資料輸入與輸出瓶頸

- 輸入與輸出設備 (螢幕、鍵盤、磁碟) 的傳輸與處理速度比 CPU 或記憶體慢很多，所以當測資數量很大 **(10^7 位元組)** 時，使用一般的輸入與輸出函式花的時間會是運算時間的很多倍，造成邏輯正確的程式評測時執行超時 (**TLE**, Time Limit Exceeded)

資料輸入與輸出瓶頸

- 輸入與輸出設備 (螢幕、鍵盤、磁碟) 的傳輸與處理速度比 CPU 或記憶體慢很多，所以當**測資數量很大 (10^7 位元組)** 時，使用一般的輸入與輸出函式花的時間會是運算時間的很多倍，造成邏輯正確的程式評測時執行**超時** (**TLE**, Time Limit Exceeded)

```
for (i=0; i<10000000; i++)
    x = getchar();
```

資料輸入與輸出瓶頸

- 輸入與輸出設備 (螢幕、鍵盤、磁碟) 的傳輸與處理速度比 CPU 或記憶體慢很多，所以當**測資數量很大 (10^7 位元組)** 時，使用一般的輸入與輸出函式花的時間會是運算時間的很多倍，造成邏輯正確的程式評測時執行**超時** (**TLE**, Time Limit Exceeded)

```
for (i=0; i<10000000; i++)
    x = getchar();
```

由硬碟讀資料需要 1.535 s

資料輸入與輸出瓶頸

- 輸入與輸出設備 (螢幕、鍵盤、磁碟) 的傳輸與處理速度比 CPU 或記憶體慢很多，所以當**測資數量很大 (10^7 位元組)** 時，使用一般的輸入與輸出函式花的時間會是運算時間的很多倍，造成邏輯正確的程式評測時執行**超時** (**TLE**, Time Limit Exceeded)

```
for (i=0; i<10000000; i++)
    x = getchar();
```

由硬碟讀資料需要 1.535 s

```
for (i=0; i<10000000; i++)
    printf("%c", 'A');
```

資料輸入與輸出瓶頸

- 輸入與輸出設備 (螢幕、鍵盤、磁碟) 的傳輸與處理速度比 CPU 或記憶體慢很多，所以當**測資數量很大 (10^7 位元組)** 時，使用一般的輸入與輸出函式花的時間會是運算時間的很多倍，造成邏輯正確的程式評測時執行**超時** (**TLE**, Time Limit Exceeded)

```
for (i=0; i<10000000; i++)
    x = getchar();
```

由硬碟讀資料需要 1.535 s

```
for (i=0; i<10000000; i++)
    printf("%c", 'A');
```

寫資料到硬碟需要 0.85 s

資料輸入與輸出瓶頸

- 輸入與輸出設備 (螢幕、鍵盤、磁碟) 的傳輸與處理速度比 CPU 或記憶體慢很多，所以當**測資數量很大 (10^7 位元組)** 時，使用一般的輸入與輸出函式花的時間會是運算時間的很多倍，造成邏輯正確的程式評測時執行**超時** (**TLE**, Time Limit Exceeded)

```
for (i=0; i<10000000; i++)
    x = getchar();
```

由硬碟讀資料需要 1.535 s

```
for (i=0; i<10000000; i++)
    printf("%c", 'A');
```

寫資料到硬碟需要 0.85 s

寫資料到螢幕需要 240 s

資料輸入與輸出瓶頸

- 輸入與輸出設備 (螢幕、鍵盤、磁碟) 的傳輸與處理速度比 CPU 或記憶體慢很多，所以當**測資數量很大 (10^7 位元組)** 時，使用一般的輸入與輸出函式花的時間會是運算時間的很多倍，造成邏輯正確的程式評測時執行**超時** (**TLE**, Time Limit Exceeded)

```
for (i=0; i<10000000; i++)
    x = getchar();
```

由硬碟讀資料需要 1.535 s

```
for (i=0; i<10000000; i++)
    printf("%c", 'A');
```

寫資料到硬碟需要 0.85 s
寫資料到螢幕需要 240 s

- 各個線上評測平台 (UVa, ZeroJudge, …) 上都會遇見這種測資

資料輸入與輸出瓶頸

- 輸入與輸出設備 (螢幕、鍵盤、磁碟) 的傳輸與處理速度比 CPU 或記憶體慢很多，所以當測資數量很大 **(10^7 位元組)** 時，使用一般的輸入與輸出函式花的時間會是運算時間的很多倍，造成邏輯正確的程式評測時執行超時 (**TLE**, Time Limit Exceeded)

```
for (i=0; i<10000000; i++)
    x = getchar();
```

由硬碟讀資料需要 1.535 s

```
for (i=0; i<10000000; i++)
    printf("%c", 'A');
```

寫資料到硬碟需要 0.85 s
寫資料到螢幕需要 240 s

- 各個線上評測平台 (UVa, ZeroJudge, …) 上都會遇見這種測資
- 大學程式設計先修檢測 **(APCS)** 的目標應該是運算思維、處理方法設計、及程式設計，因此到目前為止還沒有針對這樣子設計的測資，如果目標只是參加檢測的話，建議可以跳過下面幾頁

初步加快速度

- **C/C++ stdio.h / cstdio**

初步加快速度

- **C/C++ stdio.h / cstdio**

```
for (i=0; i<10000000; i++)
    x = getchar_unlocked();
```

初步加快速度

- **C/C++ stdio.h / cstdio**

```
for (i=0; i<10000000; i++)
    x = getchar_unlocked();
```

FreeBSD　　0.591 (getchar)　⇒　0.031 (getchar_unlocked)

初步加快速度

- **C/C++ stdio.h / cstdio**

```
for (i=0; i<10000000; i++)
    x = getchar_unlocked();
```

FreeBSD　　0.591 (getchar) ⇒ 0.031 (getchar_unlocked)

Nonlocking stdio functions

getc_unlocked, getchar_unlocked
fread_unlocked, fwrite_unlocked
putc_unlocked, putchar_unlocked
fgets_unlocked, fputs_unlocked

初步加快速度

- **C/C++ stdio.h / cstdio**

Nonlocking stdio functions

getc_unlocked, getchar_unlocked
fread_unlocked, fwrite_unlocked
putc_unlocked, putchar_unlocked
fgets_unlocked, fputs_unlocked

```
for (i=0; i<10000000; i++)
    x = getchar_unlocked();
```

FreeBSD　0.591 (getchar) ⇒ 0.031 (getchar_unlocked)

讀取正整數

```
1 int scani() {
2     register int x, c=getchar_unlocked();
3     while(c<'0'||c>'9'){if(c<0)return-1;c=getchar_unlocked();}
4     for(x=0;c>='0'&&c<='9';c=getchar_unlocked())
5         x = (x<<1) + (x<<3) + (c&15);
6     return x;
7 }
```

初步加快速度

- **C/C++ stdio.h / cstdio**

Nonlocking stdio functions

getc_unlocked, getchar_unlocked
fread_unlocked, fwrite_unlocked
putc_unlocked, putchar_unlocked
fgets_unlocked, fputs_unlocked

```
for (i=0; i<10000000; i++)
    x = getchar_unlocked();
```

FreeBSD 0.591 (getchar) ⇒ 0.031 (getchar_unlocked)

讀取正整數

```
1 int scani() {
2     register int x, c=getchar_unlocked();
3     while(c<'0'||c>'9'){if(c<0)return-1;c=getchar_unlocked();}
4     for(x=0;c>='0'&&c<='9';c=getchar_unlocked())
5         x = (x<<1) + (x<<3) + (c&15);
6     return x;
7 }
```

```
1 while ((n=scani())>=0) {
2     for (i=0; i<n; i++) {
3         x = scani(), y = scanfi();
          …
4     }
5 }
```

初步加快速度

- **C/C++ stdio.h / cstdio**

```
for (i=0; i<10000000; i++)
    x = getchar_unlocked();
```

Nonlocking stdio functions

getc_unlocked, getchar_unlocked
fread_unlocked, fwrite_unlocked
putc_unlocked, putchar_unlocked
fgets_unlocked, fputs_unlocked

FreeBSD　0.591 (getchar) ⇒ 0.031 (getchar_unlocked)

讀取正整數

```
1 int scani() {
2     register int x, c=getchar_unlocked();
3     while(c<'0'||c>'9'){if(c<0)return-1;c=getchar_unlocked();}
4     for(x=0;c>='0'&&c<='9';c=getchar_unlocked())
5         x = (x<<1) + (x<<3) + (c&15);
6     return x;
7 }
```

```
1 while ((n=scani())>=0) {
2     for (i=0; i<n; i++) {
3         x = scani(), y = scanfi();
          …
4     }
5 }
```

- **C++ iostream**

初步加快速度

Nonlocking stdio functions

getc_unlocked, getchar_unlocked
fread_unlocked, fwrite_unlocked
putc_unlocked, putchar_unlocked
fgets_unlocked, fputs_unlocked

- **C/C++ stdio.h / cstdio**

```
for (i=0; i<10000000; i++)
    x = getchar_unlocked();
```

FreeBSD 0.591 (getchar) ⇒ 0.031 (getchar_unlocked)

讀取正整數

```
1 int scani() {
2     register int x, c=getchar_unlocked();
3     while(c<'0'||c>'9'){if(c<0)return-1;c=getchar_unlocked();}
4     for(x=0;c>='0'&&c<='9';c=getchar_unlocked())
5         x = (x<<1) + (x<<3) + (c&15);
6     return x;
7 }
```

```
1 while ((n=scani())>=0) {
2     for (i=0; i<n; i++) {
3         x = scani(), y = scanfi();
          …
4     }
5 }
```

- **C++ iostream**

```
1 ios_base::sync_with_stdio(false);
2 cin.tie(0);
3 cin >> x >> y; …
4 cout << x << y; …
```

初步加快速度

Nonlocking stdio functions

getc_unlocked, getchar_unlocked
fread_unlocked, fwrite_unlocked
putc_unlocked, putchar_unlocked
fgets_unlocked, fputs_unlocked

- **C/C++ stdio.h / cstdio**

```
for (i=0; i<10000000; i++)
    x = getchar_unlocked();
```

FreeBSD 0.591 (getchar) ⇒ 0.031 (getchar_unlocked)

讀取正整數

```
1 int scani() {
2     register int x, c=getchar_unlocked();
3     while(c<'0'||c>'9'){if(c<0)return-1;c=getchar_unlocked();}
4     for(x=0;c>='0'&&c<='9';c=getchar_unlocked())
5         x = (x<<1) + (x<<3) + (c&15);
6     return x;
7 }
```

```
1 while ((n=scani())>=0) {
2     for (i=0; i<n; i++) {
3         x = scani(), y = scanfi();
          …
4     }
5 }
```

- **C++ iostream**

```
1 ios_base::sync_with_stdio(false);
2 cin.tie(0);
3 cin >> x >> y; …
4 cout << x << y; …
```

https://chino.taipei/note-2016-0311C-的輸出入cin-cout和scanf-printf誰比較快？/

自行實作輸入/輸出緩衝機制

- 輸入/輸出函式其實非常花時間，測資的數量很多時如果使用標準的函式來處理，光讀寫資料就已經超過時間限制

自行實作輸入/輸出緩衝機制

- 輸入/輸出函式其實非常花時間，測資的數量很多時如果使用標準的函式來處理，光讀寫資料就已經超過時間限制
- 輸入：一次讀取100KB資料到記憶體，順序把資料拿出來處理

自行實作輸入/輸出緩衝機制

- 輸入/輸出函式其實非常花時間，測資的數量很多時如果使用標準的函式來處理，光讀寫資料就已經超過時間限制
- 輸入：一次讀取100KB資料到記憶體，順序把資料拿出來處理

```
01 // 假設輸入資料中每一列有一個整數
02 int ic, offset=0, x;
03 char ib[102500], *end, *ptr;
```

自行實作輸入/輸出緩衝機制

- 輸入/輸出函式其實非常花時間，測資的數量很多時如果使用標準的函式來處理，光讀寫資料就已經超過時間限制
- 輸入：一次讀取100KB資料到記憶體，順序把資料拿出來處理

```
01 // 假設輸入資料中每一列有一個整數
02 int ic, offset=0, x;
03 char ib[102500], *end, *ptr;

04 while ((ic=fread_unlocked(ib+offset, 1, 102400, stdin))>0) {
05     if (ic<102400) end = ib+offset+ic-1;
06     else for (end=ib+offset+ic-1; *end!='\n'; end--);
07     ptr = ib-1;
08     while (ptr<end) {
09         for (x=0,ptr++; ptr<=end&&*ptr>='0'&&*ptr<='9'; ptr++)
10             x = (x<<1) + (x<<3) + (*ptr&15);
11         // x 是每一列輸入的整數，處理 x
12     }
13     for (ptr=ib,end++; end<ib+offset+ic; ptr++,end++)  *ptr = *end;
14     offset = ptr - ib;
15 }
```

自行實作輸入/輸出緩衝機制 (續)

- 輸出：順序處理資料，寫資料到200KB的記憶體中，緩衝區滿了再一次輸出

自行實作輸入/輸出緩衝機制 (續)

- 輸出：順序處理資料，寫資料到200KB的記憶體中，緩衝區滿了再一次輸出

```
01 int oc=0, x, y, i;
02 char ob[200100], t[20];
```

自行實作輸入/輸出緩衝機制 (續)

- 輸出：順序處理資料，寫資料到200KB的記憶體中，緩衝區滿了再一次輸出

```
01 int oc=0, x, y, i;
02 char ob[200100], t[20];

03 while (more data) {
04     // ... results is an integer x

05     for (i=0; x>0; x/=10,i++) t[i] = x % 10 + '0';
06     while (i>0) ob[oc++] = t[--i]; ob[oc++] = '\n';

07     if (oc>200000) {
08         y=ob[200000]; ob[200000]=0; printf("%s", ob); ob[0]=y;
09         for (i=200001; i<oc; i++)
10             ob[i-200000] = ob[i];
11         oc = oc - 200000;
12     }
13 }
14 ob[oc]=0; printf("%s", ob);
```

二分搜尋法概念

- data[] 陣列裡存放由小到大順序排好的資料

data[]

0	1	2	3	4	5	6	7	8	9	10
2	3	4	5	6	8	13	16	18	23	24

二分搜尋法概念

- data[] 陣列裡存放由小到大順序排好的資料

data[]

0	1	2	3	4	5	6	7	8	9	10
2	3	4	5	6	8	13	16	18	23	24

- 想要確定 **16** 這筆資料放在陣列的什麼地方 (data[7])

二分搜尋法概念

- data[] 陣列裡存放由小到大順序排好的資料

data[]

0	1	2	3	4	5	6	7	8	9	10
2	3	4	5	6	8	13	16	18	23	24

- 想要確定 16 這筆資料放在陣列的什麼地方 (data[7])
  - 最簡單的方法就是由 data[0] 開始『一個一個比對』

二分搜尋法概念

- data[] 陣列裡存放由小到大順序排好的資料

data[]

0	1	2	3	4	5	6	7	8	9	10
2	3	4	5	6	8	13	16	18	23	24

- 想要確定 16 這筆資料放在陣列的什麼地方 (data[7])
  - 最簡單的方法就是由 data[0] 開始『一個一個比對』
  - 但是這樣子完全沒有運用到資料順序已經排好的特性…資料都整理好了, 竟然當成沒有整理過的資料來使用

二分搜尋法概念

- data[] 陣列裡存放由小到大順序排好的資料

data[]

0	1	2	3	4	5	6	7	8	9	10
2	3	4	5	6	8	13	16	18	23	24

- 想要確定 16 這筆資料放在陣列的什麼地方 (data[7])
  - 最簡單的方法就是由 data[0] 開始『一個一個比對』
  - 但是這樣子完全沒有運用到資料順序已經排好的特性…資料都整理好了, 竟然當成沒有整理過的資料來使用
  - 因為資料已經排好順序, 如果發現 16 > 8, 8 之前所有的資料 (data[0]~data[4]) 都不需要考慮了, 一定都小於 16

二分搜尋法概念

- data[] 陣列裡存放由小到大順序排好的資料

data[]

0	1	2	3	4	5	6	7	8	9	10
2	3	4	5	6	8	13	16	18	23	24

- 想要確定 **16** 這筆資料放在陣列的什麼地方 (data[7])
  - 最簡單的方法就是由 data[0] 開始『一個一個比對』
  - 但是這樣子完全沒有運用到資料順序已經排好的特性…資料都整理好了, 竟然當成沒有整理過的資料來使用
  - 因為資料已經排好順序, 如果發現 **16** > 8, 8 之前所有的資料 (data[0]~data[4]) 都不需要考慮了, 一定都小於 **16**
- **二分搜尋法 (Binary Search, 二元搜尋法)**

二分搜尋法概念

- data[] 陣列裡存放由小到大順序排好的資料

data[]

0	1	2	3	4	5	6	7	8	9	10
2	3	4	5	6	8	13	16	18	23	24

- 想要確定 16 這筆資料放在陣列的什麼地方 (data[7])
  - 最簡單的方法就是由 data[0] 開始『一個一個比對』
  - 但是這樣子完全沒有運用到資料順序已經排好的特性…資料都整理好了, 竟然當成沒有整理過的資料來使用
  - 因為資料已經排好順序, 如果發現 16 > 8, 8 之前所有的資料 (data[0]~data[4]) 都不需要考慮了, 一定都小於 16
- 二分搜尋法 (Binary Search, 二元搜尋法)

搜尋目標 16

0	1	2	3	4	5	6	7	8	9	10
2	3	4	5	6	8	13	16	18	23	24

資料陣列中共有 11 個元素

二分搜尋法概念

- data[] 陣列裡存放由小到大順序排好的資料

data[]

0	1	2	3	4	5	6	7	8	9	10
2	3	4	5	6	8	13	16	18	23	24

- 想要確定 16 這筆資料放在陣列的什麼地方 (data[7])
  - 最簡單的方法就是由 data[0] 開始『一個一個比對』
  - 但是這樣子完全沒有運用到資料順序已經排好的特性…資料都整理好了, 竟然當成沒有整理過的資料來使用
  - 因為資料已經排好順序, 如果發現 16 > 8, 8 之前所有的資料 (data[0]~data[4]) 都不需要考慮了, 一定都小於 16
- 二分搜尋法 (Binary Search, 二元搜尋法)

搜尋目標 16

0	1	2	3	4	5	6	7	8	9	10
2	3	4	5	6	8	13	16	18	23	24

資料陣列中共有 11 個元素

最接近中間的元素是 8

二分搜尋法概念

- data[] 陣列裡存放由小到大順序排好的資料

data[]

0	1	2	3	4	5	6	7	8	9	10
2	3	4	5	6	8	13	16	18	23	24

- 想要確定 16 這筆資料放在陣列的什麼地方 (data[7])
  - 最簡單的方法就是由 data[0] 開始『一個一個比對』
  - 但是這樣子完全沒有運用到資料順序已經排好的特性…資料都整理好了, 竟然當成沒有整理過的資料來使用
  - 因為資料已經排好順序, 如果發現 16 > 8, 8 之前所有的資料 (data[0]~data[4]) 都不需要考慮了, 一定都小於 16
- 二分搜尋法 (Binary Search, 二元搜尋法)

搜尋目標 16

0	1	2	3	4	5	6	7	8	9	10
2	3	4	5	6	8	13	16	18	23	24

資料陣列中共有 11 個元素

最接近中間的元素是 8

16 > 8, 16 位於後半段

二分搜尋法概念

- data[] 陣列裡存放由小到大順序排好的資料

data[]

0	1	2	3	4	5	6	7	8	9	10
2	3	4	5	6	8	13	16	18	23	24

- 想要確定 16 這筆資料放在陣列的什麼地方 (data[7])
  - 最簡單的方法就是由 data[0] 開始『一個一個比對』
  - 但是這樣子完全沒有運用到資料順序已經排好的特性…資料都整理好了, 竟然當成沒有整理過的資料來使用
  - 因為資料已經排好順序, 如果發現 16 > 8, 8 之前所有的資料 (data[0]~data[4]) 都不需要考慮了, 一定都小於 16
- 二分搜尋法 (Binary Search, 二元搜尋法)

搜尋目標 16

6	7	8	9	10
13	16	18	23	24

資料陣列中共有 5 個元素

二分搜尋法概念

- data[] 陣列裡存放由小到大順序排好的資料

data[]

0	1	2	3	4	5	6	7	8	9	10
2	3	4	5	6	8	13	16	18	23	24

- 想要確定 **16** 這筆資料放在陣列的什麼地方 (data[7])
  - 最簡單的方法就是由 data[0] 開始『一個一個比對』
  - 但是這樣子完全沒有運用到資料順序已經排好的特性…資料都整理好了, 竟然當成沒有整理過的資料來使用
  - 因為資料已經排好順序, 如果發現 **16** > 8, 8 之前所有的資料 (data[0]~data[4]) 都不需要考慮了, 一定都小於 **16**
- **二分搜尋法 (Binary Search, 二元搜尋法)**

搜尋目標 **16**

6	7	8	9	10
13	**16**	**18**	**23**	**24**

資料陣列中共有 **5** 個元素

最接近中間的元素是 **18**

二分搜尋法概念

- data[] 陣列裡存放由小到大順序排好的資料

data[]

0	1	2	3	4	5	6	7	8	9	10
2	3	4	5	6	8	13	16	18	23	24

- 想要確定 16 這筆資料放在陣列的什麼地方 (data[7])
  - 最簡單的方法就是由 data[0] 開始『一個一個比對』
  - 但是這樣子完全沒有運用到資料順序已經排好的特性…資料都整理好了, 竟然當成沒有整理過的資料來使用
  - 因為資料已經排好順序, 如果發現 16 > 8, 8 之前所有的資料 (data[0]~data[4]) 都不需要考慮了, 一定都小於 16
- **二分搜尋法 (Binary Search, 二元搜尋法)**

搜尋目標 16

6	7	8	9	10
13	16	18	23	24

資料陣列中共有 5 個元素

最接近中間的元素是 18

16 < 18, 16 位於前半段

二分搜尋法概念

- data[] 陣列裡存放由小到大順序排好的資料

data[]

0	1	2	3	4	5	6	7	8	9	10
2	3	4	5	6	8	13	16	18	23	24

- 想要確定 16 這筆資料放在陣列的什麼地方 (data[7])
  - 最簡單的方法就是由 data[0] 開始『一個一個比對』
  - 但是這樣子完全沒有運用到資料順序已經排好的特性…資料都整理好了, 竟然當成沒有整理過的資料來使用
  - 因為資料已經排好順序, 如果發現 16 > 8, 8 之前所有的資料 (data[0]~data[4]) 都不需要考慮了, 一定都小於 16
- 二分搜尋法 (Binary Search, 二元搜尋法)

搜尋目標 16

6	7
13	16

資料陣列中共有 2 個元素

二分搜尋法概念

- data[] 陣列裡存放由小到大順序排好的資料

data[]

0	1	2	3	4	5	6	7	8	9	10
2	3	4	5	6	8	13	16	18	23	24

- 想要確定 16 這筆資料放在陣列的什麼地方 (data[7])
  - 最簡單的方法就是由 data[0] 開始『一個一個比對』
  - 但是這樣子完全沒有運用到資料順序已經排好的特性…資料都整理好了, 竟然當成沒有整理過的資料來使用
  - 因為資料已經排好順序, 如果發現 16 > 8, 8 之前所有的資料 (data[0]~data[4]) 都不需要考慮了, 一定都小於 16
- **二分搜尋法 (Binary Search, 二元搜尋法)**

搜尋目標 16

6	7
13	16

資料陣列中共有 2 個元素

最接近中間的元素是 13

二分搜尋法概念

- data[] 陣列裡存放由小到大順序排好的資料

data[]

0	1	2	3	4	5	6	7	8	9	10
2	3	4	5	6	8	13	16	18	23	24

- 想要確定 16 這筆資料放在陣列的什麼地方 (data[7])
  - 最簡單的方法就是由 data[0] 開始『一個一個比對』
  - 但是這樣子完全沒有運用到資料順序已經排好的特性…資料都整理好了, 竟然當成沒有整理過的資料來使用
  - 因為資料已經排好順序, 如果發現 16 > 8, 8 之前所有的資料 (data[0]~data[4]) 都不需要考慮了, 一定都小於 16
- 二分搜尋法 (Binary Search, 二元搜尋法)

搜尋目標 16

6	7
13	16

資料陣列中共有 2 個元素

最接近中間的元素是 13

16 > 13, 16 位於後半段

二分搜尋法概念

- data[] 陣列裡存放由小到大順序排好的資料

data[]

0	1	2	3	4	5	6	7	8	9	10
2	3	4	5	6	8	13	16	18	23	24

- 想要確定 16 這筆資料放在陣列的什麼地方 (data[7])
  - 最簡單的方法就是由 data[0] 開始『一個一個比對』
  - 但是這樣子完全沒有運用到資料順序已經排好的特性…資料都整理好了, 竟然當成沒有整理過的資料來使用
  - 因為資料已經排好順序, 如果發現 16 > 8, 8 之前所有的資料 (data[0]~data[4]) 都不需要考慮了, 一定都小於 16
- 二分搜尋法 (Binary Search, 二元搜尋法)

搜尋目標 16

7

16

資料陣列中共有 1 個元素

二分搜尋法概念

- data[] 陣列裡存放由小到大順序排好的資料

data[]

0	1	2	3	4	5	6	7	8	9	10
2	3	4	5	6	8	13	16	18	23	24

- 想要確定 **16** 這筆資料放在陣列的什麼地方 (data[7])
  - 最簡單的方法就是由 data[0] 開始『一個一個比對』
  - 但是這樣子完全沒有運用到資料順序已經排好的特性…資料都整理好了, 竟然當成沒有整理過的資料來使用
  - 因為資料已經排好順序, 如果發現 **16** > 8, 8 之前所有的資料 (data[0]~data[4]) 都不需要考慮了, 一定都小於 **16**
- **二分搜尋法 (Binary Search, 二元搜尋法)**

搜尋目標 **16**

7

16

資料陣列中共有 **1** 個元素

最接近中間的元素是 **16**

二分搜尋法概念

- data[] 陣列裡存放由小到大順序排好的資料

data[]

0	1	2	3	4	5	6	7	8	9	10
2	3	4	5	6	8	13	16	18	23	24

- 想要確定 16 這筆資料放在陣列的什麼地方 (data[7])
  - 最簡單的方法就是由 data[0] 開始『一個一個比對』
  - 但是這樣子完全沒有運用到資料順序已經排好的特性…資料都整理好了, 竟然當成沒有整理過的資料來使用
  - 因為資料已經排好順序, 如果發現 16 > 8, 8 之前所有的資料 (data[0]~data[4]) 都不需要考慮了, 一定都小於 16
- 二分搜尋法 (Binary Search, 二元搜尋法)

搜尋目標 16

7

16

資料陣列中共有 1 個元素

最接近中間的元素是 16

16 == 16, 找到了