CS 2100: Data Structures & Algorithms 1

Advanced Sorts (Part I)

Mergesort; divide and conquer

Dr. Nada Basit // basit@virginia.edu

Spring 2022

Friendly Reminders

- The University updated the mask policy. As per my Request on Mar 28, 2022 (see Collab), I would greatly appreciate if you would do me a kind favor by **continuing to wear your masks** in CS 2100 (Ridley G008). I know it is a lot to ask, and it is **voluntary**, but I appreciate your understanding.
- If you forget your mask (or mask is lost/broken), I have a few available
  - Just come up to me at the start of class and ask!
- No eating or drinking in the classroom, please
- Our lectures will be **recorded** (see Collab) – please allow 24-48 hrs to post
- If you feel **unwell**, or think you are, please stay home
  - *We will work with you!*
  - At home: eye mask instead! Get some rest ☺

{Reminder} How to Sort?

- Some "straightforward" sorting algorithms
  - Insertion Sort, Selection Sort, Bubble Sort
  - **Each is $O(n^2)$**

- More efficient sorting algorithms
  - Quicksort, Mergesort, Heapsort
  - **Each is O(n log n)**

Best Sorts are O(n log n)

{Reminder} Sorting using Collections.sort()

- Collections and Arrays classes provide `.sort()` methods!
  - Utilizes `compareTo()` or `Comparator` to ***determine order*** when comparing elements
  - "under the hood", it's a *variant* of something called ***Mergesort***
  - Θ(n log n) worst-case -- as good as we can do
  - We'll discuss how **Mergesort** works soon!

Mergesort

A divide-and-conquer style algorithm

Mergesort Introduction

- General sorting algorithm
- Is ***recursive***
- An example of a **divide-and-conquer** algorithm
- Is $\mathbf{o(n^2)}$ – *strictly faster* than $\mathbf{n^2}$
- is faster than the **adjacent sorts** in most situations
  - Bubble sort and Insertion sort were: $\Theta(\mathbf{n^2})$

Divide-and-Conquer

Certain algorithms follow this paradigm; usually they are recursive too

Scenario

- Imagine you worked for the Post Office.

- One day the automated sorting machine broke down and you have lots of pieces of mail to sort! ☹

- If you had **100 pieces of mail to sort**, executing a sort algorithm (e.g. $O(n^2)$) on **this one pile of 100 pieces** will take **10,000t** (t=time)

Divide and Conquer

- However, sorting **2 piles of 50** would take 2 x 2,500t
- Sorting **4 piles of 25** will take 4 x 645t (2,500t)
- **Diving the problem reduced the overhead!**

Using recursion to break a problem down into smaller pieces to improve algorithm performance. (Run each of these smaller pieces *in parallel!*)

- **Binary Search** is an example of this.

Divide and Conquer:
putting recursion to work for you!

- An **algorithm design strategy**, one of many you will learn
- Strategy: It is often easier to solve several **small instances** of a problem than one large one.
  - **divide** the problem into **k** smaller instances of the same problem
  - **conquer** (***solve***) each the **k** problems **recursively**
  - **combine** the solutions to obtain the solution for ***original*** input
- Note: Must have a base case to solve *really small* problems ***directly***

General Strategy for Divide and Conquer

```
Solve(A)        // solve for input A
  n = size(A)  // size of our problem is n
  // base case
  if (n <= smallsize) // problem <= some threshold
    solution = directlySolve(A);      // solve directly
  else // recursive case
    divide A into A_1, A_2, …, A_k.      // divide
    for each i in {1, …, k}
      S_i = solve(A_i);   // conquer each sub-problem
    solution = combine(S_1, …, S_k);     // combine parts
  return solution;    // return solution to original problem
```

General Strategy for Divide and Conquer

- Runtime is equal to time to divide + recurse + time to merge

```
solveProblem(input)
  if input is small, then brute-force
  else if input is big
    divide problem into n smaller problems
    recursively invoke solveProblem on smaller problems
    merge solutions to small problems into bigger solution
    return bigger solution
```

Why Divide and Conquer?

- Sometimes it's the simplest approach
- Divide and Conquer is often more efficient than "obvious" approaches
  - E.g. **Binary Search** instead of Sequential Search
  - E.g. **Merge Sort or Quicksort** instead of Selection Sort
- Not necessarily efficient: *Maybe the same or worse than another approach*
- No standard implementation: *May or may not be implemented recursively*
- Divide and Conquer algorithms illustrate a **top-down strategy**
  - Given a large problem, identify and break into smaller subproblems; solve then combine the results

[Aside]

Binary Search

Recursive divide-and-conquer strategy

Binary Search: Non-Recursive (*aka Iterative*)

```
int binSearch ( int[] array, int target ) {
   int first = 0;   int last = array.length-1;
   while ( first <= last ) {
      mid = (first + last) / 2;        // calculate middle ('mid')
      if ( target == array[mid] ) return mid;   // found it!
      else if ( target < array[mid] )      // must be in 1st half
            last = mid - 1;
      else                                  // must be in 2nd half
            first = mid + 1
  }
  return -1; // only got here if not found above
}
```

Binary Search: Recursive [pseudocode]

```
public static int binarySearch(int[] list, int value) {
        return binSearch(list, target, 0, list.length -1); // initially entire list is valid
}
public static int binSearch(int[] list, int first, int last, int target) {
    //Base Case: if no where left to look (if low > high) return (-1)
    //Calculate mid (an int)
    //Print mid – the item that is being compared
    //if mid is equal to target, return mid
    //else if mid is less than the target,  first = mid + 1 (target is in the top half)
    //else (mid is greater than the target), last = mid – 1 (target is in the bottom half)
    //return [a recursive call to binSearch, passing values list, first, last, target]
}
```

- No loop! Recursive calls takes its place - But don’t think about that if it confuses you!
- Base cases checked first? (Why? Zero items? One item?)

Binary Search: Recursive

```
int binSearch(int[] array, int first, int last, int target) {
    if (first <= last) {
        int mid = (first + last) / 2;
        if (target == array[mid])
            return mid;
        if (target < array[mid])
            return binSearch(array, first, mid - 1, target);
        else if (target > array[mid]);
            return binSearch(array, mid + 1, last, target);
    }
    return -1;
}
```

Merge Sort

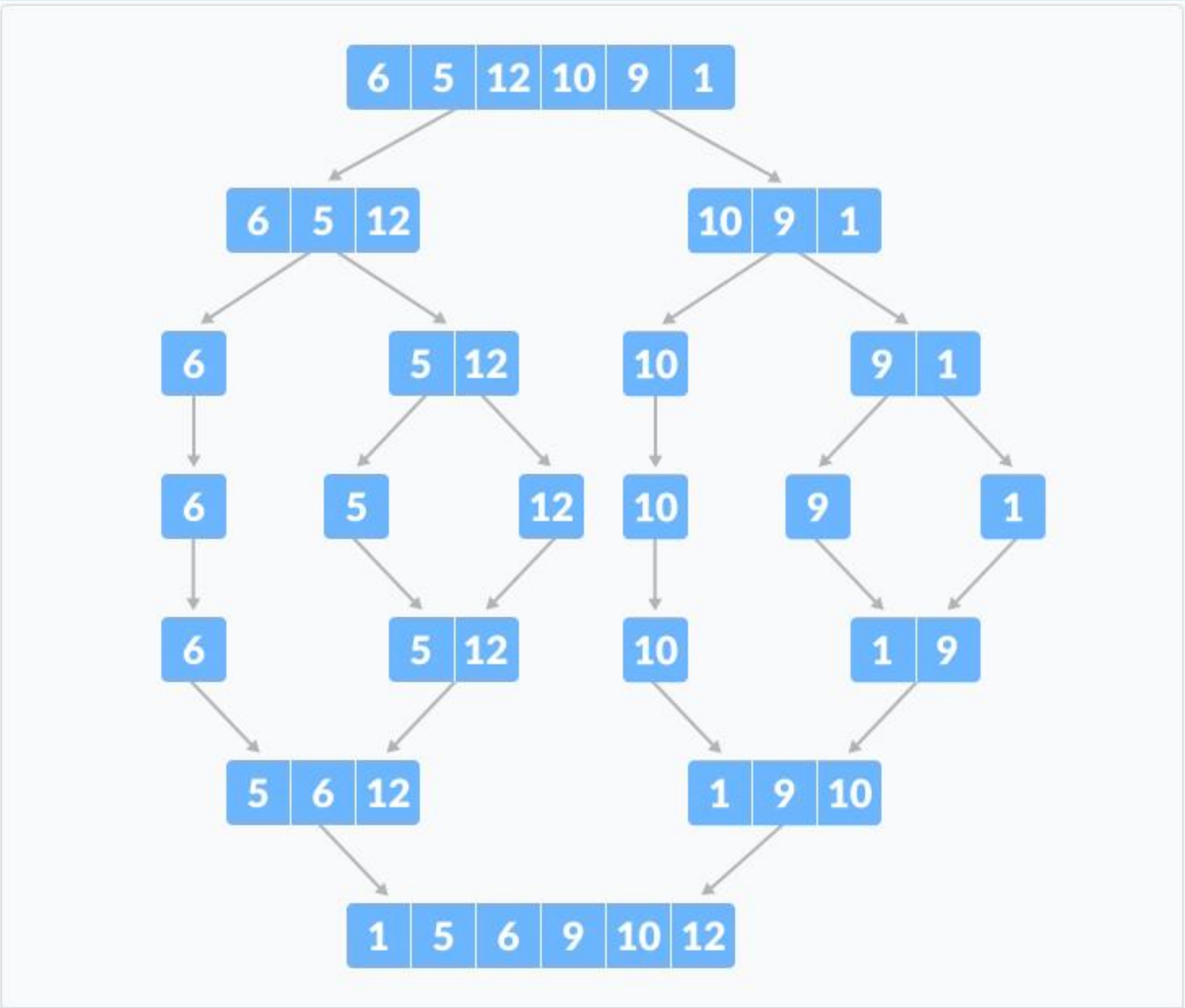
Divide-and-conquer stragegy

Algorithm: Mergesort

- Specification:
  - Input: Array E and indexes **first** and **last**
  - Output: Sorted rearrangement of the same elements in E between **first** & **last**

- Mergesort is a classic example of Divide and Conquer:
  - **Divide**: split the array into two halves (left and right / first and last)
  - **Conquer**: call mergesort() to recursively sort the two halves
  - **Combine**: combine the 2 sorted halves into one final sorted array
    - This is the "**merge**" step, and where it gest its name!

  [Base case: 1 element (is sorted) or: 2 elements (compare and swap)]

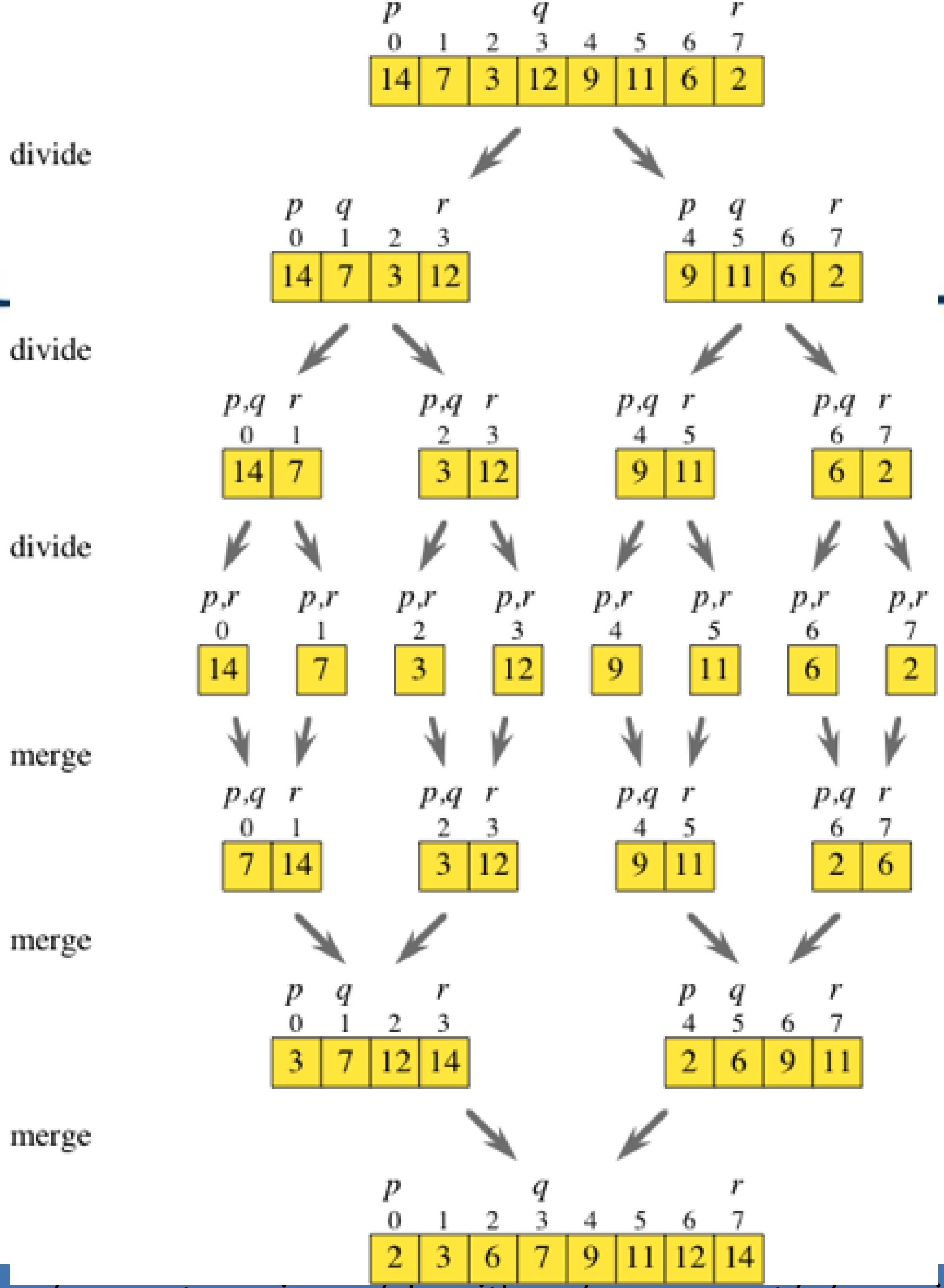


Images © programiz.com

Animation: Mergesort

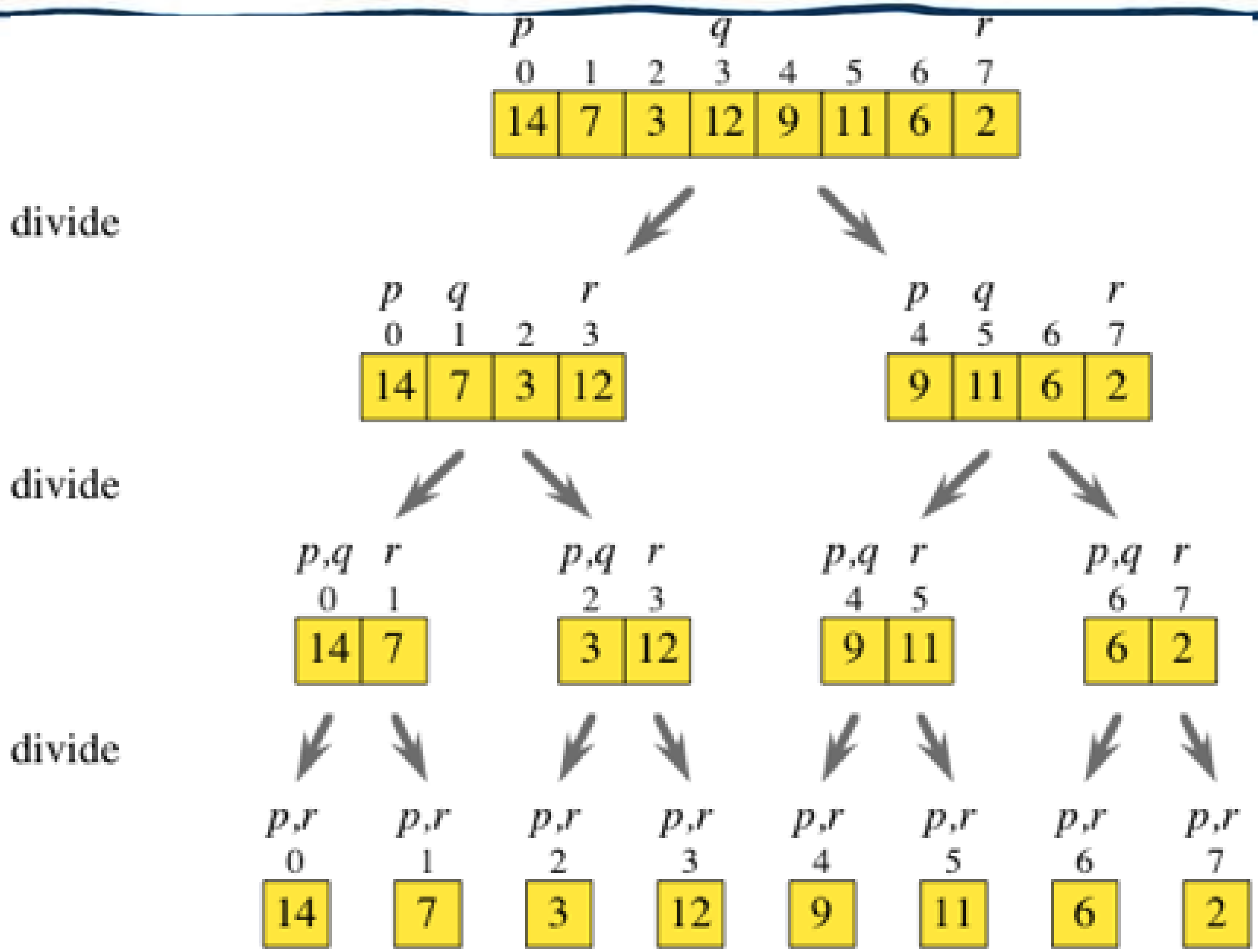
6 5 3 1 8 7 2 4

Mergesort (divide and conquer)



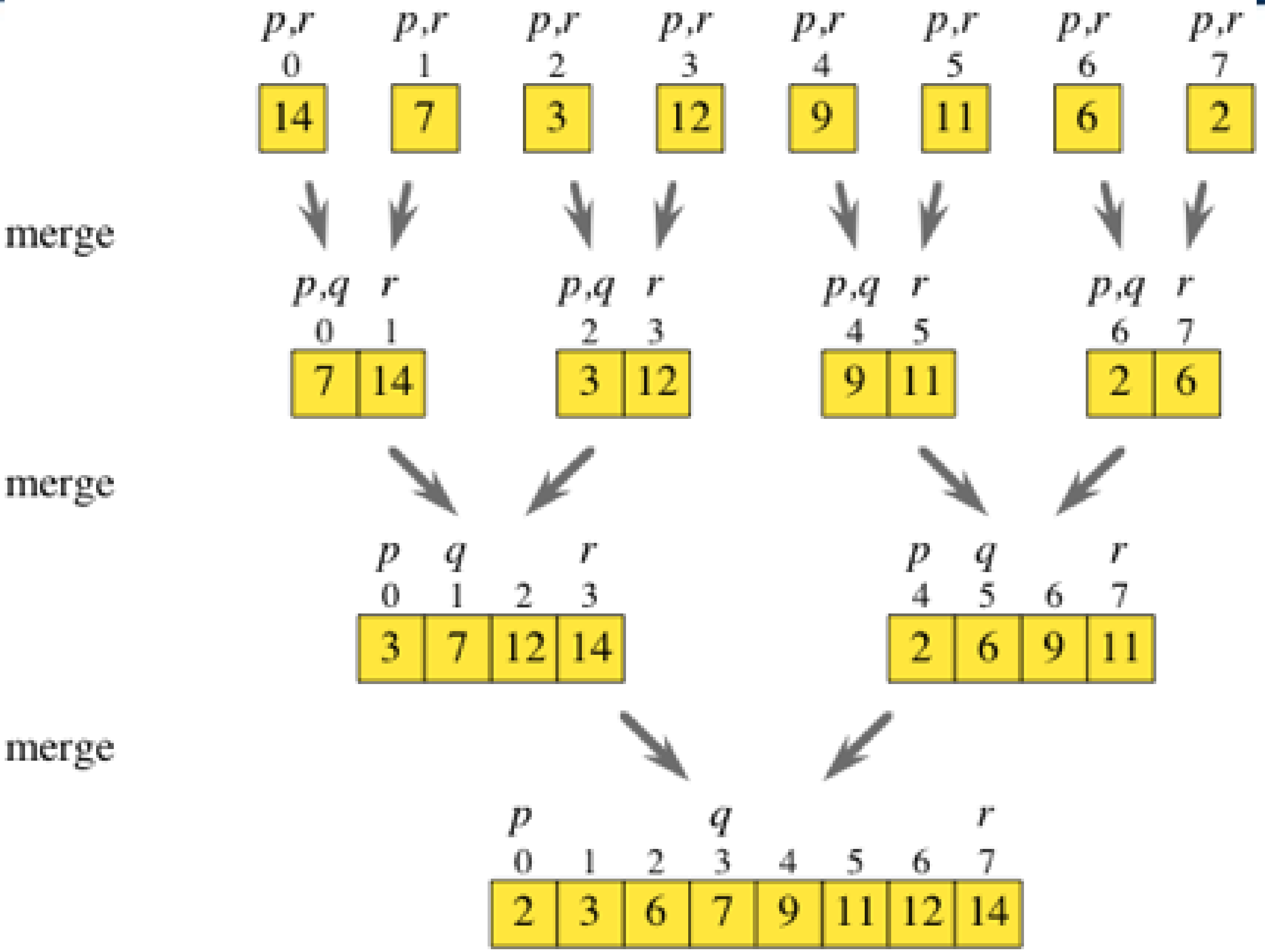
https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/overview-of-merge-sort

Mergesort: Divide stage



https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/overview-of-merge-sort

Mergesort: Conquer stage (merge)



Mergesort: Do it by hand ~ divide and merge stages

6 5 3 1 8 7 2 4

Exercise: Trace Mergesort Execution

- Can you trace MergeSort() on this list? ***(even** # of elements)*

A = {8, 3, 2, 9, 7, 1, 5, 4}; ← original list; to be sorted (8 *elements*)

8, 3, 2, 9 7, 1, 5, 4	← ***divide*** into 2 lists of 4
8, 3 2, 9 7, 1 5, 4	← ***divide*** 2 lists of 4 into 4 lists of 2
8, 3, 2, 9, 7, 1, 5, 4	← ***divide*** into **SINGLE** items
3,8 2,9 1,7 4,5	← ***merge*** single items into pairs
2,3,8,9 1,4,5,7	← ***merge*** 4 pairs into 2 lists of 4
{1,2,3,4,5,7,8,9}	← ***merge*** 2 lists of 4 into 1 list (Result)

Exercise: Trace Mergesort Execution

- Can you trace MergeSort() on this list? ***(odd # of elements)***

  A = {8, 3, 2, 9, 7, 1, 5, 4, 6}; ← original list; to be sorted (9 *elements*)

  8, 3, 2, 9 7, 1, 5, 4, 6 ← ***divide*** two lists are not even (ok!)

  8, 3 2, 9 7, 1 5, 4 6 ← ***divide*** into pairs + 1

  8, 3, 2, 9, 7, 1, 5, 4, 6 ← ***divide*** into **SINGLE** items

  ---

  3,8 2,9 1,7 4,5,6 ← ***merge*** single items into pairs + 3

  2,3,8,9 1,4,5,6,7 ← ***merge***

  {1,2,3,4,5,6,7,8,9} ← ***merge*** into 1 list (Result)

Efficiency of Mergesort

- ***Mergesort*** is **O(n lg n)**
  same order-class as the most efficient sorts (quicksort and heapsort)

- It **is** more efficient than Selection Sort, Bubble Sort, and Insertion Sort
- The **Divide and Conquer approach** matters, and in this case, is a "win"!

- Most of the work is done in the "**merge**" portion of the algorithm
- Most implementations use a "scratch array"
  - An extra array of size n which is then copied back into the original array

Merge part of Mergesort

Merging Sorted Sequences

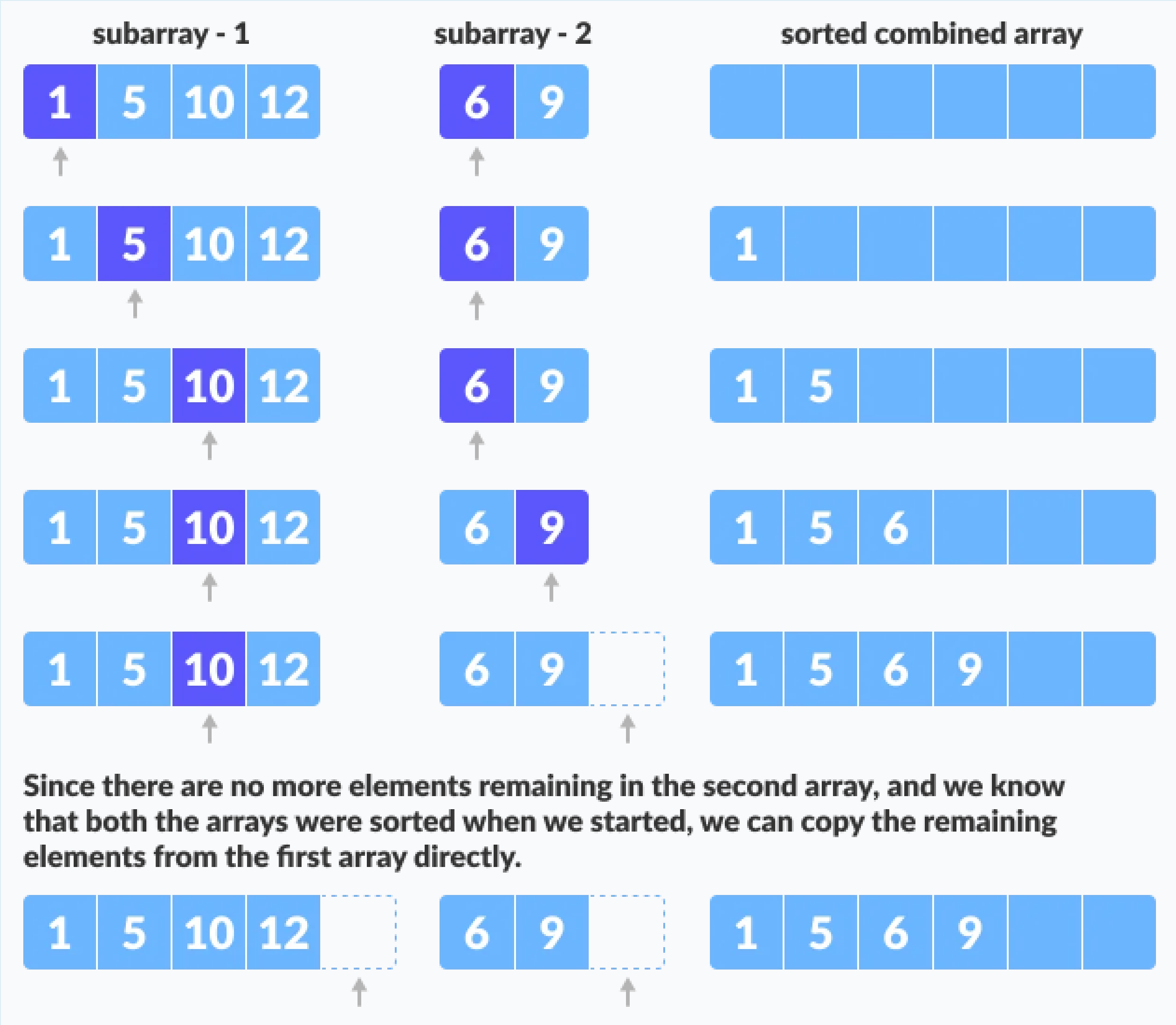
- Problem:
  - Given two sequences **A** and **B** sorted in non-decreasing order, merge them to create one sorted sequence **C**
  - Input size: C has n items, and A and B have n/2
- Strategy:
  - Determine the first item in C: it should be the smaller of the first item in A and the first in B.
  - Suppose it is the first item of A. Copy that to C.
  - Then continue merging B with "rest of A" (without the item copied to C). Repeat!

Algorithm: Merge (pseudocode)

- merge(A, B, C)
  - if (A is empty)
    - append what is left in B to C
  - else if (B is empty)
    - append what is left in A to C
  - else if (first item in A <= first item in B)
    - append first item in A to C
    - merge (rest of A, B, C)
  - else // first item in B is smaller
    - append first item in B to C
    - merge (A, rest of B, C)
  - return

```
// sequence A and B; merge into C
// maintain current index of sub-arrays (A & B)
// and destination array (C)
int a_ptr = 0;  int b_ptr = 0;  int c_ptr = 0;
// until end of A or B is reached, pick the
// larger among elements pointed to in A and B
// and place in correct position in C array
while (a_ptr < A.len && b_ptr < B.len) {
   if(A[a_ptr] <= B[b_ptr]) { // ele in A smaller
      C[c_ptr] = A[a_ptr];
      a_ptr++; // increment A pointer
   } else {  // ele in B smaller
      C[c_ptr] = B[b_ptr];
      b_ptr++; // increment B pointer
   }
   c_ptr++; // adjust C pointer for next ele
}
// when run out of elements in either A or B
// pick up the remaining ele and put into C
while (a_ptr < A.len) // copy rest of A into C
   C[c_ptr] = A[a_ptr]; a_ptr++; c_ptr++;

while (b_ptr < B.len) // copy rest of B into C
   C[c_ptr] = B[b_ptr]; b_ptr++; c_ptr++;
```



Images © programiz.com

Examining Merge: *Small* Example

- Merge A and B: **A**= 3,8 **B**= 2,9 C= {} to hold sorted list
- 3,8 <u>2</u>,9 C={2}
- <u>3</u>,8 9 C={2, 3}
- <u>8</u> 9 C={2, 3, 8}
- <u>9</u> C={2, 3, 8, 9}
- Done!

Red: elements at **head** of list
<u>Red Underlined</u>: smallest
(this is the element that gets added to the sorted list)
Green: rest of items in the list
C: growing sorted list

Mergesort Analysis

- What is the runtime?
  - Divide the list (constant)
  - Two recursive sorts
  - Merge (linear)
- Total: **T(n) = 2T(n/2) + n = Θ(nlog(n))**
  - Uhhhhh...why?
- How to tell that this is true?
- **T(n) = 2T(n/2) + n = Θ(nlog(n))**
  - Solve for a closed form (will see in DSA2)
  - Draw out tree and count (we did this!)
  - Master theorem (nice...will see in DSA2)

Mergesort: Do it by hand ~ divide and merge stages

6 5 3 1 8 7 2 4

Each row has linear work (n) for merge to do

There are $\log(n)$ rows → $n*\log(n)$

Meregesort Method Signature

- Typically, Mergesort is done like this instead:

```
//Sort the list between indices i and j
public void mergeSort(T[] list, int i, int j);
```

- And **recursive** calls done like this
  - Doesn't make new arrays when dividing
  - Just ask mergesort to only work on one portion of interest
  - **merge()** still uses **scratch array**, copies *back* to list

```
int mid = (j+i)/2;
mergeSort(list, i, mid);
mergeSort(list, mid+1, j);
```

Algorithm: Meregesort

```
public static void mergeSort (Element[] E, int first, int last){
 if (first < last) {                 // base case == 1 element
   int mid = (first + last)/2;       // calculate middle
   mergeSort(E, first, mid);         // sort first half
   mergeSort(E, mid+1, last);        // sort second half
   merge(E, first, mid, last);       // merge two sorted halves
 }
}
```