概論、C++ 技巧以及簡單演算法技巧

hansonyu123

2016 年 9 月 7 日、12 日

1 接下來你會遇到的事

1.1 主要比賽們

日期	項目	地點	概述
9/6	校內能力初賽	建中	考完了 (?
10/4	校內能力複賽	建中	選出約 12 個校隊代表參加北市能力競賽。
11 月	北市賽	臺北某學校	選出 10 人參加全國賽。
12 月	全國賽	師大	選出 10 人進入 TOI 一階。
2 月	校隊補選	建中	選出遞補已進入 TOI 一階的校隊人選，與其他校隊一同參加 TOI 入營考。
3 月	TOI 入營考	師大	選出另外 20 人進入 TOI 一階。
3/4 月	TOI 一階	師大（住宿）	14 天中有兩次考試，在 30 人中選出 12 人進入 TOI 二階（並獲得參加 APIO 與推薦之資格）。
4 月	TOI 二階	師大（住宿）	同上，由 12 人選出 4 人代表臺灣參加 IOI。
5 月	APIO	師大	由進入 TOI 二階的人參加，沒什麼用處 (?
7/28	IOI 2017	伊朗德黑蘭	為國爭光拿獎牌。

1.2 次要比賽們

日期	項目	地點	概述
10/11 月	北市軟體競賽	臺北某學校	初賽筆試、決賽上機。
11/12 月	NPSC	臺大	同校三人組隊報名。有初賽、決賽，每校只有三隊可以進決賽。聽說獎品不錯。

1.3 各種 OJ 們

簡稱	網址	概述
TIOJ	tioj.infor.org	建中的 OJ，名字採用遞迴縮寫，相信大家比賽的時候都用過。聽說歷史悠久，有很多不錯的題目，但爛題也不少。
UVa	uva.onlinejudge.org	（英文）可說是史上第一個 OJ，1997 年就開放了，題目數量極多。可以反覆練習相關算法。
ACM-ICPC	icpcarchive.ecs.baylor.edu	（英文）某個大學程式設計競賽的題目們，據說教授喜歡從這裡找題目。
POI	main.edu.pl/en	（英文、波蘭文）波蘭資奧的網站，裡面有蠻多不錯的題目，只是這裡不能用 C++11。
ZJ	zerojudge.tw	在眾多水題中夾雜著幾題難題。feedback 非常好心，有移植一些 UVa 還有入營考的題目。考試前刷水題可以增加自信 (?
CF	codeforces.com	俄羅斯的程式設計競賽平臺，隔一段時間就有比賽（通常一週 1~2 次），但是通常會在臺灣時間半夜。如果作息調整得來的話可以參加比賽，順便爬積分（世界排名）。不然也可以當作一般的 OJ，而且看得到別人的 code，可以觀摩別人是怎麼寫程式的。

1.4 演算法線上資源

名稱	網址	概述
演算法筆記	www.csie.ntnu.edu.tw/~u91029	內容超多樣豐富的演算法專門網站。不盡然和競賽相關，也有一些些錯誤，但是由於各種經典問題幾乎都有介紹到，依然是個非常好用的參考網站。
cplusplus C++ reference	www.cplusplus.com/reference en.cppreference.com/w/cpp	想要對 C++ 內建程式庫等有完整的了解，這兩個都是值得參考的網站。C++ reference 比較完整，但稍微難懂一點。也可以搭配本講義後面的章節服用。
資訊之芽	www.csie.ntu.edu.tw/~sprout/algo2016	這是一個每年二到五月每週六上課的課程，但是它的講義、課堂 ppt、回家作業等東西都會放在網路上，也有其中幾年的教學影片，課講得還算不錯，可以聽聽。

1.5 演算法書本資源

名稱	ISBN	概述
Introduction to Algorithms	978-0-262-03384-8	大學演算法課程必備書籍，選訓也會發。偏重於嚴謹的論證，看這本書需要有一定的數學程度。有例題可以練習。
提升程式設計的資料結構力	978-986-276-679-8	這次上課的參考教材，但是內容有些參差不齊，應該只會偶爾提到。

2 什麼是複雜度

2.1 O-notation

在介紹複雜度之前，我們得先瞭解複雜度的記法：O-notation。如果我們說

$$f(x) = O(g(x)) \text{ as } x \to \infty$$

就代表存在常數 c 以及 x_0，使得 $|f(x)| \leq c|g(x)| \ \forall x \geq x_0$ 。用好懂一點的語言來講，就是 $g(x)$ 趨近無窮的速度不比 $f(x)$ 來得慢。

舉例來說，$x = O(x^2), 2x = O(123456x + 7), x^{213} = O(2^x)$。

我們可以注意到 O-notation 代表的是一個上界，所以在一般的演算法分析上，為了使其簡單易懂，通常會依循三個原則：用盡可能小的上界、忽略常數、忽略成長較慢的項。例如，雖然 $7x^2 + 15x$ 可以是 $O(x^3), O(x^2 + x), O(7x^2)$，但是我們通常會寫它是 $O(x^2)$。

2.2 時間複雜度

所謂的時間複雜度，就是在估計一個演算法運行所需時間的一個函數。通常，我們會以「進行基本運算的次數」來估計。以 C++ 為例，加減乘除、取餘數、位元運算、設值、邏輯運算、比大小等等的都算是一個「基本運算」。因為基本運算都可以在一個固定的時間內完成，所以「進行基本運算的次數」和「演算法所需時間」只有常數的差別。 然而實際計算進行運算的次數不太切實際，所以我們通常都只關心它的成長速度 — 也就是使用 O-notation。注意我們在 O-notation 中忽略了常數，是因為常數在通常情況差異不大，並不對運算速度有太重要的影響（儘管偶有例外）。

只關心它的成長速度也是有原因的。比如說算法 A 的時間複雜度為 $A(n) = 1000n^3 = O(n^3)$，算法 B 的時間複雜度為 $B(n) = n^4 = O(n^4)$。雖然在 n 很小（比如是 1）的時候，算法 A 比算法 B 還要慢得許多，但是當 n 逐漸變大時，算法 A 所需的時間開始愈來愈靠近算法 B，最後甚至比算法 B 還要來得少的許多。比如說在 $n = 10^6$ 時，算法 A 所需的時

間已經是算法 B 的千分之一了。而通常我們在意的是，在面對海量的輸入的時候，哪個演算法能有較好的表現，這時算法 A 就脫穎而出了。

再舉一個例子：

Algorithm 1: Insertion sort

```
function insertion_sort(array, length)
        for i = 1 to length(A) - 1
        x = A[i]
        j = i
        while j > 0 and A[j] > x
                A[j] = A[j - 1]
                j = j - 1
        end while
        A[j] = x
        end for
end function
```

這是典型插入排序（insertion sort）的演算法。如果此程序獲得的數列（長度為 n）一開始已是由小到大排序好的，那麼只會進行 $n-1$ 次的比較，因此時間複雜度為 $O(n)$。然而若獲得的數列一開始是由大排到小，那麼需要進行 $n(n-1)/2$ 次比較，時間複雜度為 $O(n^2)$。可見就算輸入大小相同，時間複雜度也不一定相同。然而我們在意的是在一定的時間內，演算法是否一定可以執行完畢，所以通常我們需要考慮的是最糟的情況（Worst case）。因此，我們將插入排序的（最糟）複雜度視為 $O(n^2)$。

知道如何估計時間複雜度後，要如何以此來估計一個演算法能不能在時限內執行完畢呢？如果你是 C 或 C++ 的使用者，你可以假設 1 秒可以進行 10^8 次運算。不過在常數很大或很小的時候，這樣的估計需要一點點修正，修正的依據就靠經驗啦。

2.3　空間複雜度

和時間複雜度類似，可用來估計一個演算法需要多少空間。只要知道基本型別佔的空間是多少，估計所需要的空間並不困難。除非題目有特別卡記憶體，不然空間通常不很重要。

2.4　題外話：均攤分析以及隨機演算法

前面的時間複雜度在一些場合下並不合適，取而代之的是其他定義的複雜度。

第一個例子是均攤複雜度。試想一個演算法，在通常的時候時間複雜度是 $O(1)$，不

過每執行 $O(n)$ 次，就恰有一次的時間複雜度是 $O(n)$。若使用前面的定義，這樣的演算法的複雜度應該是 $O(n)$，但若我們執行這個演算法夠多次，比如說執行了 $O(n)$ 次，那麼總複雜度也只有 $O(n)$。如果我們將所需的時間平均地分攤給每次執行，那麼每次執行所需要的時間便只有 $O(1)$。這就是均攤分析以及均攤複雜度的意義。這樣的演算法並不少見，比如說 vector 的 push_back 和並查集（disjoint set）都是。

第二個例子是平均複雜度。有些演算法在接受不同的輸入時會有不一樣的表現，而每次的表現好壞都會不同。比如說，最好可能只需要 $O(n)$，然而最糟需要 $O(n^2)$。這種情況，我們比較在意的是假設輸入隨機，所需時間的期望值是多少，也就是平均複雜度。算法就跟一般算期望值是一樣的。一個常見的例子是 treap 的插入，最糟情況下時間複雜度是 $O(n)$，然而平均複雜度只有 $O(\log n)$。如果使用好的 hash function，雜湊也會有類似的現象。這些東西未來有機會就會提到。

如果沒辦法一下子瞭解這節的東西也沒關係，未來遇到就會知道了。

3 好用的 C++ 內建資料結構

相信大家都是 C 或者 C++ 的使用者。

如果你是 C 的使用者，強烈建議你現在開始使用 C++。語法相容，但又有強力的 library，是在競賽時間壓力下不可或缺的強力工具。

如果你是 C++ 的使用者，強烈建議你把 C++ 內建又好用的東西摸熟。這樣就可以省去許多功夫，開心使用前人為你寫好的東西了。

3.1 什麼是資料結構

就是字面上的意思：拿來儲存資料的結構。

一個最簡單的例子就是陣列，它直接把一堆同樣型別的資料排成一排好好管理。但除了陣列之外，還有許多資料結構（有些甚至不是內建的）是需要瞭解的。以下就講些簡單常用的內建資料結構吧。

這些資料結構在「標準模板庫」(Standard Template Library，簡稱 STL）之中，是一個 C++ 的程式庫。注意，這個程式庫的所有東西都在 namespace std 底下。

3.2 型別模板（C++）

不過在繼續之前，有個重要的 C++ 觀念要講。

試想有一個作為容器用的型別 C，可以容納一些型別是給定的型別 T 的東西。那你在宣告這樣的一個變數時，理論上你應該告訴編譯器這個變數容納的型別是哪個型別。然而 C 語言中並沒有這樣的語法，所以在這樣的需求下模板（template）就在 C++ 誕生了。

型別模板的功用就是生出一堆新的型別。比方說剛剛的 C 好了，存放 int 型別的 C 和

存放 double 型別的 C，型別應該要是不同的。所以在宣告一個變數時，不是寫「C (變數名)」，而是寫「C<(int 或 double)> (變數名)」以指定這個變數的型別要是存放 int 的那種型別，還是存放 double 的那種型別。

有些型別模板的參數不只一個，寫好寫滿就對了（例：priority_queue<A, B, C> 就需要三個型別）。如果沒有寫好寫滿，會自動採用預設的型別（前提是要有預設的）。

想瞭解更多的話，關鍵字搜「模板」跟「C++」吧。

註：C++11 開始，模板的參數數是可變的。

3.3 迭代器（Iterator）

設想有個容器 C，裡面已經裝一些東西了。我該如何遍歷 C 中的所有元素呢？要知道 C 可能長得不像陣列，沒有「下標」這種東西。為了解決這個問題，C++ STL 為每個容器提供一個成員型別，叫做「迭代器」。

你可以把迭代器想像成是指標（事實上，指標也算一種迭代器）。如果你今天有一個迭代器 i，存取 i 指向的內容的方法，跟指標一樣，是在前面加個星號（*i）。而迭代器分成三種，取決於迭代器能進行的運算，由功能強到弱排序如下：

1. 隨機存取（Random Access）迭代器：這類的迭代器能夠和整數做加減法，加 s 代表從這項開始往後數 s 項，減 s 代表往前數 s 項。（當然，遞增、遞減運算也沒有問題。）你可以把指標當作這種迭代器。

2. 雙向（Bidirectional）迭代器：這類的迭代器只能做遞增（++）和遞減（--）運算，分別代表後一項和前一項。

3. 單向（Forward）迭代器：這類的迭代器只能做遞增（++）的運算，代表後一項。

而按照迭代器使用的方法，分成兩種：

1. 輸入（Input）迭代器：當你只有要讀取迭代器指向的內容時，這時迭代器當作輸入迭代器使用。所有的迭代器都可以當作輸入迭代器。

2. 輸出（Output）迭代器：當你要直接更改迭代器指向的內容時，這時迭代器當作輸出迭代器使用。除了常數（const）迭代器（也就是規定不能更動迭代器指向的內容）以外，所有的迭代器都可以當作輸出迭代器。

C++ 內建的迭代器都很和藹，只要是可以做的運算，複雜度都是 $O(1)$。

為了滿足人們的需求，C++ 內建的容器通常有兩種迭代器：正常的迭代器以及逆向迭代器。假如容器的型別是 C，因為迭代器是原本型別的成員型別，宣告時名稱分別是 C::iterator 和 C::reverse_iterator。前者會從前迭代到後，後者會從後迭代到前。另外，每個容器 c 的兩種迭代器各有兩個迭代器代表頭尾，分別是 c.begin()、c.end() 和 c.rbegin()、c.rend()。c.begin() 指向 c 的第一項，而 c.end() 指向 c 的最後一項的後一項。也就是說，*c.end()、*c.rend() 是不存在的，如果你這樣寫會造成不可預期的後果。

3.4 vector

vector 位於標頭檔 <vector> 裡。vector 可被視為是動態陣列的實現。通常的陣列，長度在宣告時就確定了，然而 vector 可以做到長度隨意伸縮。

列舉一下常用語法（假設變數名為 v）：

1.（建構式）vector<T> v(size_type a, const T& b)：一開始這個 v 會被 b 填滿，共填 a 個。如果只有指定 a，那麼 b 是 T 的預設值；如果什麼都沒指定，v 會是一個空的 vector。複雜度 $O(a)$。

2. v[i]：v 中的第 i 項，當平常陣列用就好。如果 i 的範圍不在 $[0, size)$，會發生無法預期的結果（undefined behavior）。複雜度 $O(1)$。

3. v.size()：這個函式會回傳 v 目前的長度。複雜度 $O(1)$。

4. v.push_back(T a)：在 v 的尾端加一個 a。均攤複雜度 $O(1)$。

5. v.pop_back()：刪除 v 的最末項。如果 v 是空的，會發生無法預期的結果。複雜度 $O(1)$。

6. v.empty()：回傳一個 bool，代表 v 是否是空的。複雜度 $O(1)$。

7. v.clear()：清空 v。複雜度 $O(size)$。原本 v 的空間會被保留，不會釋放掉。

8. v.resize(size_type a, const T& b)：強制將 v 的長度變為 a。如果比原本短，則將 v 原本的末段捨去，複雜度 $O(D(size - a))$，D 是解構 T 的時間。如果比原本長，在 v 的後面加 b 直到足夠為止（如果只有指定 a，那麼 b 是 T 的預設值），通常複雜度 $O(C(a - size))$，但如果需重新配置記憶體（reallocate），複雜度 $O(Ca)$，其中 C 是建構（複製）T 的時間。

9. v.reserve(size_type n)：預留放至少 n 個 T 的空間。如果需重新配置記憶體，複雜度 $O(size)$。如果 $n < size$，這個函數不造成任何影響。

vector 的迭代器屬於隨機存取迭代器。

比較需要講的是 vector 重新配置記憶體的耗時較長，所以如果能預先知道記憶體最多需要多少，就在一開始建構的時候開滿或者先 reserve 吧！（儘管對時間只有常數的影響）

另外，vector<bool> 有特化成一個 bool 佔的空間只有 1 bit，是 bool[] 的 1/8。

3.5 string

string 位於標頭檔 <string> 裡，等價於 basic_string<char>（如果不知道這個，可能會看不懂編譯訊息）。string 的用法很像 vector<char>，但因字串太常使用了，所以有經過一些優化。除此之外，還有一些好用的東西（假設變數名為 s）：

1. s = t：如果 t 是一個 string 或是 C 式字串，s 會變得跟 t 一樣。複雜度不明，但通常是 $O(size_s + size_t)$。
2. s += t：如果 t 是一個 string 或是 C 式字串，在 s 的尾端加上 t。複雜度通常是 $O(size_s + size_t)$。
3. s.c_str()：這個函式會回傳跟 s 一樣的 C 式字串。在 C++11 中保證複雜度為 $O(1)$。
4. s （比較大小或相等的符號）t：回傳比較 s 跟 t 字典序的結果。通常複雜度是 $O(\max(size_s, size_t))$。
5. cin >> s：輸入字串至 s，直到讀到空白字元。
6. cout << s：輸出字串 s。
7. getline(cin, s, char c)：輸入字串至 s，直到讀到字元 c。未指定時，c 是換行符號（`'\n'`）。

string 的迭代器屬於隨機存取迭代器。

除了列舉的之外，vector 有的 string 都有。除此之外，s.size() 有個同義的函式 s.length()，可能是怕人打錯才多加了這個函式吧。

順帶一提，關於 string 是不是「容器」其實也有些爭議，但它有大部分容器的性質，所以在此仍然將其歸類為容器。

3.6 deque

deque 位於標頭檔 <deque> 裡。deque 可以視為可以在最前面加東西、刪東西的 vector，除此之外它就是 vector 了。

假設變數名是 d，想要移除第一項，就用 d.pop_front()。想要在前面加一個東西 a，就用 d.push_front(a)。這兩個函式不會使迭代器失效，但會改變 deque 的下標。

雖說功能比 vector 強，但代價是時間和空間幾乎翻倍，所以沒事別用 deque。

3.7 list

list 位於標頭檔 <list> 裡。list 是個「雙向鏈結（doubly linked）結構」，也就是說對於 list 中的每一項，都可以 $O(1)$ 知道它的前一項和後一項。如此做的好處是，如果我要一次性加入一堆東西，只需要 $O(1)$ 的代價，比 vector 優。然而代價是，存取第 i 項的複雜度是 $O(i)$，因此沒有內建的下標運算。同樣列舉一下常用語法（假設變數名為 s）：

1. （建構式）list<T> s(size_type a, const T& b)：同 vector。
2. s.push_front(T a)、s.push_back(T a)、s.pop_front()、s.pop_back()：同 deque。
3. s.size()：回傳 s 中有幾項。相當需要注意的是 C++98 中這個函式的複雜度只有保證 $O(size)$，C++11 則保證 $O(1)$。

4. s.empty()：回傳一個 bool，代表 v 是否是空的。複雜度 $O(1)$。

5. s.insert(iterator p, T a)：在 p 指的那一項前面插入一個 a 並回傳一個指向 a 的迭代器。複雜度 $O(1)$。

6. s.insert(iterator p, size_type n, T a)：在 p 指的那一項前面插入 n 個 a。複雜度 $O(n)$。

7. s.erase(iterator p)：把 p 指的那項刪掉並回傳指向之後那項的迭代器。注意刪完之後 p 就失效了。複雜度 $O(1)$。

8. s.erase(iterator first, iterator last)：把 [first,last) 指到的東西全砍光光，回傳 last。複雜度和砍掉的東西個數呈線性關係。

9. s.splice(iterator p, list& x, iterator first, iterator last)：first 和 last 是 x 的迭代器。這個函式會把 [first,last) 指到的東西從 x 中移除並加到 p 指的那項前面。注意到 x 會因為這個函式而改變。如果沒有指定 last，那只將 first 從 x 刪去並加入 s。如果 first 和 last 都沒指定，那會將 x 中所有東西移到 s 中使 x 變為空的。複雜度是轉移元素個數的線性。

list 的迭代器屬於雙向迭代器。

可以看出 list 最大的功能是可以用 $O(1)$ 的代價進行一些別的容器做不到的事（insert、erase）。然而 list 最大的問題是所佔空間過於龐大，而且實用性低。C++11 中多了一個 forward_list 以改善空間過大的問題，代價是迭代器變成單向迭代器（也因此沒有 reverse_iterator）。

3.8 Container adaptor

它的翻譯好像叫什麼「適配器」的，滿難聽的⋯⋯ 這個東西主要是接收一個容器（前面講的那些），然後取其精華 (?) 改造它，成為新的容器。因此，Container adaptor 本身也是型別模板。

不過在取其精華的過程中，捨棄了一些東西。注意適配器通常不會有迭代器。

接下來介紹三個常見的 container adaptor。

3.9 stack

stack 位於標頭檔 <stack> 裡。可以把它想像成一疊書，每次可以放一本書在最上面，也可以從最上面拿一本書走。簡單來說就是秉持著「後進先出」（LIFO）的精神。

stack 這個模板需要的型別參數有兩個：T 和 C，其中 T 是內容物的型別，而 C 是採用的容器。為了方便改造，stack 對 C 有些要求：要有 empty、size、back、push_back、pop_back 這些函式，而在內建的容器中能夠勝任這角色的有 vector、deque 和 list。

stack 常用的語法列舉如下（假設變數名為 s）：

1.（建構式）stack<T, C> s(C& a)：s 一開始會有一份 a 的複製品。如果沒有指定 C 的話，C 是 deque<T>。如果沒有指定 a 的話，s 一開始會是空的。複雜度 $O(size)$。

2. s.size()、s.empty()：同 vector。

3. s.top()：存取最後一個進入 s 的元素，即「一疊書中最上面的那一本」。複雜度 $O(1)$。

4. s.push()：將一個元素加入 s 中。複雜度 $O(1)$。

5. s.pop()：將最後一個進入 s 的元素移除。複雜度 $O(1)$。

至於 C 應該要選用什麼，個人建議是 vector<T>。而且事實上，stack 能做的事 vector 都能做到，所以平常用 vector 就可以了。只是用 stack 可以增加程式的可讀性。

3.10 queue

queue 位於標頭檔 <queue> 裡。可以把它想像成排隊等著結帳的人群，要嘛有新的人來排在隊伍的尾端，要嘛最前面有一個人結完帳要走了。簡單來說，就是秉持著「先進先出」（FIFO）的精神。

queue 這個模板同樣需要兩個型別參數 T 和 C，跟 stack 一樣。不過不同的是，queue 對 C 的要求不太一樣：要有 empty、size、front、back、push_back、pop_front 這些函式，而在內建的容器中能夠勝任這角色的只有 deque 和 list。

queue 常用的語法列舉如下（假設變數名為 q）：

1.（建構式）queue<T, C> q(C& a)：q 一開始會有一份 a 的複製品。如果沒有指定 C 的話，C 是 deque<T>。如果沒有指定 a 的話，q 一開始會是空的。複雜度 $O(size)$。

2. q.size()，q.empty()：同 vector。

3. q.front()：存取第一個進入 sq 的元素，即「隊伍中最前面的人」。複雜度 $O(1)$。

4. q.back()：存取最後一個進入 q 的元素，即「隊伍最末端」。複雜度 $O(1)$。

4. q.push()：將一個元素加入 q 中。複雜度 $O(1)$。

5. q.pop()：將第一個進入 q 的元素移除。複雜度 $O(1)$。

建議 C 就依照預設的即可。不過和 stack 一樣，要用 queue 不如用 deque。

3.11 priority_queue

<queue> 中其實還藏有一威力極大的適配器：priority_queue。priority_queue 利用幾個內建函式實現「二叉堆」（binary heap）結構，一個在任何時候維持最頂的元素永遠都是最大的資料結構。priority_queue 雖然實作容易，但應用廣泛，每次都手刻一次會很浪費時

間。以後會見到更多 priority_queue 的應用。

priority_queue 這個模板需要三個型別參數 T、Con 和 Cmp。T 代表內容物的型別（需可以比較大小），Con 代表使用的容器，而 Cmp 代表使用的比大小的依據（之後會再詳細說明）。Con 的要求是擁有隨機存取迭代器以及 empty、size、front、push_back、pop_back 這些函式，而滿足這些條件的內建函式有 vector（預設值）和 deque。在詳細地認識 Cmp 之前，只需要知道 Cmp 是 less<T>（預設值）時 priority_queue 是最大堆，而是 greater<T> 的時候 priority_queue 是最小堆。

priority_queue 常用的語法列舉如下（假設變數名為 pq）：

1.（建構式）priority_queue<T, Con, Cmp> pq：建構一個空的 pq。複雜度 $O(1)$。

2.（建構式）priority_queue<T, Con, Cmp> pq(iterator first, iterator last)：建構一個 pq，內含 [first,last) 指到的東西，這裡 iterator 可以是任何迭代器。複雜度 $O(size)$。

3. pq.size()、pq.empty()：同 vector。

4. pq.top()：回傳 pq 中最大（最小）的元素（無法修改）。複雜度 $O(1)$。

5. pq.push(T a)：將 a 加入 pq 中。複雜度 $O(\log size)$。

6. pq.pop()：將 pq 中最大（最小）的元素移除。複雜度 $O(\log size)$。

比較需要注意的是在建構的時候直接餵內容物的時間複雜度是 $O(size)$，而一個一個 push 進去的時間複雜度是 $O(size \log size)$。雖然一般情況下沒什麼差，不過有必要的時候請記得 priority_queue 的建構式可以減少複雜度。

3.12 pair

pair 位於標頭檔 <utility> 裡面（注意沒有 <pair> 這個標頭檔）。pair 其實很單純，就是把兩個（可能不同型別的）變數綁在一起，變成一個變數。為此，pair 需要接收兩個型別，分別代表一對的第一項和第二項的型別。

pair 常用的語法列舉如下（假設變數名為 p）：

1.（建構式）pair<A, B> p(A a, B b)：建構一個把型別 A 和型別 B 綁在一起的 p，其中第一項是 a，第二項是 b。

2. p = s：如果 s 也是同型別的 pair，把 p 變得跟 s 一樣。

3. p（比較大小或相等的符號）s：如果 s 也是同型別的 pair，先比第一項，如果一樣再比第二項。

4. p.first、p.second：存取第一項、第二項。

在使用 pair 時，常常會使用到一個非成員函式 make_pair。make_pair 的好處在於，你不用特別指明第一項和第二項的型別，編譯器會自行幫你解析。用法是 make_pair(A a，B b)，函式會回傳一個 pair<A, B>，其中第一項是 a，第二項是 b。

另外，C++11 開始允許可變長度的模板，所以也有 pair 的推廣版 tuple（在標頭檔 <tuple> 中）。有興趣的可以自己看看。

3.13 set

set 位於標頭檔 <set> 裡。set 實現了自平衡二元查找樹，用白話文來講，可以 $O(\log n)$ 插入、刪除或查詢一個值有沒有在其中。特別的是，裡面的元素不會重複，因此我們會把元素的值稱為鍵值（key）。

set 的常用語法列舉如下（假設變數名為 s）：

1.（建構式）set<K> s：建構一個空的 s。複雜度 $O(1)$。

2. s.size()、s.empty()：同 vector。

3. s.insert(K k)：在 s 中放入一個鍵值為 k 的元素。如果本來就有了，什麼事都不會做。複雜度 $O(\log size)$。

4. s.erase(iterator first, iterator last)：刪除 [first,last)。如果沒指定 last，只刪除 first。只刪除一個時均攤複雜度 $O(1)$，刪除多個時複雜度是刪除個數的線性。

5. s.erase(K k)：刪除所有鍵值為 k 的元素並回傳刪除的項數（在 set 中只會是 0 或 1）。複雜度 $O(\log size)$。

6. s.find(K k)：回傳指向鍵值為 k 的元素的迭代器。如果沒有這種東西，回傳 m.end()。複雜度 $O(\log size)$。

7. s.count(K k)：回傳有幾個鍵值為 k 的元素（在 set 中只會是 0 或 1）。複雜度 $O(\log size)$。

8. s.lower_bound(K k)：回傳迭代器指向第一個鍵值大於等於 k 的項。複雜度 $O(\log size)$。

9. s.upper_bound(K k)：回傳迭代器指向第一個鍵值大於 k 的項。複雜度 $O(\log size)$。

set 的迭代器是雙向迭代器。set::iterator 會由小迭代到大，set::reverse_iterator 則會由大迭代到小。比較容易被忽視的是 set 的迭代器在遞增和遞減的時候，理論上不只是均攤複雜度，連複雜度也是 $O(1)$。但有些實作只保證均攤複雜度。

要注意的是 lower_bound 和 upper_bound 的微妙差別：一個是大於等於、一個是大於。用途通常是找鍵值在 [l,u) 的那些項，找法是 [s.lower_bound(l), s.upper_bound(u))。記住，C++ 通常是左閉右開區間。

3.14 map

map 位於標頭檔 <map> 裡。map 可以當成 set 的每一個元素都對應到另一個值，也就是可以用 $O(\log n)$ 插入、刪除或尋找一個鍵值對應的值。因此，map 這個模板需要兩個型別參數 K 和 T，其中 K 是鍵值的型別（需要可以比大小），而 T 代表對應到的值的型別。

另外，map 中的每一個元素其實是 pair<K, T>，所以迭代器指向的東西是一個 pair，第一項是鍵值，第二項是對應的值。

map 常用的語法列舉如下（假設變數名為 m）：

1.（建構式）map<K, T> m：建構一個空的 m。複雜度 $O(1)$。

2. m.size()、m.empty()、m.erase(iterator first, iterator last)、m.erase(K k)、m.find(K k)、m.count(K k)、m.lower_bound(K k)、m.upper_bound(K k)：同 set。

3. m[k]：存取鍵值 k 對應的值。如果 k 沒有對應的值，會插入一個元素，使 k 對應到預設值並回傳之。複雜度 $O(\log size)$。

4. m.insert(pair<K, T> k)：如果沒有鍵值為 k.first 的值，插入一個鍵值為 k.first 的值對應到 k.second，並回傳一個 pair，first 是指向剛插入的元素的迭代器、second 是 true；如果已經有了，回傳一個 pair，first 是指向鍵值為 k.first 的元素的迭代器，second 是 false。複雜度 $O(\log size)$。

和 set 的迭代器一樣，map 的迭代器是雙向迭代器。

3.15 multiset, multimap

在 <set>、<map> 中分別還有 multiset 和 multimap。和前面大致相同，唯一的差別在於 multiset 和 multimap 中鍵值可以重複出現，不像 set 和 map 鍵值不能一樣。如此一來，count 和 erase 回傳的值便不一定是 0 或 1。此外，由於 multimap 中一個鍵值可能對應到許多不同的值，因此也不支援下標操作。

在 multiset 和 multimap 中有一個特別好用的函式 equal_range(K k)，會回傳一個 iterator 的 pair，第一項代表 lower_bound(k)，第二項代表 upper_bound(k)。這兩項迭代器之間的項就是那些鍵值是 k 的項。雖然這個函式 set 跟 map 也有，但在 set 和 map 中就顯得有點雞肋。

3.16 (C++11)unordered_(multi)set, unordered_(multi)map

在 C++11 以後，unordered 系列常常擔任優化掉 map 和 set $O(\log n)$ 複雜度的角色。unordered_(multi)set 和 unordered_(multi)map 分別在標頭檔 <unordered_set> 和 <unordered_map> 裡。這四個模板需要的前一（二）個型別參數和 set（map）一樣，而接著是一個型別

Hash，代表要使用的雜湊函數的函數型別（之後會提）或指標。不過 C++11 有預設的內建型別（含任意型別的指標）的雜湊函數，所以除非情況特殊，你可以不用理會這一項。
你可以把 unordered 系列當成 (multi)map 和 (multi)set 來用。不過有幾點不同：

1. unordered 系列的迭代器為單向迭代器。

2. unordered 系列沒有將所有項依鍵值排序（這也是它名字的由來），因此迭代器在遍歷容器時不會依鍵值的大小順序遍歷。

3. 因為沒有排序，所以理所當然的沒有 lower_bound、upper_bound。

4. 比起 (multi)map 和 (multi)set，unordered 系列的期望複雜度少一個 log。

在宣告變數（建構式）時，你可以指定 bucket 至少有幾個。如果鍵值是內建型別而且你沒有指定 bucket 至少有幾個，那麼它會很耐斯地幫你搞定。

3.17 bitset

bitset 位於標頭檔 <bitset> 裡。你可以把它想像成有一堆 0 跟 1 的一個陣列，也就是說 bitset 的大小是固定的。但是它每一項只會用到 1 bit 的空間，而 bitset 的位元運算是被優化過的（雖然只限於同大小的 bitset），運作起來非常快速，對常數有極巨大的影響，同時也可以達到壓縮空間的效用，是常數優化的好幫手。

bitset 這個模板需要一個整數 n 作為模板的參數，代表 bitset 的長度。

bitset 的一些常用語法列舉如下（假設變數名為 b）：

1.（建構式）bitset<N> b(a)：用 a 初始化一個長度為 N 的 bitset。這裡 a 可以是 unsigned long、string 或 C 式字串。如果沒有指定 a，或者如果 b 有一些地方沒被 a 初始化，那些地方預設為 0。

2. b.count()：回傳 b 有幾個位元是 1。複雜度 $O(N)$。

3. b.size()：回傳 b 有幾個位元。複雜度 $O(1)$。

4. b （位元運算）：不管是一元還是二元的位元運算都可以。如果是兩個 bitset 的二元位元運算，兩個 bitset 的長度需一致。複雜度 $O(N)$。

5. b[a]：存取第 a 位。複雜度 $O(1)$。

6. b.set()：將所有位元設為 1。複雜度 $O(N)$。

7. b.reset()：將所有位元設為 0。複雜度 $O(N)$。

8. b.flip()：將所有位元的 0、1 互換（反白）。複雜度 $O(N)$。

9. b.to_string()：回傳一個字串和 b 的內容一樣。複雜度 $O(N)$。

10. b.to_ulong()：回傳一個 unsigned long 和 b 的內容一樣（在沒有溢位的範圍內）。複雜度 $O(N)$。

bitset 不是容器，而且它也沒有迭代器。

通常而言，如果要估計常數的話，相較於直接使用陣列，空間是 1/8、count 約是 1/6、位元運算約是 1/30。當然這些都不是絕對的。

要注意的是，上述的複雜度沒有明文規定，不過通常是如此。

3.18 習題

上面那些充其量只是「字典」。若要完全瞭解如何使用，最簡單的方法就是直接操練啦。以下都會提示可能會用到的東西，但不代表不用那些東西就解不出來。

1. (ZJ b298) 有 $N \le 10^4$ 家廠商，其中有 $M \le 10^6$ 組廠商關係 a, b，代表 a 是 b 的上游廠商。這 N 個廠商中有 L 個是有問題的，而有問題的廠商的下游廠商也會有問題。接著有 $Q \le 10^4$ 個詢問，回答指定的廠商是否有問題。（vector）

2. (ZJ c123)(UVa 514) 有一個 1 到 $N \le 1000$ 的排序（單筆可到 10^6），請問你可不可以透過一個暫存用的 stack 使得你從 stack 取出元素的順序剛好是 1 到 N。（stack）

3. (ZJ a813) 有 $N \le 10^6$ 棟房子由左至右排成一列。第 i 棟房子的高度是 H_i。如果 $a < b$ 而且第 a 棟房子和第 b 棟房子之間沒有其它房子高度超過 H_a 或 H_b，那麼從第 a 棟房子可以看得到第 b 棟房子，從第 b 棟房子也可以看到第 a 棟房子。如果從第 i 棟房子能看見的房子數為 C_i，求 $C_1 + C_2 + \cdots + C_N$。（stack）

4. (ZJ d424)(UVa 105)(TIOJ 1202) 給你不超過 5000（單筆可以到 10^5）個矩形，每個都有一邊貼齊 x 軸且都在 $+y$ 的部分。請求出它們的輪廓。（priority_queue）

5. (ZJ b231)(2009 入營考 pC) 有 $N \le 1000$ 本書（事實上 $N \le 10^5$ 都可以），每本書都有所需要的印刷時間和裝訂時間。你可以同時裝訂任意多本書，但同一時間只能印刷至多一本書。每本書需要先印完再裝訂。請問你至少要花多久才能將所有書都印刷裝訂完畢。（priority_queue）

6. (TIOJ 1807) 給你一張圖，有 $m \le 10^3$ 個點和 $n \le 10^9$ 條邊，每個點的編號在 1 到 n（別懷疑，你沒看錯）之間。請判斷它是不是簡單圖。（map, set）

7. (ZJ b291) 輸入的第一行有一個正整數 $N \le 1000$。接下來 N 行的每一行會出現一個字串（字串只包含小寫英文且長度不超過 20）代表動物種類，一個數字 $M \le 100$ 代表動物個數，緊接著是一個字串代表動物出現的地方。請輸出若干行，每一行列出一個地方出現的所有動物及個數（地點依第一次出現的順序排序，動物也依第一次出現的順序排序）。（vector, map, string）

4 <algorithm>

C++ 很貼心地給了一個標頭檔 <algorithm>，裡面充滿了各種函式。然而這些函式並不知道它們接收到的東西長得怎樣，所以這個時候迭代器就發揮功勞了。

<algorithm> 的函式往往都是以迭代器做為參數，因此這些函式不太需要關於接收到的東西的資訊（例如大小等，因為這些東西通常都蘊含在迭代器中了）。不過，有些函式還需要接收一個「函數」作為參數。你可以直接餵它函數指標，也可以選擇餵它一個「函數物件」。

4.1 函數物件

相信大家都知道 int 可以做加減乘除、遞增遞減、模、位元運算、比較大小等等的運算；一個指標則可以加減、可以用星號 dereference；一個迭代器則可以 dereference，可以遞增，可能可以遞減，也可能可以做加減法運算；一個 vector 或 deque 則有下標運算 “[]”。這些我們統稱為「運算子」（operator）。而在眾多的運算子中，還有一種 “()”。由於它寫起來就像是函數，因此提供 “()” 這種運算子的物件我們姑且稱作為「函數物件」，使用起來就真的像函數一樣。

而函數物件的型別就稱作「函數型別」。

4.2 less, greater

前面一直提到的 less 和 greater 事實上是函數型別的模板。它們都位於標頭檔 <functional> 裡。對於一個有小於（<）運算子的型別 T，less<T>(T a, T b) 會回傳一個 bool 等價於 a<b。而 greater 則是使用大於運算子。可以看出，less 和 greater 只是將運算子包裝成函數物件的模板，而其它運算子對應的函數型別模板也全都在標頭檔 <functional> 中。然而 <functional> 中最常用的就是這兩個了。

通常不太需要為了 less 和 greater 多寫一行引入，因為 <algorithm>、<queue>、<set>、<map> 等等，其實都有引入 <functional>。

4.3 好用的簡單函式

介紹完函數物件後就進入這個章節的主題吧。

<algorithm> 中的一些函式其實還滿常見的，像是 swap、min、max、random_shuffle（打亂一個範圍的元素）、iter_swap（交換兩迭代器指向的物件）等等的。相信大家都很熟悉所以就不再多說了。

4.4 簡化迴圈用的函式

C++ 作者在他出的書裡提到，迴圈是低階程式在用的，高階程式應避免使用迴圈。若使用了，也要儘量少。為了貫徹他的理想，C++ 實作了許多將迴圈包裝起來的函式，常用的列舉如下：for_each, find, find_if, find_end, count, count_if, search, search_n, copy, copy_backward, replace, replace_if, fill, fill_n, remove, remove_if, unique, reverse, rotate, partition, stable_partition, min_element, max_element, lexicographical_compare。

如果跟上面那整坨東西搞好關係的話，你的程式碼會變得很短。如果你會 C++11 以後的 lambda function 語法（inline 生成函數物件）的話，你的程式碼會變得超級無敵短。但我們現在寫的程式是競賽用途，不求高階不求美觀，所以如果你不會這些將迴圈包裝起來的函式其實也無傷大雅。

4.5 sort, nth_element

常常遇到要你寫 sort 的題目覺得很煩嗎？insertion sort 跟 quicksort 都會 TLE 嗎？寫 merge sort、heapsort 寫到心煩了嗎？不用擔心，C++ 自己根本就有內建的 sort 和 nth_element，而且是採用效率極高的 introsort 和 introselect。

先簡單介紹一下 introselect，是一個找到序列中第 k 小的元素的方法。在 introselect 之前，人們都是使用 quickselect，其想法是隨便戳一項（稱之為 pivot），將比它小的元素放在左邊，比它大的元素放在右邊，再看看你要的東西是在左邊還右邊，繼續遞迴下去。這樣當 pivot 剛好在正中間時複雜度會是好好的 $O(n)$（這也是期望複雜度），然而運氣差一點可能會退化為 $O(n^2)$。而 introselect 即為結合 quickselect 常數小但複雜度不保證，以及 median of medians（中位數的中位數，有興趣可以 wiki）保證複雜度但常數較大的特性而形成的排序法。想法是一開始先 quickselect，如果發現遞迴太深（代表戳不中比較中間的項）的話再改用 median of medians。而 nth_element 採用的就是 introselect，複雜度保證 $O(n)$。Median of medians 蠻難寫的，所以請記得有這個函式可以用。

Introsort 也是同樣的道理。一開始先使用 quicksort（隨便戳一項之後將比它小的元素放在左邊，比它大的元素放在右邊再把左右遞迴排序好），如果發現遞迴太深就改用 heapsort（可以把它想像成把一堆東西塞進 priority_queue 後再一個一個拔出來），而如果東西很少的話就直接使用 insertion sort。語法如下：

1. sort(iter first, iter last, Cmp cmp)：將 [first,last) 依照 cmp 排序，使得若 a 嚴格地在 b 前面，則 cmp(a,b) 為真。這裡要求 iter 是隨機存取迭代器，而 cmp 是一個接受兩個參數，回傳一個 bool 的函數指標或函數物件。不指定 cmp 時會由小排到大。複雜度 $O(Cn\log n)$，其中 C 是 cmp 的複雜度。

2. stable_sort：跟 sort 一樣，然而保證如果有兩項 a 跟 b 的值一樣且 a 一開始在 b 前面，那麼最後 a 也會在 b 前面。一般來說，有額外空間的話時間複雜度 $O(n \log n)$、額外空間複雜度 $O(n)$。但是如果沒辦法找到額外的空間，時間複雜度會變成 $O(n \log^2 n)$。（少數實作會提供不管有沒有額外空間都可以達到 $O(n \log n)$ 複雜度的演算法，稱為 block sort 或 wikisort。）

3. nth_element(iter first, iter nth, iter last, Cmp cmp)：將排序後應該會在第 nth 位置的元素 x 移到第 nth 個位置，並且讓應該在 x 前面的所有元素都在 x 前面，反之亦同。參數要求同 sort。平均複雜度 $O(n)$，但未保證最差複雜度。在多數實作當中，最差複雜度為 $O(n \log n)$。

4.6 專對付已排序序列的函式

有些函式是專門處理已排序序列的函式。如果你把它用在未排序的數列，你會得到一個無法預期的回傳值。常見的列舉如下：

1. lower_bound(iter first, iter last, T t, Cmp cmp)：找到 [first, last) 中第一項 a，使得 cmp(a, t) 不為真，並回傳指向 a 的迭代器。要求 [first, last) 經 Cmp 排序過。若未指定 Cmp，則假設 [first, last) 由小排至大。簡單來說，跟 map 的 lower_bound 是一樣的意思。若 iter 是隨機存取迭代器，複雜度 $O(\log n)$。若不然，複雜度 $O(n)$。如果 iter 是 set 和 map 系列的迭代器，請直接採用 set 和 map 的成員函式。

2. upper_bound(iter first, iter last, T t, Cmp cmp)：找到 [first, last) 中第一項 a，使得 cmp(t, a) 為真，並回傳指向 a 的迭代器。其餘同 lower_bound。

3. equal_range(iter first, iter last, T t, Cmp cmp)：回傳一個 pair，第一項是 lower_bound，第二項是 upper_bound。其餘同 lower_bound。

4. merge(iter1 first1, iter1 last1, iter2 first2, iter2 last2, iter3 res, Cmp cmp)：假設 [first1, last1) 和 [first2, last2) 是兩個經 cmp 排序好的序列，將兩個序列合併並排序，輸出至以 res 為首的序列並回傳指向序列末端的迭代器。iter1、iter2 可以是任何迭代器，iter3 可以是任何輸出迭代器。記得讓 res 後面有足夠的空間。複雜度線性。

5. set_union(iter1 first1, iter1 last1, iter2 first2, iter2 last2, iter3 res, Cmp cmp)：假設 A = [first1, last1) 和 B = [first2, last2) 是兩個經 cmp 排序好的序列，將兩個序列取聯集排序輸出至以 res 為首的序列並回傳指向序列末端的迭代器。其餘同 merge。

6. set_intersection：同 set_union，但此函式取的是交集。

7. set_difference：同 set_union，但此函式取的是餘集（$A - B$）。

8. set_symmetric_difference：同 set_union，但此函式取的是對稱餘集（$A \cup B - A \cap B$）。

4.7 (next/prev)_permutation

最後來介紹 next_permutation 和 prev_permutation 這兩個函式。

相信大家都知道什麼是字典序：從第一項開始逐項比較大小，直到分出大小為止。而這兩個函式即是找出比目前的字典序還要大一點（小一點）的排列。如果找不到更大（更小）的了，會將原本的序列由小到大（由大到小）排好。雖然一次的複雜度最高是線性，但如果執行了 $n!$ 次，均攤複雜度只有常數，是個不錯用的函式。語法如下：

1. next_permutation(iter first, iter last, Cmp cmp)：將 [first, last) 變成原本的某個排序，使得字典序是比原本大的所有排序中最小的。如果成功了，這個函式會回傳 true；否則回傳 false，並將 [first, last) 變成字典序最小的那個排序。cmp 是拿來比較兩元素大小的依據，如果不指定的話會用小於運算子。iter 至少需要是雙向迭代器。

2. prev_permutation(iter first, iter last, Cmp cmp)：將 [first, last) 變成原本的某個排序，使得字典序是比原本小的所有排序中最大的。如果成功了，這個函式會回傳 true；否則回傳 false，並將 [first, last) 變成字典序最大的那個排序。其餘同 next_permutation。

4.8 習題

同樣地，會有可能需要用到什麼東西的提示，但是不代表說要全用喔。

1. (ZJ a233) 裸排序題 BJ4

2. (TIOJ 1807) 給你一張圖，有 $m \le 10^3$ 個點和 $n \le 10^9$ 條邊，每個點的編號在 1 到 n 之間。請判斷它是不是簡單圖。(vector, sort)

3. (TIOJ 1617)(IOI 2000)(Interactive) 有 $n \le 1499$ 個數（奇數個），每次詢問你可以問某三個數的中位數是誰，而你最多可以做 7777 次詢問。請找出全部的中位數。(nth_element)

4. (ZJ d242)(UVa 481) 給你一個序列（長度單筆最大可到 10^5），找最長的嚴格遞增子序列。(vector, upper_bound)

5 動態規劃 (Dynamic Programming)

動態規劃（簡稱 DP）簡單來說就是以空間複雜度為代價，換取時間複雜度的優化。方法就是將之後可能會用到的計算結果存起來，以後就不用再重新算一遍了。

我們以兩種計算費氏數列的方法為例：

Algorithm 2: Fibonacci without DP

```
function fibo(i)
        if i = 0
                return 0
        else if i = 1
                return 1
        else
                return f(i - 1)+f(i - 2)
        end if
end function
```

Algorithm 3: Fibonacci with DP

```
array dp = {0}
function fibo(i)
        if dp[i] != 0
                return dp[i]
        else if i = 0
                return 0
        else if i = 1
                return dp[1] = 1
        else
                return dp[i] = f(i - 1) + f(i - 2)
        end if
end function
```

大家可以試著在自己的電腦上跑跑看。前一種方法在跑第四十幾項時就開始變得吃力了，而後一種方法計算到 1000000 項都沒問題（當然會有溢位的情況）。

我儘量將兩段程式碼寫得很像就是為了方便比較。可以發現第二段程式碼和第一段程式碼的差別在於，第二段程式碼中有將答案存在陣列中，未來如果還需要該答案的話可以直接從陣列取出；然而第一段程式碼中，就算是已經算過的項，未來還是要重新算一次。

5.1 狀態數、轉移式以及複雜度

那要如何估計 DP 的複雜度呢？首先，我們需要知道它的狀態數以及轉移式。

所謂的狀態數，就是你所存的答案的總數。以 Algorithm 3 為例，要計算第 n 項，需要儲存第 0 項到第 $n-1$ 項的答案，因此狀態數是 $O(n)$。

而所謂的轉移式，就是你如何利用已知的結果，計算出新的東西。同樣以 Algorithm 3 為例，$f(i)$ 的值的計算方法就是 $f(i-1)+f(i-2)$。我們就稱 $f(i)=f(i-1)+f(i-2)$ 為這個 DP 的轉移式，而計算這個式子的複雜度就稱為轉移的複雜度。在這個情況下，轉移

複雜度為 $O(1)$。

知道這兩個東西之後，複雜度也很好估計了。最慘的情況下，每個你存的狀態都需要去計算，而每次計算（轉移）的複雜度你也知道了，因此總複雜度就是把兩個乘在一起。以 Algorithm 3 為例，總複雜度為 $O(n) \times O(1) = O(n)$。

事實上，Algorithm 2 的時間複雜度是 $O(\varphi^n)(\varphi = \frac{1+\sqrt{5}}{2})$、空間複雜度是 $O(1)$，由此可見 DP 犧牲一些空間複雜度，換取的空間複雜度的極大改進。

5.2 滾動 DP

DP 的技巧有許多種，通稱為 DP 優化，每種學問都很深，可以有效地減少時間複雜度。這些我們之後會再提。現在我們先來看一種減少空間複雜度的技巧：滾動 DP。

我們可以用同樣的時間複雜度，但只用 $O(1)$ 的空間來達到剛剛的目標：

Algorithm 4: Fibonacci with rolling DP

```
function fibo(i)
        if i = 0
                return 0
        else
                prev_val = 0, now_val = 1
                for now_index = 1 to i
                        prev_val += now_val
                        swap prev_val and now_val
                end for
                return now_val
        end if
end function
```

其想法是：既然每次計算的時候只需要前兩項，那剩下那些用不到的就可以丟掉了。

通常這種實作在 prev_val 和 now_val 分別是陣列的時候行不通，因為將 now_val、prev_val 的每一個值交換會浪費許多時間。然而在 C/C++ 中，可以透過交換指標達成這個目的。

5.3 經典 DP 題

懶得找 judge，自己寫開心就好。

1. 最大子陣列和：給你一個整數陣列 $a_0, a_1, \cdots, a_{n-1}$。求一組 i, j 使得 $a_i + a_{i+1} + \cdots + a_j$ 最大。時間複雜度 $O(n)$，空間複雜度 $O(1)$。

2. 最長公共子序列（Longest Common Sequence, LCS）：給你兩個字串，長度分別為 m 和 n，請找出他們的最長公共子序列。時間複雜度 $O(mn)$。如果需要列出其中一個最長公共子序列，空間複雜度 $O(mn)$。如果只需求出最長的長度，空間複雜度 $O(\min(m,n))$。

3. (ZJ d637) 0/1 背包問題：你有一個包包能裝 W 公斤的東西。有 N 樣物品，已知所有物品分別的重量和價值。請問這個包包最多可以裝價值多高的物品？時間複雜度 $O(NW)$，空間複雜度 $O(W)$。

4. 無限背包問題：同前一題，然而同一樣物品有任意多件，你可以裝任意多個同一樣物品。時間複雜度 $O(NW)$，空間複雜度 $O(W)$。

5.（旅行推銷員問題）給你一張 n 個點的圖，給你任兩點之間的距離。求恰經過每個點一次後回到出發點的迴路長度最小值。時間複雜度 $O(n^2 2^n)$、空間複雜度 $O(n2^n)$。（提示：以一堆 0 和 1 為狀態，並用二進位轉換成一個數儲存。此技巧稱為位元 DP。）

5.4 習題

在做這些題目的時候，仔細想想，狀態要存什麼？轉移式怎麼寫？再回頭檢視複雜度是不是好的。

1. (ZJ d652) 有 $n \leq 50$ 個怪物排成一列（n 其實可以到 400）。每隻怪物都有一個大小 W_i。對於三個相鄰的怪物 A、B、C，A 和 C 可以一起把 B 給殺掉，並且產生汙染 $P = W_A W_B W_C$。要求這些怪物殺來殺去以後只留下最前面跟最後面的怪物。請問汙染的總和最小是多少？保證答案在 10^9 以內。

2. (ZJ d645) 背包問題，有些東西有限量，有些東西不限量。請想出一個複雜度 $O(nmw)$ 的作法，其中 n 是物品個數，m 是限量的最大值，w 是背包可容納的重量。（註：利用 DP 優化，可將複雜度降低至 $O(nw)$。）

3. (ZJ d054) 有幾種用單位小正方形還有由 3 個單位小正方形組成的 L 形拼滿一個 2 乘 n 的矩形的方法？（$n \leq 40$，但其實可以到 $n \leq 10^6$）

4. (ZJ a128)(ACM ICPC World Finals) 你有一個 $x \times y$ 的方格表（$x, y \leq 10^4$）。給你 $n \leq 15$ 個數，請問你是否能沿著某條橫線或直線切割方格表 $n-1$ 次，使得切出來的 n 塊面積跟那 n 個數一樣。（提示：位元 DP。）

5. (2016 三模 pB)(No judge) 我們稱一個物體的耐撞力為 H，若該物體從 H 公尺高度內自由落體下時不會壞掉，但超過 H 公尺時便會損壞。給你一個陣列 $P_1, P_2, \cdots, P_U$，其中 P_i 代表在物體從 i 公尺下落所需的實驗經費。物體在壞掉之前可以進行任意多次實驗，但壞掉之後就不能進行任何實驗。已知 H 在 [0,U] 中，問在要求該物體最多只能壞掉一次的情況下，保證可以得知確切 H 值的最小實驗花費。複雜度 $O(U^2)$。

（註：利用 DP 優化，可以快速找到轉移來源，將複雜度降低至 $O(U\log U)$。這也是原題要求的複雜度。）

7. (2015 一模 pA)(No judge) 給一個 $n\times n$（$n\le 22$）的表格，每格裡有一個數字 $V_{i,j}\le 10^6$。選擇其中若干格使得任兩格在八方位不相鄰（也就是說有公共邊或公共角都不行），求總和的最大值。（提示：位元 DP。）

6 貪婪演算法 (greedy)

遇到要求最大（最小值）時，每次決策都選擇當下最佳的選擇，這就是貪婪演算法。然而一般的情況下，貪婪演算法不保證正確性，因此什麼時候可以 greedy、要怎麼 greedy，都需要一定的練習才可以熟練。

6.1 貪婪失敗的實例

以 0/1 背包問題為例。考慮以下三種貪婪的準則：

1. 每次都將重量最小的物品塞進包包。複雜度 $O(n\log n)$。

2. 每次都將價值最高的物品塞進包包。複雜度 $O(n\log n)$。

3. 每次都將 CP 值（價值除以重量）最高的物品塞進包包。複雜度 $O(n\log n)$。

可以發現三者的複雜度都比前面的 DP 快許多，然而問題就在於三者都不保證正確性。有興趣的人不妨試著對三種準則分別構造反例。

從這個例子，可以發現雖然 greedy 可以有低時間複雜度，卻不一定正確。不過好的 greedy 演算法會給出跟正解足夠接近的解。而在某些問題裡，greedy 更是會保證正確性。

6.2 貪婪成功的實例

我們考慮下列的問題：給你 n 個線段。請問最多能取出幾個線段，內部互不重疊。我們再考慮以下三種貪婪的準則：

1. 將線段依線段長度排序，再由小到大選取。如果有重疊就不選，沒有重疊就選。複雜度 $O(n\log n)$。

2. 將線段依左端點排序，從最左邊的線段開始取。如果有重疊就不取，沒有重疊就取。複雜度 $O(n\log n)$。

3. 將線段依右端點排序，從最左邊的線段開始取。如果有重疊就不取，沒有重疊就取。複雜度 $O(n\log n)$。

對於第一個準則，考慮線段 [0,5][4,6][5,10] 就發現這準則不一定正確。
對於第二個準則，考慮線段 [0,3][1,2][2,3] 就發現這準則也不一定正確。
然而可以證明第三個準則保證正確（請讀者自行思考），這是一個 greedy 成功的案例。
這個例子告訴我們就算可以 greedy，也要選擇一種好的方法，不能胡亂地貪婪。

6.3 習題

當你 greedy 失敗的時候，要仔細思考：為什麼這樣 greedy 會失敗？我該怎麼樣改變我的準則才可以將剛剛的反例克服？當然，也要適時放棄 greedy 的想法。

1. (No judge) 有 n 位病人要看醫生。已知每個病人看醫生需要花的時間，而且只有一位醫生。請求出病人總等待時間的最小值。時間複雜度 $O(n \log n)$。

2. (Codeforces 665C) 給你一個長度 $\leq 2 \cdot 10^5$ 的字串。請用最少的 edit distance（改變最少個字元）使得相鄰兩個字元皆相異。

3. (Codeforces 701A) 給你 $n \leq 100$ 個數。請將這 n 個數兩兩分組使得每組和相同。保證做得到。

4. (ZJ b231)(2009 入營考 pC) 有 $N \leq 1000$ 本書（事實上 $N \leq 10^5$ 都可以），每本書都有所需要的印刷時間和裝訂時間。你可以同時裝訂任意多本書，但同一時間只能印刷至多一本書。每本書需要先印完再裝訂。請問你至少要花多久才能將所有書都印刷裝訂完畢。

7 二分搜

7.1 一般的二分搜

通常一般的二分搜是在解決以下這種問題：如果有一個遞增的函數 f 定義在區間 $[a, a+n)$ 上，請求出滿足 $f(s) \geq c$ 的最小整數 s。

如果你從 a 開始暴搜，直到找到一個滿足條件的 s，那麼複雜度是 $O(n)$。這時我們可以使用二分搜來解決這樣的問題，優化時間複雜度。想法是對於某個在 $(a, a+n)$ 中的整數 k，如果 $f(k-1) \geq c$，那麼 $s < k$，也就是說你要求的答案會落在區間 $[a, k)$ 中。反之，如果 $f(k-1) < c$，那麼 $s \geq k$，也就是說你要求的答案會落在 $[k, a+n)$。為了讓兩種情況的可能性都儘量低，你可以發現你應該要取 k 愈接近 $a + n/2$ 愈好。如此一來，每次候選區間的長度都會縮小一半，因此複雜度為 $O(\log n)$。

實務上，這種函數 f 常常不能直接得出某一點的值 $f(a)$（甚至只能確認它和 c 的大小關係），而需要 $O(M)$ 的時間來計算。顯然地，這時複雜度是 $O(M \log n)$。

順帶一提，lower_bound 和 upper_bound 便是用二分搜實作的。

實作上要注意的是加一和減一不要搞混、左閉右開和閉區間不要搞混，不然很有可能就變成無窮迴圈。以下是虛擬碼：

Algorithm 5: Binary Search

```
function binary_search(array[], first, last, val)
        while first + 1 < last
                mid = (first + last) / 2
                if (array[mid - 1] < val)
                        first = mid
                else
                        last = mid
                end if
        end while
        return first
end function
```

7.2 題外話：三分搜

利用二分搜這種「縮短候選人長度」的想法，我們可以找出滿足特定性質的函數的最小值，這種技巧稱為三分搜。三分搜處理的問題如下：有一個在 $[a, a+n)$ 中先嚴格遞減再嚴格遞增的函數 f，請求出 f 在 $[a, a+n)$ 的最小值。

取在 $[a, a+n)$ 中的兩個整數 $x < y$。如果 $f(x) < f(y)$，那麼最小值一定落在 $[a, y)$。如果 $f(x) > f(y)$，那麼最小值一定落在 $(x, a+n)$。如果 $f(x) = f(y)$，那麼最小值一定落在 (x, y)。為了讓候選區間每次都會縮短一定的比例，通常都取 x 跟 y 為區間的三等分點（取中間一點的話常數會變小）。複雜度仍然是 $O(\log n)$。

7.3 對答案二分搜

有許多問題都喜歡叫你求「滿足條件的最小值」這種東西。如果這個問題滿足「單調性」，那或許可以考慮對答案二分搜。

什麼是「單調性」呢？考慮一個函數 P，如果 s 滿足條件，那麼 $P(s) = 1$，反之則為 0。如果 P 有單調性，我們就說這個問題有單調性。這樣的好處是，我們可以直接用前面的方法二分搜出要求的 s。如果計算 P 的複雜度並不大時，這樣的方法可以有非常好的表現效率。在你沒辦法快速求出 s 而只能快速確認一個 s 是否符合條件時，這是一個非常好的方法。

7.4 習題

1. (TIOJ 1839)(IOI 2013)(Interactive) 有 $n \leq 5000$ 個開關，分別（不照順序地）連著 n 個門。對於每個開關，要嘛開的時候門會開，要嘛關的時候門會開，反之則反。你最多可以詢問 70000 次，對於每次詢問，你可以給一個 n 個開關的配置，程式會告訴你第一個關著的門是幾號門。請找出開關和門之間的對應關係，以及會讓所有門都開的開關配置。

2. (TIOJ 1341)(IOI 2007)(Interactive) 有圖，詳見題敘。
（提示：一開始將候選區間不斷倍增，直到確定所求答案位於候選區間內。有人稱這種方法為倍增法。）

3. (TIOJ 1815)(IOI 2013) 你有 $T \leq 10^6$ 個玩具、$A \leq 5 \cdot 10^4$ 個弱雞機器人和 $B \leq 5 \cdot 10^4$ 個小不點機器人。每個玩具的重量為 W_i、大小為 S_i。每個弱雞機器人每次可以拿起的玩具重量不能超過 X_i（大小不限制），而小不點機器人每次可以拿起的玩具大小不能超過 Y_i（重量不限制）。兩種機器人一次都只能拿一種玩具，每拿起一個玩具並放在好好的地方需要花一分鐘，但不同的機器人可以同時拿玩具。請問你至少需要用幾分鐘才能收拾完所有玩具（或不可能收完）?
（提示：greedy。）