Small-World Problem

Problem 1: The Small-World Phenomenon (SWP)

- What does the phrase “it’s such a small world” mean?
  - How far apart are two random individuals?
  - Six degrees of separation
- Why should we care?
  - Spreading of rumors, disease, and so on?
- Want to test if the SWP holds for a system of interactions (people, proteins, computers, etc.)
- How do we test it algorithmically?
  - Need a mathematical model of the relation “knows”

A Model

- A network with the small-world property is characterized by small path lengths.
- Use an undirected graph $G(V, E)$.
  - Individuals correspond to nodes of G.
  - x knows y if and only if $\{x, y\} \in E$.
- Our goal is to compute the distribution and basic statistical properties of pairwise shortest path lengths.
- Special case: What do you do if the graph is not connected?
- The SWP problem then takes as input a graph $G(V, E)$ and outputs the distribution of distances d_{uv} between all pairs of nodes u, v in G.

A Naïve Distance Algorithm

- The SWP problem requires an algorithm to find the distance between two nodes.
- A simple algorithm is based on a direct application of the definition of distance. Given two nodes u and v, check if there is a path $u = u_0, u_1, \dots, u_k = v$ of length k for increasing values of k, starting at 1.
- Remaining problems are as follows:
  - How do you check if there is a path of length k?
  - When does the algorithm stop?

A Brute Force Algorithm

DISTANCE(V, E, u, v)

▷ *Input.* Undirected graph $G(V, E)$ and $u, v \in V$

▷ *Output.* Length of shortest path from u to v in G or ∞ if not connected

```
k ← 1
while k < |V|
      do u_0 ← u
         u_k ← v
         for each subset U ⊂ V − {u, v} of size k − 1
              do for each permutation ⟨u_1, . . . , u_{k−1}⟩ of U
                      do foundPath ← TRUE
                         for i ← 1 to k
                              do if {u_{i−1}, u_i} ∉ E
                                      then foundPath ← FALSE
                                           break
                         if foundPath = TRUE
                            then return k
         k ← k + 1
return ∞
```

Efficiency

- Runtime depends on how input is provided
  - Assume that $G(V, E)$ is given by adjacency matrix
- Running time also depends on the input size $(n + m)$ as well as the actual input
  - n is number of vertices, m is number of edges
- Review
  - Number of subsets of size k of a set of size n

$$\binom{n}{k} = \frac{n!}{(n-k)!\,k!}$$

  - Number of permutations of a set of size n

$$n!$$

Analysis

DISTANCE(V, E, u, v)

1	1	$k \leftarrow 1$	
n	2	**while** $k < \lvert V \rvert$	
$n-1$	3	**do** $u_0 \leftarrow u$	
$n-1$	4	$u_k \leftarrow v$	
	5	**for** each subset $U \subset V - \{u, v\}$ of size $k-1$	$\binom{n-2}{k-1}$
	6	**do for** each permutation $\langle u_1, \ldots, u_{k-1} \rangle$ of U	$(k-1)!$
	7	**do** *foundPath* $\leftarrow$ TRUE	1
	8	**for** $i \leftarrow 1$ **to** k	$k+1$
	9	**do if** $\{u_{i-1}, u_i\} \notin E$	k
	10	**then** *foundPath* $\leftarrow$ FALSE	1
	11	**break**	1
	12	**if** *foundPath* $=$ TRUE	1
	13	**then return** k	0
$n-1$	14	$k \leftarrow k + 1$	
1	15	**return** ∞	

Lines 7,12   Line 9   Lines 10,11

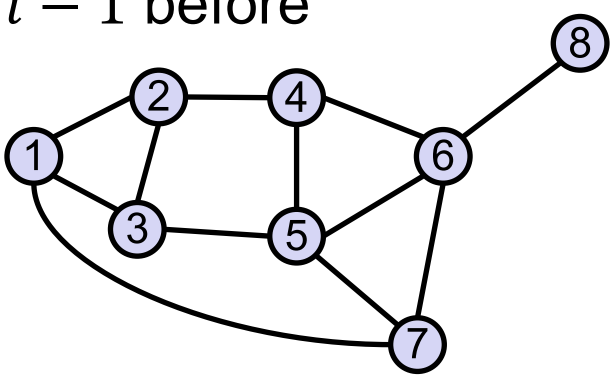
$$T(n) = 2 + n + 3(n-1) + \sum_{k=1}^{n-1} \left[\binom{n-2}{k-1} (k-1)! \, (2 + (k+1) + k + 2 \right]$$

END

SWP Queues

Improving Brute Force

- Recall that our brute force algorithm looks for the shortest path by considering potential paths in order of increasing length: $1, 2, 3, \ldots$
- We can improve performance by remembering partial shortest paths that have been found in a data structure (i.e., remember all nodes at distance 1, 2, 3, and so on, in that order).
- *Key insight:* If $d_{uv} = k$, then there must be w such that $d_{uw} = k - 1 \Rightarrow$ find *all* nodes at distance $i - 1$ before finding all nodes at distance i.
- Use Queues to hold these partial paths



Data Structures

- A ***data structure*** is a policy for organizing data in computer memory with the goal of supporting a specific set of operations efficiently.
  - *Example:* Given a list of values $\langle x_1, x_2, \ldots, x_n \rangle$, determine if an arbitrary value x appears in the list
- A data structure is the basis for implementing an abstract data type (a class).
  - The ADT defines the *logical form* of the data
  - The data structure implements the *physical form*
  - *Example:* dictionaries, adjacency matrix, adjacency list
- Algorithms + data structures = programs.

Queue ADT

- A *queue* is a dynamic set that supports inserts and deletes; a delete operation (*dequeue*) removes the element that has been in the set for the longest time
- Enforces a first-in, first-out (FIFO) policy
- Operations
  - Queue(*Type*) → *Queue*
    - Constructor, creates an empty set of elements of type *Type*
  - Empty(*Queue*) → *Boolean*
    - True if set has no elements
  - Enqueue(*Queue, Type*) → *Queue*
    - Adds an element to the set
  - Dequeue(*Queue*) → *Type*
    - Removes and returns the oldest element in the set (error if empty)

Queue Data Structure

- Queues can be implemented in a number of ways
  - Some implementations are more efficient than others
  - We will explore the details of these implementations later in the course
- When implemented efficiently:
  - Enqueue, dequeue, and empty run in constant time
- It is important to use efficient structures to hold data in help make our algorithms effecient

END

Breadth-First Search

Breadth-First Search (BFS)

- Graph *exploration:* visit all nodes and edges of a graph.
  - Done to compute some properties (e.g., paths between nodes, existence of cycles, connectedness, etc.)
  - Many applications (Web, social networks, routing, games, etc.)
- BFS is a method to systematically explore the nodes of a graph starting from a designated ***source node*** s
- Visit s and its neighbors; then, for each neighbor, visit its neighbors, and so on until no more nodes can be visited!
- A graph is explored level by level—level 0: $\{s\}$.
  - Level i*:* vertices reachable by path of length i but not shorter
  - Level i built from level $i - 1$ by trying all outgoing edges but ignoring vertices from previous levels

BFS

- *Question:* How do we make sure that all the neighbors of a node are visited *before* any of their neighbors are (i.e., how do we guarantee that all nodes at level i are visited before any node of level $i+1$)?
- *Answer:* We do so by using a queue to store the nodes pending exploration.
  - All nodes at level i appear earlier in the queue than any node at level $i+1$.

BFS Algorithm

$\text{BFS}(V, E, s)$

$\triangleright$ *Input.* Undirected graph $G(V, E)$ and source $s \in V$

$\triangleright$ *Output.* Distance d_v from s to v, $\forall v \in V$

Initialize an empty queue Q
for each $u \in V$
    do $d_u \leftarrow \infty$
$d_s \leftarrow 0$
$\text{Enqueue}(Q, s)$
while Q is not empty
    do $u \leftarrow \text{Dequeue}(Q)$
        for each neighbor v of u
            do if $d_v = \infty$
                then $d_v \leftarrow d_u + 1$
                    $\text{Enqueue}(Q, v)$
return d

Efficiency

BFS(V, E, s)

$O(n)$

```
Initialize an empty queue Q
for each u ∈ V
        do d_u ← ∞
d_s ← 0
```

```
ENQUEUE(Q, s)
```

$O(n + m)$

```
while Q is not empty
        do u ← DEQUEUE(Q)
            for each neighbor v of u
                do if d_v = ∞
                    then d_v ← d_u + 1
                         ENQUEUE(Q, v)
```

```
return d
```

- Each node is enqueued/dequeued at most once.
- Each edge is examined at most twice.
- As implemented, BFS runs in $O(n + m)$ time, n = num verts, m = num edges

END

Introduction to Paradigms

Introduction to Paradigms

- An ***algorithm design paradigm*** is a general approach to solving a problem that is applicable to other problems.
- Similar to a design pattern
  - From Object-Oriented programming
- Distills the common structure of “similar” algorithms
  - Similar implementations
  - Similar restrictions on use
  - Similar efficiency
  - Gives us a language to talk about problem solving

Algorithm Design Paradigms

- Techniques include:
  - brute force
  - incremental
  - divide-and-conquer
  - decrease-and-conquer
  - randomization
  - iterative improvement
  - backtracking
  - branch-and-bound
  - dynamic programming
  - greedy solutions
  - approximation

END

Incremental Design

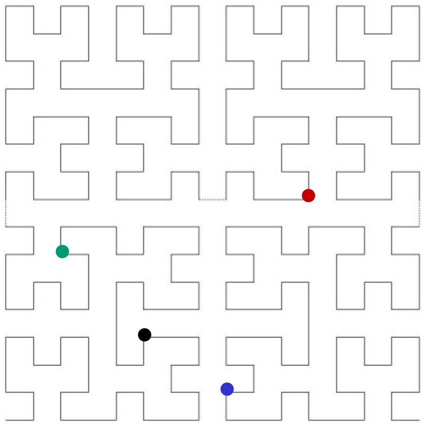
Sorting

- *Input:* a sequence $X = \langle x1, x2, \ldots, xn \rangle$ of n values taken from a *totally ordered* set *U* (not necessarily numbers)
- *Output:* a permutation π of X such that $x_{\pi(i)} \leq x_{\pi(i+1)}$, for $1 \leq i < n$
- Some input instances

  $\langle 20, 5, 4, 13, 9 \rangle$

  $\langle$Bill, Tom, Katie, Mary, Bob$\rangle$

  $\langle (5, 3), (11, 8), (8, 1), (10,4), (2,6) \rangle$



A Sorting Algorithm

Step	Sort(A, n)
1	**for** $i \leftarrow 1$ **to** $n-1$
2	**do** key $\leftarrow A[i]$
3	$j \leftarrow i-1$
4	**while** $j \geq 0$ and $A[j] >$key
5	**do** $A[j+1] \leftarrow A[j]$
6	$j \leftarrow j-1$
7	$A[j+1] \leftarrow$ key

- How is this general?
- Why is it correct?
- How efficient is it?
  - Best and worst cases?

A Python Implementation

```
# sort a list A of integers in ascending order
import random  # package for random generation/manipulation
n = 10         # number of elements
A = [i+1 for i in range(n)] # create list A of integers 1...n
random.shuffle(A)           # randomly permute A

print('Before sorting: ')
print(A)

# Insertion sort
for i in range(1,n):
    key = A[i]
    j = i-1
    while j>=0 and A[j]>key:
        A[j+1] = A[j]
        j = j-1
    A[j+1] = key

print('After sorting: ')
print(A)
```

Incremental Design

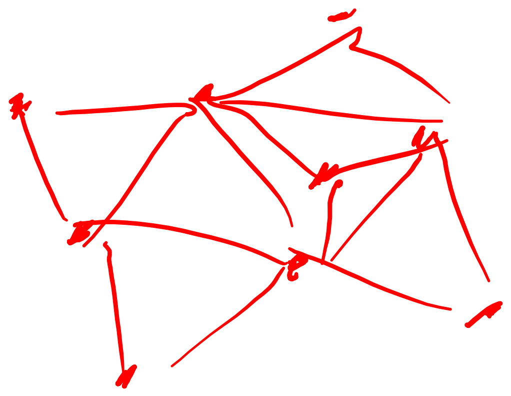
- The sorting algorithm is an instance of an ***incremental design,*** summarized as follows:
  - Consider a problem that takes as input a list of values $x_1, x_2, \ldots, x_n$.
  - Let c denote some suitably chosen constant.

1. Solve the problem for $x_1, \ldots, x_c$ by any method of your choice.
2. **For** $i \leftarrow c + 1$ **to** *n* **do**
   extend the solution for $x_1, \ldots, x_{i-1}$ to a solution for $x_1, \ldots, x_i$.

```
Sort(A, n)
for i ← 2 to n
    do  key ← A[i]
        j ← i − 1
        while j > 0 and A[j] > key
            do  A[j + 1] ← A[j]
                j ← j − 1
        A[j + 1] ← key
```

Incremental Design

- Incremental Tessellation



Incremental Design Paradigm

Incremental approach

- Solve $\langle a_1 \rangle$
- For $i = 2, 3, \ldots, n$:
  - Solve $\langle a_1, ,\ldots, a_i - 1, a_i \rangle$ using solution of $\langle a_1, ,\ldots, a_i - 1 \rangle$

END

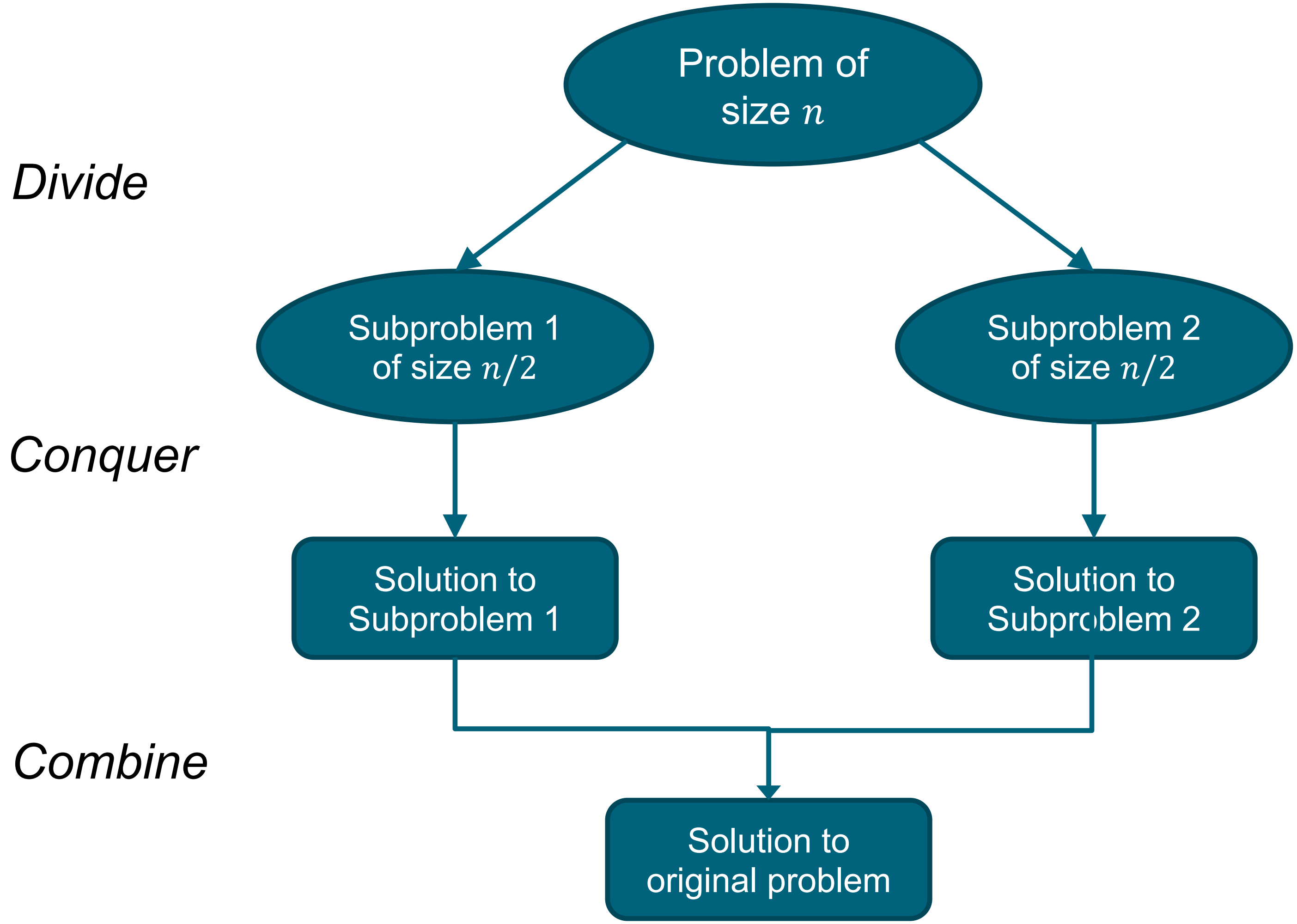
Divide and Conquer

DAC Template

Given a problem of size *n:*

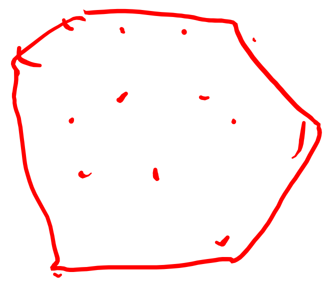
1. **Divide** the problem into *k* subproblems of size *n/k* each
2. **Conquer** by solving each subproblem independently
3. **Combine** the *k* solutions to subproblems into a solution to the original problem

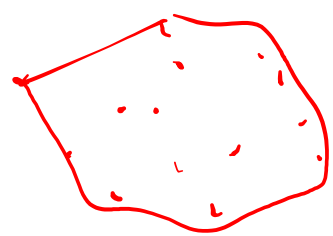
DAC With Two Subproblems



Divide and Conquer Problems

- There are many problems that can be solved using DAC
  - Binary Search, Merge/Quick sort
  - Polynomial/Matrix multiplication, Exponentiation
  - Order Statistics
  - Stock Profit
  - Closest Point Pairs, Merge/Quick Hull



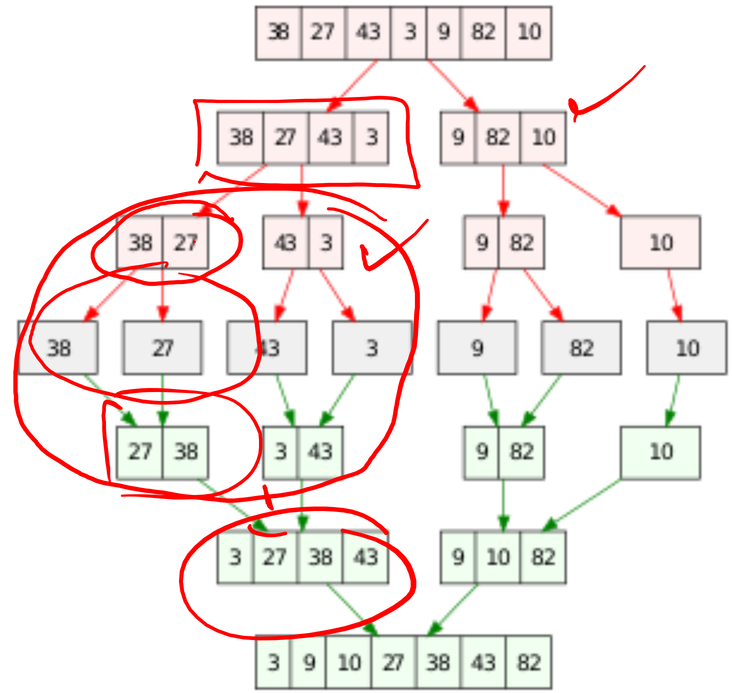


END

Sorting

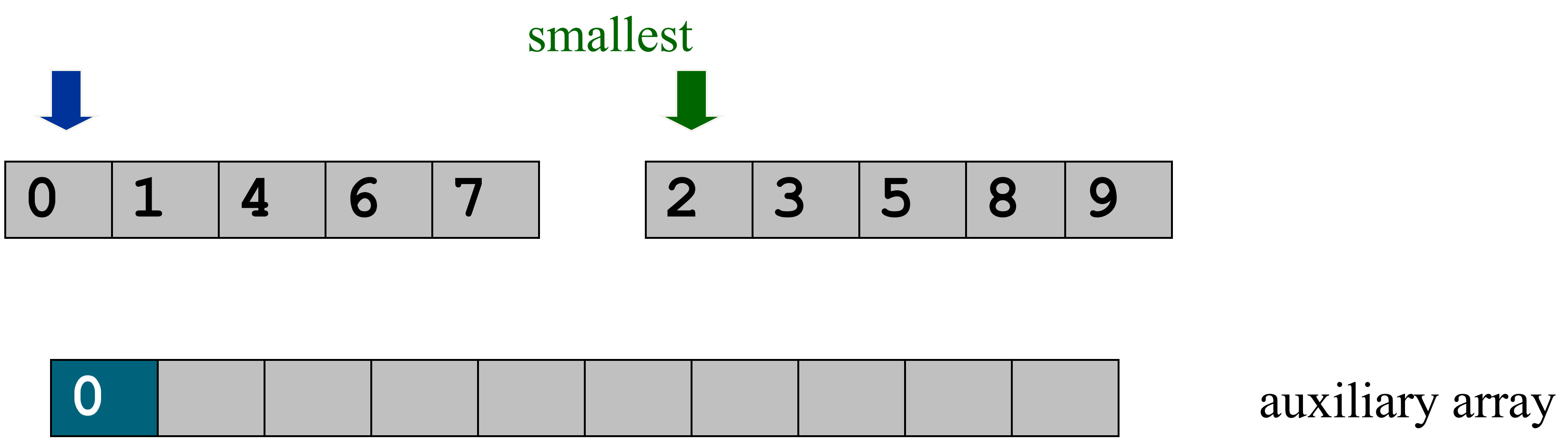
Sorting Revisited

```
def mergeSort(L):
    if len(L) < 2:
        return L[:]
    else:
        mid = len(L)//2
        Left = mergeSort(L[:mid])
        Right = mergeSort(L[mid:])
        return merge(Left, Right)
```



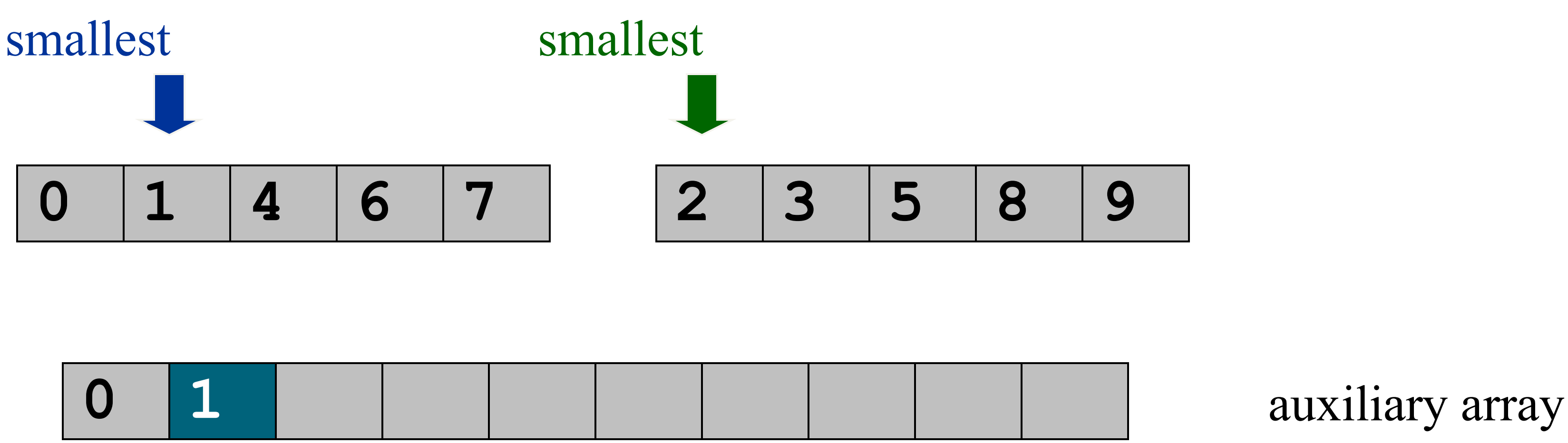
Merging

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



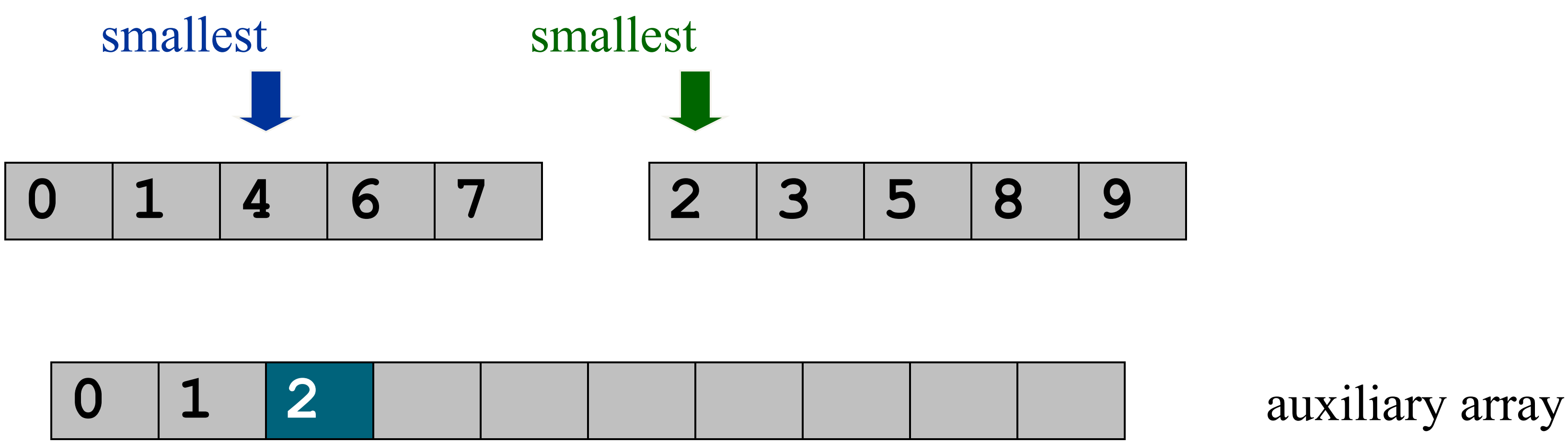
Merging

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



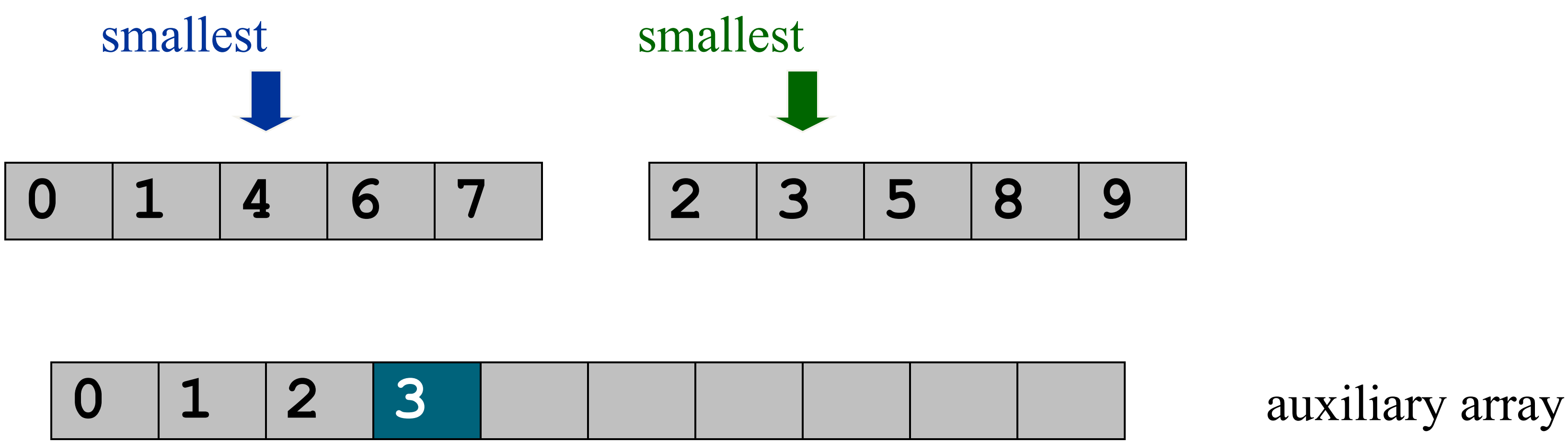
Merging

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



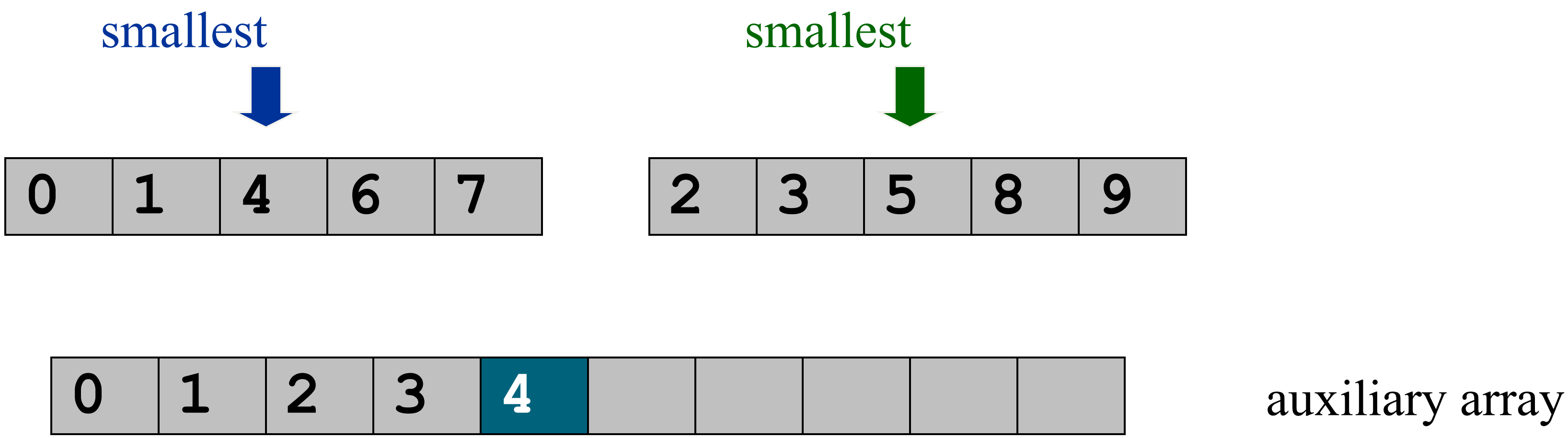
Merging

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



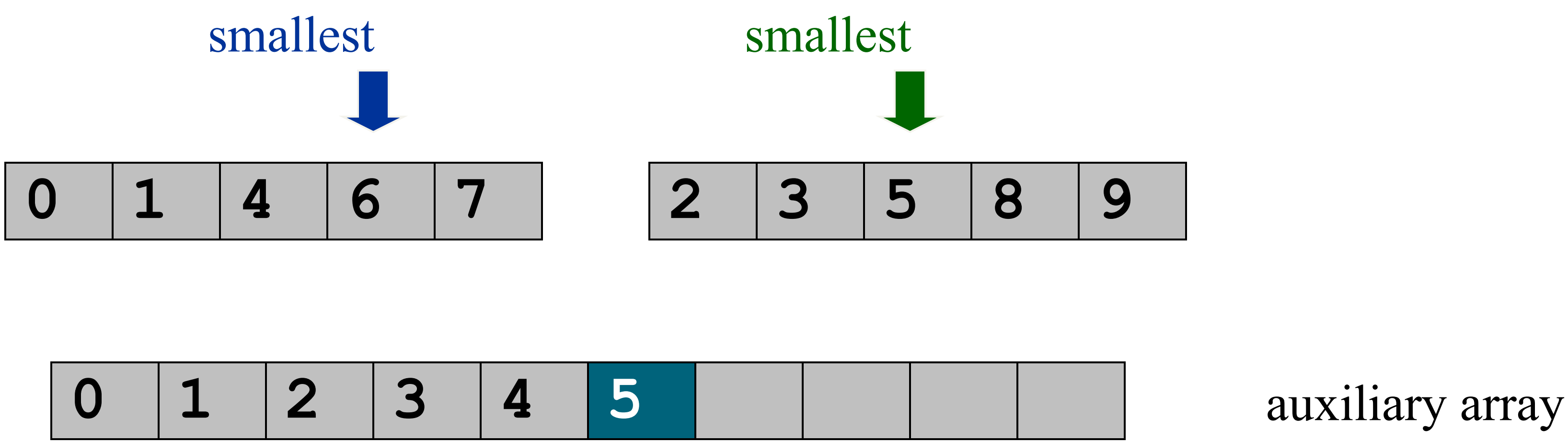
Merging

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



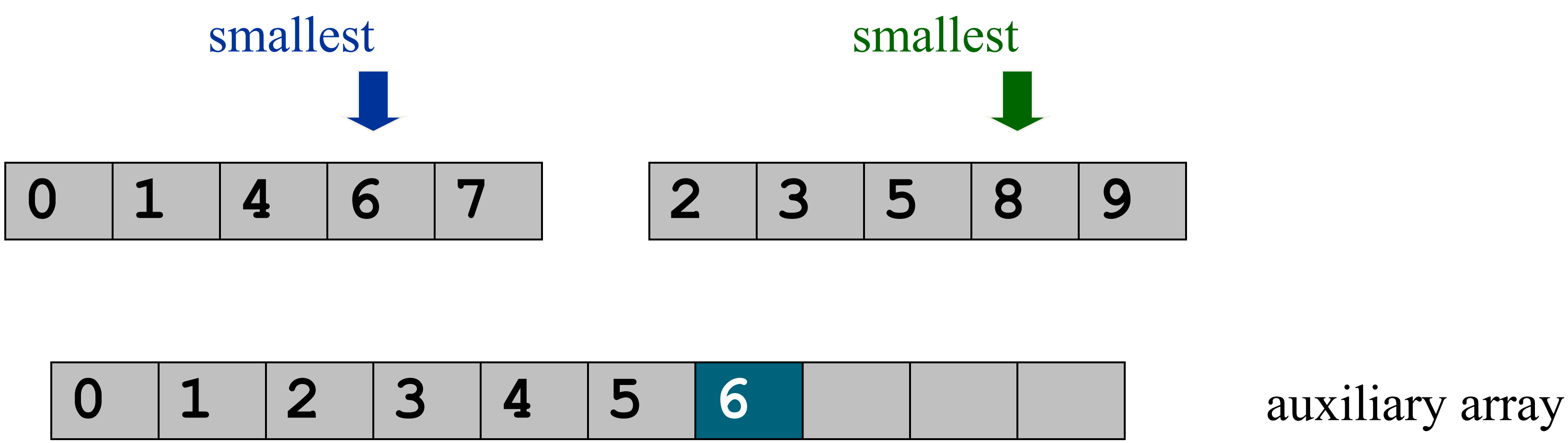
Merging

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



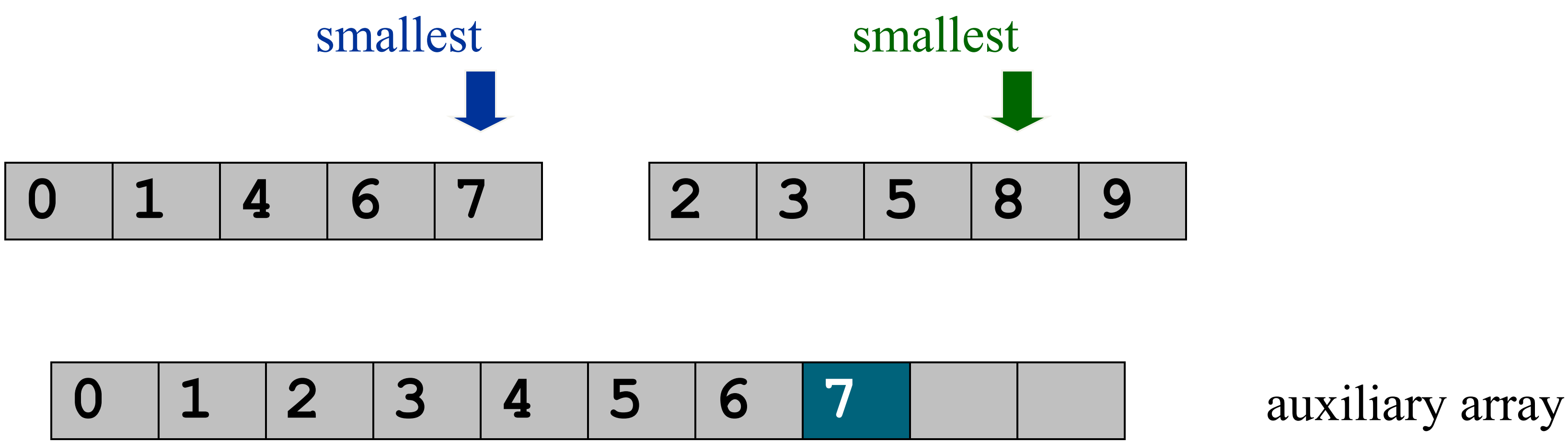
Merging

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



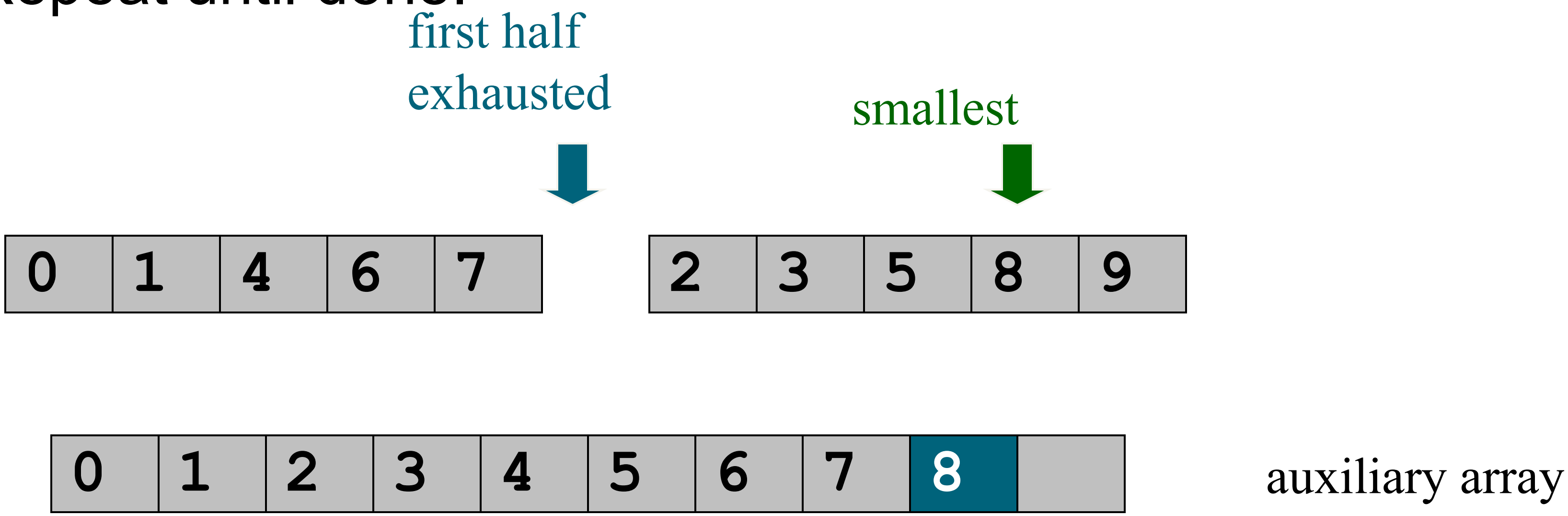
Merging

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



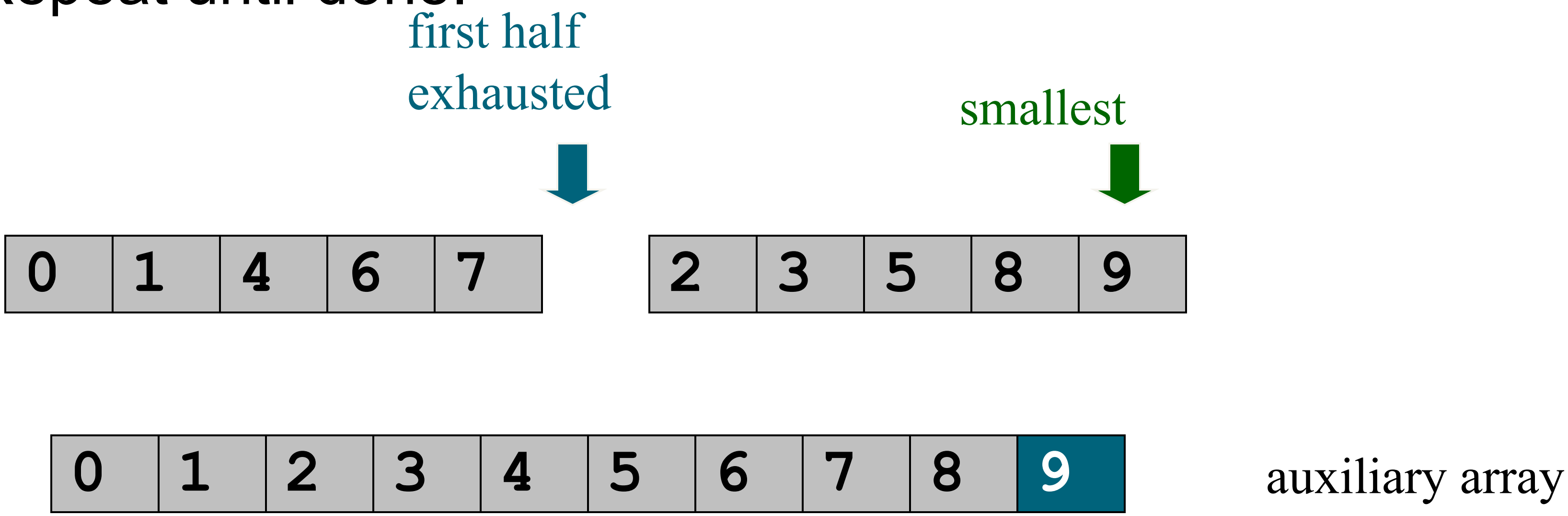
Merging

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



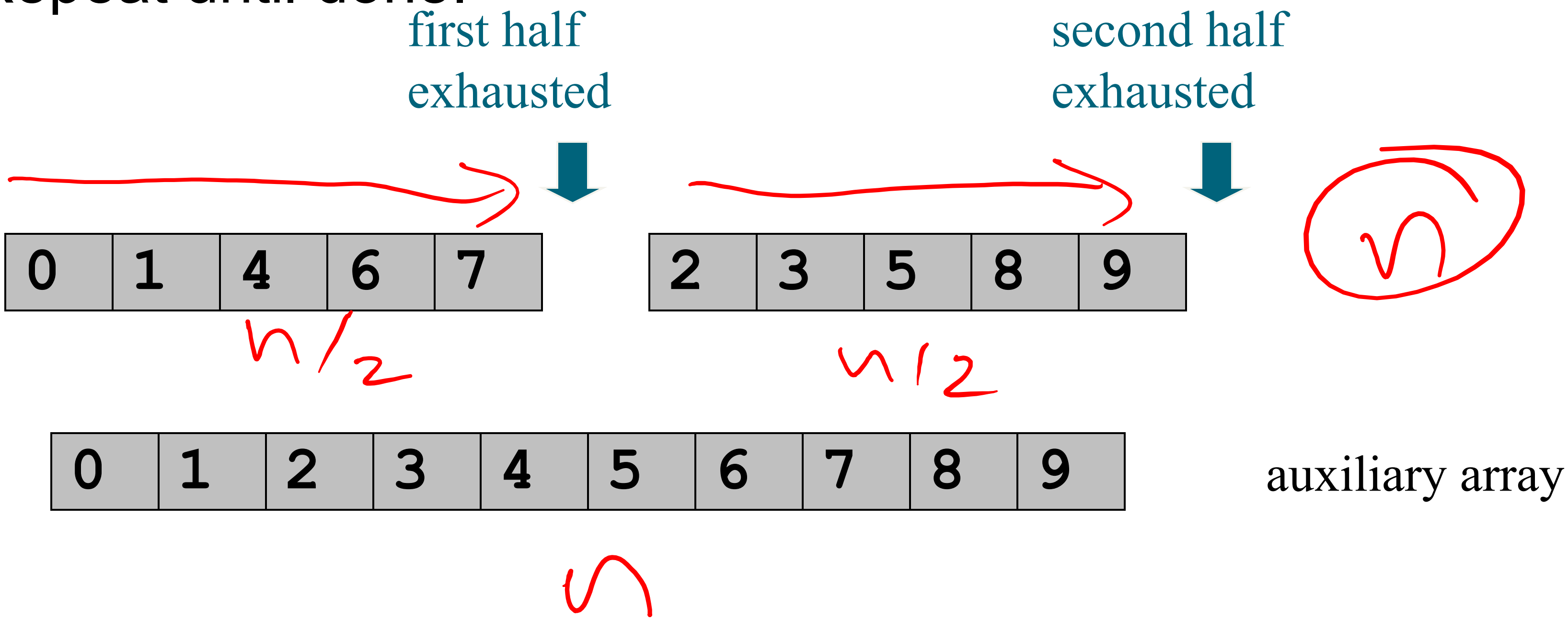
Merging

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



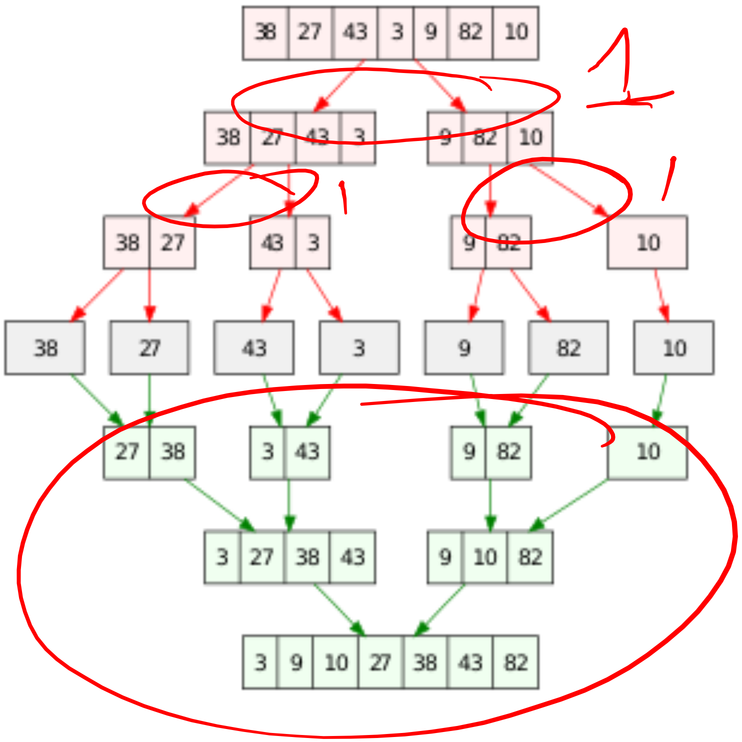
Merging

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



Merge

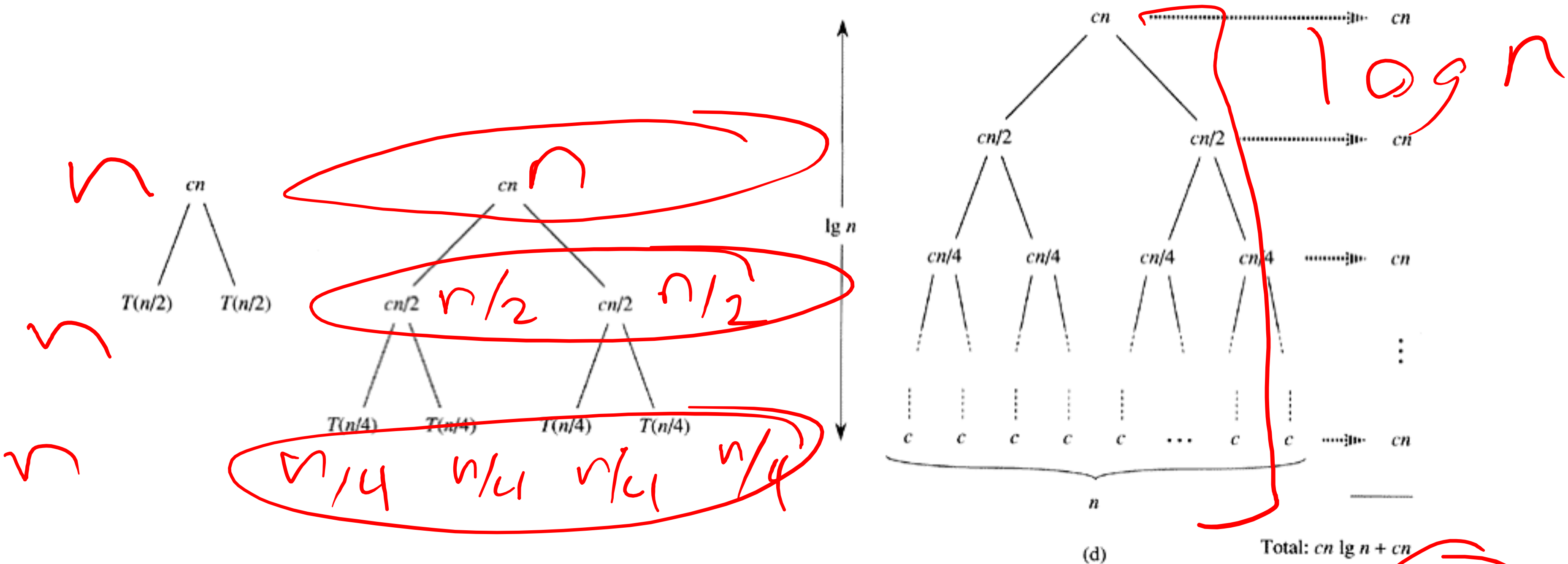
```
def merge(A, B):
      out = []
      i,j = 0,0
      while I < len(A) and j <
   len(B):
            if A[i] < B[j]:
                  out.append(A[i])
                  i += 1
            else:
                  out.append(B[j])
                  j += 1
      while I < len(A):
            out.append(A[i])
            i += 1
      while j < len(B):
            out.append(B[j])
            j += 1
      return out
```



Running Time of Merge Sort

- How long does merge take?
- Accessing an arbitrary element of a list takes $O(1)$ time.
- Complexity is $T(n) = 2 \cdot T(n/2) + n$ (for $n \geq 2$).
- What does this resolve to?

Recursion Tree

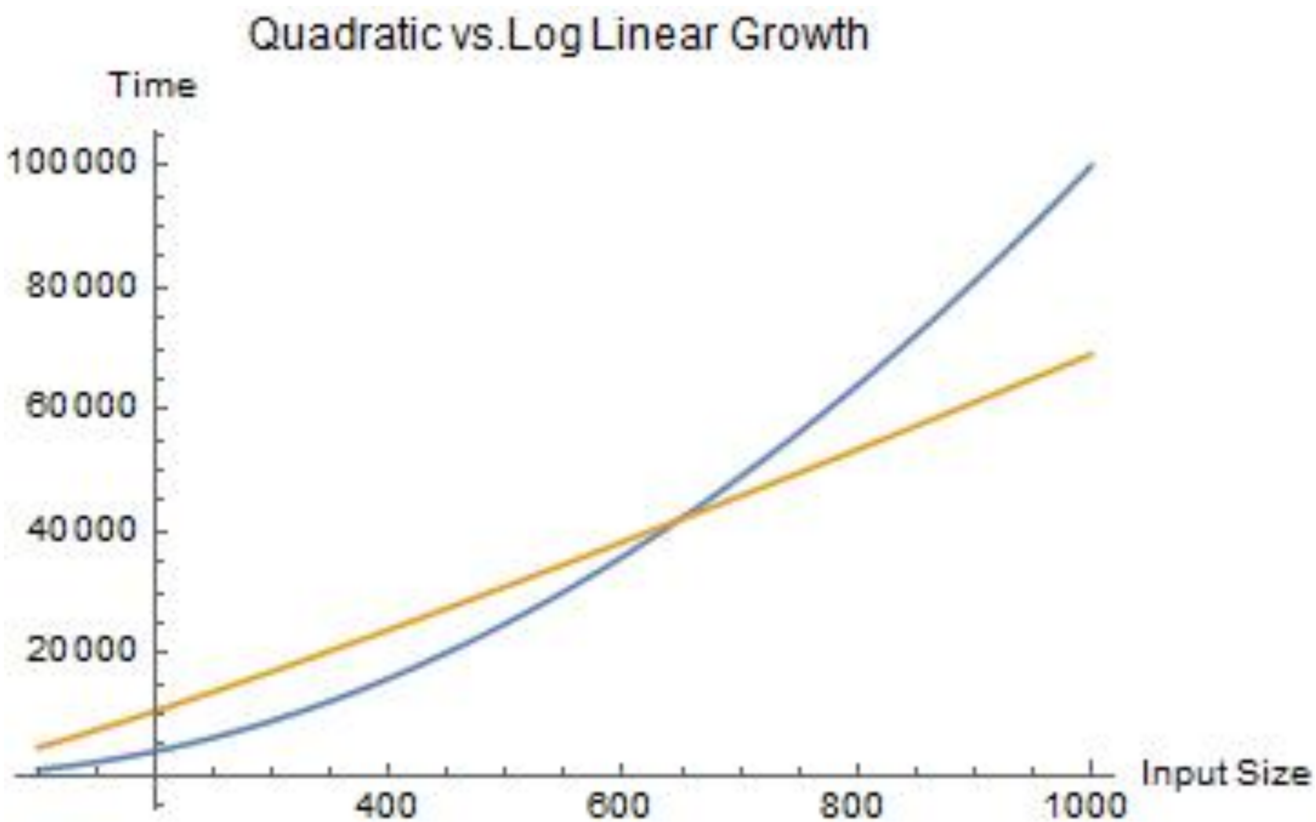


$$T(n) = 2 \cdot T(n/2) + n \in \Theta(n \log n)$$

n^2

Insertion vs. Merge Sort

Comparing $0.1n^2$ vs. $10\, n \log n$



Running Time Comparison

$T_1(n) = 0.1\ n^2$ $\qquad$ $T_2(n) = 10\ n \log n$

Input Size, n	$\Theta(n^2)$	$\Theta(n \log n)$
10	10 μsec	332 μsec
100	1 msec	6.64 msec
1,000	100 msec	100 msec
10,000	10 sec	1.3 sec
100,000	17 min	16 sec
1,000,000	28 hours	3 min
10,000,000	116 days	39 min

END

Paradigms Overview

Paradigms

- We have introduced the idea of paradigms.
- We will cover the following in detail:
  - Brute Force/Incremental
  - Divide and conquer
  - Dynamic programming
  - Greedy
  - Backtracking

Brute Force/Incremental

- Brute Force
  - Try every possibility
  - Often implement the algorithm ”as stated”
  - Usually highly inefficient
- Incremental
  - Build up solution by adding to existing solution

Divide and Conquer

- Divide problem into subproblems
- Solve subproblems
- Combine subproblems to form solution
- Fairly fast
- Can’t always combine subproblems properly

Dynamic Programming

- Similar to divide and conquer
- Less repeated work for overlapping problems

Greedy

- Solve a problem by making local decisions
- The local decisions combine to form a correct global solution to the problem
- Very fast
- But often, the local choices won't combine to form a global solution, so can't use Greedy

Backtracking

- Make a choice and explore to see where it leads
- If doesn't lead to a solution, try another option
- Slow
  - Sometimes so slow that you can't solve in a reasonable amount of time

END

Acknowledgements

- Introduction to Algorithms, 3rd edition, by T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein; MIT Press, 2009
- Wikipedia.org