alpha(N))$，此處 $\alpha(N)$ 是反艾克曼函數。

11.6 樹鍊剖分

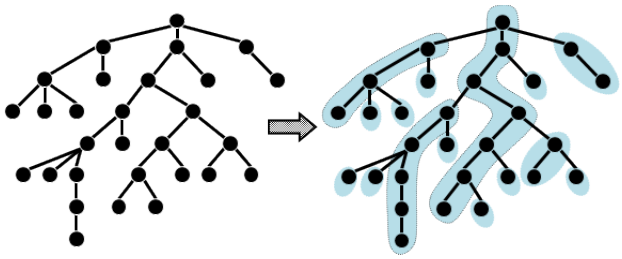
咦！樹鍊剖分，聽起來就很恐怖，到底是啥啊？不用怕，他真的沒有想像中難，他只不過是一種 LCA 的求法而已嘛！當然，他能做的事當然不只 LCA。

何謂樹鍊剖分

樹鍊剖分，又稱輕重鍊剖分 (Heavy-Light Decomposition)，也是用來處理樹上路徑相關問題的好工具。他是一種把序列上的操作搞到樹上路徑的方法，線段樹可以對序列做的事，他幾乎都可以把它搬到樹上來做 (路徑查詢啦、點權更改等等)，真的非常有趣喔！

定義 11.6.1: 重鍊定義

對於一棵樹的每一個節點 u，令其子樹的大小為 $\mathrm{sz}(u)$，則若存在一個 u 的子節點 v 使得 $\frac{\mathrm{sz}(u)}{2} < \mathrm{sz}(v)$，那 u 和 v 就存在一條**重邊**（圖中粗邊為重邊，細邊為輕邊）。



由圖看出，那些重邊會形成一條鍊，而那些鍊就是所謂的**重鍊**！

直接講樹鍊剖分沒什麼意思，讓我們直接來看例題吧！

習題 11.6.1: UVa 12424

給你一棵 N 個節點的樹，每個節點有一種顏色，和 Q 個操作，每一個操作都是以下兩個的其中之一：

1. 將某個點的顏色改掉
2. 查詢兩個節點之間的簡單路徑哪個顏色出現最多次，並且輸出

$(N, Q \leq 10^5$，顏色數 $\leq 10)$

看到這個題目，不知道大家有甚麼想法呢？首先，套一句老話，如果對這個題目沒有想法，就先思考一下簡單的版本：

習題 11.6.2: 上一題的簡單辦

給你一個序列，你需要處理兩種操作：

1. 將某個位置的數字改掉
2. 查詢兩個位置之間哪種數字最多

(序列長度、操作數 $\leq 10^5$，數字種類 ≤ 10)

這個問題好像就不那麼難，只要對每一種顏色開一顆支援單點修改，區間找和的資料分結構 (BIT、線段樹等)，每次修改次單點修改，查詢就把 10 種顏色都區間查詢一遍就好了。

回到原本的題目，我們的目標就是**構造一個節點的序列，使得樹上的一條簡單路徑經過的節點分布在序列中盡量少個連續的區塊**。

若是把樹轉換成一個序列，最直覺的想法就是所謂的**樹壓平**，也就是**任意選定一個根後，依照樹的前序遍歷順序（也就是進入時間）將所有節點由小到大排成一個序列**。這樣做的好處就是，一個子樹會落在連續的一段序列中，所以只要知道是哪些序列，然後再對那些序列詢問，就可以化簡成上面的問題了！因此，若這個問題問的是每個子樹最多的顏色，那用樹壓平的方法可以在 $O(N + Q \log N)$ 的複雜度做完。

很可惜的，這題的主要重點是路徑而非子樹，所以不能用樹壓平解決。

通常，**把無根樹轉為有根樹會有助於解題**，而這麼做之後，兩點之間的簡單路徑就是其中之一向上走到兩者的 LCA，再向下走到另一個點。因為如此，LCA 在路徑問題中佔了蠻重要的地位。那這題可以用倍增法來解決嗎？利用倍增法的確可以在往上跳的過程中順便進行區間查詢，不過要怎麼把節點變成序列，使得在往上跳的過程中不經過太多區塊？因為每個節點都是往根的方向跳，所以很難構造出這個序列。因此，我們在這裡提供另外一種 LCA 的作法 (其實就是樹鍊剖分！)。

用樹鍊剖分解 LCA

首先，DFS 第一次找出每個子樹的大小，並對每一個節點找出他子節點中最大子樹是哪一棵，也順便紀錄每個節點的父節點和每個點的深度。

接著進行第二次 DFS。注意，我這裡直接把細節都說出來喔！從根節點開始，走向他最大的子樹，接下來繼續，直到走到葉為止，這就是第一條重鍊！走的時候，要順便紀錄每個點所屬的重鍊的最高的點，也可以同時為這些點重新編號（也就是第二次 DFS 的進入順序，紀錄了新的序列中每個節點在哪裏）。而找出第一條重鍊後，其他不在這條鍊中的點，也要從自己開始找重鍊。這樣說明可能不是很清楚，不過待會看 code 的時候或許可以體會到喔！為甚麼我們要將點重新編號呢？還記得我們的目標是將這顆樹轉換成一個序列嗎？這個新的編號，就是新序列的順序喔！我們終於拿到一個序列了，接著就對這個序列建構一棵線段樹 (或是其他資料結構) 吧！好，預處理到此可說是告一段落。我們做了那麼多事，到底是為了甚麼？先賣個關子，讓我們繼續看下去。我們終於要來處理 LCA 了！在介紹方法之前，我們先來看看會用到剛剛預處理的哪些東西：

1. 每個節點深度 dep[x]

2. 每個節點自己所在的重鍊中，深度最小的節點 link_top[x]
3. 每個原編號轉換成的新編號 link[x]
4. 每個節點的父節點 pa[x]

這部分看 code 蠻好了解的，我就借卦長的 code 來讓大家看看 XD！(Source：http://sunmoon-template.blogspot.com/2015/07/heavy-light-decomposition.html)

```
#include<vector>
#define MAXN 100005
int siz[MAXN],max_son[MAXN],pa[MAXN],dep[MAXN];
/*節點大小、節點大小最大的孩子、父母節點、深度*/
int link_top[MAXN],link[MAXN],cnt;
/*每個點所在鏈的鏈頭、樹鏈剖分的DFS序、時間戳*/
std::vector<int >G[MAXN];/*用vector存樹*/
void find_max_son(int x){
    siz[x]=1;
    max_son[x]=-1;
    for(auto i : G[x]){
        if( i== pa[x] )continue;
        pa[i] = x;
        dep[i] = dep[x]+1;
        find_max_son(*i);
        if(max_son[x]==-1 || siz[i]>siz[max_son[x]] ) max_son[x] =
i;
        siz[x]+= siz[i];
    }
}

void build_link(int x,int top){
    link[x] = ++cnt;/*記錄x點的時間戳*/
    link_top[x]=top;
    if(max_son[x] == -1)return;
    build_link(max_son[x], top);/*優先走訪最大孩子*/
    for(auto i : G[x]){
        if( i==max_son[x] || i == pa[x])continue;
        build_link(i,i);
    }
}
```

接下來呢？其實很簡單，要找兩點之間的 LCA 時，首先他判斷這兩點是不是在在同一條重鍊上，如果是的話，那頭的深度小的那點就是 LCA 囉 XD。不是的話怎麼辦？我們可以發現，**如果把深度比較低的那個點換成它那條重鍊的深度最小的節點的父節點之後，兩者的 LCA 並不會改變** (因為他跳過的部分絕不可能是 LCA)。因此就這樣一直換，直到兩者在同一條重鍊中囉！

```
int find_lca(int a,int b){
    /*求LCA，可以在過程中對區間進行處理*/
    int ta = link_top[a], tb = link_top[b];
    while(ta!=tb){
```

```
5           if(dep[ta] < dep[tb]){
6               swap(ta, tb);
7               swap(a, b);
8           }
9           //這裡可以對a所在的鏈做區間處理
10          //區間為(link[ta], link[a])
11          ta = link_top[ a=pa[ta] ];
12      }
13      /*最後a,b會在同一條鏈，若a!=b還要在進行一次區間處理*/
14      return dep[a] < dep[b] ? a : b;
15  }
```

Code 中的註解應該也寫得頗為清楚，在將低的節點往上跳的同時，可以順便進行區間操作，因此，我們的 UVa 12424 這題可以說是得到了大致的解法。剩下的問題就是，這麼做的複雜度有比較好嗎？

樹鍊剖分的複雜度分析

已知每次進行區間操作的複雜度是 $O(\log N)$，那對於每一組要求 LCA 的點對，總共要進行幾次「跳躍」呢？這裡我們考慮從一條重鍊上的某一個點跳到他所處的重鍊的深度最小的節點 (稱作 h) 的父節點 (稱作 p) 的時候其子樹的大小，因為 h 的子樹並非 p 的子節點中的最大子樹 (否則他們會在同一條重鍊中)，因此 p 必有一個子節點的子樹的大小比 h 的子樹要來的大，因此，每跳躍一次，跳躍後的節點的子樹都會是原本的兩倍以上。就這個角度來看，一個節點總共最多也只能跳 $\log N$ 次，總共有兩個節點，因此跳躍的次數為 $O(\log N)$。最終，我們可以了解到每次詢問的總複雜度是 $O(\log N)$(跳躍次數)$\times$ $O(\log N)$(區間操作複雜度)！因此，以上的題目可以在總複雜度 $O(N + Q\log^2 N)$ 做完。以下附上程式碼。

跟著蕭電這樣做

要注意的是，樹鍊剖分可以套上其他各式各樣資料結構，包括 Treap、BIT、線段樹，還有其他持久化資料結構。

11.7 習題

仔細思考這些題目有那裏要用到 LCA，都是比較難的題目！

習題 11.7.1: UVa 11354 Bond （TIOJ 1163）

非常厲害的 Ovuvuevuevue Enyetuenwuevue Ugbemugbem Osas（簡稱 Osas）要環遊非洲！他選了 N 個城市，編號為 $1, 2, 3, \ldots, N$，並且他的地圖上寫了其中一些城市間有 M 條雙向的路，長度為 d_i。他要走 Q 次，每一次想要從 s_i 走到 t_i。對於一個從 s_i 走到 t_i 的走法，Osas 的疲累度就是路徑上經過路中最長的那一個。請問，對於每一個 s_i 和 t_i，他可以走的最小疲累度為何（也就是對於每一個 s_i 到 t_i 的路徑中，最大路長最小為何）？($1 \le N \le 50000$，$1 \le M \le 10^5$)

習題 11.7.2: TIOJ 1798 Can You Arrive？

Osas 現在有了困境——到了先進的新加坡（他非洲逛完了），想要搭地鐵前往美食中心，Newton Hawker！但是，他有一個困擾：新加坡有 N 個站，其中被 $N-1$ 條邊連著（恰好是一棵樹！），但是只有 K 種車子行駛，第 i 種車子會在 x_i 到 y_i 之間的唯一路徑往返（因為是樹！），而他現在有 Q 個詢問，請問從編號為 a 的站可不可以經過若干次轉車到達編號為 b 的站？($K \leq N \leq 10^6$，$Q \leq 10^6$)

習題 11.7.3: TIOJ 1445 機器人組裝大賽

現在的 Osas 已經今非昔比，是個科技的人才！他現在參加了一個機器人組裝比賽，其中有 N 個零件，有 M 種方法可以連結它們，第 i 種方法可以連結第 a_i 和 b_i 個零件，代價有 w_i。但是：他發現比賽場地也有一個勁敵，名稱為 Kkwazzawazzakkwaquikkwalaquaza '* Zzabolazza（簡稱 Zzabolazza），他已經找到一個最佳的方法連結所有的零件了（也就是代價最小），請幫助 Osas，找到一個代價盡量小的零件連接方法（一定要全部連！不行的話輸出-1），並且與 Zzabolazza 的連接方法不完全相同）。

塊狀數組（分塊） 12

12.1 前言

分塊可以處理 RMQ （區間最小值）的問題，另外也支援單點修改的操作。

12.2 暴力美學

RMQ 的暴力做法是 $O(N)$ 輸入加上 $O(N)$ 暴力掃最小值。另一種暴力求法是對每組 (l,r)$O(N)$ 建二維的表來 $O(1)$ 查詢。但這樣兩種做法在側資量大時會 TLE 。因此我們需要更有效的演算法。

12.3 原理

在上述第二種做法當中，會發現其實有幾段間會被很多組 $[l, r)$ 被用到；也就是說，一段區間詢問可以分成常用的區間加上不常用的區間，而經常使用到的區間可以預處理存下來，以加快詢問速度。當然我們可以用複雜的數學算出預處理哪些區間可以使複雜度最低，但為了簡化，我們多半以數列分塊的形式來解決這個問題。於是我們將整個序列分成 k 段，每段有 $\frac{N}{k}$ 個數字，而這 k 段就視為上述經常被用到的區間。接著存下每段的最小值，如此一來只要詢問包含到這塊的區間，就可以直接存取他的最小值, 不必再重算一次了！

12.4 操作流程

預處理

$O(N)$ 暴力掃一遍，紀錄每個區間的最小值。

查詢

假設詢問區間 $[l, r)$ 的最小值，答案即是每塊被包含於 $[l, r)$ 的區間（直接存取剛剛計算的最小值）以及剩下不滿一個區間的值（暴力掃）。

單點修改

只要修改輸入的點，並更新那塊的最小值（暴力掃），即可 $O(\frac{N}{k})$ 完成。

複雜度

預處理為 $O(N)$。查詢時要存取區間內每塊的最小值，複雜度 $O(k)$；以及暴力掃不滿一個區間的值，複雜度 $O(\frac{N}{k})$。查詢總複雜度為 $O(k+\frac{N}{k})$，根據算幾不等式，k 取為 $\sqrt{N}$ 時複雜度最小，為 $O(\sqrt{N})$。故整體為預處理 $(O(N),$ 詢問 $O(\sqrt{N})$ 的 RMQ。

以下是程式碼：

```
#define inf 2147483647
#define maxn 100
int a[maxn],b[(int)sqrt(maxn)+1],k;
void init(int n){
    k=(int)sqrt(n);
    for(int i=0;i<(n+k-1)/k;i++)
        b[i]=inf;
    for(int i=0;i<n;i++)
        b[i/k]=min(b[i/k],a[i]);
}
int query(int l,int r){
    int minn=inf;
    while(l%k&&l<=r)
        minn=min(minn,a[l++]);
    while((r+1)%k&&l<=r)
        minn=min(minn,a[r--]);
    while(l<r)
        minn=min(minn,b[l/k]),l+=k;
    return minn;
}
void modify(int ind,int v){
    if(b[ind/k]==a[ind]){
        a[ind]=v;
        b[ind/k]=inf;
        for(int i=(ind/k)*k;i<(ind/k+1)*k;i++)
            b[ind/k]=min(b[ind/k],a[i]);
    }
    a[ind]=v;
}
```

12.5 其他應用

分塊可以應用在帶修改區間求和，預處理存下每個區間的和，查詢和 RMQ 相同，修改則更新該點和該塊的值。詢問複雜度為 $O(1)$
而分塊也可以應用到例如區間眾數... 等方面。

12.6 關於區間修改

事實上分塊也可以支援區間修改，複雜度 $O(\sqrt{N})$。大致上就是 b 陣列暴力 $O(\sqrt{N})$ 修改。a 陣列頭尾不滿一整塊的也是暴力 $O(\sqrt{N})$ 修改，而其他的就分塊打懶標（詳細懶標的觀念可能以後就會提到了吧）。

字串入門 13

13.1 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。

哪就讓我們趕快進入此次的主題－字串吧！

13.2 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 L_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 $A_{i...L_A-1}$。

13.3 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 13.3.1: 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 $O(L_P L_T)$。

```
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 $h(A) = A_0 p^{L_A-1} + A_1 p^{L_A-2} + ... + A_{L_A-1}$ (L_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 $h(A_{1...L_A-1}), h(A_{1...L_A-2}), ..., h(A_{1...1})$。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 $O(L_A)$ 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 $O(L_A)$ 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 $O(L_A)$ 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 $L_A - L_B + 1$ 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 $O(L_A + L_B)$。

Hash 固然是字串匹配的好幫手，不過千萬不要絕 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。

13.4 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。

13.5 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

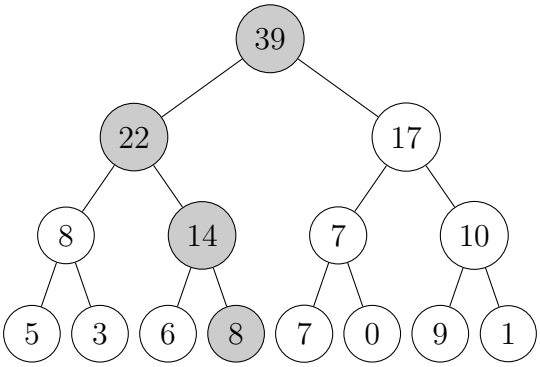
相信有了失配函數之後，大家都很快就知道怎麼進行字串匹配了。我們先對欲尋找的字串 P，建立它的失配函數陣列，然後用兩個變數 i, j 在字串 T 與 P 上爬行。

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！$j = L_P + 1$ 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++;
        }
    }
}
```

13.6 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 $O(L \lg N)$。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！看看它的英文自首，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：
首先先插入"she"：



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！

13.7 習題

排版跑掉了 郭死

習題 13.7.1: 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。(字串長度 $\leq 10^6$)

習題 13.7.2: 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)？(字串長度 $\leq 2 \times 10^5$)(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 13.7.3: 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 13.7.4: 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？(字串長度 $\leq 10^5$)

離線處理淺談 - 強大的工具 14

如果一個演算法在計算答案之前不需要知道所有輸入，我們稱這個演算法為**在線**演算法；反之若在計算答案的時候要先知道所有輸入，我們稱此為**離線**演算法。例如插入排序在排序所有數字時可以一個一個讀進來再各自插入，選擇排序則是必須先讀進所有數字後，才能按照順序把最小的數字放到前面。

犧牲了即時更新性所換來的通常是計算量或 coding 複雜度的減少，我們先以 RMQ 當作一個簡單的例子。

習題 14.0.1: 經典題 - RMQ (ZJ d539)

給定一個序列 $a_1, a_2, \ldots, a_n$，之後有 q 次詢問 $[l, r]$ 中的最大值。$n, q \leq 10^5$。

我們可以讀進所有詢問 $[l, r]$，並依照左界由大到小排序。如此一來，對於固定的左界只需要維護一個前綴 max 就足以回答所有詢問，我們可以想到用 BIT 來維護這個東西，對於左界的遞減剛好對應到單點更新，就這樣輕鬆完成不須線段樹的 RMQ 啦！雖然這個例子可能不一定能感受到離線演算法的威力，但大家應該了解到了，如果離線計算的話可以用很多奇怪的方法去減輕時間、空間或 coding 複雜度。

14.1 莫隊 (Mo's Algorithm)

從前面的題目可以發現，在面對一些區間詢問的題目時，有時我們可以藉由將詢問以某種方式排序來降低操作整體的複雜度，其中有一種演算法稱為**莫隊**。

莫隊演算法通常適用於那些當區間左右界稍微改變時，能以很小的代價維護新區間的題目，簡單來說，我們要想辦法排序詢問使得兩兩之間的「距離」都很相近，讓我們可以從前一個詢問的答案用不多的時間推出後一個詢問的答案以節省時間複雜度。如果能花 $F(n)$ 的時間從 $[l, r)$ 的答案推到 $[l \pm 1, r)$ 或 $[l, r \pm 1)$，由 $[l_1, r_1)$ 的答案推出 $[l_2, r_2)$ 的答案所需要的時間就是 $F(n) \cdot (|l_2 - l_1| + |r_2 - r_1|)$，這也可以看成座標平面上兩點的曼哈頓距離，我們可以用曼哈頓最小生成樹的演算法達到最差 $O(F(n) \cdot n\sqrt{q})$ 的複雜度，但是否有更簡單的方法能達到相同的複雜度呢？

我們可以想到先將詢問依照左界排序，這樣當我們要查詢時左界只需要修改最多 $O(n)$，不過聰明的大家應該會發現右界的順序是亂的！這就會導致我們的修改次數最多會退化至 $O(nq)$，為了解決這個問題，我們決定也以某種方式對右界排序。

我們把原序列分成 m 塊 (每塊有 n/m 個元素)，並且依照左界所在的塊排序，若在同一塊則以右界排序。左界在同一塊間的移動次數總和最多就是 $q \cdot n/m$，而不同塊之間的移動次數最多是 $m \cdot 2(n/m) = 2n$；右界在同一塊間的移動次數總和最多是 nm ，而不同塊之間右界的移動次數最多也同樣是 nm；因此總複雜度為 $O(F(n) \cdot (nq/m + nm + 2n))$，可以發現取 $m = \sqrt{q}$ 會得到最佳複雜度 $O(F(n) \cdot n\sqrt{q})$ ，這也和曼哈頓最小生成樹可得到的結果相同，我們通常採用這種寫法。莫濤提出的莫隊算法，除了將詢問排序之外，剩下都只是暴力計算，但是就是因為把詢問好好排序，得以把複雜度降低了一個根號。

實作上有一些細節，就是在更新左、右界時，盡量不要讓右界比左界小（也就是不要讓區間不合理），這部分的程式碼通常會用數個while來實現，其架構大概長的像下面這樣。

```
struct Query{
    int l, r, id, block;
} Q[MAXQ];

bool cmp(Query &a, Query &b){
    return (a.block!=b.block) ? a.block<b.block : a.r<b.r;
}

int n, m, res, ans[MAXQ];
void add(int pos){ ... } // 維護增加一個數的改變
void sub(int pos){ ... } // 維護減少一個數的改變

void MO(){
    int K = 400; // 有時會依範圍直接寫一個固定的大小
    for(int i = 0; i < q; i++) Q[i].block = Q[i].l/K;
    sort(Q, Q+q, cmp); // 有時沒排序也沒過(X
    // 初始化也很重要，要注意一開始的區間
    int l = 1, r = 0;
    for(int i = 0; i < q; i++){
        // 先擴展再縮減
        // 這邊是左閉右閉的區間寫法，邊界問題自己要注意
        while(l > Q[i].l) add(--l);
        while(r < Q[i].r) add(++r);
        while(l < Q[i].l) sub(l++);
        while(r > Q[i].r) sub(r--);
        ans[Q[i].id] = res;
    }
    for(int i = 0; i < q; i++) cout << ans[i] << '\n';
}
```

雖然莫隊通常是用來解決不帶修改的題目，但大陸人也有研發出可以解決待修改問題的莫隊，想法大概是把時間視為第三個維度，變成三維曼哈頓距離的感覺 (前提是要能在時間軸上前後移動)，用類似的方法可以有 $O(n^{\frac{5}{3}})$ 的複雜度。

14.2 操作分治

操作分治，又名 CDQ 分治。既然名字裡面有個分治，顧名思義就是利用了 Divide & Conquer 的算法。通常我們都希望問題的維度越低越好，但操作分治的做法卻是**加上了一個時間維度，對時間分治，並保持其中一個維度有序**，使問題的維度由 n 到 $n+1$ 再降到 $n-1$。

儘管這個想法聽起來莫名其妙，但在處理帶修改的可離線問題時非常有效。尤其是原本是帶修改的二維問題（加上時間變成三維），可能可以透過操作分治降低複雜度。

實際上操作分治可以分為三個步驟:

- 計算左邊區間的答案
- 計算右邊區間不受左邊區間影響時的答案
- 計算左邊區間對右邊區間的影響

注意這邊是對時間分治，所以左邊右邊代表了時間上的先後；而以上三個步驟的順序也可能會依照題目改變。直接上例題吧！

習題 14.2.1: 經典題 (No judge)

一個二維平面上，有 n 次操作，每次操作可以在一個點上加上權重，或者詢問某個座標的左下角所有權重的和。$n \leq 10^5, |x|, |y| \leq 10^9$。

離散化是不可避免的，但就算離散化後用二維 BIT 或線段樹依然會 MLE，我們嘗試用操作分治降低問題的維度。

首先依照對每個操作加上一個時間維度，並直接對其分治。對每個區間來說，左右兩個區間的答案會變為兩個子問題，因此我們需要處理的只有「左邊區間的修改」對「右邊區間的查詢」的影響。可以發現，如果按照 x 座標排序好，我們就可以直接動態維護 y 座標前綴和 (BIT) 以得到答案；實作上有點類似 merge sort 中 merge 的部分，下面的 code 大概顯示了如何運作。

```
Query Q[MAXQ];
void merges(int L,int M,int R){
    int i = L, j = M;
    vector<query> tmp;
    init(); // BIT
    while(i<M || j<R){
        if((j==R) || ((i<M&&Q[i].x<Q[j].x){
            if(Q[i].type == ADD)
                add(Q[i].y,1); // BIT
            tmp.push_back(Q[i++]);
        }else{
            if(Q[j]].type == QRY)
                ans[Q[j].id] += query(Q[j].y); // BIT
            tmp.push_back(Q[j++]);
        }
    }
    for(int i=0;i<R-L;i++)
```

```
18          Q[i+L] = tmp[i];
19  }
20  void CDQ(int l=0, int r=n){
21      if(r-l == 1) return;
22      int mid = l+(r-l)/2;
23      CDQ(l,mid),CDQ(mid,r);
24      merges(l,mid,r);
25  }
```

每次把一個區間 $[l,r)$ 的兩部份以 BIT 合併求解的複雜度是 $O((r-l)\log n)$ (在離散化的前提下)，總時間複雜度 $T(n)=2T(n/2)+O(n\log n)$ ，由主定理得知 $T(n)=O(n\log^2 n)$。

雖然這題可能也可以用動態開點四叉樹或其他奇怪的資料結構解決，但操作分治寫法在空間、coding 複雜度都表現得很好，並且時間複雜度的常數也很小。

14.3 整體二分搜

讀到這裡的同學應該都了解什麼是二分搜了吧！二分搜的檢查有時會有些代價，而如果直接各自二分搜這些代價加起來會使複雜度爛掉，我們想辦法讓這些代價能夠共用而減輕複雜度，因此就誕生了**整體二分搜**的演算法，顧名思義就是整體一起進行二分搜。

習題 14.3.1: 經典題 - 靜態區間第 k 小

給定一個序列 $a_1,a_2,\ldots,a_n$，之後有 q 個詢問 (l,r,k) ，請對每個詢問輸出 $[l,r]$ 第 k 小的數字。

區間第 k 大可能在嵌套或持久化資料結構也會提到，但利用持久化線段樹的寫法空間複雜度很高（筆者寫了兩天還一直 MLE，QAQ），也不是我們今天的主題。

類似於lower_bound，考慮對答案二分搜：每次詢問一個 x 並檢查每個詢問區間有幾個數不大於 x ，如此可以分成答案要比 x 大及答案要比 x 小的兩種詢問；注意原序列的數也可以分成比 x 大及比 x 小的兩種，因此能把原序列也分成兩部分，變為兩個子問題遞迴解決。

至於我們要怎麼知道每個區間有多少數不大於 x 呢？我們可以用 BIT 維護「$\le x$ 的數的個數」的前綴和，如此對於右半部遞迴的前綴和只需要以原序列中大於 x 的數修改前半部遞迴時的 BIT 即可，省下每次都重算原序列或重置 BIT 的時間複雜度。整體二分的核心程式碼大概長的像這樣 (參考自日月卦長的部落格)，當然不同題目一定會需要修改細節部分。

```
1  // 原序列的數可視為操作一起放進陣列
2  // V,L,R 是操作們的索引，傳遞 int 比傳遞物件效率好
3  void totalBinarySearch(int l,int r,vector<int> &V){
4      int mid = l+(r-l>>1);
5      vector<int> L,R;
6      // 將V中對應的原序列數及詢問以mid為界分到L,R
7      split(V,L,R,mid);
8      // 二分搜答案不大於 mid 的
```

```
9       totBS(l,mid,L);
10      // 在BIT中記錄不大於 mid 的數造成的影響
11      update(L);
12      // 二分搜答案大於 mid 的
13      totBS(mid+1,R);
14      // 復原操作，以免影響到之後的搜索
15      undo_update(L);
16  }
17  // split的實作，利用BIT
18  void split(vector<int> &V,vector<int> &L,vector<int> &R,    int x){
19      for(int id:V){
20          if(...){ //是原序列數a_i
21              if(a_i < x) {
22                  add(i,1); // BIT
23                  L.push_back(id);
24              }else R.push_back(id);
25          }
26      for(int id:V){
27          if(...){ // 是詢問(l,r,k)
28              int cnt = query(r)-query(l-1); // BIT
29              if(cnt < x) L.push_back(id);
30              else R.push_back(id);
31          }
32      }
33      // 離開時記得 undo 讓 BIT 和進來前相同
34      undo_update(L);
35      V.clear(); // 節省空間
36  }
```

我們來簡單分析一下整體二分搜的複雜度：原序列和詢問都在遞迴樹中出現，每個原序列的數以及每一個詢問出現的次數都是樹高 $O(\log(n+q))$ ，也是 BIT 修改及查詢的時間複雜度，而每次 BIT 修改最多 $O(\log n)$（在離散化的前提下），因此我們的均攤時間複雜度是 $O((n+q)\log n\log(n+q))$。整體二分搜的一個明顯優點是空間複雜度低，只有 $O(n+q)$ ，畢竟除了遞迴呼叫使用的空間外也只開了一條 BIT，想當然爾出題者可能藉由記憶體限制的方式強制離線，又因為 code 短且不需要實作持久化樹狀資料結構，因此整體二分在競賽中被廣泛使用。

另外，在這題中由於我們是利用 BIT 來維護，undo 的成本很低，因此可以輕鬆的以 DFS 的方式呼叫遞迴，但有時候對於一些資料結構要 undo 其實是很困難的 (例如 DSU)，這時候我們就需要用到 BFS 的技巧。每次操作只在資料結構中增加內容，當二分搜進入新的一層後再將整個資料結構重置，便能避免掉昂貴的 undo 操作。

14.4 習題

習題的唷

習題 14.4.1: XOR and Favorite Number(CF 617E)

給定 k 及一個長度為 n 的正整數序列 s ，對於 Q 次詢問 l, r ，每次輸出 $[s_l, \dots s_r]$ 中有幾對 (i, j) 使得 $s_i \oplus s_{i+1} \cdots \oplus s_j = k$，其中 $\oplus$ 代表 XOR 運算。$n \le 10^5; s_i, k \le 10^6$。

習題 14.4.2: 區間眾數 (ZJ b417)

給定一個長度為 n 的正整數序列 s ，對於 m 次詢問 l, r，每次輸出 $[s_l, \dots s_r]$ 眾數的個數以及有幾種數字是眾數。$n, m \le 10^5, 1 \le s_i \le n$。

習題 14.4.3: 區間逆序數對 (TIOJ 1694)

給定一個長度為 n 的正整數序列 s ，對於 q 次詢問 l, r ，每次輸出 $[s_l, \dots s_r]$ 逆序數對的個數。$n, q \le 10^6$，$s_i \le 10^9$。

習題 14.4.4: Coding Days (TIOJ 1840)

給定一長度為 N 的序列以及 Q 筆操作, 每筆操作可能為下列兩種:

1. `1 l r k`: 請輸出區間 $[l, r]$ 中第 k 小的值。
2. `2 p v`: 將序列中的第 p 個值改成 v。
3. `3 x v`: 關於這項操作，詳細的內容請觀察題目敘述。

$N \le 50000, Q \le 10000$，序列中的數皆可以用`int`儲存。

習題 14.4.5: Stamp Rally (Atcoder)

給定一張 N 個點、M 條邊的無向圖，每條邊編號為 1~M。有 Q 次詢問 (x, y, s) ，輸出最小的 i ，使得存在分別以 x 和 y 為起點的兩條路徑，路徑上的邊編號都不大於 i，且兩條路徑上共有 s 個不同的點（包含 x, y ，可以經過重複的邊或點但不會重複算到）。$N, M, Q \le 10^5$。

樹堆 Treap 15

15.1 前言

treap=tree(binary search tree)+heap，是一種隨機平衡二元搜尋樹，對於任意的序列，他的插入、刪除、查詢操作期望複雜度皆為 $O(\log N)$。一般的 BST 雖然期望深度也是 $O(\log N)$，但是只要刻意餵給他排序好的序列，就一定會退化成鍊狀，這就是為什麼我們需要 treap 了。因為其期望複雜度低且程式碼相對簡單，故有些時候能拿來代替自平衡二元搜尋樹。

15.2 原理

treap 不同於一般二元搜尋樹的是，每個節點除了紀錄這個節點的數字（我們稱之為 key 值）外，同時會記錄一個 pri 值，struct大概長這樣：

```
int randseed=7122;
int rand(){return randseed=randseed*randseed%0xdefaced;}
struct node{
    int key,pri;
    node *l,*r;
    node(){};
    node(int _key):key(_key),pri(rand()){l=r=nullptr;}
};
```

樹中的 key 值會保持 BST 的性質（所以他是一種二元搜尋樹），pri 值則會保持 heap（二叉堆）的性質。這樣可以幹嘛呢 ??? 我們先對這個結構熟悉一下：

唯一性

在介紹二元搜尋樹時，我們知道對於同一組數字建立二元搜尋樹，他可以有很多種不同的樣子。但在 treap 中當每對 key,pri 都固定時，其建構出來的 treap 是唯一的。你可以想像，BST 的性質當中可以固定左右的關係（左邊的 key 值一定小於右邊），而 heap 的性質中可以固定上下的關係（上面的 pri 值一定小於下面），上下左右關係都固定的情形下，這棵 treap 自然就固定了啊！(你說這樣一點都不嚴謹？講的好像我知道怎麼嚴謹證明一樣)

當 pri 值 $=1\sim n$ 時

這個情況下，我們可以想像一棵最普通的二元搜尋樹（只有 key 值），我們依原本節點 pri 值從 $1\sim n$ 的順序一一插入，因為越後面插入的節點一定在越下面，因此 pri 值（= 插入時間）大的一定在越下面。如此一來，tree 和 heap 的性質就同時都滿足了。換句話說，每個 $1\sim n$ 的排列各自代表一種插入順序。

當 pri 值隨機時

根據上面的結論，pri 值的大小可以視為插入時間的先後，那如果 pri 值隨機的話，不就是隨機順序插入的意思嗎？而我們都知道，隨機順序插入的二元搜尋樹，其期望深度 $O(\log N)$（深度超過 $2\log N$ 的機率是 $\frac{1}{n^2}$），這也就是為什麼他的操作都可以 $O(\log N)$ 達成了。

15.3 實作方法

聽到這裡，你可能會想說：講那麼多，到底要怎麼在操作的同時保持這些性質啊？事實上，維持這個性質的方法有很多，主要有 merge-split treap 和旋轉式 treap。這邊要介紹的是 merge-split treap，因為這種方法好刻 code 又短。

merge-split tree 顧名思義，需要有 merge（合併）、split（分裂）兩個主要操作。而這兩個操作的實作方法其實就是遞迴，要詳細解釋也沒什麼意思，所以就直接切到程式碼的部分吧。

merge

這個操作是要將 a,b 兩個 treap 合併成一個，**其中 a 的所有 key 值都小於 b**。

```
node *merge(node *a,node *b){ //將根節點為a,b的treap合併
    if(!a) return b; //base case
    if(!b) return a; //base case
    if(a->pri<b->pri){
        a->r=merge(a->r,b);
        return a;
    }else{
        b->l=merge(a,b->l);
        return b;
    } //pri值小的當父節點，大的當子節點。
}
```

split

這個操作是要將一個 treap 的 key 值 $\le k$ 的都丟到 a，其餘 $>k$ 的丟到 b。

```
void split(node *s,node *&a,int k,node *&b){
    if(!s) a=b=nullptr; //base case
```

```
    else if(s->key<=k) //s的key較小，故s和其左子樹都在左邊
        a=s,split(s->r,k,a->r,b); //分割右子樹
    else
        b=s,split(s->l,k,a,b->l); //分割左子樹
}
```

15.4 延伸操作

上面兩個操作看起來實在是沒有什麼實際用途，所以我們要來介紹一下利用它們組合而成的延伸操作。

insert

insert的做法只需要先將原本的 treap 拆開，再把左、新節點、右依序合併就好了。以下是程式碼：

```
void insert(node *&root,int t){
    node *a,*b;
    split(root,a,t,b);
    root=merge(merge(a,new node(t)),b);
}
```

erase

erase基本上就是反著做insert就好了。以下為程式碼：

```
void erase(node *&root,int t){
    node *a,*b,*c;
    split(root,a,t,b);
    root=b;
    split(root,b,t,c);
    root=merge(a,c);
}
```

15.5 比 BST 更強大的功能

treap 是一個進階版的 BST，因此能改裝成具有更多功能的東西。其中，treap 最吸引人的功能就是 merge 和 split 了。不過我們先從一些初階的東西開始講起。

記錄 size

要記錄每個子樹的 size，就是自己的 size= 左子樹的 size+ 右子樹的 size+1。這簡單的運算一切就交給遞迴就好了（事實上就是線段樹的懶標）。這邊就不附上 code 了（後面會有）。

名次樹（rank tree）

rank tree 就是要能查詢各個 rank 的值以及每個值得 rank。在 search by key 時，可以用兩種方式（假設要查詢 key= k 的 rank）：

1. 將 key$\leq k$ 的 split 出來，最後根節點的 size 就是 rank 了。
2. 同 BST 的查詢，一開始 rank= 1，每次若不是向左子樹走時 rank+= 左子樹的 size。

而 search by rank 時，假設要查詢 rank= k 的 key 值，從根節點開始：如果左子樹的 size+1 =rank，代表此節點的 key 值就是答案；如果 size$\geq$rank, 則向左子樹遞迴；如果 size+1 <rank，則向右子樹遞迴，並且 rank-= 左子樹的 size+1。

split by rank

既然可以 search br rank，下一步就可以 split by rank 了！作法其實都一樣，只是要記的在遞迴後面加上 pull()，才能保持住正確的 size。

剛剛講了這麼多，重點就是為了下面這個神奇的東西：

序列轉 treap

什麼是序列轉 treap 呢 ??? 一般而言 treap 當中都是按照數字大小作為 key 值，達到 BST 的效果。然而現在，我們希望這棵 treap 中序尋訪的結果恰好就是原序列；也就是以 index 值作為 key 值。然而我們會知道，其實這邊的 key 值恰好就會是他的 rank，在查詢、split 時都可以用 rank 來進行就好，所以我們通常將 key 值省略不記（到後面你就會知道記與不記的差別了）。

懶人標記（lazy tag）

懶標你們一定不陌生，沒錯，他就是在線段樹上出現過的東西。這裡的 treap 恰好也是紀錄一個序列，所以，當然也可以使用懶標啦！假設我現在要對 $[l, r)$ 進行區間加值，我們可以先利用 split by rank 將整棵 treap 進行 split，分成 $[0, l)$，$[l, r)$，$[r, n)$ 這三棵 treap，然後在 $[l, r)$ 這棵 treap 的根上打懶標，最後再 merge 回去，就完成區間加值了。

區間操作

從上面的例子你應該可以發現，在 treap 當中我們可以在 $\log N$ 的時間內切出任意的區間，因此讓區間操作變得非常容易。這邊我們就舉交換區間的例子來讓大家更深刻體會 treap 的美妙吧！

習題 15.5.1: 交換區間

給定一序列及兩種操作：

1. 將 $[l_1, r_1)$，$[l_2, r_2)$ 兩個區間的位置交換。
2. 查詢序列第 k 的數字是多少。

每次都要搬動一個區間非常麻煩，但使用 treap 的話，只要切成 $[0, l_1)$，$[l_1, r_1)$，$[r_1, l_2)$，$[l_2, r_2)$，$[r_2, n)$ 五段，在依 $[0, l_1)$，$[l_2, r_2)$，$[r_1, l_2)$，$[l_1, r_1)$，$[r_2, n)$ 的順序 merge 回來就好了！是不是很簡單？我們來看 code 吧：

```
int rs=1e9+7;
int rand(){return rs=(rs*rs)%0xdefaced;}
struct node;
int s(node *a);
struct node{
    int a,pri,si;
    node *l,*r;
    node(){}
    node(int _a):a(_a),si(1),pri(rand()){l=r=nullptr;}
    void pull(){si=s(l)+s(r)+1;}
};
int s(node *a){return a?a->si:0;}
node *merg(node *a,node *b){
    if(!a) return b;
    if(!b) return a;
    if(a->pri<b->pri)
        return a->r=merg(a->r,b),a->pull(),a;
    else
        return b->l=merg(a,b->l),b->pull(),b;
}
void split(node *n,node *&a,int k,node *&b){
    if(!n) return a=b=nullptr,void();
    if(k>s(n->l)+1){
        a=n;
        split(n->r,a->r,k-s(n->l)-1,b);
        a->pull();
    }else{
        b=n;
        split(n->l,a,k,b->l);
        b->pull();
    }
}
int query(node *n,int k){ //0-base
    if(s(n->l)+1==k) return n->a;
    if(s(n->l)+1<k) return k-=s(n->l)+1,query(n->r,k);
    else return query(n->l,k);
}
void change(node *&n,int l1,int r1,int l2,int r2){
```

```
39      //0-base,[)
40      node *a,*b,*c,*d,*e;
41      split(n,a,l1,b);n=b;
42      split(n,b,r1-l1+1,c);n=c;
43      split(n,c,l2-r1+1,d);n=d;
44      split(n,d,r2-l2+1,e);
45      n=merg(merg(merg(merg(a,d),c),b),e);
46  }
```

pbds 16

16.1 前言

這份講義要介紹的是一個 GNU C++ 中的函式庫－－ pbds(policy-based data structure)，俗稱平板電視。他跟 STL 有一點類似，但是他其中的某些功能是 STL 無法達到的。大家應該都知道 STL 在競賽中的重要性，因此 pbds 也絕對是幫助你邁向程式設計競賽成功的好幫手！話不多說，我們就趕快來看看怎麼用 pbds 吧！

16.2 pbds 介紹

接下來就向各位介紹 pbds 的用法及其中的資料結構。
首先，無論是使用 pbds 中的人和資料結構，都需要打上這兩行：

```
#include <ext/pb_ds/detail/standard_policies.hpp>
using namespace __gnu_pbds;//there are two _ before gnu, but only one _ bet
```

以下就開始正式開始介紹 pbds 中的資料結構囉！

Hash Table

就跟 STL 中的unordered_map幾乎一樣 (可用中括號隨機存取、insert、delete 、clear、find、查看 size、自定 hash function，但不能count)，它分成gp_hash_table和 cc_hash_table，使用並無差異，只是處理碰撞的方式不同。
而使用時還要加上：#include<ext/pb_ds/assoc_container.hpp>。讓我們直接來看一下範例程式碼吧！

```
#include<bits/stdc++.h>
#include<ext/pb_ds/detail/standard_policies.hpp>
#include<ext/pb_ds/assoc_container.hpp>
using namespace std;
using namespace __gnu_pbds;
int main()
```

```
{
  gp_hash_table<int,int> r;
  for(int i = 0;i < 5000000;i++)
  {
    r.insert({i,i+1});
  }
  cout << (r.find(7))->second << endl;
  for(int i = 0;i < 5000000;i++)
  {
    r.erase(i);
  }
  if(r.find(7) == r.end())
  {
    cout << "No!" << endl;
  }
}
```

之後我又將cc_hash_table換成gp_hash_table和unordered_map。跑一樣的程式碼，cc_hash_table跑 2.42 秒左右，gp_hash_table跑 2.53 秒左右，unordered_map則跑了 5.06 秒。因此若覺得unordered_map常數太大的人可以考慮用用看 pbds 的 hash table 喔！

Priority Queue

對，這也跟 STL 的priority_queue差不多，STL 的priority_queue支援的操作它都支援，而為了防止與 STL 中的priority_queue搞混，在使用上要特別注意在宣告的時候要加上__gnu_pbds::，以確保用到的是 pbds 中的priority_queue。宣告方式也有些微差異，可在底下範例中看到。相較於 STL 的priority_queue只有一種，pbds 的priority_queue可以在宣告時加上一個 tag，以決定priority_queue的實作方式。使用不同的實作方式也就會導致不同操作有不同複雜度，簡單整理如下：

	push	pop	modify	erase	join
std::priority_queue	最差 $\Theta(n)$ 均攤 $\Theta(\log n)$	最差 $\Theta(\log n)$	最差 $\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$

其中的 pairing heap 和 binomial heap 較常用到。
或許會有人好奇，要怎麼修改priority_queue內部的元素呢？
首先，pbds 的priority_queue的 iterator 宣告如下：

```
__gnu_pbds::priority_queue<int>::point_iterator it;
```

此外，將元素push到 pbds 的priority_queue的時候會有回傳值，即為該元素所在的 iterator，只要在push的時候紀錄一下，就可以利用modify函式和 iterator 來進行修改囉！
使用時別忘了加#include<ext/pb_ds/priority_queue.hpp>。
給個範例程式：

```cpp
#include<bits/stdc++.h>
#include<ext/pb_ds/detail/standard_policies.hpp>
#include<ext/pb_ds/priority_queue.hpp>
using namespace std;
using namespace __gnu_pbds;
int main()
{
  __gnu_pbds::priority_queue<int,less<int>,binomial_heap_tag> pq;//型態 比較
  __gnu_pbds::priority_queue<int,less<int>,binomial_heap_tag>::point_iterat
  for(int i = 0;i < 10;i++)
  {
    it[i] = pq.push(i);
  }
  pq.modify(it[0],7122);
  cout << pq.top() <<endl;
}
```

有沒有覺得寫 dijkstra 的時候會派上用場呢？

接著示範合併 (join)，注意被合併的priority_queue的大小會歸零。

```cpp
#include<bits/stdc++.h>
#include<ext/pb_ds/detail/standard_policies.hpp>
#include<ext/pb_ds/priority_queue.hpp>
using namespace std;
using namespace __gnu_pbds;
int main()
{
  __gnu_pbds::priority_queue<int,less<int>,pairing_heap_tag> a,b;
  for(int i = 0;i < 5;i++)
  {
    a.push(i);
    b.push(i+5);
  }
  a.join(b);
  cout << a.size() << " " << b.size() <<endl;
  cout << a.top() <<endl;
}
```

tree

接下來講的就是大家常常抱怨的set和map的進化版了啊，就是 pbds 中的平衡二元樹。它除了insert、erase、lower_bound、upper_bound等set已經有的操作之外，還可以找 k 小值 (find_by_order，回傳 iterator(0-based)) 和求排名 (order_of_key，回傳一整數 (0-based))，還支援類似 merge-split treap 的join(join的時候兩棵樹的類型要相同，沒重複元素，且 key 值要左小右大) 還有split。

它的宣告長這樣：

```
tree<int,null_type,less<int>,rb_tree_tag,tree_order_statistics_node_update>
```

前三格就是 STL map中的 key、對應的資料型態，和比較函式。第四格是各種樹的 tag，有rb_tree_tag、splay_tree_tag、ov_tree_tag，其中的rb_tree_tag較為常用。其他兩個的用處我不是很了解，但它在一般使用上的常數較高，盡量避免使用。最後就是更新方式了，不知道要打甚麼就照抄吧！
它其實還有一些更進階的用法，像是自行定義點的更新方式，不過我也不會，而且蛋餅說還不如自己刻一顆 treap，所以這邊我就不多說大家有興趣的自行研究喔！也記得要加上#include<ext/pb_ds/assoc_container.hpp>。
嗯，一樣來看範例程式。

```
#include<iostream>
#include<ext/pb_ds/assoc_container.hpp>
#include<ext/pb_ds/tree_policy.hpp>
using namespace std;
using namespace __gnu_pbds;
int main()
{
  tree<int,null_type,less<int>,rb_tree_tag,tree_order_statistics_node_updat
  for(int i = 0;i < 10;i++)
  {
    a.insert(i*2);
  }
  a.erase(18); // tr = {0,2,4,6,8,10,12,14,16}
  cout << *a.lower_bound(11) << endl; // 原本set和map支援的操作都有
  cout << *a.find_by_order(0) << endl;
  cout << *a.find_by_order(5) << endl;
  cout << a.order_of_key(9) << endl;
  cout << a.order_of_key(10) << endl;
  a.split(6,b);
  for(auto i : a)
  {
    cout << i << " ";
  }
  cout << endl;
  for(auto i : b)
  {
    cout << i << " ";
  }
  cout << endl;
  a.join(b);
  cout << a.size() << endl;
}
```

Trie

大家可能會覺得剛剛的資料結構或許很厲害，但其實都只是 STL 之中不同資料結構的變形。但接下來要介紹的資料結構，STL 之中就沒有了呢！它，就是大名鼎鼎的 Trie！

首先先介紹宣告方式，它的宣告方式比較複雜，我盡量想辦法簡單一點。它完整的宣告長這樣：

```
1 trie<string,null_type, trie_string_access_traits<>, pat_trie_tag, trie_pref
```

第一個參數一定要是字串型態，第二格可以是其他的資料型態 (類似map的感覺)，不用的話就用null_type，第三格是比較函式，第四格是 trie 的種類，最後則是更新方式。一般使用時後三格可以照打，有興趣者可自行研究。那你可以對它做啥操作呢？

首先就是最基本的insert、delete、find 字串，也可以進行合併 (join) 和分割 (split)(要注意的是，join的兩個 trie 一定要第一個字典序比較小，第二個比較大)。它還有一個很特別的操作，就是prefix_range的這個操作，它會回傳一個 pair，代表以某個前綴開頭的字串的開頭的 iterator 和結尾的 iterator。這樣講不太清楚，不如直接看範例吧！

要記得加上#include<ext/pb_ds/assoc_container.hpp>。

```
1  #include <iostream>
2  #include <ext/pb_ds/assoc_container.hpp>
3  #include <ext/pb_ds/detail/standard_policies.hpp>
4  using namespace std;
5  using namespace __gnu_pbds;
6  int main()
7  {
8    trie<string, null_type, trie_string_access_traits<>,pat_trie_tag,trie_pre
9    a.insert("her");
10   a.insert("hers");
11   b.insert("she");
12   b.insert("his");
13   a.join(b);
14   auto temp = a.prefix_range("h");
15   for(auto i = temp.first;i != temp.second;i++)
16   {
17     cout << *i << " ";
18   }
19   cout <<endl;
20   a.delete("hers");
21   a.split("r",b);
22   cout << a.size() << " " << b.size() <<endl;
23 }
```

16.3 習題

其實 pbds 只是一個方便使用的工具，裡面很多的資料結構都可以自己手刻出來，因此我也不知道要放甚麼習題呢！大家好好思考下面的題目吧！

習題 16.3.1: 忍者調度問題 (TIOJ4544 1429)

給定一個 N 個節點的有根樹，每個節點有兩個值 (C 和 L)，再給一個整數 M，你要找到任意個節點，使的他們 C 值的和小於 M，且選定的節點數乘上所有選定節點的某個共同祖先的 L 值最大，輸出該最大值。($N \leq 10^5$，$M \leq 10^9$)

字串 II 17

17.1 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{ //maxn means how many kinds of characters
    node *p,*next[maxn],*fp; //記錄父節點、子節點們和failure pointer
    int index; //第幾個單字的結尾，-1代表不是結尾
    char ch;
    node(){}
    node(node *_p,char _c):p(_p),ch(_c){
        for(int i=0;i<maxn;i++) next[i]=nullptr;
        fp=nullptr; index=-1;
    }
};
```

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(vector<string> &dict){
    node *root=new node(nullptr,'\0');
    root->p=root;
```

```
    for(int i=0;i<dict.size();i++){
        node *ptr=root;
        for(auto &j:dict[i]){
            if(!ptr->next[j-'a'])
                ptr->next[j-'a']=new node(ptr,j);
            ptr=ptr->next[j-'a'];
        }
        ptr->index=i;
    }
    //failure pointer part
    return root;
}
```

failure pointer

為了方便，這裡的 failure pointer 定義和 failure function 會有點差異。而建立的方法我們知道原本 failure function 在計算時可以for迴圈從頭開始一一往後建，也就是說計算某個點時其前面所有 failure function 都要先被算到。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比他上面的節點都已經算完了。說了這麼多，不如直接看程式碼吧（這邊會接續上面的程式碼）：

```
node *build(vector<string> &dict){
    //failure pointer part
    queue<node *> q;
    for(int i=0;i<maxn;i++){ //根及第一層的fp都指向根
        if(root->next[i]){
            root->next[i]->fp=root;
            for(int j=0;j<maxn;j++)
                if(root->next[i]->next[j])
                    q.push(root->next[i]->next[j]);
        }
    }
    while(q.size()){
        node *t=q.front(),*ptr=t->p->fp; q.pop();
        while(ptr!=root&&!ptr->next[t->ch-'a'])
            ptr=ptr->fp;
        if(ptr->next[t->ch-'a']) t->fp=ptr->next[t->ch-'a'];
        else t->fp=root;
        for(int i=0;i<maxn;i++) if(t->next[i]) q.push(t->next[i]);
    }
    return root;
}
```

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 ACsearch(node *r,string &s,vector<string> &dict){
    //root of trie,input string and dictionary
    node *ptr=r;
    int i=0;
    cout<<"string:\tposition:"<<endl;
    while(i<s.size()){
        if(ptr->next[s[i]-'a'])
            ptr=ptr->next[s[i]-'a'],i++;
        else if(ptr==r)
            i++;
        else
            ptr=ptr->fp;
        if(ptr->index!=-1)
            cout<<dict[ptr->index]<<"\t"<<i-(int)dict[ptr->index].size()<<e
    }
    while(ptr!=r){
        ptr=ptr->fp;
        if(ptr->index!=-1)
            cout<<dict[ptr->index]<<"\t"<<i-(int)dict[ptr->index].size()<<e
    } //makes ptr back to root
}
```

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 只在位置 0 和 4，位置 2 的 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”abababb” 的那條路，經過第一個 ab 時在 tire 上有標記結尾，經過第三個 ab 時，因為結尾加上的'\n'不斷失配而往 fp 走，走到指向第一個 ab 的地方時就會被算到。然而經過第二個 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（若是頭尾還是會被算到）。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。因此我們應該要瓊舉這些字串，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！

因此，實作上我們可以記錄所有點被走過幾次，然後在 ACsearch 跑完之後將每個節點被算到的次數不斷向他的 failure pointer 加。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（用 DFS 就好了），如此就可以正確又高效率的完成了！所以來看一下 code 吧！(由於互相包含版紀錄位置很麻煩，我也不知道複雜度是不是好的，所以這邊就放計數就好了)

```
待補
```