隨機算法

Lecture by WiwiHo

Credit: boook HNO2

大綱

- Introduction
- 一些你學過的問題
- Hash
- 怎麼隨機
- 隨機與數學
- 更多題目

Introduction

生活中的隨機

- 考試不會寫，亂猜一個答案
- 不知道午餐要吃什麼，列一個清單然後抽籤
- 請瀏覽器幫我設一個隨機密碼
- 班上抽打掃工作
    - 然後每次都抽到回收 QQ

演算法裡的隨機

- 題目好難不會做，用隨機亂弄一個答案好了
- 100% 正確的演算法好難寫，犧牲一點正確性能不能省一點功夫？
- 複雜度低的演算法可能不太好寫而且常數又很大，我的演算法偶爾稍微跑慢一點好像也沒什麼關係 (?)
- 有些題目根本就沒有 100% 對又很快的作法啊

隨機的種類

- Monte Carlo Algorithm:
  犧牲演算法的正確性，輸出的答案有一定機率會是錯的
  (但一定很快！)
  e.g. Hash
- Las Vegas Algorithm:
  犧牲演算法執行時間的穩定性，執行時間可能會很長
  (但一定是對的！)
  e.g. Quicksort

一些你學過的問題

Quicksort

```
function QUICKSORT(arr)
    if arr.size ≤ 1 then
        return
    end if
    pivot ← arr.size − 1
    把所有 < arr[pivot] 的數字移到前面、> 的移到後面，= 的放在中間
    QUICKSORT(< arr[pivot] 的部分)
    QUICKSORT(> arr[pivot] 的部分)
end function
```

- 這個演算法有什麼問題？

Quicksort

- 如果最後一個數字是所有數字中第 k 大的，那麼分完之後的兩邊大小最多就是 $k-1$ 和 $n-k$；要是每次都分成差不多一半，那麼時間是 $O(n\log n)$
- 可是如果不是呢...？

Quicksort

- 如果最後一個數字是所有數字中第 k 大的，那麼分完之後的兩邊大小最多就是 $k-1$ 和 $n-k$；要是每次都分成差不多一半，那麼時間是 $O(n\log n)$
- 可是如果不是呢...？
- $arr = [1, 2, 3, \ldots, n]$
- $T(n) = T(n-1) + O(n),\ T(1) = O(1)$

Quicksort

- 如果最後一個數字是所有數字中第 k 大的，那麼分完之後的兩邊大小最多就是 $k-1$ 和 $n-k$；要是每次都分成差不多一半，那麼時間是 $O(n\log n)$
- 可是如果不是呢...？
- $arr = [1,2,3,\ldots,n]$
- $T(n) = T(n-1) + O(n),\ T(1) = O(1)$
- $T(n) = O(n^2)$ QQ

Quicksort

- 因為 worst case 一定是在所有數字都相異的時候，所以我們接下來都假設數字相異
- 出題者好壞，故意弄這種測資
- 如果測資都是隨機的就好了...

Quicksort

- 因為 worst case 一定是在所有數字都相異的時候，所以我們接下來都假設數字相異
- 出題者好壞，故意弄這種測資
- 如果測資都是隨機的就好了...
- 咦，排序前長怎樣根本不重要，乾脆自己把它打亂好了！
- 先把序列 random shuffle，這樣無論是第幾大的數，被選成 pivot 的機率都是 $1/n$

Quicksort

- 因為 worst case 一定是在所有數字都相異的時候，所以我們接下來都假設數字相異
- 出題者好壞，故意弄這種測資
- 如果測資都是隨機的就好了...
- 咦，排序前長怎樣根本不重要，乾脆自己把它打亂好了！
- 先把序列 random shuffle，這樣無論是第幾大的數，被選成 pivot 的機率都是 $1/n$
- 感性理解：期望時間複雜度 $O(n \log n)$

尋找第 K 大

- 還記得分治課的 median of medians 嗎？
- 在 median of medians 的作法裡，我們藉由「挑出每個 5 個一組的人當中的中位數，再挑出它們的中位數」作為 pivot，然後把序列分成兩半

尋找第 K 大

- 還記得分治課的 median of medians 嗎？
- 在 median of medians 的作法裡，我們藉由「挑出每個 5 個一組的人當中的中位數，再挑出它們的中位數」作為 pivot，然後把序列分成兩半
- 那時候我們就偷偷暴雷了，它和 Quicksort 類似，原理在於盡可能把序列分成長度相近的兩半
- 隨機選 pivot！

Hash

小複習

- 還記得手寫作業出現過的 Hash 嗎？
- Hash 的常見用途
  - 比較兩個東西是不是一樣（字串、等一下會有的各種怪東西）
  - 值域壓縮（Hash table）
- 字串的 Rolling hash：
  $h(s_1 s_2 \ldots s_n) = s_1 \times C^{n-1} + s_2 \times C^{n-2} + \cdots + s_n \bmod M$

Sprout

小複習

- 還記得手寫作業出現過的 Hash 嗎？
- Hash 的常見用途
  - 比較兩個東西是不是一樣（字串、等一下會有的各種怪東西）
  - 值域壓縮（Hash table）
- 字串的 Rolling hash：
  $h(s_1 s_2 \ldots s_n) = s_1 \times C^{n-1} + s_2 \times C^{n-2} + \cdots + s_n \bmod M$
- 一樣的東西，hash 值一定一樣，但反過來就不一定
  - 被 hash 的物件通常有超級多種！
- 誤判的機率是多少？

矩陣乘法

Problem 矩陣乘法

給三個 $N \times N$ 矩陣 A, B, C，求是否 $AB = C$。

- $N \leq 10000$

矩陣乘法

- $N \times M$ 乘 $M \times K$ 的暴力矩陣乘法複雜度是 $O(NMK)$
  - 目前已知的確定性演算法好像都太慢了 QQ

矩陣乘法

- $N \times M$ 乘 $M \times K$ 的暴力矩陣乘法複雜度是 $O(NMK)$
  - 目前已知的確定性演算法好像都太慢了 QQ
- 秉持著 hash 的精神，要是我們把矩陣 hash，就只要判 $h(AB) = h(C)$ 就好

矩陣乘法

- $N \times M$ 乘 $M \times K$ 的暴力矩陣乘法複雜度是 $O(NMK)$
  - 目前已知的確定性演算法好像都太慢了 QQ
- 秉持著 hash 的精神，要是我們把矩陣 hash，就只要判 $h(AB) = h(C)$ 就好
- 只是要把一個矩陣 hash 的話好像很容易（例如直接 rolling hash）

矩陣乘法

- $N \times M$ 乘 $M \times K$ 的暴力矩陣乘法複雜度是 $O(NMK)$
  - 目前已知的確定性演算法好像都太慢了 QQ
- 秉持著 hash 的精神，要是我們把矩陣 hash，就只要判 $h(AB) = h(C)$ 就好
- 只是要把一個矩陣 hash 的話好像很容易（例如直接 rolling hash）
- 但是要有辦法算 $h(AB)$...

不一樣的 hash

- 沒有人說 hash 完的東西一定要是一個數字，既然暴力的問題在於矩陣很大，那我是不是可以把它 hash 成一個小一點的矩陣 (?)

不一樣的 hash

- 沒有人說 hash 完的東西一定要是一個數字，既然暴力的問題在於矩陣很大，那我是不是可以把它 hash 成一個小一點的矩陣 (?)
- 令 R 是一個 $N \times 1$ 矩陣，矩陣 A 的 hash 是 AR

不一樣的 hash

- 沒有人說 hash 完的東西一定要是一個數字，既然暴力的問題在於矩陣很大，那我是不是可以把它 hash 成一個小一點的矩陣 (?)
- 令 R 是一個 $N \times 1$ 矩陣，矩陣 A 的 hash 是 AR
- $h(AB) = (AB)R = ABR = A(BR)$，只要判 $ABR = CR$ 就好
- 時間複雜度變成 $O(N^2)$！

其實是一樣的 hash

- 其實我們在做的事情也可以看成是把矩陣的每一列 hash 成一個數字
- 字串的 rolling hash 就是把字串當成一個 $1 \times N$ 矩陣，hash 是乘上 $[1, C, C^2, \ldots]^T$
- 數字太大的話可以 mod M

成雙成對

Problem 成雙成對

給一個數列 $a_1, a_2, ..., a_N$，接下來有 Q 筆詢問，每筆詢問求一個區間 $[l, r]$ 裡是不是每種數字都出現偶數次。

- $N \leq 10^6$
- $Q \leq 10^6$
- $0 \leq a_i \leq 2^{31} - 1$

成雙成對

- Hash: 答案是 Yes 時一定輸出 Yes，答案是 No 時有機率會錯
- 所以我們先想辦法唬爛一個 Yes 時一定會對的版本 (?)

成雙成對

- Hash: 答案是 Yes 時一定輸出 Yes，答案是 No 時有機率會錯
- 所以我們先想辦法唬爛一個 Yes 時一定會對的版本 (?)
- 把數字裝進一個盒子裡，如果盒子裡有兩個數字長一樣，那我們希望它們互相打架然後消失，要是最後盒子裡沒有東西，答案就是 Yes，反之就是 No

成雙成對

- Hash: 答案是 Yes 時一定輸出 Yes，答案是 No 時有機率會錯
- 所以我們先想辦法唬爛一個 Yes 時一定會對的版本 (?)
- 把數字裝進一個盒子裡，如果盒子裡有兩個數字長一樣，那我們希望它們互相打架然後消失，要是最後盒子裡沒有東西，答案就是 Yes，反之就是 No
- XOR

成雙成對

- $a \oplus a = 0$
  - 一樣的數字會自己打架然後不見
- 如果區間裡全部數字都出現偶數次，那麼它們全部 xor 起來一定是 0

成雙成對

- $a \oplus a = 0$
  - 一樣的數字會自己打架然後不見
- 如果區間裡全部數字都出現偶數次，那麼它們全部 xor 起來一定是 0
- $1 \oplus 2 \oplus 3$？

隨機

- 如果所有數字都是隨機的，xor 和剛好是 0 的機率就是 $1/C$
- 本來的數值是什麼不重要！
- 把每一種數字都替換成另一種隨機數字
- 其實我們是在把盒子 hash
- 你的作業題目：K 的倍數的版本

Hash table

- 用 Hash function 可以把值域壓縮的特性來儲存和查找資料

Hash table

- 用 Hash function 可以把值域壓縮的特性來儲存和查找資料
- 碰撞的處理方式：open addressing, chaining...
- C++ 裡的 hash table：
  - STL unordered 系列
  - `__gnu_pbds::gp_hash_table`: open addressing
  - `__gnu_pbds::cc_hash_table`: chaining
  - C++ STL: Order of magnitude faster hash tables with Policy Based Data Structures
  - Blowing up unordered_map, and how to stop getting hacked on it

碰撞的機率

- 在使用 hash table 的時候，要是碰撞的狀況很多，那就要額外花很多力氣處理碰撞
- 在用 hash 檢查兩個東西是不是一樣的時候，碰撞就完蛋了！
- 所以，碰撞的機率如何？

碰撞的機率

- 在使用 hash table 的時候，要是碰撞的狀況很多，那就要額外花很多力氣處理碰撞
- 在用 hash 檢查兩個東西是不是一樣的時候，碰撞就完蛋了！
- 所以，碰撞的機率如何？
- 在理想的狀況下，如果 hash function 的值域大小是 m，那兩個不同物件 hash 值一樣的機率就是 $1/m$
- 然而，要是今天有個人盯著你的 code 硬是要搞你，他肯定是做得到的

碰撞的機率

- 在使用 hash table 的時候，要是碰撞的狀況很多，那就要額外花很多力氣處理碰撞
- 在用 hash 檢查兩個東西是不是一樣的時候，碰撞就完蛋了！
- 所以，碰撞的機率如何？
- 在理想的狀況下，如果 hash function 的值域大小是 m，那兩個不同物件 hash 值一樣的機率就是 $1/m$
- 然而，要是今天有個人盯著你的 code 硬是要搞你，他肯定是做得到的
- 一些解決方法：用一些不尋常的 hash 方式、準備多個 hash function 隨機抽一種用
- Universal hashing：準備一個 hash function 的 set H，使得對於任意 $x \neq y$，都有 $\Pr_{h \in H}(h(x) = h(y)) \leq 1/m$

小結

- hash function：把物件打到整數的函數
    - 什麼都可以是物件
    - 函數可以長得奇形怪狀
- 易於檢查是否相同
- 易於計算

怎麼隨機

大家都會

```cpp
#include <iostream>

using namespace std;

int main(){
    srand(time(NULL));
    cout << rand() << "\n";
}
```

怎麼產生隨機數

- 你會怎麼自己產生一個隨機數字
- 你產生的方法真的是隨機的嗎？(什麼是隨機？)

怎麼產生隨機數

- 你會怎麼自己產生一個隨機數字
- 你產生的方法真的是隨機的嗎？(什麼是隨機？)
- 亂數生產器（aka 隨機函數）：會產生一個看起來很隨機的數列的函數
- 亂數生產器的輸入：seed
  - 一樣的 seed 要生出一樣的結果！

亂數生產器

- 一個例子：

$$f(A,B,C,x) = Ax^2 + Bx + C \bmod 107$$

$$f(23,11,20,50..60) = \{76,56,82,47,58,8,4,46,27,54,20\}$$

不好的隨機函數

$$f(A,B,C,x) = Ax^2 + Bx + C \bmod 107$$

- 根據模除性質，如果 $x \equiv y \pmod{107}$，那麼 $f(A,B,C,x)$ 和 $f(A,B,C,y)$ 會是一樣的，也就是說它們有長度為 107 的循環節，好短 QQ
- 我們想像的隨機需要有什麼特性？

不好的隨機函數

$$f(A, B, C, x) = Ax^2 + Bx + C \bmod 107$$

- 根據模除性質，如果 $x \equiv y \pmod{107}$，那麼 $f(A, B, C, x)$ 和 $f(A, B, C, y)$ 會是一樣的，也就是說它們有長度為 107 的循環節，好短 QQ
- 我們想像的隨機需要有什麼特性？
  - 不能預測？(密碼學肯定很在乎)

不好的隨機函數

$$f(A,B,C,x) = Ax^2 + Bx + C \bmod 107$$

- 根據模除性質，如果 $x \equiv y \pmod{107}$，那麼 $f(A,B,C,x)$ 和 $f(A,B,C,y)$ 會是一樣的，也就是說它們有長度為 107 的循環節，好短 QQ
- 我們想像的隨機需要有什麼特性？
  - 不能預測？(密碼學肯定很在乎)
  - 好的分布？

rand()

- rand() 有什麼問題？
- 在你的電腦上試試看 `cout << RAND_MAX;`，在 Windows 上，它很小
- 在不同電腦上可能有不同結果！
  - 在做題目好像沒關係，可是有些時候你可能需要確保你的程式在不同環境下有相同結果

mt19937

- 聽起來 rand() 不太好用，怎麼辦呢？自己寫隨機函數？
- 自己寫不失為一個好方法，不過也有現成的函數可以用
- 梅森旋轉演算法（Mersenne twister）
- mt19937：周期長達 $2^{19937}-1$ 的亂數生產器
- 真的不可預測嗎？

```
mt19937 rng(123123);
cout << rng() << "\n"; // unsigned int
```

random shuffle

- `shuffle(v.begin(), v.end(), rng)`
- 怎麼自己寫 random shuffle？

seed

- 怎麼選 seed？
- `time(NULL)` 不好嗎？
- `chrono::system_clock::now().time_since_epoch().count()`

time

Defined in header `<time.h>`

```
time_t time( time_t *arg );
```

Returns the current calendar time encoded as a `time_t` object, and also stores it in the `time_t` object pointed to by `arg` (unless `arg` is a null pointer)

Parameters

arg - pointer to a `time_t` object where the time will be stored, or a null pointer

Return value

Current calendar time encoded as `time_t` object on success, `(time_t)(-1)` on error. If `arg` is not a null pointer, the return value is also stored in the object pointed to by `arg`.

Notes

The encoding of calendar time in `time_t` is unspecified, but most systems conform to POSIX specification and return a value of integral type holding the number of seconds since the Epoch. Implementations in which `time_t` is a 32-bit signed integer (many historical implementations) fail in the year 2038.

時間剪枝

- 當我們的隨機算法有機率會錯，我們就會需要做很多次來提高正確率
- 固定要做的次數是一種方法
- 也可以做到快沒時間為止

```
int start = clock();
while(clock() - start <= 0.99 * TIME_LIMIT *
    CLOCKS_PER_SEC){
     // ...
}
```

隨機與數學

簡單的練習

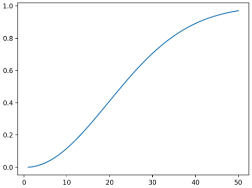
- 題目有 T 筆測資，每筆測資你的程式會有 p 的機率答對，並且你做了 k 次取最好的，請問你 AC 的機率是多少

簡單的練習

- 題目有 T 筆測資，每筆測資你的程式會有 p 的機率答對，並且你做了 k 次取最好的，請問你 AC 的機率是多少
- 答錯的機率：$1 - p$
- 單筆測資答錯：猜了 k 次全部都錯，機率是 $(1-p)^k$
- 單筆測資答對：$1 - (1-p)^k$
- 全部答對：$(1-(1-p)^k)^T$

生日悖論

- 假設一年有 365 天，班上有 n 個人，有任兩個人生日相同的機率是多少？
- $1-\frac{P_n^{365}}{365^n}$




```
2: 0.002739726027397249
3: 0.008204165884781345
4: 0.016355912466550326
5: 0.02713557369979358
6: 0.040462483649111536
7: 0.05623570309597365
8: 0.07433529235166902
9: 0.09462383388916673
10: 0.11694817771107768
11: 0.141141378321733
12: 0.16702478883806438
13: 0.19441027523242937
14: 0.223102512004973
15: 0.25290131976368635
16: 0.2836040052528499
17: 0.31500766529656066
18: 0.34691141787178936
19: 0.37911852603153673
20: 0.41143838358057994
21: 0.4436883351652058
22: 0.4756953076625501
23: 0.5072972343239854
24: 0.5383442579145288
25: 0.5686997039694639
26: 0.598240820135939
27: 0.626859282263242
28: 0.6544614723423994
29: 0.680968537477777
30: 0.7063162427192686
31: 0.7304546337286438
32: 0.7533475278503207
```

Hash 成功率

- 如果我們希望我們拿到的不同東西 hash 出來一定要不一樣，hash 的成功率是多少？
- 假設所有東西都各自獨立、均勻地 map 到 $[0, M)$ 內[1]，那麼對於某個固定的東西，其他隨便一個東西和它碰撞的機率是 $1/M$
- 聽起來還不錯，那如果我有 N 個東西呢？
- M 個東西裡面選 N 個，任兩個不重複的機率是多少？

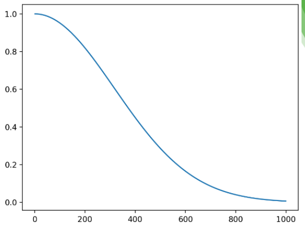
[1]這東西有一個術語叫 uniform hashing

Hash 成功率

- M 個東西選 $N \le M$ 個，選到不重複的機率？

$$\frac{P^M_N}{M^N} = \frac{M!}{(M-N)!M^N} = \frac{M(M-1)\cdots(M-N+1)}{M^N}$$

$M = 10^5$



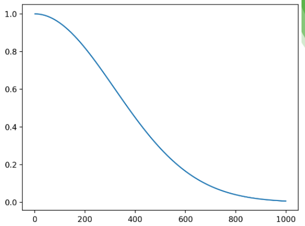
Hash 成功率

- M 個東西選 $N \le M$ 個，選到不重複的機率？

$$\frac{P^M_N}{M^N} = \frac{M!}{(M-N)!M^N} = \frac{M(M-1)\cdots(M-N+1)}{M^N}$$

- 它下降得好快 QQ
- 事實上，當 N 到達 $\sqrt{M}$ 左右時，不碰撞的機率就會小於 0.5 了…

$M = 10^5$



Hash mod 怎麼選

- Hash 時用的 mod 要怎麼選？
- 首先它要夠大
  - 至少要是數量平方的量級
- 最好要選質數
- 可以拿很多個不同的大質數分別做 hash，每一個的結果都一樣才算一樣
  - 關鍵字：中國剩餘定理

期望值

- 還記得剛剛講到的 quicksort 和 nth_element 嗎？
- **期望**時間複雜度是怎麼算的？
- 一些名字很像的複雜度
  - 期望（expected）時間複雜度：
    同一個程式對同樣的測資執行很多次取平均的時間
    (跟測資無關，很衰的時候才會很久)
  - 平均（average）時間複雜度：
    同一個程式對各種不同的測資執行取平均的時間
    (跟測資有關，出題者很壞就會很久)
  - 均攤（amortized）時間複雜度：
    一堆操作總共花費的時間取平均當成單一操作的時間
    (是真正的時間)

期望值

$$E[X] = \sum_{x\ \text{是一種可能的}\ X} x \times \Pr(X = x)$$

- 白話文：對於所有可能的狀況，把它發生的機率乘上我們關心的數值後加總
- Linearity of Expectation: $E[X+Y] = E[X] + E[Y]$
  - 不管 X、Y 是否獨立！
- $E[c \times X] = c \times E[X]$
- $E[X \times Y]$ 不一定是 $E[X] \times E[Y]$

Sprout

Quicksort 複雜度

- 直觀的想法：Quicksort 其實是一種分治，所以時間複雜度應該是 $O(n \times \text{層數})$，期望上每次會砍大概一半然後遞迴，所以層數是 $\log n$

Quicksort 複雜度

- 直觀的想法：Quicksort 其實是一種分治，所以時間複雜度應該是 $O(n \times \text{層數})$，期望上每次會砍大概一半然後遞迴，所以層數是 $\log n$
- 嚴謹的證明？
  - 假設我們在排序一個 $1 \sim n$ 的 permutation
  - $X_{i,j} =$ 第 i 個東西有沒有跟 j 比較
  - $E[\text{比較次數}] = E[\sum X_{i,j}] = \sum E[X_{i,j}]$
  - 什麼條件下 i 會和 j 比？

3-SAT randomized algorithm

Problem MAX-3-SAT

有 N 個布林變數 $x_1, x_2, \ldots, x_N$，還有 M 個條件，每個條件都形如

$$\neg^? x_i \vee \neg^? x_j \vee \neg^? x_k$$

i, j, k 相異。($\neg^?$ 代表可能有 $\neg$ 或可能沒有，像是 $x_1 \vee \neg x_2 \vee x_3$。)
求一組 $x_1, x_2, \ldots, x_N$ 使得被滿足的條件數量盡量多。

不幸的是，光是判斷有沒有解都是 NPC 問題，所以我們放寬一點要求：求一組解使得至少 $7M/8$ 個條件被滿足。

Sprout

3-SAT randomized algorithm

- 隨便猜一個解，期望上會滿足幾個條件？

$$
\begin{aligned}
&E[\sum \text{第 } i \text{ 個條件被滿足}] \\
&= \sum E[\text{第 } i \text{ 個條件被滿足}] \\
&= \sum \left(1 - \left(\frac{1}{2} \times \frac{1}{2} \times \frac{1}{2}\right)\right) \\
&= \sum \frac{7}{8} = \frac{7M}{8}
\end{aligned}
$$

- 這件事告訴我們解一定是存在的，如果不存在任何一組解使得滿足條件數量 $\geq \frac{7M}{8}$，期望值就不可能是 $\frac{7M}{8}$

3-SAT randomized algorithm

- 一直亂猜直到猜中為止，期望上要猜幾次？

3-SAT randomized algorithm

- 一直亂猜直到猜中為止，期望上要猜幾次？
- p_i = 滿足恰 i 個條件的機率，p = 成功機率

$$
\begin{aligned}
\frac{7M}{8} &= \sum_{0 \le i \le M} i p_i \\
&= \sum_{0 \le i < \frac{7M}{8}} i p_i + \sum_{\frac{7M}{8} \le i \le M} i p_i \\
&\le \left(\frac{7M}{8} - \frac{1}{8}\right) \sum_{0 \le i < \frac{7M}{8}} p_i + M \sum_{\frac{7M}{8} \le i \le M} p_i \\
&\le \left(\frac{7M}{8} - \frac{1}{8}\right) \cdot 1 + M \cdot p
\end{aligned}
$$

$$p \ge \frac{1}{8M}$$

3-SAT randomized algorithm

- 一直亂猜直到猜中為止，期望上要猜幾次？
- p_i = 滿足恰 i 個條件的機率，p = 成功機率

$$
\begin{aligned}
\frac{7M}{8} &= \sum_{0 \le i \le M} i p_i \\
&= \sum_{0 \le i < \frac{7M}{8}} i p_i + \sum_{\frac{7M}{8} \le i \le M} i p_i \\
&\le \left(\frac{7M}{8} - \frac{1}{8}\right) \sum_{0 \le i < \frac{7M}{8}} p_i + M \sum_{\frac{7M}{8} \le i \le M} p_i \\
&\le \left(\frac{7M}{8} - \frac{1}{8}\right) \cdot 1 + M \cdot p
\end{aligned}
$$

$$p \ge \frac{1}{8M}$$

Derandomization

- 哇，期望值這麼厲害！但難道沒有確定性的演算法解決這個問題嗎？

Derandomization

- 哇，期望值這麼厲害！但難道沒有確定性的演算法解決這個問題嗎？
- 假設我們現在想決定 x_1，如果我們強制決定它是 T 或 F，剩下的變數一樣隨機...
- $x_1 = T$ 或 F 時，成立的條件數量期望值分別是 $E[Y \mid x_1 = T]$ 和 $E[Y \mid x_1 = F]$，$Y =$ 滿足的條件數量

$$E[Y] = \frac{1}{2}E[Y \mid x_1 = T] + \frac{1}{2}E[Y \mid x_1 = F] \geq \frac{7M}{8}$$

Derandomization

- 哇，期望值這麼厲害！但難道沒有確定性的演算法解決這個問題嗎？
- 假設我們現在想決定 x_1，如果我們強制決定它是 T 或 F，剩下的變數一樣隨機...
- $x_1 = T$ 或 F 時，成立的條件數量期望值分別是 $E[Y \mid x_1 = T]$ 和 $E[Y \mid x_1 = F]$，$Y =$ 滿足的條件數量

$$E[Y] = \frac{1}{2}E[Y \mid x_1 = T] + \frac{1}{2}E[Y \mid x_1 = F] \geq \frac{7M}{8}$$

- $E[Y \mid x_1 = T]$ 和 $E[Y \mid x_1 = F]$ 至少要有一個 $\geq \frac{7M}{8}$！
- 對 $x_1 = T$ 和 $x_1 = F$ 各算一次期望值，取較大那個，按照這個方式一一決定每一個 x_i

Derandomization

- 哇，期望值這麼厲害！但難道沒有確定性的演算法解決這個問題嗎？
- 假設我們現在想決定 x_1，如果我們強制決定它是 T 或 F，剩下的變數一樣隨機...
- $x_1 = T$ 或 F 時，成立的條件數量期望值分別是 $E[Y \mid x_1 = T]$ 和 $E[Y \mid x_1 = F]$，Y = 滿足的條件數量

$$E[Y] = \frac{1}{2}E[Y \mid x_1 = T] + \frac{1}{2}E[Y \mid x_1 = F] \geq \frac{7M}{8}$$

- $E[Y \mid x_1 = T]$ 和 $E[Y \mid x_1 = F]$ 至少要有一個 $\geq \frac{7M}{8}$！
- 對 $x_1 = T$ 和 $x_1 = F$ 各算一次期望值，取較大那個，按照這個方式一一決定每一個 x_i
- 最後，我們得到了一組解，而且它的期望值不會比一開始的差，而且其實沒有任何隨機成分！
- 時間複雜度 $O(NM)$

更多題目

樹同構判斷

Problem CSES Tree Isomorphism I

給你兩棵 N 個點的**有根**樹，求它們是否同構 (isomorphism)。
樹同構的定義：你有辦法為每個點的子節點排順序，使得兩棵樹長得一模一樣。
$N \leq 10^5$

假設我們時間很多

- 你有辦法為每個點的子節點**排順序**，使得兩棵樹長得一模一樣
- 幫子節點們選一個排序方式

假設我們時間很多

- 你有辦法為每個點的子節點**排順序**，使得兩棵樹長得一模一樣
- 幫子節點們選一個排序方式
- 例如：每個節點有一個序列，這個序列是把對它為根的子樹 DFS 時往下往上走的順序記錄下來，往下走是 0，往上走是 1，對子節點的這個序列的字典序排序
- 節點 v 的序列：$0\langle c_1$ 的序列$\rangle\langle c_2$ 的序列$\rangle \ldots 1$
  $c_i =$ 排序後第 i 個子節點

時間沒有很多的話

- 這方法很容易比較，但是不好計算
- 把每個節點的序列 hash，然後直接照 hash 值大小排序
- 易於計算！
- Bonus: 無根樹怎麼辦？

時間沒有很多的話

- 這方法很容易比較，但是不好計算
- 把每個節點的序列 hash，然後直接照 hash 值大小排序
- 易於計算！
- Bonus: 無根樹怎麼辦？
- 注意到這裡的 hash 得要是把那個序列直接當成字串 hash，如果你是寫成

$$h(v) = h(c_1) \times C^k + h(c_2) \times C^{k-1} + \cdots + h(c_k) \times C + 1$$

之類的這種東西，這是錯的

最近點對

Problem CSES Minimum Euclidean Distance

平面上有 N 個點，求它們之間最小的歐氏距離是多少。

- 分治好難寫 QQ，可不可以隨機？

最近點對

- 假設我們現在有 $i-1$ 個點在平面上，並且我們知道它們的最近點對距離是 d，然後我們要加入第 i 個點並找出新的最近點對距離
- 如果距離變短，那最近點對的其中一個點一定是 i
- i 周圍畫一個半徑為 d 的圓內有其他點
- 但是要檢查半徑為 d 的圓內有沒有其他點太難了...

最近點對

- 退而求其次，我們找到一些可能跟它距離小於 d 的點如果這些點數很少，那就很棒
- 把整個平面切成 $(d/2) \times (d/2)$ 的網格
- 檢查所有 i 所在的格子和周圍 8 格和再往外一圈共 25 格裡的點
- 如果找到距離小於 d 的點，就把網格重畫不然把 i 直接加進它在的那個格子裡

時間分析

- 每個格子內最多只有 1 個點 $=>$ 最多只要檢查 25 個點
- 每次重畫格子要花 $O(i)$ 的時間，不然加入只要花 $O(1)$ 的時間（用 hash table）
- 如果加入點的順序是隨機的，那麼期望總時間是

$$\begin{aligned} & E[\sum \text{加入第 } i \text{ 個點花的時間}] \\ = & \sum E[\text{加入第 } i \text{ 個點花的時間}] \\ = & \sum (p_i \times O(i) + (1 - p_i) \times O(1)) \end{aligned}$$

- $p_i =$ 第 i 個點要重畫格子的機率

時間分析

- 重畫格子的機率是多少？
- 第 i 個點需要重畫格子代表它在前 i 個點的最近點對裡
- 如果先把順序 random shuffle，這個機率就是 $2/i$
- $p_i \times O(i) + (1 - p_i) \times O(1) = O(1)$
- 期望總時間複雜度 $O(N)$

長度為 8 的簡單路

Problem 長度為 8 的簡單路

給一個 N 點 M 邊的有向圖，還有兩個點 S, T，求一條從 S 到 T 恰有 8 個點的簡單路徑（含頭尾），保證有解。

$1 \leq N \leq 100, 1 \leq M \leq 2000$

- 不用簡單的話要怎麼做？

換個問題

Problem 長度為 8 的簡單路 - 塗色版

給一個 N 點 M 邊的有向圖，還有兩個點 S, T。有 6 種顏色，S, T 以外**每個點是其中一種顏色**，S, T 沒有顏色。求一條從 S 到 T **每種顏色出現恰一次**的簡單路徑。

換個問題

Problem 長度為 8 的簡單路 - 塗色版

給一個 N 點 M 邊的有向圖，還有兩個點 S, T。有 6 種顏色，S, T 以外**每個點是其中一種顏色**，S, T 沒有顏色。求一條從 S 到 T **每種顏色出現恰一次**的簡單路徑。

- 開 $2^6 \times N = 64N$ 個節點，節點 (msk, v)（$0 \le msk < 2^6$, $1 \le v \le N$）代表走到了 v、已經經過 msk 是 1 的 bit 的那些顏色
- $(0, S)$ 是否能走到 $(2^6 - 1, T)$
- 總節點數 $2^6 N$，邊數 $2^6 M$，總時間複雜度 $O(2^6 M)$

回到本來的問題

- 經過每種顏色恰一次 => 沒有點被經過兩次
- 如果我們隨機給本來的圖塗上顏色，再做這個修改版的問題
- 剛好答案的路徑中間 6 個點，每個點顏色都不同，那我們就會得到答案
- 機率：$\frac{6!}{6^6} \approx 0.01543209876\ldots$

好低怎麼辦

- 對的機率好低就多做幾次！
- $1 \leq N \leq 100, 1 \leq M \leq 2000$
- 做一次要 $64M \approx 10^5$ 的時間
- 做一次的錯誤率：$(1 - 6!/6^6) \approx 0.98$
  做 100 次的錯誤率：$(1 - 6!/6^6)100 \approx 0.21$
  做 500 次的錯誤率：$(1 - 6!/6^6)500 \approx 0.004$
  做 1000 次的錯誤率：$(1 - 6!/6^6)1000 \approx 1.76 \times 10^{-7}$