Finding all **EQUAL PAIRS** in Two Sorted Arrays

*It takes a really bad school to ruin a good student and a really fantastic school to rescue a bad student.*

*Dennis J. Frailey*
Definition: Equal Pair

Given two sets of data $X = \{ x_1, x_2, \ldots, x_m \}$ and $Y = \{ y_1, y_2, \ldots, y_n \}$, an equal pair is an element $x_i$ in $X$ and an element $y_j$ in $Y$ such that $x_i = y_j$. 
Given two arrays \( x[\ ] \) and \( y[\ ] \), each of which contains sorted and distinct integers in ascending order, write a C program to report all equal pairs.
First Thought: 1/5

1. Suppose arrays $x[]$ and $y[]$ have $m$ and $n$ elements.
2. The first thought may be this:
   a) For each $i$, compare $x[i]$ with every $y[j]$.
   b) Report any found equal pairs.

```c
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
        }
    }
}
```
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
        }
    }
}

First Thought: 2/5

1. If \( x[i] \) and \( y[j] \) are not an equal pair, obviously nothing should happen.
2. The else part is empty and we should move on to the next iteration, comparing \( x[i] \) and \( y[j+1] \).
```c
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. If \(x[i]\) and \(y[j]\) are equal, we should record this.
2. The **then** part may have to store \(i\) and \(j\) and increase the “count” by 1 before moving on to comparing \(x[i]\) and \(y[j+1]\).
1. We need arrays `xx[ ]` and `yy[ ]` for saving the positions of the found equal pairs.
2. The `int` `k` is a counter to keep track of the number of equal pairs found so far.
3. The `break` is used to get out of the j-loop as once an equal pair is found, we could skip the remaining of the j-loop.
4. Why? The arrays have distinct numbers.
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. Let us write a function
   EQUAL_PAIRS() for this problem:
   a) int x[ ] and y[ ] are the input arrays with m and n elements.
   b) xx[ ] and yy[ ] are the returned arrays that store the locations.
   c) k, the function value, is the number of equal pairs.
Is This Good?: 1/2

1. Let us determine the number of comparisons used.
2. In the worst case, if \( x[i] \) is not equal to any element of array \( y[j] \), \( n \) comparisons are needed for the \( j \)-loop.
3. The \( i \)-loop iterates \( m \) times, and, in the worst case, the total number of comparisons is \( O(m \times n) \), not very good.
4. If \( m=n \), it is an \( O(n^2) \) solution.

```c
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. This could be an acceptable solution for beginners.

2. However, it does not use all of the given conditions such as sorted to its maximum.

3. It is easy to improve this version to some degree; but these could just be minor.

4. Well, let us see how we could improve this version for beginners.
Improvements? 1/5

1. Let us assume $x[0] < y[0]$.
2. Find a $x[p]$ such that $x[p-1] < y[0] \leq x[p]$
3. $x[p-1] < y[0] \leq x[p]$
4. In this way, we do not compare $x[0]$ to $x[p-1]$ because they cannot be found in $x[]$.
5. Hence, we have to handle $x[p..m-1]$ and $y[0..n-1]$ using $p$ comparisons.
6. Can we do more? YES!

```java
for (i = 0; i < m; i++)
  if (x[i] >= y[0]) {
    p = i;
    break;
  }
```
1. If $y[n-1] > x[m-1]$, we could cut the comparison range further.

2. Find a $y[n-q]$ such that
   
   $$y[n-(q-1)] < x[m-1] <= y[n-q]$$
   
3. In this way, from $y[n-q]$ to $y[n-1]$ cannot be in $x[]$ because they are all larger than the entries in $x[]$.

4. The ranges are reduced to $x[p..m-1]$ and $y[0..n-q]$. 

**Improvements? 2/5**
1. Thus, $p$ comparisons are used to cut the first $p$ elements from $x[]$ and $q$ comparisons to cut $q$ elements from $y[]$.

2. The remaining elements in $x[]$ is $m-p$ and the remaining elements in $y[]$ is $n-q$.

3. Therefore, each $x[i]$ in the $x[]$ is compared with every entries in $y[0..(n-q)]$ which requires $n-q$ comparisons.

Improvements? 3/5
1. Because there are \( m-p \) entries in \( x[] \), the total number of comparisons is \((m-p)\times(n-q)\).

2. We also need \( p \) comparisons to cut \( x[] \) and \( q \) comparisons for the \( y[] \).

3. The total number of comparisons is \((m-p)\times(n-q) + (p+q)\).

4. The worst case is still \( O(m\times n) \).

5. There is no improvement in the worst case!
1. These modifications indeed can speed up the original version. 
2. However, in doing so does not improve the worst-case in a big way. 
3. Therefore, we have to think differently to break the $O(m \times n)$ barrier.

**Improvements? 5/5**

```c
for (j = n-1; j >= 0; j--)
    if (y[j] < x[m-1]) {
        q = n-j-1;
        break;
    }
```
Is there a better way?

Of course, the immediate answer could be the BINARY SEARCH, if you have reached the data structures course.
Basic Idea: 1/2

1. We have an array \( y[n] \) whose elements are distinct and sorted in ascending order.
2. Is a given data item \( DATA \) in the array? If it is, where is it?

```plaintext
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
```
Basic Idea: 2/2

1. Because half of the elements is cut after each comparison, in no more than \( \log_2(n) \) iterations we find the item or nothing.

2. The worst complexity is \( O(\log_2(n)) \)

```java
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
```
```c
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;
}
```

**Solution: 1/6**

1. This a possible solution.
2. For each `x[i]`, search array `y[]` to find it.
3. If `x[i]` is found, save positions to `xx[]` and `yy[]` and increase the count of equal pairs (`int k`).
4. Each iteration requires at most 2 comparisons.
4. Because each search requires $O(\log_2(n))$ comparisons, the total number of comparisons is $O(m \times \log_2(n))$.  

int EQUALPAIRS(int x[], int y[], int m, int n, int xx[], int yy[])
{
    int i, j, k = 0; // binary search
    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;
}

Solution: 2/6

1. Use x[i] to search y[],
   complexity: $O(m \times \log_2(n))$.
2. Use y[j] to search x[],
   complexity: $O(n \times \log_2(m))$.

3. Which way is better?
4. Of course, one should choose the longer array (i.e., the larger of m and n) to be searched by the shorter array.

6. Any justification?
1. The left table shows the values of $x$ and $\log_2(x)$.

2. It is clear that the increase of $\log_2(x)$ is much slower than that of the $x$.

3. Therefore, using the shorter array to search the longer array appears to be more efficient.

4. We need a proof rather than an observation!
1. We want to know whether using the shorter array to search the longer one \((m \log_2(n))\) uses less comparisons than using the longer array to search the shorter one \((n \log_2(m))\).

2. Divide both sides by \(\log_2(n) \times \log_2(m)\).

3. We have a function \(f(x) = \frac{x}{\log_2(x)}\).

If function \(f(x)\) is an increasing one, then \(m < n\) means \(m/\log_2(m) < n/\log_2(n)\) and in turn it gives \(m \log_2(n) < n \log_2(m)\).
Solution: 5/6

1. If \( x = 1 \), \( \log_2(1) = 0 \) and \( f(x) \) is \( \infty \).

2. Compute the derivative of \( f(x) \):

\[
\frac{df}{dx} = \ln(2) \cdot \frac{d}{dx} \left( \frac{x}{\ln(x)} \right)
\]

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

\[
= \ln(2) \frac{\ln(x) - 1}{(\ln(x))^2}
\]

If function \( f(x) \) is an increasing one, then \( m < n \) means \( \frac{m}{\log_2(m)} < \frac{n}{\log_2(n)} \) and in turn it gives \( m\log_2(n) < n\log_2(m) \).
1. The minimum of $f(x)$ is at $x = e$ because $f'(e) = 0$!

2. $f(e) = e / \log_2(e) = 1.884169$.

3. $(e, 1.884169)$ is the minimum.

4. If $x > e$, $f(x)$ is increasing!

5. Hence, as long as $m$ and $n$ are larger than or equal to 2, using the shorter array to search the longer one is the way to go!

If function $f(x)$ is an increasing one, then $m < n$ means $m / \log_2(m) < n / \log_2(n)$ and in turn it gives $m \log_2(n) < n \log_2(m)$. 

**Solution: 6/6**
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
            //
        }
    }
}

Improvements: 1/3

1. An obvious improvement is that if \( x[i] \) is not in the range of \( y[0] \) and \( y[n-1] \), then **DO NOT SEARCH**.
2. This is because if \( x[i] \) is not in the range of \( y[] \), you won’t find it in \( y[0..n-1] \).
3. In doing so, the worst case is still the same. For example:
\[
\begin{align*}
  x[] &= \{1, 3, 5, 7, 9\} \\
  y[] &= \{0, 2, 4, 6, 8, 10, 12\}.
\end{align*}
\]
Improvements: 2/3

1. Just like in the previous solution, we could ignore portions of arrays \( x[] \) and/or \( y[] \) to cut down the number of comparisons.

2. However, this does not improve the worst case because \( p \) and \( q \) could be 0!

```
for (j = n-1; j >= 0; j--)
    if (y[j] < x[m-1]) {
        q = n-j-1;
        break;
    }
```
1. If $x[i]$ is equal to $y[j]$, then elements in $x[i+1..m-1]$ cannot be found in $y[0..j]$.
2. This is because $x[]$ and $y[]$ are sorted with distinct data.
3. Hence, if $x[i] = y[j]$, the search range of $x[i+1]$ is $y[j+1]$ to $y[n-1]$.
4. It does not improve the worst case complexity.
5. It is an important observation for developing an optimal solution.
An Optimal Solution

We have all the needed tools to do this
1. If \( x[i] = y[j] \), then elements in \( x[i+1..m-1] \) cannot be found in \( y[0..j] \).
2. Similarly, \( y[j+1..n-1] \) cannot be found in \( x[0..i] \).
3. This is because \( x[] \) and \( y[] \) are sorted with distinct data.
4. Hence, if \( x[i] = y[j] \), the search range would be restricted to \( x[i+1..m-1] \) and \( y[j+1..n-1] \).
Idea: 2/5

1. If $x[i] < y[j]$, what is the next step?
2. $x[i]$ cannot be found in $y[j..n-1]$.
3. Therefore, try the next element $x[i+1]$.
4. Consequently, if $x[i] < y[j]$, what we need to do is $i++$. 
Idea: 3/5

1. If $x[i] > y[j]$, what is the next step?
2. $y[j]$ cannot be found in $x[i..m-1]$.
3. Therefore, try the next element $y[j+1]$.
4. Consequently, if $x[i] > y[j]$, what we need to do is $j++$. 
1. We have three cases:
   a) If $x[i] < y[j]$, set $i++$ to advance $x[i]$ to the next.
   b) If $x[i] > y[j]$, set $j++$ to advance $y[j]$ to the next.
   a) If $x[i] = y[j]$, set $i++$ and $j++$ to advance both $x[i]$ and $y[j]$ to the next.

2. In this way, two comparisons per iteration are needed!
1. i and j must start with 0!
2. As long as there are $x[i]$ and $y[j]$ in the arrays, the process continues.
3. Therefore, the loop structure looks like this:

```java
i = j = 0;
while (i < m && j < n) {
    //
    // do the comparisons
    //
}
```
1. We have three cases:
   a) If $x[i] < y[j]$, set $i++$ to advance $x[i]$ to the next.
   b) If $x[i] > y[j]$, set $j++$ to advance $y[j]$ to the next.
   c) If $x[i] = y[j]$, set $i++$ and $j++$ to advance both $x[i]$ and $y[j]$ to the next.
**Analysis: 1/5**

1. **How many comparisons?**
2. If \( x[i] < y[j] \), then \( i++ \) and uses \( 1 \) comparison.
3. If \( x[i] > y[j] \), then \( j++ \) and uses \( 2 \) comparisons.
4. If \( x[i] = y[j] \), then \( i++ \) and \( j++ \) and uses \( 2 \) comparisons.
1. How many comparisons?

2. If \( i \) takes \( m \) moves and \( j \) takes \( n \) moves, the total number of comparisons is \( m+2n \)!

3. Is this possible?

4. Yes, it is possible.

5. If \( x[m-1] \) is equal to \( y[n-1] \), then “\( i \) takes \( m \) moves and \( j \) takes \( n \) moves.”

6. This requires \( m+2n \) comparisons.

7. This is a \( O(m+2n) \) solution.
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;
}
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;
}

1. Assume $m < n$.  
2. If $x[m-1] < y[0]$, then only $m$ comparisons are needed.  
3. On the other hand, if $x[0] > y[n-1]$, then every comparison is a “>”, the total number of comparisons is $2n$.  

Analysis: 4/5
1. Assume $m < n$.
2. Therefore, call `EQUAL_PAIRS()` with the longer array as the $x[]$ and the shorter array as $y[]$ so that $m+2n$ is smaller.
Initially, \( i \) and \( j \) are both 0.
Example: 2/5

- Because $x[0] < y[0]$, do $i++$ until $x[3] > y[0]$.
- We have 3 <s and 1 >.
- Then, $i=3$ and $j$ moves to 1.

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

```plaintext
0  1  2  3  4  5  6
1  2  3   8 14  …

0     1     2       3       4       5       6       7

< Comparison steps : 3
> Comparison steps : 1
• Previous total : 0
• Total comparisons : 3 + 1*2 = 5

No of < : 3
No of > & = : 1
```
Example: 3/5

- Because \(x[3] > y[1]\), do \(j++\) until \(x[3] < y[4]\).
- We have 3 >s and 1 <.
- Then, \(j=4\) and \(i\) moves to 4.

\[
\begin{array}{cccccccc}
1 & 2 & 3 & 8 & 14 & \ldots \\
\end{array}
\]
Example: 4/5

- We have 3 $>$s and 1 $=$.
- Then, $i = 4$ and $j = 7$. 

Comparison steps:

1. $<$ Previous total: 1
2. $>$ Total comparisons: $12 + (0 + 4 \times 2) = 20$

<table>
<thead>
<tr>
<th>$i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x[]$</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>8</td>
<td>14</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$j$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y[]$</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

- No of $<$ = 0
- No of $>$ & $=$ = 4
Example: 5/5

- The total number of comparisons is \(20 = 4(<)+2\times8(>)\).
- This is equal to \(m+2n = 4+2\times8\).
- The next step is \(i++\) and \(j++\).
Improvements?

- It is easy to avoid unnecessary comparisons when $x[0] > y[n-1]$ or $x[m-1] < y[0]$.
- The technique to remove some unnecessary comparisons discussed on Slides 11--15 may also be used.
- These could be minimal with respect to complexity.
- If $p$ and $q$ comparisons are used to trim the first and second parts of $x[]$ and $y[]$, respectively, the number of comparisons is $(p+q)+[(m-p)+2(n-q)] = (m+2n)-q$. 
A Summary
What did we learn?

- This **EQUAL PAIR** problem is a good programming problem for beginners.
- It could be a little bit challenging if **not all given conditions are used** properly and fully.
- We started with a naïve $O(m \times n)$ solution, moved on to a better one $O(\min(m,n) \log_2(\max(m,n)))$ and finally reached an optimal one $O(m+2n)$.

What if the sorted condition is dropped? Well, one could sort both arrays and the solution cannot be linear! Why?
References

The End

Happy Programming!