Enumeration

Lecture by WiwiHo
Credit: 8e7, 李昕威, baluteshih, TreapKing, Ccucumber12

Sprout

開始之前

- 影片看了嗎？
- 作業寫了嗎？

Sprout

資訊之芽 Instagram 上線啦

大綱

- 枚舉
- 搜尋

Sprout

枚舉各種東西

Sprout

枚舉的物件

- 影片開頭有個枚舉回文的問題
  - 如果枚舉開頭和結尾，那麼有 $O(n^2)$ 個東西要枚舉，檢查一個東西的時間是 $O(n)$

枚舉的物件

- 影片開頭有個枚舉回文的問題
  - 如果枚舉開頭和結尾，那麼有 $O(n^2)$ 個東西要枚舉，檢查一個東西的時間是 $O(n)$
  - 但如果是枚舉回文中心再小到大枚舉長度，那就可以回文的特性，一邊枚舉長度一邊檢查是不是回文
  - 複雜度變成 $O(n^2)$

枚舉的物件

- 影片開頭有個枚舉回文的問題
  - 如果枚舉開頭和結尾，那麼有 $O(n^2)$ 個東西要枚舉，檢查一個東西的時間是 $O(n)$
  - 但如果是枚舉回文中心再小到大枚舉長度，那就可以回文的特性，一邊枚舉長度一邊檢查是不是回文
  - 複雜度變成 $O(n^2)$

枚舉的物件

- 影片開頭有個枚舉回文的問題
  - 如果枚舉開頭和結尾，那麼有 $O(n^2)$ 個東西要枚舉，檢查一個東西的時間是 $O(n)$
  - 但如果是枚舉回文中心再小到大枚舉長度，那就可以回文的特性，一邊枚舉長度一邊檢查是不是回文
  - 複雜度變成 $O(n^2)$
- 枚舉的精神
  - 利用問題的特性，找出好的**枚舉物件**與**枚舉順序**

枚舉的物件

- 剛才我們枚舉回文
- 被枚舉的物件可以奇形怪狀

Sprout

枚舉排列

Problem 枚舉排列

給你一個數字 N，請列出所有 $1 \sim N$ 的排列。

- 排列總共有 $N!$ 種

枚舉排列：決策樹

- 還記得影片裡樂透那題那個樹狀圖嗎？

枚舉排列：決策樹

- 還記得影片裡樂透那題那個樹狀圖嗎？
- 枚舉的過程可以視為是「不斷在做決策」

Sprout

枚舉排列：決策樹

- 還記得影片裡樂透那題那個樹狀圖嗎？
- 枚舉的過程可以視為是「不斷在做決策」
- 像是枚舉排列的過程就可以是不斷決定下一個數字要拿什麼

Sprout

枚舉排列：決策樹

- 還記得影片裡樂透那題那個樹狀圖嗎？
- 枚舉的過程可以視為是「不斷在做決策」
- 像是枚舉排列的過程就可以是不斷決定下一個數字要拿什麼
- 決策樹上每一個節點的子節點就是一個可能的下一步決策

Sprout

枚舉排列：決策樹

- 還記得影片裡樂透那題那個樹狀圖嗎？
- 枚舉的過程可以視為是「不斷在做決策」
- 像是枚舉排列的過程就可以是不斷決定下一個數字要拿什麼
- 決策樹上每一個節點的子節點就是一個可能的下一步決策
- 所謂的枚舉就是走過所有可能的決策路線

枚舉排列

- 總之每一步在做決策時，我們先有了一些選好的數字，我們要選下一個數字

枚舉排列

- 總之每一步在做決策時，我們先有了一些選好的數字，我們要選下一個數字
- 因為我們現在是要排列，所以不能選到選過的數字

Sprout

枚舉排列

- 總之每一步在做決策時，我們先有了一些選好的數字，我們要選下一個數字
- 因為我們現在是要排列，所以不能選到選過的數字

```
int n;
vector<bool> use; // initialize to length n + 1
vector<int> now; // 目前排列的長相
void f(){
    if((int)now.size() == n){
        // output now
        return;
    }
    for(int i = 1; i <= n; i++){
        if(use[i]) continue; // 用過了就不能再用
        use[i] = true;
        now.push_back(i);
        f();
        now.pop_back();
        use[i] = false; // 記得改回來！
    }
}
```

枚舉子集

- 其實我們剛剛枚舉排列的過程就會枚舉到所有子集了

枚舉子集

- 其實我們剛剛枚舉排列的過程就會枚舉到所有子集了
- 不過排列的順序是重要的，子集的順序是不重要的

Sprout

枚舉子集

- 其實我們剛剛枚舉排列的過程就會枚舉到所有子集了
- 不過排列的順序是重要的，子集的順序是不重要的
- 只拿比上一個元素更大的元素，就可以避免同個子集枚舉到兩次

```
void f(int lst = 0){ // 上一個選的數字
    // output now
    for(int i = lst + 1; i <= n; i++){ // 只枚舉 lst+1..n 這些選項
        if(use[i]) continue; // 用過了就不能再用
        use[i] = true;
        now.push_back(i);
        f();
        now.pop_back();
        use[i] = false; // 記得改回來！
    }
}
```

迴圈式寫法

- 遞迴枚舉好像稍微有點麻煩……

迴圈式寫法

- 遞迴枚舉好像稍微有點麻煩……
- 一個 $0 \sim N-1$ 的子集 S 可以用一個整數 x 表示，x 的第 i 個 bit 是 1 就代表 $i \in S$

迴圈式寫法

- 遞迴枚舉好像稍微有點麻煩……
- 一個 $0 \sim N-1$ 的子集 S 可以用一個整數 x 表示，x 的第 i 個 bit 是 1 就代表 $i \in S$

```
for(int i = 0; i < (1 << n); i++){
    for(int j = 0; j < n; j++)
        if(1 << j & i) cout << j + 1 << " ";
    cout << "\n";
}
```

- 那是不是也可以把排列用一個整數表示啊，像是用字典序當編號 (?)

迴圈式寫法

- 遞迴枚舉好像稍微有點麻煩……
- 一個 $0 \sim N-1$ 的子集 S 可以用一個整數 x 表示，x 的第 i 個 bit 是 1 就代表 $i \in S$

```
for(int i = 0; i < (1 << n); i++){
    for(int j = 0; j < n; j++)
        if(1 << j & i) cout << j + 1 << " ";
    cout << "\n";
}
```

- 那是不是也可以把排列用一個整數表示啊，像是用字典序當編號 (?)
- 可以是可以，但 STL 有好用 function

```
vector<int> now(n);
iota(now.begin(), now.end(), 1);
do{
    // output now
}
while(next_permutation(now.begin(), now.end()));
```

遞迴 vs 迴圈

- 既然有那麼簡單的寫法，那幹嘛不都用迴圈就好了 (?)

Sprout

遞迴 vs 迴圈

- 既然有那麼簡單的寫法，那幹嘛不都用迴圈就好了 (?)
- 有時候遞迴枚舉的過程可以幫你做到一些事情

Sprout

遞迴 vs 迴圈

- 既然有那麼簡單的寫法，那幹嘛不都用迴圈就好了 (?)
- 有時候遞迴枚舉的過程可以幫你做到一些事情

Problem 枚舉大子集

有一個數列 $a_1, a_2, \ldots, a_N$，列出所有 $1 \sim N$ 的集合 S，滿足 $\sum_{i \in S} a_i \geq K$。

- 如果寫一個枚舉 $i = 1 \sim 2^N - 1$ 的迴圈，然後在裡面花 $O(N)$ 時間算出子集的元素總和，那時間複雜度是 $O(2^N N)$
- 用遞迴呢？

遞迴 vs 迴圈：枚舉大子集

- 我們可以另外維護一個數，記錄目前總和，遞迴枚舉的過程可以順便維護它

遞迴 vs 迴圈：枚舉大子集

- 我們可以另外維護一個數，記錄目前總和，遞迴枚舉的過程可以順便維護它
- 這樣時間複雜度就是遞迴函數被 call 了幾次，也就是子集合的數量
- $O(2^N)$

遞迴 vs 迴圈：排列

- 那在枚舉排列的時候有一樣的效果嗎？

Sprout

遞迴 vs 迴圈：排列

- 那在枚舉排列的時候有一樣的效果嗎？
- 如果是用剛剛的寫法，答案是**沒有**，因為每次選下一個數的時候我們都會枚舉過全部的 $1 \sim N$

遞迴 vs 迴圈：排列

- 那在枚舉排列的時候有一樣的效果嗎？
- 如果是用剛剛的寫法，答案是**沒有**，因為每次選下一個數的時候我們都會枚舉過全部的 $1 \sim N$
- 只有跑到長度 $= N$ 時不用再枚舉下去，所以光是枚舉部分的時間就是

$$O\left(NP_1^N + NP_2^N + \cdots + NP_{N-1}^N + P_N^N\right) = O\left(N \cdot \frac{N}{(N-1)!} + \cdots + N \cdot \frac{N!}{1!} + N!\right)$$

這個東西是 $O(N \times N!)$

遞迴 vs 迴圈：排列

- 那在枚舉排列的時候有一樣的效果嗎？
- 如果是用剛剛的寫法，答案是**沒有**，因為每次選下一個數的時候我們都會枚舉過全部的 $1 \sim N$
- 只有跑到長度 $= N$ 時不用再枚舉下去，所以光是枚舉部分的時間就是

$$O\left(NP_1^N + NP_2^N + \cdots + NP_{N-1}^N + P_N^N\right) = O\left(N \cdot \frac{N}{(N-1)!} + \cdots + N \cdot \frac{N!}{1!} + N!\right)$$

  這個東西是 $O(N \times N!)$
- 為什麼枚舉子集的就可以少一個 N？因為枚舉子集的時候並不會枚舉到那些用過的東西
  - 還記得我們是從 $lst + 1$ 開始枚舉的嗎？

遞迴 vs 迴圈：排列

- 事實上可以用個 linked list 維護沒用過的數字，這樣就可以讓過程變 $O(N!)$

Sprout

遞迴 vs 迴圈：排列

- 事實上可以用個 linked list 維護沒用過的數字，這樣就可以讓過程變 $O(N!)$
- 喔也太麻煩了吧，那 `next_permutation` 的複雜度是什麼？

遞迴 vs 迴圈：排列

- 事實上可以用個 linked list 維護沒用過的數字，這樣就可以讓過程變 $O(N!)$
- 喔也太麻煩了吧，那 `next_permutation` 的複雜度是什麼？
- 如果你用 `next_permutation` 枚舉過一次全部排列，總時間複雜度是 $O(N!)$

遞迴 vs 迴圈：排列

- 事實上可以用個 linked list 維護沒用過的數字，這樣就可以讓過程變 $O(N!)$
- 喔也太麻煩了吧，那 `next_permutation` 的複雜度是什麼？
- 如果你用 `next_permutation` 枚舉過一次全部排列，總時間複雜度是 $O(N!)$
- 他找到下一個排列的方法其實非常簡單：找到最長遞減後綴 $a_{i+1}, \ldots, a_N$，然後找到 $> a_i$ 的最小元素 a_j，交換 a_i, a_j 後把 $a_{i+1}, \ldots, a_N$ 反轉

遞迴 vs 迴圈：排列

- 事實上可以用個 linked list 維護沒用過的數字，這樣就可以讓過程變 $O(N!)$
- 喔也太麻煩了吧，那 `next_permutation` 的複雜度是什麼？
- 如果你用 `next_permutation` 枚舉過一次全部排列，總時間複雜度是 $O(N!)$
- 他找到下一個排列的方法其實非常簡單：找到最長遞減後綴 $a_{i+1}, \ldots, a_N$，然後找到 $> a_i$ 的最小元素 a_j，交換 a_i, a_j 後把 $a_{i+1}, \ldots, a_N$ 反轉
- 時間複雜度就是所有排列的最長遞減後綴長度總和
- 第 i 個位置在最長遞減後綴裡的條件是它和它後面都遞減，所以它在 P_{i-1}^N 個排列裡會被算到

$$P_0^N + P_1^N + \cdots + P_{N-1}^N = \frac{N!}{N!} + \frac{N!}{(N-1)!} + \cdots + \frac{N!}{1!}$$

這個東西是 $O(N!)$

枚舉與減枝

Problem

八皇后問題 給定一個西洋棋盤，要在上面放八個皇后。皇后的攻擊範圍為上下左右與四個對角線。請輸出一個放法使的任兩個皇后之間不會互相攻擊。

- 要枚舉什麼？

枚舉與減枝

Problem

八皇后問題 給定一個西洋棋盤，要在上面放八個皇后。皇后的攻擊範圍為上下左右與四個對角線。請輸出一個放法使的任兩個皇后之間不會互相攻擊。

- 要枚舉什麼？
- 每個格子放或不放

Sprout

枚舉與減枝

Problem

八皇后問題 給定一個西洋棋盤，要在上面放八個皇后。皇后的攻擊範圍為上下左右與四個對角線。請輸出一個放法使的任兩個皇后之間不會互相攻擊。

- 要枚舉什麼？
- 每個格子放或不放
- 一橫排只能放一個皇后！下一個位子只要枚舉下一排的就好

枚舉與減枝

Problem

八皇后問題 給定一個西洋棋盤，要在上面放八個皇后。皇后的攻擊範圍為上下左右與四個對角線。請輸出一個放法使的任兩個皇后之間不會互相攻擊。

- 要枚舉什麼？
- 每個格子放或不放
- 一橫排只能放一個皇后！下一個位子只要枚舉下一排的就好
- 一直排只能放一個皇后！不要放在已經用過的直排上

枚舉與減枝

Problem

八皇后問題 給定一個西洋棋盤，要在上面放八個皇后。皇后的攻擊範圍為上下左右與四個對角線。請輸出一個放法使的任兩個皇后之間不會互相攻擊。

- 要枚舉什麼？
- 每個格子放或不放
- 一橫排只能放一個皇后！下一個位子只要枚舉下一排的就好
- 一直排只能放一個皇后！不要放在已經用過的直排上
- 一個對角線只能放一個皇后！不要放在用過的對角線上（how??）
  - $x+y, x-y$

枚舉：小結

- 在比較複雜的題目，枚舉是一種**透過固定某些東西，減少我們需要考慮的事情**的方法
- 所以我們需要想想要固定什麼，才能讓問題變好做
- 同時，如何避免浪費時間枚舉不重要的東西也是一個重點
- 甚至，用好的順序枚舉也會對枚舉之後要做的問題有幫助

搜尋

Sprout

二分搜的實作

- 影片上有個丟雞蛋問題，有一個藏起來的數字 x，如果你在 $> x$ 樓丟雞蛋，蛋就會破掉，$\leq x$ 樓就不會，求 x

二分搜的實作

- 影片上有個丟雞蛋問題，有一個藏起來的數字 x，如果你在 $> x$ 樓丟雞蛋，蛋就會破掉，$\leq x$ 樓就不會，求 x
- 可以想成蛋會不會破是一個函數 $f(i) = 0$ 代表不會，反之則會

二分搜的實作

- 影片上有個丟雞蛋問題，有一個藏起來的數字 x，如果你在 $> x$ 樓丟雞蛋，蛋就會破掉，$\leq x$ 樓就不會，求 x
- 可以想成蛋會不會破是一個函數 $f(i) = 0$ 代表不會，反之則會

i	1	2	$\ldots$	$x-1$	x	$x+1$	$x+2$	$\ldots$	2147483647
$f(i)$	0	0	$\ldots$	0	0	1	1	$\ldots$	1

二分搜的實作

- 影片上有個丟雞蛋問題，有一個藏起來的數字 x，如果你在 $> x$ 樓丟雞蛋，蛋就會破掉，$\leq x$ 樓就不會，求 x
- 可以想成蛋會不會破是一個函數 $f(i) = 0$ 代表不會，反之則會

i	1	2	...	$x-1$	x	$x+1$	$x+2$	...	2147483647
$f(i)$	0	0	...	0	0	1	1	...	1

- 左邊都是 0，右邊都是 1，我們要找的答案是最後一個 0 的位置

二分搜的實作：左閉右閉

$$0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1$$

- 我們維護一個「已知答案範圍」$[l, r]$，代表我們知道答案一定在 $[l, r]$ 這個範圍裡

二分搜的實作：左閉右閉

$$0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1$$

- 我們維護一個「已知答案範圍」$[l, r]$，代表我們知道答案一定在 $[l, r]$ 這個範圍裡
- 取 mid 是 $[l, r]$ 之中的某個位置

二分搜的實作：左閉右閉

$$0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1$$

- 我們維護一個「已知答案範圍」$[l, r]$，代表我們知道答案一定在 $[l, r]$ 這個範圍裡
- 取 mid 是 $[l, r]$ 之中的某個位置
- 如果 $f(mid) = 0$，我們知道答案**是** mid **或更右邊的人**，所以 $l \leftarrow mid$

二分搜的實作：左閉右閉

$$0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1$$

- 我們維護一個「已知答案範圍」$[l, r]$，代表我們知道答案一定在 $[l, r]$ 這個範圍裡
- 取 mid 是 $[l, r]$ 之中的某個位置
- 如果 $f(mid) = 0$，我們知道答案**是** mid **或更右邊的人**，所以 $l \leftarrow mid$
- 如果 $f(mid) = 1$，我們知道答案**只能是** mid **左邊的人（絕不會是** mid **!）**，所以 $r \leftarrow mid - 1$

二分搜的實作：左閉右閉

$$0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1$$

- 我們維護一個「已知答案範圍」$[l, r]$，代表我們知道答案一定在 $[l, r]$ 這個範圍裡
- 取 mid 是 $[l, r]$ 之中的某個位置
- 如果 $f(mid) = 0$，我們知道答案**是** mid **或更右邊的人**，所以 $l \leftarrow mid$
- 如果 $f(mid) = 1$，我們知道答案**只能是** mid **左邊的人（絕不會是** mid **!）**，所以 $r \leftarrow mid - 1$
- 最後 $l = r$ 的時候就找到答案了

二分搜的實作：左閉右閉

$$
\begin{aligned}
l &\leftarrow mid && \text{if } f(mid) = 0 \\
r &\leftarrow mid - 1 && \text{if } f(mid) = 1
\end{aligned}
$$

- 所以 mid 要選誰？
- 我們要避免陷入無窮迴圈，所以要 $mid \neq l$ 且 $mid - 1 \neq r$
  - $mid \leq r$ 所以 $mid - 1$ 絕不是 r
  - mid 不能是 l

二分搜的實作：左閉右閉

$$
\begin{aligned}
l &\leftarrow mid && \text{if } f(mid) = 0 \\
r &\leftarrow mid - 1 && \text{if } f(mid) = 1
\end{aligned}
$$

- 所以 mid 要選誰？
- 我們要避免陷入無窮迴圈，所以要 $mid \neq l$ 且 $mid - 1 \neq r$
  - $mid \leq r$ 所以 $mid - 1$ 絕不是 r
  - mid 不能是 l
- 如果要在 $\lfloor \frac{l+r}{2} \rfloor$ 和 $\lceil \frac{l+r}{2} \rceil$ 挑一個，那要選的上高斯
  - 直覺來說，靠右一點就不會選到 l
  - 想想 $r = l + 1$ 的狀況

二分搜的實作：左閉右閉

如果今天我想要找第一個 1 呢？那就變成

$$\begin{aligned} l &\leftarrow mid + 1 && \text{if } f(mid) = 0 \\ r &\leftarrow mid && \text{if } f(mid) = 1 \end{aligned}$$

- 變成 mid 不能是 r 了！所以要選靠左的，也就是下高斯

二分搜的實作：左閉右閉

- 會不會根本就沒有 0 或 1 啊？

二分搜的實作：左閉右閉

- 會不會根本就沒有 0 或 1 啊？
- 你可以直接假裝最左邊多一個 0，最右邊多一個 1

Sprout

二分搜的實作：左閉右閉

- 會不會根本就沒有 0 或 1 啊？
- 你可以直接假裝最左邊多一個 0，最右邊多一個 1
- 可以寫 $f(i)$ 的時候特判一下戳到最左或最右是 0 和 1
- 或者
  - 找最後一個 0 的時候，只在最左邊多加一個 0
  - 找第一個 1 的時候，只在最右邊多加一個 1

  就不用特判，多加的那格永遠問不到
  - 就是你要找什麼就把那個多放一個
  - 同時也是遠離 mid 的那一側多加一個

二分搜的實作：半開半閉

- 喔好麻煩喔，怎麼還要分 case QQ

二分搜的實作：半開半閉

- 喔好麻煩喔，怎麼還要分 case QQ
- 現在我們改成，一樣記錄 l, r 兩個位置，但是 l 指向我們**所知的**最後一個 0，r 指向我們所知的第一個 1

二分搜的實作：半開半閉

- 喔好麻煩喔，怎麼還要分 case QQ
- 現在我們改成，一樣記錄 l, r 兩個位置，但是 l 指向我們**所知的**最後一個 0，r 指向我們所知的第一個 1
- 所以世界長這樣

$$\underset{}{0}\ \underset{}{0}\ \underset{}{0}\ \underset{}{0}\ \underset{}{0}\ \underset{}{0}\ \underset{l}{0}\ \underset{}{?}\ \underset{}{?}\ \underset{}{?}\ \underset{}{?}\ \underset{}{?}\ \underset{}{?}\ \underset{}{?}\ \underset{}{?}\ \underset{r}{1}\ \underset{}{1}\ \underset{}{1}\ \underset{}{1}\ \underset{}{1}$$

二分搜的實作：半開半閉

- 喔好麻煩喔，怎麼還要分 case QQ
- 現在我們改成，一樣記錄 l, r 兩個位置，但是 l 指向我們**所知的**最後一個 0，r 指向我們所知的第一個 1
- 所以世界長這樣

$$\underset{}{0}\ \underset{}{0}\ \underset{}{0}\ \underset{}{0}\ \underset{}{0}\ \underset{}{0}\ \underset{l}{0}\ ?\ ?\ ?\ ?\ ?\ ?\ ?\ ?\ \underset{r}{1}\ 1\ 1\ 1\ 1$$

- mid 是 0 就把 l 改成 mid，反之把 r 改成 mid

二分搜的實作：半開半閉

- 喔好麻煩喔，怎麼還要分 case QQ
- 現在我們改成，一樣記錄 l, r 兩個位置，但是 l 指向我們**所知的**最後一個 0，r 指向我們所知的第一個 1
- 所以世界長這樣

$$\underset{}{0}\ \underset{}{0}\ \underset{}{0}\ \underset{}{0}\ \underset{}{0}\ \underset{}{0}\ \underset{l}{0}\ ?\ ?\ ?\ ?\ ?\ ?\ ?\ ?\ \underset{r}{1}\ 1\ 1\ 1\ 1$$

- mid 是 0 就把 l 改成 mid，反之把 r 改成 mid
- 最後的終止條件是 $r = l + 1$

二分搜的實作：半開半閉

- 喔好麻煩喔，怎麼還要分 case QQ
- 現在我們改成，一樣記錄 l, r 兩個位置，但是 l 指向我們**所知的**最後一個 0，r 指向我們所知的第一個 1
- 所以世界長這樣

$$\underset{}{0}\ \underset{}{0}\ \underset{}{0}\ \underset{}{0}\ \underset{}{0}\ \underset{}{0}\ \underset{l}{0}\ ?\ ?\ ?\ ?\ ?\ ?\ ?\ ?\ \underset{r}{1}\ 1\ 1\ 1\ 1$$

- mid 是 0 就把 l 改成 mid，反之把 r 改成 mid
- 最後的終止條件是 $r = l + 1$
- 你會發現 $r > l + 1$ 的時候 $\frac{l+r}{2}$ 無論取上還是下高斯都不是 l 也不是 r，所以就沒有剛剛無窮迴圈的問題了

二分搜的實作：半開半閉

- 喔好麻煩喔，怎麼還要分 case QQ
- 現在我們改成，一樣記錄 l, r 兩個位置，但是 l 指向我們**所知的**最後一個 0，r 指向我們所知的第一個 1
- 所以世界長這樣

$$\underset{}{0}\ \underset{}{0}\ \underset{}{0}\ \underset{}{0}\ \underset{}{0}\ \underset{}{0}\ \underset{l}{0}\ \underset{}{?}\ \underset{}{?}\ \underset{}{?}\ \underset{}{?}\ \underset{}{?}\ \underset{}{?}\ \underset{}{?}\ \underset{}{?}\ \underset{r}{1}\ \underset{}{1}\ \underset{}{1}\ \underset{}{1}\ \underset{}{1}$$

- mid 是 0 就把 l 改成 mid，反之把 r 改成 mid
- 最後的終止條件是 $r = l + 1$
- 你會發現 $r > l + 1$ 的時候 $\frac{l+r}{2}$ 無論取上還是下高斯都不是 l 也不是 r，所以就沒有剛剛無窮迴圈的問題了
- 最終 l 是最後一個 0，r 是第一個 1，取你要的當答案即可

二分搜的實作：半開半閉

- 喔好麻煩喔，怎麼還要分 case QQ
- 現在我們改成，一樣記錄 l, r 兩個位置，但是 l 指向我們**所知的**最後一個 0，r 指向我們所知的第一個 1
- 所以世界長這樣

$$\underset{l}{0}\ 0\ 0\ 0\ 0\ 0\ 0\ ?\ ?\ ?\ ?\ ?\ ?\ ?\ ?\ \underset{r}{1}\ 1\ 1\ 1\ 1$$

- mid 是 0 就把 l 改成 mid，反之把 r 改成 mid
- 最後的終止條件是 $r = l + 1$
- 你會發現 $r > l + 1$ 的時候 $\frac{l+r}{2}$ 無論取上還是下高斯都不是 l 也不是 r，所以就沒有剛剛無窮迴圈的問題了
- 最終 l 是最後一個 0，r 是第一個 1，取你要的當答案即可
- 要補 0,1 的話就兩側都補，那兩個位置都永遠不會被問到

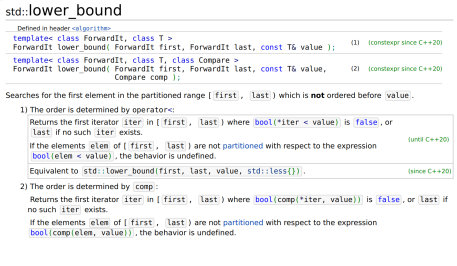
STL 裡的二分搜

- `lower_bound(l, r, val, comp)`, `upper_bound(l, r, val, comp)`
  - 預設的 `comp` = `less<>`

STL 裡的二分搜

- `lower_bound(l, r, val, comp)`, `upper_bound(l, r, val, comp)`
  - 預設的 `comp` = `less<>`
- 直白的說法：`lower_bound` 是第一個**不小於** `val` 的位置，`upper_bound` 是第一個**不小於等於**的位置
  - `lower_bound` 在找這個的第一個 1：$f(i) = \begin{cases} 0 & \text{if } v_i < \texttt{val} \\ 1 & \text{otherwise} \end{cases}$
  - `upper_bound` 在找這個的第一個 1：$f(i) = \begin{cases} 0 & \text{if } v_i \leq \texttt{val} \\ 1 & \text{otherwise} \end{cases}$

STL 裡的二分搜



std::lower_bound

Defined in header <algorithm>

```
template< class ForwardIt, class T >
ForwardIt lower_bound( ForwardIt first, ForwardIt last, const T& value );          (1)  (constexpr since C++20)

template< class ForwardIt, class T, class Compare >
ForwardIt lower_bound( ForwardIt first, ForwardIt last, const T& value,            (2)  (constexpr since C++20)
                       Compare comp );
```

Searches for the first element in the partitioned range [first , last) which is **not** ordered before value .

1) The order is determined by operator<:

Returns the first iterator iter in [first , last) where bool(*iter < value) is false , or last if no such iter exists. (until C++20)

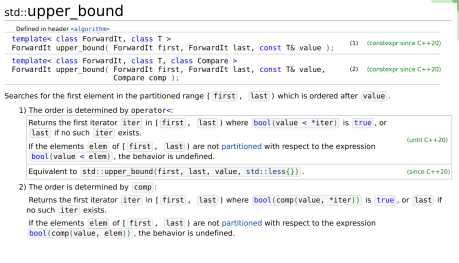
If the elements elem of [first , last) are not partitioned with respect to the expression bool(elem < value) , the behavior is undefined.

Equivalent to std::lower_bound(first, last, value, std::less{}) . (since C++20)

2) The order is determined by comp :

Returns the first iterator iter in [first , last) where bool(comp(*iter, value)) is false , or last if no such iter exists.

If the elements elem of [first , last) are not partitioned with respect to the expression bool(comp(elem, value)) , the behavior is undefined.



std::upper_bound

Defined in header <algorithm>

```
template< class ForwardIt, class T >
ForwardIt upper_bound( ForwardIt first, ForwardIt last, const T& value );          (1)  (constexpr since C++20)

template< class ForwardIt, class T, class Compare >
ForwardIt upper_bound( ForwardIt first, ForwardIt last, const T& value,            (2)  (constexpr since C++20)
                       Compare comp );
```

Searches for the first element in the partitioned range [first , last) which is ordered after value .

1) The order is determined by operator<:

Returns the first iterator iter in [first , last) where bool(value < *iter) is true , or last if no such iter exists. (until C++20)

If the elements elem of [first , last) are not partitioned with respect to the expression bool(value < elem) , the behavior is undefined.

Equivalent to std::upper_bound(first, last, value, std::less{}) . (since C++20)

2) The order is determined by comp :

Returns the first iterator iter in [first , last) where bool(comp(value, *iter)) is true , or last if no such iter exists.

If the elements elem of [first , last) are not partitioned with respect to the expression bool(comp(value, elem)) , the behavior is undefined.

- `lower_bound` 在找這個的第一個 1：$f(i) = \begin{cases} 0 & \text{if comp}(v_i, \text{val}) \\ 1 & \text{otherwise} \end{cases}$
- `upper_bound` 在找這個的第一個 1：$f(i) = \begin{cases} 1 & \text{if comp}(\text{val}, v_i) \\ 0 & \text{otherwise} \end{cases}$
- 記得 set/map 有它自己的 lower/upper bound

例題：收費站設置

Problem 收費站設置

在一條直線的高速公路上有 n 個可以放收費站的點，位置分別是 $a_1, a_2, \ldots, a_n$。你要架設 k 個收費站，而你希望讓任兩個收費站的間距遠一點。具體來說，你要讓 $\min(a_{i+1} - a_i)$ 盡量大。請輸出這個最大可能的數值。

- $n \leq 10^5, 0 \leq a_i \leq 10^9$

例題：收費站設置

- 要枚舉什麼？

Sprout

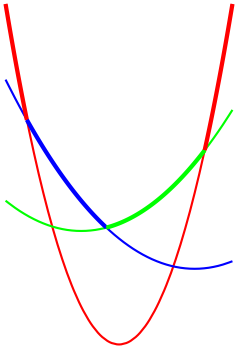
例題：收費站設置

- 要枚舉什麼？
  - 答案？每個站**至少**要距離多遠
- 如果某個距離 x 可以是答案，那**更小的** x 也都可以是答案

例題：收費站設置

- 要枚舉什麼？
  - 答案？每個站**至少**要距離多遠
- 如果某個距離 x 可以是答案，那**更小的** x 也都可以是答案
- $f(x) = 0$ 代表 x 可以是答案，找最後一個 0

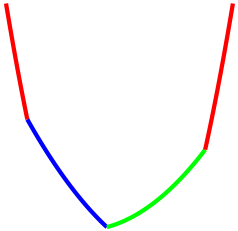
三分搜



- 記得影片裡那個一堆 U 函數取 max，找這個的最小值的問題嗎？
- 三分搜的對象：**單峰函數**

三分搜

- 影片上列出的一堆 case：
  - Case 1: $S(a) < S(b) < S(c) < S(d)$：最低點一定不在最右邊
  - Case 2: $S(a) > S(b) < S(c) < S(d)$：最低點一定不在最右邊
  - Case 3: $S(a) > S(b) > S(c) < S(d)$：最低點一定不在最左邊
  - Case 4: $S(a) > S(b) > S(c) > S(d)$：最低點一定不在最左邊
- 看中間那兩項就好了
- ~~他根本就沒有列出全部 case~~



三分搜：單峰函數

- (以下假設是找最小值，找最大值的就自己把遞增遞減倒過來)
- 單峰函數是什麼意思？中間高兩邊低？
- 更精準地說，是**最小值左邊遞減，右邊遞增**的函數

三分搜：單峰函數

- (以下假設是找最小值，找最大值的就自己把遞增遞減倒過來)
- 單峰函數是什麼意思？中間高兩邊低？
- 更精準地說，是**最小值左邊遞減，右邊遞增**的函數
- 遞減遞增？可以水平線嗎

Sprout

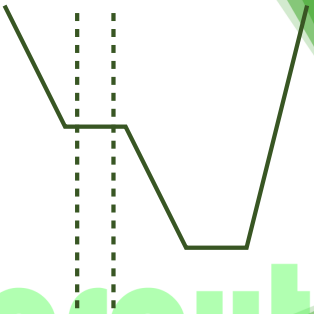
三分搜：單峰函數

- (以下假設是找最小值，找最大值的就自己把遞增遞減倒過來)
- 單峰函數是什麼意思？中間高兩邊低？
- 更精準地說，是**最小值左邊遞減，右邊遞增**的函數
- 遞減遞增？可以水平線嗎
- 答案是不太行
  - 廢話要是可以的話我幹嘛特別提出來
  - 那為什麼不行呢？不太行又是怎麼一回事？

Sprout

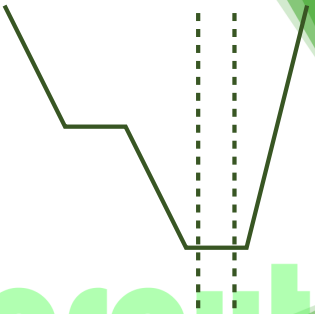
三分搜：水平線

- 那兩刀切在這就不知道要丟哪邊了 QQ



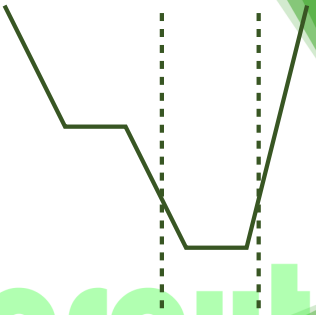
三分搜：水平線

- 那兩刀切在這就不知道要丟哪邊了 QQ
- 但要是極值發生的地方是一段水平線，那切在那裡是可以的



三分搜：水平線

- 那兩刀切在這就不知道要丟哪邊了 QQ
- 但要是極值發生的地方是一段水平線，那切在那裡是可以的
- 注意到如果兩刀切的地方不在同一段水平線，但剛好是一樣的值，這樣不管丟掉哪邊都可以
  - 和兩刀都在極值那段水平線的狀況一樣



三分搜：水平線

- 那兩刀切在這就不知道要丟哪邊了 QQ
- 但要是極值發生的地方是一段水平線，那切在那裡是可以的
- 注意到如果兩刀切的地方不在同一段水平線，但剛好是一樣的值，這樣不管丟掉哪邊都可以
  - 和兩刀都在極值那段水平線的狀況一樣
- 結論是，只有極值那一段可以是水平線，其他地方都不行

三分搜：水平線

- 那兩刀切在這就不知道要丟哪邊了 QQ
- 但要是極值發生的地方是一段水平線，那切在那裡是可以的
- 注意到如果兩刀切的地方不在同一段水平線，但剛好是一樣的值，這樣不管丟掉哪邊都可以
  - 和兩刀都在極值那段水平線的狀況一樣
- 結論是，只有極值那一段可以是水平線，其他地方都不行
- 注意到二分搜的時候是可以有水平線的，因為我們永遠都知道要被丟掉的是哪一邊

三分搜：終止條件

- 三分搜什麼時候要停？
- 三分搜的對象往往是一個實數函數，所以不會有 $l = r$ 的一天

Sprout

三分搜：終止條件

- 三分搜什麼時候要停？
- 三分搜的對象往往是一個實數函數，所以不會有 $l = r$ 的一天
- 搜到 l, r 的差小於某個數值為止
  - 範圍大的時候會因為浮點數精度而達不到這個數值
- 搜固定次數
  - 搜 k 次後，範圍大小是本來的 $\left(\frac{2}{3}\right)^k$

整數三分搜

- 如果今天三分搜的對象是一個整數函數怎麼辦？

整數三分搜

- 如果今天三分搜的對象是一個整數函數怎麼辦？
- 假設**第一個**極值在 x
  - $\forall i \leq x,\ f(i) - f(i-1) < 0$
  - $\forall i > x,\ f(i) - f(i-1) \geq 0$

整數三分搜

- 如果今天三分搜的對象是一個整數函數怎麼辦？
- 假設**第一個**極值在 x
  - $\forall i \leq x,\ f(i) - f(i-1) < 0$
  - $\forall i > x,\ f(i) - f(i-1) \geq 0$
- 尋找最後一個 $f(i) - f(i-1) < 0$ 的 i
  - 變成二分搜了！

例題：最小包覆圓的限制情況

Problem 最小包覆圓的限制情況

給你二維平面上的 N 個點，求包覆所有點的最小圓半徑。特別的是，這個包覆圓必須恰與 x 軸交於一點。

- $N \leq 10^5$
- |座標範圍| $\leq 10^7$
- 所有點的 y 座標 > 0

例題：最小包覆圓的限制情況

- 半徑同時是圓心的 y 座標

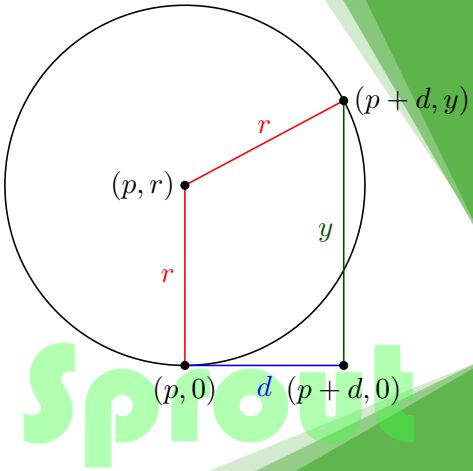
例題：最小包覆圓的限制情況

- 半徑同時是圓心的 y 座標
- 我們先試著枚舉圓心的 x 座標好了，然後來算算看半徑 r 至少要是多少

Sprout

例題：最小包覆圓的限制情況

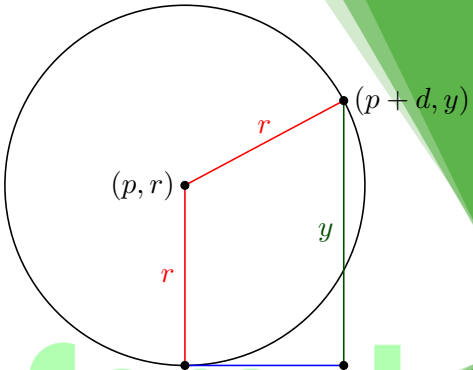
- 假設圓心 x 座標是 p
- 我們想知道它要蓋到 $(p+d, y)$ 這個點，y 座標兼半徑 r 至少要是多少



例題：最小包覆圓的限制情況

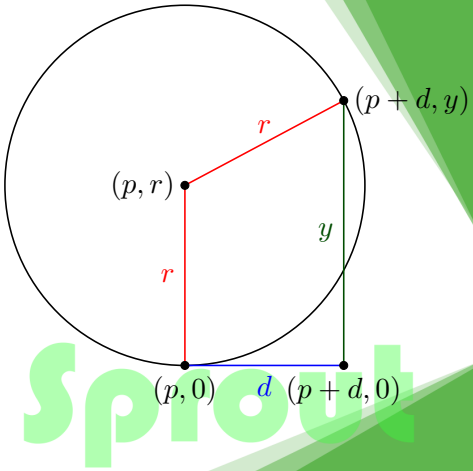
- 假設圓心 x 座標是 p
- 我們想知道它要蓋到 $(p+d, y)$ 這個點，y 座標兼半徑 r 至少要是多少

$$
\begin{aligned}
r^2 &= d^2 + (y-r)^2 \\
&= d^2 + y^2 + r^2 - 2yr \\
r &= \frac{d^2 + y^2}{2y}
\end{aligned}
$$



例題：最小包覆圓的限制情況

- 可是 p 可以有很多種耶 QQ

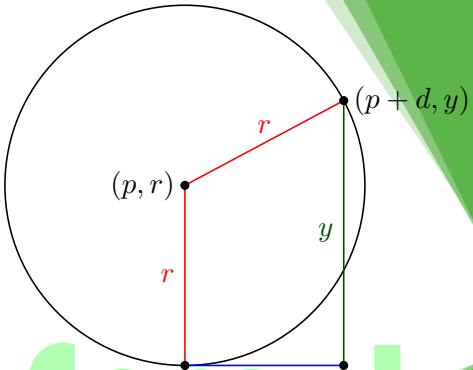


例題：最小包覆圓的限制情況

- 可是 p 可以有很多種耶 QQ
- 再看一次 r

$$r = \frac{(x-p)^2 + y^2}{2y}$$

它是一個對 p 的二次函數耶

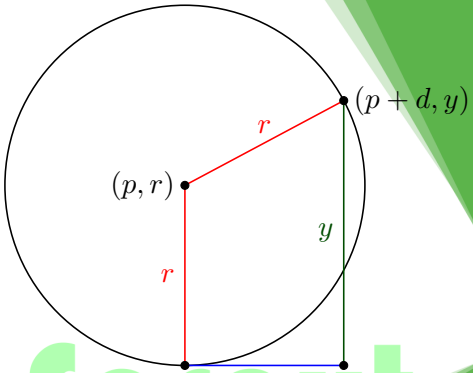


例題：最小包覆圓的限制情況

- 可是 p 可以有很多種耶 QQ
- 再看一次 r

$$r = \frac{(x-p)^2 + y^2}{2y}$$

  它是一個對 p 的二次函數耶
- 一群二次函數的 max 的最小值



例題：最小包覆圓的限制情況

- 還有什麼東西是我們可以枚舉的？

Sprout

例題：最小包覆圓的限制情況

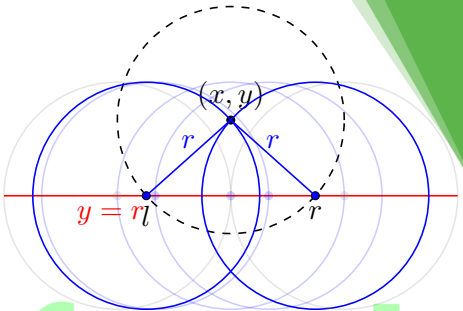
- 還有什麼東西是我們可以枚舉的？
- 能不能直接枚舉半徑 r？
  - 現在變成 y 座標是固定的，我們要找 x 座標

例題：最小包覆圓的限制情況

- 還有什麼東西是我們可以枚舉的？
- 能不能直接枚舉半徑 r？
  - 現在變成 y 座標是固定的，我們要找 x 座標
- 對於一個點 (x, y)，有哪些 x 座標可以把圓心擺在上面，使得這個圓蓋住它呢？

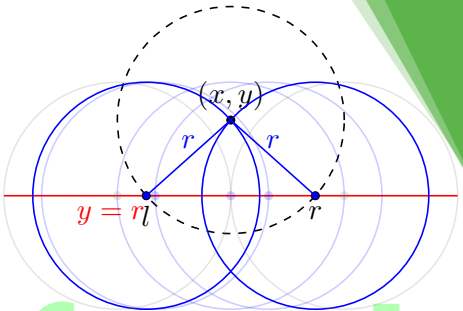
例題：最小包覆圓的限制情況

- l 到 r 中間的點都可以是圓心



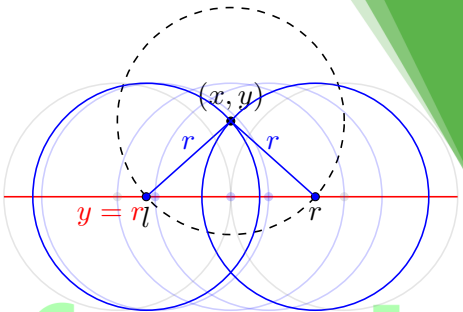
例題：最小包覆圓的限制情況

- l 到 r 中間的點都可以是圓心
- 算出每個點的這個範圍，只要這些範圍都有交集，那就可以選這個 r



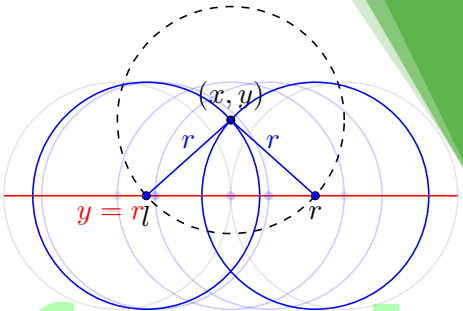
例題：最小包覆圓的限制情況

- l 到 r 中間的點都可以是圓心
- 算出每個點的這個範圍，只要這些範圍都有交集，那就可以選這個 r
- 越大的半徑越好



例題：最小包覆圓的限制情況

- l 到 r 中間的點都可以是圓心
- 算出每個點的這個範圍，只要這些範圍都有交集，那就可以選這個 r
- 越大的半徑越好
- 二分搜 r！



二分搜 vs 三分搜

- 二分搜做 k 次，範圍大小會是本來的 $\left(\frac{1}{2}\right)^k$
- 三分搜做 k 次，範圍大小會是本來的 $\left(\frac{2}{3}\right)^k$

二分搜 vs 三分搜

- 二分搜做 k 次，範圍大小會是本來的 $\left(\frac{1}{2}\right)^k$
- 三分搜做 k 次，範圍大小會是本來的 $\left(\frac{2}{3}\right)^k$
- 假設我們希望範圍縮小成本來的 $\frac{1}{C}$ 倍……
  - 二分搜大概要做 $\log_2 C$ 次
  - 三分搜大概要做 $\log_{\frac{3}{2}} C$ 次，別忘了三分搜每次搜其實都是 2 個詢問

二分搜 vs 三分搜

- 二分搜做 k 次，範圍大小會是本來的 $\left(\frac{1}{2}\right)^k$
- 三分搜做 k 次，範圍大小會是本來的 $\left(\frac{2}{3}\right)^k$
- 假設我們希望範圍縮小成本來的 $\frac{1}{C}$ 倍……
  - 二分搜大概要做 $\log_2 C$ 次
  - 三分搜大概要做 $\log_{\frac{3}{2}} C$ 次，別忘了三分搜每次搜其實都是 2 個詢問
- 二分搜總共要詢問 $\log_2 C$ 次
- 三分搜總共要詢問 $2\log_{\frac{3}{2}} C$ 次
  - 整數三分搜轉差分二分搜總共要詢問 $2\log_2 C$ 次

二分搜 vs 三分搜

- 二分搜做 k 次，範圍大小會是本來的 $\left(\frac{1}{2}\right)^k$
- 三分搜做 k 次，範圍大小會是本來的 $\left(\frac{2}{3}\right)^k$
- 假設我們希望範圍縮小成本來的 $\frac{1}{C}$ 倍……
  - 二分搜大概要做 $\log_2 C$ 次
  - 三分搜大概要做 $\log_{\frac{3}{2}} C$ 次，別忘了三分搜每次搜其實都是 2 個詢問
- 二分搜總共要詢問 $\log_2 C$ 次
- 三分搜總共要詢問 $2\log_{\frac{3}{2}} C$ 次
  - 整數三分搜轉差分二分搜總共要詢問 $2\log_2 C$ 次
- $2\log_{\frac{3}{2}} C \approx 3.419 \log_2 C$

二分搜：為什麼是二分？

- 為什麼二分搜是二分搜，不是三分四分五分十分？
- 二分搜的對象：0000011111 函數

Sprout

二分搜：為什麼是二分？

- 為什麼二分搜是二分搜，不是三分四分五分十分？
- 二分搜的對象：0000011111 函數
- 如果我們把搜尋範圍切成 k 半，然後我們的切割點是

$$l = a_0 \quad a_1 \quad a_2 \quad \dots \quad a_{k-2} \quad a_{k-1} \quad a_k = r$$

- 其中我們已經知道 $f(l) = 0, f(r) = 1$，我們會花 $k-1$ 次詢問來知道所有 a_i

二分搜：為什麼是二分？

- 為什麼二分搜是二分搜，不是三分四分五分十分？
- 二分搜的對象：0000011111 函數
- 如果我們把搜尋範圍切成 k 半，然後我們的切割點是

$$l = a_0 \quad a_1 \quad a_2 \quad \dots \quad a_{k-2} \quad a_{k-1} \quad a_k = r$$

- 其中我們已經知道 $f(l) = 0, f(r) = 1$，我們會花 $k-1$ 次詢問來知道所有 a_i
- 新的 l 是最後一個 i 滿足 $f(i) = 0$，新的 r 是第一個 i 滿足 $f(i) = 1$

二分搜：為什麼是二分？

- 為什麼二分搜是二分搜，不是三分四分五分十分？
- 二分搜的對象：0000011111 函數
- 如果我們把搜尋範圍切成 k 半，然後我們的切割點是

$$l = a_0 \quad a_1 \quad a_2 \quad \dots \quad a_{k-2} \quad a_{k-1} \quad a_k = r$$

- 其中我們已經知道 $f(l) = 0, f(r) = 1$，我們會花 $k-1$ 次詢問來知道所有 a_i
- 新的 l 是最後一個 i 滿足 $f(i) = 0$，新的 r 是第一個 i 滿足 $f(i) = 1$
- 區間大小變成本來的 $\frac{1}{k}$

二分搜：為什麼是二分？

- 為什麼二分搜是二分搜，不是三分四分五分十分？
- 二分搜的對象：0000011111 函數
- 如果我們把搜尋範圍切成 k 半，然後我們的切割點是

$$l = a_0 \quad a_1 \quad a_2 \quad \dots \quad a_{k-2} \quad a_{k-1} \quad a_k = r$$

- 其中我們已經知道 $f(l) = 0, f(r) = 1$，我們會花 $k-1$ 次詢問來知道所有 a_i
- 新的 l 是最後一個 i 滿足 $f(i) = 0$，新的 r 是第一個 i 滿足 $f(i) = 1$
- 區間大小變成本來的 $\frac{1}{k}$
- 搜尋次數是 $\log_k C$，詢問總次數是 $(k-1)\log_k C = \frac{k-1}{\log_2 k}\log_2 C$
  - 對於所有整數 $k > 2$，這都比二分搜還差

三分搜：為什麼是三分？

- 同樣的，假設我們要搜一個 U 函數的最小值，每次搜的時候切 k 段
- 只要還有兩個不是邊邊的切割點存在，你就可以比一比它們的大小，然後把側邊的某一段丟掉

三分搜：為什麼是三分？

- 同樣的，假設我們要搜一個 U 函數的最小值，每次搜的時候切 k 段
- 只要還有兩個不是邊邊的切割點存在，你就可以比一比它們的大小，然後把側邊的某一段丟掉
- 所以，我們一定可以找三個點 a_{i-1}, a_i, a_{i+1}，使得我們確定答案在 $[a_{i-1}, a_{i+1}]$ 之間
- 範圍大小變成本來的 $\frac{2}{k}$

三分搜：為什麼是三分？

- 同樣的，假設我們要搜一個 U 函數的最小值，每次搜的時候切 k 段
- 只要還有兩個不是邊邊的切割點存在，你就可以比一比它們的大小，然後把側邊的某一段丟掉
- 所以，我們一定可以找三個點 a_{i-1}, a_i, a_{i+1}，使得我們確定答案在 $[a_{i-1}, a_{i+1}]$ 之間
- 範圍大小變成本來的 $\frac{2}{k}$
- 搜尋次數是 $\log_{\frac{k}{2}} C$，總詢問次數是 $(k-1)\log_{\frac{k}{2}} C = \frac{k-1}{\log \frac{k}{2}} \log C$

三分搜：為什麼是三分？

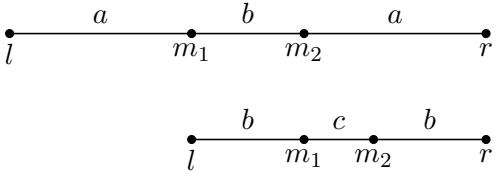
- 同樣的，假設我們要搜一個 U 函數的最小值，每次搜的時候切 k 段
- 只要還有兩個不是邊邊的切割點存在，你就可以比一比它們的大小，然後把側邊的某一段丟掉
- 所以，我們一定可以找三個點 a_{i-1}, a_i, a_{i+1}，使得我們確定答案在 $[a_{i-1}, a_{i+1}]$ 之間
- 範圍大小變成本來的 $\frac{2}{k}$
- 搜尋次數是 $\log_{\frac{k}{2}} C$，總詢問次數是 $(k-1)\log_{\frac{k}{2}} C = \frac{k-1}{\log \frac{k}{2}} \log C$
- 所以四分搜詢問次數比較少 XD

Sprout

三分搜：為什麼是三分？

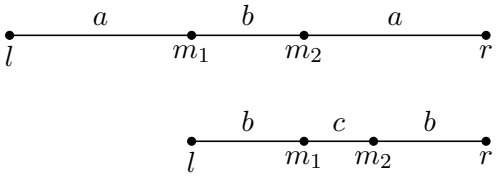
- 同樣的，假設我們要搜一個 U 函數的最小值，每次搜的時候切 k 段
- 只要還有兩個不是邊邊的切割點存在，你就可以比一比它們的大小，然後把側邊的某一段丟掉
- 所以，我們一定可以找三個點 a_{i-1}, a_i, a_{i+1}，使得我們確定答案在 $[a_{i-1}, a_{i+1}]$ 之間
- 範圍大小變成本來的 $\frac{2}{k}$
- 搜尋次數是 $\log_{\frac{k}{2}} C$，總詢問次數是 $(k-1)\log_{\frac{k}{2}} C = \frac{k-1}{\log \frac{k}{2}} \log C$
- 所以四分搜詢問次數比較少 XD
  - 甚至四分搜中間那個切點，前面其實已經問過了，所以真正的次數是 $2\log_2 C$

三分搜：到底誰比較好？



- 其實三分搜每次搜尋都要兩次詢問挺虧的，有沒有辦法像四分搜一樣，讓其中一個切割點剛好上次也是切割點啊？
- 從上面那張圖看起來就是可以

三分搜：到底誰比較好？



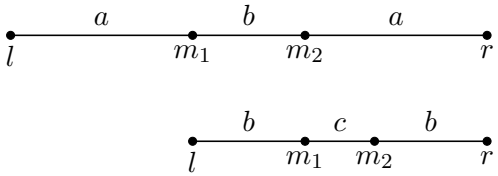
$$c+b=a$$

$$\frac{a}{b}=\frac{b}{c}$$

$$\frac{c+b}{b}=\frac{b}{c}$$

Sprout

三分搜：到底誰比較好？

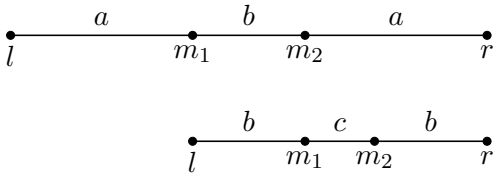


$$c+b=a$$
$$\frac{a}{b}=\frac{b}{c}$$
$$\frac{c+b}{b}=\frac{b}{c}$$

- $\frac{b}{c}=\frac{a}{b}=\varphi\approx 1.618$

Sprout

三分搜：到底誰比較好？



$$c+b=a$$

$$\frac{a}{b}=\frac{b}{c}$$

$$\frac{c+b}{b}=\frac{b}{c}$$

- $\frac{b}{c}=\frac{a}{b}=\varphi\approx 1.618$
- $\frac{a}{b}=\frac{a+b}{a}=\frac{2a+b}{a+b}$
- $\frac{a+b}{2a+b}=\frac{1}{\varphi}=\varphi-1\approx 0.618$

Sprout

黃金比例搜

- 這東西有個名字叫黃金比例搜（Golden ratio search）

黃金比例搜

- 這東西有個名字叫黃金比例搜（Golden ratio search）
- 每次搜尋完，範圍大小是本來的 $\frac{a+b}{2a+b} = \frac{1}{\varphi}$

黃金比例搜

- 這東西有個名字叫黃金比例搜（Golden ratio search）
- 每次搜尋完，範圍大小是本來的 $\frac{a+b}{2a+b} = \frac{1}{\varphi}$
- 搜尋次數是 $\log_{\varphi} C$，每次搜尋只要一次詢問

黃金比例搜

- 這東西有個名字叫黃金比例搜（Golden ratio search）
- 每次搜尋完，範圍大小是本來的 $\frac{a+b}{2a+b} = \frac{1}{\varphi}$
- 搜尋次數是 $\log_\varphi C$，每次搜尋只要一次詢問
- 總詢問次數是 $\log_\varphi C \approx 1.44 \log_2 C$
- 動動腦：搜整數可以用黃金比例搜嗎？

搜尋：小結

- 這些搜尋法其實和枚舉一樣，是為了**固定某個東西使得問題變好做**
- 只不過用這些特殊的搜尋法會更有效率地找到**這個固定的東西應該要是什麼才會讓答案最好**
- 黃金比例搜……沒有很重要，大多數時候還是直接寫正常的三分搜
- 如果是三分搜，兩個切割點都要詢問的話，其實兩個切割點越接近中間，讓每次縮小的比例接近 $\frac{1}{2}$ 是最好的
  - 但在不是整數時我們不這麼做是因為，距離太近可能會因為浮點數誤差比不出來，兩個切割點總是需要一個足夠大的安全距離，那不如直接三等分比較好寫