Ch5. 搜尋 Search

本章要介紹的是「搜尋」。

首先，跟排序一樣，會定義什麼叫搜尋，並說明搜尋為何如此重要。

接著，會說明有哪些搜尋問題，以及在不同的資料下，各會遇到哪些搜尋狀況。

再來，會說明一些常見的搜尋方式：

A. 循序搜尋 Sequential Search
B. 二分搜尋 Binary Search
C. 插補搜尋 Interpolation Search
D. 黃金切割搜尋 Golden Section Search
E. 二元搜尋樹搜尋 Binary Tree Search
F. 雜湊搜尋 Hashing Search
G. 費氏搜尋 Fibonacci Search

不過，因為其中的「二元搜尋樹搜尋」、「雜湊搜尋」、「費氏搜尋」主要是依賴相應的資料結構實作，所以本書主要介紹「循序搜尋」、「二分搜尋」、「插補搜尋」與「黃金切割搜尋」，其中，最重要、最常考到的是「二分搜尋」。

接著，會把搜尋做總結，並做一些實戰練習。

1. 搜尋簡介

首先，搜尋是要在一個集合內，找出具特定鍵值 Key 的元素，該集合可能為未排序或已排序。

不同的資料結構會影響到搜尋的方式與效率，視資料是以陣列、雜湊表、二元樹，還是其他結構儲存，在搜尋時的適用方式與效率都會有很大不同。

同樣的，搜尋也會有 Worst / Average / Best case。通常 Best case 都是 $O(1)$，代表第一個檢查的元素正好就是要找的目標資料。

（1）搜尋的分類

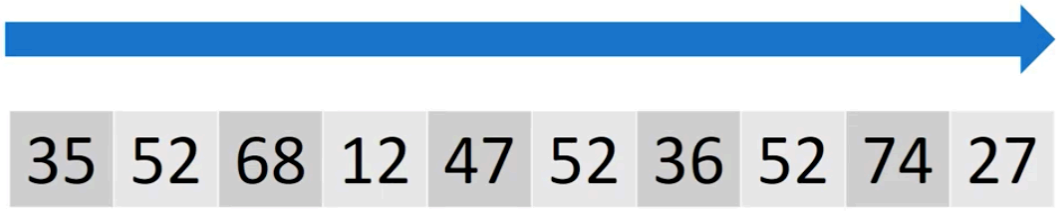
搜尋一樣可以分為「內部搜尋」和「外部搜尋」。內部搜尋指的是搜尋過程中，資料可以全部載入記憶體中，外部搜尋則代表搜尋過程中會用到外部記憶體。

另外，Successful Search 代表成功找到該筆資料在資料結構中的位置，Unsuccessful Search 則代表確定該筆資料並不存在於要找的結構中。

本章中介紹搜尋時，使用的資料結構皆為陣列，不過同樣的方法也可以應用在各種 Sequential container 上，比如 向量 Vector、鏈結串列 Linked list 也適用。

2. 循序搜尋

循序搜尋，顧名思義就是從資料結構中的第一筆資料開始，一筆一筆向後找，直到最後一筆。



這其實就是一種暴力解，比如在上圖的陣列中要找到 74 這筆資料，就從 35、52、68、... 如果一筆資料不符合要找的值，就繼續往後找。

進行循序搜尋時，資料不需要事先經過排序（要記得如排序等資料「前處理」也會花費時間），而因為最壞狀況下目標可能是容器中最後一筆，前面每筆資料都要被檢查，所以複雜度為 $O(n)$。

（1）實作循序搜尋

循序搜尋	
1	// 傳入引數中，data[]是資料結構
2	// len 是資料筆數、target 是要找的目標
3	int sequential_search(int data[], int len, int target){
4	for (int i=0 ; i<len ; i++){
5	// 目前檢查的資料是目標資料時，回傳索引值
6	if (data[i]==target)
7	return i;
8	// 找到陣列結尾仍然沒有找到時，回傳 -1
9	else if (i==len-1)
10	return -1;
11	}
12	}

當容器中有數筆資料都與目標資料的值相同時，會回傳的是最前面那一筆的索引值。

（2）循序搜尋的複雜度

Worst case：檢查到容器結尾才找到，$O(n)$
Best case：在容器第一筆資料就找到，$O(1)$

Average case：

平均要找的次數是（資料可能出現在第一筆、第二筆、第三筆、…、最後一筆）

$\frac{1+2+3+\cdots+n}{n} = \frac{n+1}{2}$，複雜度為 $O\left(\frac{n}{2}\right) = O(n)$。

3. 二分搜尋法

二分搜尋法是最重要的搜尋演算法之一，包括插補搜尋、黃金切割搜尋等，都是二分搜尋法的延伸。

二分搜尋法顧名思義，是把要找的資料分成兩邊，每次檢查一筆資料後，就用刪去法刪去其中一半的可能。

但是應用二分搜尋法時，資料必須事先排序好，才知道要刪除哪一半，且使用的資料結構必須支援隨機存取（透過索引值直接取出資料），否則效能會很低落。

每次檢查一筆資料後，會分成三種狀況：

A. 確定找到，或者找不到要搜尋的目標（兩種結束條件）
B. 尚未找到，但確定目標會出現在陣列的前半部
C. 尚未找到，但確定目標會出現在陣列的後半部

（1）二分搜尋法的進行過程

給定如上已經排序好的陣列，要找到 27 的位置（索引值），先取出資料的中位數，索引值為 5（$\left\lfloor \frac{10}{2} \right\rfloor = 5$）的 52 來檢查：

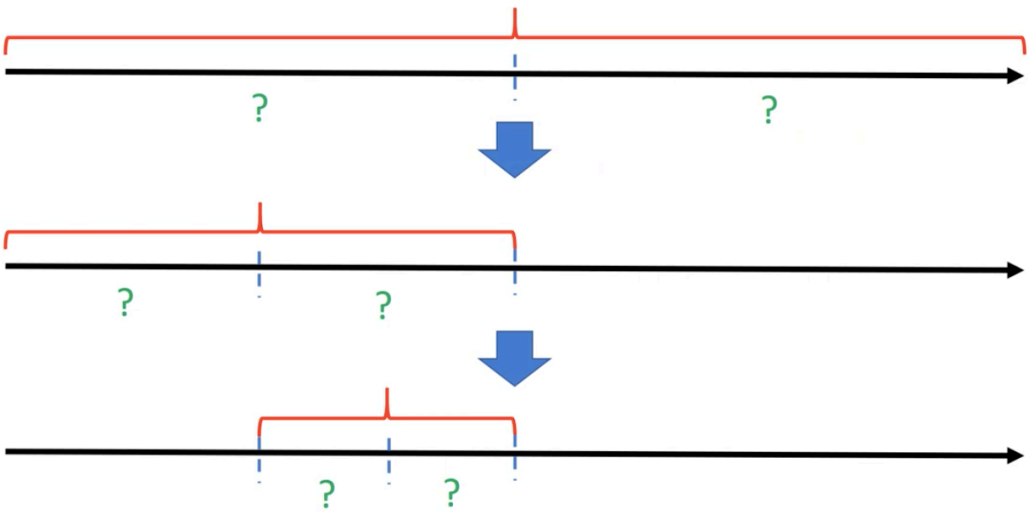
因為 $52 > 27$，所以 27 會出現在 52 的前面，也就是陣列的前半段，此時，陣列的後半段就可以不再理會。

接下來，再取出索引值為 $\left\lfloor \frac{5}{2} \right\rfloor = 2$ 的 35 來檢查：

因為 $35 > 27$，所以 27 會出現在 35 前面，只需再往前面檢查即可。

再來，取出索引值 $\left\lfloor \frac{2}{2} \right\rfloor = 1$ 的 27，發現符合值符合目標，回傳索引值 1。

如下圖所示，二分搜尋法就是每次都把剩餘資料切成左右兩邊，並且決定目標會出現在中位數的左邊還是右邊，繼續針對那個區間尋找。



（2）二分搜尋法的複雜度

Worst case 下，要一路找到最後一個中位數才找到，共要檢查 $\log_2 n$（例如 8 筆資料最多要找 3 次）次中位數，因此複雜度為 $O(\log_2 n)$。

Best case 則是第一次取中位數時，正好就與目標相符，複雜度為 $O(1)$。

平均而言，要找的次數是

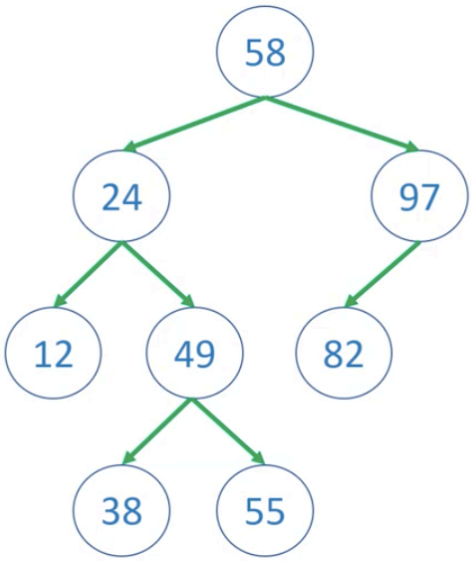
$$\frac{[1+2+3+\cdots+\log_2 n]}{[\log_2 n]}$$

$$=\frac{[\log_2 n]+1}{2}$$

複雜度為 $O(\log_2 n)$。

（3）二元搜尋樹

另外，若用二元搜尋樹來進行搜尋，則如圖中所示，資料就是樹的每個節點。二元搜尋樹的特性是每個節點的左子節點編號一定比父節點的編號小，右子節點編號一定比父節點編號大。



要在樹上找到一個特定編號，可以依照下列步驟，由根節點開始向下進行：

A. 當要找的編號與現處節點一致，結束搜尋

B. 當要找的編號跟現處節點不一致

   a. 尋找的編號比節點編號小，往左節點移動

   b. 尋找的編號比節點編號大，往右節點移動

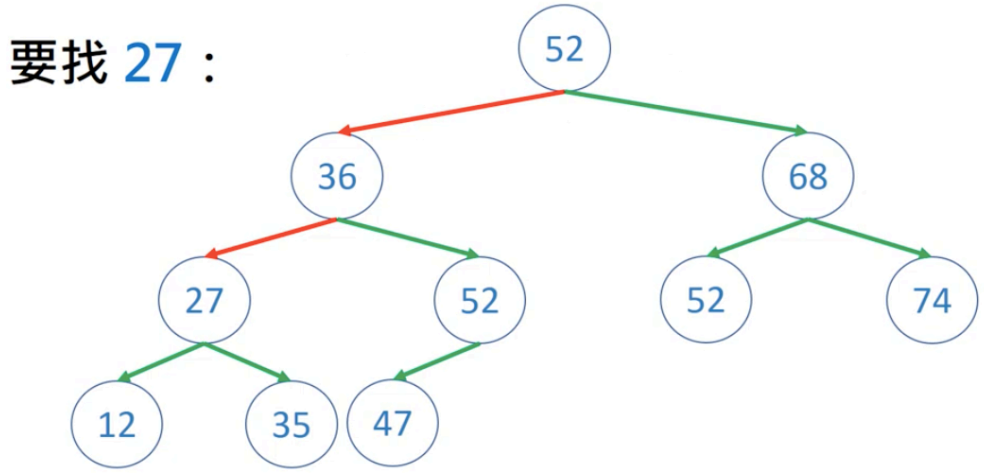
C. 當現處節點為葉節點，代表要找的編號不存在樹上，結束搜尋，並回傳空指標

若是完滿二元樹，每經過一次分岔，就可以刪去一半的資料，因此經過 n 次搜尋後，可以找完一個有 2^{n-1} 個節點的二元搜尋樹。

因為搜尋次數約等於 $\log_2$(資料數目)，所以時間複雜度同樣為 $O(\log_2 n)$。

12	27	35	36	47	52	52	52	68	74

如果要在一個資料順序對應到二元搜尋樹的陣列（對應方式請參考資料結構相關書籍）當中尋找一筆資料，根據上面的規則，從 52 開始，因為 $52 > 27$，所以往左搜尋左子樹，左子樹的根節點 36 也大於 27，因此繼續往 36 的左子樹尋找，順利找到 27。



（4）二分搜尋法的實作

二分搜尋法

```
1  #include <iostream>
2  using namespace std;
3
4  // 傳入搜尋的容器 data[]
5  // 目前尋找的區間 [lower, upper]
6  // 要找的目標 target
7  int Binary_Search(int data[], int lower, int upper, int target){
8
9      // 上界慢慢變小，下界慢慢變大
10     // 直到要找的區間已經沒有資料時，代表目標不存在容器中
11     // 回傳 -1
12     if (upper<lower)
13         return -1;
14
15     // 中位數索引值
16     int middle = (lower+upper)/2;
17
18     // 中位數資料與目標值相同
19     if (data[middle]== target){
20         return middle;
21     }
22     // 尋找中位數的左邊
23     else if (data[middle]>target){
24         return Binary_Search(data, lower, middle-1);
25     }
26     // 尋找中位數的右邊
27     else if (data[middle]<target){
28         return Binary_Search(data, middle+1, upper, target);
29     }
30
31 }
```

```
32
33  int main()
34  {
35      int target;
36
37      // 已排序的陣列
38      int arr[10] = {12,27,35,36,47,52,52,52,68,74};
39      cout << "Please enter the target you would like to search:" << endl;
40      cin >> target;
41
42      // 回傳 target 的索引值
43      cout << "Index of " << target << " is " << Binary_Search(arr, 0, 9, target);
44
45      return 0;
46  }
```

執行結果

```
Please enter the target you would like to search:
>> 35
Index of 35 is 2

Please enter the target you would like to search:
>> 38
Index of 38 is -1
```

（5）用二分搜尋法求出根號近似解

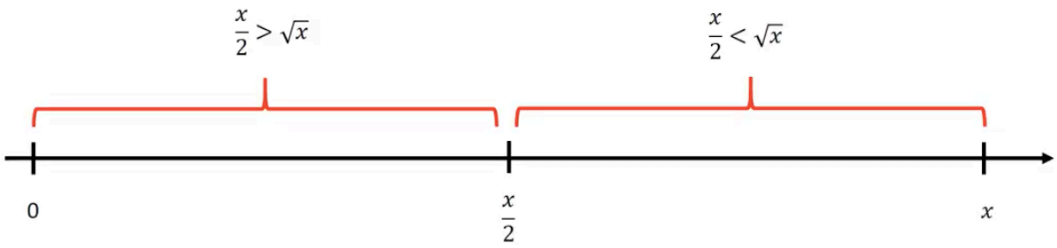
A. 題目

讓使用者輸入一個大於 1 的數 x 與可容許誤差 E，透過二分搜尋法求出「$\sqrt{x} = y$」，使得誤差 $y^2 - x$ 在 E 以內。

B. 範例輸入與輸出

Please enter a number(>1) and error:
>> 5 0.000001
The square root is 2.23607

C. 解題邏輯



因為 x 大於 1，所以 $\sqrt{x}$ 的值一定介於 0 與 x 間。

利用二分搜尋法，把該區間分為左右兩半，並比較中位數 $\frac{x}{2}$ 與 $\sqrt{x}$ 何者較大，就可以決定繼續往哪邊向下搜尋。

如何決定 $\frac{x}{2}$ 與 $\sqrt{x}$ 何者較大呢？把兩邊平方，比較 $\frac{x^2}{4}$ 與 x，就可能知道應搜尋左邊或右邊，接著，再把該區間分為一半往下搜尋，直到誤差小於可容許誤差時，結束搜尋。

可容許誤差越大時，需要進行的搜尋次數越少，相反的，可容許誤差越小，就要進行越多次搜尋，不過因為二分搜尋法每次都可以把誤差減少一半左右，所以通常不需進行太多次，比如誤差若要縮小到原始誤差的 $\frac{1}{1000}$，也只要進行

10 次搜尋即可（$\frac{1}{2^{10}} < \frac{1}{1000}$）。

用二分搜尋法求出根號近似解

```
1  #include <iostream>
2  #include <cmath> // for abs
3  using namespace std;
4
5  // 傳入值 x 是要求根號的數字
6  // lower 到 upper 是目前的區間
7  // error 是可容許誤差
8  double Square(double x, double lower, double upper, double error){
9      double middle = (lower+upper)/2;
10
11     // 誤差小於容許值時，回傳 middle
12     if (abs(middle*middle-x) < error){
13         return middle;
14     }
15     // 搜尋 middle 左邊
16     else if (middle*middle > x){
17         return Square(x, lower, middle, error);
18     }
19     // 搜尋 middle 右邊
20     else if (middle*middle < x){
21         return Square(x, middle, upper, error);
22     }
23
24 }
25
26 int main()
27 {
28     double x, e;
29     cout << "Please enter a number(>1) and error:" << endl;
30
```

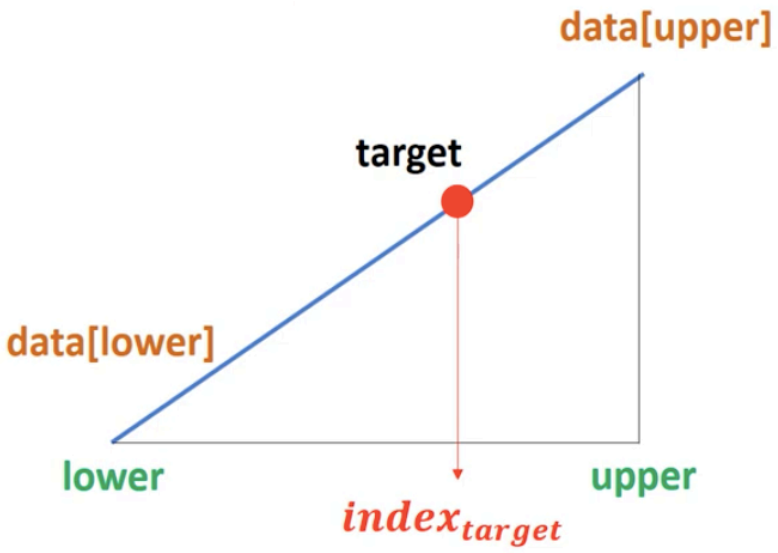
```
31      cin >> x >> e;
32      // 下界也可傳入 1，如 Square(x, 1, x, e)
33      cout << "The square root is " << Square(x, 0, x, e) << endl;
34
35      return 0;
36  }
```

執行結果

```
Please enter a number(>1) and error:
>> 57 0.00001
7.54983
```

4. 插補搜尋

插補搜尋是二分逼近法的改良版，用「內插法」來找出資料最可能存在的索引值位置。因為假設資料的值平均分佈，比起二分搜尋直接把資料切成兩半，插補搜尋會透過目標和「最大值」、「最小值」分別的距離，來猜資料最可能位在哪個索引值。



像上面的三角形一般，$lower$ 是區間內最低的索引值、$upper$ 是區間內最高的索引值，如果資料平均分佈在 $data[lower]$ 和 $data[upper]$ 之間（相鄰資料間都相差同樣的值），把每一筆資料的值畫出來，都會落在三角形的斜邊上。

而把 $target$ 的值標在斜邊上後，就可以透過公式來把對應到底邊上的值算出來，也就知道 $target$ 這個值最可能在 $lower$ 到 $upper$ 間的哪個索引值。

（1）插補搜尋的目標

公式如下：因為 $data[lower] - target - index_{target}$ 和 $data[lower] - data[upper] - upper$ 兩個三角形為相似三角形：

$$\frac{target - data[lower]}{index_{target} - lower} = \frac{data[upper] - data[lower]}{upper - lower}$$

移項之後，可以得到：

$$index_{target} = lower + \frac{(target - data[lower])(upper - lower)}{data[upper] - data[lower]}$$

透過上面的公式，就可以不用總是假設要找尋的目標會出現在區間中間，而是去找資料平均分布時最可能出現的索引值。

比如若資料是 1、2、3、...、10 共十筆，而要找的目標是 8，那麼顯然目標會比較接近 10，而非正好位在中間，如果使用二分搜尋法，仍然會以中間的 5 或 6 來切出兩個區間，但如果改用插補搜尋，就可以很快找到精確的位置。

不過，當實際資料並不符合平均分佈的假設時，插補搜尋不一定會比二分逼近法來得好。

（2）插補搜尋的實作

插補搜尋

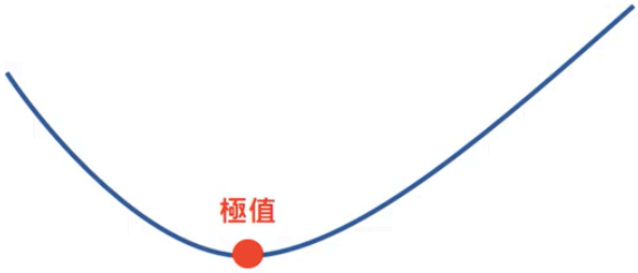
```
1  #include <iostream>
2  using namespace std;
3
4  int Interpolation_search (int data[], int lower, int upper, int target){
5
6      // 邊界情況：區間內沒有資料
7      if (upper<lower)
8          return -1;
9
10     // 一般情況
11
12     // 透過公式算出目標最可能在的索引值
13     int upper_data = data[upper];
14     int lower_data = data[lower];
15     int index = lower+(target-lower_data)*(upper-lower)/(upper_data-
16     lower_data);
17
```

```
18        // 與二分搜尋法一樣分成三種情況處理
19        // data[index] 正好就是目標
20        if (data[index]==target){
21              return index;
22        }
23        // data[index] 比目標的值大
24        // 尋找左邊區間
25        else if (data[index]>target){
26              return Interpolation_search(data, lower, index-1, target);
27        }
28        // data[index] 比目標的值小
29        // 尋找左邊區間
30        else if (data[index]<target){
31              return Interpolation_search(data, index+1, upper, target);
32        }
33
34  }
35
36  int main()
37  {
38        int target;
39
40        int arr[10] = {12,27,35,36,47,52,52,52,68,74};
41        cout << "Please enter the target you would like to search:" << endl;
42
43        cin >> target;
44
45        // 回傳 target 的索引值
46        cout << "Index of " << target << " is " << Interpolation_Search(arr, 0, 9,
47        target);
48
49        return 0;
50
51  }
```

執行結果
Please enter the target you would like to search: >> 27 Index of 27 is 1 Please enter the target you would like to search: >> 28 Index of 28 is -1

5. 黃金切割搜尋

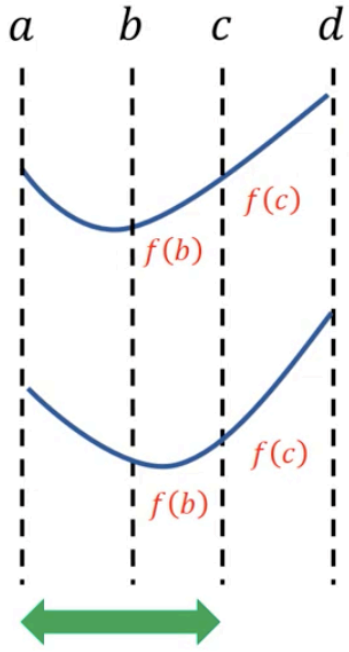
黃金切割搜尋只適用於特定的狀況，可以用來找出函式在某個區間內的極值，不過只適用在「單峰函式 unimodal function」上。



單峰函式指的是函式在區間內的二階微分只有正值或只有負值，也就是說，在該區間內只會出現「一個」極值而已。

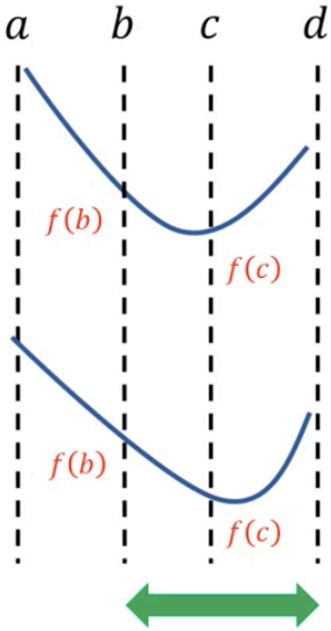
（1）黃金切割搜尋的原理

要找出一個單峰函式的極值時，二分搜尋法並不適用，因為即使從中間切一刀後，仍然無法判別極值會出現在左邊還是右邊。所以面對一個單峰函式時，必須同時切兩刀，如下圖中的 b 和 c，a 和 d 則分別是區間下界與上界。

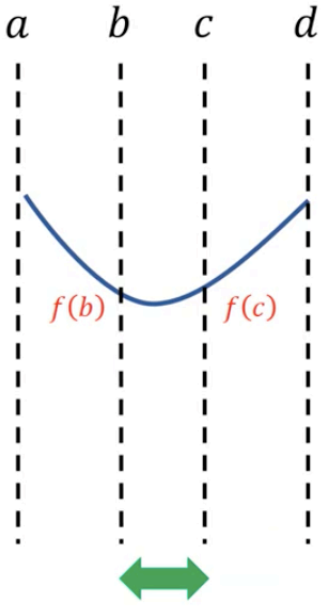


如果在 b 和 c 這兩個位置得到的函式值 $f(b)$ 和 $f(c)$ 有 $f(c) > f(b)$，那麼一定是上面兩種狀況之一：極值出現在 b 的左邊，或者極值出現在 b 和 c 之間。

不管是兩種狀況中哪一個，極值都是出現在 [a,c]，就可以排除極值在 [c,d] 的可能性。

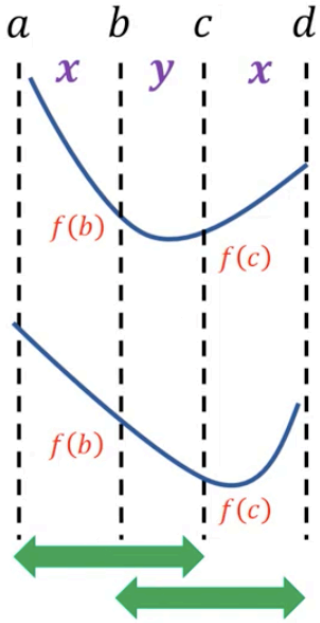


反之，如果 $f(b) > f(c)$，那麼也有兩種情形：極值出現在 c 的右邊，或者出現在 b 和 c 之間。這兩種狀況下，極值都會出現在 [b,d]，也就可以排除極值在 [a,b] 的可能性。

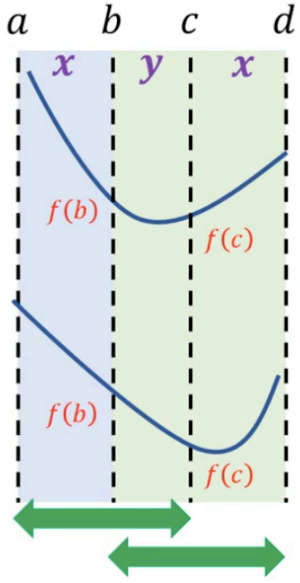


另外，如果 $f(b)=f(c)$，則極值會出現在 [b,c]，不過因為這種情況很少出現，所以可以與上面兩種情形任一合併處理。

（2）選擇切割點



黃金切割搜尋中，每次都要切出兩刀，怎麼切會使得效率最好呢？因為把區間限縮後，還要繼續往下搜尋，所以如果上一輪切割出的兩刀，其中一刀變成下一輪的上 / 下界，而另外一刀則正好可以沿用為下一輪兩刀中的其中一刀，則每一輪實質上就只需要多切一刀就好了（每切一刀都要取出該處的函式值，會耗費時間，所以執行越少次越好）。



比如上圖中，因為 $f(b) > f(c)$，所以下一輪會針對 [b,d] 搜尋，此時 b 變成下一輪的下界，而若 c 能夠被沿用為下一輪的其中一刀（下一輪兩刀中左邊那一刀），則下一輪只要多切右邊那一刀即可。

這要求特定的長度關係：

$$\overline{ab}:\overline{ad} = \overline{bc}:\overline{bd}$$

又因為兩端一定要對稱（因為有時會取到 [a,c]，有時會取到 [b,d]），所以可以假設 $\overline{ab}:\overline{bc}:\overline{cd} = x:y:x$。

這樣一來，可以列出等式：

$$\frac{\overline{ab}}{\overline{ad}} = \frac{\overline{bc}}{\overline{bd}}$$

$$\frac{x}{2x+y} = \frac{y}{x+y}$$

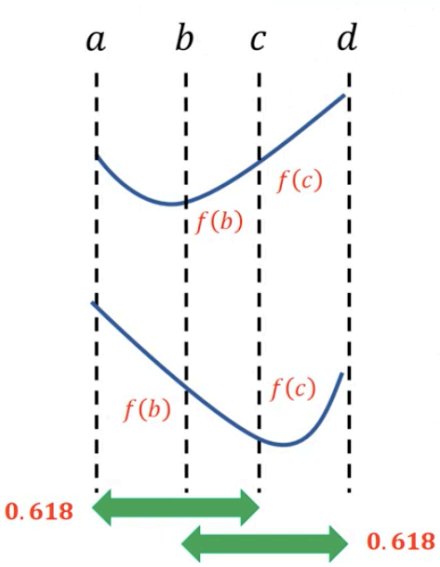
求解上式，會得到：

$$\frac{x}{y} = \frac{1+\sqrt{5}}{2}$$

也就是說，$\overline{ac}:\overline{ad}$ 和 $\overline{bd}:\overline{ad}$ 的值都是

$$\frac{x+y}{2x+y} = \frac{-1+\sqrt{5}}{2} \approx 0.618$$

用這個比例，就可以找到 b 和 c 應該切的位置，0.618 正好就是「黃金比例」。



上圖中，先根據黃金比例切出 b 和 c，並且得到 $f(b)$ 和 $f(c)$ 的值。

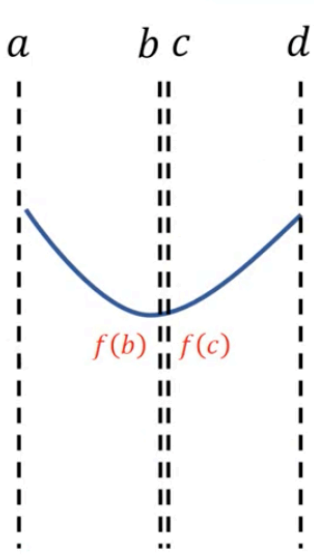
A. 如果 $f(c) > f(b)$

極值落在 [a,c]，下一輪要再切出 e，使得 $\overline{ce}:\overline{ac} = 0.618:1$

B. 如果 $f(c) < f(b)$

極值落在 [b,d]，下一輪要再切出 e，使得 $\overline{be}:\overline{bd} = 0.618:1$

（3）黃金切割搜尋的效率



為什麼不直接在中點附近取兩個很接近的點（二分搜尋），而要使用黃金比例來切割呢？不是只要任意切兩刀，都可以縮減區間嗎？更何況切在中點附近時，每輪排除的區間是 50%，似乎比黃金切割的 38.2%（$1-0.618$）來得大。

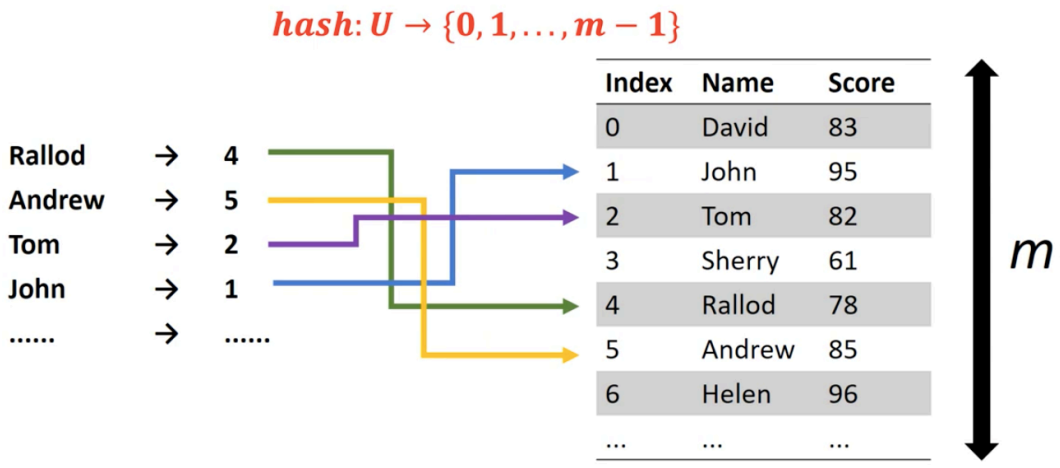
這是因為如果選擇中點附近取兩個很接近的點，那麼下一輪這兩個點一定都不能延用成切割點，因此必須要再呼叫兩次切割函式。

這樣一來，事實上「每次」呼叫切割函式只縮減了 25% 的總區間，而黃金切割搜尋則因為可以複用上一輪的運算結果，呼叫一次函式就可以縮減 38.2% 的區間，也就是說，在同樣的呼叫次數下，黃金切割搜尋可以排除更大比例的區間，效率較高。

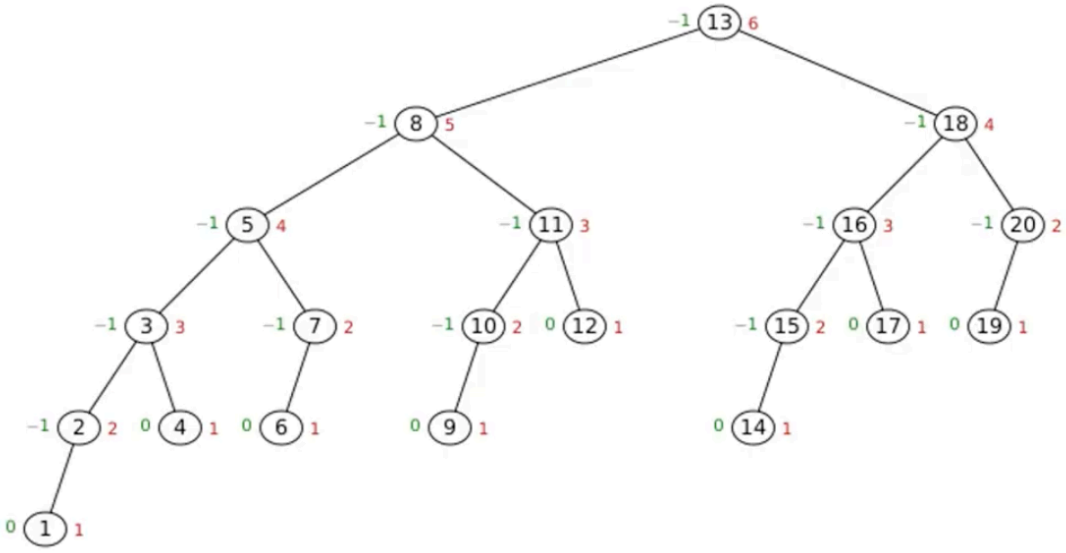
（4）雜湊搜尋與費氏搜尋

在進入搜尋的總結前，很快提一下雜湊搜尋與費氏搜尋。

雜湊搜尋指的是給定任意的輸入（目標），經過「雜湊函式」的轉換後，會輸出一個介於 0 到 $m-1$ 之間的值（索引值）。



只要有適當的雜湊函式，每次要搜尋特定值時，複雜度都只要 $O(1)$。



費氏搜尋則是結合費氏數列和樹的形式儲存資料，利用這種儲存方式，可以讓搜尋的速度快上很多。

（5）LeetCode #852. 山形陣列中的最大值 Peak Index in a Mountain Array

A. 題目

如果一個陣列 arr 符合以下條件，就把它叫做「山形陣列」：

a. $arr.length \geq 3$

b. 存在某個 i，且 $0 < i < arr.length - 1$，使得

$arr[0] < arr[1] < \cdots < arr[i-1] < arr[i]$

$arr[i] > arr[i+1] > \cdots > arr[arr.length - 1]$

給定一個山形陣列 arr，回傳所有符合條件的 i，使得

$arr[0] < \cdots < arr[i-1] < arr[i] > arr[i+1] > \cdots > arr[arr.length - 1]$

B. 出處

https://leetcode.com/problems/peak-index-in-a-mountain-array/

C. 範例輸入與輸出

輸入：arr = [0,1,0]

輸出：1

輸入：arr = [0,10,5,2]
輸出：1

D. 解題邏輯

本題其實也是要找出一個單峰函式的極值，只是本節的討論是以找「最小值」來說明，此題則要找「最大值」。

本題有三種解法：循序搜尋、二分搜尋和黃金切割搜尋。

循序搜尋找出山形陣列中的最大值	
1	class Solution{
2	public:
3	int peakIndexInMountainArray(vector<int>& arr){
4	
5	// 極值不會出現在開頭或結尾，所以 i 從 1 到 arr.size()-2
6	for (int i=1 ; i<arr.size()-1 ; i++){
7	
8	if (arr[i]>arr[i-1] && arr[i]>arr[i+1])
9	return i;
10	
11	}
12	
13	// leetcode 要求處理例外
14	return -1;
15	}
16	
17	};

二分搜尋找出山形陣列中的最大值（在很靠近中間的位置切兩刀）

```
1  class Solution{
2  public:
3      int peakIndexInMountainArray (vector<int>& arr){
4
5          int lower = 1;
6          int upper = arr.size()-2;
7
8          while (upper>=lower){
9
10             // 避免溢位，不直接相加除以 2
11             int middle = lower + (upper-lower)/2;
12
13             // 找到目標值
14             if (arr[middle]>arr[middle-1]&&arr[middle]>arr[middle+1]){
15                 return middle;
16             }
17             // 這次的 middle 位在左邊山坡上，要往右邊區間找
18             else if (arr[middle]<arr[middle+1]){
19                 lower = middle+1;
20             }
21             // 這次的 middle 位在右邊山坡上，要往左邊區間找
22             else {
23                 upper = middle-1;
24             }
25
26         } // end of while
27
28         // leetcode 要求處理例外
29         return -1;
30
31     }
32
33 };
```

黃金切割尋找山形陣列中的最大值

```
1   class Solution{
2
3       // 回傳黃金切割的索引值
4       // 也可以選擇不使用 0.382 和 0.618，改用數學式算出實際精確值
5
6       int gold_1(int lower, int upper){
7           return lower + (upper-lower)*0.382;
8       }
9
10      int gold_2(int lower, int upper){
11          return lower + (upper-lower)*0.618;
12      }
13
14  public:
15
16      int peakIndexInMountainArray(vector<int>& arr){
17
18          int lower = 0;
19          int upper = arr.size()-1;
20
21          int cut_1 = gold_1(lower, upper);
22          int cut_2 = gold_2(lower, upper);
23
24          // cut_1 和 cut_2 切出的區間有資料時可以繼續執行
25          while (cut_2>cut_1){
26
27              // 如果極值在 cut_2 左邊
28              // cut_2 變為上界
29              // cut_1 變為下一輪的 cut_2
30              // 新切出下一輪的 cut_1（區間為 [lower, cut_2]）
31              if (arr[cut_1]>arr[cut_2]){
32                  upper = cut_2;
```

```
                    cut_2 = cut_1;
                    cut_1 = gold_1(lower, cut_2);
                }

                // 如果極值在 cut_1 右邊
                // cut_1 變為下界
                // cut_2 變為下一輪的 cut_1
                // 新切出下一輪的 cut_2（區間為 [cut_1, upper]）
                else {
                    lower = cut_1;
                    cut_1 = cut_2;
                    cut_2 = gold_2(cut_1, upper);

                }

            }

            // 因為這題不能無限切割下去（索引值必須是整數）
            // 所以上面迴圈只能先限縮 lower 和 upper 的值
            // 仍然要在這個區間內尋找極值

            // 因為極值不會出現在開頭及尾端，要調整 lower 和 upper
            lower = (lower>0 ? lower : 1);
            upper = (upper<arr.size()-1 ? upper : arr.size()-2);

            for (int i=lower ; i<=upper ; i++){
                if(arr[i]>arr[i-1]&&arr[i]>arr[i+1])
                    return i;
            }
            return -1;

        } // end of peakIndexInMountainArray

}; // end of Solution
```

6. 搜尋總結

接下來，把本章介紹的幾種搜尋方法做個總結。

首先，資料可以分成「已經過處理」與「未經過處理」兩種。

A. 如果未經過處理，則只能使用循序搜尋
B. 如果經過事先排序，就能使用二分搜尋、插補搜尋
C. 如果事先建立好雜湊表，就能使用雜湊搜尋

不過資料「前處理」花費的時間也要納入考慮，只想要搜尋一次時，可以使用循序搜尋就好。通常是在未來需要進行非常多次的搜尋時，才選擇先排序好，使往後的每次搜尋都能大幅加速，就是「前人種樹、後人乘涼」的概念。

本章介紹的各種搜尋法中，以二分搜尋最常考，請讀者務必熟悉其原理與實作。

（1）各種搜尋的複雜度

搜尋複雜度	最好複雜度	平均複雜度	最壞複雜度
線性搜尋	$O(1)$	$O(n)$	$O(n)$
二分搜尋	$O(1)$	$O(\log_2 n)$	$O(\log_2 n)$
二元樹搜尋	$O(1)$	$O(\log_2 n)$	$O(\log_2 n)$
插補搜尋	$O(1)$	$O(\log_2 n)$	$O(n)$
雜湊搜尋	$O(1)$	$O(1)$	$O(1)$

二分搜尋和二元樹搜尋每次都會把要找尋的區間減半，因此平均複雜度都為 $O(\log_2 n)$。

插補搜尋則在最壞狀況下需要 $O(n)$，因為若資料分佈非常不平均，如在 [1,99,100,100,100,...,100] 中搜尋 99，因為會一直去找陣列的後端，所以效率不佳。不過插補搜尋的平均複雜度仍是 $O(\log_2 n)$。

7. 實戰演練

接下來，有三題實戰練習，都與實務上最常用的二分搜尋法有關。

（1）LeetCode #35. 找尋插入位置 Search Insert Position

A. 題目

給定一個已排序的整數陣列（數字不重複）和目標值，如果目標在陣列中，回傳其索引值，如果不在陣列中，則回傳其應該插入在哪個索引值，才能使陣列仍然符合排序。

B. 出處
https://leetcode.com/problems/search-insert-position/

C. 範例輸入與輸出
輸入：nums = [1,3,5,6], target = 5
輸出：2

輸入：nums = [1,3,5,6], target = 2
輸出：1

上面的例子中 target 是 2 時，要插入在索引值 1 才能維持陣列排序。

找尋插入位置 Search Insert Position	
1	class Solution{
2	
3	// 使用二分搜尋法
4	int binary_search(vector<int>& nums, int lower, int upper, int target){
5	
6	// 沒有找到時，回傳應該插入的位置 lower
7	if (upper<lower)
8	return lower;
9	

```
10        int middle = lower + (upper-lower)/2;
11
12        // 比較 nums[middle] 與 target 的大小
13        if (nums[middle]==target){
14              return middle;
15        }
16        else if (nums[middle]>target){
17              return binary_search(nums, lower, middle-1, target);
18        }
19        else {
20              return binary_search(nums, middle+1, upper, target);
21        }
22
23  }
24
25  public:
26
27        int searchInsert(vector<int>& nums, int target){
28              // 回傳二分搜尋的結果
29              return binary_search(nums, 0, nums.size()-1, target);
30        }
31  };
```

找尋插入位置 Search Insert Position 的迴圈版本（較遞迴佔用記憶體空間少）

```
1   class Solution{
2   public:
3
4         int searchInsert(vector<int>& nums, int target){
5
6               int lower = 0;
7               int upper = nums.size()-1;
8
9               // 記錄要插入的位置
10              int position;
```

```
11
12              while(upper>=lower){
13
14                  int middle = lower+(upper-lower)/2;
15
16                  if (nums[middle]==target){
17                      return middle;
18                  }
19                  else if (nums[middle]>target){
20                      upper = middle-1;
21                      position = middle;
22                  }
23                  else {
24                      lower = middle+1;
25                      position = middle+1;
26                  }
27
28              } // end of while
29
30              // 沒有找到 target
31              return position;
32
33          } // end of searchInsert
34
35  }; // end of Solution
```

（2）LeetCode #278. 第一個錯誤版本 First Bad Version

A. 題目

你是一個產品經理，且正在帶領一個團隊開發新產品，但最新版本的產品無法通過品管測試。因為每個版本都是修改先前的版本而成，所以只要有一個版本發生錯誤，後面的每個版本也都會是錯的。

假設有 n 個版本 [1,2,...,n]，而你想找到第一個發生錯誤的版本。有一個 API 會根據傳入的版本回傳該版本是否錯誤：bool isBadVersion(version)。

設計一個函式來找到第一個錯誤的版本，並最小化呼叫該 API 的次數。

B. 出處
https://leetcode.com/problems/first-bad-version/

C. 範例輸入與輸出
輸入：n = 5（bad = 4）
輸出：4

呼叫 isBadVersion(3) -> false
呼叫 isBadVersion(5) -> true
呼叫 isBadVersion(4) -> true

發現第一個錯誤的版本是 4。

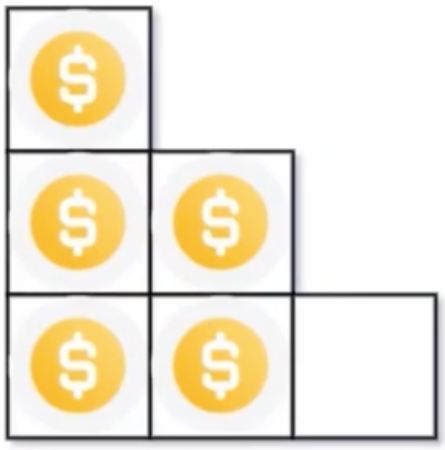
第一個錯誤版本 First Bad Version	
1	class Solution{
2	public:
3	int firstBadVersion(int n){
4	
5	int lower = 0;
6	int upper = n;
7	

```
8           // 使用二分搜尋
9           while (lower<upper){
10
11              int middle = lower + (upper-lower)/2;
12
13              // 檢查 middle 是否是錯誤的
14              bool bad = isBadVersion(middle);
15
16              // 如果 middle 是錯誤的，第一個錯誤版本在前面
17              if (bad==true){
18                  upper = middle-1;
19              }
20              // 如果 middle 是對的，第一個錯誤版本在後面
21              else {
22                  lower = middle+1;
23              }
24
25          } // end of while
26
27          // 回傳找到的第一個錯誤版本索引值 lower
28          return lower;
29
30      } // end of firstBadVersion
31
32  }; // end of Solution
```

（3）LeetCode #441. 安排硬幣 Arranging Coins

A. 題目

你有 n 個硬幣，且你想要用這些硬幣來排出一個階梯的形狀。這個階梯有 k 個橫列，第 i 個橫列剛好會有 i 個硬幣，最下面的那個橫列可以不用填滿。



給定一個整數 n，回傳該階梯完整的「橫列」數。

B. 出處

https://leetcode.com/problems/arranging-coins/

C. 範例輸入與輸出

輸入：n = 5

輸出：2

第三個橫列沒辦法填滿，所以回傳 2

D. 解題邏輯

應該回傳「k 個 row」的情形，最多可以放 $\frac{(k+1)[(k+1)+1]}{2}$ 個硬幣，因此：

$$\frac{k(k+1)}{2} < n < \frac{(k+1)[(k+1)+1]}{2}$$

安排硬幣 Arranging Coins

```
1  class Solution{
2
3  // 避免溢位，用函式計算 middle 層可以放的硬幣數
4  long long int sum(long long int middle){
5      return middle*(middle+1)/2;
6  }
7
8  public:
9
10     int arrangeCoins(int n){
11
12         int lower = 0;
13         // k 超過 2*sqrt(n) 時，k*(k+1)/2 一定超過 n
14         // 因此初始的上界 upper 可以取 2*sqrt(n)
15         // 若再向下取到 sqrt(2n) 會有邊界處理問題
16         int upper = 2*sqrt(n);
17
18         while (upper>lower){
19
20             int middle = lower+(upper-lower)/2;
21
22             // middle 層正好放的下 n 個硬幣
23             if (sum(middle)==n){
24                 return middle;
25             }
26
27             // middle 層可以放超過 n 個硬幣
28             else if (sum(middle)>n){
29                 upper = middle;
30             }
31
32             // middle 層放不下 n 個硬幣
33             else if (sum(middle)<n){
```

```
34                              lower = middle+1;
35                          }
36
37                  } // end of while
38
39                  return lower-1;
40
41          } // end of arrangeCoins
42
43  }; // end of Solution
```