Indian Institute of Science
Bangalore, India
भारतीय विज्ञान संस्थान
बंगलौर, भारत

DS221

Data Structures, Algorithms & Data Science Platforms

Instructor: Chirag Jain (slides from Prof. Simmhan)

Slides contributed by:
Yogesh Simmhan, Venkatesh Babu & Sathish Vadhiyar, CDS, IISc

CDS
Department of Computational and Data Sciences

L5: Algorithm Types

Algorithms

Algorithm classification

- Algorithms that use a *similar problem-solving approach* can be grouped together
  - A classification scheme for algorithms
- Classification is neither exhaustive nor disjoint
- The purpose is not to be able to classify an algorithm as one type or another, but to *highlight the various ways in which a problem can be attacked*

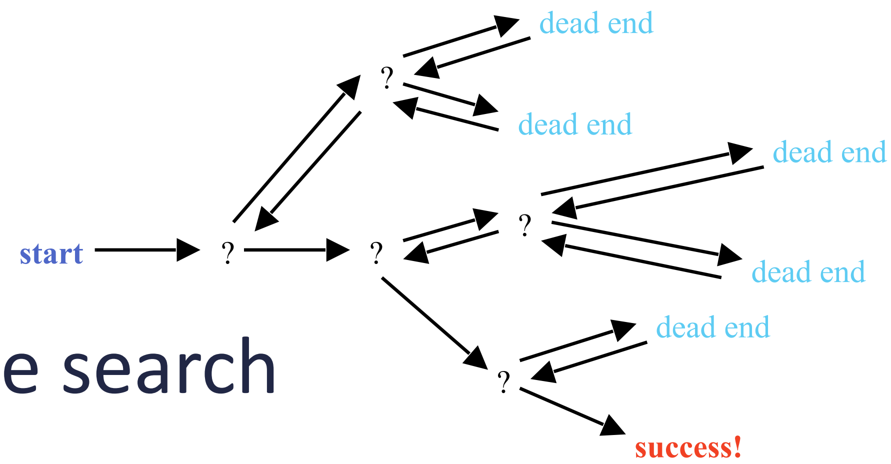
A short list of categories

- Algorithm types we will consider include:
  1. Simple recursive algorithms
  2. Backtracking algorithms
  3. Divide and conquer algorithms
  4. Dynamic programming algorithms
  5. Greedy algorithms
  6. Branch and bound algorithms
  7. Brute force algorithms
  8. Randomized algorithms

Simple Recursive Algorithms

- A simple recursive algorithm:
  1. Solves the base cases directly
  2. Recurs with a simpler subproblem
  3. Does some extra work to convert the solution to the simpler subproblem into a solution to the given problem
- These are "simple" because several of the other algorithm types are inherently recursive
- *Any seen so far?*
  - Tree traversal
  - Binary search over sorted array

Backtracking algorithms



- Uses a depth-first recursive search over solution space
  - Test to see if a solution has been found, and if so, returns it; otherwise
  - For each choice that can be made at this point,
    - Make that choice
    - Recurse
    - If the recursion returns a solution, return it
  - If no choices remain, return failure

- *Any seen so far?*
  - DFS traversal

Sample backtracking algo.

Sudoku: Fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 sub-grids that compose the grid contain all of the digits from 1 to 9.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Divide and Conquer

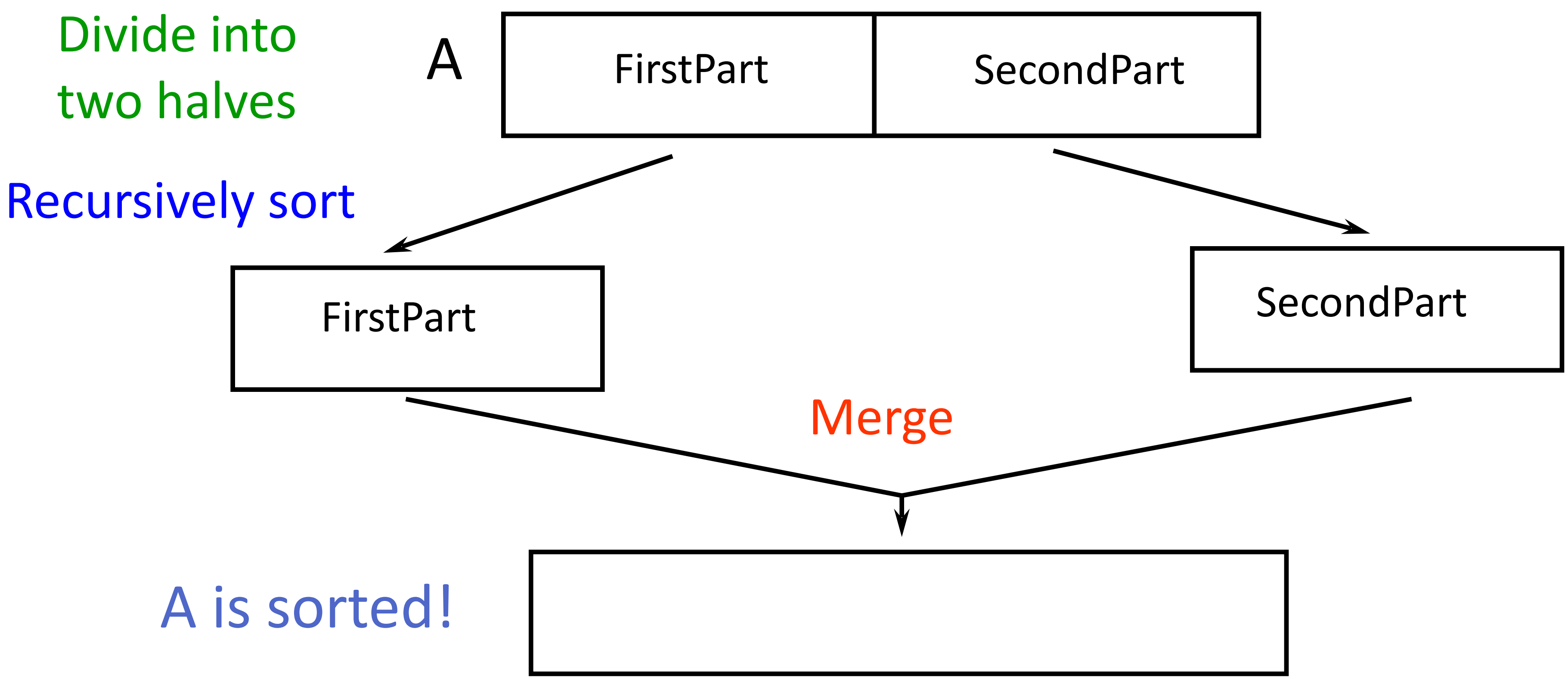
- A divide and conquer algorithm consists of two parts:
  - *Divide* the problem into smaller subproblems of the same type, and solve these subproblems recursively
  - *Combine* the solutions to the subproblems into a solution to the original problem
- *Traditionally, an algorithm is only called "divide and conquer" if it contains at least two recursive calls*

Binary search tree lookup?

- Compare the key to the value in the root
  - If the two values are equal, report success
  - If the key is less, search the left subtree
  - If the key is greater, search the right subtree
- This is *not* a divide and conquer algorithm because, although there are two recursive calls, only one is used at each level of the recursion

- *Sorting algorithms are good examples. E.g. Merge Sort, Quick Sort*

Merge Sort: Idea



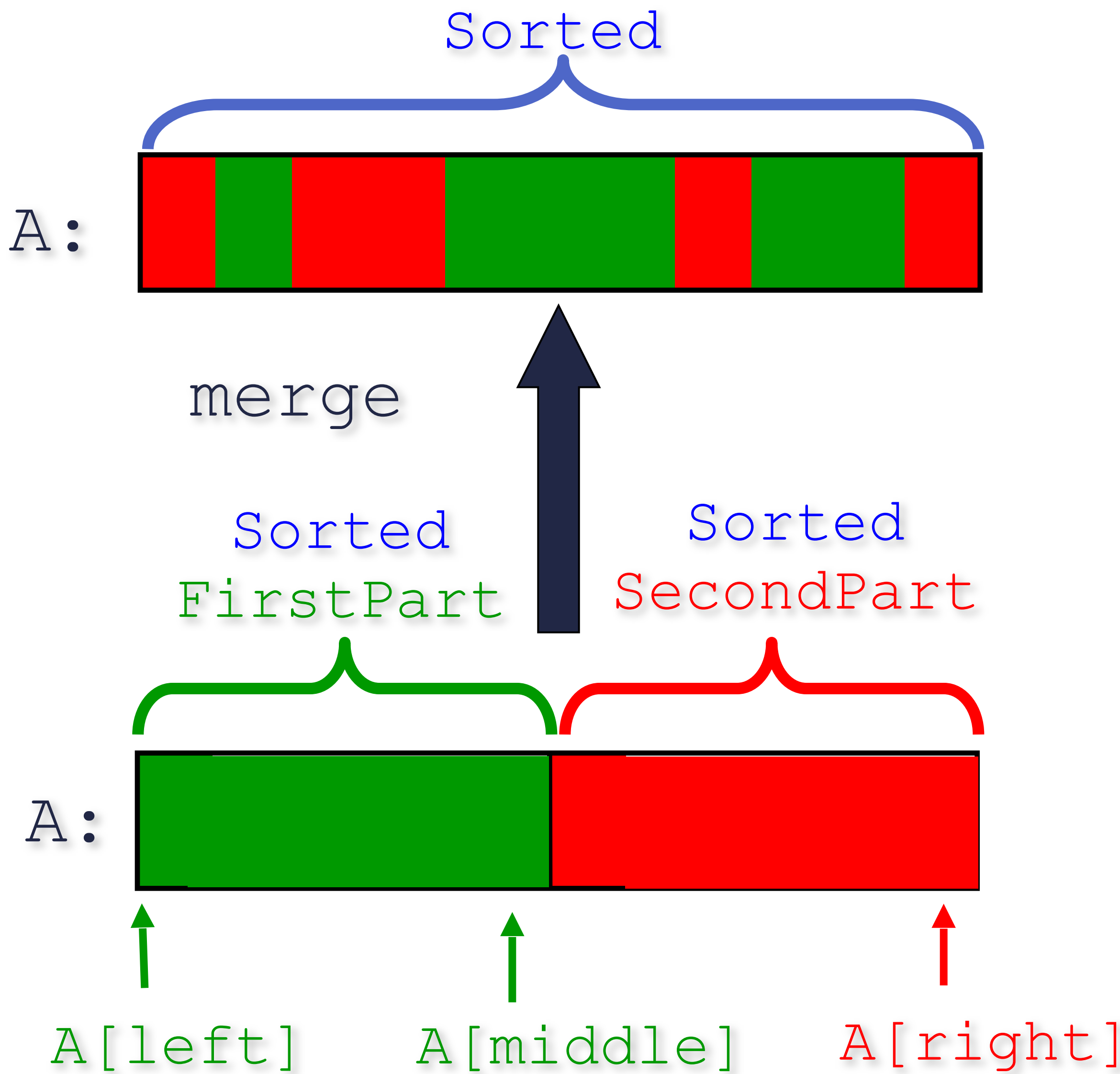
Merge Sort: Algorithm

```
MergeSort (A, left, right)
  if (left >= right) return
  else {
        middle = Floor((left+right)/2)
        MergeSort(A, left, middle)
        MergeSort(A, middle+1, right)
        Merge(A, left, middle, right)
   }
}
```

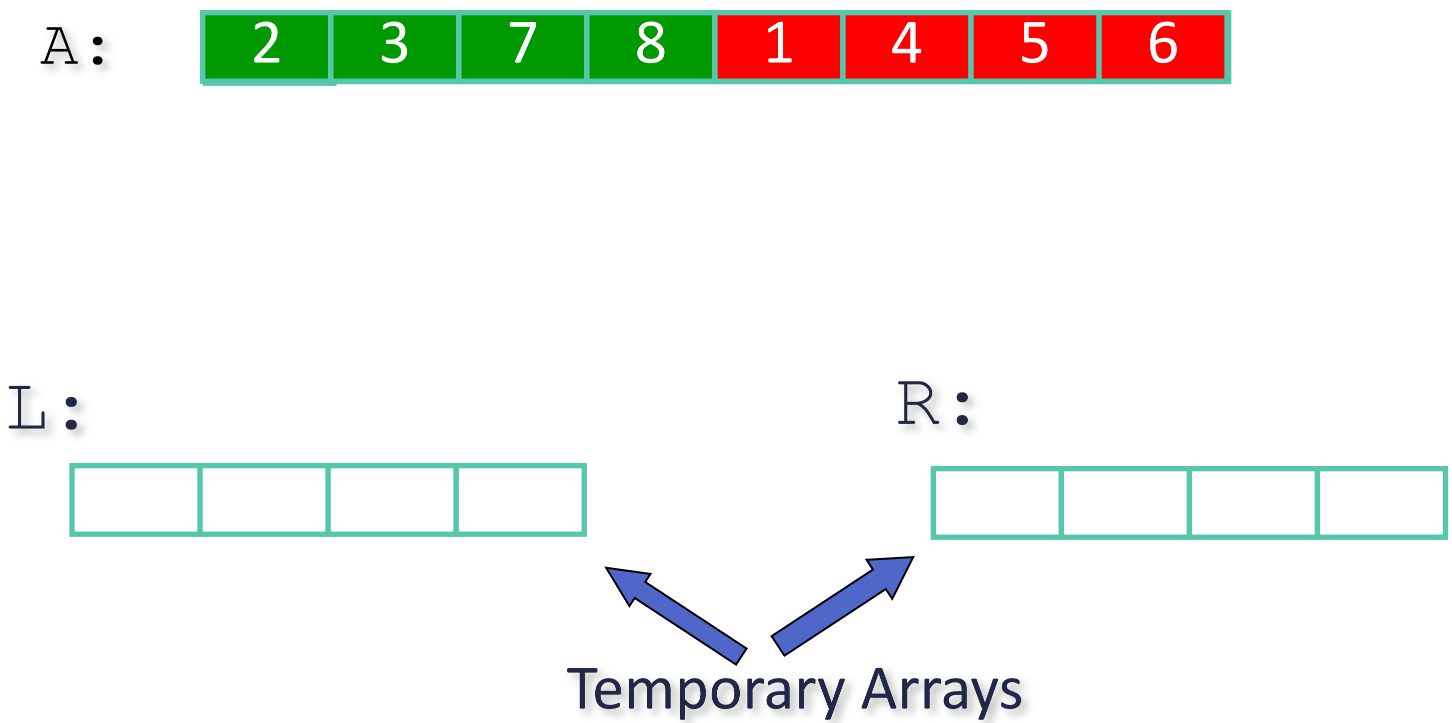
Recursive Call

Merge: Given two sorted arrays, merges them into a single sorted array

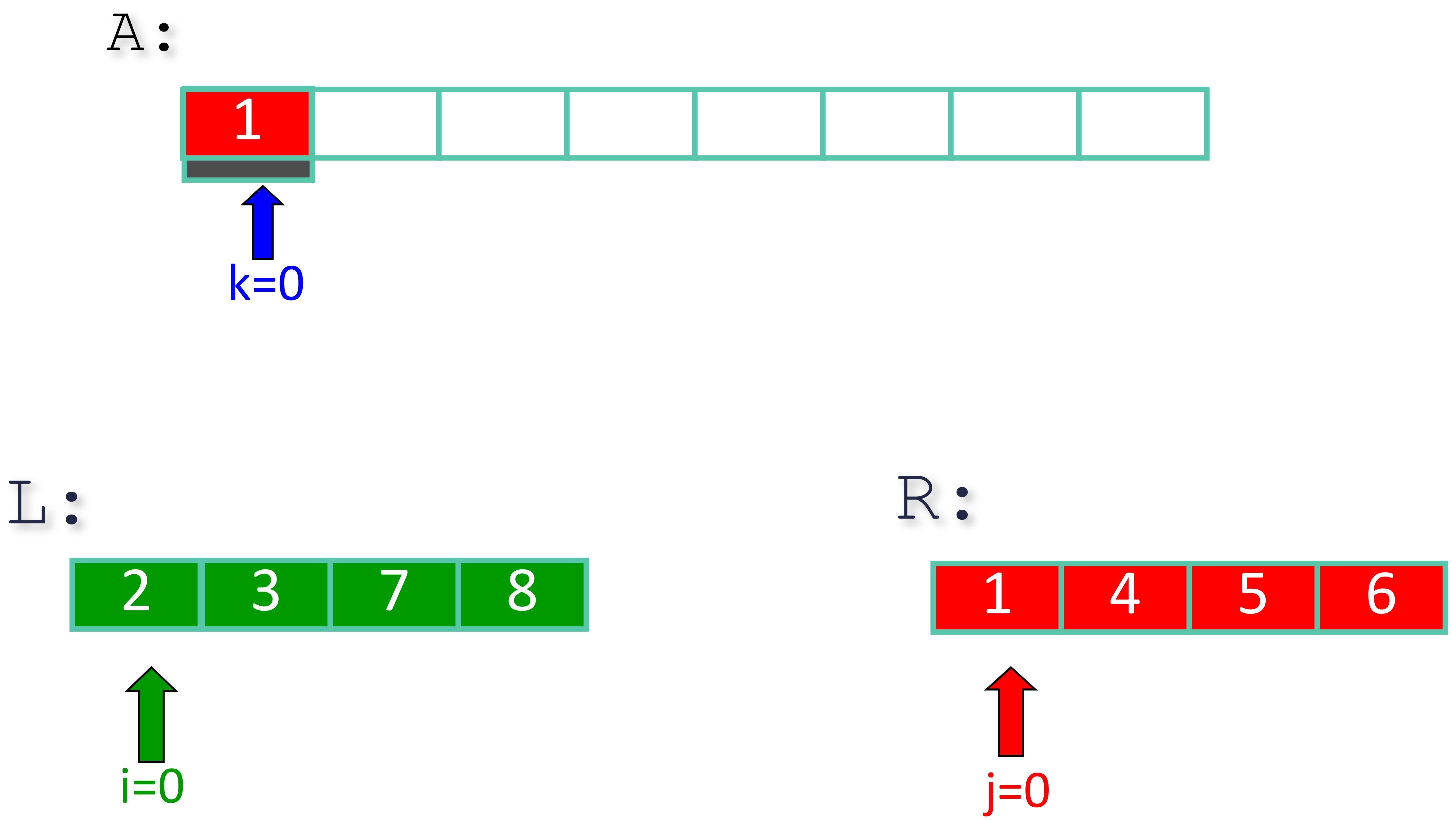
Merge-Sort: Merge



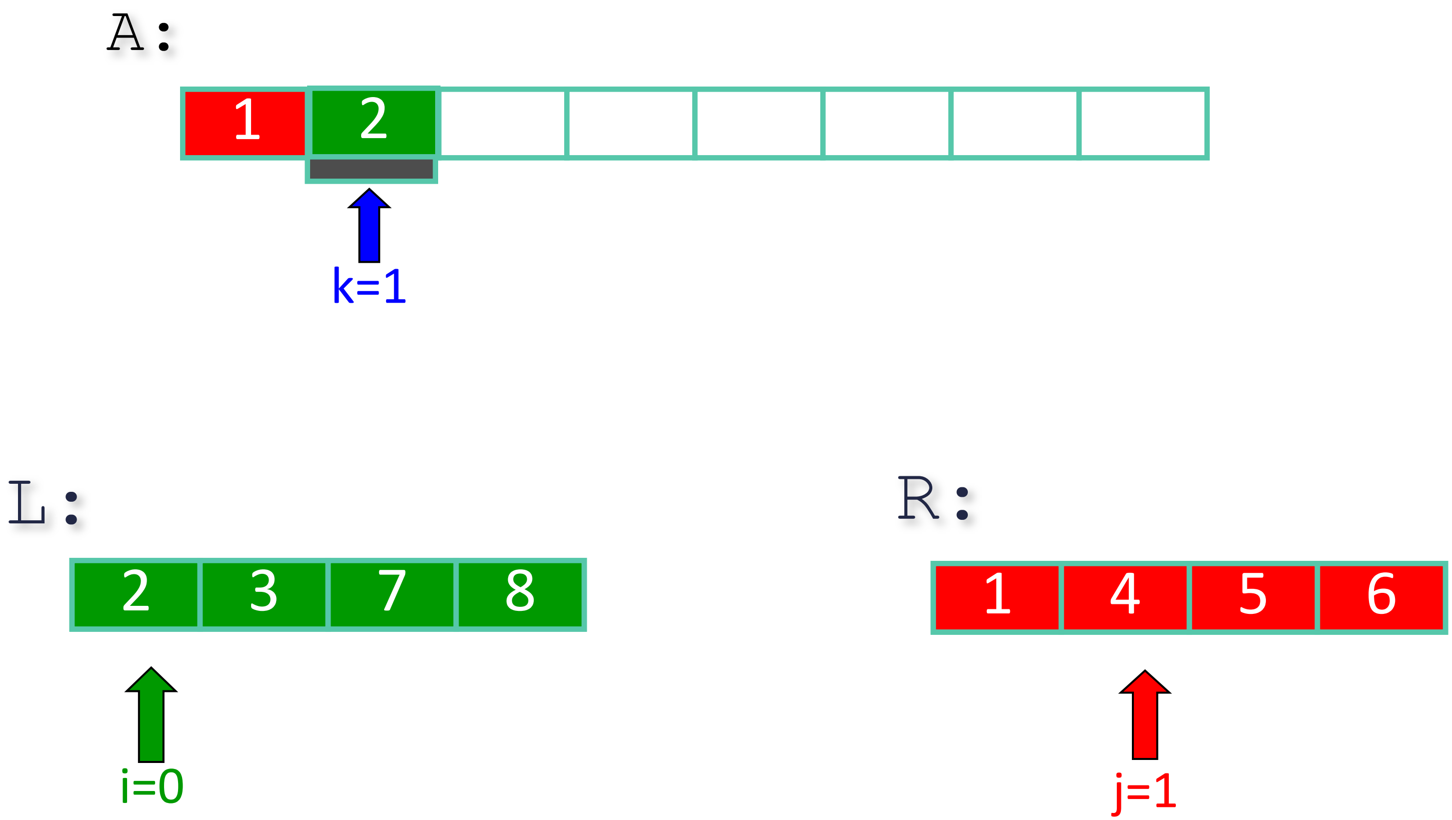
Merge-Sort: Merge



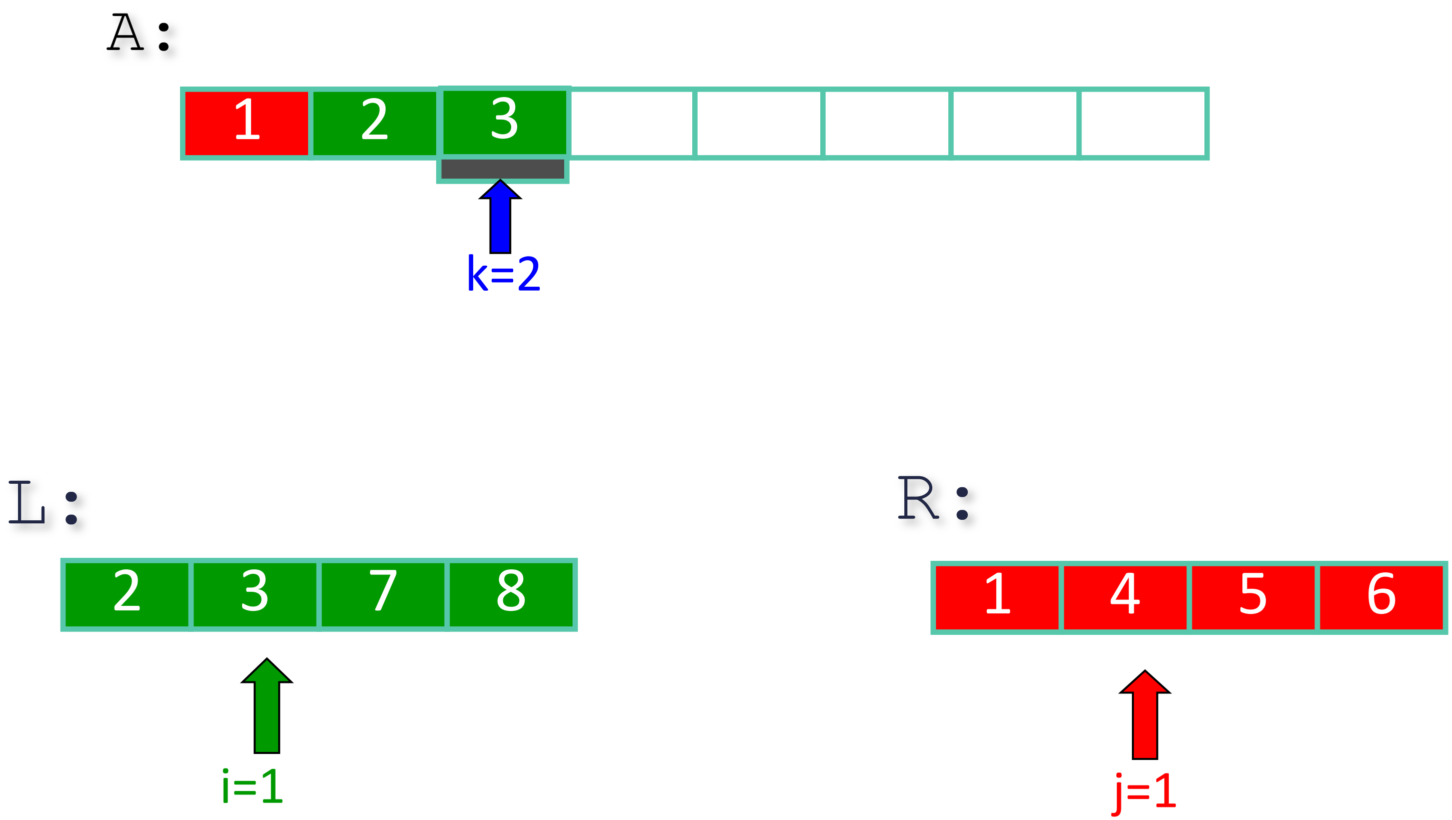
Merge-Sort: Merge



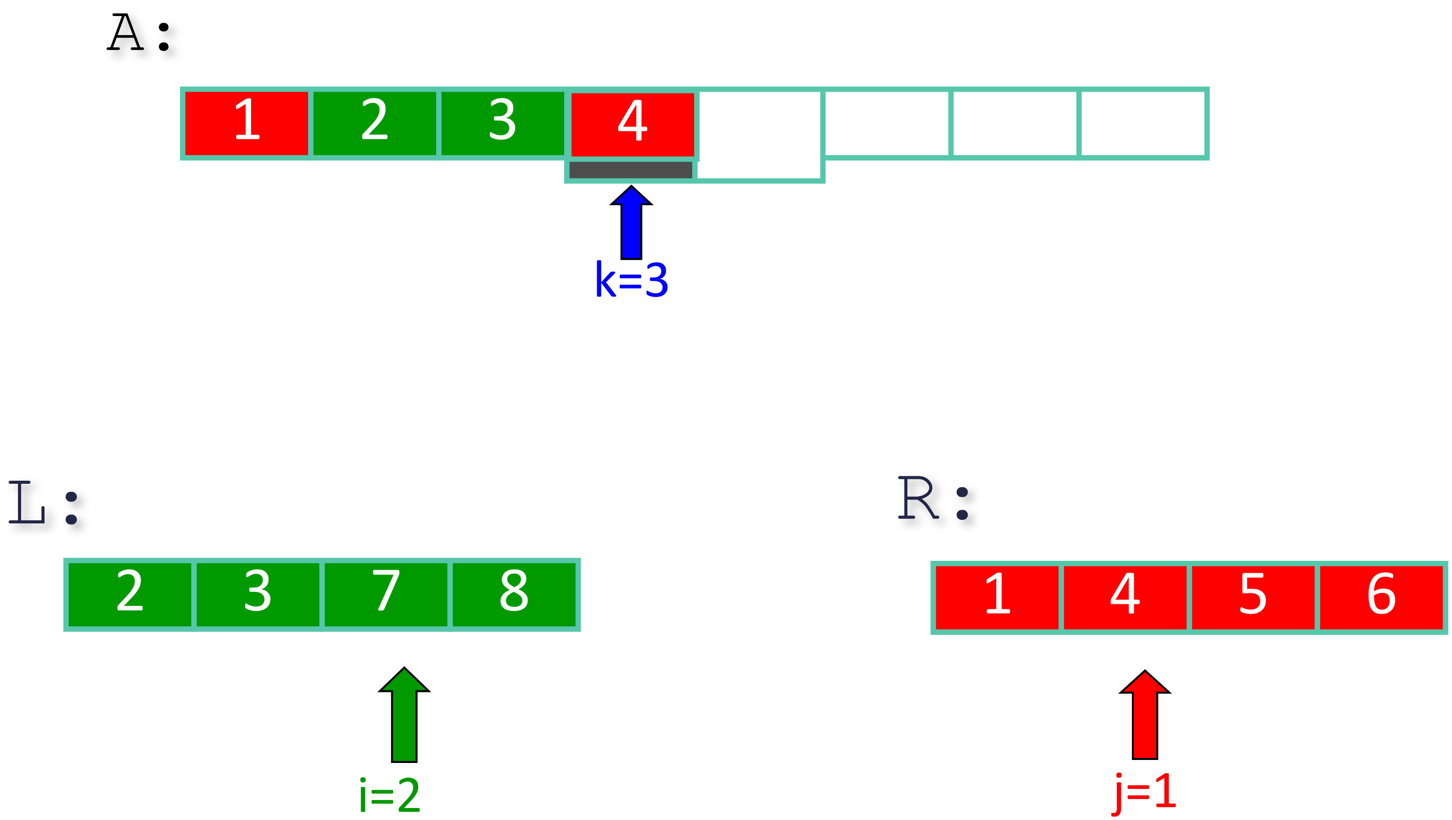
Merge-Sort: Merge



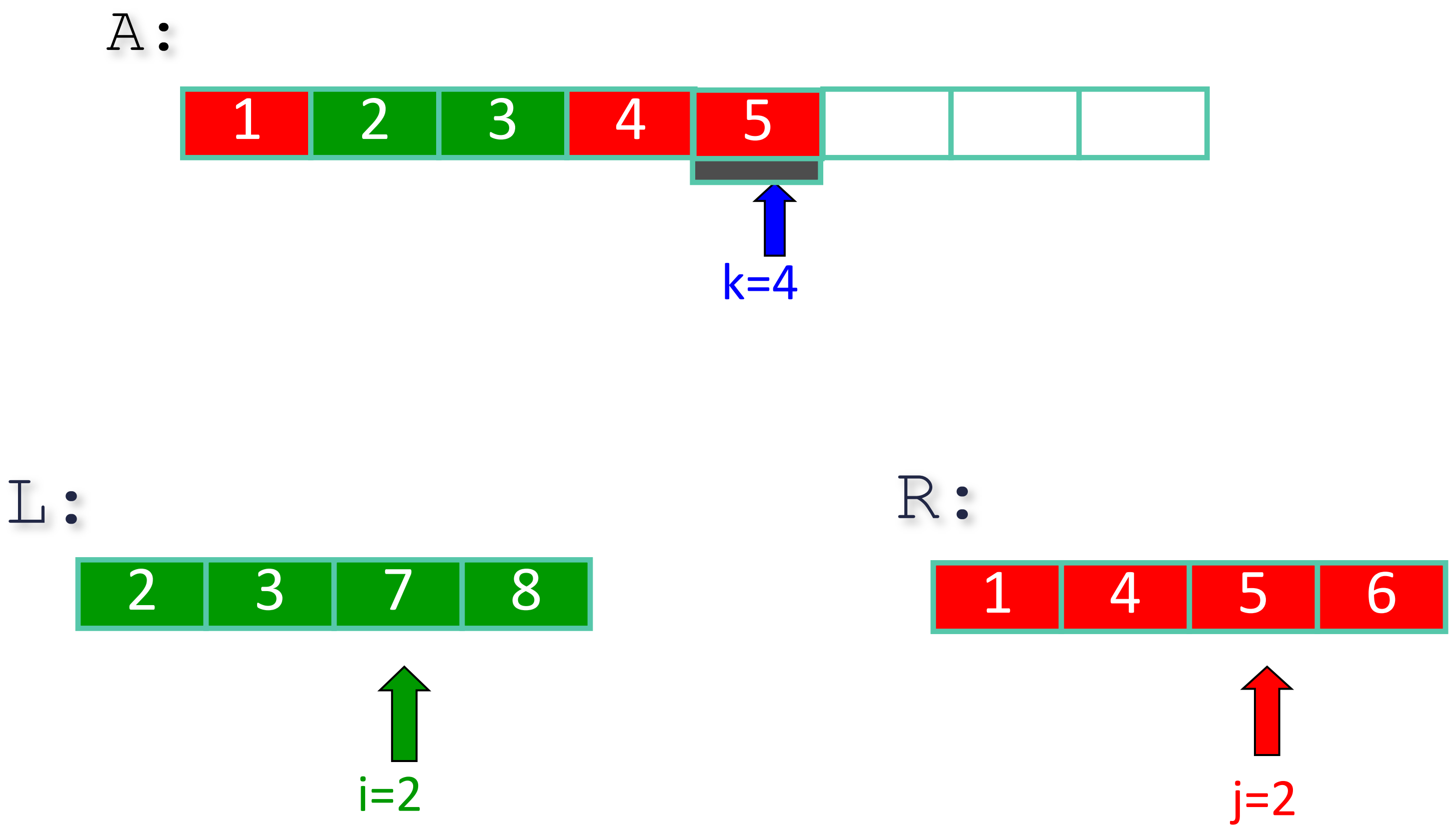
Merge-Sort: Merge



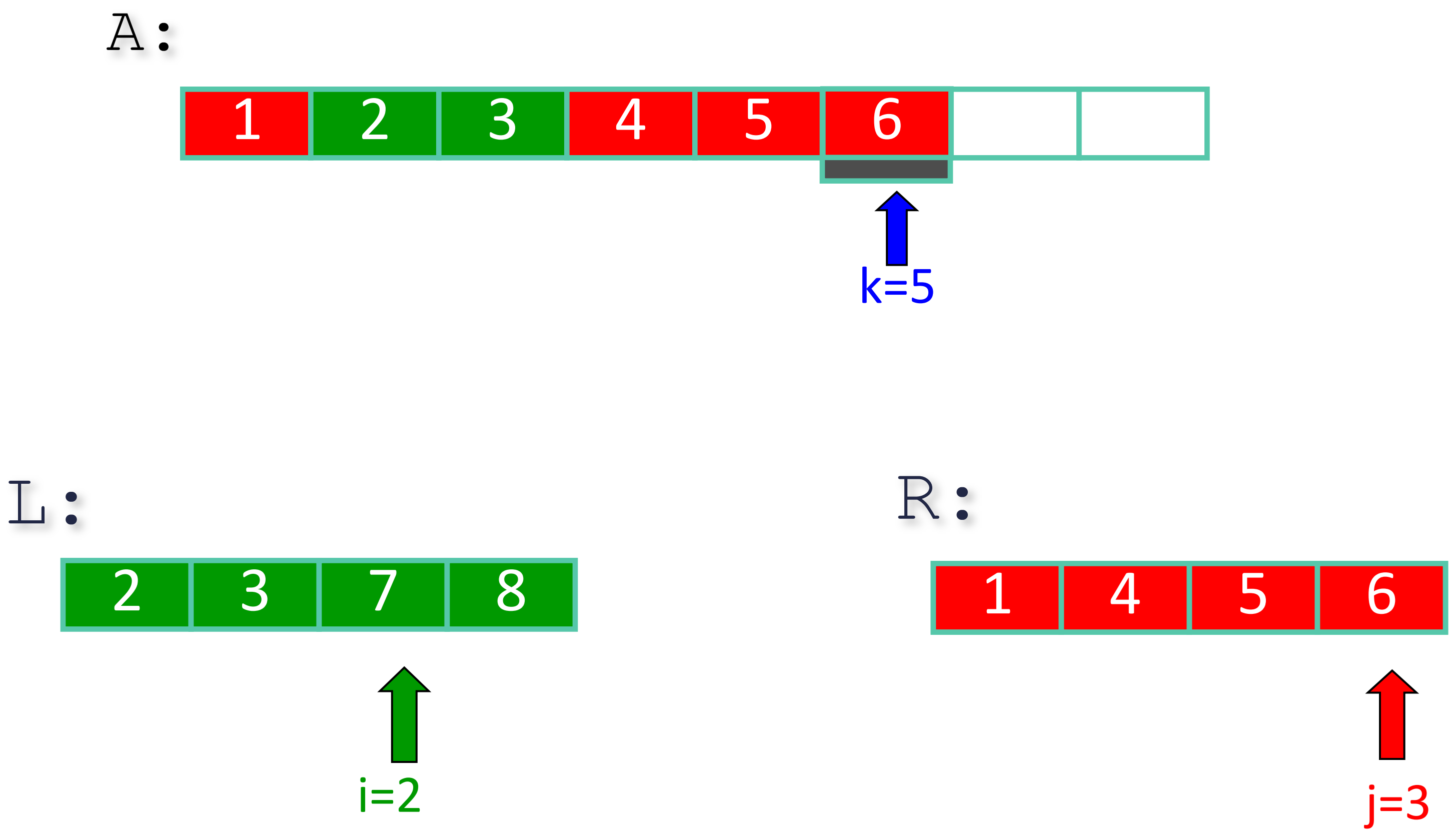
Merge-Sort: Merge



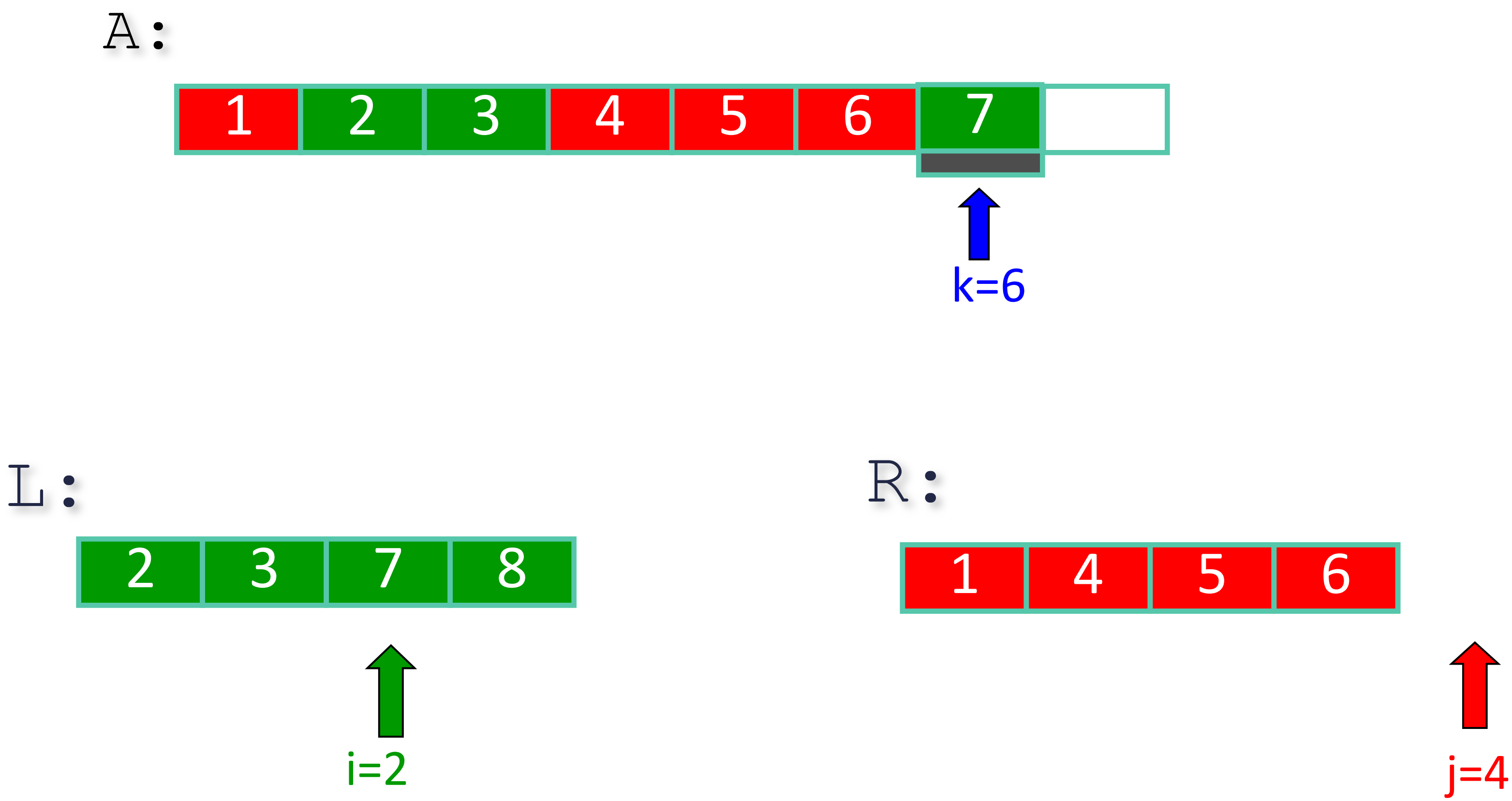
Merge-Sort: Merge



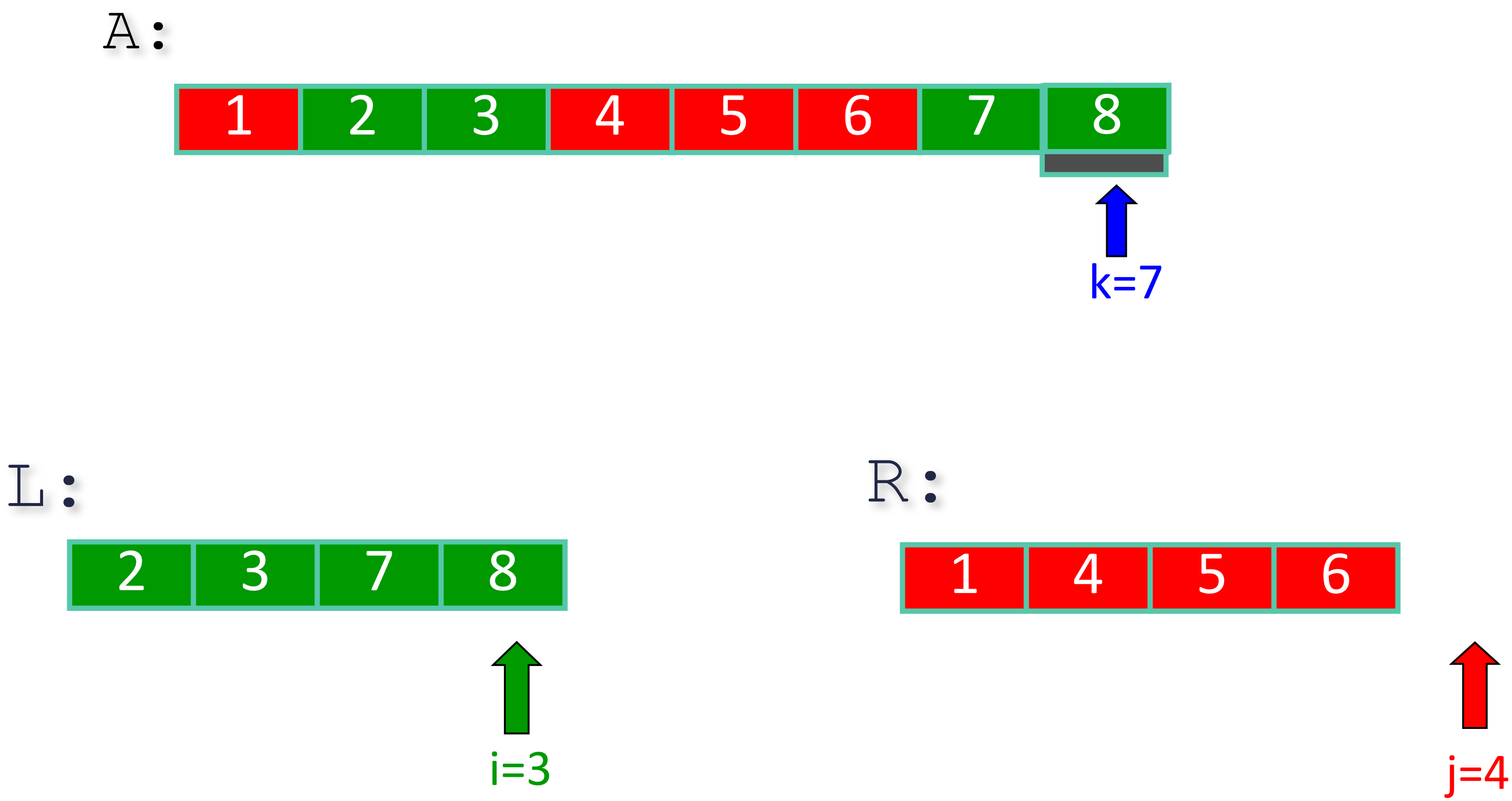
Merge-Sort: Merge



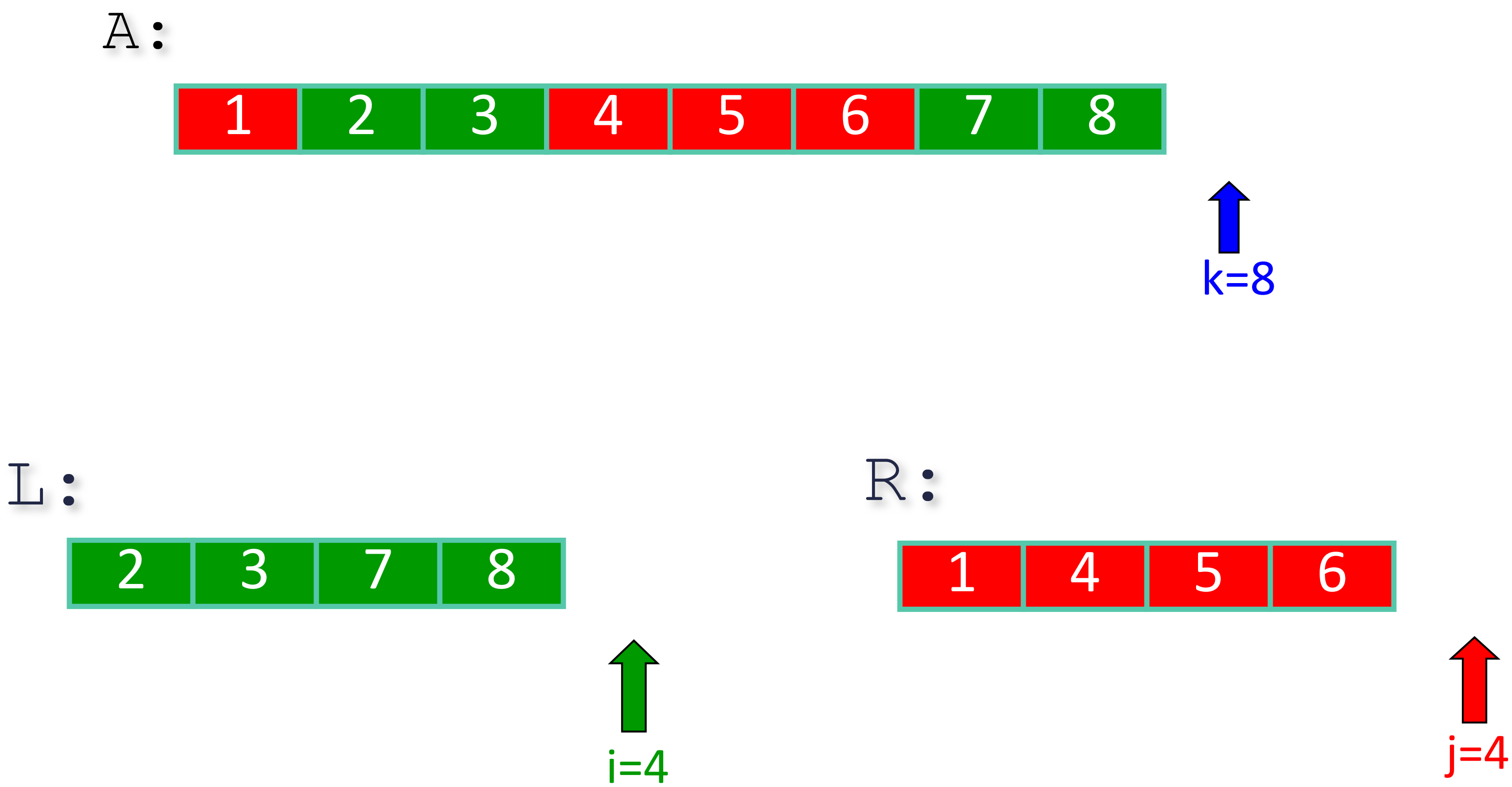
Merge-Sort: Merge



Merge-Sort: Merge



Merge-Sort: Merge

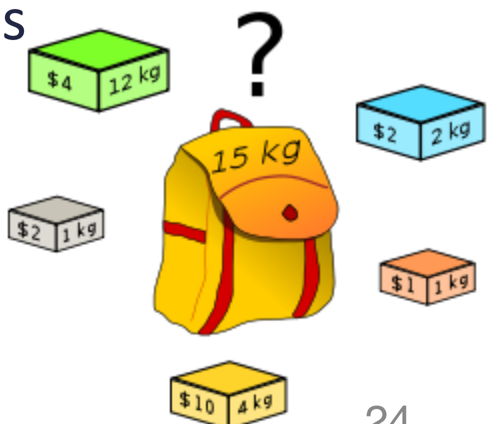


Greedy algorithms

- An optimization problem is one in which you want to find, not just *a* solution, but the *best* solution
- A "greedy algorithm" sometimes works well for optimization problems
- A greedy algorithm works in phases: At each phase:
  - You take the best you can get right now, without regard for future consequences
  - You hope that by choosing a *local* optimum at each step, you will end up at a *global* optimum
- *Any seen so far?*
- Djikstra's Shortest path problem
  - Greedily pick the shortest among the vertices touched so far

Knapsack Problem

- We are given a set of n items, where each item i is specified by a weight w_i and a value v_i. We are also given a weight bound W (the capacity of knapsack).
- The goal is to find the subset of items of maximum total value such that sum of their weights is at most W (they all fit into the knapsack).
  - Exponential time to try all possible subsets
  - O(n.W) is achievable using *dynamic programming (DP)*



© Keenan Pepper

Knapsack Problem

- 0-1 Knapsack:
  - *n* items (can be the same or different)
  - Must **leave or take** (i.e. 0-1) each item (e.g. bars of gold)
  - Greedy does not guarantee maximum value (why?)

- Fractional Knapsack:
  - *n* items (can be the same or different)
  - Can take **fractional part** of each item (e.g. gold dust)
  - Greedy guarantees maximum value (why?)

http://www.radford.edu/~nokie/classes/360/greedy.html

Greedy Solution 1

- From the remaining objects, select the object with maximum value that fits into the knapsack
- *Does not guarantee an optimal solution*
- E.g., n=3, w=[100,10,10], v=[20,15,15], weight bound W=105

Greedy Solution 2

- Select the one with minimum weight that fits into the knapsack
- *Also, does not guarantee optimal solution*
- E.g., n=2, w=[10,20], v=[5,100], W=25

Greedy Solution 3

- Select the one with maximum value density v_i/w_i that fits into the knapsack
- E.g., n=3, w=[20,15,15], v=[40,25,25], W=30
- Greedy still *does not guarantee optimal solution*
- Greedy works…if fractional items possible!

Dynamic Programming (DP)

- A dynamic programming algorithm "remembers" past results and uses them to find new results
  - *Memoization*
- Dynamic programming is generally used for optimization problems
  - Multiple solutions exist, need to find the "best" one
  - Requires "optimal substructure" and "overlapping subproblems"
    - **Optimal substructure**: Optimal solution can be constructed from optimal solutions to subproblems
    - **Overlapping subproblems**: Solutions to subproblems can be stored and reused in a bottom-up fashion
- *This differs from Divide and Conquer, where subproblems generally need not overlap*

Fibonacci numbers

- $n_i = n_{(i-1)} + n_{(i-2)}$
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …
- To find the n^{th} Fibonacci number:
  - If n is zero or one, return 1; otherwise,
  - Compute fibonacci(n-1) and fibonacci(n-2)
  - Return the sum of these two numbers

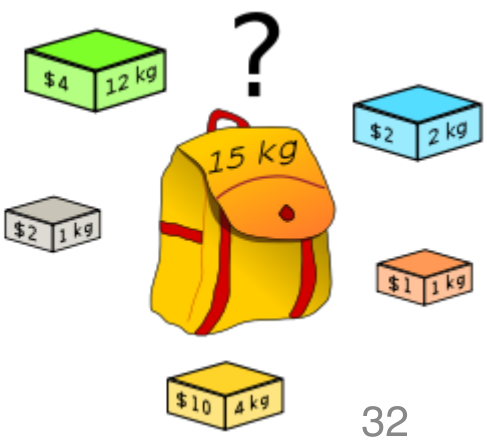
- This is a *recursive* algorithm
- Recursion leads to an *expensive* algorithm
  - Exponential time, that is, $O(2^n)$
    - *Binary tree of height 'n' with f(n) having two children, f(n-1), f(n-2)*

Fibonacci numbers again

- To find the n^{th} Fibonacci number:
  - If *n* is zero or one, return one; otherwise,
  - *Look up in a table if present, otherwise recursively compute* fibonacci(n-1),
  - Similarly, lookup or recursively compute fibonacci(n-2)
  - Find the sum of these two numbers
  - *Store the result in a table* and return it
- Since finding the n^{th} Fibonacci number involves finding all smaller Fibonacci numbers, the second recursive call has little work to do
- The table may be preserved and used again later
- Other examples: *Floyd–Warshall All-Pairs Shortest Path (APSP)* algorithm, *Towers of Hanoi*, …

Back to 0-1 Knapsack Problem

- Input: set of n items, where each item i is specified by a weight w_i and a value v_i, weight bound W
- Find the subset of items of maximum total value such that sum of their weights is at most W.
  - Solvable using *dynamic programming (DP)* - *How?*



DP for 0-1 Knapsack

```
// n = # items still to choose from, W = capacity left
MaxValue(n, W)
{
   if (n==0) return 0;
   if (arr[n][W] is known) return arr[n][W];
   if (wn > W)
      result = MaxValue (n-1, W);
   else
      result = max{vn + MaxValue(n-1, W-wn),
                        MaxValue(n-1, W)};
   arr[n][W] = result;   // store
   return result;
}
```

Brute force algorithm

- A brute force algorithm simply tries *all* possibilities until a satisfactory solution is found
- Such an algorithm can be:
  - Optimising: Find the *best* solution. This may require finding all solutions, or if a value for the best solution is known, it may stop when any best solution is found
    - Example: Finding the best path for a traveling salesman
  - Satisfying: Stop as soon as a solution is found that is *good enough*

Improving brute force algorithms

- Often, brute force algorithms require exponential time
- Various *heuristics* and *optimisations* can be used
  - Heuristic: A "rule of thumb" that helps you decide which possibilities to look at first
  - Optimisation: In this case, a way to eliminate certain possibilities without fully exploring them

Randomised algorithms

- A randomised algorithm uses a random number at least once during the computation to make a decision
  - Example: In Quicksort, using a random number to choose a pivot
  - Example: Trying to factor a large number by choosing random numbers as possible divisors