程式寫作講堂

EP01: 從兩個已經排好大小的陣列中尋找等值對

需要一間十分差的學校才能毀掉一位好學生
但是
卻要一間極好的學校才能救起一位差學生

Dennis J. Frailey

本集內容

- 我們討論一道很簡單的程式設計習題，我們會
  - 從一個初學者都可能會想到的粗淺解法開始
  - 進步到使用一點最基本資料結構的較快解法
  - 最後到達一個最佳解答
  - 每一種解法都會用比較次數衡量該解法的效率
  - 也會討論到一些改良
- 影片說明欄有這系列影片的網頁、您可以從該處下載投影片和程式

何謂等值對?

已知兩組資料 X = { $x_1, x_2, \cdots, x_m$} 和 Y = { $y_1, y_2, \cdots, y_n$}, 一組 <u>等值對</u> 就是在 X 中的一個 x_i 和在 Y中的一個 y_j 它們的值相等 $x_i = y_j$。

問題

已知兩個陣列 x[ ] 和 y[ ]，
每一個陣列都包含了一組自小到大排好、
而且互不相等的整數，
請寫一個C（或用任何其它語言）程式，
找出這兩個陣列之間的所有等值對。

粗淺的想法: 1/5

```
int x[m], y[n];

for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
        if (x[i] == y[j]) {
            // an equal pair found
            // report it
        }
        else {
            // not an equal pair
            // perhaps do nothing
        }
    }
}
```

1. 假設 `x[]` 和 `y[]` 各有 `m` 和 `n` 個數
2. 粗淺的想法可能如此:
   a) 對每一個 `i` 而言，拿 `x[i]` 和每一個 `y[j]` 比較。
   b) 若相同，就找到一組**等值對**。

粗淺的想法: 2/5

```
int x[m], y[n];

for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
        if (x[i] == y[j]) {
            // an equal pair found
            // report it
        }
        else {
            // not an equal pair
            // perhaps do nothing
        }
    }
}
```

1. 若 `x[i]` 和 `y[j]` 不相等，當然不必做任何事。
2. 於是，`else` 部分就是空白，我們比下一組，也就是 `x[i]` 和 `y[j+1]`。

粗淺的想法: 3/5

```
int x[m], y[n];

for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
        if (x[i] == y[j]) {
            // an equal pair found
            // report it
        }
        else {
            // not an equal pair
            // perhaps do nothing
        }
    }
}
```

1. 若 `x[i]` 和 `y[j]` 相同，我們得紀錄這一組。
2. 在 `then` 部份，我們得保存 `i` 和 `j` 的值，並且把一個“計數器”加1。然後，再接著比下一組 `x[i]` 和 `y[j+1]`。

粗淺的想法: 4/5

```
int x[m], y[n];
int xx[ ], yy[ ], k = 0;

for (i = 0; i < m; i++)
    for (j = 0; j < n; j++) {
        if (x[i] == y[j]) {
            xx[k] = i;
            yy[k] = j;
            k++;
            break;
        }
}
```

因為陣列元素相異，一旦找到一組，y[j+1]就是個不同的值。

1. 用兩個陣列 xx[ ]和 yy[ ] 分別保存 i 和 j 的值。
2. int k 是一個計數器，每找到一組等值對就加1。
3. 程式中的 break 是用來在找到一組後跳出 j 迴圈的，也就是說在找到 x[i]和 y[j] 相等之後，我們可以跳過剩下的 y[ ]元素。
4. 為什麼? 因為陣列中的元素相異。

粗淺的想法: 5/5

```
int EQUAL_PAIRS(int x[], int y[],
            int m, int n,
            int xx[], int yy[])
{
      int i, j, k;

      k = 0;
      for (i = 0; i < m; i++)
            for (j = 0; j < n; j++)
                  if (x[i] == y[j]) {
                        xx[k] = i;
                        yy[k] = j;
                        k++;
                        break;
                  }
      return k;
}
```

1. 左邊的 EQUAL_PAIRS() 函數是個可能的解答:
   a) int x[ ]和 y[ ]是輸入陣列，各有 m和 n個元素。
   b) xx[ ]和 yy[ ]是輸出陣列，它們分別儲存在x[ ]和 y[ ]中等值對的位置。
   c) k 是函數值，這是等值對的數目。

這個解答好嗎？：1/2

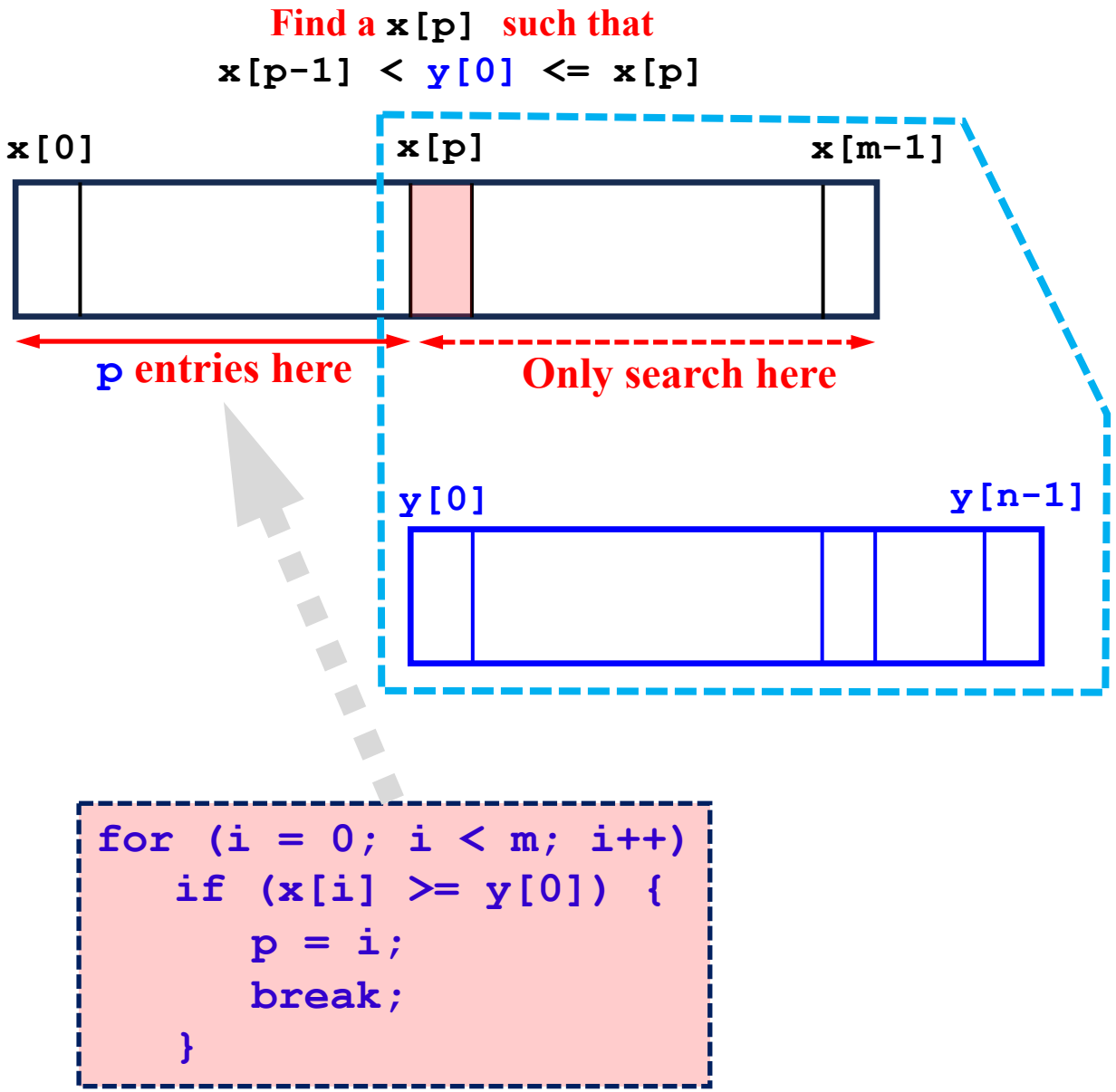
```
int EQUAL_PAIRS(int x[], int y[],
                int m, int n,
                int xx[], int yy[])
{
     int i, j, k;

     k = 0;
     for (i = 0; i < m; i++)
          for (j = 0; j < n; j++)
               if (x[i] == y[j]) {
                    xx[k] = i;
                    yy[k] = j;
                    k++;
                    break;
               }
     return k;
}
```

1. 我們看看比較了多少次。
2. 在最壞的情況下，若x[i]不等於 y[ ]中的任何元素，於是j-迴圈繞n次，所以用了n個比較。
3. 而外面的i-迴圈也繞 m 次，於是總共比了 $O(m \times n)$ 次。
4. 若 $m=n$，這是一個比了 $O(n^2)$ 次的解答。

這個解答好嗎？：2/2

```
int EQUAL_PAIRS(int x[], int y[],
            int m, int n,
            int xx[], int yy[])
{
      int i, j, k;

      k = 0;
      for (i = 0; i < m; i++)
            for (j = 0; j < n; j++)
                  if (x[i] == y[j]) {
                        xx[k] = i;
                        yy[k] = j;
                        k++;
                        break;
                  }
      return k;
}
```

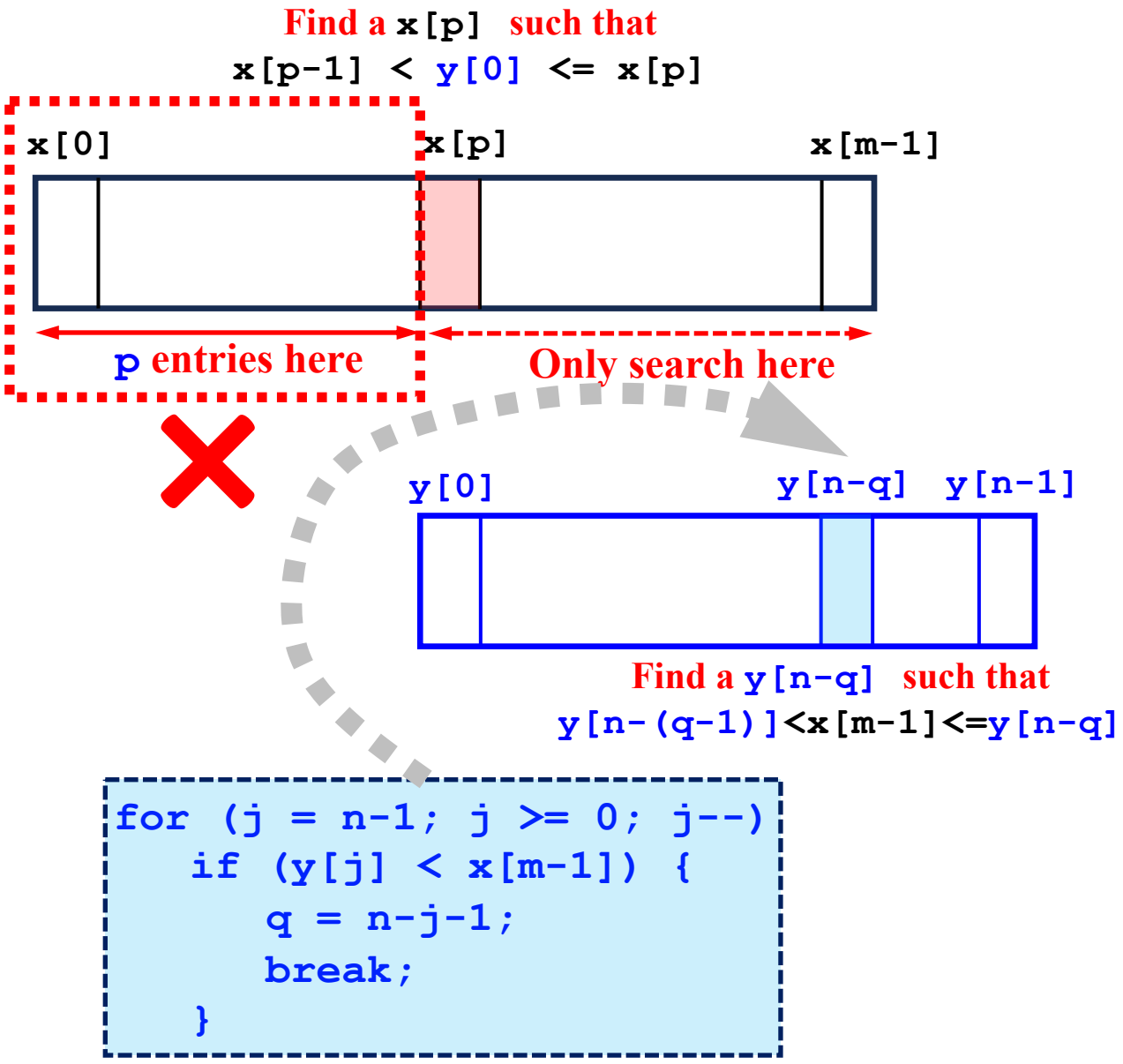
1. 對初學者來說，這或許是個可以接受的解法。
2. 不過，這個解法卻沒有用到所有的已知條件（譬如已經自小到大排好）。
3. 我們很容易把這個解法改得比較好，但可能都是次要的。
4. 不過，以初學者的觀點試著改良也很有意義。



改良? 1/5

1. 我們假設 `x[0] < y[0]`。
2. 找一個 `x[p]` 滿足下式:
   `x[p-1] < y[0] <= x[p]`
3. 於是就不必比較從 `x[0]` 到 `x[p-1]`，因為它們不可能在`y[]`中。
4. 所以， 用`p`次比較，可以把需要的工作降到只比較 `x[p..m-1]` 和 `y[0..n-1]`。
5. 可以做得更多嗎? 可以!

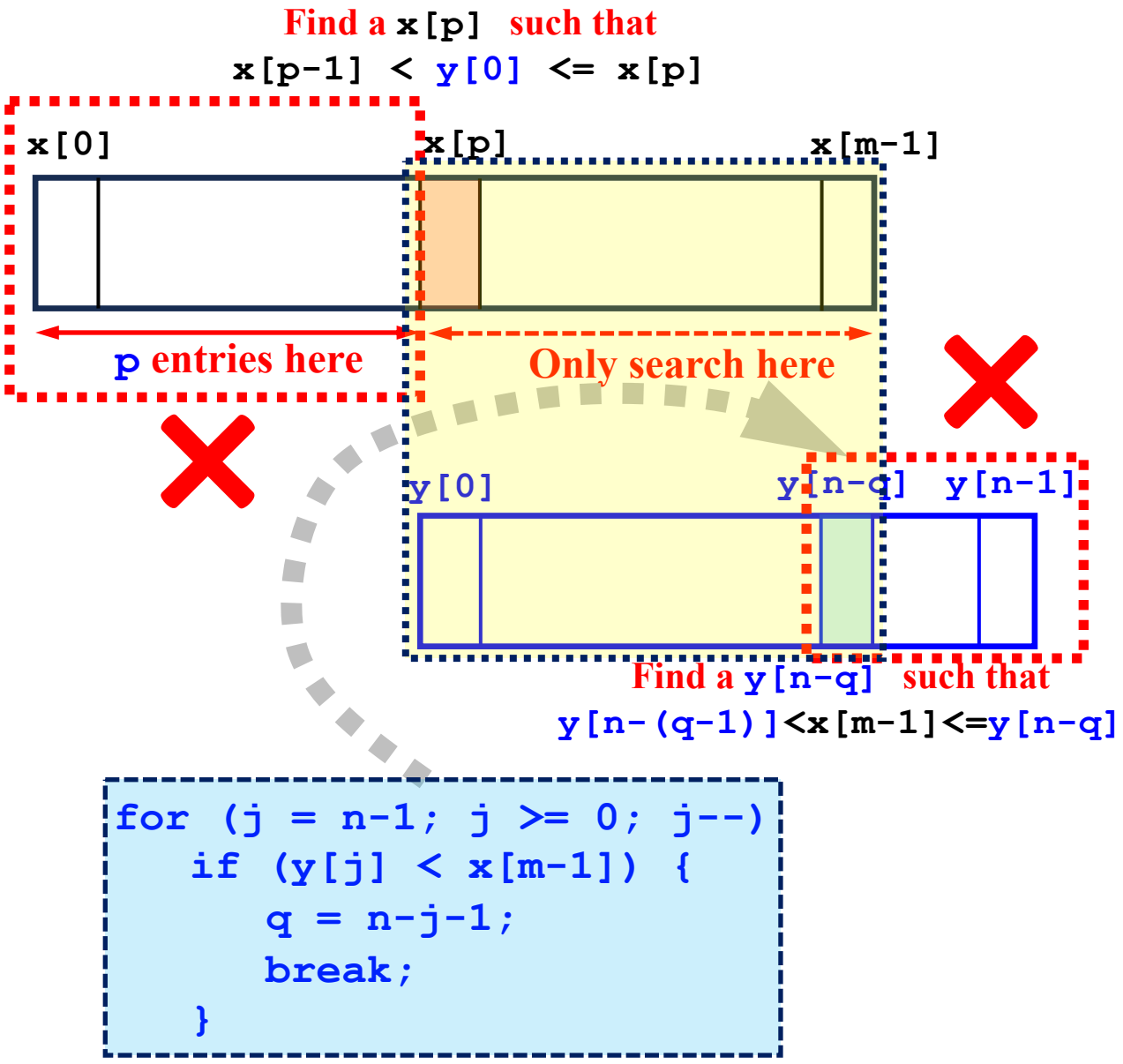
改良? 2/5




```
for (j = n-1; j >= 0; j--)
   if (y[j] < x[m-1]) {
      q = n-j-1;
      break;
   }
```

1. 若 `y[n-1] > x[m-1]`，我們還可以把比較的範圍再縮短些。
2. 找一個 `y[n-q]` 滿足下式

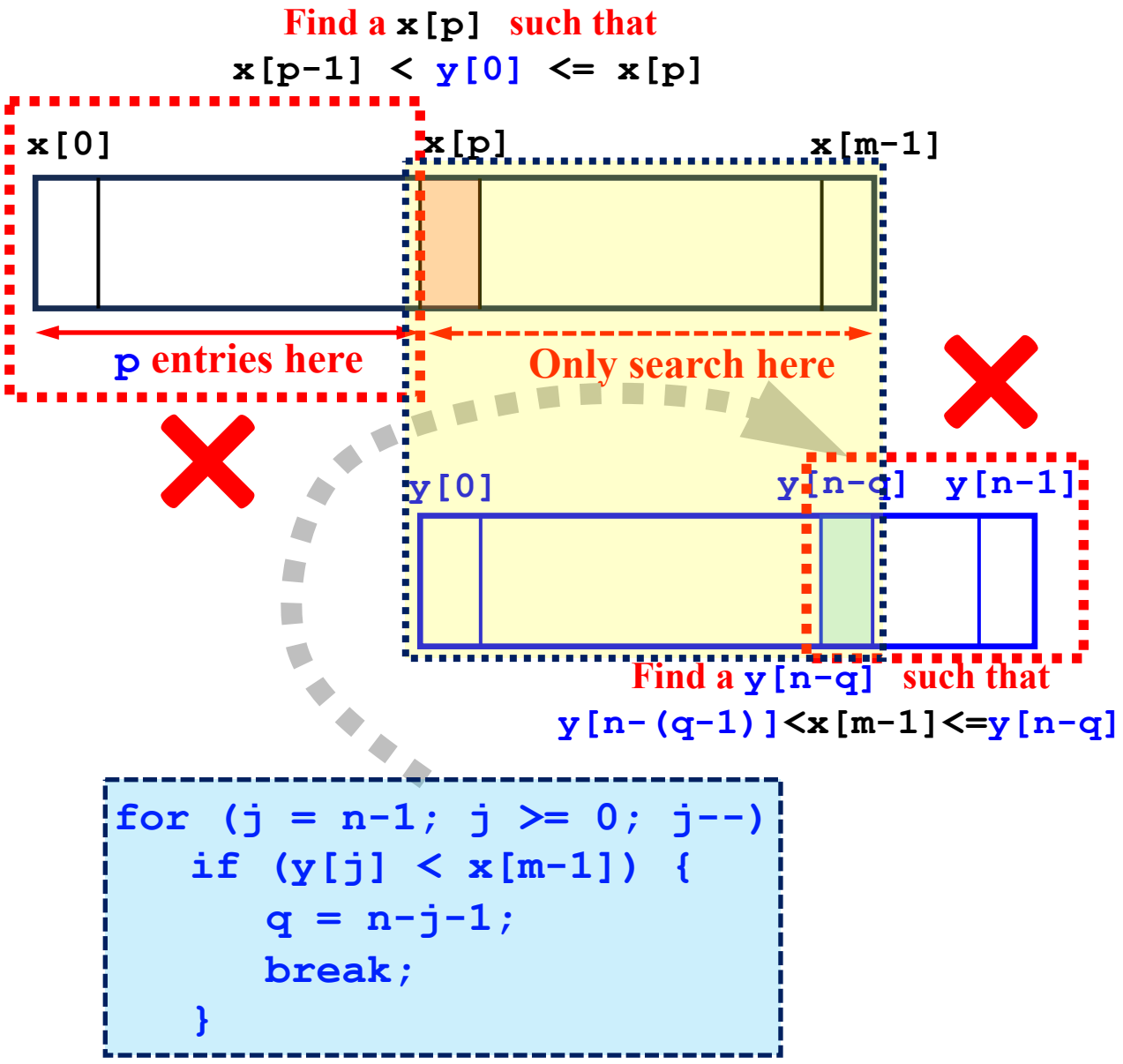
   `y[n-(q-1)] < x[m-1] <= y[n-q]`
3. 這樣，從 `y[n-q]` 到 `y[n-1]` 就不可能在 `x[]` 內，因為它們比每一個 `x[]` 中的值都來得大。
4. 於是，用q次比較把比較的範圍縮小到`x[p..m-1]`和`y[0..n-q]`。




```
for (j = n-1; j >= 0; j--)
   if (y[j] < x[m-1]) {
      q = n-j-1;
      break;
   }
```

改良? 3/5

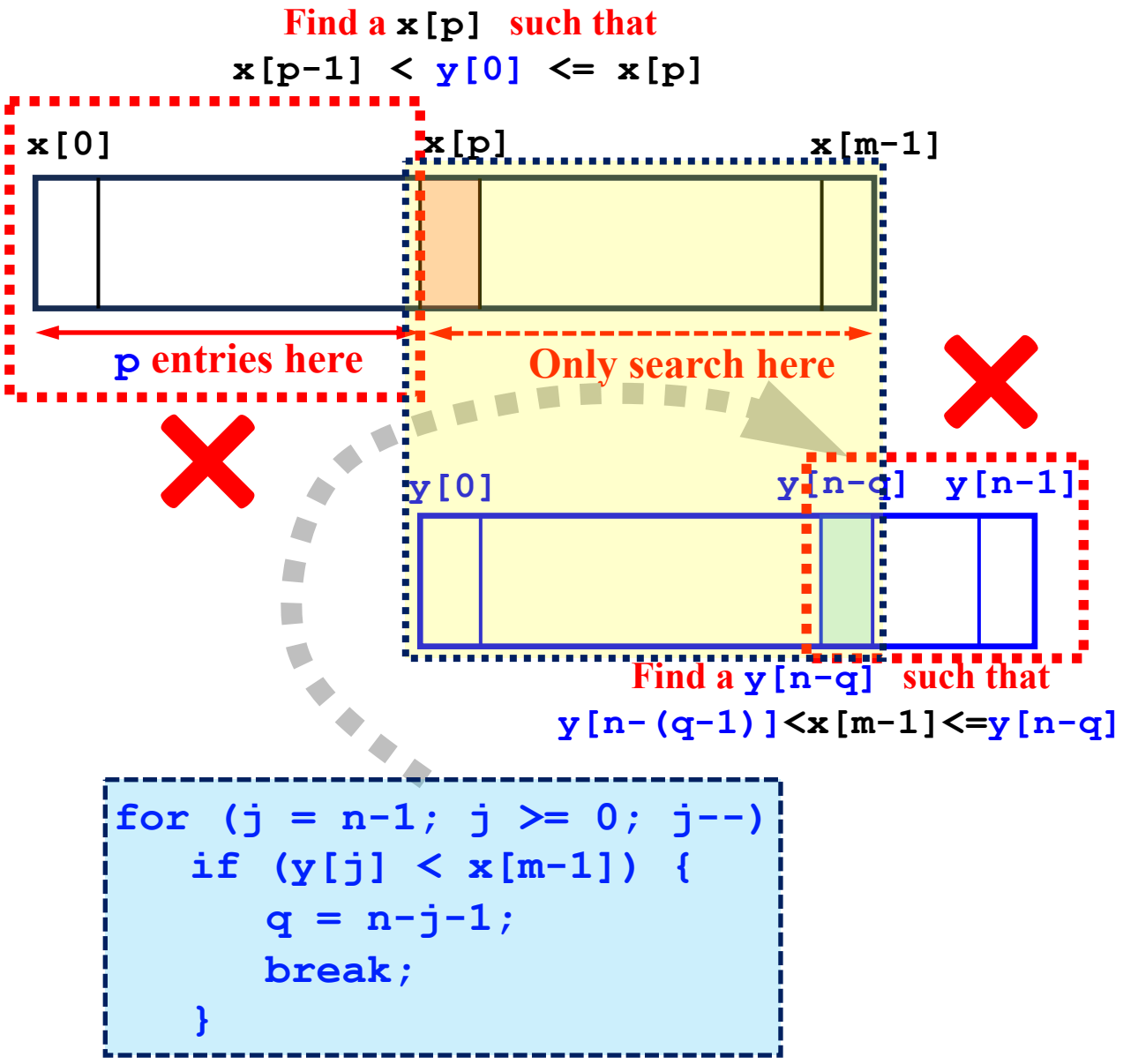
1. 這樣，我們用 `p` 個比較去掉 `x[]`中的頭 `p` 個元素；又用 `q` 個比較去掉 `y[]`中的後 `q`個元素。
2. 結果是： `x[]`中剩下`m-p` 個元素、而 `y[]`中剩下 `n-q`個元素。
3. 因此，每一個剩下來在 `x[]`中的元素 `x[i]` 和`y[]` 中剩下的部分 `y[0..(n-q)]`比較，總共需要個 *n-q* 比較。




```
for (j = n-1; j >= 0; j--)
   if (y[j] < x[m-1]) {
      q = n-j-1;
      break;
   }
```

改良? 4/5

1. 因為x[ ]中剩下 $m-p$ 個元素，總比較次數為 $(m-p)\times(n-q)$ 。
2. 我們得用 p 個比較去掉 x[] 的頭 p 個元素，又用 q 個比較去掉 y[] 中尾部的 q 個元素。
3. 在最壞情況下的比較數是 $(m-p)\times(n-q)+(p+q)$。
4. 所以，最壞的情況$O(m\times n)$沒有變，而且還得做額外的去頭去尾的工作。




```
for (j = n-1; j >= 0; j--)
   if (y[j] < x[m-1]) {
      q = n-j-1;
      break;
   }
```

改良? 5/5

1. 加上一些去頭去尾的額外手續，我們的確可以做到比原來的快一點。
2. 不過，這並沒有對最壞的情況有很大的改善。
3. 所以，我們必須有一點不同的想法才能突破 $O(m \times n)$ 的限制。

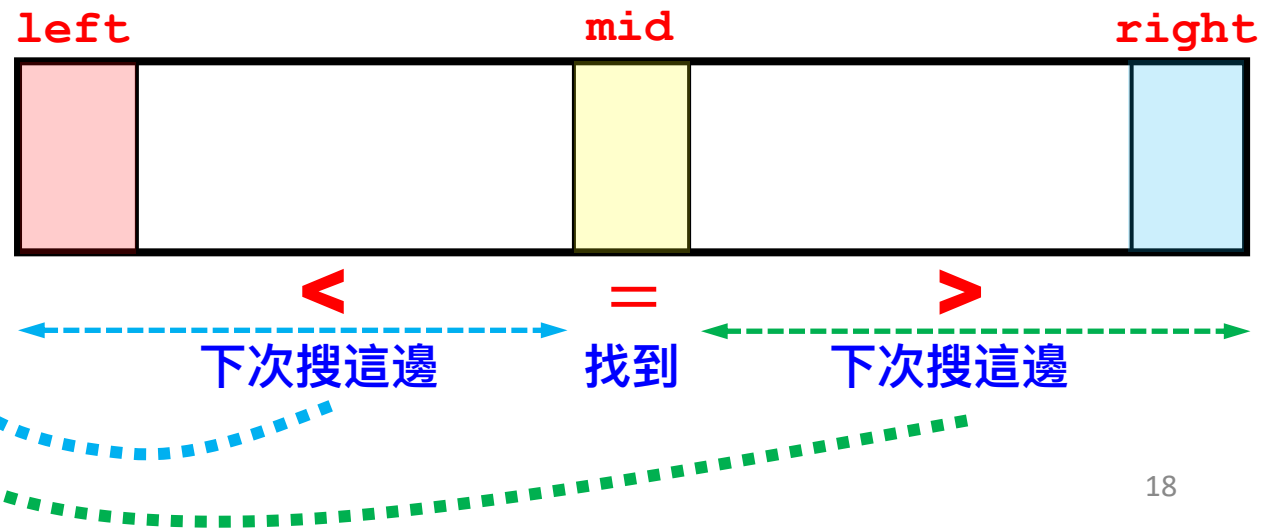
有沒有更好的辦法?

如果您學過資料結構，您可能馬上就會想到：二分搜尋（Binary Search）

基本想法: 1/2

```
left  = 0;
right = n-1;
while (left <= right) {
    mid = (left + right)/2;
    if (DATA == y[mid]) {
        // found at mid
        // return mid
    }
    else if (DATA < y[mid])
        right = mid - 1;
    else
        left  = mid + 1;
    }
}
// return not-found
```

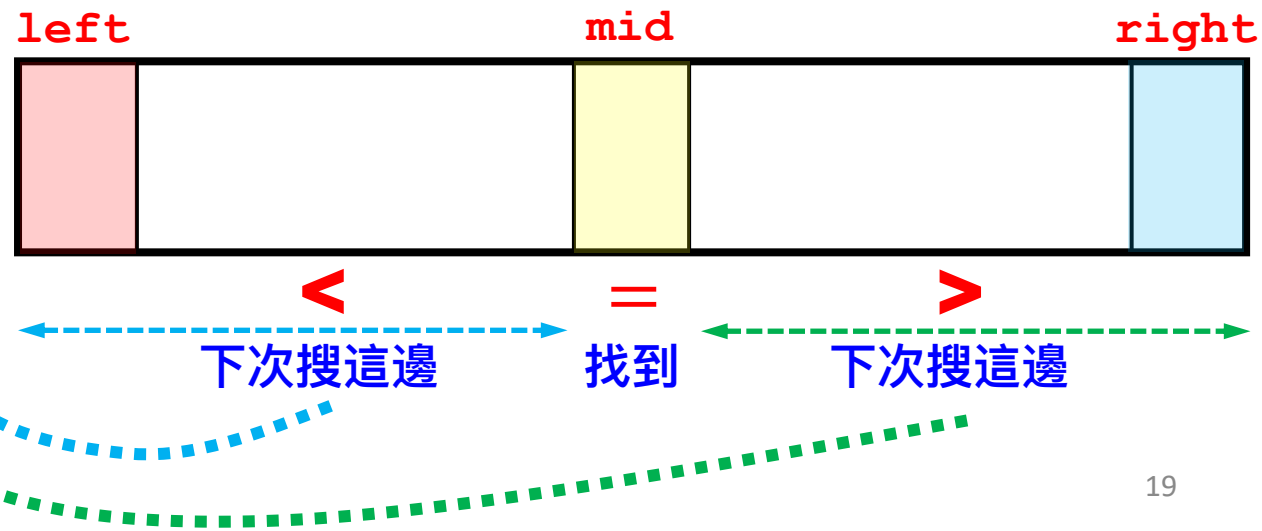
1. 陣列y[ ]中各元素自小到大排好、而且全不相同。
2. 若 DATA 是一個給定的資料，它在陣列 y[ ]中嗎？若在，又在何處?



基本想法: 2/2

```
left  = 0;
right = n-1;
while (left <= right) {
    mid = (left + right)/2;
    if (DATA == y[mid]) {
        // found at mid
        // return mid
    }
    else if (DATA < y[mid])
        right = mid - 1;
    else
        left  = mid + 1;
    }
}
// return not-found
```

1. 每次比較後搜尋的範圍都減半，所以這個迴圈最多繞 $\log_2(n)$ 次就知道有否找到DATA。
2. 所以最壞情況下會用到$2\times\log_2(n)$亦即$O(\log_2(n))$次比較。



```
int EQUAL_PAIRS(int x[], int y[],
      int m, int n, int xx[], int yy[])
{
    int i, j, k = 0;
    int left, right, mid;

    for (i = 0; i < m; i++) {
        left  = 0;   right = n-1;
        while (left <= right) {
            mid = (left + right)/2;
            if (x[i] == y[mid]) {
                xx[k] = i;  yy[k] = mid;
                k++;
                break;
            }
            else if (x[i] < y[mid])
                right = mid - 1;
            else
                left  = mid + 1;
        }
    }
    return k;
}
```

binary search

解答: 1/6

1. 左邊是個可能的解答。
2. 對每一個 `i`，在`y[]`中找`x[i]`。
3. 若找到了 `x[i]`，把位置存入 `xx[]` 和 `yy[]`、並且把等值對數目 (`int k`) 加`1`。
4. 程式中每繞一次得用 2 個比較。
5. 因為每一次搜尋需要 $O(\log_2(n))$ 個比較，而`x[]` 有`m`個元素，總比較數為 $O(m \times \log_2(n))$ 。

```
int EQUAL_PAIRS(int x[], int y[],
      int m, int n, int xx[], int yy[])
{
    int i, j, k = 0;
    int left, right, mid;

    for (i = 0; i < m; i++) {
        left  = 0;   right = n-1;
        while (left <= right) {
            mid = (left + right)/2;
            if (x[i] == y[mid]) {
                xx[k] = i;  yy[k] = mid;
                k++;
                break;
            }
            else if (x[i] < y[mid])
                right = mid - 1;
            else
                left  = mid + 1;
        }
    }
    return k;
}
```

binary search

解答: 2/6

1. 用 `x[i]` 搜尋 `y[]` 的比較數目 $O(m \times \log_2(n))$.
2. 用 `y[j]` 搜尋 `x[]` 的比較數目 $O(n \times \log_2(m))$.
3. 哪種方式比較好?
4. 直覺的看法是：用短的陣列去搜長的陣列比較好，因為$\log_2()$比較小。
6. 有沒有證明呢?

解答: 3/6

x	$\log_2(x)$	四捨五入
1	0	0
10	3.322	4
100	6.644	7
1,000	9.966	10
10,000	13.288	14
100,000	16.610	17
1,000,000	19.932	20
10,000,000	23.253	24

1. 左邊的表提示了x 和對應的 $\log_2(x)$。
2. 很明題地，$\log_2(x)$ 的增長比 x 的增長慢很多。
3. 所以，使用短陣列去搜長陣列比較快。
4. 這是個觀察，而我們需要証明。

解答: 4/6

$$m\log_2(n) \;?\Leftrightarrow\; n\log_2(m)$$

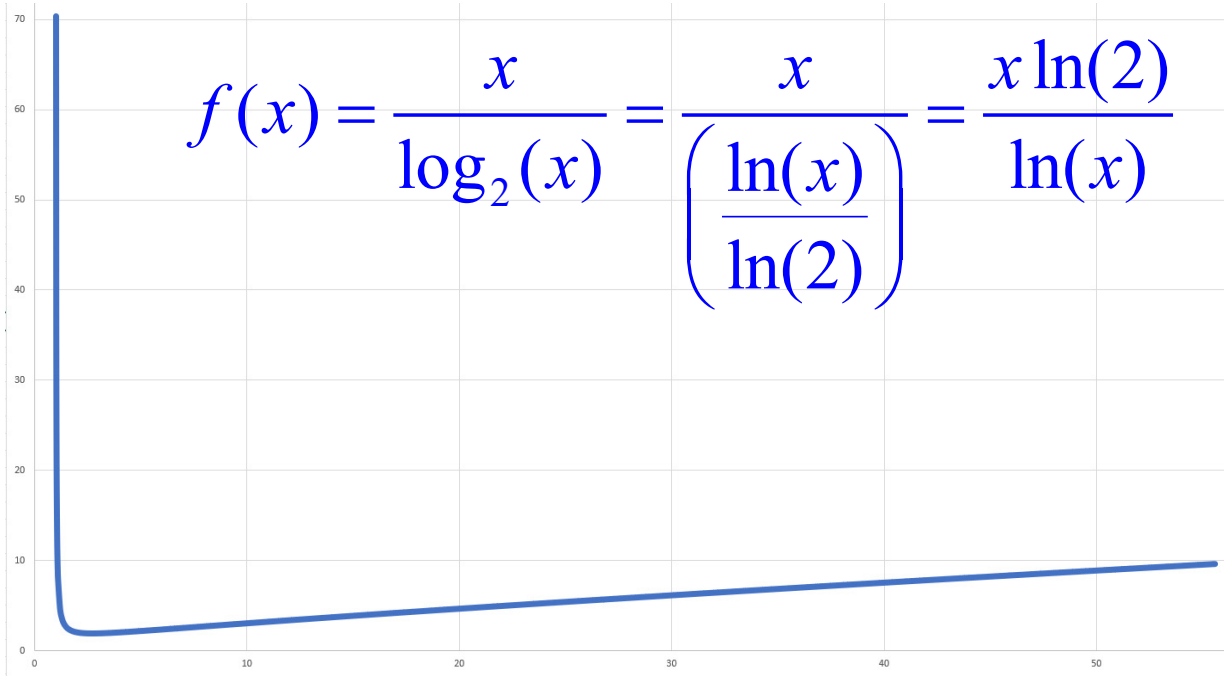
$$\frac{m}{\log_2(m)} \;?\Leftrightarrow\; \frac{n}{\log_2(n)}$$

Let $f(x)=\dfrac{x}{\log_2(x)}$

若 $f(x)$ 是個上升（增加）函數，
$m < n$ 表示 $m/\log_2(m) < n/\log_2(n)$；
因此就相當於$m\log_2(n) < n\log_2(m)$。

1. 假設$m < n$，我們要證明用短陣列去搜長陣列的比較數目 ($m\log_2(n)$)要比用長陣列搜短陣列的比較數目 ($n\log_2(m)$) 來得少。
2. 兩邊同除 $\log_2(n)\times\log_2(m)$.
3. 我們得到一個函數：
   $f(x)=x/\log_2(x)$.

解答: 5/6



1. 若 x=1，$\log_2(1)=0$ 而且 $f(x) = \infty$ 。
2. 計算 $f(x)$ 的導數如下：

$$\frac{df}{dx} = \ln(2)\frac{d}{dx}\left(\frac{x}{\ln(x)}\right)$$

$$= \ln(2)\frac{\ln(x)\frac{dx}{dx} - x\frac{d(\ln(x))}{dx}}{\left(\ln(x)\right)^2}$$

$$= \ln(2)\frac{\ln(x)-1}{\left(\ln(x)\right)^2}$$

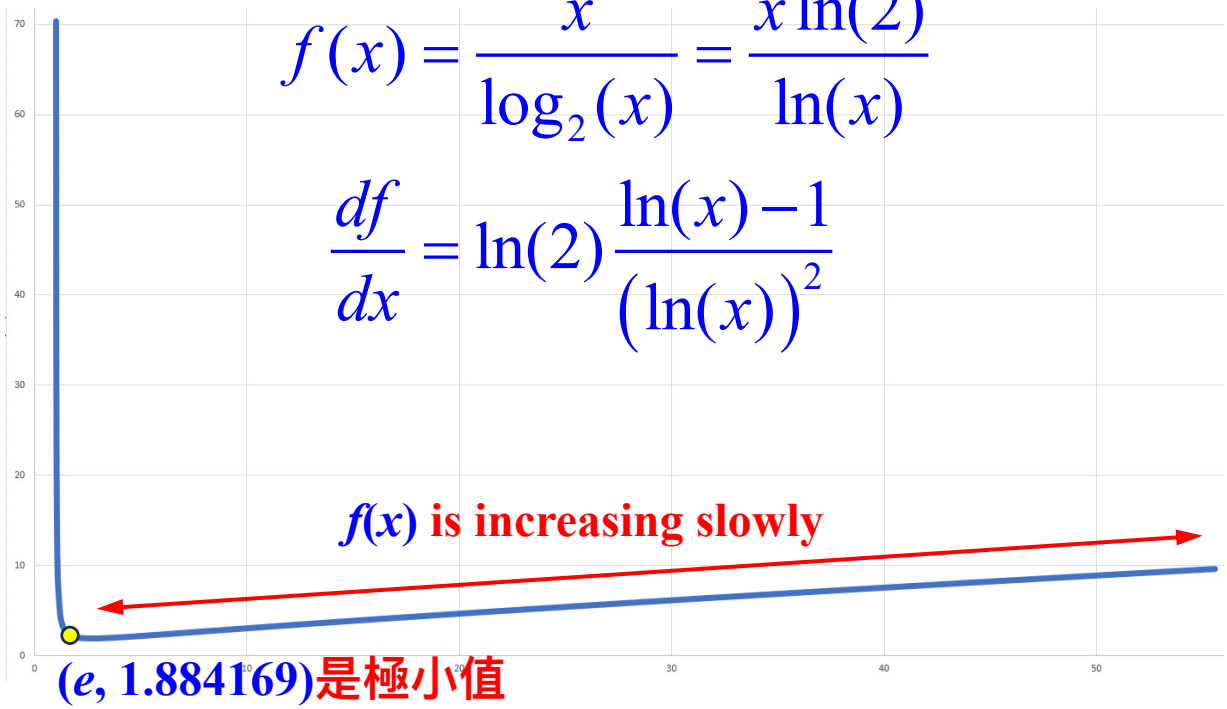
$$\frac{d(\ln(x))}{dx} = \frac{1}{x}$$

若 $f(x)$ 是個上升（增加）函數，
$m < n$ 表示 $m/\log_2(m) < n/\log_2(n)$；
因此就相當於$m\log_2(n) < n\log_2(m)$。

解答: 6/6

$$f(x)=\frac{x}{\log_2(x)}=\frac{x\ln(2)}{\ln(x)}$$

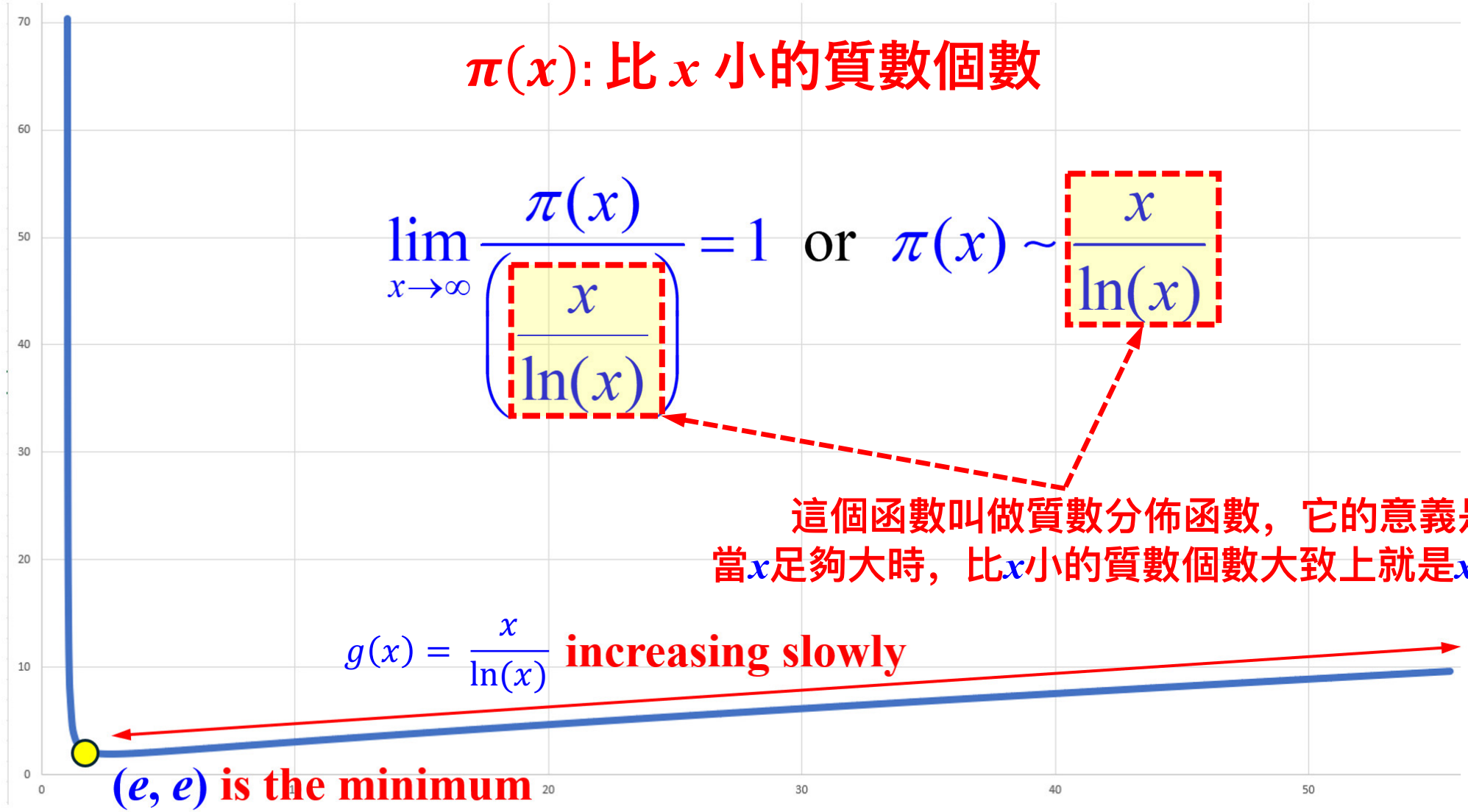
$$\frac{df}{dx}=\ln(2)\frac{\ln(x)-1}{(\ln(x))^2}$$



1. $f(x)$ 的極小值在 $x=e$，因為 $f'(e)=0$ 。
2. $f(e)=e/\log_2(e)=1.884169$.
3. $(e, 1.884169)$ 是極小值所在。
4. 若 $x>e$， $f(x)$的值是上升！
5. 所以，只要 m 和 n 大於等於 2，用短陣列去搜長陣列會用較少的比較次數。

若 $f(x)$ 是個上升（增加）函數，
$m<n$ 表示 $m/\log_2(m)<n/\log_2(n)$；
因此就相當於$m\log_2(n)<n\log_2(m)$。

題外話：質數分佈函數


π(x): 比 x 小的質數個數
lim x→∞ π(x) / (x/ln(x)) = 1 or π(x) ~ x/ln(x)
這個函數叫做質數分佈函數，它的意義是：
當x足夠大時，比x小的質數個數大致上就是x/ln(x) 。
g(x) = x/ln(x) increasing slowly
(e, e) is the minimum
0
10
20
30
40
50
60
70
0
20
30
40
50

改良: 1/3

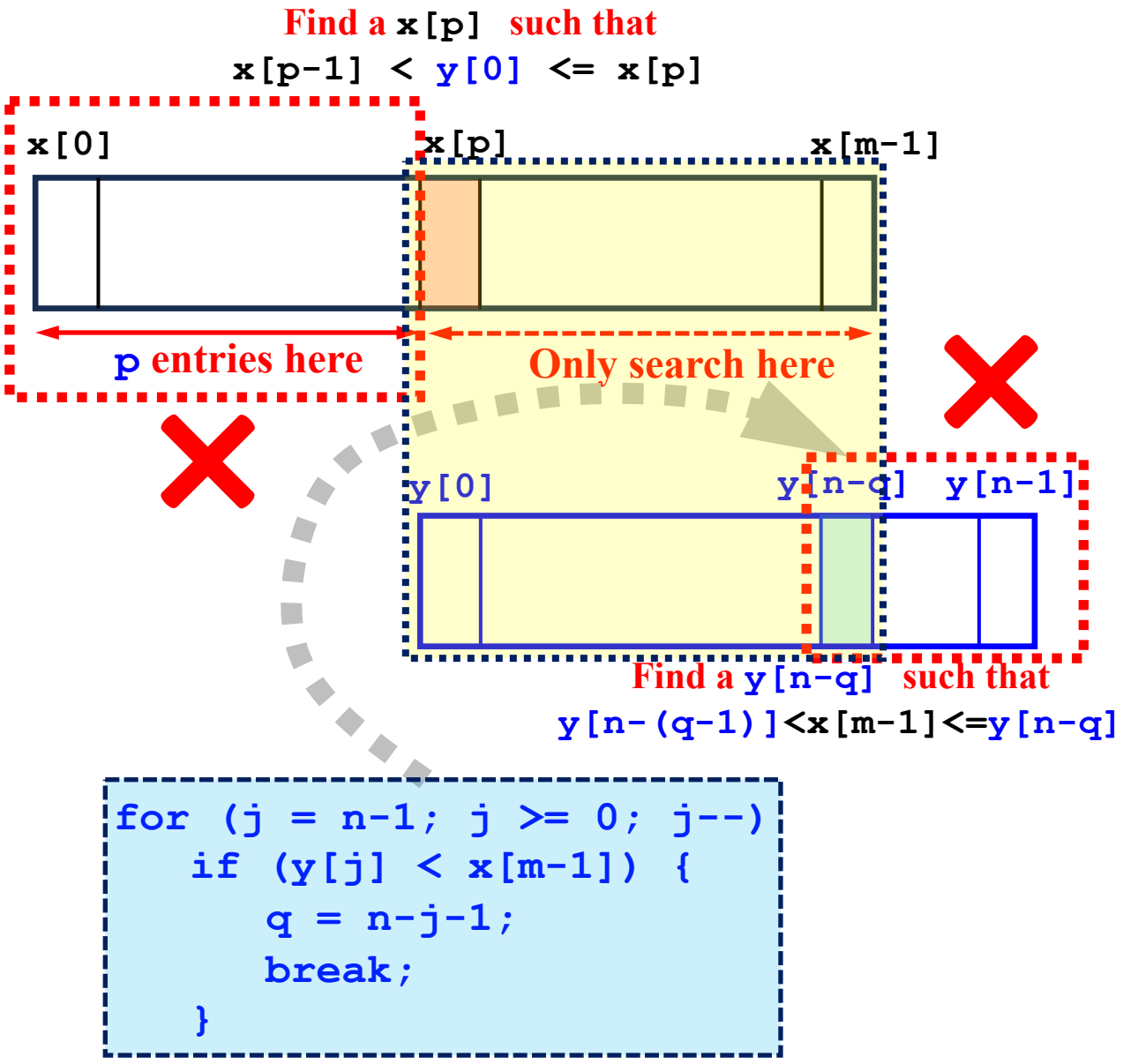
```
int EQUAL_PAIRS(int x[], int y[],
      int m, int n, int xx[], int yy[])
{
    int i, j, k = 0;
    int left, right, mid;

    for (i = 0; i < m; i++) {
       if (y[0] <= x[i] &&
           x[i] <= y[n-1]) {
          //
          // do a binary search
          //
    }
```

binary search

1. 一個很顯而易見的改良就是：若 `x[i]` 不在 `y[0]` 和 `y[n-1]` 之間就不搜尋！
2. 理由就是：若`x[i]`不在`y[]`的範圍內，就找不到它！搜尋就徒勞無功。
3. 縱使如此，也不會改變最壞的情況。考慮以下兩個陣列：

   `x[]` = { 1, 3, 5, 7, 9}

   `y[]` = { 0, 2, 4, 6, 8, 10, 12}.

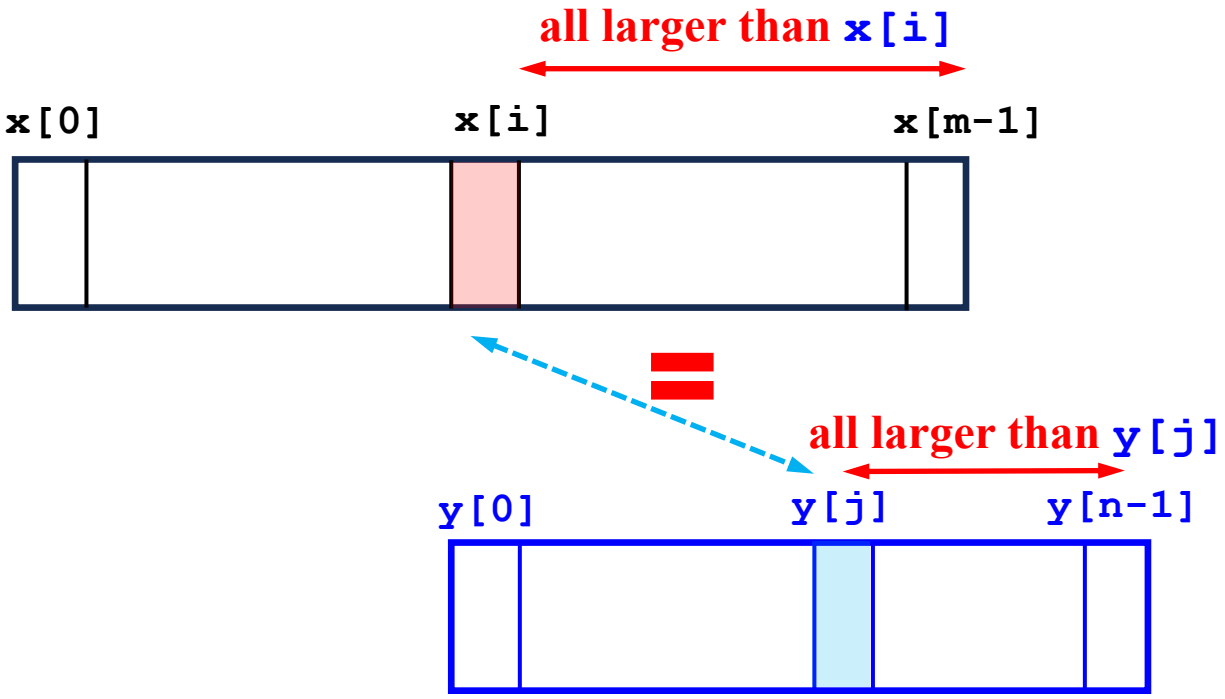



```
for (j = n-1; j >= 0; j--)
   if (y[j] < x[m-1]) {
      q = n-j-1;
      break;
   }
```

改良: 2/3

1. 和上一個方法一樣，我們也可以把 `x[]` 和 `y[]` 去頭去尾、期望再降低些比較數目。
2. 不過，這樣做對最壞的情況不會有什麼幫助，因為 p 和 q 可能都是 0!

改良: 3/3

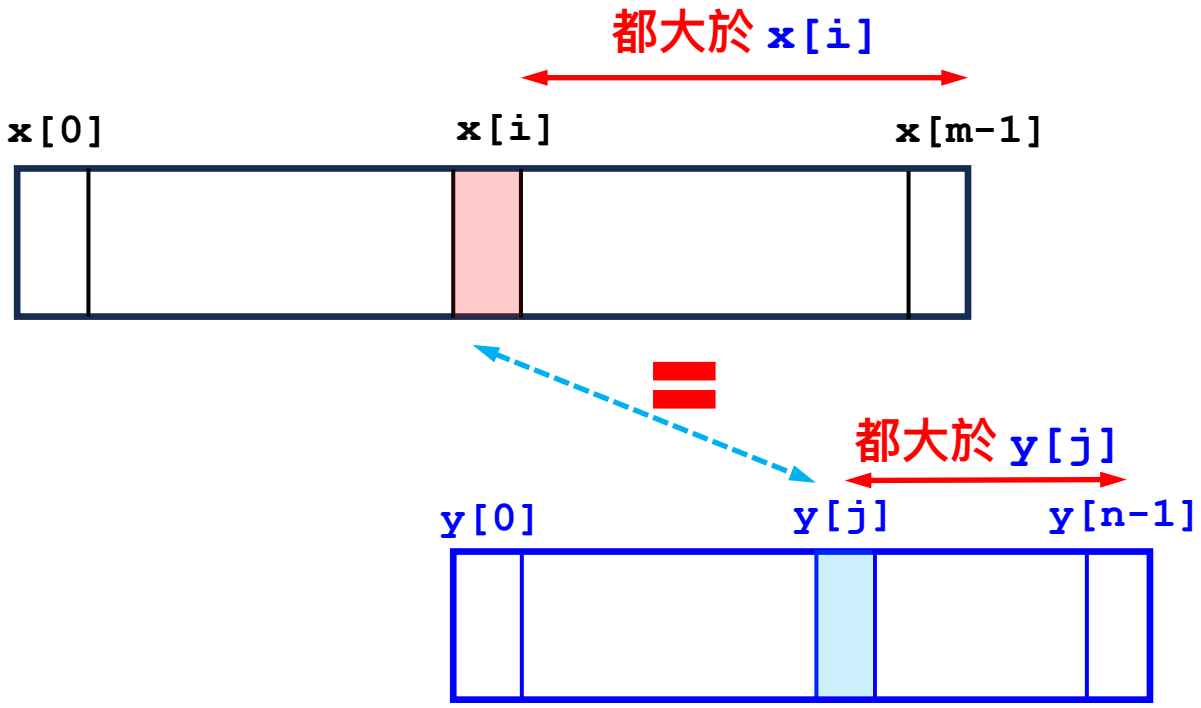


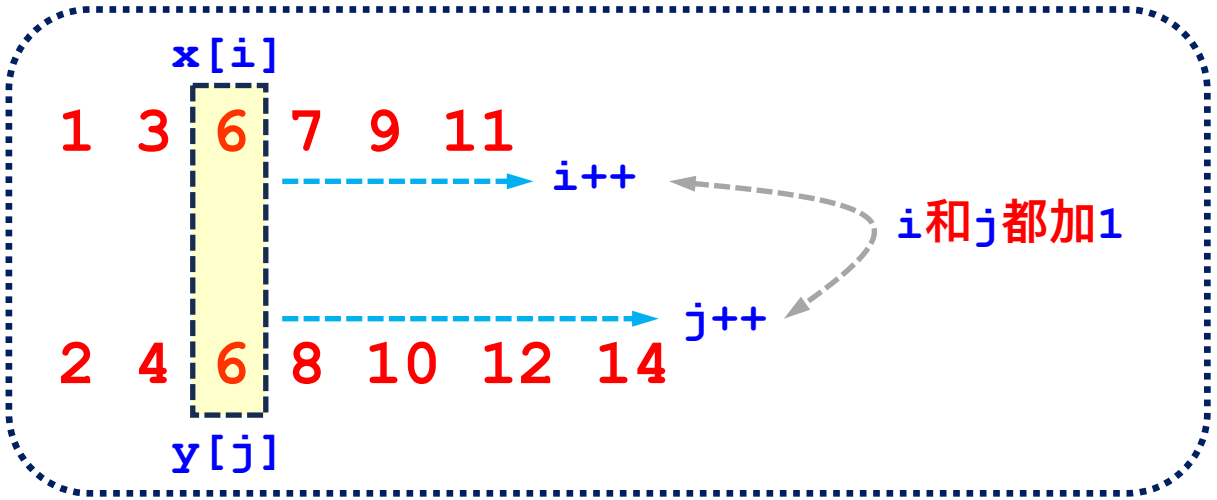
1. 若 x[i]和 y[j]相等，於是 x[i+1..m-1]這一段就不會和 y[0..j]這一段中任何元素相同。
2. 原因就是 x[]和 y[]都是自小到大排好，而且值都不相同。
3. 因此，若 x[i] = y[j]，x[i+1] 的搜尋範圍就是從 y[j+1]到 y[n-1]。
4. 然而，這樣做仍然無法改善最壞的情況（若完全沒有等值對）
5. 不過，這卻是很重要、到達最佳解的觀察。

一個最佳解

到目前為止，我們已經有了達到這個目的的所有工具

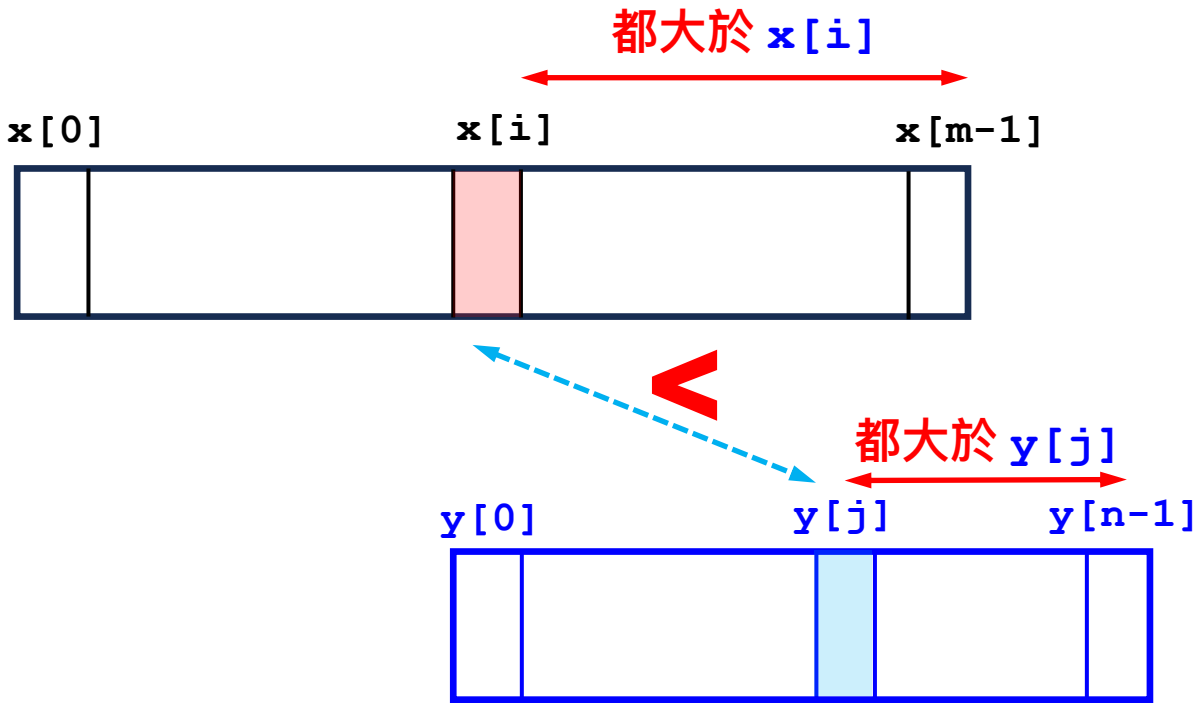
想法: 1/5



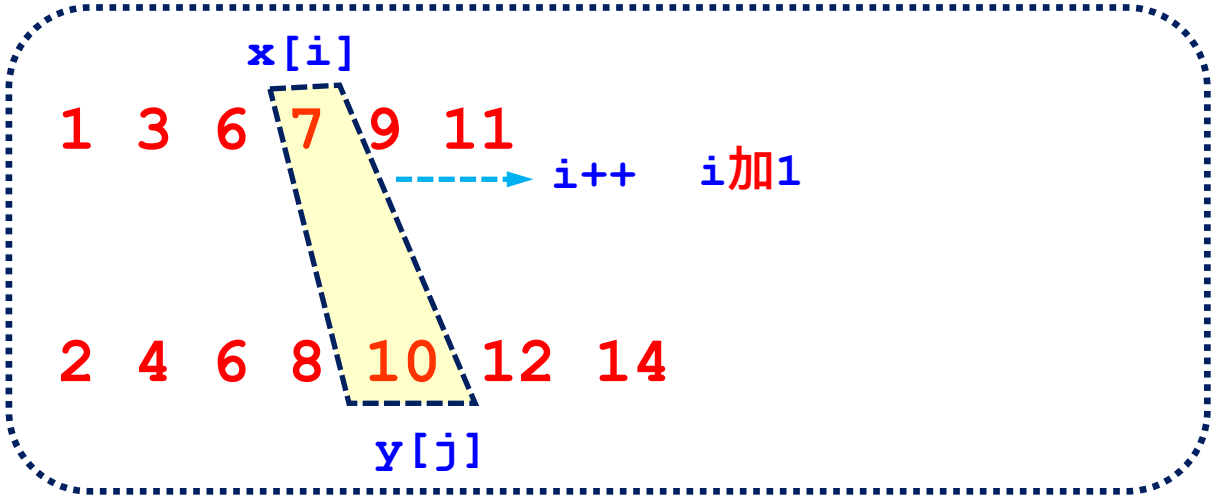


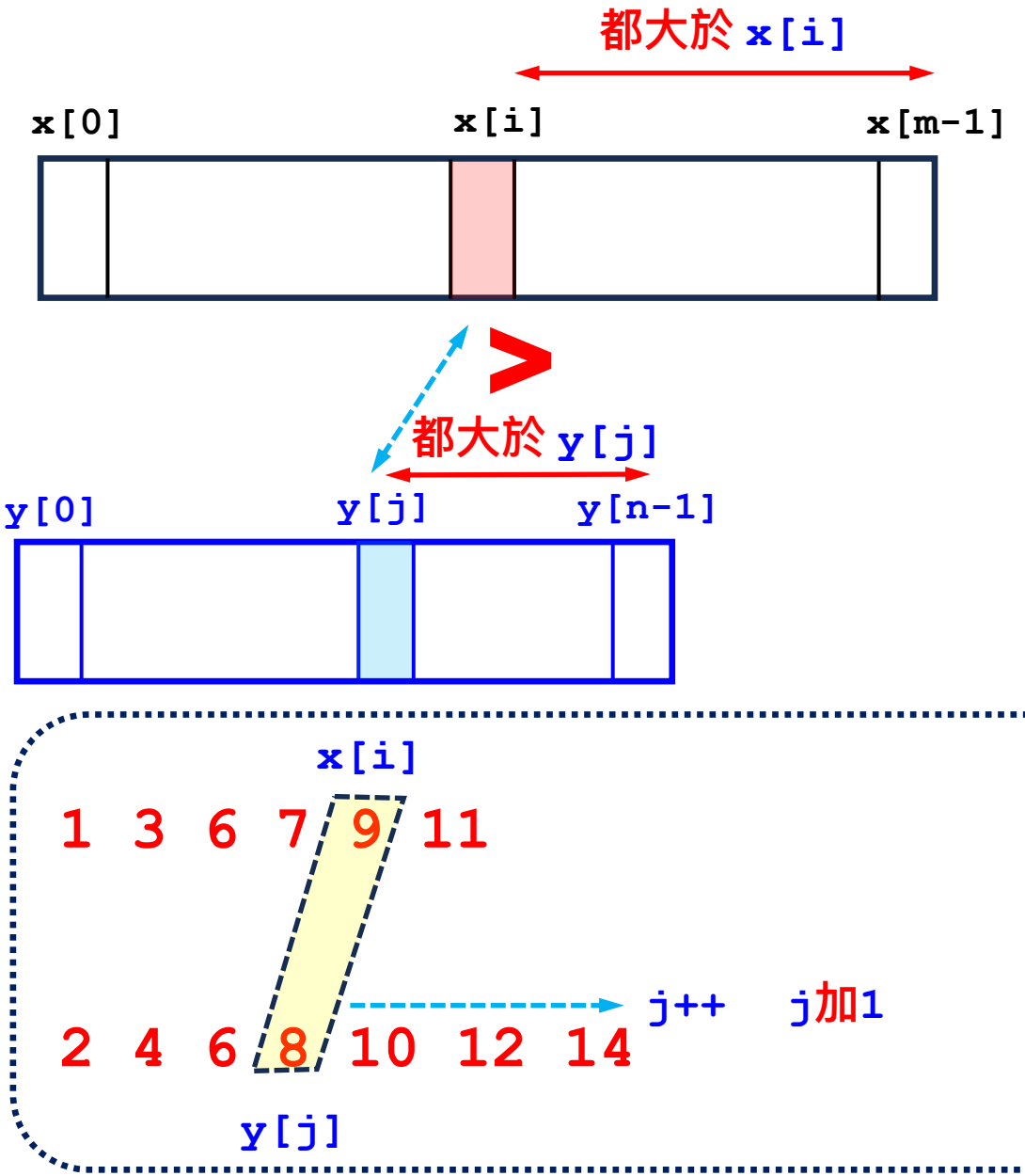
1. 若 x[i] = y[j]，所有在 x[i+1..m-1] 裡頭的元素都不可能在 y[0..j] 中出現。
2. 同樣地， y[j+1..n-1] 中的元素也不可能在x[0..i] 中出現。
3. 原因： x[] 和 y[] 都是自小到大排好、而且值都不相同。
4. 所以，若x[i] = y[j]，搜尋的範圍就限制在 x[i+1..m-1] 和 y[j+1..n-1]。

想法: 2/5



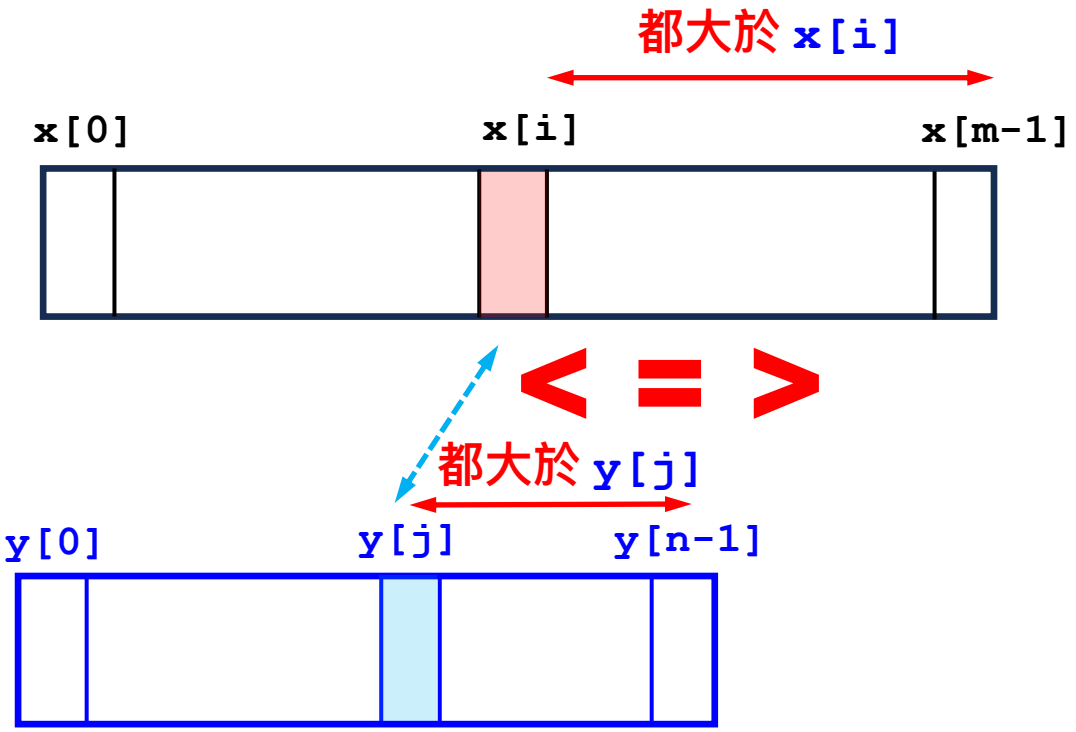
1. 若 `x[i] < y[j]`，下一步為何?
2. 當然，`x[i]`不可能在 `y[j..n-1]`裡頭。
3. 所以，我們放棄`x[i]`而改用`x[i+1]`。
4. 換言之，若`x[i] < y[j]`，我們把`i` 移到下一個`x[i+1]`。





想法: 3/5

1. 若 `x[i] > y[j]`，下一步為何?
2. 當然，`y[j]`不可能在 `x[i..m-1]`裡頭。
3. 所以，我們放棄 `y[j]`而改用 `y[j+1]`。
4. 換言之，若 `x[i] > y[j]`，我們把 `j` 移到下一個元素。



想法: 4/5

1. 我們有三種情況:
   a) 若x[i] < y[j], 執行 i++
      把 x[i] 移到下一個位置
   b) 若x[i] > y[j], 執行 j++
      把 y[j]移到下一個位置
   c) 若 x[i] = y[j], 執行 i++
      和 j++ 把x[i]和y[j]兩者
      都移到下一個位置
2. 這樣, 每繞一次最多用兩次
   比較

想法: 5/5

```
i = j = 0;
while (i < m && j < n) {
    //
    // do the comparisons
    //
}
```

1. i和 j 都得從 0 開始!
2. 只要 x[i] 和 y[j] 中還有未比過的，這個過程就一直反覆。
3. 所以，迴圈結構如左:

```
int EQUAL_PAIRS(int x[], int y[],
          int m, int n, int xx[], int yy[])
{
      int i, j, k;

      i = j = k = 0;
      while (i < m && j < n) {
            if (x[i] < y[j])
                  i++;
            else if (x[i] > y[j])
                  j++;
            else {
                  xx[k] = i;
                  yy[k] = j;
                  i++;
                  j++;
                  k++;
            }
      }
      return k;
}
```

解答

1. 我們有三種情況:
   a) 若`x[i] < y[j]`, 執行 `i++`
      把 `x[i]` 移到下一個位置
   b) 若`x[i] > y[j]`, 執行 `j++`
      把 `y[j]`移到下一個位置
   c) 若 `x[i] = y[j]`, 執行 `i++`
      和 `j++` 把`x[i]`和`y[j]`兩者
      都移到下一個位置

```
int EQUAL_PAIRS(int x[], int y[],
          int m, int n, int xx[], int yy[])
{
      int i, j, k;

      i = j = k = 0;
      while (i < m && j < n) {
            if (x[i] < y[j])
                  i++;
            else if (x[i] > y[j])
                  j++;
            else {
                  i++;
                  j++;
            }
      }
      return k;
}
```

分析: 1/5

1. 用了多少次比較?
2. 若`x[i] < y[j]`，於是 `i++`
   用了 1 次比較
1. 若 `x[i] > y[j]`，於是 `j++`
   用了 2 次比較
1. 若`x[i] = y[j]`，於是 `i++` 和 `j++` 用了 2 次比較

```
int EQUAL_PAIRS(int x[], int y[],
          int m, int n, int xx[], int yy[])
{
      int i, j, k;

      i = j = k = 0;
      while (i < m && j < n) {
            if (x[i] < y[j])
                  i++;
            else if (x[i] > y[j])
                  j++;
            else {
                  i++;
                  j++;
            }
      }
      return k;
}
```

分析: 2/5

1. 用了多少次比較?
2. 若 i 移了 m 次而 j 移了 n 次，總比較次數為 $m+2n$!
3. 這事實上可能嗎?
4. 當然可能!
5. 若x[m-1] 和 y[n-1] 相等，於是 “i移了 m 次而且j 移了 n 次”。
6. 於是總共有 $m+2n$ 次比較，也就是個 $O(m+2n)$ 的解答。

```
int EQUAL_PAIRS(int x[], int y[],
          int m, int n, int xx[], int yy[])
{
      int i, j, k;

      i = j = k = 0;
      while (i < m && j < n) {
            if (x[i] < y[j])
                  i++;
            else if (x[i] > y[j])
                  j++;
            else {
                  i++;
                  j++;
            }
      }
      return k;
}
```

分析: 3/5

1. 假設 $m < n$.
2. 若對每一個 `i`而言都有 `x[i] = y[i]` ，於是每次 `j++` 時都要 **2** 次比較，總比較次數為 $2m$。

```
int EQUAL_PAIRS(int x[], int y[],
          int m, int n, int xx[], int yy[])
{
      int i, j, k;

      i = j = k = 0;
      while (i < m && j < n) {
            if (x[i] < y[j])
                  i++;
            else if (x[i] > y[j])
                  j++;
            else {
                  i++;
                  j++;
            }
      }
      return k;
}
```

分析: 4/5

1. 假設 $m < n$.
2. 若 `x[m-1] < y[0]`，每一次比較都是 “<”，只需要 m 次比較。
3. 另一方面，若`x[0] > y[n-1]`，則每一次比較都是 “>”，總比較數是 $2n$。

```
int EQUAL_PAIRS(int x[], int y[],
          int m, int n, int xx[], int yy[])
{
      int i, j, k;

      i = j = k = 0;
      while (i < m && j < n) {
            if (x[i] < y[j])
                  i++;
            else if (x[i] > y[j])
                  j++;
            else {
                  i++;
                  j++;
            }
      }
      return k;
}
```

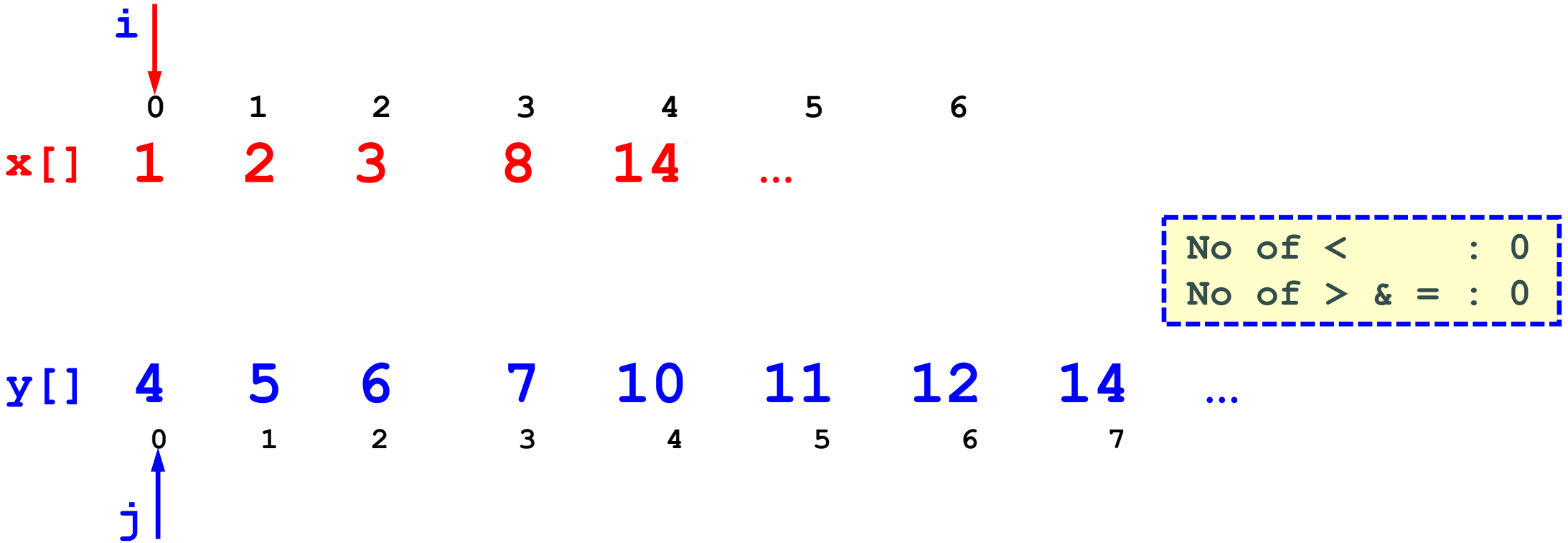
分析: 5/5

1. 假設 $m < n$.
2. 在叫用`EQUAL_PAIRS()`時，給`x[]`長的陣列、給`y[]`短的陣列，可以使 $m+2n$比較小。

範例: 1/5

```
< Comparison steps   : 0
> Comparison steps   : 0
♦ Previous total     : 0
♦ Total comparisons  : 0
```

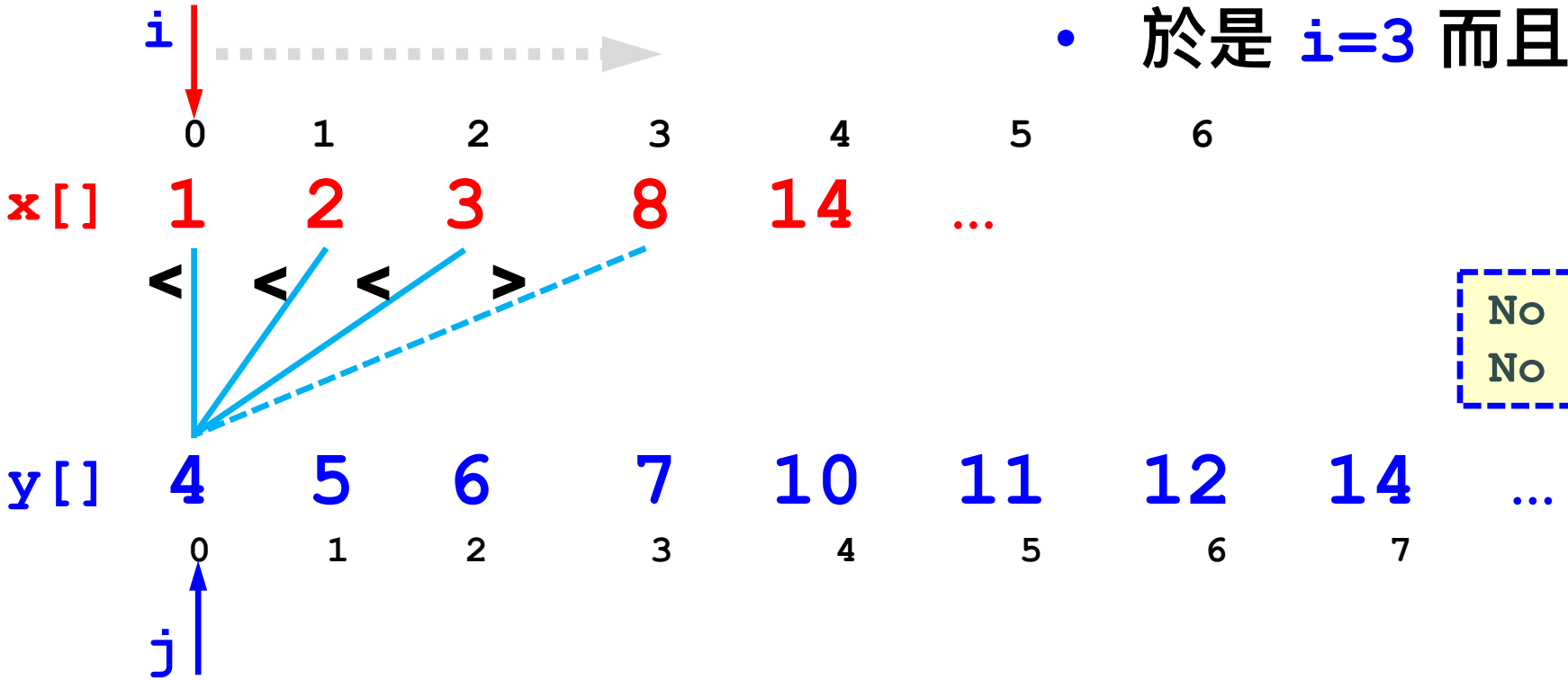
- 開始時，i和 j 為 0。



範例: 2/5

```
< Comparison steps   : 3
> Comparison steps   : 1
♦ Previous total     : 0
♦ Total comparisons  : 3+1×2 = 5
```

- 因為 x[0]<y[0]，執行 i++
- 直到滿足 x[3]>y[0] 為止。
- 我們有3次 < 和 1 次 >。
- 於是 i=3 而且j 移到 1 。

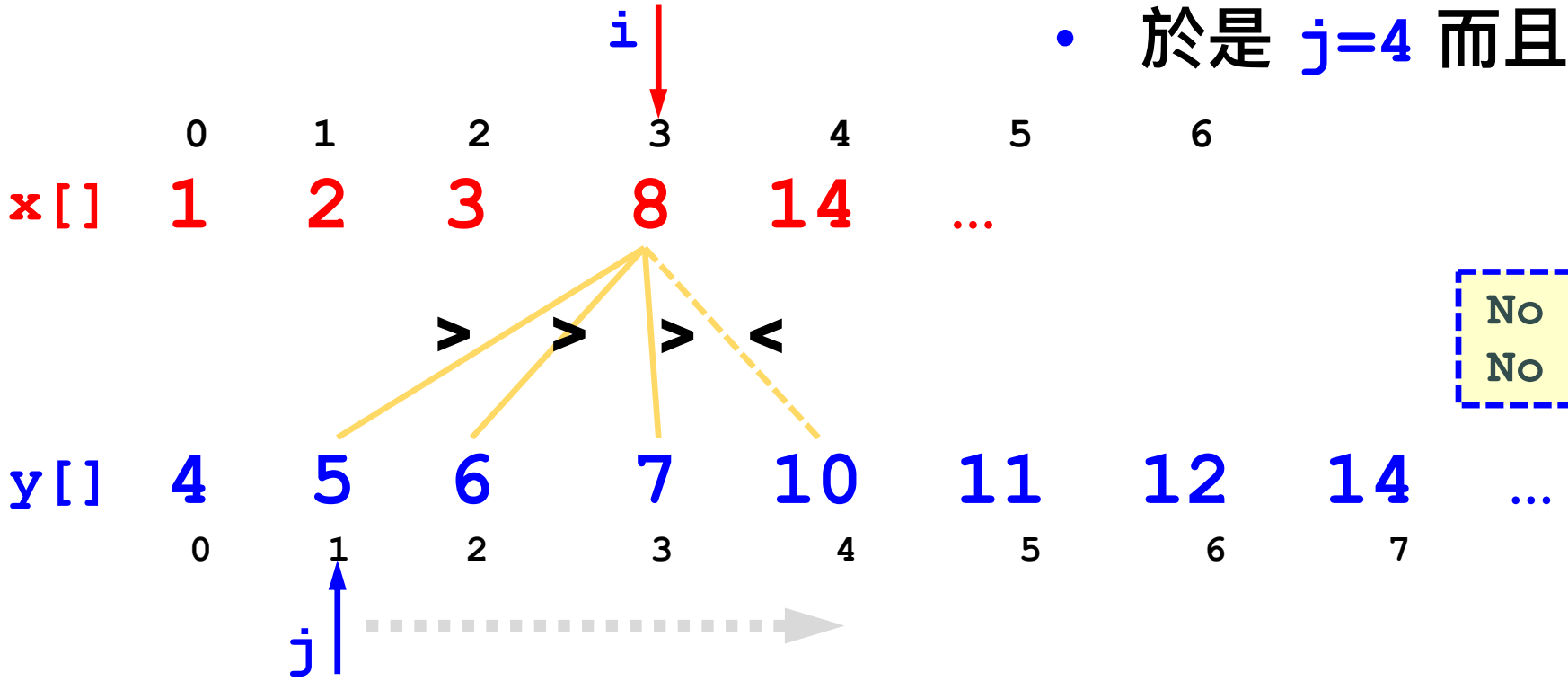



```
No of <      : 3
No of > & = : 1
```

範例: 3/5

```
< Comparison steps   : 1
> Comparison steps   : 3
♦ Previous total     : 5
♦ Total comparisons  : 5+(1+3×2) = 12
```

- 因為 x[3]>y[1], 執行 j++ 直到滿足 x[3]<y[4] 為止。
- 我們有3次 > 和 1 次 <。
- 於是 j=4 而且i 移到 4 。

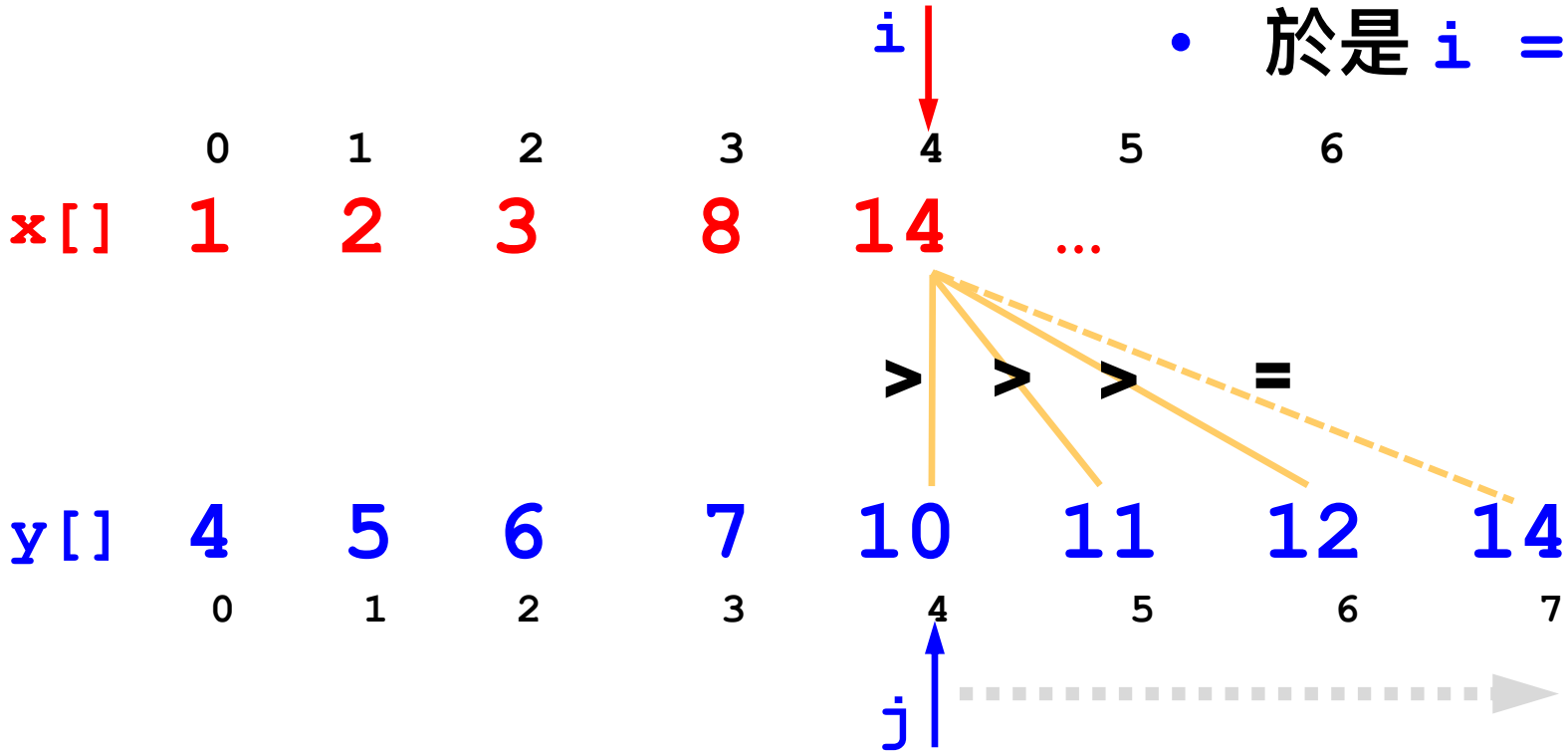



```
No of <      : 1
No of > & = : 3
```

範例: 4/5

```
< Comparison steps   : 1
> Comparison steps   : 1
♦ Previous total     : 12
♦ Total comparisons  : 12+(0+4×2) = 20
```

- 因為 x[4]>y[4], 執行 j++ 直到滿足 x[4] = y[7] 為止。
- 我們有3次 > 和 1 次 =。
- 於是 i = 4而且 j = 7.

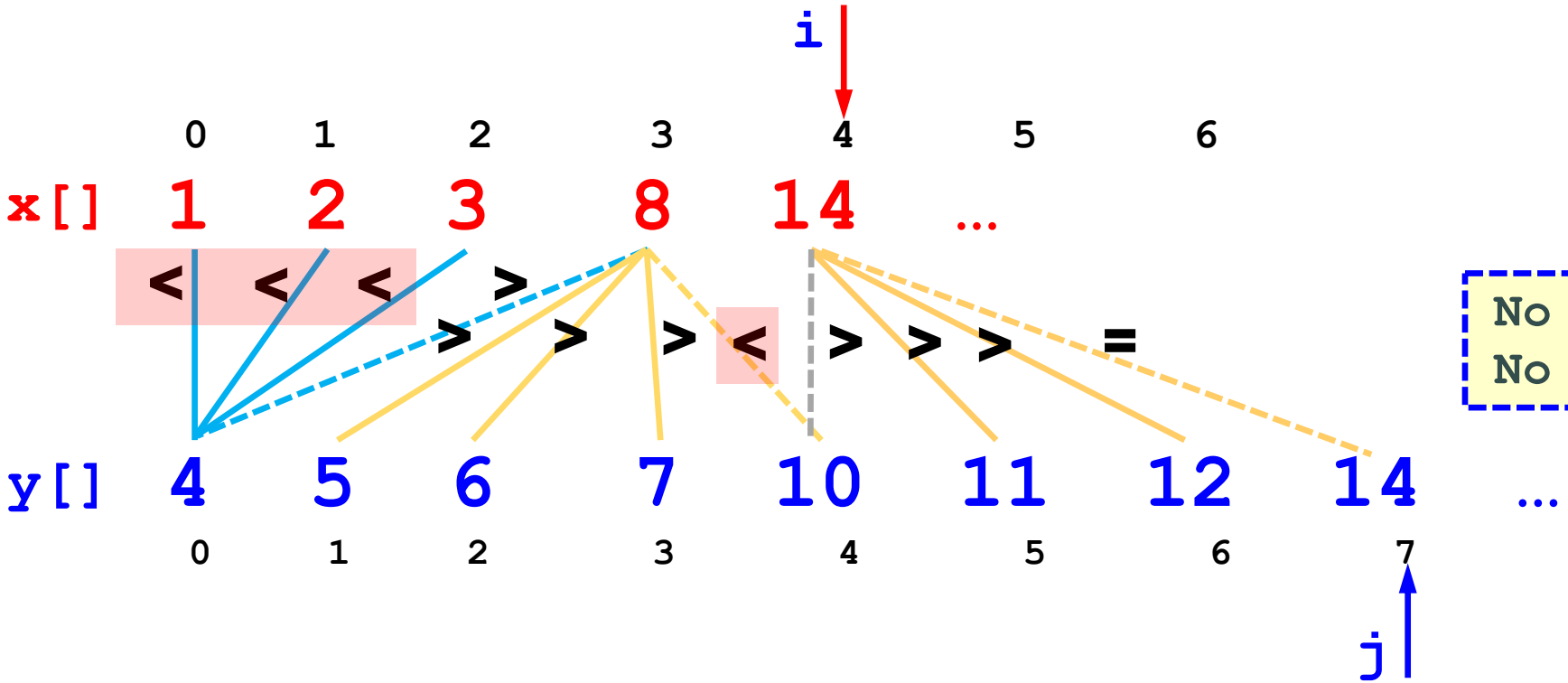



```
No of <      : 0
No of > & = : 4
```

範例: 5/5

♦ Total comparisons : 20

- 總比較次數為 $20 = 4(<)+2\times8(>)$。
- 這正好等於 $m+2n = 4+2\times8$ 。
- 下一步是 i++ 和 j++。



No of < : 4
No of > & = : 8

改良?

- 當下式為真時，很容易可以降低比較次數:

  `x[0] > y[n-1]` 或 `x[m-1] < y[0]`.

- 前面討論過的技巧也有可能降低比較次數。
- 不過以最壞情況而言，功效是有限的。
- 若使用 p次和 q 次比較分別刪去 `x[]` 的前段一部份和 `y[]` 後段一部份，總比較次數為

  $$(p+q)+[(m-p)+2(n-q)] = (m+2n)-q$$

  但 q 可能為0、而 p 則沒有作用。

總結

我們學到了什麼?

- 這道等值對題是初學者很好的程式設計習題。
- 若要用到所有的已知條件，對初學者可能有點難度。
- 我們從一個很粗淺的 $O(m \times n)$ 解法開頭、發展成一個用二分搜尋法的 $O(\min(m,n)\log_2(\max(m,n))$ 解法，最後到達使用 $O(m+2n)$ 次比較的最佳解法。

$$O(n^2) \longrightarrow O(n\log_2(n)) \longrightarrow O(n)$$

想一想

- 如果我們把元素相異的條件拿掉呢? 譬如說，若 x[3..5] 有 4、 4和 4，而 y[7..8] 是 4 和 4，那麼程式就應該報告等值對為 x[3] 和 y[7]、x[3] 和 y[8]、x[4] 和 y[7]、 x[4] 和 y[8]、 x[5] 和 y[7]、 x[5] 和 y[8]。請修改程式達成這個目的。
- 如果把自小到大排好的條件去掉，我們可以把兩個陣列排序，解答自然就不可能是線性的，若只排一個陣列呢?

題外話

- 往後各講的題目有不少會出自拙著名題精選百則
- 這本書是在1989年出版、至今35年，而今對各題的了解當然有所長進，而且寫書有頁數限制，因此影片中的講解會比書裡頭的更細膩、更詳細、也更完整。

參考文獻

1. M. Rem, Small Programming Exercise 2, *Science of Computer Programming*, Vol. 3 (1983), pp. 313—319.
2. 冼鏡光，名題精選百則，儒林圖書公司，1989年初版，2010年第三版。

謝謝收看

Happy Programming!