搜尋與枚舉

sam571128

September 28, 2021

搜尋? 枚舉?

搜尋和枚舉在打競賽中，是最基礎的技巧之一

不過，他卻也是決定比賽勝負的關鍵之一

很多我們無法找到一個能快速解決問題的演算法時

暴力搜尋卻能夠解決我們的問題

搜尋? 枚舉?

2021 TOI 初選

- 記分板
- 高一保障: 130 分，女生保障: 124 分

在這場比賽中，不用會任何的演算法，會搜尋枚舉即可拿到 160 分左右

搜尋與枚舉為何重要? 因為他沒有任何先備條件

單純只要會「寫程式」，就可以做到這件事

線性搜尋?

搜尋

我想著 1 到 n 的其中一個數字，每次猜完，你會知道你的數字太大或太小，猜猜看我想的數字是多少?

線性搜尋?

搜尋

我想著 1 到 n 的其中一個數字，每次猜完，你會知道你的數字太大或太小，猜猜看我想的數字是多少?

遇到這種問題，一般人應該都會很直覺的想到

既然我要猜這個數字，那我從頭到尾找一次，不就好了嗎?

線性搜尋?

猜數字

我想著 1 到 n 的其中一個數字，每次猜完，你會知道你的數字太大或太小，猜猜看我想的數字是多少?

```
for(int i = 0; i < n; i++){
    if(arr[i]==x){
        cout << "Find!\n";
    }
}
```

線性搜尋?

猜數字

我想著 1 到 n 的其中一個數字，每次猜完，你會知道你的數字太大或太小，猜猜看我想的數字是多少?

```
for(int i = 0; i < n; i++){
    if(arr[i]==x){
        cout << "Find!\n";
    }
}
```

時間複雜度: $O(n)$

更快的搜尋?

猜數字

我想著 1 到 n 的其中一個數字，每次猜完，你會知道你的數字太大或太小，猜猜看我想的數字是多少?

注意到這個問題，基本上我們可以轉換成是一個排序好的陣列

而我們要在其中，尋找某個特定的元素 x

更快的搜尋?

搜尋

給定一個排序好的陣列，詢問 x 是否存在這個陣列中

這個搜尋方式即為大家所熟知的「二分搜尋法 (Binary Search)」

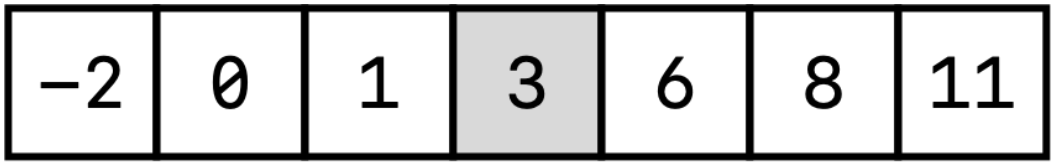
詳細作法如下

更快的搜尋?

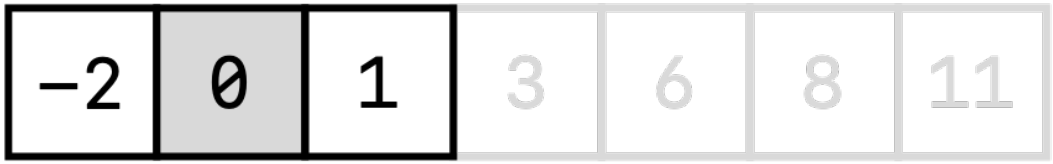
假設今天我們有一個陣列: $\{-2, 0, 1, 3, 6, 8, 11\}$

我們要在裡面尋找 -1 這個值

更快的搜尋?



-1 < 3
search left side



-1 < 0
search left side

-1 > -2
search right side

empty list
-1 is not in original list

更快的搜尋?

會發現我們每次都淘汰了一半的數字

因此像原本陣列大小為 7

$7/2 = 3 \Rightarrow 3/2 = 1 \Rightarrow 1/2 = 0$

總操作為三次，也就是 $\lceil \log_2(7) \rceil = 3$

會發現這個演算法其實只有 $O(\log n)$ 的複雜度!

$O(\log n)$ 比 $O(n)$ 快了超級多!!

回到猜數字

既然我們一開始知道我們的答案範圍為 1 到 n

那麼我們每次就去猜這個範圍的正中間，也就是 $\frac{1+n}{2}$

假設得到的回答是「太大」，那麼我們的範圍就縮成 $[1, \frac{1+n}{2} - 1]$

反之，範圍縮成 $[\frac{1+n}{2} + 1, n]$

回到猜數字

那麼假設答案範圍為 l 到 r

那麼我們每次就去猜這個範圍的正中間，也就是 $\frac{l+r}{2}$

假設得到的回答是「太大」，那麼我們的範圍就縮成 $[l, \frac{l+r}{2} - 1]$

反之，範圍縮成 $[\frac{l+r}{2} + 1, r]$

回到猜數字

```cpp
int l = 1, r = 100;

while(l < r){
    int m = (l+r)/2;
    int result = ask(m);
    if(result==1){
        //以 1 表示太大
        r = mid-1;
    }else if(result==-1){
        //以 -1 表示太小
        l = mid+1;
    }else if(result==0){
        //以 0 表示
        cout << "Find!\n";
    }
}
```

二分搜尋真的有必要嗎?

但回過頭來看，只能在排序好的陣列做到這一點是不是感覺很沒用?

二分搜尋真的有必要嗎?

不過並非如此，二分搜尋不只能用在這種地方

只要題目出現所謂的「單調性」，即可使用此算法

二分搜尋的另外一種題目

當題目出現，請找到做到某件事情時的最大或最小 x 值

此時，二分搜尋法就是可以拿來考慮做的事情!

二分搜尋的另外一種題目

在這裡舉一題 APCS 考古題「基地台」為例

第 4 題　基地台

問題描述

為因應資訊化與數位化的發展趨勢，某市長想要在城市的一些服務點上提供無線網路服務，因此他委託電信公司架設無線基地台。某電信公司負責其中 N 個服務點，這 N 個服務點位在一條筆直的大道上，它們的位置(座標)係以與該大道一端的距離 P[i]來表示，其中 i=0~N-1。由於設備訂製與維護的因素，每個基地台的服務範圍必須都一樣，當基地台架設後，與此基地台距離不超過 *R* (稱為基地台的半徑)的服務點都可以使用無線網路服務，也就是說每一個基地台可以服務的範圍是 D=2R(稱為基地台的直徑)。現在電信公司想要計算，如果要架設 K 個基地台，那麼基地台的最小**直徑**是多少才能使每個服務點都可以得到服務。

基地台架設的地點不一定要在服務點上，最佳的架設地點也不唯一，但本題只需要求最小直徑即可。以下是一個 N=5 的例子，五個服務點的座標分別是 1、2、5、7、8。

假設 K=1，最小的直徑是 7，基地台架設在座標 4.5 的位置，所有點與基地台的距離都在半徑 3.5 以內。假設 K=2，最小的直徑是 3，一個基地台服務座標 1 與 2 的點，另一個基地台服務另外三點。在 K=3 時，直徑只要 1 就足夠了。

二分搜尋的另外一種題目

簡單來說，有 n 個要覆蓋的點，以及 k 個基地台

問基地台的覆蓋直徑最少要是多少才能覆蓋所有點

二分搜尋的另外一種題目

如果我們以 k 個基地台去想，要找直徑基本上很困難

因為究竟要怎麼放基地台才好呢?

二分搜尋的另外一種題目

如果我們以 k 個基地台去想，要找直徑基本上很困難

因為究竟要怎麼放基地台才好呢?

這樣基本上十分困難，因為基地台的放置方式會影響我們的直徑

二分搜尋的另外一種題目

換個想法，假設我們固定一種直徑，去找是否能用 $<= k$ 個基地台覆蓋

這樣其實簡單很多了吧!

二分搜尋的另外一種題目

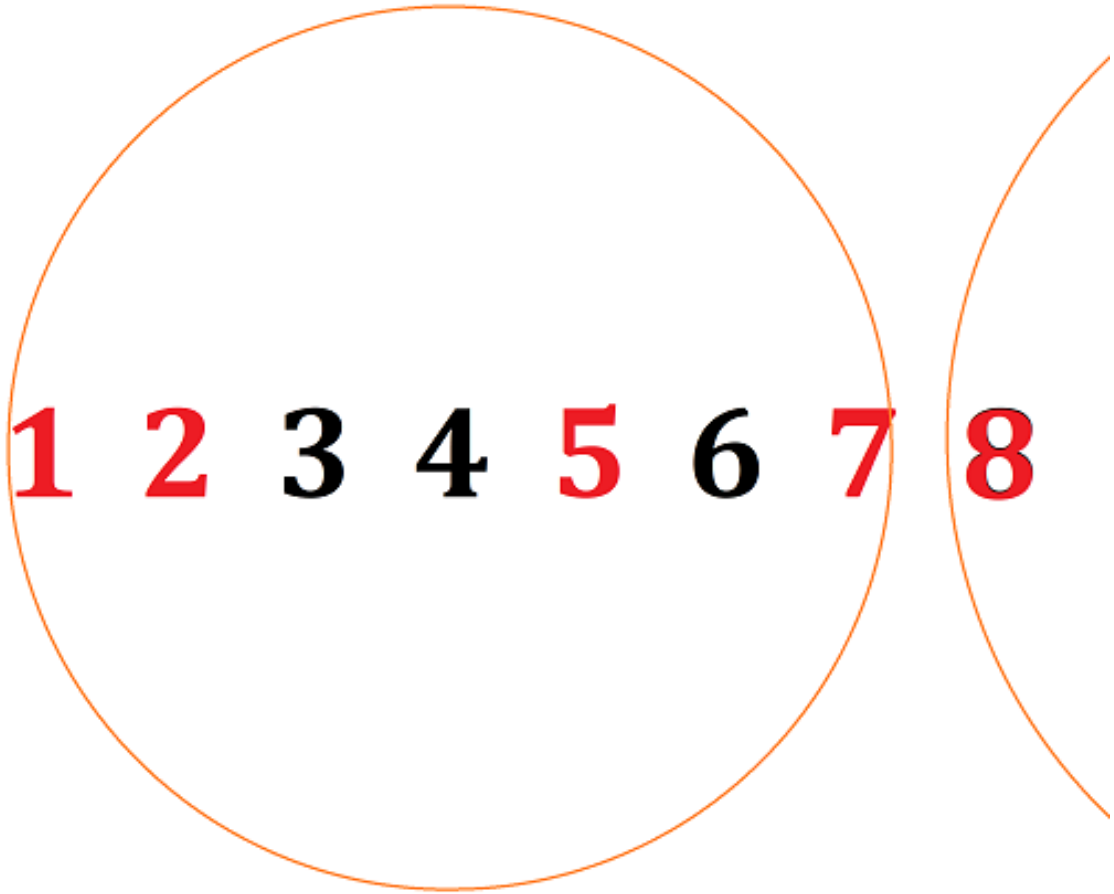
由於我們固定直徑時，要檢查需要多少基地台，方式會很簡單

先將必須覆蓋的點排序，我們每次都從最左邊的點開始放基地台

範例如下圖:

二分搜尋的另外一種題目

假設必須覆蓋點為 $\{1, 2, 5, 7, 8\}$，直徑為 6



二分搜尋的另外一種題目

當我們知道如何尋找某個特定直徑符不符合答案後

我們可以另外觀察到一點，也就是需要 $<= k$ 個基地台的直徑有**單調性**

又或者說，**當直徑大於某個特定直徑** x 時

我們就只需要 $<= k$ 個基地台即可覆蓋所有點

二分搜尋的另外一種題目

因此，我們就可以使用**二分搜尋法**解決這個題目

(check 是檢查該直徑是否能滿足用 $<= k$ 個基地台覆蓋)

```
int l = 1, r = 1e9;

while(l+1 < r){
    int m = (l+r)/2;

    if(check(m)) r = mid;
    else l = mid + 1;
}

if(check(l)) cout << l << "\n";
else cout << r << "\n";
```

二分搜尋的另外一種題目

應該能稍微感覺到二分搜尋法的神奇之處吧!

只要改變一下想法，就可以用二分搜解決了!

二分搜的幾種寫法

第一種: 左閉右閉 ($[l, r]$)

```
int l = 1, r = 1e9;

while(l < r){
    int m = (l+r)/2;

    if(check(m)) r = mid;
    else l = mid + 1;
}

cout << l << "\n";
```

二分搜的幾種寫法

第二種: 左閉右開 ($[l, r)$)

```
int l = 1, r = 1e9;

while(l+1 < r){
    int m = (l+r)/2;

    if(check(m)) r = mid;
    else l = mid;
}

cout << l << "\n";
```

二分搜的幾種寫法

第三種: 倍增法二分搜 (Binary Lifting)

```
int now = 0, r = 1e9;

for(int i = LOG; i >= 0; i--){
    if(now+(1<<i) <= r && check(now+(1<<i)))
        now += (1<<i);
}
```

二分搜的幾種寫法

另外提醒一下，如果是在一個陣列二分搜，STL 有內建

```
int pos = lower_bound(arr,arr+n,x)-arr;
```

而在 vector 或 deque 等中的寫法如下

```
int pos = lower_bound(v.begin(),v.end(),x)-v.begin();
```

二次函數的極值?

經典問題

給你一個二次函數 $f(x) = ax^2 + bx + c$，詢問他的最大或最小值

二次函數的極值?

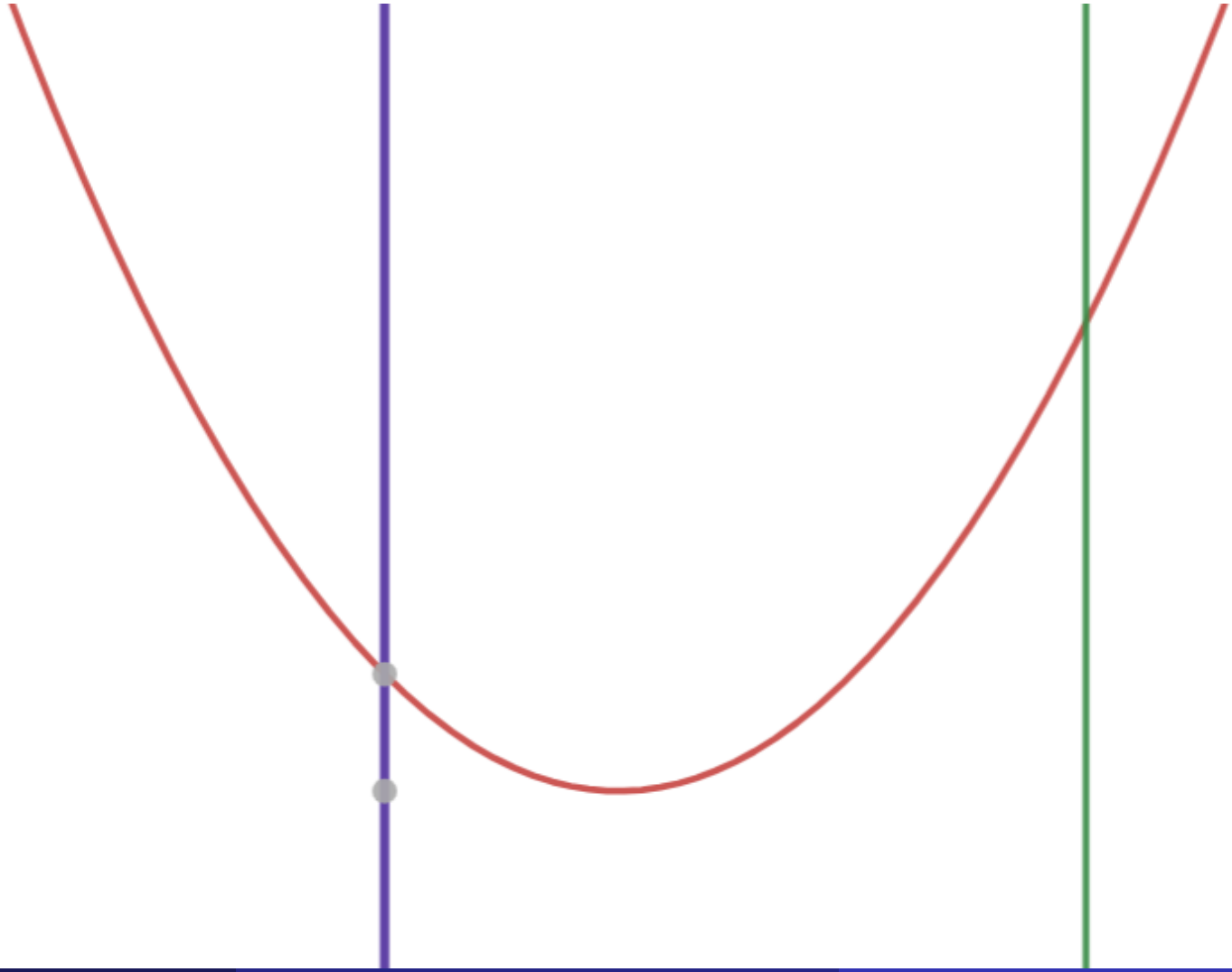
遇到這個問題的時候怎麼辦呢?

二次函數不像我們在做二分搜時那樣容易，他會遞增再遞減

那這樣怎麼做呢?

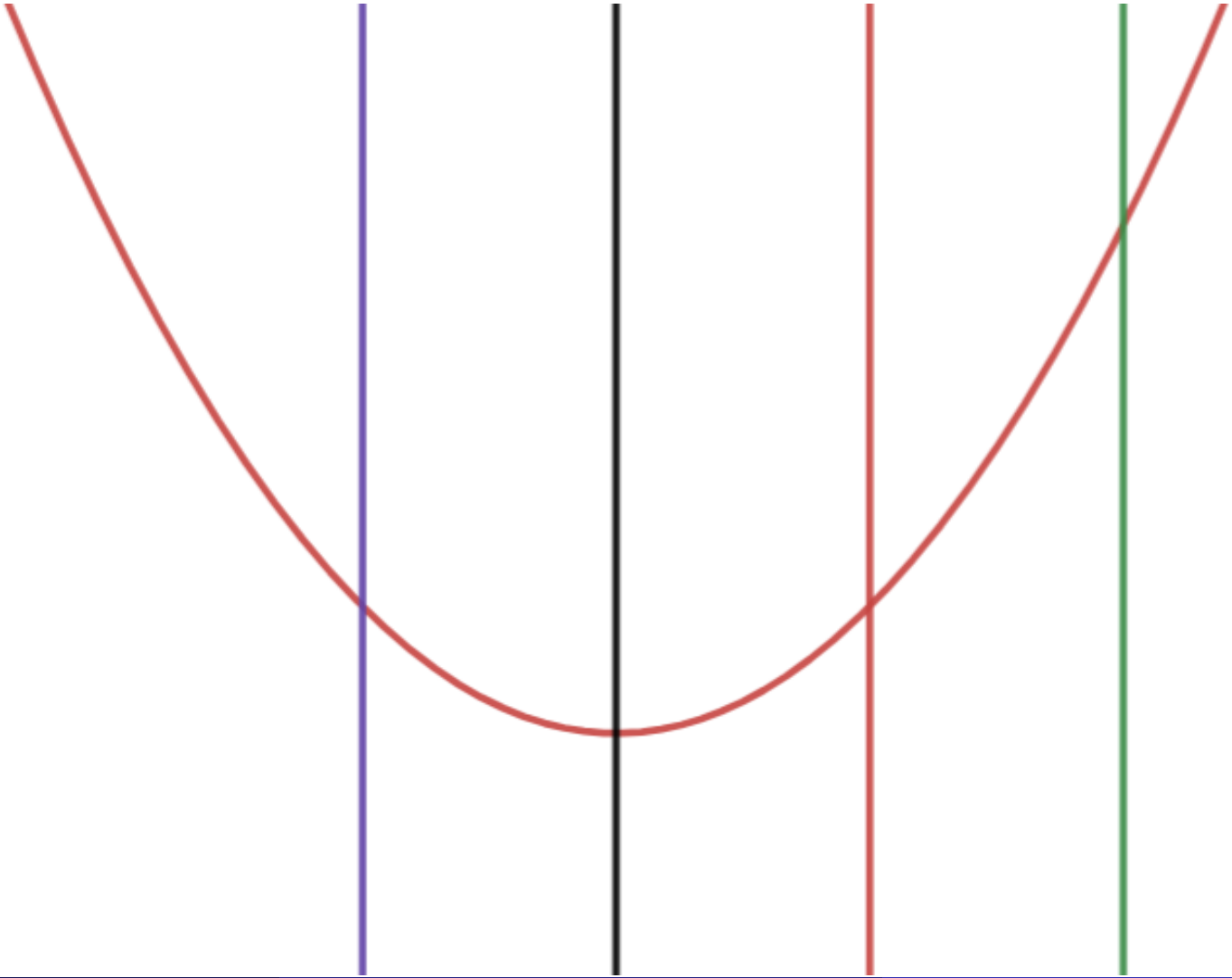
二次函數的極值?

假設我們要尋找的區間如下圖 $[l, r]$



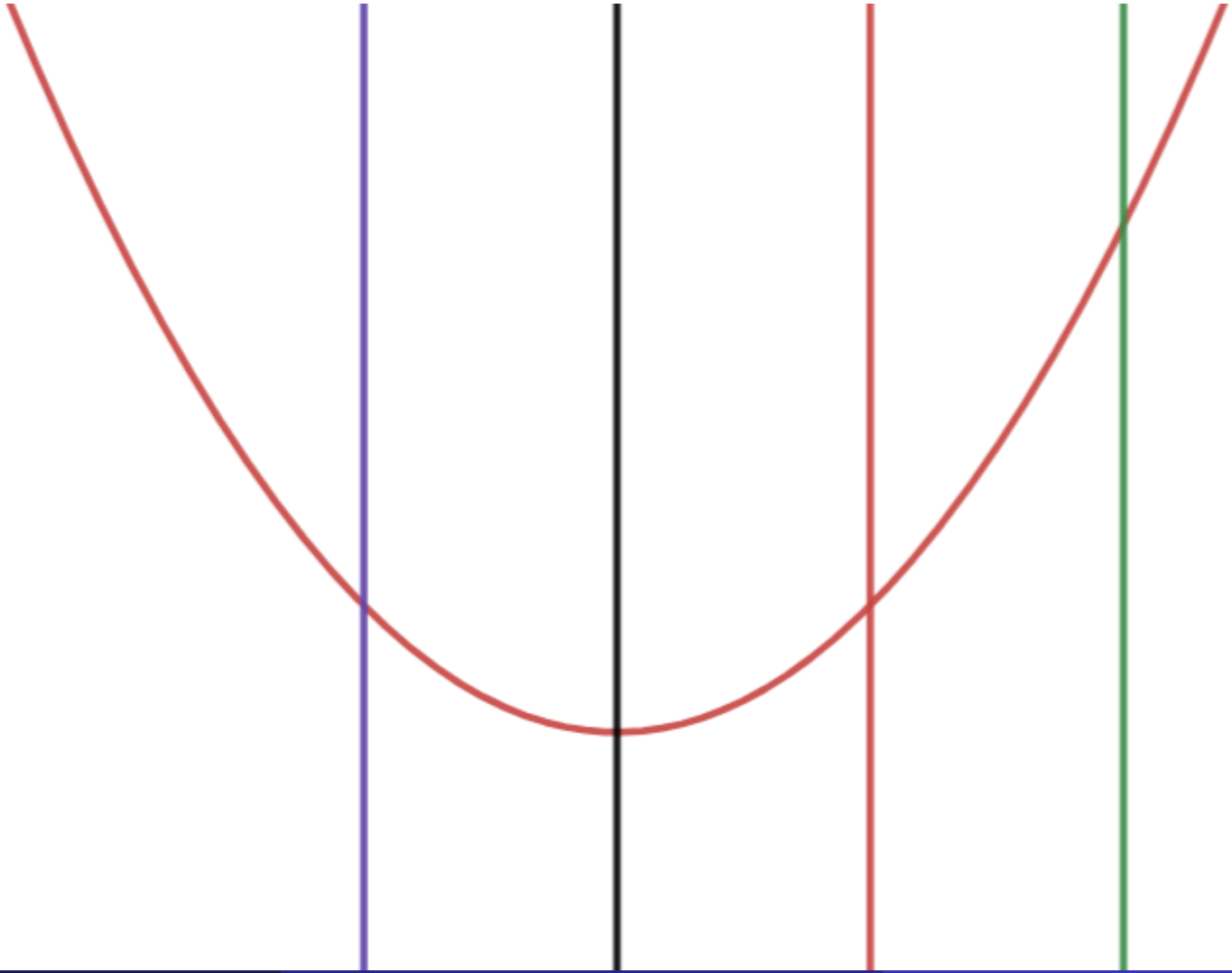
二次函數的極值?

我們可以在這個區間多切出兩條線，平分成三等份



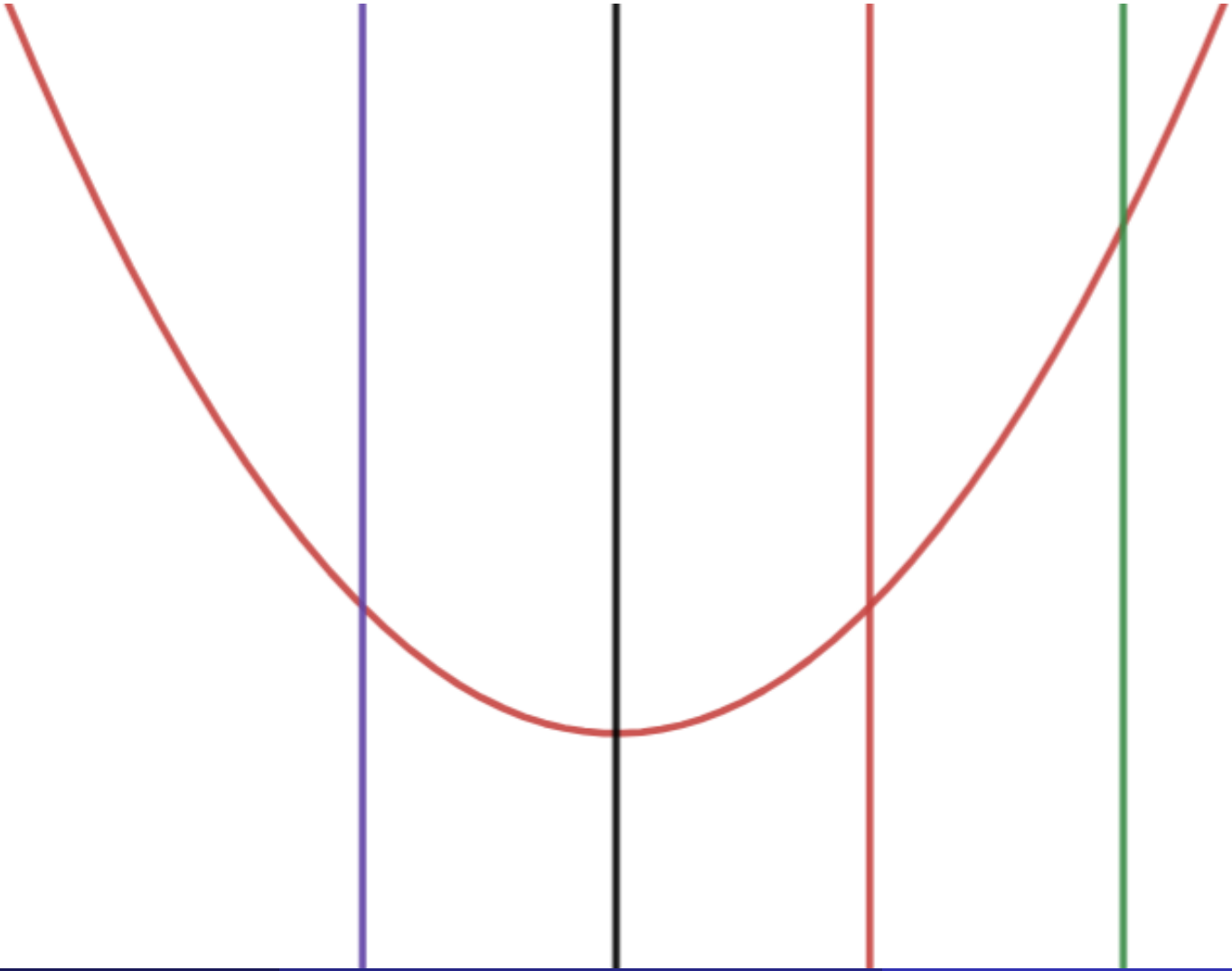
二次函數的極值?

這樣中間我們會有兩個值 ml, mr



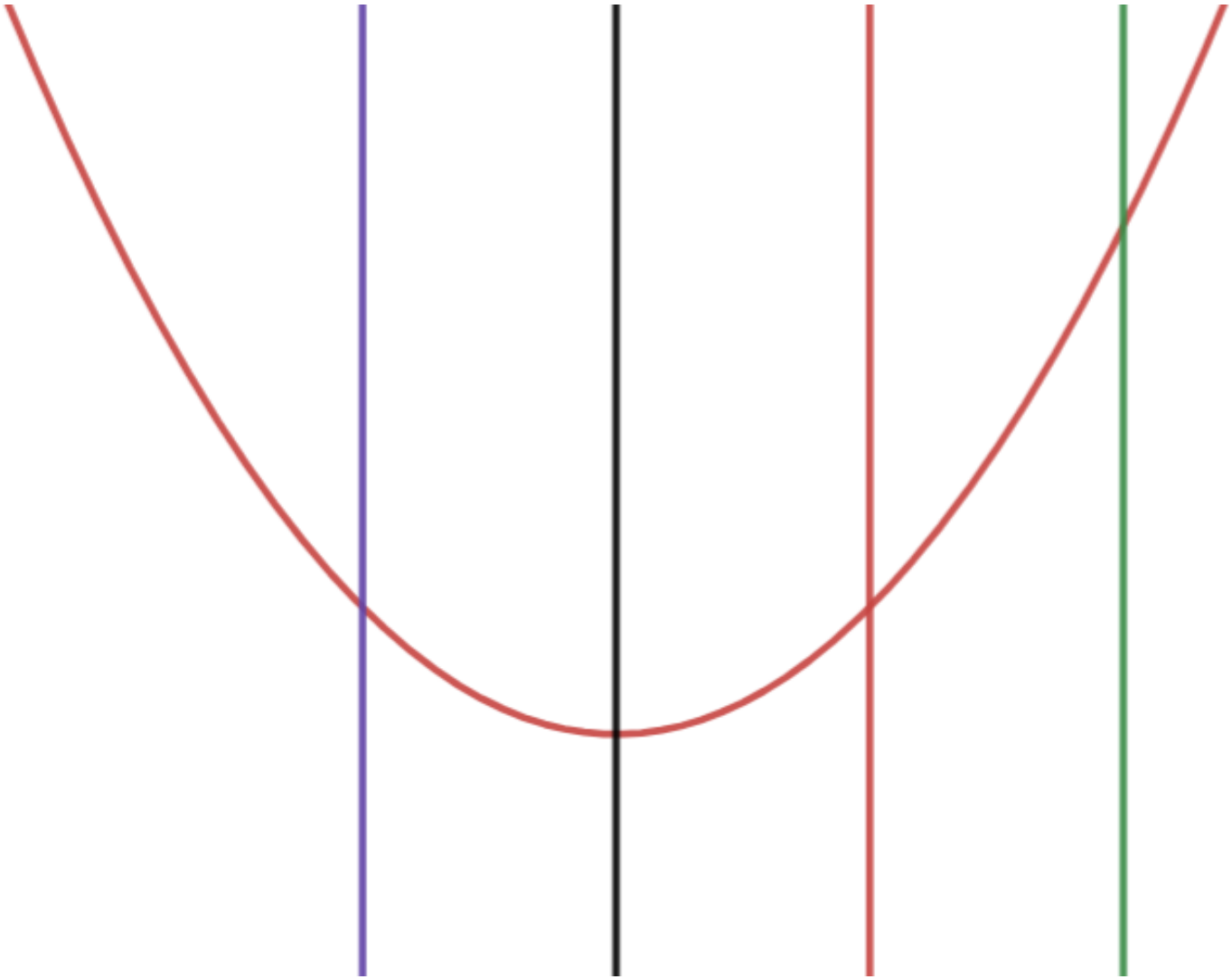
二次函數的極值?

當 $f(ml) > f(mr)$，我們就把 $l = ml$，反之，$r = mr$



二次函數的極值?

持續這樣進行之後，我們就會找到極值了!



二次函數的極值?

這樣的做法我們稱為**三分搜**

實作方式如下:

```
double f(double x){
    return x*x-3*x+1;
}

signed main(){
    fastio


    double l = 0, r = 100;
    for(int i = 0;i < 100;i++){
        double ml = l+(r-l)/3;
        double mr = r-(r-l)/3;
        if(f(ml) > f(mr)) l = ml;
        else r = mr;
    }

    cout << l << "\n";
}
```

二次函數的極值?

實作方式如下:

```
double f(double x){
    return x*x-3*x+1;
}

signed main(){
    fastio


    double l = 0, r = 100;
    for(int i = 0;i < 100;i++){
        double ml = l+(r-l)/3;
        double mr = r-(r-l)/3;
        if(f(ml) > f(mr)) l = ml;
        else r = mr;
    }

    cout << l << "\n";
}
```

搜尋的方式?

我們剛剛講的東西都比較偏向在某個有單調性或凹凸函數上找極值

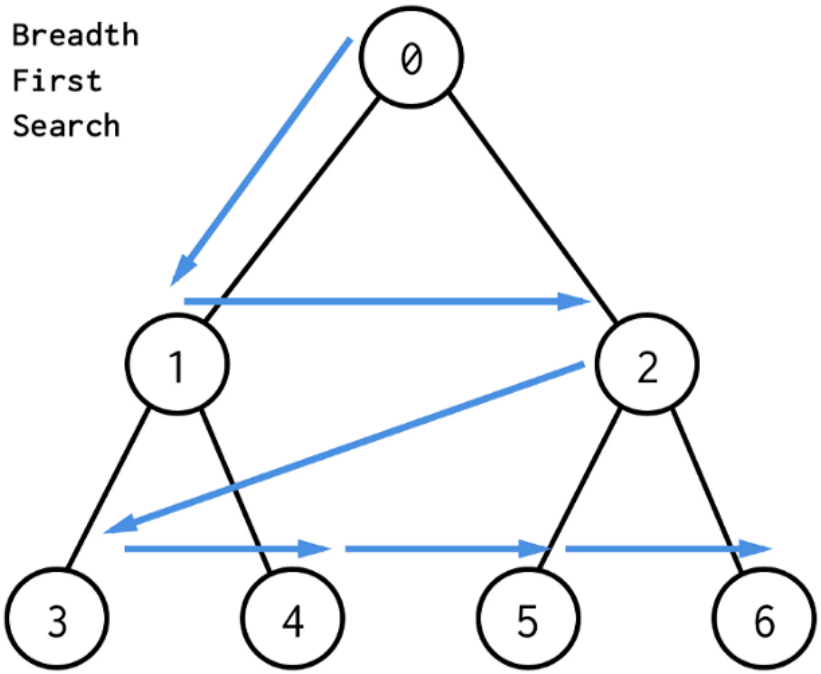
那麼有其他搜尋方法嗎?

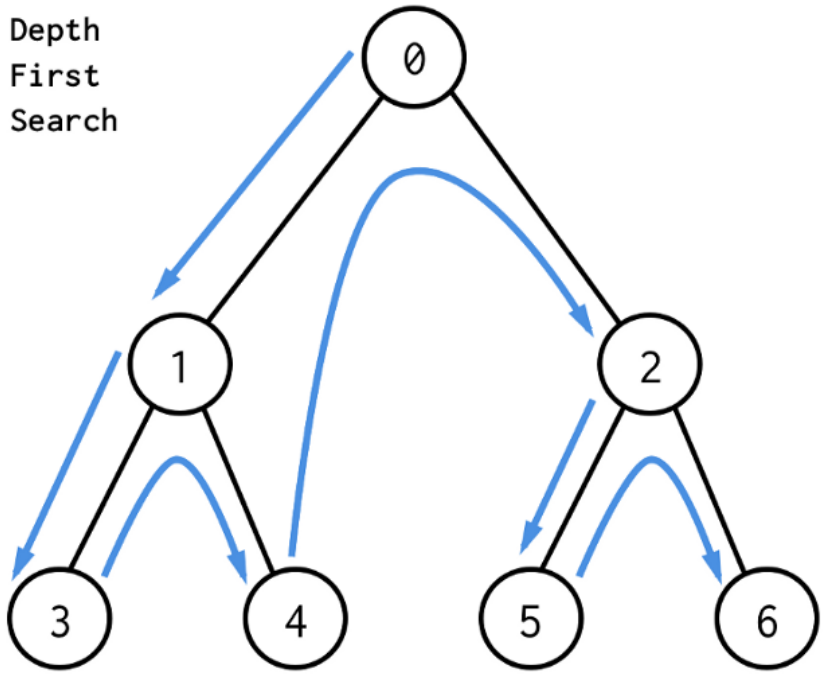
搜尋的方式?

而這裡我們來介紹兩種不同的搜尋方式:

- DFS (深度優先搜尋)
- BFS (廣度優先搜尋)

搜尋的方式?





搜尋的方式?

DFS 就是依深度往下去做搜尋

BFS 則是依照近的點開始做

搜尋的方式?

在實作方面，DFS 常用遞迴的方式模擬，而 BFS 則是用 queue 來進行

可以用搜尋的問題?

DFS

- 數獨
- 八皇后
- 各種需要尋找答案的題目

BFS

- 量杯問題
- 0/1 最短路
- 各種與最少步數有關的問題

數獨

而我們今天要來討論「**數獨**」

給你一個 9×9 的棋盤，問你是否能用 1 到 9 依照規則填滿棋盤

- 每一行，每一列數字不重複
- 每 3×3 的數字不重複

數獨

SUDOKU

8		6			3		9	
	4			1			6	8
2			8	7				5
1		8			5		2	
	3		1				5	
7		5		3		9		
	2	1			7		4	
6				2		8		
	8	7	6		4			3

ANSWER:

8	7	6	5	4	3	1	9	2
5	4	3	2	1	9	7	6	8
2	1	9	8	7	6	4	3	5
1	9	8	7	6	5	3	2	4
4	3	2	1	9	8	6	5	7
7	6	5	4	3	2	9	8	1
3	2	1	9	8	7	5	4	6
6	5	4	3	2	1	8	7	9
9	8	7	6	5	4	2	1	3

數獨

會發現這個問題，如果我們要去找到他的答案，應該很困難吧

總共的可能性有 9^{9^9} 這麼多種可能性，電腦根本算不完阿

數獨

不過這個問題，因為有了數字不重複的限制

一旦我們發現我們填的數字絕對不符合規則時，我們就不繼續，並收回上一步

這樣的作法，我們稱為**回朔法** (Backtracking)

數獨

實際按照這個方式去找答案，其實總共要花費的次數並不會太多!

因此，我們可以利用暴搜的方式找到一組答案

數獨

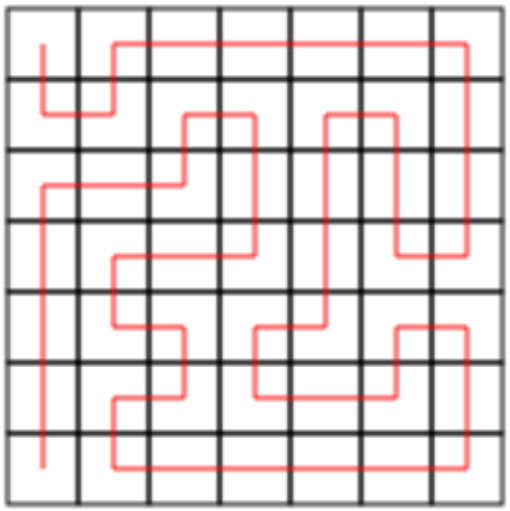
新手在回朔法的題目容易會寫出很多的 bug，建議大家都要實際練習看看數獨

Grid Paths

Grid Paths (CSES)

有一個 7×7 的網格，給你字串 (表示著第幾步要往哪個方向走)，問總共有幾種路徑可以從左上走到左下並走完所有網格?

For example, the path



corresponds to the description DRURRRRDDDLUULDDDLDRRURDDLLLLLURULURRUULDLLDDDD.

Grid Paths

根據這個問題，我們來談談**暴搜剪枝**這件事情

Grid Paths

你可能會想，如果我們直接去搜尋所有的路徑，當步數到的時候，根據他的方式走，這樣能不能過?

Grid Paths

很可惜的，直接這樣做，會發現我們無法在時間內做完

Test results

test	verdict	time	
#1	ACCEPTED	0.16 s	»
#2	ACCEPTED	0.01 s	»
#3	ACCEPTED	0.05 s	»
#4	ACCEPTED	0.35 s	»
#5	ACCEPTED	0.12 s	»
#6	ACCEPTED	0.21 s	»
#7	ACCEPTED	0.87 s	»
#8	ACCEPTED	0.17 s	»
#9	ACCEPTED	0.04 s	»
#10	ACCEPTED	0.56 s	»
#11	TIME LIMIT EXCEEDED	--	»
#12	TIME LIMIT EXCEEDED	--	»
#13	TIME LIMIT EXCEEDED	--	»
#14	TIME LIMIT EXCEEDED	--	»
#15	ACCEPTED	0.28 s	»
#16	ACCEPTED	0.16 s	»
#17	ACCEPTED	0.08 s	»
#18	ACCEPTED	0.13 s	»
#19	ACCEPTED	0.01 s	»
#20	TIME LIMIT EXCEEDED	--	»

Grid Paths

那我們想想，如果我們在走完所有格子前走到終點，我們是不是就能停止?

Grid Paths

那我們想想，如果我們在走完所有格子前走到終點，我們是不是就能停止?

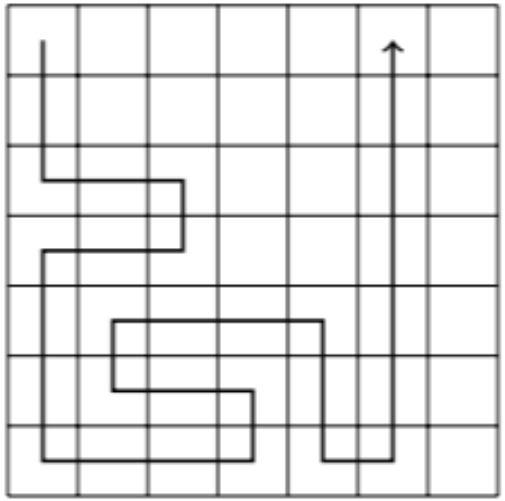
Using this observation, we can terminate the search immediately if we reach the lower-right square too early.

- running time: 119 seconds
- number of recursive calls: 20 billion

很可惜的還是過不了

Grid Paths

另外一個想法，如果我們無法往前走了，但左右都能走，那麼我們也可以直接停止

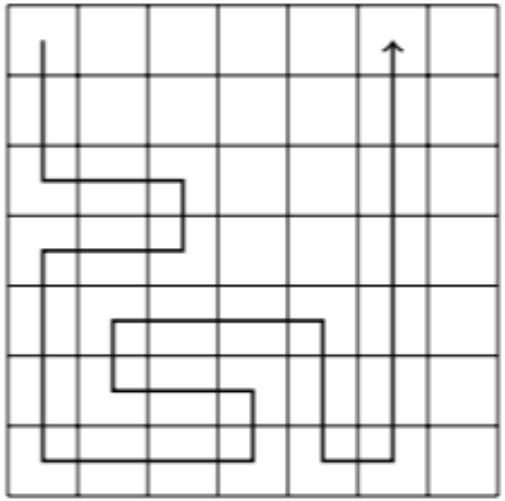


In this case, we cannot visit all squares anymore, so we can terminate the search. This optimization is very useful:

- running time: 1.8 seconds
- number of recursive calls: 221 million

Grid Paths

另外一個想法，如果我們無法往前走了，但左右都能走，那麼我們也可以直接停止



In this case, we cannot visit all squares anymore, so we can terminate the search. This optimization is very useful:

- running time: 1.8 seconds
- number of recursive calls: 221 million

Grid Paths

經過上述幾種**剪枝**方式之後，就可以通過這個問題了

這就是暴搜剪枝的神奇之處!

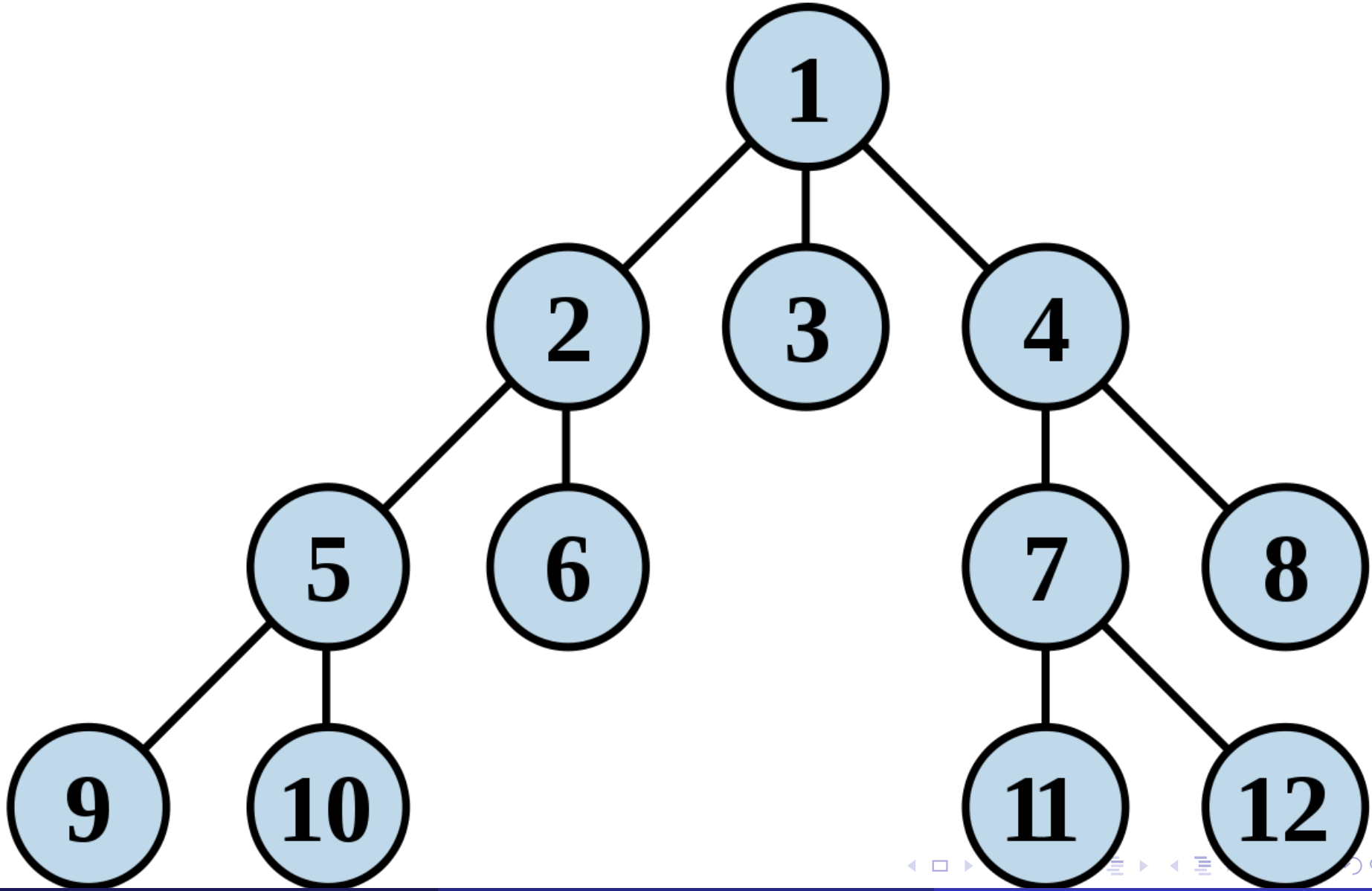
BFS

接著我們要來講講 BFS 這個東西，基本上不論是搜尋中的 BFS 還是之後我們會提到的圖論中的 BFS，都會需要使用到 queue 這個東西，利用 queue 的「First In First Out」，最先到達的狀態就要最早走

BFS

接著我們要來講講 BFS 這個東西，基本上不論是搜尋中的 BFS 還是之後我們會提到的圖論中的 BFS，都會需要使用到 queue 這個東西，利用 queue 的「First In First Out」，最先到達的狀態就要最早走

BFS



BFS

照著這樣的方式做之後，當我們找到某個狀態時，一定是由最少步數走到

BFS

接著讓我們來看看他的經典題「**倒水問題**」

倒水問題

倒水問題

給你 n $(n \leq 5)$ 個量杯，每個量杯有各自的刻度。對於這些量杯，每一步你可以做以下三種操作:

1. 裝滿量杯
2. 把量杯的水倒掉
3. 把 A 量杯的水倒給 B，直到 A 是空的或 B 是滿的

問你是否能量出某個特定的水量 x，如果可以，請輸出**最少步數**

倒水問題

倒水問題

給你 n $(n \leq 5)$ 個量杯，每個量杯有各自的刻度。對於這些量杯，每一步你可以做以下三種操作:

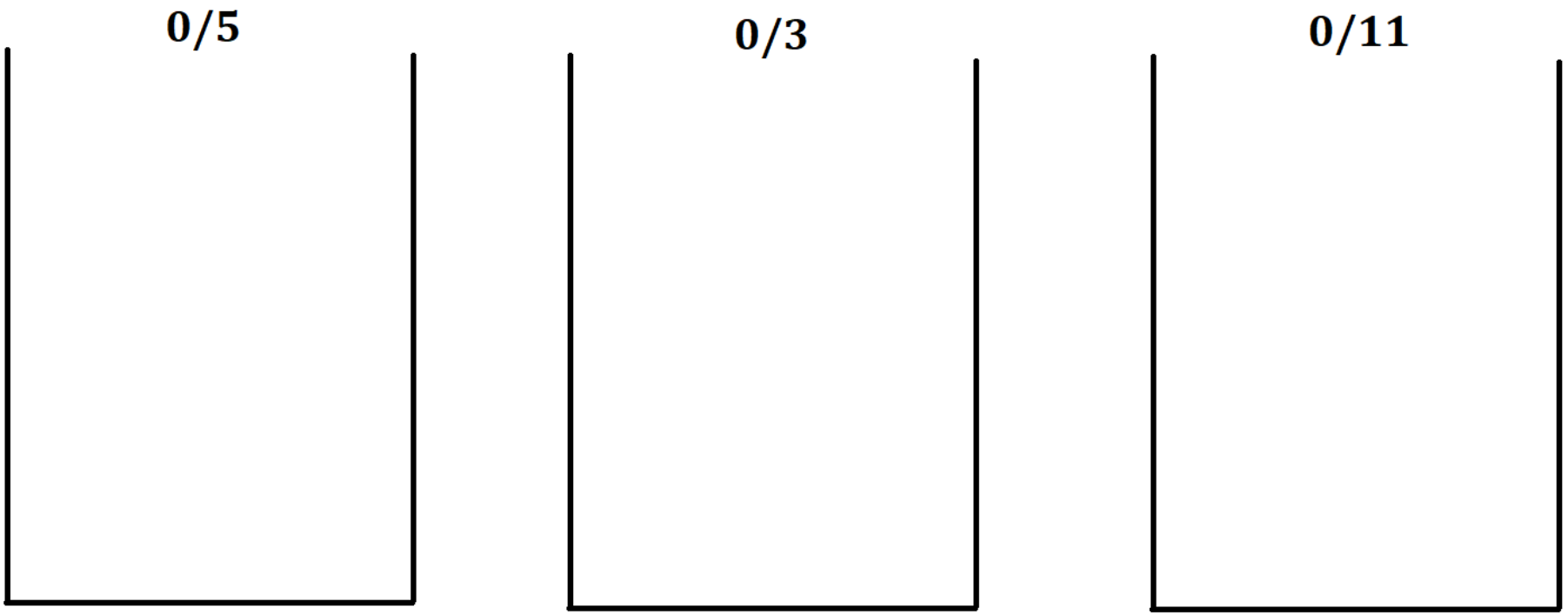
1. 裝滿量杯
2. 把量杯的水倒掉
3. 把 A 量杯的水倒給 B，直到 A 是空的或 B 是滿的

問你是否能量出某個特定的水量 x，如果可以，請輸出**最少步數**

由於他問的是**最少步數**，我們可以使用 BFS 來尋找這個答案

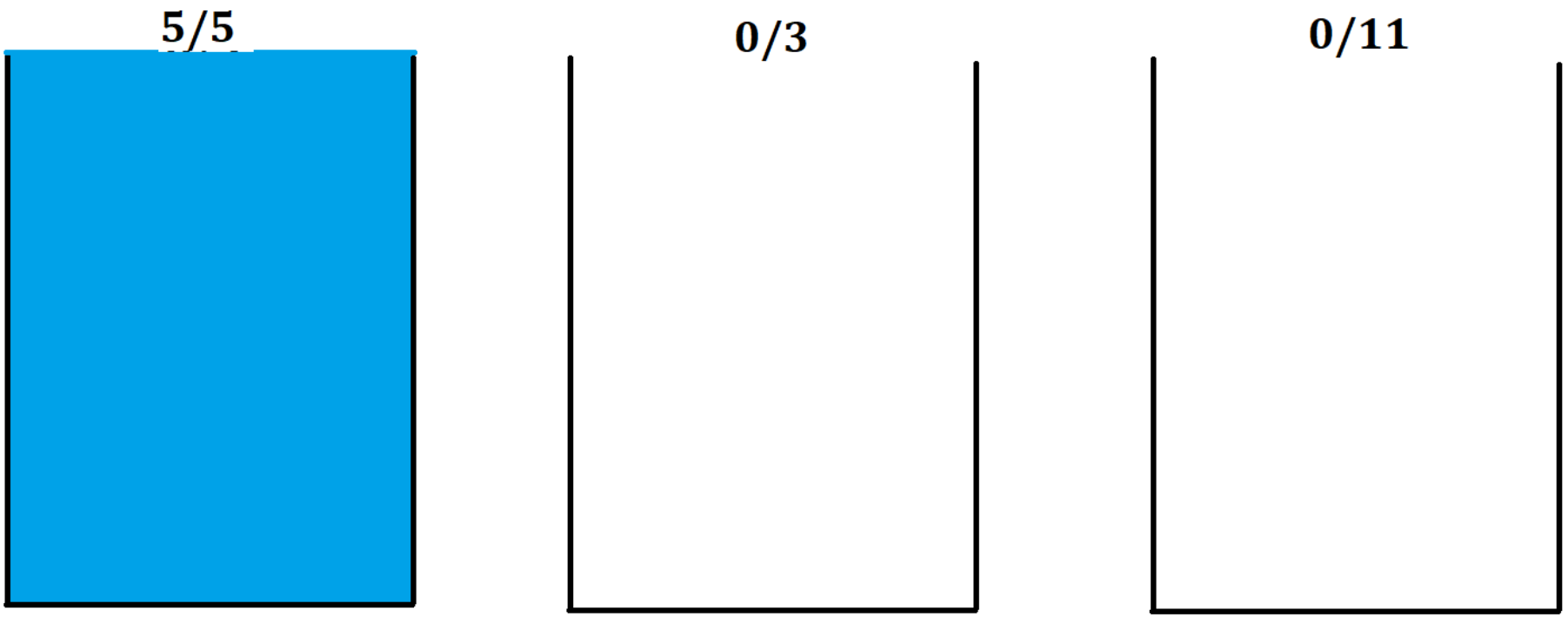
倒水問題

首先，我們要先知道，這個問題的初始狀態是每個量杯都沒有裝水



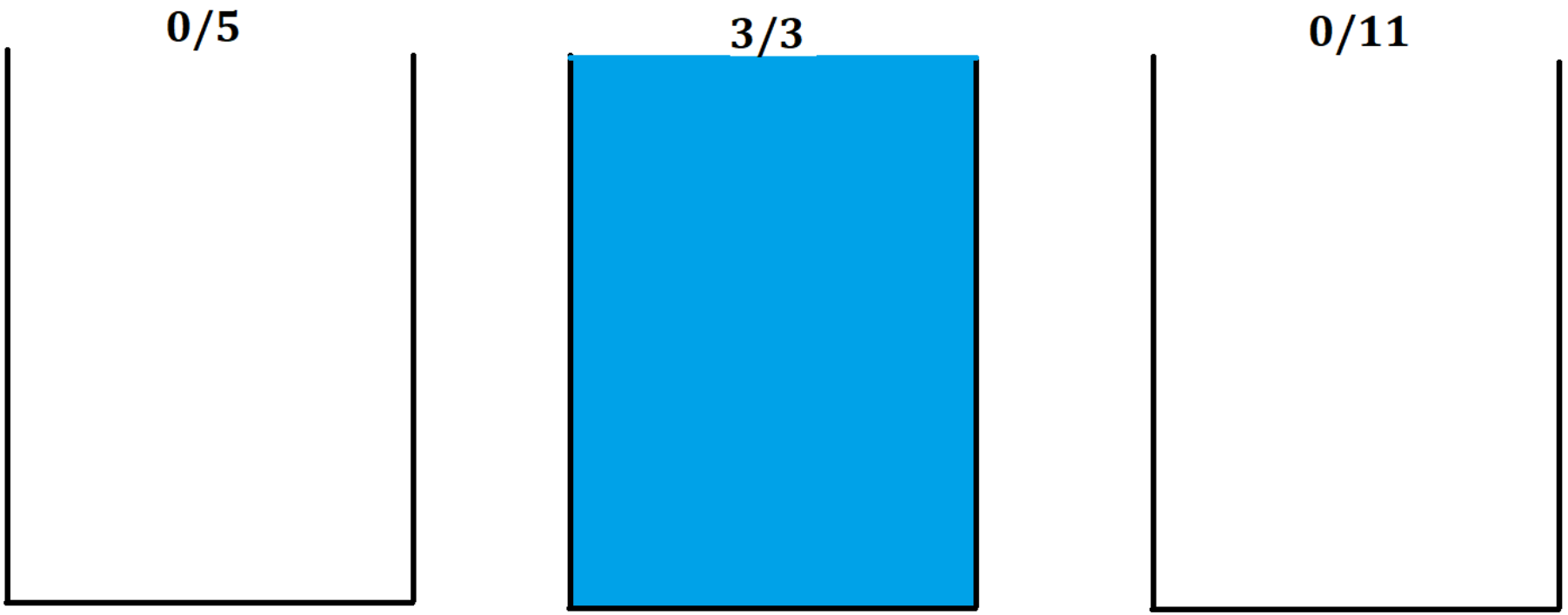
倒水問題

第一種可能性



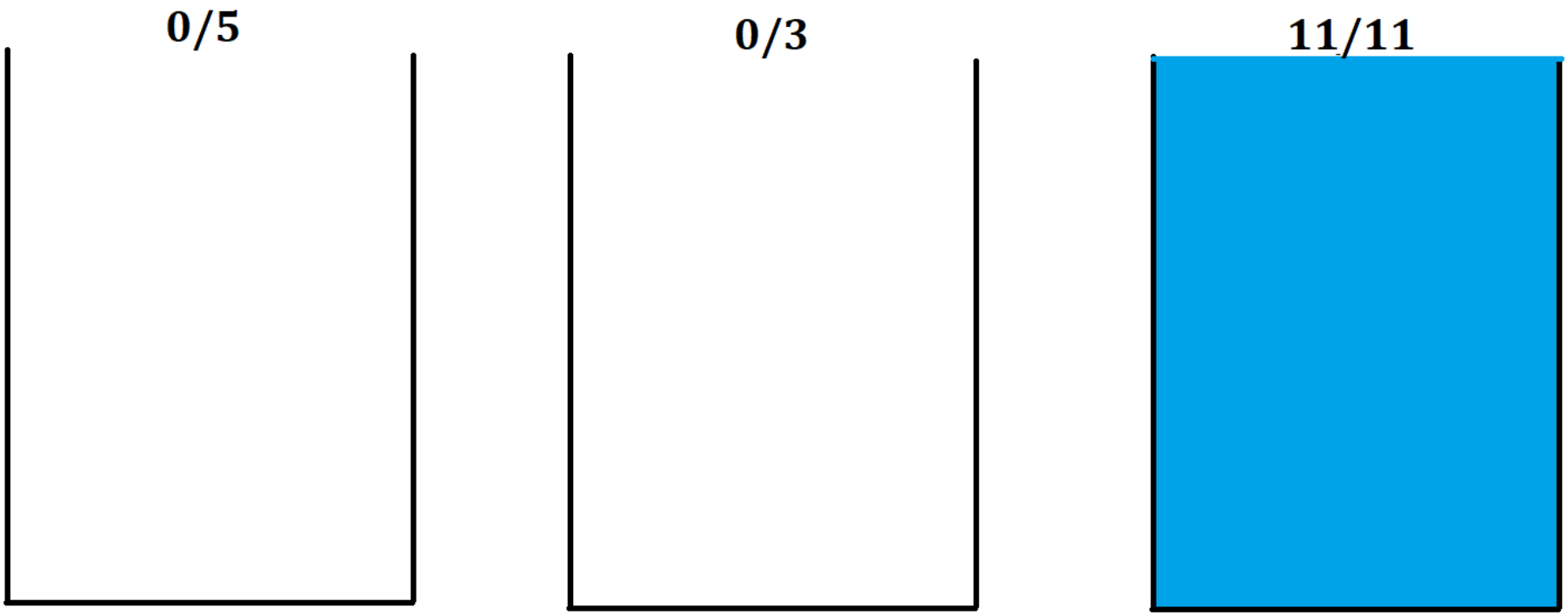
倒水問題

第二種可能性



倒水問題

第三種可能性



倒水問題

照著這樣做，每個狀態都去分別做這幾種操作，最後就能找到我們要的答案了。

但是萬一沒有答案怎麼判斷呢?

我們這裡可以先引用之後在講數學會講到的**貝祖定理**，能不能量出 x 若且為若 x 是所有量杯的刻度的最大公因數的倍數

照這樣進行，最後就能找到答案了

枚舉

講完搜尋之後，我們要來談談**枚舉**

讓我們來看看一個簡單的問題

枚舉

經典問題

給你一個 n 項的陣列，問總和最大的子陣列的總和為何?

枚舉

經典問題

給你一個 n 項的陣列，問總和最大的子陣列的總和為何?

看到這個題目，你可能會想說，那麼我去**枚舉開始和結束**的點不就好了?

枚舉

經典問題

給你一個 n 項的陣列，問總和最大的子陣列的總和為何?

看到這個題目，你可能會想說，那麼我去**枚舉開始和結束**的點不就好了?

枚舉

經典問題

給你一個 n 項的陣列，問總和最大的子陣列的總和為何?

```
int ans = 0;
for(int i = 0;i < n;i++){
    for(int j = i;j < n;j++){
        int sum = 0;
        for(int k = i;k <= j;k++){
            sum += arr[k];
        }
        ans = max(sum,ans);
    }
}
cout << ans << "\n";
```

枚舉

經典問題

給你一個 n 項的陣列，問總和最大的子陣列的總和為何?

還不夠好! 如果我們固定起始點開始找呢?

枚舉

經典問題

給你一個 n 項的陣列，問總和最大的子陣列的總和為何?

```
int ans = 0;
for(int i = 0;i < n;i++){
    int sum = 0;
    for(int j = i;j < n;j++){
        sum += arr[j];
        ans = max(ans,sum);
    }
}
cout << ans << "\n";
```

枚舉

背包問題

有 n $(n \leq 20)$ 個物品放在桌上，你有一個容量為 C 的背包，問最多能裝幾個物品?

枚舉

背包問題

有 n $(n \leq 20)$ 個物品，每個物品有各自的重量 w_i, 放在桌上，你有一個容量為 C 的背包，問最多能裝幾個物品?

這個問題問學過 **DP(動態規劃)** 的人應該都會認為很容易，不過僅僅是在物品的總和有限制時，當物品的重量總和沒有被限制時，背包問題會是 **NP 問題**

枚舉

背包問題

有 n $(n \leq 20)$ 個物品，每個物品有各自的重量 w_i, 放在桌上，你有一個容量為 C 的背包，問最多能裝幾個物品?

作法很簡單，今天每個物品只有兩種可能性，**拿**或**不拿**，答我們可以直接用 **DFS** 去跑所有的物品，每個物品考慮拿與不拿

枚舉

```
void solve(int i, int sum, int cnt){
    if(sum <= c) ans = max(cnt,ans);
    if(i == n) return;
    solve(i,sum+w[i],cnt+1);
    solve(i,sum,cnt);
}
```

時間複雜度: $O(2^n)$

枚舉

這樣的寫法可以輕鬆的解決這個問題，不過有另外一種不用遞迴的做法

但是我們要先提一下所謂的**位元運算**

枚舉

符號	運算
&	AND (當同一位的數字相同時為 1)
|	OR (當兩個數字同一位至少有一個 1 時為 1)
^	XOR (當兩個數字的同一位只有一個 1 時為 1)
<<	左移一位 (等同於 $\times 2$)
>>	右移一位 (等同於 $/2$)

各種運算

枚舉

因為二進位的性質，小於 2^n 次方的數字，總共位數會是 n，而且每個數字都會分別屬於某個特定的組別，我們可以去看每個數有哪幾位是 1，利用位元去枚舉我們要的答案，俗稱「**位元枚舉**」

枚舉

程式碼如下:

```
for(int mask = 0; mask < (1<<n); mask++){
    int sum = 0, cnt = 0;
    for(int i = 0;i < n;i++){
        if(mask&(1<<i)) sum += w[i], cnt++;
    }
    if(sum <= c){
        ans = max(ans, cnt);
    }
}
```