DS221 | 19 Sep – 19 Oct, 2017

Data Structures, Algorithms & Data Science Platforms

Yogesh Simmhan

simmhan@cds.iisc.ac.in

L5: Algorithm Types

Graph ADT, Algorithms

Some slides courtesy:
Venkatesh Babu & Sathish Vadhiyar, CDS, IISc

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*

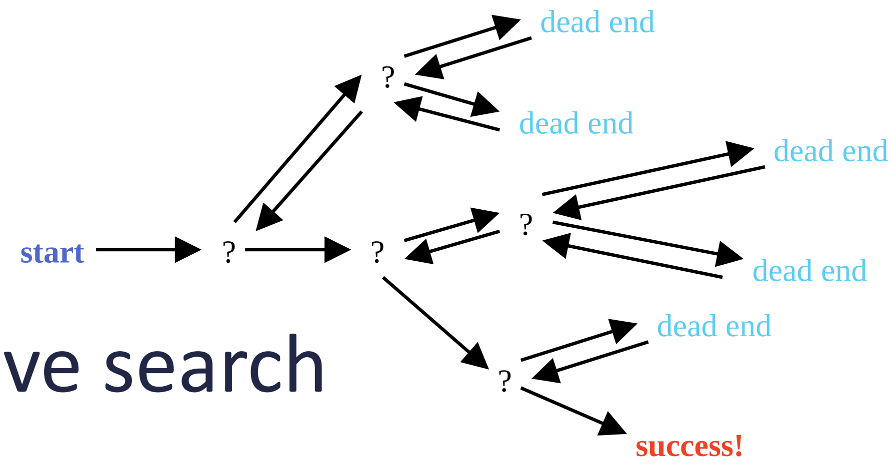
A short list of categories

- 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

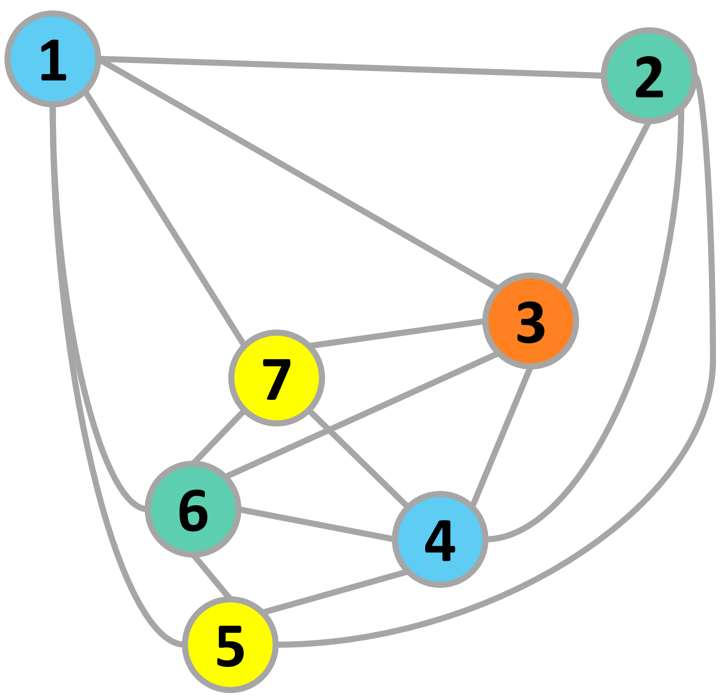


- Uses a depth-first recursive search over solution space
  - Test to see if a solution has been found, and if so, returns it; otherwise
  - For each choice that can be made at this point,
    - Make that choice
    - Recurse
    - If the recursion returns a solution, return it
  - If no choices remain,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

- *Any seen so far?*
  - DFS traversal

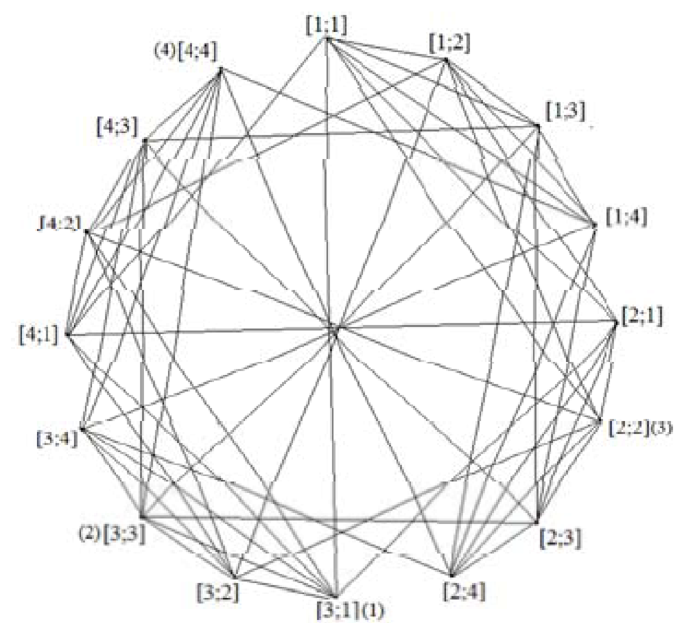
Sample backtracking algo.

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



4x4 Sudoku

[1;1]	[1;2]	[1;3]	[1;4]
[2;1]	[2;2] 3	[2;3]	[2;4]
[3;1] 1	[3;2]	[3;3] 2	[3;4]
[4;1]	[4;2]	[4;3]	[4;4] 4

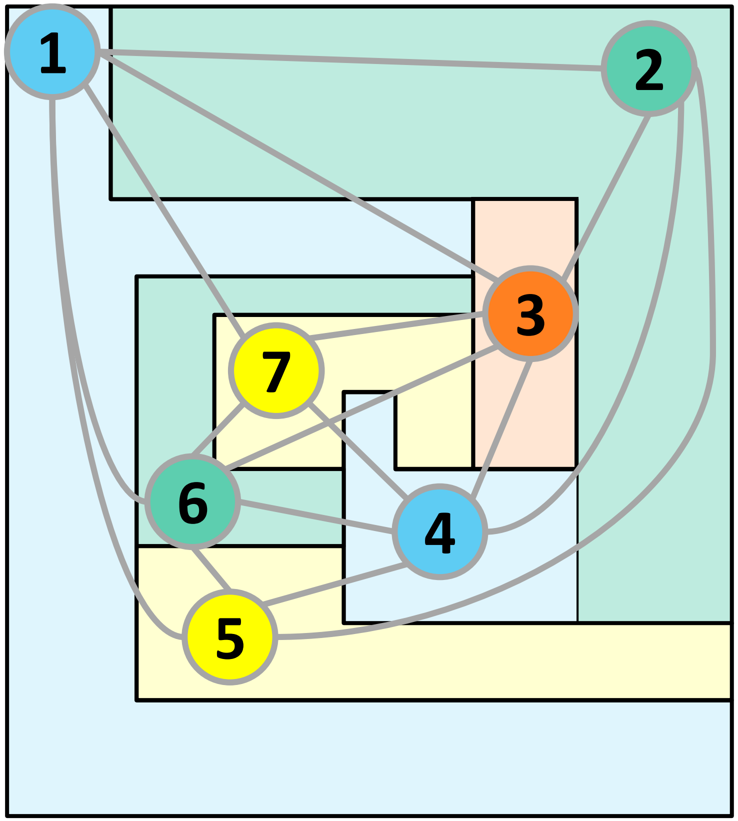


The above mentioned graph has 16 vertices and 56 edges.

Sample backtracking algo.

The ***Four Color Theorem*** *states that any map on a plane can be colored with no more than four colors, so that no two countries with a common border are the same color*

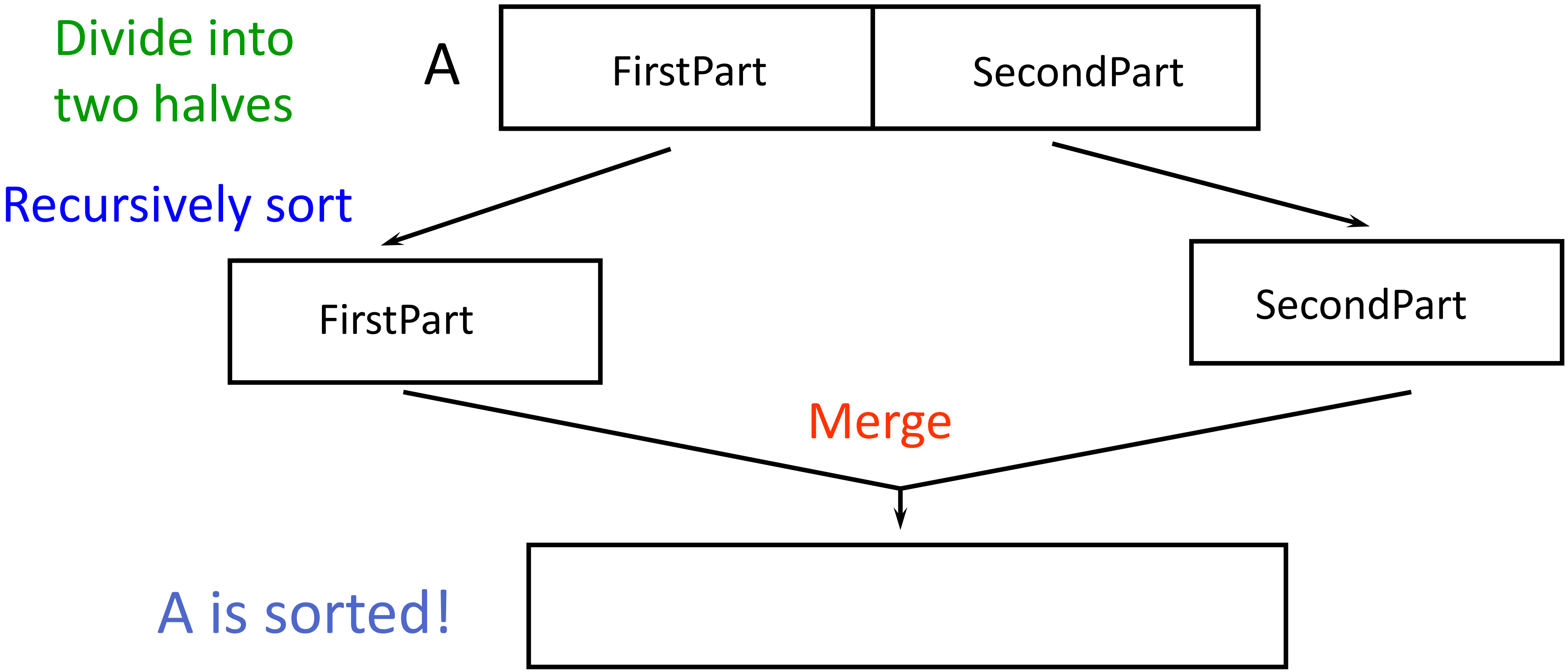
```
boolean explore(int ctry, int col){
 if (ctry >= map.size) return true;
 if (okToColor(ctry, col)) {
 map[ctry] = col;
   for (int c=RED; c<=BLUE; c++){
     if (explore(ctry+1, c))
       return true;
   }
 } else
   return false;
}
```



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*
- *E.g. Merge Sort, Quick Sort*

Merge Sort: Idea

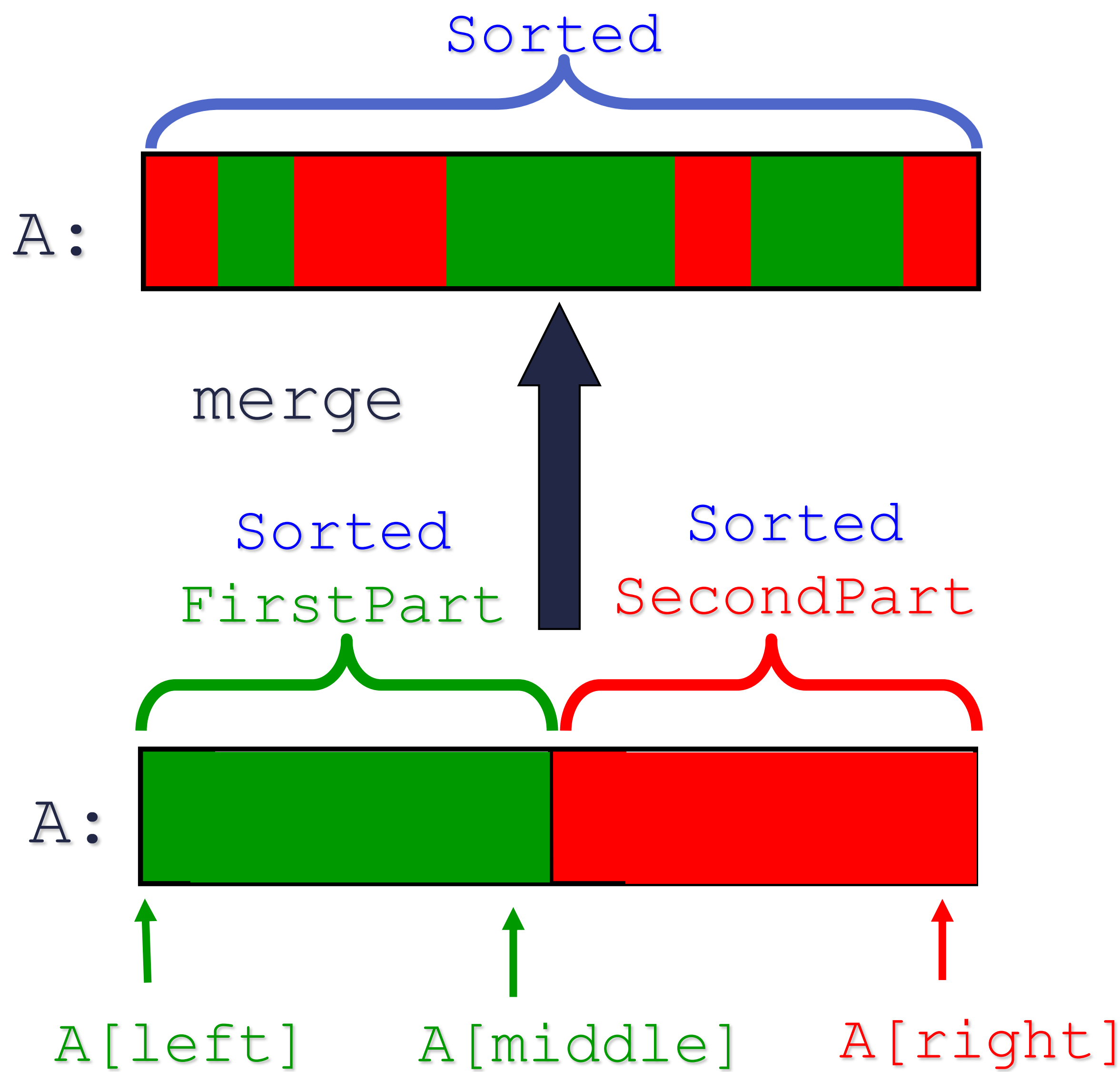


Merge Sort: Algorithm

```
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-Sort: Merge



Merge-Sort: Merge

A:

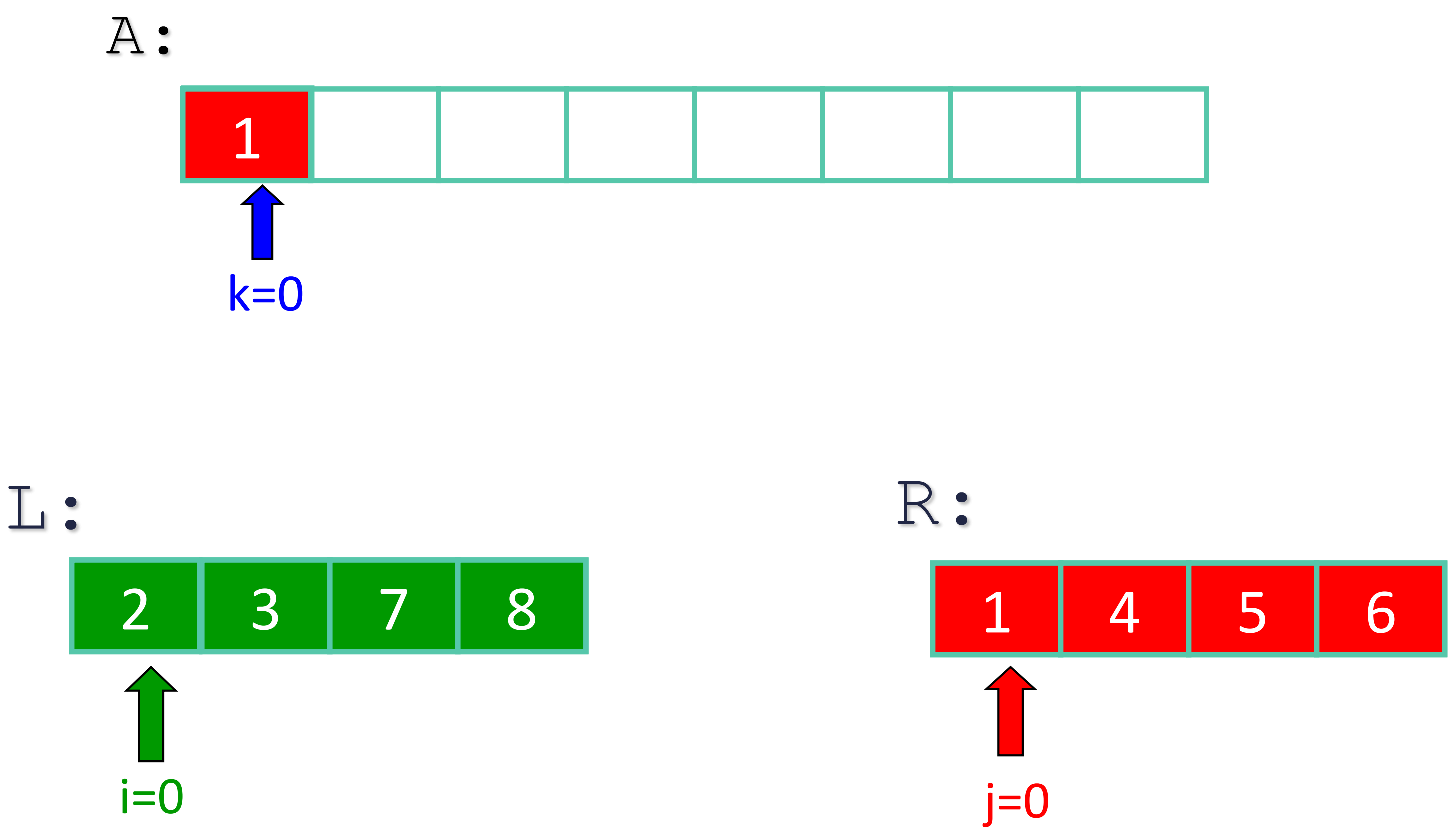
2	3	7	8	1	4	5	6

L:

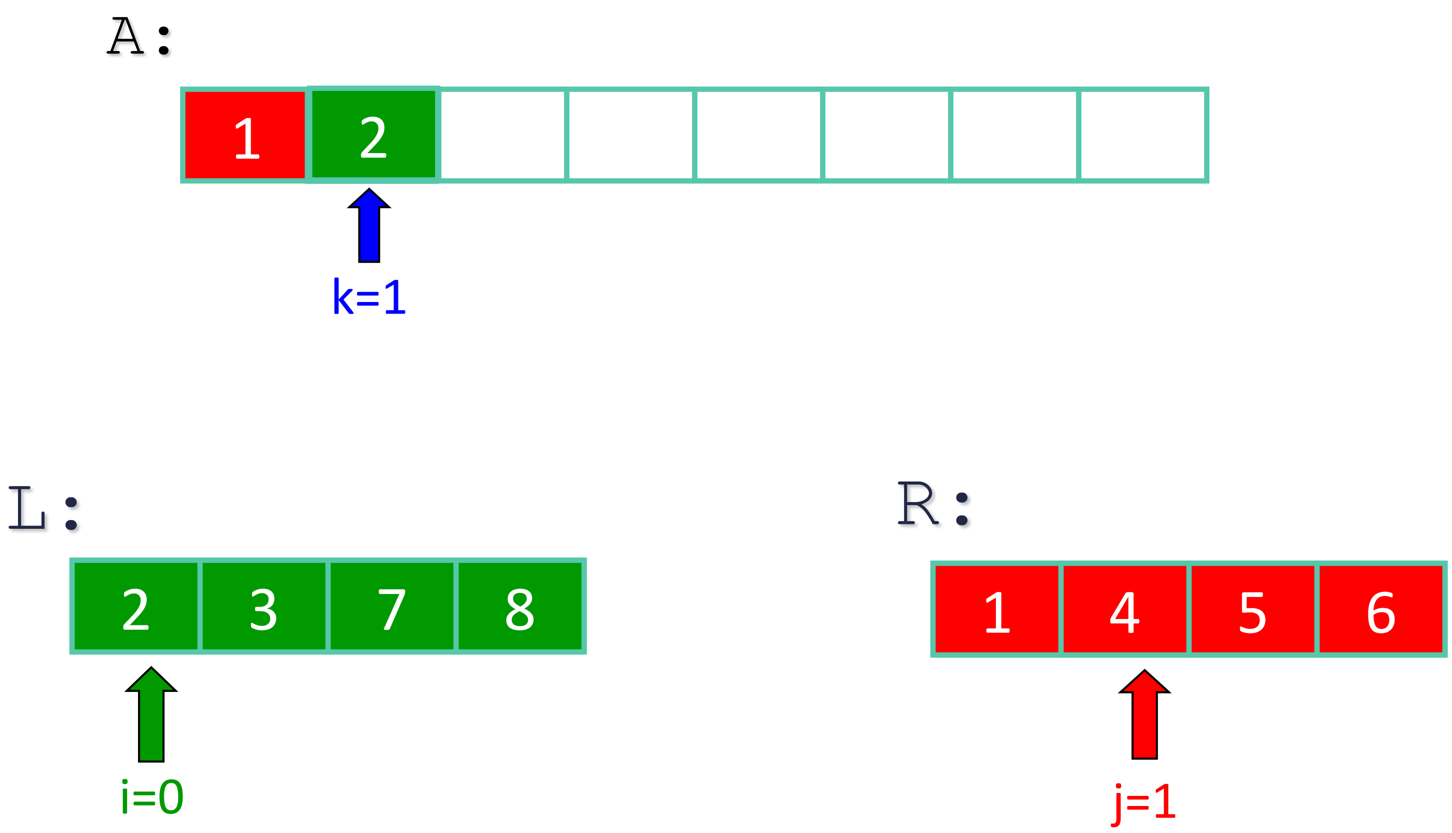
R:

Temporary Arrays

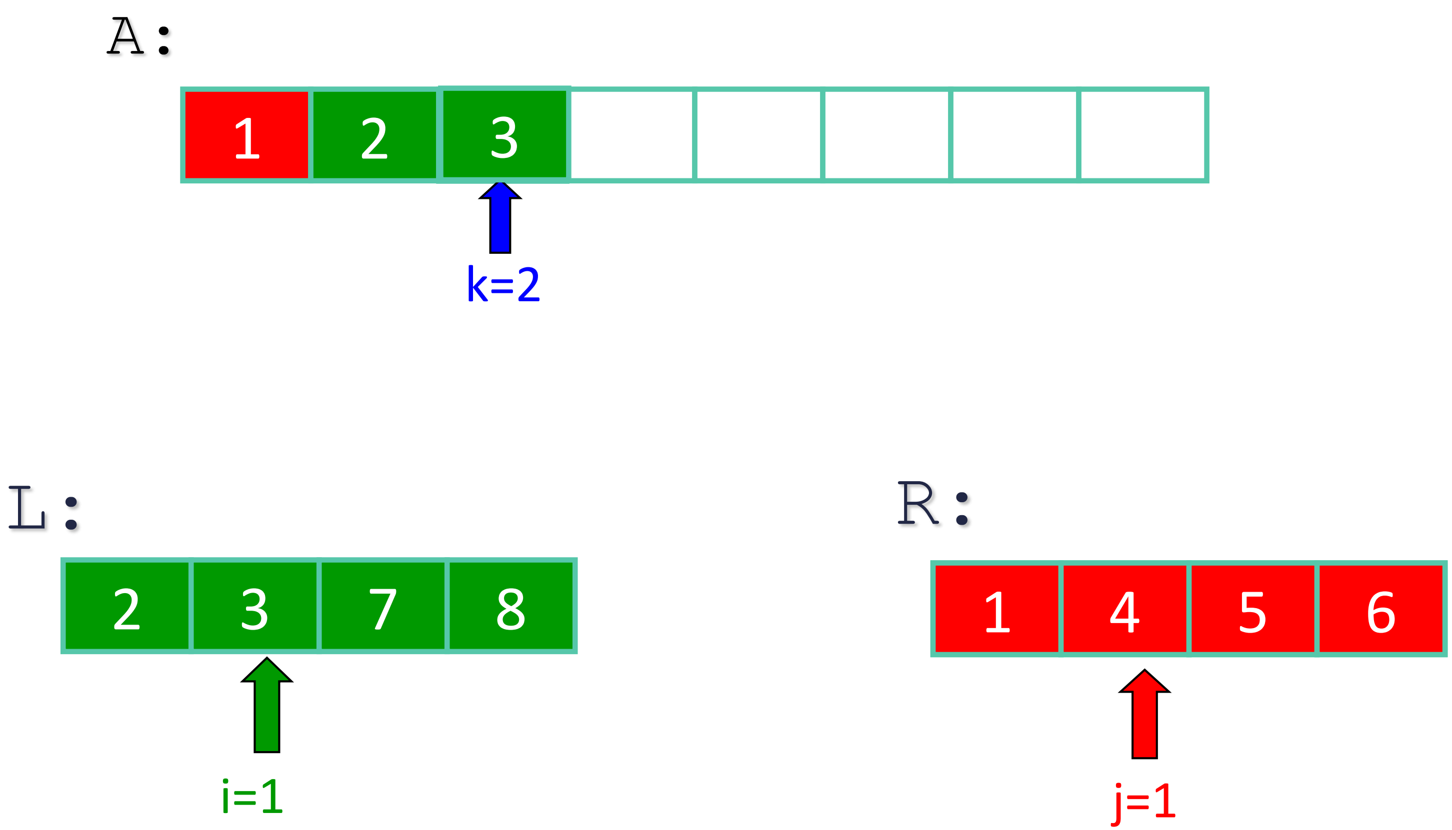
Merge-Sort: Merge



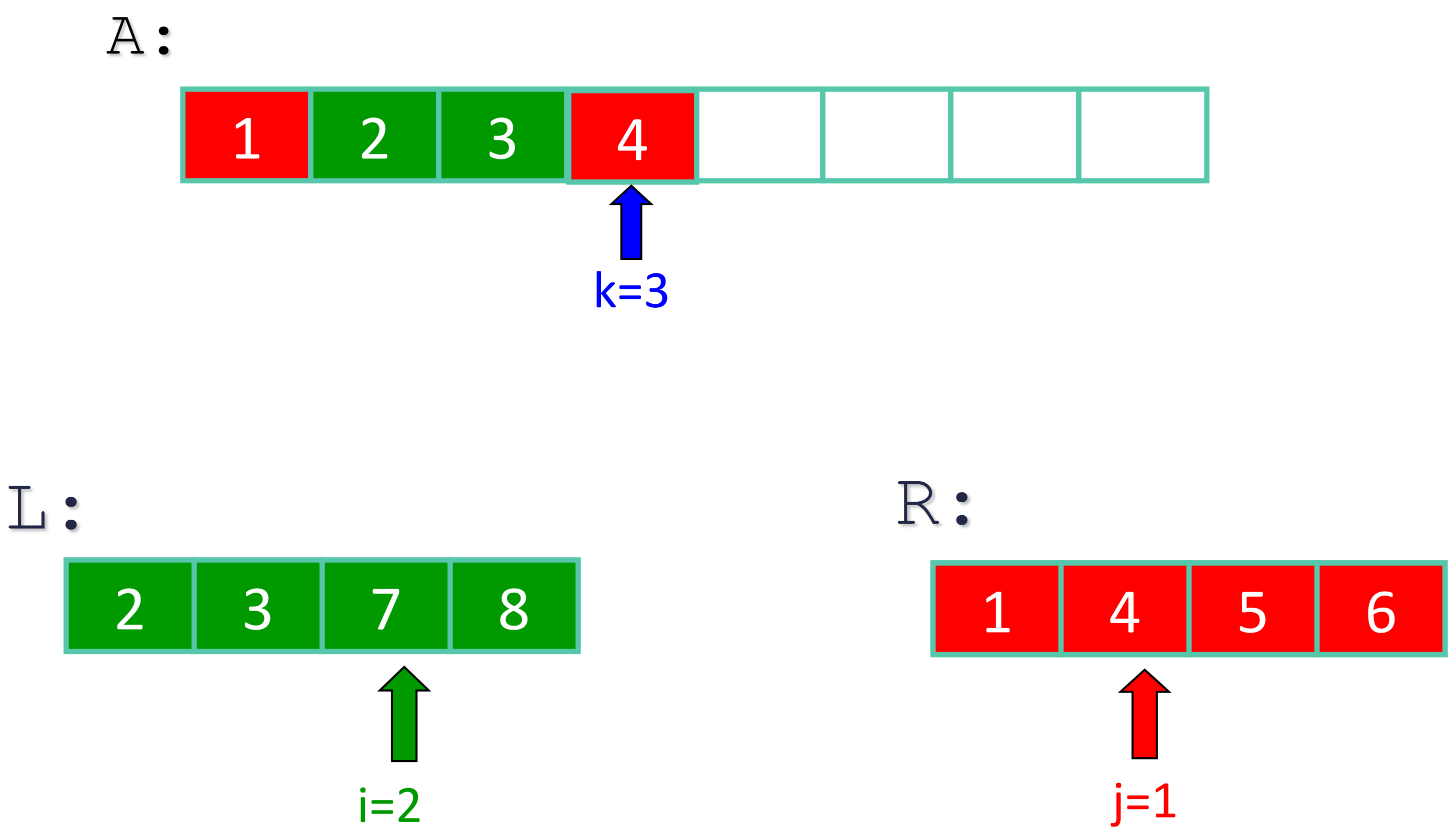
Merge-Sort: Merge



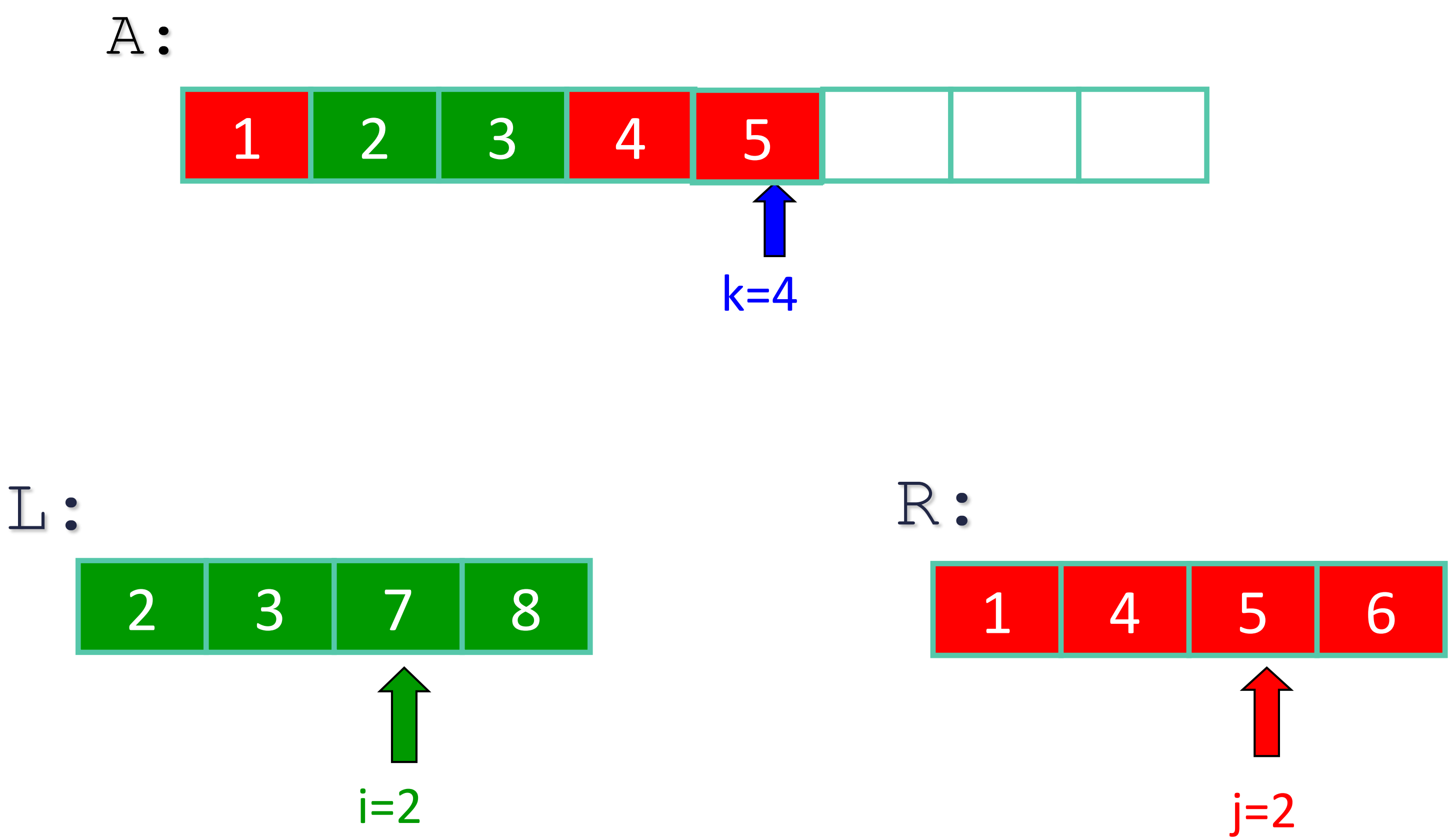
Merge-Sort: Merge



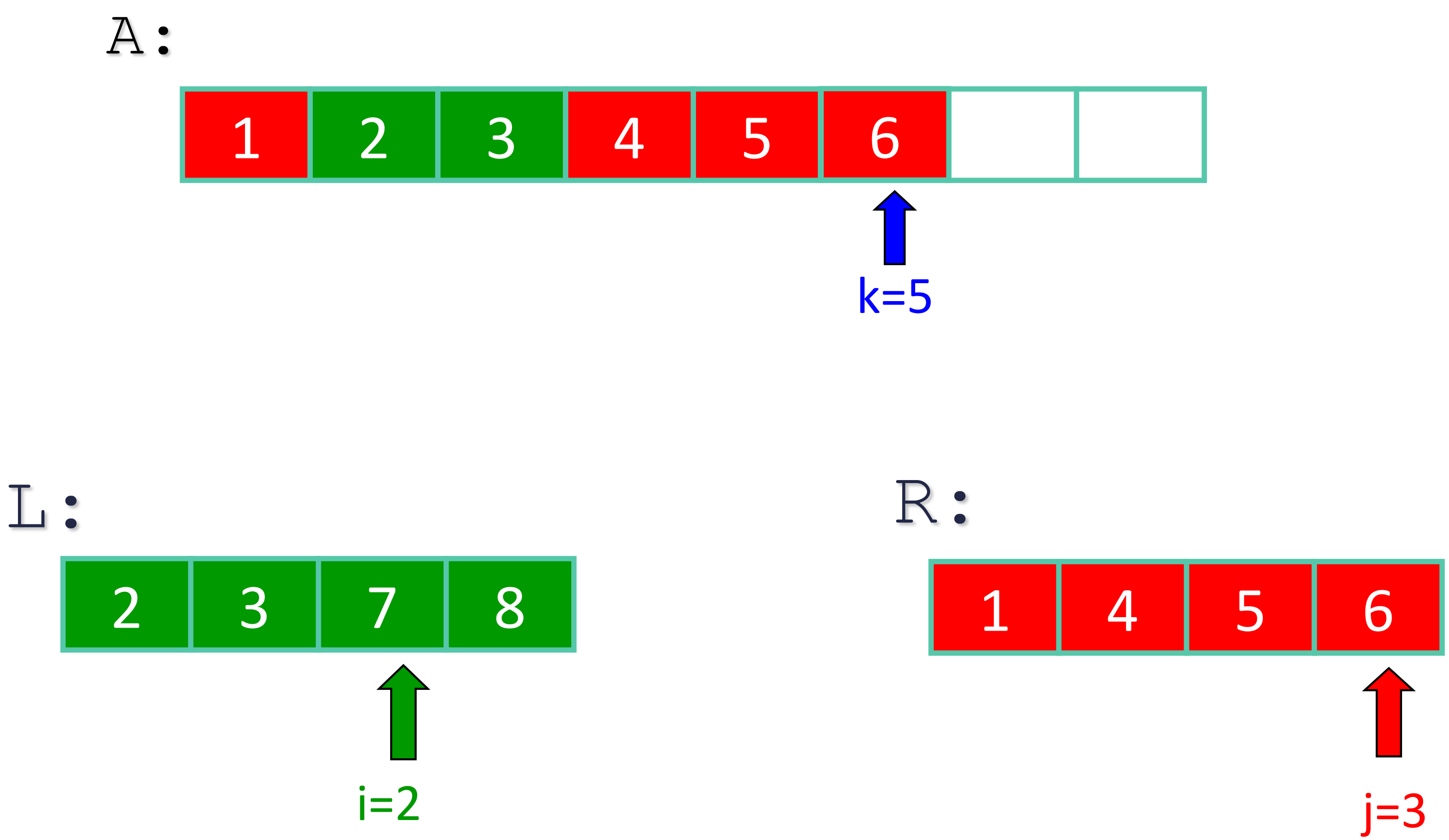
Merge-Sort: Merge



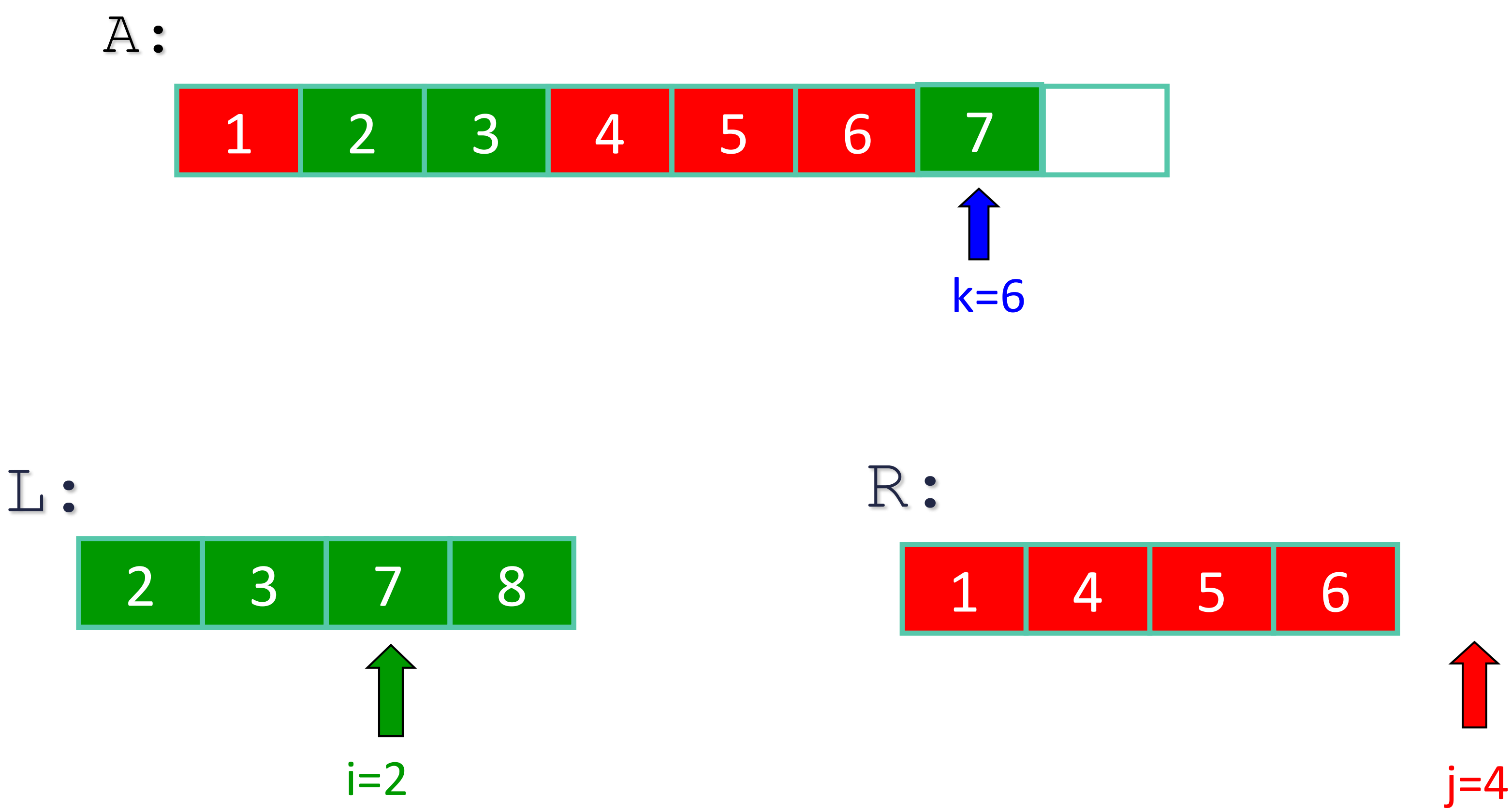
Merge-Sort: Merge



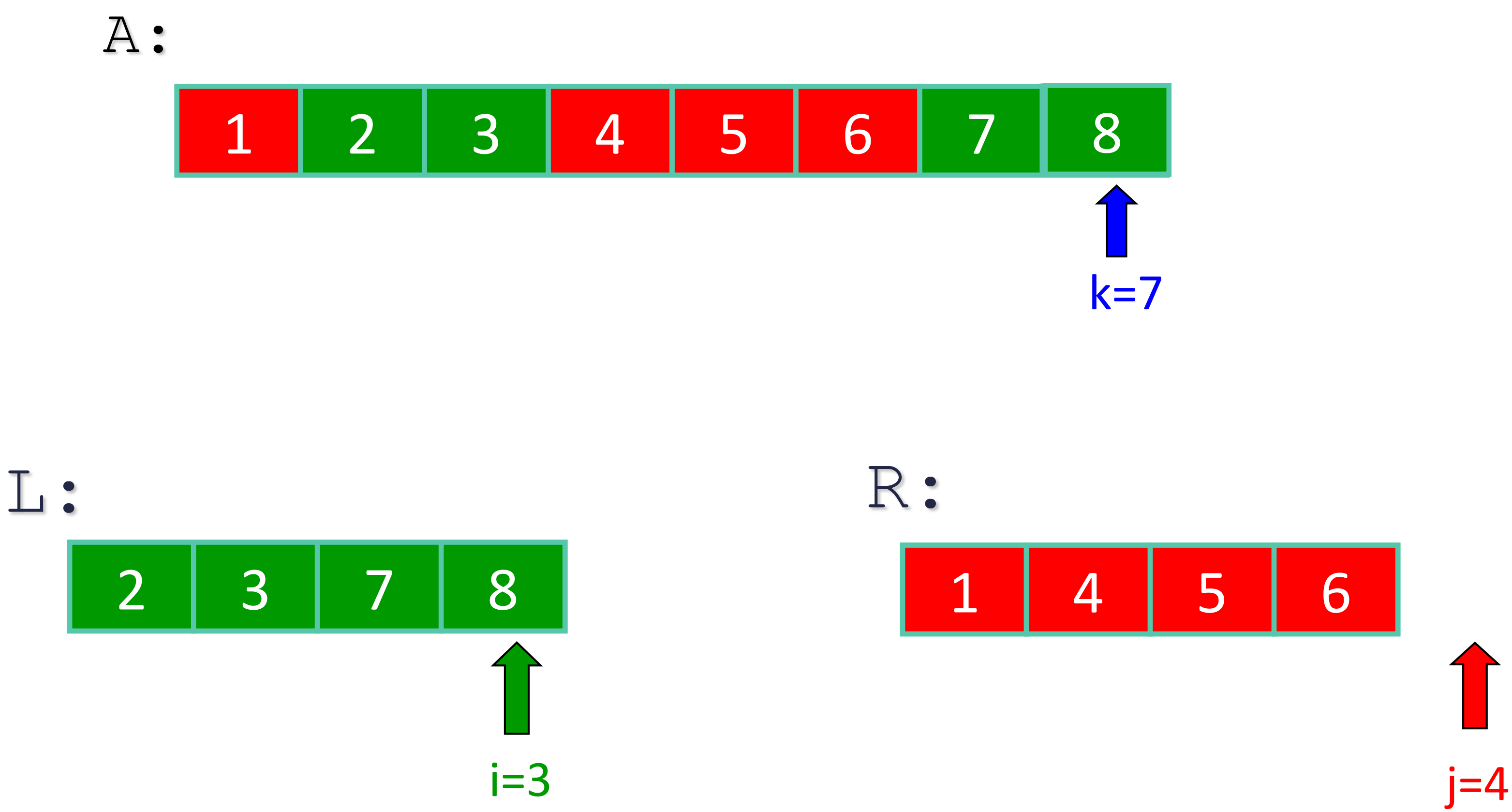
Merge-Sort: Merge



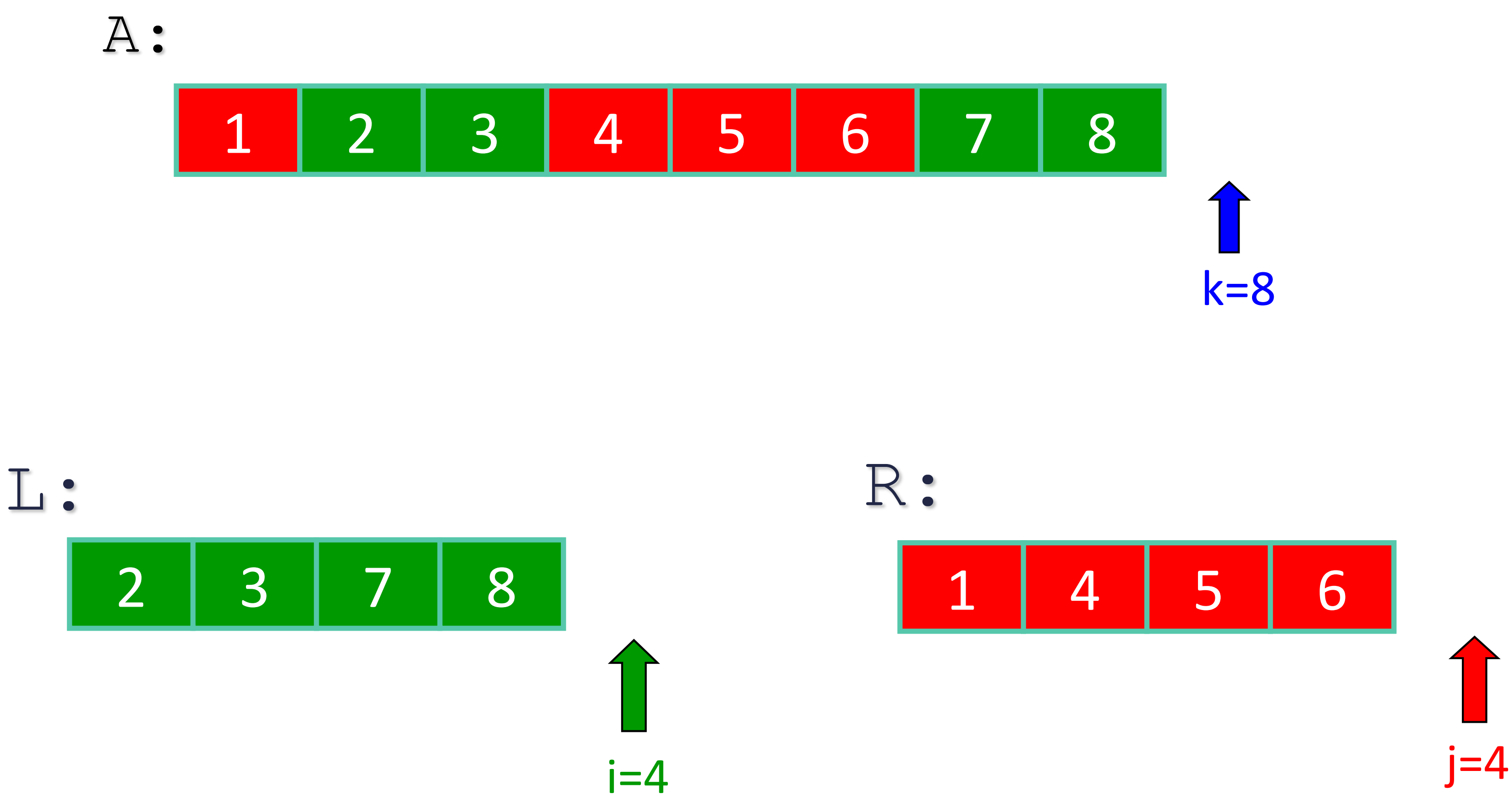
Merge-Sort: Merge



Merge-Sort: Merge



Merge-Sort: Merge



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

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
- This is also an *expensive* algorithm
  - It requires O(fibonacci(n)) time
  - This is equivalent to 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,
  - Compute, *or look up in a table*, fibonacci(n-1) and 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*, …

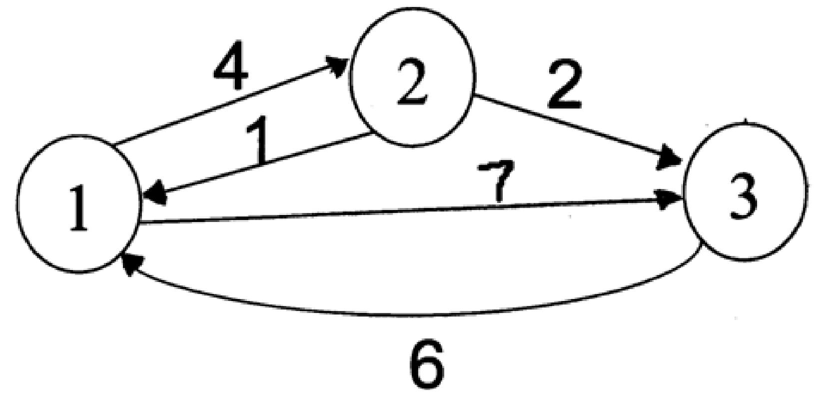
http://www.cs.ucf.edu/~sarahb/COP3503/Lectures/DynProg_FloydWarshall.ppt

Floyd Warshall APSP

- Shortest distances between all pairs of ***v*** vertices
  - O(v³) complexity. *Similar to Djikstra's on each vertex!*
- Test if the current shortest path from **i** to **j** is improved by a path from **i** to **k** and then **k** to **j**

$$D^{(0)} = \begin{bmatrix} 0 & 4 & 7 \\ 1 & 0 & 2 \\ 6 & \infty & 0 \end{bmatrix}$$

$$D^{(2)} = \begin{bmatrix} 0 & 4 & 6 \\ 1 & 0 & 2 \\ 6 & 10 & 0 \end{bmatrix}$$



$$D^{(1)} = \begin{bmatrix} 0 & 4 & 7 \\ 1 & 0 & 2 \\ 6 & 10 & 0 \end{bmatrix}$$

$$D^{(3)} = \begin{bmatrix} 0 & 4 & 6 \\ 1 & 0 & 2 \\ 6 & 10 & 0 \end{bmatrix}$$

Consider thru' Vertex 1:

D(3,2) = D(3,1) + D(1,2)

Consider thru' Vertex 2:

D(1,3) = D(1,2) + D(2,3)

Consider thru' Vertex 3:

Nothing changes.

Floyd Warshall APSP

- Looking at this example, we can come up with the following algorithm:
  - Let D store the matrix with the initial edge-weight information initially, infinity for non-existent edges.
  - Update D with the calculated shortest paths

```
For k=1 to n {
  For i=1 to n {
     For j=1 to n
        D[i,j] = min(D[i,j], D[i,k]+D[k,j])
  }
}
```

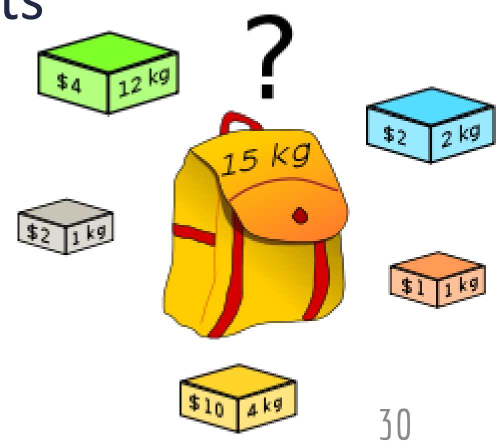
  - The final D matrix will store all the shortest paths.

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 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 (they all fit into the knapsack).
  - Exponential time to try all possible subsets
  - O(n.S) using DP



© Keenan Pepper

Knapsack Problem

- 0-1 Knapsack:
  - *n* items (can be the same or different)
  - Have **only one** of each
  - Must **leave or take** (i.e. 0-1) each item (e.g. bars of gold)
  - DP works, greedy does not

- Fractional Knapsack:
  - *n* items (can be the same or different)
  - Can take **fractional part** of each item (e.g. gold dust)
  - Greedy works and DP algorithms work

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 minimum size that fits into the knapsack
- *Also, does not guarantee optimal solution*
- E.g., n=2, s=[10,20], v=[5,100], c=25

Greedy Solution 3

- Select the one with the maximum value density v_i/s_i that fits into the knapsack
- E.g., n=3, s=[20,15,15], v=[40,25,25], c=30
- Greedy works…if fractional items possible!

DP for 0-1 Knapsack

```
// Recursive algorithm: either we use the last element or we don't.
Value(n,S)    // S = space left, n = # items still to choose from
{
   if (n == 0) return 0;
   if (arr[n][S] != unknown) return arr[n][S];  // <- added this
   if (s_n > S) result = Value(n-1,S);
   else result = max{v_n + Value(n-1, S-s_n), Value(n-1, S)};
   arr[n][S] = result;                           // <- and this
   return result;
}
```

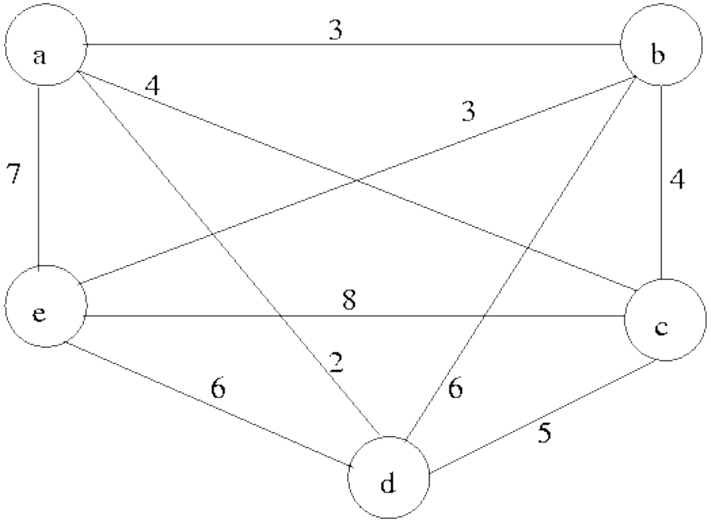
Branch & Bound algorithms

- Branch and bound algorithms are generally used for optimization problems. *Similar to backtracking.*
  - As the algorithm progresses, a tree of subproblems is formed
  - The original problem is considered the “root problem”
  - A method is used to construct an upper and lower bound for a given problem
  - At each node, apply the bounding methods
    - If the bounds match, it is deemed a feasible solution to that particular subproblem
    - If bounds do *not* match, partition the problem represented by that node, and make the two subproblems into children nodes
  - Continue, using the best known feasible solution to trim sections of the tree, until all nodes have been solved or trimmed

Example branch and bound algorithm

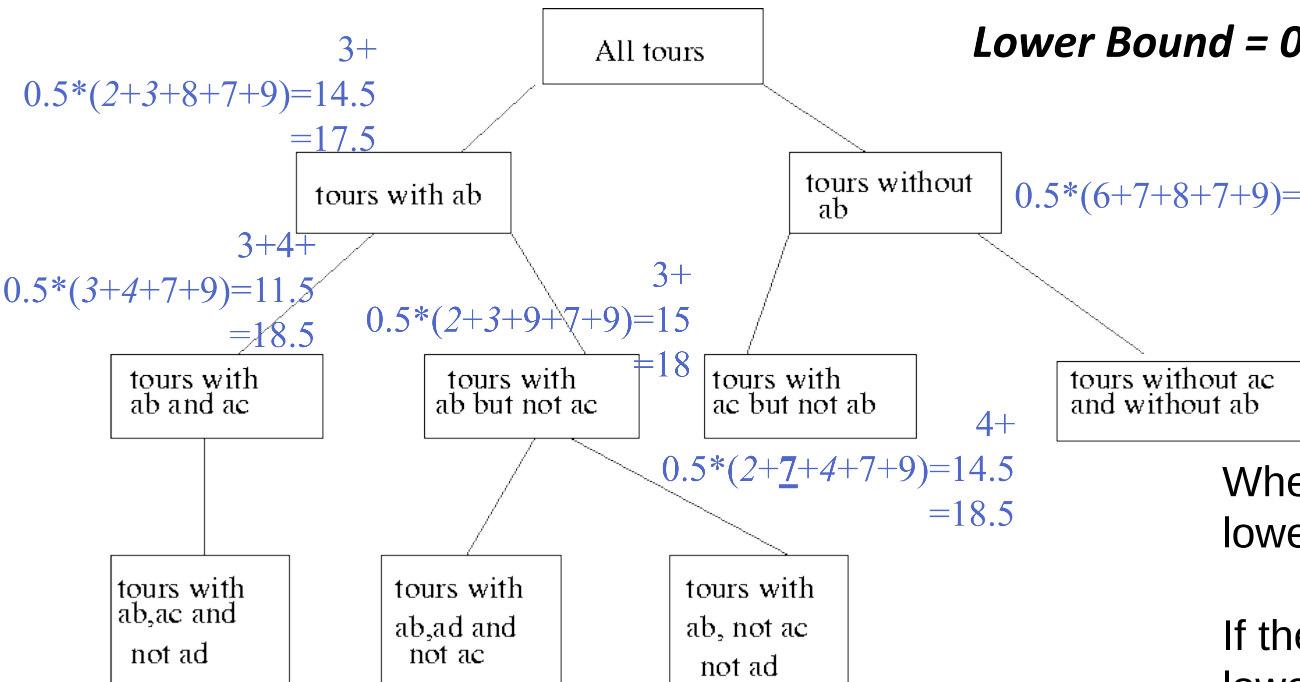
- *"Suppose it is required to minimize an objective function. Suppose that we have a method for getting a lower bound on the cost of any solution among those in the set of solutions represented by some subset. If the best solution found so far costs less than the lower bound for this subset, we need not explore this subset at all."*
- **Traveling salesman problem**: A salesman has to visit each of *n* cities once each and return to the original city, while minimize total distance traveled
  - Split into two subproblems, whether to take an out edge from a vertex or not.
  - If current best solution smaller than the lower bound of a subset, do not explore.
  - Lower bound given by *0.5*(sum of tours on two edges, for all vertices)*

http://lcm.csa.iisc.ernet.in/dsa/node187.html



Node	Least cost edges	Total cost
a	(a, d), (a, b)	2+3=5
b	(a, b), (b, e)	3+3=6
c	(c, b), (c, a)	4+4=8
d	(d, a), (d, c)	2+5=7
e	(e, b), (e, d)	3+6=9

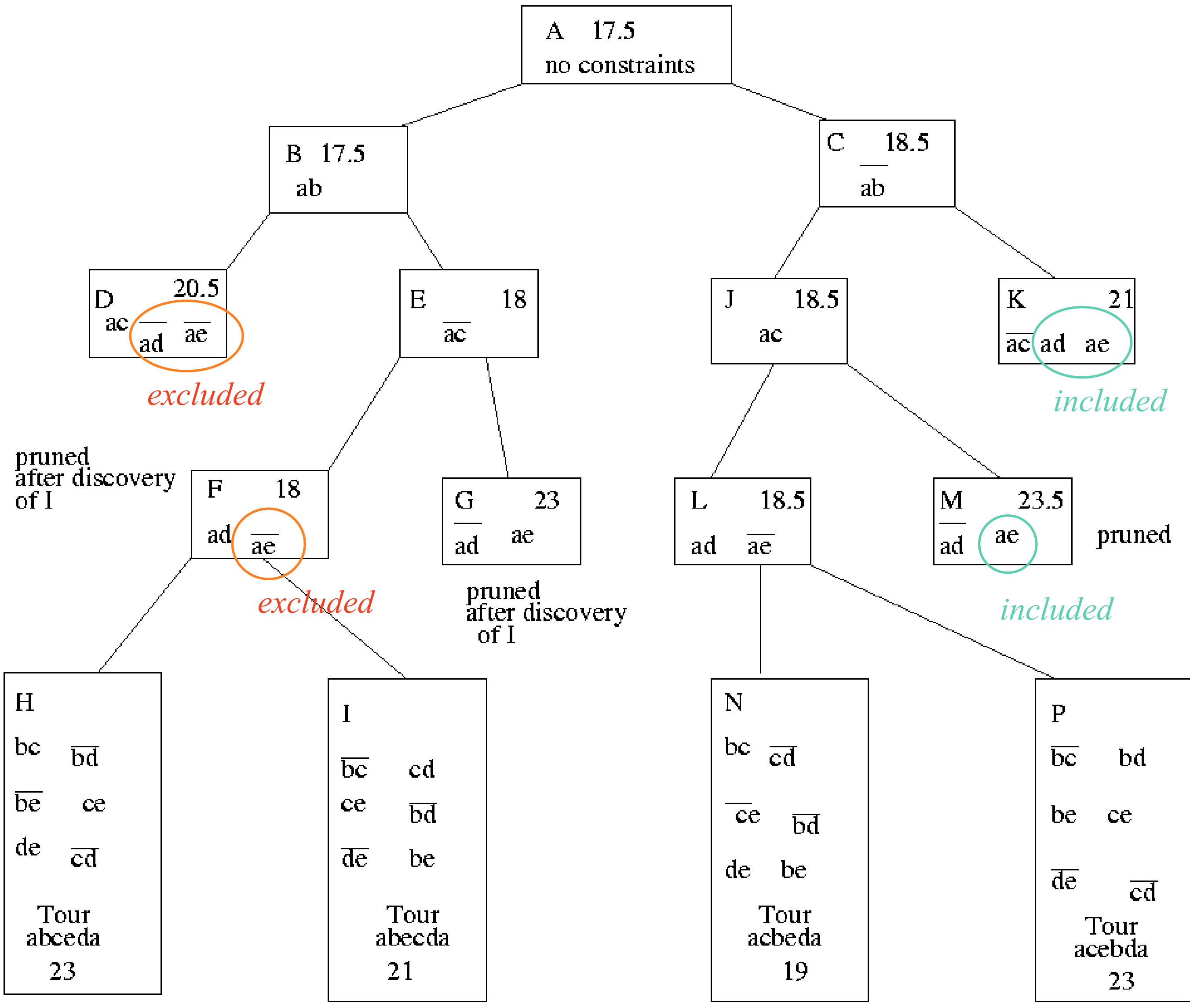
Lower Bound = 0.5 * (5 + 6 + 8 + 7 + 9) = 17.5



When we branch, we compute lower bounds for both children.

If the lower bound for a child is >= lowest cost found so far, we prune that child.

http://lcm.csa.iisc.ernet.in/dsa/node187.html



If excluding (x, y) makes it impossible for x or y to have two adjacent edges in the tour, include (x, y).

If including (x, y) would cause x or y to have more than two edges adjacent in the tour, or complete a non-tour cycle with edges already included, exclude (x, y).

http://lcm.csa.iisc.ernet.in/dsa/node187.html

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
- *E.g. k-means clustering*

Reading

- Online resources on algorithm types
- https://www.cs.cmu.edu/~avrim/451f09/lectures/lect1001.pdf