Enumeration & Search

Lecture by LittleCube

Credit: WiwiHo (2024, link)

Sprout

課程影片

Q & A?

Sprout

聰明的枚舉

Sprout

影片範例

影片開頭有個枚舉回文的問題：

- 枚舉開頭、結尾就會需要 $O(n^2)$，每次檢查 $O(n)$
- 枚舉中間跟長度：算過的東西可以繼續用，$O(n)$ 個開頭、全部長度只需要 $O(n)$

影片範例

影片開頭有個枚舉回文的問題：

- 枚舉開頭、結尾就會需要 $O(n^2)$，每次檢查 $O(n)$
- 枚舉中間跟長度：算過的東西可以繼續用，$O(n)$ 個開頭、全部長度只需要 $O(n)$
- 「要枚舉什麼」：我們要枚舉什麼？
- 「要怎麼枚舉」：什麼順序可以讓既有的資訊繼續用？

枚舉各種物件

Sprout

排列

枚舉排列：

Problem Enumerate Permutation

你有一個序列 $S = \{1, 2, \ldots, n\}$，列出他的所有排列

排列

這樣寫一定不行...

```
void enumeration(int n)
{
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            if (j != i)
                for (int k = 1; k <= n; k++)
                    if (k != i && k != j)
                        for(...)
                            ...
}
```

排列

非常經典的遞迴實作：

```
void enumeration(vector<int> current, vector<int> remain)
{
    if (remain.size() == 0ull)
        current;// 這裡就枚舉到一個排列了
    for (int i = 0; i < (int)remain.size(); i++)
    {
        current.emplace_back(remain[i]);
        remain.erase(remain.begin() + i);
        enumeration(current, remain);
        remain.insert(remain.begin() + i, current.back());
        current.pop_back();
    }
}
```

排列

- 枚舉下一個物件
- 更新、遞迴
- 回溯成原本的狀況

Sprout

排列

- 枚舉下一個物件
- 更新、遞迴
- 回溯成原本的狀況

$O(n^2 \cdot n!)$

排列

用一個陣列紀錄哪些已經用過了

```
void enumeration(vector<int> &used, vector<int> &perm)
{
    for (int i = 0; i < (int)used.size(); i++)
        if (!used[i])
        {
            used[i] = 1;
            perm.emplace_back(i);
            enumeration(used, perm);
            perm.pop_back();
            used[i] = 0;
        }
}
```

排列

用一個陣列紀錄哪些已經用過了

```
void enumeration(vector<int> &used, vector<int> &perm)
{
    for (int i = 0; i < (int)used.size(); i++)
        if (!used[i])
        {
            used[i] = 1;
            perm.emplace_back(i);
            enumeration(used, perm);
            perm.pop_back();
            used[i] = 0;
        }
}
```

$O(n \cdot n!)$

排列

- 輸出就要 $O(n \cdot n!)$，上面的方法其實相當夠用
- 有時候我們只需要處理相鄰的元素就可以知道某些性質的話，有沒有辦法拿掉一個 n？

排列

- 輸出就要 $O(n \cdot n!)$，上面的方法其實相當夠用
- 有時候我們只需要處理相鄰的元素就可以知道某些性質的話，有沒有辦法拿掉一個 n？
- 上面用一個 linked-list 維護就可以只枚舉剩下的元素！

排列

- 輸出就要 $O(n \cdot n!)$，上面的方法其實相當夠用
- 有時候我們只需要處理相鄰的元素就可以知道某些性質的話，有沒有辦法拿掉一個 n？
- 上面用一個 linked-list 維護就可以只枚舉剩下的元素！
- 可是這樣好麻煩喔，但我們有內建的 `next_permutation`

排列

```
void enumeration(int n)
{
    vector<int> perm(n);
    iota(perm.begin(), perm.end(), 1);
    do
    {
        // actually do something
    }
    while (next_permutation(perm.begin(), perm.end()));
}
```

排列

- `next_permutation` 會給下一個字典序的排列
- 如果字典序是最大的，回傳 `false`
- （通常）總共會花 $O(n!)$，每次最壞 $O(n)$

- 一般的實作：
- 找到最長（往前走）遞減的後綴 $a_{i+1}, \ldots, a_n$，
- 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

子集合

枚舉集合：

Problem Enumerate Subset

你有一個集合 $S = \{1, 2, \ldots, n\}$，列出所有他的子集合

子集合

- 我們當然可以遞迴！
- 同樣的想法：枚舉下一個是不是在集合內、然後遞迴

Sprout

子集合

```cpp
using namespace std;
void enumeration(vector<int> &cur, int i, int n)
{
    // cur 是其中一個子集
    for (int j = i; j < n; j++)
    {
        cur.emplace_back(j);
        enumeration(cur, j + 1, n);
        cur.pop_back();
    }
}
```

- 複雜度是 $O(n2^n)$？

子集合

```cpp
using namespace std;
void enumeration(vector<int> &cur, int i, int n)
{
    // cur 是其中一個子集
    for (int j = i; j < n; j++)
    {
        cur.emplace_back(j);
        enumeration(cur, j + 1, n);
        cur.pop_back();
    }
}
```

- 複雜度是 $O(n2^n)$？
- 沒錯，但也是 $O(2^n)$

子集合

- 想法：每個元素在不在可以用 0 1 表示
- $\{5, 3, 1, 0\}$ 可以寫成 101011
- 如果 n 其實不大，可以用整數的 bit 表示！

子集合

```
using namespace std;
void enumeration(int n)
{
    for (int mask = 0; mask < (1 << n); mask++)
    {
        vector<int> subset;
        for (int i = 0; i < n; i++)
            if (mask & (1 << i))
                subset.emplace_back(i);
    }
}
```

- 這樣要花 $O(n2^n)$
- 小心 bit 是 0-indexed！

Sprout

子集合

枚舉子集合：

Problem Enumerate Subset

你有一個集合 $S = \{1, 2, \ldots, n\}$，列出所有他的子集合的子集合

子集合

```
using namespace std;
void enumeration(int n)
{
    for (int mask = 0; mask < (1 << n); mask++)
        for (int sub = mask; sub < (1 << n); sub++)
            if ((sub & mask) == sub)
            {
                // actually do something
            }
}
```

- 枚舉兩次子集
- 集合 AND 表示交集、OR 表示聯集

Sprout

子集合

```
using namespace std;
void enumeration(int n)
{
    for (int mask = 0; mask < (1 << n); mask++)
        for (int sub = mask; sub < (1 << n); sub++)
            if ((sub & mask) == sub)
            {
                // actually do something
            }
}
```

- 枚舉兩次子集
- 集合 AND 表示交集、OR 表示聯集
- 要花 $O(4^n)$ 跑外面兩個迴圈

Sprout

子集合

- 可是所有 (集合, 子集合) 其實只有 3^n 個
- 每個元素只有都不在、在外面、兩個都在

子集合

- 可是所有 (集合, 子集合) 其實只有 3^n 個
- 每個元素只有都不在、在外面、兩個都在
- 有一個方式是先枚舉裡面的，再枚舉沒有在子集合內的子集合
- 可是直接寫有點小麻煩

子集合

- 很酷的實作方法

```
using namespace std;
void enumeration(int n)
{
    for (int mask = 0; mask < (1 << n); mask++)
        for (int sub = mask;; sub = (sub - 1) & mask)
        {
            // actually do something
            if (sub == 0) break;
        }
}
```

子集合

```
mask            0101110101
sub             0001110000
sub - 1         0001101111
sub - 1 & mask  0001100101
```

子集合

```
mask             0101110101
sub              0001110000
sub - 1          0001101111
sub - 1 & mask   0001100101
```

基本上可以想像成「只看 mask 有的地方，做二進位減法！」

搜索

Sprout

DFS vs BFS

在圖足夠小可以存起來的狀況

	BFS	DFS
時間複雜度	都是 $O(n+m)$	
空間複雜度	都是 $O(n+m)$	
實作	都差不多	
特點	可以找到最短路	用遞迴常數比較大

DFS vs BFS

在圖太大一定要探索的情況：假設 m 是最遠探索的距離，b 是每個節點可以往外擴張的數量

	BFS	DFS
時間複雜度	都是 $O(b^m)$	
空間複雜度	$O(b^m)$	$O(m)$
實作	都差不多	
特點	可以找到最短路	空間非常小！

IDDFS

- 想要有 DFS 的空間優勢，但又不想要不小心跑很久或找到次好的路徑

IDDFS

- 想要有 DFS 的空間優勢，但又不想要不小心跑很久或找到次好的路徑
- IDDFS (Iterative Deepening DFS)
- 維護允許的最深深度，遞迴就只可以這麼深
- 一開始最深深度是 0，找不到就 $+1$ 重跑一遍 DFS

IDDFS

- 想要有 DFS 的空間優勢，但又不想要不小心跑很久或找到次好的路徑
- IDDFS (Iterative Deepening DFS)
- 維護允許的最深深度，遞迴就只可以這麼深
- 一開始最深深度是 0，找不到就 $+1$ 重跑一遍 DFS
- 在一般的圖上面很慢，但是在搜索這種很多分支的圖，多一層新走的點會遠遠超過之前走過的點！

Bi-directional BFS

- 圖不至於太大，而且終點唯一（或是很少）

Bi-directional BFS

- 圖不至於太大，而且終點唯一（或是很少）
- 從起點跟終點一起 BFS，撞到的時候就是找到答案了！

Sprout

Bi-directional BFS

- 圖不至於太大，而且終點唯一（或是很少）
- 從起點跟終點一起 BFS，撞到的時候就是找到答案了！
- 時間複雜度是 $O(b^{m/2}) = O((\sqrt{b})^m)$

Bi-directional BFS

- 圖不至於太大，而且終點唯一（或是很少）
- 從起點跟終點一起 BFS，撞到的時候就是找到答案了！
- 時間複雜度是 $O(b^{m/2}) = O((\sqrt{b})^m)$

- 有點同樣精神的東西叫折半枚舉，但我們今天應該來不及講到

剪枝

Problem CSES Chessboard and Queens

有一個 8×8 的平板，中間有一些位置是不能放的。
有多少種方法可以放 8 隻皇后，彼此都不能互相攻擊？ 皇后可以攻擊在同一直、橫或斜排的所有格子。

剪枝

Problem CSES Chessboard and Queens

有一個 8×8 的平板，中間有一些位置是不能放的。
有多少種方法可以放 8 隻皇后，彼此都不能互相攻擊？ 皇后可以攻擊在同一直、橫或斜排的所有格子。

最慢的應該是全空的版面，先考慮這種狀況

剪枝

- 直接硬寫一個暴力，最後再檢查

Sprout

剪枝

- 直接硬寫一個暴力，最後再檢查
- 會需要跑到 $\binom{64}{8} = 4426165368 \approx 4 \times 10^9$ 種狀況

剪枝

- 直接硬寫一個暴力，最後再檢查
- 會需要跑到 $\binom{64}{8} = 4426165368 \approx 4 \times 10^9$ 種狀況
- 可是有些時候中間就沒救了，例如說不小心放在兩個一樣的橫排上面

剪枝

- 每個橫排只枚舉一個
- ```
  int col[N];
  void solve(int i)
  {
      if (i == N)
      {
          check(); // 檢查有沒有同直排或同斜排
          return;
      }
      for (int j = 0; j < N; j++)
      {
          col[i] = j;
          solve(i + 1);
      }
  }
  ```
  ```

剪枝

- 每個橫排只枚舉一個
- ```
  int col[N];
  void solve(int i)
  {
      if (i == N)
      {
          check(); // 檢查有沒有同直排或同斜排
          return;
      }
      for (int j = 0; j < N; j++)
      {
          col[i] = j;
          solve(i + 1);
      }
  }
  ```
- solve 被呼叫 $19173961 \approx 1.9 \times 10^7$ 次、check 被呼叫 $16777216 \approx 1.6 \times 10^7$ 次

Sprout
  ```

剪枝

- 每個橫排只枚舉一個，直排有遇到就跳過！
-
```
int x[N], y[N], col[N];
void solve(int i)
{
    if (i == N)
    {
        check(); // 檢查有沒有同斜排
        return;
    }
    for (int j = 0; j < N; j++)
        if (col[j] == 0)
        {
            x[i] = i, y[i] = j;
            col[j] = 1;
            solve(i + 1);
            col[j] = 0;
        }
}
```

剪枝

- 每個橫排只枚舉一個，直排有遇到就跳過！
- ```
  int x[N], y[N], col[N];
  void solve(int i)
  {
      if (i == N)
      {
          check(); // 檢查有沒有同斜排
          return;
      }
      for (int j = 0; j < N; j++)
          if (col[j] == 0)
          {
              x[i] = i, y[i] = j;
              col[j] = 1;
              solve(i + 1);
              col[j] = 0;
          }
  }
  ```
- solve 被呼叫 $109601 \approx 1.1 \times 10^5$ 次、check 被呼叫 40320 次
  ```

剪枝

- 每個橫排只枚舉一個，直排有遇到就跳過，斜排有遇到也跳過！
- ```
  int x[N], y[N], diag[N * 2], cdiag[N * 2], col[N];
  void solve(int i)
  {
      if (i == N)
      {
          check(); // 一定是合法的
      }
      for (int j = 0; j < N; j++)
          if (col[j] == 0 && diag[i + j] == 0 && cdiag[i - j + N] == 0)
          {
              x[i] = i, y[i] = j;
              col[j] = diag[i + j] = cdiag[i - j + N] = 1;
              solve(i + 1);
              col[j] = diag[i + j] = cdiag[i - j + N] = 0;
          }
  }
  ```
  ```

剪枝

- 每個橫排只枚舉一個，直排有遇到就跳過，斜排有遇到也跳過！
- ```
  int x[N], y[N], diag[N * 2], cdiag[N * 2], col[N];
  void solve(int i)
  {
      if (i == N)
      {
          check(); // 一定是合法的
      }
      for (int j = 0; j < N; j++)
          if (col[j] == 0 && diag[i + j] == 0 && cdiag[i - j + N] == 0)
          {
              x[i] = i, y[i] = j;
              col[j] = diag[i + j] = cdiag[i - j + N] = 1;
              solve(i + 1);
              col[j] = diag[i + j] = cdiag[i - j + N] = 0;
          }
  }
  ```
- solve 被呼叫 2057 次、check 被呼叫 92 次
  ```

剪枝

- 最後一份在我自己的電腦上可以跑到 13 皇后
- 但第一份 8 皇后就不行了

Sprout

剪枝

- 最後一份在我自己的電腦上可以跑到 13 皇后
- 但第一份 8 皇后就不行了
- 回到原本題目，其實只要多判現在格子是不是不能放就可以了

Sprout

剪枝

Problem CSES Grid Paths

有一個 7×7 的網格，你要繞整個格子一遍走過所有路徑
有一些步數你必須要往限制的方向走，請問你有多少種走法？（全空總共有 88418 種）

剪枝

- 如果走到剩下的位置不連通就可以直接停掉了！
- 但是，檢查連通要花很多時間
- 不如找一些比較簡單可以檢查的性質剪枝，每次遞迴花很多時間剪枝不見得會賺

Sprout

枚舉

- 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

Sprout

搜尋

Sprout

整數二分搜

- 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：$f(x) = 0$ 代表不會，反之 $f(x) = 1$ 則會

整數二分搜

- 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：$f(x) = 0$ 代表不會，反之 $f(x) = 1$ 則會
- 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**

整數二分搜

第一種實作想法

- 我們想要維護一個區間 $[l, r]$ 表示答案只會出現在 $[l, r]$ 裡面
- 戳一個點 m 會有這樣的狀況：

整數二分搜

第一種實作想法

- 我們想要維護一個區間 $[l, r]$ 表示答案只會出現在 $[l, r]$ 裡面
- 戳一個點 m 會有這樣的狀況：
  - 如果 m 是 0，那我們知道答案在右邊：$[m, r]$ 裡面
  - 反之，我們知道答案在嚴格左邊：$[l, m-1]$ 裡面

整數二分搜

第一種實作想法

- 我們想要維護一個區間 $[l, r]$ 表示答案只會出現在 $[l, r]$ 裡面
- 戳一個點 m 會有這樣的狀況：
  - 如果 m 是 0，那我們知道答案在右邊：$[m, r]$ 裡面
  - 反之，我們知道答案在嚴格左邊：$[l, m-1]$ 裡面
- 最後 $l = r$ 的時候就找到答案了

整數二分搜

- m 要選誰？

Sprout

整數二分搜

- m 要選誰？
- 如果 m 是 l，那第一種狀況的時候會陷入無窮迴圈！
- 另一邊沒有這個問題，$m = r$ 還是會讓區間變小

整數二分搜

- m 要選誰？
- 如果 m 是 l，那第一種狀況的時候會陷入無窮迴圈！
- 另一邊沒有這個問題，$m = r$ 還是會讓區間變小
- 我們還想要讓 m 越靠近中間越好，所以好想要是 $\lfloor \frac{l+r}{2} \rfloor$ 或 $\lceil \frac{l+r}{2} \rceil$

整數二分搜

- m 要選誰？
- 如果 m 是 l，那第一種狀況的時候會陷入無窮迴圈！
- 另一邊沒有這個問題，$m = r$ 還是會讓區間變小
- 我們還想要讓 m 越靠近中間越好，所以好想要是 $\lfloor \frac{l+r}{2} \rfloor$ 或 $\lceil \frac{l+r}{2} \rceil$
- 選 $\lceil \frac{l+r}{2} \rceil$ 就不會發生第一個狀況！
  - 直覺來說，靠右邊一點才不會選到 l
  - 你可以證明只有 $l + 1 = r$ 的時候這件事情才重要

整數二分搜

換成找**第一個 1 的位置**

- 戳一個點 m 會有這樣的狀況：

整數二分搜

換成找**第一個 1 的位置**

- 戳一個點 m 會有這樣的狀況：
  - 如果 m 是 1，那我們知道答案在左邊：$[l, m]$ 裡面
  - 反之，我們知道答案在嚴格右邊：$[m+1, r]$ 裡面

整數二分搜

換成找**第一個 1 的位置**

- 戳一個點 m 會有這樣的狀況：
  - 如果 m 是 1，那我們知道答案在左邊：$[l, m]$ 裡面
  - 反之，我們知道答案在嚴格右邊：$[m+1, r]$ 裡面
- m 是 r，那第一種狀況的時候會陷入無窮迴圈！

整數二分搜

換成找**第一個 1 的位置**

- 戳一個點 m 會有這樣的狀況：
  - 如果 m 是 1，那我們知道答案在左邊：$[l, m]$ 裡面
  - 反之，我們知道答案在嚴格右邊：$[m+1, r]$ 裡面
- m 是 r，那第一種狀況的時候會陷入無窮迴圈！
- 選 $\left\lfloor \frac{l+r}{2} \right\rfloor$ 就不會發生第一個狀況！

整數二分搜

- 會不會根本就沒有 0 或 1 啊？

Sprout

整數二分搜

- 會不會根本就沒有 0 或 1 啊？
- 找 0 的時候你可以假裝最左邊多一個 0

Sprout

整數二分搜

- 會不會根本就沒有 0 或 1 啊？
- 找 0 的時候你可以假裝最左邊多一個 0
- 剛好這個位置用上高斯一定不會要用到！

Sprout

整數二分搜

- 會不會根本就沒有 0 或 1 啊？
- 找 0 的時候你可以假裝最左邊多一個 0
- 剛好這個位置用上高斯一定不會要用到！
- 找 1 的時候你可以假裝最右邊多一個 1，也會有同樣的方便性質

Sprout

整數二分搜

- 會不會根本就沒有 0 或 1 啊？
- 找 0 的時候你可以假裝最左邊多一個 0
- 剛好這個位置用上高斯一定不會要用到！
- 找 1 的時候你可以假裝最右邊多一個 1，也會有同樣的方便性質
- **要特別注意有負數的時候上下高斯會不太一樣！**

整數二分搜

第二種實作想法

- 不想要分 case QQ

整數二分搜

第二種實作想法

- 不想要分 case QQ
- 換成我們想知道 **0 / 1 的分界點在哪裡**
- 讓 l 是我們所知的最後一個 0，r 是第一個 1
- 最後 $l = r + 1$ 的時候就找到答案了

整數二分搜

第二種實作想法

- 不想要分 case QQ
- 換成我們想知道 **0 / 1 的分界點在哪裡**
- 讓 l 是我們所知的最後一個 0，r 是第一個 1
- 最後 $l = r + 1$ 的時候就找到答案了
- 戳 m 的時候可以隨便取你喜歡的方向，反正 $r > l + 1$ 一定不會有剛剛討厭的問題！

整數二分搜

第二種實作想法

- 不想要分 case QQ
- 換成我們想知道 **0 / 1 的分界點在哪裡**
- 讓 l 是我們所知的最後一個 0，r 是第一個 1
- 最後 $l = r + 1$ 的時候就找到答案了
- 戳 m 的時候可以隨便取你喜歡的方向，反正 $r > l + 1$ 一定不會有剛剛討厭的問題！
- 三個問題一次解決：
  - 不管你要找 0 還是 1 都可以（自己選 l, r 要哪個）
  - 不用思考上下高斯問題
  - 兩邊剛好戳不到，多加的 0 跟 1 根本不需要多處理

整數二分搜

STL 幫你寫好的一些二分搜

- `lower_bound(first, last, value, comp);`
- 會從 [`first`, `last`) 找出第一個大於等於（不小於）`value` 的位置
- `upper_bound(first, last, value, comp);`
- 會從 [`first`, `last`) 找出第一個大於（不小於等於）`value` 的位置

- `comp` 是當作小於的比較函數

Sprout

整數二分搜

`comp` 是當作小於的比較函數

- 你可以用內建的：(預設是) `less<>()` 跟 `greater<>()`
- 你可以自己寫一個有 `bool operator()(const T &a, const T &b)` 的 struct
- 你可以傳一個 lambda function！

Sprout

二分搜

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 $a_1 < a_2 < \cdots < a_n$。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 $\min |a_i - a_j| \quad (i \neq j,\ i, j$ 兩點都有收費站$)$ 盡量大。請輸出這個最大可能的數值。

- $2 \leq k \leq n \leq 10^5$

Sprout

二分搜

- 要枚舉什麼？

Sprout

二分搜

- 要枚舉什麼？
- 答案？ 每個站**至少**要距離多遠

Sprout

二分搜

- 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，我們要找最後一個 0

二分搜

- 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，我們要找最後一個 0

- 戳一個 x 要花 $O(n)$ 檢查

二分搜

- 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，我們要找最後一個 0

- 戳一個 x 要花 $O(n)$ 檢查
- 上下界要選 1 到 $a_n - a_1$，所以總複雜度是 $O(n \log a_n)$

二分搜

- 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，我們要找最後一個 0

- 戳一個 x 要花 $O(n)$ 檢查
- 上下界要選 1 到 $a_n - a_1$，所以總複雜度是 $O(n \log a_n)$
- 可以只花 $O(n \log n)$ 嗎？

例題：Safe Distance

Problem SWERC 2020 Problem C Safe Distance

你想要通過一個 $X \times Y$ 的空間，從座標 $(0, 0)$ 走到 (X, Y)，這個空間裡面有 n 個人，第 i 個人在平面上的 (x_i, y_i) 位置。
不過你想要跟大家都保持社交距離，具體來說，你想要讓你路徑上最近的人距離越大越好。路途中你不可以走出這個房間。請輸出這個最大可能的數值。這個數值只要在相對或絕對誤差 10^5 以內就可以了。

- $1 \leq n \leq 1000$
- $1 \leq X, Y \leq 10^6$

例題：Safe Distance

- 相對或絕對誤差是什麼？

例題：Safe Distance

- 相對或絕對誤差是什麼？
- 絕對誤差是相減絕對值 $|a' - a|$
- 相對誤差是相除的差異 $|1 - \frac{a'}{a}|$

例題：Safe Distance

- 相對或絕對誤差是什麼？
- 絕對誤差是相減絕對值 $|a' - a|$
- 相對誤差是相除的差異 $|1 - \frac{a'}{a}|$
- a 很小的時候相對誤差會很麻煩，所以（電腦計算）通常都是要求兩個其中一個就可以

例題：Safe Distance

- 相對或絕對誤差是什麼？
- 絕對誤差是相減絕對值 $|a' - a|$
- 相對誤差是相除的差異 $|1 - \frac{a'}{a}|$
- a 很小的時候相對誤差會很麻煩，所以（電腦計算）通常都是要求兩個其中一個就可以

- 要枚舉什麼？

例題：Safe Distance

- 相對或絕對誤差是什麼？
- 絕對誤差是相減絕對值 $|a' - a|$
- 相對誤差是相除的差異 $|1 - \frac{a'}{a}|$
- a 很小的時候相對誤差會很麻煩，所以（電腦計算）通常都是要求兩個其中一個就可以

- 要枚舉什麼？
- 還是答案？ 你要離大家都**至少**多遠

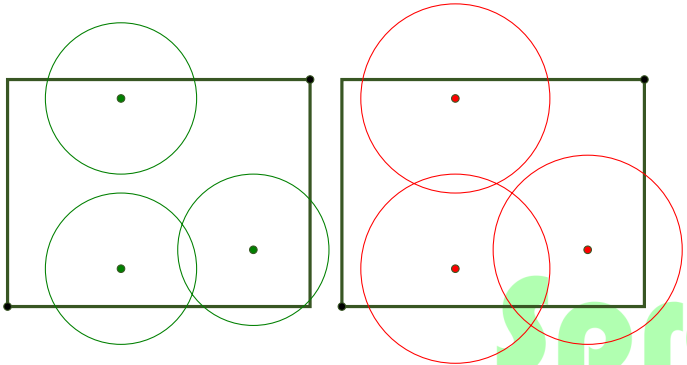
例題：Safe Distance

- 相對或絕對誤差是什麼？
- 絕對誤差是相減絕對值 $|a' - a|$
- 相對誤差是相除的差異 $|1 - \frac{a'}{a}|$
- a 很小的時候相對誤差會很麻煩，所以（電腦計算）通常都是要求兩個其中一個就可以

- 要枚舉什麼？
- 還是答案？ 你要離大家都**至少**多遠
- 好像還是可以二分搜？

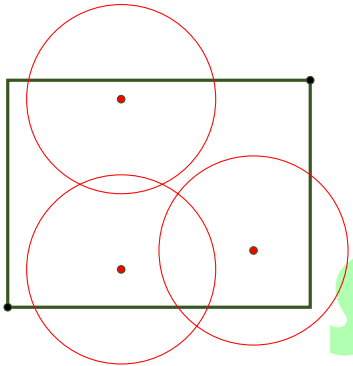
例題：Safe Distance

對於任何一個半徑 r，我們要怎麼知道有沒有這樣的路徑？



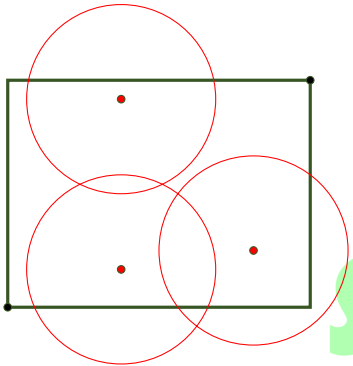
例題：Safe Distance

Key insight: 如果有一群人畫出半徑 x 的圓連通起來而且包含左上的任何一個邊邊跟右下的任何一個邊邊，就一定不能通過！



例題：Safe Distance

Key insight: 如果有一群人畫出半徑 x 的圓連通起來而且包含左上的任何一個邊邊跟右下的任何一個邊邊，就一定不能通過！ **反之就一定可以（為什麼）？**



例題：Safe Distance

- 每次用 $O(n^2)$ DFS 看有沒有這樣的連通狀況
- 好像可以跟整數一樣二分搜欸，真的嗎？

例題：Safe Distance

看看原本的二分搜框架：

- 我們想知道 **0 / 1 的分界點在哪裡**
- 讓 l 是我們所知的最後一個 0，r 是第一個 1
- 最後 $l = r + 1$ 的時候就找到答案了

例題：Safe Distance

看看原本的二分搜框架：

- 我們想知道 **0 / 1 的分界點在哪裡**
- 讓 l 是我們所知的最後一個 0，r 是第一個 1
- 最後 $l = r + 1$ 的時候就找到答案了？？？？？

實數二分搜

- 我們根本沒有這種 $+1$ 就是答案概念

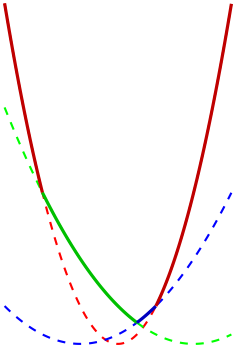
實數二分搜

- 我們根本沒有這種 $+1$ 就是答案概念
- 策略 1：如果 $r - l$ 夠小就可以停了
  - 設多小通常就是題目的誤差範圍
  - 比較直覺
  - 要求距離太小可能不小心就會無窮迴圈！

實數二分搜

- 我們根本沒有這種 $+1$ 就是答案概念
- 策略 1：如果 $r - l$ 夠小就可以停了
  - 設多小通常就是題目的誤差範圍
  - 比較直覺
  - 要求距離太小可能不小心就會無窮迴圈！
- 策略 2：外面跑一個固定次數
  - 有點反直覺
  - 只要次數設的合理就不會出事

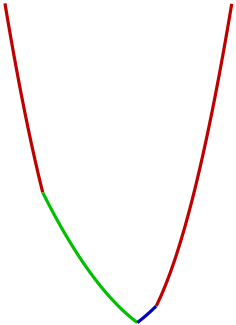
三分搜



- 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？
- 一般三分搜的對象：單峰函數

三分搜

- 影片上列出了一大堆 case，但其實根本不是四個點都重要！
- 戳兩個點 $a < b$
  - 如果 $f(a) < f(b)$ 就不會在 b 的右邊
  - 如果 $f(a) > f(b)$ 就不會在 a 的左邊



三分搜

- （以下假設是找最小值，找最大值的就自己加個負號）
- 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

Sprout

三分搜

- （以下假設是找最小值，找最大值的就自己加個負號）
- 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
- 嚴不嚴格遞減重要嗎？

Sprout

三分搜

這三個函數哪些可以三分搜？

三分搜

- 只能搜嚴格遞減、水平、嚴格遞增的函數

Sprout

例題：最小包覆圓 Special Case

Problem 最小包覆圓 Special Case

平面上有 N 個 y 座標是正的點，你想要找到一個圓滿足

- 他跟 x 軸相切（只交在一點上）
- 他包含住所有點

這樣的圓最小半徑可以是多少？

- $1 \leq N \leq 10^5$

例題：最小包覆圓 Special Case

- 我們先枚舉圓心好了，他的 y 座標剛好是半徑
- 如果有另一個點在 (a, b)，那包住他的條件是

$$(a - x)^2 + (b - y)^2 \leq y^2$$

例題：最小包覆圓 Special Case

- 我們先枚舉圓心好了，他的 y 座標剛好是半徑
- 如果有另一個點在 (a, b)，那包住他的條件是

$$(a-x)^2 + (b-y)^2 \leq y^2 \Longrightarrow (a-x)^2 + b^2 \leq 2by$$

例題：最小包覆圓 Special Case

- 我們先枚舉圓心好了，他的 y 座標剛好是半徑
- 如果有另一個點在 (a, b)，那包住他的條件是

$$(a-x)^2 + (b-y)^2 \leq y^2 \Longrightarrow (a-x)^2 + b^2 \leq 2by$$

所以所有點的限制都長的像是

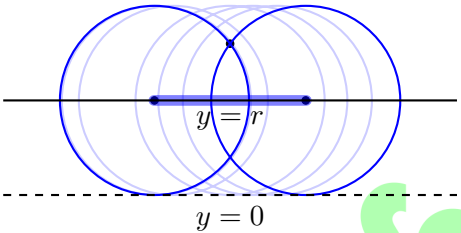
$$y \geq \frac{(a-x)^2 + b^2}{2b}$$

所有條件都要滿足，然後要求 y 的最小值，不就是求這些二次函數的 max 的最小值嗎？

例題：最小包覆圓 Special Case

- 如果我們枚舉半徑 r 呢？
- 如果有另一個點在 (a, b)，那圓心條件的是

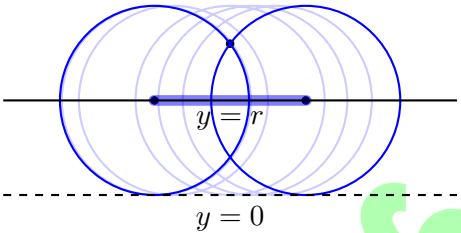
$$(a - x)^2 + (b - r)^2 \leq r^2$$



例題：最小包覆圓 Special Case

- 如果我們枚舉半徑 r 呢？
- 如果有另一個點在 (a, b)，那圓心條件的是

$$(a - x)^2 + (b - r)^2 \leq r^2$$



- 算出每個點的交集，如果有的話這個 r 就是好的，二分搜找最小的！

整數三分搜

- 二分搜會有不小心掉進無窮迴圈的問題，那整數上三分搜會不會有？

Sprout

整數三分搜

- 二分搜會有不小心掉進無窮迴圈的問題，那整數上三分搜會不會有？
- 其實中間只有戳兩個點而已，不如就戳 $f(m)$ 以及 $f(m+1)$

整數三分搜

- 二分搜會有不小心掉進無窮迴圈的問題，那整數上三分搜會不會有？
- 其實中間只有戳兩個點而已，不如就戳 $f(m)$ 以及 $f(m+1)$
- 我們其實想找到第一個 $f(m) \leq f(m+1)$ 的 m
- 變成一般的二分搜！

二分搜 vs 三分搜

在求整數上的單峰函數最小值問題：

- 二分搜問 2 次（大約）把剩餘的狀況變成 $\frac{1}{2}$
- 三分搜問 2 次（大約）把剩餘的狀況變成 $\frac{2}{3}$

二分搜 vs 三分搜

在求整數上的單峰函數最小值問題：

- 二分搜問 2 次（大約）把剩餘的狀況變成 $\frac{1}{2}$
- 三分搜問 2 次（大約）把剩餘的狀況變成 $\frac{2}{3}$

- 差分二分搜需要 $2\log_2 C$ 次計算 $f(x)$
- 三分搜需要 $2\log_{3/2} C$ 次計算 $f(x)$

二分搜 vs 三分搜

在求整數上的單峰函數最小值問題：

- 二分搜問 2 次（大約）把剩餘的狀況變成 $\frac{1}{2}$
- 三分搜問 2 次（大約）把剩餘的狀況變成 $\frac{2}{3}$

- 差分二分搜需要 $2\log_2 C$ 次計算 $f(x)$
- 三分搜需要 $2\log_{3/2} C$ 次計算 $f(x)$
- 他們兩個差了大約 1.71 倍！（三分搜比較慢）

二分搜

- 為什麼二分搜是二分搜，不是三分四分五分十分？

Sprout

二分搜

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

$$l = a_0 < a_1 < a_2 < \cdots < a_{k-1} < a_k = r$$

二分搜

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

$$l = a_0 < a_1 < a_2 < \cdots < a_{k-1} < a_k = r$$

- 只有在「我們看到的 0 - 1」中間才可能是 0 跟 1 的分界點

二分搜

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

$$l = a_0 < a_1 < a_2 < \cdots < a_{k-1} < a_k = r$$

- 只有在「我們看到的 0 - 1」中間才可能是 0 跟 1 的分界點
- 區間大小會變成 $\frac{1}{k}$，所以總詢問次數是 $(k-1)\log_k n$
- k 越小詢問次數最少，所以會取 $k = 2$

二分搜

- 為什麼三分搜是三分搜，不是四分五分六分十分？

二分搜

- 為什麼三分搜是三分搜，不是四分五分六分十分？
- 如果我們把搜尋範圍切成 k 半，然後我們的切割點是

$$l = a_0 < a_1 < a_2 < \cdots < a_{k-1} < a_k = r$$

二分搜

- 為什麼三分搜是三分搜，不是四分五分六分十分？
- 如果我們把搜尋範圍切成 k 半，然後我們的切割點是

$$l = a_0 < a_1 < a_2 < \cdots < a_{k-1} < a_k = r$$

- 只有在「我們看到的最小值」的左邊右邊兩塊可以出現「最小值」！

Sprout

二分搜

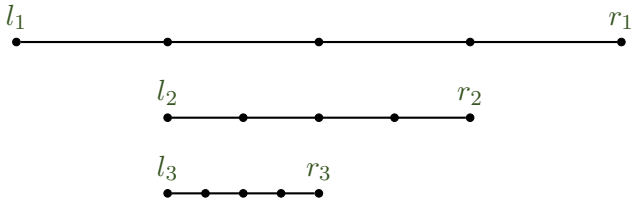
- 為什麼三分搜是三分搜，不是四分五分六分十分？
- 如果我們把搜尋範圍切成 k 半，然後我們的切割點是

$$l = a_0 < a_1 < a_2 < \cdots < a_{k-1} < a_k = r$$

- 只有在「我們看到的最小值」的左邊右邊兩塊可以出現「最小值」！
- 區間大小會變成 $\frac{2}{k}$，所以總詢問次數是 $(k-1)\log_{k/2} n$
- 最好的其實是四分搜！

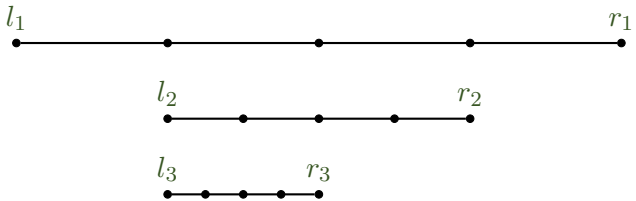
四分搜？

四分搜好像不只這麼好欸？！



四分搜？

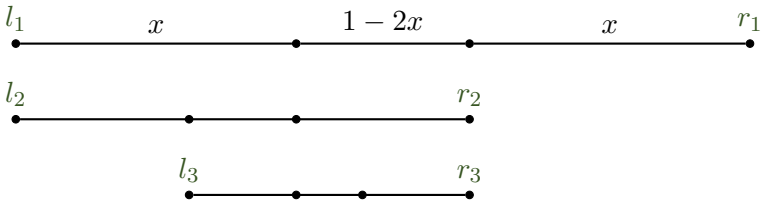
四分搜好像不只這麼好欸？！



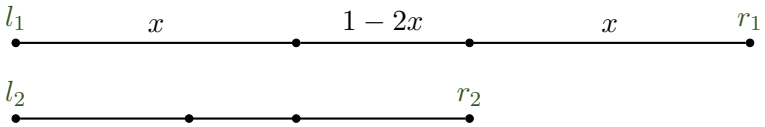
從第二次開始，中間的點其實都已經算過了！

三分搜？

我們可以同樣做在三分搜上嗎？

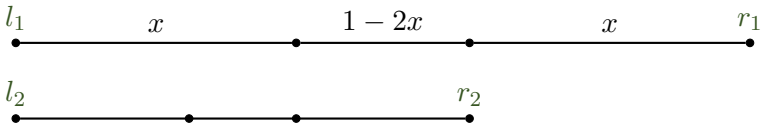


三分搜？



$$\frac{x}{1-2x}=\frac{1-x}{x}\Longrightarrow 2x^2-3x+1=x^2, x=\frac{3\pm\sqrt{5}}{2}$$

三分搜？



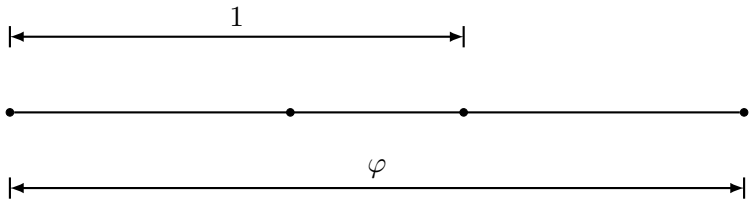
$$\frac{x}{1-2x} = \frac{1-x}{x} \Longrightarrow 2x^2 - 3x + 1 = x^2, x = \frac{3 \pm \sqrt{5}}{2}$$

這個比例 $x = \frac{3-\sqrt{5}}{2}$ 跟黃金比例有關！

$$\frac{3-\sqrt{5}}{2} = 1 - \frac{1}{\varphi}$$

Sprout

三分搜？



- 這東西有個名字叫黃金比例搜（Golden ratio search）
- 搜尋次數是 $\log_\varphi n + O(1)$，每次搜尋只要一次詢問
- 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**
- 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 1:2 是最好的
- 但在不是整數時我們不這麼做是因為，我們不知道可以多近