CSE 2202
Design and Analysis of Algorithms – I

Lecture 9

Algorithm Types

Divide and Conquer

ALGORITHM STRATEGIES

General Concepts

- Algorithm strategy
  - Approach to solving a problem
  - May combine several approaches
- Algorithm structure
  - Iterative $\Rightarrow$ execute action in loop
  - Recursive $\Rightarrow$ reapply action to subproblem(s)
- Problem type
  - Decision $\Rightarrow$ find Yes/No answer
  - Satisfying $\Rightarrow$ find any satisfactory solution
  - Optimization $\Rightarrow$ find best solutions (vs. cost metric)

Some Algorithm Strategies

- Divide and conquer algorithms
- Dynamic programming algorithms
- Greedy algorithms
- Backtracking algorithms
- Branch and bound algorithms
- Heuristic algorithms

Divide and Conquer

- Based on dividing problem into subproblems
- Approach
  1. Divide problem into smaller subproblems
     - Subproblems must be of same type
     - Subproblems do not need to overlap
  2. Solve each subproblem recursively
  3. Combine solutions to solve original problem
- Usually contains two or more recursive calls

Divide and Conquer – Examples

- Binary Search
- Quicksort
  - Partition array into two parts around pivot
  - Recursively quicksort each part of array
  - Concatenate solutions
- Mergesort
  - Partition array into two parts
  - Recursively mergesort each half
  - Merge two sorted arrays into single sorted array
- Counting Inversion

Dynamic Programming Algorithm

- Based on remembering past results
- Approach
  1. Divide problem into smaller subproblems
     - Subproblems must be of same type
     - Subproblems must overlap
  2. Solve each subproblem recursively
     - May simply look up solution
  3. Combine solutions into to solve original problem
  4. Store solution to problem
- Generally applied to optimization problems

Fibonacci Algorithm

- Fibonacci numbers
  - fibonacci(0) = 1
  - fibonacci(1) = 1
  - fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
- Recursive algorithm to calculate fibonacci(n)
  - If n is 0 or 1, return 1
  - Else compute fibonacci(n-1) and fibonacci(n-2)
  - Return their sum
- Simple algorithm $\Rightarrow$ exponential time $O(2^n)$

Dynamic Programming – Example

- Dynamic programming version of fibonacci(n)
  - If n is 0 or 1, return 1
  - Else solve fibonacci(n-1) and fibonacci(n-2)
    - Look up value if previously computed
    - Else recursively compute
  - Find their sum and store
  - Return result
- Dynamic programming algorithm $\Rightarrow$ O(n) time
  - Since solving fibonacci(n-2) is just looking up value

Dynamic Programming - Example

- 0-1 Knapsack
- Longest Common Subsequence
- Longest Increasing Sequence
- Sum of Subset
- Warshall’s All pairs shortest path
- Bellman Ford’s Single Source Shortest Path
- Matrix Chain Multiplication

Greedy Algorithm

- Based on trying best current (local) choice
- Approach
  - At each step of algorithm choose best local solution
- Avoid backtracking, exponential time $O(2^n)$
- Hope local optimum lead to global optimum

Greedy Algorithm – Example

Kruskal's Minimal Spanning Tree Algorithm

```
sort edges by weight (from least to most)
tree = $\varnothing$
for each edge (X,Y) $\boxed{\text{in order}}$
    if it does not create a cycle
        add (X,Y) to tree
        stop when tree has N–1 edges
```

Picks best local solution at each step

Greedy Algorithm - Example

- Dijkstra’s Single Source Shortest Path
- Minimum Spanning Tree – Prim & Kruskal
- Fractional Knapsack Problem
- Huffman Coding

Backtracking Algorithm

- Based on depth-first recursive search
- Approach
  1. Tests whether solution has been found
  2. If found solution, return it
  3. Else for each choice that can be made
     a) Make that choice
     b) Recur
     c) If recursion returns a solution, return it
  4. If no choices remain, return failure

Backtracking Algorithm – Example

- Find path through maze
  - Start at beginning of maze
  - If at exit, return true
  - Else for each step from current location
    - Recursively find path
    - Return with first successful step
    - Return false if all steps fail

Backtracking Algorithm – Example

- Color a map with no more than four colors
  - If all countries have been colored return success
  - Else for each color c of four colors and country n
    - If country n is not adjacent to a country that has been colored c
      - Color country n with color c
      - Recursively color country n+1
      - If successful, return success
  - Return failure

Backtracking - Example

- 8 Queen Problem
- Graph Coloring
- Sum of Subset
- Hamiltonian Cycle
- Travelling Salesman Problem (TSP)
- Permutation & Combination Generation

Branch and Bound Algorithm

- Based on limiting search using current solution
- Approach
  - Track best current solution found
  - Eliminate partial solutions that can not improve upon best current solution
  - Reduces amount of backtracking
- Not guaranteed to avoid exponential time $O(2^n)$

Branch and Bound – Example

- Branch and bound algorithm for TSP
  - Find possible paths using recursive backtracking
  - Track cost of best current solution found
  - Stop searching path if cost > best current solution
  - Return lowest cost path
- If good solution found early, can reduce search
- May still require exponential time $O(2^n)$

Heuristic Algorithm

- Based on trying to guide search for solution
- Heuristic $\Rightarrow$ “rule of thumb”
- Approach
  - Generate and evaluate possible solutions
    - Using “rule of thumb”
    - Stop if satisfactory solution is found
- Can reduce complexity
- Not guaranteed to yield best solution

Heuristic Algorithm – Example

- Heuristic algorithm for TSP
  - Find possible paths using recursive backtracking
    - Search 2 lowest cost edges at each node first
  - Calculate cost of each path
  - Return lowest cost path from first 100 solutions
- Not guaranteed to find best solution
- Heuristics used frequently in real applications

DIVIDE & CONQUER

Divide and Conquer Algorithms

- Example
  - Binary Search
  - Merge Sort
  - Quick Sort
  - Counting
  - Closest Pair of Points

Divide and Conquer

Divide the problem into a number of subproblems

– There must be base case (to stop recursion).

Conquer (solve) each subproblem **recursively**

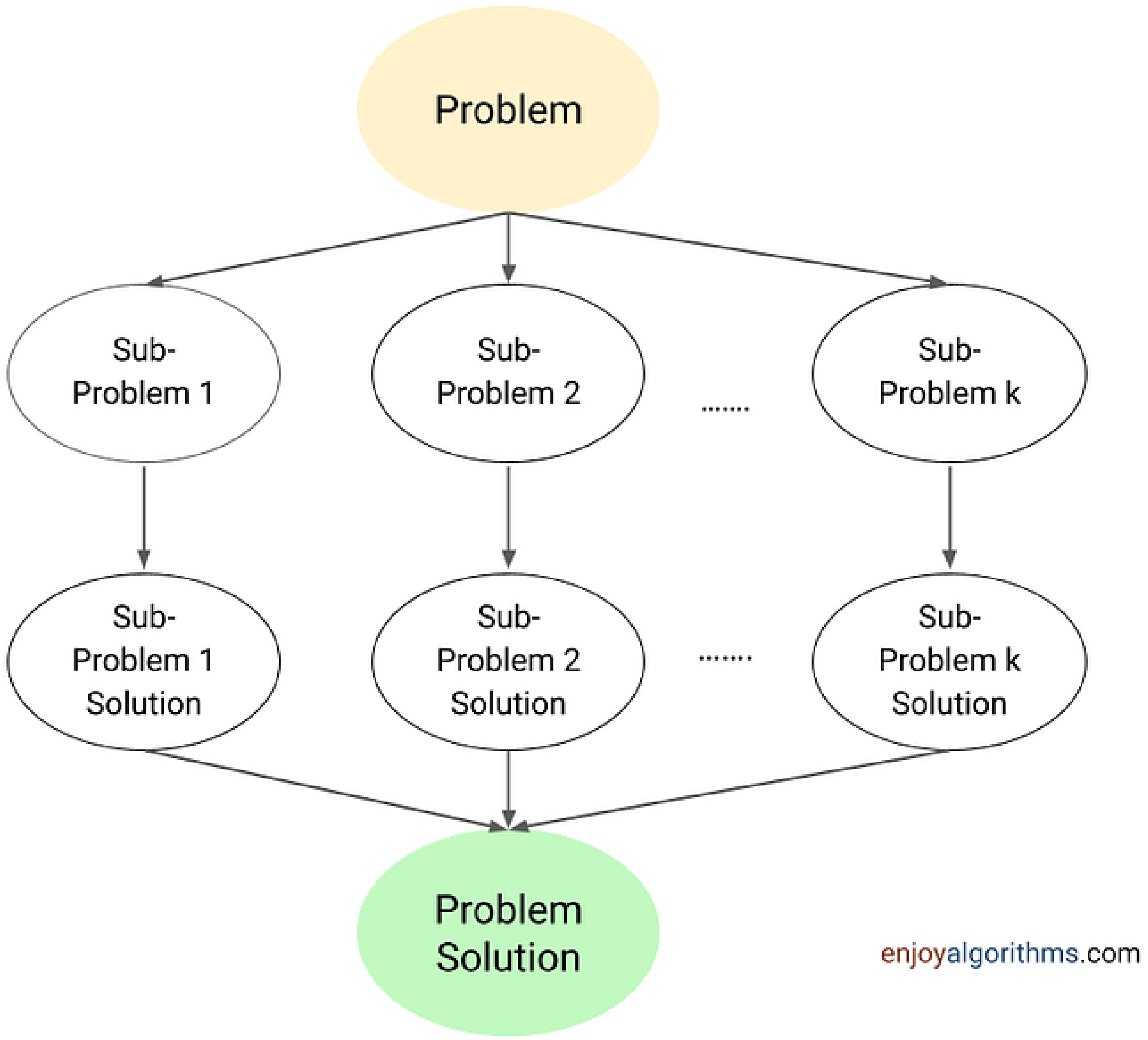
Combine (merge) solutions to subproblems into a solution to the original problem

Divide and Conquer

Divide
Dividing the problem into smaller sub-problems

Conquer
Solving each sub-problems recursively

Combine
Combining sub-problem solutions to build the original problem solution



Divide-and-Conquer

Most **common** usage.

- Break up problem of size n into **two** equal parts of size ½n.
- Solve two parts recursively.
- Combine two solutions into overall solution in **linear time.**

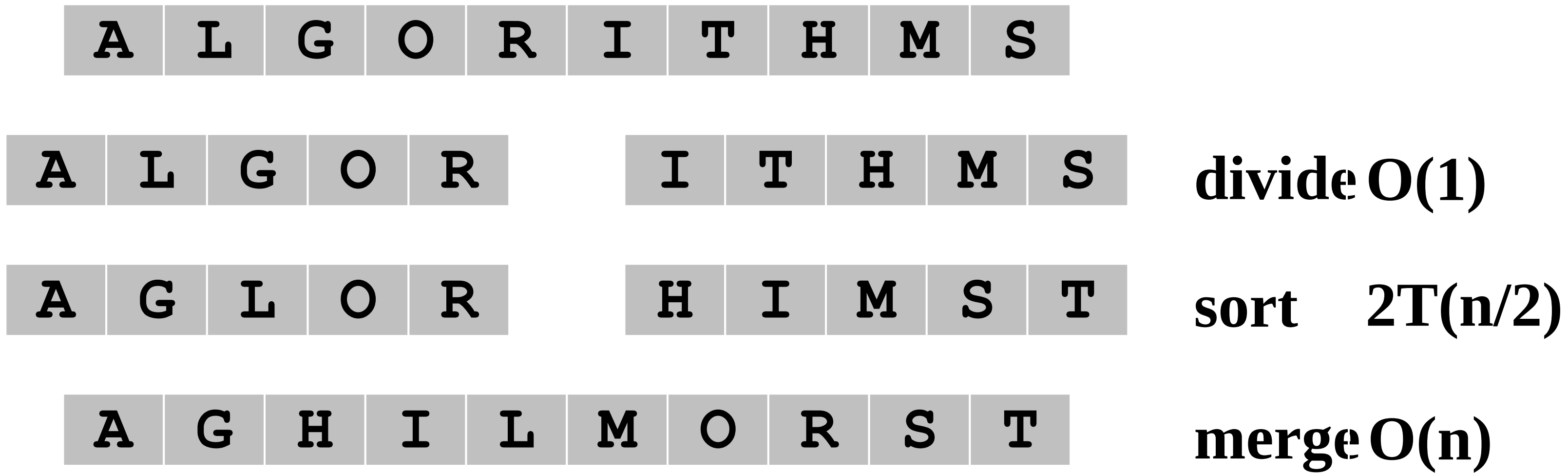
Consequence.

- Brute force: n^2.
- Divide-and-conquer: n log n.

Mergesort

Mergesort.

- Divide array into two halves.
- Recursively sort each half.
- Merge two halves to make sorted whole.

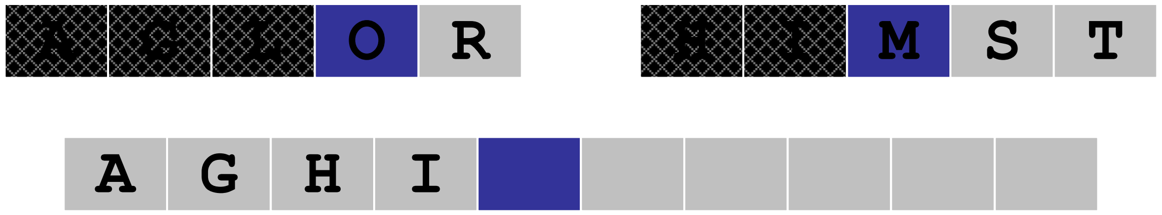


Merging

Merging. Combine two pre-sorted lists into a sorted whole.

How to merge efficiently?

- Linear number of comparisons.
- Use temporary array.



A Useful Recurrence Relation

Def. T(n) = number of comparisons to mergesort an input of size n.

Mergesort recurrence.

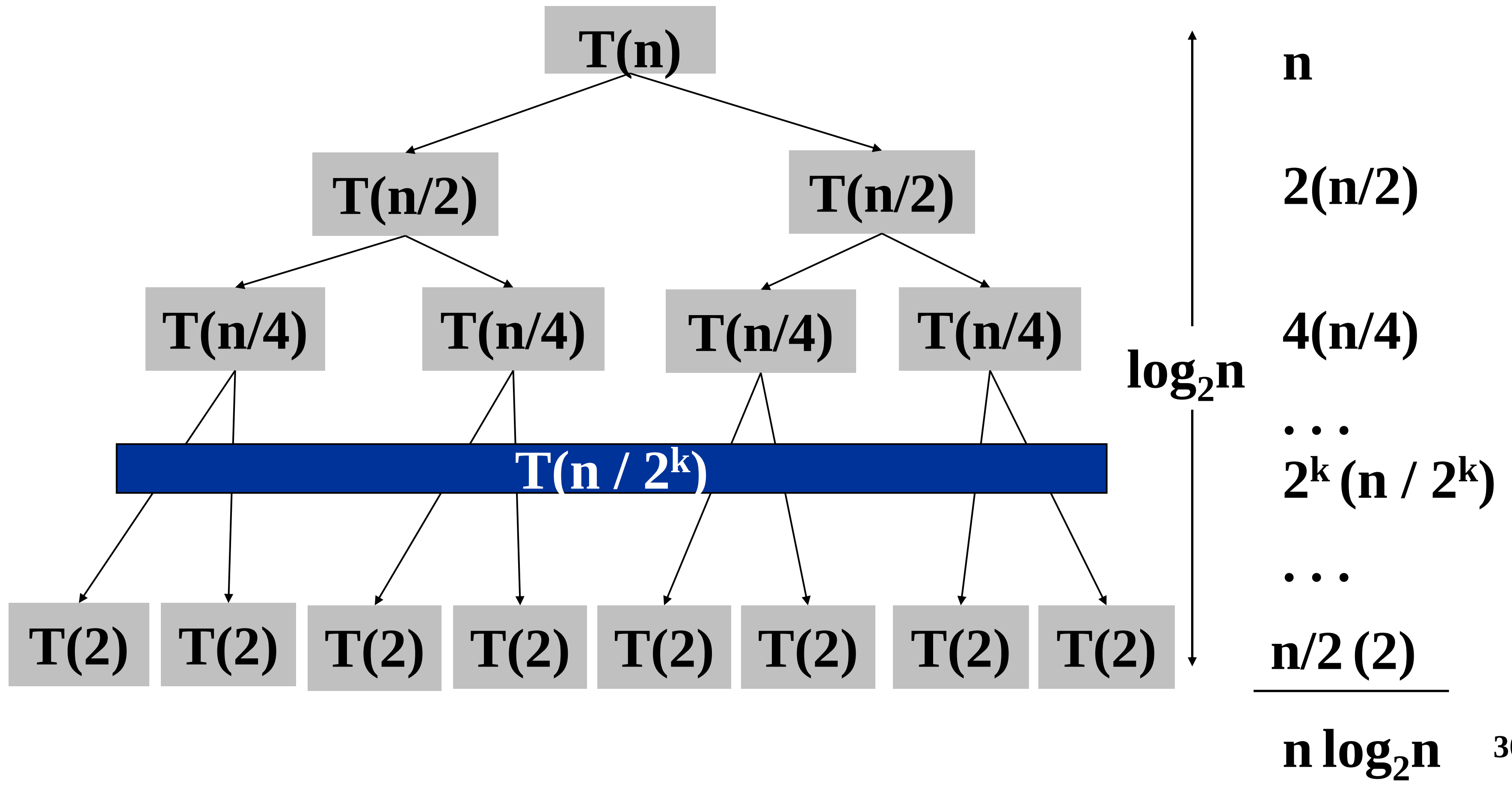
$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + n & \text{otherwise} \end{cases}$$

Solution. T(n) = O(n $\log_2$ n).

Assorted proofs. We describe several ways to prove this recurrence. Initially we assume n is a power of 2 and replace $\leq$ with =.

Proof by Recursion Tree

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$



Proof by Telescoping

Claim. If T(n) satisfies this recurrence, then T(n) = n $\log_2$ n. ↑

assumes n is a power of 2

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Pf. For n > 1:

$$\begin{aligned} \frac{T(n)}{n} &= \frac{2T(n/2)}{n} + 1 \\ &= \frac{T(n/2)}{n/2} + 1 \\ &= \frac{T(n/4)}{n/4} + 1 + 1 \\ &\cdots \\ &= \frac{T(n/n)}{n/n} + \underbrace{1 + \cdots + 1}_{\log_2 n} \\ &= \log_2 n \end{aligned}$$

Proof by Induction

Claim. If T(n) satisfies this recurrence, then T(n) = n $\log_2$ n. $\uparrow$

assumes n is a power of 2

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Pf. (by induction on n)

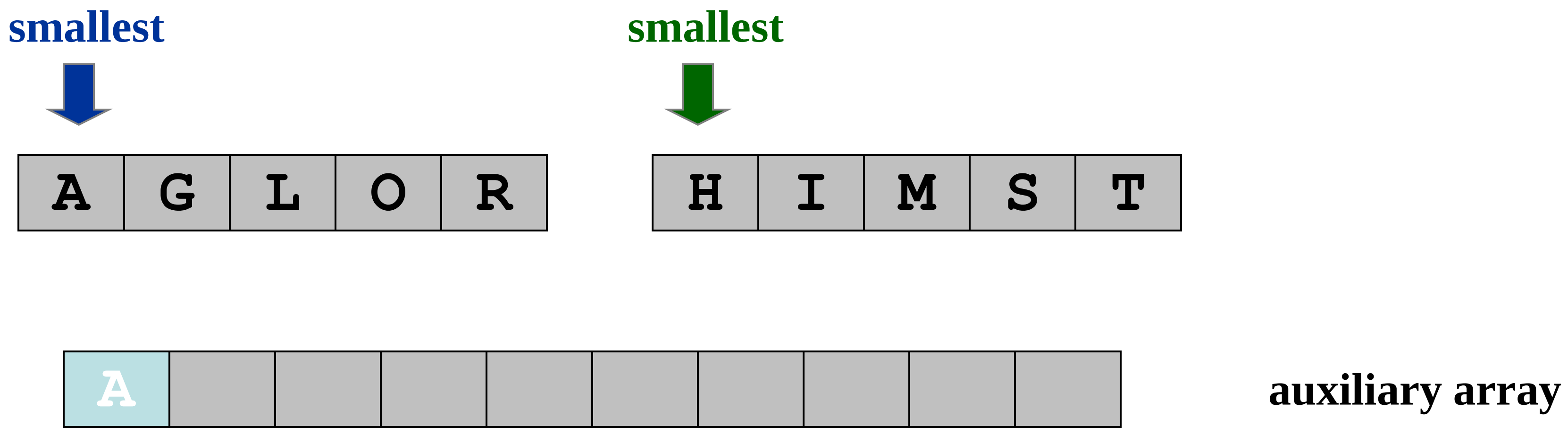
- Base case: n = 1.
- Inductive hypothesis: T(n) = n $\log_2$ n.
- Goal: show that T(2n) = 2n $\log_2$ (2n).

$$\begin{aligned} T(2n) &= 2T(n) + 2n \\ &= 2n\log_2 n + 2n \\ &= 2n\left(\log_2(2n) - 1\right) + 2n \\ &= 2n\log_2(2n) \end{aligned}$$

Merging

Merge.

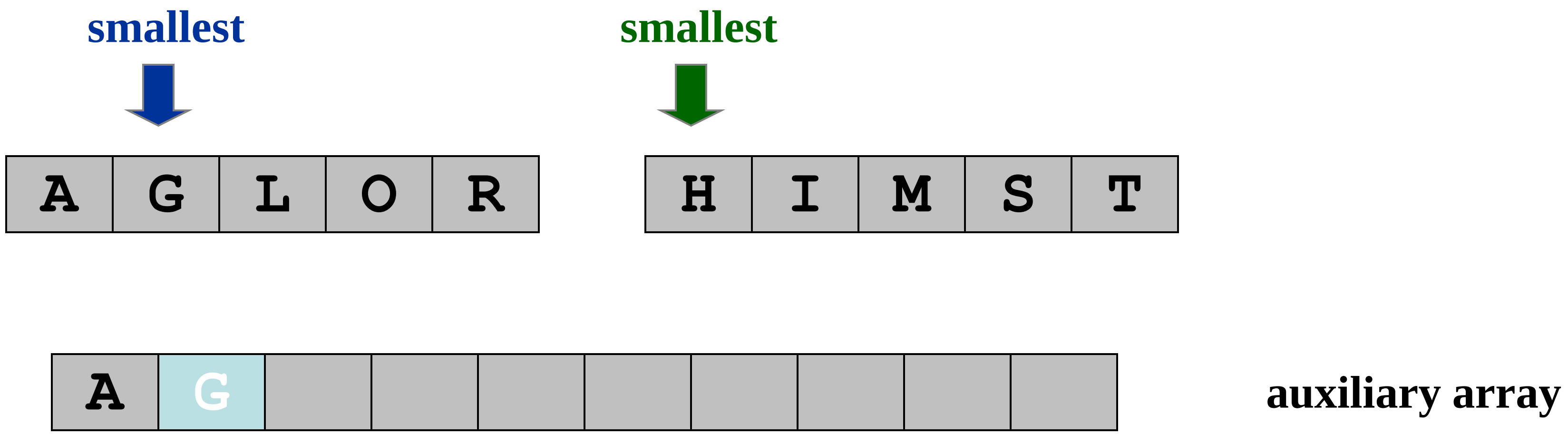
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Merge.

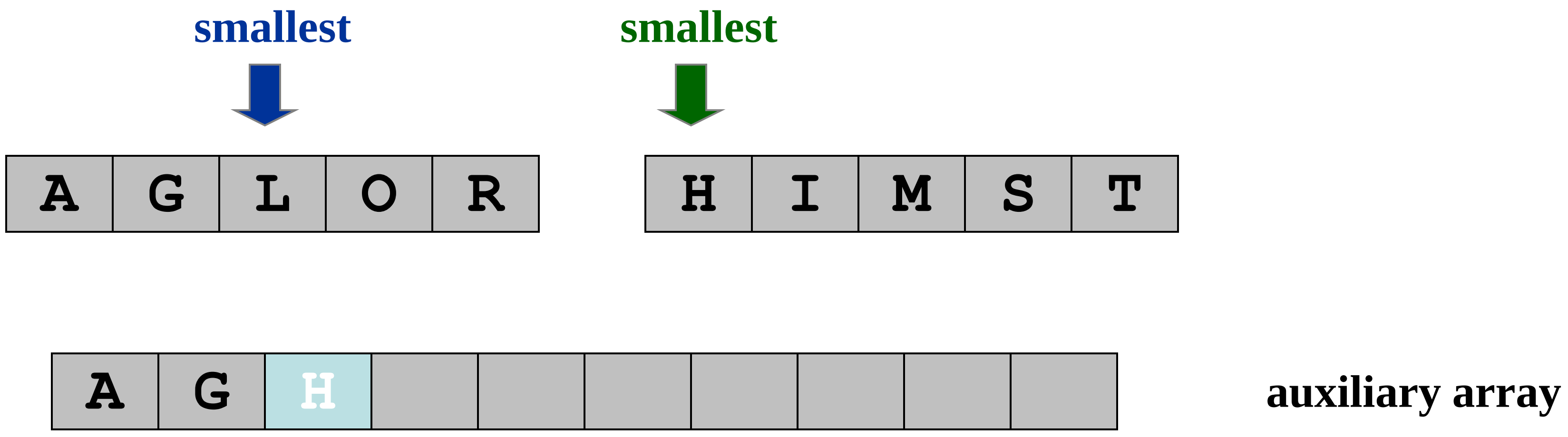
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Merge.

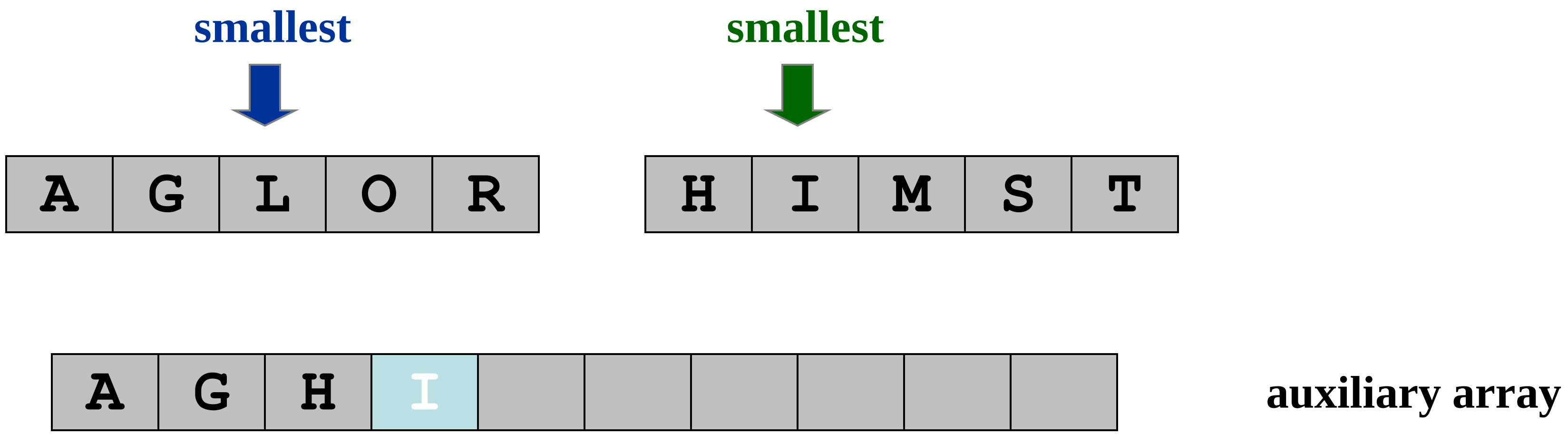
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Merge.

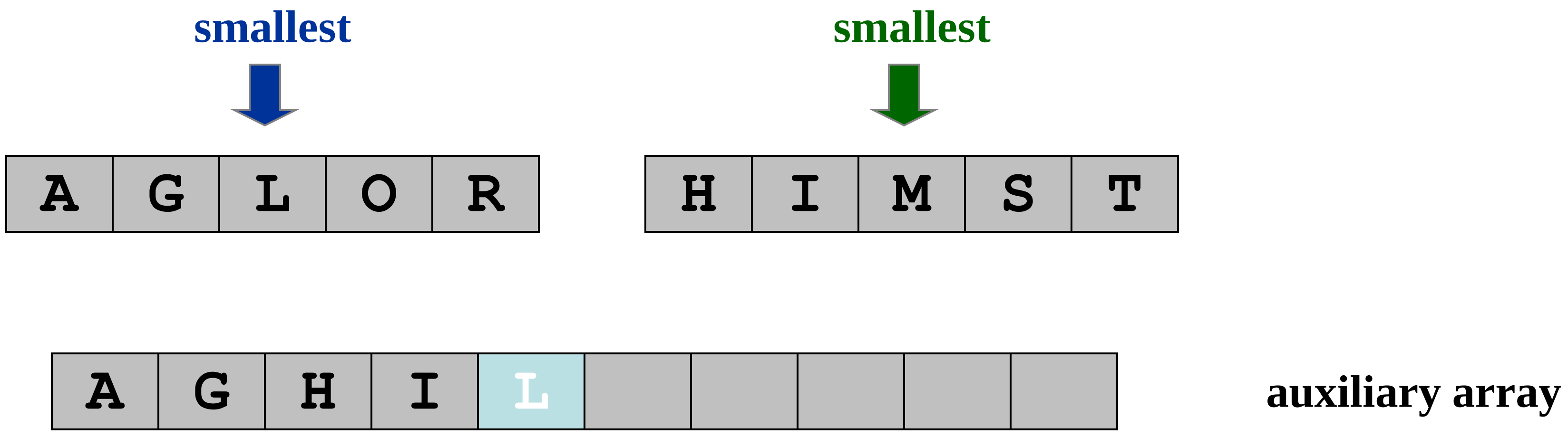
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Merge.

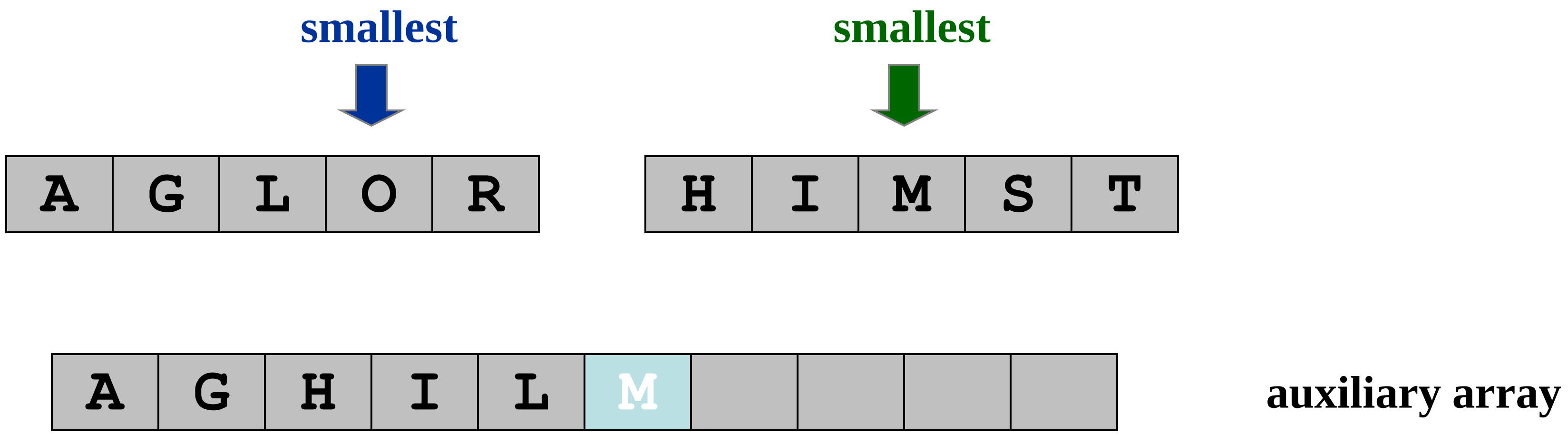
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Merge.

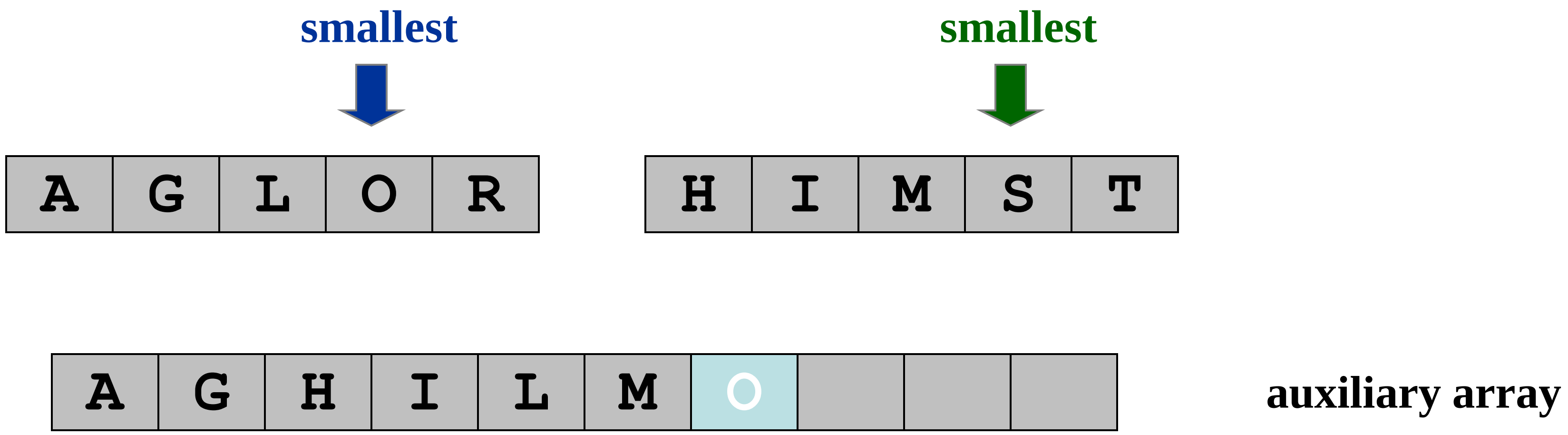
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Merge.

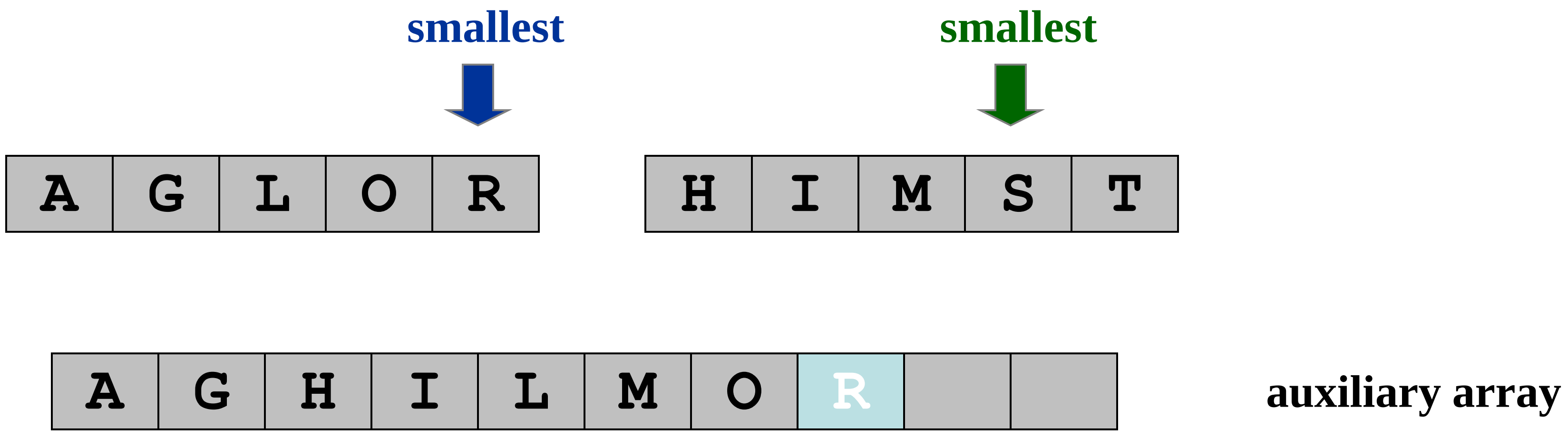
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Merge.

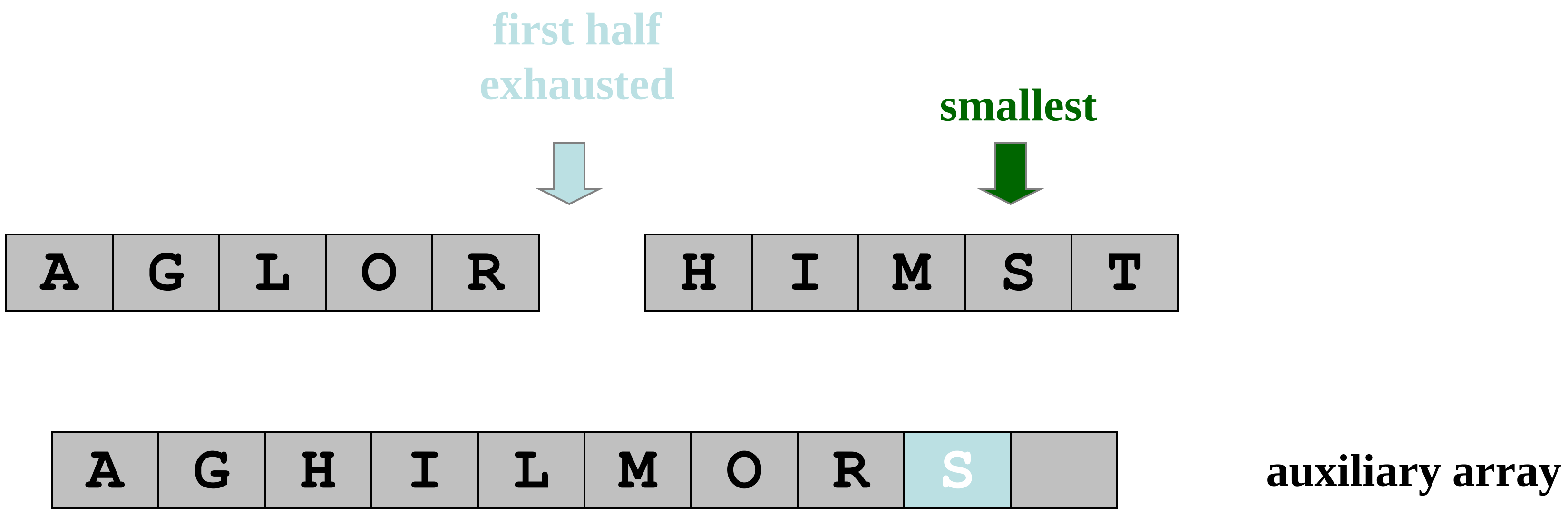
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Merge.

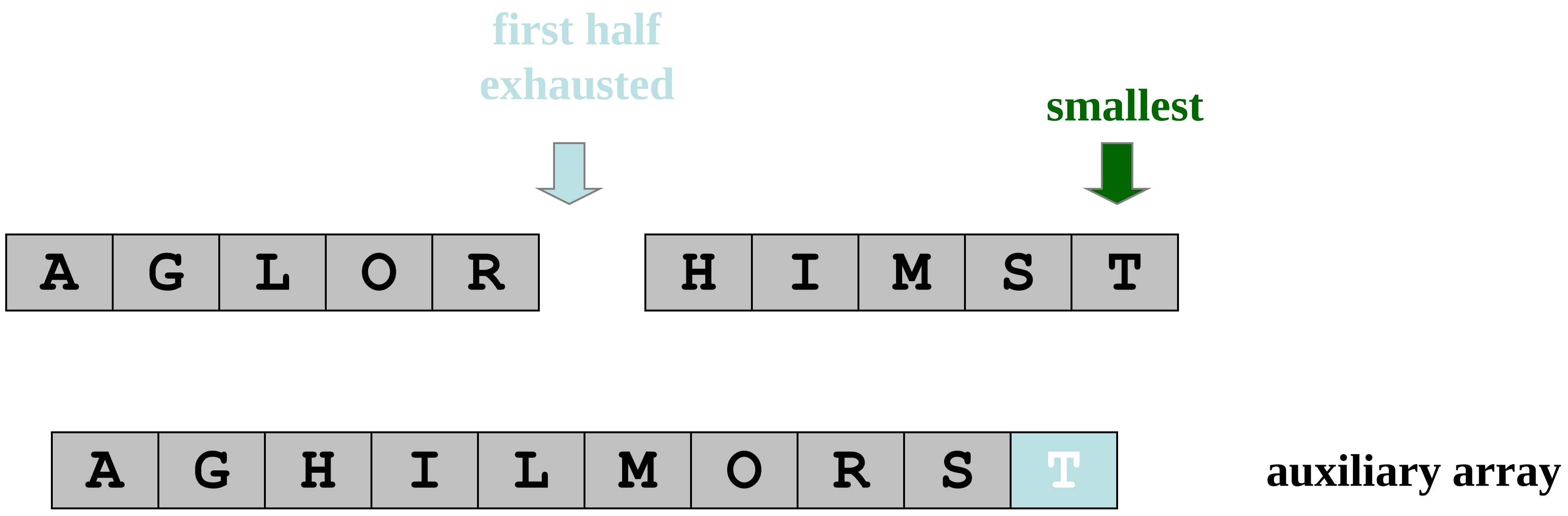
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Merge.

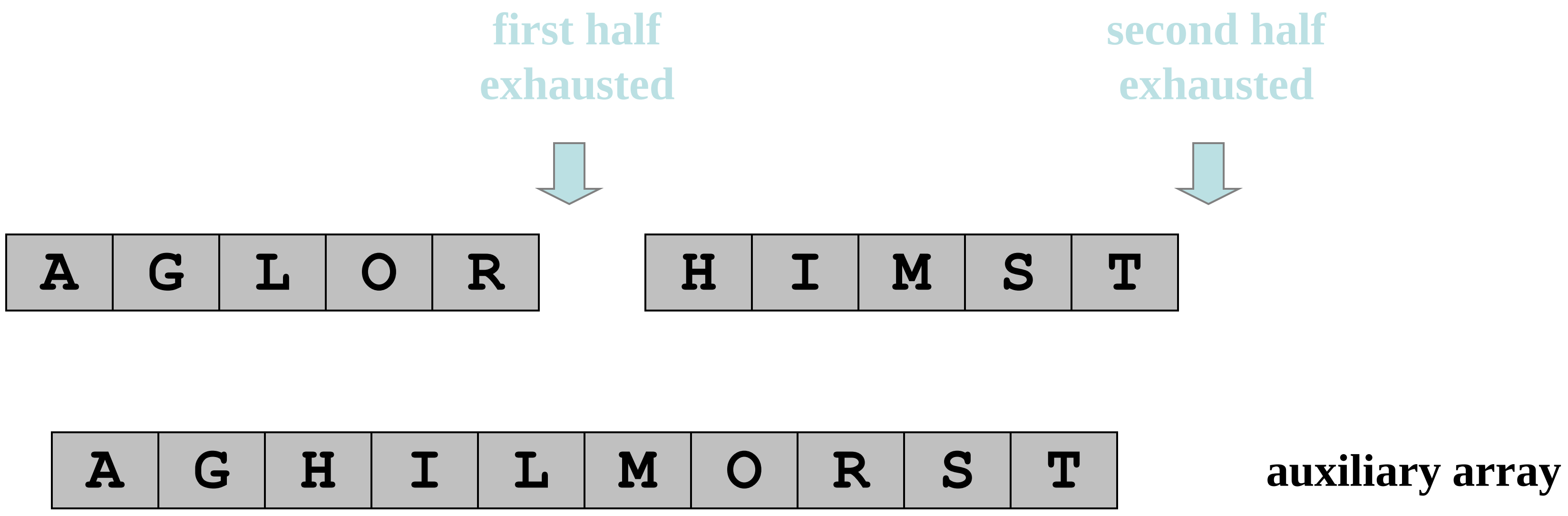
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Merging

Merge.

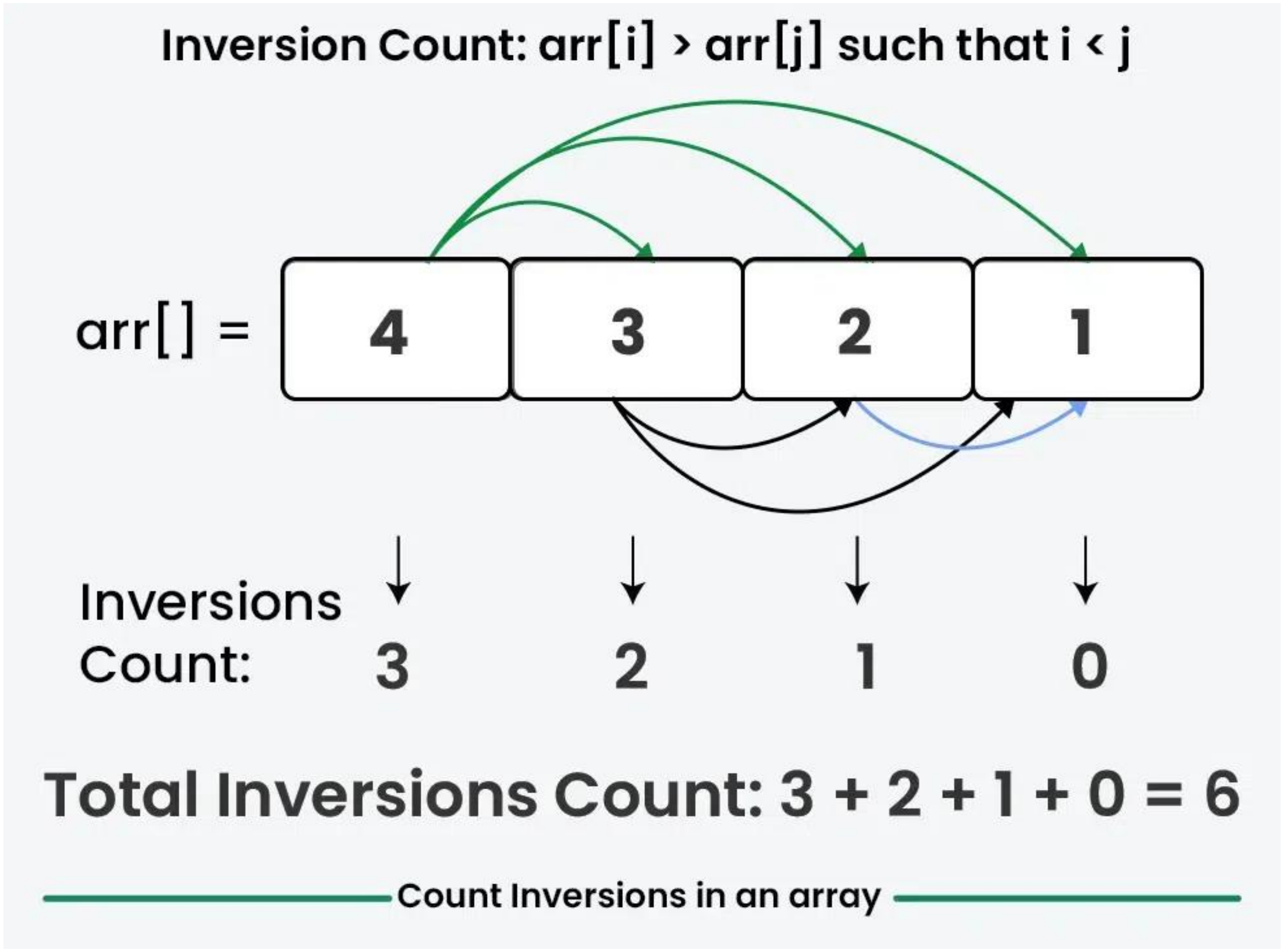
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into auxiliary array.
- Repeat until done.



Counting Inversions

https://www.topcoder.com/thrive/articles/count-inversions-in-an-array

Counting Inversions


Inversion Count: arr[i] > arr[j] such that i < j
arr[] =
4
3
2
1
Inversions Count:
3
2
1
0
Total Inversions Count: 3 + 2 + 1 + 0 = 6
Count Inversions in an array

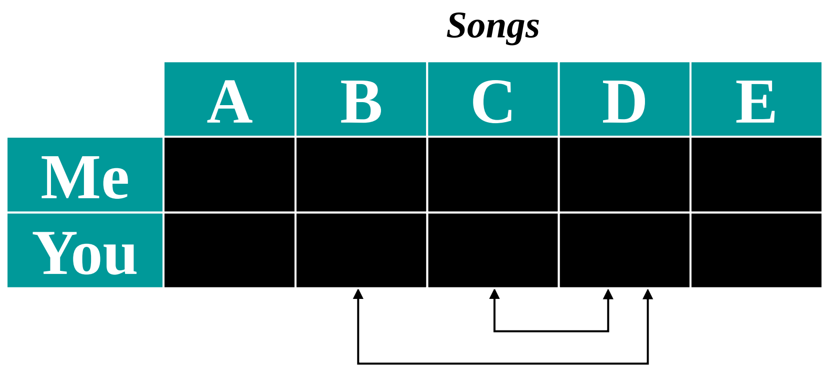
Counting Inversions

Music site tries to match your song preferences with others.

- You rank n songs.
- Music site consults database to find people with similar tastes.

Similarity metric: number of inversions between two rankings.

- My rank: 1, 2, …, n.
- Your rank: $a_1, a_2, \ldots, a_n$.
- Songs i and j inverted if $i \le j$, but $a_i > a_j$.



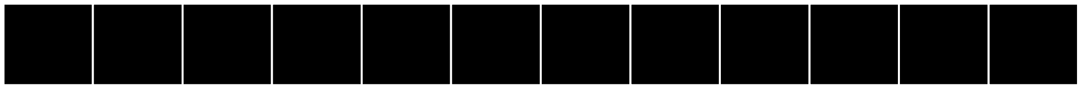
<u>Inversions</u>

3-2, 4-2

Brute force: check all $\Theta(n^2)$ pairs i and j.

Counting Inversions: Divide-and-Conquer

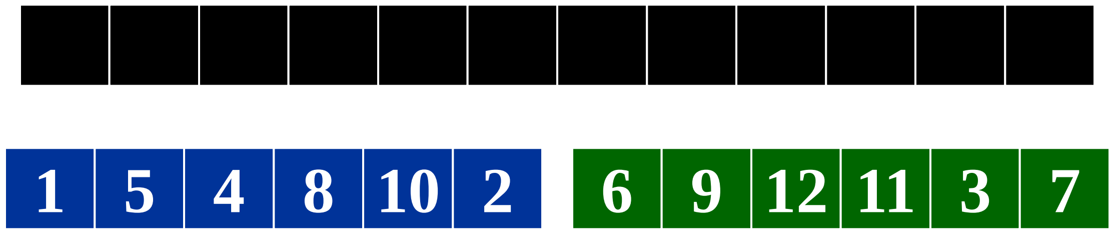
Divide-and-conquer.



Counting Inversions: Divide-and-Conquer

Divide-and-conquer.

- Divide: separate list into two pieces.

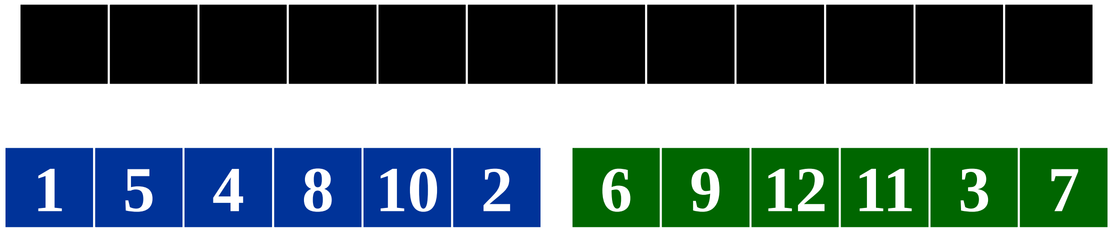


Divide: O(1).

Counting Inversions: Divide-and-Conquer

Divide-and-conquer.

- Divide: separate list into two pieces.
- Conquer: recursively count inversions in each half.



Divide: O(1).

Conquer: 2T(n / 2)

5 blue-blue inversions

8 green-green inversions

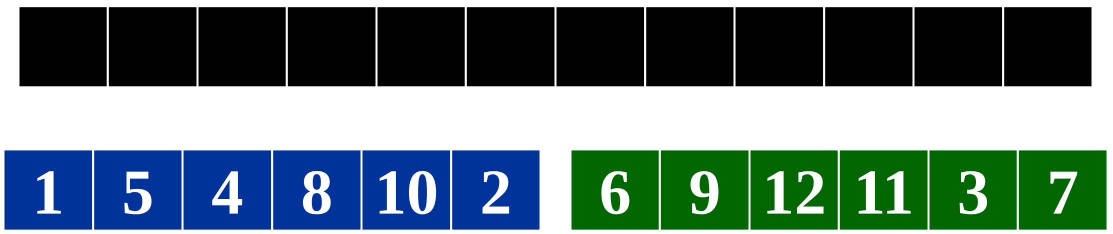
5-4, 5-2, 4-2, 8-2, 10-2

6-3, 9-3, 9-7, 12-3, 12-7, 12-11, 11-3, 11-7

Counting Inversions: Divide-and-Conquer

Divide-and-conquer.

- Divide: separate list into two pieces.
- Conquer: recursively count inversions in each half.
- Combine: count inversions where a_i and a_j are in different halves, and return sum of three quantities.



1	5	4	8	10	2	6	9	12	11	3	7

Divide: O(1).

Conquer: 2T(n / 2)

5 blue-blue inversions

8 green-green inversions

9 blue-green inversions

5-3, 4-3, 8-6, 8-3, 8-7, 10-6, 10-9, 10-3, 10-7

Combine: ???

Total = 5 + 8 + 9 = 22.

Counting Inversions: Divide-and-Conquer

How can we combine the results in O(n) time?

!!!Try Yourself!!!

!!!Try Yourself!!! – Algorithm to find Second MAX from an array

Derive an Divide & Conquer algorithm to find the Second MAX from an array of N elements and find the complexity of your algorithm.

3.4 (due Sep 28, 2006) We are given two arrays of integers $A[1..n]$ and $B[1..n]$, and a number X. Design an algorithm which decides whether there exist $i, j \in \{1, \ldots, n\}$ such that $A[i] + B[j] = X$. Your algorithm should run in time $O(n \log n)$.

!!!Try Yourself !!! - Two Dimensional Search

You are given an $m \times n$ matrix of numbers A, sorted in increasing order within rows and within columns. Assume $m = O(n)$. Design an algorithm that finds the location of an arbitrary value x, in the matrix or report that the item is not present. Is your algorithm optimal?