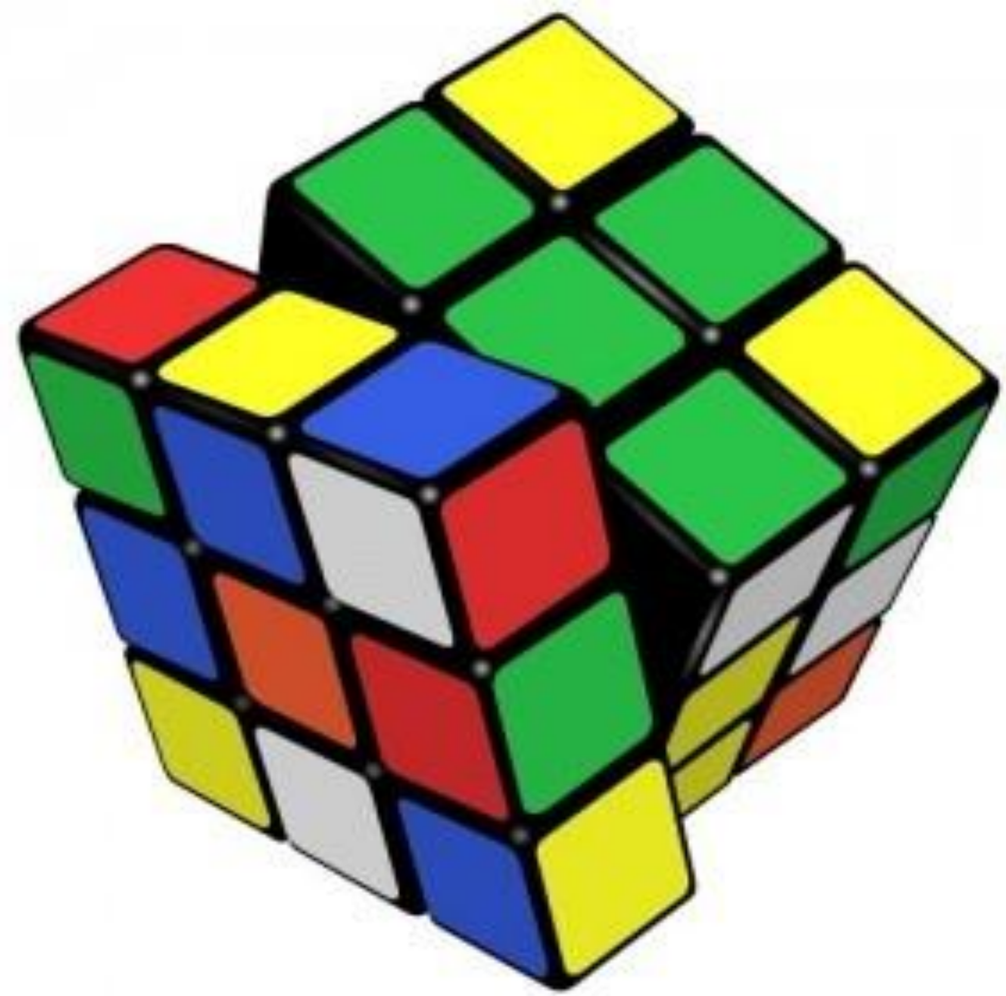
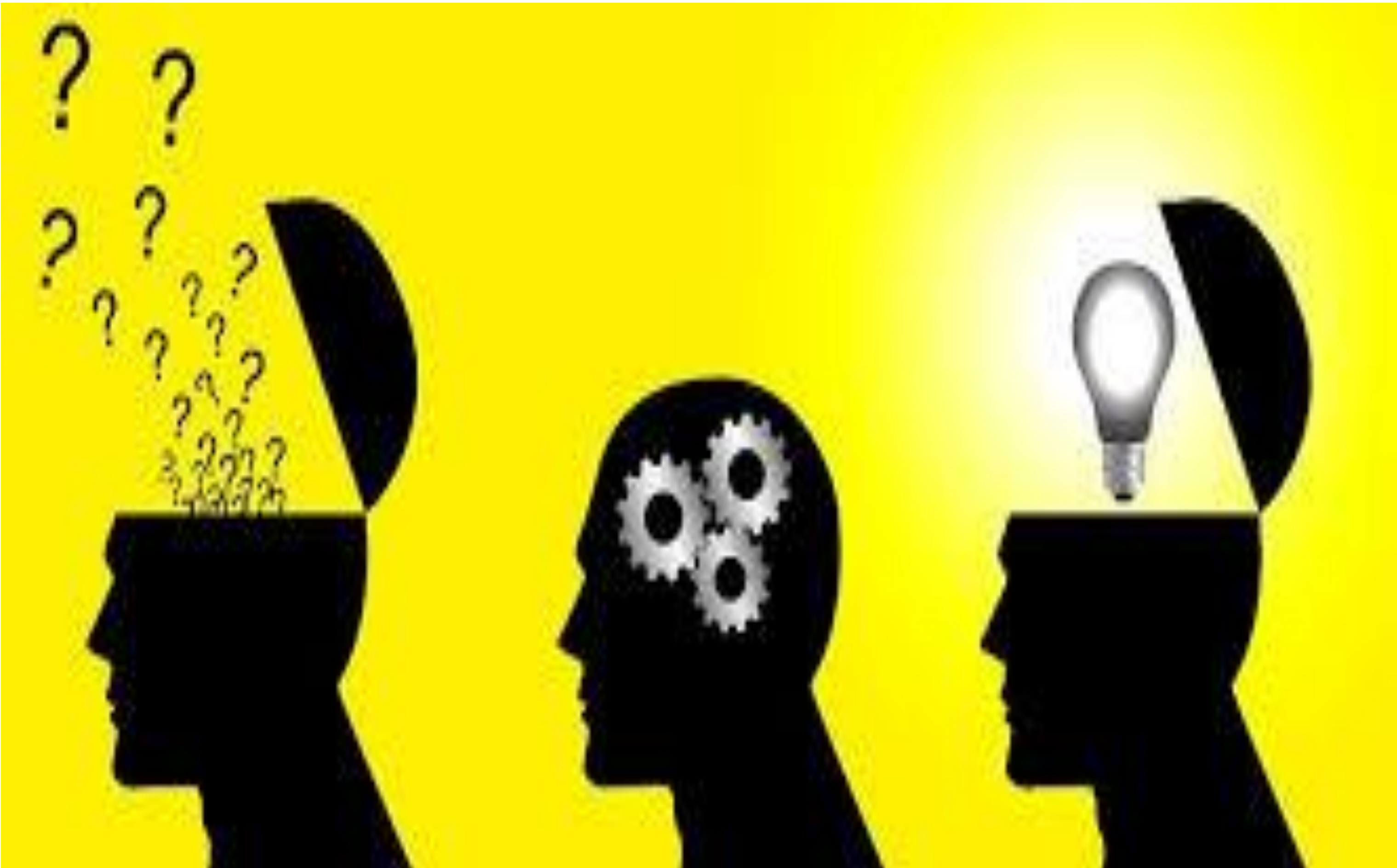
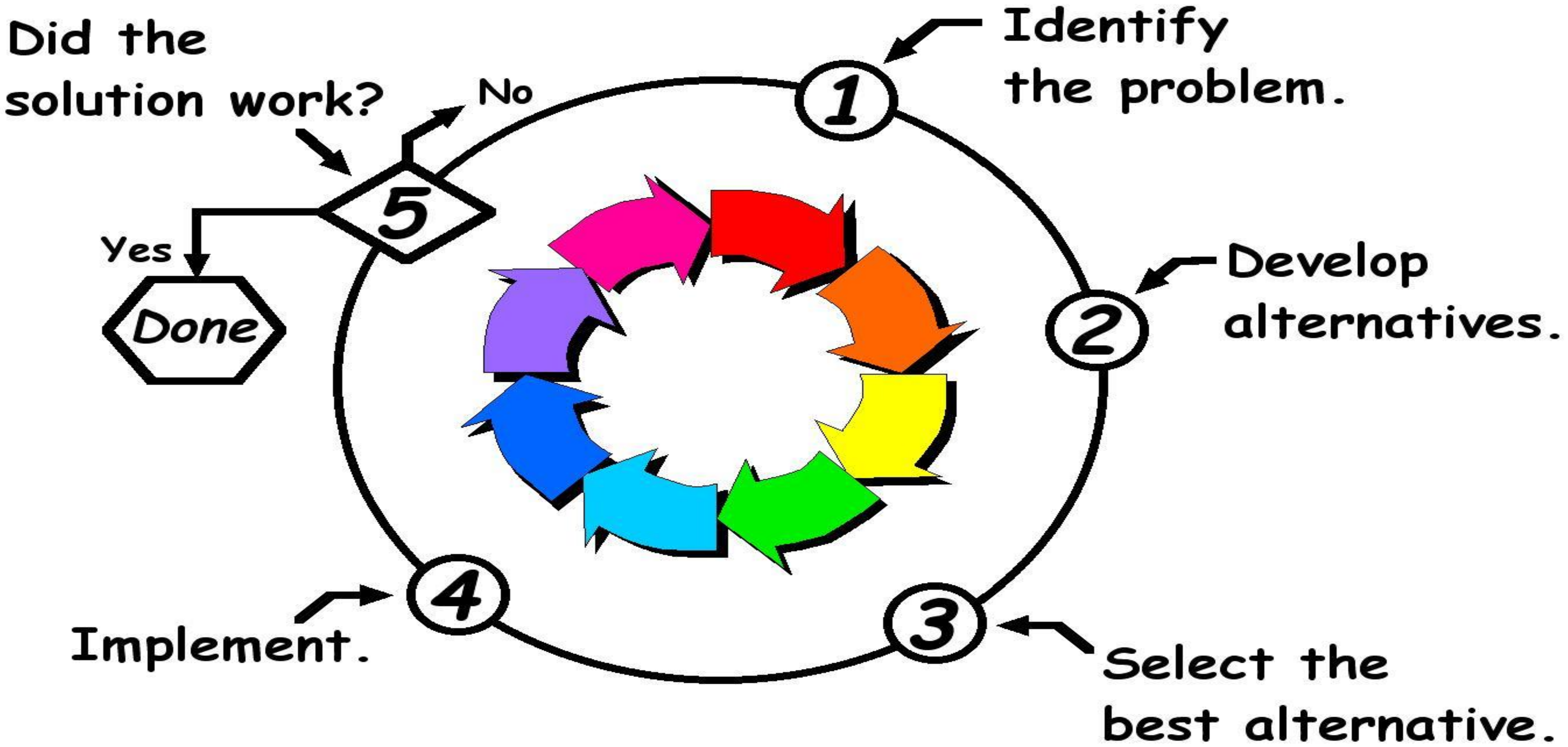
Problem solving using Algorithms





Steps to solve a problem...



Few Ways to solve problem

Basically they are-

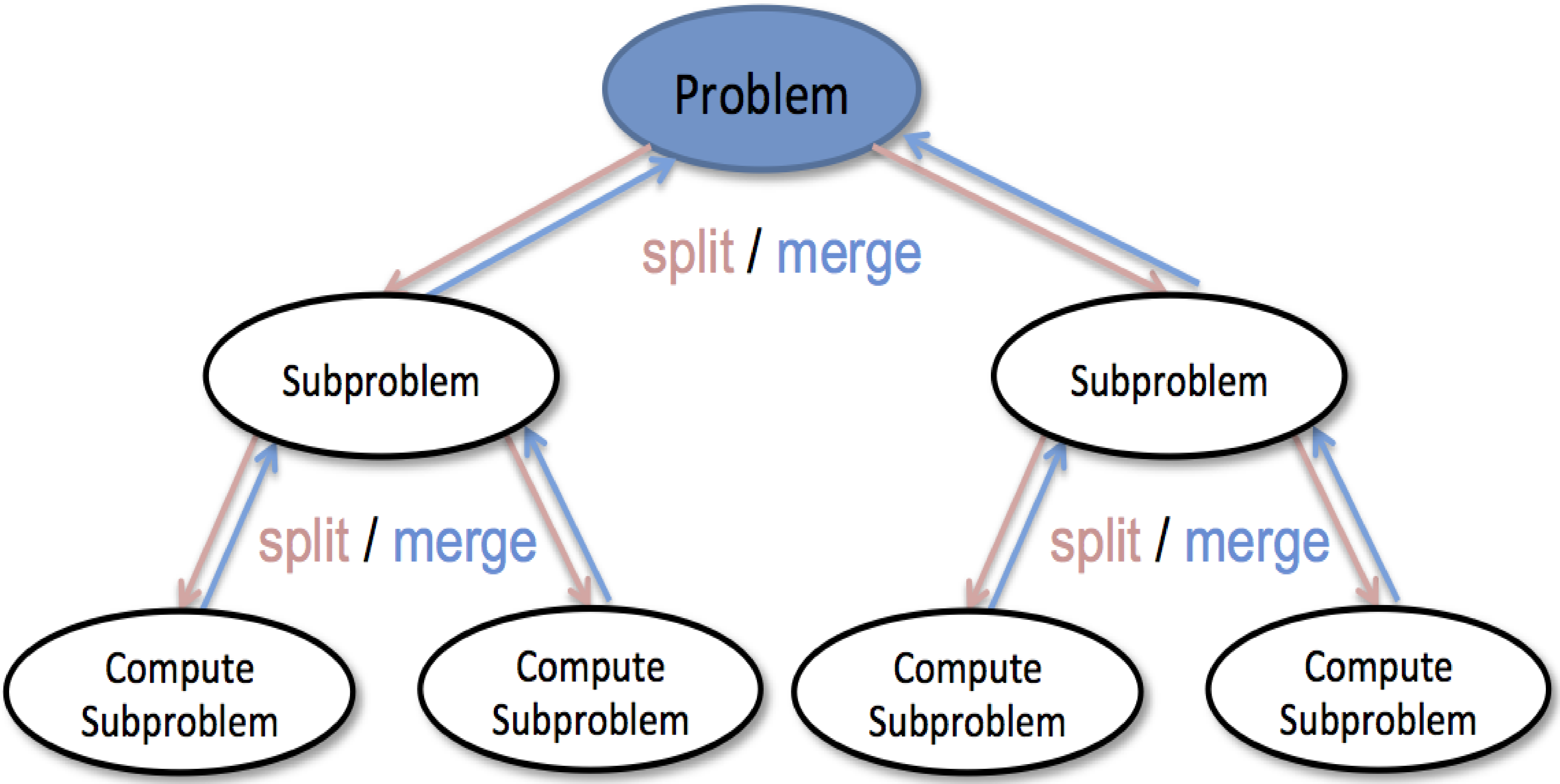
- Simple Iterative algorithms
- Divide and conquer
- Dynamic programming
- Greedy algorithm
- Backtracking algorithm
- Branch & bound algorithms
- and others.......................................

❖ ***All of above are also called as Algorithm Design Techniques or Strategies.***

Recursion

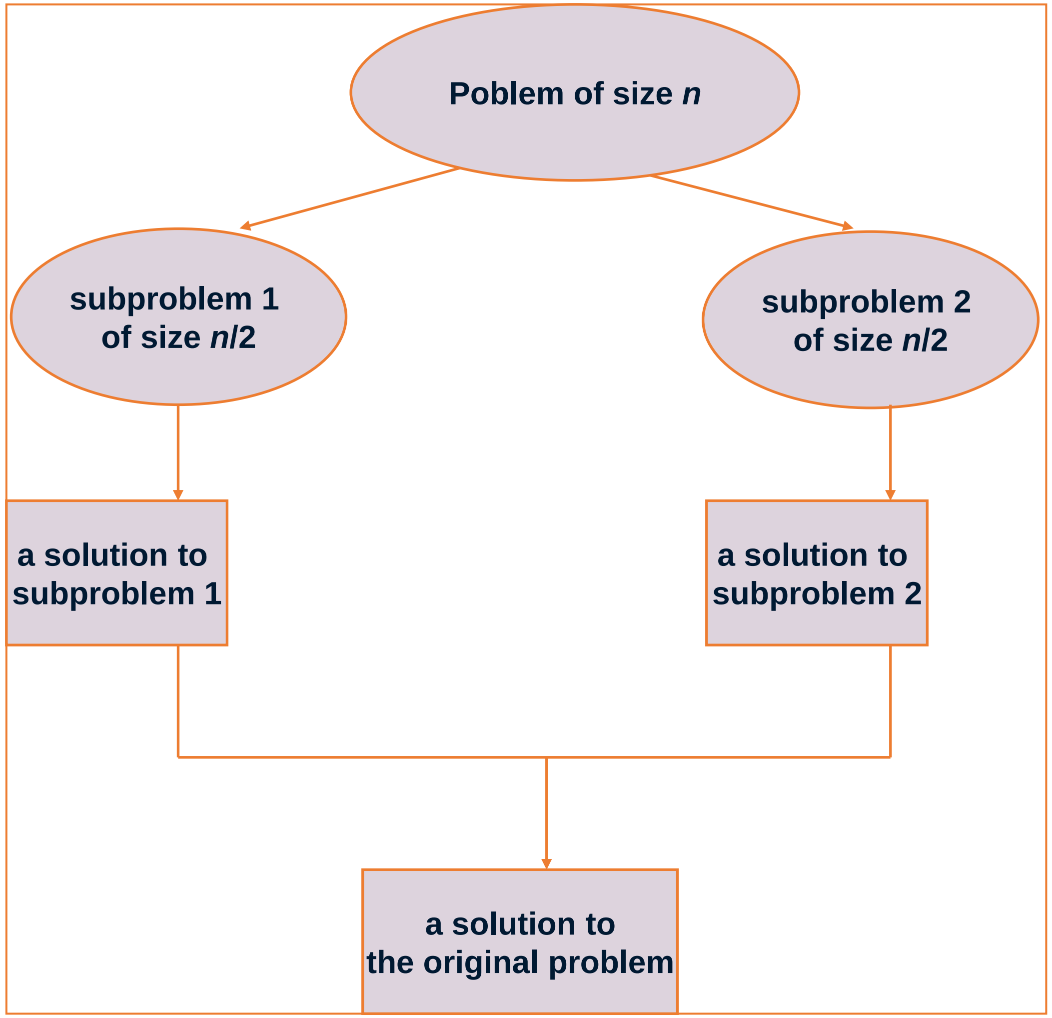
- Function calling itself
- 2 IMP things to remember :
  - initial value
  - stopping condition or Criteria
- How to design recursion :
  1. Divide the problem into one or more simple small parts of problem
  2. Call function on each part
  3. Combine solution of parts to get solution of problem.

DIVIDE AND CONQUER



Divide and Conquer

- **Basic Idea**
- Divide instance of problem into two or more smaller instances
- Solve smaller instances recursively
- Obtain solution to original (larger) instance by combining these solutions
- Divide and Conquer algorithms consist of two parts:
  - ***Divide:*** Smaller problems are solved recursively (except, of course, the base cases).
  - ***Conquer:*** The solution to the original problem is then formed from the solutions to the subproblems.



❖ Divide and Conquer used to derive efficient parallel algorithms.

Divide and Conquer Examples

- Sorting: merge sort and quick sort
- Binary search
- Binary tree traversals
- Matrix multiplication: Strassen’s algorithm
- Multiplication of large integers

- Closest-pair and convex-hull algorithms

The Divide and Conquer Algorithm

```
Divide_Conquer(problem P)
{
 if Small(P) return S(P);
 else {

   divide P into smaller instances P1, P2, ..., Pk, k≥1;

   Apply Divide_Conquer to each of these subproblems;

   return Combine(Divide_Conque(P1), Divide_Conque(P2),...,
 Divide_Conque(Pk));
 }
}
```

Analysis of Recursive algorithm

- Time complexity:

$$T(n) = \begin{cases} 2T(n/2)+S(n)+M(n) & , n \geq c \\ b & , n < c \end{cases}$$

where S(n) : time for splitting

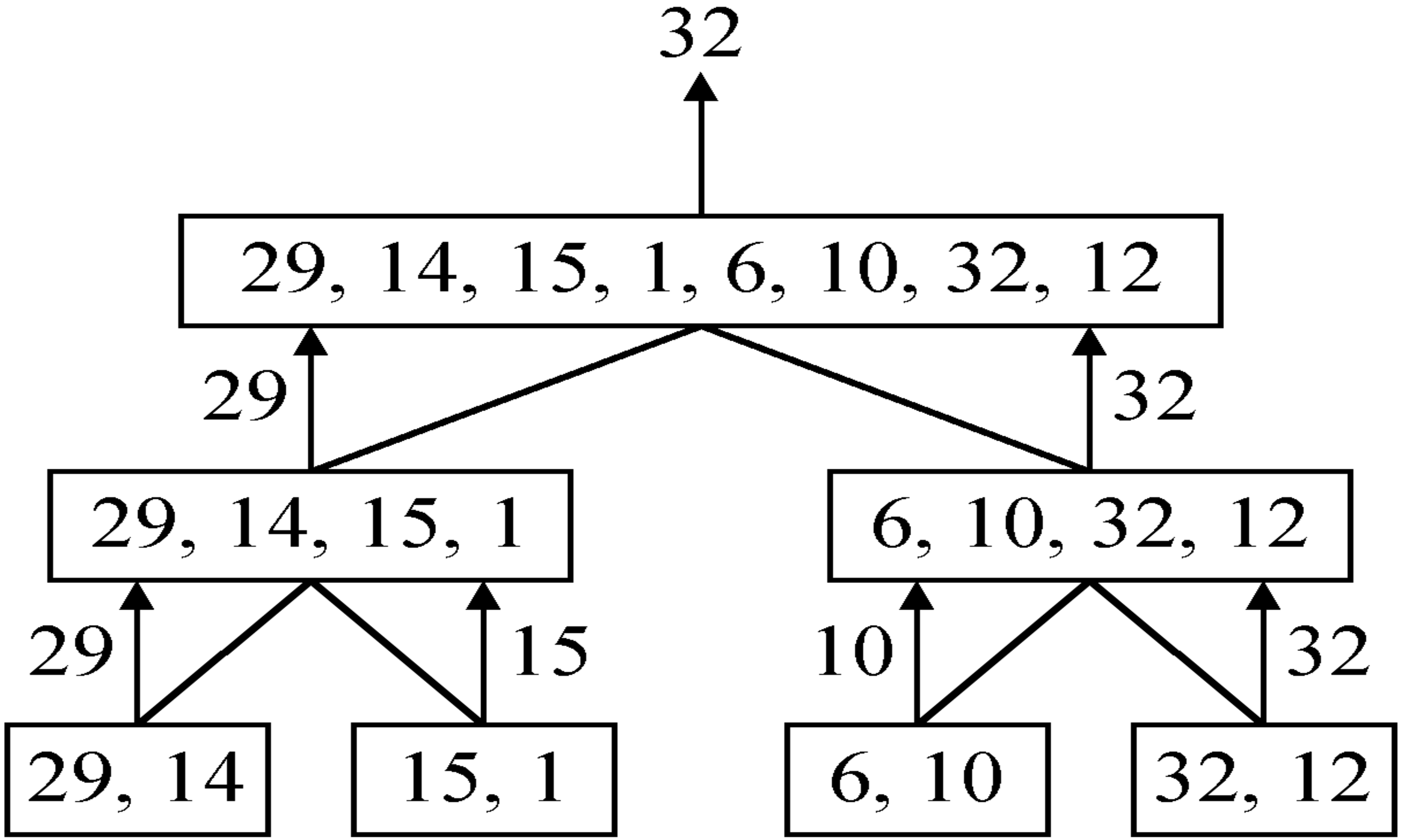
M(n) : time for merging

b : a constant

c : a constant

A simple example

- finding the maximum of a set S of n numbers



Time complexity of Finding Maximum

T(n): Time Complexity Function

$$T(n)=\begin{cases} 2T(n/2)+1 & , n>2 \\ 1 & , n\leq 2 \end{cases}$$

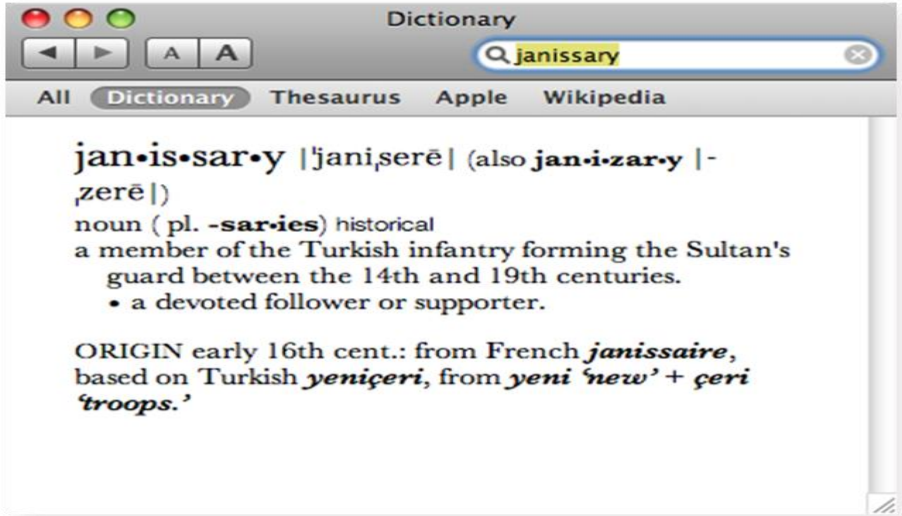
- Calculation of T(n):

Assume $n = 2^k$,

$$\begin{aligned} T(n) &= 2T(n/2)+1 \\ &= 2(2T(n/4)+1)+1 \\ &= 4T(n/4)+2+1 \\ &\quad\vdots \\ &=2^{k-1}T(2)+2^{k-2}+\ldots+4+2+1 \\ &=2^{k-1}+2^{k-2}+\ldots+4+2+1 \\ &=2^k-1 = n-1 \end{aligned}$$

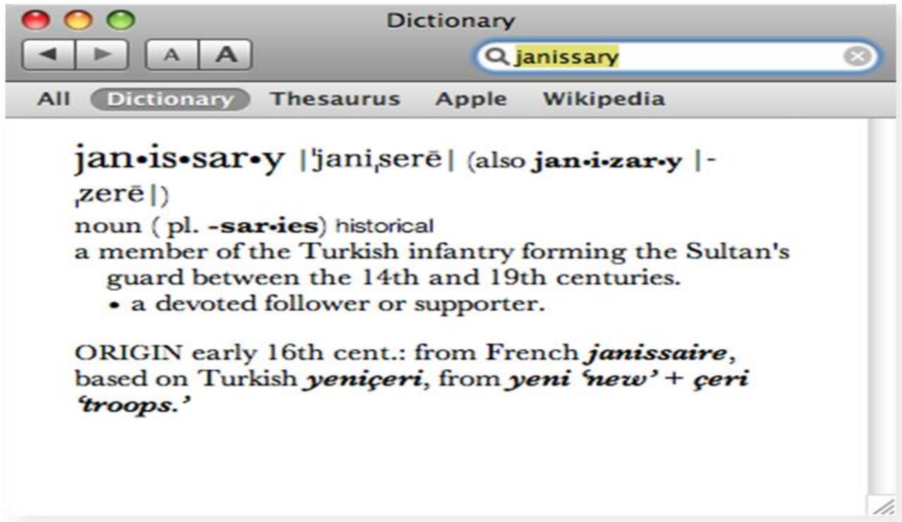
Searching in a Dictionary

- To get a general sense of how the divide and conquer strategy improves search, consider how people find information in a phone book or dictionary
- suppose you want to find “janissary” in a dictionary
  - open the book near the middle
  - the heading on the top left page is “kiwi”, so move back a small number of pages.
  - here you find “hypotenuse”, so move forward find “ichthyology”, move forward again..
- The number of pages you move gets smaller (or at least adjusts in response to the words you find)



Searching in a Dictionary

- A detailed specification of this process:
  1. The goal is to search for a word w in entire region of the book.
  2. At each step pick a word x in the middle of the current region
  3. There are now two smaller regions: the part before x and the part after x
  4. If w comes before x, repeat the search on the region before x, otherwise search the region following x (go back to step 3)
- Note: at first a “region” is of a group of pages, but eventually a region is a set of words on a single page



A Note About Organization

- **An important note: an efficient search depends on having the data organized in some fashion**
  - if books in a library are scattered all over the place we would have to do an iterative search
  - start at one end of the room and progress toward the other
- If books are sorted or carefully cataloged we can try a binary search or other method.

Unordered Linear Search

- int UnsorteddLinearSearch (int A[], int n, int data)

```
{
        for (int i = 0; i < n; i++)
        {
                if (A[i] == data)
                return i;
        }
        return -1;
}
```

Sorted/Ordered Linear Search

```
int SortedLinearSearch(int A[], int n, int data)
{
        for (int i = 0; i < n; i++)
        {
                if (A[i] == data)
                return i;
                else if(A[i] > data)
                return -1;
        }
        return -1;
}
```

Binary Search

- The binary search algorithm uses the divide-and-conquer strategy to search through an array.

- The array ***must be sorted.***

- ***IDEA:***
  - 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: 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: $\boxed{1}$

Mid: $\boxed{6}$

R: $\boxed{12}$

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

$x < v(\text{Mid})$

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

Algorithm binary-search

Input: A sorted sequence of n elements stored in an array.

Output: The position of x (to be searched).

Step 1: If only one element remains in the array, solve it directly.

Step 2: Compare x with the middle element of the array.

Step 2.1: If x = middle element, then output it and stop.

Step 2.2: If x < middle element, then recursively solve the problem with x and the left half array.

Step 2.3: If x > middle element, then recursively solve the problem with x and the right half array.

```
//Iterative Binary Search Algorithm
int BinarySearchIterative[int A[], int n, int data)
{
        int low =  0;
        int high = n-1;
        while (low <= high)
        {
                mid  = low + (high-low)/2; //To avoid overflow

                if (A[mid] == data)
                        return mid;
                else if (A[mid] < data)
                        low =  mid + 1;
                else
                        high =  mid - 1;
        }
        return -1;
}
```

```
//Recursive Binary Search Algorithm
int BinarySearchRecursive[int A[], int low, int high, int data)
{
        int mid  = low + (high-low)/2; //To avoid overflow

        if (A[mid] == data)
                return mid;
        else if (A[mid] < data)
                return BinarySearchRecursive (A, mid + 1, high, data);
        else
                return BinarySearchRecursive (A, low, mid - 1 , data);

        return -1;
}
```

Implementation	Search-Worst Case	Search-Avg. Case
Unordered Array	n	$\frac{n}{2}$
Ordered Array	$logn$	$logn$
Unordered List	n	$\frac{n}{2}$
Ordered List	n	$\frac{n}{2}$
Binary Search (arrays)	$logn$	$logn$
Binary Search Trees (for skew trees)	n	$logn$

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…

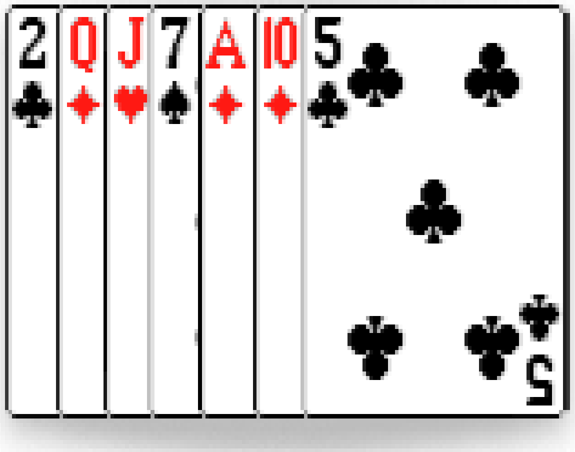
Mergesort

- Mergesort algorithm is one of two important divide-and-conquer sorting algorithms (the other one is quicksort).
- The merge sort algorithm uses “Bottom up” approach
  - start by solving the smallest pieces of original problem
  - keep combining their results into larger solutions
  - eventually the original problem will be solved
- It is a recursive algorithm.
  - Divides the list into halves,
  - Sort each halve separately, and
  - Then merge the sorted halves into one sorted array.

Merging Cars by key [Aggressiveness of driver]. Most aggressive goes first.

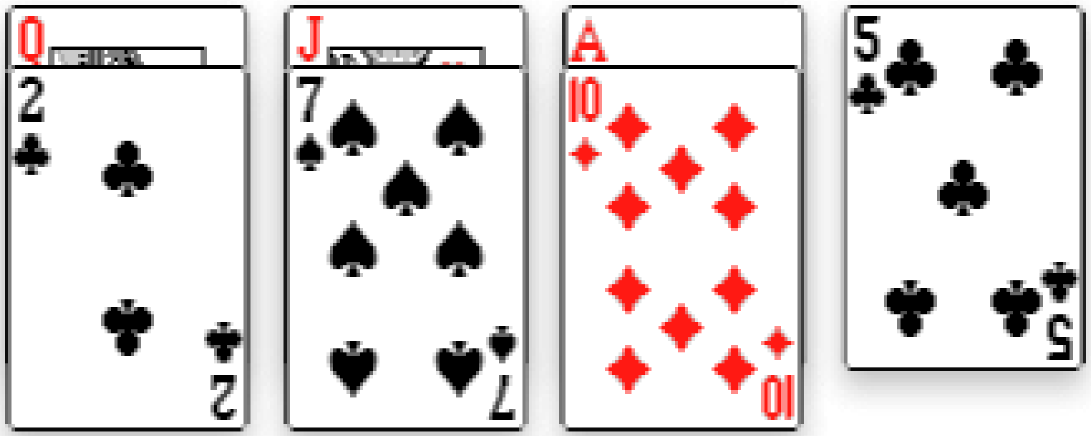
Example: sorting playing cards

initial hand

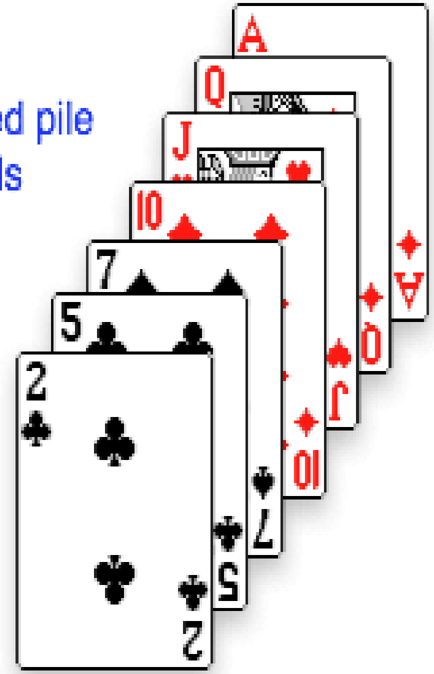


- divide the cards into groups of two
- sort each group -- put the smaller of the two on the top
- merge groups of two into groups of four
- merge groups of four into groups of eight
- ...

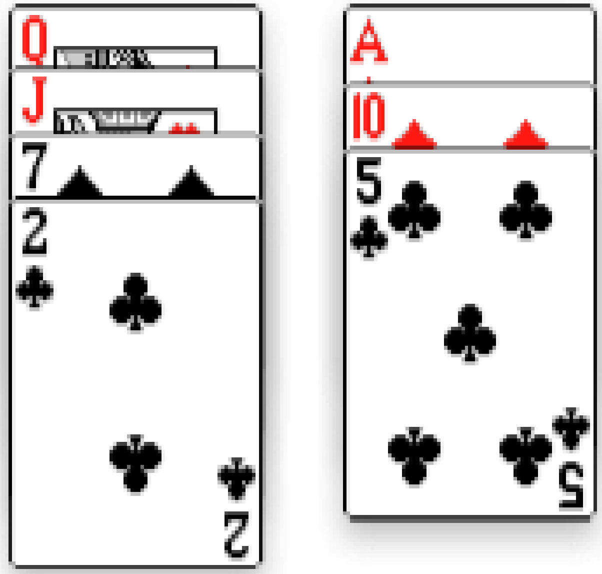
sorted piles of size two



final sorted pile of all cards



sorted piles of size four



In this example:

compare 2 with 5, pick up the 2
compare 5 with 7, pick up the 5
compare 7 with 10, pick up the 7
....

Merge Sort Procedure

Step 1 − if it is only one element in the list it is already sorted, return.

Step 2 − divide the list recursively into two halves until it can no more be divided.

Step 3 − merge the smaller lists into new list in sorted order.

Another Example

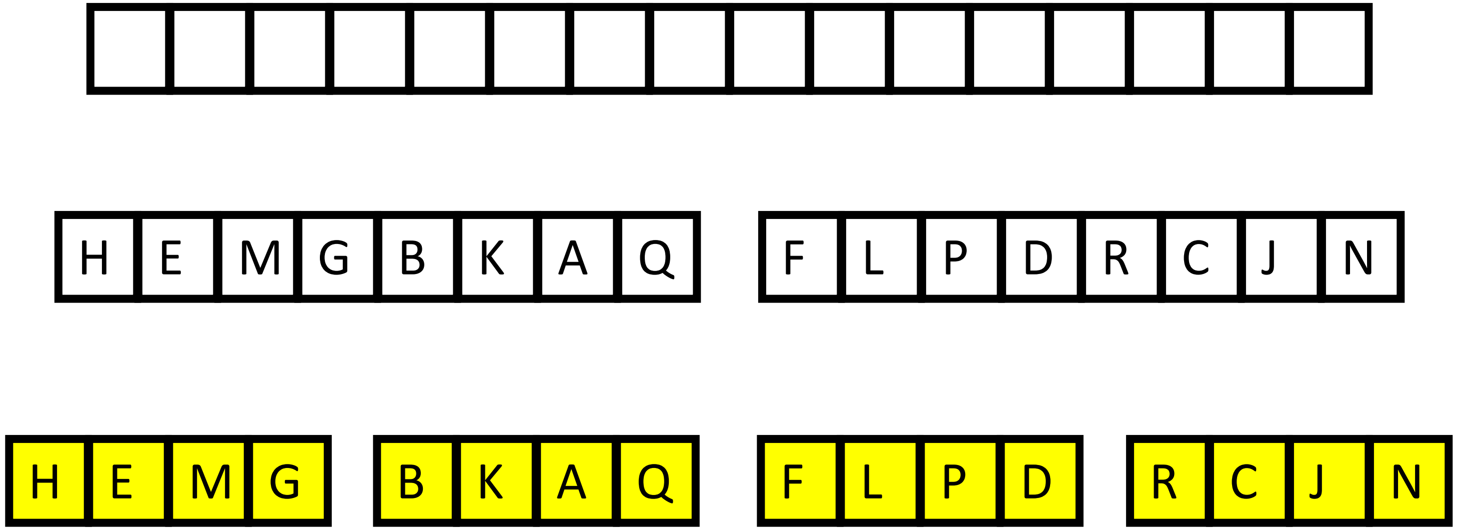
- Subdivide the sorting task

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

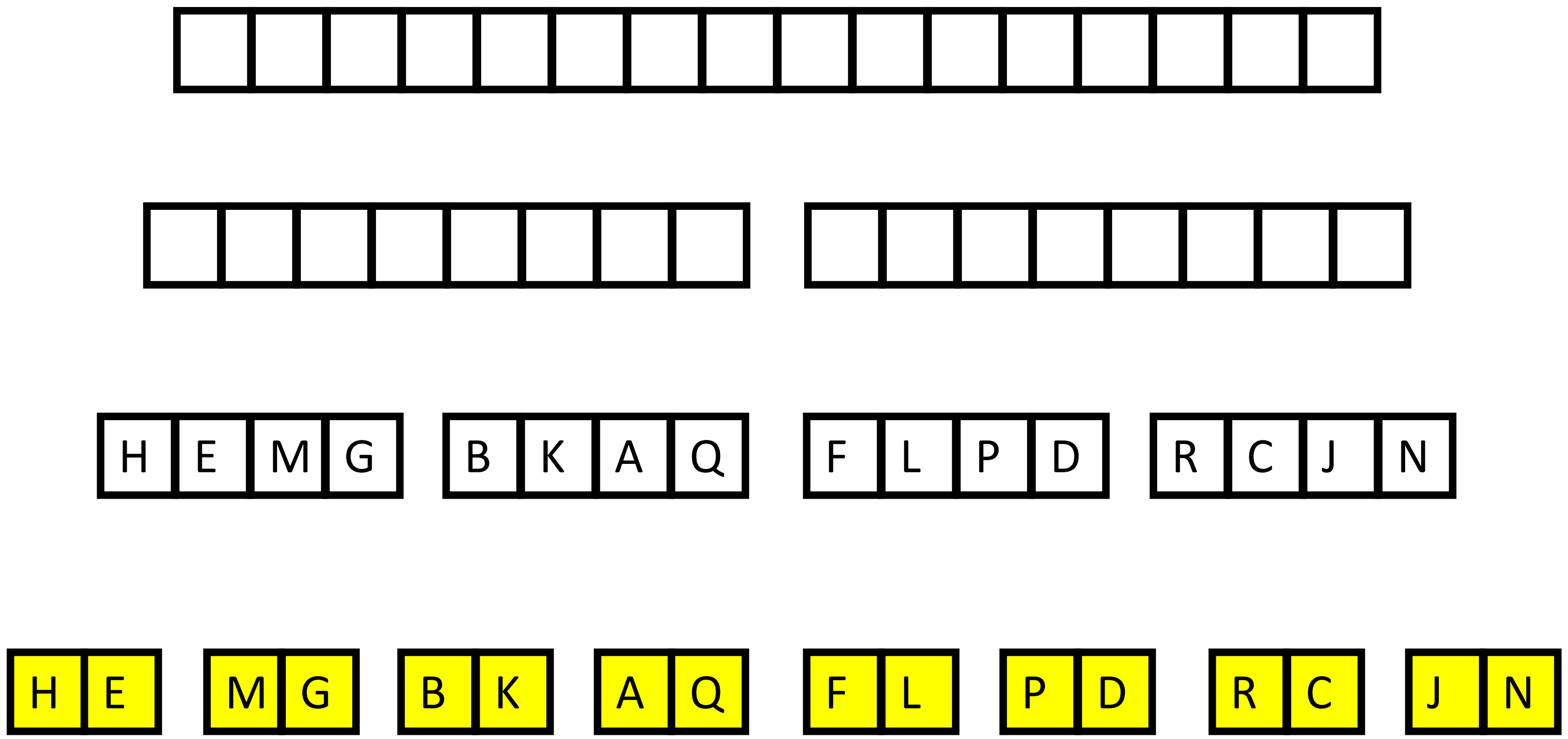
H	E	M	G	B	K	A	Q

F	L	P	D	R	C	J	N

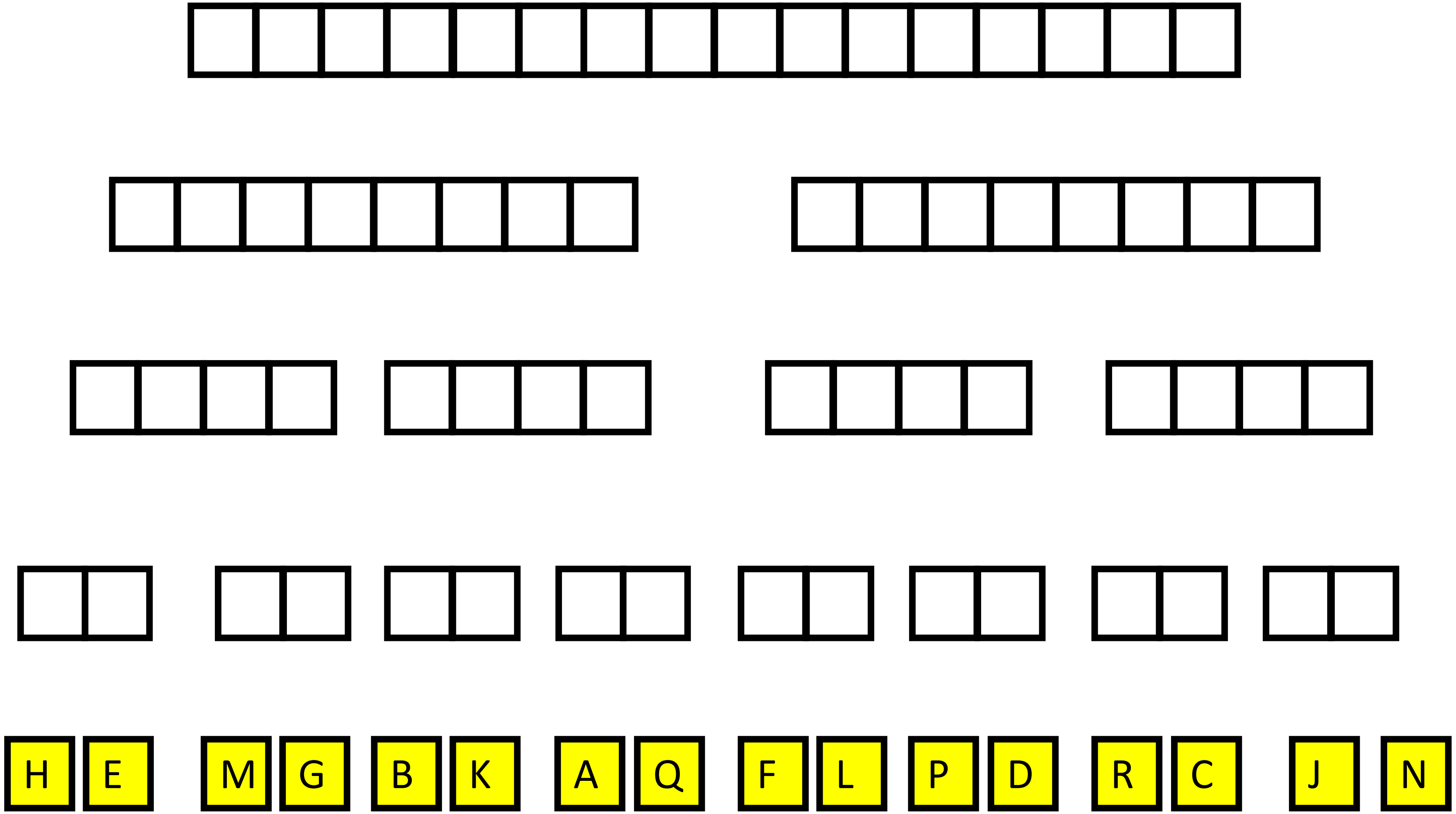
Subdivide again



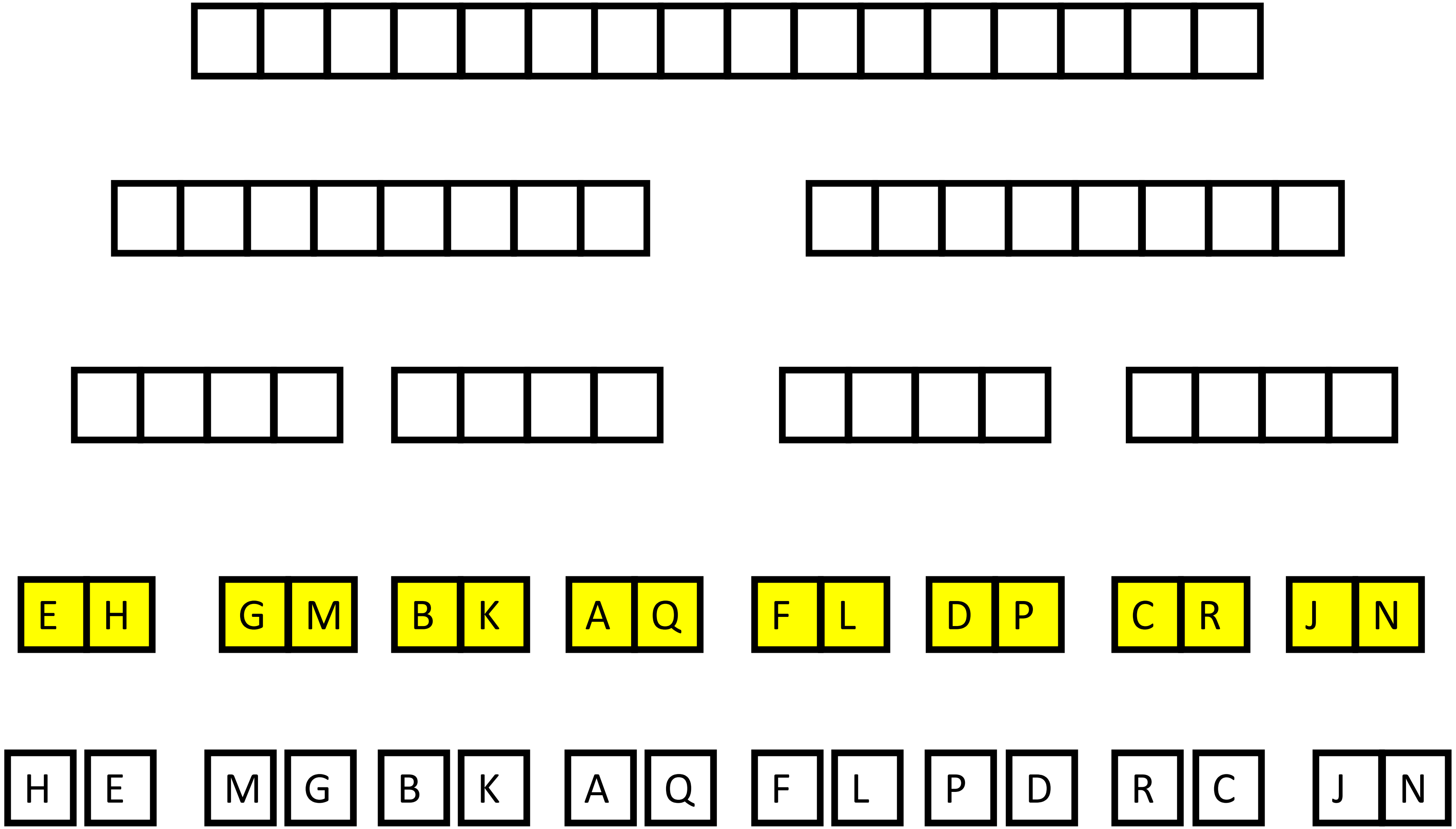
And again



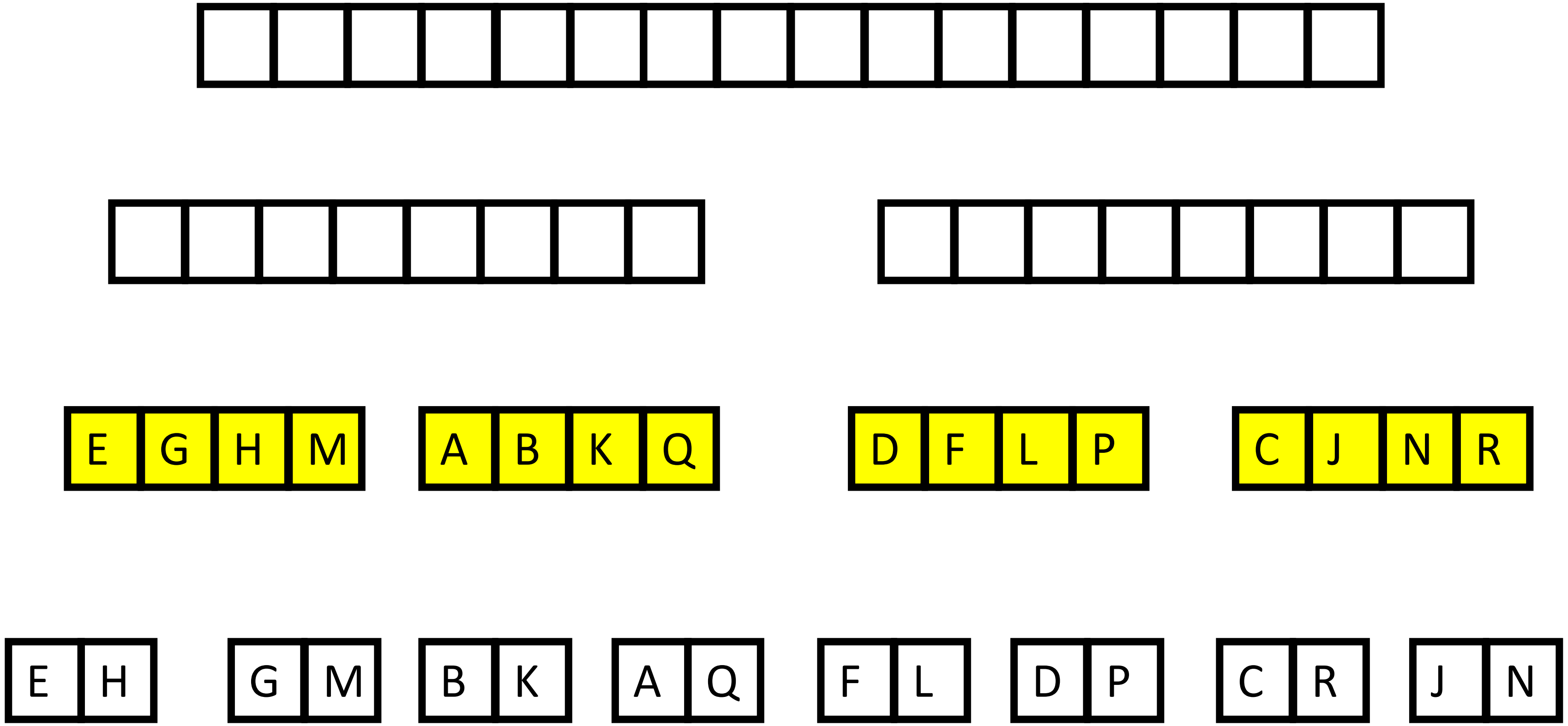
And one last time



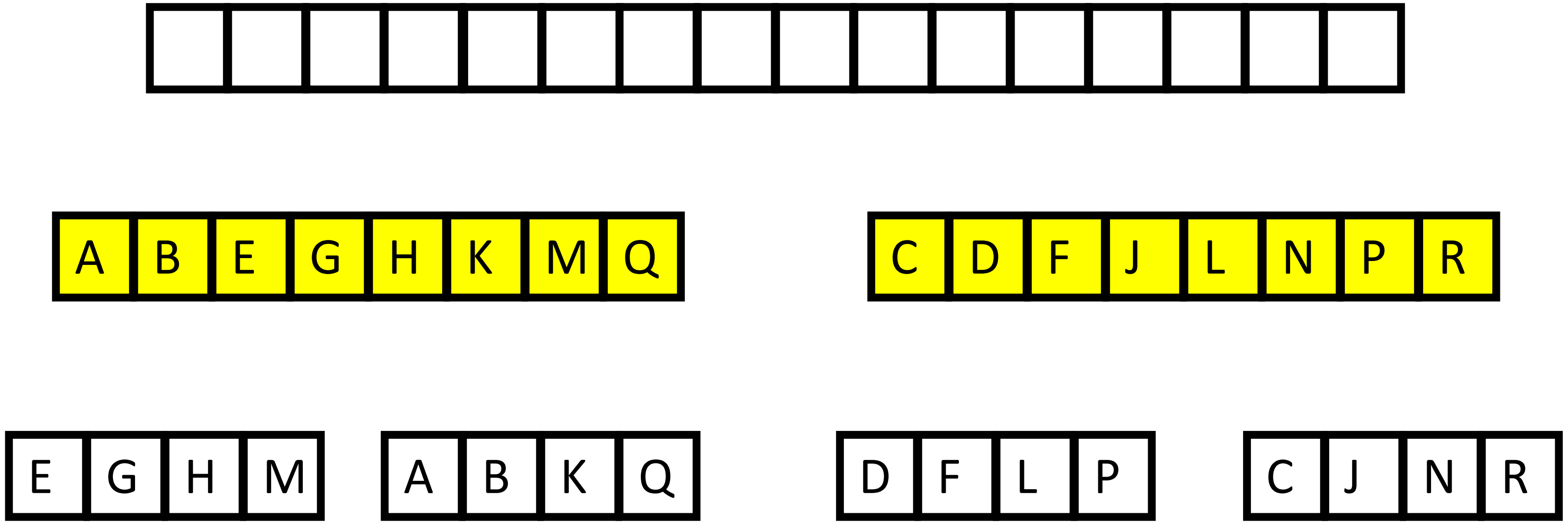
Now merge



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

Merging two sorted Array : Example

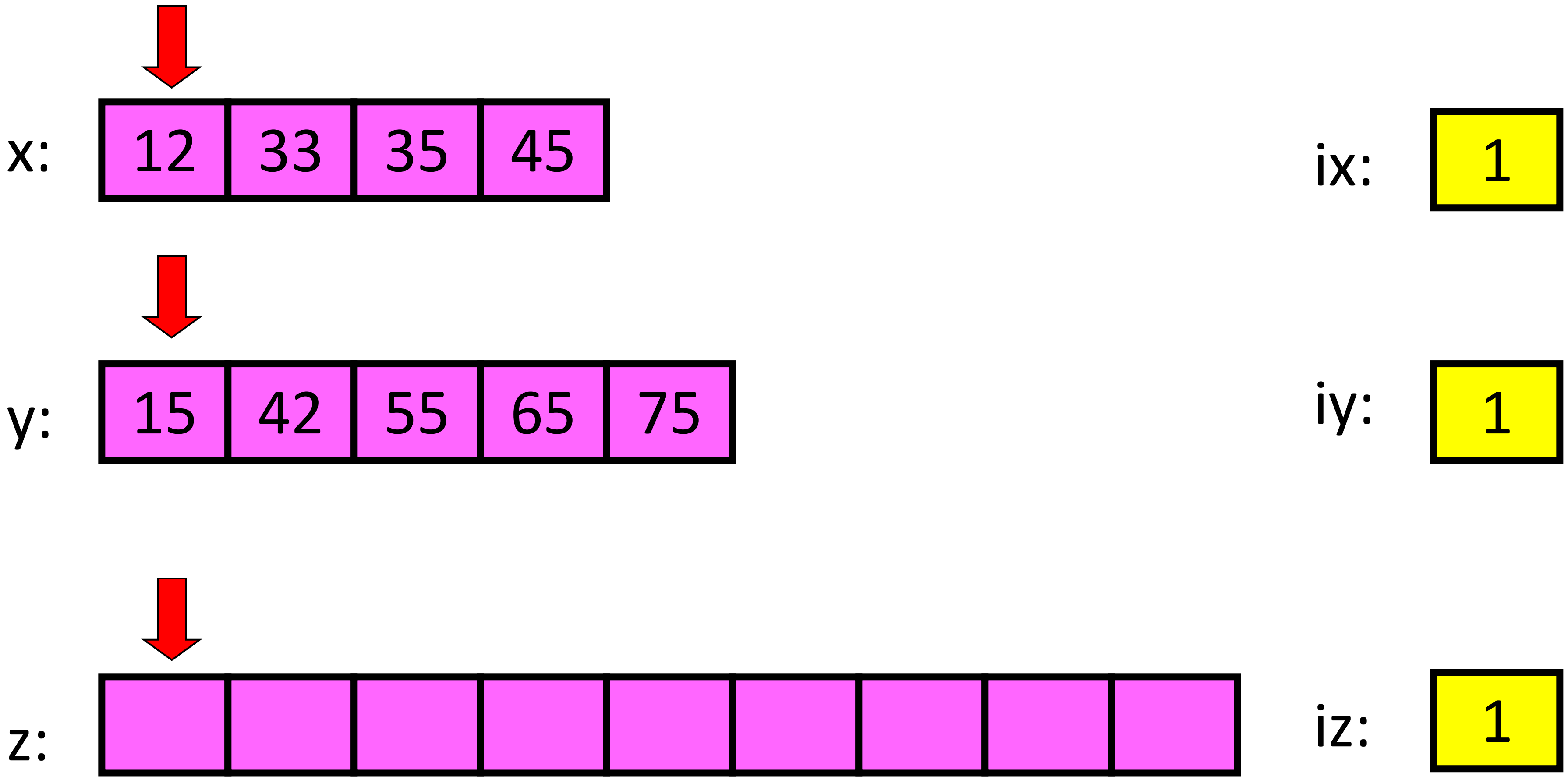
- The central sub-problem is the merging of two sorted arrays into one single sorted array

12	33	35	45

15	42	55	65	75

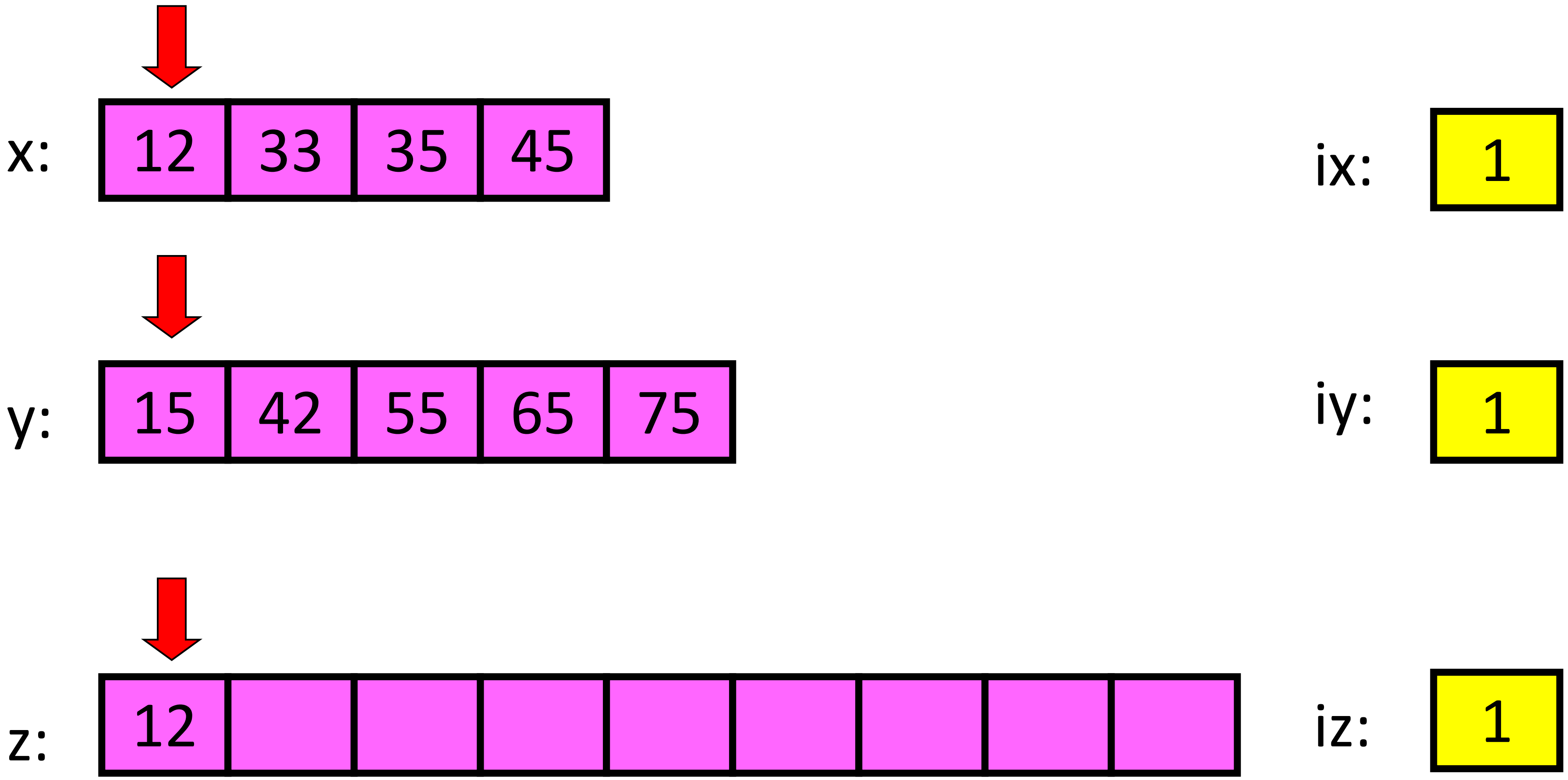
12	15	33	35	42	45	55	65	75

Merge



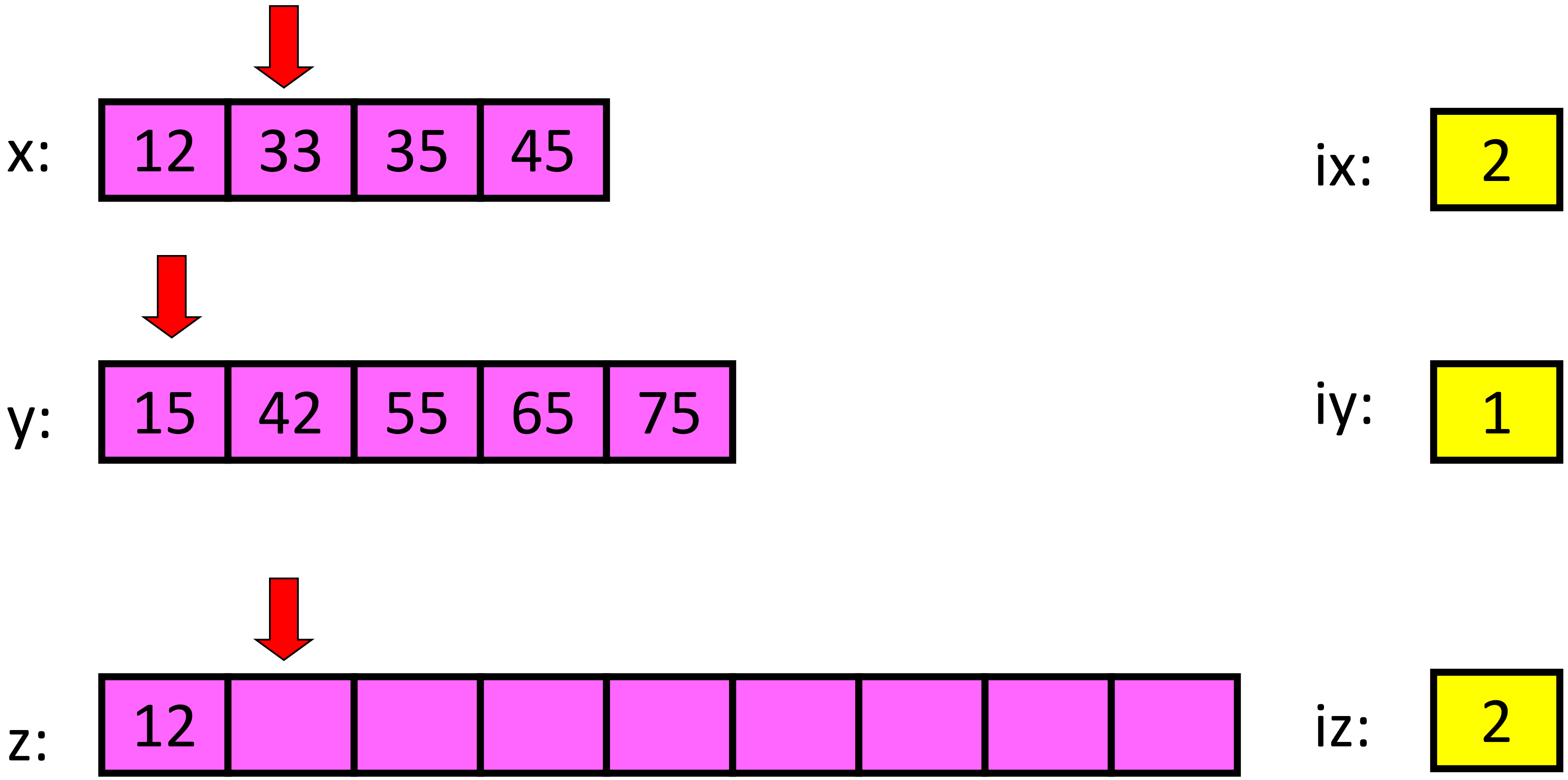
ix<=4 and iy<=5: x(ix) <= y(iy) ???

Merge



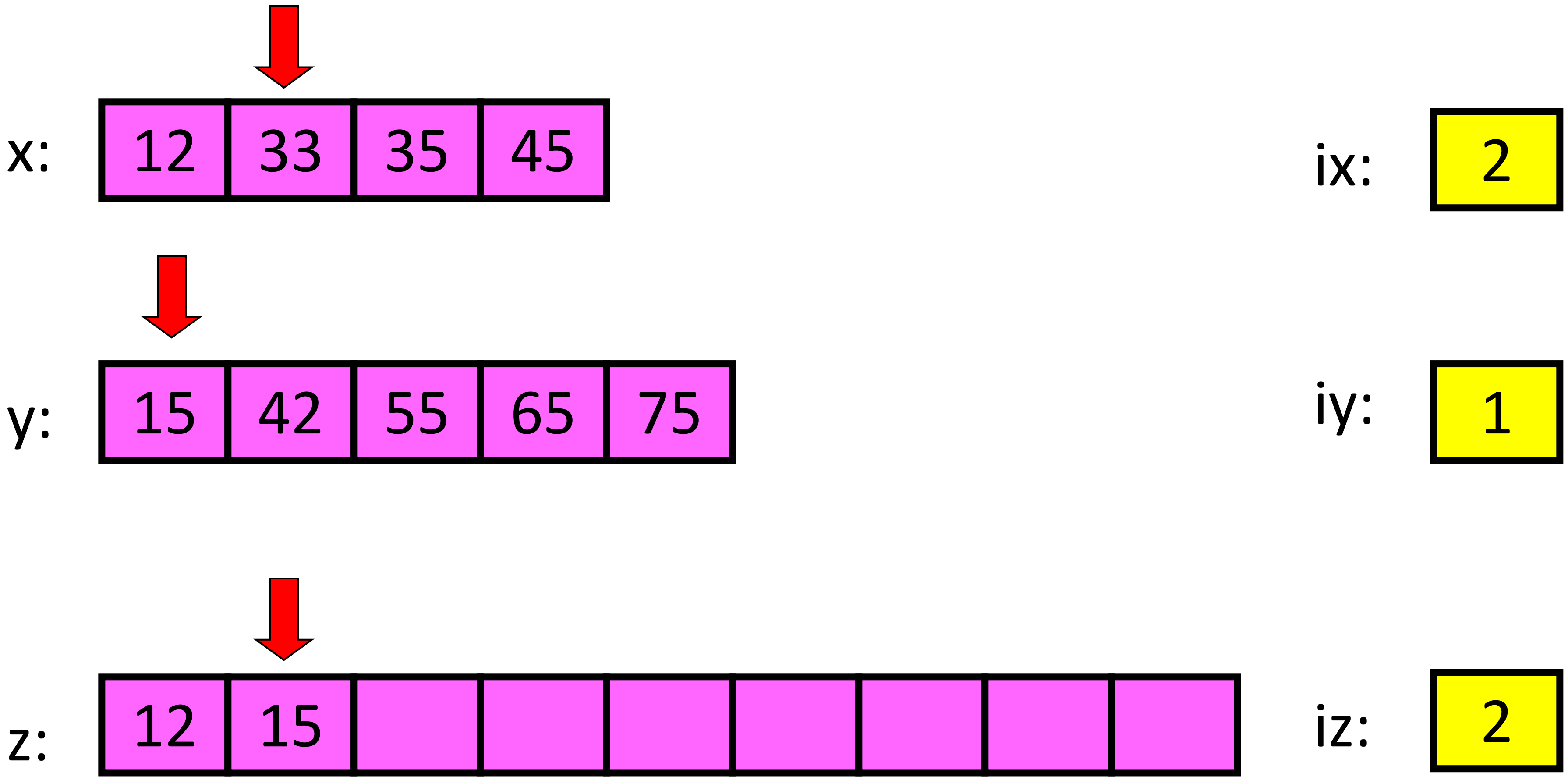
ix<=4 and iy<=5: x(ix) <= y(iy) YES

Merge



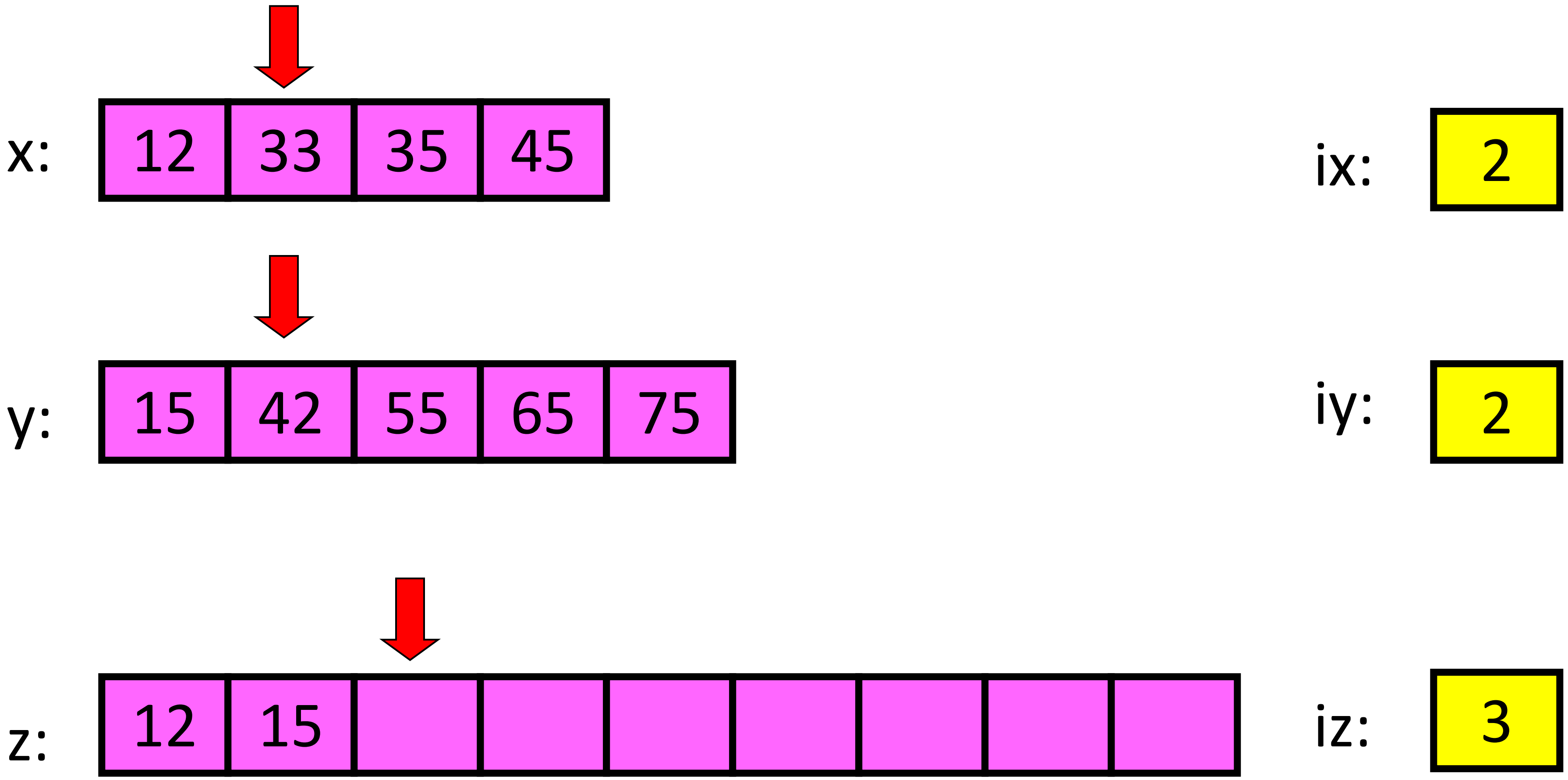
ix<=4 and iy<=5: x(ix) <= y(iy) ???

Merge



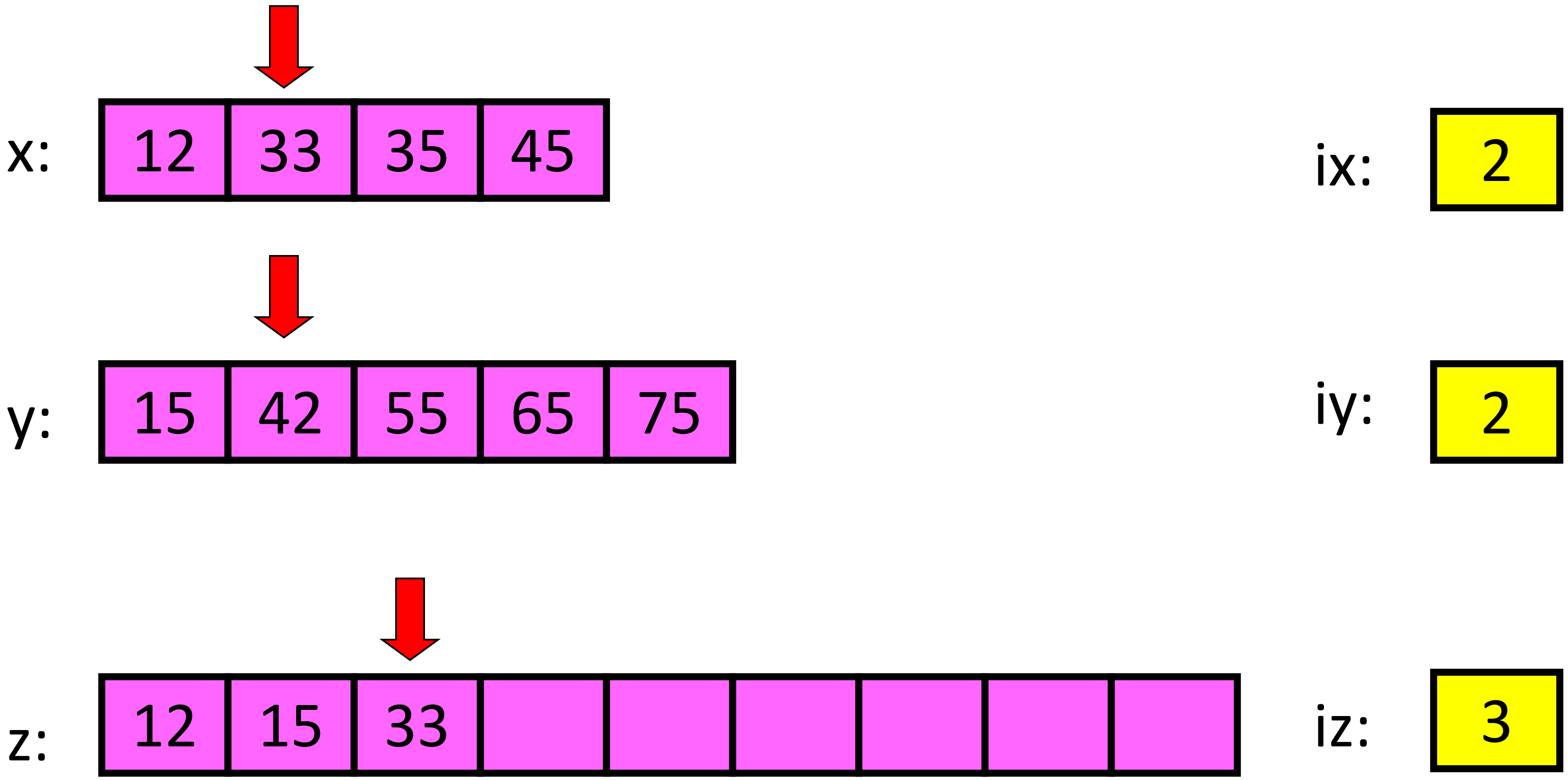
ix<=4 and iy<=5: x(ix) <= y(iy) NO

Merge



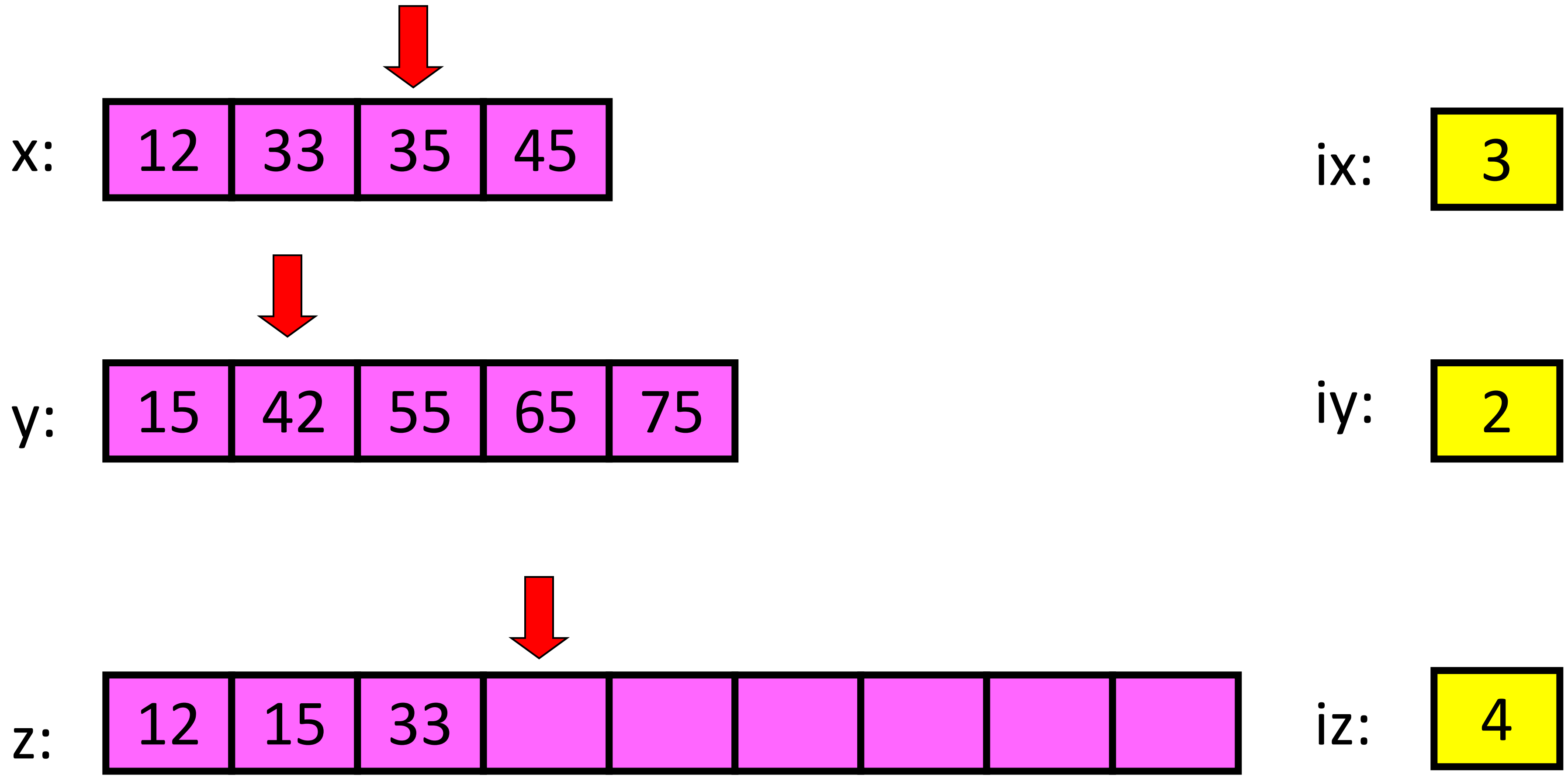
ix<=4 and iy<=5: x(ix) <= y(iy) ???

Merge



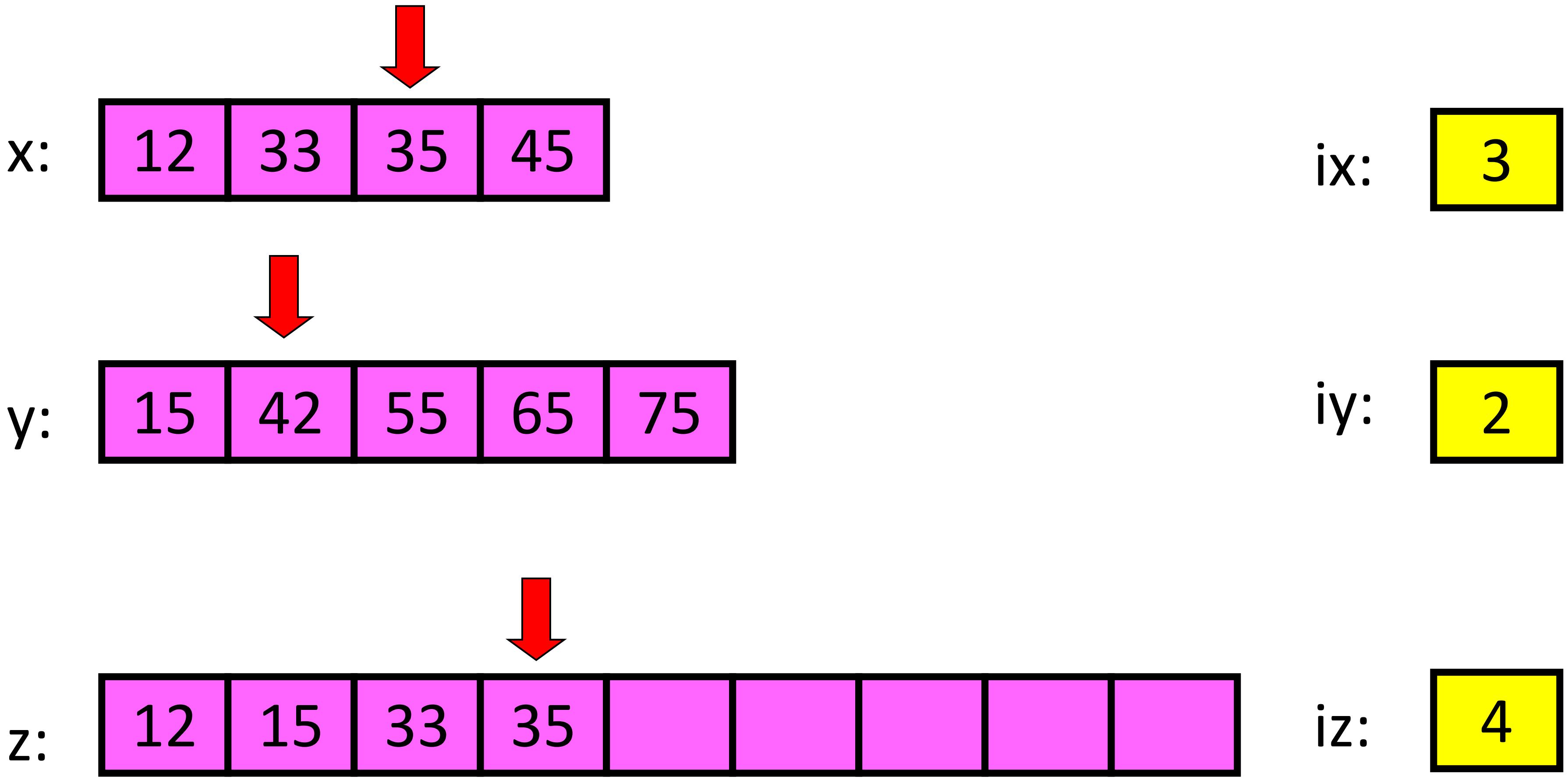
ix<=4 and iy<=5: x(ix) <= y(iy) YES

Merge



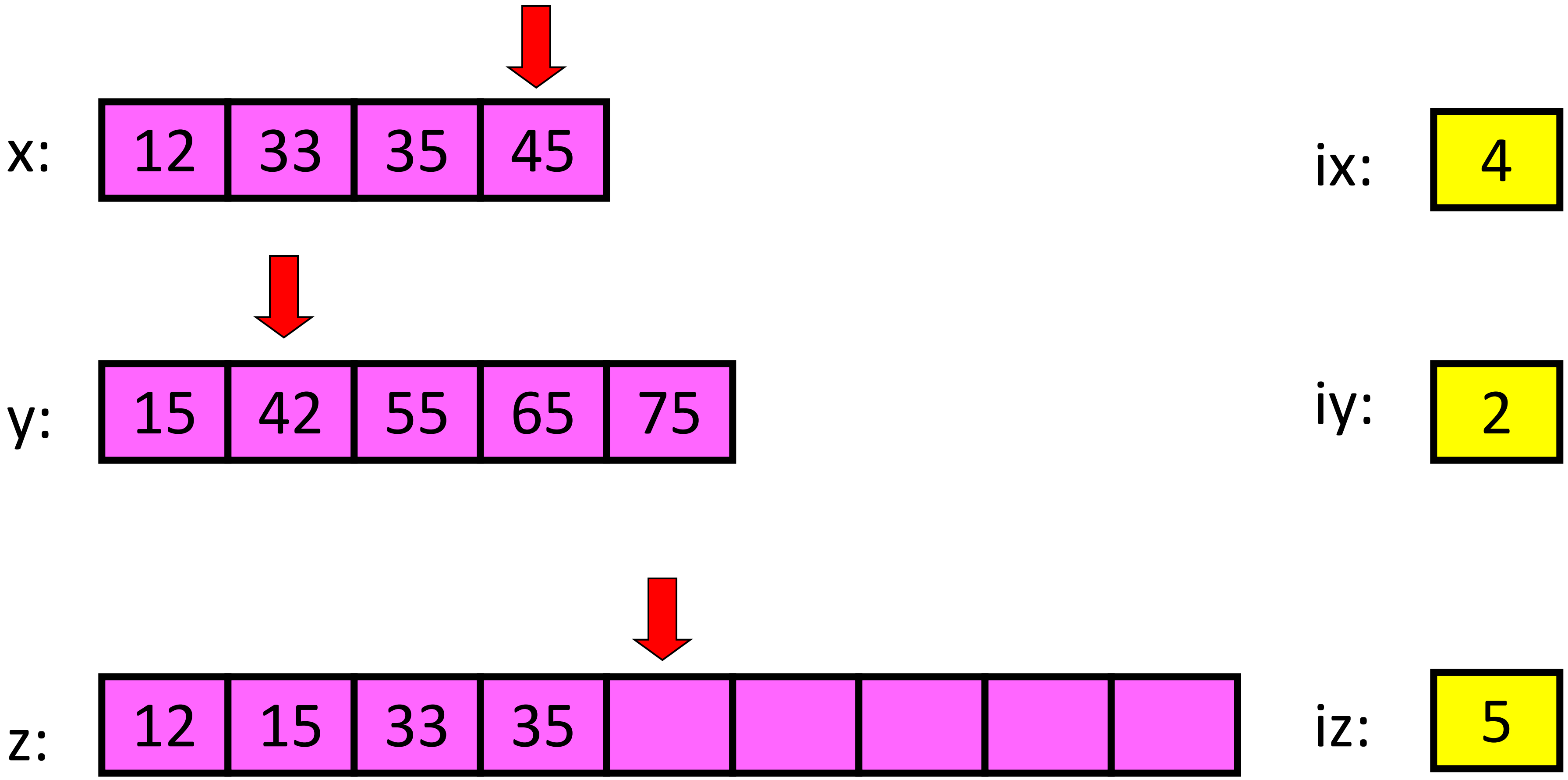
ix<=4 and iy<=5: x(ix) <= y(iy) ???

Merge



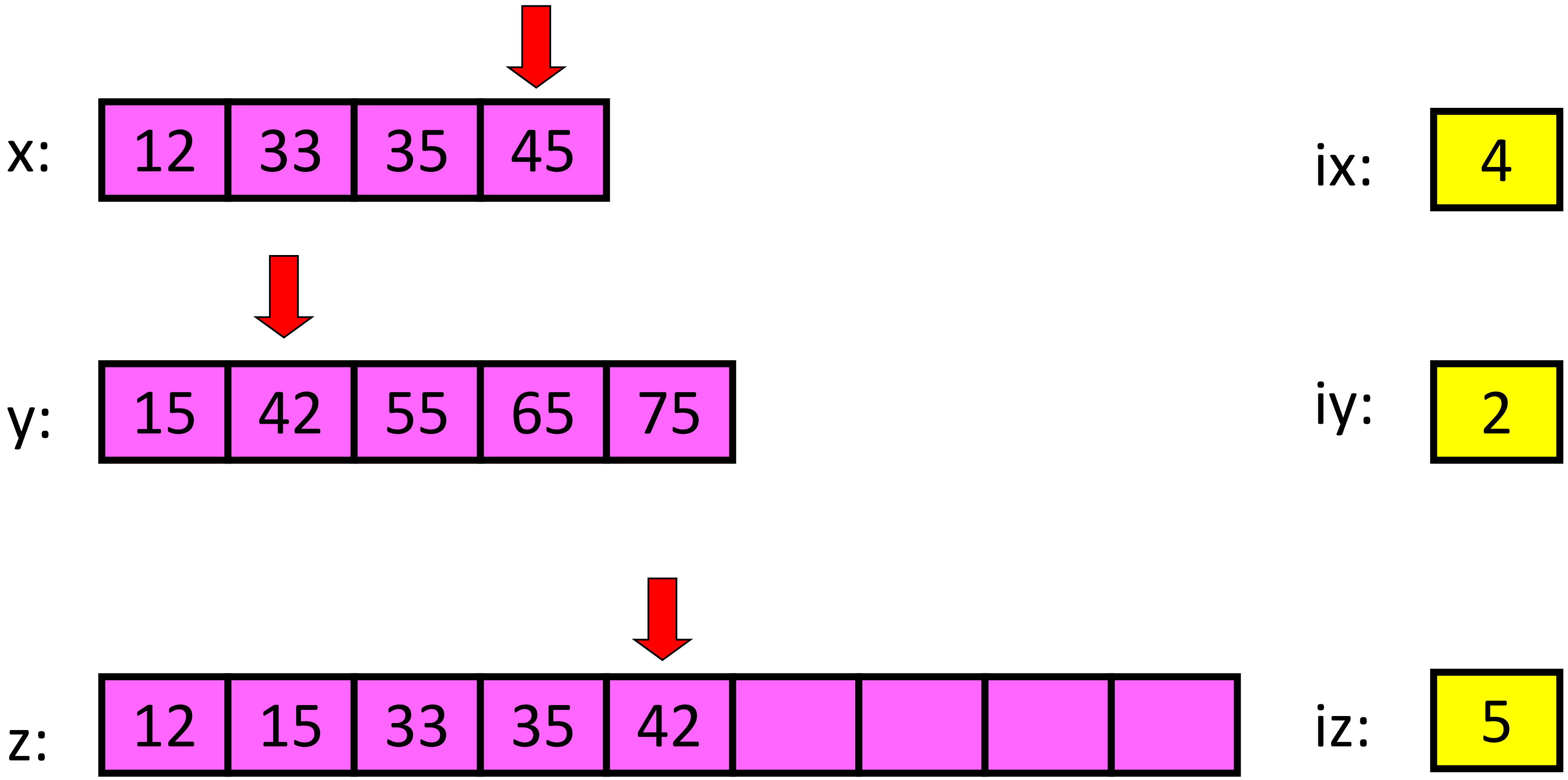
ix<=4 and iy<=5: x(ix) <= y(iy) YES

Merge



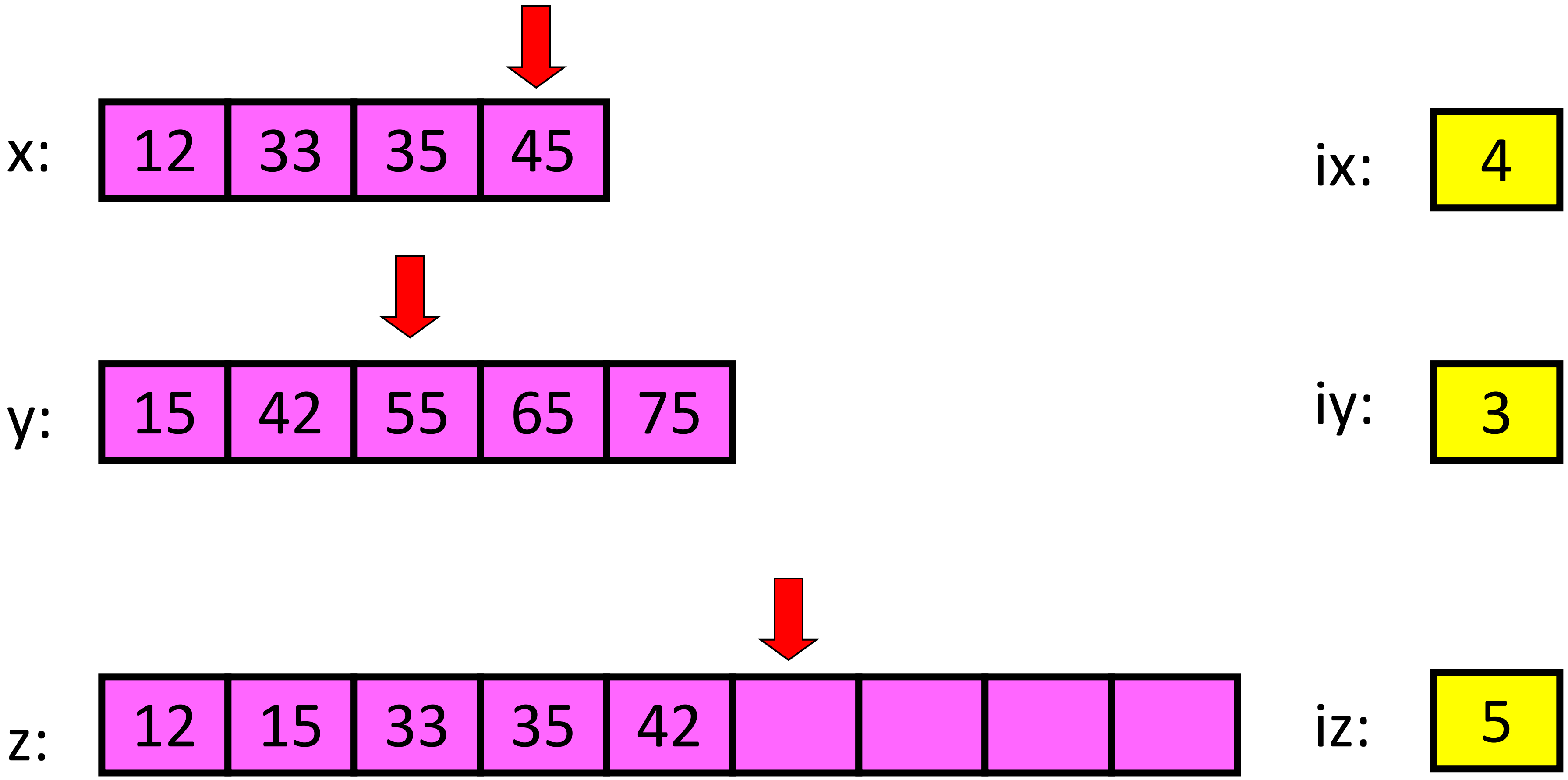
ix<=4 and iy<=5: x(ix) <= y(iy) ???

Merge



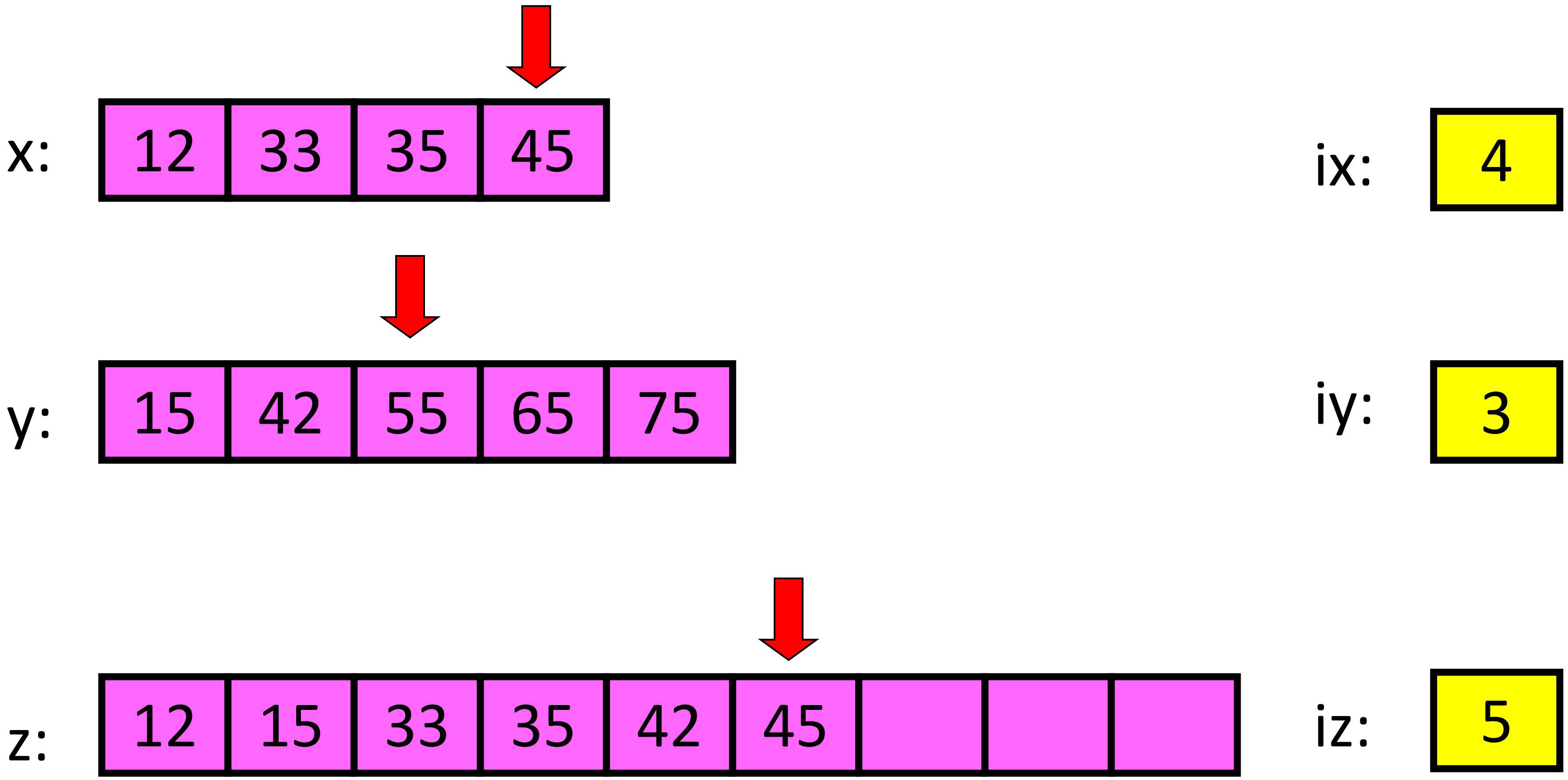
ix<=4 and iy<=5: x(ix) <= y(iy) NO

Merge



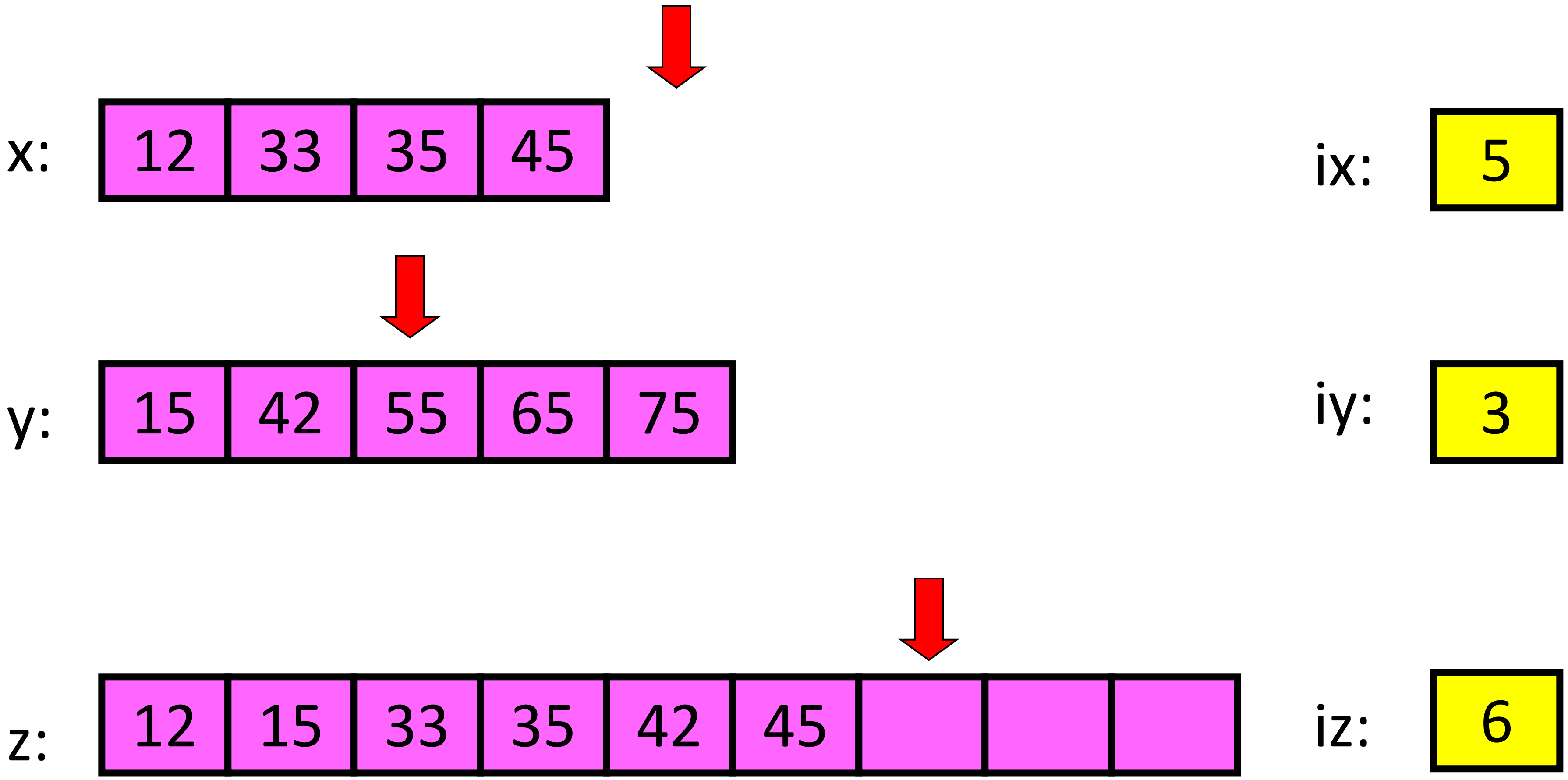
ix<=4 and iy<=5: x(ix) <= y(iy) ???

Merge



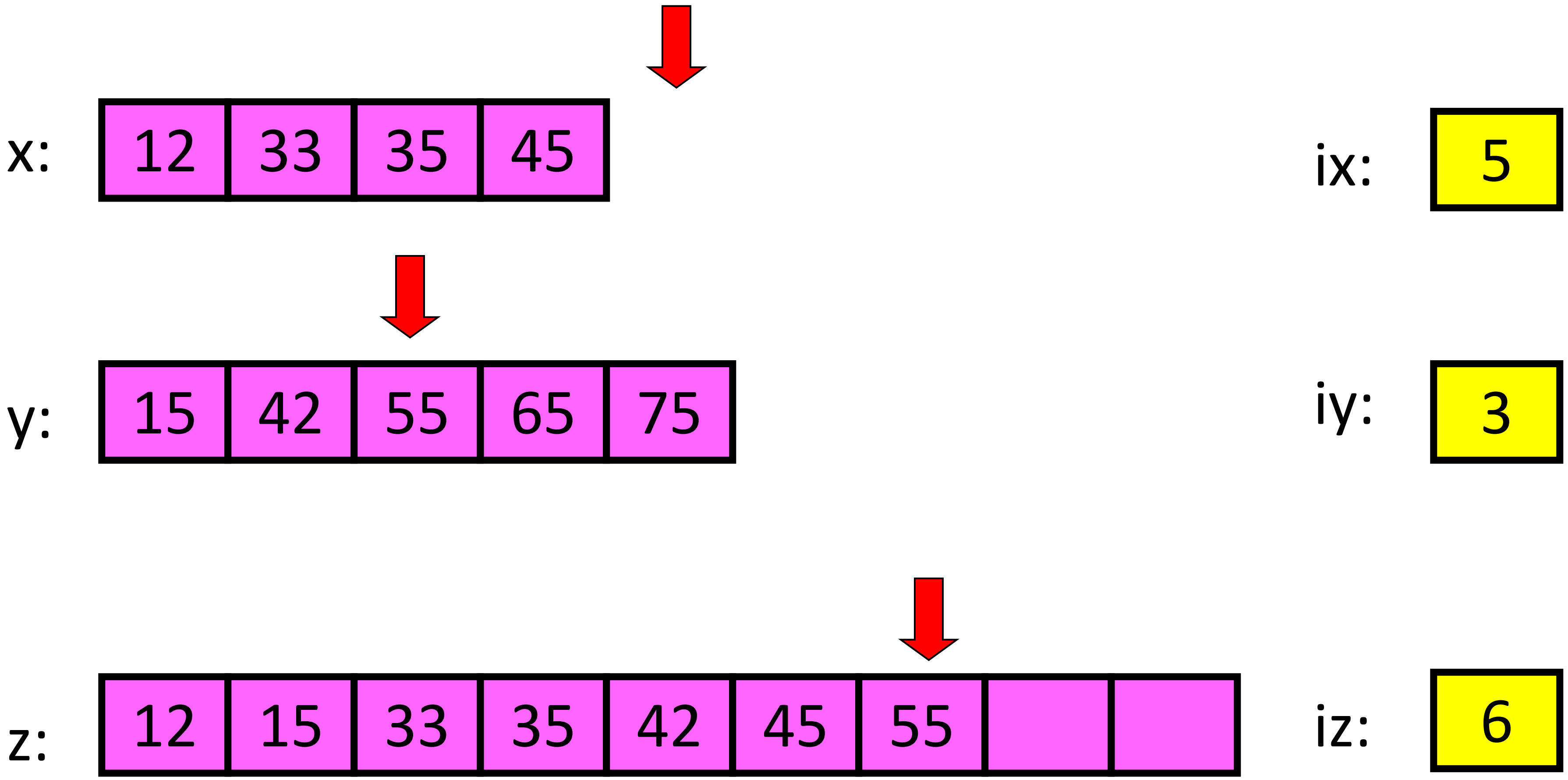
ix<=4 and iy<=5: x(ix) <= y(iy) YES

Merge



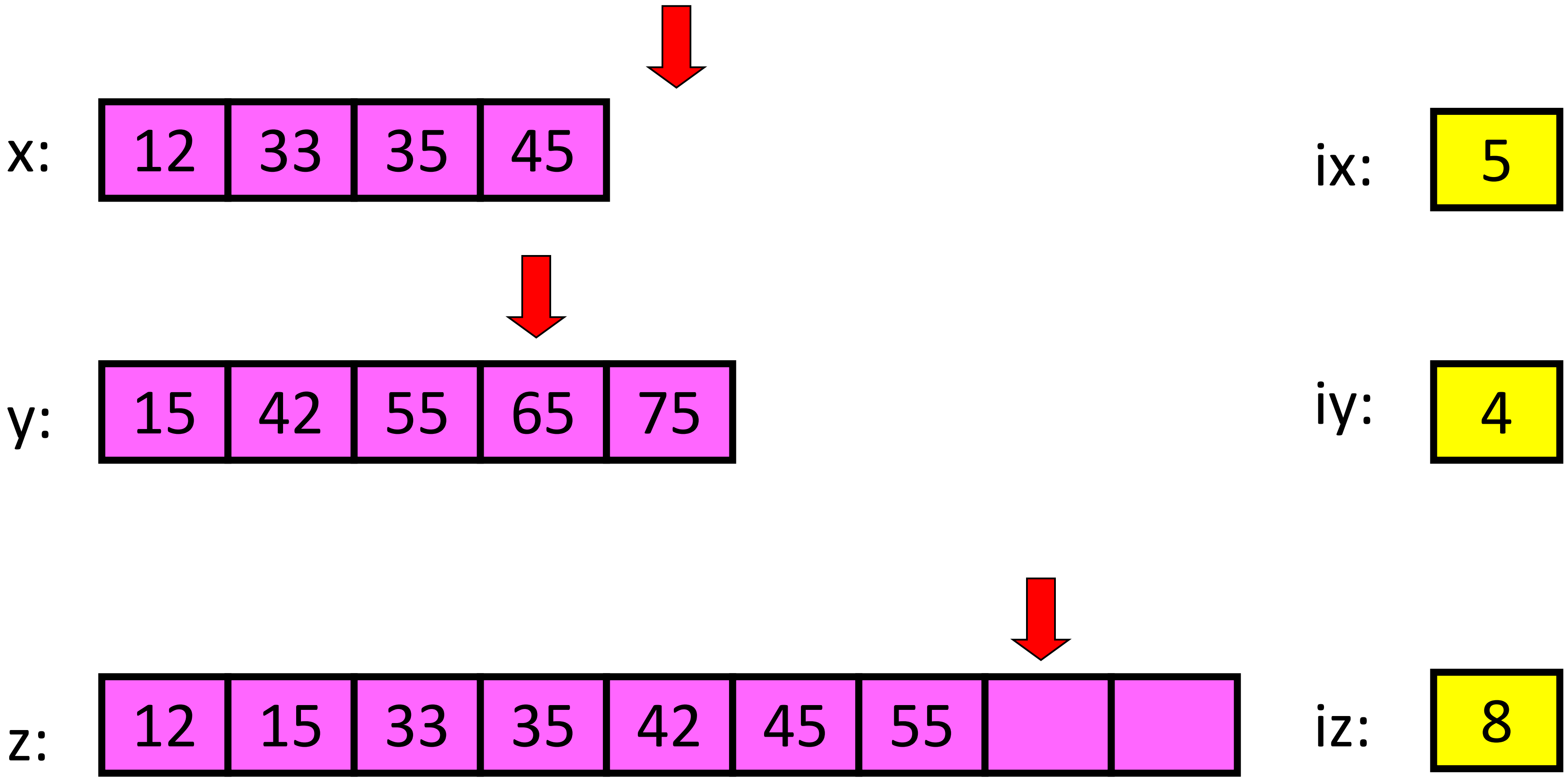
ix > 4

Merge



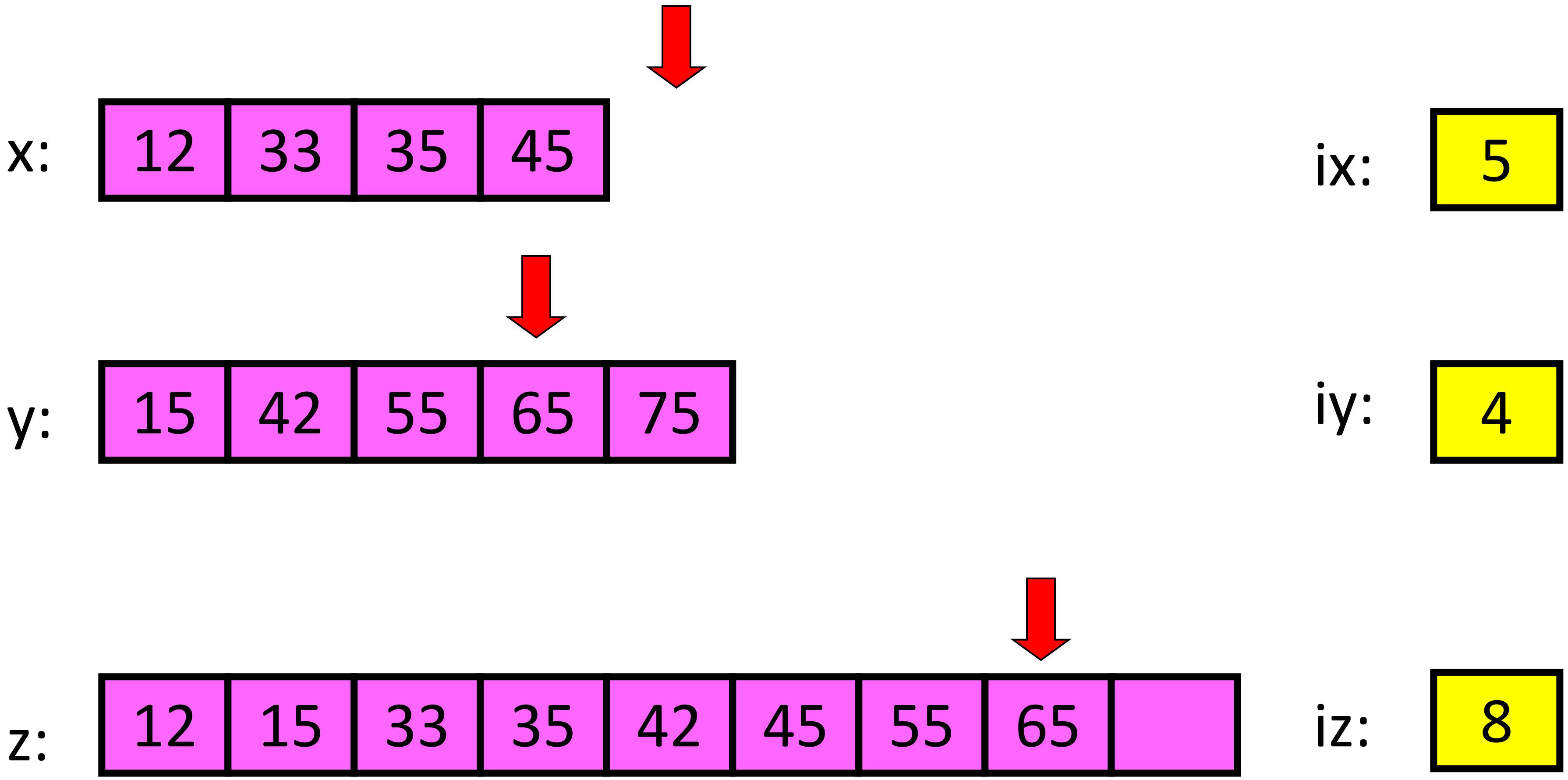
ix > 4: take y(iy)

Merge



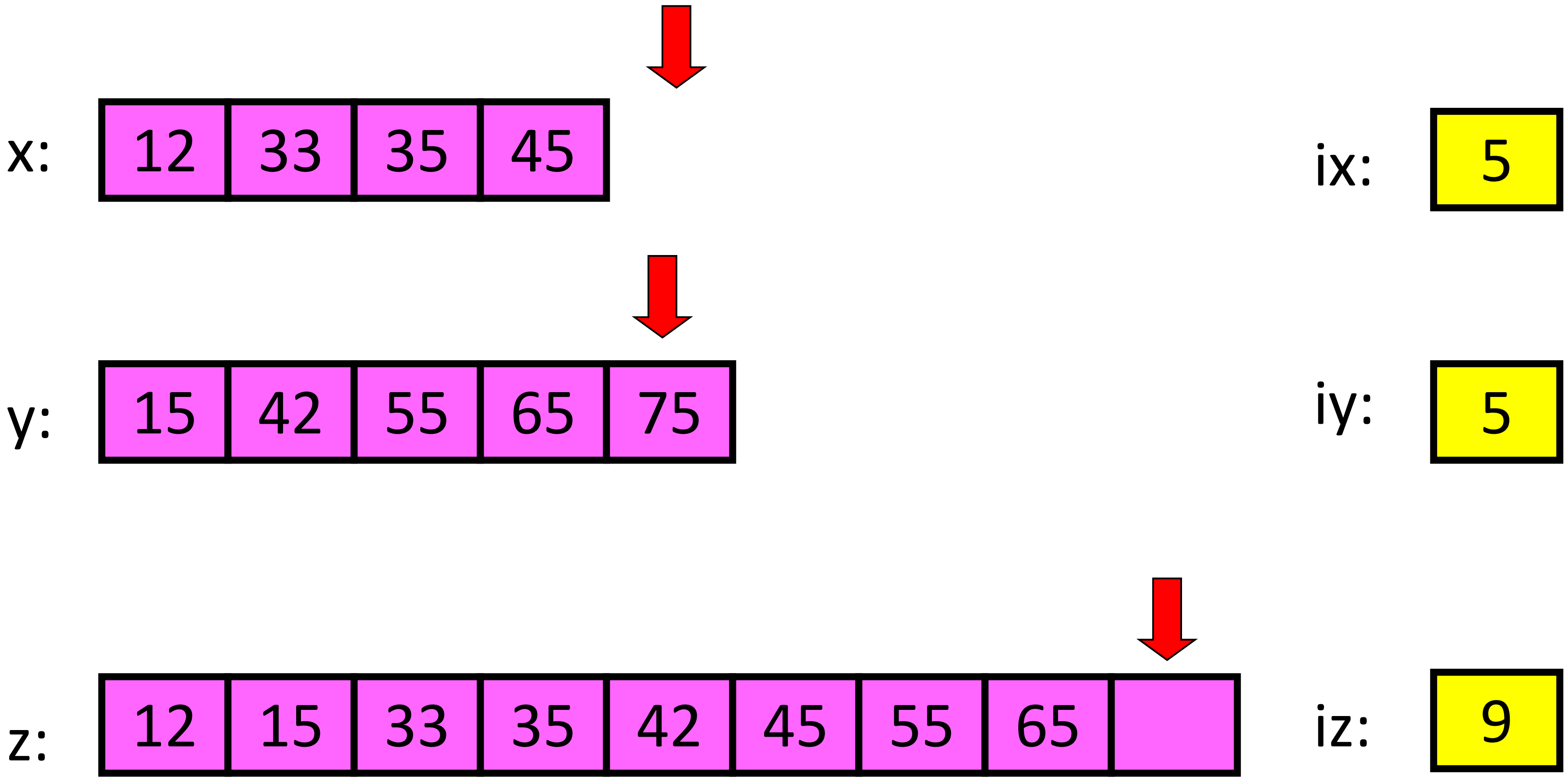
iy <= 5

Merge



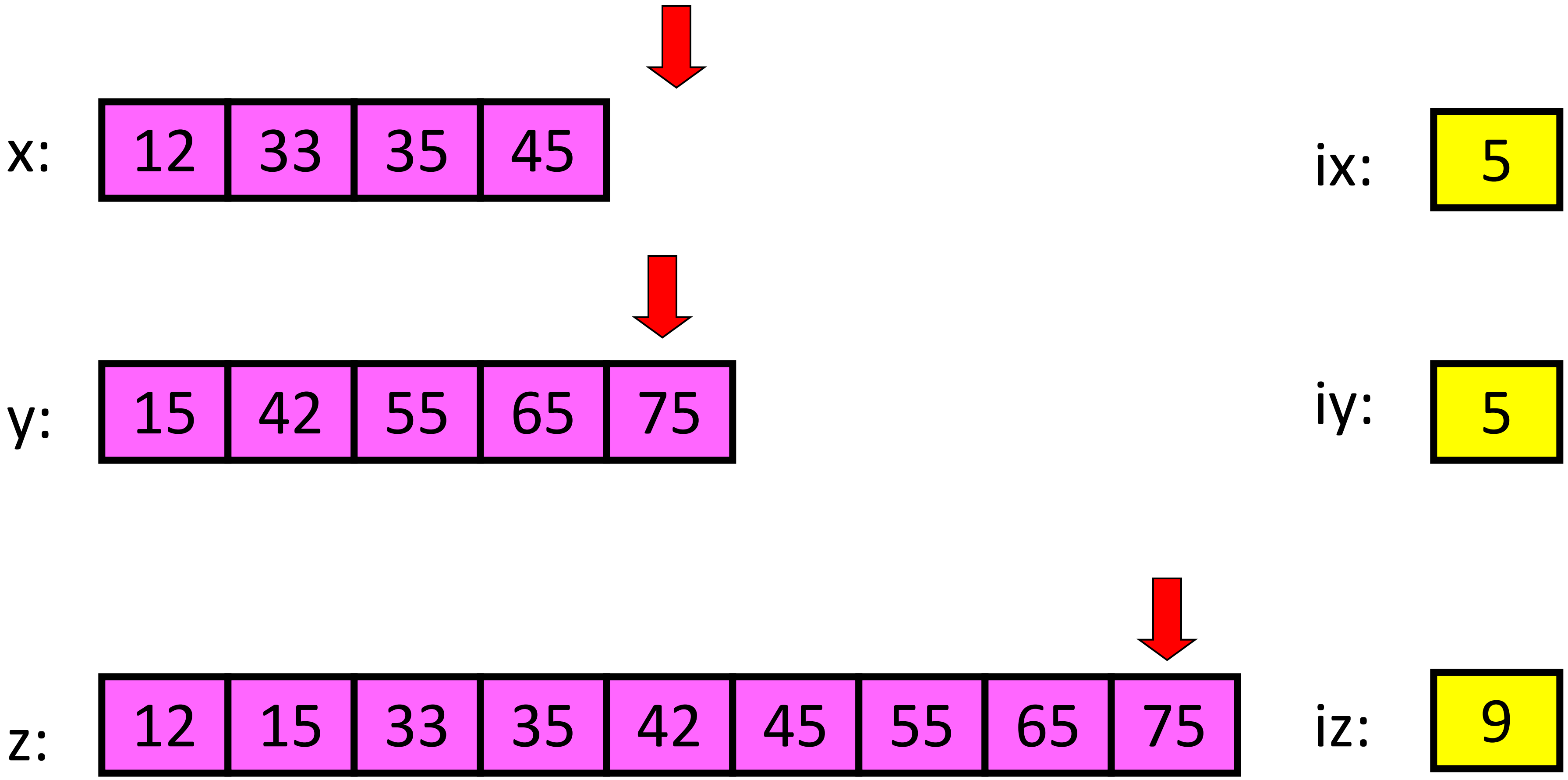
iy <= 5

Merge



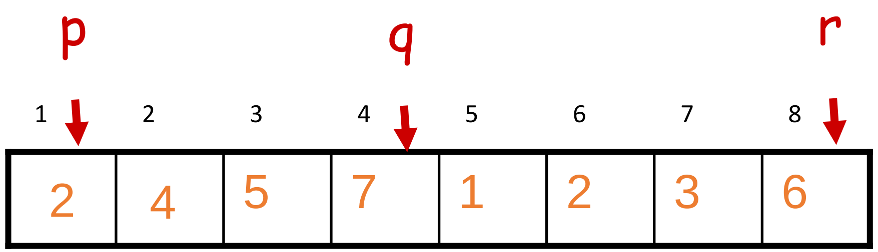
iy <= 5

Merge



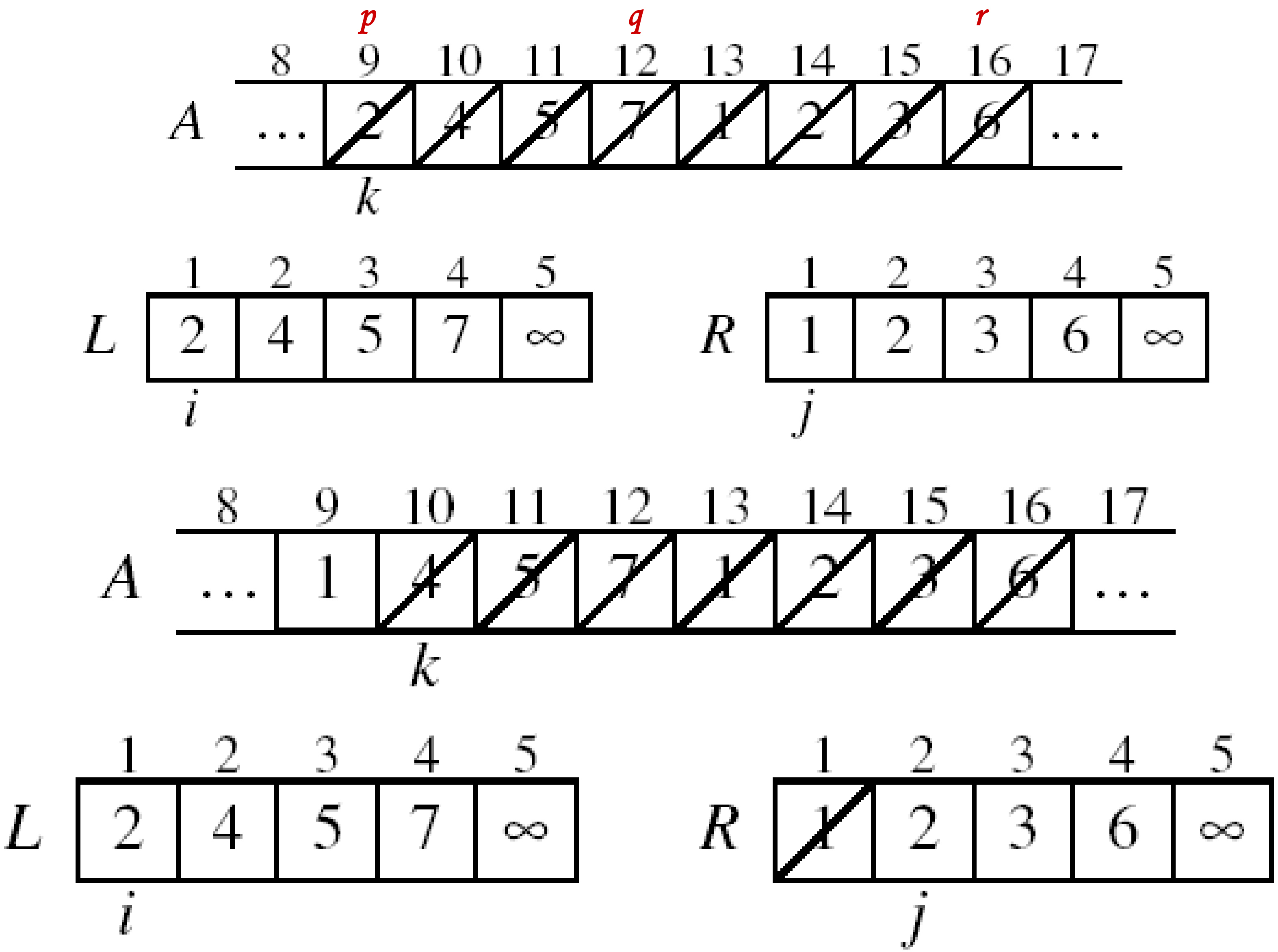
iy <= 5

Merging

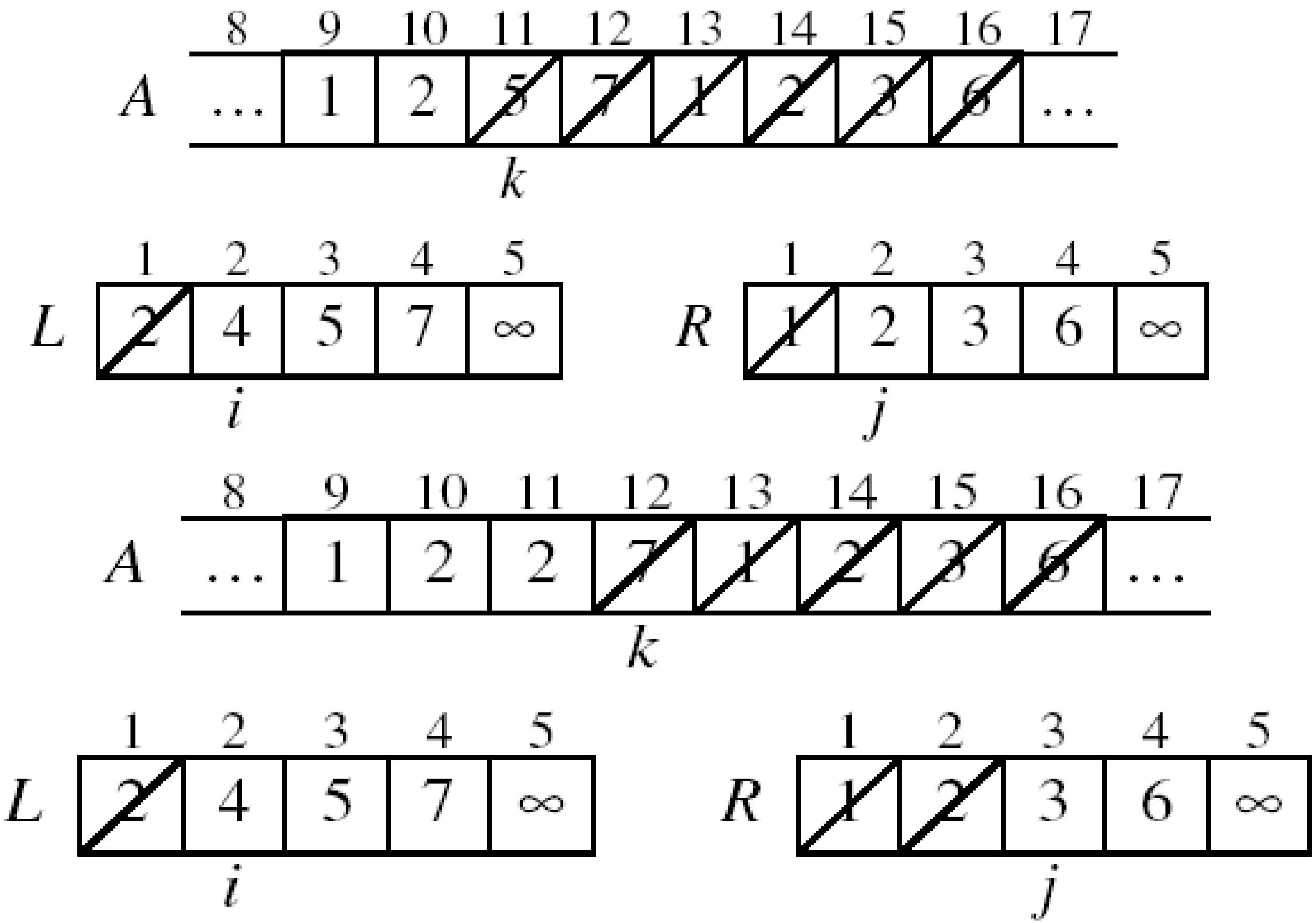


- **Input:** Array A and indices p, q, r such that $p \leq q < r$
  - Subarrays $A[p \,..\, q]$ and $A[q+1 \,..\, r]$ are sorted
- **Output:** One single sorted subarray $A[p \,..\, r]$

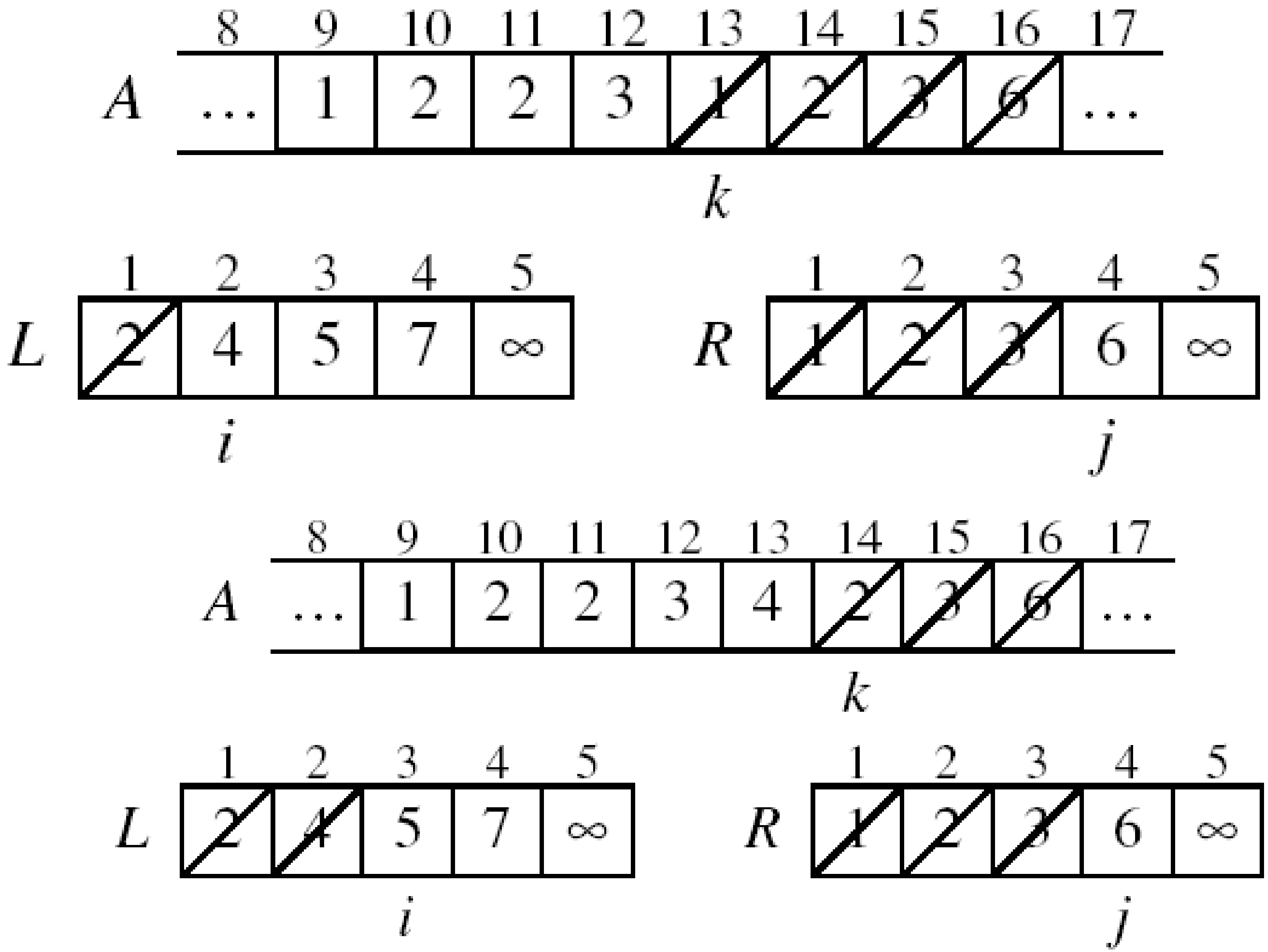
Example: MERGE(A, 9, 12, 16)



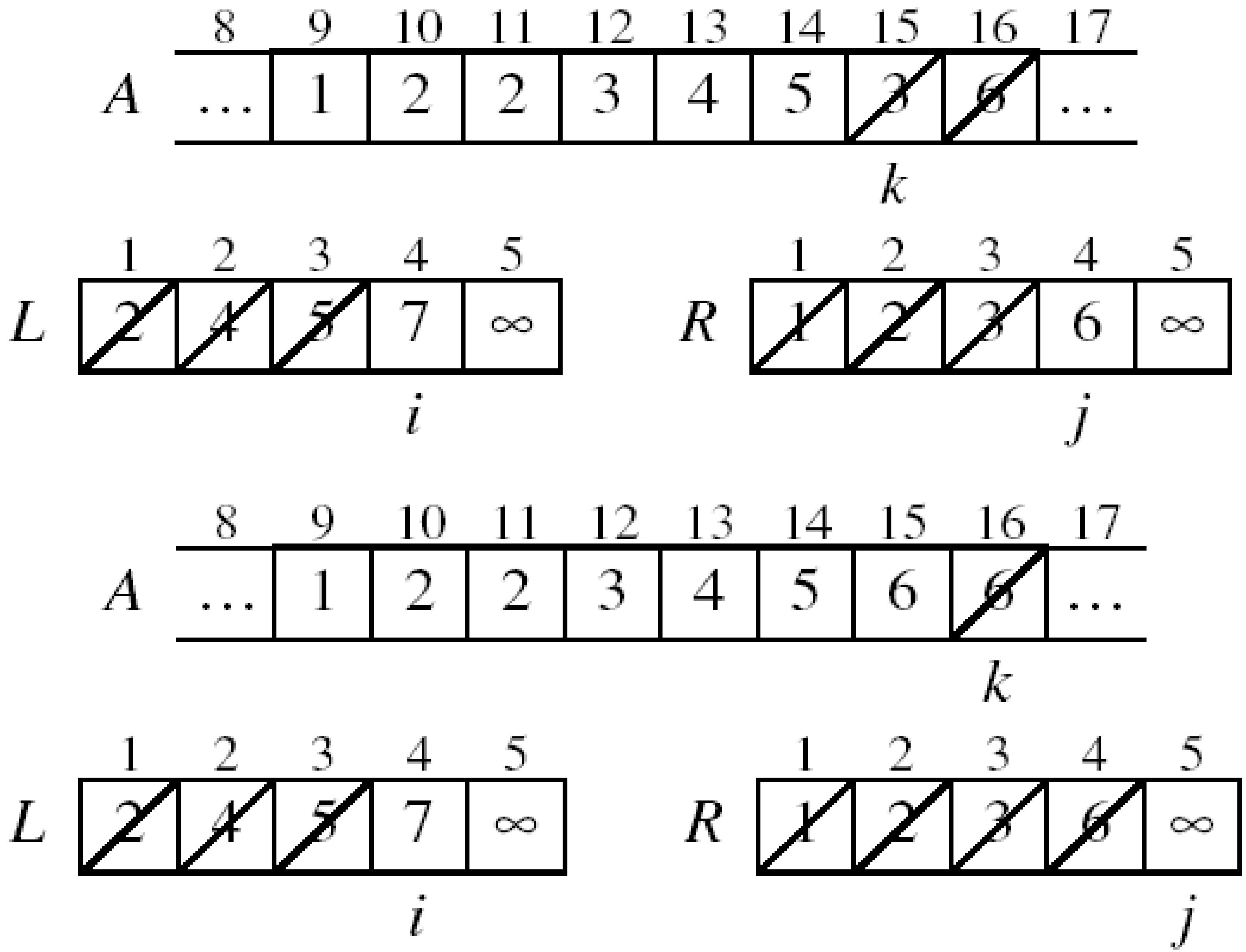
Example: MERGE(A, 9, 12, 16)



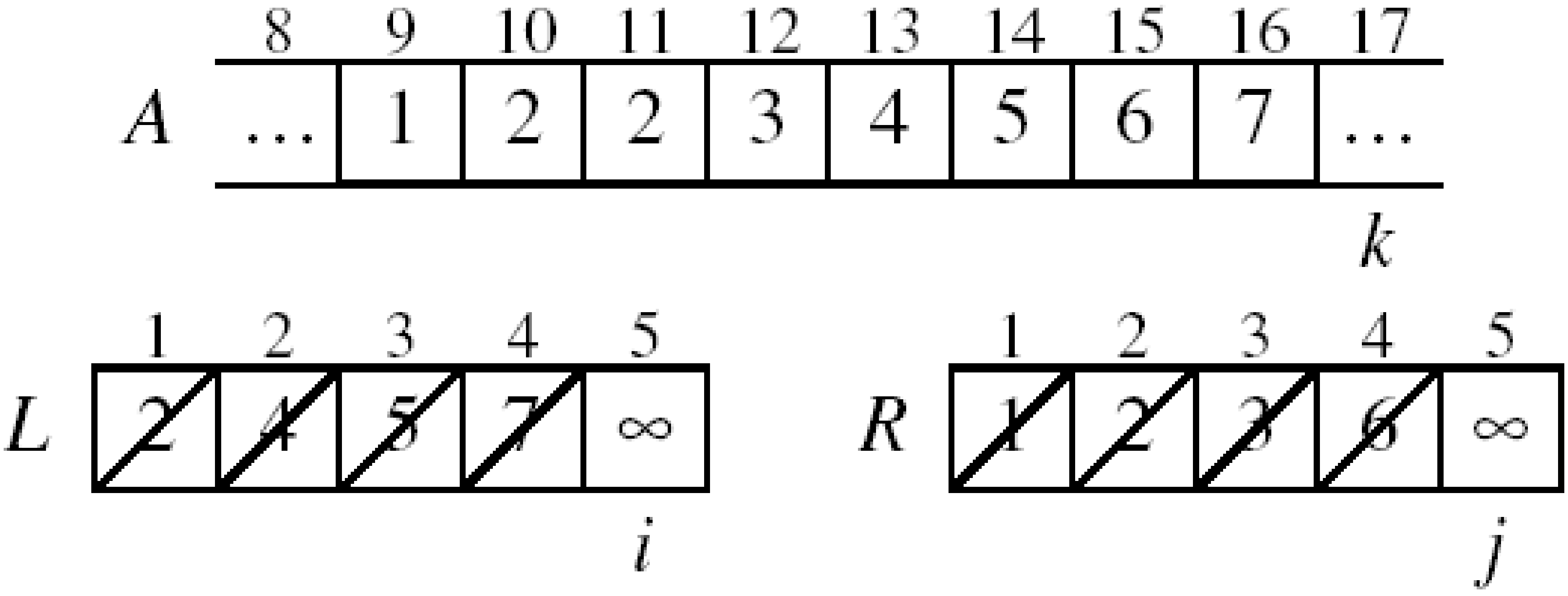
Example (cont.)



Example (cont.)

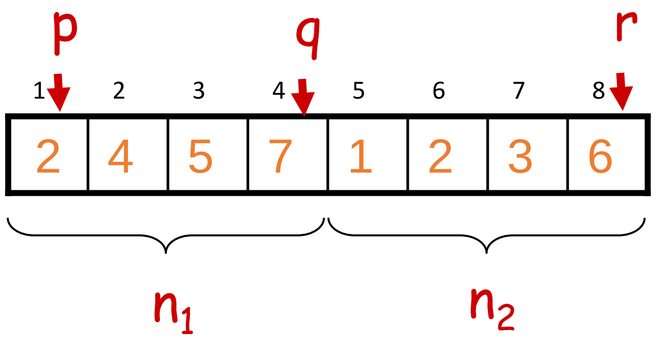


Example (cont.)



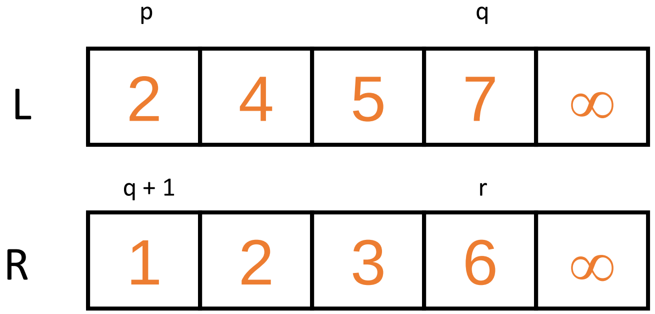
Done!

Merge - Pseudocode



Alg.: MERGE(A, p, q, r)

1. Compute n_1 and n_2
2. Copy the first n_1 elements into $L[1\ ..\ n_1 + 1]$ and the next n_2 elements into $R[1\ ..\ n_2 + 1]$
3. $L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$
4. $i \leftarrow 1$; $j \leftarrow 1$
5. **for** $k \leftarrow p$ **to** r
6.   **do if** $L[\ i\ ] \leq R[\ j\ ]$
7.   **then** $A[k] \leftarrow L[\ i\ ]$
8. $i \leftarrow i + 1$
9.   **else** $A[k] \leftarrow R[\ j\ ]$
10. $j \leftarrow j + 1$



Merge function

```
void merge(int A[], int p, int q, int r)
{
/* Create L ← A[p..q] and M ← A[q+1..r] */
    int n1 = q - p + 1;
    int n2 =  r - q;
     int L[n1], M[n2];
    for (i = 0; i < n1; i++)
        L[i] = A[p + i];
    for (j = 0; j < n2; j++)
        M[j] = A[q + 1 + j];
  /* Maintain current index of sub-arrays and
main array */
    int i, j, k;
    i = 0;
    j = 0;
    k = p;
/* Until we reach either end of L or M,
pick larger among elements L and M and
place them in the correct position at
A[p..r] */
    while (i < n1 && j < n2)
    {
        if (L[i] <= M[j])
        {
            arr[k] = L[i];
            i++;
        }
    else
        {
            arr[k] = M[j];
            j++;
        }
        k++;
    }
```

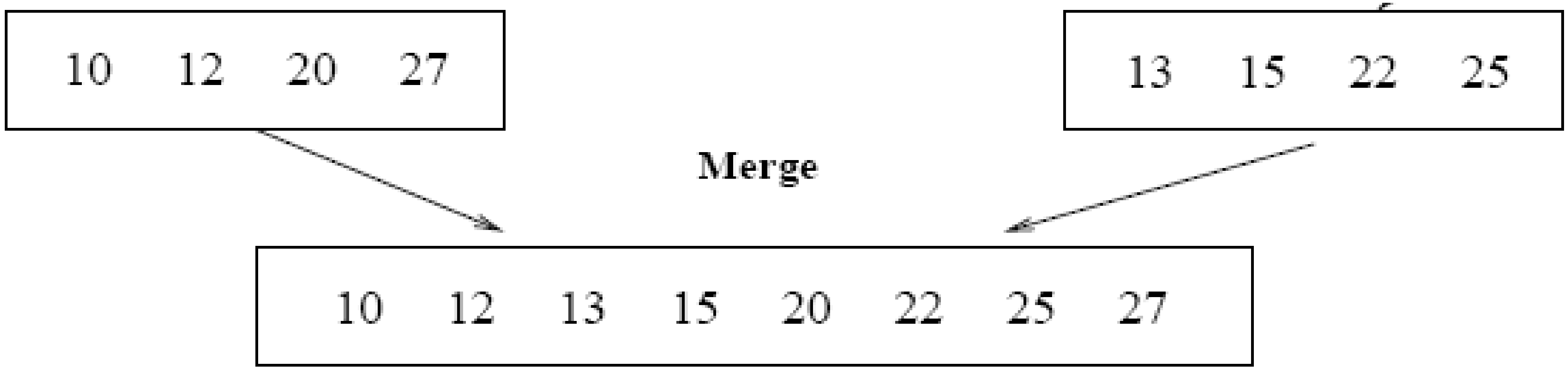
Merge function

```
/* When we run out of elements in either L or M, pick up the remaining elements
and put in A[p..r] */
    while (i < n1)
    {
        A[k] = L[i];
        i++;
        k++;
    }
     while (j < n2)
    {
        A[k] = M[j];
        j++;
        k++;
    }
}
```

```
void Merge(int A[], int temp[], int left, int mid, int right) {
        int i, left_end, size, temp_pos;
        left_end = mid - 1;
        temp_pos = left;
        size = right - left + 1;
        while ((left <= left_end) && (mid <= right)) {
                if(A[left] <= A[mid]) {
                        temp[temp_pos] = A[left];
                        temp_pos = temp_pos + 1;
                        left = left +1;
                }
                else {  temp[temp_pos] = A[mid];
                        temp_pos = temp_pos + 1;
                        mid = mid + 1;
                }
        }
while (left <= left_end) {
        temp[temp_pos] = A[left];
        left = left + 1;
        temp_pos = temp_pos + 1;
}
while (mid <= right) {
        temp[temp_pos] = A[mid];
        mid = mid + 1;
        temp_pos = temp_pos + 1;
}
for (i = 0; i <= size; i++) {
        A[right] = temp[right];
        right = right - 1;
}
```

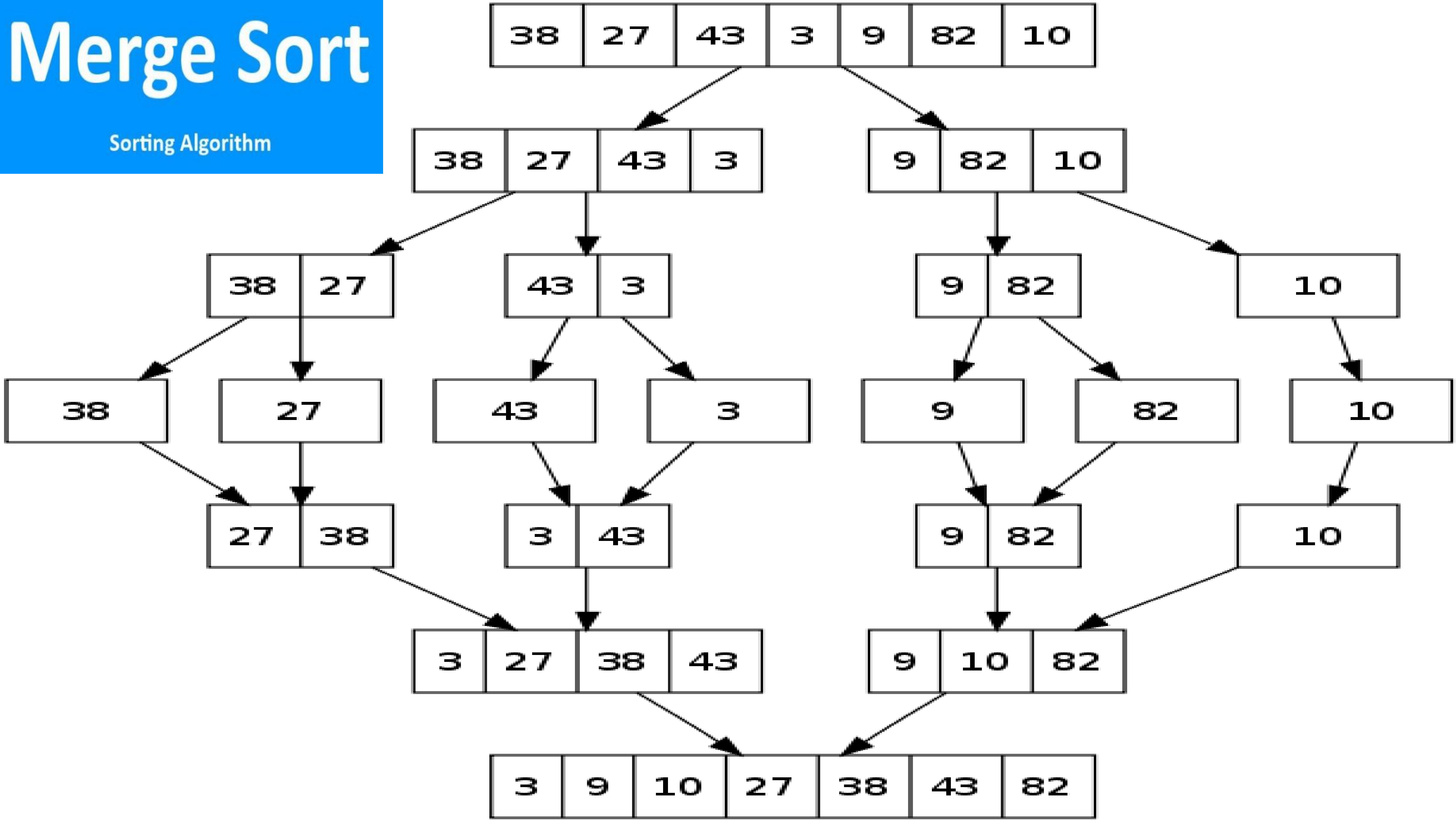
Running Time of Merge

- Initialization (copying into temporary arrays):
  - $\Theta(n_1 + n_2) = \Theta(n)$
- Adding the elements to the final array:
  - n iterations, each taking constant time $\Rightarrow \Theta(n)$
- Total time for Merge: $\Theta(n)$

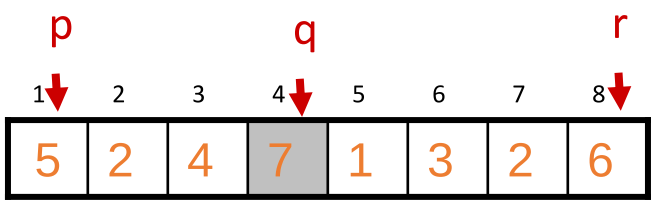


Merge Sort

Sorting Algorithm



Merge Sort Algorithm




```
Alg.: MERGE-SORT(A, p, r)

if p < r                                 ▷ Check for base case

   then q ← ⌊(p + r)/2⌋                  ▷ Divide

        MERGE-SORT(A, p, q)              ▷ Conquer

        MERGE-SORT(A, q + 1, r)          ▷ Conquer

        MERGE(A, p, q, r)                ▷ Combine
```

```
void Mergesort(int A[], int temp[], int left, int right) {
        int mid;
        if(right > left) {
                mid = (right + left) / 2;
                Mergesort(A, temp, left, mid);
                Mergesort(A, temp, mid+1, right);
                Merge(A, temp, left, mid+1, right);
        }
}
```

MERGE-SORT Running Time

- **Divide:**
  - compute q as the average of p and r: $D(n) = \Theta(1)$
- **Conquer:**
  - recursively solve 2 sub-problems, each of size $n/2 \Rightarrow 2T(n/2)$
- **Combine:**
  - MERGE on an n-element subarray takes $\Theta(n)$ time: $C(n) = \Theta(n)$

$$T(n) = \begin{cases} 2T(n/2) + \Theta(n) & \text{if } n > 1 \\ \Theta(1) & \text{if } n = 1 \end{cases}$$

Solve the Recurrence Relation of Merge Sort

$$T(n) = \begin{cases} 2T(n/2) + \Theta(n) & \text{if } n > 1 \\ \Theta(1) & \text{if } n = 1 \end{cases}$$

Above function can be rewritten as :

$$T(n) = \begin{cases} 2T(n/2) + cn & \text{if } n > 1 \\ c & \text{if } n = 1 \end{cases}$$

Using Master's Theorem:

Compare n with $f(n) = cn$

Case 2: $T(n) = \Theta(n\log n)$

Merge Sort - Discussion

- Running time insensitive of the input

- Advantages:
  - Mergesort is extremely efficient algorithm with respect to time.
  - Guaranteed to run in $\Theta(nlogn)$

- Disadvantage
  - Mergesort requires an extra array whose size equals to the size of the original array.
  - So, it requires extra space $\approx N = O(n)$

QUICK SORT

- **Quicksort** (sometimes called **partition-exchange sort**) is an efficient sorting algorithm, serving as a systematic method for placing the elements of an array in order.
- Quicksort can operate in-place on an array, requiring small additional amounts of memory to perform the sorting.
- It is very similar to selection sort, except that it does not always choose worst-case partition.
- Mathematical analysis of quicksort shows that, on average, the algorithm takes $O(n \log n)$ comparisons to sort n items.
- In the worst case, it makes $O(n^2)$ comparisons, though this behaviour is rare.

Quicksort Procedure

- Select a pivot which divide or partition the list in two part.
  - e.g., pivot = A[n-1] or A [1] or any random element
- Rearrange the list so that it starts with the pivot followed by a ≤ sublist (a sublist whose elements are all smaller than or equal to the pivot) and a ≥ sublist (a sublist whose elements are all greater than or equal to the pivot)
- Exchange the pivot with the last element in the first sublist, then the pivot will be now in its final position
- Sort the two sublists recursively using quicksort.

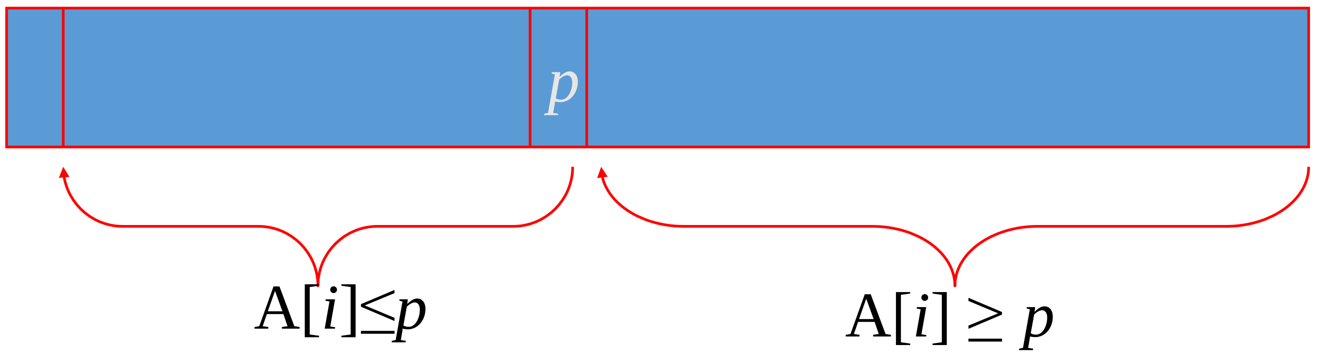
Summary :

Step 1 – Make the right-most index value pivot

Step 2 – partition the array using pivot value

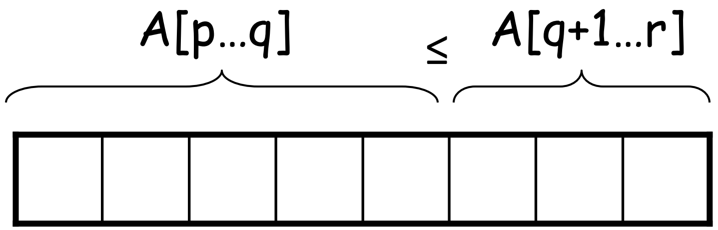
Step 3 – quicksort left partition recursively

Step 4 – quicksort right partition recursively



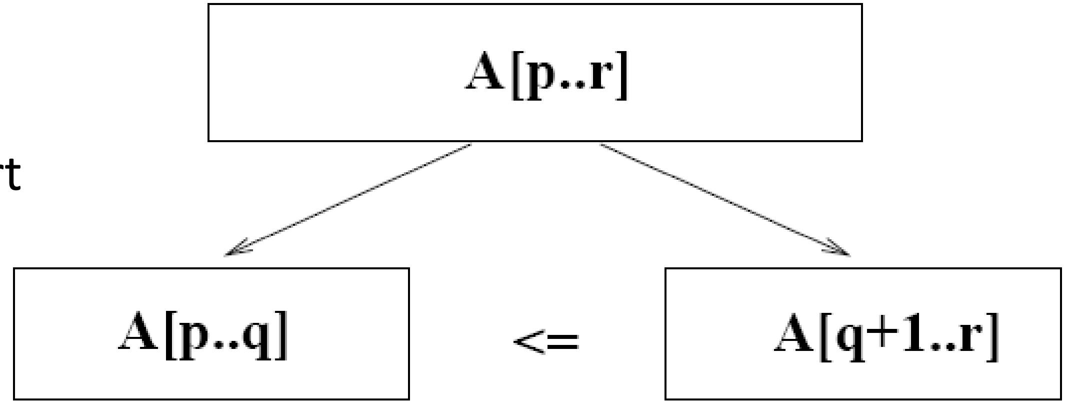
Quicksort

- Sort an array $A[p\ldots r]$



Divide

- Partition the array A into 2 subarrays $A[p..q]$ and $A[q+1..r]$, such that each element of $A[p..q]$ is smaller than or equal to each element in $A[q+1..r]$
- Need to find index q to partition the array



Conquer

Recursively sort $A[p..q]$ and $A[q+1..r]$ using Quicksort

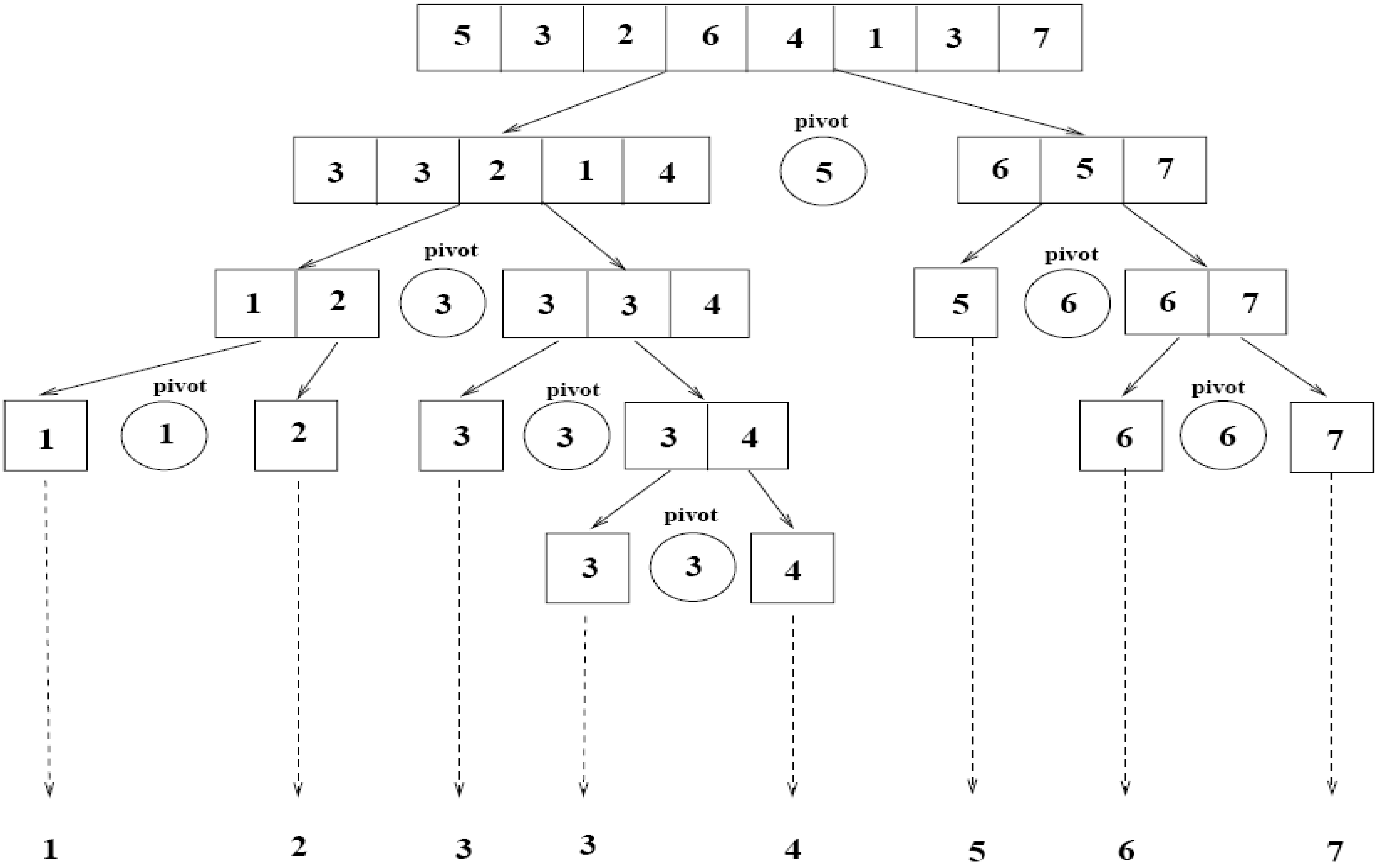
Combine

Trivial: the arrays are sorted in place

No additional work is required to combine them

The entire array is now sorted

Example



5 3 2 6 4 1 3 7
3 3 2 1 4
pivot
5
6 5 7
1 2
pivot
3
3 3 4
5
pivot
6
6 7
1
pivot
1
2
3
pivot
3
3 4
6
pivot
6
7
3
pivot
3
4
1 2 3 3 4 5 6 7

Partitioning the Array

Alg. PARTITION (A, p, r)

$x \leftarrow A[r]$ //selects an last element (r) as a ***pivot*** element

$i \leftarrow p - 1$

for $j = p$ to $r-1$

{

if $A[j] \leq x$

$i = i+1$

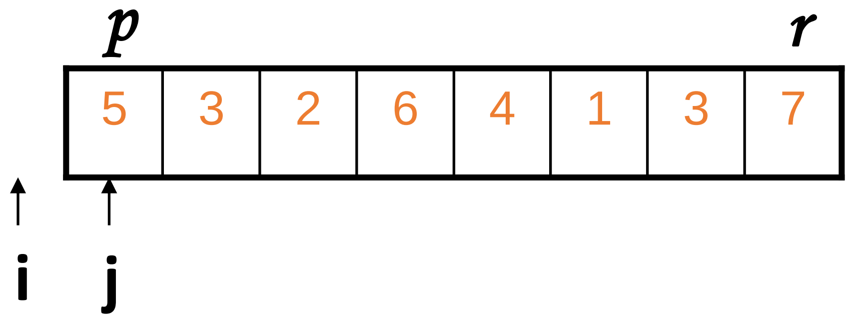
exchange $A[i] \leftrightarrow A[j]$

}

exchange $A[i+1] \leftrightarrow A[r]$

return $i + 1$

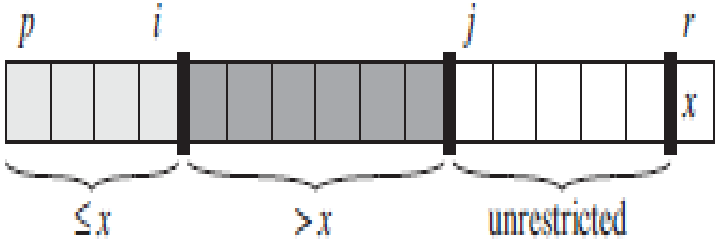
A:

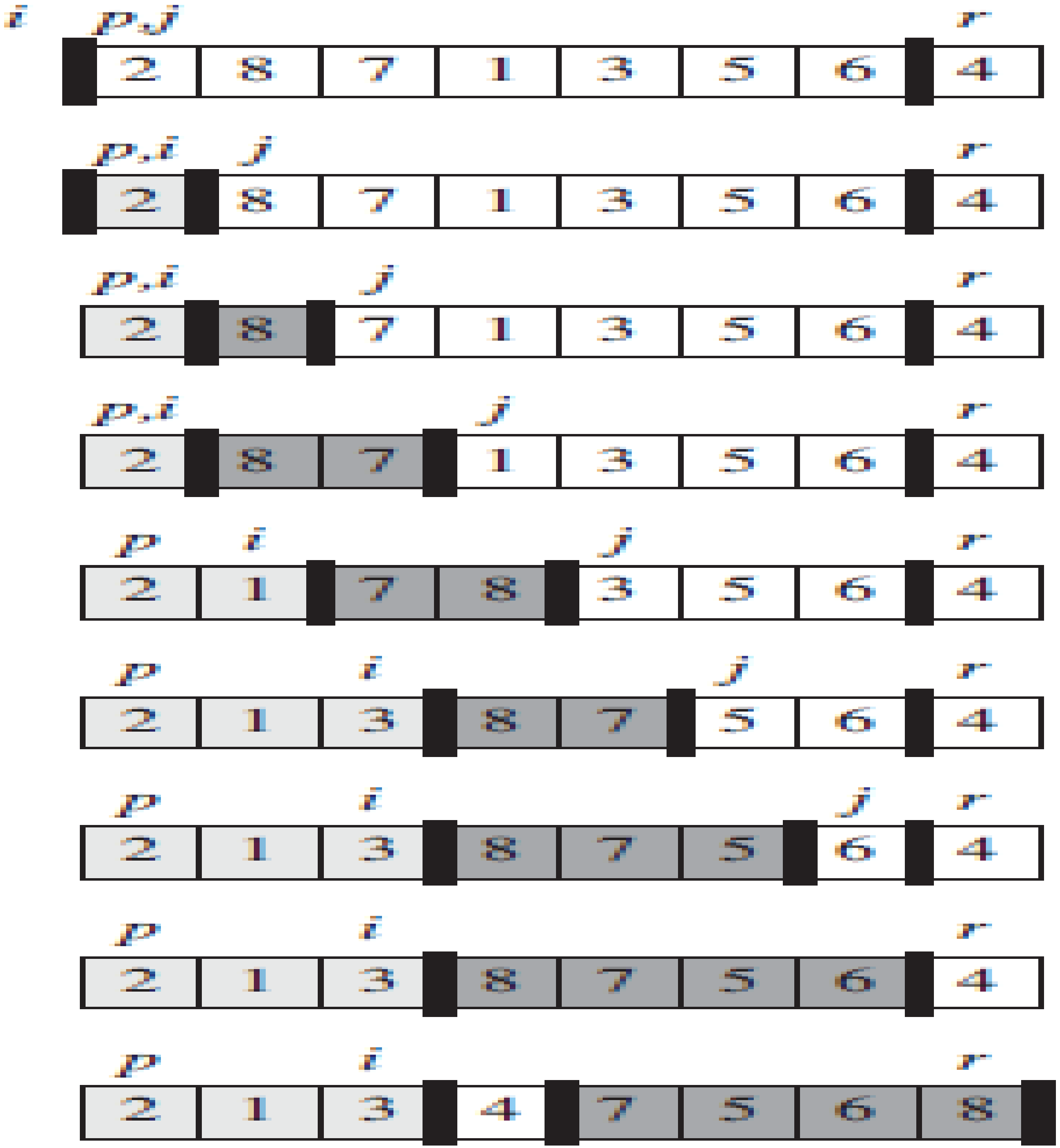


Running time: $\Theta(n)$

$n = r - p + 1$

Example


p
i
j
r
x
≤x
>x
unrestricted


i
p,j
r
2 8 7 1 3 5 6 4
p,i
j
r
2 8 7 1 3 5 6 4
p,i
j
r
2 8 7 1 3 5 6 4
p,i
j
r
2 8 7 1 3 5 6 4
p
i
j
r
2 1 7 8 3 5 6 4
p
i
j
r
2 1 3 8 7 5 6 4
p
i
j
r
2 1 3 8 7 5 6 4
p
i
r
2 1 3 8 7 5 6 4
p
i
r
2 1 3 4 7 5 6 8

QUICKSORT

Alg.: QUICKSORT(A, p, r)

if p < r // Initially: p=1, r= A.length

then $q \leftarrow$ PARTITION(A, p, r)

QUICKSORT (A, p, q-1)

QUICKSORT (A, q+1, r)

Recurrence Equation of Above Algorithm:

$T(n) = T(q) + T(n - q) + f(n)$ Where f(n) depend on PARTITION() function

Worst Case Partitioning

- Worst-case partitioning
  - The pivot divides the list of size n into two sub-lists of sizes 0 and n-1.
  - One region has one element and the other has n – 1 elements
  - Maximally unbalanced
  - The number of key comparisons

$$= n-1 + n-2 + \dots + 1$$

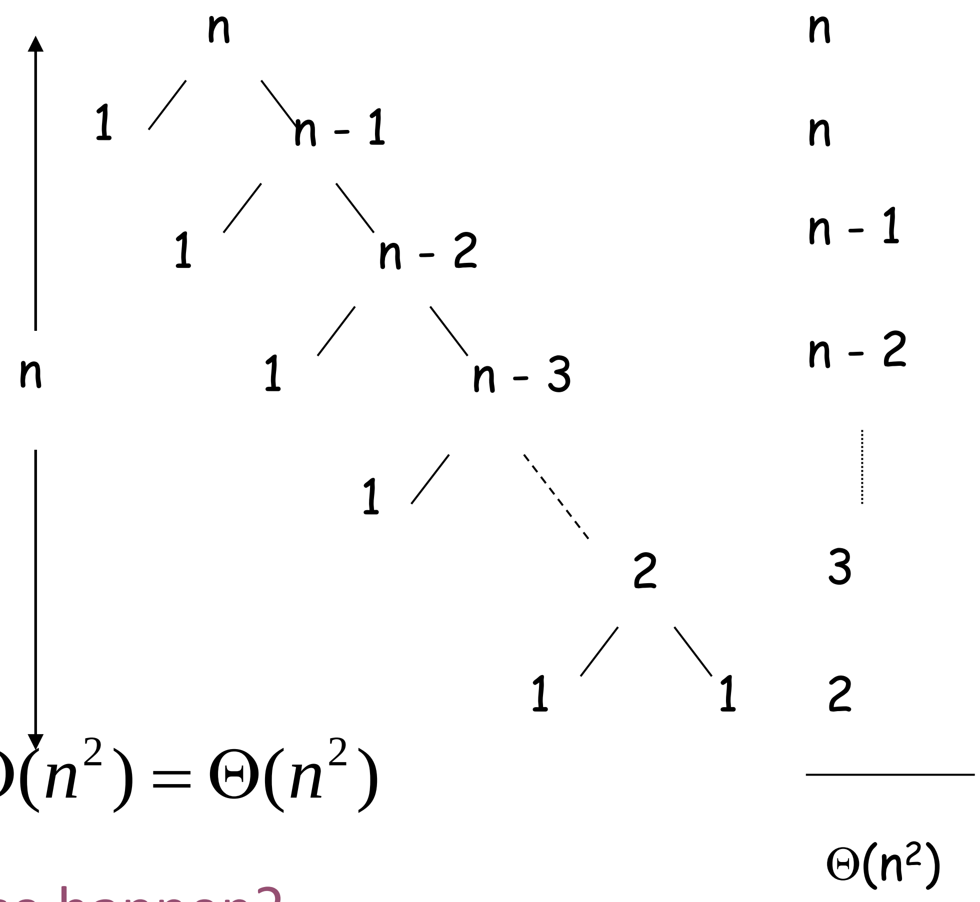
$$= \mathbf{n^2/2 - n/2} \quad \Rightarrow \quad \mathbf{O(n^2)}$$

- Recurrence: q=1

$T(n) = T(1) + T(n-1) + n,$

$T(1) = \Theta(1)$

$$T(n) = T(n-1) + n = n + \left(\sum_{k=1}^{n} k\right) - 1 = \Theta(n) + \Theta(n^2) = \Theta(n^2)$$



When does the worst case happen?

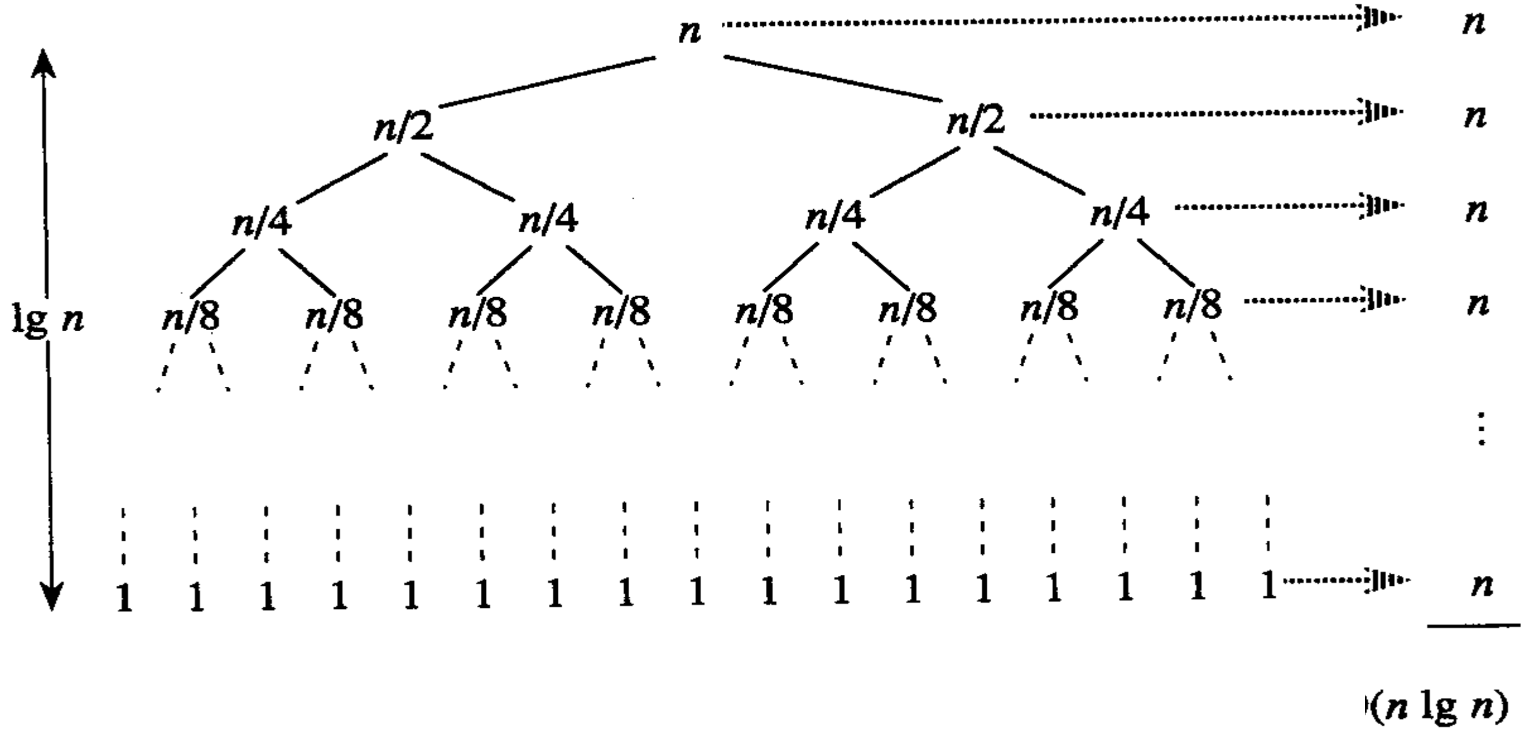
Best Case Partitioning

In each partition, the problem is always divided into two sub-problems with almost equal size. i.e.,Partitioning produces two regions of size n/2.

- Recurrence: q=n/2

  $T(n) = 2T(n/2) + \Theta(n)$

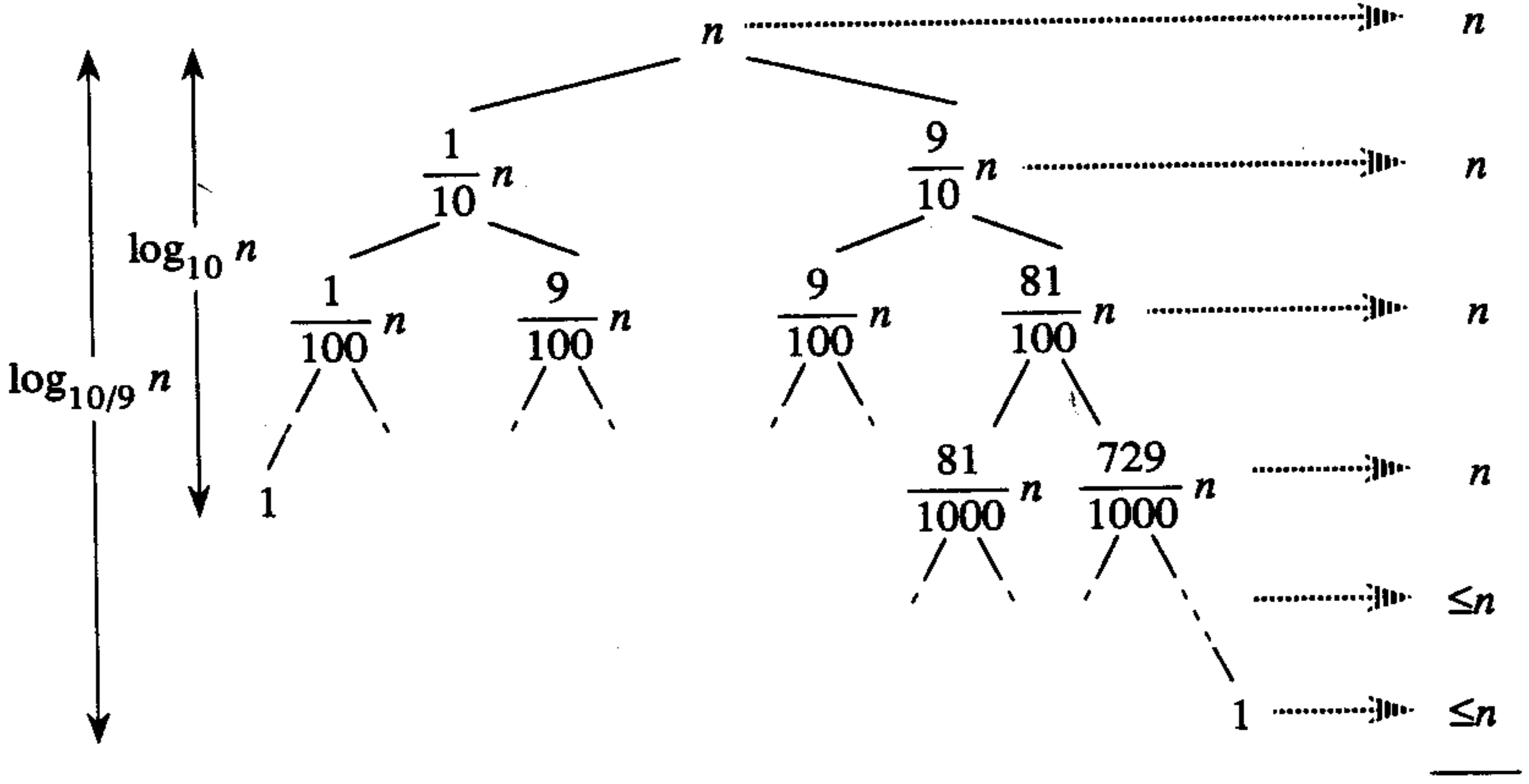
  $T(n) = \Theta(n\log n)$ // Using Master theorem



Case Between Worst and Best

- 9-to-1 proportional split

$$Q(n) = Q(9n/10) + Q(n/10) + n$$



Using Recurrence Method: $\Theta(n \lg n)$

Quicksort – Analysis

Based on whether the partitioning is balanced or not .

- *Best case*: split in the middle — $\Theta(n \log n)$
  - $T(n) = 2T(n/2) + \Theta(n)$ //2 subproblems of size n/2 each
- *Worst case*: sorted array! — $\Theta(n^2)$
  - $T(n) = T(n-1) + n+1$ //2 subproblems of size 0 and n-1 respectively
- *Average case*: random input array — $\Theta(n \log n)$

Randomized version of Quicksort

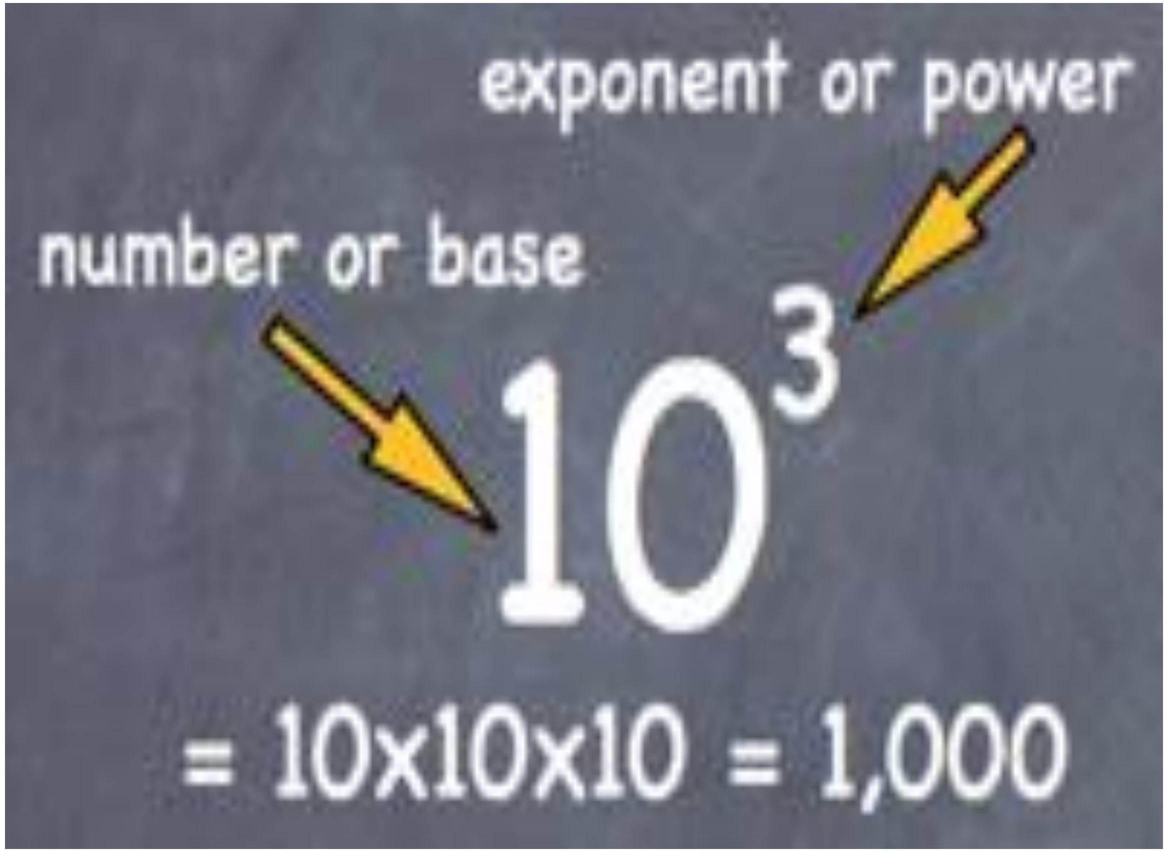
- Instead of always using A[r] as the pivot, we will select a randomly chosen element from the subarray A[p…r].
- We do so by first exchanging element A[r] with an element chosen at random from the subarray A[p…r].
- By randomly sampling the range p … r, we ensure that the pivot element x is equally likely to be any of the r - p + 1 elements in the subarray.
- Because we randomly choose the pivot element, we expect the split of the input array to be reasonably well balanced on average.
- The expected running time of RANDOMIZED-QUICKSORT is $\Theta(n \log n)$

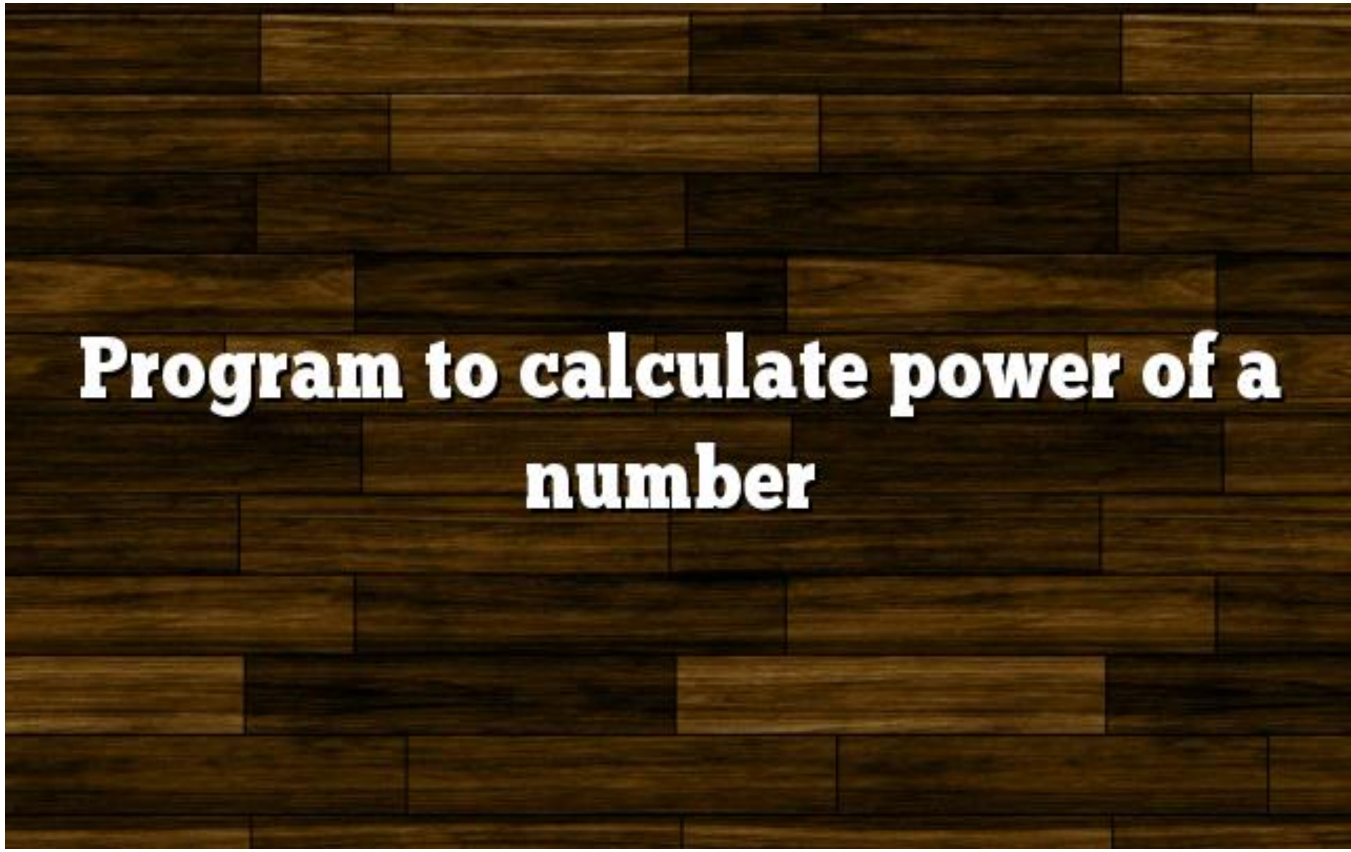
Randomized Quicksort Algorithm

```
Algo: RANDOMIZED-PARTITION (A, p, r)
i = RANDOM (p, r)
exchange A[r] with A[i]
return PARTITION (A, p, r)
```

```
Alg.: RANDOMIZED-QUICKSORT(A, p, r)
      if p < r
         then q ← RANDOMIZED-PARTITION(A, p, r)
                  RANDOMIZED-QUICKSORT (A, p, q-1)
                  RANDOMIZED-QUICKSORT (A, q+1, r)
```

Calculate power(base,exponent) in optimized way [Favorite question for many interviews]

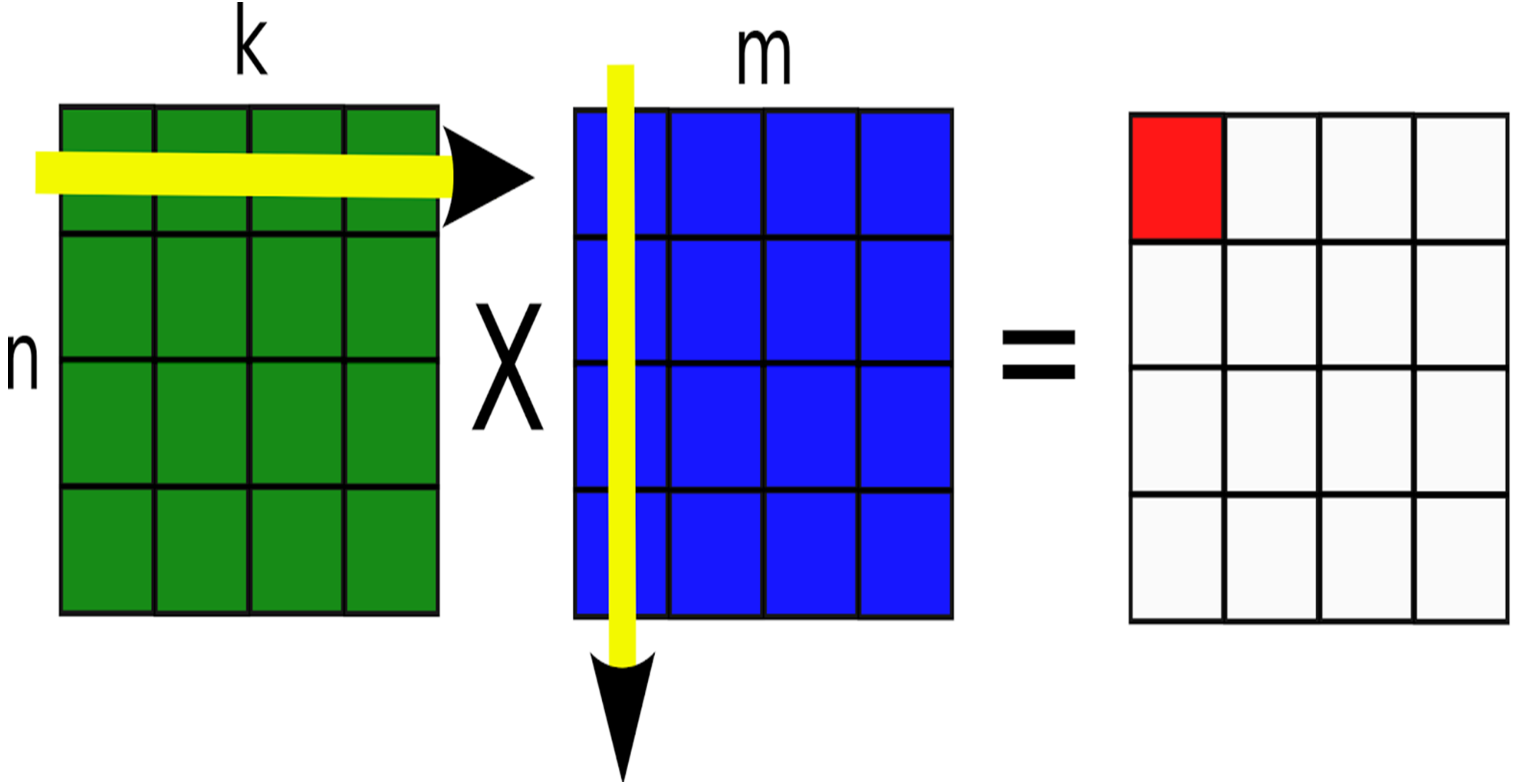




```
public long pow_REC(long base, long exp)
    {

        if(exp ==1)
        {
            return base;
        }
        else
        {
            return base*pow_REC(base, exp-1);
        }

    }
```

```
public long pow_DAC(long base, long exp)
    {
        if(exp ==0)
        {
            return 1;
        }

        if(exp ==1)
        {
            return base;
        }

        if(exp % 2 == 0)
        {
            long half = pow_DAC(base, exp/2);
            return half * half;
        }
        else
        {
            long half = pow_DAC(base, (exp -1)/2);
            return base * half * half;
        }
     }
```

Matrix multiplication



Matrix multiplication

- Multiplication of 2 matrices is a fundamental numerical operation.
  - The standard method of matrix multiplication of two n x n matrices takes $T(n) = O(n^3)$.
- Let A, B and C be $n \times n$ matrices

$$C = AB$$

$$C(i, j) = \sum_{1 \le k \le n} A(i, k)B(k, j)$$

Naïve Algorithm for multiplication of two matrices A and B:

```
// Initialize Matix C.
for i = 1 to n
  for j = 1 to n
    for k = 1 to n
          C [i, j] += A[i, k] * B[k, j];
```

Divide and Conquer Approach

- We can use a Divide and Conquer solution by separating a matrix into 4 quadrants:

$$\left[\begin{array}{cc|cc} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ \hline a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{array}\right] \text{ X } \left[\begin{array}{cc|cc} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ \hline b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{array}\right] = \left[\begin{array}{cc|cc} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ \hline c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{array}\right]$$

- Divide matrices A and B in 4 sub-matrices of size N/2 x N/2.

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \quad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} \quad C = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

if , $C = AB$
then we have :

$$\begin{aligned} c_{11} &= a_{11}b_{11} + a_{12}b_{21} \\ c_{12} &= a_{11}b_{12} + a_{12}b_{22} \\ c_{21} &= a_{21}b_{11} + a_{22}b_{21} \\ c_{22} &= a_{21}b_{12} + a_{22}b_{22} \end{aligned}$$

- ✓ In the above method, we do 8 multiplications for matrices of size N/2 x N/2 and 4 additions. Addition of two matrices takes $O(N^2)$ time.
- ✓ So the time complexity can be written as: **$T(n) = 8T(n/2) + O(n^2)$**
- ✓ From Master's Theorem, time complexity of above method is **$O(n^3)$**.
- ✓ **Simple Divide and Conquer also leads to $O(n^3)$. can there be a better way?**

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
n = A.rows
let C be a new n × n matrix
if n == 1
    c11 = a11 · b11
else partition A, B, and C as in equations (4.9)
    C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
    C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
    C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
    C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
return C
```

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1\,, \\ 8T(n/2) + \Theta(n^2) & \text{if } n > 1\,. \end{cases}$$

Strassen’s matrix multiplication

❖In divide and conquer method, the main component for high time complexity is 8 recursive calls.

❖The idea of Strassen’s method is to reduce the number of recursive calls to 7.

❖Strassen’s method is similar to simple D & C method in the sense that this method also divide matrices to sub-matrices of size N/2 x N/2.

❖Strassen’s Matrix multiplication can be performed only on **square matrices**where **n** is a **power of 2**. Order of both of the matrices are **n × n**.

- $P = (A_{11} + A_{22})(B_{11} + B_{22})$
  $Q = (A_{21} + A_{22})B_{11}$
  $R = A_{11}(B_{12} - B_{22})$
  $S = A_{22}(B_{21} - B_{11})$
  $T = (A_{11} + A_{12})B_{22}$
  $U = (A_{21} - A_{11})(B_{11} + B_{12})$
  $V = (A_{12} - A_{22})(B_{21} + B_{22})$.
- $C_{11} = P + S - T + V$
  $C_{12} = R + T$
  $C_{21} = Q + S$
  $C_{22} = P + R - Q + U$

Time complexity of Strassen's matrix multiplication

- **Strassen's matrix multiplication** requires 7 multiplications and 18 additions or subtractions.
- Addition and Subtraction of two matrices takes $O(N^2)$ time. So time complexity can be written as
- Time complexity: $7T(n/2)+O(n^2)$
- From Master's Theorem, time complexity of above method is $O(N^{Log 7})$ which is approximately $O(N^{2.8074})$

	Mult	Add	Recurrence Relation	Runtime
Regular	8	4	$T(n) = 8T(n/2) + O(n^2)$	$O(n^3)$
Strassen	7	18	$T(n) = 7T(n/2) + O(n^2)$	$O(n^{\log_2 7}) = O(n^{2.81})$

Strassen's Algorithm Correctness

- Let's verify one of these:

Given:

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \qquad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} \qquad C = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

if $C = AB$

we know:

$$c_{21} = a_{21}b_{11} + a_{22}b_{21}$$

Strassen's Algorithm states:

- $\mathbf{c_{21} = q + s = (a_{21}\, b_{11} + a_{22}\, b_{11}) + (a_{22}\, b_{21} - a_{22}\, b_{11})}$

where $\mathbf{q = (a_{21} + a_{22}) * b_{11}}$ **and** $\mathbf{s = a_{22} * (b_{21} - b_{11})}$

Multiplication of Large Integers

Consider the problem of multiplying two (large) n-digit integers represented by arrays of their digits such as:

A = 123456789   B = 87654321

The grade-school algorithm:

$$
\begin{array}{r}
a_1\ a_2 \dots\ a_n \\
\underline{b_1\ b_2 \dots\ b_n} \\
(d_{10})\ d_{11} d_{12} \dots d_{1n} \\
(d_{20})\ d_{21} d_{22} \dots d_{2n} \qquad\quad \\
\dots \dots \dots \dots \dots \dots \dots \qquad\quad \\
\underline{(d_{n0})\ d_{n1} d_{n2} \dots d_{nn}} \qquad\qquad\quad
\end{array}
$$

Efficiency: $\Theta(n^2)$ single-digit multiplications and $\Theta(n)$ single-digit additions

First Divide-and-Conquer Algorithm

A small example: $A * B$ where $A = 2135$ and $B = 4014$

$A = (21 \cdot 10^2 + 35)$, $B = (40 \cdot 10^2 + 14)$

So, $A * B = (21 \cdot 10^2 + 35) * (40 \cdot 10^2 + 14)$

$= 21 * 40 \cdot 10^4 + (21 * 14 + 35 * 40) \cdot 10^2 + 35 * 14$

In general, if $A = A_1A_2$ and $B = B_1B_2$ (where A and B are *n*-digit, A_1, A_2, B_1, B_2 are *n/*2-digit numbers),

$A * B = (A_1 \cdot 10^{n/2} + A_2) * (B_1 \cdot 10^{n/2} + B_2)$

$= A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2$

Recurrence for the number of one-digit multiplications $M(n)$:

$$M(n) = 4M(n/2), \quad M(1) = 1$$

Solution: $M(n) = n^2$

Second Divide-and-Conquer Algorithm

$A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2$

The idea is to decrease the number of multiplications from 4 to 3:

$(A_1 + A_2) * (B_1 + B_2) = A_1 * B_1 + (A_1 * B_2 + A_2 * B_1) + A_2 * B_2,$

$\Rightarrow$ $(A_1 * B_2 + A_2 * B_1) = (A_1 + A_2) * (B_1 + B_2) - A_1 * B_1 - A_2 * B_2,$

which requires only 3 multiplications at the expense of (4-1) extra add/sub.

$U = (A_1 + A_2) * (B_1 + B_2)$

$V = A_1 * B_1$

$W = A_2 * B_2$

$Z = V \cdot 10^n + (U - V - W) \cdot 10^{n/2} + W$

Recurrence for the number of multiplications M(n):

$$M(n) = 3M(n/2), \quad M(1) = 1$$

Solution: M(n) $= 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}$

Example of Large-Integer Multiplication

2135 * 4014

$= (21*10^2 + 35) * (40*10^2 + 14)$

$= (21*40)*10^4 + c1*10^2 + 35*14$

where $c1 = (21+35)*(40+14) - 21*40 - 35*14$

21*40 $= (2*10 + 1) * (4*10 + 0)$

$= (2*4)*10^2 + c2*10 + 1*0$

where $c2 = (2+1)*(4+0) - 2*4 - 1*0$, etc.

This process requires 9 digit multiplications as opposed to 16.

Multiplication of large Integers

$a = a_1 a_0$ and $b = b_1 b_0$

$c = a * b$

$= (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0)$

$=(a_1 * b_1)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0)$

For instance: $a = 123456$, $b = 117933$:

Then $c = a * b = (123*10^3+456)*(117*10^3+933)$

$=(123 * 117)10^6 + (123 * 933 + 456 * 117)10^3 + (456 * 933)$

Multiplication of large integers

- $a = a_1a_0$ and $b = b_1b_0$
- $c = a * b$
  $=(a_1 * b_1)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0)$
  $=c_2 10^n + c_1 10^{n/2} + c_0,$

  where
  $c_2 = a_1 * b_1$ is the product of their first halves
  $c_0 = a_0 * b_0$ is the product of their second halves
  $c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's halves and the sum of the b's halves minus the sum of c_2 and c_0.

Multiplication of large integers

- $c = c_2 10^n + c_1 10^{n/2} + c_0$,

  where

  $c_2 = a_1 * b_1$

  $c_0 = a_0 * b_0$

  $c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$

  Multiplication of n-digit numbers requires three multiplications of n/2-digit numbers


Backspace
Screen
Home
Page Up
Thank You
Page Down
Shift
Insert