APCS 與競賽–
從入門到進階
Book1

ShangJhe Li
WeiHsuan Tai

June 26, 2025

目錄

程式碼

前言

林燈基金會贊助的蘭燈之星 APCS 競賽已經舉辦了數年，我也擔任了許多次的講師，但每年的課程內容都要由講師重新設計，也沒有一套傳承好的教材可以使用。這樣除了讓講師們每年都要花費大量時間準備課程外，還會讓學生們在學習上遇到許多困難，因爲每位講師的教學風格與教材內容都不盡相同。此外，這樣也沒辦法起到傳承的效果，讓新講師可以快速上手。因此，我決定編寫一套教材，讓未來的講師們可以輕鬆一些，也期望未來的講師們能夠在此基礎上進行擴充與修改，讓這套教材能夠更完善，並且能夠持續傳承下去。

本教材係基於李尚哲編寫的《APCS 與競賽入門》一書進行改編，主要是針對蘭燈之星 APCS 競賽的課程進行順序的調整以及擴充，希望能夠加入一些在臺灣大學學到的內容。

演算法競賽的實力養成是一個漫長的過程，還望同學們可以持之以恆，用對方法，開創出屬於你們的一片天地，在全國賽的戰場上向前衝。

Chapter 1

如何寫出可以被看懂的程式

1.1 縮排

1.1.1 爲什麼要縮排

你可能想說：「我程式可以執行就好了啊！」。但是你可以看看這個範例。

```
#include<bits/stdc++.h>
using namespace std;
int main(){
int n,a=0,sum=0;
cin>>n;
for(int i=0;i<n;++i){
cin>>a;
if(a<120)
sum+=a;
}
cout<<sum<<"\n";
}
```

Code 1.1: 縮排的錯誤示範

請問你能在 20 秒內了解這個程式在幹嘛嗎？如果可以，那你就比筆者強大了^ ^。(至少在程式碼識讀上)

即便如此，如果你遇到問題，被你問到的人都絕對不會想要看到這樣的程式碼，你如果傳這種給其他人，大概有 99.99% 的機率被退回。

1.1.2 怎麼做比較好

程式撰寫風格是非常自由，但一個大原則就是要有可讀性。我所使用的縮排原則是，只要遇到函式 (Function)、for, while 迴圈、if, switch 判斷式，都會在接下來多空 2-4 個空格 (在 vscode 裡面)。

以下是一個範例。

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

int main() {
    int a = 0, sum = 0;
    cin >> n;
    for(int i = 0; i < n; ++i) {
        cin >> a;
        if(a < 120)
            sum += a;
    }
    cout << sum << "\n";
}
```

Code 1.2: 縮排的合理示例

在這樣的階層式架構下，只要是 APCS 觀念題有拿到 3, 4 級分的人應該都可以快速理解這段程式碼。

Tip: 程式碼撰寫上應以縮排的方式增加可讀性。在所有的程式碼中應用這樣的習慣，將有助於維護你寫的程式碼們。

1.2 空格

1.2.1 聲明

以下的建議僅供參考，空格的方式一直以來都是吵不完的議題，我將說明爲何要這樣做。請依照你的個人需求選擇你的空格方式。(注意可讀性即可)

1.2.2 一般情況

因爲會把程式碼的字體縮小，方便我閱讀更多行的程式碼，所以我會在運算符號間加上空格，這樣可以更容易懂。

```cpp
void solve() {
    int a = 0, sum = 0;
    cout << sum << "\n";
}
```

Code 1.3: 我的空格習慣 (一般情況)

```cpp
void solve(){
    int a=0,sum=0;
    cout<<sum<<"\n";
}
```

Code 1.4: 不加上空格的版本

應該可以看出來我的程式碼比較壓縮，如果不喜歡這一種可以用第二種。基本上都可以被廣泛接受。

1.2.3 特殊情況

看到一般情況應該就會想到有特殊的情形。而爲了可讀性，我在下列情形會加上空格。

1. 使用 if 而不想要換行時。
2. 把兩個指令塞在同一行時。
3. 運用指標時。
4. 運算式太複雜時。

以下程式是範例。

```
void solve(){
    if(a<0) a=0;
    b++; c++;
    d->sum += e->sum;
    cout<<a + (b-c) - d->sum<<"\n";
}
```

Code 1.5: 特殊情況們

不過盡可能避免這樣的特殊情況，因爲這樣仍會讓程式碼變得難以閱讀。

Tip: 請依照你習慣的方式，有一致性與可讀性的加上空格。你會發現本章節強調可讀性，因爲我實在是看過好多沒有辦法閱讀的程式，我還必須手動排版，造成每個人時間的浪費。

1.3 變數命名

這個章節我們要來探討變數名稱的意義。在業界，變數名稱要盡可能包含所有資訊，讓共事的夥伴可以藉由變數名稱讀懂程式碼。然而，有些人認爲競賽只要用盡可能簡潔的變數名稱就好。

1.3.1 在簡短與明確中抉擇

我承認競賽上不太可能用向實務一樣長的變數名稱，因爲競賽必須與時間賽跑。但太過簡短，甚至沒有意義的名稱可能導致除錯困難，因此會有一些折衷的方案，希望盡可能同時解決兩邊的問題。

1.3.2 縮寫

在命名變數時，使用英文縮寫讓程式碼變的簡短，是一個常見的方式。以下是一個簡單的表格，列舉出我可能會用到的縮寫代表的意義。

縮寫	ct	isv	mx	mn	idx	num
意義	count	is valid	max	min	index	number

1.3.3 組合型單字

對於組合型單字，我們可以使用兩種方式命名。

1. 加上底線，Ex: `item_number`。
2. 除了第一個單詞，後面的單詞第一個字母大寫，Ex: `itemNumber`

1.3.4 其他

其他的命名方式可以參考 Code Complete 第 11 章「變數名稱的力量」。

1.4 範例與練習

Example 1.4.1. 快速冪。

```
using ll=long long;
const ll MOD=1e9+7;

ll POW(ll a,ll x){
    ll ret=1;
    while(x>0){
        if(x%2==1) ret=ret*a%MOD;
        a=a*a%MOD;
        x/=2;
    }
    return ret;
}
```

Code 1.6: 排版範例

Problem 1.4.2. 請對以下程式碼排版。

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

using ll=long long;
const ll INF=0x3f3f3f3f;
const ll LINF=0x3f3f3f3f3f3f3f3f;
const ll MOD=1e9+7;
```

```
8
9  int n;
10 int dp[20][(1<<16)+10];
11 int a[20];
12 int main(){
13 ios::sync_with_stdio(0);cin.tie(0);cin>>n;
14 for(int i=1;i<=n;++i) cin>>a[i];
15 for(int i=1;i<=n;++i) dp[i][0]=0;
16 for(int i=1;i<(1<<n);++i){
17 for(int k=1;k<=n;++k){
18 for(int l=1;l<=n;++l){
19 if( (i-(1<<l))^i == (1<<k) ){dp[k][i];
20 }}
21 }}
22 }
```

1.5 更進一步

可以去看這一部影片 https://www.youtube.com/watch?v=CFRhGnuXG-4 ，重點是不要有太多層的迴圈。

Example 1.5.1. 五層的範例

```
1  bool check(int a,int b){
2      if(a+b>100){
3          if(a-b<50){
4              for(int i=1;i<a;i*=2;++i){
5                  if(a*i%1000==200){
6                      for(int j=0;j<a+b;++j){
7                          a=(a+b)%100;
8                          b=(a-b)%100;
9                      }
10                 }
11                 return a%2;
12             }
13         }else{
14             return a%2;
15         }
16     }else{
17         return b%2;
18     }
19 }
```

Code 1.7: 太多層的例子

同樣的程式碼可以寫成這樣，會比較容易掌握你要表達的意義。

```
1  void modify(int &a,int &b){
2      for(int j=0;j<a+b;++j){
3          a=(a+b)%100;
4          b=(a-b)%100;
5      }
6  }
```

```
7
8  bool check(int a,int b){
9      if(a+b<=100){
10         return b%2;
11     }
12
13     if(a-b>=50){
14         return a%2;
15     }
16
17     for(int i=1;i<a;i*=2;++i){
18         if(a*i%1000==200){
19             modify(a,b);
20             return a%2;
21         }
22     }
23 }
```

Code 1.8: 改進的版本

除此之外，我們也可以嘗試遵守一個原則，就是一個函數只做一件事情，並且不要超過 30 行，這樣會讓程式碼更容易閱讀。

Chapter 2

基礎資料結構

2.1 STL

雖然至今爲止我們都當作你們已經會使用 STL 了。不過可能有一些同學甚至還不知道我在講什麼。所以我決定加上這個小節，告訴大家什麼是 STL。

不過其實這個章節的作者是戴偉璿

2.1.1 什麼是 STL

想像一個狀況，你今天想要使用電腦，那你會怎麼做？零零散散購買主機板、CPU、記憶體、顯示卡、硬碟等等自己組裝固然可以，但直接購買一台完整的電腦是否比較方便？STL 就是這樣的東西。STL 是 C++ 的標準模板庫（Standard Template Library），它提供了一些常用的資料結構，讓我們可以更方便地使用 C++ 進行程式設計。

STL 的目的是爲了讓我們可以更快速地開發程式，而不需要從頭開始實作所有的資料結構和演算法。白話來說，他其實就是把東西打包好方便我們使用。

以下逐一介紹 STL 的各種資料結構。

2.1.2 string

string 是 C++ 中用來處理字串的資料結構，你可以把他看成是一個字元陣列，但它提供了更多的功能，例如自動管理記憶體、支援字串操作等。string 的使用方式與陣列類似，但它有一些特別的功能，例如可以自動調整大小、支援字串連接等。

以下是一些常用的 string 操作：

```
    // declare 宣告
    string s;

    // give value 輸入
    cin>>s;
    getline(cin,s);

    // 取得某一個位置的值，和陣列一樣
    s[5];

    // size 長度
    // good
    cout<<s.size()<<s.length();
    // bad
    cout<<strlen(s);

    // clear 清空
    s.clear();

    // 連接
    string s1="abc",s2="def";
    s=s1+s2;

```

Code 2.1: string 用法

在使用 string 時，請注意以下幾點：

- string 的大小可以動態調整，不需要預先指定大小。
- string 支援字串操作，例如連接、比較、搜尋等。
- 在取得字串長度時，不建議使用 strlen()，因爲他的的時間複雜度是 $\mathcal{O}(n)$，而 string 的 size() 和 length() 方法的時間複雜度是 $\mathcal{O}(1)$。
- string 的字元編碼是 UTF-8，所以可以處理多種語言的字串。

2.1.3 vector

vector 是 C++ 中用來處理動態陣列的資料結構，它提供了自動管理記憶體的功能，並且支援隨機存取。vector 的使用方式與陣列類似，但它有一些特別的功能，例如可以自動調整大小、支援插入和刪除等。

以下是一些常用的 vector 操作：

```
    // declare 宣告
    vector<int> v;
    vector<char> v[10005];// 這樣宣告的是10005個vector，可以看成是二維陣列
    // 而不是內有10005個元素的vector
    // 如果需要宣告有10005個元素的vector
    vector<int> v(10005);
    vector<int> v(10005,-1); // 所有元素皆爲-1

    // add element 新增元素
    // faster 因爲使用建構式而非複製。
    v.emplace_back(x);
    // slower
    v.push_back(x);

    // erase back 從最後一個移除元素
    v.pop_back()

    // go through 遍歷
    for(auto i:v){
        // i:the elements of vector v
    }

    // size
    v.size()

    // clear
    v.clear();

```

Code 2.2: vector 用法

2.1.4 stack

堆疊（stack）是一種先進後出（Last In First Out, LIFO）的資料結構。在 C++ 中，堆疊通常使用 stack 容器來實現。堆疊的操作主要包括入堆疊（push）、出堆疊（pop）和查詢頂部元素（top）。堆疊的使用方式與陣列類似，但它有一些特別的功能，例如只能在頂部進行插入和刪除操作。可以把堆疊想像成一堆疊起來的盤子，你只能放新的盤子在最上面或是從最上面拿走盤子。

以下是一些常用的 stack 操作：

```
    // declare 宣告
    stack<int> st;
    stack<char> st;
    // faster
    stack<int,vector<int>> st;

    // add element to back 就是 push_back()
    st.push(x);
    st.emplace(x); // emplace_back() in vector

```

```
11      // query the last element 頂部元素
12      st.top();
13
14      // move out the last element 移除頂部元素
15      st.pop();
16
17      // check is empty
18      st.empty();
19
20      // size
21      st.size()
22
```

Code 2.3: stack 用法

值得注意的是，stack 的操作 vector 也可以實現，但他的速度比 vector 慢許多。

2.1.5 queue

佇列（queue）是一種先進先出（First In First Out, FIFO）的資料結構。在 C++ 中，佇列通常使用 queue 容器來實現。佇列的操作主要包括入佇列（push）、出佇列（pop）和查詢前端元素（front）。和堆疊不同，佇列的操作是先進先出，你可以把佇列想像成一個排隊的隊伍，最先到的人最先被服務，最慢到的人只能排在最後面。

以下是一些常用的 queue 操作：

```
1       // declare
2       queue<int> qu;
3
4       // add element to back
5       qu.push(x);
6
7       // query the first element
8       qu.front();
9
10      // move out the last element
11      qu.pop();
12
13      // check is empty
14      qu.empty();
15
16      // size
17      qu.size()
18
```

Code 2.4: queue 用法

2.1.6 deque

雙端佇列，其實就是堆疊和佇列的結合，你可以在兩端進行插入和刪除操作。在 C++ 中，雙端佇列通常使用 deque 容器來實現。

```
1  // declare
2  deque<int> dq;
3
4  // add element to back
5  dq.push_back(x);
6  dq.push_front(x);
7
8  // query the last element
9  dq.front();
10 dq.back();
11
12 // move out the last element
13 dq.pop_back();
14 dq.pop_front();
15
16 // 也可以像陣列一樣用
17 dq[10];
18
19 // check is empty
20 dq.empty();
21
22 // size
23 dq.size()
24
```

Code 2.5: deque 用法

2.1.7 priority queue

顧名思義，優先佇列（priority queue）是一種特殊的佇列。他的操作看似與一般佇列相同–從後面加入元素，從前面提出。但這個佇列會維護一個 heap 結構，確保你每次提出的元素都是目前佇列中最大的（或最小的）元素。

一言以蔽之，他的本質上是一個使用 vector 維護的 heap 資料結構。

```
1  // declare (最大堆)
2  priority_queue<int> pq;
3  // 最小堆
4  priority_queue<int,vector<int>,greater<int>> pq;
5
6  // add element into the pq
7  pq.push(x);
8
9  // query the element (對最大堆來說就是最大值)
10 // (對最小堆來說就是最小值)
11 pq.top();
12
13 // move out the max/min element
```

```
    pq.pop();

    // check is empty
    pq.empty()

    // size
    pq.size()

```

Code 2.6: priority queue 用法

2.1.8 map

有時候，我們的資料對應到的不一定是編號。舉例而言，我有一大堆員工，如果要使用員工姓名快速查詢某個員工的工號該怎麼做？線性搜索當然是可行的，但這樣時間複雜度很高。而 C++ 的 STL 提供了一個稱爲 map 的資料結構，可以在 $\mathcal{O}(\log n)$ 的時間內完成查詢。map 是一種關聯容器，它將鍵（key）和值（value）對應起來。在 map 中，每個鍵都是唯一的，並且可以用來快速查詢對應的值。map 的使用方式與陣列類似，但它有一些特別的功能，例如可以使用任意類型的鍵、支援自動排序等。

題外話，map 與 set 都是使用紅黑樹實作的資料結構，那是一種稱爲平衡樹的資料結構。未來在進階資料結構時會提到一種稱爲 Treap 的平衡樹。

```
// declare
map<int,int> mp;
// 兩個可以不一樣
map<string,int> mp;

// add element into mp (key 爲 y，value 爲 x)
mp.insert({y,x});
mp.emplace(y,x);
mp[y]=x;

// query the element
mp[y];
mp.begin();
mp.end();

// move out the element (key 爲 x)
mp.erase(x);

// check the element with key x is present in the map container

// return int (但只有 0, 1)
mp.count(x)

// return iterator
map<int,int>::iterator it=mp.find(x)

// check is empty
mp.empty()

```

```
30 // size
31 mp.size()
```

Code 2.7: map 用法

2.1.9 set

set 的中文就是集合，他也就是一個集合，所有的操作基本上都是針對集合的操作。

```
1  // declare
2  set<int> S;
3
4  // add element to back
5  S.insert(x);
6  S[y]=x;
7
8  // query the element
9  S[y]
10 S.begin();
11 S.end();
12
13 // check the x is in the set
14 // return iterator
15 set<int>::iterator it=S.find(x)
16 //return
17 S.count(x)
18
19 // 找出不小於x的最小值
20 it=S.lower_bound(x);
21 // 找出大於x的最小值
22 it=S.upper_bound(x);
23 // move out the last element
24 S.erase(x);
25
26 // check is empty
27 S.empty()
28
29 // size
30 S.size()
```

Code 2.8: set 用法

2.1.10 bitset

bitset 可以看成是一個 bool 陣列，但他在做位元運算時會比 bool 陣列快上許多。有時候會用到，但真的很少需要它。

```
1 // declare
2 bitset<100010> bt;
3
```

```cpp
// give value
string str;
cin>>str;
bt=bitset<100010>(str);

// find one index 存取
bt[5];

// set all element to 0
bt.reset();

// set all element to 1
bt.set();

// bitwise operation
bitset<10> a,b;
for(auto &i:a) cin>>i;
for(auto &i:b) cin>>i;

cout<<a&b<<"\n";
cout<<a|b<<"\n";
cout<<a^b<<"\n";
```

Code 2.9: bitset 用法

2.1.11 小節

這個章節基本上都是背誦的內容，但是其實用性非常高，建議同學藉由實作上多多使用這些東西來熟悉 STL 們。

Tip: 不要用背的，而是藉由不斷的使用讓大腦自然地記起來。

2.1.12 範例與練習

Problem 2.1.1. LeetCode 3. Longest Substring Without Repeating Characters

題目敘述

給你一個字串 s，找出最長的子字串滿足裡面沒有相同的字母。

輸入說明

$0 \leq s.length \leq 5 \times 10^4$ s 裡面可能有空格。

輸出說明

輸出長度。

範例測試

範例輸入 1	範例輸出 1
abcabcbb	3
範例輸入 2	範例輸出 2
bbbbb	1
範例輸入 3	範例輸出 3
pwwkew	3

2.2 實作 STL 的資料結構

既然官方的函式庫裡面有這樣的東西，就表示我們可以做一個出來。

2.2.1 實作 vector

核心概念：倍增法 (Doubling Strategy)

在許多動態資料結構中，我們需要在執行期間擴充儲存空間。以 Vector (或 C++ 的 std::vector) 爲例，當我們不斷使用 push_back 新增元素時，其內部預先分配的記憶體空間總有被用完的時候。

這時，我們必須：

1. 配置一塊**更大的**新記憶體。
2. 將舊記憶體中的所有元素**複製**到新記憶體。
3. **釋放**舊的記憶體空間，以防記憶體洩漏。
4. 將指標指向新的記憶體位置。

關鍵問題是：新記憶體應該要多「大」?

- **線性增長 (不好)**：如果每次都只增加一個固定的量 (例如 capacity + 10)，隨著元素數量 n 的增長，重新配置記憶體的次數會越來越頻繁，導致 push_back 的平均時間複雜度變爲 $O(n)$。
- **倍增法 (Amortized O(1))**：如果我們每次都將空間擴充爲原來的兩倍 (**空間** $\times 2$)，雖然單次擴充的成本很高 (需要 $O(n)$ 的時間複製元素)，但這種情況很少發生。經過均攤分析 (Amortized Analysis)，可以證明，在經過一系列 push_back 操作後，每個操作的**平均時間複雜度爲常數時間** $O(1)_{\textbf{amortized}}$。這是一種空間換取時間的經典策略，確保了 Vector 的效能。

Vector.h

這是一個簡化版的 Vector 結構，用於儲存整數。

```
struct Vector {
    // 建構子：初始化一個大小爲 _sz 的 Vector
    // 容量 (capacity) 也會被設爲 _sz
    Vector(int _sz = 0) : sz(_sz), capacity(_sz) {
        if (sz != 0) value = new int[sz];
        else value = nullptr;
    }
    // 解構子：釋放動態配置的記憶體
    ~Vector() {
        if (value != nullptr) delete [] value;
    }
    // 調整 Vector 的大小
    Vector& resize(int new_size);
    // 在尾端新增一個元素
    Vector& push_back(int val);
    // 移除尾端的元素
    Vector& pop_back();
    // 運算子重載，提供類似陣列的存取方式
    int& operator[](int index) { return value[index];}

    int *value;     // 指向動態配置陣列的指標
    int sz;         // 目前儲存的元素數量
    int capacity;   // 目前已配置的記憶體容量
};
```

Code 2.10: Vector

Vector 實作

以下是 push_back 和 resize 的完整實作，這是 Vector 的核心。

```
#include "Vector.h"

// push_back 是倍增法的關鍵所在
Vector& Vector::push_back(int val) {
    // 1. 檢查容量是否已滿
    if (sz == capacity) {
        // 2. 計算新容量。如果原容量爲 0，則新容量設爲 1，否則加倍。
        int new_capacity = (capacity == 0) ? 1 : capacity * 2;

        // 3. 配置新記憶體
        int *new_value = new int[new_capacity];

        // 4. 將舊資料複製到新記憶體
        for (int i = 0; i < sz; ++i) {
            new_value[i] = value[i];
        }

        // 5. 釋放舊記憶體
        if (value != nullptr) {
            delete[] value;
        }

```

```
23          // 6. 更新指標和容量
24          value = new_value;
25          capacity = new_capacity;
26      }
27
28      // 7. 在尾端加入新元素，並更新大小
29      value[sz] = val;
30      sz++;
31
32      return *this;
33  }
34
35  // pop_back 較爲簡單，只需減少 size 即可
36  // 爲了效率，我們不縮減容量 (std::vector 也是如此)
37  Vector& Vector::pop_back() {
38      if (sz > 0) {
39          sz--;
40      }
41      return *this;
42  }
43
44  // resize 處理大小變更，可能需要擴充或只是修改 sz
45  Vector& Vector::resize(int new_size) {
46      // 如果新大小超過目前容量，則需要重新配置記憶體
47      if (new_size > capacity) {
48          int new_capacity = new_size; // 直接配置所需大小
49          int *new_value = new int[new_capacity];
50
51          // 複製舊資料
52          for (int i = 0; i < sz; ++i) {
53              new_value[i] = value[i];
54          }
55
56          if (value != nullptr) {
57              delete[] value;
58          }
59
60          value = new_value;
61          capacity = new_capacity;
62      }
63      // 對於新增加的空間，可以選擇性地初始化爲 0
64      for(int i = sz; i < new_size; ++i) {
65          value[i] = 0;
66      }
67
68      // 最後更新 size
69      sz = new_size;
70      return *this;
71  }
```

Code 2.11: Vector.cpp 的建議實作

應用

即便我們知道有函式庫可以用，倍增法依舊是一個很有用的技巧，尤其是在處理**未知範圍的搜索**問題時。

此外，有了動態陣列 (如 std::vector) 之後，**堆疊 (Stack)** 與 **佇列 (Queue)** 就可以被輕易地實作出來了。

2.2.2 實作 Stack

首先我們來實作堆疊。你會發現，只要利用 std::vector 內建的功能，就可以非常方便地模擬出堆疊的結構。我們將 vector 的**尾端**視爲堆疊的**頂端 (top)**。

- push(value): 將新元素放到堆疊頂端 → vector.push_back(value)
- pop(): 移除堆疊頂端的元素 → vector.pop_back()
- top(): 查看堆疊頂端的元素 → vector.back()

```
#include <iostream>
#include <vector>

struct Stack {
    std::vector<int> values;
    Stack& push(int value);
    Stack& pop();
    int top();
    int size();
};

// 將元素推入堆疊頂端
Stack& Stack::push(int value) {
    // your code below
    values.push_back(value);
    // your code above
    return *this;
}

// 從堆疊頂端移除元素
Stack& Stack::pop() {
    // your code below
    // 加上 .empty() 檢查可以讓程式更穩健，避免在空堆疊上操作
    if (!values.empty()) {
        values.pop_back();
    }
    // your code above
    return *this;
}

// 回傳堆疊頂端的元素值
int Stack::top() {
    // your code below
```

```
34      // 如果堆疊是空的，回傳 -1，否則回傳最尾端的元素
35      if (values.empty()) {
36          return -1;
37      }
38      return values.back();
39      // your code above
40  }
41
42  int Stack::size() {
43      return values.size();
44  }
45
46  const int PUSH{1};
47  const int POP{2};
48  const int TOP{3};
49
50  int main(void) {
51      // 關閉 C++ stream 與 C stdio 的同步，加速 cin/cout
52      std::ios_base::sync_with_stdio(false);
53      std::cin.tie(NULL);
54
55      int operation_num;
56      std::cin >> operation_num;
57
58      Stack st;
59      for (int i{0}; i < operation_num; ++i) {
60          int op, value;
61          std::cin >> op;
62          switch (op) {
63          case PUSH:
64              std::cin >> value;
65              st.push(value);
66              std::cout << st.size() << "\n";
67              break;
68          case POP:
69              st.pop();
70              break;
71          case TOP:
72              std::cout << st.top() << "\n";
73              break;
74          }
75      }
76  }
```

Code 2.12: 用 std::vector 實作 Stack

2.2.3 實作 Queue

接著我們來實作佇列 (Queue)。佇列的特性是「先進先出 (FIFO)」。如果直接使用 `std::vector`，從尾端推入 (`push_back`) 很有效率，但從頭部彈出 (`erase(begin())`) 的時間複雜度是 $O(N)$，因爲需要移動後方所有元素，這在資料量大時會非常慢。

爲了解決這個問題，我們將利用 `std::vector` 模擬一個更高效的結構：**環狀陣列**

(Circular Array)。

核心概念：環狀陣列

我們使用兩個指標 `l` (left/front) 和 `r` (right/rear) 來標記佇列的頭部與尾部。

- `l`: 指向佇列的第一個元素。
- `r`: 指向佇列**最後一個元素的下一個位置**。
- `sz`: 佇列中實際的元素數量。
- `capacity`: 底層 `vector` 的總容量。

當 `l` 或 `r` 指標移動到陣列末端時，我們會讓它「繞回」到陣列的開頭，這就是「環狀」的概念，可以透過模數運算 (%) 實現。

當佇列已滿 (`sz == capacity`) 且需要再次推入元素時，我們會進行**擴容**，也就是題目提示的 `resize`。擴容時，我們會將環狀陣列中的元素「拉直」，並複製到一個更大的新陣列中。

程式實作

```cpp
#include <iostream>
#include <vector>

struct Queue {
    // 建構子：初始化所有成員變數
    Queue(): sz(0), l(0), r(0), capacity(0) {}
    std::vector<int> values;
    int l, r, sz, capacity;
    void set_capacity(int new_capacity);
    Queue& push(int val);
    Queue& pop();
    int front();
    int size() { return sz; }
};

// 重新設定容量，並將環狀陣列拉直
void Queue::set_capacity(int new_capacity) {
    std::vector<int> new_values(new_capacity);
    // 將舊陣列中的元素按順序複製到新陣列
    for (int i = 0; i < sz; ++i) {
        new_values[i] = values[(l + i) % capacity];
    }
    // 更新 vector、指標和容量
    values = new_values;
    capacity = new_capacity;
    l = 0;       // 新的頭部在索引 0
    r = sz;      // 新的尾部在 sz 的位置
}
```

```
29
30 // 將元素推入佇列尾端
31 Queue& Queue::push(int val) {
32     // Your code below
33     // 如果佇列已滿，進行擴容 (使用倍增法)
34     if (sz == capacity) {
35         int new_capacity = (capacity == 0) ? 1 : capacity * 2;
36         set_capacity(new_capacity);
37     }
38     // 將新元素放入尾端
39     values[r] = val;
40     // 更新尾端指標，並用 % 實現環狀移動
41     r = (r + 1) % capacity;
42     // 增加實際大小
43     sz++;
44     // Your code above
45     return *this;
46 }
47
48 // 從佇列頭部彈出元素
49 Queue& Queue::pop() {
50     // Your code below
51     // 如果佇列內沒有任何元素，跳過指令
52     if (sz == 0) {
53         return *this;
54     }
55     // 更新頭部指標，並用 % 實現環狀移動
56     l = (l + 1) % capacity;
57     // 減少實際大小
58     sz--;
59     // Your code above
60     return *this;
61 }
62
63 // 取得頭部元素的值
64 int Queue::front() {
65     // Your code below
66     // 如果佇列是空的，回傳 -1
67     if (sz == 0) {
68         return -1;
69     }
70     // 回傳頭部指標所指的元素
71     return values[l];
72     // Your code above
73 }
74
75 const int PUSH{1};
76 const int POP{2};
77 const int FRONT{3};
78
79 int main(void) {
80     std::ios::sync_with_stdio(0);std::cin.tie(0);
81     int operation_num;
82     std::cin >> operation_num;
83
84     Queue st;
```

```
85      for (int i{0}; i < operation_num; ++i) {
86          int op, value;
87          std::cin >> op;
88          switch (op) {
89          case PUSH:
90              std::cin >> value;
91              st.push(value);
92              std::cout << st.size() << "\n";
93              break;
94          case POP:
95              st.pop();
96              break;
97          case FRONT:
98              std::cout << st.front() << "\n";
99              break;
100         }
101     }
102 }
```

Code 2.13: 用環狀陣列實作 Queue

2.2.4 實作 Priority Queue

堆積是一種特殊的樹狀資料結構，它在陣列的基礎上，透過父子節點的索引關係來模擬樹的行爲，因此效率很高。以下我們以 ** 最大堆積 (Max Heap)** 爲例，其必須滿足以下性質：

1. **堆積性質 (Heap Property)**: 父節點 (Parent Node) 的值總是**大於等於**其子節點 (Child Nodes) 的值。這確保了最大值永遠在樹的根節點。

2. **結構性質 (Shape Property)**: 堆積是一個**完全二元樹 (Complete Binary Tree)**。這意味著樹的每一層都是滿的，除了最底層；且最底層的節點都盡量靠左對齊。這個性質讓我們能用陣列來緊湊地儲存它。

二元樹 (Binary Tree) 是一種每個節點最多只能有兩個子節點的樹狀結構。

Heap 主要操作

Build Heap (建堆): 將一個無序的陣列轉換成滿足堆積性質的結構。這個過程也稱爲 **Heapify**。

Insert (插入): 在堆積中插入一個新元素，同時維持堆積性質。

Extract Max (取出最大值): 移除並回傳堆積中的最大元素（即根節點），同樣要維持堆積性質。

操作演算法詳解

爲了實現上述操作，我們需要兩個核心的輔助函式：sift_down(下沉)和 sift_up(上浮)。

Sift Down (下沉) 當某個節點的值小於其子節點，破壞了堆積性質時，我們讓它與其**較大的子節點**交換位置，並一路向下重複此過程，直到它不再小於其子節點，或成爲葉節點爲止。這個操作是 **Build Heap** 和 **Extract Max** 的基礎。

Sift Up (上浮) 當我們在堆積末端加入一個新元素時，這個新元素可能比其父節點大。我們讓它與其父節點交換位置，並一路向上重複此過程，直到它的值小於等於其父節點，或已到達根節點爲止。這個操作是 **Insert** 的基礎。

程式碼實作與詳解

接下來，我們將實作一個最大堆積。我們使用 std::vector 來儲存資料，並透過索引計算來模擬父子關係。

- 索引爲 'i' 的節點：
- 其左子節點索引爲 '2*i + 1'
- 其右子節點索引爲 '2*i + 2'
- 其父節點索引爲 '(i - 1) / 2'

下面的程式碼模板已經提供了這些索引計算的輔助函式。

```
#include <iostream>
#include <vector>
#include <functional> // for std::function
#include <algorithm>  // for std::swap

inline int left_child(int node) {
    return (node << 1) | 1;
}

inline int right_child(int node) {
    return (node << 1) + 2;
}

inline int parent(int node) {
    return (node - 1) >> 1;
}

class Heap {
public:
    Heap(const std::vector<int> &elements) : values(elements) {
        heapify();
```

```
22      }
23      void heapify();
24      void push(int val);
25      int pop_top();
26  private:
27      // 輔助函式：讓節點 i 下沉以維持堆積性質
28      void sift_down(int i);
29      // 輔助函式：讓節點 i 上浮以維持堆積性質
30      void sift_up(int i);
31      std::vector<int> values;
32  };
33
34  // --- Helper Functions ---
35  void Heap::sift_down(int i) {
36      int max_index = i;
37      int l = left_child(i);
38      // 檢查左子節點是否存在且大於目前節點
39      if (l < values.size() && values[l] > values[max_index]) {
40          max_index = l;
41      }
42      int r = right_child(i);
43      // 檢查右子節點是否存在且大於目前最大值節點
44      if (r < values.size() && values[r] > values[max_index]) {
45          max_index = r;
46      }
47      // 如果最大值不是目前節點 i，則交換並遞迴下沉
48      if (i != max_index) {
49          std::swap(values[i], values[max_index]);
50          sift_down(max_index);
51      }
52  }
53
54  void Heap::sift_up(int i) {
55      // 當節點不是根節點且比父節點大時，持續上浮
56      while (i > 0 && values[i] > values[parent(i)]) {
57          std::swap(values[i], values[parent(i)]);
58          i = parent(i);
59      }
60  }
61
62  // --- Public Methods ---
63  void Heap::heapify() {
64      // Your code below
65      // 從最後一個非葉節點開始，由下往上對每個節點執行 sift_down
66      // 最後一個元素的索引是 values.size() - 1
67      // 其父節點就是最後一個非葉節點
68      for (int i = (values.size() / 2) - 1; i >= 0; --i) {
69          sift_down(i);
70      }
71      // Your code above
72  }
73
74  void Heap::push(int val) {
75      // Your code below
76      // 1. 將新元素加入到陣列尾端
77      values.push_back(val);
```

```
78      // 2. 對新元素執行 sift_up，以維持堆積性質
79      sift_up(values.size() - 1);
80      // Your code above
81  }
82
83  int Heap::pop_top() {
84      // Your code below
85      if (values.empty()) {
86          return -1; // 或拋出異常
87      }
88      // 1. 保存根節點的值
89      int top_element = values[0];
90      // 2. 將最後一個元素移到根節點
91      values[0] = values.back();
92      // 3. 移除陣列的最後一個元素
93      values.pop_back();
94      // 4. 如果堆積非空，對新的根節點執行 sift_down
95      if (!values.empty()) {
96          sift_down(0);
97      }
98      // Your code above
99      return top_element;
100 }
```

Code 2.14: 用 Vector 實作 Heap

2.2.5 實作 Set 與 Map

Set 和 Map 是兩個非常常用的資料結構，分別用於儲存唯一元素和鍵值對。它們通常使用平衡二元搜尋樹 (如紅黑樹) 來實現，以確保操作的時間複雜度爲 $O(\log n)$。所以這對目前的你們來說，實作起來會有點困難。在競賽上會使用 Treap 取代。Treap 將會在後面介紹。

2.3 並查集

2.3.1 前言

並查集又稱爲 DSU(Disjoint Set Union)，顧名思義就是專門處理合併與查詢集合的資料結構。

我們常常會遇到這樣的問題，兩個人是否同組，合併兩個組別的問題。如果使用 set 來執行操作，則查詢需要 $O(\log(n))$，合併需要 $O(n\log(n))$。而如果使用 DSU，則可以將查詢的操作時間複雜度降到 $O(\alpha(n))$，其中 $\alpha(n)$ 是阿克曼函數 $A(n,n)$ 的反函數，比較直觀的說法就是他在正常資訊競賽的變數範圍內都不會大於 4，因此可以視爲常數。

2.3.2 概念

我們幾乎都是使用樹的概念存放 DSU，因此如果你還不知道樹是什麼東西，你可以先翻到樹的介紹。

DSU 裡面的每個元素都會存放著一個訊息，就是他的上面有誰。你可以想像成，你上面是你的學長姐，你學長姐上面是你們的老師，你們老師的上面是學校校長，而你們都屬於宜蘭高中。因此，我們可以畫這樣的圖以供參考。

查詢

圖中的箭頭表示該節點的上面是誰。查詢時，只要不斷的往上找就可以找到最頂部的節點，如果兩個節點的最頂部相同，則我們稱這兩個節點在同一個集合中。

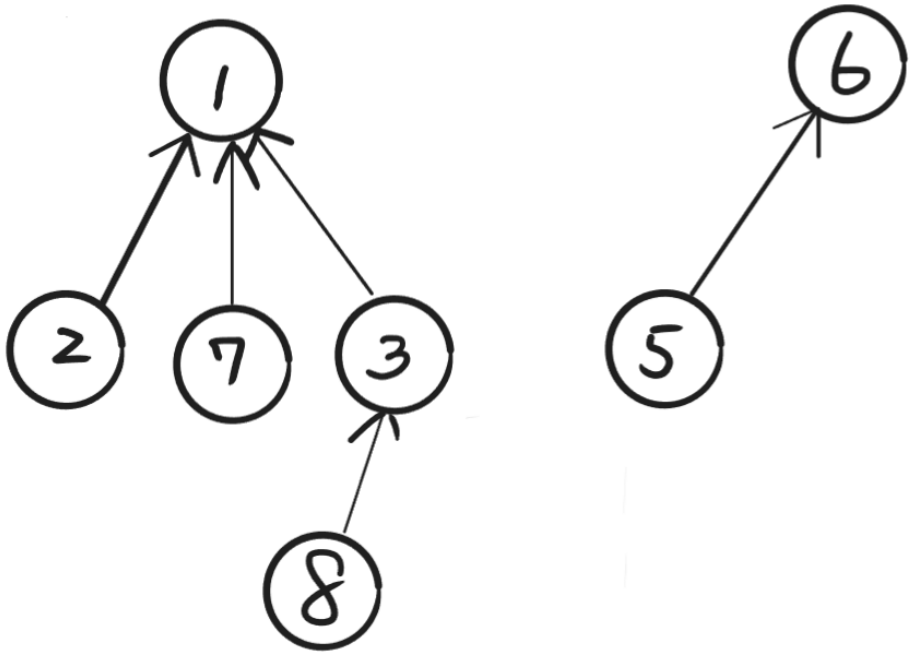


Figure 2.1: 並查集示意圖

合併

合併兩個組別 (a,b) 非常簡單，我們只要將節點 a 的頂，與節點 b 的頂結合就可以了，也就是說，將 a 的頂設爲 b 的頂的上面，或是反過來，都可以完成。

2.3.3 啓發式合併

啓發式合併藉由將大的集合的頂，設在小的集合上面，以增進運行效率。原本沒有啓發式合併的並查集，期望複雜度爲 $O(n)$，加上啓發式合併之後可以降到 $O(\log(n))$。

2.3.4 路徑壓縮

如果我們在查詢的時候，將所有經過的節點直接連上最頂的那一個，我們可以進一步改善複雜度到平均 $O(\alpha(n))$，在競賽中可以視爲常數 4。

2.3.5 實作

實作上，因爲通常加上路徑壓縮就足以勝任大部分任務，所以我們可以省略啓發式合併。

```cpp
const int N=100010;
int dsu[N];

int find(int a){
    if(dsu[a]==a){
        return a;
    }
    // 路徑壓縮
    return dsu[a]=find(dsu[a]);
}

int Union(int a,int b){
    dsu[find(a)]=dsu[find(b)];
}
```

Code 2.15: DSU 實作

2.3.6 範例與練習

Problem 2.3.1. 請實作啓發式合併。

Problem 2.3.2. UVA793 A - Network Connections

題目敘述

有 n 台電腦，編號爲 1 至 n，接下來有若干個指令，指令爲`c a b`代表連接 a 和 b，指令爲`q a b`代表詢問 a 與 b 是否相連，最後請輸出總共有幾次詢問是得到「相連」的答案，以及總共有幾次詢問是得到「不相連」的答案。

範例測試

範例輸入 1	範例輸出 1
10 c 1 5 c 2 7 q 7 1 c 3 9 q 9 6 c 2 5 q 7 5	1,2

小小細節

UVA 因爲很老，有一些奇怪的規定，例如最後一行不要換行等等，總之要遵守才會得到 *AC*。

Problem 2.3.3. ZJ d831 畢業旅行

題目敘述

多年來友情的羈絆，終於將在這畢業的季節開花結果。

這幾天，班上同學們無時無刻都熱烈討論著畢業旅行的地點。小明說，如果要去六福村，可以順便去小人國；小美說，如果去了恆春的話，墾丁就在幾十公里外了，一定也要去玩；小華表示，小鬼湖跟大鬼湖好像很近，似乎都是很有趣的地方。

身爲班長，聽到同學這麼多「去了哪裡也可以去哪裡」的資訊後，你決定要爲班上的同學們，找到一個能玩最多景點的畢業旅行。

輸入說明

有多組測試資料，以 EOF 結束。

每組測試資料的第一行有兩個正整數 $n(n \leq 10^6)$ 和 $m(m \leq 10^5)$，表示景點有 n 個，編號爲 $0-(n-1)$。接下來有 m 行，每行有兩個整數 a 和 b $(0 \leq a, b < n)$，表示去了 a 的同時也可以去 b(反過來也一樣)。

輸出說明

輸出一個數字，表示畢業旅行最多可以玩的景點數量。

範例測試

範例輸入 1	範例輸出 1
6 4 0 1 2 3 1 3 5 4 1000000 0 1000000 1 0 999999	4 1 2

Problem 2.3.4. 洛谷 P1536 村村通

題目敘述

某市調查城鎮交通狀況，得到現有城鎮道路統計表。表中列出了每條道路直接連通的城鎮。市政府「村村通工程」的目標是使全市任何兩個城鎮間都可以實現交通（但不一定有直接的道路相連，只要相互之間可達即可）。請你計算出最少還需要建設多少條道路？

輸入說明

輸入包含若干組測試數據，每組測試數據的第一行給出兩個用空格隔開的正整數，分別是城鎮數目 n 和道路數目 m ；隨後的 m 行對應 m 條道路，每行給出一對用空格隔開的正整數，分別是該條道路直接相連的兩個城鎮的編號。簡單起見，城

鎮從 1 到 n 編號。

注意：兩個城市間可以有多條道路相通。

在輸入數據的最後，爲一行一個整數 0，代表測試數據的結尾。

輸出說明

對於每組數據，對應一行一個整數。表示最少還需要建設的道路數目。

範例測試

範例輸入 1	範例輸出 1
4 2 1 3 4 3 3 3 1 2 1 3 2 3 5 2 1 2 3 5 999 0 0	1 0 2 998

2.4 樹狀數組

樹狀數組又稱爲 BIT，是個快速動態求取區間和的助手。

2.4.1 回顧

還記得前綴和嗎，他也是用來求區間和的，但你應該會發現，若當中有一個值需要被修改，那整列都會被動到，因此複雜度就會增加。

以下是一個比較表，可以讓你了解 BIT 的強大之處。

資料結構	查詢區間和	單點修改值
純陣列	$O(n)$	$O(1)$
前綴和	$O(1)$	$O(n)$
BIT	$O(\log(n))$	$O(\log(n))$

2.4.2 用途與概念

BIT 用於在快速求取區間和的同時，又能保證快速修改。感覺像是陣列與前綴和的優點結合下的產物，不過其複雜程度也是三者之中最高的。示意圖中數字代表它儲存的區間。

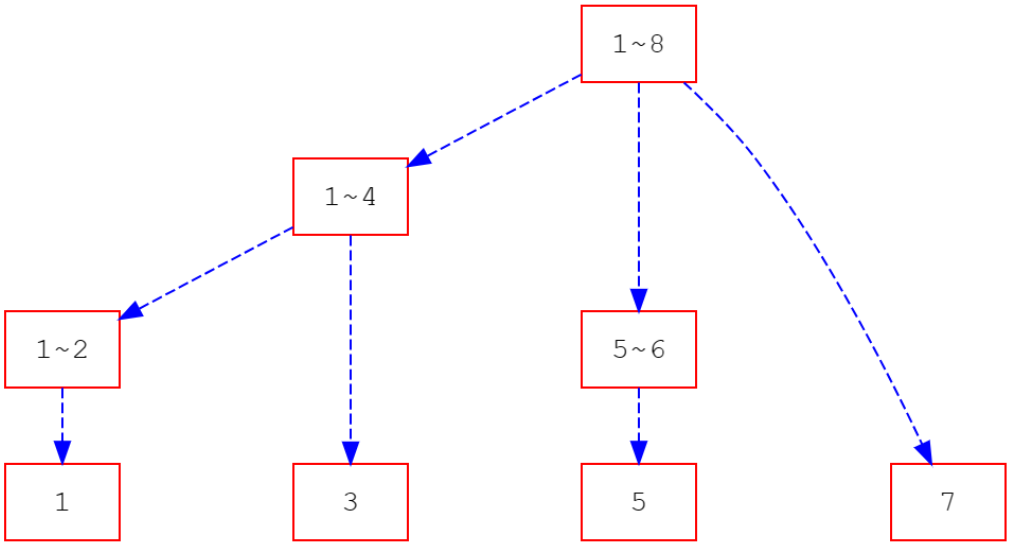


Figure 2.2: BIT 示意圖

如果想要知道1-5的和，可以查詢1-4+5。以下爲查詢所需區間表。

1	1–2	1–3	1–4	1–5	1–6	1–7	1–8
1	1–2	1–2 & 3	1–4	1–4 & 5	1–4 & 5–6	1–4 & 5–6 & 7	1–8

可以發現，在元素數量爲 8 時，最多會需要查詢 3 個區間就可以得到1-n的值，接著就像前綴和那樣，用1~R - 1~(L-1)就可以求出所有區間的和了。

2.4.3 實作

BIT 一般都是用陣列實作，用區間最大值作爲存放位置 (例如1-4放在4)。那搜尋與修改呢？這部分就比較複雜了，需要了解一些二進位制，如果你已經理解二進位制了，那就繼續往下看吧。

lowbit

lowbit 是在二進位下右邊看過來最前面的 1，例如：6(000110) 就是 2(000010)

查詢

可以發現，x 重複 $-lowbit(x)$ 會變成 0，且途中會經過所有需要的區間。以 7 爲例。

$$7(000111) \rightarrow 6(000110) \rightarrow 4(000100 \rightarrow 0(000000))$$

只要將所有經過的區間加在一起，就可以得到區間和了。這也呼應爲何他查尋的複雜度爲 $O(log(n))$，因爲 n 最多只會有 $log(n)$ 個 bit。

修改

修改也有些相似，變成 x 加上 $lowbit(x)$，就會將上面有包含到他的區間也都更新到。以 3 爲例。

$$3(000011) \rightarrow 4(000100) \rightarrow 8(001000)$$

因爲每一次都會進位，所以這樣最多也是 $O(log(n))$。也可以發現 $lowbit$ 的重要性。

程式碼

```cpp
#define lowbit(x) (x&-x)
// 與 int lowbit(int x){ return x&-x;}
// 用define或函式雖然不會讓你的程式碼寫起來比較快，但可以使其較易於理解。

using ll=long long;

ll bt[200010];
ll a[200010];

// build 函式是透過逐個更新陣列的所有元素來建構BIT
void build(int n){
    for(int i=1;i<=n;++i)
        for(int x=i;x<200005;x+=lowbit(x))
            bt[x]+=a[i];
}

// 單點加值
void add(int x,int k){
    a[x]+=k;
    for(int i=x;i<=200005;i+=lowbit(i))
        bt[i]+=k;
}

// 由單點加值衍生出來的單點修改
void modify(int x,int k){
    add(x,k-a[i]);
}

// 搜尋1~x的區間和
ll find_sum(int x){
    ll ret=0;
    for(int i=x;i>0;i-=lowbit(i))
        ret+=bt[i];
    return ret;
}

// 用計算(1~r) - (1~(l-1))算出所有區間的區間和
ll query(int l,int r){
    return find_sum(r)-find_sum(l-1);
}
```

Code 2.16: BIT

2.4.4 範例與練習

Problem 2.4.1. ZJ d796 區域調查 (POJ.1195 Mobile phones 改編)

題目敘述

給一個矩陣 $T(1,1), T(1,2), ...T(N,M)$，求 $T(x1,y1)$ 到 $T(x2,y2)$ 的總和或者是修改 $T(x1,y1)$ 的值。

輸入說明

每組輸入的第一行會有兩個正整數 $N\ Q$ $(1 \leq N \leq 250, Q \leq 5 \times 10^5)$

接下來會有 N 行，每行上會有 N 個元素 M $(0 \leq M \leq 32767)$

接下來會有 Q 行，倘若第一個數字爲 1，則接下來會有四個數字

x_1, y_1, x_2, y_2，$1\ x_1, y_1, x_2, y_2 \leq 250$

請輸出元素 $S = (x, y) \mid x_1 \leq x \leq x_2, y_1 \leq y \leq y_2$ 符合的所有元素總和

倘若第一個數字爲 2，則接下來會有三個數字

$x1, y1, V$，$1 \leq x_1, y_1 \leq 250, 0 \leq V \leq 32767$

請修改 (x1 , y1)= V ，此行不必輸出。

輸出說明

若爲調查，則輸出區域中的元素總和，若爲修改，則不必輸出。

範例測試

範例輸入 1	範例輸出 1
5 10 3 2 2 3 7 4 4 0 3 8 2 4 7 2 3 5 9 6 1 4 7 1 7 1 1 2 2 2 1 1 5 4 5 5 2 2 1 7 1 3 2 1 5 1 2 5 4 5 1 1 2 2 1 2 2 2 7 2 4 5 5 1 3 3 4 5 1 4 3 2 2	2 42 15 13 24 33

Tip: BIT 同樣可以做成二維。

Problem 2.4.2. ZJ d847 98 學年度中投區資訊學科能力競賽 2D rank finding problem

題目敘述

二度空間上的排名計算問題 (2D rank finding problem)：給定二度平面空間 $(2D)$ 上的點 $A = (a1, a2)$ 與點 $B = (b1, b2)$，其大小關係定義爲若 $A > B$ 若且唯若 $a1 > b1$ 且 $a2 > b2$，亦即 A 點在 B 點的右上方。值得注意的是，並非任意兩點均可以決定大小關係，如下圖中的點 A 與點 E，點 D 與點 E 等，無法決定這兩點的大小關係故爲無法比較 $(incomparable)$。給定 N 個點 $(x1, y1), (x2, y2), \cdots, (xn, yn)$，定義某一個點的排名 $(rank)$ 爲所給的點集合中，比該點小的點的個數。

設計一個程式，從檔案讀取點的名稱與座標，計算出在所給定的集合中，所有點的排名值。

輸入說明

有多組測試資料

每組的第一行有一個數字 N $(1 \leq N \leq 10000)$

接下來會有 N 行，每行上會有兩個數字 $x,\ y$ $(1 \leq x, y \leq 1000)$

輸出說明

請按照輸入的順序，求出對於 (x, y) 有多少個點 (a, b)

在它的左下方 $a < x, b < y$

範例測試

範例輸入 1	範例輸出 1
5 961 404 640 145 983 888 539 71 437 532	2 1 4 0 0

Problem 2.4.3. 低地距離 (2020 年 10 月 APCS)

題目敘述

輸入一個長度爲 $2n$ 的陣列，其中 $1-n$ 的每個數字都剛好各 2 次。

i 的低窪值的定義是兩個數值爲 i 的位置中間，有幾個小於 i 的的數字。

以 $[3, 1, 2, 1, 3, 2]$ 爲例，1 的低窪值爲 $0, 2$ 的低窪值值爲 $1, 3$ 的低窪值爲 3。

請對於每個 $1-n$ 的數字都求其低窪值（兩個相同的數字之間有幾個數字比它小），輸出低窪值的總和，答案可能會超過 $C++\ int$ 的上限。

輸入說明

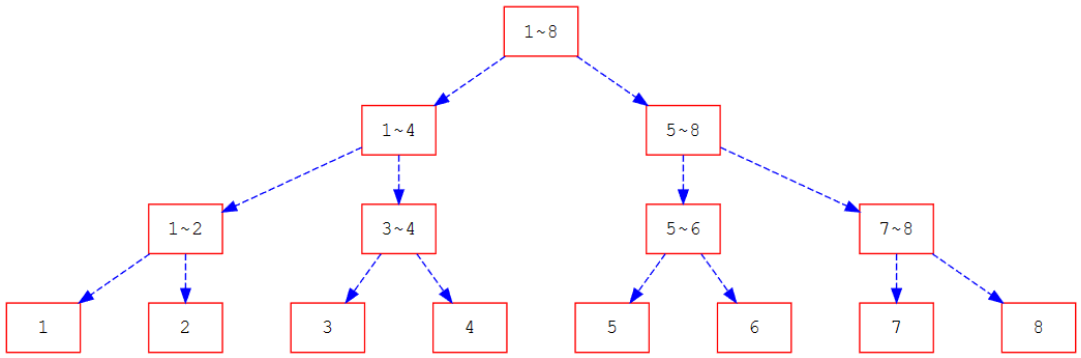
第一行有一個正整數 $n,\ n \leq 10^5$

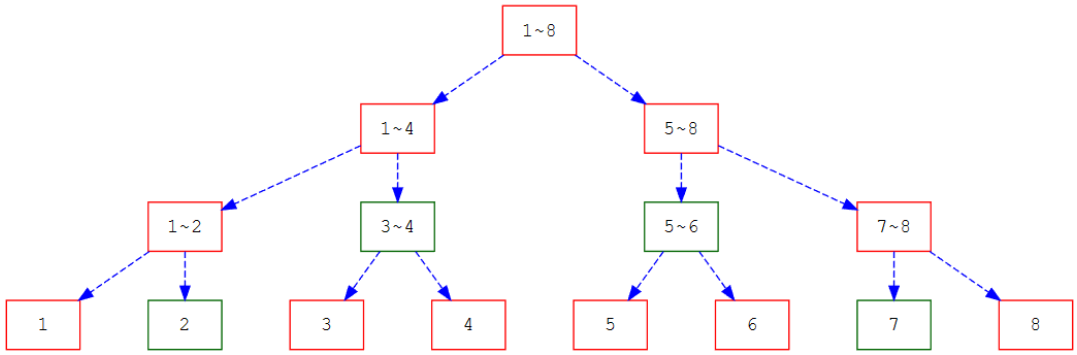
第二行有 $2n$ 個正整數，以空格分隔，保證 $1-n$ 每個數字都恰好出現兩次。

輸出說明

輸出 $1-n$ 每個數字的低窪值總和。

範例測試





範例輸入 1	範例輸出 1
3 3 1 2 1 3 2	4

2.5 線段樹

競賽上的版本。

2.5.1 用法與概念

線段樹除了可以用來快速解區間和問題，還可以用來執行許多與區間有關的操作。建構區間方式大致上如圖。

每個區間視情況放不同的數值，例如：最大/小，或是區間總和等。

接著，每個區間就都可以分爲 $O(\log(n))$ 個區間，例如2-7可以分爲2，3-4，5-6，7。

查詢時皆以最大區間爲出發點，如圖就會是從1-8這個區間開始，如果要查詢的區間是2-7。因爲1-8這個區間並沒有完全包含2-7，因此需要往下遞迴，分成1-4和5-8再次查詢。

接著，因爲1-4和5-8仍然沒有完全被2-7包含，因此要再次遞迴，這次是分解

成1-2,3-4,5-6以及7-8。

這次3-4和5-6都有被完全包含，因此可以直接回傳這個區間的值。而1-2和7-8還是沒有。所以這兩個區間還要再次向下查詢。

查詢時最重要的是，若區間完全被包含就直接回傳，若完全沒被包含就不往那邊搜尋，否則再將區間分成兩塊向下遞迴。

2.5.2 實作

可以發現他是一顆二元樹，於是我們有兩種做法：指標型與陣列型。以下實作以區間總和爲範例。

陣列型

> **Tip:** 設根節點 idx 爲 1，在完滿二元樹中，左子樹就會是 $idx \times 2$，右子樹就是 $idx \times 2 + 1$。

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

const int N=100010;

int a[N];
// 陣列seg的大小建議是N*4，否則你要先知道大於N的最小2^k值
// 131072 -> 262144 所以其實開 262200 就可以了
int seg[N*4];
int n,MXN=1;

// 不同於指標型的是：陣列型可以O(n)預建構
void build(int lb=1,int rb=MXN,int idx=1){
    //終止條件：區間長度爲1
    if(lb==rb) return;

    // 設mid爲區間中點，均分爲左右區間
    // >>1相當於/2，但稍快
    int mid=lb+rb>>1;

    // idx*2   爲左子樹
    // idx*2+1 爲右子樹
    build(lb,mid,idx*2);
    build(mid+1,rb,idx*2+1);

    seg[idx]=seg[idx*2]+seg[idx*2+1];
}

void init(){
    // 爲了讓他長度爲2^k
    while(MXN<n) MXN<<=1;
    // 歸零
    for(int i=MXN+1;i<MXN*2;i++) seg[i]=0;
    // 以陣列的內容對其初始化
    for(int i=1;i<=n;++i) seg[MXN+i-1]=a[i];
```

```
36      // 將所有節點都建構好
37      build();
38  }
39
40  void modify(int x,int k){
41      // 此部分也與指標型不同，陣列可以直接依據座標修改
42      x=x+MXN-1;
43      seg[x]=k;
44
45      // 向上更新節點
46      while(x>1){
47          x>>=1;
48          seg[x]=seg[x*2]+seg[x*2+1];
49      }
50  }
51
52  int query(int l,int r,int lb=1,int rb=MXN,int idx=1){
53      // 終止條件：所在區間位於欲查詢區間之中
54      if(l<=lb && rb<=r) return seg[idx];
55      // 設mid爲區間中點，均分爲左右區間
56      // >>1相當於/2，但稍快
57      int mid=lb+rb>>1;
58
59      int ret=0;
60      if(l<=mid)   ret+=query(l,r, lb ,mid,idx*2);
61      if(r>=mid+1) ret+=query(l,r,mid+1,rb,idx*2+1);
62
63      return ret;
64  }
65
66  int main(){
67      ios::sync_with_stdio(0);cin.tie(0);
68
69      cin>>n;
70      for(int i=1;i<=n;++i) cin>>a[i];
71      // 一定要記得init()
72      init();
73  }
```

Code 2.17: 陣列型線段樹

指標型

```
1   #include<bits/stdc++.h>
2   using namespace std;
3
4   int n;
5
6   struct node{
7       // value
8       int val;
9       // 左右子樹的指標
10      node *rch,*lch;
11      // 建構式
12      node(){
13          val=0;
```

```
14          rch=lch=nullptr;
15      }
16      // 帶有初始值的建構式
17      node(int v){
18          val=v;
19          rch=lch=nullptr;
20      }
21      // 當左右子樹改變時，應重新計算父節點的值
22      void pull(){
23          // 歸零
24          val=0;
25          // 必須先有左右子樹才能拜訪
26          // if(l) 相當於 if(l!=nullptr)
27          if(lch) val+=lch->val;
28          if(rch) val+=rch->val;
29      }
30      // 單點修改
31      void modify(int p,int v,int lb=1,int rb=n){
32          // 終止條件：區間長度爲1
33          if(lb==rb){
34              val=v;
35              return;
36          }
37          // 若左右子樹沒有結點則開一個新的
38          if(!lch) lch=new node();
39          if(!rch) rch=new node();
40          // 設mid爲區間中點，均分爲左右區間
41          // >>1相當於/2，但快很多
42          int mid=lb+rb>>1;
43          // 左邊走左，右邊走右
44          if(p<=mid) lch->modify(p,v, lb ,mid);
45          if(mid<p)  rch->modify(p,v,mid+1,rb);
46          // 還記得子樹修改完要做什麼？
47          pull();
48      }
49
50      int query(int l,int r,int lb=1,int rb=n){
51          // 終止條件：所在區間位於欲查詢區間之中
52          if(l<=lb && rb<=r){
53              return val;
54          }
55          // 同modify
56          int mid=lb+rb>>1;
57
58          int ret=0;
59          if(lch && l<=mid) ret+=lch->query(l,r, lb ,mid);
60          if(rch && mid<r)  ret+=rch->query(l,r,mid+1,rb);
61          // 爲求保險
62          pull();
63          return ret;
64      }
65  };
66
67  node *rt=new node();//root
```

Code 2.18: 指標型線段樹

2.5.3 範例與練習

Problem 2.5.1. ZJe409 Segment Tree

題目敘述

你需要使用線段樹支援兩種操作。

1. 將 $A[x]$ 的值更新爲 y
2. 要查詢 $A[X]$-$A[Y]$ 之中最大值 $maxA$ 及最小值 $minA$ 的差

Problem 2.5.2. 戰術資料庫 (110 宜中資訊社校內賽 pF)

題目敘述

要求能在一個陣列中做以下操作。

1. 搜尋區間和
2. 搜尋區間最大和最小值
3. 單點加減值

輸入說明

輸入第一行有一個數字 n, q ，下一行有 n 個數字，緊接著有 q 筆操作，可能爲以下五種。

1. 搜尋區間和 $find\ sum\ (l)\ (r)$
2. 搜尋區間最大和最小值 $find\ max/min(l)\ (r)$
3. 單點加減值 $plus/minus\ (position)\ (k)$

相鄰數字間以空白隔開。$n, q \leq 100000, a[i] \leq 100000$

輸出說明

對於每個搜尋指令做出回答。

範例測試

範例輸入 1	範例輸出 1
7 10 1 2 3 4 5 6 7 find max 2 5 find min 1 4 minus 3 1 plus 2 4 find sum 1 7 plus 7 -3 find sum 1 3 minus 6 0 plus 1 1 find max 1 7	5 1 31 9 6

2.6 ZKW 線段樹

2.6.1 概念與簡介

ZKW 線段樹是一種高效的線段樹實現方式，由清華大學張昆瑋（ZKW）提出，以非遞迴、堆式儲存和位元運算爲主要特色，在算法競賽以程式碼精簡和高效能資料結構設計而廣受推崇。

1. **非遞迴：**自底向上的更新讓 ZKW 線段樹使用迴圈而非遞迴來實現查詢和修改操作，這樣可以避免遞迴帶來的額外開銷。
2. **堆式儲存：**ZKW 線段樹強制以完滿二元樹的結構儲存資料，用類似堆 (Heap) 的索引方式，這樣可以提高存取效率。
3. **位元運算：**ZKW 線段樹利用二進制編號的特性，使用位元運算來表示節點之間的關係，這樣可以更快速地進行查詢和更新操作。

2.6.2 實作

建樹操作

這裡運用到的概念是完滿二元樹的特性，這也是爲甚麼要強制儲存成完滿二元樹結構的原因。當然，如果資料數量並非 2 的次方，無法填滿完滿二元樹的話，我們可以補 0 或給特定初始值 (依使用要求而定)。以下是完滿二元樹的特性:

- 完滿二元樹的節點編號從 1 開始，根節點爲 1。
- 對於節點編號爲 idx 的節點，其左子節點編號爲 $idx \times 2$，右子節點編號爲 $idx \times 2 + 1$。
- 當前節點編號是父節點編號除以 2 的結果 (無條件捨去)。
- 每層節點按順序排列，第 n 層有 2^{n-1} 個節點。

```
1     void init(){
2         // maxn用來定位最後一層的開始編號
3         // 小於n+2的原因則是爲了左右各預留一個節點用於置放查詢指標
4         while(maxn<n+2) maxn<<=1;
5         // seg用來儲存線段樹的節點，初始值爲0
6         for(int i=0;i<n;++i)seg[i+maxn+1]=輸入資料;
7         // 由下而上建樹，operation(a,b)是合併兩個節點的操作，依要求而定
8         for(int i=maxn-1;i;--i)seg[i]=operation(seg[i<<1],seg[i<<1|1]);
9     }
10
```

Code 2.19: ZKW 建樹

查詢操作

單點查詢的部分很簡單，就是從葉節點開始，不斷向父節點走，沿路累加權重即可。但區間查詢就需要用到位元運算的特性了，這裡我們要以擴增左右區段的方式進行查詢。舉例而言，假設要查詢的是閉區間 [l,r]，我們就會將其轉成開區間 (l-1,r+1) 這時觀察二進制節點的特性我們可以總結出對於查詢區間 [l,r] 的規律:

- 如果 l 是左節點，則兄弟節點必在查詢區間內。
- 如果 r 是右節點，則兄弟節點必在查詢區間內。
- 查詢終止的條件是 l,r 互爲兄弟節點。

以下是查詢的實作：

```
1   int query(int l,int r){
2       int num=初始值;
3       for(l+=maxn,r+=maxn+2;l^r^1;l>>=1,r>>=1){
4           // 當l是左節點時，將兄弟節點累計進去
5           if(~l&1)num=operation(num,seg[l^1]);
6           // 當r是右節點時，將兄弟節點累計進去
7           if(r&1) num=operation(num,seg[r^1]);
8       }
9       return num;
10  }
11
```

Code 2.20: ZKW 區間查詢

修改操作

單點修改的部分很單點查詢類似，就是從葉節點開始，不斷向父節點走，沿路修改父節點。然而區間修改的話就需要用到懶人標記了，ZKW 的懶人標記概念與一般的懶人標記類似，但實作較爲複雜，這裡就不再過多贅述。

以下是修改的實作：

```
1   void modify(int idx,int num){
2       idx+=maxn+1;
3       seg[idx] = num;
4       while(idx){
5           idx>>=1;
6           seg[idx]=operation(seg[idx<<1],seg[idx<<1|1]);
7       }
8   }
9
```

Code 2.21: ZKW 區間查詢

Bonus：此處，請講師示範如何使用 Scratch 實做 ZKW 線段樹。

Chapter 3

樹論

3.1 基本知識

3.1.1 什麼是樹

樹是一個連通圖，裡面沒有環，也沒有自己連自己的邊。因此，樹如果有 n 個節點，就會有 $n-1$ 個邊。移除任何一個邊都會導致他變成兩棵樹，新增任何邊都會導致出現一個環。

樹上的任意兩個點擁有唯一路徑，因此我們有機會在 $O(\log(n))$ 的時間內求出路徑長。也可以發展屬於樹的高效演算法。

只有連接到一個邊的節點我們稱爲葉節點。如果有根 (root)，則無論根連接幾個邊我們都稱他爲根結點。

3.1.2 資料儲存

通常來說，樹的儲存與圖相同，主要以鄰接串列儲存，不過由於樹的特性，如果是有根樹的話，我們可以使用一個陣列儲存，每個節點將儲存他的 parent node。而如果邊有權重，則可以另外定義 edge。

```
const int N=100010;
struct edge{
    int to,dis;// 或 w,c etc.
};

vector<int> g[N];
vector<edge> g[N];
int p[N];// 存放 parent
```

Code 3.1: 樹的儲存

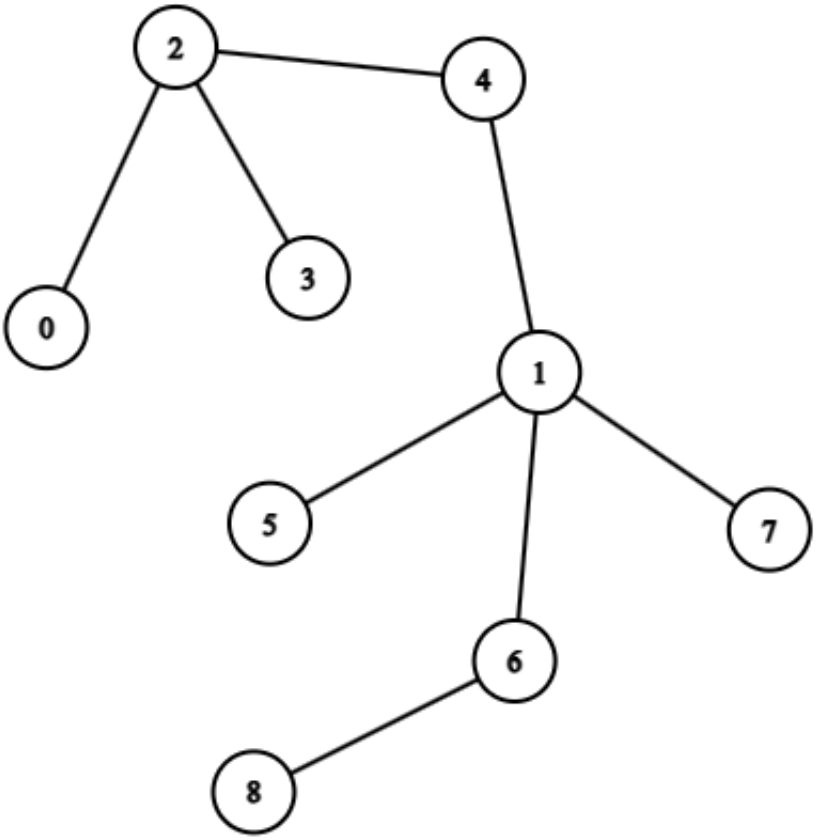


Figure 3.1: 樹示意圖

3.1.3 樹的遍歷

樹上可以使用兩種搜索，大致跟圖相同，分別是深度優先搜索 (DFS) 與廣度優先搜索 (BFS)。

深度優先搜索與廣度優先搜索都會定根，就是決定 root 是誰，如果題目沒有指定，也可以直接選擇 0 或 1。

以下都將以無權重圖演示。

```
void dfs(int now,int parent){
    for(auto next:g[now]) if(next!=parent){
        dfs(next,now);
    }
}
```

Code 3.2: DFS in Tree

```
bool isv[N];
void bfs(int root){
    queue<int> nodes;
    nodes.push(root);
    while(!nodes.empty()){// 也可以使用 nodes.size()
        int now=nodes.front();
```

```
7           nodes.pop();
8           // do something below
9
10          for(auto next:g[now]) if(!isv[next]){
11              isv[next]=true;
12              nodes.push(next);
13          }
14      }
15  }
```

Code 3.3: BFS in Tree

在樹上，我們幾乎都使用 DFS，因爲兩者都會跑完所有節點，而 DFS 的碼量較小，簡單說就是偷懶。

3.1.4 範例與練習

Example 3.1.1. 洛谷 P5908 貓貓和企鵝

題目敘述

王國裡有 n 個居住區，它們之間有 $n-1$ 條道路相連，並且保證從每個居住區出發都可以到達任何一個居住區，並且每條道路的長度都爲 1。

除了 1 號居住區外，每個居住區住著一隻小企鵝，有一天一隻貓貓從 1 號居住區出發，想要去拜訪一些小企鵝。可是貓貓非常懶，它只願意去距離它在 d 以內的小企鵝們。

貓貓非常懶，因此希望你告訴他，他可以拜訪多少隻小企鵝。

輸入說明

第一行兩個整數 n, d，意義如題所述。

第二行開始，共 $n-1$ 行，每行兩個整數 u, v，表示居民區 u 和 v 之間存在道路。

輸出說明

一行一個整數，表示貓貓可以拜訪多少隻小企鵝。

範例測試

範例輸入 1	範例輸出 1
5 1 1 2 1 3 2 4 3 5	2

我們可以在 dfs 函式內加上一個參數 dis，表示至今走過的距離。

```
1  int d;
2  int dfs(int now,int parent,int dis=0){
3      if(dis>d) return 0;
4
5      int ret=0;
```

```
6       if(now!=1) ret=1;
7
8       for(auto next:g[now]) if(next!=parent){
9           ret+=dfs(next,now,dis+1);
10      }
11
12      return ret;
13  }
14
15  int main(){
16      // input
17      dfs(1,1,0);
18  }
```

Code 3.4: 貓貓和企鵝題解

Problem 3.1.2. CF 1676 G White-Black Balanced Subtrees

題目敘述

給定一棵由 n 個節點組成的根樹，節點從 1 到 n 進行編號，根節點爲 1。還有一個字符串 s 表示每個節點的顏色：如果 $s_i = B$，則節點 i 是黑色，如果 $s_i = W$，則節點 i 是白色。

樹的子樹被稱爲平衡子樹，如果白色節點的數量等於黑色節點的數量。計算平衡子樹的數量。

樹是一個無環的連通無向圖。根樹是一棵樹中選定的節點，該節點被稱爲根。在這個問題中，所有樹都有根 1。

該樹由包含 $n-1$ 個數字的父節點數組 $a_2, \cdots, a_n$ 來指定：對於所有 $i = 2, \cdots, n$，a_i 是編號爲 i 的節點的父節點。節點 u 的父節點是一個節點，它是從 u 到根之間的一個簡單路徑上的下一個節點。

節點 u 的子樹是所有通過 u 的節點集合，形成從 u 到根的簡單路徑。請注意，一個節點包含在其子樹中，並且根的子樹是整棵樹。

輸入說明

輸入的第一行包含一個整數 $t(1 \le t \le 10^4)$——測試案例的數量。

每個測試案例的第一行包含一個整數 $n(2 \le n \le 4000)$——樹中的節點數量。

每個測試案例的第二行包含 $n-1$ 個整數 $a_2, \cdots, a_n$ $(1 \le ai < i)$ ——節點 $2, \cdots, n$ 的父節點。

每個測試案例的第三行包含一個長度爲 n 的字符串 s，由字符 B 和 W 組成——樹的顏色。

保證所有測試案例中 n 的值的總和不超過 2×10^5。

輸出說明

對於每個測試案例，輸出一個整數——平衡子樹的數量。

範例測試

範例輸入 1	範例輸出 1
3 7 1 1 2 3 3 5 WBBWWBW 2 1 BW 8 1 2 3 4 5 6 7 BWBWBWBW	2 1 4

Problem 3.1.3. CF 115 A Party

題目敘述

一家公司有 n 名員工，編號從 1 到 n。每個員工可能沒有直接的經理，或者恰好有一位不同編號的員工作爲他的直接經理。如果滿足以下條件之一，則員工 A 被稱爲員工 B 的上級：

員工 A 是員工 B 的直接經理。員工 B 有一位直接經理員工 C，且員工 A 是員工 C 的上級。公司不會存在管理循環，即不存在一位員工是其直接經理的上級。

今天公司將舉辦一個派對。這涉及將所有 n 名員工分爲若干組：每位員工必須屬於且只能屬於一個組。此外，在任何一個組內，不能有兩個員工 A 和 B，其中 A 是 B 的上級。

最少需要形成多少組？

輸入說明

第一行包含整數 $n(1 \le n \le 2000)$ —員工的數量。

接下來的 n 行包含整數 $p_i(1 \le p_i \le n$ 或 $p_i = -1)$。每個 p_i 表示第 i 名員工的直接經理。如果 p_i 是 -1，表示第 i 名員工沒有直接經理。

保證不會有員工是自己的直接經理 $(p_i \ne i)$。同時，不會存在管理循環。

輸出說明

輸出一個整數，表示在派對中將形成的最小群組數量。

範例測試

範例輸入 1	範例輸出 1
5 -1 1 2 1 -1	3

Tip: 像這種圖裡面有很多樹的情況我們通常稱爲樹林。

3.2 樹直徑

3.2.1 概念

樹直徑就是樹上的最長距離，例如前面的樹的示意圖，他的直徑就是 5，從節點 0 一路走到節點 8。

如果要求樹直徑，我們有兩種方法，分別是兩次 DFS 以及動態規劃。若你還不知道動態規劃是什麼，你只要先知道可以透過把額外的資料存在陣列中，就可以得到樹直徑，這樣就可以了。

3.2.2 兩次 DFS

兩次 DFS 可以用在邊權重不爲負的情形。我們首先要隨便找一個點 n。

1. 找到離 n 最遠的點 u。
2. 接著找到離 u 最遠的點 v。

找完以後 $u \to v$ 就是這棵樹的其中一個直徑。我們可以在 DFS 的過程中，同時維護距離，如此就會在第二次 DFS 後直接完成計算。

```
struct pii{
    int mx,node;
};

pii dfs(int n,int p,int dis){
    pii ret={dis,n};
    for(auto u:g[n]) if(u.to!=p){
        pii tp=dfs(u.to,n,dis+u.dis);
        if(tp.mx>ret.mx){
            ret=tp;
        }
    }
    return ret;
}

int main(){
    // input
    pii a=dfs(1);
    pii b=dfs(a.node);

    cout<<b.mx<<"\n";
}
```

Code 3.5: 兩次 DFS

3.2.3 樹上 DP

我們將會維護兩個陣列，分別是 mx 以及 smx，裡面存放的是最長以及非嚴格次長，同樣以樹示意圖爲例。將 2 設爲根節點，同樣用 DFS 向下遞迴，我們看到節點 1。他往下分別有節點 5, 6, 7，其中節點 6 底下還有節點 8。所以 5, 6, 7 底下的最長長度分別爲 0, 1, 0。

於是我們可以將最長與次長的邊，透過節點 1 做爲橋樑連接在一起，對所有節點都做過一次就可以找到直徑了。

```cpp
#define pii pair<int,int>
#define to first
#define dis second
const int INF=0x3f3f3f3f;

vector<pii> g[500010];
int mx[500010],smx[500010],ans=-INF;

void dfs(int x,int p){
    if(g[x].size()<=1) mx[x]=smx[x]=0;
    else mx[x]=smx[x]=-INF;
    for(auto i:g[x]) if(i.to!=p) {
        dfs(i.to,x);
        if(mx[x]<mx[i.to]+i.dis){
            smx[x]=mx[x];
            mx[x]=mx[i.to]+i.dis;
        }else if(smx[x]<mx[i.to]+i.dis){
            smx[x]=mx[i.to]+i.dis;
        }
    }
    ans=max(ans,mx[x]+smx[x]);
}

int main(){
    // input
    dfs(1,0);
    cout<<ans;
}
```

Code 3.6: 樹直徑 DP 算法

3.2.4 範例與練習

Problem 3.2.1. 洛谷 P3304 [SDOI2013] 直徑

題目敘述

小 Q 最近學習了一些圖論知識。根據課本，有如下定義。樹：無迴路且連通的無向圖，每條邊都有正整數的權值來表示其長度。如果一棵樹有 N 個節點，可以證明其有且僅有 $N-1$ 條邊。

路徑：一棵樹上，任意兩個節點之間最多有一條簡單路徑。我們用 $dis(a,b)$ 表示點 a 和點 b 的路徑上各邊長度之和。稱 $dis(a,b)$ 爲 a,b 兩個節點間的距離。

直徑：一棵樹上，最長的路徑爲樹的直徑。樹的直徑可能不是唯一的。

現在小 Q 想知道，對於給定的一棵樹，其直徑的長度是多少，以及有多少條邊滿足所有的直徑都經過該邊。

輸入說明

第一行包含一個整數 N，表示節點數。接下來 $N-1$ 行，每行三個整數 a, b, c ，表示點 a 和點 b 之間有一條長度爲 c 的無向邊。

$2 \leq N \leq 200000$，所有點的編號都在 $1 \cdots N$ 的範圍內，邊的權值 $\leq 10^9$。

輸出說明

共兩行。第一行一個整數，表示直徑的長度。第二行一個整數，表示被所有直徑經過的邊的數量。

範例測試

範例輸入 1	範例輸出 1
6 3 1 1000 1 4 10 4 2 100 4 5 50 4 6 100	1110 2

Problem 3.2.2. 洛谷 P6722 「MCOI-01」村莊

題目敘述

今天，珂愛善良的 0x3 喵醬騎著一匹小馬來到了一個村莊。

「嘿，這個村莊的佈局.....」「好像之前我玩 Ciste 的地方啊 qwq」

0x3 喵醬有一張地圖，地圖上有關於這個村莊的資訊。然後 0x3 喵醬要根據這張地圖來判斷 Ciste 是否有解。

注：Ciste 是《請問您今天要來點兔子嗎》中的一種藏寶圖遊戲。

村莊被簡化爲一個 n 個節點（編號爲 1 到 n）和 $n-1$ 條邊構成的無向連通圖。

0x3 喵醬認爲這個無向圖的資訊與滿足以下條件的新圖有關：

新圖的節點集合與原圖相同在新圖中，節點 u 和節點 v 之間存在無向邊當且僅當在原圖中 $dis(u,v) \geq k$（k 是給定的常數，$dis(u,v)$ 表示節點編號爲 u 的節點到節點編號爲 v 的節點的最短路徑長度）0x3 喵醬還認爲，如果這個" 新圖" 是二分圖，則 Ciste 有解；如果" 新圖" 不是二分圖，則 Ciste 無解。(如果您不知道二分圖，請參考提示)

現在 0x3 喵醬想請您判斷一下這個 Ciste 是否有解。

輸入說明

第一行包含一個正整數 T，表示有 T 組測試數據。對於每組測試數據，第一行包含兩個正整數 n 和 k。接下來 $n-1$ 行，每行包含三個正整數 x、y 和 v，表示節點編號爲 x 的節點到節點編號爲 y 的節點有一條權重爲 v 的無向邊。輸入數據保證合法。

$n \leq 10^5$，$T \leq 10$，$v \leq 1000$，$k \leq 1000000$

輸出說明

對於每一組測試數據，輸出一行，如果 Ciste 有解則輸出"Yes"，否則輸出"Baka Chino"。

Tip: 二分圖又稱作二部圖，是圖論中的一種特殊模型。設 $G = (V, E)$ 是一個無向圖，如果頂點 V 可分割爲兩個互不相交的子集 (A, B)，並且圖中的每條邊 (i, j) 所關聯的兩個頂點 i 和 j 分別屬於這兩個不同的頂點集 $(i \in A, j \in B)$，則稱圖 G 爲一個二分圖。

Problem 3.2.3. 實作一個演算法可以計算出次長樹直徑。

3.3 樹重心

3.3.1 概念

樹的重心就是拔除他後，形成的若干棵樹分別的節點數量中的最大值，會最小的那個節點。

也可以說，以重心爲根後，所有子樹的大小都不超過總結點的一半。

如果我們想要知道樹重心在哪裡的話，我們可以應用 DP 技巧。首先，我們同樣是隨意找一個節點當作根，接著向下 DFS。接著，對於每個節點，我們都找出它的最大子樹。最後，不要忘記，如果把它當作根節點，那它上面的所有節點都將會是他的子樹，所以還要考慮上面的所有節點。

3.3.2 實作

```cpp
#include<bits/stdc++.h>
using namespace std;
#define INF 1000000000

vector<int> child[100010];
// sz[x] 是 x 的子樹大小，dp[x] 是 x 下面最大的子樹大小。
int sz[100010],dp[100010],ans=-1,n,mn=INF;
// vt 就是 isv，表示有沒有經過這個點。
bool vt[100010];

void dfs(int a){
    sz[a]=1;
    for(auto c:child[a]){
        if(!vt[c]){
            vt[c]=true;
            dfs(c);
            sz[a]+=sz[c];
            dp[a]=max(dp[a],sz[c]);
        }
```

```
20      }
21      dp[a]=max(dp[a],n-sz[a]);
22      if(mn>dp[a]){
23          mn=dp[a];
24          ans=a;
25      }else if(mn==dp[a] && ans>a){
26          ans=a;
27      }
28  }
29
30  int main(){
31      ios::sync_with_stdio(0);cin.tie(0);
32
33      int t;
34      cin>>t;
35      for(int iptNum=0;iptNum<t;iptNum++){
36          mn=INF;
37          for(int i=0;i<100010;i++){
38              child[i].clear();
39              sz[i]=0;
40              dp[i]=0;
41              vt[i]=false;
42          }
43          cin>>n;
44          for(int i=1;i<n;i++){
45              int c,p;
46              cin>>p>>c;
47              child[p].emplace_back(c);
48              child[c].emplace_back(p);
49          }
50          dfs(0);
51          cout<<ans<<"\n";
52      }
53      return 0;
54  }
```

Code 3.7: 樹重心

3.3.3 範例與練習

Problem 3.3.1. CF 708C Centroids

題目敘述

給定一個由 n 個節點組成的樹。如果從樹中刪除該節點後，每個連通分量的大小都不超過 $\frac{n}{2}$，則該節點稱爲重心。

現在，你可以進行最多一次的邊替換操作。邊替換是指從樹中刪除一條邊 (保留相應的節點)，然後插入一條新的邊 (不添加新的節點)，使得圖形仍然是一棵樹。你需要判斷每個節點是否可以透過最多一次的邊替換成爲重心。

輸入說明

第一行包含一個整數 $n(2 \leq n \leq 4 \times 10^5)$ ，表示樹中的節點數量。接下來的 $n-1$

行，每行包含一對節點索引 u_i 和 $v_i(1 \leq u_i, v_i \leq n)$，表示相應邊的兩個端點。

輸出說明

輸出 n 個整數，第 i 個整數等於 1 表示第 i 個節點可以透過最多一次的邊替換成爲中心節點，等於 0 則表示不能。

範例測試

範例輸入 1	範例輸出 1
5 1 2 1 3 1 4 1 5	1 0 0 0 0
範例輸入 2	範例輸出 2
3 1 2 2 3	1 1 1

Tip: 先找出重心，再利用它的性質。

Problem 3.3.2. 洛谷 P1395 會議

題目敘述

在一個村莊裡住著 n 個村民，這 n 個村民的家通過 $n-1$ 條路徑相連，每條路徑的長度爲 1。現在，村長計劃在其中一個村民的家裡舉辦一場會議，村長希望選擇一個村民的家作爲會議地點，使得所有村民到會議地點的距離之和最小。如果有多個村民的家都滿足條件，則選擇村民編號最小的家作爲會議地點。

輸入說明

第一行包含一個整數 n，表示村民的數量。

接下來的 $n-1$ 行，每行包含兩個整數 a 和 b，表示村民 a 的家和村民 b 的家之間存在一條路徑。

$n \leq 5 \times 10^4$

輸出說明

輸出一行，包含兩個整數 x 和 y。x 表示會議地點所在的村民家的編號。y 表示所有村民到會議地點的距離之和的最小值。

範例測試

範例輸入 1	範例輸出 1
4 1 2 2 3 3 4	2 4

3.4 最近共同祖先

3.4.1 概念

考慮一顆有根樹，我們希望找到任何兩個節點的最近同祖先 (又稱爲 LCA)。我們以樹示意圖爲例，將節點 2 定爲根，我們希望找到節點 5 和 8 的最近共同祖先。那麼最近共同祖先便是 1。

我們可以很快的想到一個 $O(n)$ 的方法找出任兩節點的最近共同祖先。首先，我們先跑過一次 DFS 預處理每個節點到根的距離，接著，如果需要查詢 u 和 v 的 LCA，我們遵循幾個條件。

- 如果 u 離根的距離短，重複讓 v 與根的距離縮短，反之就讓 u 與根的距離縮短。
- 讓 u,v 與根的距離保持相等向上尋找。
- 如果找到同一個點就是正確的。

```
1  int dis[N],par[N];
2  void dfs(int n,int p,int d){
3      dis[n]=d;
4      for(auto u:g[n]) if(u!=p){
5          par[u]=n;
6          dfs(u,n,d+1);
7      }
8  }
9
10 int query(int u,int v){
11     if(dis[u]>dis[v]){
12         swap(u,v);
13     }
14     while(dis[v]>dis[u]){
15         v=par[v];
16     }
17
18     while(u!=v){
19         u=par[u];
20         v=par[v];
21     }
22
23     return u;
24 }
```

Code 3.8: $O(n)$ LCA 算法

不過這樣的速度並不是很讓人滿意，有沒有辦法讓搜尋的速度提升呢，答案是可以的，只需要犧牲一點點預處理的時間就可以了。

3.4.2 倍增法

剛剛我們在找 LCA 的時候我們都是一個一個向上的，如果我們可以一次跳很多步，是否就會快上許多呢？沒有錯，我們可以要建構從節點 u 向上 $1 \sim n$ 個節點是誰，在做二分搜，可是這樣預處理的時間將會是 $O(n^2)$。太慢了。所以我們退而求其次，建構 u 向上 $1, 2, 4, \cdots, 2^k, \cdots, 2^{\log_2(n)}$ 步的節點。這樣會花費 $O(n \log(n))$ 的時間做預處理，也可以在 $O(\log(n))$ 的時間內完成查詢的工作。

3.4.3 實作

```cpp
vector<int> g[100010];
int lca[30][100010];
int lev[100010],root;

void init(int x=root,int p=root){
    lca[0][x]=p;
    lev[x]=lev[p]+1;
    for(auto i:g[x]) if(i != p)
        init(i,x);
}

void build(int n){
    for(int i=1;i<=log2(n);++i)
        for(int j=1;j<=n;++j)
            lca[i][j]=lca[i-1][lca[i-1][j]];
}

int query(int a,int b){
    if(lev[a]<lev[b]) swap(a,b);
    for(int i=25;i>=0;--i)
        if(lev[lca[i][a]]>=lev[b])
            a=lca[i][a];
    if(a==b) return a;

    for(int i=25;i>=0;--i){
        if(lca[i][a]!=lca[i][b]){
            a=lca[i][a];
            b=lca[i][b];
        }
    }
    return lca[0][a];
}
```

Code 3.9: $O(\log(n))$ 查詢 LCA 算法

3.4.4 範例與練習

Problem 3.4.1. 洛谷 P3379 【模板】最近公共祖先 (LCA)

題目敘述

給定一棵有根多叉樹，求指定兩個節點的最近公共祖先。

輸入說明

第一行包含三個正整數 N, M, S，分別表示樹的節點個數、詢問的個數和樹的根節點序號。

接下來 $N-1$ 行，每行包含兩個正整數 x, y，表示節點 x 和節點 y 之間有一條直接連接的邊（保證可以構成樹）。

接下來 M 行，每行包含兩個正整數 a, b，表示詢問節點 a 和節點 b 的最近公共祖先。

100% 的數據，$1 \leq N, M \leq 500000$，$1 \leq x, y, a, b \leq N$，不保證 $a \neq b$

輸出說明

輸出包含 M 行，每行包含一個正整數，依次爲每一個詢問的結果。

範例測試

範例輸入 1	範例輸出 1
5 5 4 3 1 2 4 5 1 1 4 2 4 3 2 3 5 1 2 4 5	4 4 1 4 4

Problem 3.4.2. CF 191 C Fools and Roads

題目敘述

他們說伯蘭德有兩個問題，愚蠢的人和道路。此外，伯蘭德有 n 個城市，由愚蠢的人居住，並由道路連接。伯蘭德的所有道路都是雙向的。由於伯蘭德有很多愚蠢的人，所以每對城市之間都有一條路徑 (否則愚蠢的人會生氣)。此外，每對城市之間最多只有一條簡單路徑 (否則愚蠢的人會迷路)。

但這還不是伯蘭德的特點結束。在這個國家，愚蠢的人有時互相訪問，從而破壞了道路。愚蠢的人並不聰明，所以他們只使用簡單的路徑。

簡單路徑是通過每個伯蘭德城市不超過一次的路徑。

伯蘭德政府知道愚蠢的人使用的路徑。幫助政府計算每條道路上可以走的不同愚蠢的人數量。

愚蠢的人的路徑將在在輸入中給出說明。

輸入說明

第一行包含一個整數 $n (2 \leq n \leq 10^5)$——城市的數量。

接下來的 $n-1$ 行，每行包含兩個以空格分隔的整數 u_i，v_i $(1 \leq u_i, v_i \leq n, ui \neq$

vi)，表示城市 u_i 和 v_i 之間有一條連接的道路。

下一行包含整數 $k(0 \leq k \leq 10^5)$——互相訪問的愚蠢人對數。

接下來的 k 行包含兩個以空格分隔的數字。第 i 行 $(i > 0)$ 包含數字 $a_i, b_i(1 \leq a_i, b_i \leq n)$。這意味著第 $2i-1$ 個愚蠢的人住在城市 a_i，並訪問住在城市 b_i 的第 2_i 個愚蠢的人。給定的數對描述了簡單的路徑，因爲在每對城市之間只有一條簡單的路徑。

輸出說明

輸出 $n-1$ 個整數，數字應以空格分隔。第 i 個數字應該等於第 i 條道路上可以走的愚蠢的人數量。道路從輸入中出現的順序開始編號，從 1 開始。

範例測試

範例輸入 1	範例輸出 1
5 1 2 1 3 2 4 2 5 2 1 4 3 5	2 1 1 1

3.5 樹上尤拉路徑

3.5.1 什麼是尤拉路徑

通常而言，尤拉路徑是指所有的邊都只通過一次的路徑，不過樹上很明顯沒有這東西。所以樹上的尤拉路徑指的是每個邊都通過兩次的路徑。我們可以藉由 DFS 來記錄這樣的路徑。

3.5.2 用途

藉由這方法，配合資料結構，同樣可以 $O(\log(n))$ 完成 LCA 的查詢。不僅如此，有一些其他問題也會用到這個操作。

3.5.3 建構

```
vector<int> g[100010] ;
vector<int> et ;
int in[100010], out[100010] ;

void dfs(int x, int p) {
    et.push_back(x) ;
    in[x] = et.size() - 1 ;
```

```
8       for(auto i:g[x]) if(i != p) {
9           dfs(i, x) ;
10          et.push_back(x) ;
11      }
12      out[x] = et.size() - 1 ;
13  }
```

Code 3.10: 樹上尤拉路徑

如果你希望它可以用來解決 LCA 問題，你還需要多存一個變數，就是深度。而此時兩個節點的 LCA 就會是兩個節點之間的深度最小值位置。

3.5.4 範例與練習

Problem 3.5.1. CF 620 E New Year Tree

題目敘述

新年假期結束了，但 Resha 不想丟掉新年樹。他邀請了他最好的朋友 Kerim 和 Gural 幫助他重新裝飾新年樹。

新年樹是一個有 n 個頂點且根在頂點 1 的無向樹。

你需要處理兩種類型的查詢：

1. 將頂點 v 的子樹中的所有頂點顏色更改爲顏色 c。
2. 查找頂點 v 的子樹中不同顏色的數量。

輸入說明

第一行包含兩個整數 n 和 m $(1 \leq n, m \leq 4 \times 10^5)$ —樹中頂點的數量和查詢的數量。

第二行包含 n 個整數 c_i $(1 \leq c_i \leq 60)$ —第 i 個頂點的顏色。

接下來的 $n-1$ 行，每行包含兩個整數 x_j 和 y_j $(1 \leq x_j, y_j \leq n)$ —第 j 條邊的兩個頂點。保證給出的是正確的無向樹。

最後的 m 行包含查詢的描述。每個描述以整數 $t_k (1 \leq t_k \leq 2)$ 開始—第 k 個查詢的類型。對於第一類型的查詢，接下來是兩個整數 v_k 和 c_k $(1 \leq v_k \leq n, 1 \leq c_k \leq 60)$ —子樹將用顏色 c_k 重新著色的頂點號碼 v_k。對於第二類型的查詢，接下來是一個整數 v_k $(1 \leq v_k \leq n)$ —需要查找其子樹中不同顏色數量的頂點號碼。

輸出說明

對於每個第二類型的查詢，輸出一個整數 a —查詢中給定的頂點的子樹中不同顏色的數量。

每個數字應該以單獨的行按照輸入中出現的查詢順序進行輸出。

3.6 輕重鍊剖分

3.6.1 概念

輕重鍊剖分是樹上的特殊方法，他可以藉由預處理的方式，將查詢區間最大最小值與區段總和的時間複雜度降到 $O(\log^2(n))$ 以下，相較於原來的 $O(n)$ 而言會快上許多。

首先同樣要先定根，接著我們透過一次 DFS 確定每個節點的子樹大小。

在輕重鍊剖分中，我們會優先從子樹最大的節點往下連接，又稱爲重邊，而其他的邊則稱爲輕邊。這樣可以保證我們的鍊的數量不會超過 $O(\log(n))$ 條。

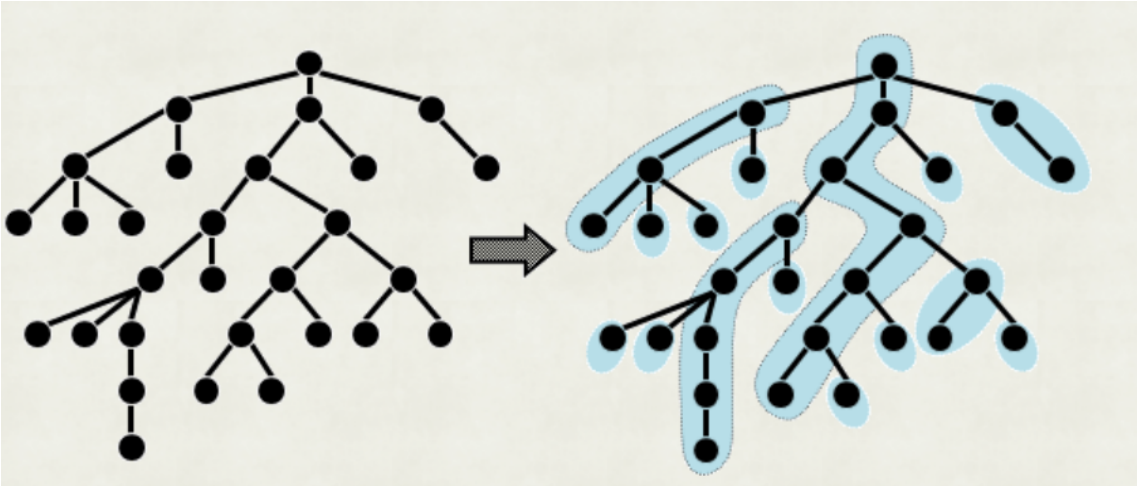


Figure 3.2: 輕重鍊剖分示意圖

3.6.2 實作

實作上我們會需要很多個陣列。

```
using ll=long long;
const int N=100010;
const ll INF=0x3f3f3f3f3f3f3f3f;
using pii=pair<int,int>;

int sz[N],fa[N],to[N],fr[N],dep[N],dfn[N];
// sz[x]  = x 的子樹大小
// fa[x]  = x 的 parent node
// to[x]  = x 所有子節點中，子樹最大的
// dep[x] = x 到根的距離
// dfn[x] = dfs(x)時的標記
// fr[x]  = x 所在的鍊的頭

int n;
vector<int> g[N];

```

```
17 void dfs(int x,int p){
18     sz[x]=1, fa[x]=p, to[x]=-1;
19     dep[x]=(p!=-1) ? dep[p]+1 : 0;
20 
21     for(auto u:g[x]) if(u!=p){
22         dfs(u,x);
23         sz[x]+=sz[u];
24         if(to[x]==-1 || sz[u]>sz[to[x]])
25             to[x]=u;
26     }
27 }
28 
29 void link(int x,int f){
30     // time stamp
31     static int tk=1;
32     fr[x]=f;
33     dfn[x]=tk++;// dfn[x]=tk, tk++;
34     // extend the chain
35     if(to[x]!=-1) link(to[x],f);
36 
37     for(auto u:g[x]){
38         if(u==fa[x] || u==to[x]) continue;
39         link(u,u);
40     }
41 }
42 
43 vector<pii> QueryPath(int u,int v){
44     // return the intervals on the path from u to v
45     int fu=fr[u],fv=fr[v];
46     vector<pii> ret;
47     while(fu!=fv){
48         if(dep[fu]<dep[fv])
49             swap(fu,fv), swap(u,v);
50         ret.emplace_back(dfn[fu],dfn[u]);
51         u=fa[fu];
52         fu=fr[u];
53     }
54     if(dep[u]>dep[v]) swap(u,v);
55     // now u is LCA
56     ret.emplace_back(dfn[u],dfn[v]);
57     return ret;
58 }
59 
60 struct Seg{
61     struct node{
62         ll val,tag;
63 
64         node *rch,*lch;
65 
66         node(){
67             val=tag=0;
68             rch=lch=nullptr;
69         }
70 
71         node(ll v){
72             val=v, tag=0;
```

```
73              rch=lch=nullptr;
74          }
75
76          void push(int l,int r){
77              int len=r-l+1>>1;
78              if(lch){
79                  lch->val+=tag*len;
80                  lch->tag+=tag;
81              }
82
83              if(rch){
84                  rch->val+=tag*len;
85                  rch->tag+=tag;
86              }
87              tag=0;
88          }
89
90          void pull(){
91              val=0;
92              if(lch) val+=lch->val;
93              if(rch) val+=rch->val;
94          }
95
96          void modify(int p,ll v,int lb=1,int rb=n){
97              if(lb==rb){
98                  val=v;
99                  return;
100             }
101
102             if(!lch) lch=new node();
103             if(!rch) rch=new node();
104
105             push(lb,rb);
106
107             int mid=lb+rb>>1;
108
109             if(p<=mid) lch->modify(p,v, lb ,mid);
110             if(mid<p)  rch->modify(p,v,mid+1,rb);
111
112             pull();
113         }
114
115         ll query(int l,int r,int lb=1,int rb=n){
116             if(l<=lb && rb<=r){
117                 return val;
118             }
119
120             push(lb,rb);
121
122             int mid=lb+rb>>1;
123
124             ll ret=0;
125             if(lch && l<=mid) ret+=lch->query(l,r, lb ,mid);
126             if(rch && mid<r)  ret+=rch->query(l,r,mid+1,rb);
127
128             pull();
```

```
129
130             return ret;
131         }
132
133         void add(int l,int r,ll v,int lb=1,int rb=n){
134             if(l<=lb && rb<=r){
135                 val+=v*(rb-lb+1);
136                 tag+=v;
137                 return;
138             }
139
140             push(lb,rb);
141
142             int mid=lb+rb>>1;
143
144             int ret=0;
145             if(lch && l<=mid) lch->add(l,r,v, lb ,mid);
146             if(rch && mid<r)  rch->add(l,r,v,mid+1,rb);
147
148             pull();
149         }
150     };
151
152     node *rt=new node();//root
153
154     void modify(int p,int v){
155         rt->modify(p,v);
156     }
157
158     ll query(int l,int r){
159         return rt->query(l,r);
160     }
161
162     void add(int l,int r,ll v){
163         rt->add(l,r,v);
164     }
165 };
166
167 Seg seg;
168
169 void modify(int v,int p){
170     p=dfn[p];
171     seg.modify(v,p);
172 }
173
174 ll query(int u,int v){
175     vector<pii> pt=QueryPath(u,v);
176     ll ret=0;
177     for(auto p:pt){
178         ret+=seg.query(p.first,p.second);
179     }
180     return ret;
181 }
182
183 ll add(int u,int v,ll k){
184     vector<pii> pt=QueryPath(u,v);
```

```
185     for(auto p:pt){
186         seg.add(p.first,p.second,k);
187     }
188 }
```

Code 3.11: 輕重鍊剖分與線段樹

3.6.3 範例與練習

Problem 3.6.1. 第一屆卓越盃 G. Safe Tariff(Hard)

題目敘述

太平洋上有 n 個島嶼國家，爲了方便起見將其命名爲 1 n 。這些島嶼國家皆使用一個共同貨幣，他們稱其爲國際銀幣。各個國家之間有許多貿易往來，其中一種便是金磚航運。並且在這些國家之間有一些固定的航線往來，這些航線也是交流與物產貿易的主要手段。每一條航線都是往返固定兩個國家。若一航線是往返國家 u ，則以 (u,v) 或 (v,u) 表示該航線，兩種表示方法相同，代表聚客 u 與國家 v 可以直航交易兩地。如果國家 u 與國家 v 之間沒有航線直接往返也可能透過複數航線進行貿易。若存在一系列的航線 $(u,p_1),(p_1,p_2)...(p_r,v)$ 則國家 u 與國家 v 之間便可以進行貿易，反之則不行。

爲了保護太平洋的生態，所有國家的共識是在任兩個島嶼國家之間都能夠進行貿易往來的前提下維持盡可能少的航線。顯而易見的，在 n 個國家的情況下僅需要 $n-1$ 條航線就足夠了。

每個國家爲了保護自己內部的產業不受到外部的低價商品傾銷、惡性競爭。因此對各項商品都設定有關稅。而在太平洋上關稅課徵的方式是針對貨物數量，在每一航線上船上所 載的商品都要被抽一次 金。依據商品的種類不同有不同的抽 方式；而針對金磚，則是每一個保險箱要課徵 c 個國際銀幣。然而因爲金磚航運生意普普，因此**所有**國家起初並不是非常重視，也就沒有對其課徵關稅。

黃瓜學長是一個專做金磚航運的 **"爲何年輕人都不愛讀近體詩"** 企業的老闆，客户所在的國家與當下金磚能出貨的國家不同，可能會被課徵關稅也不同。他希望可以收取合理的價格以避免無法獲利，或是被投訴價格過高不符合公平貿易。因此他需要計算航線上會被課徵多少關稅。

並且，由於最近發生金融危機，加上疫情高漲，許多原來不課徵關稅的國家也嗅到了這股商機，選擇開始課徵關稅。所以黃瓜學長必須隨時調整售價來因應關稅的變動。以免傷害到 **"爲何年輕人都不愛讀近體詩"** 的競爭力。

輸入說明

第一行輸入 2 個數字 $n\ q$ ，代表有 n 個國家，q 次操作

接下來有 $n-1$ 行，每行有 2 個數字 $u\ v$ ，代表國家 (u,v) 之間有直接的航道。

接下來有 q 行，每行有 3 個數字 $op\ a\ b$

若 $op=1$ ，代表黃瓜學長收到訂單 (從國家 a 輸送到國家 b)

否則 $op=2$ ，代表國家 a ，將關稅改爲 b 國際銀幣/保險箱

$op\in 1,2$，$n,q\leq 10^5$，$a,b\leq n$

輸出說明

對於 $op == 1$ 輸出所需關稅 (輸出國 (a) 不會課關稅，但輸入國 (b) 會)。

範例測試

範例輸入 1	範例輸出 1
7 10 6 5 7 5 4 3 4 5 4 2 6 1 1 3 7 1 3 1 2 6 207 1 7 4 1 6 4 2 2 683 2 3 119 1 5 6 2 1 579 2 1 947	0 0 0 0 207

Problem 3.6.2. 洛谷 P2680 運輸計畫

題目敘述

公元 2044 年，人類進入了宇宙紀元。

L 國有 n 個星球，還有 $n-1$ 條雙向航道，每條航道建立在兩個星球之間，這 $n-1$ 條航道連通了 L 國的所有星球。

小 P 掌管一家物流公司，該公司有很多個運輸計劃，每個運輸計劃形如：有一艘物流飛船需要從 u_i 號星球沿最快的宇航路徑飛行到 v_i 號星球去。顯然，飛船駛過一條航道是需要時間的，對於航道 j，任意飛船駛過它所花費的時間爲 t_j，並且任意兩艘飛船之間不會產生任何干擾。

爲了鼓勵科技創新，L 國國王同意小 P 的物流公司參與 L 國的航道建設，即允許小 P 把某一條航道改造成蟲洞，飛船駛過蟲洞不消耗時間。

在蟲洞的建設完成前，小 P 的物流公司就預接了 m 個運輸計劃。在蟲洞建設完成後，這 m 個運輸計劃會同時開始，所有飛船一起出發。當這 m 個運輸計劃都完成時，小 P 的物流公司的階段性工作就完成了。

如果小 P 可以自由選擇將哪一條航道改造成蟲洞，試求出小 P 的物流公司完成階段性工作所需要的最短時間是多少？

輸入說明

第一行包括兩個正整數 n, m，表示 L 國中星球的數量及小 P 公司預接的運輸計劃的數量，星球從 1 到 n 編號。

接下來 $n-1$ 行描述航道的建設情況，其中第 i 行包含三個整數 a_i, b_i 和 t_i，表示第 i 條雙向航道修建在 a_i 與 b_i 兩個星球之間，任意飛船駛過它所花費的時間爲 t_i。

接下來 m 行描述運輸計劃的情況，其中第 j 行包含兩個正整數 u_j 和 v_j，表示第 j 個運輸計劃是從 u_j 號星球飛往 v_j 號星球。

數據保證 $1 \leq a_i, b_i \leq n$，$0 \leq t_i \leq 1000$，$1 \leq u_i, v_i \leq n$

輸出說明

一個整數，表示小 P 的物流公司完成階段性工作所需要的最短時間。

範例測試

範例輸入 1	範例輸出 1
6 3 1 2 3 1 6 4 3 1 7 4 3 6 3 5 5 3 6 2 5 4 5	11

Tip: 需要快讀快寫，記得往前翻。

Chapter 4

圖論初階

4.1 基本知識

4.1.1 圖是什麼

圖 G 由點 V 與邊 E 構成，記做 $G = (V, E)$ 。聽起來很難？
其實就是像這樣的東西。

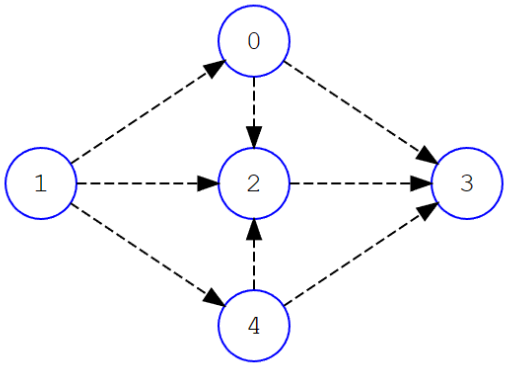


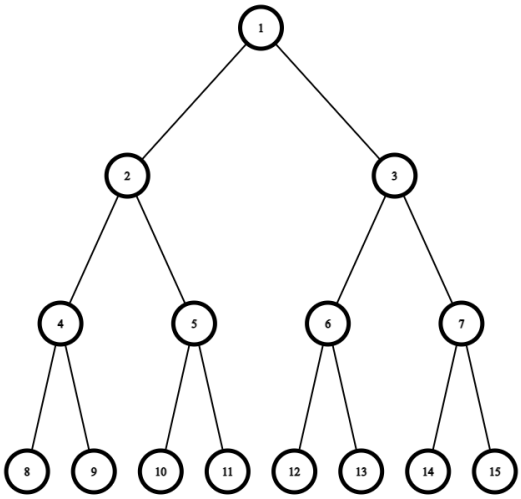
Figure 4.1: “圖” 的示意圖

4.1.2 線上好用工具

`https://csacademy.com/app/graph_editor/`
他畫出來的圖長這樣。

4.1.3 用途

所以這就是圖，不過這樣的東西到底會問怎樣的問題呢？



Example 4.1.1. 圖的基本問題

輸入一個有向圖 G 與一個起點 s

- 請計算由 s 出發可以到達的點數 (不包含 s)。
- 並且計算這些可以到達的點與 s 的距離和。
- 假設每個邊的長度均爲 1。
- 兩點之間可能有多個邊。
- 邊的起點與終點未必不同。

4.1.4 存圖

鄰接串列 (Adjacency Link)

因爲比賽中多數都用不到鄰接矩陣 (Adjacency Matrix)，所以我先講鄰接串列。

簡單說就是用 vector 陣列存放點和點之間的關係。

```
1 const int N=100010;
2 vector<int> g[N];
```

Code 4.1: 存圖

其中，g[x] 就是從 x 出發可以到達的所有點 (注意 g[x] 是一個 vector)。如果你要新增一個從 a 到 b 的邊，則分爲兩種情況。

1. 有向圖 `g[a].push_back(b);`
2. 無向圖還要加上 `g[b].push_back(a);`

會這樣做是因爲，無向圖的邊是可以雙向通行的，所以如果 a 可以到 b，則 b 也可以到 a。

鄰接矩陣

鄰接矩陣使用一個g[n][n]表示圖形，其中g[a][b]>0表示 a 可以到 b，這樣的存圖方式也很直覺，但是會需要花費 $O(n^2)$ 的記憶體空間。相較於上一個存圖方式的 $O(n+m)$ 而言會高許多，因爲競賽上常常出現 $n \leq 10^5$, $m \leq 2 \times 10^5$ 的情況。

4.1.5 BFS(廣度優先搜索)

依照到原點所需要經過的邊數由小到大的順序訪問點。以程式來實作的話需要一些簡單的資料結構 (queue)。

```
vector<int> g[N];
bool isv[N];

void BFS(int s){
    queue<int> q;
    q.push(s);

    while(!q.empty()){
        int now=q.front();
        q.pop();
        for(auto e:g[now]) if(!isv[e]){
            q.push(e);
            isv[e]=true;
        }
    }
}
```

Code 4.2: 圖的 BFS

不過，如果要解決前面的例題，我們還需要其他東西，幫忙協助計算。

```
struct info{
    int to,dis;
};

vector<int> g[N];
bool isv[N];

int bfs(int s){
    queue<info> q;
    q.push({s,0});

    int ret=0;

    while(!q.empty()){
        info now=q.front();
        q.pop();

        ret+=now.dis;

```

```
20          for(auto e:g[now.to]) if(!isv[e]){
21              q.push({e,now.dis+1});
22              isv[e]=true;
23          }
24      }
25
26      return ret;
27  }
```

Code 4.3: 圖基本問題題解

4.1.6 DFS(深度優先搜索)

跟剛剛 BFS 走的順序不一樣，在 DFS 的時候，我們只要有路就一直走，直到沒有路我們才考慮上一個有其他路的點。聽起來有點複雜，不過程式的寫法與 Tree 上面的有點像，所以還是很簡單的。

```
1  vector<int> g[N];
2  bool isv[N];
3
4  void DFS(int n){
5      for(auto u:g[n]) if(!isv[u]){
6          isv[u]=true;
7          DFS(u);
8      }
9  }
```

Code 4.4: 圖基本問題題解

需要注意的是，isv 標記的順序要在 DFS 往下前，否則如果遇到還就會陷入無窮迴圈。

因爲 DFS 明顯比較好寫，因此在可以用 DFS 的情況下幾乎會使用他。

4.1.7 範例與練習

Example 4.1.2. AP325 P-7-2 開車蒐集寶物

題目敘述

參加一個蒐集寶物的遊戲，你拿到一個地圖，地圖上有 n 個藏寶點，每個藏寶點有若干價值的寶物，由於你的團隊是最頂尖的，只要能到達藏寶點一定可以取得該藏寶點的寶藏。

從地圖上看得到一共有 m 條道路，每條道路連接兩個藏寶點，而且每條道路都是雙向可以通行的。

在遊戲的一開始，你可以要求直升機將你的團隊運送到某個藏寶點，而且你可以獲得一部車與充足的油料，但是直升機的載送只有一次，所以你必須決定要從哪裡開始才可以獲得最多的寶藏總價值。

輸入說明

第一行是兩個正整數 n 與 m ，代表藏寶地點數與道路數，地點是以 0 $n-1$ 編號，

第二行 n 個非負整數，依序是每一個地點的寶藏價值，每個地點的寶藏價值不超過 100 。

接下來有 m 行，每一行兩個整數 a 與 b 代表一個道路連接的兩個地點編號。

n 不超過 5×10^4，m 不超過 5×10^5 。

兩點之間可能有多條道路，有些道路的兩端點可能是同一地點。

輸出說明

最大可以獲得的寶藏總價值。

範例測試

範例輸入 1	範例輸出 1
7 6 5 2 4 2 1 1 8 5 1 1 3 1 4 2 0 2 0 3 3	9

Tip: 通常題目給的圖不一定可以從起點到達所有的點，這樣我們會說這個圖是非連通的，反之從起點可以到任何地方的稱爲連通圖。

我們可以在 DFS 的過程中記錄並回傳總價值，所以我們的 DFS 不再是 void，而可以是 int (或 long long 之類的)。

因爲圖非連通，所以我們需要對每一個點都做確認，並找出所有的連通分量 (也就是每一個連通的子圖) 裡面所有點的總和的最大值。

```
int val[N];
vector<int> g[N];
bool isv[N];

int dfs(int n){
    int ret=val[n];
    for(int next:g[n]){
        if(!isv[next]){
            isv[next]=true;
            ret+=dfs(next);
        }
    }
    return ret;
}

int main(){
    // input

```

```
19      int ans=0;
20
21      for(int i=0;i<n;++i){
22          if(!isv[i]){
23              isv[i]=true;
24              ans=max(ans,dfs(i));
25          }
26      }
27
28      cout<<ans<<"\n";
29  }
```

Code 4.5: 開車蒐集寶物題解

Problem 4.1.3. CSES 1192 Counting Rooms

題目敘述

給你一張圖，請你算出房間數量。地圖有 $n \times m$ 個格子，每一個格子不是地板就是牆壁。你可以往上下左右的地板行走，所有依照這個方式走的到的位置都算同一個房間。

輸入說明

第一行輸入兩個數字 n, m ，緊接著是 n 行 m 列的字元 (字串)，# 表示牆壁，. 表示地板。

輸出說明

一個整數表示房間數。

範例測試

範例輸入 1	範例輸出 1
5 8 ######## #..#...# ####.#.# #..#...# ########	3

Tip: 可以使用常數陣列表達相對方位，如下：
`const int dr[]{1,0,-1,0}, dc[]{0,1,0,-1};`

Problem 4.1.4. CSES 1193 Labyrinth

題目敘述

給你一張圖，請你從 A 走到 B。地圖有 $n \times m$ 個格子，每一個格子不是地板就是牆壁。你可以往上下左右的地板行走，所有依照這個方式走的到的位置都算同一個房間。

輸入說明

第一行輸入兩個數字 n, m ，緊接著是 n 行 m 列的字元 (字串)，# 表示牆壁，. 表示地板，A 是起點，B 是終點。

輸出說明

如果存在路徑，首先輸出 “YES”，否　打印 “NO”。

如果存在路徑，輸出最短路徑的長度，以及路徑描述，路徑描述是由字元 L(左)、R(右)、U(上) 和 D(下)　成的字串。可以打印任何有效的解答。

範例測試

範例輸入 1	範例輸出 1
5 8 ######## #.A#...# #.##.#B# #......# ########	YES 9 LDDRRRRRU

Problem 4.1.5. CSES 1666 Building Roads

題目敘述

給你 n 個點 m 條無向邊。請你找出要讓這張圖連通的最少所需新增邊數。還有給出具體方案。節點編號爲 $1 \cdots n$。

輸入說明

第一行輸入兩個數字 n, m ，接下來有 m 行，輸入邊 a_i, b_i。

保證沒有自環或重邊。

輸出說明

第一行輸出需要加上幾條邊，第二行以後給出具體加哪條，如果有多解，則輸出任何一個都可以。

範例測試

範例輸入 1	範例輸出 1
4 2 1 2 3 4	1 2 3

4.2 拓鋪排序

4.2.1 概念

在講拓鋪排序前，我們要先講 DAG(Directed Acyclic Graph)，也就是有向無環圖。

有向無環圖，顧名思義就是邊有方向，然後不存在任何的環，也就是說，就算你沒有加上 isv 判定，也不會有 TLE 的問題。

Figure 6.1 就是一個 DAG 的範例。

至於什麼是拓鋪排序，其實我也忘了，所以我去查資料 (太少用)，是這樣的，如

果有一條邊 (a, b)，則我們說 a 一定要在 b 之前被走訪過，

我們可以藉由計算入度 (in degree) 來處理，如果入度爲 0 就將他 push 進到 queue 裡面。

Tip: 入度 ind[x] 就是有幾條邊指向 x。

4.2.2 實作

```
vector<int> ans;
void topological_sort(){
    queue<int> topo;
    int f=0,b=0;
    for(int i=0;i<n;i++){
        if(ind[i]==0){
            topo.push(i);
            b++;
        }
    }

    while(!topo.empty()){
        int now=topo.front();
        topo.pop();
        ans.push_back(now);
        for(auto next:g[now]){
            if(--ind[next]==0){
                topo.push(next);
                b++;
            }
        }
    }
}
```

Code 4.6: 拓鋪排序

4.2.3 範例與練習

Problem 4.2.1. TIOJ 1092 A. 跳格子遊戲

題目敘述

Mimi, Moumou 兩人是青梅竹馬，從小就玩在一起（雖然他們現在也才小學三年級而已）。愛玩的兩人，平常的遊戲玩久了之後就覺得無聊沒勁，常常湊在一起發明新的遊戲。

這天他們又開始在設計新遊戲了，遊戲的規則如下：

先在地上畫 N 個圓圈，編號 1 到 N，並規定 1 號圓圈是起點、N 號是終點

在這 N 個圓圈之間任意互相畫”單向”箭頭，箭頭起點和終點各有一圓圈且兩圓圈不爲同一圓，假設有一箭頭從圓 a 連到圓 b，則遊戲中可以從 a 跳到 b。

檢查 1, 2 步驟所畫出的圖，確保任何編號 1　N-1 的圓都有路徑可以走到終點，同時圖中也不可以有迴圈產生（也就是對於任何一個圓，絕對不會有路徑可以從自己走到自己），對任兩圓 a, b 最多只有一個箭頭從 a 指向 b。

兩個人進行遊戲，開始時先由一人站在起點（1 號圓），另一個人站在圖外。

遊戲中假設甲站在一個圓 C 中而乙在圖外，則乙要從 C 所指出去的箭頭中選一個箭頭，站到這個箭頭所指的圓上，然後甲則離開 C 走到圖外。

兩人重複步驟 5 的動作，直到其中一人到達終點，到達終點的人爲贏家。

輸入說明

輸入檔含有多筆測資，每筆資料第一行有兩個數字 $N, E(1 \leq N \leq 10000,\ E \leq 10N)$。接下來 E 行每行有兩個數字 $a, b(1 \leq a, b \leq N)$，表示有箭頭從 a 指向 b。你可以假設輸入檔所代表的圖都是有效的。最後一行則是遊戲開始時站在起點上的人名。N 和 E 都是零 “0 0” 表示檔案結束。

輸出說明

假設遊戲爲 Mimi 和 Moumou 兩人進行，且兩人皆十分聰明，如果有必勝的走法那就一定會贏。

對每筆測資輸出遊戲結束後贏家的名字 (Mimi 或 Moumou)，佔一行。

範例測試

範例輸入 1	範例輸出 1
5 6 1 2 1 3 2 3 2 4 3 5 4 5 Mimi 0 0	Moumou

Problem 4.2.2. Sprout OJ 165 陣線推進

題目敘述

如果你有玩過策略遊戲，尤其是 AOC，的話，一定曉得「戰線」是一種很重要的概念。大致上這概念可以這樣說明：如果對方設置陣地得宜，那麼在攻破某些陣地之前，一定要先攻破某些陣地，否則會導致我軍不是被城牆擋住，就是會被兩邊夾擊、全軍覆沒。

一般來說兩個玩家單挑對決時，因爲通常都是互相快攻，彼此的陣地都還只有本陣，這方面問題就不大；但是當玩家數量變多，例如變成 4v4 團體戰的時候，隨著遊戲時間增常，陣地之間的關係也容易變得非常複雜，想攻破 A 要攻破 B 和 C、想攻破 B 又要先攻破 D 和 E 等等。爲了避免兵力的大幅損傷，你派出了大量的斥候犧牲小我完成大我，先確認了各個陣地的配置以及位置，接著便是擬定作戰計畫的時候了。如果已經知道要攻破某個陣地前要先攻破哪些陣地，你能否給出一個方案使得根據這個方案進行陣地攻略可以滿足所有要求條件、在全程都

不腹背受敵的情況下與對方決一死戰呢？

輸入說明

輸入的第一行是一個正整數 T。接下來會有 T 組測試資料，每組測試資料的第一行是兩個非負整數 n,m，分別代表陣地的數量與陣地間的依賴關係數量；第二行到第 $m+1$ 行每行由兩個整數 a 和 b $(0 \leq a, b \leq n-1)$ 組成，代表要攻破陣地 b 之前必須先攻破陣地 a，其中陣地從 0 $n-1$ 依序編號。

輸出說明

如果存在題目所要求的方案，爲了方便起見請輸出字典序最小的一個。

否則代表一場喋血的戰役在所難免，只好輸出"QAQ"(不含引號) 了。

Tip: 誰說只能用 queue 呢？

4.3 二分圖判定

4.3.1 概念

有沒有一種方式可以將圖中的點分成兩組，使組內沒有成員之間的邊。如下圖。

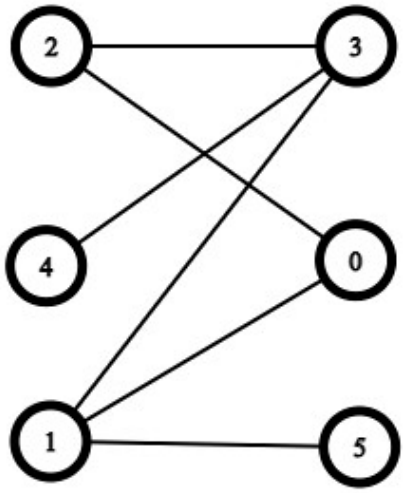


Figure 4.2: 二分圖

判別方法

想像將圖上色，用 12 表示，相鄰的點都用不同色，如果沒有辦法使用就代表這張圖不是二分圖。實作上，如果對方已經跟你有相同顏色表示沒有辦法。

4.3.2 實作

```cpp
int color[N];
bool isBipartiteGraph(int n){
    for(auto u:g[n]){
        // 用 color==0 代替 bool isv[N];
        if(color[u]==0){
            // 有一個邪門的技巧 color[u]=color[n]^3;
            color[u]=color[n]%2+1;
            if(!isBipartiteGraph(u))
                return false;
        }else if(color[u]==color[n]){
            return false;
        }
    }
    return true;
}
```

Code 4.7: 二分圖判定

4.3.3 範例與練習

Problem 4.3.1. TIOJ 1209 圖論之二分圖測試

題目敘述

給你一個圖 (Graph)，請問這個圖是否爲一個二分圖 (bipartite graph)？

所謂的二分圖，就是存在一種分法，把所有頂點分成兩個點集 X 和 Y，其中 X 以及 Y 內部的頂點互不相鄰。

輸入說明

輸入可能包含多筆測試資料。

每筆測試資料的第一列有兩個整數 $n, m (1 \leq n \leq 40,000$，$0 \leq m \leq 500,000)$，分別代表一個圖的點數和邊數。點的編號是從 1 到 n。

接下來有 m 列，每列有兩個以空白隔開的正整數，代表一條邊所連的兩個端點編號。當 $n = m = 0$ 時代表輸入結束。

輸出說明

對於每筆測試資料，若該圖是二分圖，請輸出 Yes，否則輸出 No。

範例測試

範例輸入 1	範例輸出 1
3 2 2 3 1 2 3 3 1 2 2 3 3 1 0 0	Yes No

Problem 4.3.2. ZJ g598 眞假子圖

題目敘述

情報調查局內有 n 個工作人員，調查局負責人將這些人秘密分成兩組 A 和 B 並不讓其他人知道，並將合作名單分配給組長，合作名單是由很多個 pair 組成，每個 pair x, y 代表 x 和 y 需要合作完成任務，並且保證 x 和 y 不會同時在 A 組或是同時在 B 組。

組長不小心將這個合作名單分配遺失，僅剩下其中 m 個 pair，爲了要復原這些失去的資料，組長派出了另外 p 個調查員編號 1 到 p 去調查這個合作關係，每一個調查員都會回傳恰好 k 個 pair 的資料回來

有些調查員回傳的資料和組長手上的資料會產生矛盾 (意即加上這 k 個 pair 和組長手上存留的 m 個 pair 會使得這些人是被分成 A，B 兩組這件事產生矛盾)，請將回傳錯誤結果的調查員編號由小到大輸出出來，保證至少一個且最多三個。

另外保證若調查員的 k 個 pair 的結果和組長存留的 m 個 pair 不會產生矛盾，則保證調查員的資料一定和原本 A，B 分組吻合。

輸入說明

第一行先輸出兩個正整數 n 和 m

第二行來有 2m 個非負整數兩兩形成一個數對，表示目前還留存的 m 個 pair

第三行有兩個正整數 p 和 k

並且接下來的 p 行每行有 2k 個非負整數, 兩兩形成一對代表某個調查員找到的 k 個 pair

輸出說明

由小到大輸出會形成矛盾的調查員編號，每個編號各自獨立一行。

範例測試

範例輸入 1	範例輸出 1
7 5 0 1 0 2 1 3 2 3 4 5 2 3 0 6 2 4 3 6 0 6 0 3 3 5	2

Problem 4.3.3. AtCoder ABC 282D Make Bipartite 2

題目敘述

給定一個包含 N 個頂點和 M 條邊（一個簡單圖不包含自環或多邊）的簡單無向圖 G。對於 $i = 1, 2, \cdots, M$，第 i 條邊連接頂點 u_i 和頂點 v_i。

請打印滿足以下兩個條件的整數對 (u, v) 的數量，其中 $1 \leq u < v \leq N$。

圖 G 沒有連接頂點 u 和頂點 v 的邊。

在圖 G 中添加一條連接頂點 u 和頂點 v 的邊會得到一個二部圖。

$2 \leq N \leq 2 \times 10^5$

$0 \leq M \leq \min 2 \times 10^5, N(N-1)/2$

$1 \leq u_i, v_i \leq N$

輸入說明

輸入以以下格式從標準輸入中給出。

$N\ M$

$u_1\ v_1$

$u_2\ v_2$

$\vdots$

$u_M\ v_M$

輸出說明

輸出答案。

範例測試

範例輸入 1	範例輸出 1
5 4 4 2 3 1 5 2 3 2	2

4.4 最短路徑

4.4.1 概念

最短路徑應該是目前最有功能的演算法了，他在導航裡面頻繁地出現。

以下不同的演算法概念各不相同，甚至有動態規劃的 (在後面的章節 QQ)。其中，Bellman-Ford 與 Dijkstra 處理的是從某個確定的點出發，到所有點的最短距離。而 Floyd-Warshall 則可以算出所有點對的最短路徑，不過當然就會花更多時間啦。

4.4.2 Bellman-Ford

概念

我們需要先了解 "鬆弛的概念"，鬆弛就是

尋找兩點的最短路徑時，最簡單的方法就是，先找一條，然後再找找看有沒有更短的。最後留下最短的。

找更短的路徑並不困難。我們可以一直找一直找，直到找到最短路徑。

不過比較快的方法是找捷徑，就是盡可能縮短距離。

因此，在 Bellmen-Ford 演算法中，我們每一次都對每個邊確認，他是否可以讓他要到的地方可以有更近的距離，這樣做過 n 次之後，理論上就可以知道一個點到所有點的最短距離了。

也因此，這個算法的複雜度爲 $O(nm)$，其中 n 表示點的數量，m 表示邊的數量。

實作

```
const int INF=0x3f3f3f3f;

int dis[N];
void Bellman_Ford(int s){
    fill(dis,dis+N,INF);
    dis[s]=0;

    while(true){
        bool update=false;
        for(auto e:es){
            if(dis[e.from]!=INF && dis[e.from]+e.dis<dis[e.to]){
                update=true;
                dis[e.to]=dis[e.from]+e.dis;
            }
        }

        if(!update){
            break;
        }
    }
}
```

Code 4.8: Bellmen-Ford 算法

4.4.3 Dijkstra

概念

這個演算法與前一個並不一樣。雖然同樣是鬆弛，可是如果所有的邊權重皆爲正，也就是不可能距離越走越短的話，我們可以使用資料結構讓計算複雜度下降。

提出這樣的 Greedy 想法的就是 Dijkstra。我們每一次都往距離最近的點走，如此一來，因爲我們邊的權重都是正的，所以不會有其他的邊可以更快走到那。

實作上，因爲我們要找不斷找最近的，所以會需要一直求最小值，所以我們可以使用`priority_queue`。

實作

```
#define to second
#define w first
#define INF 0x3f3f3f3f
using pii=pair<int,int>;

vector<pii> g[100010];
```

```
7
8  int main(){
9      // input
10
11     vector<int> dis(n+1,INF);
12     priority_queue<pii,vector<pii>,greater<pii>> pq;
13
14     dis[f]=0;
15     for(auto i:g[f]){
16         pq.push(make_pair(dis[f]+i.to,i.w));
17     }
18
19     for(int i=0;i<n-1;i++){
20         auto u=pq.top();
21         pq.pop();
22
23         if(dis[u.to]!=INF) continue;
24
25         dis[u.to]=u.w;
26         for(auto j:g[u.to]){
27             if(dis[j.w]==INF){
28                 pq.push(make_pair(dis[u.to]+j.to,j.w));
29             }
30         }
31     }
32 }
```

Code 4.9: Dijkstra 算法

4.4.4 Floyd-Warshall

概念

dp[i][j][k] 表示經過前 k 個點“鬆弛”後從 i 到 j 的最短距離。

那什麼是經過第 k 個點鬆弛呢？就是確認，如果從 a 到 b 先經過 k 會不會比較快。

因此經過一系列的通靈後，我們可以得知下式。

```
dp[i][j][k] = min(dp[i][j][k-1],dp[i][k][k-1]+dp[k][j][k-1]);
```

由於 dp 過程中只會用到 dp[i][j][k-1] ，所以可以重複利用陣列。最後就像底下的 dp 式一樣可以只使用二維。

```
dp[i][j] = min(dp[i][j],dp[i][k]+dp[k][j])
```

實作

實作上需要注意的有兩個，第一個是我們需要使用鄰接矩陣，第二個是沒有邊的點要打標記，或是設爲 INF(超大的數字)。

```
1
2  void init(){
3      for(int i=1;i<=n;++i)
4          for(int j=1;j<=n;++j)
```

```
            dp[i][j]=INF;
}

void FloydWarshall(){
    for(int k=1;k<=n;++k){
        for(int i=1;i<=n;++i){
            for(int j=1;j<=n;++j){
                dp[i][j]=min(dp[i][j],dp[i][k]+dp[k][j]);
            }
        }
    }
}

// 也可以透過巨集簡化
#define REP(i,s,n) for(int i=(s);i<=(n);++i)

void FloydWarshall(){
    REP(k,1,n) REP(i,1,n) REP(j,1,n){
        dp[i][j]=min(dp[i][j],dp[i][k]+dp[k][j]);
    }
}
```

Code 4.10: Floyd-Warshall 算法

4.4.5 範例與練習

Problem 4.4.1. CSES 1671 Shortest Routes I

題目敘述

有 n 個城市和 m 個城市之間的航班連接。你的任務是確定從 Syrjälä 到每個城市的最短路徑長度。

輸入說明

第一行有兩個整數 n 和 m，表示城市數量和航班連接數量。城市編號爲 $1, 2, ..., n$，城市 1 爲 Syrjälä。

接下來的 m 行描述了航班連接。每行有三個整數 a, b 和 c：一個航班從城市 a 開始，到達城市 b，航班長度爲 c。每個航班都是單程的。

你可以假設從 Syrjälä 到所有其他城市都是可行的。

$1 \leq n \leq 10^5$，$1 \leq m \leq 2 \cdot 10^5$，$1 \leq a, b \leq n$，$1 \leq c \leq 10^9$。

輸出說明

輸出 n 個整數，表示從 Syrjälä 到城市 $1, 2, \cdots, n$ 的最短路徑長度。

範例測試

範例輸入 1	範例輸出 1
3 4 1 2 6 1 3 2 3 2 3 1 3 4	1

Problem 4.4.2. CSES 1672 Shortest Routes II

題目敘述

有 n 個城市和 m 條道路相連。你的任務是處理 q 個查詢，其中你需要確定兩個給定城市之間的最短路徑長度。

輸入說明

第一行有三個整數 n，m 和 q，表示城市數量、道路數量和查詢數量。

接下來有 m 行描述道路。每行有三個整數 a，b 和 c，表示城市 a 和城市 b 之間有一條長度爲 c 的道路。所有道路都是雙向的。

最後有 q 行描述查詢。每行有兩個整數 a 和 b，表示要求解的兩個城市之間的最短路徑長度。

$1 \le n \le 500$，$1 \le m \le n^2$，$1 \le q \le 10^5$，$1 \le a, b \le n$，$1 \le c \le 10^9$

輸出說明

對於每個查詢，輸出最短路徑的長度。如果不存在路徑，輸出 -1。

範例測試

範例輸入 1	範例輸出 1
4 3 5 1 2 5 1 3 9 1 2 2 1 1 3 1 4 3 2	5 5 8 -1 3

Problem 4.4.3. CSES 1673 High Score

題目敘述

你玩一個遊戲，遊戲中有 n 個房間和 m 條通道。你的初始分數爲 0，每通過一條通道，你的分數增加 x，其中 x 可以爲正數或負數。你可以通過一條通道多次。

你的任務是從第一個房間走到第 n 個房間。你能獲得的最大分數是多少？

輸入說明

第一行有兩個整數 n 和 m，表示房間數量和通道數量。房間編號爲 $1, 2, \cdots, n$。

接下來有 m 行描述通道。每行有三個整數 a，b 和 x，表示通道從房間 a 開始，到房間 b 結束，通過該通道可以增加分數 x。所有通道都是單向通道。

你可以假設從第一個房間到第 n 個房間是可行的。

$1 \le n \le 2500$，$1 \le m \le 5000$，$1 \le a, b \le n$，$-10^9 \le x \le 10^9$

輸出說明

輸出一個整數，表示你能獲得的最大分數。如果你能獲得任意大的分數，輸出-1。

範例測試

範例輸入 1	範例輸出 1
4 5 1 2 3 2 4 -1 1 3 -2 3 4 7 1 4 4	5

Problem 4.4.4. TIOJ 1096 E. 漢米頓的麻煩

題目敘述

轉圈圈捷運公司一向是以多樣性的路線，繞不完的圈圈，以及每條路線上搭配的行程而享譽國際。不過也因爲這樣，太多的路線交錯複雜，再也沒有人能夠分清楚哪一段是哪的路線，以及總共有多少的圈圈。

轉圈圈捷運公司的董事長漢米頓，一直有想要作一件事情，就是想要把捷運路線上，每一種圈圈的路線要坐上一遍，而他的這個計畫已經開始實施了。

但是，幾天之後他發現了他的想法幾乎不可能實現：他已經轉到快要吐出來的！而且當他嘗試著要把所有的環狀路線全部都列出來，寫在他的記事本上的時候，他才發現這是一個很難的問題，他根本不知道有多少條的圈圈路線！他列出了很多，也坐過了很多圈圈的路線，可是他沒有辦法知道說又沒有漏了哪一條環狀路線沒有坐過。

漢米頓花了很多的時間，才發現了這個問題很難，很難。不過漢米頓並不放棄，雖然他以他一個人的力量沒有辦法做到，但是他想到了一個辦法！

轉圈圈捷運公司在近日內，會舉辦一個轉圈圈換獎品的活動，如果搭乘捷運的旅客所搭乘的路線，起點和終點是同一個的話，就可以得到一份價值 150 元的獎品。當然，這條路線至少要有搭上捷運才算，直接進站又出站的話可是不能算的！

轉圈圈捷運公司在計算車資的時候，是依照旅客從進站到出站的時間來計算的，如果坐了 15 分鐘的車，那麼車費就會是 15 元。

漢米頓對於自己的想到的這個辦法很滿意，但是他還是有一點擔心：如果有人坐環狀路線的車資比 150 元還要少，那公司可就虧大了 ... 所以現在的他正在想辦法找找看有沒有哪一條環狀路線上的車資是會小於 150 元的。

看到漢米頓這麼煩惱，聰明的你應該想到好方法幫幫他了吧！可以寫個程式幫忙他找到在轉圈圈捷運公司的路線中，所花費時間最少的一條環狀路線，要坐多久呢？

輸入說明

輸入檔中有許多組輸入，每組輸入的第一行有一個數字 N （$1 \le N \le 100$），代表

總共有 N 個捷運站。接下來有 N 行，每行有 N 個數字、以空白分隔，第 i 行的第 j 個數字 T_{ij} 代表編號 i 的捷運站有一條路線到編號 j 的捷運站，車程 T_{ij} 分鐘（$0 \leq T_{ij} \leq 1000$），如果這個數字是 0 代表兩站之間沒有直接相連。$N = 0$ 表示檔案結束。

輸出說明

對每組輸入輸出一行，包含一個數字，表示花費最短的環狀路線所花的時間。如果找不到任何一條環狀路線，則輸出 -1。

範例測試

範例輸入 1	範例輸出 1
4 0 0 2 0 4 0 0 0 0 3 0 1 0 1 0 0 0	8

4.5 最小生成樹

4.5.1 概念

最小生成樹常常用在處理道路規劃問題，雖然我們沒有直接的看見他在生活中的應用，但是肯定有，而且競賽常常會考喔。

那什麼是最小生成樹？就是這個樹的邊集 $\in$ 圖的邊集合的子集，所有這樣的集合中，邊權總和最小的那一個。

同樣的，兩種演算法的概念並不相同，所以就讓我分別講解吧。

4.5.2 Kruskal

概念

我們盡可能加入權重最小的邊，如果加入後會產生環就不加，就這樣跑完所有的邊就會找到最小生成樹了。

判斷會不會產生環需要 DSU，把加入的邊 a,b 兩端所屬的連通分量合併，這恰好是並查集最擅長的操作。

排序會是複雜度最高的操作，因此時間複雜度爲 $O(m \log(n))$。

實作

```
struct edge{
    int f,t,w;
};
bool operator<(edge a,edge b){
    return a.w<b.w;
```

```
}

int dsu[10010],rk[10010];
void init(int n){
    for(int i=0;i<=n;i++){
        dsu[i]=i;
        rk[i]=1;
    }
}

int find(int a){
    if(dsu[a]==a)
        return a;
    return find(dsu[a]);
}

bool same(int a,int b){
    return find(a)==find(b);
}

void uni(int a,int b){
    if(rk[find(a)]==rk[find(b)]){
        dsu[find(b)]=dsu[find(a)];
        rk[a]++;
    }else if(rk[find(a)]>rk[find(b)]){
        dsu[find(b)]=dsu[find(a)];
    }else{
        dsu[find(a)]=dsu[find(b)];
    }
}

int main(){
    // input

    sort(g.begin(),g.end());

    int ans=0,ct=0;

    init(n);
    for(auto e:g){
        if(!same(e.f,e.t)){
            uni(e.f,e.t);
            ct++;
            ans+=e.w;
        }
    }
}
```

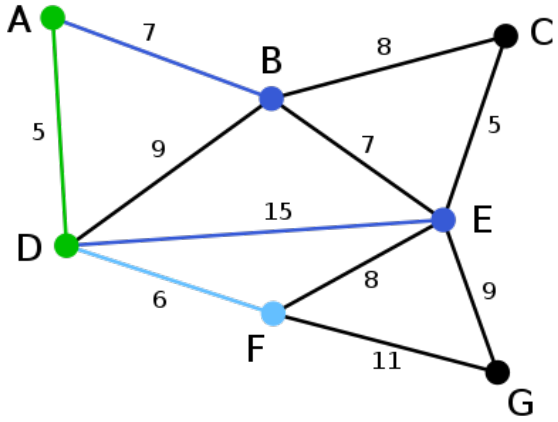
Code 4.11: Kruskal 算法

4.5.3 Prim

因爲我沒有用過 Prim，而且兩者的時間複雜度同爲 $O(m \log(n))$。

不過還是稍微介紹一下概念。他有點像 Dijkstra，不過是每次都把距離已經確定的點集合最近的邊加入。

如圖，下一個頂點爲距離 D 或 A 最近的頂點。B 距 D 爲 9，距 A 爲 7，E 距 D 爲 15，F 距 D 爲 6。因此，F 距 D 或 A 最近，因此將頂點 F 與相應邊 DF 圖上比較亮的顏色。



我們同樣可以利用priority_queue提升效率，所以同樣是 $O(m\log(n))$。

4.5.4 範例與練習

Problem 4.5.1. Sprout OJ 734 模板題

Problem 4.5.2. 洛谷 P1194 買禮物

題目敘述

又到了一年一度的明明生日了，明明想要買 B 樣東西，巧的是，這 B 樣東西價格都是 A 元。

但是，商店老闆說最近有促銷活動，也就是：如果你買了第 I 樣東西，再買第 J 樣，那麼就可以只花 $K_{I,J}$ 元，更巧的是，$K_{I,J}$ 竟然等於 $K_{J,I}$。

現在明明想知道，他最少要花多少錢。

輸入說明

第一行兩個整數，A, B。

接下來 B 行，每行 B 個數，第 I 行第 J 個爲 $K_{I,J}$。

我們保證 $K_{I,J} = K_{J,I}$ 並且 $K_{I,I} = 0$。

特別的，如果 $K_{I,J} = 0$，那麼表示這兩樣東西之間不會導致優惠。

$1 \le B \le 500, 0 \le A, K_{I,J} \le 1000$

輸出說明

一個整數，爲最小要花的錢數。

範例測試

範例輸入 1	範例輸出 1
3 3 0 2 4 2 0 2 4 2 0	7

Problem 4.5.3. 洛谷 P1396 營救

題目敘述

"咚咚咚……"，"查水表！" 原來是查水表來了，現在哪裡找這麼熱心上門的查表員啊！小明感動得熱淚盈眶，開起了門 ...

媽媽下班回家，街坊鄰居說小明被一群陌生人強行押上了警車！媽媽豐富的經驗告訴她小明被帶到了 t 區，而自己在 s 區。

該市有 m 條大道連接 n 個區，一條大道將兩個區相連接，每個大道有一個擁擠度。小明的媽媽雖然很著急，但是不願意擁擠的人潮衝亂了她優雅的步伐。所以請你幫她規劃一條從 s 至 t 的路線，使得經過道路的擁擠度最大值最小。

輸入說明

第一行有四個用空格隔開的 n，m，s，t，其含義見【題目敘述】。

接下來 m 行，每行三個整數 u, v, w，表示有一條大道連接區 u 和區 v，且擁擠度爲 w。

$1 \le n \le 10^4$，$1 \le m \le 2 \times 10^4$，$w \le 10^4$，$1 \le s, t \le n$。且從 s 出發一定能到達 t

輸出說明

輸出一行一個整數，代表最大的擁擠度。

範例測試

範例輸入 1	範例輸出 1
3 3 1 3 1 2 2 2 3 1 1 3 3	2

Chapter 5

進階資料結構

5.1 懶人標記

5.1.1 概念

之前的線段樹只有單點修改的功能，而若需要區間加值的話則會很費時，因此有人想到一個方法：若要修改的區間剛好完全包含線段樹上的某個區間的話，就可以先在上面打一個標記。等到需要動到他下面的子區間再向下放，因爲我們一次最多在 $O(\log n)$ 個區間打上標記，所以區間修改也可以從 $O(n\log(n))$ 進步到 $O(\log(n))$ 。

5.1.2 實作

首先我們需要在 node 裡面增加一些東西，因爲標記稱爲 Tag，所以以下程式碼中的 tag 表示我們的標記。

```
struct node{
    // value 和 tag
    int val,tag;

    // 子樹的指標
    node *rch,*lch;

    // 建構式
    node(){
        val=tag=0;
        rch=lch=nullptr;
    }

    // 帶有初始值的建構式
    node(int v){
        val=v, tag=0;
        rch=lch=nullptr;
    }
};
```

Code 5.1: 懶人標記的 node

接著，我們的 node 裡面會有一些函式，我以區間加值 (將一段區間都加上某個數) 爲例。

```
struct node{
    // 單點修改
    void modify(int p,int v,int lb=1,int rb=n){
        // 終止條件：區間長度爲1
        if(lb==rb){
            val=v;
            return;
        }

        // 若左右子樹沒有結點則開一個新的
        if(!lch) lch=new node();
        if(!rch) rch=new node();

        push(lb,rb);

        // 設mid爲區間中點，均分爲左右區間
        // >>1相當於/2，但快很多
        int mid=lb+rb>>1;

        if(p<=mid) lch->modify(p,v, lb ,mid);
        if(mid<p)  rch->modify(p,v,mid+1,rb);

        // 還記得子樹修改完要做什麼？
        pull();
    }

    int query(int l,int r,int lb=1,int rb=n){
        if(l<=lb && rb<=r){
            return val;
        }

        push(lb,rb);

        int mid=lb+rb>>1;

        int ret=0;
        if(lch && l<=mid) ret+=lch->query(l,r, lb ,mid);
        if(rch && mid<r)  ret+=rch->query(l,r,mid+1,rb);

        pull();

        return ret;
    }

    // 區間加值
    void add(int l,int r,int v,int lb=1,int rb=n){
        // 終止條件：所在區間位於欲查詢區間之中
        if(l<=lb && rb<=r){
            // 由於區間爲閉區間[lb,rb]因此要加1
```

```
50              val+=v*(rb-lb+1);
51              tag+=v;
52              return;
53          }
54
55          push(lb,rb);
56
57          int mid=lb+rb>>1;
58
59          int ret=0;
60          if(lch && l<=mid) lch->add(l,r,v, lb ,mid);
61          if(rch && mid<r)  rch->add(l,r,v,mid+1,rb);
62
63          pull();
64      }
65  };
```

Code 5.2: 懶人標記的 node

另一種方式是用陣列，我們可以另外開一個陣列稱爲 tag。具體的程式架構與上述的指標型線段樹大致相同，就留給各位自行練習了。

5.1.3 範例與練習

Example 5.1.1. TIOJ 1224 矩形覆蓋面積計算

題目敘述

給你很多平面上的矩形，請求出它們覆蓋的總表面積。

輸入說明

輸入檔案只包含一筆測試資料。

第一行包含一個正整數 n，表示矩形的數量 $(1 \leq n \leq 100,000)$。

接下來的 n 行，每行包含四個整數 L, R, D, U $(0 \leq L < R \leq 1,000,000$；$0 \leq D < U \leq 1,000,000)$，代表矩形的左、右、下、上四個邊界座標。

輸出說明

請輸出覆蓋的總面積。

範例測試

範例輸入 1	範例輸出 1
2 1 10 1 10 0 2 0 2	84

想法

想像有一條線從上往下 (或從下往上) 掃描，每個矩形的上邊表示進入，下邊表示出來，我們可以藉由維護這個區間有多少地方是有被矩形覆蓋的直到有所變化 (進或出)，這所有區間長度總和，乘上上次有變化到這次有變化前高度就是這一段的面積。

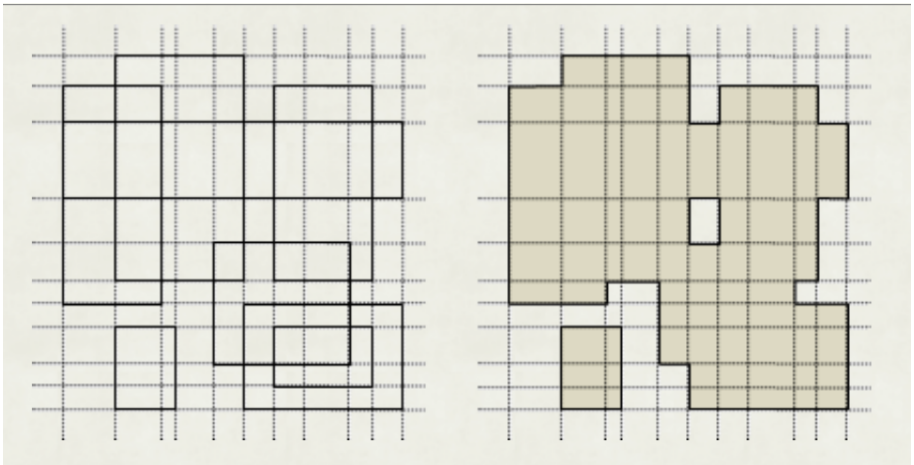


Figure 5.1: 矩形覆蓋面積示意圖

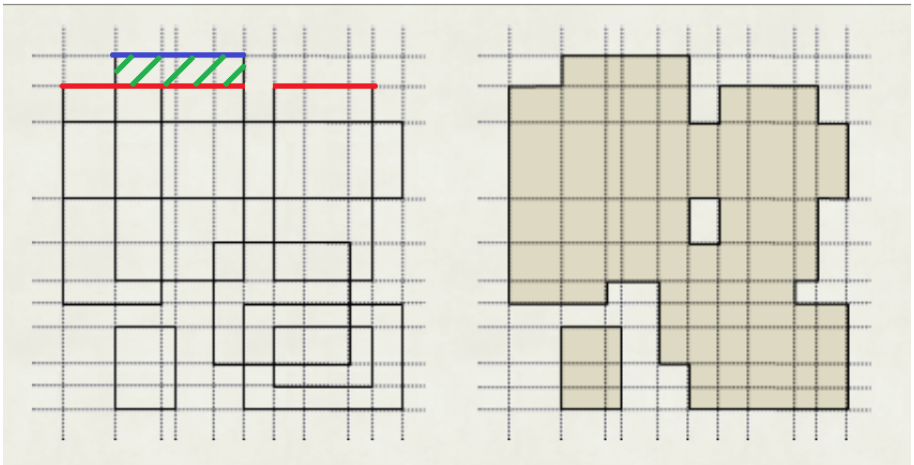


Figure 5.2: 第一個變化

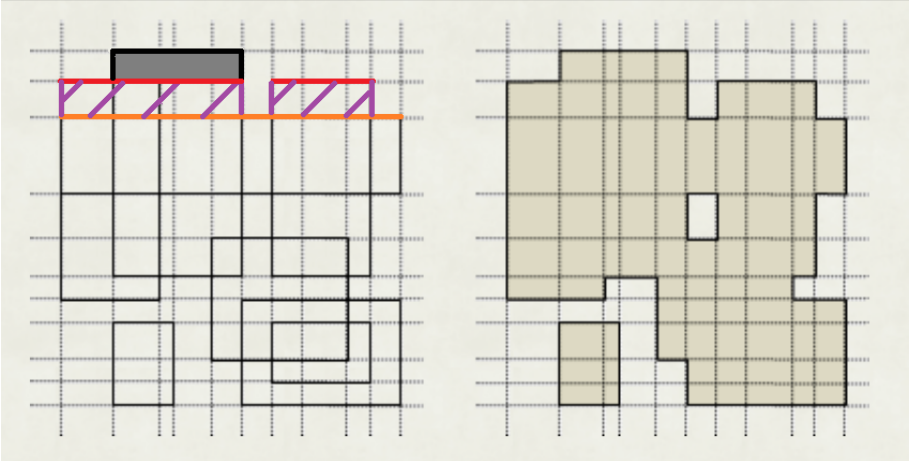


Figure 5.3: 第二個變化

實作

首先，我們需要一棵可以區間加值的線段樹。接著對所有的變化做排序，最後一一計算分部的面積。因爲這題比較特殊，所以我貼上完整的程式碼。

注意：因爲我們只要查詢所有的和，所以我們可以使用跟節點的值就好。

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

struct node{
    int ct=0,sum=0;
}seg[3050000];

struct line{
    int l,r,x,m;
    line(){
        l=r=x=0;
    }
    line(int a,int b,int c,int i){
        l=a;r=b;x=c;m=i;
    }
};
bool operator<(line a,line b){
    return a.x<b.x;
}

int n,MXN;
vector<line> lns;

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

void init(){
    MXN=1;
    while(MXN<1000010)
        MXN<<=1;
}

void add(int l,int r,int v,int lb=1,int rb=MXN,int idx=1){
    if(l<=lb && rb<=r){
        seg[idx].ct+=v;
        if(seg[idx].ct>0) seg[idx].sum=rb-lb+1;
        else pull(idx);
        return;
    }

    int mid=lb+rb>>1;
    if(l<=mid) add(l,r,v,lb,mid,idx*2);
    if(mid+1<=r) add(l,r,v,mid+1,rb,idx*2+1);

    if(seg[idx].ct>0) seg[idx].sum=rb-lb+1;
    else pull(idx);
}

int main(){
```

```
51      ios::sync_with_stdio(0);cin.tie(0);
52
53      cin>>n;
54      for(int i=0;i<n;++i){
55          int l,r,d,u;
56          cin>>l>>r>>d>>u;
57          lns.emplace_back(line(l+1,r,d,1));
58          lns.emplace_back(line(l+1,r,u,-1));
59      }
60      sort(lns.begin(),lns.end());
61      init();
62
63      long long sum=0,pre=lns.front().x;
64      for(auto ln:lns){
65          if(ln.x != pre){
66              sum+=(seg[1].sum)*(ln.x-pre);
67              pre=ln.x;
68          }
69          add(ln.l,ln.r,ln.m);
70      }
71      sum+=(seg[1].sum)*(lns.back().x-pre);
72
73      cout<<sum;
74  }
```

Code 5.3: 矩形覆蓋面積題解

Problem 5.1.2. 洛谷 P3870 【TJOI2009】開關

題目敘述

現有 n 盞燈排成一排，從左到右依次編號爲 1，2，……，n。然後依次執行 m 項操作。

操作分爲兩種：

1. 指定一個區間 $[a, b]$，然後改變編號在這個區間內的燈的狀態 (把開著的燈關上，關著的燈打開)。
2. 指定一個區間 $[a, b]$，要求你輸出這個區間內有多少盞燈是打開的。

燈在初始時都是關著的。

輸入說明

第一行有兩個整數 n 和 m，分別表示燈的數目和操作的數目。

接下來有 m 行，每行有三個整數，依次爲：c、a、b。其中 c 表示操作的種類。

1. 當 c 的值爲 0 時，表示是第一種操作。
2. 當 c 的值爲 1 時，表示是第二種操作。

a 和 b 則分別表示了操作區間的左右邊界。

$2 \le n \le 10^5$，$1 \le m \le 10^5$，$1 \le a, b \le n$，$c \in \{0, 1\}$

輸出說明

每當遇到第二種操作時，輸出一行，包含一個整數，表示此時在查詢的區間中打開的燈的數目。

範例測試

範例輸入 1	範例輸出 1
4 5 0 1 2 0 2 4 1 2 3 0 2 4 1 1 4	1 2

Problem 5.1.3. 洛谷 P1438 無聊的數列

題目敘述

無聊的 YYB 總喜歡搞出一些正常人無法搞出的東西。有一天，無聊的 YYB 想出了一道無聊的題：無聊的數列。。。(K 峰：這題不是傻 X 題嗎)

維護一個數列 a_i，支持兩種操作：

`1 l r K D`：給出一個長度等於 $r-l+1$ 的等差數列，首項爲 K，公差爲 D，並將它對應加到 $[l, r]$ 範圍中的每一個數上。即：令 $a_l = a_l + K, a_{l+1} = a_{l+1} + K + D \ldots a_r = a_r + K + (r-l) \times D$。

`2 p`：詢問序列的第 p 個數的值 a_p。

輸入說明

第一行兩個整數數 n, m 表示數列長度和操作個數。

第二行 n 個整數，第 i 個數表示 a_i。

接下來的 m 行，每行先輸入一個整數 opt。

若 $opt = 1$ 則再輸入四個整數 l r K D；

若 $opt = 2$ 則再輸入一個整數 p。

$0 \le n, m \le 10^5, -200 \le a_i, K, D \le 200, 1 \le l \le r \le n, 1 \le p \le n$

輸出說明

對於每個詢問，一行一個整數表示答案。

範例測試

範例輸入 1	範例輸出 1
5 2 1 2 3 4 5 1 2 4 1 2 2 3	6

5.2 動態開點

5.2.1 前言

其實前面的指標型線段樹就是了，不過還是讓我來帶你們了解一下原理。

5.2.2 概念

首先是爲什麼要動態開點？當然是爲了省記憶體。如果你有越多節點，你消耗的記憶體就越多。有時候會遇到數值範圍過大的情形，我們就可以做動態開點。那一般而言什麼是過大的範圍呢？只要超過 10^6 的級距就可以算是過大的範圍。

有一個常見的問題是，爲什麼不使用離散化 (編到這邊我才發現我好像沒有編進第三章，於是繼續趕工)。答案是因爲，有時候我們需要確切的值，例如上一個單元的矩形覆蓋面積問題，如果你用離散化，那你的值就跑掉了，這樣就會算出錯誤的答案。

5.2.3 實作

其實之前那個就是了，所以我不重複貼上。需要注意的是，小心不要存到 nullptr 的節點，因爲會吃到 RE，或 SEG。還有空節點在計算的時候要小心，有些線段樹的空節點是有值的。

5.3 2D 線段樹

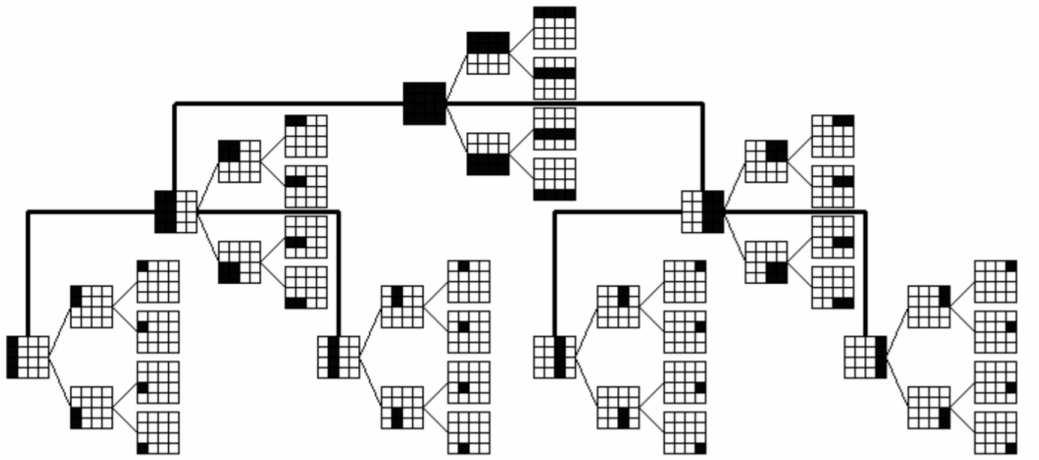
5.3.1 前言

又稱爲樹套樹，有點像是二維前綴和，或是 BIT。不過因爲難很多所以放在進階資料結構篇。

5.3.2 概念

我們將 n 個線段樹，放進線段樹裡面。修改的時候先找到那一棵你要改的樹，再修改裡面你要的節點。至於查詢，我們可以使用與 1D 相同的概念，查詢 $O(\log n)$ 棵樹裡面的 $O(\log m)$ 個節點。

因此，兩種操作的時間複雜度都是 $O(\log n \times \log m)$，通常來說 $n \doteqdot m$，所以也可以記爲 $O(\log^2(n))$。



5.3.3 實作

我們會先實作 seg1D，再用 seg2D 存放。

```
int N,M;
void amax(int &a,int b){
    a=max(a,b);
    return;
}

struct seg1D{
    int val=0;
    seg1D *lch=nullptr,*rch=nullptr;

    void pull(){
        val=0;
        if(lch) amax(val,lch->val);
        if(rch) amax(val,rch->val);
    }

    void modify(int x,int p,int lb=0,int rb=N){
        if(lb==rb){
            amax(val,p);
            return;
        }
        int mid=lb+rb>>1;

        if(x<=mid){
            if(!lch)lch=newseg1D;
            lch->modify(x,p,lb,mid);
        }

        if(mid<x){
            if(!rch) rch=new seg1D;
            rch->modify(x,p,mid+1,rb);
        }
        pull();
    }

```

```
36      int query(int l,int r,int lb=0,int rb=N){
37          if(l<=lb && rb<=r) return val;
38
39          int mid=lb+rb>>1;
40          int ret=0;
41          if(l<=mid && lch) amax(ret,lch->query(l,r,lb,mid));
42          if(mid<r && rch) amax(ret,rch->query(l,r,mid+1,rb));
43          return ret;
44      }
45  };
46
47  struct seg2D{
48      seg1D seg;
49      seg2D *lch=nullptr,*rch=nullptr;
50
51      void modify(int x,int y,int p,int lb=0,int rb=N){
52          seg.modify(y,p);
53          if(lb==rb) return;
54
55          int mid=lb+rb>>1;
56          if(x<=mid){
57              if(!lch) lch=new seg2D;
58              lch->modify(x,y,p,lb,mid);
59          }
60
61          if(mid<x){
62              if(!rch) rch=new seg2D;
63              rch->modify(x,y,p,mid+1,rb);
64          }
65      }
66
67      int query(int xl,int xr,int yl,int yr,int lb=0,int rb=N){
68          if(xr<xl || yr<yl) return 0;
69          if(xl<=lb && rb<=xr) return seg.query(yl,yr);
70          int mid=lb+rb>>1;
71          int ret=0;
72
73          if(xl<=mid && lch) amax(ret,lch->query(xl,xr,yl,yr,lb,mid));
74          if(mid<xr && rch) amax(ret,rch->query(xl,xr,yl,yr,mid+1,rb));
75          return ret;
76      }
77  };
```

Code 5.4: 2D 線段樹

5.3.4 誰說只能放線段樹

其實，你要把 1D 線段樹換成其他東西也可以，例如 BIT，或是 Treap(好期待)。

5.3.5 範例與練習

Problem 5.3.1. ZJ c571 三維偏序

給定 n 個物件，每個物件有三個參數 x_i, y_i, z_i，請問最多可以選幾個物件，使得任意排序後，三個參數皆嚴格遞增。

Problem 5.3.2. CF19D Points

題目敘述

皮特和鮑勃發明了一個新奇的遊戲。鮑勃拿了一張紙，並在上面繪製了一個笛卡爾坐標系：點 $(0,0)$ 位於左下角，x 軸向右延伸，y 軸向上延伸。皮特給鮑勃提供了三種類型的請求：

- add x y：在紙上標記一個坐標爲 (x,y) 的點。對於每個此類型的請求，保證在請求時點 (x,y) 尚未在紙上標記。
- remove x y：在紙上擦除先前標記的坐標爲 (x,y) 的點。對於每個此類型的請求，保證在請求時點 (x,y) 已經在紙上標記。
- find x y：在紙上找到所有標記的點，這些點位於點 (x,y) 的右上方。在這些點中，鮑勃選擇最左邊的點，如果不止一個，選擇最下面的點，並將其坐標返回給皮特。

鮑勃能夠回答 10、100 或 1000 個請求，但當請求的數量增加到 2×10^5 時，鮑勃無法處理。現在他需要一個能夠回答所有皮特請求的程式。請幫助鮑勃！

輸入說明

第一行輸入一個數字 n $(1 \leq n \leq 2 \times 10^5)$ ，表示請求的數量。接下來 n 行描述了每個請求。add x y 表示標記一個點，remove x y 表示擦除一個點，find x y 表示查找底部最左邊的標記點。輸入中所有座標均爲非負數且不超過 10^9。

輸出說明

對於每個 find x y 的請求，輸出一行結果，表示底部最左邊的標記點的座標。如果點 (x,y) 的右上方沒有任何標記點，輸出 -1。

範例測試

範例輸入 1	範例輸出 1
7 add 1 1 add 3 4 find 0 0 remove 1 1 find 0 0 add 1 1 find 0 0	1 1 3 4 1 1

5.3.6 其他

二維線段樹幾乎不支援懶人標記，如果需要的話可以考如果需要的話可以考慮看看 KD 樹，或是四分樹。(有空我再研究看看)

5.4 持久化

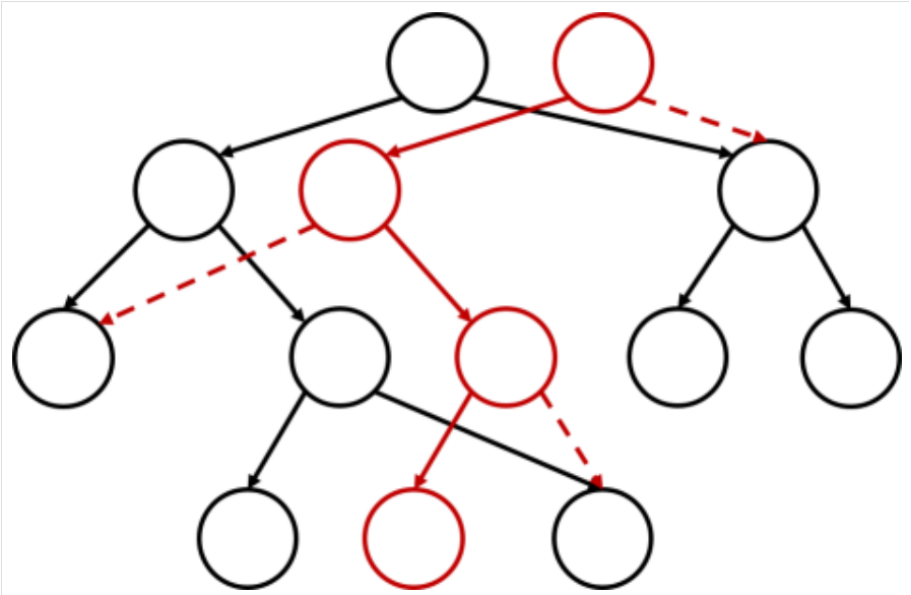
5.4.1 概念

持久化的概念就是保存所有版本，像是我們用電腦時常常使用的 ctrl+z。

想想如果我們要問第 k 次操作後的某段區間，我們有兩種已知做法。

1. 對詢問的 k 做排序，依序處理完再重排輸出。
2. 用 n 棵線段樹，但會花費過多空間。

而持久化就是讓我們可以用最小的空間，備份所有版本。以線段樹而言就可以用共用節點的方式進行。



5.4.2 實作

```
struct node{
    node *lch,*rch;
    int val;

    node(){
        lch=rch=nullptr;
        val=0;
    }

    void pull(){
        val=lch->val + rch->val;
    }
};

```

```
15 void build(int l,int r,node *&p){
16     p=new node();
17     if(l==r) return;
18     int mid=(l+r)>>1;
19     build(l,mid,p->lch);
20     build(mid+1,r,p->rch);
21     p->pull();
22 }
23
24 void modify(int pos,int val,int lb,int rb,node *pre,node *&cur){
25     // 對於有改變的節點開新的
26     cur=new node();
27
28     if(lb==rb){
29         cur->val=pre->val+val;
30         return;
31     }
32
33     // 共用節點
34     cur->lch=pre->lch;
35     cur->rch=pre->rch;
36     int mid=(lb+rb)>>1;
37
38     if(pos<=mid) modify(pos,val,lb,mid,pre->lch,cur->lch);
39     if(mid<pos) modify(pos,val,mid+1,rb,pre->rch,cur->rch);
40
41     cur->pull();
42 }
43
44 int query(int l,int r,int lb,int rb,node *ver){
45     if(l<=lb && rb<=r) return ver->val;
46
47     int ret=0;
48     int mid=(lb+rb)>>1;
49     if(l<=mid) ret+=query(l,r,lb,mid,ver->lch);
50     if(mid<r) ret+=query(l,r,mid+1,rb,ver->rch);
51 }
```

Code 5.5: 持久化線段樹

5.4.3 範例與練習

Example 5.4.1. 洛谷 P3834 【模板】可持久化線段樹 2

題目敘述

這是一個非常經典的可持久化權值線段樹入門題——靜態區間第 k 小。

數據已經過加強，請使用可持久化權值線段樹。同時請注意常數優化。

給定 n 個整數構成的序列 a，對於指定的閉區間 $[l,r]$ 查詢其區間內的第 k 小值。

輸入說明

第一行包含兩個整數，分別表示序列的長度 n 和查詢的個數 m。

第二行包含 n 個整數，第 i 個整數表示序列的第 i 個元素 a_i。

接下來 m 行每行包含三個整數 l, r, k ，表示查詢區間 $[l, r]$ 內的第 k 小值。

$1 \leq n, m \leq 2 \times 10^5$，$|a_i| \leq 10^9$，$1 \leq l \leq r \leq n$，$1 \leq k \leq r - l + 1$

輸出說明

對於每次詢問，輸出一行一個整數表示答案。

範例測試

範例輸入 1	範例輸出 1
5 5 25957 6405 15770 26287 26465 2 2 1 3 4 1 4 5 1 1 2 2 4 4 1	6405 15770 26287 25957 26287

持久化線段樹作法

我們需要使用值域線段樹，存放每個數字出現過幾次，並使用持久化操作，記錄所有版本，從左到右第 N 個版本就是區間 $[1, N]$ 的線段樹。

如此一來，我們就可以執行查詢了，查詢時，我們想要找到第 k 小的數，而第 k 小的數字有一個特點，他大於前面的洽好 $k-1$ 個數，所以我們可以在線段樹上做二分搜，如果他已經比超過 k 個數字大了，就找小一點的數字，反之則找大一點的數字。

接者，因爲值域很大，所以我們可以考慮做離散化，而不是用動態開點 (因爲時間卡的很緊)。

```
struct node{
    node *lch,*rch;
    int val;

    node(){
        lch=rch=nullptr;
        val=0;
    }

    void pull(){
        val=lch->val + rch->val;
    }
};

void build(int l,int r,node *&p){
    p=new node();
    if(l==r) return;
    int mid=(l+r)>>1;
    build(l,mid,p->lch);
    build(mid+1,r,p->rch);
    p->pull();
}

```

```
24 void modify(int pos,int lb,int rb,node *pre,node *&cur){
25     // 對於有改變的節點開新的
26     cur=new node();
27 
28     if(lb==rb){
29         cur->val=pre->val+1;
30         return;
31     }
32 
33     // 共用節點
34     cur->lch=pre->lch;
35     cur->rch=pre->rch;
36     int mid=(lb+rb)>>1;
37 
38     if(pos<=mid) modify(pos,lb,mid,pre->lch,cur->lch);
39     if(mid<pos) modify(pos,mid+1,rb,pre->rch,cur->rch);
40 
41     cur->pull();
42 }
43 
44 int query(int val,int lb,int rb,node *l,node *r){
45     if(lb==rb) return lb;
46     int k=r->lch->val - l->lch->val;
47     int mid=(lb+rb)>>1;
48     if(val<=k) return query(val,lb,mid,l->lch,r->lch);
49     else return query(val-k,mid+1,rb,l->rch,r->rch);
50 }
51 
52 int main(){
53     // input
54     for(int i=0;i<n;++i){
55         modify(v[i],1,mxn,ver[i],ver[i+1]);
56     }
57 
58     int l,r,k;
59     while(m--){
60         // input 查詢範圍與 k
61         query(k,1,mxn,ver[l-1],ver[r])-1;
62     }
63 }
```

Code 5.6: 區間第 k 小題解

Problem 5.4.2. 洛谷 P4587 [FJOI2016] 神秘數

題目敘述

一個可重複數字集合 S 的神秘數定義爲最小的不能被 S 的子集的和表示的正整數。例如 $S = 1, 1, 1, 4, 13$，有：$1 = 1$，$2 = 1+1$，$3 = 1+1+1$，$4 = 4$，$5 = 4+1$，$6 = 4+1+1$，$7 = 4+1+1+1$。

8 無法表示爲集合 S 的子集的和，故集合 S 的神秘數爲 8。

現給定長度爲 n 的正整數序列 a，m 次詢問，每次詢問包含兩個參數 l, r，你需要求出由 $a_l, a_{l+1}, \cdots, a_r$ 所組成的可重集合的神秘數。

輸入說明

第一行一個整數 n，表示數字個數。第二行 n 個正整數，從 1 編號。

第三行一個整數 m，表示詢問個數。接下來 m 行每行包含兩個整數 l, r。

$1 \leq n, m \leq 10^5$，$\sum a \leq 10^9$

輸出說明

對於每次詢問，輸出一行對應的答案。

範例測試

範例輸入 1	範例輸出 1
5 1 2 4 9 10 3 1 1 1 2 1 3	2 4 8

5.5 Treap

這應該是第一個單元標題是英文的了。不過他有中文名，樹堆

5.5.1 前言

Treap 是一個隨機平衡的二元搜尋樹，這裡所說的 Treap 是以 merge/split 的方式實作的，這樣實作的 Treap 有很多好處，我們先說明他的基本規則，以及 merge/split 函式 (function) 在做什麼，接著說明如何實作。

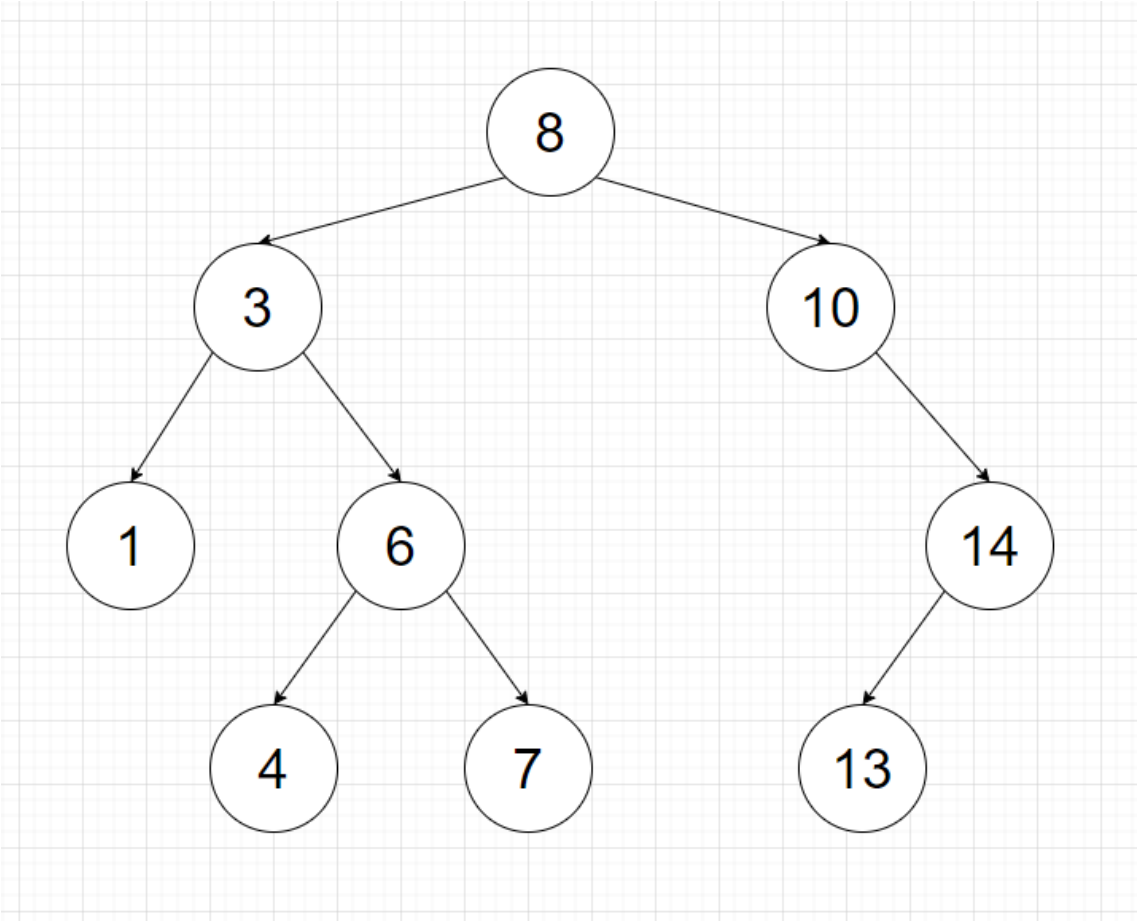
樹堆的名稱來由是樹 (Tree) 和堆積 (Heap) 結合而成，每個節點會至少存放兩個數值 val,pri ，val (也有人叫他 key) 就是內容，而 pri 則是用以平衡。

5.5.2 規則與前置作業

所有節點的 pri 值必須比他的左右子樹小 (大也可以)，而且樹上的 val 滿足 "二元搜尋樹" 的性質，爲了避免有人忘記或不知道什麼是 "二元搜尋樹" 的性質，以下用一個我出過的題目說明。

Example 5.5.1. 第一屆卓越盃 C.Safe Sorter

- 如果該節點還沒有任何保險箱，此保險箱就會佔據該節點
- 否則若這個要放入的保險箱裡面的金塊比在該節點的保險箱還要多，他會被往右邊送
- 否則就往左邊送
- 必須送到他占用一個節點爲止



如圖

- 第一個放進來的保險箱有 8 個金塊
- 第二個有 10 個，因此他被放在右邊
- 第三個有 14 個，因此第一步會先往右送，發現右邊的節點也有保險箱了，因而再次被往右送
- 第四個有 3 個，於是被放在左邊
- 其他依此類推

而所謂隨機平衡就是他的平衡方式來自所有節點被賦予的一個隨機的 pri 值。Treap 當中最重要的就是以下兩種操作，並且其中一種與 pri 有關。

- merge(a,b) : 將兩顆樹 a, b 合併。前提: a, b 滿足所有在 a 裡面的節點所存放的值皆小於所有在 b 裡面的節點
- split(a,b,p) : 將一棵樹分成 a, b 並同時滿足 a 中的值皆小於等於 ($\leq$) p ，而 b 中的值皆大於 ($>$) p

兩者詳細步驟是依照遞迴定義的，首先是 merge。

merge

- 如果 a,b 至少其中一個是空指標，回傳非空的那一個 (兩個都空救回傳空指標)
- 否則爲了同時維持 BST 和 heap 的性質，我們先判斷哪一個的 pri(priority) 值更大，我的實作是將 pri 值小的放上面 (較接近樹根的位置)
- 如果 a 的 pri 值較小，我們會回傳 a 作爲呼叫此函數的樹的根，不過在此之前，我們要先決定 a 的右子樹 (因爲 b 也還沒被合併完) ，所以要向下遞迴
- b 的情況剛好相反，因爲 b 裡面的值都大於 a 裡面的值

split

split 較爲複雜，因爲同時要滿足 Treap 的兩種性質。

- 若 T(要被分割的 Treap) 爲空指標的話，代表整棵樹都被分完了，因此將 a,b 都設爲空指標。
- 否則一樣要分成兩個條件: T 的根結點的值小於等於 p ，和大於 p
- 如果 T 的根結點的值小於等於 p ，選擇將 T 的根結點與他的左子樹給 a 並繼續向下分解 T 的右子樹給 a 的右子樹與 b
- 否則將 T 的根結點與他的右子樹給 b 並繼續向下分解 T 的左子樹給 b 的左子樹與 a

名次樹

首先介紹一下名次樹，名次樹是透過在節點內多存一個數值 → 子樹大小 (size)，以實現更多功能，例如: k 在這棵樹裡面排第幾，刪除比 k 大的第一個數，還有排名第 k 的樹爲何等。

爲了在 Treap 中實現這個功能，我們要多實作兩個函數 → pull & SplitBySize(splitSz)。

首先，我先將節點完整的 struct ，還有尋找某個指標底下的樹的 size 的函式寫出來，方便後續講解。順帶一題，爲了維持名次樹的正確性，每次只要樹有經過更動就要執行 pull 。

```
1 struct node{
2     // key 是排序依據
3     // pri 是 heap 的依據
4     int key,pri,sz;
5     node *lch,*rch;
6
7     node(int _key){
8         key=_key;
9         sz=1;
```

```
10          lch=rch=nullptr;
11          // RandomInt() 之後會實作
12          pri=RandomInt();
13      }
14
15      void pull(){
16          // 自己也算
17          sz=1;
18          // 判斷式相當於 lch!=nullptr
19          // 要加上左右子樹的大小才是完整的
20          if(lch) sz+=lch->sz;
21          if(rch) sz+=rch->sz;
22      }
23  };
24
25  int fs(node *a){
26      // 如果 a 爲空指標則 sz(回傳值) 爲 0
27      return a ? a->sz : 0;
28  }
```

Code 5.7: node of Treap

SplitBySize

這個函式會與 Split 有點像，差別在於分開的依據不同，SplitBySize 會將整棵樹分爲 a, b ，其中 a 存有前 s 小的所有數，b 則存剩下的。詳細步驟如下。

- 與 Split 相同，若 T 爲空則表示分完，將 a,b 都設爲空指標
- 否則如果 T 的左子樹的元素個數不到 s ，將 T 和他的左子樹都給 a ，然後繼續從 T 的右子樹當中切出 $s-(T$的左子樹大小$)-1(T$自己$)$ 個元素給 a 的右子樹
- 否則將 T 和他的右子樹都給 b ，然後從 T 的左子樹當中切出 s 個元素給 a

隨機數

C++ 本身就有內建隨機函式可供取用。就直接放在下面供大家參考。

```
1  // 簡單作法，但可能被 hack
2  #include<crand>
3
4  int RandomInt(){
5      return rand();
6  }
7
8  // 較長，但較不易被破解
9  #include<random>
10
11 unsigned seed=chrono::steady_clock().now().time_since_epoch().count();
12 mt19937 rng(seed);
13
14 int RandomInt(){
15     return rng();
16 }
```

Code 5.8: C++ 隨機數

最後附上程式碼。

```
node *merge(node *a,node *b){
    if(!a || !b) return a ? a : b;

    if(a->pri < b->pri){
        a->rch=merge(a->rch,b);
        a->pull();
        return a;
    }else{
        b->lch=merge(a,b->lch);
        b->pull();
        return b;
    }
}

void split(node *T,node *&a,node *&b,int p){
    if(!T){
        a=b=nullptr;
        return;
    }

    if(T->key <= p){
        a=T;
        split(T->rch,a->rch,b,p);
        a->pull();
    }else{
        b=T;
        split(T->lch,a,b->lch,p);
        a->pull();
    }
}

void splitSz(node *T,node *&a,node *&b,int s){
    if(!T){
        a=b=nullptr;
        return;
    }

    if(fs(T->lch)<s){
        a=T;
        splitSz(T->rch,a->rch,b,s-fs(T->lch)-1);
        a->pull();
    }else{
        b=T;
        splitSz(T->lch,a,b->lch,s);
        b->pull();
    }
}
```

Code 5.9: Treap 核心

5.5.3 基本功能

有了這些函式之後，後面的大多數操作都會變很簡單，舉凡插入 (insert)，刪除 (erase)，尋找 (find) 等等。因爲他們都可以用 merge and split 湊出來。而且很多功能甚至不只有一種實作方法。讓我們看下去。

Insert

insert 可以被簡單地分成幾個步驟。假設我要插入的數字爲 v 。

- 將整棵樹 (我都存在 rt 指標中) 切成小於等於 v 和大於 v 的兩棵樹 rt, b
- 另外用 v new 一個節點 a(v) 出來 (不會動態記憶體分配的可以先去複習一下)。
- 合併 rt 和 a ，並將合併後的樹給 rt
- 合併 rt 和 b ，並將合併後的樹給 rt

Erase

erase 一樣可以被簡單地分成幾個步驟。假設我要刪除的數字爲 v 。

1. 刪除所有等於 v 的數 (整數才可以)

- 將整棵樹 (我都存在 rt 指標中) 切成小於等於 v 和大於 v 的兩棵樹 rt, b
- 再將 rt 指向的樹切成小於等於 v-1 和大於 v-1 的兩棵樹 rt, a
- 刪除 a 整棵樹
- 合併 rt 和 b ，並將合併後的樹給 rt

2. 刪除一個等於 v 的數 (整數才可以)

- 將整棵樹 (我都存在 rt 指標中) 切成小於等於 v-1 和大於 v-1 的兩棵樹 rt, b
- 再將 b 指向的樹切成左邊一個元素，剩下都放右邊的 b, a
- 刪除 b
- 合併 rt 和 b ，並將合併後的樹給 rt

如果是浮點數可能要換方法，不過基本上不會用到。

Count

假設我要數的數字爲 v ，就是問 v 有幾個。

- 將整棵樹 (我都存在 rt 指標中) 切成小於等於 v 和大於 v 的兩棵樹 rt, b

- 再將 rt 指向的樹切成小於等於 v-1 和大於 v-1 的兩棵樹 rt, a
- 用一個變數將 a 的 size 存起來
- 合併 rt 和 a ，並將合併後的樹給 rt
- 合併 rt 和 b ，並將合併後的樹給 rt

Kth Small Element

假設我要找第 p 小的數字。

- 將整棵樹 (我都存在 rt 指標中) 切成左邊 p 個元素，剩下都放右邊的 rt, b
- 再將 rt 指向的樹切成左邊 p-1 個元素，剩下都放右邊的 rt, a
- 用一個變數將 a 的值存起來
- 合併 rt 和 a ，並將合併後的樹給 rt
- 合併 rt 和 b ，並將合併後的樹給 rt

最後附上程式碼。

```
void insert(int v){
    node *a=new node(v),*b;
    split(rt,rt,b,v);
    rt=merge(rt,a);
    rt=merge(rt,b);
}

// 這個函式是先刪除左右子樹，再刪除自己，並透過遞迴來刪掉整棵樹。
void delete_tree(node *n){
    if(n->lch) delete_tree(n->lch);
    if(n->rch) delete_tree(n->rch);
    delete(n);
}

// erase all element between (v-1~v] -> 對應 1
void erase(int v){
    node *a,*b;
    split(rt,rt,b,v);
    split(rt,rt,a,v-1);// or v-eps
    // it's better to use delete_tree function
    delete(a);
    rt=merge(rt,b);
}

// erase one element between (v-1~v] -> 對應 2
void erase(int v){
    node *a,*b;
    split(rt,rt,b,v-1);
    splitSz(b,b,a,1);
    delete(b);
```

```
31      rt=merge(rt,b);
32  }
33
34  int count(int v){
35      node *a,*b;
36      split(rt,rt,b,v);
37      split(rt,rt,a,v-1);// or v-eps
38      int ret=fs(a);
39      rt=merge(rt,a);
40      rt=merge(rt,b);
41      return ret;
42  }
43
44  int kth(int v){
45      node *a,*b;
46      splitSz(rt,rt,b,p);
47      splitSz(rt,rt,a,p-1);
48      int ret=a->key;
49      rt=merge(rt,a);
50      rt=merge(rt,b);
51      return ret;
52  }
```

Code 5.10: Treap 基本功能

說到這邊，你以爲只是平衡樹嗎？不不不，祂可厲害了。

5.5.4 進階功能

如果我們將整棵樹的中序遍歷當成序列的順序的話 (如下圖)，我們可以實現需多更強大的功能。可以說，線段樹能做到的事 Treap 都可以做到，但 Treap 甚至可以做到許多線段樹做不到的事。

而且，此時我們可以把 key 拔掉。然後只使用 splitSz 和 merge 。

Insert

假設我希望插入的數字位於第 p 個，且值爲 v 。

- 將整棵樹 (我都存在 rt 指標中) 切 p-1 個給 rt ，其他剩下的分給 b
- 另外用 v new 一個節點 a(v) 出來
- 合併 rt 和 a ，並將合併後的樹給 rt
- 合並 rt 和 b ，並將合併後的樹給 rt

Erase

假設我要刪除第 p 個數。

- 將整棵樹 (我都存在 rt 指標中) 切 p-1 個給 rt ，其他給 b

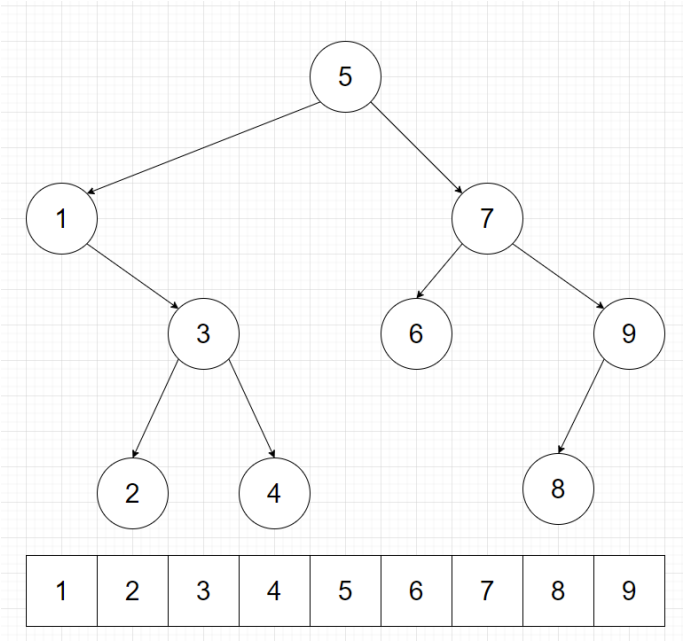


Figure 5.4: Treap 表示數列示意圖

- 再將 b 指向的樹切一個給 b 剩下給 a
- 刪除 a
- 合併 rt 和 b ，並將合併後的樹給 rt

區間操作

在介紹區間操作之前，我們還需要在節點上加入更多的資訊和函式，所以 node 裡面會有更多程式碼。(以下程式碼加上了很多功能，實際上不需要那麼多)

```
const int INF=0x3f3f3f3f

struct Treap{
    struct node{
        // 此節點的值
        int val;
        // 維持 Treap 的性質
        int pri,sz;
        // 用在區間查詢
        int sum,mx,mn;
        // 可以支援區間反轉
        bool rev_tag;
        // 可以支援區間加值
        int add_tag;
        // 左右子樹
        node *lch,*rch;

        node(int _val){
            val=sum=mx=mn=_val;
            lch=rch=nullptr;
            sz=1;
```

```
22              pri=rng();
23          }
24
25          void pull(){
26              sz=1;
27              sum=mx=mn=val;
28
29              if(lch){
30                  sz+=lch->sz;
31                  sum+=lch->sum;
32                  mx=max(mx,lch->mx);
33                  mn=min(mn,lch->mn);
34              }
35
36              if(rch){
37                  sz+=rch->sz;
38                  sum+=rch->sum;
39                  mx=max(mx,rch->mx);
40                  mn=min(mn,rch->mn);
41              }
42          }
43
44          // 讓他底下的區間都反轉
45          void rev(){
46              rev_tag=!rev_tag;
47          }
48
49          // 再更改節點前用
50          void push(){
51              if(rev_tag){
52                  swap(lch,rch);
53                  if(lch)
54                      lch->tag=true;
55                  if(rch)
56                      rch->tag=true;
57              }
58
59              if(lch)
60                  lch->add_tag+=add_tag;
61              if(rch)
62                  rch->add_tag+=add_tag;
63              add_tag=0;
64          }
65      };
66  };
```

Code 5.11: Treap 表示數列使用的 node

同時，我們在原來的 merge/split 操作程式碼中，都要加上 push ，具體位置如下。

```
1   struct Treap{
2       node *merge(node *a,node *b){
3           if(!a || !b) return a ? a : b;
4
5           if(a->pri < b->pri){
```

```
6               a->push();
7               a->rch=merge(a->rch,b);
8               return a;
9           }else{
10              b->push();
11              b->lch=merge(a,b->lch);
12              return b;
13          }
14      }
15
16      void splitSz(node *T,node *&a,node *&b,int s){
17          if(!T){
18              a=b=nullptr;
19              return;
20          }
21
22          T->push();
23
24          if(fs(T->lch)<s){
25              a=T;
26              splitSz(T->rch,a->rch,b,s-fs(T->lch)-1);
27              a->pull();
28          }else{
29              b=T;
30              splitSz(T->lch,a,b->lch,s);
31              b->pull();
32          }
33      }
34  };
```

Code 5.12: push 放的位置

接下來的操作都是把區間切下來，做完我要做的事，再接回去就可以了。

區間查詢 (極值、總和)

假設現在要查詢 $[l, r]$ 的區間。

- 從 rt 切下 r 個節點給 rt ，其它給 b
- 接著從 rt 切下 l-1 個節點給 rt ，其餘給 a
- 此時 a 就存有區間 $[l, r]$ ，而他的 sum, mx, mn 就會是這個區間的總和、最大值，以及最小值
- 最後先按照順序把它合併，再回傳答案即可

單點修改

先將所要修改的位置切下來，修改完後 merge 回去，在修改完後記得要 push 。

區間操作

假設現在要對 $[l, r]$ 的區間進行操作。

- 從 rt 切下 r 個節點給 rt ，其它給 b

- 接著從 rt 切下 l-1 個節點給 rt ，其餘給 a
- 此時 a 就存有區間 $[l, r]$ ，直接對他操作就可以了
- 最後先按照順序把它合併

當然，與線段樹相同，區間操作通常需要懶人標記。

最後附上程式碼。

```
int query(int l,int r){
    node *a,*b;
    splitSz(rt,rt,b,r);
    splitSz(rt,rt,a,l-1);
    // 下一行放 sum, mx, mn 看需求
    int ret=a->sum;
    rt=merge(rt,a);
    rt=merge(rt,b);
    return ret;
}

// 單點查詢就用查詢長度爲 1 的區間重復利用函式就可以了
// 不過會比較慢
int get(int p){
    return query(p,p);
}

void modify(int p,int v){
    node *a,*b;
    splitSz(rt,rt,a,p-1);
    splitSz(a,a,b,1);
    a->val=v;
    rt=merge(rt,a);
    rt=merge(rt,b);
}

// 區間反轉
void reverse(int l,int r){
    node *a,*b;
    splitSz(rt,rt,b,r);
    splitSz(rt,rt,a,l-1);
    if(a) a->rev();
    rt=merge(rt,a);
    rt=merge(rt,b);
}

// 區間加值
void add(int l,int r,int v){
    node *a,*b;
    splitSz(rt,rt,b,r);
    splitSz(rt,rt,a,l-1);
    a->val+=v;
    a->push();
    rt=merge(rt,a);
    rt=merge(rt,b);
}
```

Code 5.13: Treap 進階操作

5.5.5 解構式

對於多筆測資，可能會需要釋放記憶體。由於 Treap 二元樹的特性，可以透過遞迴刪除整棵樹。

```
void DeleteTree(node *n){
    if(n->lch) DeleteTree(n->lch);
    if(n->rch) DeleteTree(n->rch);
    delete(n);
}

~Treap(){
    if(rt) DeleteTree(rt);
}
```

Code 5.14: Treap 解構式

5.5.6 一些神奇的東西

因爲幾乎不會用到，只是好玩寫的，所以不解釋。

```
int size(){
    return fs(rt);
}

bool empty(){
    return size()==0;
}

void push_back(num v){
    insert(size()+1,v);
}

void push_front(num v){
    insert(1,v);
}

void pop_back(){
    erase(size()+1);
}

void pop_front(){
    erase(1);
}

void resize(int sz){
    while(size()<sz){
        push_back(0);
```

```
28      }
29      while(size()>sz){
30          pop_back();
31      }
32  }
33
34  void resize(int sz,num v){
35      while(size()<sz){
36          push_back(v);
37      }
38      while(size()>sz){
39          pop_back();
40      }
41  }
42
43  Treap(){}
44  Treap(int sz){
45      resize(sz);
46  }
47
48  Treap(int sz,int v){
49      resize(sz,v);
50  }
```

Code 5.15: 其他功能

5.5.7 持久化

正在研究 ...

5.5.8 小結

Treap 還有更多操作可以使用，我並未，也不可能一一列舉。你們只要記得 merge 和 split 可以創造出無限可能。

不過小心，Treap 會使用大量記憶體與時間。雖然同樣是 $O(\log(n))$。同級距下 Treap 會比較慢，當然，有一些用值域線段樹的問題，用 Treap 會跑的比較快，畢竟這次複雜度不相同了 $O(\log C) > O(\log n), C \leq 10^{12}, n \leq 10^5$。

5.5.9 範例與練習

Problem 5.5.2. 第一屆卓越盃 F.Safe Sequence(Hard)

題目敘述

你是一家銀行負責保管保險箱的保險箱管理處處長，由於最近銀行生意並不是非常好，因此銀行行長決定跟員工玩一個遊戲。以下爲遊戲會用到的操作。

1. 新增一個保險箱到保險箱序列第 x 個後面，且該保險箱此時已經有 y 個金條

2. 反轉 $[l,r]$ 的保險箱 註 1
3. 詢問區間 $[l,r]$ 的金條總數
4. 從目前保險箱序列中的第 x 個位置拿出 y 個金條
5. 在目前保險箱序列中的第 x 個位置放入 y 個金條

輸入說明

第一行輸入 1 個數字 t ，代表有 t 比測試資料
每筆測資第一行輸入 2 個數字 n,q ，表示一開始的保險箱有幾個
第二行輸入 n 個數字 $a_1,a_2...a_n$ ，a_i 代表第 i 個保險箱原來有幾塊金條
接下來有 q 行，分別對應一個操作 (以下說明)
對應題目敘述的第 k 項操作

1. 輸入 1 x y
2. 輸入 2 l r
3. 輸入 3 l r
4. 輸入 4 x y
5. 輸入 5 x y

$t \leq 100$，$n,q,a_i,x,y \leq 10^4$

輸出說明

僅有操作 3 需要輸出區間 $[l,r]$ 的金條總數

範例測試

範例輸入 1	範例輸出 1
1 4 2 1 2 3 4 2 1 4 3 1 3	10

註 1：由於銀行裝備有" 量子軌道位移系統"，因此無須擔心保險箱管理處員工因搬運過多保險箱而導致橫紋肌溶解，或是出現保險箱太重而搬不動的問題。

Problem 5.5.3. CF 702F T-Shirts

題目敘述

大批量的 T 恤在商店在春季之前上架。總共有 n 種 T 恤上架販售。第 i 種 T 恤有兩個整數參數—c_i 和 q_i，其中 c_i 表示第 i 種 T 恤的價格，q_i 表示第 i 種 T 恤的品質。假設商店中有無限數量的每種 T 恤可供販售，但是通常價格和品質並無關聯。

根據預測，在接下來的一個月內將有 k 名顧客光臨該商店，第 j 個顧客打算花費不超過 b_j 來購買 T 恤。

所有顧客都使用相同的策略。首先，顧客希望購買最多可能的最高品質的 T 恤，然後從剩餘的 T 恤中購買最多可能的最高品質的 T 恤，以此類推。同時，在多個品質相同的 T 恤中，顧客會選擇價格更低的那件。顧客不喜歡相同的 T 恤，因此每位顧客不會購買多於一件同款的 T 恤。

請根據所描述的策略，確定每位顧客將購買的 T 恤數量。所有顧客之間是獨立的，一位顧客的購買不會影響其他顧客的購買。

輸入說明

第一行包含一個正整數 n $(1 \leq n \leq 2 \times 10^5)$ —T 恤類型的數量。

接下來的 n 行，每行包含兩個整數 c_i 和 q_i $(1 \leq c_i, q_i \leq 10^9)$ —第 i 種 T 恤的價格和品質。

接下來一行包含一個正整數 k $(1 \leq k \leq 2 \times 10^5)$ —顧客的數量。

接下來一行包含 k 個正整數 $b_1, b_2, \cdots, b_k$ $(1 \leq b_j \leq 10^9)$，其中第 j 個數字表示第 j 位顧客打算花費的總金額。

輸出說明

輸出的第一行應包含一個長度爲 k 的整數序列，其中第 i 個數字應該等於第 i 位顧客將購買的 T 恤數量。

範例測試

範例輸入 1	範例輸出 1
3 7 5 3 5 4 3 2 13 14	2 3

Problem 5.5.4. TIOJ 1382 約瑟問題

題目敘述

還記得約瑟夫問題嗎!? 沒錯就是那位約瑟夫人。這次問題是約瑟問題，不是約瑟夫問題唷!!

有個古老的經典問題是這樣的：n 個人圍成圓圈，從頭開始每 k 個一數殺掉，最後問 Joseph 今天晚餐吃什麼。

很難對吧。今天問題簡單多了，而且是普遍級的，不殺人的

有 n 個人依照編號順去坐著圍成圓圈，每個人的椅子都被裝上“強制脫出裝置”，接著有張神秘的紙條上面寫著 n 個數，代表每一次要數幾個人之後彈出 (開始時從第一位開始數)

輸入說明

包含多組測試資料，請以 EOF 做爲結束 (測試資料不超過 10 組)

每筆測試資料的第一行爲一個數字 n 代表有多少人

第二行有 n 個數字 $a_1, a_2, \cdots, a_n$，a_i 代表第 i 次數到第 a_i 個人被彈出

$1 \leq n, a_i \leq 10^5$

輸出說明

輸出彈出的順序，編號是 1 n。

範例測試

範例輸入 1	範例輸出 1
5 2 3 2 3 1	2 5 3 4 1

參考資料

https://www.youtube.com/watch?v=CFRhGnuXG-4

https://www.luogu.com.cn/

https://codeforces.com/

https://tioj.ck.tp.edu.tw/problems

https://zerojudge.tw/Problems

https://leetcode.com/

https://cses.fi/book/book.pdf

https://judge.tcirc.tw/

https://zh.wikipedia.org

https://atcoder.jp/