Announcements

- Extra office hours today (instead of DIS sections); Zoom links on Canvas
- P6 due tonight at 11pm
- Test 2B feedback and grade estimation on website
- Final exam: Mon, 5/18, 9am. “2.5 hr” take-home, 48 hr submission window
- Optional review session: Sunday, 5/17, 2pm, Zoom (see Canvas)
- Please fill out course evaluation, *worth one <u>BONUS</u> point*, which can be used against any point lost on the final exam (150 points).
- Regular office/consulting hours end today. Study period hours are posted on Canvas and course website.

- Previous Lecture (and exercise):
  - Algorithms for sorting and searching
    - Insertion Sort
    - (Read about Bubble Sort in *Insight*)
    - Linear Search
    - Binary Search
  - Efficiency (complexity) analysis: analyze loops, count number of operations, use timing functions
  - Time efficiency vs. memory efficiency

- Today, Lecture 26:
  - Another “divide and conquer” strategy: **Merge Sort**
  - Review recursion
  - Semester wrap-up

Binary search is efficient, but we need to sort the vector in the first place so that we can use binary search

- Many different algorithms out there...
- We saw insertion sort (and read about bubble sort)
- Let's look at **merge sort**
- Another example of the "divide and conquer" approach (like binary search) but using recursion

Which task fundamentally requires less work: sort a length 1000 array, or merge* two length 500 sorted arrays into one?

A. Sort

B. Merge

C. The same

**Merge two sorted arrays so that the resultant array is sorted (not concatenate two arrays)*

Comparison counting

How many comparisons (between elements) are required to run *insertion sort* on the following vector?

[9, 13, 24, 96, 12, 18, 56]

A. 6

B. 7

C. 12

D. 21

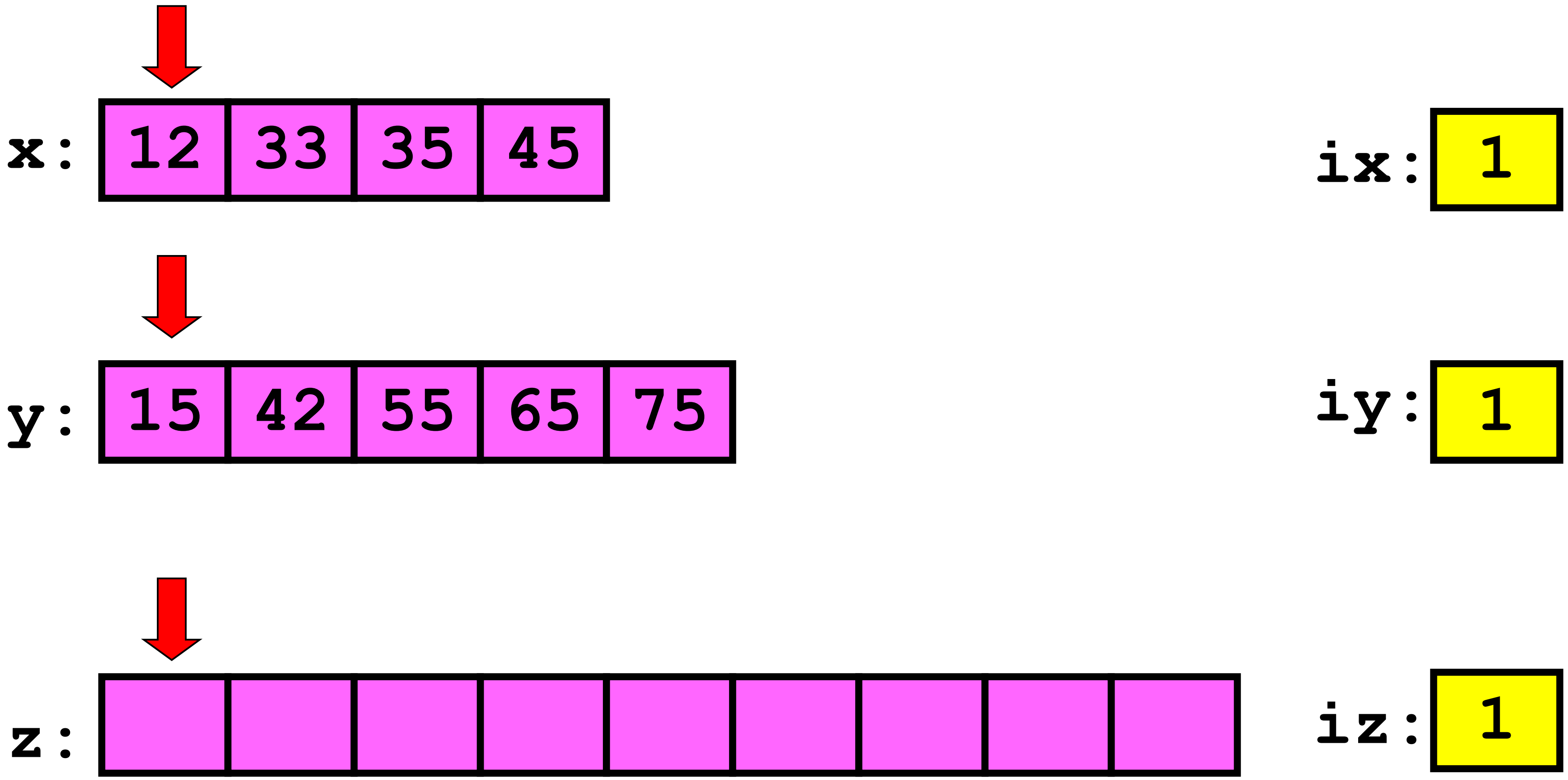
The central sub-problem is the merging of two sorted arrays into one single sorted array

12	33	35	45

15	42	55	65	75

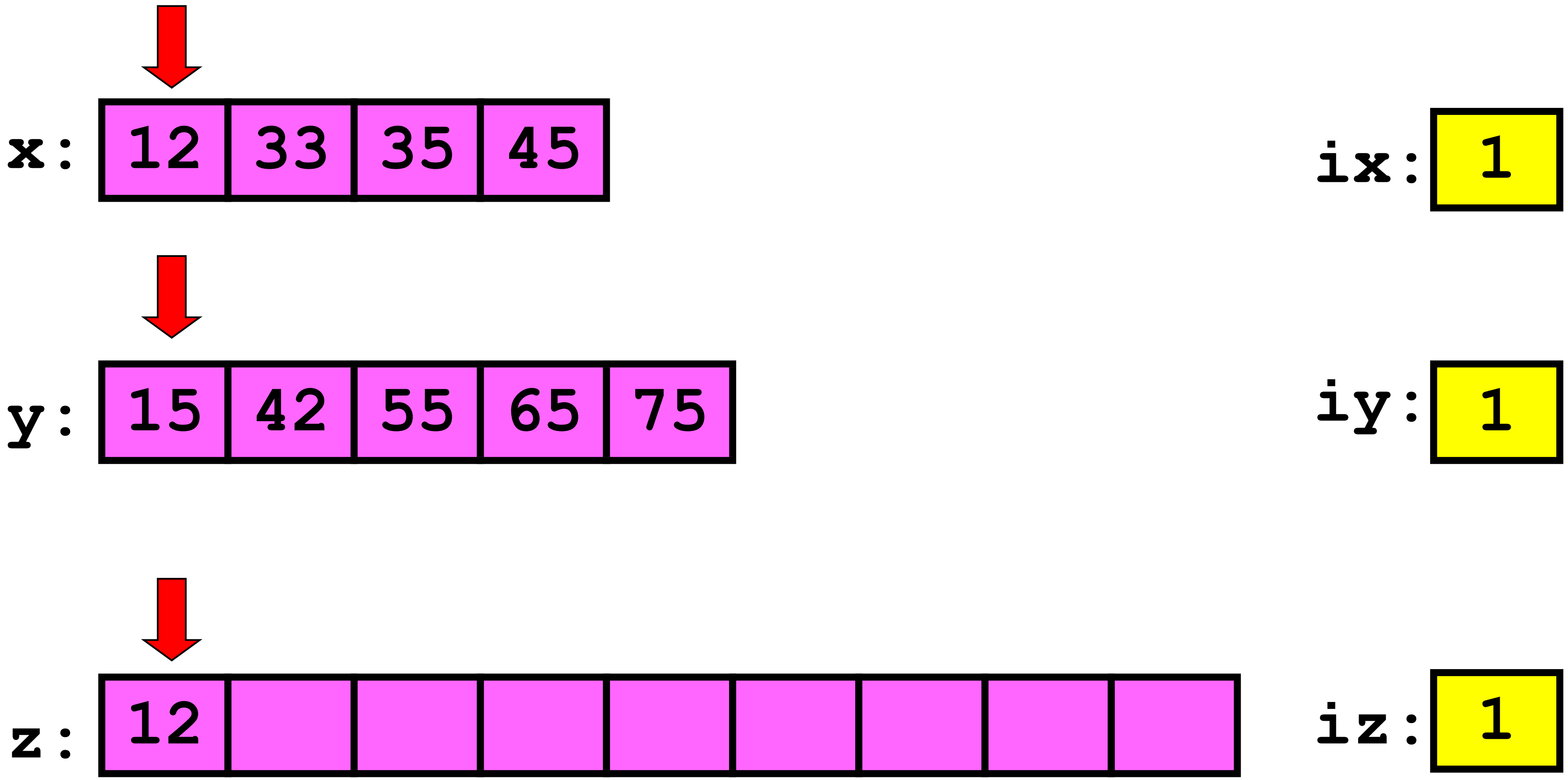
12	15	33	35	42	45	55	65	75

Merge



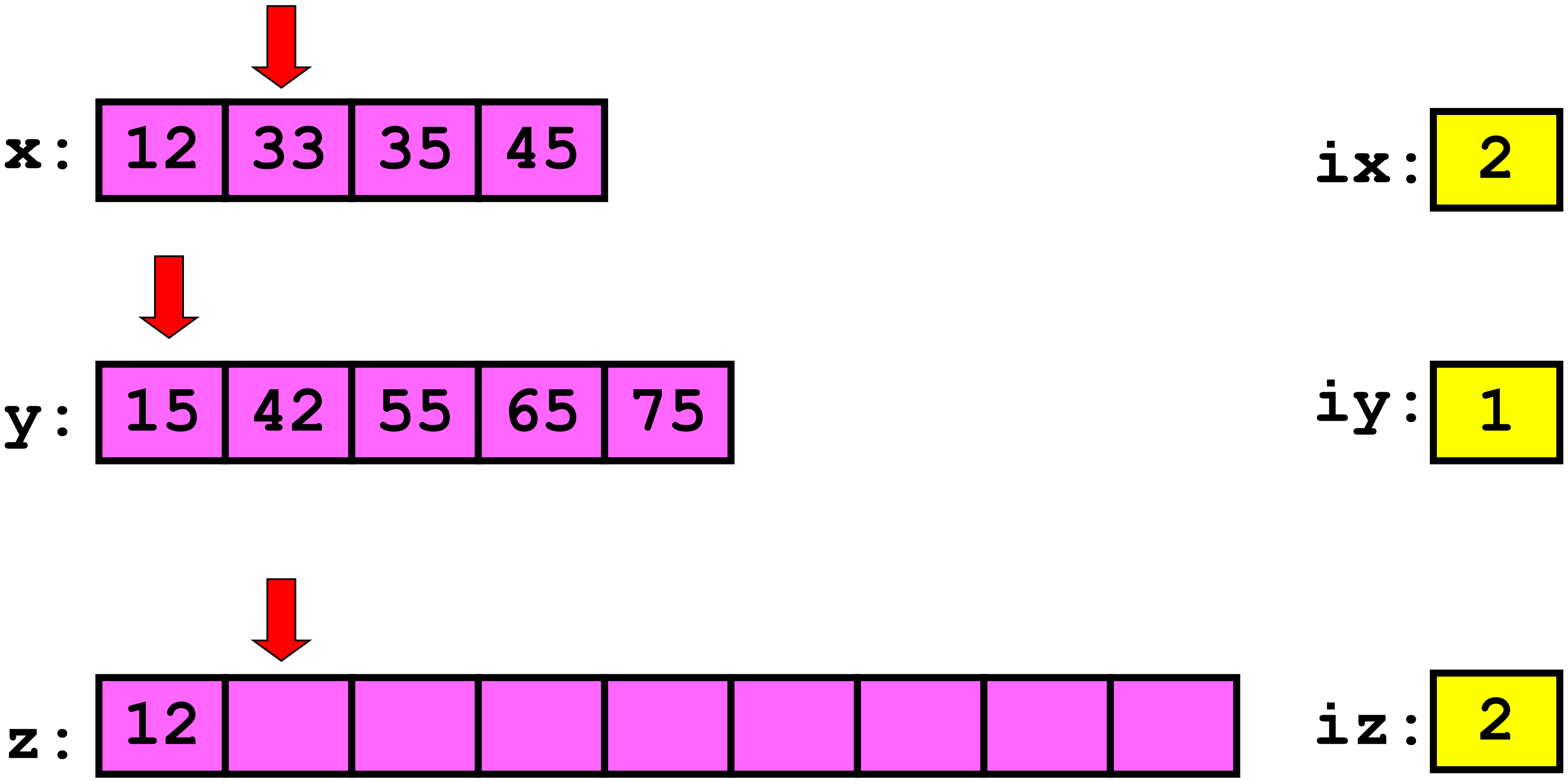

```
ix<=4 and iy<=5:   x(ix) <= y(iy)   ???
```

Merge



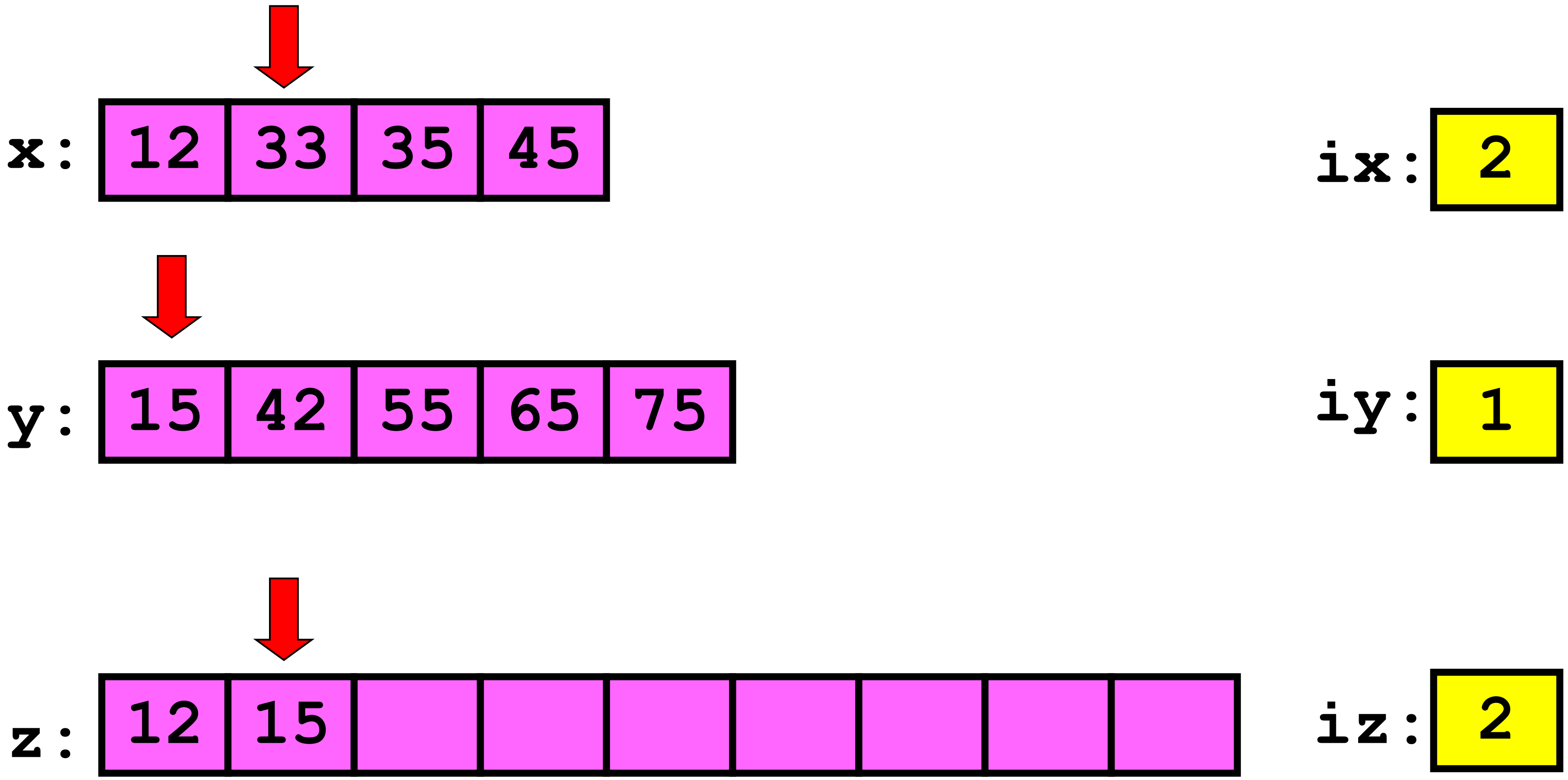
`ix<=4 and iy<=5:   x(ix) <= y(iy)`   **YES**

Merge



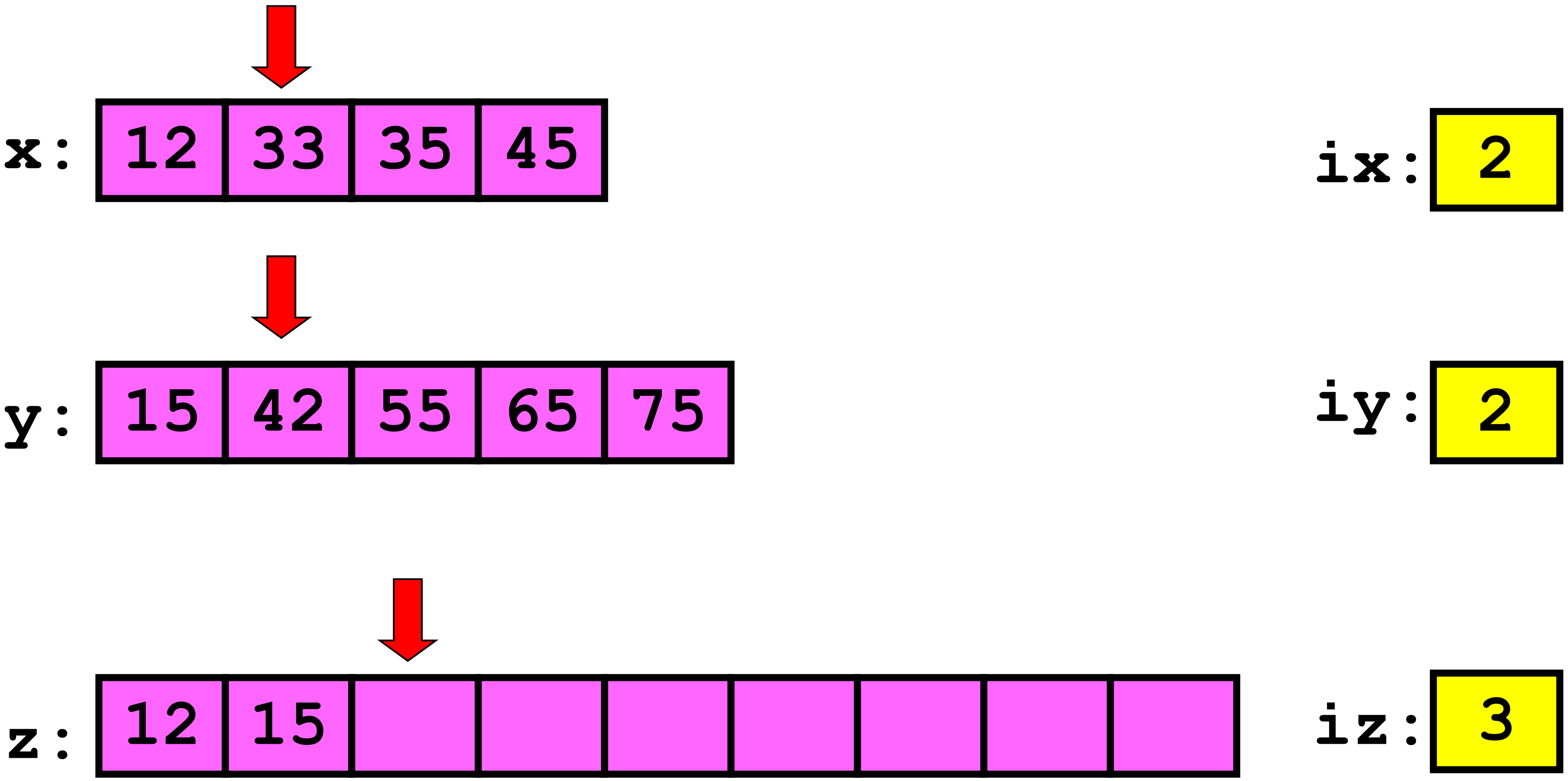
`ix<=4 and iy<=5:   x(ix) <= y(iy)   ???`

Merge



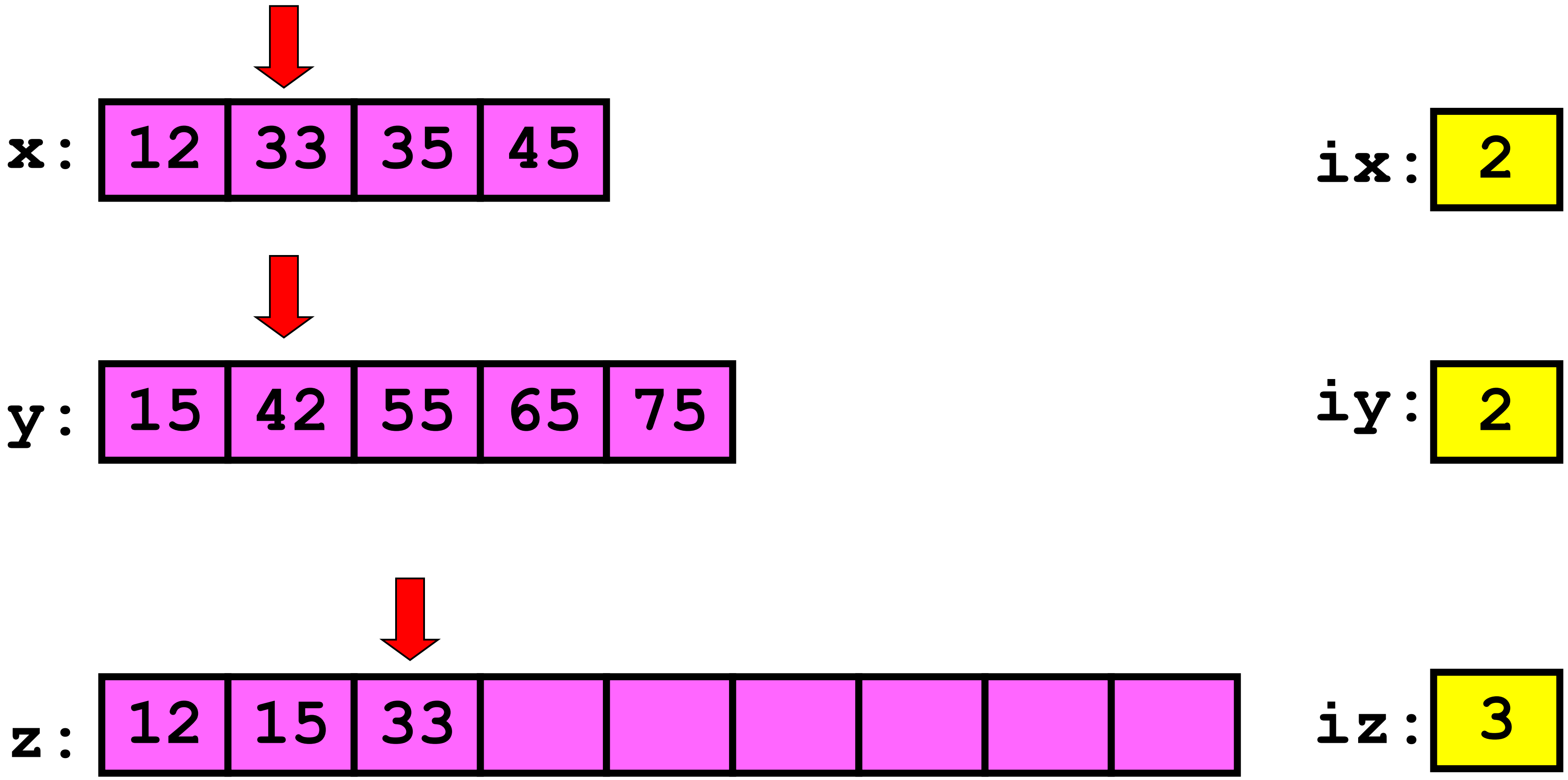
ix<=4 and iy<=5: x(ix) <= y(iy) NO

Merge



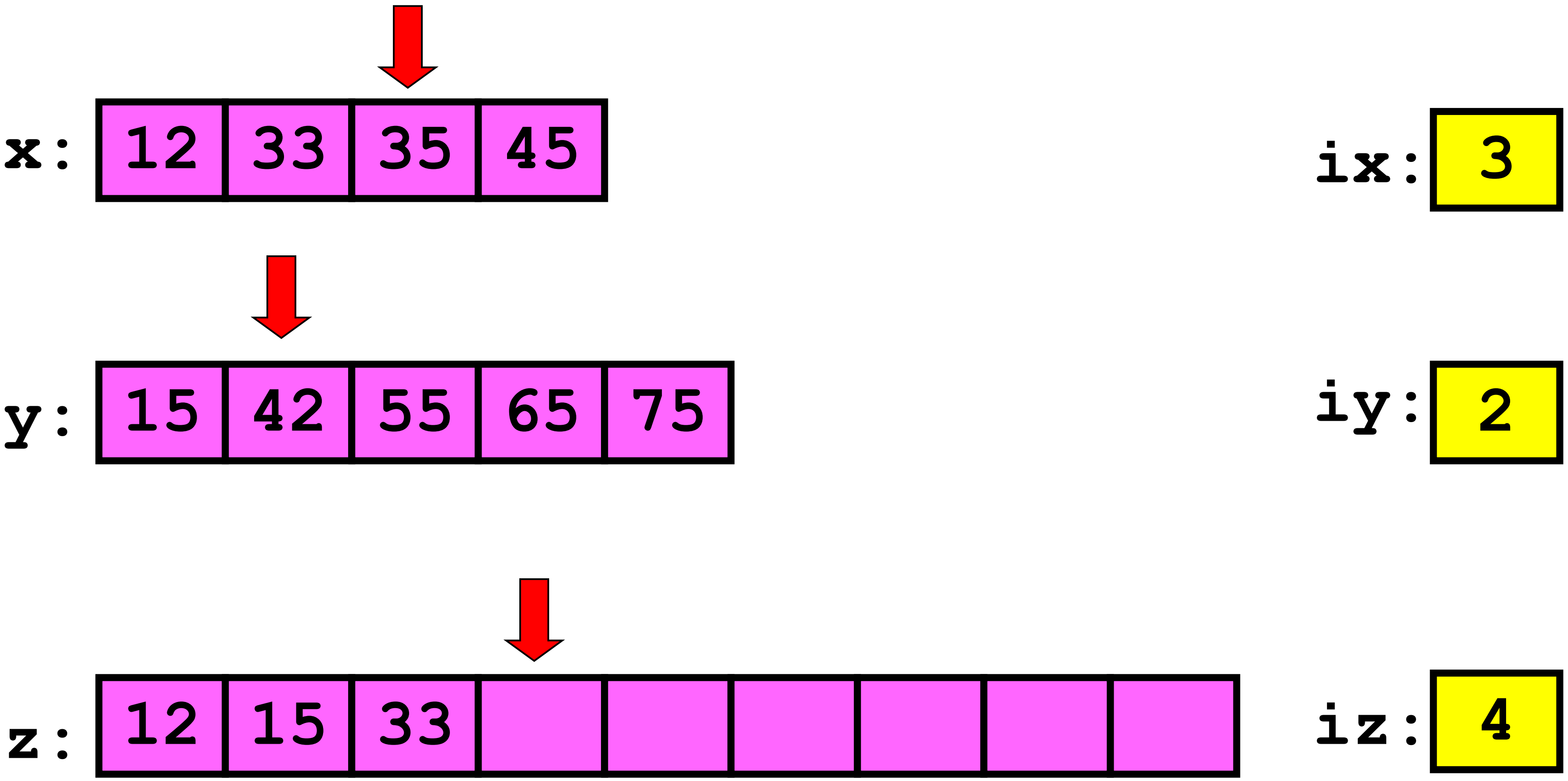
`ix<=4 and iy<=5:   x(ix) <= y(iy)   ???`

Merge



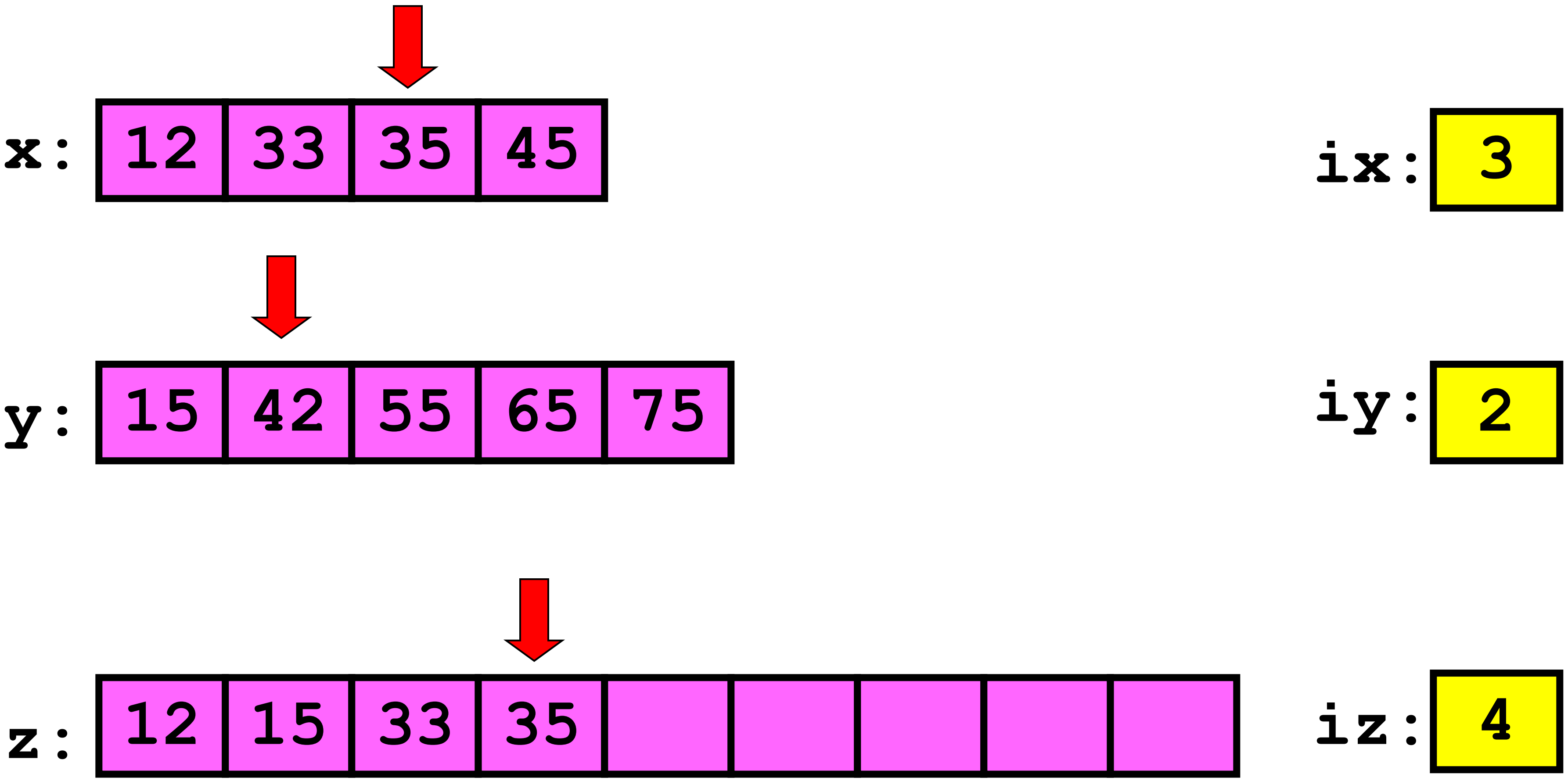
`ix<=4 and iy<=5:   x(ix) <= y(iy)`   **YES**

Merge



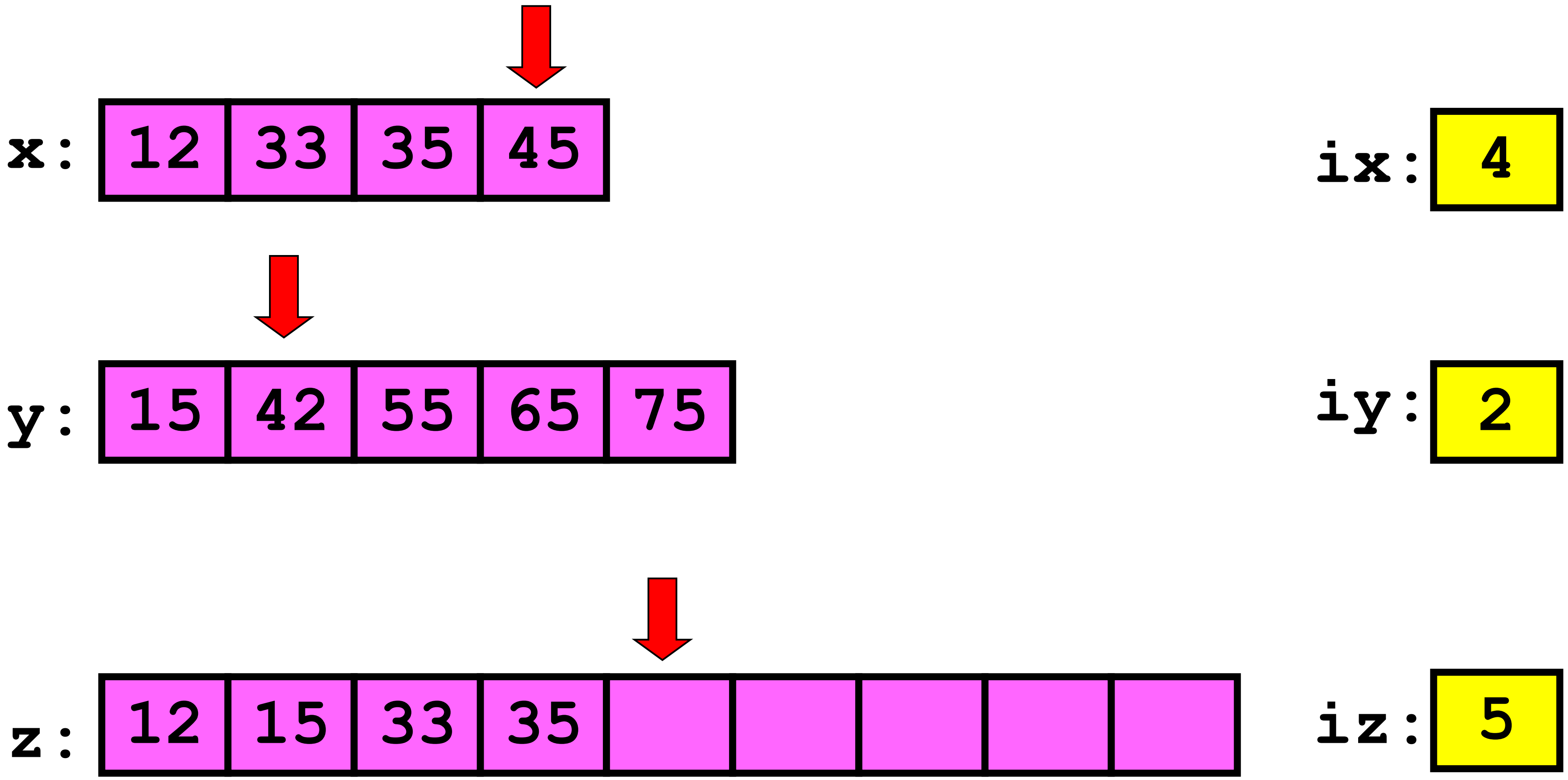
`ix<=4 and iy<=5:   x(ix) <= y(iy)   ???`

Merge



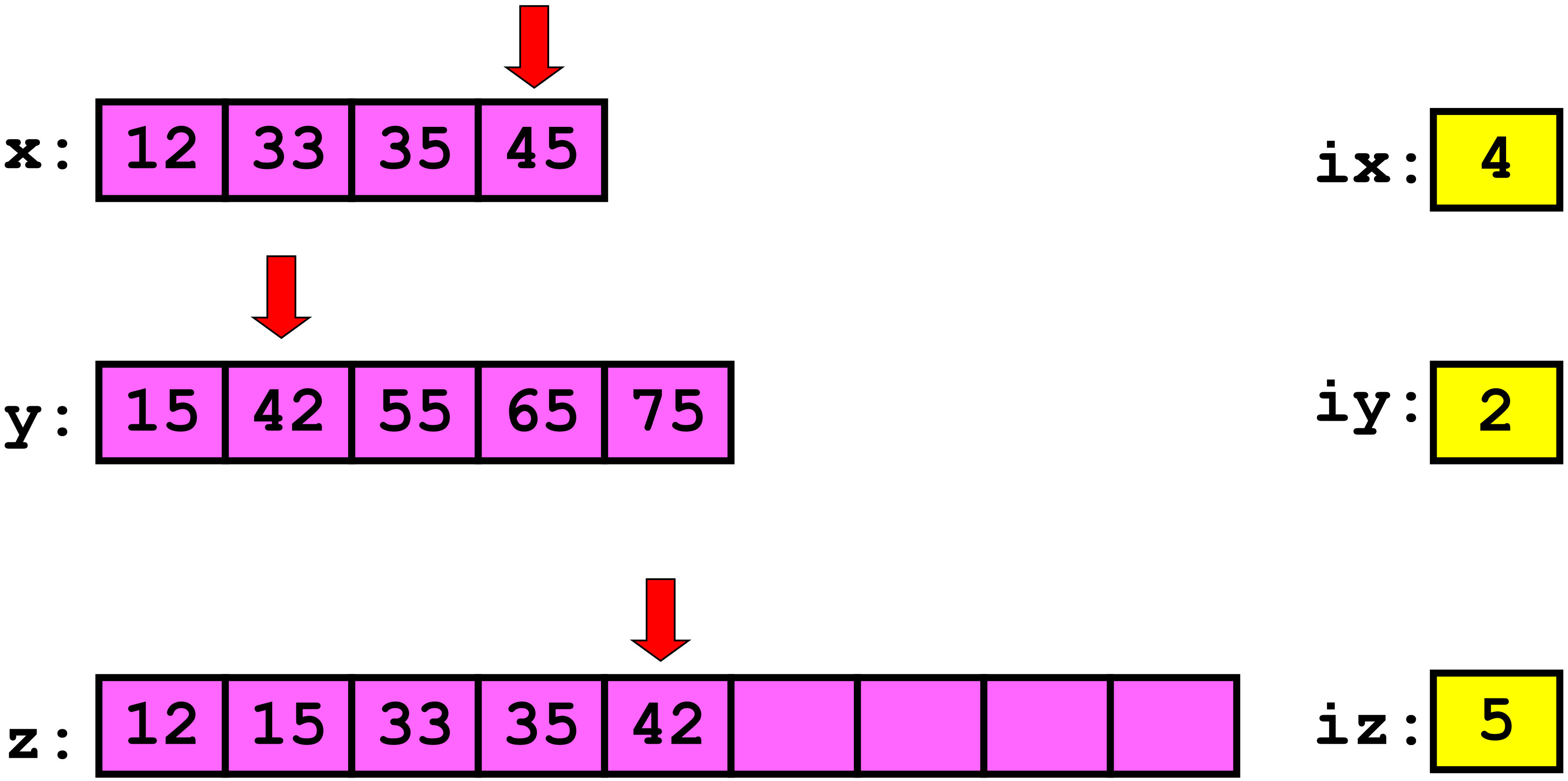
`ix<=4 and iy<=5:   x(ix) <= y(iy)`   **YES**

Merge



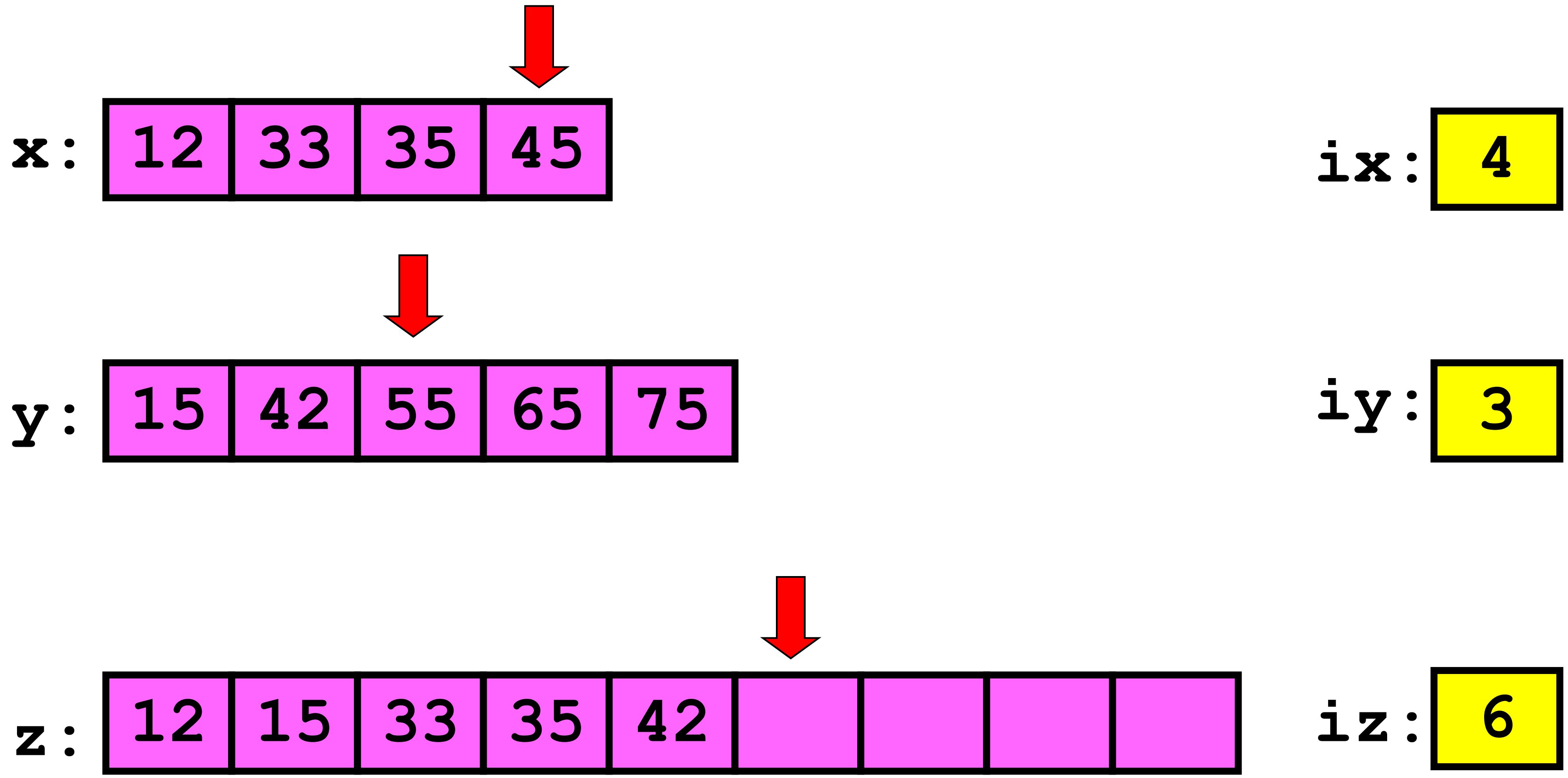
`ix<=4 and iy<=5:   x(ix) <= y(iy)   ???`

Merge



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

Merge




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

Merge

x:

12	33	35	45

ix: 4

y:

15	42	55	65	75

iy: 3

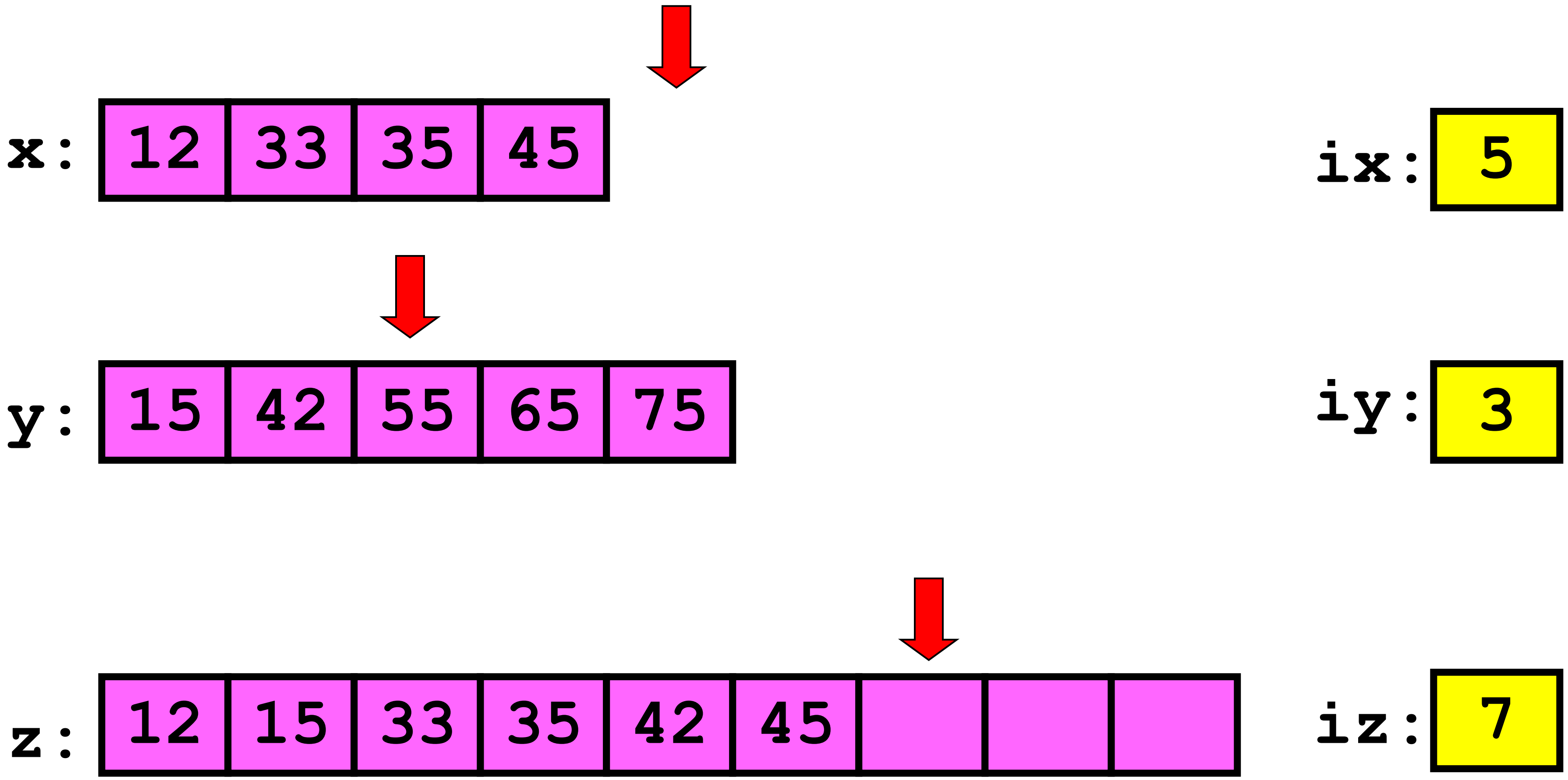
z:

12	15	33	35	42	45			

iz: 6

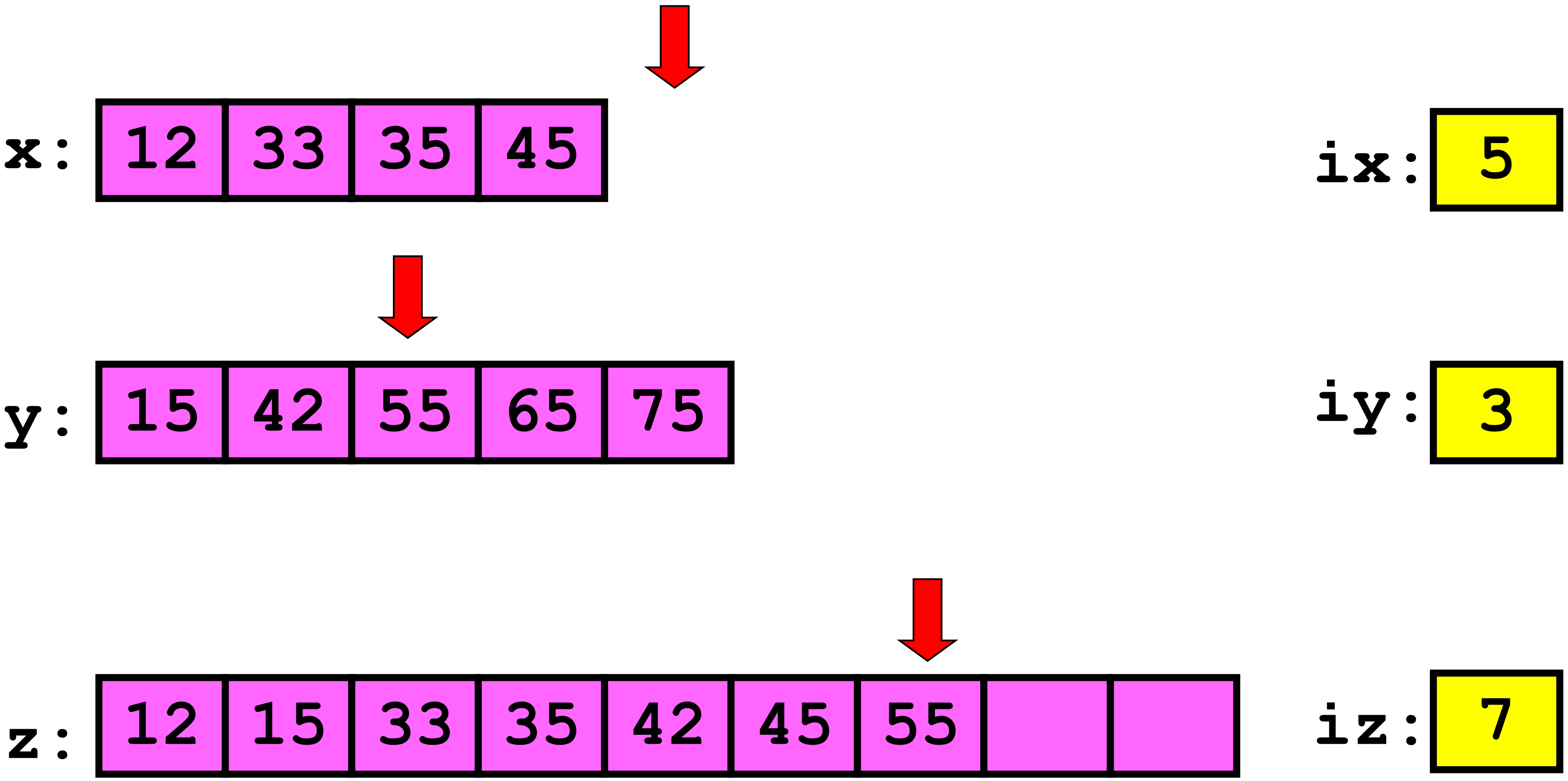
`ix<=4 and iy<=5:   x(ix) <= y(iy)`   **YES**

Merge



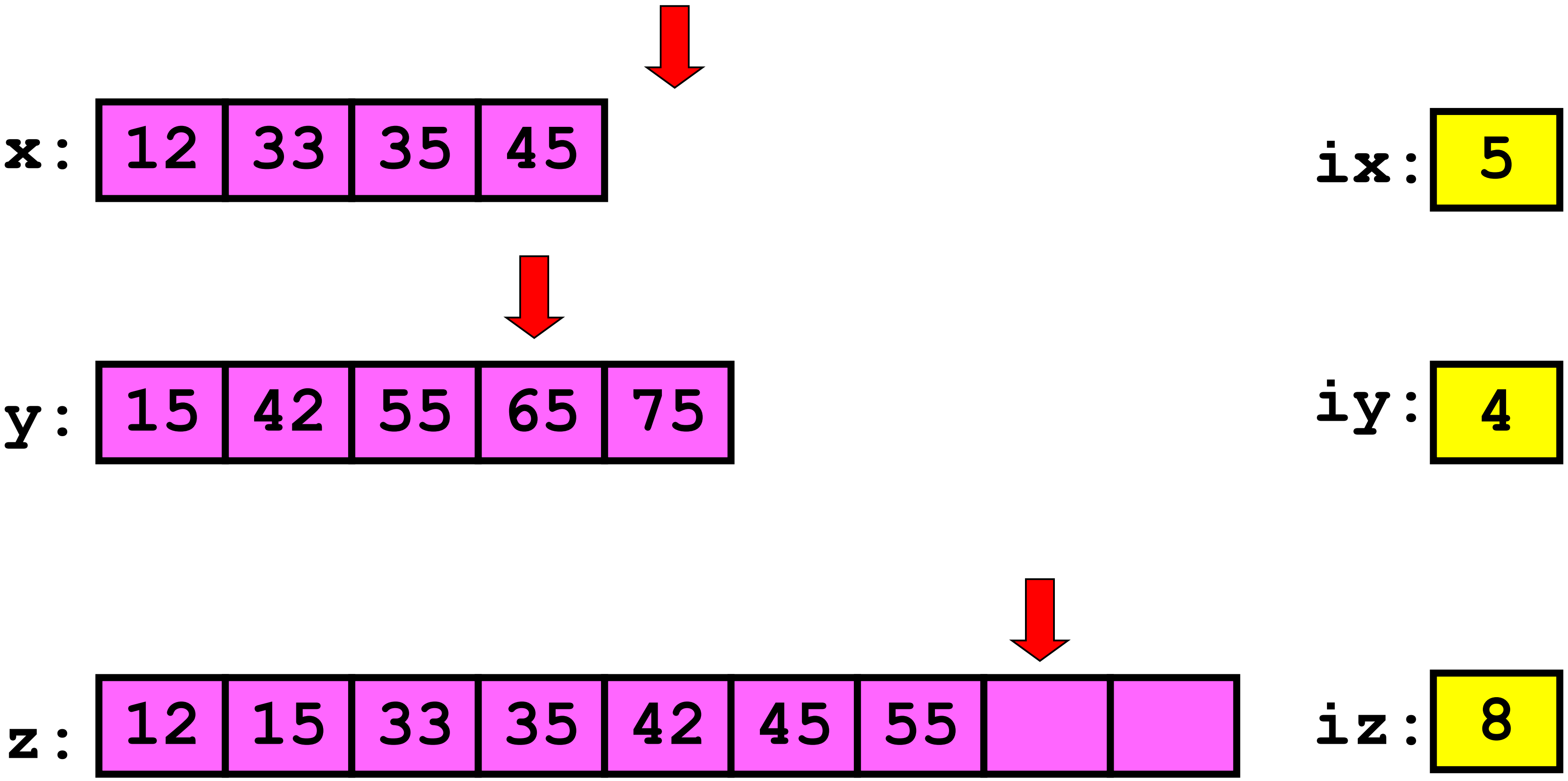
`ix > 4`

Merge



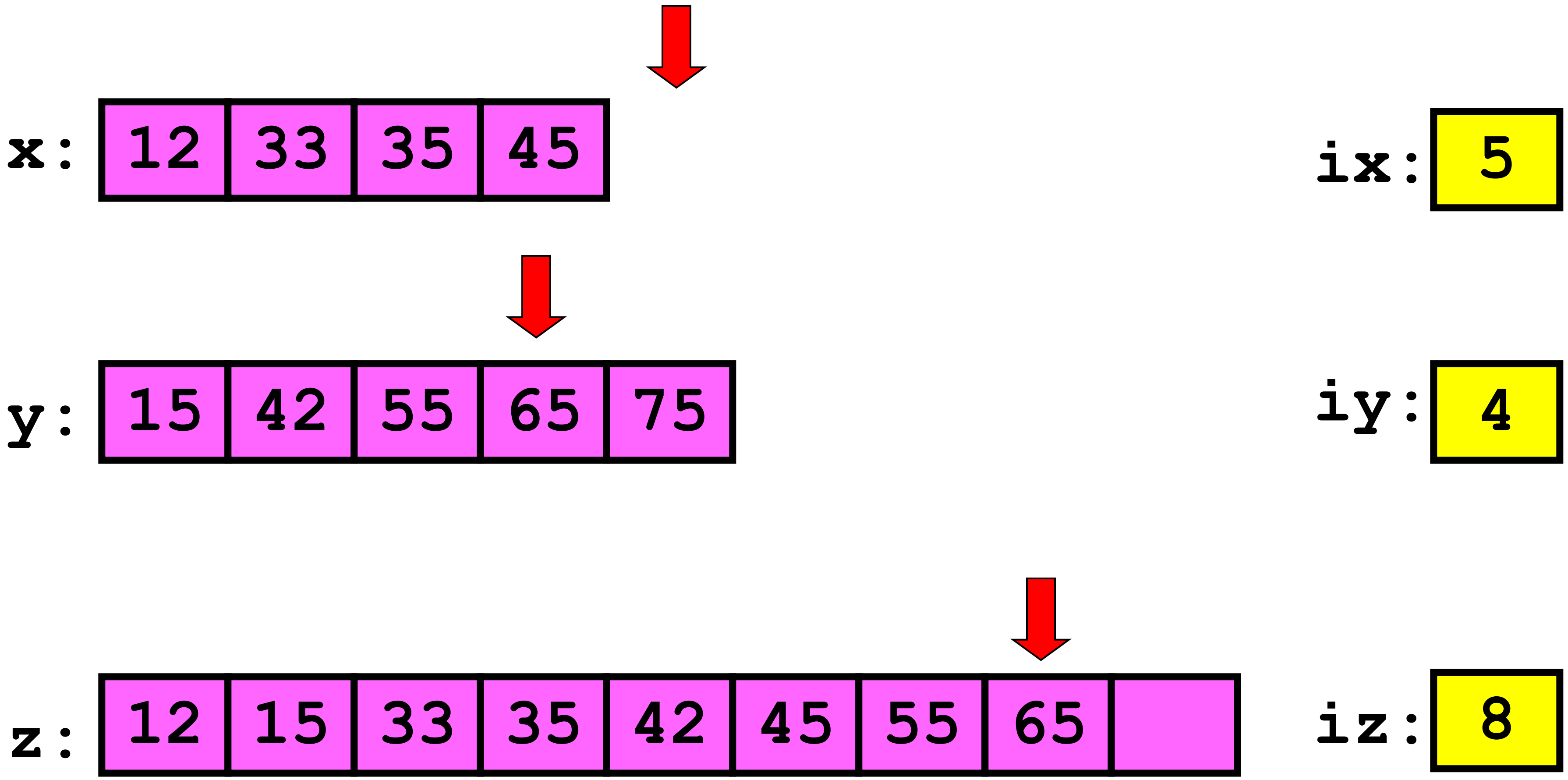
`ix > 4:   take y(iy)`

Merge



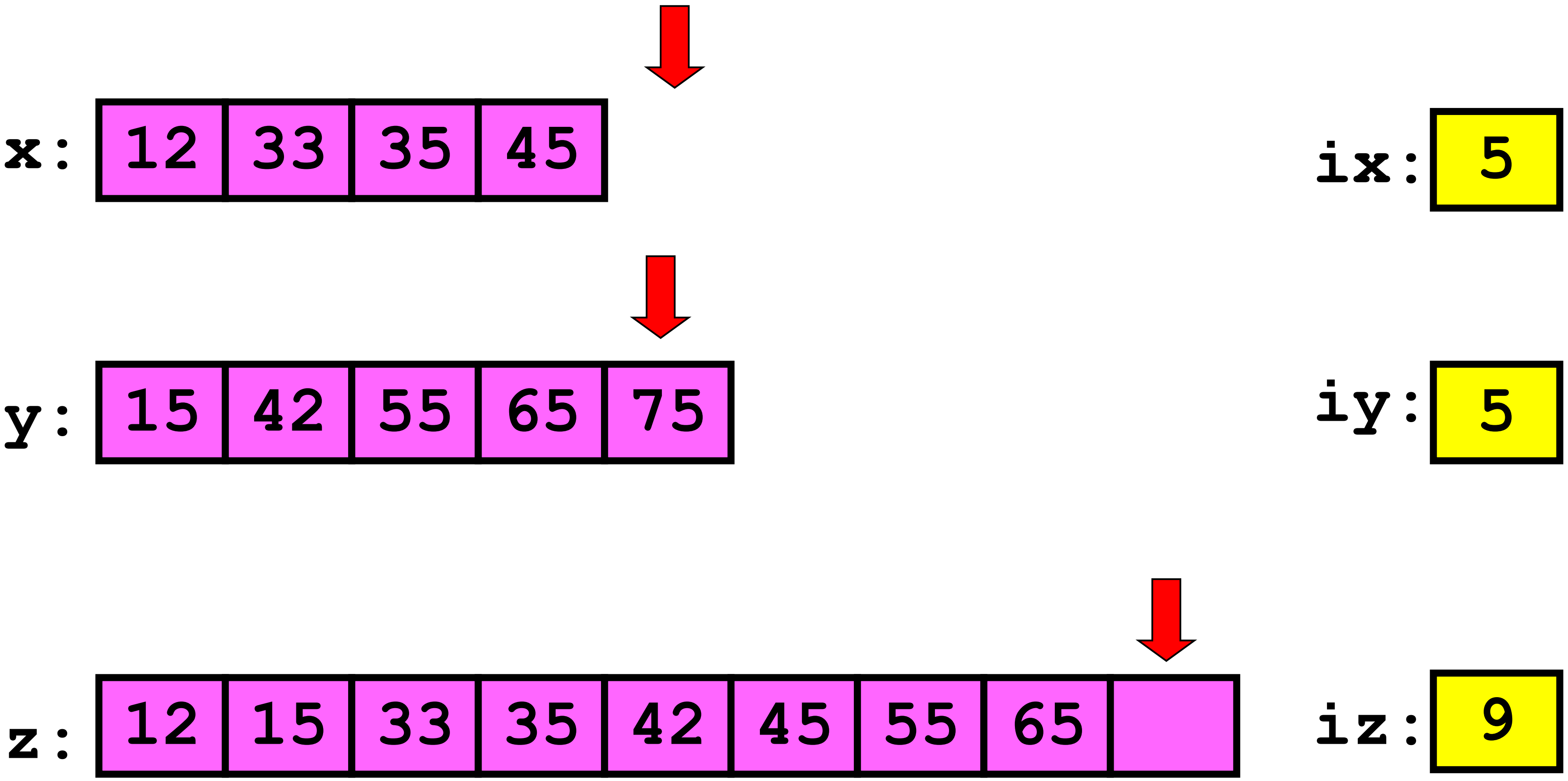
`iy <= 5`

Merge



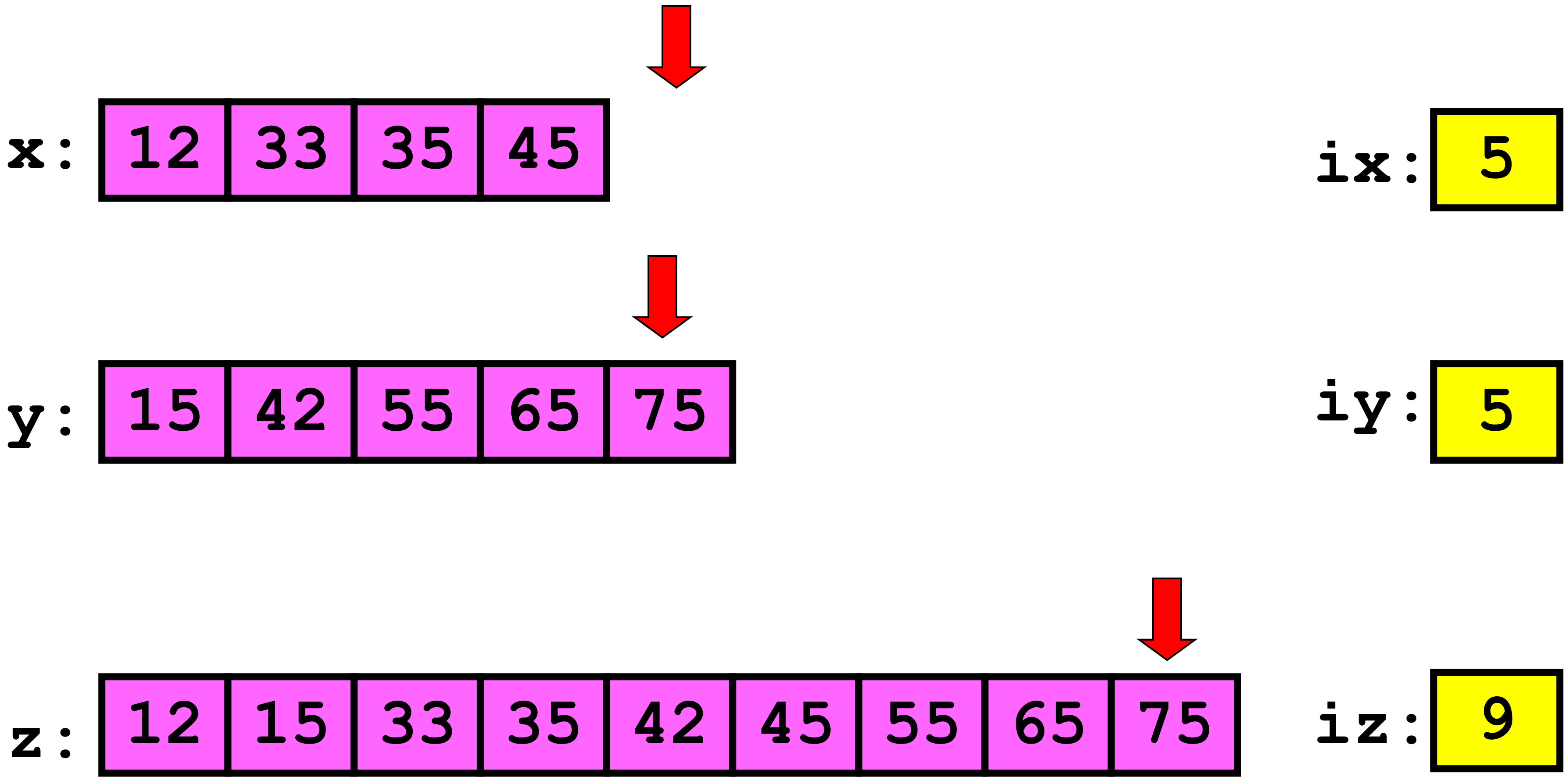

```
iy <= 5
```

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
```

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.

Cost of dividing work

Suppose each comparison we make costs $1

Given a vector with N elements,

- Insertion sort costs $N(N-1)/2
- Merge costs $(N-1)

(worst case)

Consider a vector with 8 elements

- Sorting by ourselves: $26
- Sorting by delegating work:
  - Left delegate (4 elements): $6
  - Right delegate (4 elements): $6
  - Merge (8 elements): $7
  - **Profit: $7!**

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…

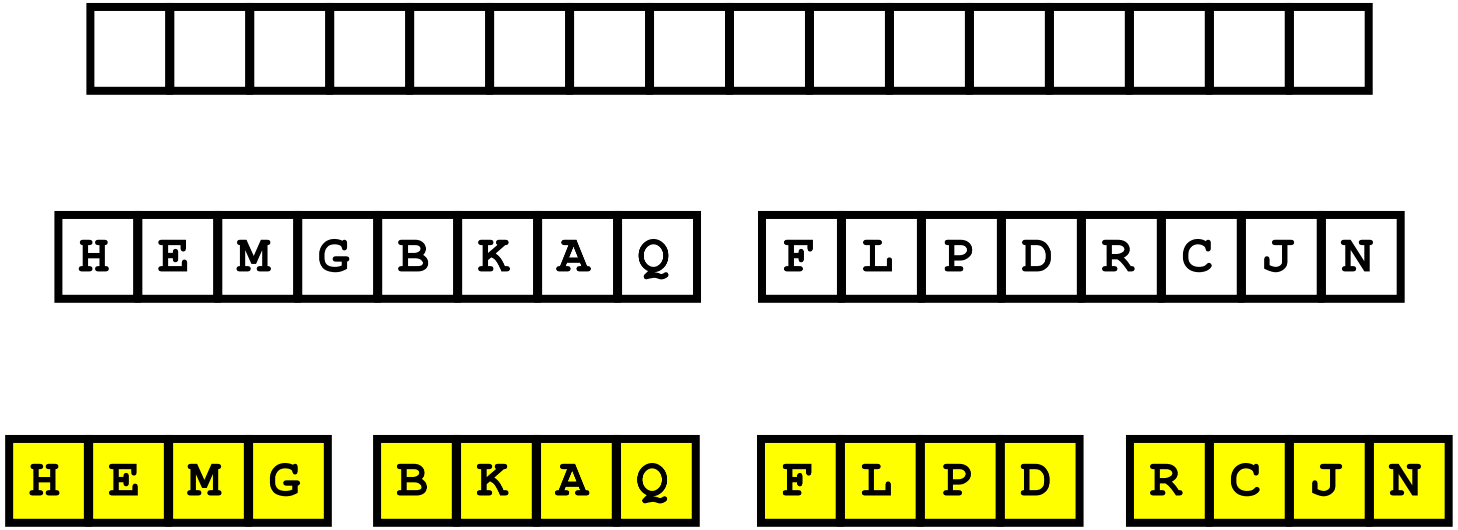
Subdivide the sorting task

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

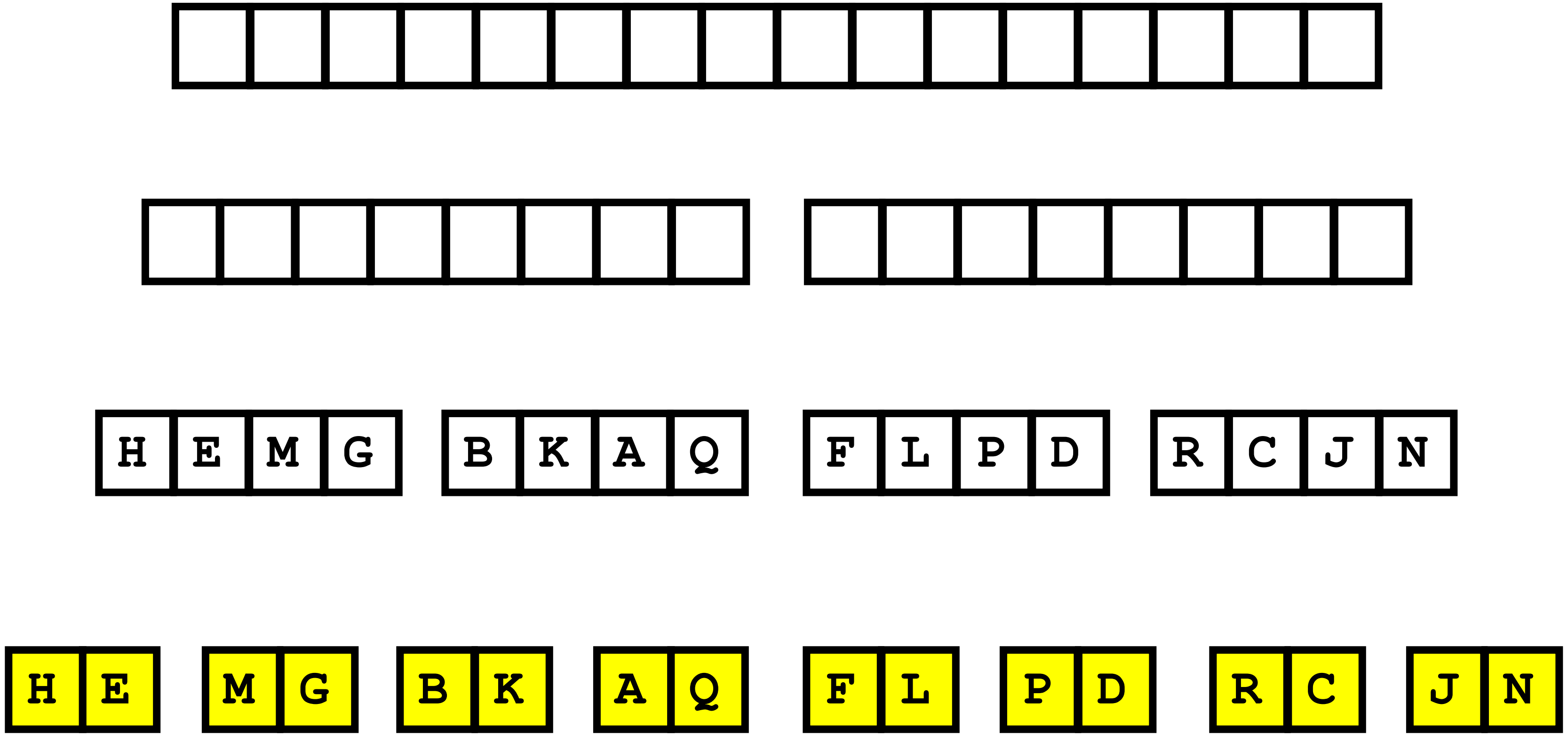
H	E	M	G	B	K	A	Q

F	L	P	D	R	C	J	N

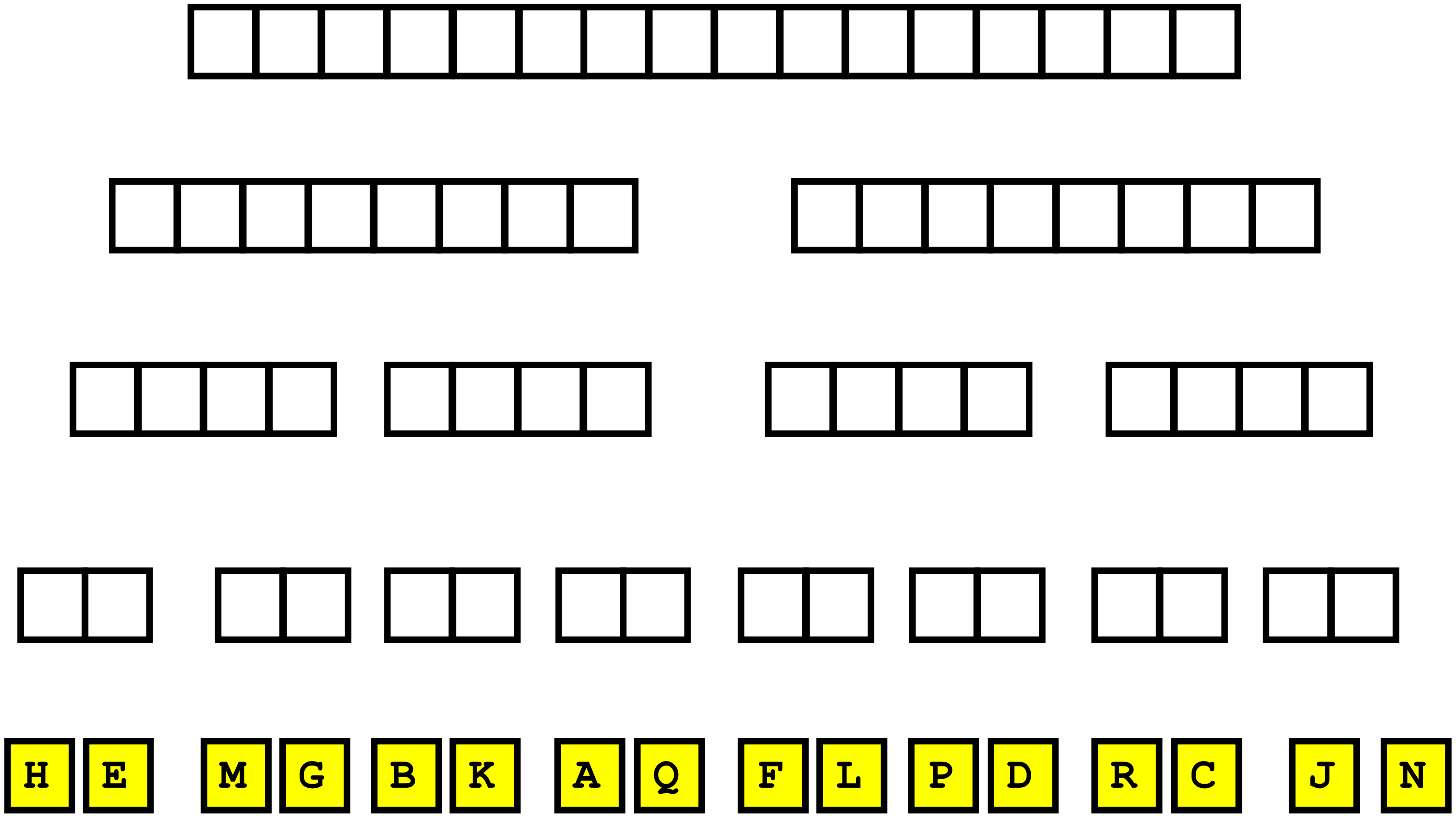
Subdivide again



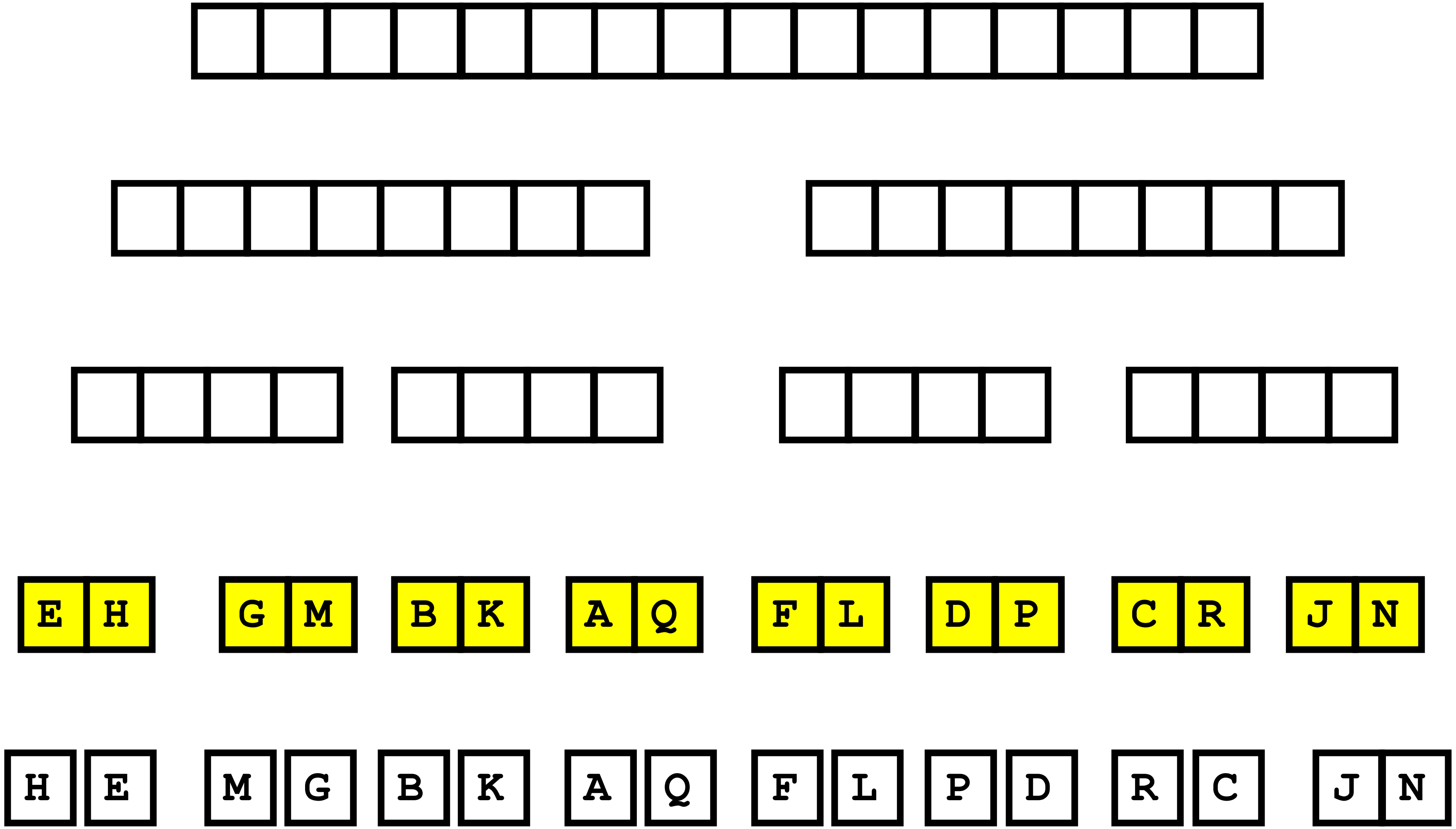
And again



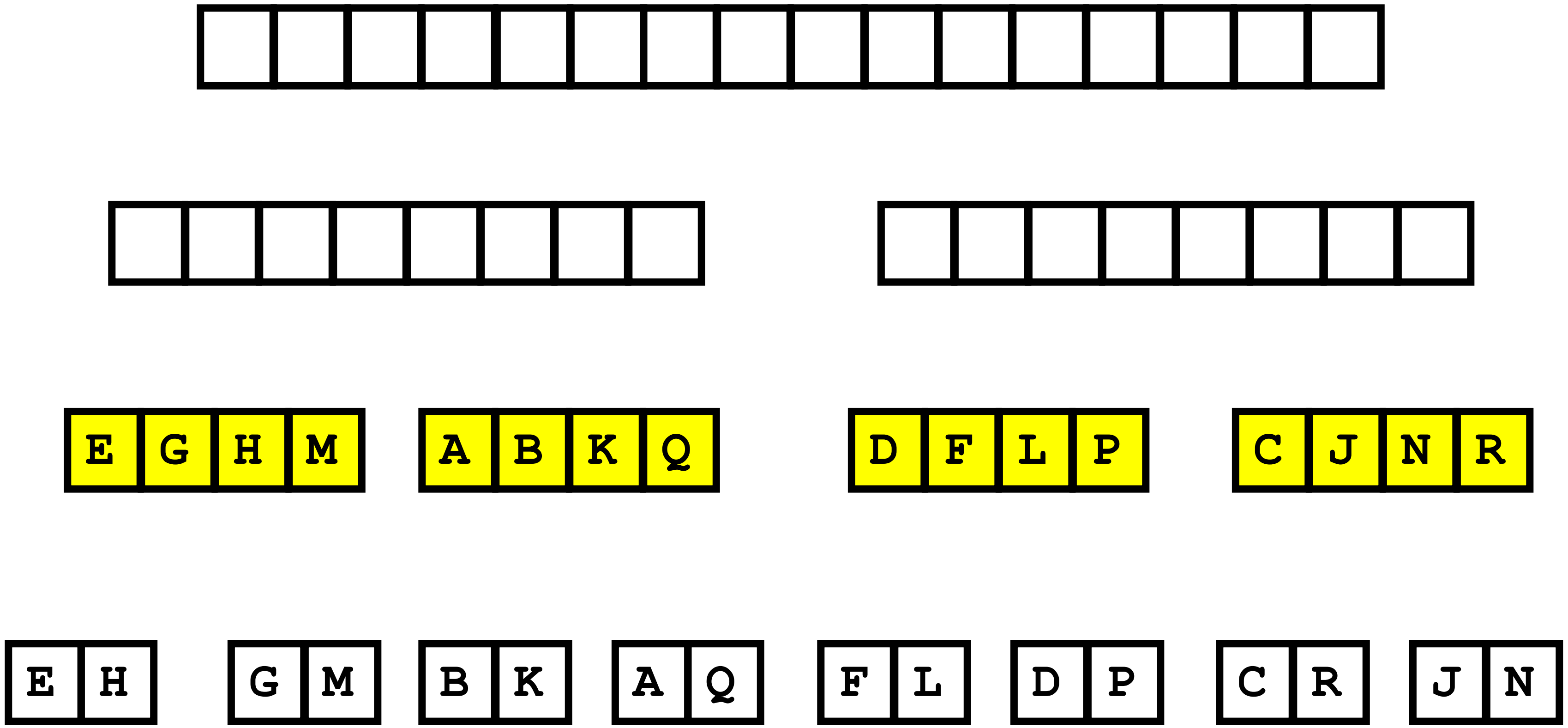
And one last time



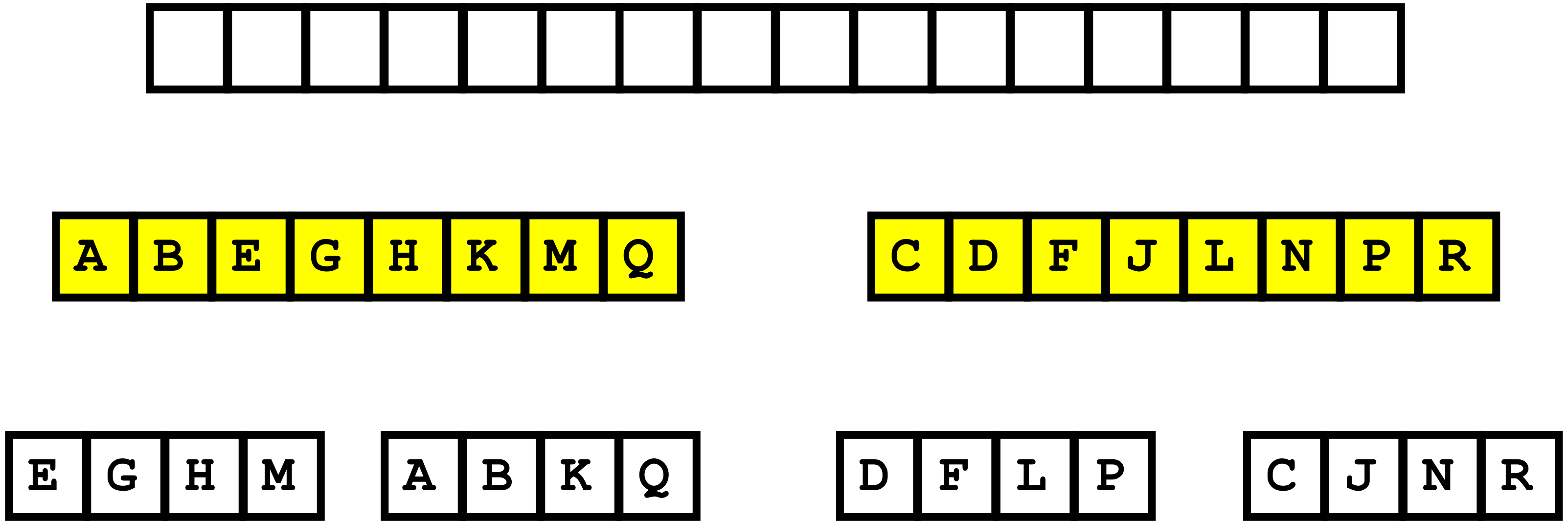
Now merge



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 (task is trivial)
      % Base case
else
      % Divide work
      % Delegate subproblems

      % Merge results
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
      % Divide work
      % Delegate subproblems

      % Merge results
end
```

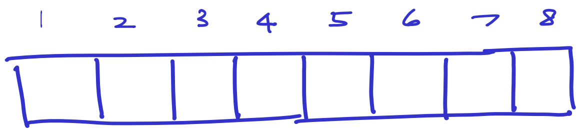
```
function y = mergeSort(x)
% x is a vector.  y is a vector
% consisting of the values in x
% sorted from smallest to largest.

n = length(x);
if n==1
      y = x;
else
      m  = floor(n/2);
      yL = 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
```

mergeSort – 1st call

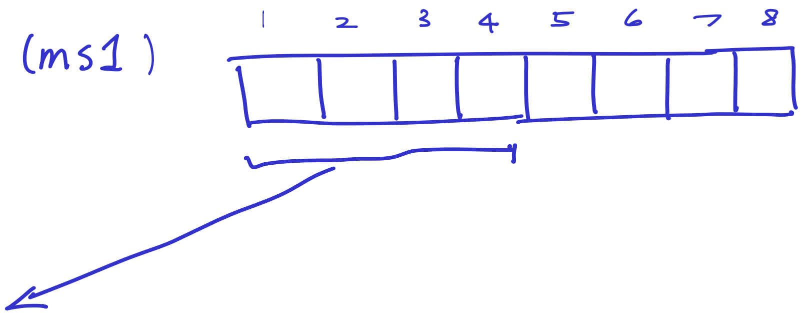
(ms1)



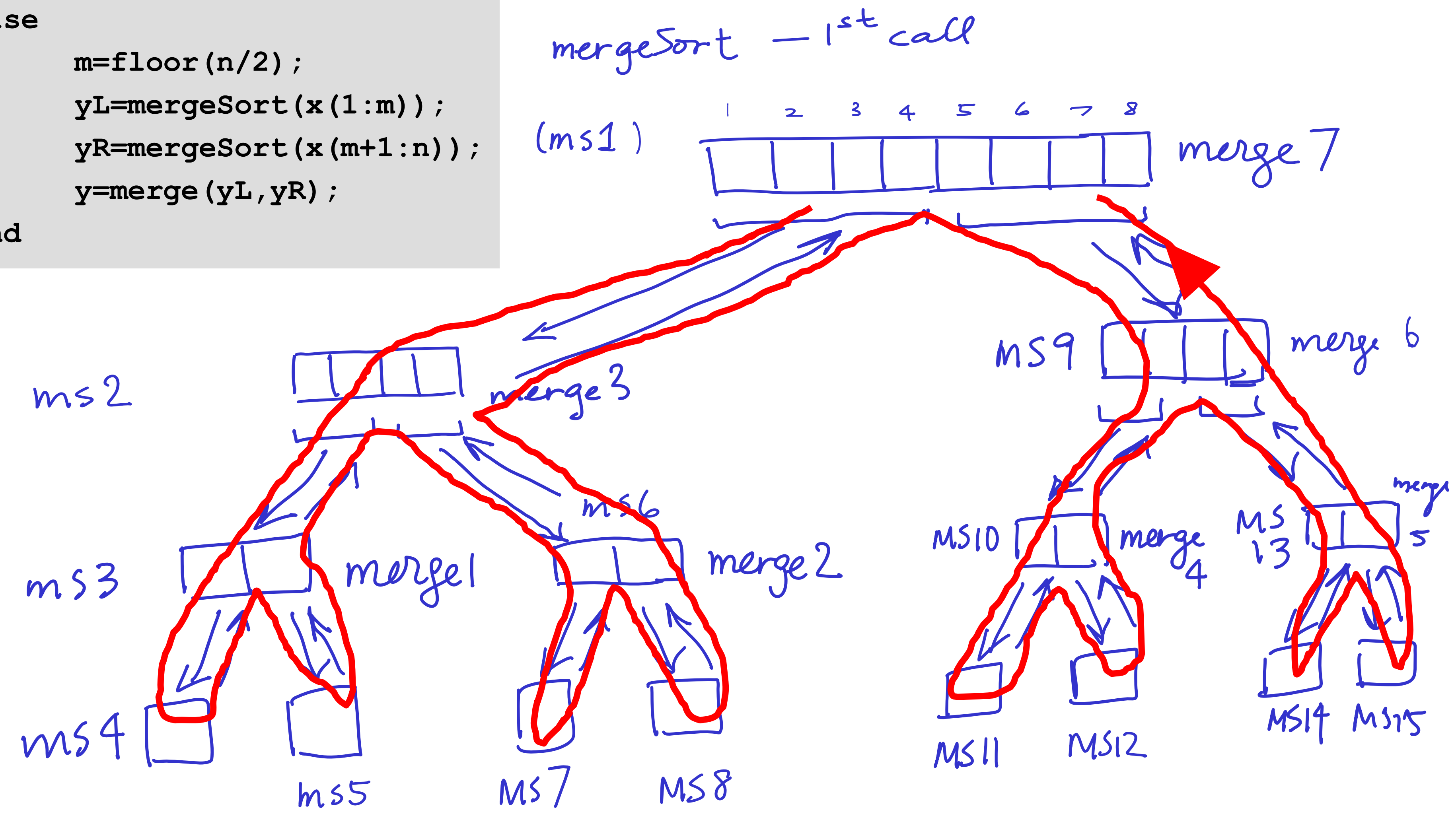
```
function y=mergeSort(x)
n=length(x);
if n==1
      y=x;
else
      m=floor(n/2);
      yL=mergeSort(x(1:m));
      yR=mergeSort(x(m+1:n));
      y=merge(yL,yR);
end
```

mergeSort – 1st call

(ms1)



```
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
```



How do merge sort and insertion sort compare?

- Insertion sort: (worst case) makes k comparisons to insert an element in a sorted array of k elements. For an array of length N:

  $1+2+\ldots+(N-1) = N(N-1)/2$, say N^2 for big N

- Merge sort:

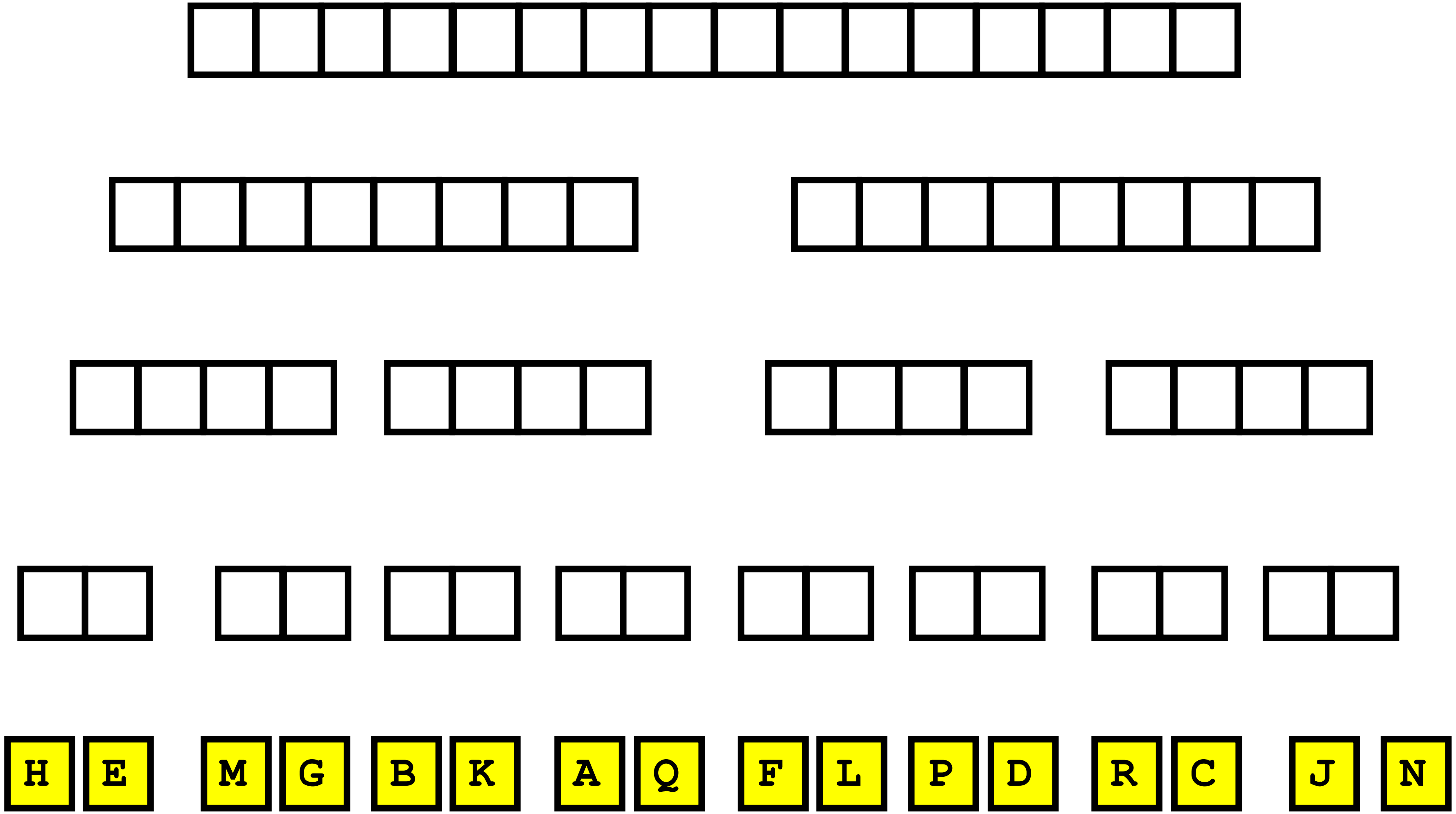
```
function y = mergeSort(x)
% x is a vector.  y is a vector
% consisting of the values in x
% sorted from smallest to largest.

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
```

All the comparisons between vector values are done in `merge`

Merge sort: about $\log_2(N)$ "levels";

about N comparisons each level



How do merge sort and insertion sort compare?

- Insertion sort: (worst case) makes i comparisons to insert an element in a sorted array of i elements. For an array of length N:

  $1+2+\ldots+(N-1) = N(N-1)/2$, say N^2 for big N

- Merge sort: $N \cdot \log_2(N)$

- Insertion sort is done *in-place*; merge sort (recursion) requires extra memory (call frames plus merge area)

See **`compareInsertMerge.m`**

How to choose??

- Depends on application
- Merge sort is especially good for sorting large data sets
  - Easily adapted to work with files if data is too big for memory
- Sort "stability" matters for object handles (elements may compare equal, but are actually distinct)
  - Insertion, Merge are intrinsically stable. QuickSort is not, but MATLAB's `sort()` does extra work to stabalize
- Insertion sort is "order N^2" at worst case, but what about an average case?
  - Insertion good for "fixing" a mostly sorted array, or adding just a few new elements

What we learned…

- Develop/implement algorithms for problems
- Develop programming skills
  - Design, implement, document, test, and debug
- Programming "tool bag"
  - Functions for reducing redundancy
  - Control flow (if-else; loops)
  - Recursion
  - Data structures, type
  - Graphics
  - File handling

What we learned… (cont'd)

- Applications and concepts
  - Image processing
  - Object-oriented programming—custom type
  - Sorting and searching (you should know the algorithms covered)
  - Approximation and error
  - Simulation, sensitivity analysis
  - Computational effort and efficiency

Where to go from here?

- Mathworks.com – Many free tutorials available on specific topics, e.g., signal processing, Simulink, …, etc.
- More detailed intro to scientific and engineering uses: "*Getting Started with MATLAB*" by Rudra Pratap. Excellent for independent, non-course-based learning
- Just play, i.e., experiment, with MATLAB programs! Many programs available in MATLAB "Community" forum "File Exchange"

Some courses for future consideration

- ENGRD/CS 2110 Object-oriented programming and data structure
  - CS 2111 Programming practicum

Highly recommended companion to CS2110

- CS 2800 Discrete Math (logic, proof, probability theory)
- CS 3220 Computational Mathematics for Computer Science
- Short language courses (e.g., Python, C++)

Computing gives us *insight* into a problem

- Computing is <u>not</u> about getting one answer!
- We build models and write programs so that we can "play" with the models and programs, learning—gaining insights—as we vary the parameters and assumptions
- Good models require domain-specific knowledge (and experience)
- Good <u>programs</u> …
    - are correct and have been thoroughly tested
    - are modular and cleanly organized
    - are well-documented
    - use appropriate data structures and algorithms
    - are reasonably efficient in time and memory


Best wishes
and
good luck with all your exams!