DS221: Introduction to Scalable Systems

Topic: Algorithms and Data Structures

L6: 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*

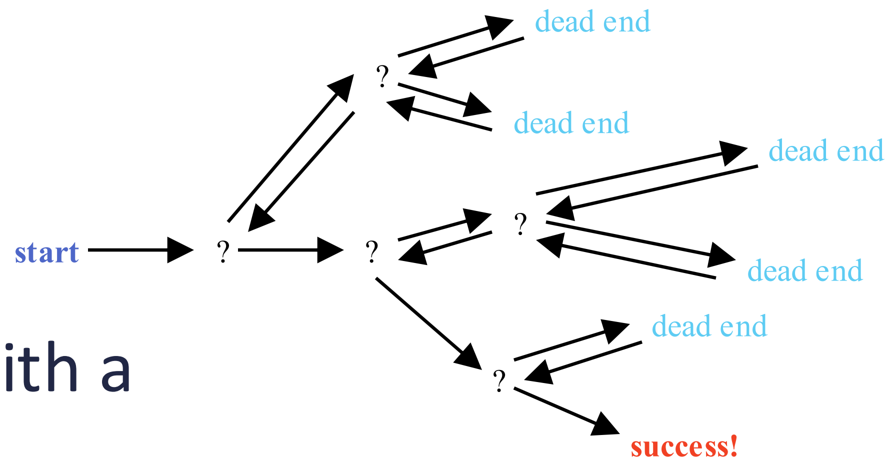
A short list of categories

- 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. Breaks the problem into a smaller subproblem and recurses.
  3. Does some extra work to convert the solutions to the simpler subproblems into a solution to the given problem
- *Any seen so far?*
  - Tree traversal
  - Binary search over sorted array

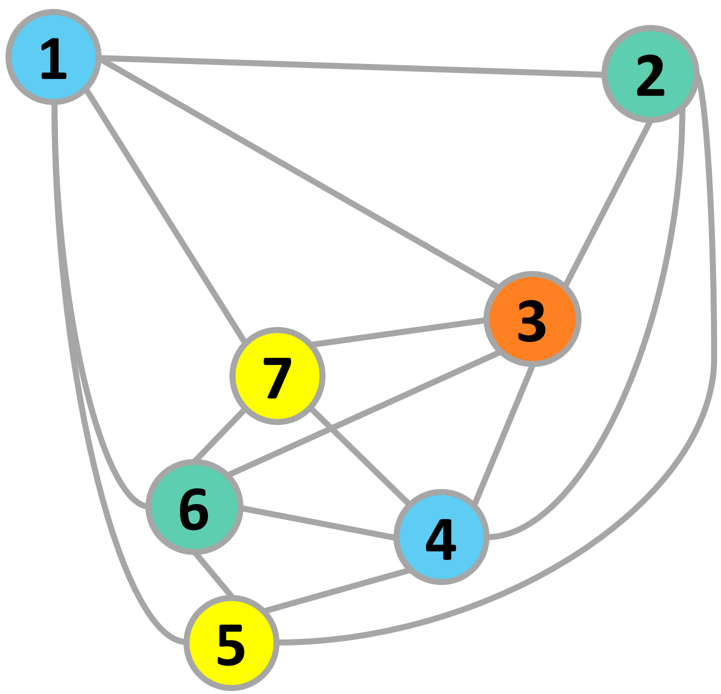
Backtracking algorithms



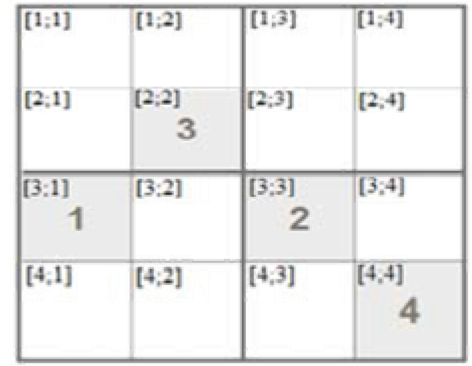
- Explore the solution space with a depth-first recursive search
- At each step:
  - Check if a complete solution is reached → return it
  - Otherwise, for every available choice:
    - Make the choice
    - Recurse
    - If recursion yields a solution → return it
- If all choices fail → return failure

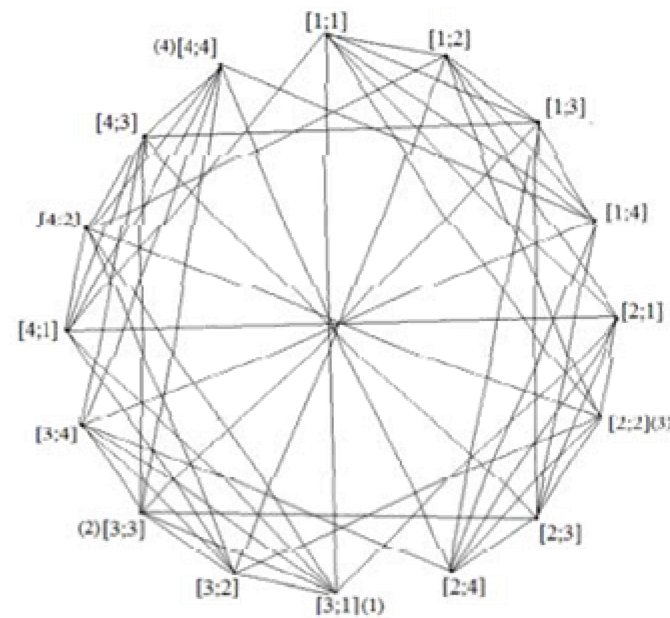
Example

Graph coloring: *Color the vertices of a graph such that no two adjacent vertices have the same color*



4x4 Sudoku





The above mentioned graph has 16 vertices and 56 edges.

Graph Coloring

- m-color problem
  - Given a graph, find out if its vertices can be colored with no more than m colors
  - $O(m^V)$
- The **Four-Color Theorem** states that any map on a plane can be colored with no more than four colors, so that no two neighbouring countries with a common border have the same color

https://en.wikipedia.org/wiki/Four_color_theorem

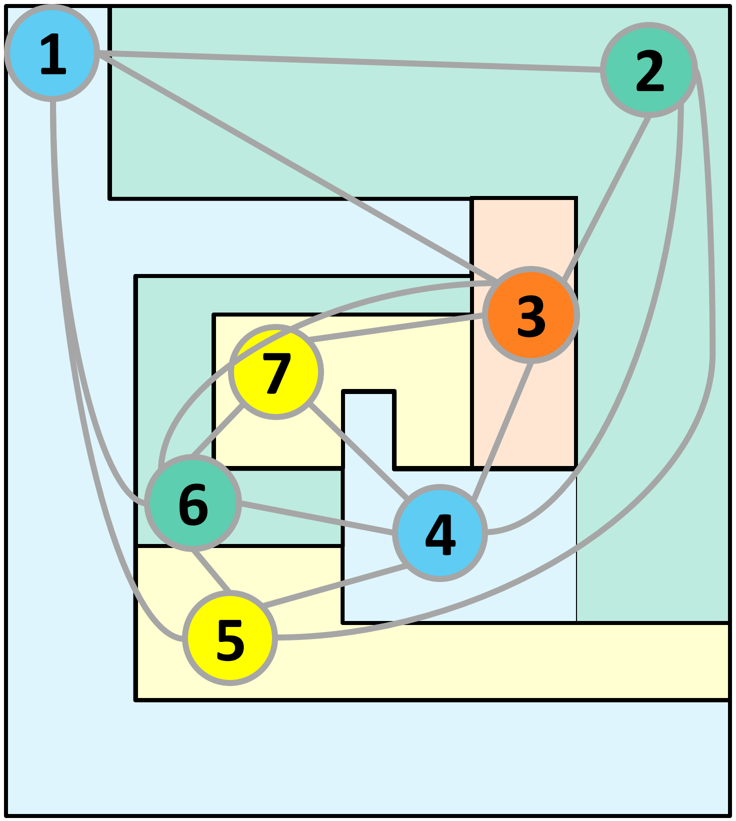
Sample backtracking algorithm

```
boolean explore(int ctry) {
  if (ctry > map.size) return true

  for (c = RED; c <= BLUE; c++)
  {
      if (okToColor(ctry, c)) {
          map[ctry] = c;
          if (explore(ctry + 1))
              return true;
          map[ctry] = NONE
      }
  }

  return false
}
```

Map (each vertex is a country)



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*

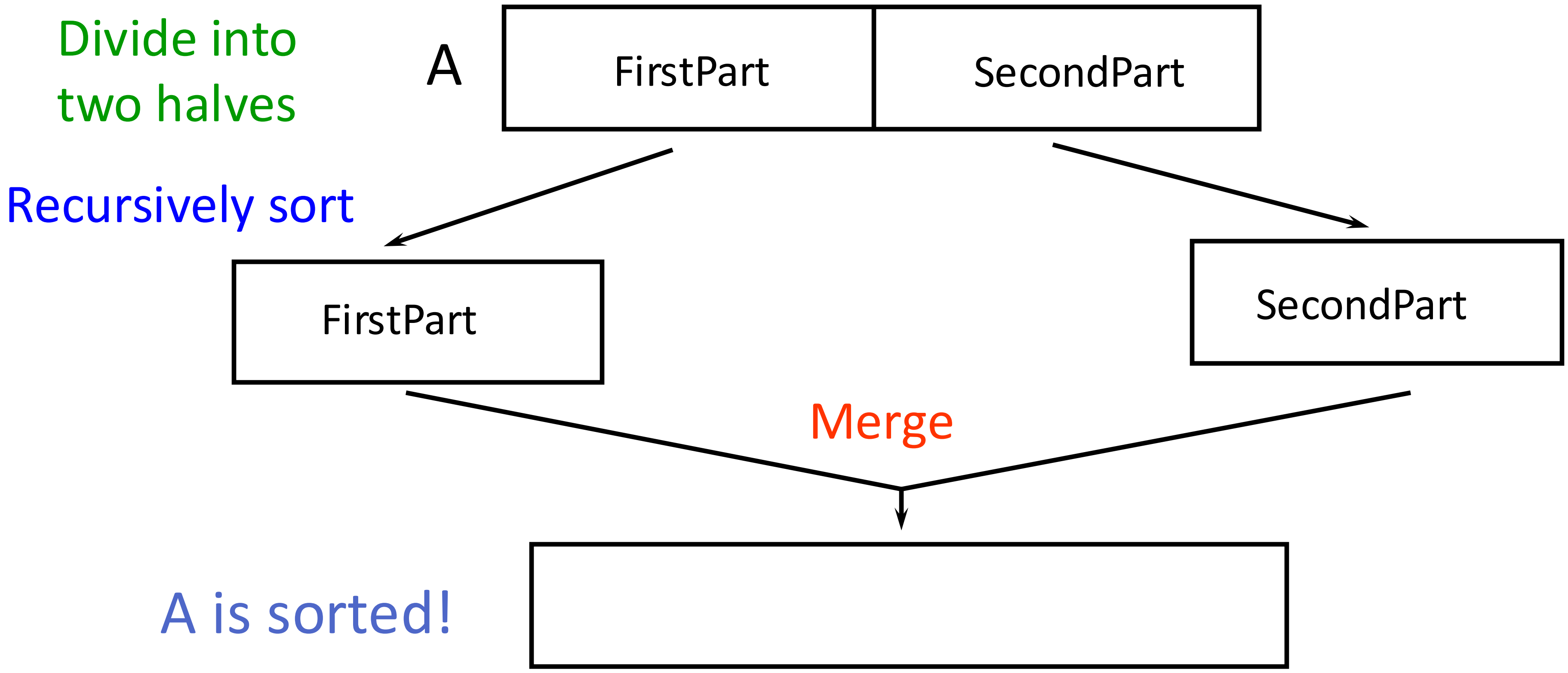
Binary search tree lookup?

- 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 even though the code contains two recursive calls, only **one branch** is actually explored at each recursion level.

- *E.g. Recursive binary search over an unsorted array. Search all elements.*
- *E.g. Merge Sort, Quick Sort*

Merge Sort: Idea



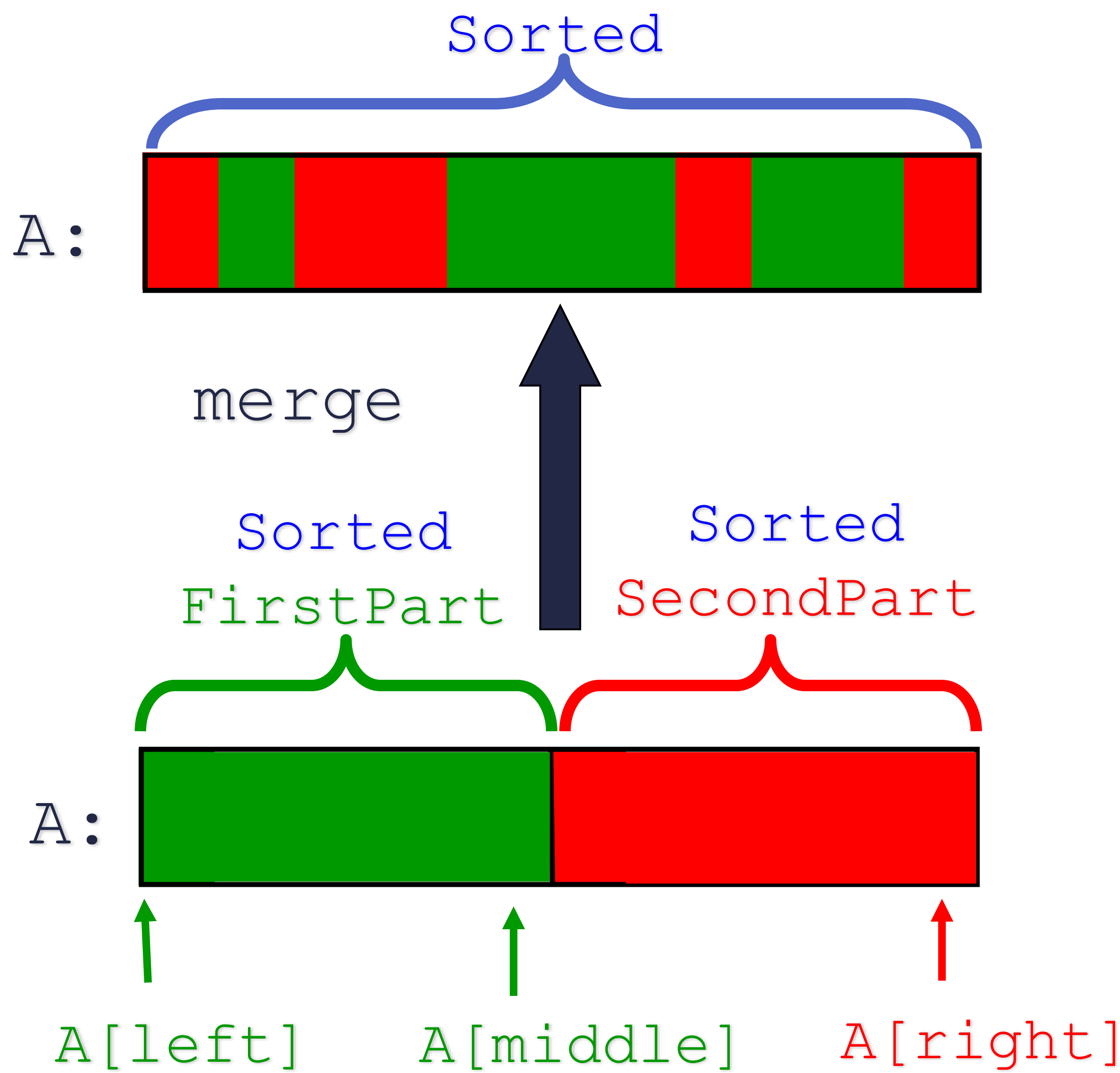
Merge Sort: Algorithm

```
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)
   }
}
```

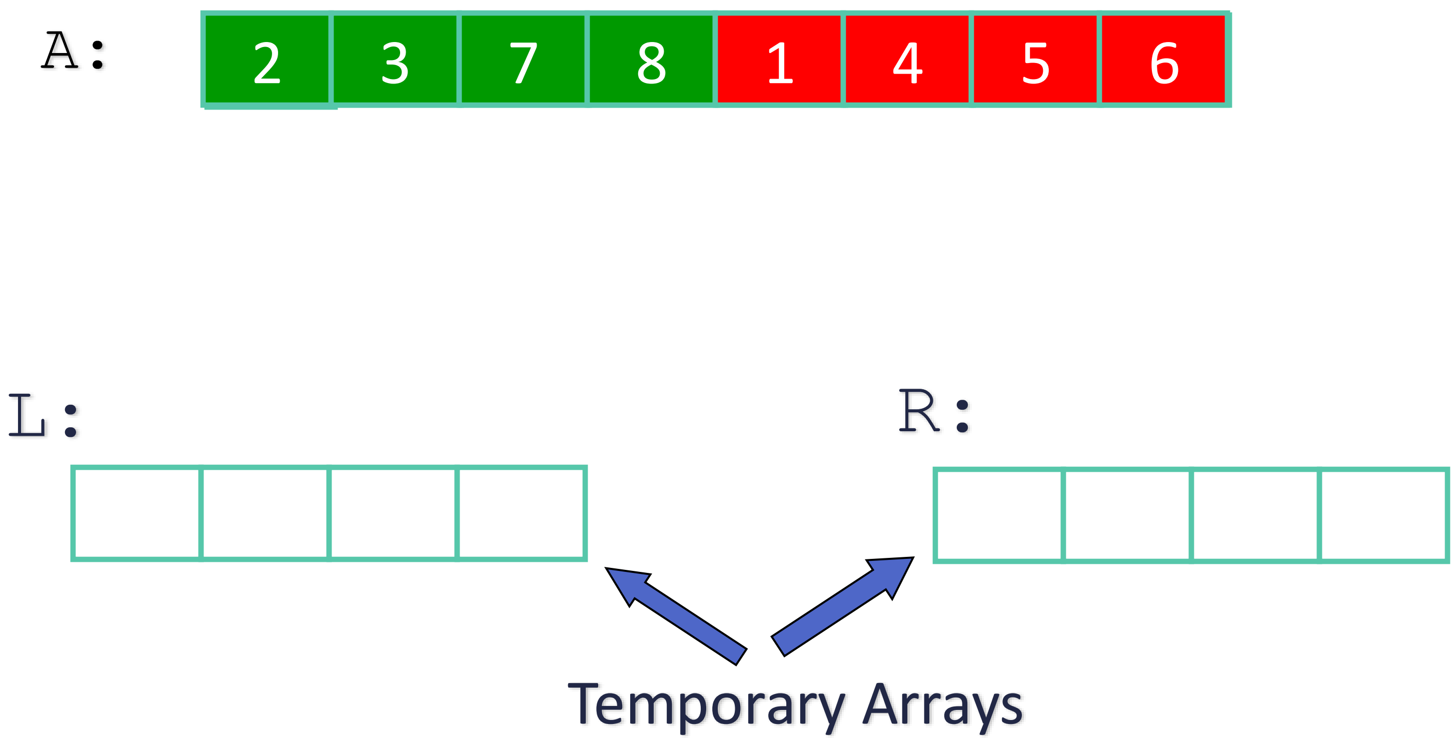
Recursive Call

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

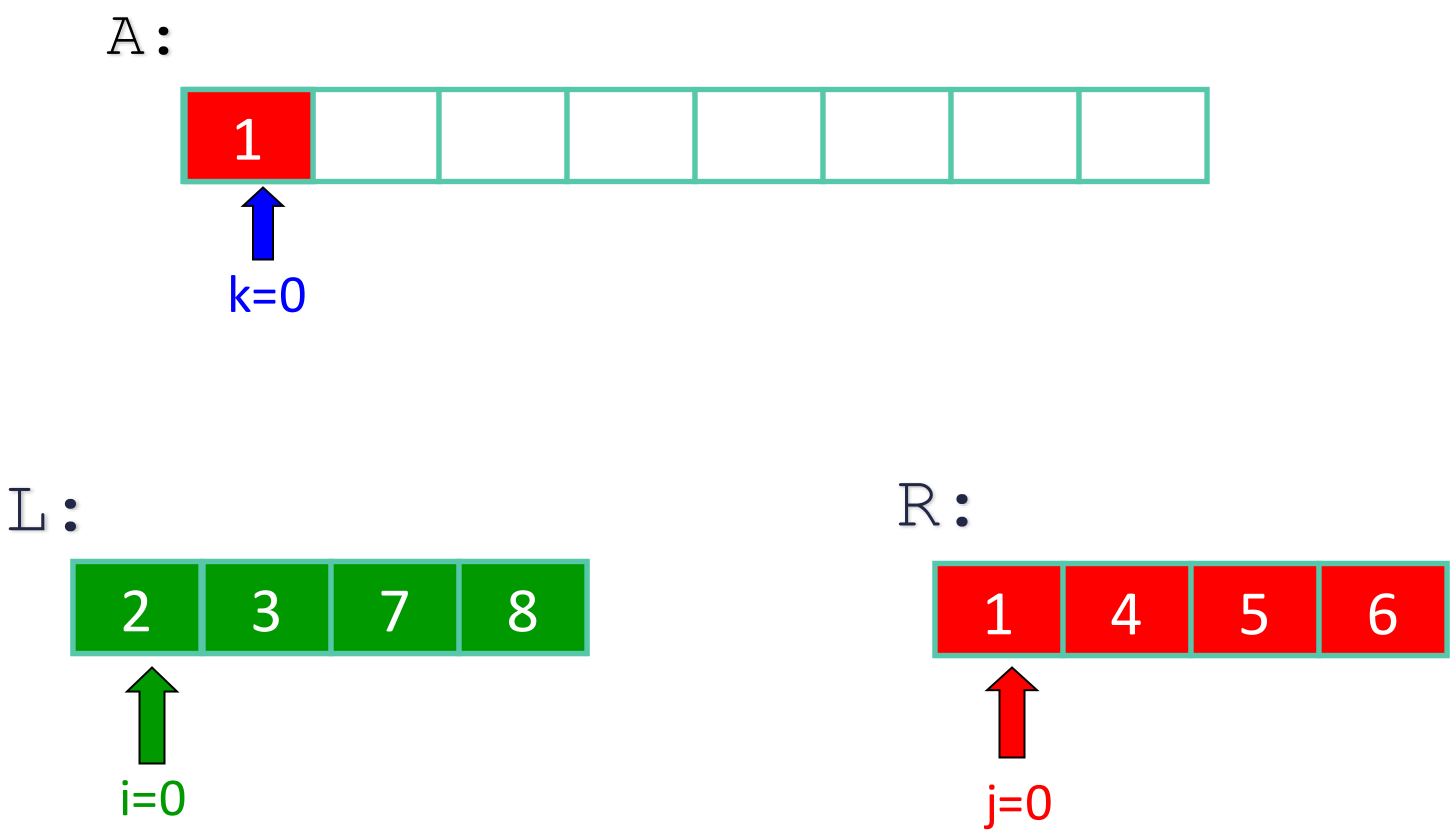
Merge-Sort: Merge



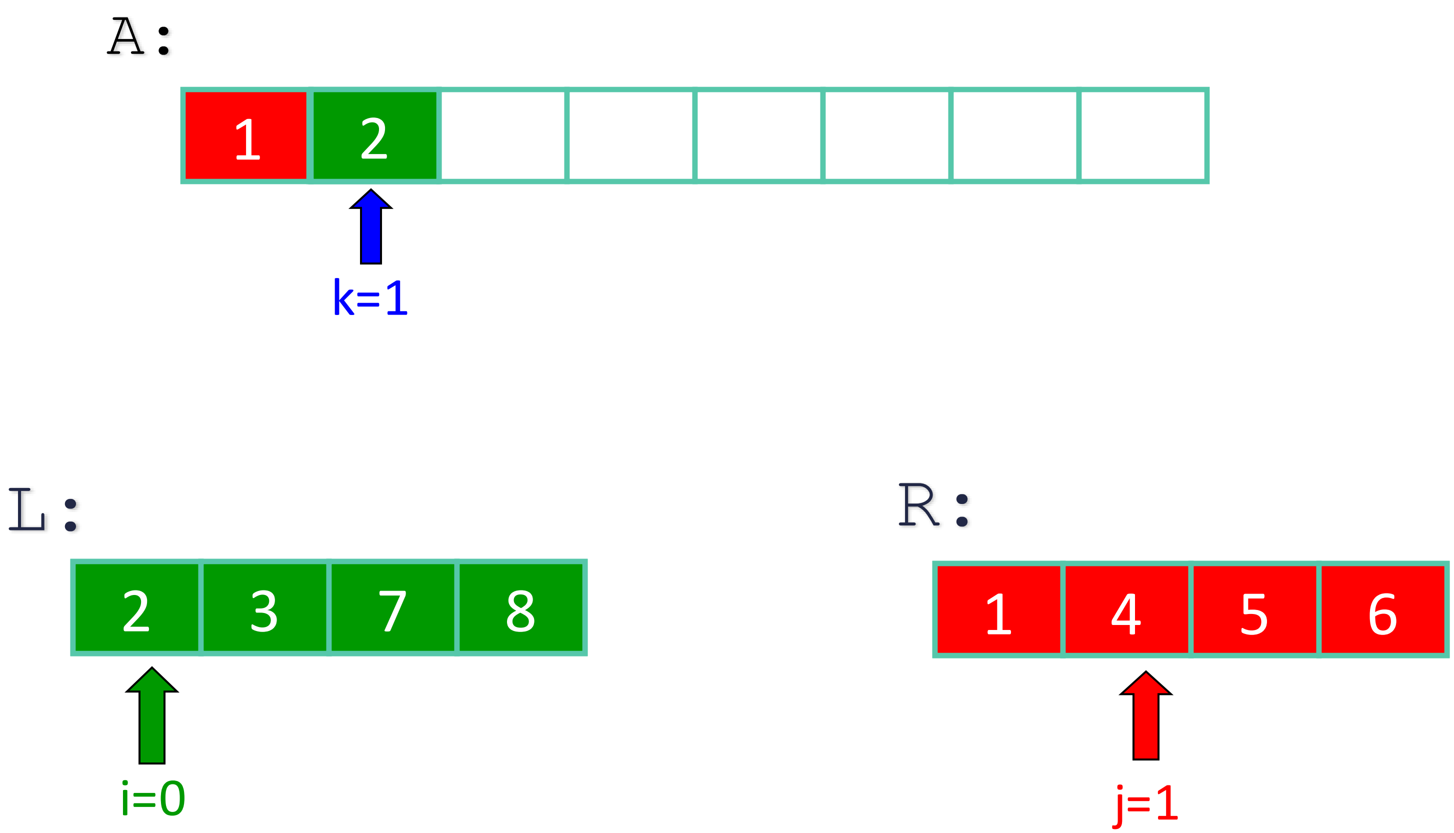
Merge-Sort: Merge



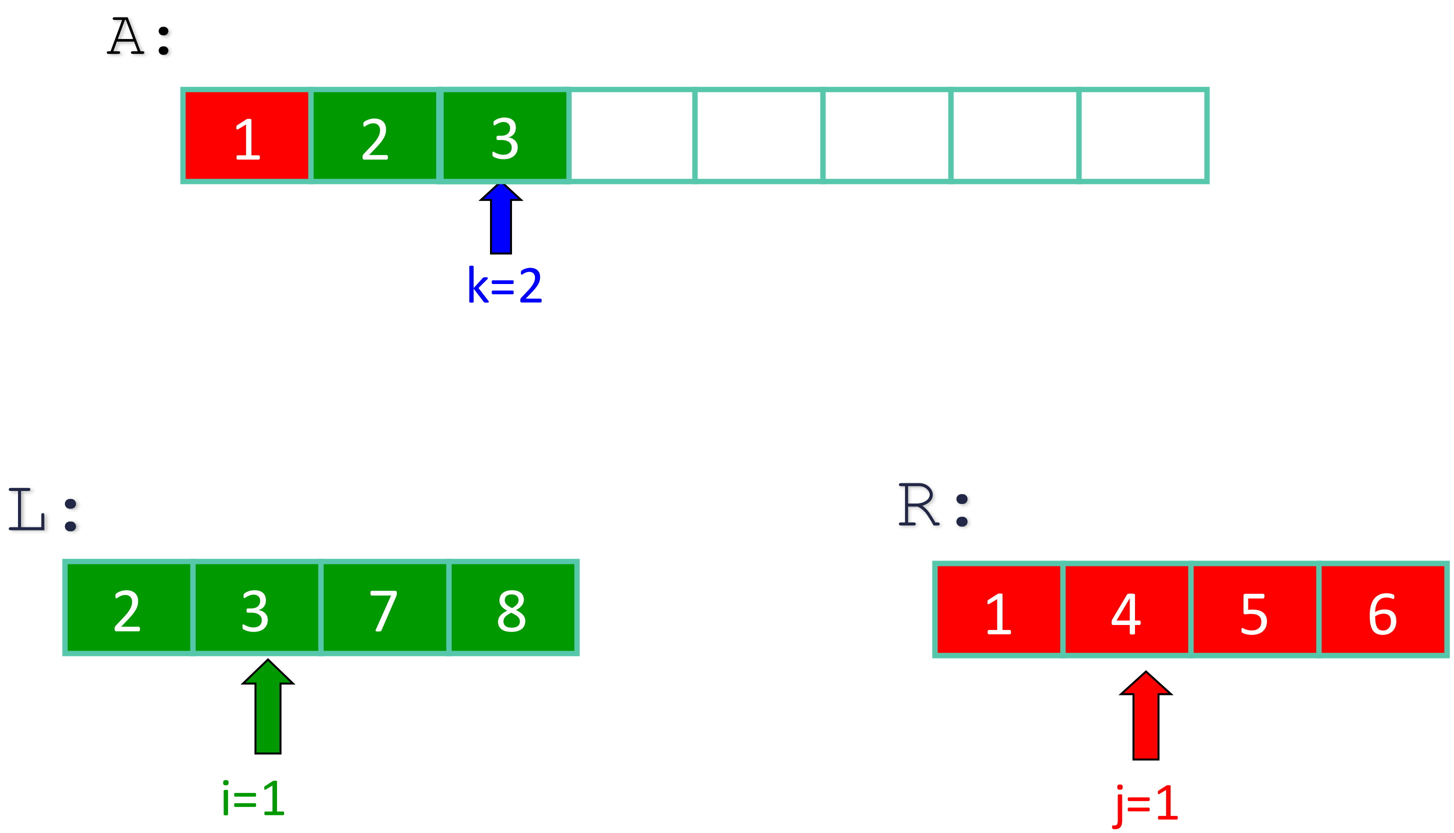
Merge-Sort: Merge



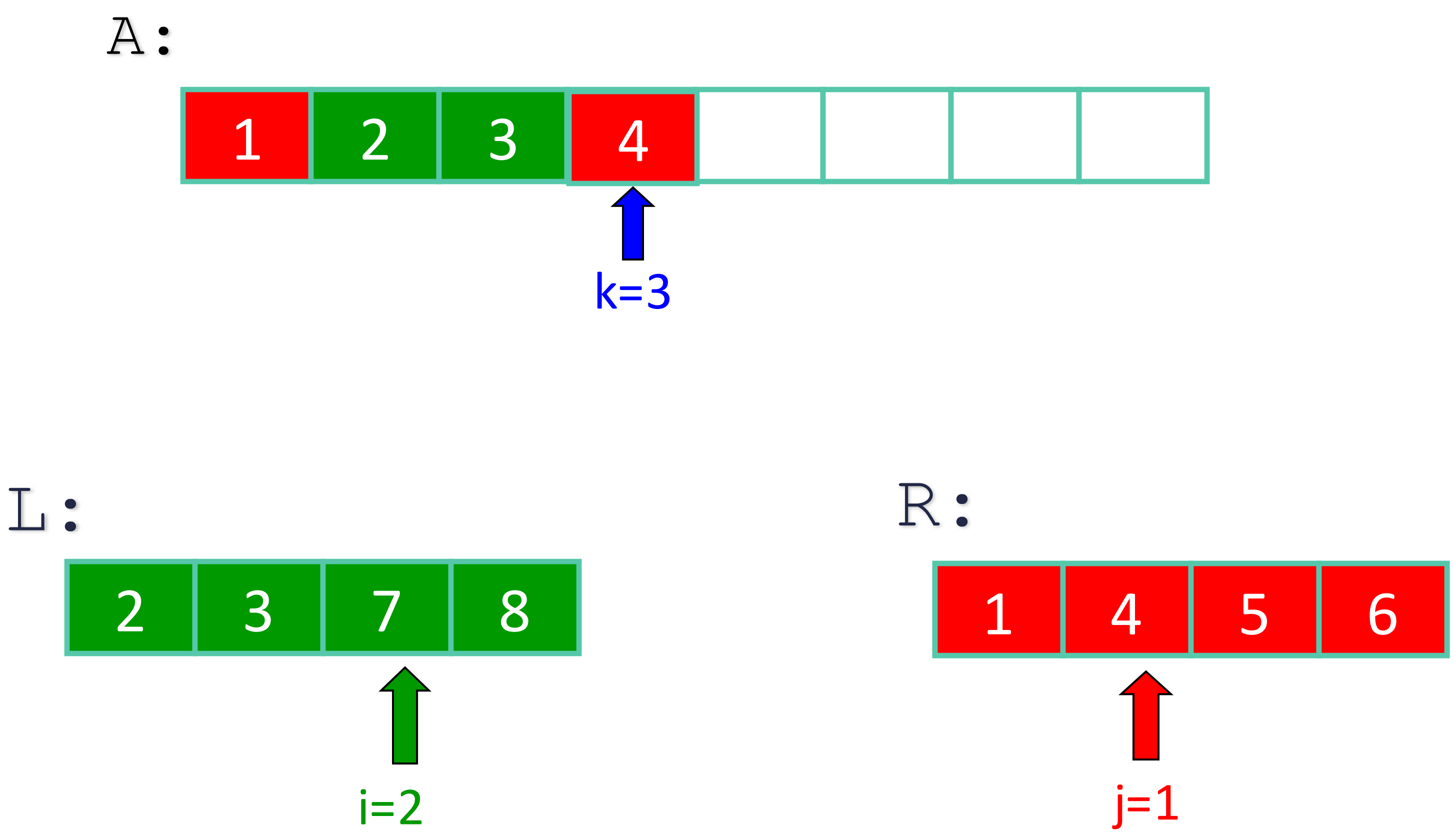
Merge-Sort: Merge



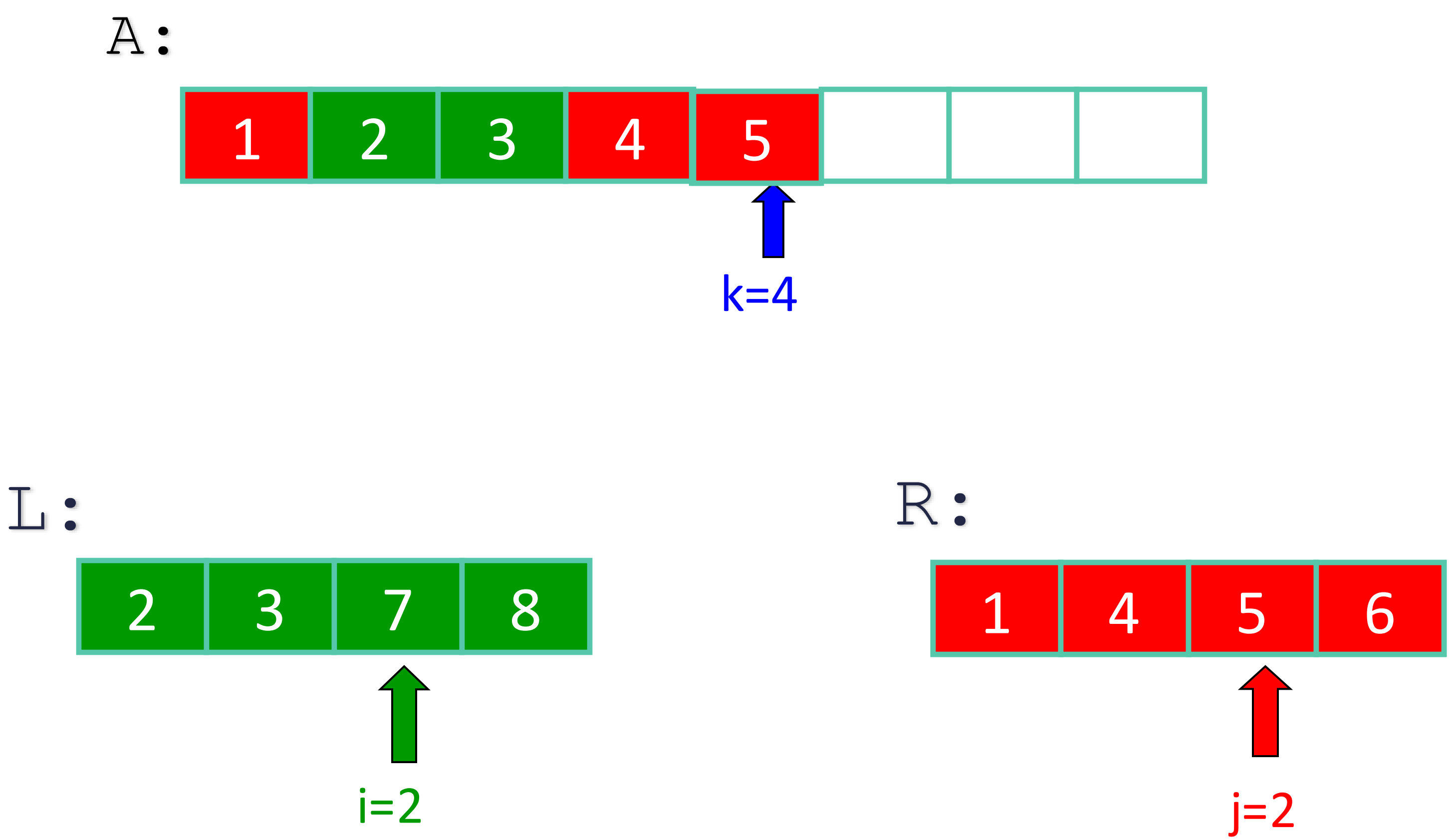
Merge-Sort: Merge



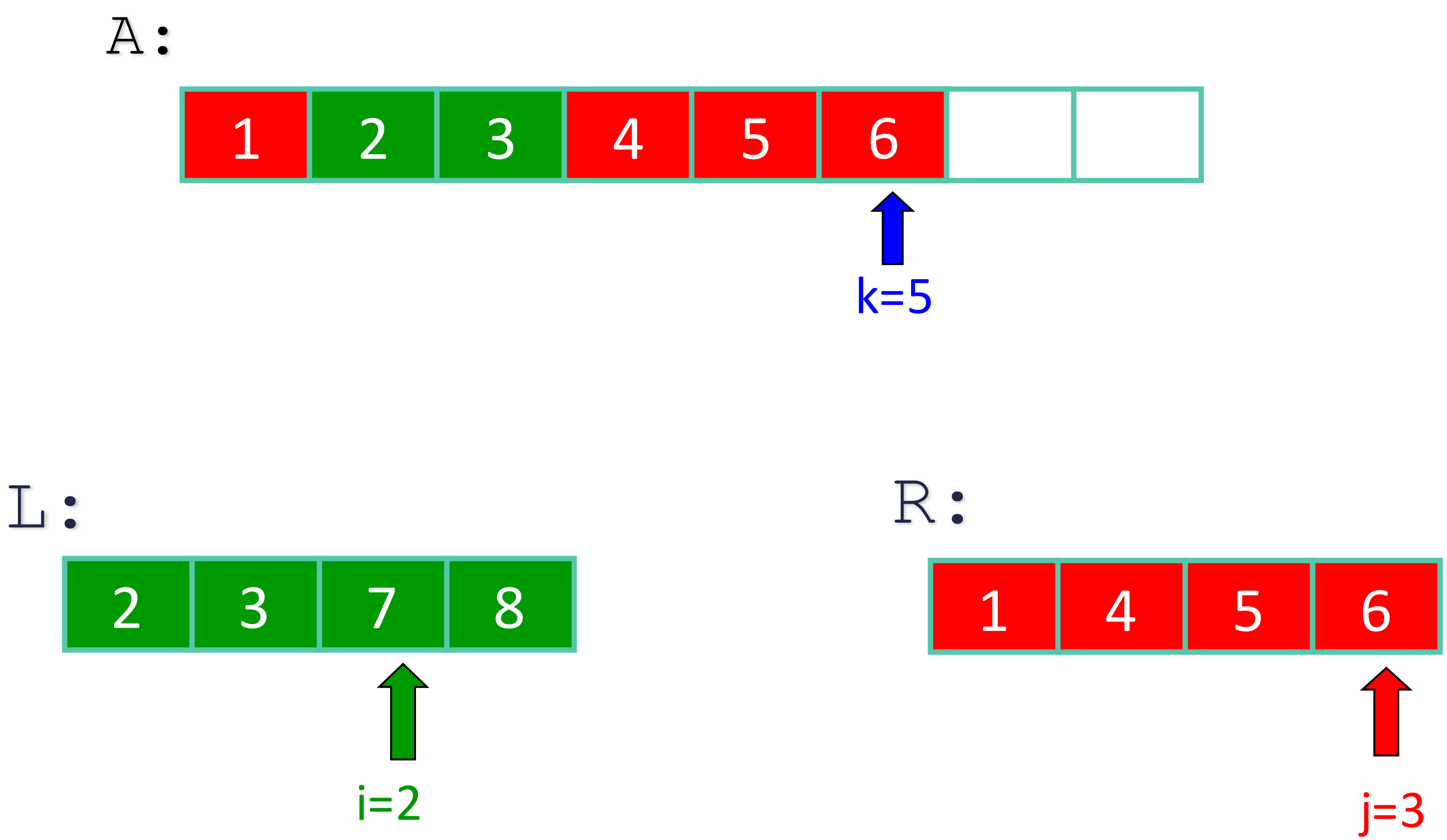
Merge-Sort: Merge



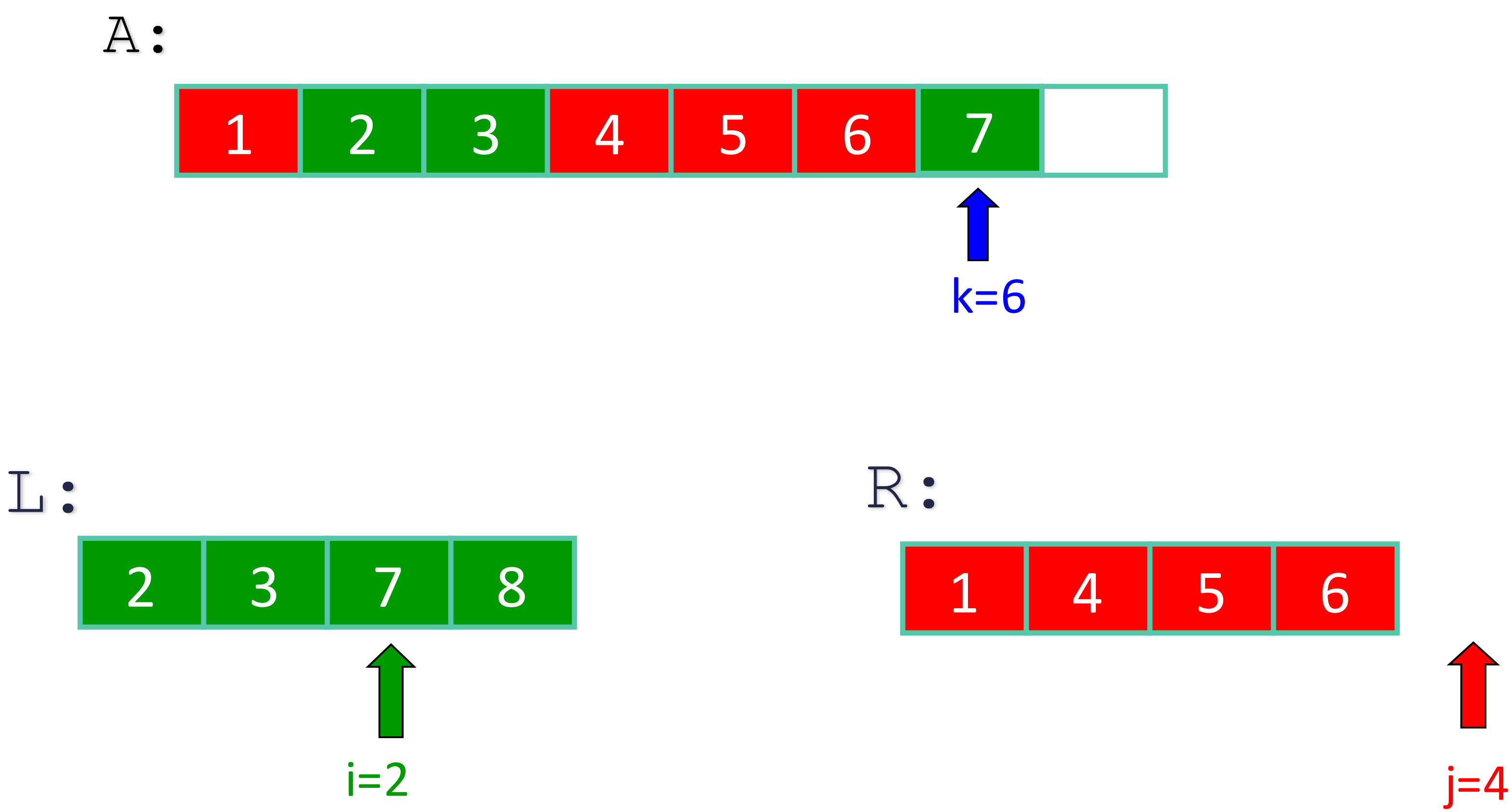
Merge-Sort: Merge



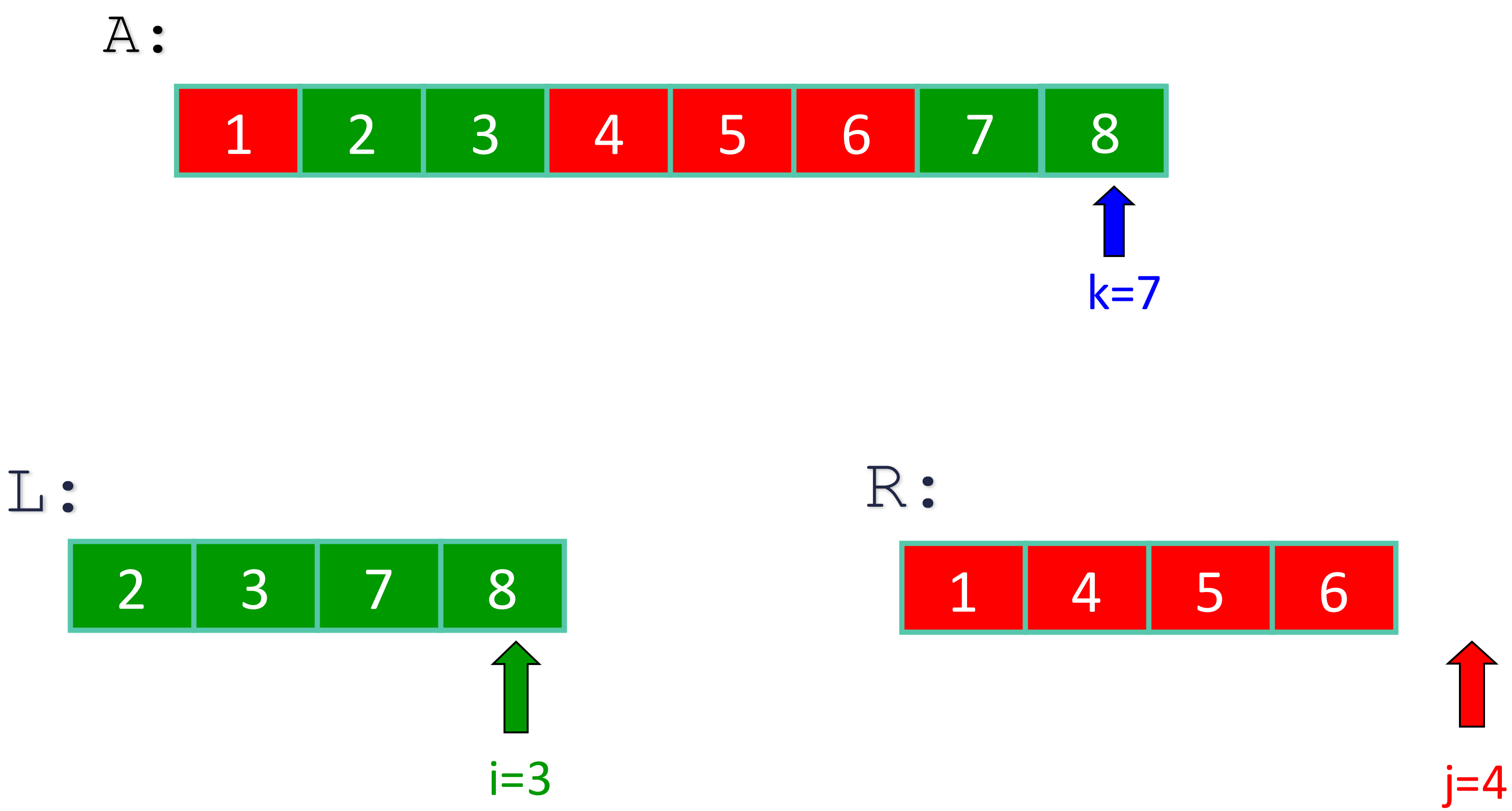
Merge-Sort: Merge



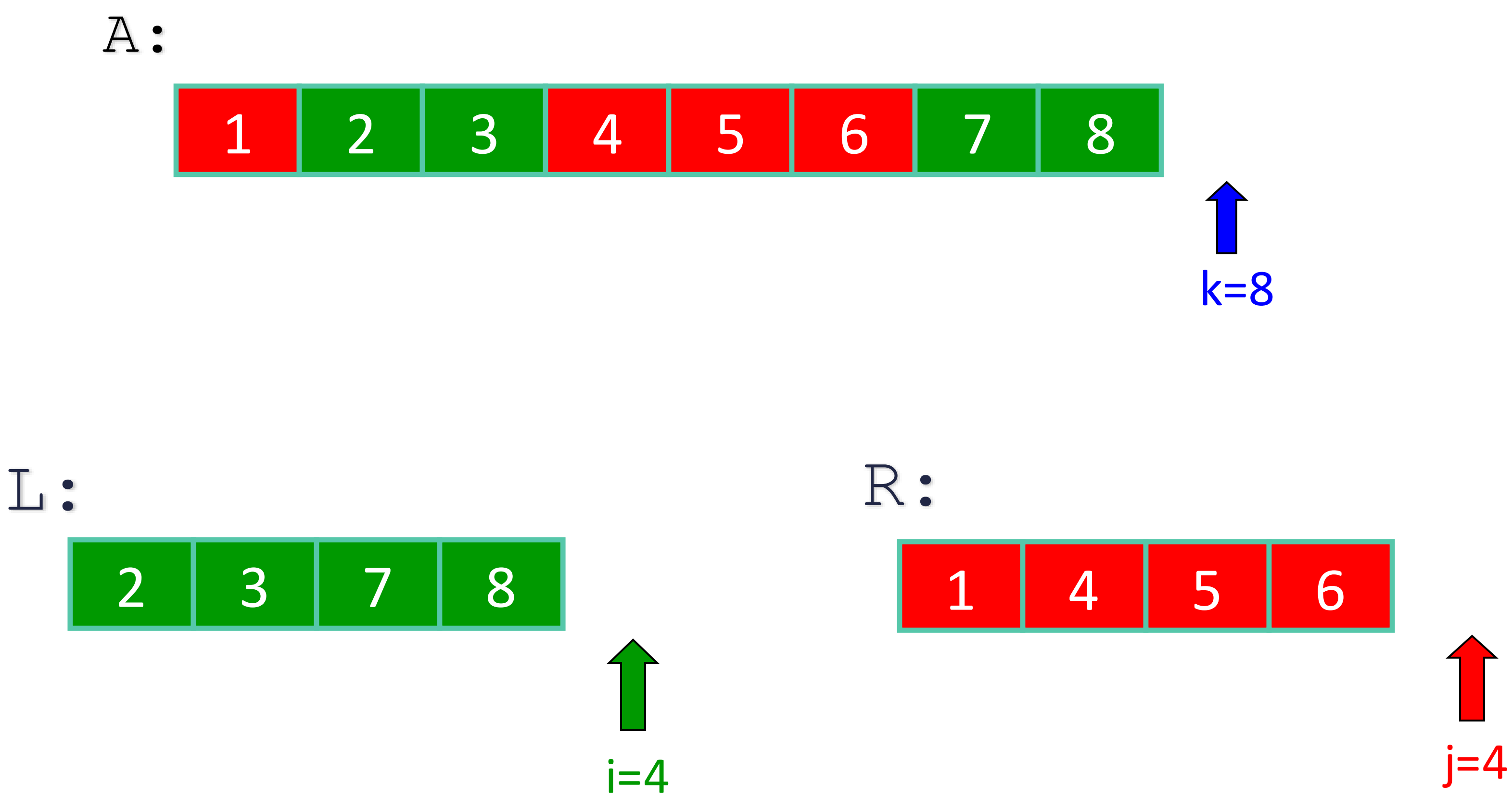
Merge-Sort: Merge



Merge-Sort: Merge

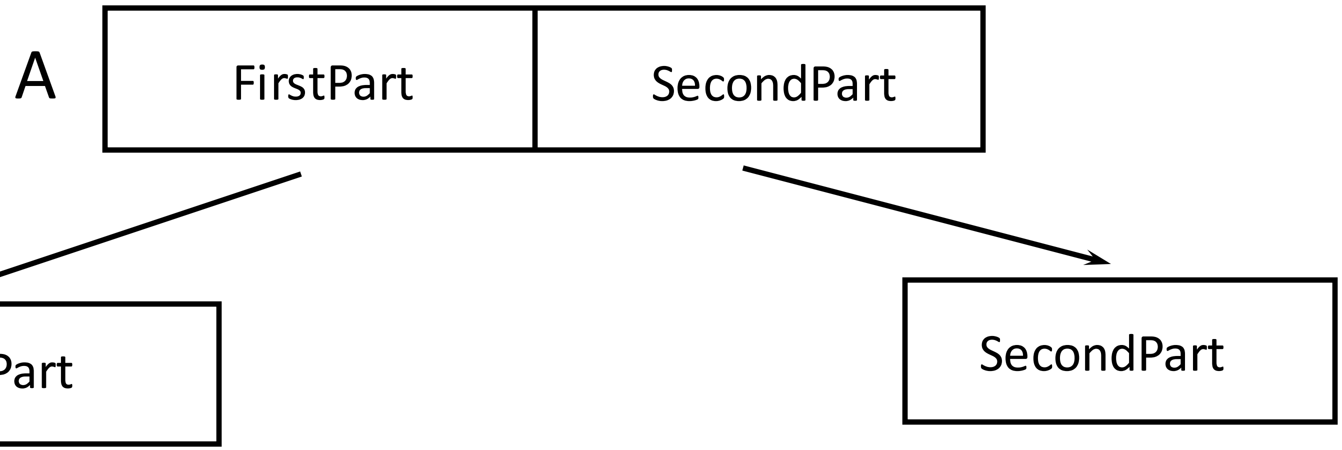


Merge-Sort: Merge



Merge Sort

Divide into two halves



Why split into two, why not k (k>2) ?

Live demo

- Comparison of merge sort implementations (k=2,10,100)

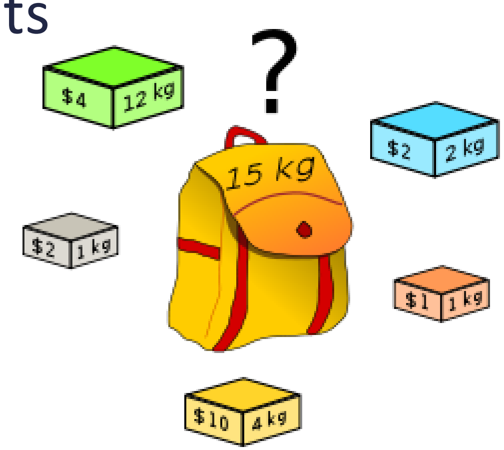
https://github.com/cjain7/DS221-Chirag-LiveDemos/tree/main/Lecture_10

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 unprocessed vertices

Knapsack Problem

- We are given a set of n items, where each item i is specified by a size s_i and a value v_i. We are also given a size bound S (the size of our knapsack).
- The goal is to find the subset of items of maximum total value such that sum of their sizes is at most S (i.e., they all fit into the knapsack).
  - Exponential time to try all possible subsets



Knapsack Problem

- 0-1 Knapsack:
  - *n* items
  - Must **leave or take** (i.e. 0-1) each item (e.g. bars of gold of different sizes and values)
- Fractional Knapsack:
  - *n* items
  - Can take **fractional part** of each item (e.g. gold dust)

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, s=[100,10,10], v=[20,15,15], S=105

Greedy Solution 2

- Select the one with the maximum value density v_i/s_i that fits into the knapsack
- *Still does not guarantee an optimal solution*
- E.g., n=3, s=[20,15,15], v=[40,25,25], S=30
- However, this greedy approach is optimal for fractional knapsack!
  - Proof?

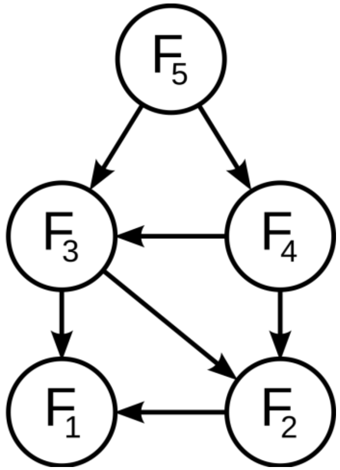
Greedy Solution 2 [fractional knapsack]

- **Greedy algorithm:**
  - Sort items by their density (v_i/s_i) in non-increasing order
  - Take as much possible from the highest-density item, then move to the next, until the size bound (capacity) S is reached
- **Proof of optimality:**
  - Without loss of generality, assume that the n items are indexed by their density, i.e., $v_1/s_1 \geq v_2/s_2 \geq \ldots \geq v_n/s_n$
  - Let $g_1, g_2, \ldots, g_n$ be the fraction of items picked by greedy
  - Let $f_1, f_2, \ldots, f_n$ be the fraction of items picked by another optimal solution **X** that is not greedy
  - Let i be the smallest index such that $f_i \neq g_i$
  - Observe that f_i must be less than g_i. And there must exist some k such that k>i and $f_k > g_k$
  - If we transfer a small amount of capacity δ from item k to item i in solution X, then then the value improves by $\delta \times (v_i/s_i - v_k/s_k)$, which is ≥ 0
  - Repeating this exchange process transforms any optimal solution into the greedy solution. The greedy solution is optimal for fractional knapsack.

Dynamic Programming (DP)

- Dynamic programming is generally used for optimization problems
  - Multiple solutions exist, need to find the best one
  - The problem should have two properties
    - **Optimal substructure**: Optimal solution can be constructed from optimal solutions to subproblems
    - **Overlapping subproblems**: Any recursive algorithm solving the problem should solve the same sub-problems over and over
- *This differs from Divide and Conquer, where subproblems generally need not overlap*

Fibonacci sequence: $F_i = F_{i-1} + F_{i-2}$



The subproblem graph for the Fibonacci sequence. The fact that it is not a tree indicates overlapping subproblems.

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 = 0, return 1. If n = 1 , return 1
  - Otherwise, compute fibonacci(n-1) and fibonacci(n-2)
  - Return the sum of these two numbers

- This is a *recursive* algorithm but it is computationally expensive
- Time complexity is exponential $O(2^n)$
  - 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=0, return 0. If n=1, return 1.
  - If n>1, then compute or retrieve following from a table:
    - fibonacci(n-1)
    - fibonacci(n-2)
  - Add these two values
  - Store the result in a table and return it
- In this algorithm, the table of previously computed values is being reused in future computations.
- Other examples: *Floyd–Warshall All-Pairs Shortest Path (APSP)* algorithm, *Towers of Hanoi*, …

Back to the 0-1 Knapsack Problem

- We are given a set of ***n*** items, where each item ***i*** is specified by a size s_i and a value v_i. We are also given a size bound ***S*** (the size of our knapsack).

- 0-1 Knapsack:
  - *n* items
  - Must **leave or take** each item

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

DP for 0-1 Knapsack

Define subproblems:
Let arr[i][Z] represent the **maximum value achievable** using the first i items and knapsack size limit Z

Recurrence:

$$\mathrm{arr}[i][Z] = \begin{cases} \mathrm{arr}[i-1][Z] & \text{if } si > Z \\ \max(\,\mathrm{arr}[i-1][Z],\, vi + \mathrm{arr}[i-1][Z-si]\,) & \text{otherwise} \end{cases}$$

Base case:
arr[i][0] = 0 for all i
Arr[0][Z] = 0 for all Z

Final solution
arr[n][S]

Pseudocode

Create two-dimensional array arr[0..n][0..S]

```
# Base case
for Z = 0 to S
    arr[0][Z] = 0
for i = 0 to n
    arr[i][0] = 0

# Fill DP table
for i = 1 to n
   for Z = 1 to S
      if sizes[i] > Z
          arr[i][Z] = arr[i-1][Z]
      else
          arr[i][Z] = max( arr[i-1][Z],  values[i] + arr[i-1][Z - sizes[i]] )

return arr[n][S]
```

Time and space complexity?

LCS problem (Try on your own)

- We are given two strings X and Y, find the **length** of their longest common substring.
- Example:
  - X = "abcdf"
  - Y = "zbcdfg”
  - Then, length of longest common substring = 4 ("bcdf")
- Hint:
  - <u>Define subproblem</u>: Compute length of the longest common substring **ending at** position i in X and position j in Y
  - Recursion? Base case? Final solution?

Brute force algorithm

- A brute force algorithm simply tries *all* possibilities until a satisfactory solution is found
- Such an algorithm can be:
  - Optimizing: 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
  - Satisficing: Stop as soon as a solution is found that is *good enough*
    - Example: Finding a traveling salesman path that is within 10% of optimal

Improving brute force algorithms

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

Randomized algorithms

- A randomized 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

k-means clustering

- **Steps**
  1. **Initialize**: Choose k initial centroids randomly.
  2. **Assignment step**: Assign each data point to the cluster with the nearest centroid (using a distance measure, e.g., Euclidean).
  3. **Update step**: Recompute each centroid as the mean of all points currently assigned to that cluster.
  4. **Repeat** steps 2–3 until:
     - Cluster assignments no longer change, or
     - A maximum number of iterations is reached.

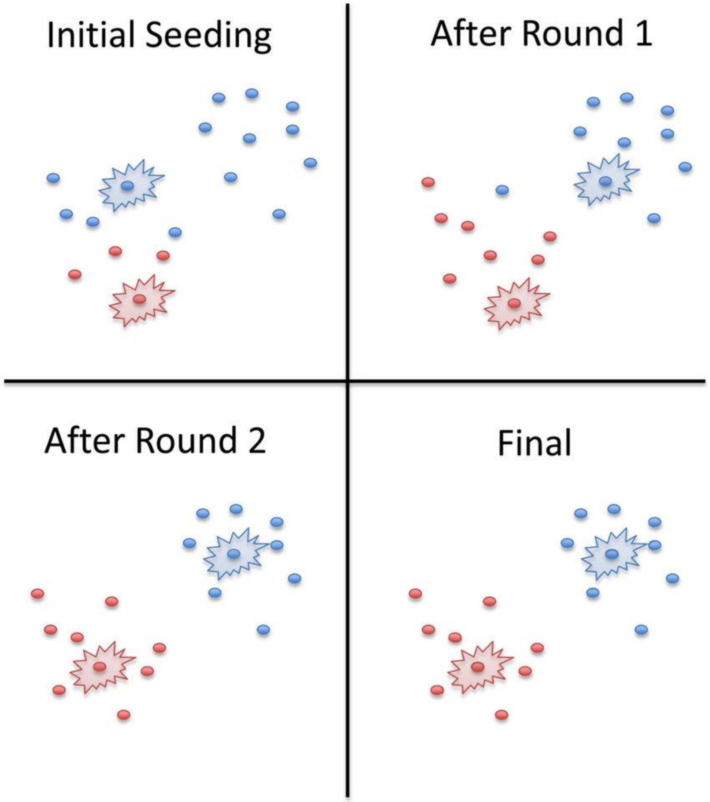


Image from https://devopedia.org/k-means-clustering

Reading

- Online resources on algorithms

- https://www.cs.cmu.edu/~avrim/451f09/lectures/lect1001.pdf