分治 ／ Divide and Conquer

Gino

2022 新化高中 x 嘉義高中 x 薇閣高中資研社暑期培訓營隊

內容大綱

- 快速冪
- 分治簡介
- Merge Sort
- 分析遞迴的工具
  - 遞迴樹
  - 主定理
- 最大連續和
- Karatsuba's Algorithm (多項式乘法)
- 逆序數對
- 最近點對
- 分治再談

健康生技公司培養綠藻以製作「綠藻粉」，再經過後續的加工步驟，製成綠藻相關的保健食品。已知該公司製作每 **1** 公克的「綠藻粉」需要 **60** 億個綠藻細胞。

請根據上述資訊回答下列問題，完整寫出你的解題過程並詳細解釋：

(1) 假設在光照充沛的環境下，**1** 個綠藻細胞每 **20** 小時可分裂成 **4** 個綠藻細胞，且分裂後的細胞亦可繼續分裂。今從 **1** 個綠藻細胞開始培養，若培養期間綠藻細胞皆未死亡且培養環境的光照充沛，經過 **15** 天後，共分裂成 4^k 個綠藻細胞，則 k 之值為何？

(2) 承 (1)，已知 **60** 億介於 2^{32} 與 2^{33} 之間，請判斷 4^k 個綠藻細胞是否足夠製作 **8** 公克的「綠藻粉」？

- 你很快的算出第一小題 $k = 18$，結果緊張之下你竟然忘了 $4^{18} = 2^{36}$。

健康生技公司培養綠藻以製作「綠藻粉」，再經過後續的加工步驟，製成綠藻相關的保健食品。已知該公司製作每 **1** 公克的「綠藻粉」需要 **60** 億個綠藻細胞。

請根據上述資訊回答下列問題，完整寫出你的解題過程並詳細解釋：

(1) 假設在光照充沛的環境下，**1** 個綠藻細胞每 **20** 小時可分裂成 **4** 個綠藻細胞，且分裂後的細胞亦可繼續分裂。今從 **1** 個綠藻細胞開始培養，若培養期間綠藻細胞皆未死亡且培養環境的光照充沛，經過 **15** 天後，共分裂成 4^k 個綠藻細胞，則 k 之值為何？

(2) 承 (1)，已知 **60** 億介於 2^{32} 與 2^{33} 之間，請判斷 4^k 個綠藻細胞是否足夠製作 **8** 公克的「綠藻粉」？

- 你很快的算出第一小題 $k = 18$，結果緊張之下你竟然忘了 $4^{18} = 2^{36}$。
- 於是你決定把 4^{18} 爆開，直接跟 480 億比大小。

會考數學

- $4^1 = 4$
- $4^2 = 16$
- $4^3 = 64$
- $\vdots$
- $4^{17} = 17179869184$
- $4^{18} = 68719476736$

- $4^1 = 4$
- $4^2 = 16$
- $4^3 = 64$
- $\vdots$
- $4^{17} = 17179869184$
- $4^{18} = 68719476736$
- 這樣一個一個乘要重複 18 次，太麻煩了！

會考數學

- 如果你已經算出 4^9 了，那把 4^9 再乘一次自己就會是 4^{18} 了！

會考數學

- 如果你已經算出 4^9 了，那把 4^9 再乘一次自己就會是 4^{18} 了！
- 乘法次數少了一半。

會考數學

- 如果你已經算出 4^9 了，那把 4^9 再乘一次自己就會是 4^{18} 了！
- 乘法次數少了一半。
- 4^9 也可以用切一半的方法乘：$4^9 = 4^4 \times 4^4 \times 4$

會考數學

- 如果你已經算出 4^9 了，那把 4^9 再乘一次自己就會是 4^{18} 了！
- 乘法次數少了一半。
- 4^9 也可以用切一半的方法乘：$4^9 = 4^4 \times 4^4 \times 4$
- 4^4 也可以用切一半的方法乘：$4^4 = 4^2 \times 4^2$

會考數學

- 如果你已經算出 4^9 了，那把 4^9 再乘一次自己就會是 4^{18} 了！
- 乘法次數少了一半。
- 4^9 也可以用切一半的方法乘：$4^9 = 4^4 \times 4^4 \times 4$
- 4^4 也可以用切一半的方法乘：$4^4 = 4^2 \times 4^2$
- 4^2 也可以用切一半的方法乘：$4^2 = 4^1 \times 4^1$

會考數學

- 如果你已經算出 4^9 了，那把 4^9 再乘一次自己就會是 4^{18} 了！
- 乘法次數少了一半。
- 4^9 也可以用切一半的方法乘：$4^9 = 4^4 \times 4^4 \times 4$
- 4^4 也可以用切一半的方法乘：$4^4 = 4^2 \times 4^2$
- 4^2 也可以用切一半的方法乘：$4^2 = 4^1 \times 4^1$
- 4^1 也可以用切一半... 呃不能再切了

會考數學

- 如果你已經算出 4^9 了，那把 4^9 再乘一次自己就會是 4^{18} 了！
- 乘法次數少了一半。
- 4^9 也可以用切一半的方法乘：$4^9 = 4^4 \times 4^4 \times 4$
- 4^4 也可以用切一半的方法乘：$4^4 = 4^2 \times 4^2$
- 4^2 也可以用切一半的方法乘：$4^2 = 4^1 \times 4^1$
- 4^1 也可以用切一半... 呃不能再切了
- 4^1 就是 4 啊！

會考數學

- $4^1 = 4$
- $4^2 = 4^1 \times 4^1 = 16$
- $4^4 = 4^2 \times 4^2 = 256$
- $4^9 = 4^4 \times 4^4 \times 4 = 262144$
- $4^{18} = 4^9 \times 4^9 = 68719476736$

- $4^1 = 4$
- $4^2 = 4^1 \times 4^1 = 16$
- $4^4 = 4^2 \times 4^2 = 256$
- $4^9 = 4^4 \times 4^4 \times 4 = 262144$
- $4^{18} = 4^9 \times 4^9 = 68719476736$
- 這樣只要做 5 次乘法，快多了！

指數運算

同樣的技巧可以套用到任意 a^b 的情況。

同樣的技巧可以套用到任意 a^b 的情況。

指數運算

給你兩個數字 a, b，請計算 a 的 b 次方等於多少。

先假設 C++ 不存在溢位問題。

指數運算

- 一直「切一半」的過程可以看成是某種遞迴。

指數運算

- 一直「切一半」的過程可以看成是某種遞迴。

$$a^b = \begin{cases} 1 & \text{if } b = 0 \\ a & \text{if } b = 1 \\ a^{\lfloor b/2 \rfloor} \times a^{\lfloor b/2 \rfloor} & \text{if } b \text{ 是偶數} \\ a^{\lfloor b/2 \rfloor} \times a^{\lfloor b/2 \rfloor} \times a & \text{otherwise} \end{cases}$$

指數運算

- 寫成 code：

```
int fastpow(int a, int b) {
    if (b == 0) return 1;
    if (b == 1) return a;

    int mid = fastpow(a, b/2);

    if (b % 2) return mid * mid * a;
    else return mid * mid;
}
```

指數運算

- 寫成 code：

```
int fastpow(int a, int b) {
    if (b == 0) return 1;
    if (b == 1) return a;

    int mid = fastpow(a, b/2);

    if (b % 2) return mid * mid * a;
    else return mid * mid;
}
```

- 複雜度分析：每次遞迴的時候會一直把 b 砍一半，直到 $b = 0$ 或 1，因此複雜度為 $O(\log b)$。

指數運算

- 寫成 code：

```
int fastpow(int a, int b) {
    if (b == 0) return 1;
    if (b == 1) return a;

    int mid = fastpow(a, b/2);

    if (b % 2) return mid * mid * a;
    else return mid * mid;
}
```

- 複雜度分析：每次遞迴的時候會一直把 b 砍一半，直到 $b = 0$ 或 1，因此複雜度為 $O(\log b)$。
- 這個技巧叫做「快速冪」（快速計算某個數字的冪次），是分治的經典應用。

什麼是分治？

- 了解快速冪之後，相信大家對分治有些感覺了。

什麼是分治？

- 了解快速冪之後，相信大家對分治有些感覺了。
- 分治，分而治之，分治不是一個特定的演算法，而是設計演算法的一種方法。

什麼是分治？

- 了解快速冪之後，相信大家對分治有些感覺了。
- 分治，分而治之，分治不是一個特定的演算法，而是設計演算法的一種方法。
- 分：Divide，把大問題分割成多個小問題。
- 治：Conquer，把小問題的答案合併起來得到大問題的解。

什麼是分治？

- 了解快速冪之後，相信大家對分治有些感覺了。
- 分治，分而治之，分治不是一個特定的演算法，而是設計演算法的一種方法。
- 分：Divide，把大問題分割成多個小問題。
- 治：Conquer，把小問題的答案合併起來得到大問題的解。
- 小問題通常和大問題形式相同，只是規模較小。

什麼是分治？

- 了解快速冪之後，相信大家對分治有些感覺了。
- 分治，分而治之，分治不是一個特定的演算法，而是設計演算法的一種方法。
- 分：Divide，把大問題分割成多個小問題。
- 治：Conquer，把小問題的答案合併起來得到大問題的解。
- 小問題通常和大問題形式相同，只是規模較小。
- 分割後的小問題如果還不夠小，則繼續遞迴分割下去，直到不能分割（遇到邊界條件）。而通常遇到邊界條件時，答案會非常明顯，所以也沒有繼續分割的必要。

什麼是分治？

- 了解快速冪之後，相信大家對分治有些感覺了。
- 分治，分而治之，分治不是一個特定的演算法，而是設計演算法的一種方法。
- 分：Divide，把大問題分割成多個小問題。
- 治：Conquer，把小問題的答案合併起來得到大問題的解。
- 小問題通常和大問題形式相同，只是規模較小。
- 分割後的小問題如果還不夠小，則繼續遞迴分割下去，直到不能分割（遇到邊界條件）。而通常遇到邊界條件時，答案會非常明顯，所以也沒有繼續分割的必要。
- 分治其實就是遞迴的延伸。

合併排序 (Merge Sort)

接下來介紹一個更為經典的分治算法。

TIOJ 1287 基礎排序

有一個長度為 n 的序列，請將這個序列由小排到大並輸出。
$n \le 10^6$

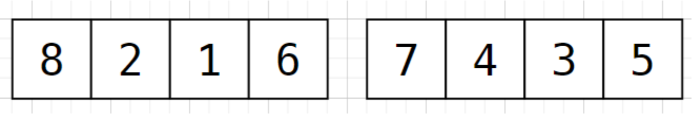
合併排序 (Merge Sort)

- 假設要排序的序列長這樣：

8	2	1	6	7	4	3	5

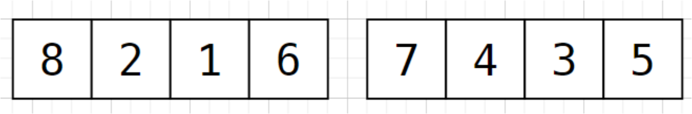
合併排序 (Merge Sort)

- 沿用快速冪「切一半」的想法，把整個序列切半：



合併排序 (Merge Sort)

- 沿用快速冪「切一半」的想法，把整個序列切半：



- 左半邊跟右半邊各為長度 $= 4$ 的序列。

合併排序 (Merge Sort)

- 沿用快速冪「切一半」的想法，把整個序列切半：

8	2	1	6		7	4	3	5

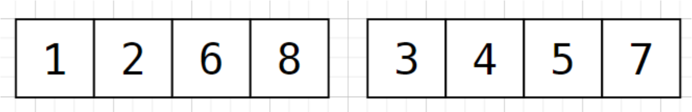
- 左半邊跟右半邊各為長度 $=4$ 的序列。
- 原本「排序 8 個數字」的問題被分解成兩個「排序 4 個數字」的子問題。

合併排序 (Merge Sort)

- 現在先不要管左右兩半是怎麼排序的，

合併排序 (Merge Sort)

- 現在先不要管左右兩半是怎麼排序的，
- 相信遞迴的力量，大膽假設左右兩邊都排序好了。



合併排序 (Merge Sort)

- 現在先不要管左右兩半是怎麼排序的，
- 相信遞迴的力量，大膽假設左右兩邊都排序好了。

1	2	6	8		3	4	5	7

- 現在要想辦法把左右兩半「合併」成一個排好的序列。

合併排序 (Merge Sort)

- 現在先不要管左右兩半是怎麼排序的，
- 相信遞迴的力量，大膽假設左右兩邊都排序好了。

1	2	6	8	3	4	5	7

- 現在要想辦法把左右兩半「合併」成一個排好的序列。
- 可以看成是我們要把數字填到以下的空格，並且要由小到大依序填好：

合併排序 (Merge Sort)

- 現在先不要管左右兩半是怎麼排序的，
- 相信遞迴的力量，大膽假設左右兩邊都排序好了。

1	2	6	8

3	4	5	7

- 現在要想辦法把左右兩半「合併」成一個排好的序列。
- 可以看成是我們要把數字填到以下的空格，並且要由小到大依序填好：

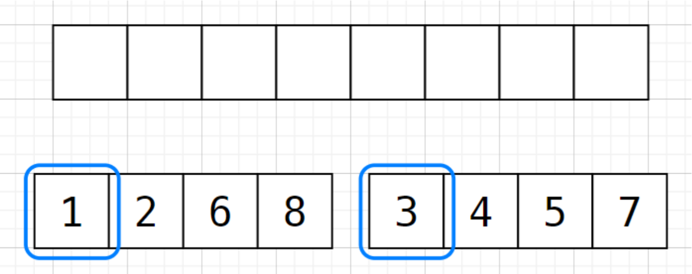
- 左右兩邊都有單調性 ⇒ 用雙指針合併！

合併排序 (Merge Sort)

- 第一格是最小的數字。

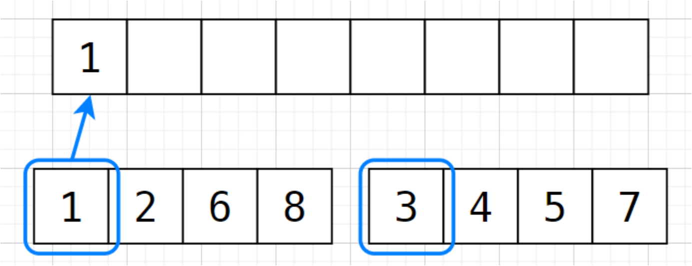
合併排序 (Merge Sort)

- 第一格是最小的數字。
- 最小的數字只可能是左半序列的開頭或右半序列的開頭。



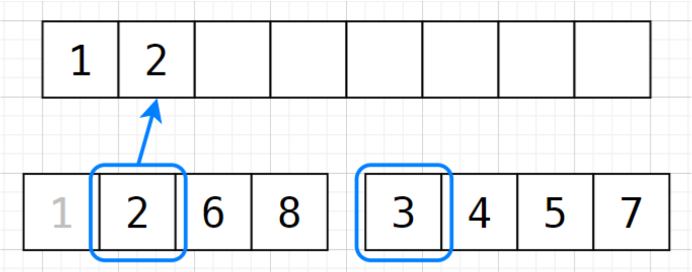
合併排序 (Merge Sort)

- 最小的數字只可能是左半序列的開頭或右半序列的開頭。
- 看誰比較小就填到第一格。



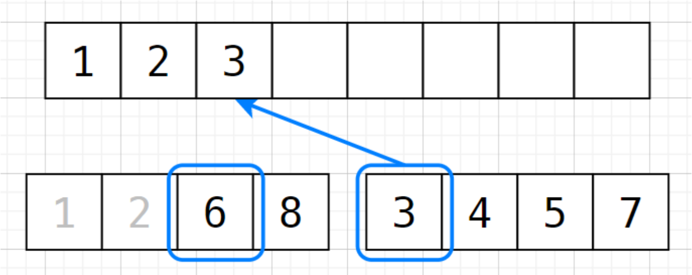
合併排序 (Merge Sort)

- 繼續填下去。



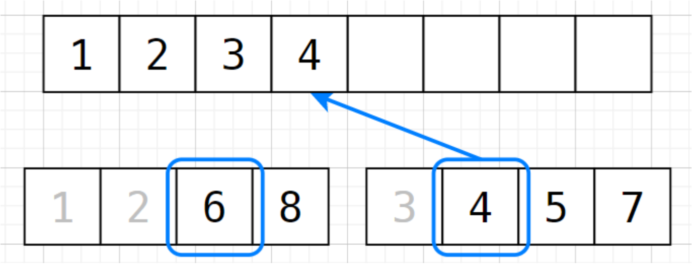
合併排序 (Merge Sort)

- 繼續填下去。



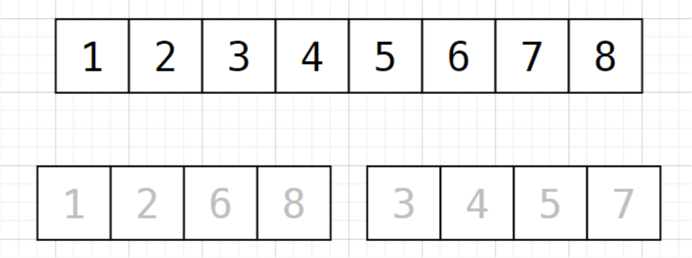
合併排序 (Merge Sort)

- 繼續填下去。



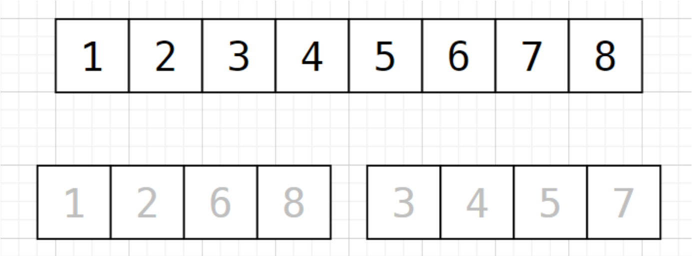
合併排序 (Merge Sort)

- 繼續填下去，直到所有數字都填好為止。



合併排序 (Merge Sort)

- 繼續填下去，直到所有數字都填好為止。



- 成功合併好兩個序列了！

合併排序 (Merge Sort)

合併排序演算法步驟

1. 邊界條件：序列長度為 $1 \Rightarrow$ 不用做任何事就已經排序好了。
2. 把序列從中間切成左右兩半。
3. 遞迴排序左半邊。
4. 遞迴排序右半邊。
5. 把左半邊跟右半邊的序列合併起來。

合併排序 (Merge Sort)

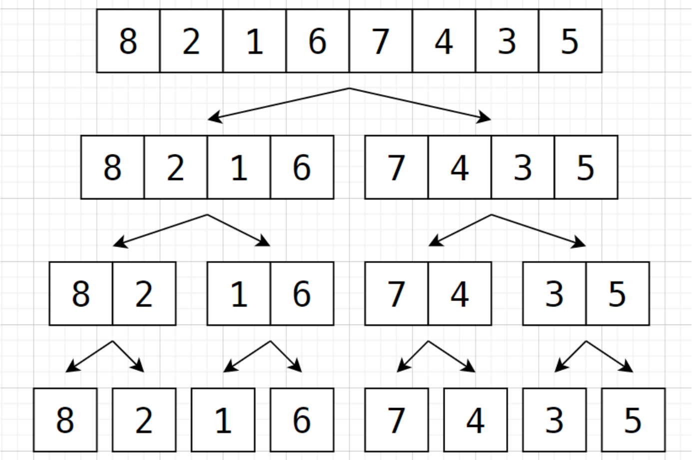
```cpp
vector<int> mergesort(vector<int> arr) {

    int n = (int)arr.size();

    // 邊界條件：陣列長度等於 1
    if (n == 1) return arr;

    // 遞迴排序左半邊跟右半邊
    int mid = n/2;
    vector<int> left, right;
    for (int i = 0; i < n; i++) {
        if (i < mid) left.push_back(arr[i]);
        else right.push_back(arr[i]);
    }
    left = mergesort(left);
    right = mergesort(right);

    // 合併左右兩半邊
    vector<int> sorted;
    int Lptr = 0, Rptr = 0;

    while (Lptr < (int)left.size() &&
           Rptr < (int)right.size()) {

        if (left[Lptr] < right[Rptr]) {
            sorted.push_back(left[Lptr]);
            Lptr++;

        } else {
            sorted.push_back(right[Rptr]);
            Rptr++;

        }

    }

    while (Lptr < (int)left.size()) {
        sorted.push_back(left[Lptr]);
        Lptr++;
    }

    while (Rptr < (int)right.size()) {
        sorted.push_back(right[Rptr]);
        Rptr++;
    }

    return sorted;

}
```

合併排序（Merge Sort）—常數比較小的版本

```cpp
vector<int> arr;

void mergesort(int l, int r) {

    // 邊界條件：陣列長度等於 1
    if (l == r) return;

    // 遞迴排序左半邊跟右半邊
    int mid = (l+r)/2;
    mergesort(l, mid);
    mergesort(mid+1, r);

    // 合併左右兩半邊
    vector<int> sorted;
    int Lptr = l, Rptr = mid+1;

    while (Lptr <= mid && Rptr <= r) {

        if (arr[Lptr] < arr[Rptr]) {
            sorted.push_back(arr[Lptr]);
            Lptr++;

        } else {
            sorted.push_back(arr[Rptr]);
            Rptr++;

        }

    }

    while (Lptr <= mid) {
        sorted.push_back(arr[Lptr]);
        Lptr++;
    }
    while (Rptr <= r) {
        sorted.push_back(arr[Rptr]);
        Rptr++;
    }

    for (int i = l, j = 0; i <= r; i++, j++) {
        arr[i] = sorted[j];
    }
}
```

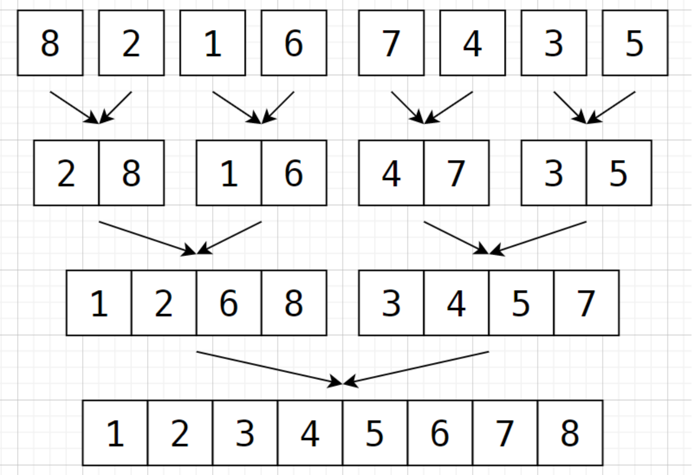
合併排序 (Merge Sort) —複雜度分析

- 我們把遞迴的過程畫出來，會發現它是一棵樹狀圖，稱之為「遞迴樹」。



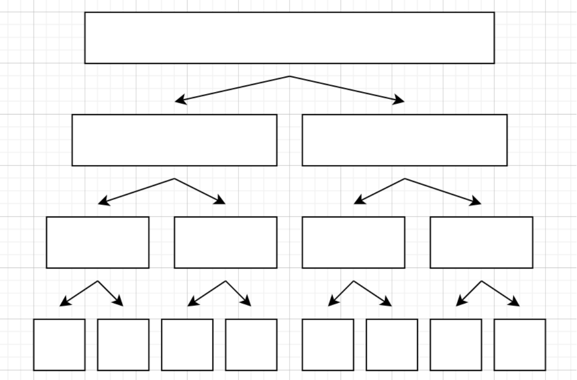
合併排序 (Merge Sort) —複雜度分析

- 我們把遞迴的過程畫出來，會發現它是一棵樹狀圖，稱之為「遞迴樹」。

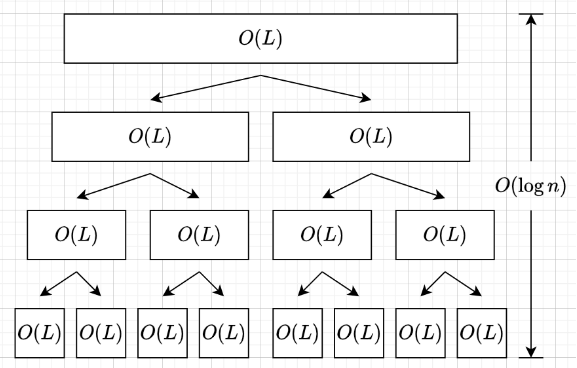


合併排序（Merge Sort）—複雜度分析

- 簡化後的遞迴樹如右。
- 只要把樹上每個節點的時間複雜度加總，就可以知道合併排序的複雜度了。

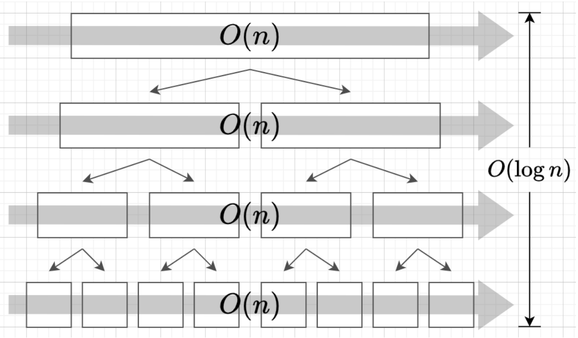


- 簡化後的遞迴樹如右。
- 只要把樹上每個節點的時間複雜度加總，就可以知道合併排序的複雜度了。
- 合併兩個序列的複雜度是線性的。
- 故每個節點的複雜度為 $O(L)$，這裡的 L 是指該節點對應到的序列長度。



合併排序（Merge Sort）—複雜度分析

- 簡化後的遞迴樹如右。
- 只要把樹上每個節點的時間複雜度加總，就可以知道合併排序的複雜度了。
- 合併兩個序列的複雜度是線性的。
- 故每個節點的複雜度為 $O(L)$，這裡的 L 是指該節點對應到的序列長度。
- 用一層一層的角度來看，每層遞迴總複雜度 $O(n)$，n 是指原本整個序列的大小。
- 而因為一直切一半的關係，所以遞迴總共 $O(\log n)$ 層。
- 故合併排序總時間複雜度為 $O(n \log n)$！



分治的複雜度分析方法

分治的複雜度分析方法

- 有些遞迴的複雜度並不能一眼就看出時間複雜度是多少，因此必須藉助一些工具來分析。

分治的複雜度分析方法

- 有些遞迴的複雜度並不能一眼就看出時間複雜度是多少，因此必須藉助一些工具來分析。
- 常見的工具：遞迴樹、主定理 (Master Theorem)、取代法 (Substitution Method)

分治的複雜度分析方法

- 有些遞迴的複雜度並不能一眼就看出時間複雜度是多少，因此必須藉助一些工具來分析。
- 常見的工具：遞迴樹、主定理 (Master Theorem)、取代法 (Substitution Method)
- 遞迴樹最直觀也最好理解，但較不嚴謹。

分治的複雜度分析方法

- 有些遞迴的複雜度並不能一眼就看出時間複雜度是多少，因此必須藉助一些工具來分析。
- 常見的工具：遞迴樹、主定理 (Master Theorem)、取代法 (Substitution Method)
- 遞迴樹最直觀也最好理解，但較不嚴謹。
- 主定理能應付大部分常見的分治演算法，利用主定理能很快得出遞迴的複雜度。

分治的複雜度分析方法

- 有些遞迴的複雜度並不能一眼就看出時間複雜度是多少，因此必須藉助一些工具來分析。
- 常見的工具：遞迴樹、主定理 (Master Theorem)、取代法 (Substitution Method)
- 遞迴樹最直觀也最好理解，但較不嚴謹。
- 主定理能應付大部分常見的分治演算法，利用主定理能很快得出遞迴的複雜度。
- 取代法是建立在數學歸納法的基礎上，搭配時間複雜度的嚴謹定義，證明遞迴的複雜度。

分治的複雜度分析方法

- 有些遞迴的複雜度並不能一眼就看出時間複雜度是多少，因此必須藉助一些工具來分析。
- 常見的工具：遞迴樹、主定理 (Master Theorem)、取代法 (Substitution Method)
- 遞迴樹最直觀也最好理解，但較不嚴謹。
- 主定理能應付大部分常見的分治演算法，利用主定理能很快得出遞迴的複雜度。
- 取代法是建立在數學歸納法的基礎上，搭配時間複雜度的嚴謹定義，證明遞迴的複雜度。
- 今天這堂課只會介紹前兩種工具。大部分情況下會這兩種就夠用了。

先備知識—時間複雜度

- 時間複雜度記號的定義：
- O (Big O) 代表的是上界，一個演算法的複雜度如果是 $O(n^2)$，代表這個演算法的執行時間不會超過 n^2 的量級。就算它真正的執行時間是 $n \log n$ 我們也可以說它的複雜度是 $O(n^2)$。

- 時間複雜度記號的定義：
- O (Big O) 代表的是上界，一個演算法的複雜度如果是 $O(n^2)$，代表這個演算法的執行時間不會超過 n^2 的量級。就算它真正的執行時間是 $n \log n$ 我們也可以說它的複雜度是 $O(n^2)$。
- Ω (Big Omega) 代表的是下界，與 Big O 正好相反。一個演算法的複雜度如果是 $\Omega(n^2)$，代表這個演算法的執行時間至少會是 n^2 的量級。

先備知識—時間複雜度

- 時間複雜度記號的定義：
- O (Big O) 代表的是上界，一個演算法的複雜度如果是 $O(n^2)$，代表這個演算法的執行時間不會超過 n^2 的量級。就算它真正的執行時間是 $n \log n$ 我們也可以說它的複雜度是 $O(n^2)$。
- Ω (Big Omega) 代表的是下界，與 Big O 正好相反。一個演算法的複雜度如果是 $\Omega(n^2)$，代表這個演算法的執行時間至少會是 n^2 的量級。
- Θ (Big Theta) 代表「剛剛好」，是 O 和 Ω 的結合。一個演算法的複雜度如果是 $\Theta(n^2)$，代表這個演算法執行時間剛剛好就是 n^2 的量級。

先備知識—時間複雜度

- 時間複雜度記號的定義：
- O (Big O) 代表的是上界，一個演算法的複雜度如果是 $O(n^2)$，代表這個演算法的執行時間不會超過 n^2 的量級。就算它真正的執行時間是 $n \log n$ 我們也可以說它的複雜度是 $O(n^2)$。
- Ω (Big Omega) 代表的是下界，與 Big O 正好相反。一個演算法的複雜度如果是 $\Omega(n^2)$，代表這個演算法的執行時間至少會是 n^2 的量級。
- Θ (Big Theta) 代表「剛剛好」，是 O 和 Ω 的結合。一個演算法的複雜度如果是 $\Theta(n^2)$，代表這個演算法執行時間剛剛好就是 n^2 的量級。
- 想知道嚴謹定義可以參考這個網站：http://alrightchiu.github.io/SecondRound/complexityasymptotic-notationjian-jin-fu-hao.html

先備知識—等比級數公式

- 等比數列：$a, ar, ar^2, ar^3, \cdots, ar^{n-1}$
- 首項為 a，公比為 r，一共有 n 項。

先備知識—等比級數公式

- 等比數列：$a, ar, ar^2, ar^3, \cdots, ar^{n-1}$
- 首項為 a，公比為 r，一共有 n 項。
- 等比級數：$S_n = a + ar + ar^2 + \cdots + ar^{n-1} = \sum_{i=1}^{n} ar^{i-1}$

先備知識—等比級數公式

- 等比數列：$a, ar, ar^2, ar^3, \cdots, ar^{n-1}$
- 首項為 a，公比為 r，一共有 n 項。
- 等比級數：$S_n = a + ar + ar^2 + \cdots + ar^{n-1} = \sum_{i=1}^{n} ar^{i-1}$
- 當公比 $r = 1$ 時，$S_n = an$
- 當公比 $r \neq 1$ 時，$S_n = \frac{a(1-r^n)}{1-r}$

先備知識—等比級數公式

- 等比數列：$a, ar, ar^2, ar^3, \cdots, ar^{n-1}$
- 首項為 a，公比為 r，一共有 n 項。
- 等比級數：$S_n = a + ar + ar^2 + \cdots + ar^{n-1} = \sum_{i=1}^{n} ar^{i-1}$
- 當公比 $r = 1$ 時，$S_n = an$
- 當公比 $r \neq 1$ 時，$S_n = \frac{a(1-r^n)}{1-r}$
- 無窮等比級數：$S_\infty = a + ar + ar^2 + \cdots = \sum_{i=1}^{\infty} ar^{i-1}$

先備知識—等比級數公式

- 等比數列：$a, ar, ar^2, ar^3, \cdots, ar^{n-1}$
- 首項為 a，公比為 r，一共有 n 項。
- 等比級數：$S_n = a + ar + ar^2 + \cdots + ar^{n-1} = \sum_{i=1}^{n} ar^{i-1}$
- 當公比 $r = 1$ 時，$S_n = an$
- 當公比 $r \neq 1$ 時，$S_n = \frac{a(1-r^n)}{1-r}$
- 無窮等比級數：$S_\infty = a + ar + ar^2 + \cdots = \sum_{i=1}^{\infty} ar^{i-1}$
- 當公比 r 滿足 $-1 < r < 1$：

$$S_\infty = \lim_{n\to\infty} S_n = \lim_{n\to\infty} \frac{a(1-r^n)}{1-r} = \frac{a}{1-r} \quad (\because \lim_{n\to\infty} r^n = 0)$$

先備知識—等比級數公式

- 等比數列：$a, ar, ar^2, ar^3, \cdots, ar^{n-1}$
- 首項為 a，公比為 r，一共有 n 項。
- 等比級數：$S_n = a + ar + ar^2 + \cdots + ar^{n-1} = \sum_{i=1}^{n} ar^{i-1}$
- 當公比 $r = 1$ 時，$S_n = an$
- 當公比 $r \neq 1$ 時，$S_n = \frac{a(1-r^n)}{1-r}$
- 無窮等比級數：$S_\infty = a + ar + ar^2 + \cdots = \sum_{i=1}^{\infty} ar^{i-1}$
- 當公比 r 滿足 $-1 < r < 1$：

$$S_\infty = \lim_{n\to\infty} S_n = \lim_{n\to\infty} \frac{a(1-r^n)}{1-r} = \frac{a}{1-r} \quad (\because \lim_{n\to\infty} r^n = 0)$$

- 不懂這些公式怎麼來的沒關係，就把它們當成黑盒子，等等計算等比數列和會用到。

先備知識—T(n)

- 分治演算法的時間複雜度我們習慣以 $T(n)$ 表示。

先備知識—T(n)

- 分治演算法的時間複雜度我們習慣以 $T(n)$ 表示。
- n 表示分治算法處理的輸入量。

先備知識—T(n)

- 分治演算法的時間複雜度我們習慣以 $T(n)$ 表示。
- n 表示分治算法處理的輸入量。
- 既然是分治，那就表示 $T(n)$ 是一個遞迴函數。

先備知識—T(n)

- 分治演算法的時間複雜度我們習慣以 $T(n)$ 表示。
- n 表示分治算法處理的輸入量。
- 既然是分治，那就表示 $T(n)$ 是一個遞迴函數。
- 以 Merge Sort 為例，每次會將問題切割成 2 個規模 $\frac{n}{2}$ 的子問題，並且要花費 $O(n)$ 時間合併。

先備知識—T(n)

- 分治演算法的時間複雜度我們習慣以 $T(n)$ 表示。
- n 表示分治算法處理的輸入量。
- 既然是分治，那就表示 $T(n)$ 是一個遞迴函數。
- 以 Merge Sort 為例，每次會將問題切割成 2 個規模 $\frac{n}{2}$ 的子問題，並且要花費 $O(n)$ 時間合併。
- 故 Merge Sort 的 $T(n)$ 可以表示為：

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

先備知識—T(n)

- 分治演算法的時間複雜度我們習慣以 $T(n)$ 表示。
- n 表示分治算法處理的輸入量。
- 既然是分治，那就表示 $T(n)$ 是一個遞迴函數。
- 以 Merge Sort 為例，每次會將問題切割成 2 個規模 $\frac{n}{2}$ 的子問題，並且要花費 $O(n)$ 時間合併。
- 故 Merge Sort 的 $T(n)$ 可以表示為：

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

- 同樣的道理，快速冪的 $T(n)$ 可以表示為：

$$T(n) = T(\frac{n}{2}) + O(1)$$

遞迴樹

- 先來介紹第一種分析遞迴的方法：遞迴樹

遞迴樹

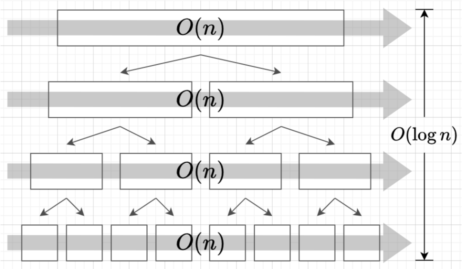
- 先來介紹第一種分析遞迴的方法：遞迴樹
- 遞迴樹非常直觀，直接把遞迴的過程畫成一個樹狀圖，

遞迴樹

- 先來介紹第一種分析遞迴的方法：遞迴樹
- 遞迴樹非常直觀，直接把遞迴的過程畫成一個樹狀圖，
- 然後把這棵樹上所有節點的時間全部加總，就可以得到整個遞迴的複雜度。

遞迴樹

- 先來介紹第一種分析遞迴的方法：遞迴樹
- 遞迴樹非常直觀，直接把遞迴的過程畫成一個樹狀圖，
- 然後把這棵樹上所有節點的時間全部加總，就可以得到整個遞迴的複雜度。
- 我們剛剛分析 Merge Sort 複雜度就是用遞迴樹來分析。



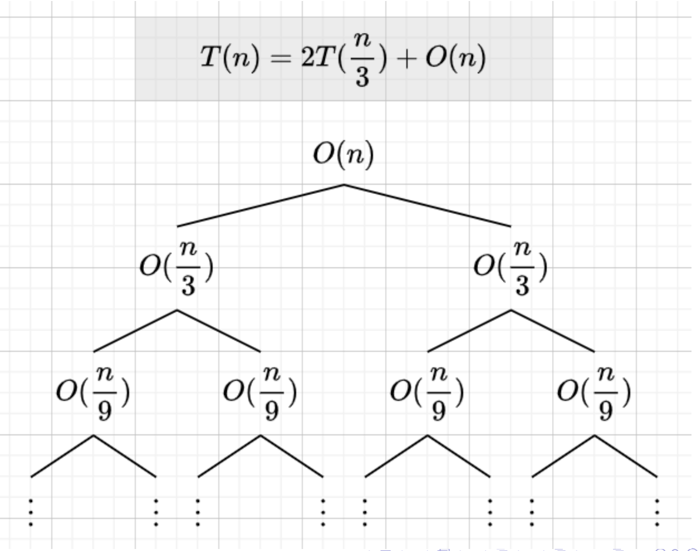
遞迴樹

例題

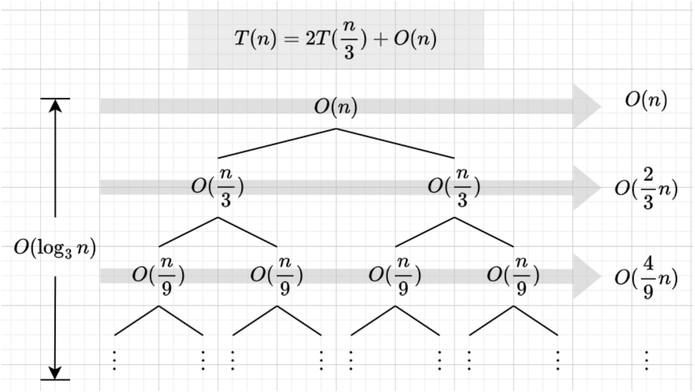
$T(n) = 2T(\frac{n}{3}) + O(n)$，求 $T(n)$ 的複雜度。

遞迴樹

- 把遞迴樹畫出來。
- 每往下一層，問題的規模就會變為 $\frac{1}{3}$ 倍，故總共有 $\log_3 n$ 層。
- 接著把每一層加總起來。



- 把遞迴樹畫出來。
- 每往下一層，問題的規模就會變為 $\frac{1}{3}$ 倍，故總共有 $\log_3 n$ 層。
- 接著把每一層加總起來。
- 你會發現，每層的總時間會是一個等比數列。
- 首項 n，公比 $\frac{2}{3}$，項數 $\log_3 n$。

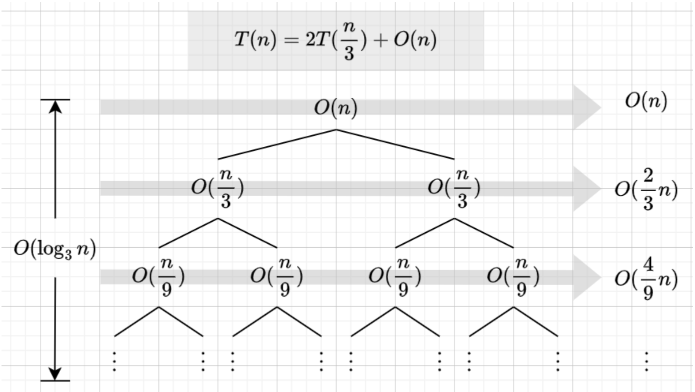


遞迴樹

- 首項 n，公比 $\frac{2}{3}$，項數 $\log_3 n$。
- 套等比級數公式：

$$
\begin{aligned}
T(n) &= O(\frac{n(1-(\frac{2}{3})^{\log_3 n})}{1-\frac{2}{3}}) \\
&= O(3n(1-(\frac{2}{3})^{\log_3 n})) \\
&= O(n(1-(\frac{2}{3})^{\log_3 n})) \\
&= O(n)
\end{aligned}
$$

- 因此，$T(n) = O(n)$。



- 原本的主定理較為複雜，這裡只介紹常見的情況。

$c_{\text{crit}} = \log_b a = \log(\#\text{subproblems}) / \log(\text{relative subproblem size})$

Case	Description	Condition on $f(n)$ in relation to c_{crit}, i.e. $\log_b a$	Master Theorem bound
1	Work to split/recombine a problem is dwarfed by subproblems. i.e. the recursion tree is leaf-heavy	When $f(n) = O(n^c)$ where $c < c_{\text{crit}}$ (upper-bounded by a lesser exponent polynomial)	... then $T(n) = \Theta\left(n^{c_{\text{crit}}}\right)$ (The splitting term does not appear; the recursive tree structure dominates.)
2	Work to split/recombine a problem is comparable to subproblems.	When $f(n) = \Theta(n^{c_{\text{crit}}} \log^k n)$ for a $k \ge 0$ (rangebound by the critical-exponent polynomial, times zero or more optional logs)	... then $T(n) = \Theta\left(n^{c_{\text{crit}}} \log^{k+1} n\right)$ (The bound is the splitting term, where the log is augmented by a single power.)
3	Work to split/recombine a problem dominates subproblems. i.e. the recursion tree is root-heavy.	When $f(n) = \Omega(n^c)$ where $c > c_{\text{crit}}$ (lower-bounded by a greater-exponent polynomial)	... this doesn't necessarily yield anything. Furthermore, if $af\left(\frac{n}{b}\right) \le kf(n)$ for some constant $k < 1$ and sufficiently large n (often called the *regularity condition*) then the total is dominated by the splitting term $f(n)$: $T(n) = \Theta(f(n))$

A useful extension of Case 2 handles all values of k:[3]

Case	Condition on $f(n)$ in relation to c_{crit}, i.e. $\log_b a$	Master Theorem bound
2a	When $f(n) = \Theta(n^{c_{\text{crit}}} \log^k n)$ for any $k > -1$	... then $T(n) = \Theta\left(n^{c_{\text{crit}}} \log^{k+1} n\right)$ (The bound is the splitting term, where the log is augmented by a single power.)
2b	When $f(n) = \Theta(n^{c_{\text{crit}}} \log^k n)$ for $k = -1$	... then $T(n) = \Theta\left(n^{c_{\text{crit}}} \log\log n\right)$ (The bound is the splitting term, where the log reciprocal is replaced by an iterated log.)
2c	When $f(n) = \Theta(n^{c_{\text{crit}}} \log^k n)$ for any $k < -1$	... then $T(n) = \Theta\left(n^{c_{\text{crit}}}\right)$ (The bound is the splitting term, where the log disappears.)

主定理

主定理 (Master Theorem)

對於以下形式的遞迴：$T(n) = aT(\frac{n}{b}) + O(n^d)$，比較 $\log_b a$ 和 d 的大小。

1. $\log_b a < d$：則 $T(n) = \Theta(n^d)$
2. $\log_b a = d$：則 $T(n) = \Theta(n^d \log n)$
3. $\log_b a > d$：則 $T(n) = \Theta(n^{\log_b a})$

主定理 (Master Theorem)

對於以下形式的遞迴：$T(n) = aT(\frac{n}{b}) + O(n^d)$，比較 $\log_b a$ 和 d 的大小。

1. $\log_b a < d$：則 $T(n) = \Theta(n^d)$
2. $\log_b a = d$：則 $T(n) = \Theta(n^d \log n)$
3. $\log_b a > d$：則 $T(n) = \Theta(n^{\log_b a})$

- 白話來說，就是比較 $\log_b a$ 和 d 的大小，誰比較大誰做主 (master)。

主定理

主定理 (Master Theorem)

對於以下形式的遞迴：$T(n) = aT(\frac{n}{b}) + O(n^d)$，比較 $\log_b a$ 和 d 的大小。

1. $\log_b a < d$：則 $T(n) = \Theta(n^d)$
2. $\log_b a = d$：則 $T(n) = \Theta(n^d \log n)$
3. $\log_b a > d$：則 $T(n) = \Theta(n^{\log_b a})$

- 白話來說，就是比較 $\log_b a$ 和 d 的大小，誰比較大誰做主 (master)。
- 而如果兩個都一樣，那就多乘一個 $\log$。

主定理

主定理 (Master Theorem)

對於以下形式的遞迴：$T(n) = aT(\frac{n}{b}) + O(n^d)$，比較 $\log_b a$ 和 d 的大小。

1. $\log_b a < d$：則 $T(n) = \Theta(n^d)$
2. $\log_b a = d$：則 $T(n) = \Theta(n^d \log n)$
3. $\log_b a > d$：則 $T(n) = \Theta(n^{\log_b a})$

- 白話來說，就是比較 $\log_b a$ 和 d 的大小，誰比較大誰做主 (master)。
- 而如果兩個都一樣，那就多乘一個 $\log$。
- 主定理的證明可以簡易地用等比級數的公式來證，有興趣的可以看這篇文章：https://blog.csdn.net/jmh1996/article/details/82827579

主定理

另外要注意的是，主定理當中 case 2 後面的 $O(n^d)$ 可以多帶 $\log$。

主定理

另外要注意的是，主定理當中 case 2 後面的 $O(n^d)$ 可以多帶 $\log$。

主定理 (Master Theorem)：case 2 延伸

當 $T(n) = aT(\frac{n}{b}) + \Theta(n^d \log^k n)$，且 $\log_b a = d, k > -1$，則：

$$T(n) = \Theta(n^d \log^{k+1} n)$$

一樣多乘一個 $\log$ 就行了。

主定理

- 還是看不懂沒關係，記重要結論就好：
- 看 $\log_b a$ 跟 d 誰比較大誰做主，一樣的話多乘一個 $\log$。

主定理練習

主定理 (Master Theorem)

對於以下形式的遞迴：$T(n) = aT(\frac{n}{b}) + O(n^d)$，比較 $\log_b a$ 和 d 的大小。

1. $\log_b a < d$：則 $T(n) = \Theta(n^d)$
2. $\log_b a = d$：則 $T(n) = \Theta(n^d \log n)$
3. $\log_b a > d$：則 $T(n) = \Theta(n^{\log_b a})$

例題一

$T(n) = 2T(\frac{n}{2}) + O(n)$，求 $T(n)$ 的複雜度。

主定理練習

主定理 (Master Theorem)

對於以下形式的遞迴：$T(n) = aT(\frac{n}{b}) + O(n^d)$，比較 $\log_b a$ 和 d 的大小。

1. $\log_b a < d$：則 $T(n) = \Theta(n^d)$
2. $\log_b a = d$：則 $T(n) = \Theta(n^d \log n)$
3. $\log_b a > d$：則 $T(n) = \Theta(n^{\log_b a})$

例題一

$T(n) = 2T(\frac{n}{2}) + O(n)$，求 $T(n)$ 的複雜度。

- $\log_b a = \log_2 2 = 1, d = 1$

主定理練習

主定理 (Master Theorem)

對於以下形式的遞迴：$T(n) = aT(\frac{n}{b}) + O(n^d)$，比較 $\log_b a$ 和 d 的大小。

1. $\log_b a < d$：則 $T(n) = \Theta(n^d)$
2. $\log_b a = d$：則 $T(n) = \Theta(n^d \log n)$
3. $\log_b a > d$：則 $T(n) = \Theta(n^{\log_b a})$

例題一

$T(n) = 2T(\frac{n}{2}) + O(n)$，求 $T(n)$ 的複雜度。

- $\log_b a = \log_2 2 = 1, d = 1$
- 套用主定理 case 2，$T(n) = \Theta(n \log n)$。

主定理練習

主定理 (Master Theorem)

對於以下形式的遞迴：$T(n) = aT(\frac{n}{b}) + O(n^d)$，比較 $\log_b a$ 和 d 的大小。

1. $\log_b a < d$：則 $T(n) = \Theta(n^d)$
2. $\log_b a = d$：則 $T(n) = \Theta(n^d \log n)$
3. $\log_b a > d$：則 $T(n) = \Theta(n^{\log_b a})$

例題二

$T(n) = 3T(\frac{n}{2}) + O(n)$，求 $T(n)$ 的複雜度。

主定理練習

主定理 (Master Theorem)

對於以下形式的遞迴：$T(n) = aT(\frac{n}{b}) + O(n^d)$，比較 $\log_b a$ 和 d 的大小。

1. $\log_b a < d$：則 $T(n) = \Theta(n^d)$
2. $\log_b a = d$：則 $T(n) = \Theta(n^d \log n)$
3. $\log_b a > d$：則 $T(n) = \Theta(n^{\log_b a})$

例題二

$T(n) = 3T(\frac{n}{2}) + O(n)$，求 $T(n)$ 的複雜度。

- $\log_b a = \log_2 3, d = 1$

主定理練習

主定理 (Master Theorem)

對於以下形式的遞迴：$T(n) = aT(\frac{n}{b}) + O(n^d)$，比較 $\log_b a$ 和 d 的大小。

1. $\log_b a < d$：則 $T(n) = \Theta(n^d)$
2. $\log_b a = d$：則 $T(n) = \Theta(n^d \log n)$
3. $\log_b a > d$：則 $T(n) = \Theta(n^{\log_b a})$

例題二

$T(n) = 3T(\frac{n}{2}) + O(n)$，求 $T(n)$ 的複雜度。

- $\log_b a = \log_2 3, d = 1$
- 套用主定理 case 3，$T(n) = \Theta(n^{\log_2 3})$。

主定理練習

主定理 (Master Theorem)

對於以下形式的遞迴：$T(n) = aT(\frac{n}{b}) + O(n^d)$，比較 $\log_b a$ 和 d 的大小。

1. $\log_b a < d$：則 $T(n) = \Theta(n^d)$
2. $\log_b a = d$：則 $T(n) = \Theta(n^d \log n)$
3. $\log_b a > d$：則 $T(n) = \Theta(n^{\log_b a})$

例題三

$T(n) = 8T(\frac{n}{2}) + O(n^2)$，求 $T(n)$ 的複雜度。

主定理練習

主定理 (Master Theorem)

對於以下形式的遞迴：$T(n) = aT(\frac{n}{b}) + O(n^d)$，比較 $\log_b a$ 和 d 的大小。

1. $\log_b a < d$：則 $T(n) = \Theta(n^d)$
2. $\log_b a = d$：則 $T(n) = \Theta(n^d \log n)$
3. $\log_b a > d$：則 $T(n) = \Theta(n^{\log_b a})$

例題三

$T(n) = 8T(\frac{n}{2}) + O(n^2)$，求 $T(n)$ 的複雜度。

- $\log_b a = \log_2 8 = 3, d = 1$

主定理練習

主定理 (Master Theorem)

對於以下形式的遞迴：$T(n) = aT(\frac{n}{b}) + O(n^d)$，比較 $\log_b a$ 和 d 的大小。

1. $\log_b a < d$：則 $T(n) = \Theta(n^d)$
2. $\log_b a = d$：則 $T(n) = \Theta(n^d \log n)$
3. $\log_b a > d$：則 $T(n) = \Theta(n^{\log_b a})$

例題三

$T(n) = 8T(\frac{n}{2}) + O(n^2)$，求 $T(n)$ 的複雜度。

- $\log_b a = \log_2 8 = 3, d = 1$
- 套用主定理 case 3，$T(n) = \Theta(n^3)$。

主定理練習

主定理 (Master Theorem)

對於以下形式的遞迴：$T(n) = aT(\frac{n}{b}) + O(n^d)$，比較 $\log_b a$ 和 d 的大小。

1. $\log_b a < d$：則 $T(n) = \Theta(n^d)$
2. $\log_b a = d$：則 $T(n) = \Theta(n^d \log n)$
3. $\log_b a > d$：則 $T(n) = \Theta(n^{\log_b a})$

例題四

$T(n) = 4T(\frac{n}{4}) + O(n \log n)$，求 $T(n)$ 的複雜度。

主定理練習

主定理 (Master Theorem)

對於以下形式的遞迴：$T(n) = aT(\frac{n}{b}) + O(n^d)$，比較 $\log_b a$ 和 d 的大小。

1. $\log_b a < d$：則 $T(n) = \Theta(n^d)$
2. $\log_b a = d$：則 $T(n) = \Theta(n^d \log n)$
3. $\log_b a > d$：則 $T(n) = \Theta(n^{\log_b a})$

例題四

$T(n) = 4T(\frac{n}{4}) + O(n \log n)$，求 $T(n)$ 的複雜度。

- $\log_b a = \log_4 4 = 1, d = 1$

主定理練習

主定理 (Master Theorem)

對於以下形式的遞迴：$T(n) = aT(\frac{n}{b}) + O(n^d)$，比較 $\log_b a$ 和 d 的大小。

1. $\log_b a < d$：則 $T(n) = \Theta(n^d)$
2. $\log_b a = d$：則 $T(n) = \Theta(n^d \log n)$
3. $\log_b a > d$：則 $T(n) = \Theta(n^{\log_b a})$

例題四

$T(n) = 4T(\frac{n}{4}) + O(n \log n)$，求 $T(n)$ 的複雜度。

- $\log_b a = \log_4 4 = 1, d = 1$
- 套用主定理 case 2 的延伸，$T(n) = \Theta(n \log^2 n)$。

主定理

- 主定理在碰到子問題規模不一樣的時候就無法使用，例如 $T(n) = T(\frac{n}{5}) + T(\frac{7n}{10}) + O(n)$。

主定理

- 主定理在碰到子問題規模不一樣的時候就無法使用，例如 $T(n) = T(\frac{n}{5}) + T(\frac{7n}{10}) + O(n)$。
- 或是後面的函式不是 $O(n^d)$ 那樣，例如 $T(n) = 2T(\frac{n}{2}) + O(3^n)$。

主定理

- 主定理在碰到子問題規模不一樣的時候就無法使用，例如 $T(n) = T(\frac{n}{5}) + T(\frac{7n}{10}) + O(n)$。
- 或是後面的函式不是 $O(n^d)$ 那樣，例如 $T(n) = 2T(\frac{n}{2}) + O(3^n)$。
- 但主定理在大部分的分治演算法下都適用，是非常方便的工具。

主定理

- 主定理在碰到子問題規模不一樣的時候就無法使用，例如 $T(n) = T(\frac{n}{5}) + T(\frac{7n}{10}) + O(n)$。
- 或是後面的函式不是 $O(n^d)$ 那樣，例如 $T(n) = 2T(\frac{n}{2}) + O(3^n)$。
- 但主定理在大部分的分治演算法下都適用，是非常方便的工具。
- 記不起來也沒關係，記住常見的遞迴複雜度就好，碰到其他情況再把樹畫出來計算：
  - $T(n) = 2T(\frac{n}{2}) + O(n) \Rightarrow T(n) = O(n \log n)$ (Merge Sort 的複雜度)
  - $T(n) = 2T(\frac{n}{2}) + O(n \log n) \Rightarrow T(n) = O(n \log^2 n)$

經典例題

最大連續和

CSES Maximum Subarray Sum

- 給一個長度為 n 的序列 a，請輸出 a 的最大連續和。
- 最大連續和定義為，所有 a 的非空連續區間中，區間和的最大值。
- $1 \leq n \leq 2 \times 10^5$

- 範例：序列 $[-1, 3, -2, 5, 3, -5, 2, 2]$ 的最大連續和為 9，發生在 $[3, -2, 5, 3]$ 那段區間。
- 有學過動態規劃的學員可能已經知道 $O(n)$ DP 的作法，但還是來練習一下用分治做這題 owo。

最大連續和

- 一樣把整個序列切一半，則最大連續和可能有三種情況：

最大連續和

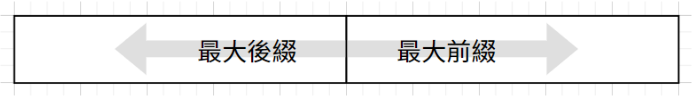
- 一樣把整個序列切一半，則最大連續和可能有三種情況：
  1. 最大連續和發生的區間在左半邊。
  2. 最大連續和發生的區間在右半邊。
  3. 最大連續和發生的區間橫跨左右兩半邊。

最大連續和

- 一樣把整個序列切一半，則最大連續和可能有三種情況：
  1. 最大連續和發生的區間在左半邊。
  2. 最大連續和發生的區間在右半邊。
  3. 最大連續和發生的區間橫跨左右兩半邊。
- 前面兩種情況交給遞迴處理。

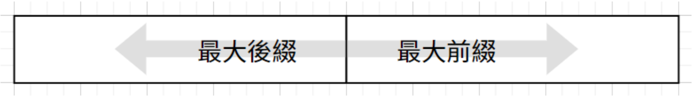
最大連續和

- 一樣把整個序列切一半，則最大連續和可能有三種情況：
  1. 最大連續和發生的區間在左半邊。
  2. 最大連續和發生的區間在右半邊。
  3. 最大連續和發生的區間橫跨左右兩半邊。
- 前面兩種情況交給遞迴處理。
- 第三種情況，等於找左半邊最大後綴 + 右半邊最大前綴，兩者合併就是橫跨兩邊的區間最大和！



最大連續和

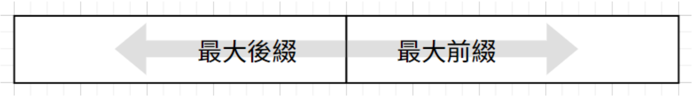
- 一樣把整個序列切一半，則最大連續和可能有三種情況：
  1. 最大連續和發生的區間在左半邊。
  2. 最大連續和發生的區間在右半邊。
  3. 最大連續和發生的區間橫跨左右兩半邊。
- 前面兩種情況交給遞迴處理。
- 第三種情況，等於找左半邊最大後綴 + 右半邊最大前綴，兩者合併就是橫跨兩邊的區間最大和！



- 找最大前後綴只需要 $O(n)$。

最大連續和

- 一樣把整個序列切一半，則最大連續和可能有三種情況：
  1. 最大連續和發生的區間在左半邊。
  2. 最大連續和發生的區間在右半邊。
  3. 最大連續和發生的區間橫跨左右兩半邊。
- 前面兩種情況交給遞迴處理。
- 第三種情況，等於找左半邊最大後綴 + 右半邊最大前綴，兩者合併就是橫跨兩邊的區間最大和！



- 找最大前後綴只需要 $O(n)$。
- $T(n) = 2T(\frac{n}{2}) + O(n)$，故 $T(n) = O(n \log n)$。

多項式乘法

(No Judge) 多項式乘法

- 設多項式 $f(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1} = \sum_{i=0}^{n-1} a_ix^i$，即 $f(x)$ 有 n 項。
- 設多項式 $g(x) = b_0 + b_1x + b_2x^2 + \cdots + b_{n-1}x^{n-1} = \sum_{i=0}^{n-1} b_ix^i$，即 $g(x)$ 也有 n 項。
- 請計算 $f(x) \times g(x)$ 的結果。

- 範例：$f(x) = (x + 2), g(x) = (2x + 1)$，則 $f(x) \times g(x) = 2x^2 + 5x + 2$。
- 方便起見我們假設 n 是 2 的冪次，不足的部分直接補係數是 0 的高次項就好。

多項式乘法

- 直接相乘要 $O(n^2)$，似乎有點慢。

多項式乘法

- 直接相乘要 $O(n^2)$，似乎有點慢。
- 考慮把 $f(x)$ 跟 $g(x)$ 分成兩半。

多項式乘法

- 直接相乘要 $O(n^2)$，似乎有點慢。
- 考慮把 $f(x)$ 跟 $g(x)$ 分成兩半。
- $f_L = a_0 + a_1 x + \cdots a_{n/2-1} x^{n/2-1}$
- $f_R = a_{n/2} + a_{n/2+1} x + \cdots a_{n-1} x^{n/2-1}$
- $g_L = b_0 + b_1 x + \cdots b_{n/2-1} x^{n/2-1}$
- $g_R = b_{n/2} + b_{n/2+1} x + \cdots b_{n-1} x^{n/2-1}$

多項式乘法

- 直接相乘要 $O(n^2)$，似乎有點慢。
- 考慮把 $f(x)$ 跟 $g(x)$ 分成兩半。
- $f_L = a_0 + a_1 x + \cdots a_{n/2-1} x^{n/2-1}$
- $f_R = a_{n/2} + a_{n/2+1} x + \cdots a_{n-1} x^{n/2-1}$
- $g_L = b_0 + b_1 x + \cdots b_{n/2-1} x^{n/2-1}$
- $g_R = b_{n/2} + b_{n/2+1} x + \cdots b_{n-1} x^{n/2-1}$
- 要計算的東西就變成了 $(f_L + x^{n/2} \cdot f_R)(g_L + x^{n/2} \cdot g_R)$

多項式乘法

- 直接相乘要 $O(n^2)$，似乎有點慢。
- 考慮把 $f(x)$ 跟 $g(x)$ 分成兩半。
- $f_L = a_0 + a_1 x + \cdots a_{n/2-1} x^{n/2-1}$
- $f_R = a_{n/2} + a_{n/2+1} x + \cdots a_{n-1} x^{n/2-1}$
- $g_L = b_0 + b_1 x + \cdots b_{n/2-1} x^{n/2-1}$
- $g_R = b_{n/2} + b_{n/2+1} x + \cdots b_{n-1} x^{n/2-1}$
- 要計算的東西就變成了 $(f_L + x^{n/2} \cdot f_R)(g_L + x^{n/2} \cdot g_R)$
- $= f_L g_L + x^{n/2}(f_L g_R + f_R g_L) + x^n f_R g_R$

多項式乘法

- $= f_L g_L + x^{n/2}(f_L g_R + f_R g_L) + x^n f_R g_R$
- 只要做 4 次乘法 $(f_L g_L, f_L g_R, f_R g_L, f_R g_R)$ 然後再做一些加法就完成了。

多項式乘法

- $= f_L g_L + x^{n/2}(f_L g_R + f_R g_L) + x^n f_R g_R$
- 只要做 4 次乘法 $(f_L g_L, f_L g_R, f_R g_L, f_R g_R)$ 然後再做一些加法就完成了。
- 那 4 次乘法可以看成是規模為 $n/2$ 的問題，而加法以及乘上 $x^{n/2}, x^n$ 都是 $O(n)$。

多項式乘法

- $= f_L g_L + x^{n/2}(f_L g_R + f_R g_L) + x^n f_R g_R$
- 只要做 4 次乘法 $(f_L g_L, f_L g_R, f_R g_L, f_R g_R)$ 然後再做一些加法就完成了。
- 那 4 次乘法可以看成是規模為 $n/2$ 的問題，而加法以及乘上 $x^{n/2}, x^n$ 都是 $O(n)$。
- 列出時間複雜度的遞迴式：$T(n) = 4T(n/2) + O(n)$。

多項式乘法

- $= f_L g_L + x^{n/2}(f_L g_R + f_R g_L) + x^n f_R g_R$
- 只要做 4 次乘法 $(f_L g_L, f_L g_R, f_R g_L, f_R g_R)$ 然後再做一些加法就完成了。
- 那 4 次乘法可以看成是規模為 $n/2$ 的問題，而加法以及乘上 $x^{n/2}, x^n$ 都是 $O(n)$。
- 列出時間複雜度的遞迴式：$T(n) = 4T(n/2) + O(n)$。
- 根據主定理，$T(n) = O(n^2)$，完全沒有變快 qwq

多項式乘法

- 我們要知道 $(f_Lg_L, f_Lg_R + f_Rg_L, f_Rg_R)$。

多項式乘法

- 我們要知道 $(f_L g_L, f_L g_R + f_R g_L, f_R g_R)$。
- 靈光一閃，如果我們計算 $S = (f_L + f_R)(g_L + g_R) = f_L g_L + f_L g_R + f_R g_L + f_R g_R$

多項式乘法

- 我們要知道 $(f_Lg_L, f_Lg_R + f_Rg_L, f_Rg_R)$。
- 靈光一閃，如果我們計算 $S = (f_L + f_R)(g_L + g_R) = f_Lg_L + f_Lg_R + f_Rg_L + f_Rg_R$
- 則 $f_Lg_L + x^{n/2}(f_Lg_R + f_Rg_L) + x^n f_Rg_R$ 可以寫成：

$$f_Lg_L + x^{n/2}(S - f_Lg_L - f_Rg_R) + x^n f_Rg_R$$

多項式乘法

- 我們要知道 $(f_L g_L, f_L g_R + f_R g_L, f_R g_R)$。
- 靈光一閃，如果我們計算 $S = (f_L + f_R)(g_L + g_R) = f_L g_L + f_L g_R + f_R g_L + f_R g_R$
- 則 $f_L g_L + x^{n/2}(f_L g_R + f_R g_L) + x^n f_R g_R$ 可以寫成：

$$f_L g_L + x^{n/2}(S - f_L g_L - f_R g_R) + x^n f_R g_R$$

- 只要計算 $S, f_L g_L, f_R g_R$，乘法次數降低成三次了！

多項式乘法

- 分析一下複雜度：
- $T(n) = 3T(\frac{n}{2}) + O(n)$

多項式乘法

- 分析一下複雜度：
- $T(n) = 3T(\frac{n}{2}) + O(n)$
- 根據主定理，$T(n) = O(n^{\log_2 3}) \approx O(n^{1.58})$

多項式乘法

- 分析一下複雜度：
- $T(n) = 3T(\frac{n}{2}) + O(n)$
- 根據主定理，$T(n) = O(n^{\log_2 3}) \approx O(n^{1.58})$
- $n = 2 \times 10^5$ 時，$n^{1.58} \approx 2.4 \times 10^8$。

多項式乘法

- 分析一下複雜度：
- $T(n) = 3T(\frac{n}{2}) + O(n)$
- 根據主定理，$T(n) = O(n^{\log_2 3}) \approx O(n^{1.58})$
- $n = 2 \times 10^5$ 時，$n^{1.58} \approx 2.4 \times 10^8$。
- 這個演算法被稱為 Karatsuba Algorithm，是由俄國數學家 Karatsuba 提出。

多項式乘法

- 分析一下複雜度：
- $T(n) = 3T(\frac{n}{2}) + O(n)$
- 根據主定理，$T(n) = O(n^{\log_2 3}) \approx O(n^{1.58})$
- $n = 2 \times 10^5$ 時，$n^{1.58} \approx 2.4 \times 10^8$。
- 這個演算法被稱為 Karatsuba Algorithm，是由俄國數學家 Karatsuba 提出。
- 類似的概念也有被應用在矩陣乘法上，有興趣的學員可以上網搜尋「Strassen's algorithm for matrix multiplication」

逆序數對

TIOJ 1080 逆序數對

- 對一個數列 S 來說，若 $i < j$ 且 $S_i > S_j$ 的話，那麼 (i, j) 就是一個逆序數對。
- 給定 S，輸出 S 總共有多少個逆序數對。
- $1 \leq n \leq 10^5$

範例：$[2, 4, 1, 3]$ 總共有 3 個逆序數對：$(2, 1)$、$(4, 1)$、$(4, 3)$。

逆序數對

- 一樣把序列分成左右兩半

逆序數對

- 一樣把序列分成左右兩半
- 這時候逆序數對 (i, j) 可以分成三種情況：

逆序數對

- 一樣把序列分成左右兩半
- 這時候逆序數對 (i, j) 可以分成三種情況：
  1. i, j 都在左半邊
  2. i, j 都在右半邊
  3. i 在左邊，j 在右邊

逆序數對

- 一樣把序列分成左右兩半
- 這時候逆序數對 (i, j) 可以分成三種情況：
  1. i, j 都在左半邊
  2. i, j 都在右半邊
  3. i 在左邊，j 在右邊
- 前兩種情況，也就是 i, j 都在同一邊可以遞迴計算。

逆序數對

- 一樣把序列分成左右兩半
- 這時候逆序數對 (i, j) 可以分成三種情況：
  1. i, j 都在左半邊
  2. i, j 都在右半邊
  3. i 在左邊，j 在右邊
- 前兩種情況，也就是 i, j 都在同一邊可以遞迴計算。
- 問題是第三種情況，一個在左一個在右，沒辦法遞迴處理。

逆序數對

- 一樣把序列分成左右兩半
- 這時候逆序數對 (i, j) 可以分成三種情況：
  1. i, j 都在左半邊
  2. i, j 都在右半邊
  3. i 在左邊，j 在右邊
- 前兩種情況，也就是 i, j 都在同一邊可以遞迴計算。
- 問題是第三種情況，一個在左一個在右，沒辦法遞迴處理。
- 只要處理橫跨兩邊的逆序數對，問題就圓滿解決了！

逆序數對

- 逆序數對的兩個條件：位置（$i < j$）、數值（$S_i > S_j$）。

逆序數對

- 逆序數對的兩個條件：位置（$i < j$）、數值（$S_i > S_j$）。
- 我們可以換個角度計算橫跨兩邊的逆序數對：
- 枚舉右半邊的每個數字 S_j，並計算左半邊有多少個數字大於 S_j。

逆序數對

- 逆序數對的兩個條件：位置（$i < j$）、數值（$S_i > S_j$）。
- 我們可以換個角度計算橫跨兩邊的逆序數對：
- 枚舉右半邊的每個數字 S_j，並計算左半邊有多少個數字大於 S_j。
- 因為 S_j 在右半邊，所以 S_j 和左半邊的數字必定能滿足「位置」的條件。

逆序數對

- 逆序數對的兩個條件：位置（$i < j$）、數值（$S_i > S_j$）。
- 我們可以換個角度計算橫跨兩邊的逆序數對：
- 枚舉右半邊的每個數字 S_j，並計算左半邊有多少個數字大於 S_j。
- 因為 S_j 在右半邊，所以 S_j 和左半邊的數字必定能滿足「位置」的條件。
- 所以左半邊每一個大於 S_j 的數字，必定能和 S_j 構成逆序數對，反之則不行。

逆序數對

- 逆序數對的兩個條件：位置（$i < j$）、數值（$S_i > S_j$）。
- 我們可以換個角度計算橫跨兩邊的逆序數對：
- 枚舉右半邊的每個數字 S_j，並計算左半邊有多少個數字大於 S_j。
- 因為 S_j 在右半邊，所以 S_j 和左半邊的數字必定能滿足「位置」的條件。
- 所以左半邊每一個大於 S_j 的數字，必定能和 S_j 構成逆序數對，反之則不行。
- 這樣做必定能算出所有橫跨兩邊的逆序數對，不多也不少。

逆序數對

- 「枚舉右半邊的每個數字 S_j，並計算左半邊有多少個數字大於 S_j。」

逆序數對

- 「枚舉右半邊的每個數字 S_j，並計算左半邊有多少個數字大於 S_j。」
- 先把左半邊的數字從大排到小。

逆序數對

- 「枚舉右半邊的每個數字 S_j，並計算左半邊有多少個數字大於 S_j。」
- 先把左半邊的數字從大排到小。
- 接著枚舉右半邊的每個數字 S_j，拿 S_j 去左半邊二分搜，就可以知道左邊有幾個數字大於 S_j。

逆序數對

- 「枚舉右半邊的每個數字 S_j，並計算左半邊有多少個數字大於 S_j。」
- 先把左半邊的數字從大排到小。
- 接著枚舉右半邊的每個數字 S_j，拿 S_j 去左半邊二分搜，就可以知道左邊有幾個數字大於 S_j。
- $T(n) = 2T(\frac{n}{2}) + O(n \log n)$

逆序數對

- 「枚舉右半邊的每個數字 S_j，並計算左半邊有多少個數字大於 S_j。」
- 先把左半邊的數字從大排到小。
- 接著枚舉右半邊的每個數字 S_j，拿 S_j 去左半邊二分搜，就可以知道左邊有幾個數字大於 S_j。
- $T(n) = 2T(\frac{n}{2}) + O(n \log n)$
- 總時間複雜度為 $O(n \log^2 n)$。

逆序數對

- 「枚舉右半邊的每個數字 S_j，並計算左半邊有多少個數字大於 S_j。」
- 如果不只左邊，連右邊也都由大到小排好了呢？

逆序數對

- 「枚舉右半邊的每個數字 S_j，並計算左半邊有多少個數字大於 S_j。」
- 如果不只左邊，連右邊也都由大到小排好了呢？
- 當你枚舉右半邊的時候，枚舉到的 S_j 會越來越小，也就代表左半邊比 S_j 大的數字也會越來越多。

逆序數對

- 「枚舉右半邊的每個數字 S_j，並計算左半邊有多少個數字大於 S_j。」
- 如果不只左邊，連右邊也都由大到小排好了呢？
- 當你枚舉右半邊的時候，枚舉到的 S_j 會越來越小，也就代表左半邊比 S_j 大的數字也會越來越多。
- 單調性出現了，用雙指針 $O(n)$ 解決！

逆序數對

- 「枚舉右半邊的每個數字 S_j，並計算左半邊有多少個數字大於 S_j。」
- 如果不只左邊，連右邊也都由大到小排好了呢？
- 當你枚舉右半邊的時候，枚舉到的 S_j 會越來越小，也就代表左半邊比 S_j 大的數字也會越來越多。
- 單調性出現了，用雙指針 $O(n)$ 解決！
- 問題來了，要怎麼排序左半邊跟右半邊？總不能又 $O(n \log n)$ 排序吧。

逆序數對

- 「枚舉右半邊的每個數字 S_j，並計算左半邊有多少個數字大於 S_j。」
- 如果不只左邊，連右邊也都由大到小排好了呢？
- 當你枚舉右半邊的時候，枚舉到的 S_j 會越來越小，也就代表左半邊比 S_j 大的數字也會越來越多。
- 單調性出現了，用雙指針 $O(n)$ 解決！
- 問題來了，要怎麼排序左半邊跟右半邊？總不能又 $O(n \log n)$ 排序吧。
- 還記得 Merge Sort 嗎？套上 Merge Sort 的框架就行了！

逆序數對

- 「枚舉右半邊的每個數字 S_j，並計算左半邊有多少個數字大於 S_j。」
- 如果不只左邊，連右邊也都由大到小排好了呢？
- 當你枚舉右半邊的時候，枚舉到的 S_j 會越來越小，也就代表左半邊比 S_j 大的數字也會越來越多。
- 單調性出現了，用雙指針 $O(n)$ 解決！
- 問題來了，要怎麼排序左半邊跟右半邊？總不能又 $O(n \log n)$ 排序吧。
- 還記得 Merge Sort 嗎？套上 Merge Sort 的框架就行了！
- 也就是做 Merge Sort 的同時可以順便計算逆序數對！

逆序數對

- 「枚舉右半邊的每個數字 S_j，並計算左半邊有多少個數字大於 S_j。」
- 如果不只左邊，連右邊也都由大到小排好了呢？
- 當你枚舉右半邊的時候，枚舉到的 S_j 會越來越小，也就代表左半邊比 S_j 大的數字也會越來越多。
- 單調性出現了，用雙指針 $O(n)$ 解決！
- 問題來了，要怎麼排序左半邊跟右半邊？總不能又 $O(n \log n)$ 排序吧。
- 還記得 Merge Sort 嗎？套上 Merge Sort 的框架就行了！
- 也就是做 Merge Sort 的同時可以順便計算逆序數對！
- $T(n) = 2T(\frac{n}{2}) + O(n)$，跟 Merge Sort 複雜度一樣是 $O(n \log n)$。

NEOJ 795 最近點對

- 平面上有 n 個點，第 i 個點座標為 (x_i, y_i)，請輸出任兩點之間的距離最小值。
- $1 \leq n \leq 2 \times 10^5$

分治再談

分治使用時機

- 分治：Divide and Conquer

分治使用時機

- 分治：Divide and Conquer
- 暴力法的癥結點在於會跑完所有的資訊，但有時候其實可以省去一些不必要的搜尋。

分治使用時機

- 分治：Divide and Conquer
- 暴力法的癥結點在於會跑完所有的資訊，但有時候其實可以省去一些不必要的搜尋。
- 當你發現一個問題從中間切下去時，如果 Conquer 的時候有一些不錯的性質可以利用（ex. 單調性），就可以考慮使用分治，避免計算不必要的資訊，從而達到優化複雜度的效果。

分治小技巧

1. 切割的區間，依照個人習慣可以選擇開區間或閉區間。我自己是習慣左閉右閉，這樣比較好想、好實作。

分治小技巧

1. 切割的區間，依照個人習慣可以選擇開區間或閉區間。我自己是習慣左閉右閉，這樣比較好想、好實作。
2. 思考分治題的時候，可以直接假設 Divide 下去左右兩邊遞迴都是好的，這時候只要專心想怎麼 Conquer 就好，也就是怎麼處理跨越兩半的答案，不用管左右邊遞迴會發生什麼事。

分治小技巧

1. 切割的區間，依照個人習慣可以選擇開區間或閉區間。我自己是習慣左閉右閉，這樣比較好想、好實作。
2. 思考分治題的時候，可以直接假設 Divide 下去左右兩邊遞迴都是好的，這時候只要專心想怎麼 Conquer 就好，也就是怎麼處理跨越兩半的答案，不用管左右邊遞迴會發生什麼事。
3. 遞迴常數頗大，當一個問題 Divide 到規模足夠小的時候，如果這時候暴力比遞迴還快，就可以改成使用暴力而不繼續遞迴下去。

OWO

1. 練習題都已經放到題單上囉，有很多有趣的構造題，也有難度非常高的題目，歡迎大家練習 > <
2. 今天這堂課到這裡結束，感謝大家的聆聽 OwO
3. 有問題的話下課後歡迎來戳我。