Function & Recursion

王淇 (LittleCube)

台南一中資訊社 TFCIS

2021.9.22

Outline

使用函數 - cmath

- 在認識函數之前，先學會使用函數

使用函數 - cmath

- 在認識函數之前，先學會使用函數
- <cmath>

使用函數 - cmath

- 在認識函數之前，先學會使用函數
- <cmath>
  - ceil(x) 回傳大於等於 x 的最小整數
  - floor(x) 回傳小於等於 x 的最大整數
  - round(x) 回傳最接近 x 的整數
  - sqrt(x) 回傳根號 x
  - cbrt(x) 回傳立方根 x
  - pow(x, p) 回傳 x^p
  - log10(x) 回傳 $\log_{10} x$
  - 當然也有三角函數跟反三角函數。

使用函數 - cmath

- 在認識函數之前，先學會使用函數
- <cmath>
  - ceil(x) 回傳大於等於 x 的最小整數
  - floor(x) 回傳小於等於 x 的最大整數
  - round(x) 回傳最接近 x 的整數
  - sqrt(x) 回傳根號 x
  - cbrt(x) 回傳立方根 x
  - pow(x, p) 回傳 x^p
  - log10(x) 回傳 $\log_{10} x$
  - 當然也有三角函數跟反三角函數。
- 暖身練習題：TOJ 19、TOJ 341

函數 - 宣告

- 從剛剛的練習來看，什麼是函數？

函數 - 宣告

- 從剛剛的練習來看，什麼是函數？
- **函數就像一台機器，你丟給他東西，他按照你給的東西做事**

函數 - 宣告

- 從剛剛的練習來看，什麼是函數？
- **函數就像一台機器，你丟給他東西，他按照你給的東西做事**
- 要怎麼定義自己的函數？

函數 - 宣告

```
type name(type1 arg1, type2 arg2, ...)
{
    //...
    return result;
}
```

函數 - 宣告

```
type name(type1 arg1, type2 arg2, ...)
{
    //...
    return result;
}
```

- type 是**回傳值類型**，可以想像成你要這個機器做完事要告訴你什麼類的東西（可能是告訴你方程式的根，這時候就是浮點數、告訴你他搜尋到誰，這時候就可能是整數）

函數 - 宣告

```
type name(type1 arg1, type2 arg2, ...)
{
    //...
    return result;
}
```

- type 是**回傳值類型**，可以想像成你要這個機器做完事要告訴你什麼類的東西 (可能是告訴你方程式的根，這時候就是浮點數、告訴你他搜尋到誰，這時候就可能是整數)
- type1 arg1, ... 是**傳入值**，可以想像成你給了這個機器什麼 (可能是給原本的方程式、給他一個清單叫他找東西)

函數 - 宣告

```
type name(type1 arg1, type2 arg2, ...)
{
    //...
    return result;
}
```

- `type` 是**回傳值類型**，可以想像成你要這個機器做完事要告訴你什麼類的東西 (可能是告訴你方程式的根，這時候就是浮點數、告訴你他搜尋到誰，這時候就可能是整數)
- `type1 arg1, ...` 是**傳入值**，可以想像成你給了這個機器什麼 (可能是給原本的方程式、給他一個清單叫他找東西)
- `return result;` 是**回傳**，可以想像當機器做到這行就會告訴你你叫他跟你講的東西，同時機器就會結束他的工作

函數 - 宣告

- 例如下面的程式碼會回傳一個正整數是不是奇數：

```
1  bool is_odd(int n)
2  {
3      if(n % 2 == 1)
4          return true;
5      else
6          return false;
7  }
```

函數 - 宣告

- 例如下面的程式碼會回傳一個正整數是不是奇數：

```
1  bool is_odd(int n)
2  {
3      if(n % 2 == 1)
4          return true;
5      else
6          return false;
7  }
```

- 注意它也可以這樣寫，因為 return 會強制終止那個函數：

```
1  bool is_odd(int n)
2  {
3      if(n % 2 == 1)
4          return true;
5      return false;
6  }
```

函數 - 宣告

- 例如下面的程式碼會回傳一個正整數是不是奇數：

```
1  bool is_odd(int n)
2  {
3      if(n % 2 == 1)
4          return true;
5      else
6          return false;
7  }
```

函數 - 宣告

- 例如下面的程式碼會回傳一個正整數是不是奇數：

```
1  bool is_odd(int n)
2  {
3      if(n % 2 == 1)
4          return true;
5      else
6          return false;
7  }
```

- 注意它也可以這樣寫：

```
1  bool is_odd(int n)
2  {
3      if(n % 2 == 1)
4          return true;
5      return false;
6  }
```

函數 - 宣告

- 例如下面的程式碼會回傳三個整數中最小的那個是多少：

```
1  int minimum(int a, int b, int c)
2  {
3      if(a <= b && a <= c)
4          return a;
5      if(b <= a && b <= c)
6          return b;
7      if(c <= a && c <= b)
8          return c;
9  }
```

函數 - 宣告

- 例如下面的程式碼會回傳三個整數中最小的那個是多少：

```
1  int minimum(int a, int b, int c)
2  {
3      if(a <= b && a <= c)
4          return a;
5      if(b <= a && b <= c)
6          return b;
7      if(c <= a && c <= b)
8          return c;
9  }
```

- 注意這個寫法也是一樣的意思：

```
1  int minimum(int a, int b, int c)
2  {
3      if (a <= b && a <= c)
4          return a;
5      else if (b <= a && b <= c)
6          return b;
7      return c;
8  }
```

函數 - 宣告

- 例如下面的程式碼會把 `arr` 這個陣列都重置成 `0`：

```
1  int arr[100];
2  void init()
3  {
4      for (int i = 0; i < 100; i++)
5          arr[i] = 0;
6  }
```

函數 - 宣告

- 例如下面的程式碼會把 `arr` 這個陣列都重置成 `0`：

```
1  int arr[100];
2  void init()
3  {
4      for (int i = 0; i < 100; i++)
5          arr[i] = 0;
6  }
```

- 注意到 `void` 是指**不回傳值**的意思，也就是不需要 `return`。

函數 - 宣告

- 例如下面的程式碼會把 `arr` 這個陣列都重置成 `0`：

```
1  int arr[100];
2  void init()
3  {
4      for (int i = 0; i < 100; i++)
5          arr[i] = 0;
6  }
```

- 注意到 `void` 是指**不回傳值**的意思，也就是不需要 `return`。
- 當然也是可以寫 `return;` 在裡面，這樣就只剩下「強制終止」的功能。

函數 - 宣告

- 例如下面的程式碼會回傳一個字串，表示某個月的季節：

```
1  #include <string>
2  using namespace std;
3
4  string season(int month)
5  {
6      if (3 <= month && month <= 5)
7          return "Spring";
8      else if (6 <= month && month <= 8)
9          return "Summer";
10     else if (9 <= month && month <= 11)
11         return "Autumn";
12     else
13         return "Winter";
14 }
```

函數 - 宣告

- 函數要宣告過才能用，所以要寫在 main 前面：

```
1  #include <iostream>
2  using namespace std;
3
4  string season(int month)
5  {
6      if (3 <= month && month <= 5)
7          return "Spring";
8      else if (6 <= month && month <= 8)
9          return "Summer";
10     else if (9 <= month && month <= 11)
11         return "Autumn";
12     else
13         return "Winter";
14 }
15
16 int main()
17 {
18     int n;
19     cin >> n;
20     cout << season(n) << "!\n";
21 }
```

函數 - 宣告

- 不過這樣 main 前面會有一大坨東西，如果不想這樣可以寫成：

```
1  #include <iostream>
2  using namespace std;
3
4  string season(int month);
5
6  int main()
7  {
8      int n;
9      cin >> n;
10     cout << season(n) << "!\n";
11 }
12
13 string season(int month)
14 {
15     if (3 <= month && month <= 5)
16         return "Spring";
17     else if (6 <= month && month <= 8)
18         return "Summer";
19     else if (9 <= month && month <= 11)
20         return "Autumn";
21     else
22         return "Winter";
23 }
```

函數 - 宣告

- 練習題：TOJ 23, TOJ 81
  能做完這兩題只需要會用函數就可以了。

函數 - Pass By Reference

- 例如下面的程式碼會交換兩個數字的值...

```
1  void swap(int a, int b)
2  {
3      int tmp = a;
4      a = b;
5      b = tmp;
6  }
```

嗎？

函數 - Pass By Reference

```
1  #include <iostream>
2  using namespace std;
3
4  void swap(int a, int b)
5  {
6      int tmp = a;
7      a = b;
8      b = tmp;
9  }
10
11 int main()
12 {
13     int a = 1, b = 2;
14     cout << a << " " << b << '\n';
15     swap(a, b);
16     cout << a << " " << b << '\n';
17 }
```

跑出來的結果是

```
1 2
1 2
```

- 到底發生什麼事？

函數 - Pass By Reference

- 這邊的問題是，函數是傳進值而不是傳進變數本身

函數 - Pass By Reference

- 這邊的問題是，函數是傳進值而不是傳進變數本身
- 換句話說剛剛發生的事是：

$$\begin{cases} \texttt{main}: & a=1, b=2 \\ \texttt{swap}: & a=1, b=2 \end{cases} \rightarrow \begin{cases} \texttt{main}: & a=1, b=2 \\ \texttt{swap}: & a=2, b=1 \end{cases}$$

函數 - Pass By Reference

- 這邊的問題是，函數是傳進值而不是傳進變數本身
- 換句話說剛剛發生的事是：

$$\begin{cases} \texttt{main}: & a=1, b=2 \\ \texttt{swap}: & a=1, b=2 \end{cases} \rightarrow \begin{cases} \texttt{main}: & a=1, b=2 \\ \texttt{swap}: & a=2, b=1 \end{cases}$$

- 要解決這個問題，我們希望函數去吃變數本身，換句話說就是讓機器直接去操作我們的東西，而不是複製一份再操作
  這個東西就稱作 Pass By Reference。

函數 - Pass By Reference

- 要用 Pass By Reference，在變數前面加個 & 就好了：

```
1 void swap(int &a, int &b)
2 {
3     int tmp = a;
4     a = b;
5     b = tmp;
6 }
```

函數 - Pass Array

- 特別講一個東西，要傳入陣列的時候比較複雜

函數 - Pass Array

- 特別講一個東西，要傳入陣列的時候比較複雜
  - 傳入一個固定大小的陣列

```
1  void init(int arr[100])
2  {
3      for (int i = 0; i < 100; i++)
4          arr[i] = 0;
5  }
```

函數 - Pass Array

- 特別講一個東西，要傳入陣列的時候比較複雜
  - 傳入一個固定大小的陣列

```
1  void init(int arr[100])
2  {
3      for (int i = 0; i < 100; i++)
4          arr[i] = 0;
5  }
```

  - 傳入一個不固定大小的陣列

```
1  void init(int n, int arr[])
2  {
3      for (int i = 0; i < n; i++)
4          arr[i] = 0;
5  }
```

函數 - Pass Array

- 特別講一個東西，要傳入陣列的時候比較複雜
  - 傳入一個固定大小的陣列

```
1  void init(int arr[100])
2  {
3      for (int i = 0; i < 100; i++)
4          arr[i] = 0;
5  }
```

  - 傳入一個不固定大小的陣列

```
1  void init(int n, int arr[])
2  {
3      for (int i = 0; i < n; i++)
4          arr[i] = 0;
5  }
```

- 最後注意一個小東西：陣列在改值的時候，不管有沒有 Pass By Reference，都會改到原本那份陣列
  詳細要等到位址與指標才會做說明。

函數 - Lambda Function

- 函數的宣告不只有一種，現在有另一種宣告方式了：Lambda Function

函數 - Lambda Function

- 函數的宣告不只有一種，現在有另一種宣告方式了：Lambda Function
- 他的缺點是比較不好懂，但優點是簡潔、可以匿名跟可以當物件操作

函數 - Lambda Function

- 函數的宣告不只有一種，現在有另一種宣告方式了：Lambda Function
- 他的缺點是比較不好懂，但優點是簡潔、可以匿名跟可以當物件操作
- 這裡就稍微講一下，更多的部分可以參考 `https://en.cppreference.com/w/cpp/language/lambda`

函數 - Lambda Function

```
1 [captures](params){body}
```

函數 - Lambda Function

```
1 [captures](params){body}
```

- captures 有三種：不填（代表不能拿函數外的東西做事）、&（表示拿函數外會直接改到他本身）、=（表示拿函數外會拿值而已）。

函數 - Lambda Function

```
1 [captures](params){body}
```

- captures 有三種：不填（代表不能拿函數外的東西做事）、&（表示拿函數外會直接改到他本身）、=（表示拿函數外會拿值而已）。
- params 跟函數傳入的東西一模一樣的寫法

函數 - Lambda Function

```
1 [captures](params){body}
```

- captures 有三種：不填（代表不能拿函數外的東西做事）、&（表示拿函數外會直接改到他本身）、=（表示拿函數外會拿值而已）。
- params 跟函數傳入的東西一模一樣的寫法
- body 也跟函數做事是一模一樣的寫法

函數 - Lambda Function

```
[captures](params){body}
```

- captures 有三種：不填（代表不能拿函數外的東西做事）、&（表示拿函數外會直接改到他本身）、=（表示拿函數外會拿值而已）。
- params 跟函數傳入的東西一模一樣的寫法
- body 也跟函數做事是一模一樣的寫法
- 例如這個就是 is_odd 的 Lambda Function 寫法：

```
auto is_odd = [](int n)
{
    if (n % 2 == 1)
        return true;
    return false;
};
```

函數

函數的部分到現在告一段落，希望你們還可以消化 (？
接下來要看遞迴的部分了 owo

遞迴 - 簡介

- 如果你想知道什麼是**遞迴**，請先去了解*遞迴*。
- 想知道更多關於*遞迴*的資訊，請參考**遞迴**。

遞迴 - 簡介

例題 - 階乘計算

輸出 $n!$。

$0 \leq n \leq 19$

- 這題當然有迴圈的寫法，但是我們來看看怎麼用遞迴思考。

遞迴 - 簡介

例題 - 階乘計算

輸出 $n!$。

$1 \leq n \leq 19$

遞迴 - 簡介

例題 - 階乘計算

輸出 $n!$。
$1 \leq n \leq 19$

- $n! = n \times (n-1) \times n - 2 \times \cdots \times 1$

遞迴 - 簡介

例題 - 階乘計算

輸出 $n!$。
$1 \leq n \leq 19$

- $n! = n \times (n-1) \times n - 2 \times \cdots \times 1$
- $n! = n \times (n-1)!$

遞迴 - 簡介

例題 - 階乘計算

輸出 $n!$。
$1 \leq n \leq 19$

- $n! = n \times (n-1) \times n - 2 \times \cdots \times 1$
- $n! = n \times (n-1)!$
- 如果我們有一個函數可以計算 $n!$，那我們就可以透過呼叫計算 $(n-1)!$ 再乘上 n 得到我們的答案。

遞迴 - 簡介

例題 - 階乘計算

輸出 $n!$。
$1 \leq n \leq 19$

- $n! = n \times (n-1) \times n - 2 \times \cdots \times 1$
- $n! = n \times (n-1)!$
- 如果我們有一個函數可以計算 $n!$，那我們就可以透過呼叫計算 $(n-1)!$ 再乘上 n 得到我們的答案。
- **好的遞迴可以幫我們減少問題的規模**

遞迴 - 簡介

例題 - 階乘計算

輸出 $n!$。

$1 \leq n \leq 19$

- 如果寫成程式碼，大概是這樣子：

```
1 long long factorial(int n)
2 {
3     return n * factorial(n - 1);
4 }
```

遞迴 - 簡介

例題 - 階乘計算

輸出 $n!$。

$1 \leq n \leq 19$

- 如果寫成程式碼，大概是這樣子：

```
long long factorial(int n)
{
    return n * factorial(n - 1);
}
```

- 不過如果你試著執行他，會跑不出答案

遞迴 - 簡介

例題 - 階乘計算

輸出 $n!$。

$1 \leq n \leq 19$

- 如果寫成程式碼，大概是這樣子：

```
long long factorial(int n)
{
    return n * factorial(n - 1);
}
```

- 不過如果你試著執行他，會跑不出答案
- 原因是因為當我呼叫 n，他會呼叫 $n-1$，再呼叫 $n-2$、$\cdots$，永無止境

遞迴 - 簡介

例題 - 階乘計算

輸出 $n!$。
$1 \leq n \leq 19$

- 如果寫成程式碼，大概是這樣子：

```
1 long long factorial(int n)
2 {
3     return n * factorial(n - 1);
4 }
```

- 不過如果你試著執行他，會跑不出答案
- 原因是因為當我呼叫 n，他會呼叫 $n-1$，再呼叫 $n-2$、$\cdots$，永無止境
- 因此我們需要**遞迴終止條件**。

遞迴 - 簡介

例題 - 階乘計算

輸出 $n!$。
$1 \leq n \leq 19$

- 在這個狀況內，當 $n = 1$ 的時候我們很容易就知道 $n! = 1! = 1$

遞迴 - 簡介

例題 - 階乘計算

輸出 $n!$。
$1 \leq n \leq 19$

- 在這個狀況內，當 $n = 1$ 的時候我們很容易就知道 $n! = 1! = 1$
- 同時這也是所有遞迴狀況中最簡單的 case，所以我們可以把他設成**遞迴終止條件**：

遞迴 - 簡介

例題 - 階乘計算

輸出 $n!$。

$1 \leq n \leq 19$

- 在這個狀況內，當 $n = 1$ 的時候我們很容易就知道 $n! = 1! = 1$
- 同時這也是所有遞迴狀況中最簡單的 case，所以我們可以把他設成**遞迴終止條件**：
- 當跑到 $n = 1$，就回傳 1 並且不要再遞迴下去

```
1  long long factorial(int n)
2  {
3      if (n == 1)
4          return 1;
5      return n * factorial(n - 1);
6  }
```

遞迴 - 遞迴搜尋

例題 - TOJ 656 枚舉子集和

有一個長度為 n 的陣列 a_n，請計算有多少個子集他的和是 k
子集的定義是在陣列選取 $0 \sim n$ 個不一定連續的元素，只要選的編號不同就視為不同的子集。
$1 \leq n \leq 20$, $-10^9 \leq k$, $a_n \leq 10^9$

遞迴 - 遞迴搜尋

例題 - TOJ 656 枚舉子集和

有一個長度為 n 的陣列 a_n，請計算有多少個子集他的和是 k
子集的定義是在陣列選取 $0 \sim n$ 個不一定連續的元素，只要選的編號不同就視為不同的子集。
$1 \leq n \leq 20$, $-10^9 \leq k$, $a_n \leq 10^9$

- 我們發現每個元素都只有取或不取，當然可以寫 20 圈的迴圈來做這件事

遞迴 - 遞迴搜尋

例題 - TOJ 656 枚舉子集和

有一個長度為 n 的陣列 a_n，請計算有多少個子集他的和是 k
子集的定義是在陣列選取 $0 \sim n$ 個不一定連續的元素，只要選的編號不同就視為不同的子集。
$1 \leq n \leq 20$, $-10^9 \leq k$, $a_n \leq 10^9$

- 我們發現每個元素都只有取或不取，當然可以寫 20 圈的迴圈來做這件事
- 但是每次遇到一個元素的情況都是取跟不取

遞迴 - 遞迴搜尋

例題 - TOJ 656 枚舉子集和

有一個長度為 n 的陣列 a_n，請計算有多少個子集他的和是 k
子集的定義是在陣列選取 $0 \sim n$ 個不一定連續的元素，只要選的編號不同就視為不同的子集。
$1 \leq n \leq 20, -10^9 \leq k, a_n \leq 10^9$

- 假設現在決定前 m 個要怎麼取，那 $m+1$ 個的狀況正好是再考慮取或不取分別看看

遞迴 - 遞迴搜尋

例題 - TOJ 656 枚舉子集和

有一個長度為 n 的陣列 a_n，請計算有多少個子集他的和是 k
子集的定義是在陣列選取 $0 \sim n$ 個不一定連續的元素，只要選的編號不同就視為不同的子集。
$1 \leq n \leq 20, -10^9 \leq k, a_n \leq 10^9$

- 假設現在決定前 m 個要怎麼取，那 $m+1$ 個的狀況正好是再考慮取或不取分別看看
- 終止條件是當所有都決定完了，看看這個方法有沒有等於 k

遞迴 - 遞迴搜尋

例題 - TOJ 656 枚舉子集和

有一個長度為 n 的陣列 a_n，請計算有多少個子集他的和是 k
子集的定義是在陣列選取 $0 \sim n$ 個不一定連續的元素，只要選的編號不同就視為不同的子集。
$1 \leq n \leq 20, -10^9 \leq k, a_n \leq 10^9$

- 假設現在決定前 m 個要怎麼取，那 $m+1$ 個的狀況正好是再考慮取或不取分別看看
- 終止條件是當所有都決定完了，看看這個方法有沒有等於 k
- 遞迴的時候我需要目前枚舉到第幾個、現在前面加起來是多少

遞迴 - 遞迴搜尋

例題 - TOJ 656 枚舉子集和

有一個長度為 n 的陣列 a_n，請計算有多少個子集他的和是 k
子集的定義是在陣列選取 $0 \sim n$ 個不一定連續的元素，只要選的編號不同就視為不同的子集。
$1 \leq n \leq 20$, $-10^9 \leq k$, $a_n \leq 10^9$

- 假設 `calculate(int m, long long sum)` 代表目前枚舉到第 `m` 個，前面選到的總和是 `sum`，而這樣的狀況有幾組總和是 k

遞迴 - 遞迴搜尋

例題 - TOJ 656 枚舉子集和

有一個長度為 n 的陣列 a_n，請計算有多少個子集他的和是 k
子集的定義是在陣列選取 $0 \sim n$ 個不一定連續的元素，只要選的編號不同就視為不同的子集。
$1 \leq n \leq 20, -10^9 \leq k, a_n \leq 10^9$

- 假設 `calculate(int m, long long sum)` 代表目前枚舉到第 `m` 個，前面選到的總和是 `sum`，而這樣的狀況有幾組總和是 k
- `calculate(m, sum) = calculate(m + 1, sum + a[m+1]) + calculate(m + 1, sum)`

遞迴 - 遞迴搜尋

例題 - TOJ 656 枚舉子集和

有一個長度為 n 的陣列 a_n，請計算有多少個子集他的和是 k
子集的定義是在陣列選取 $0 \sim n$ 個不一定連續的元素，只要選的編號不同就視為不同的子集。
$1 \leq n \leq 20, -10^9 \leq k, a_n \leq 10^9$

- 假設 `calculate(int m, long long sum)` 代表目前枚舉到第 `m` 個，前面選到的總和是 `sum`，而這樣的狀況有幾組總和是 k
- `calculate(m, sum) = calculate(m + 1, sum + a[m+1]) + calculate(m + 1, sum)`
- 當枚舉到 `m = n` 的時候，如果 `sum`$= k$ 就回傳 1

遞迴 - 遞迴搜尋

例題 - TOJ 656 枚舉子集和

有一個長度為 n 的陣列 a_n，請計算有多少個子集他的和是 k
子集的定義是在陣列選取 $0 \sim n$ 個不一定連續的元素，只要選的編號不同就視為不同的子集。
$1 \leq n \leq 20, -10^9 \leq k, a_n \leq 10^9$

- 假設 `calculate(int m, long long sum)` 代表目前枚舉到第 `m` 個，前面選到的總和是 `sum`，而這樣的狀況有幾組總和是 k
- `calculate(m, sum) = calculate(m + 1, sum + a[m+1]) + calculate(m + 1, sum)`
- 當枚舉到 `m = n` 的時候，如果 `sum`$= k$ 就回傳 1
- 題目要的答案就是 `calculate(0, 0)`（什麼都沒枚舉，也沒有加上任何東西）

遞迴 - 遞迴搜尋

```
1  int n;
2  long long arr[21], k;
3
4  int cal(int m, long long sum)
5  {
6      if(m == n)
7      {
8          if(sum == k)
9              return 1;
10         else
11             return 0;
12     }
13     else
14         return cal(m + 1, sum + arr[m + 1]) + cal(m + 1, sum);
15 }
```

遞迴 - 遞迴搜尋

練習題：

例題 - TOJ 656 枚舉子集和

有一個長度為 n 的陣列 a_n，請計算有多少個子集他的和是 k。
$1 \leq n \leq 20, -10^9 \leq k, a_n \leq 10^9$

類題 - TOJ 657 八皇后問題

有一個長度為 8×8 的棋盤，總共要放 8 隻皇后，
而棋盤上有一些缺格是不能放皇后的，
輸出有多少種放 8 隻皇后而且互不相吃的情況。

類題 - TOJ 655 分糖果

有 M 包不一定相同的糖果要分給 N 個人，有沒有辦法在不拆開包裝的情況下平分？
$N, M \leq 20$
這題要用到的技巧會比較難。

遞迴 - 遞迴優化

遞迴除了可以枚舉之外，也可以透過遞迴的情況來優化某些問題。
這部分又稱為分治演算法 (D&C, Divide and Conquer)。

遞迴 - 遞迴優化

例題 - TOJ 658 排序

有一個長度為 n 的陣列 a_n，請把他由小到大排序。
$1 \leq n \leq 2 \times 10^5$

遞迴 - 遞迴優化

在講這個演算法之前，先看簡化的版本：

Merge

有兩個已經從小到大排好的陣列 a_n、b_n
請把它們合併起來，而且合併完要由小到大排序。

遞迴 - 遞迴優化

在講這個演算法之前，先看簡化的版本：

Merge

有兩個已經從小到大排好的陣列 a_n、b_n
請把它們合併起來，而且合併完要由小到大排序。

- 策略是，從兩個的前面開始拿比較小的那個放到新的陣列前面

遞迴 - 遞迴優化

在講這個演算法之前，先看簡化的版本：

Merge

有兩個已經從小到大排好的陣列 a_n、b_n
請把它們合併起來，而且合併完要由小到大排序。

- 策略是，從兩個的前面開始拿比較小的那個放到新的陣列前面
- 因為兩個陣列都是由小到大的，這樣每次都會拿到最小的那個

遞迴 - 遞迴優化

在講這個演算法之前，先看簡化的版本：

Merge

有兩個已經從小到大排好的陣列 a_n、b_n
請把它們合併起來，而且合併完要由小到大排序。

- 策略是，從兩個的前面開始拿比較小的那個放到新的陣列前面
- 因為兩個陣列都是由小到大的，這樣每次都會拿到最小的那個
- 所以如果我們要排序一個陣列，我們可以把它剖成兩半，分別排序完再合併

遞迴 - 遞迴優化

在講這個演算法之前，先看簡化的版本：

Merge

有兩個已經從小到大排好的陣列 a_n、b_n
請把它們合併起來，而且合併完要由小到大排序。

- 策略是，從兩個的前面開始拿比較小的那個放到新的陣列前面
- 因為兩個陣列都是由小到大的，這樣每次都會拿到最小的那個
- 所以如果我們要排序一個陣列，我們可以把它剖成兩半，分別排序完再合併
- 那排序一半的陣列呢？再它剖成兩半，分別排序完再合併

遞迴 - 遞迴優化

在講這個演算法之前，先看簡化的版本：

Merge

有兩個已經從小到大排好的陣列 a_n、b_n
請把它們合併起來，而且合併完要由小到大排序。

- 策略是，從兩個的前面開始拿比較小的那個放到新的陣列前面
- 因為兩個陣列都是由小到大的，這樣每次都會拿到最小的那個
- 所以如果我們要排序一個陣列，我們可以把它剖成兩半，分別排序完再合併
- 那排序一半的陣列呢？再它剖成兩半，分別排序完再合併
- 剖到最後會只剩下 1 個元素，只有 1 個元素的陣列一定是排序好的！

遞迴 - 遞迴優化

回到原本的問題：

例題 - TOJ 658 排序

有一個長度為 n 的陣列 a_n，請把他由小到大排序。
$1 \leq n \leq 2 \times 10^5$

- 利用剛剛的策略，似乎就解決這個問題了。

遞迴 - 遞迴優化

回到原本的問題：

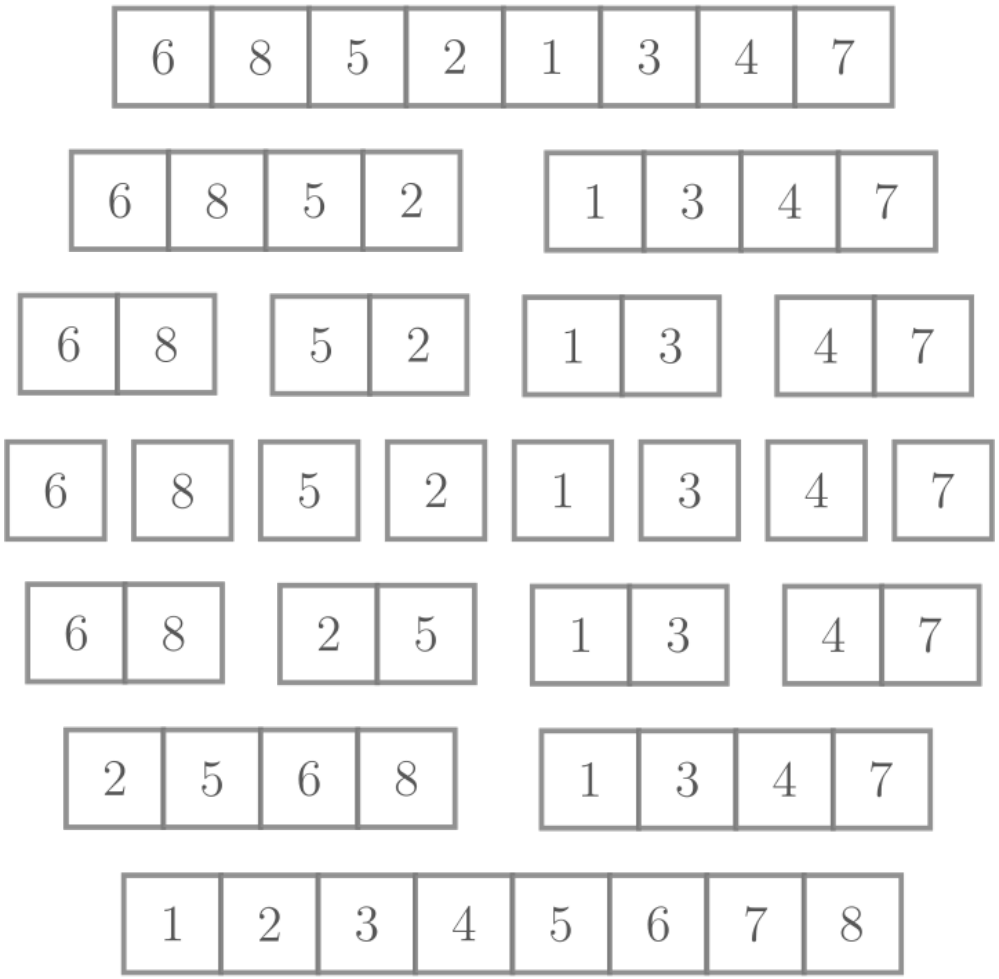
例題 - TOJ 658 排序

有一個長度為 n 的陣列 a_n，請把他由小到大排序。
$1 \leq n \leq 2 \times 10^5$

- 利用剛剛的策略，似乎就解決這個問題了。
- 這個演算法還算快，比起每次直接硬找最小值的方法好很多。

遞迴 - 遞迴優化

圖解一下剛剛的想法：



遞迴 - 遞迴優化

程式碼：

```
long long arr[200005], tmp[2000005];
void merge_sort(int L, int R)
{
    if (L == R)
        return;
    int mid = (L + R) / 2;
    merge_sort(L, mid);
    merge_sort(mid + 1, R);
    int lp = L, rp = mid + 1;
    for (int i = L; i <= R; i++)
        if (rp > R || (lp <= mid && arr[lp] <= arr[rp]))
        {
            tmp[i] = arr[lp];
            lp++;
        }
        else
        {
            tmp[i] = arr[rp];
            rp++;
        }
    for (int i = L; i <= R; i++)
        arr[i] = tmp[i];
}
```

遞迴 - 遞迴優化

練習題：

類題 - TOJ 659 猜數字（互動題）

LittleCube 藏有一個介於 $0 \sim 2^{31} - 1$ 的數字，每次你問他一個數字，他都會告訴你太大、相等、或太小。
你能用 32 次就把這個數字猜出來嗎？

例題 - TOJ 658 排序

有一個長度為 n 的陣列 a_n，請把他由小到大排序。
$1 \leq n \leq 2 \times 10^5$

類題 - TOJ 11 bubble

有一個長度為 n 的陣列 a_n，如果每次只交換相鄰兩個數字來把陣列排成由小到大，
最少需要交換幾次？
$1 \leq n \leq 2 \times 10^5$