Divide and Conquer Algorithms

Divide-and-Conquer

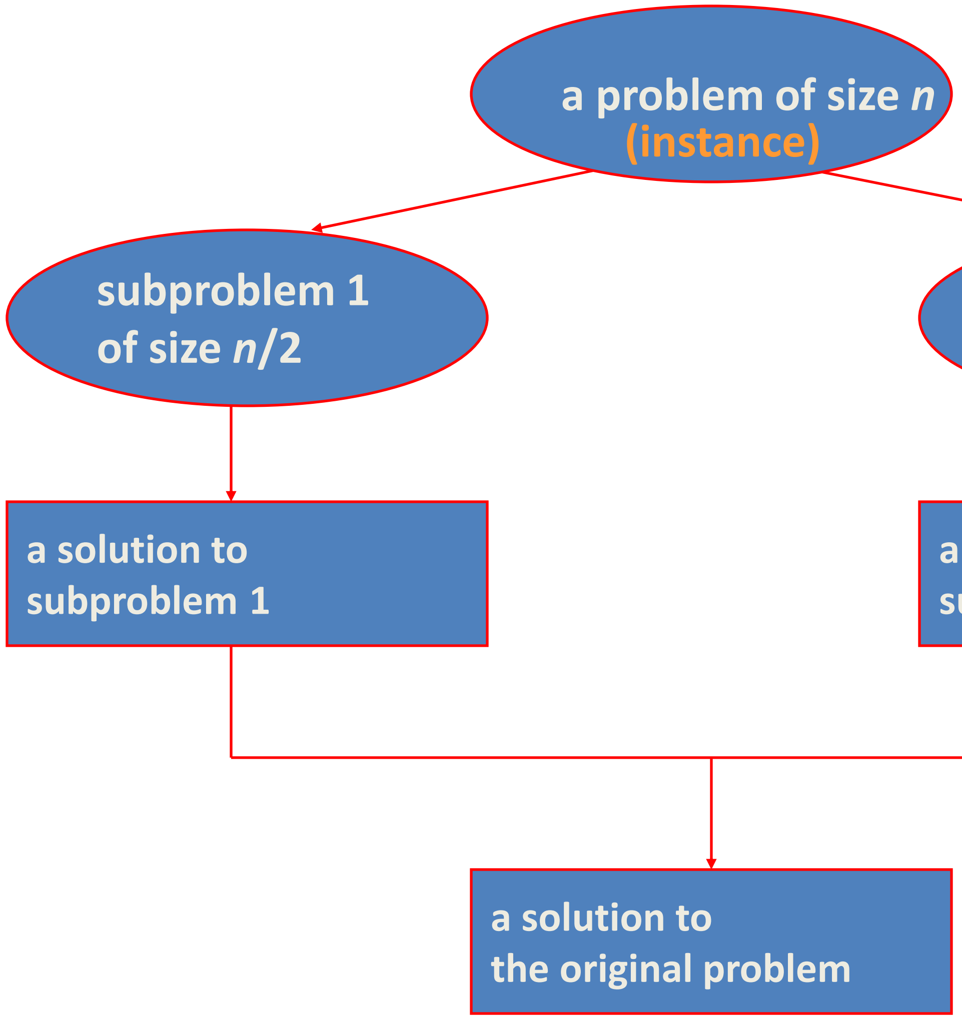
The most-well known algorithm design strategy:

1. Divide instance of problem into two or more smaller instances

2. Solve smaller instances recursively

3. Obtain solution to original (larger) instance by combining these solutions

Divide-and-Conquer Technique (cont.)



It general leads to a recursive algorithm!

Linear Search

Problem: Given a list of N values, determine whether a given value X occurs in the list.

For example, consider the problem of determining whether the value 55 occurs in:

1	2	3	4	5	6	7	8
17	31	9	73	55	12	19	7

There is an obvious, correct algorithm:

start at one end of the list,
if the current element doesn't equal the search target, move to the next element,
stopping when a match is found or the opposite end of the list is reached.

Basic principle: divide the list into the current element and everything before (or after) it; if current isn't a match, search the other case

Linear Search

```
algorithm LinearSearch takes number X, list number L, number Sz

# Determines whether the value X occurs within the list L.
# Pre:  L must be initialized to hold exactly Sz values
#

   # Walk from the upper end of the list toward the lower end,
   # looking for a match:

   while Sz > 0 AND L[Sz] != X
      Sz := Sz - 1
   endwhile

   if Sz > 0          # See if we walked off the front of the list
      display true    # if not, got a match
   else
      display false   # if so, no match

   halt
```

Binary Search

Repeatedly halving the size of the “search space” is the main idea behind the method of binary search.

An item in a sorted array of length n can be located with just $\log_2 n$ comparisons.

Binary Search

Repeatedly halving the size of the "search space" is the main idea behind the method of binary search.

An item in a sorted array of length n can be located with just $\log_2 n$ comparisons.

"Savings" is significant!

n	log2(n)
100	7
1000	10
10000	13

Binary search: target x = 70

	1	2	3	4	5	6	7	8	9	10	11	12
v	12	15	33	35	42	45	51	62	73	75	86	98

L: 1

Mid: 6

R: 12

v(Mid) <= x

So throw away the left half...

Binary search: target x = 70

	1	2	3	4	5	6	7	8	9	10	11	12
v	12	15	33	35	42	45	51	62	73	75	86	98

L: 6

Mid: 9

R: 12

x < v(Mid)

So throw away the right half…

Binary search: target x = 70

	1	2	3	4	5	6	7	8	9	10	11	12
v	12	15	33	35	42	45	51	62	73	75	86	98

L: 6

Mid: 7

R: 9

v(Mid) <= x

So throw away the left half…

Binary search: target x = 70

	1	2	3	4	5	6	7	8	9	10	11	12
v	12	15	33	35	42	45	51	62	73	75	86	98

L: 7

Mid: 8

R: 9

v(Mid) <= x

So throw away the left half…

Binary search: target x = 70

	1	2	3	4	5	6	7	8	9	10	11	12
v	12	15	33	35	42	45	51	62	73	75	86	98

L: 8

Mid: 8

R: 9

Done because

R-L = 1

```
function L = BinarySearch(a,x)
# x is a row n-vector with x(1) < ... < x(n)
# where x(1) <= a <= x(n)
# L is the index such that x(L) <= a <= x(L+1)

L = 1;  R = length(x);
# x(L) <= a <= x(R)
while R-L > 1
    mid = floor((L+R)/2);
    # Note that mid does not equal L or R.
    if a < x(mid)
        # x(L) <= a <= x(mid)
        R = mid;
    else
        # x(mid) <= a <== x(R)
        L = mid;
    end
end
```

Binary search is efficient, but how do we sort a vector in the first place so that we can use binary search?

- Many different algorithms out there...
- Let’s look at **merge sort**
- An example of the “divide and conquer” approach

Merge sort: Motivation

If I have two helpers, I'd…

- Give each helper half the array to sort
- Then I get back the sorted subarrays and merge them.

What if those two helpers each had two sub-helpers?

And the sub-helpers each had two sub-sub-helpers? And…

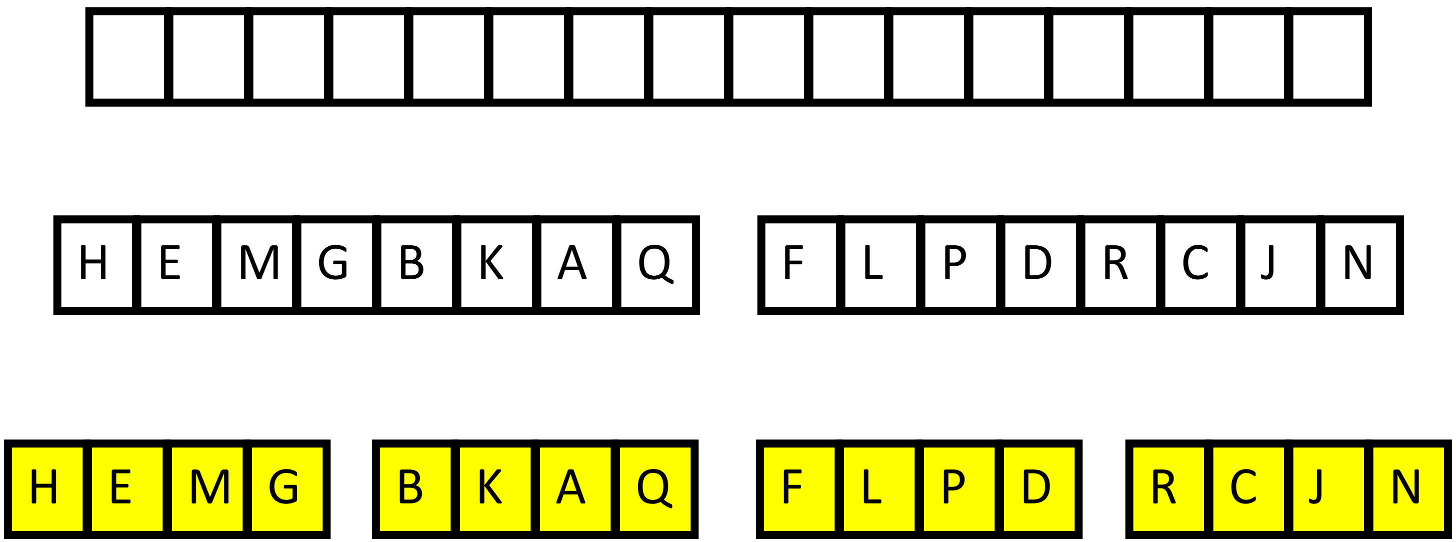
Subdivide the sorting task

H	E	M	G	B	K	A	Q	F	L	P	D	R	C	J	N

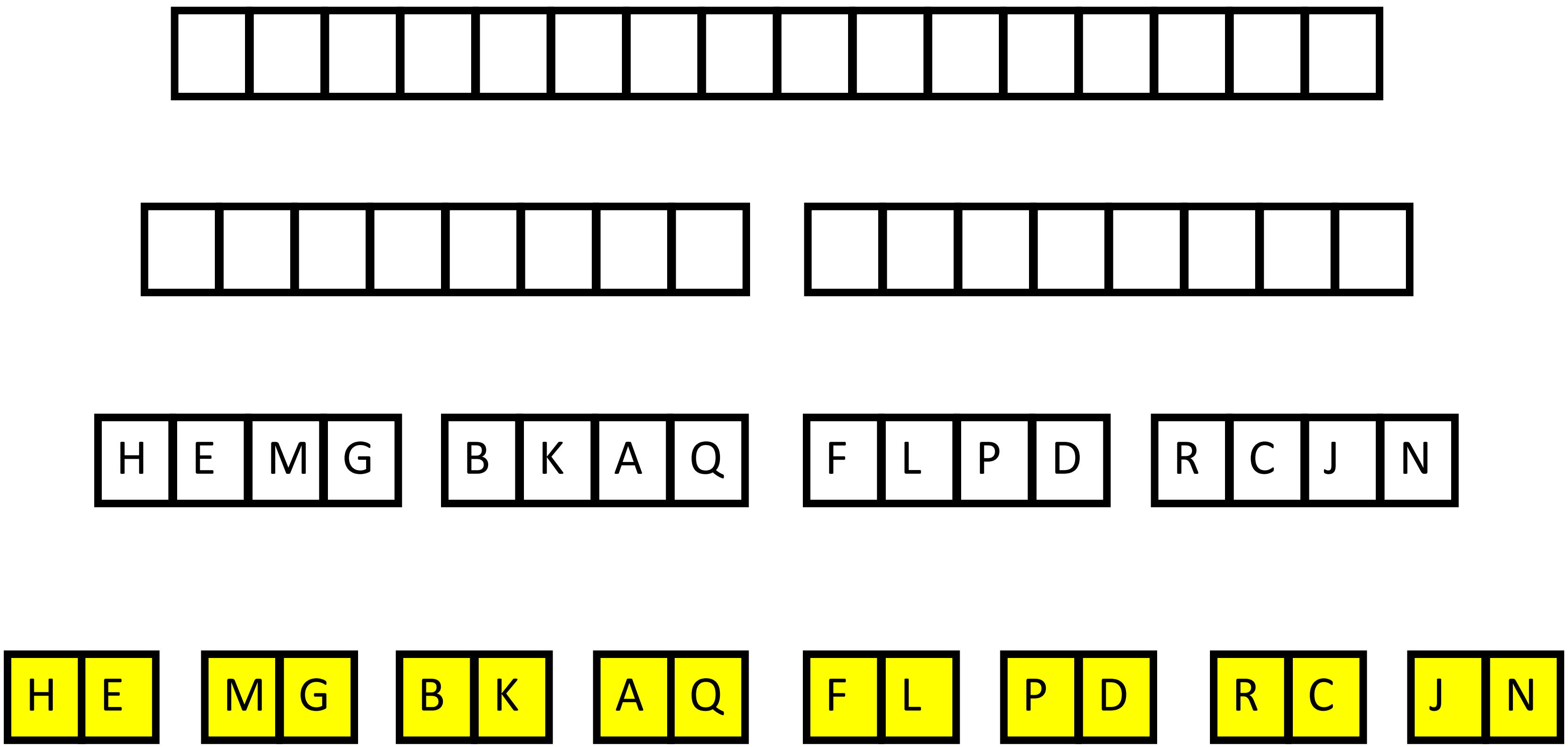
H	E	M	G	B	K	A	Q

F	L	P	D	R	C	J	N

Subdivide again



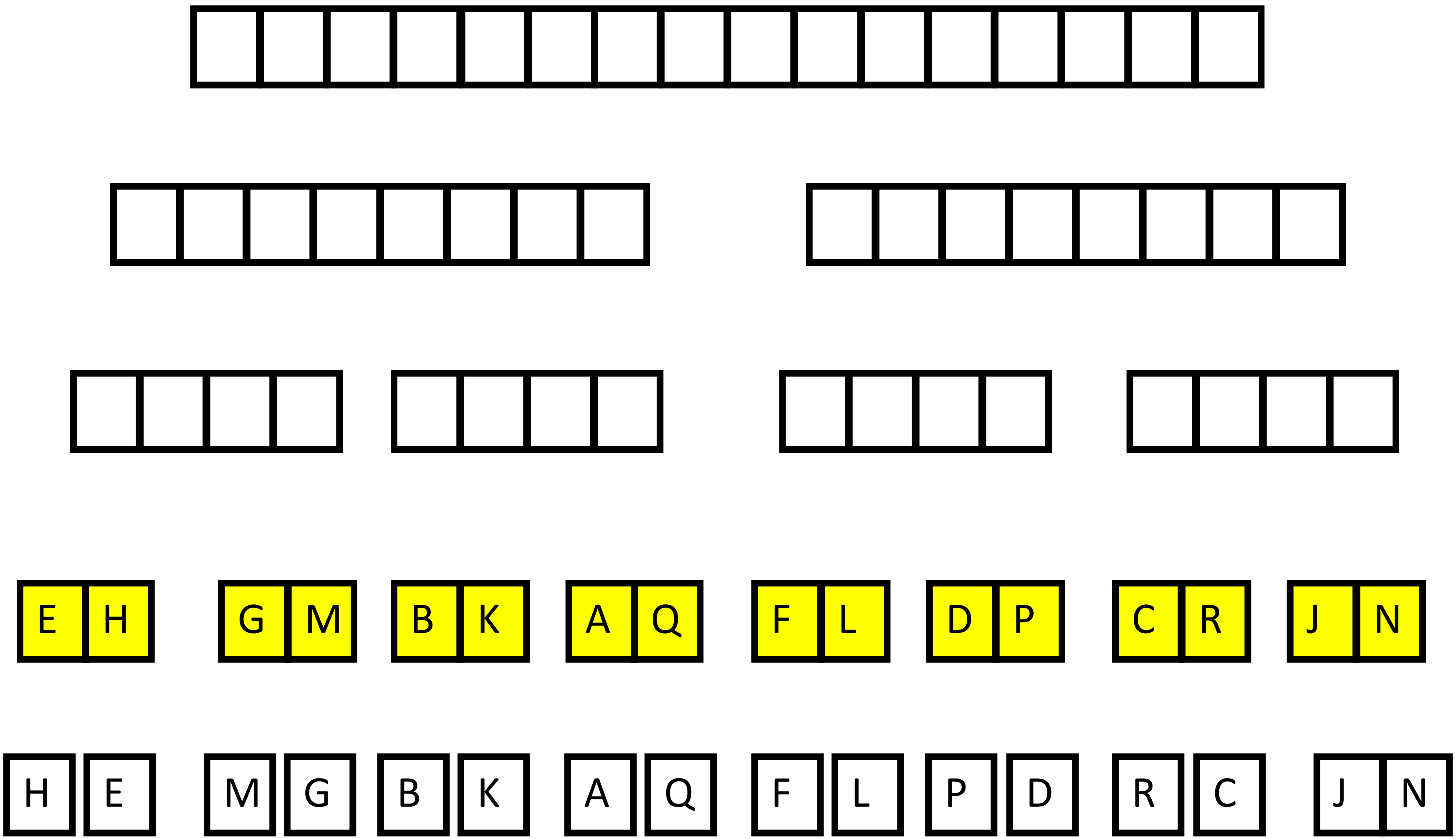
And again



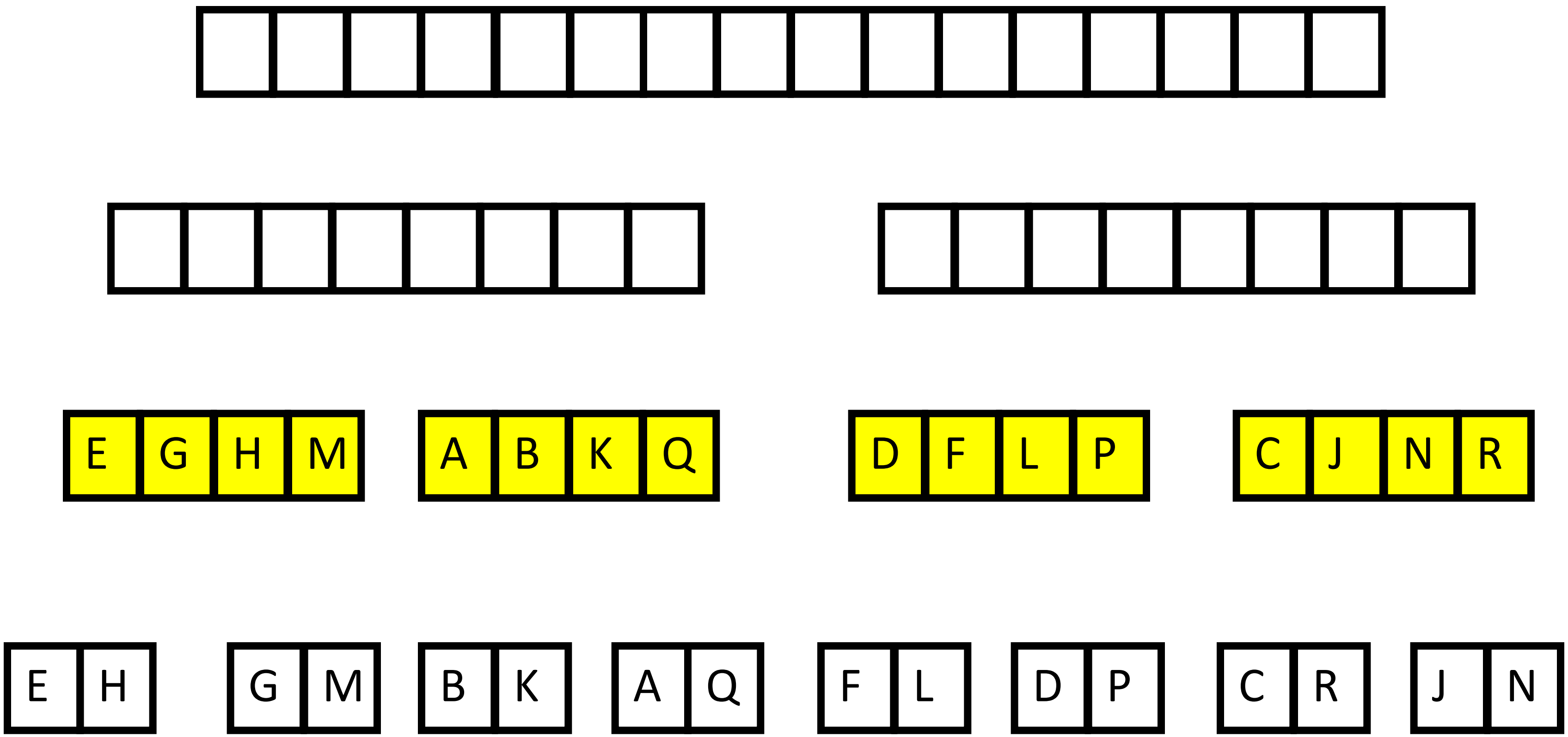
And one last time

H E M G B K A Q F L P D R C J N

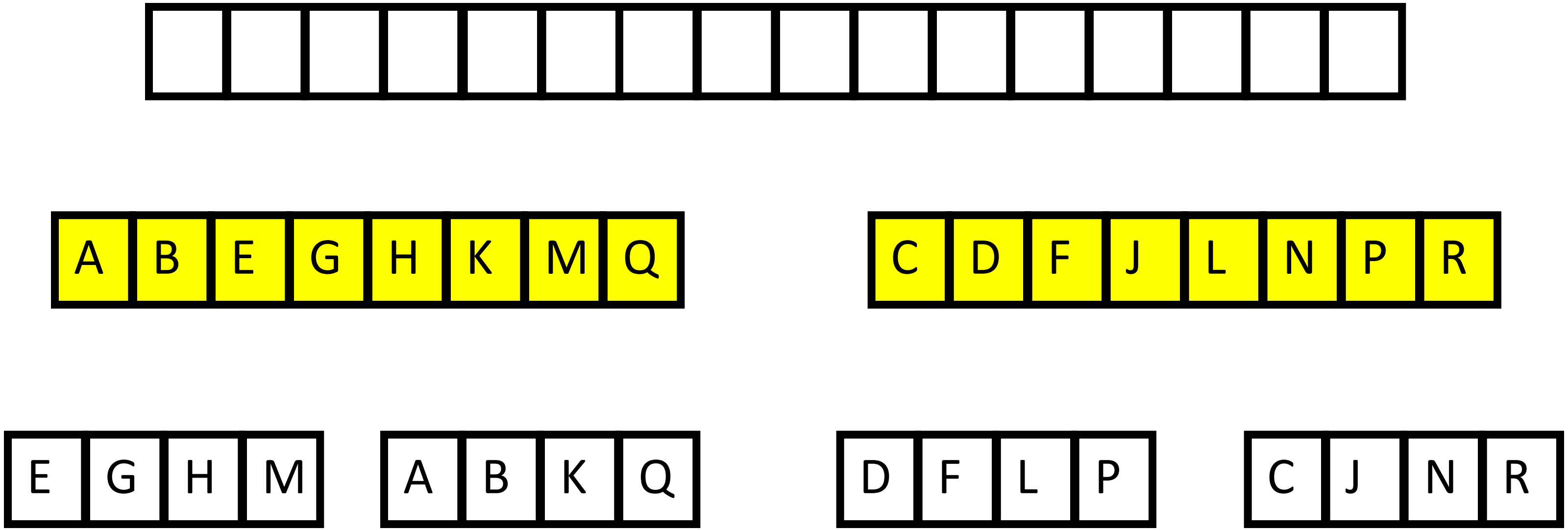
Now merge



And merge again



And again



And one last time

A	B	C	D	E	F	G	H	J	K	L	M	N	P	Q	R

A	B	E	G	H	K	M	Q

C	D	F	J	L	N	P	R

Done!

A	B	C	D	E	F	G	H	J	K	L	M	N	P	Q	R

```
function y = mergeSort(x)
# x is a vector.  y is a vector
# consisting of the values in x
# sorted from smallest to largest.

n = length(x);
if n==1
      y = x;
else
      m  = floor(n/2);
      yL = mergeSortL(x(1:m));
      yR = mergeSortR(x(m+1:n));
      y  = merge(yL,yR);
end
```

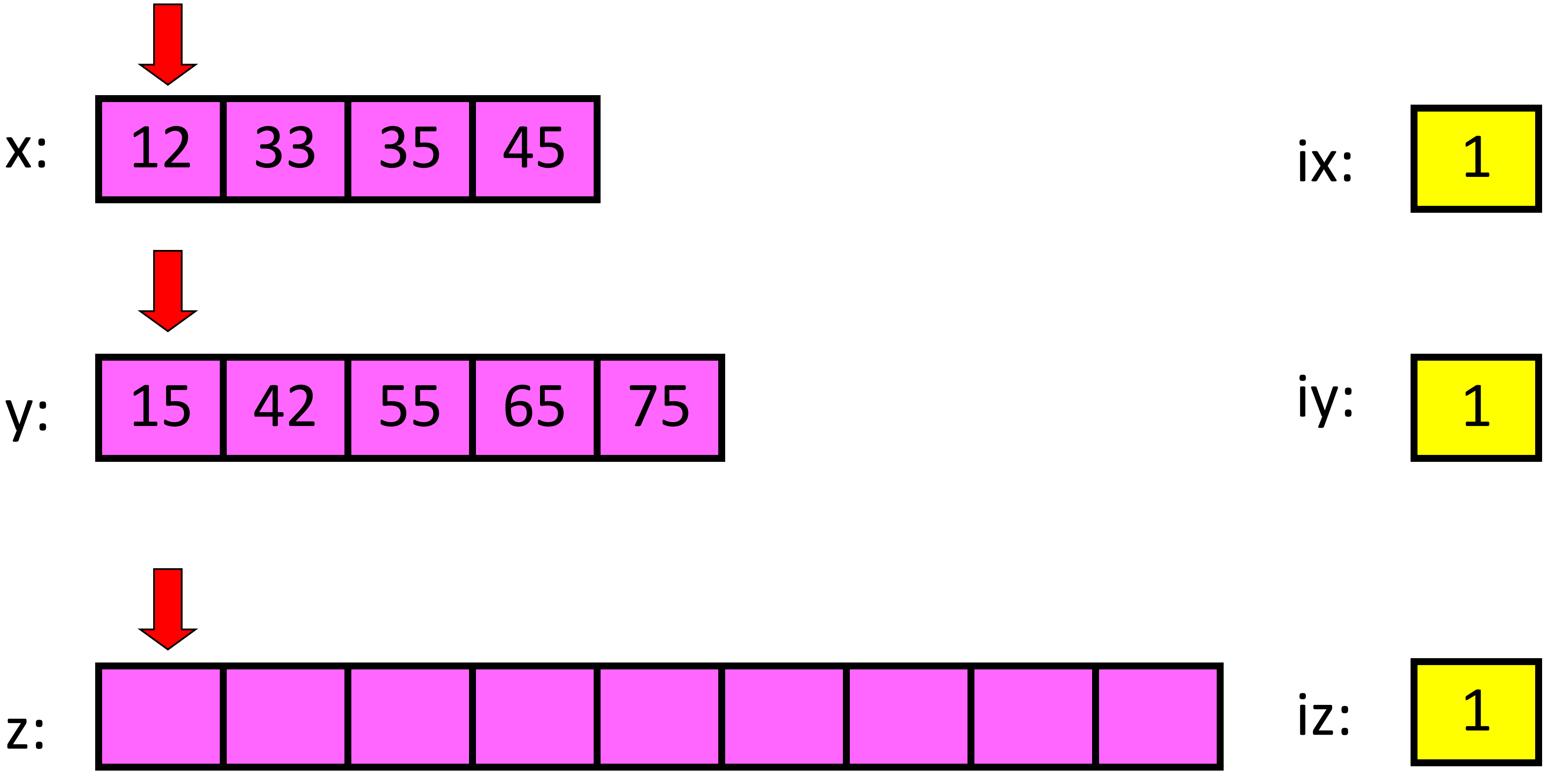
The central sub-problem is the merging of two sorted arrays into one single sorted array

12	33	35	45

15	42	55	65	75

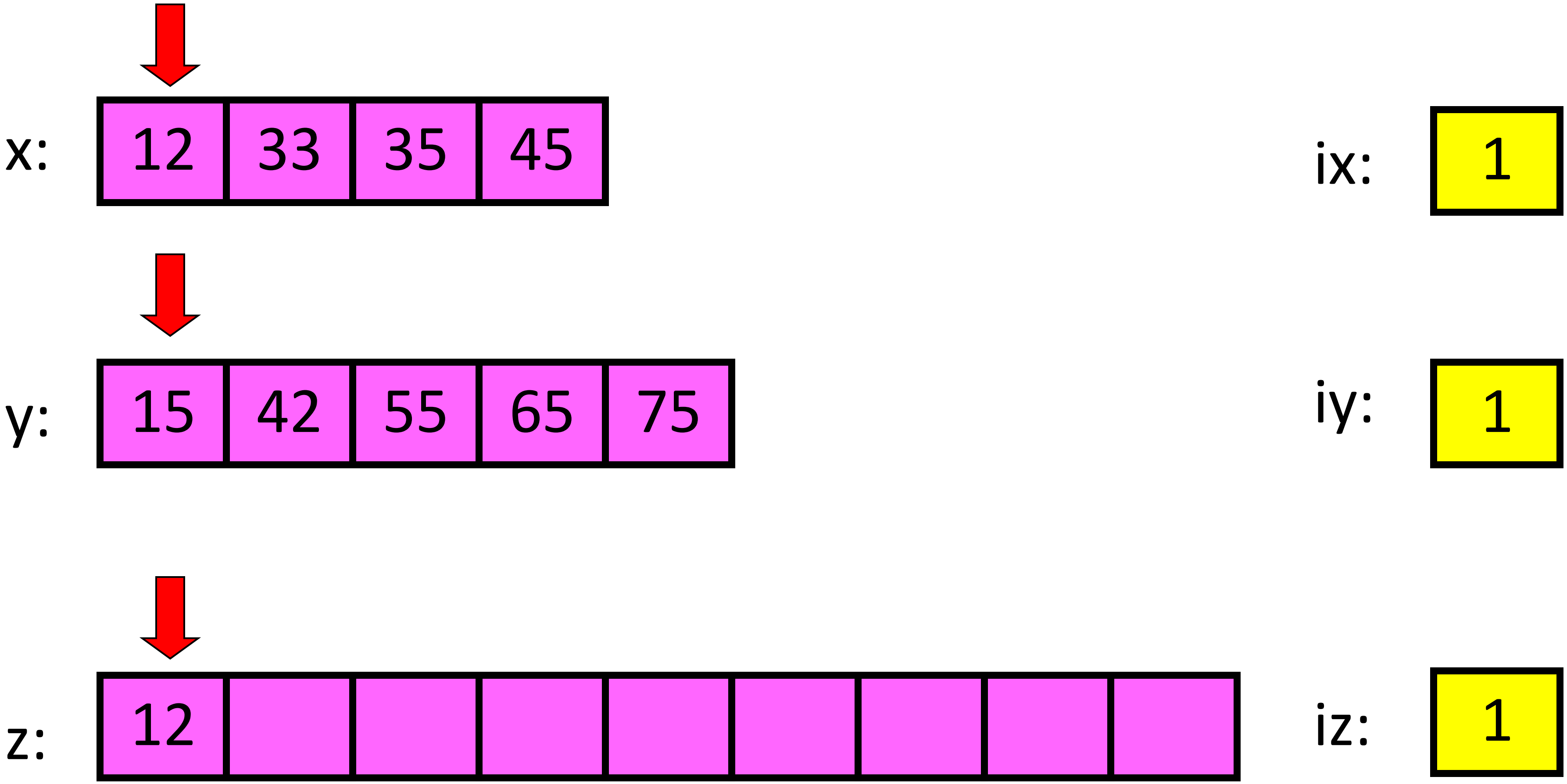
12	15	33	35	42	45	55	65	75

Merge



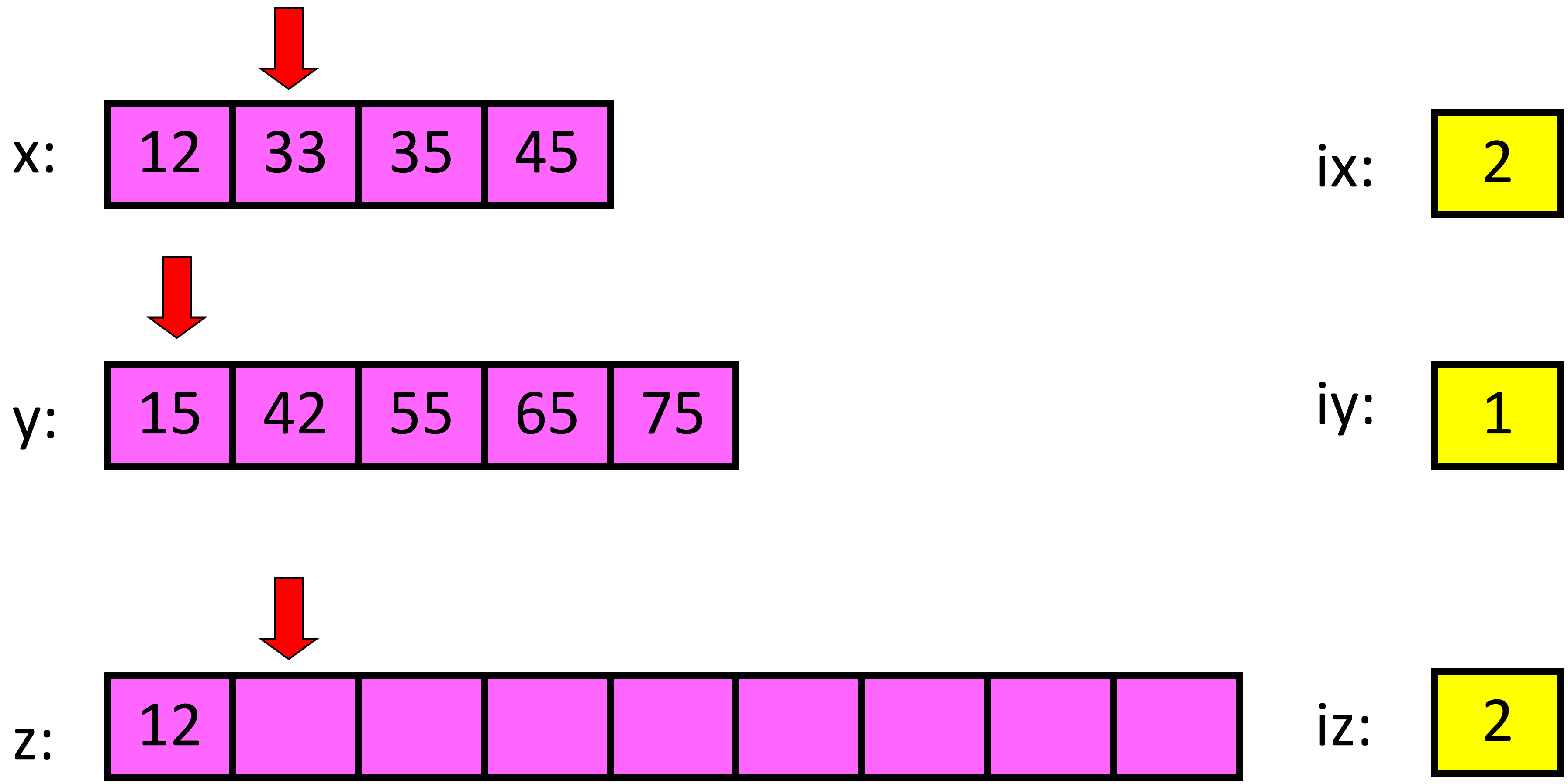
ix<=4 and iy<=5: x(ix) <= y(iy) ???

Merge



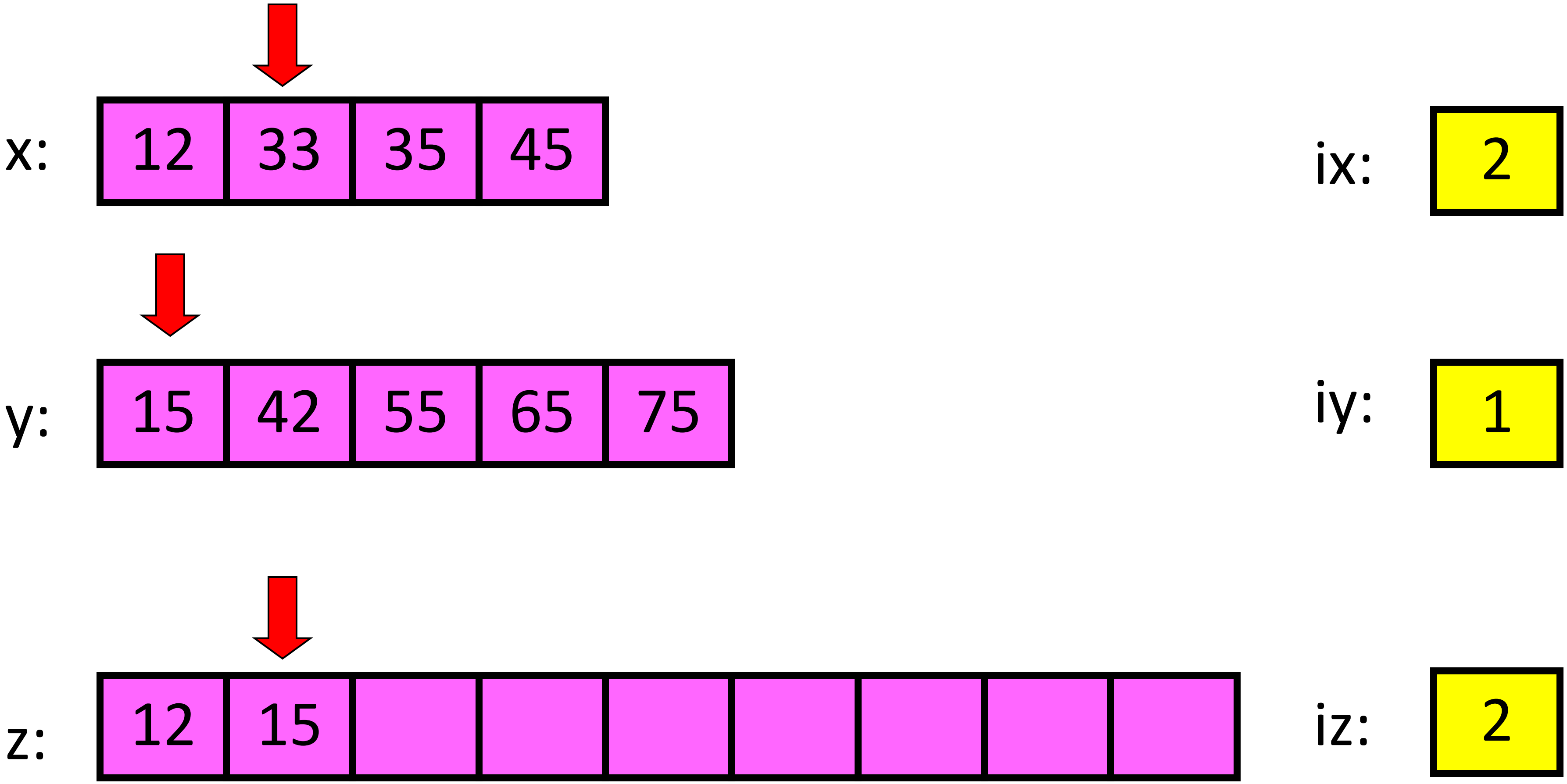
ix<=4 and iy<=5: x(ix) <= y(iy) YES

Merge



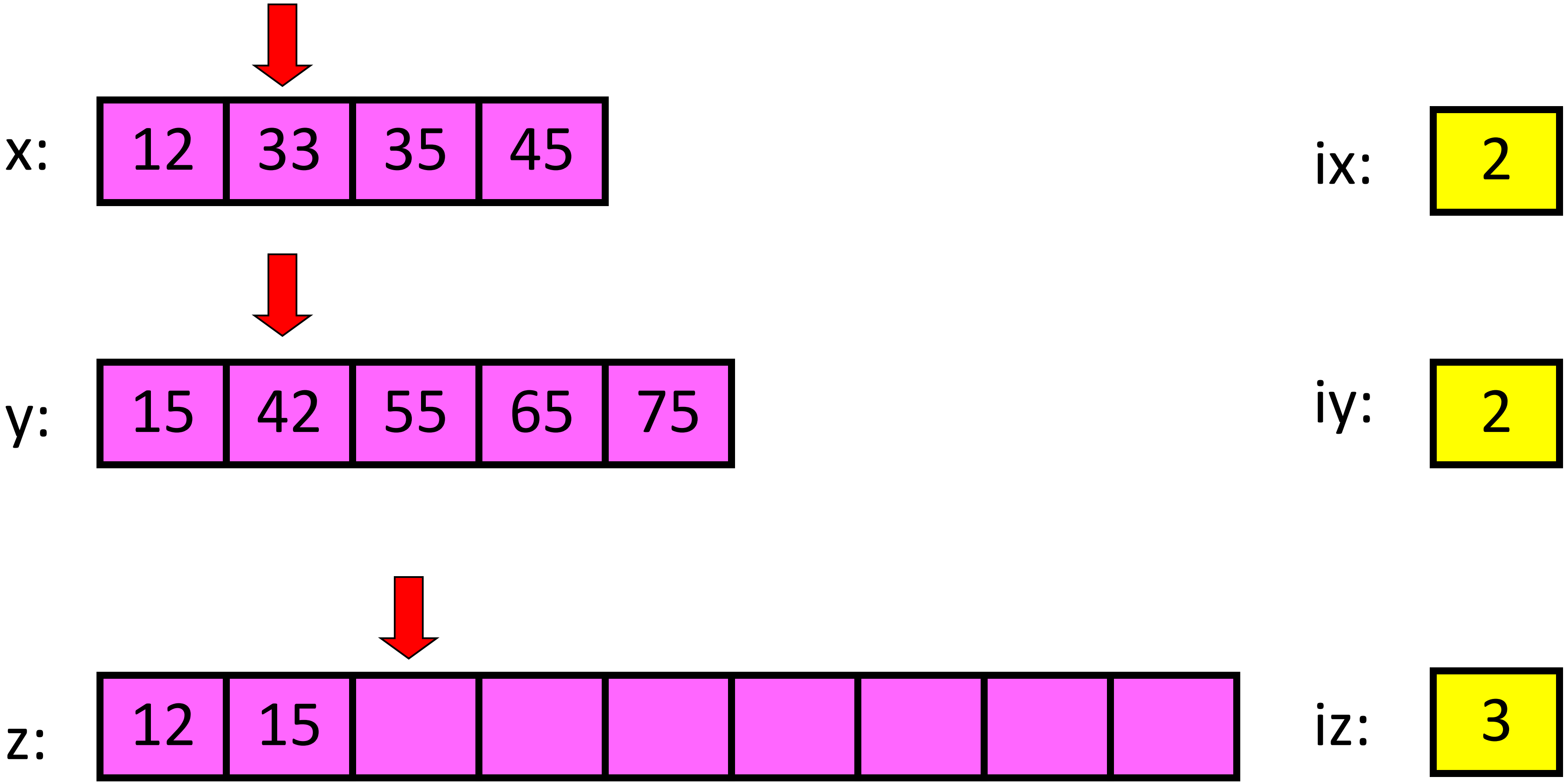
ix<=4 and iy<=5: x(ix) <= y(iy) ???

Merge



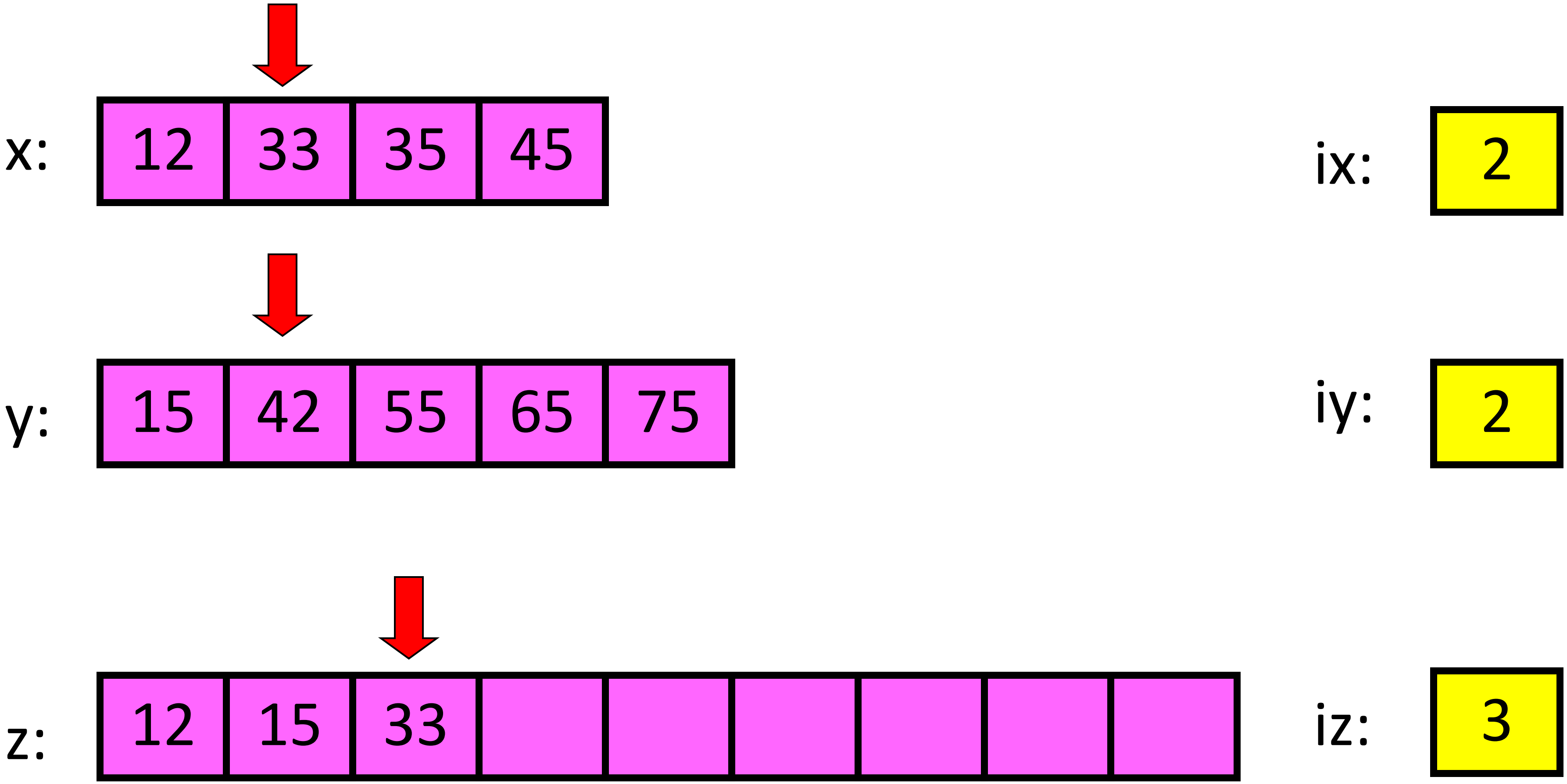
ix<=4 and iy<=5: x(ix) <= y(iy) NO

Merge



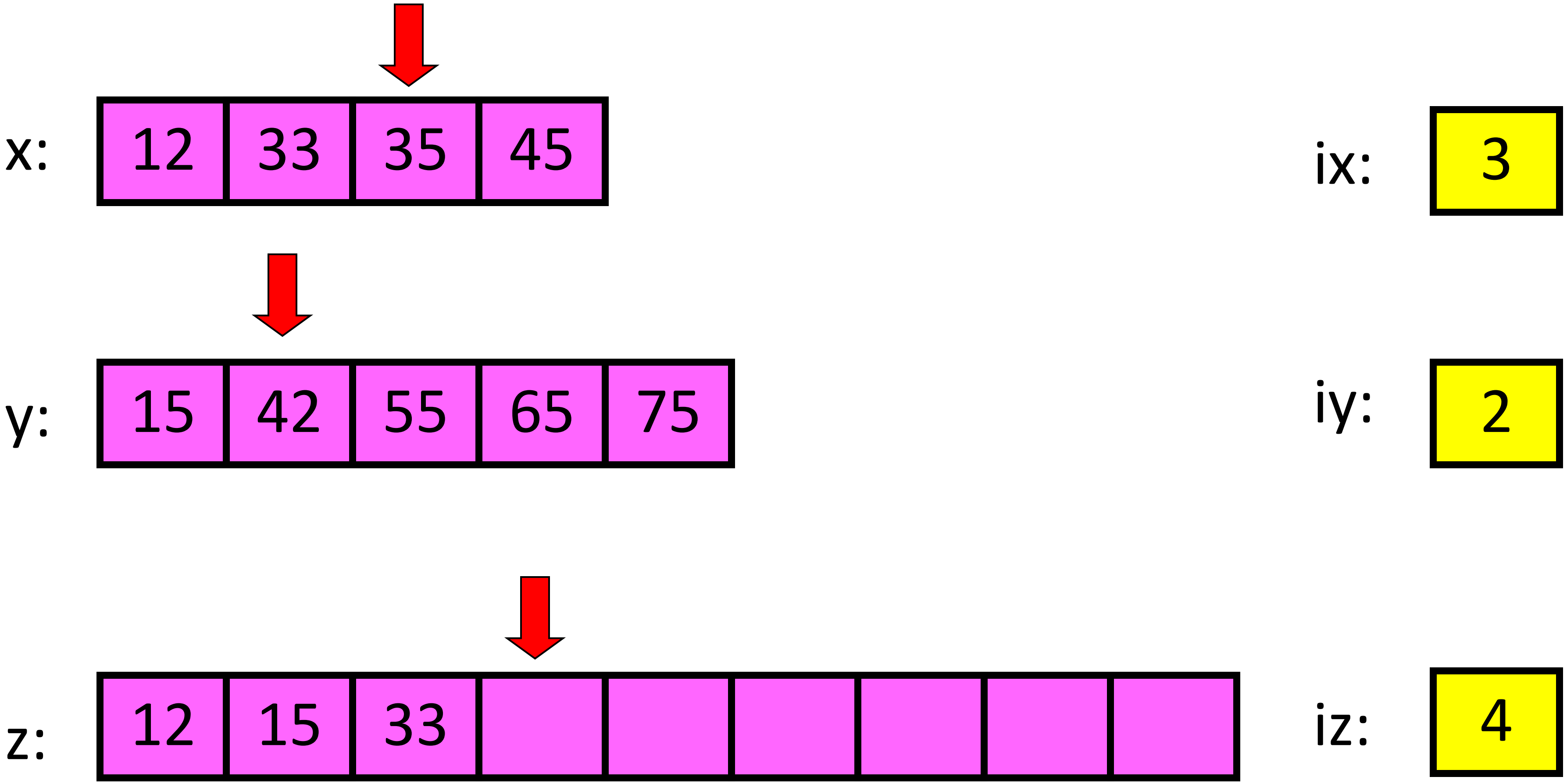
ix<=4 and iy<=5: x(ix) <= y(iy) ???

Merge



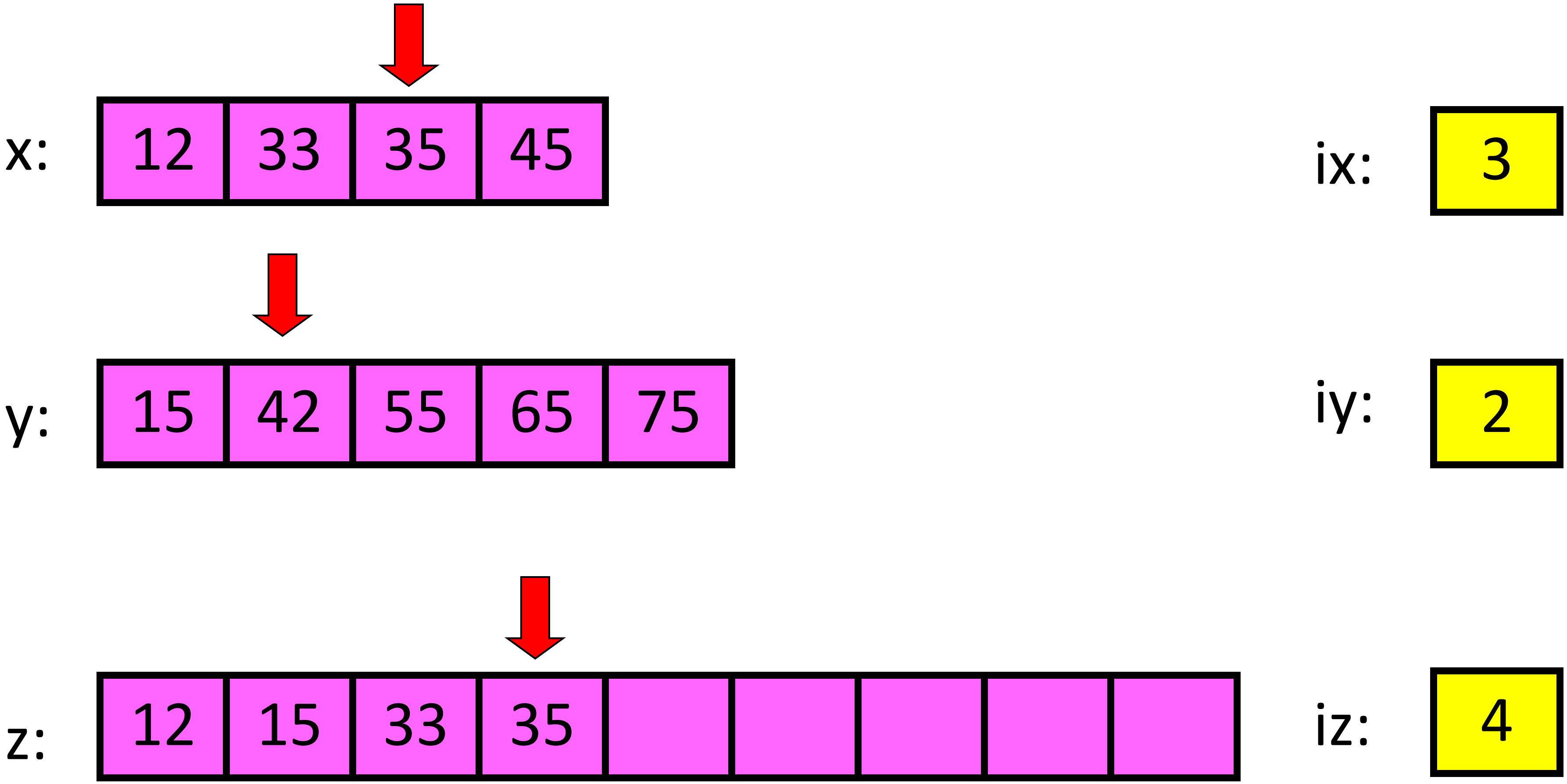
ix<=4 and iy<=5: x(ix) <= y(iy) YES

Merge



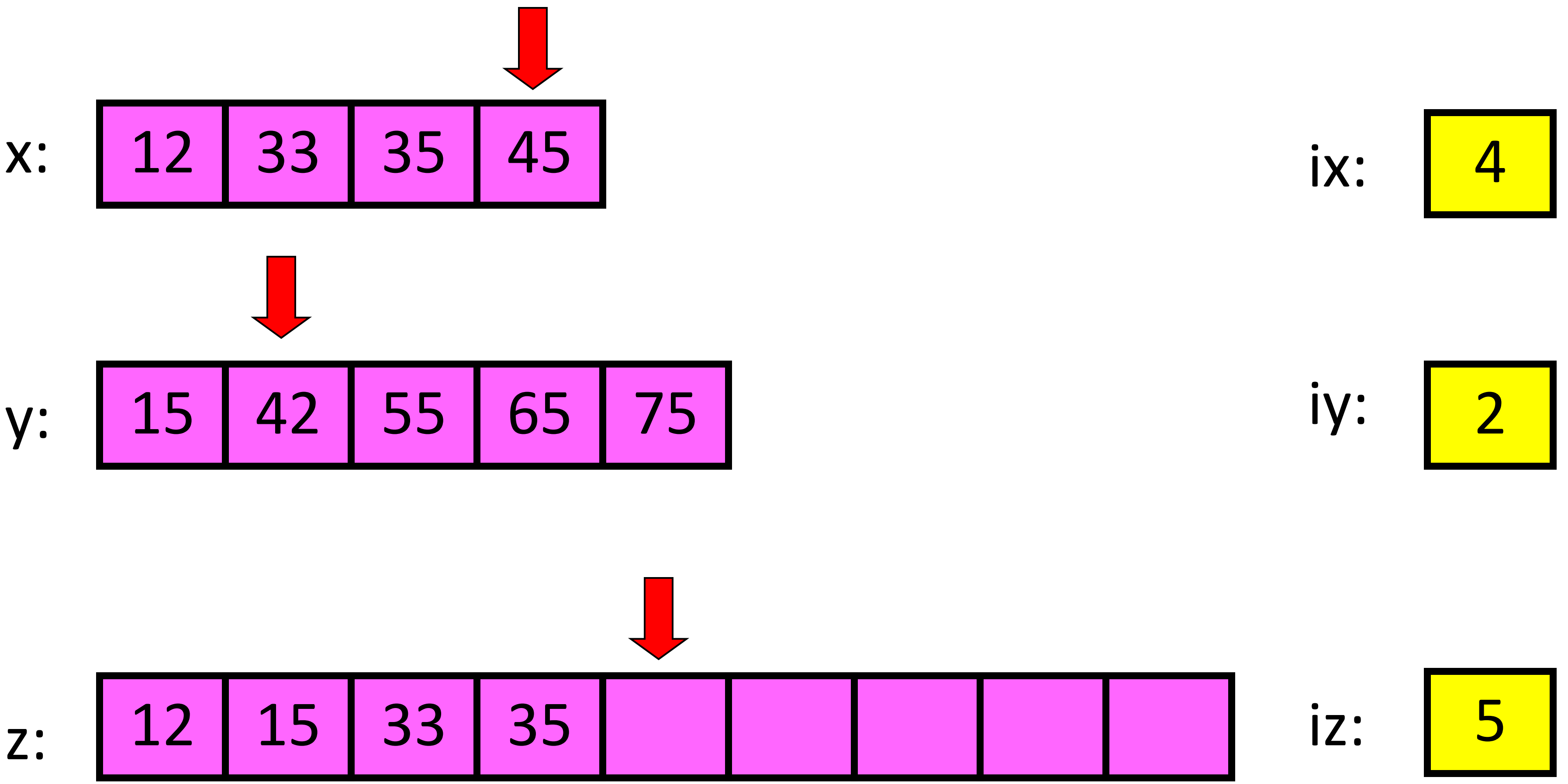
ix<=4 and iy<=5: x(ix) <= y(iy) ???

Merge



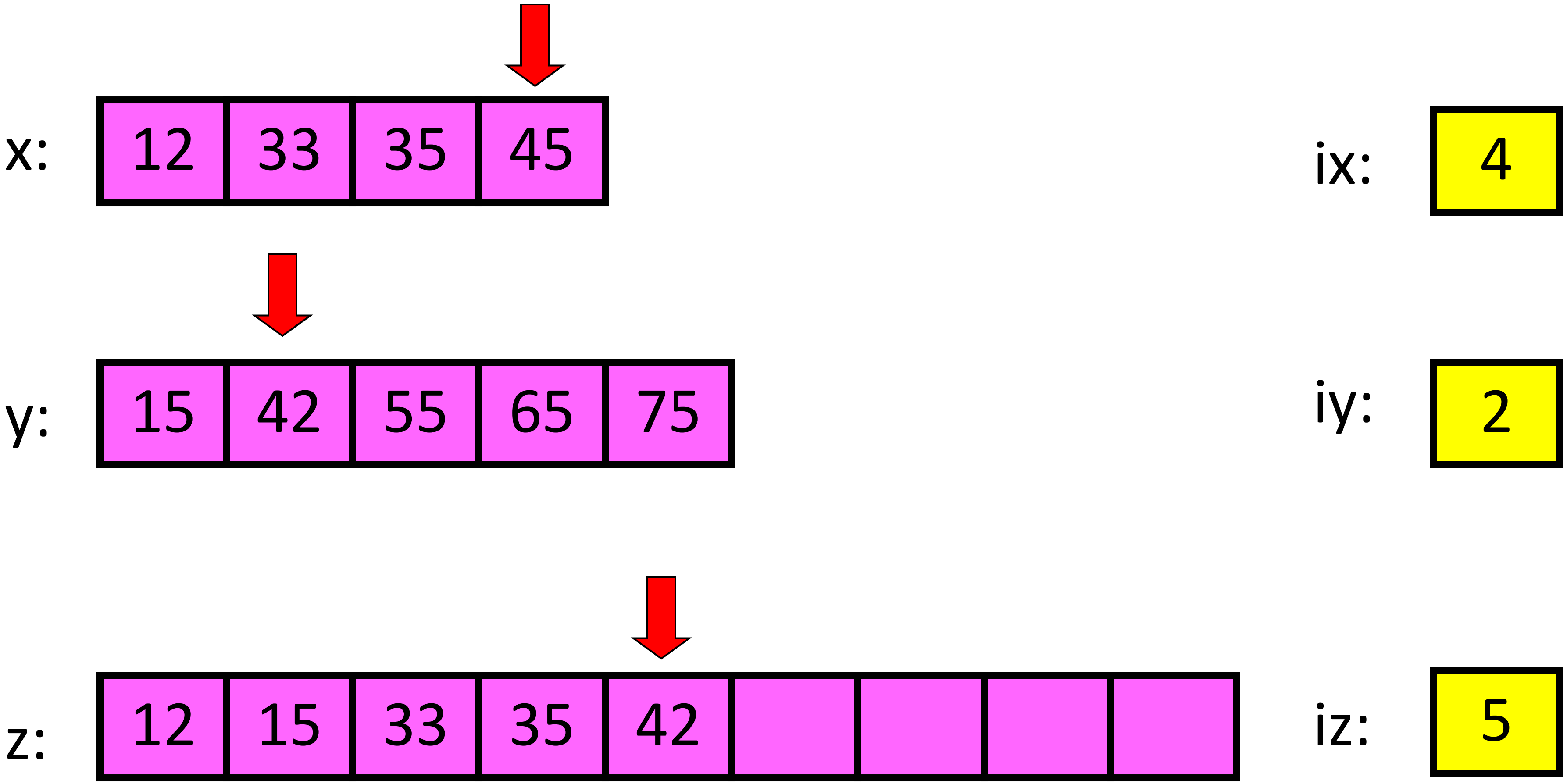
ix<=4 and iy<=5: x(ix) <= y(iy) YES

Merge



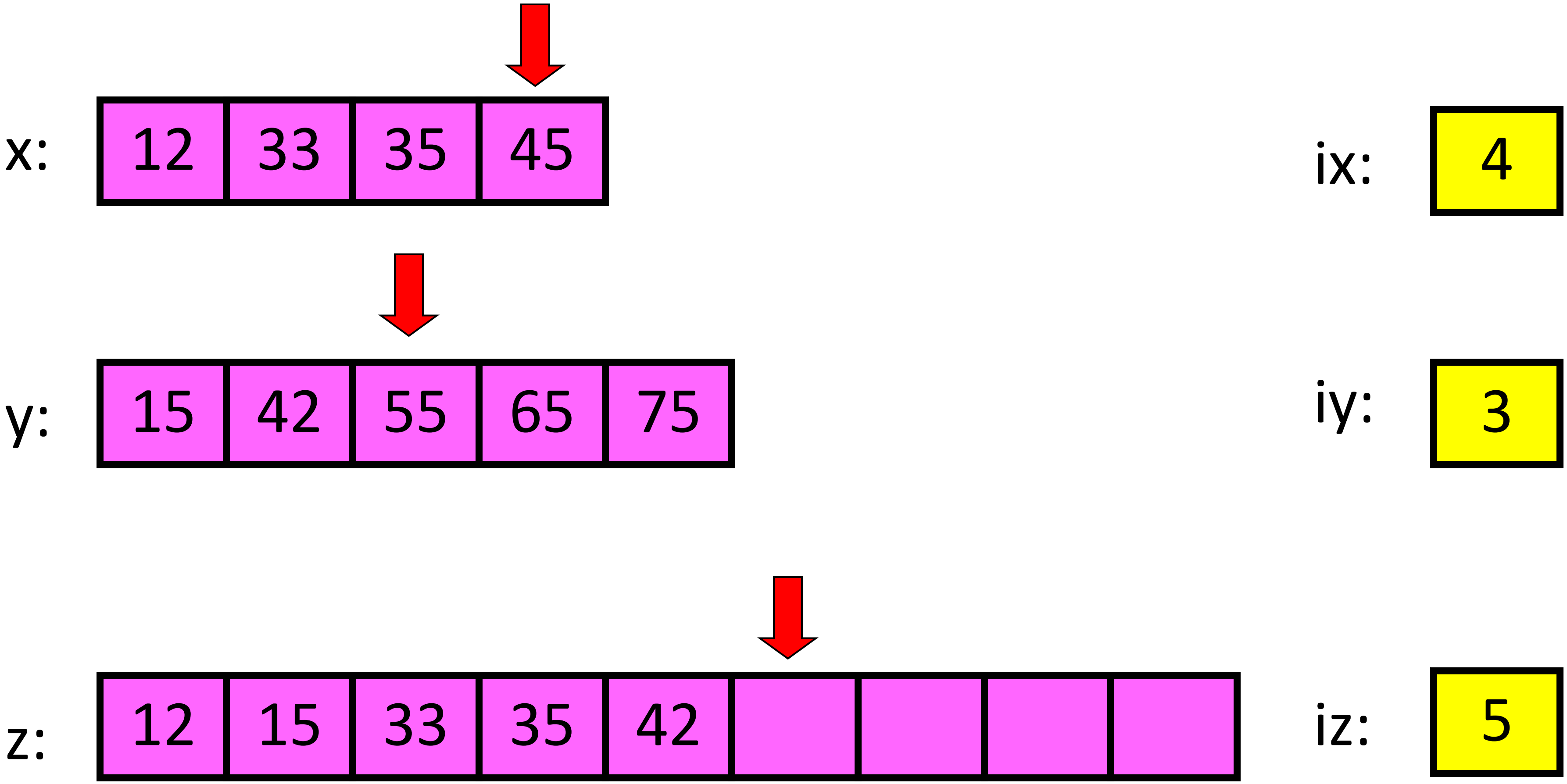
ix<=4 and iy<=5: x(ix) <= y(iy) ???

Merge



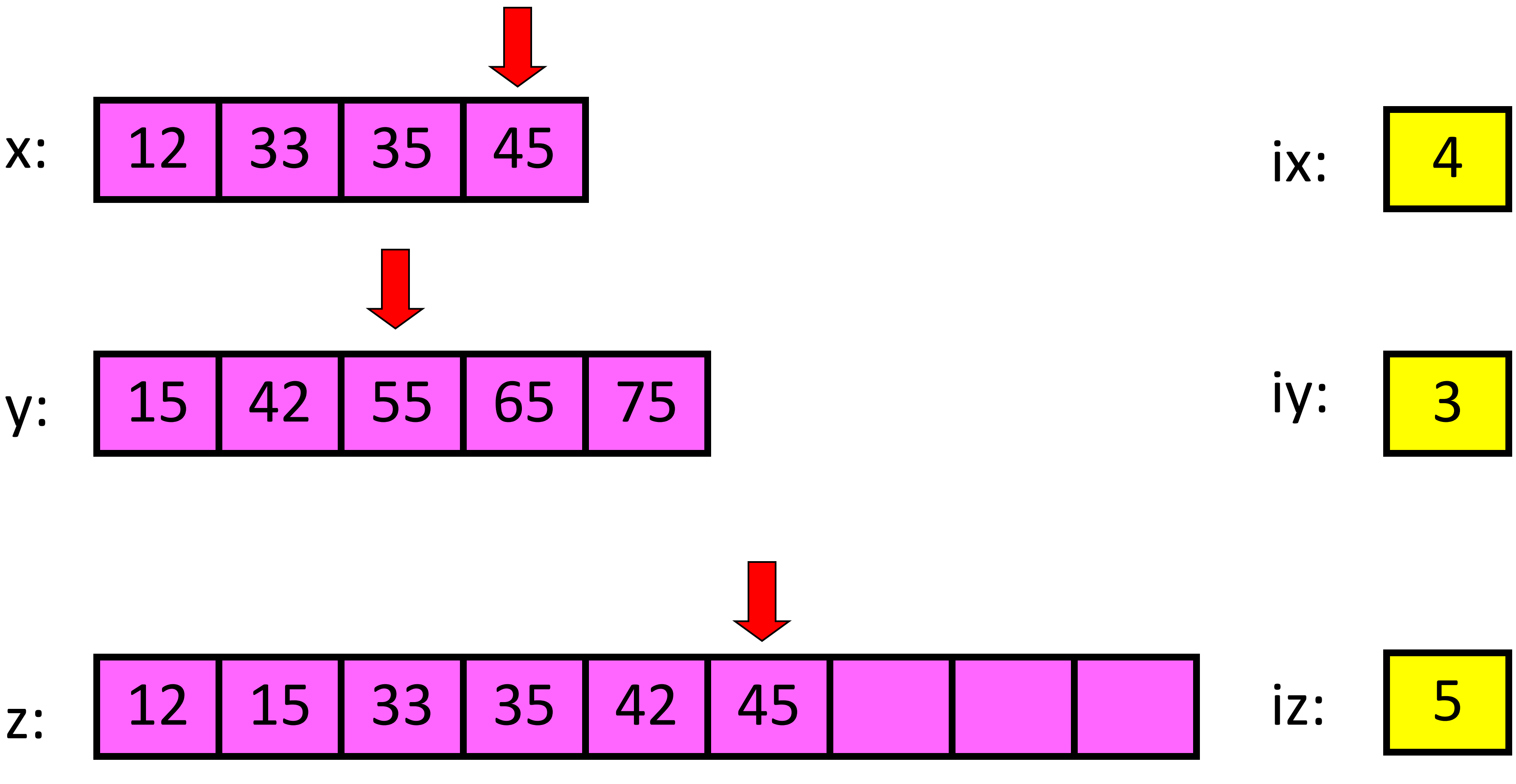
ix<=4 and iy<=5: x(ix) <= y(iy) NO

Merge



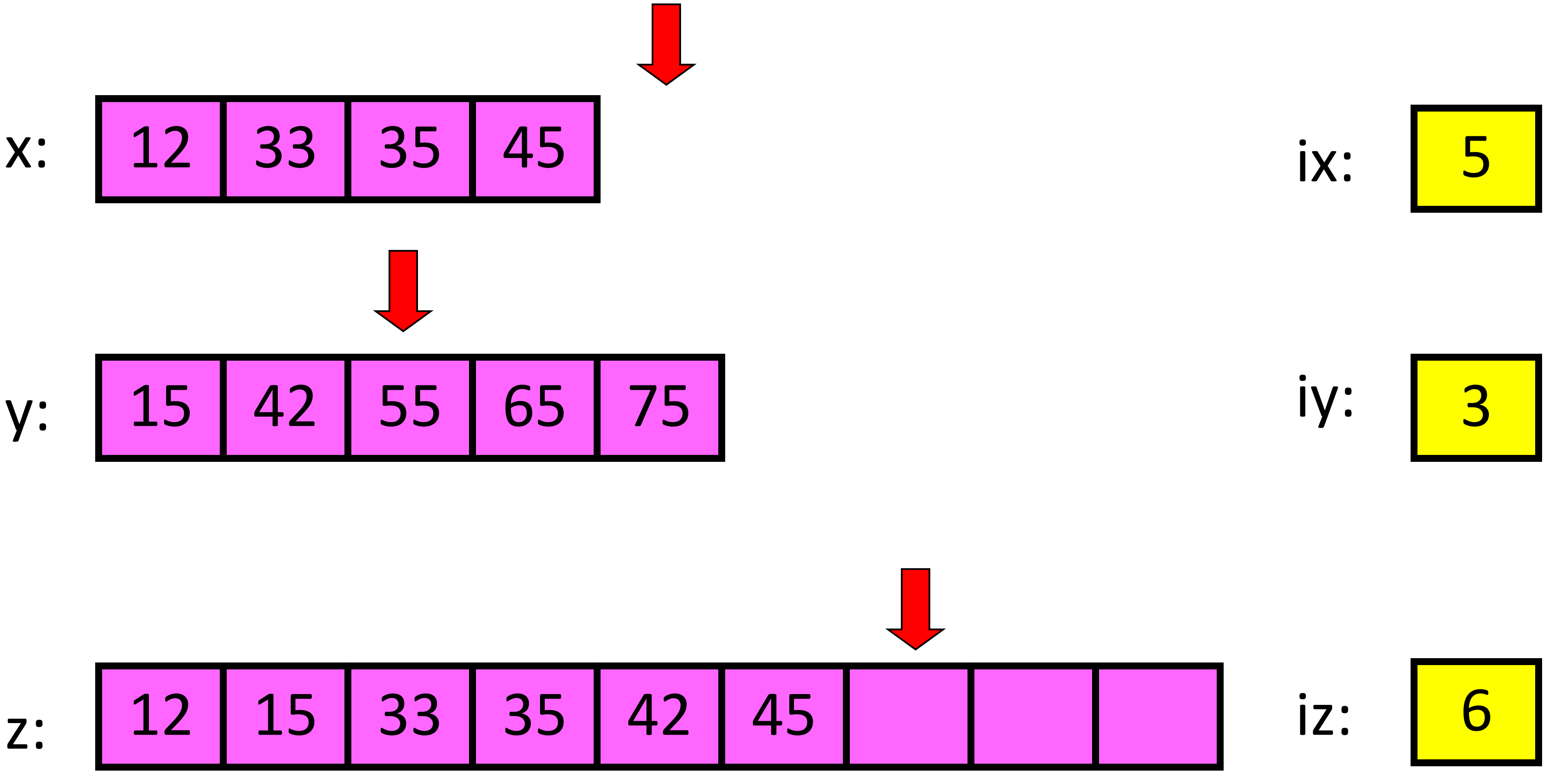
ix<=4 and iy<=5: x(ix) <= y(iy) ???

Merge



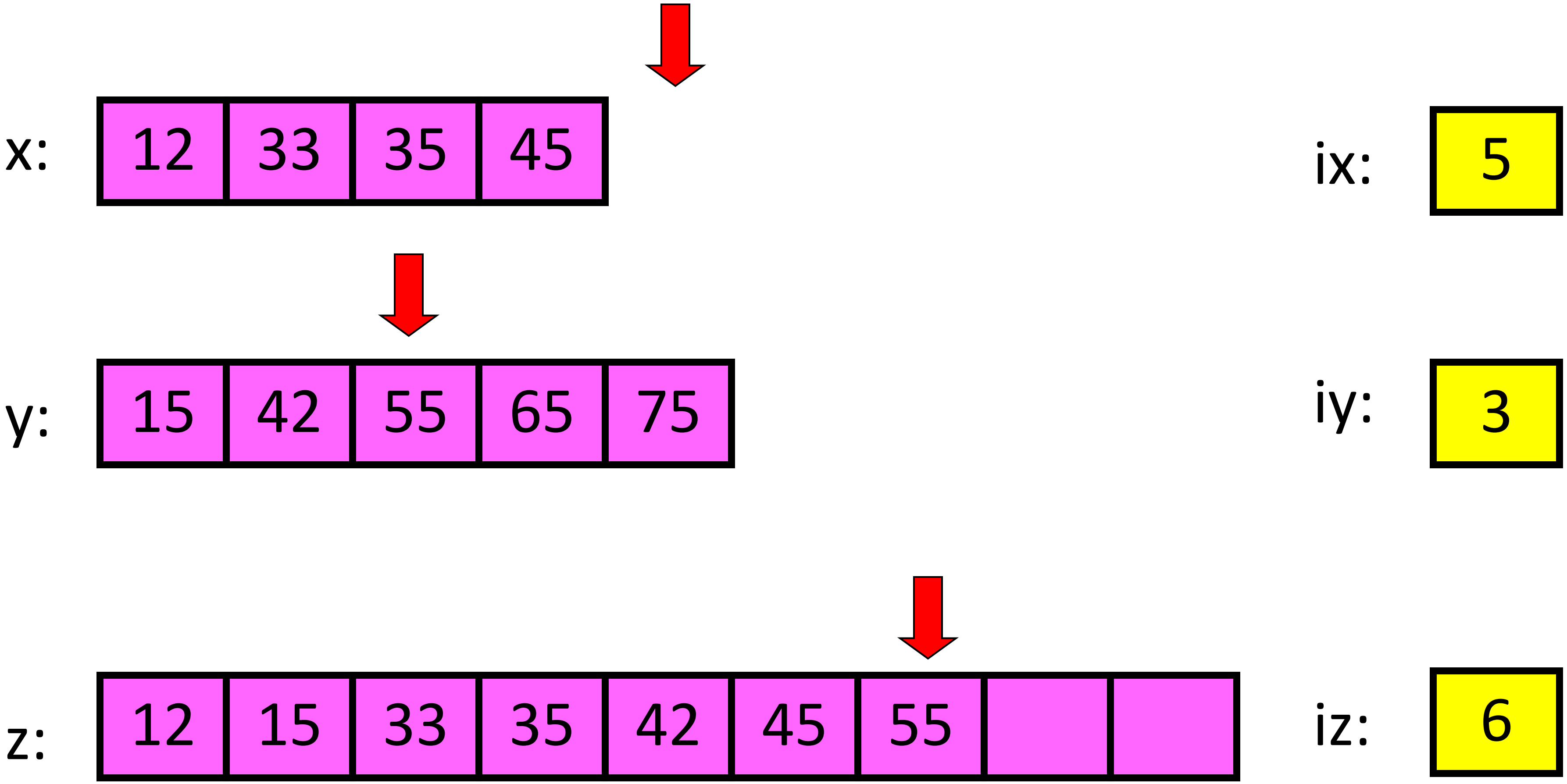
ix<=4 and iy<=5: x(ix) <= y(iy) YES

Merge



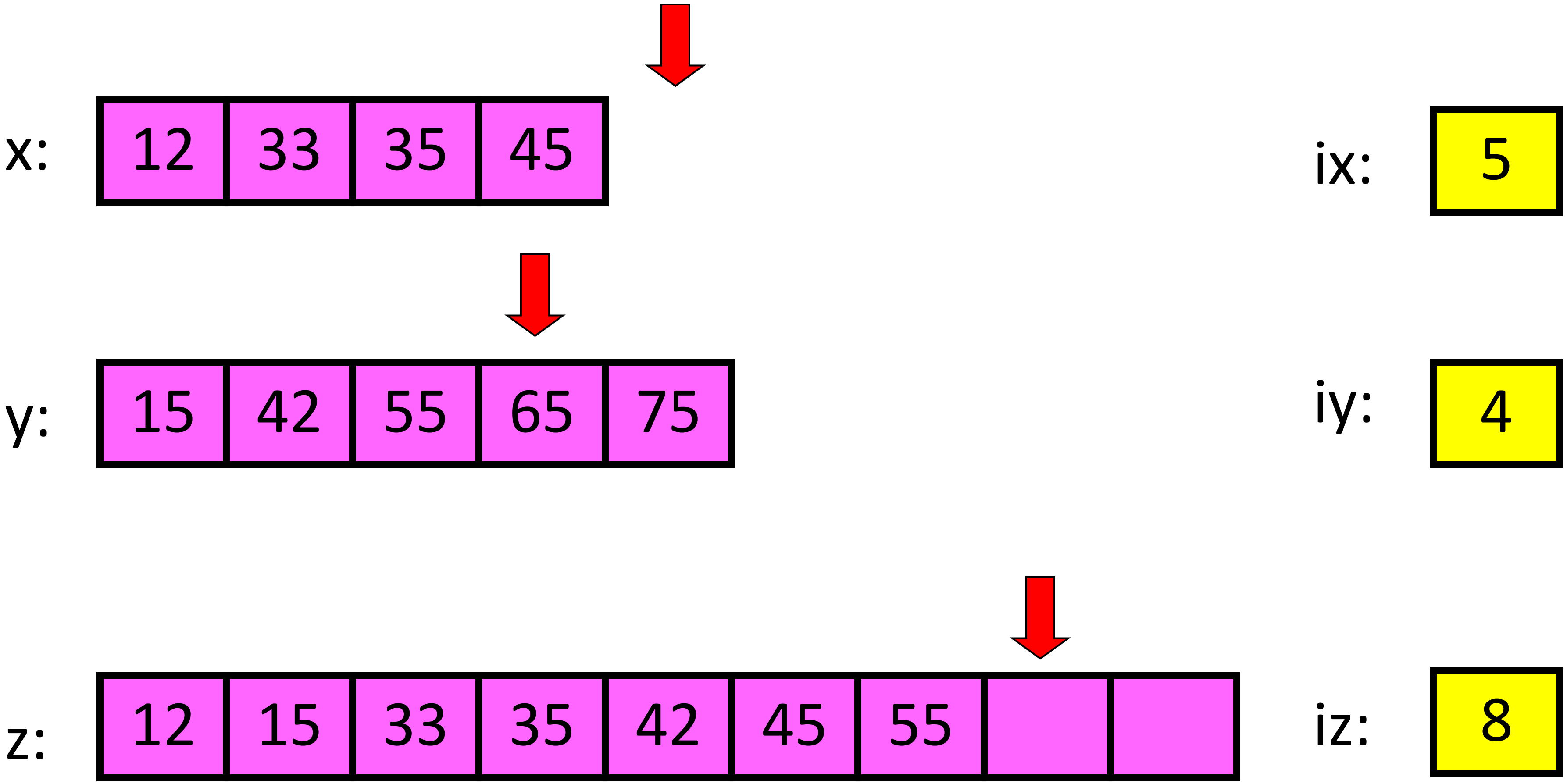
ix > 4

Merge



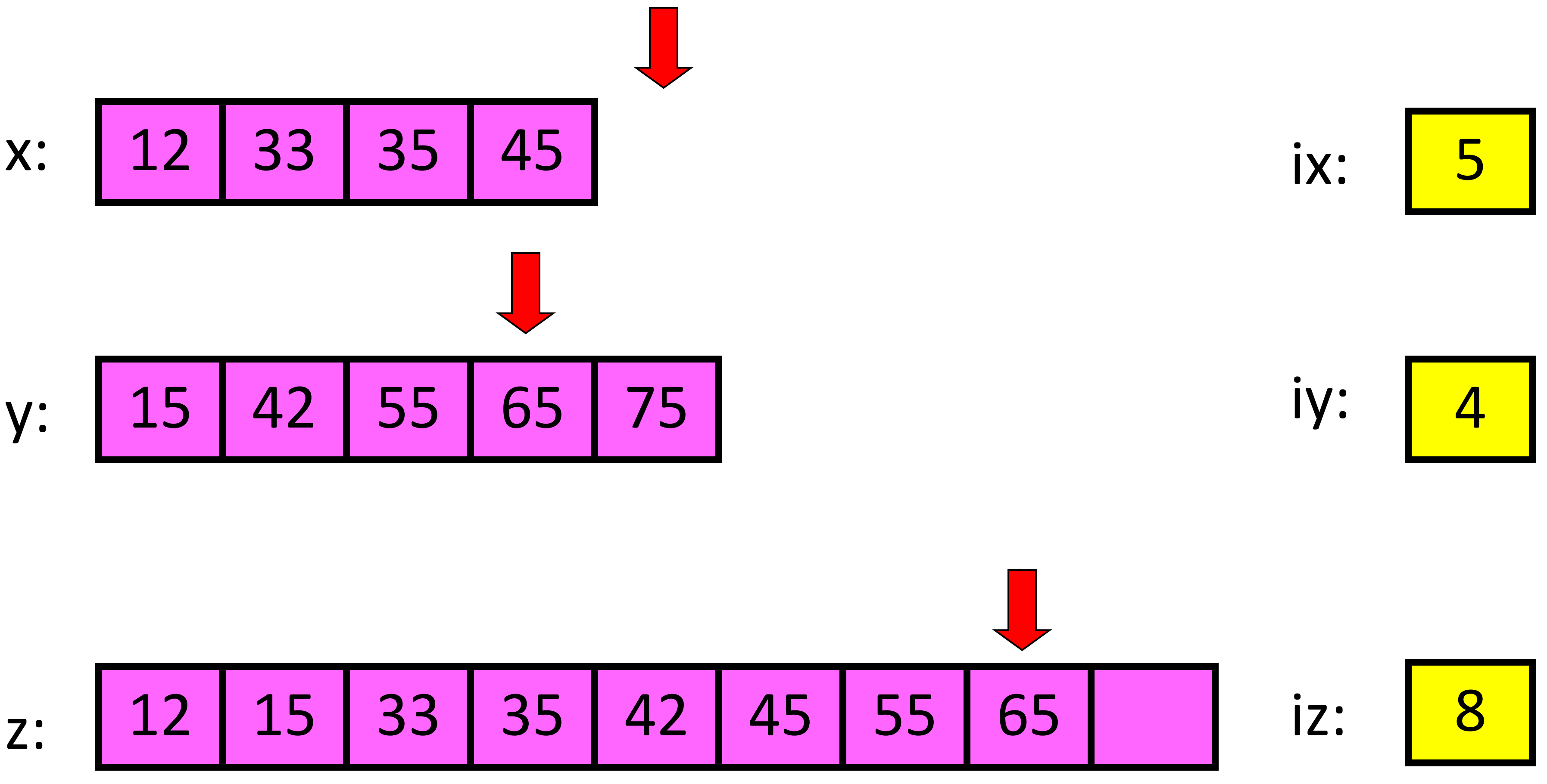
ix > 4: take y(iy)

Merge



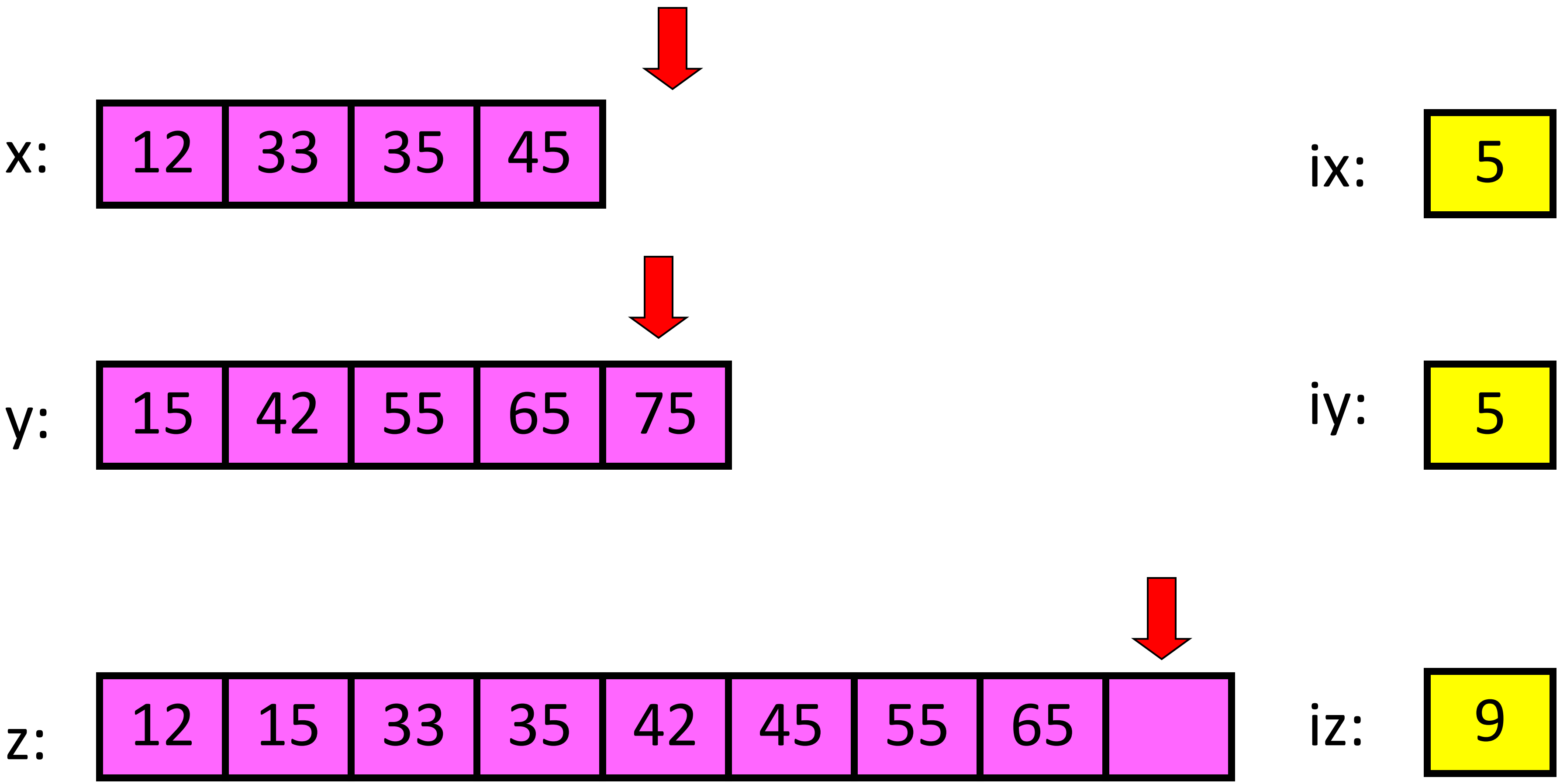
iy <= 5

Merge



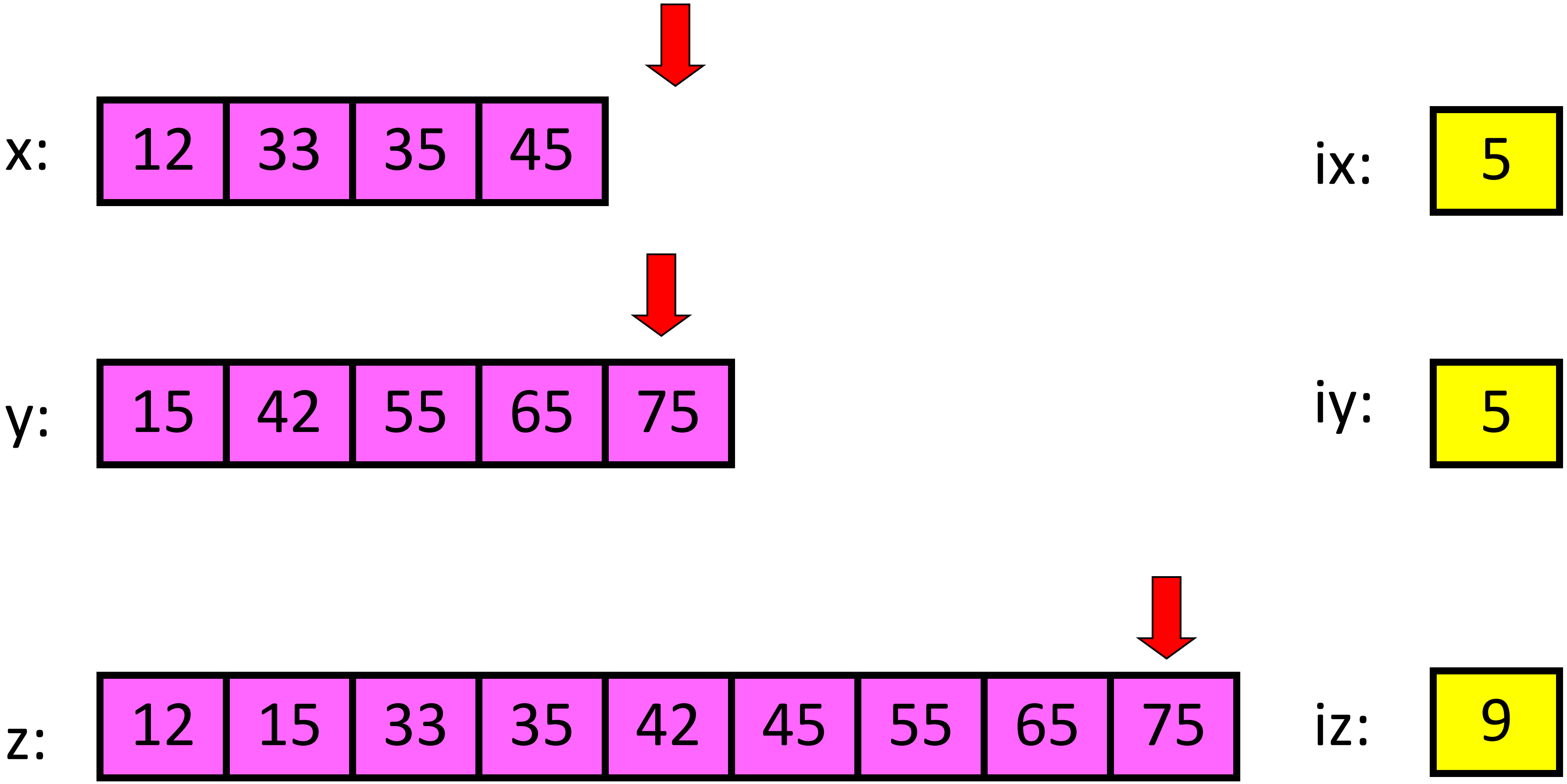
iy <= 5

Merge



iy <= 5

Merge



iy <= 5

```
function z = merge(x,y)
nx = length(x); ny = length(y);
z = zeros(1,nx+ny);
ix = 1; iy = 1; iz = 1;
```

```
function z = merge(x,y)
nx = length(x); ny = length(y);
z = zeros(1, nx+ny);
ix = 1; iy = 1; iz = 1;
while ix<=nx && iy<=ny




end
# Deal with remaining values in x or y
```

```
function z = merge(x,y)
nx = length(x); ny = length(y);
z = zeros(1, nx+ny);
ix = 1; iy = 1; iz = 1;
while ix<=nx && iy<=ny
    if  x(ix) <= y(iy)
        z(iz)= x(ix);  ix=ix+1;  iz=iz+1;
    else
        z(iz)= y(iy);  iy=iy+1;  iz=iz+1;
    end
end
# Deal with remaining values in x or y
```

```
function z = merge(x,y)
nx = length(x); ny = length(y);
z = zeros(1, nx+ny);
ix = 1; iy = 1; iz = 1;
while ix<=nx && iy<=ny
    if  x(ix) <= y(iy)
        z(iz)= x(ix);  ix=ix+1;  iz=iz+1;
    else
        z(iz)= y(iy);  iy=iy+1;  iz=iz+1;
    end
end
while ix<=nx  # copy remaining x-values
  z(iz)= x(ix);  ix=ix+1;  iz=iz+1;
end
while iy<=ny  # copy remaining y-values
  z(iz)= y(iy);  iy=iy+1;  iz=iz+1;
end
```

```
function y = mergeSort(x)
# x is a vector.  y is a vector
# consisting of the values in x
# sorted from smallest to largest.

n = length(x);
if n==1
      y = x;
else
      m  = floor(n/2);
      yL = mergeSortL(x(1:m));
      yR = mergeSortR(x(m+1:n));
      y  = merge(yL,yR);
end
```

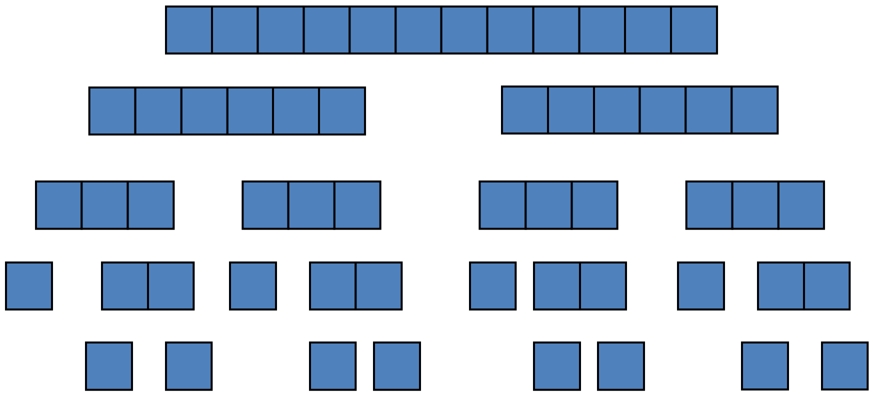
```
function y = mergeSortL(x)
# x is a vector.  y is a vector
# consisting of the values in x
# sorted from smallest to largest.

n = length(x);
if n==1
      y = x;
else
      m  = floor(n/2);
      yL = mergeSortL_L(x(1:m));
      yR = mergeSortL_R(x(m+1:n));
      y  = merge(yL,yR);
end
```

```
function y = mergeSortL_L(x)
# x is a vector.  y is a vector
# consisting of the values in x
# sorted from smallest to largest.

n = length(x);
if n==1
      y = x;
else
      m  = floor(n/2);
      yL = mergeSortL_L_L(x(1:m));
      yR = mergeSortL_L_R(x(m+1:n));
      y  = merge(yL,yR);
end
```

There should be just one mergeSort function!

```
function y = mergeSort(x)
# x is a vector.  y is a vector
# consisting of the values in x
# sorted from smallest to largest.

n = length(x);
if n==1
      y = x;
else
      m  = floor(n/2);
      yL = mergeSort(x(1:m));
      yR = mergeSort(x(m+1:n));
      y  = merge(yL,yR);
end
```

```
function y=mergeSort(x)
n=length(x);
if n==1
      y=x;
else
      m=floor(n/2);
      yL=mergeSort(x(1:m));
      yR=mergeSort(x(m+1:n));
      y=merge(yL,yR);
end
```



Divide-and-Conquer Examples

- Sorting: mergesort and quicksort
- Binary tree traversals
- Binary search
- Multiplication of large integers
- Matrix multiplication: Strassen’s algorithm
- Closest-pair algorithm