CS 344: Design and Analysis of Computer Algorithms

(Spring 2022 — Sections 5,6,7,8)

Week 2: Lectures 3 & 4
Proof by Induction, Recursion, Divide and Conquer, Merge Sort

This week’s topics:

- Community detection problem
- Refresher on mathematical induction
- Proof of correctness of algorithms by induction
- Recursion and recursive algorithms
- Divide and conquer technique: Merge sort
- A fun question to think about

This week's topics:

- Community detection problem
- Refresher on mathematical induction
- Proof of correctness of algorithms by induction
- Recursion and recursive algorithms
- Divide and conquer technique: Merge sort
- A fun question to think about
- **Disclaimer:** we are using some of the video lectures from previous iterations of this course

Announcements

- My office hours/Q&A sessions:
  - Thursdays 4pm to 5pm
  - Mondays 5pm to 6pm.
- Quiz 1 is due on Jan 24. “Homework 0” is due on Jan 25.
- Quiz 2 from the materials of this week
  - Due Monday, Jan 31, 11:59pm.
- Homework 1 (topics of lectures 1 to 5)
  - Due Tuesday, Feb 8, 11:59pm EST
- Next week classes: in-person!

An Example of Algorithm Design Process: Community Detection Problem

Problem

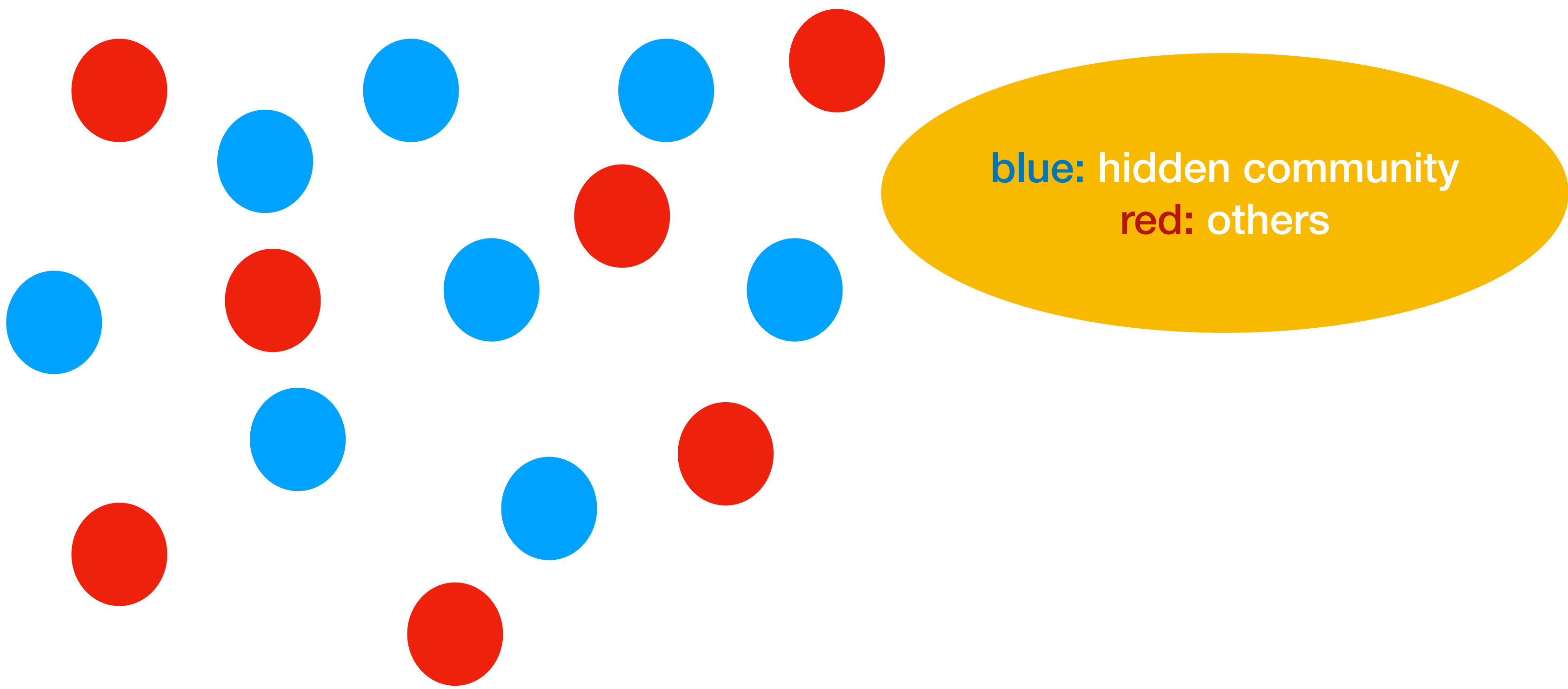
- We have n people in a party for some **odd** number n
- We know that strictly more than half of these people belong to some hidden community
- We can ask two people in the party to **greet** each other:
  - If **both** people belong to this hidden community then they greet each other warmly
  - Otherwise, they say they do not know each other
- Design an algorithm that finds every member of this hidden community using smallest number of greetings possible

Where to start?

- Gain intuition about the problem
- See if you can solve a "puzzle" version of this problem for some reasonably small value of n
- We have 15 people in a party and 8 of them belong to a hidden community
- Can we find all people in this hidden community?
  - There is a simple solution with 105 greetings
  - Can you solve the problem with ≤ 25 greetings?

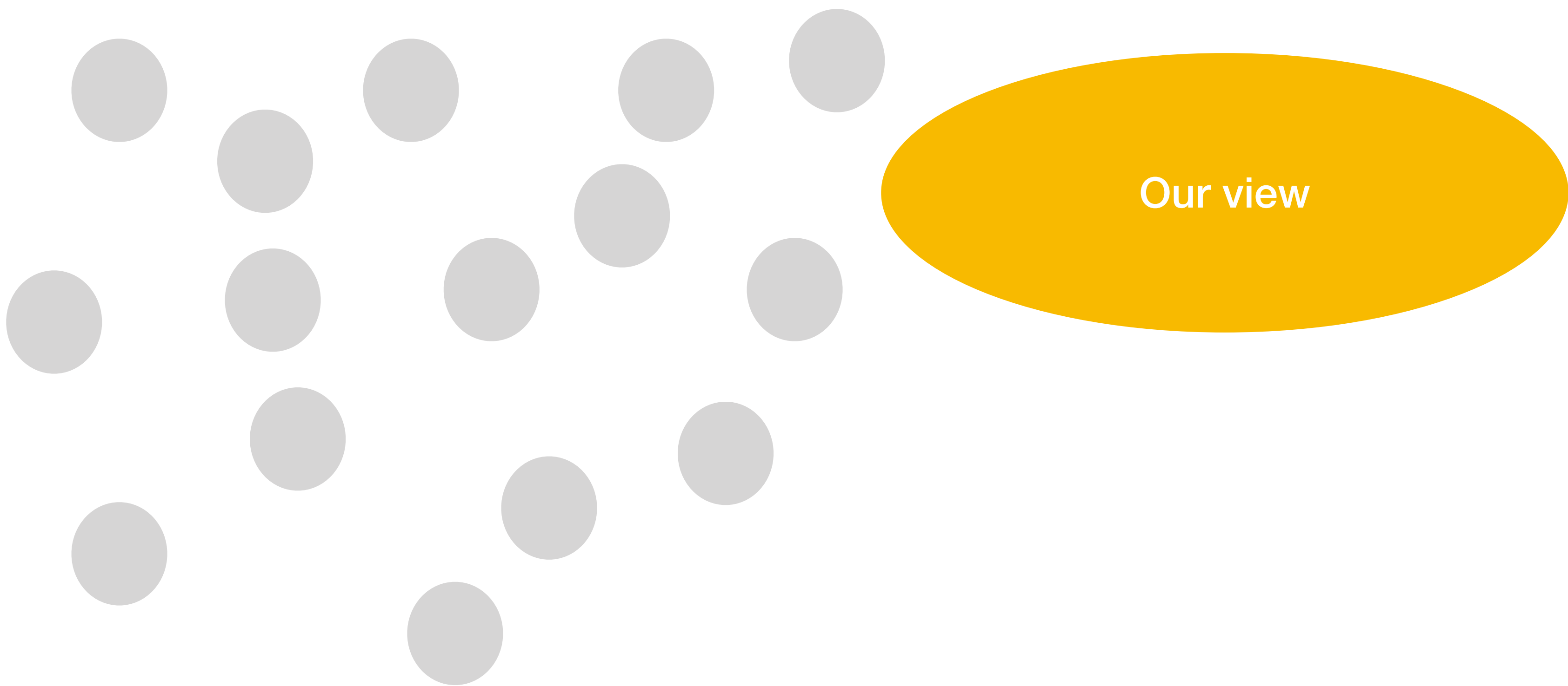
Developing Intuition

- When we have 15 people only:



Developing Intuition

- When we have 15 people only:



Developing Intuition

- When we have 15 people only:
- Suppose we ask two people to greet:
  - Case 1: they know each other

Developing Intuition

- When we have 15 people only:
- Suppose we ask two people to greet:
  - Case 1: they know each other
    - They both belong to the hidden community

Developing Intuition

- When we have 15 people only:
- Suppose we ask two people to greet:
  - Case 2: they do not know each other
    - At least one of them is outside the community

Developing Intuition

- When we have 15 people only:
- Suppose we ask two people to greet:
  - Case 2: they do not know each other
    - At least one of them is outside the community

Developing Intuition

- When we have 15 people only:
- Suppose we ask two people to greet:
  - Case 2: they do not know each other
    - At least one of them is outside the community

Developing Intuition

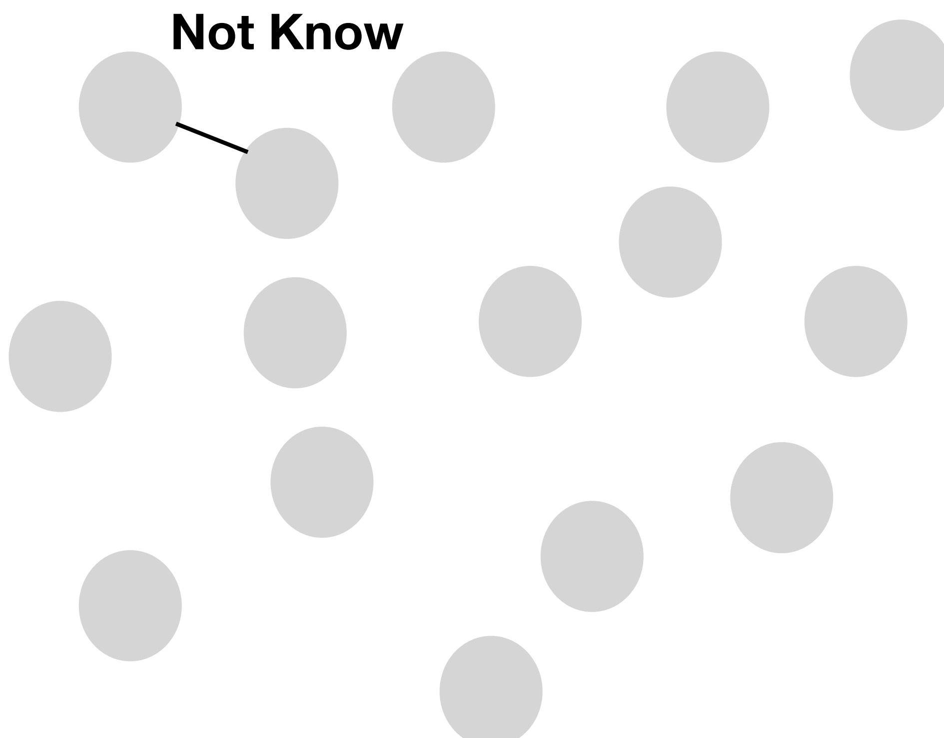
- When we have 15 people only:
- Suppose we ask two people to greet:
  - Case 2: they do not know each other
    - At least one of them is outside the community

Developing Intuition

- Simple solution: ask everyone to greet everyone!
- Whenever both sides know each other mark them both blue

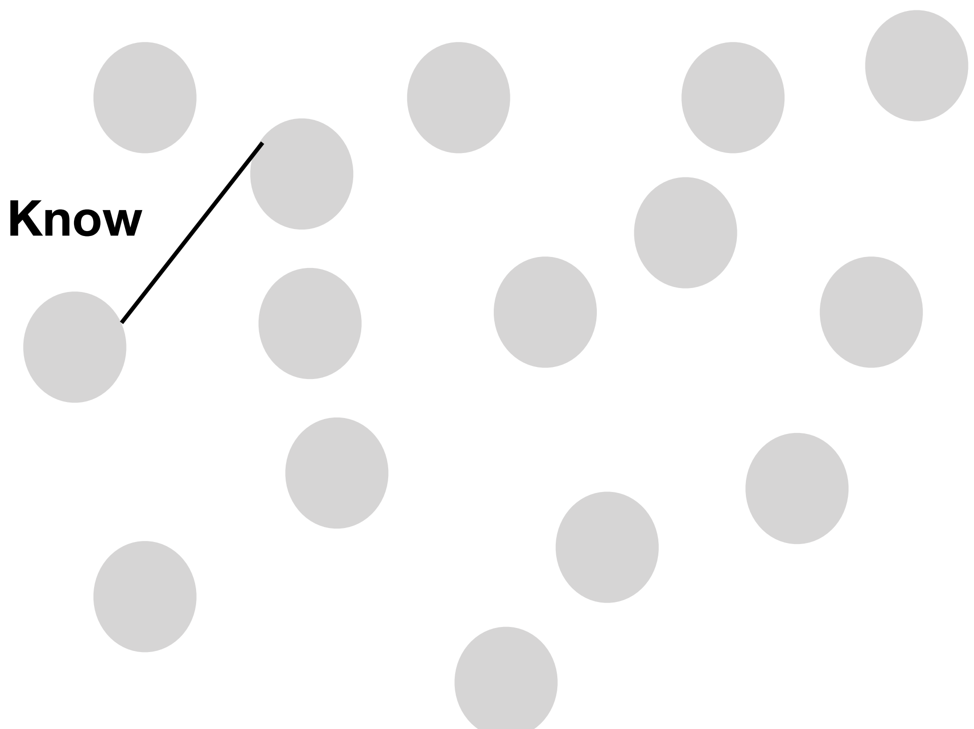
A Simple Solution

- Simple solution: ask everyone to greet everyone!
- Whenever both sides know each other mark them both blue



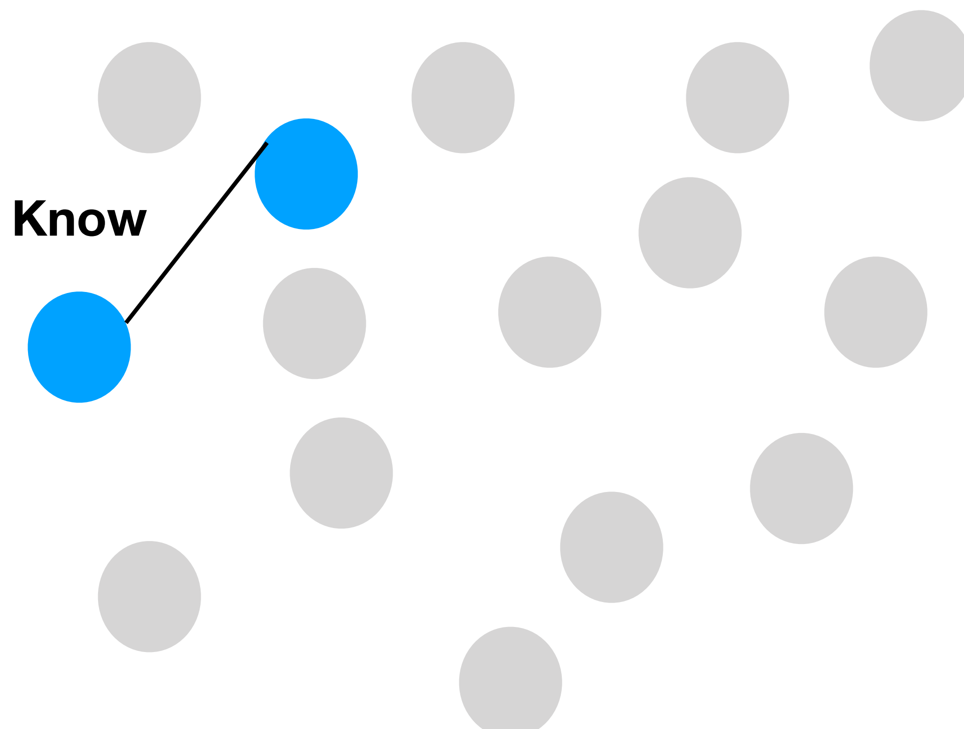
A Simple Solution

- Simple solution: ask everyone to greet everyone!
- Whenever both sides know each other mark them both blue



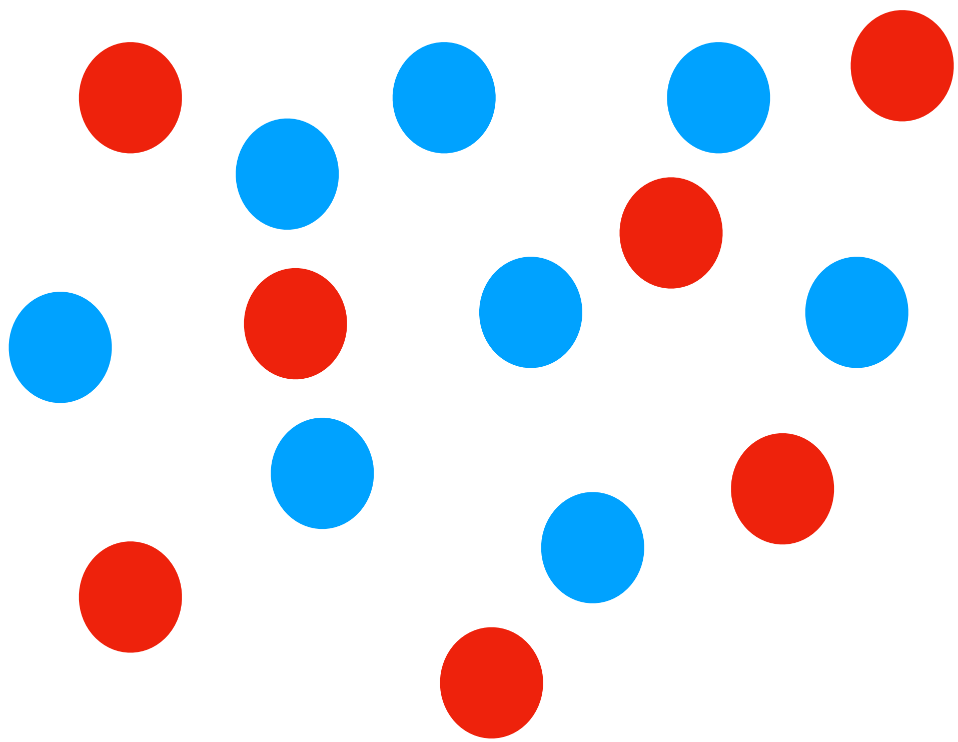
A Simple Solution

- Simple solution: ask everyone to greet everyone!
- Whenever both sides know each other mark them both blue



A Simple Solution

- Simple solution: ask everyone to greet everyone!
- Whenever both sides know each other mark them both blue



A Simple Solution

- Simple solution: ask everyone to greet everyone!
- Whenever both sides know each other mark them both blue
- Proof of correctness:
    1. We never mark any person blue if the person does not belong to the hidden community
    2. Every person of the hidden community will be marked blue when greeting them with another member
    3. The set of people marked blue is exactly the hidden community

A Simple Solution

- Simple solution: ask everyone to greet everyone!
- Whenever both sides know each other mark them both blue
- Proof of correctness:
- Efficiency analysis (=number of greetings):
  - All pairs of people greet each other
  - When we have 15 people: this is $\binom{15}{2} = \frac{15 \cdot 14}{2} = 105$
  - When we have n people: this is $\binom{n}{2} = \frac{n \cdot (n-1)}{2} = \Theta(n^2)$

Better Solution?

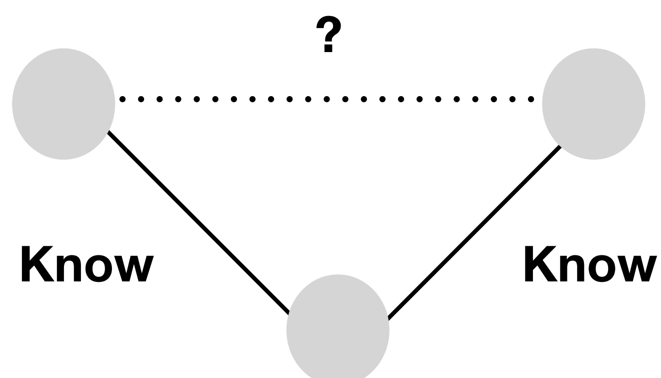
- Do we really need that many greetings?

Better Solution?

- Do we really need that many greetings?
- Some intuition why not:
  - Not all greetings seem necessary:

Better Solution?

- Do we really need that many greetings?
- Some intuition why not:
  - Not all greetings seem necessary:



Better Solution?

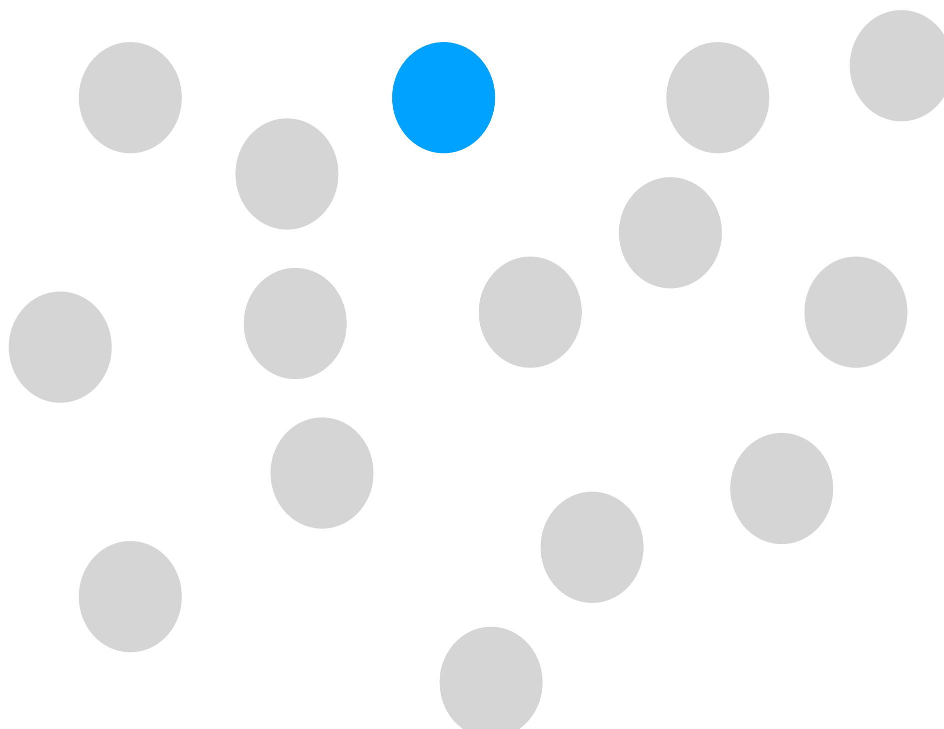
- Do we really need that many greetings?
- Some intuition why not:
  - Not all greetings seem necessary
  - We are not using full power of problem
    - Previous algorithm worked even we only had two people in the hidden community not the majority

Developing Intuition

- Break the problem into something simpler first:
- Can we find a single member of the hidden community?

Developing Intuition

- Break the problem into something simpler first:
- Can we find a single member of the hidden community?



Developing Intuition

- Break the problem into something simpler first:
- Can we find a single member of the hidden community?
- If we can do that, we can solve the problem with at most $n - 1$ more greetings

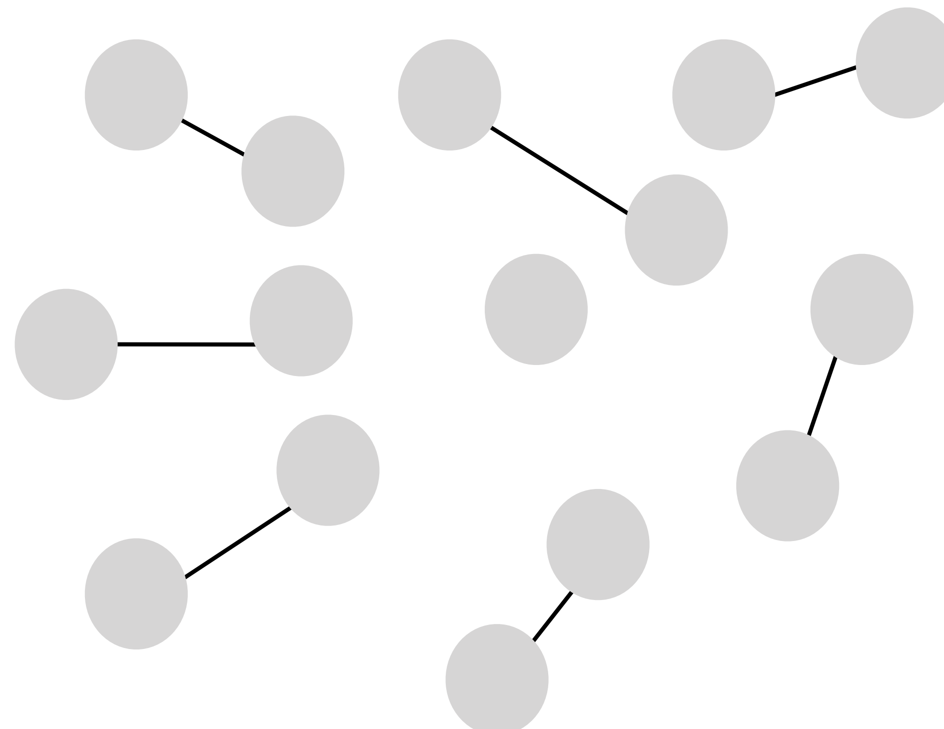
Developing Intuition

- Break the problem into something simpler first:
- Can we find a single member of the hidden community?
- If we can do that, we can solve the problem with at most $n - 1$ more greetings
- How to find a single member?

Finding a Single Member

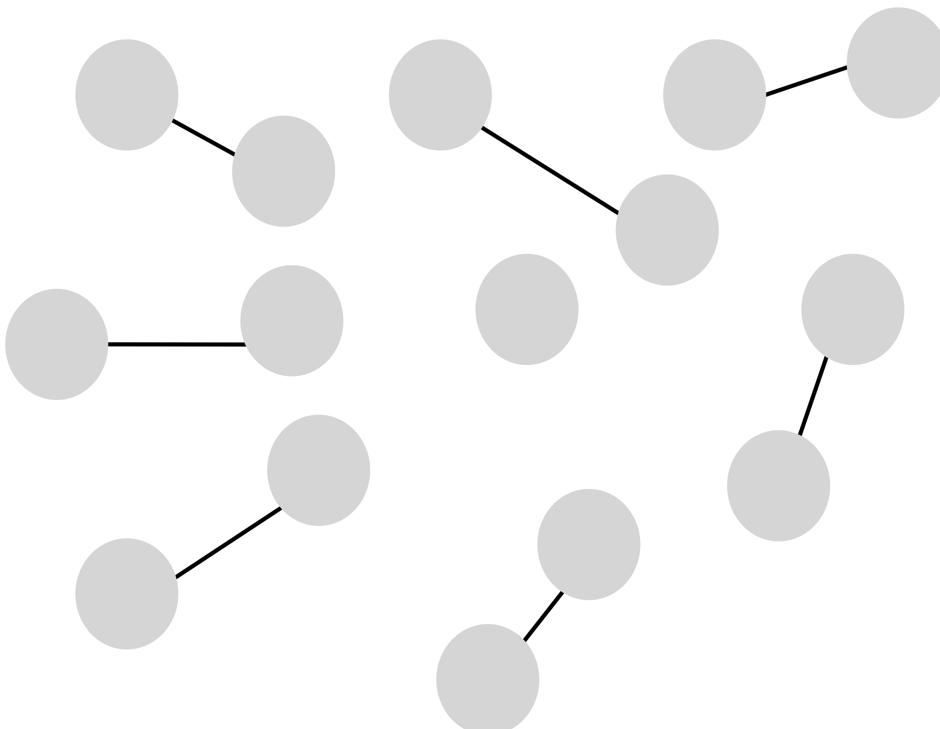
- Pair up people together into groups of size two with one extra

- Ask them to greet each other



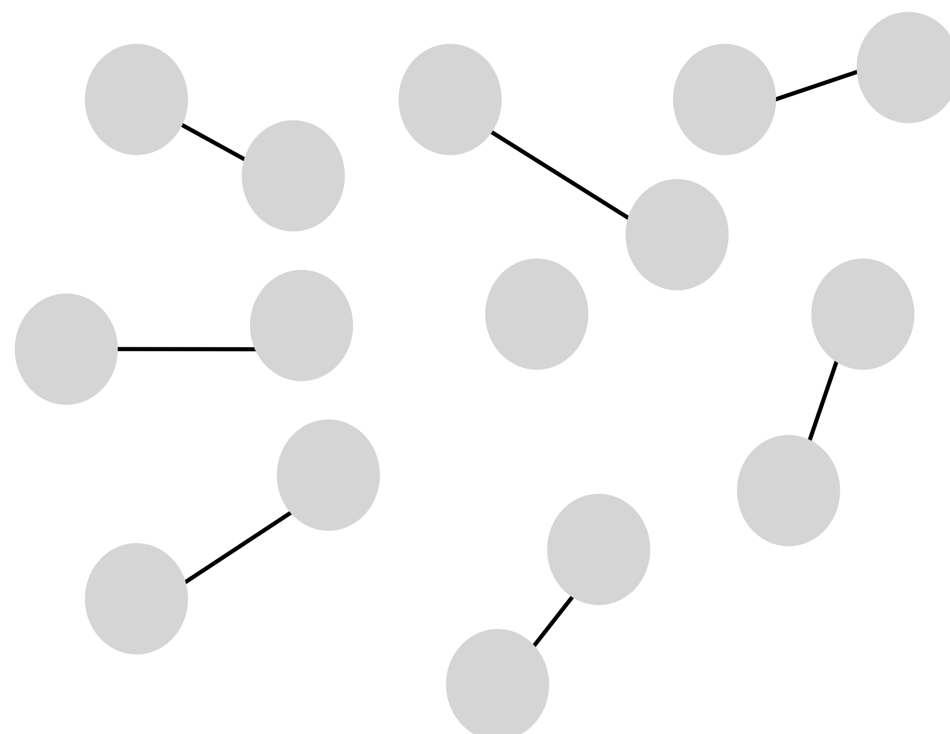
Finding a Single Member

- Pair up people together into groups of size two with one extra
- Ask them to greet each other
- Case 1: There is a pair that know each other
  - We found two members



Finding a Single Member

- Pair up people together into groups of size two with one extra
- Ask them to greet each other
- Case 1: There is a pair that know each other
- Case 2: No pair know each other
  - The extra person belongs to the hidden community



A Better Solution

- Algorithm:
  - Pair up people together into groups of size two with one extra; ask people in each group to greet
  - If there is a group that knows each other, mark both parties in the group blue
  - Otherwise mark the extra person blue
  - Pick a blue person and ask them to greet everyone else
  - Mark everyone that the blue person knows as blue
  - Return all blue persons are members of the hidden community

A Better Solution

- Proof of Correctness: Part I
  - Pair up people together into groups of size two with one extra; ask people in each group to greet
  - If there is a group that knows each other, mark both parties in the group blue
  - Otherwise mark the extra person blue
- If a group knows each other, both belong to community, thus marked blue correctly
- If no group know each other, every group has at least one non-member, so the extra person belongs to hidden community to preserve majority

A Better Solution

- Proof of Correctness: Part II
  - By part I, the blue person belongs to the hidden community
  - The set of people that know this person is exactly the people in hidden community, so algorithm returns correctly

- Pick a blue person and ask them to greet everyone else
- Mark everyone that the blue person knows as blue
- Return all blue persons are members of the hidden community

A Better Solution

- Efficiency Analysis (number of greetings)
  - Pair up people together into groups of size two with one extra; ask people in each group to greet
  - If there is a group that knows each other, mar... in ... blue
  - ...k the extra person blue
  - Pick a blue person and ask them to greet everyone else
  - Mark everyone that the blue person knows as blue
  - Return all blue persons are members of the hidden community

$\frac{n}{2}$ greetings

$n-1$ greetings

$\Theta(n)$ greetings

Mathematical Induction

Proofs

- Remember that perhaps the single most important step of algorithm design is to prove the correctness of the algorithm
  - Why the algorithm solves the given problem?
- This often involves proving mathematical statements for every possible input
  - These are often called universal statements
- Example:
  - A non-universal statement: $2^2 = 2 + 2$ but $3^2 \neq 3 + 3$
  - A universal statement: $(a+b)^2 = a^2 + b^2 + 2ab$ for **every** a,b.

Example

- Statement: For every integer $n \geq 1$, $\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$

Example

- Statement: For every integer $n \geq 1$, $\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$

- How do we prove this statement?

Example

- Statement: For every integer $n \geq 1$, $\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$
- How do we prove this statement?
- We can try to "prove" it using a calculator for small values of n:

n	LHS	RHS
1	1	$\frac{1 \cdot 2}{2} = 2$
2	$1 + 2 = 3$	$\frac{2 \cdot 3}{2} = 3$
3	$1 + 2 + 3 = 6$	$\frac{3 \cdot 4}{2} = 6$
4	$1 + 2 + 3 + 4 = 10$	$\frac{4 \cdot 5}{2} = 10$

Example

- Statement: For every integer $n \geq 1$, $\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$
- How do we prove this statement?
- We can try to “prove” it using a calculator for small values of n
- But at some point we have to give up: there is no way we can test infinite numbers (for that matter even for n, say, $n = 10^{20}$)

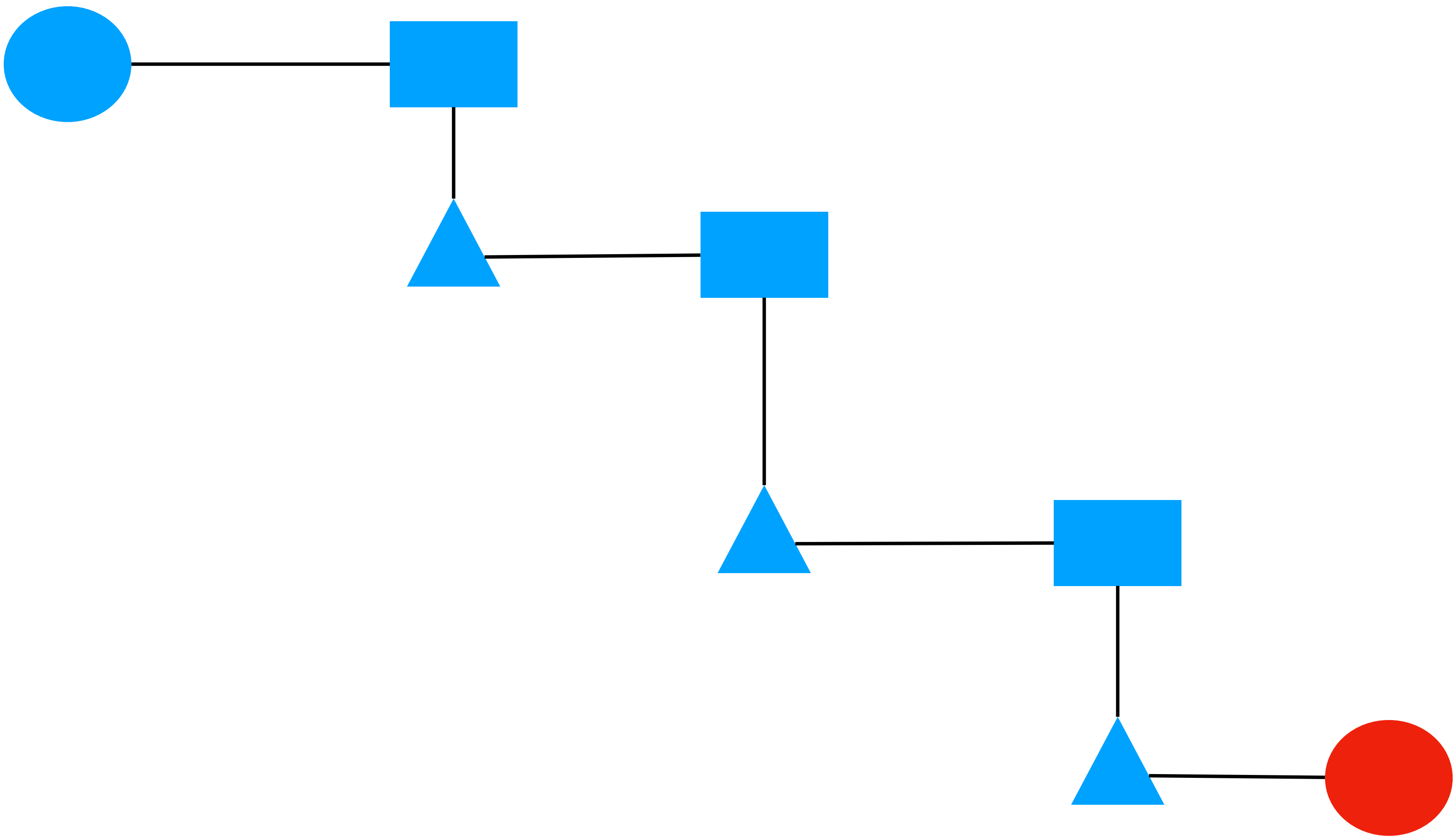
Example

- Statement: For every integer $n \geq 1$, $\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$
- How do we prove this statement?
- We can try to “prove” it using a calculator for small values of n
- But at some point we have to give up: there is no way we can test infinite numbers (for that matter even for n, say, $n = 10^{20}$)
- We need a tool for proving statements about numbers that go to infinity

Example

- Statement: For every integer $n \geq 1$, $\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$
- How do we prove this statement?
- We can try to “prove” it using a calculator for small values of n
- But at some point we have to give up: there is no way we can test infinite numbers (for that matter even for n, say, $n = 10^{20}$)
- We need a tool for proving statements about numbers that go to infinity: **Mathematical Induction**.

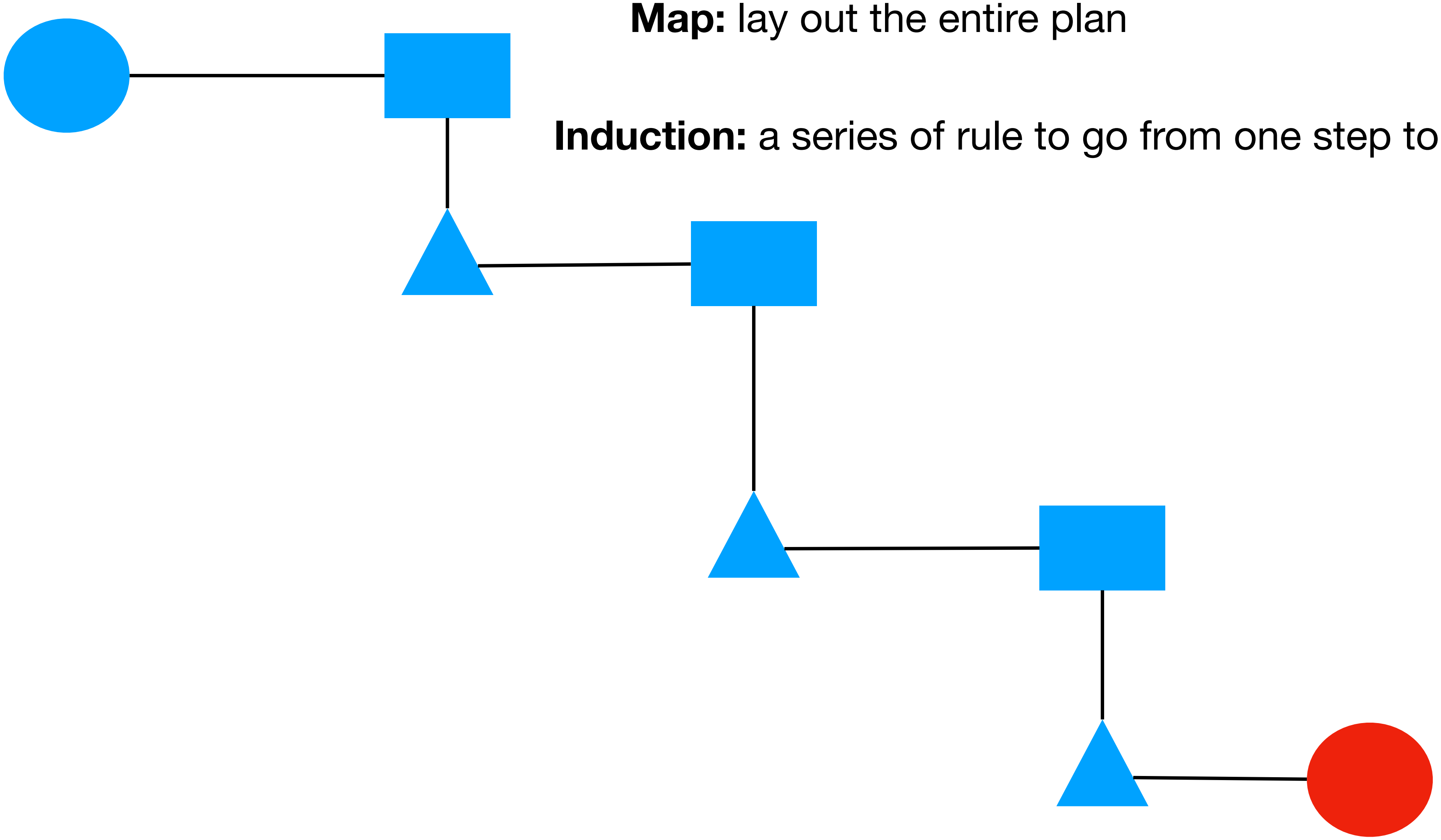
What is induction about?



What is induction about?

Map: lay out the entire plan

Induction: a series of rule to go from one step to next

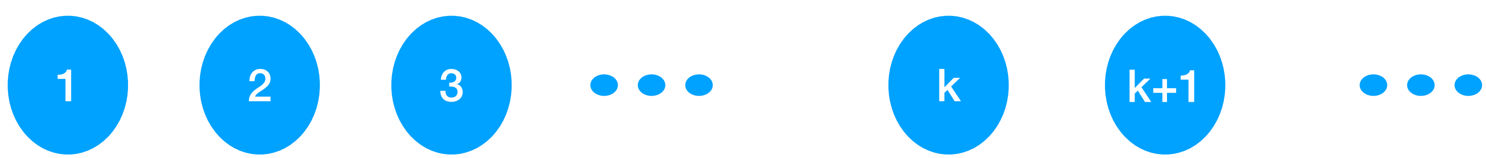


Induction

- Base case: prove that the statement is true for $n = 1$
- Induction step: prove that, for any integer $k \geq 1,$ if induction hypothesis is true for $n = k$ then it is also true for $n = k + 1$

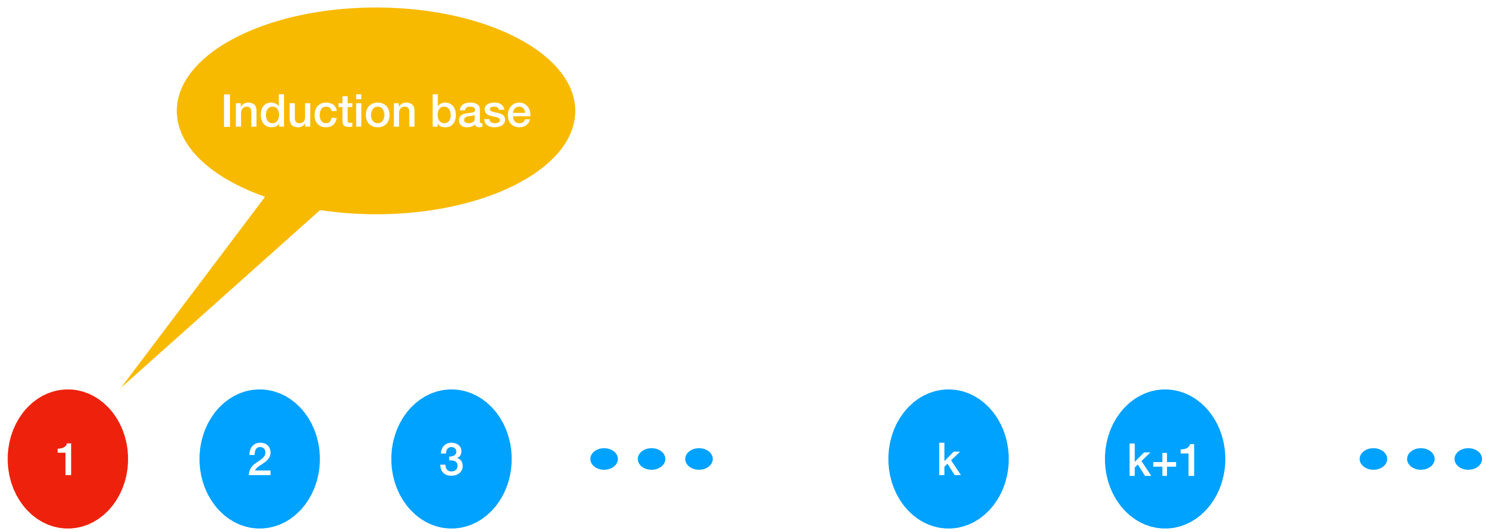
Induction

- Base case: prove that the statement is true for $n = 1$
- Induction step: prove that, for any integer $k \geq 1$, if induction hypothesis is true for $n = k$ then it is also true for $n = k + 1$



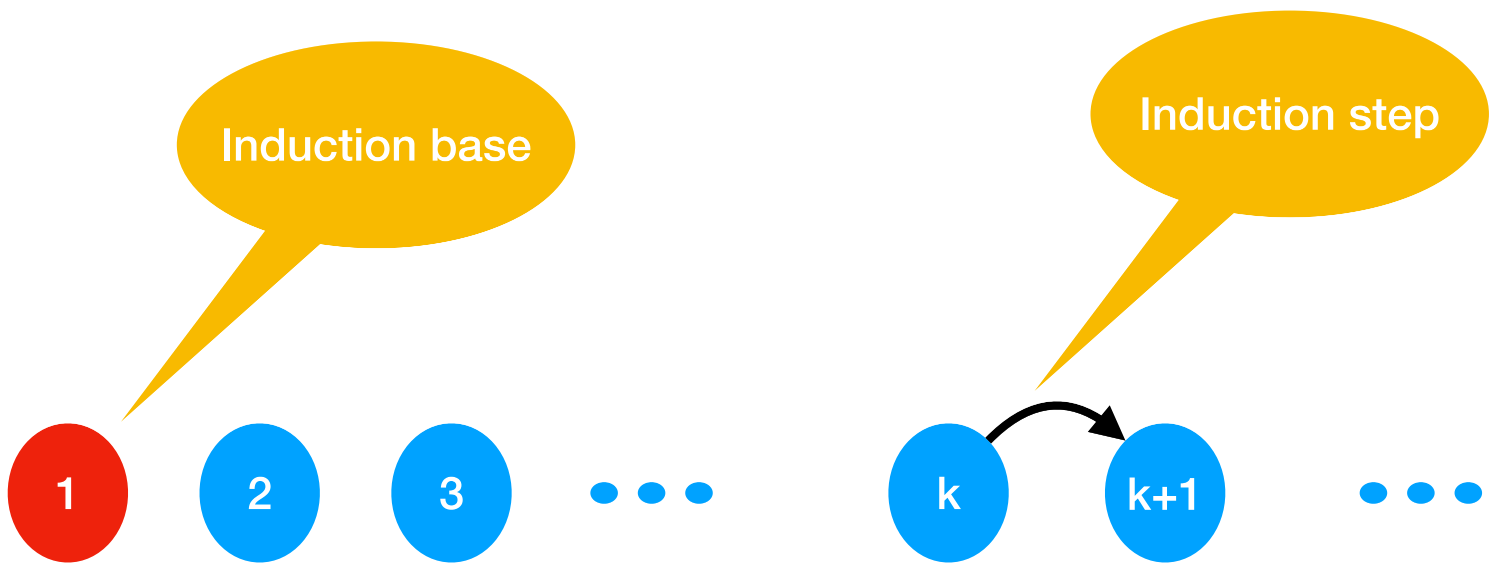
Induction

- Base case: prove that the statement is true for $n = 1$
- Induction step: prove that, for any integer $k \geq 1$, if induction hypothesis is true for $n = k$ then it is also true for $n = k + 1$



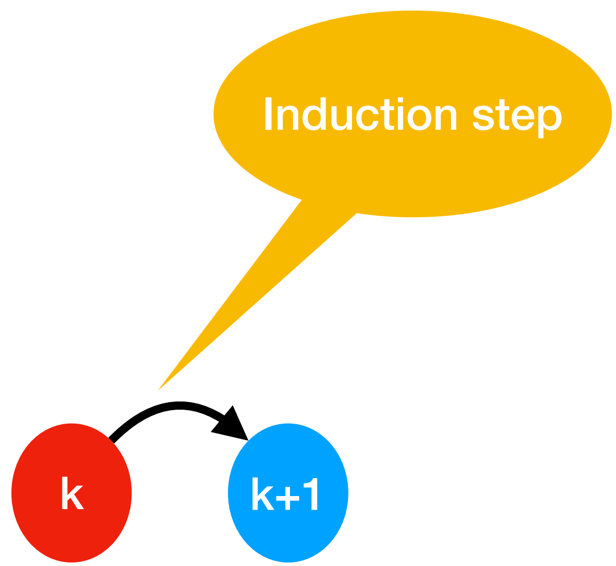
Induction

- Base case: prove that the statement is true for $n = 1$
- Induction step: prove that, for any integer $k \geq 1$, if induction hypothesis is true for $n = k$ then it is also true for $n = k + 1$



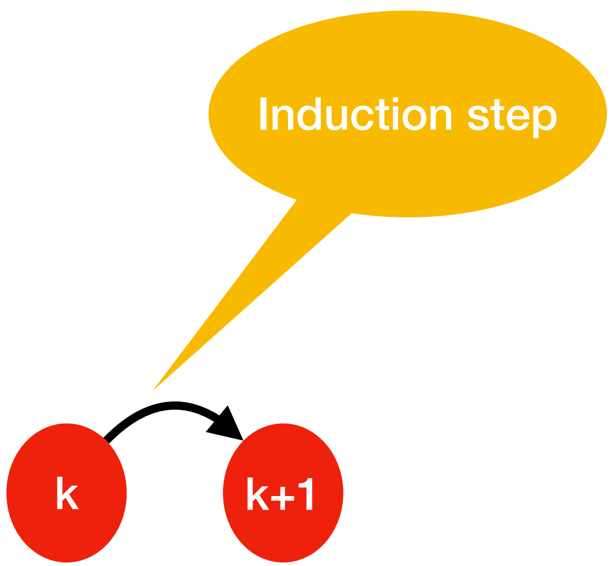
Induction

- Base case: prove that the statement is true for $n = 1$
- Induction step: prove that, for any integer $k \geq 1$, if induction hypothesis is true for $n = k$ then it is also true for $n = k + 1$



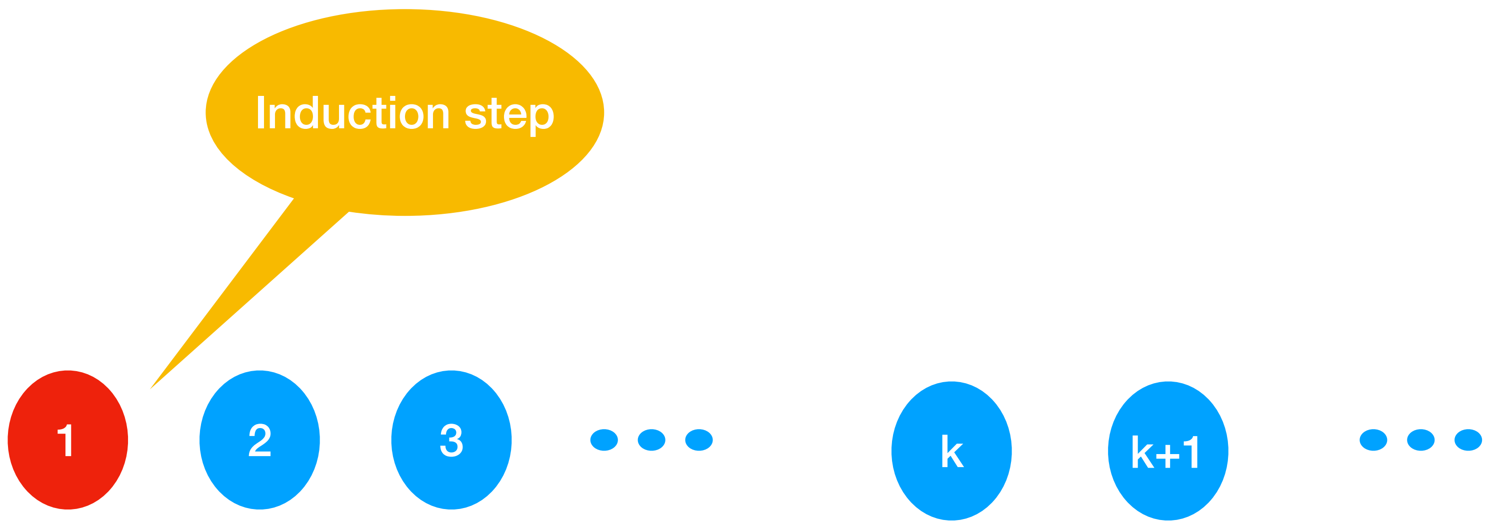
Induction

- Base case: prove that the statement is true for $n = 1$
- Induction step: prove that, for any integer $k \geq 1$, if induction hypothesis is true for $n = k$ then it is also true for $n = k + 1$



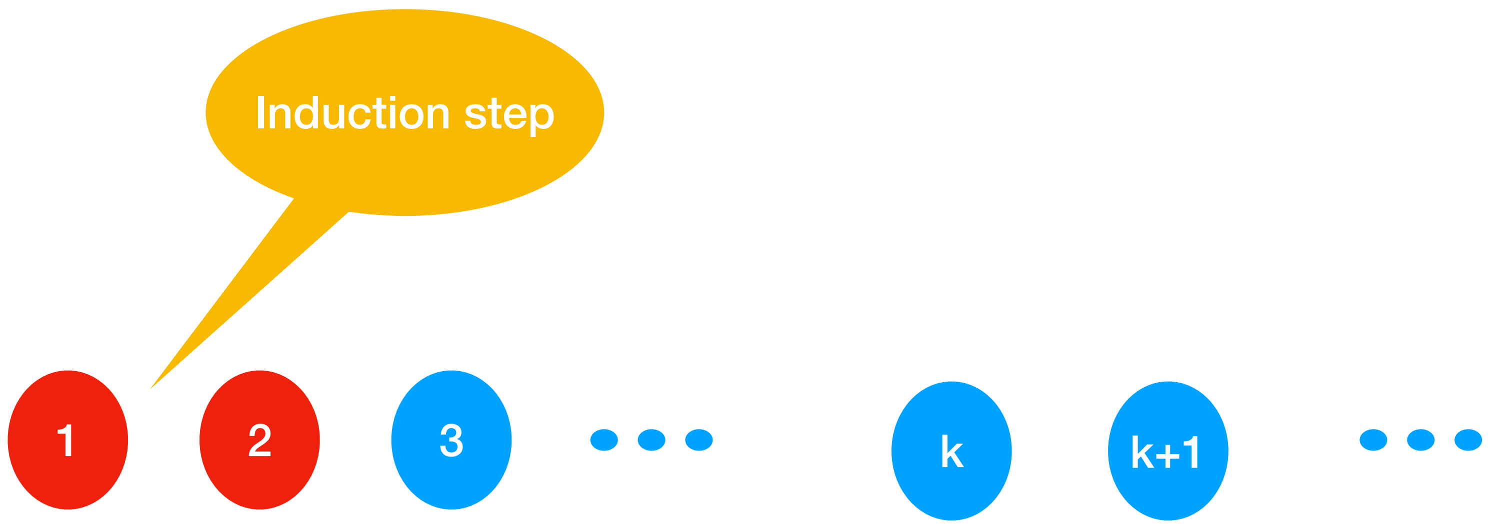
Induction

- Base case: prove that the statement is true for $n = 1$
- Induction step: prove that, for any integer $k \geq 1$, if induction hypothesis is true for $n = k$ then it is also true for $n = k + 1$



Induction

- Base case: prove that the statement is true for $n = 1$
- Induction step: prove that, for any integer $k \geq 1$, if induction hypothesis is true for $n = k$ then it is also true for $n = k + 1$

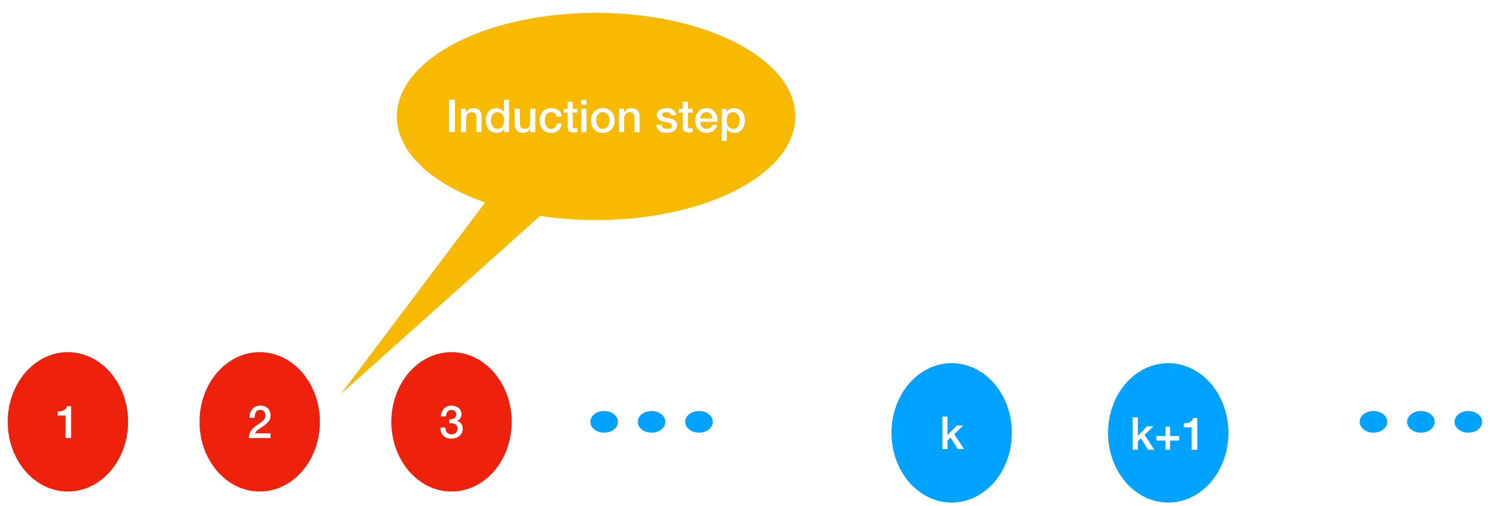


Induction

- Base case: prove that the statement is true for $n = 1$
- Induction step: prove that, for any integer $k \geq 1$, if induction hypothesis is true for $n = k$ then it is also true for $n = k + 1$

Induction

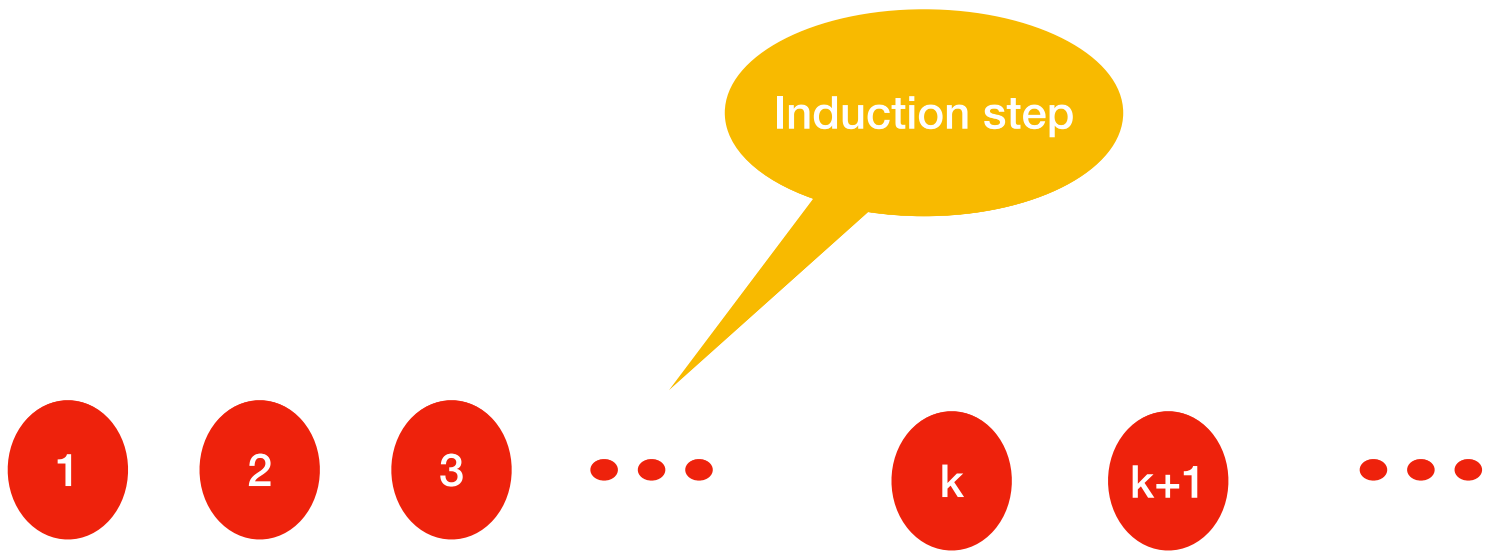
- Base case: prove that the statement is true for $n = 1$
- Induction step: prove that, for any integer $k \geq 1$, if induction hypothesis is true for $n = k$ then it is also true for $n = k + 1$



Induction

- Base case: prove that the statement is true for $n = 1$

- Induction step: prove that, for any integer $k \geq 1$, if induction hypothesis is true for $n = k$ then it is also true for $n = k + 1$



Example

- For every integer $n \geq 1$, $\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$

- Proof:
  - **Induction base:** statement is true for n=1 because LHS is 1 and RHS is $\frac{1 \cdot (1+1)}{2} = 1$

Example

- For every integer $n \geq 1$, $\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$
- Proof:
  - **Induction step:** Suppose the statement is true for $n = k$; we prove it for $n = k + 1$
  - $\sum_{i=1}^{k+1} i = (\sum_{i=1}^{k} i) + (k+1)$
  - By induction hypothesis for $n = k$, $\sum_{i=1}^{k} i = \frac{k \cdot (k+1)}{2}$
  - So $\sum_{i=1}^{k+1} i = \frac{k \cdot (k+1)}{2} + (k+1)$

Example

- For every integer $n \geq 1$, $\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$
- Proof:
  - **Induction step:** Suppose the statement is true for $n = k$; we prove it for $n = k + 1$
  - $\sum_{i=1}^{k+1} i = (\sum_{i=1}^{k} i) + (k+1)$
  - By induction hypothesis for $n = k$, $\sum_{i=1}^{k} i = \frac{k \cdot (k+1)}{2}$
  - So $\sum_{i=1}^{k+1} i = \frac{k \cdot (k+1)}{2} + (k+1) = \frac{(k+2) \cdot (k+1)}{2} = \frac{(k+1) \cdot (k+2)}{2}$

Example

- For every integer $n \geq 1$, $\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$
- Proof:
  - Both induction base and step hold.
  - Thus, by induction the statement holds.

Example

- For every integer $n \geq 1$, $\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$

- Proof:
  - Both induction base and step hold.
  - Thus, by induction the statement holds.

- And this is literally all induction is!

Proof of Correctness of Algorithms using Induction

Example: Finding Maximum

- An algorithm for finding maximum in array A[1:n]:

    1. Let candidate MAX be the element A[1]

    2. Iterate over elements A[i] for i = 1 to n: if A[i] > MAX, then let MAX=A[i].

    3. Output MAX as the maximum element of the array

Proof of Correctness

- Statement 1: Algorithm is correct = At the end of the algorithm MAX is equal to maximum of array A[1:n]

- Statement 2: After every iteration i of the for-loop, MAX is equal to maximum of array A[1:i].

Proof of Correctness

- Statement 2: After every iteration i of the for-loop, MAX is equal to maximum of array A[1:i].

- Proof by induction

  - Induction base: when i=1, MAX = A[1] and thus maximum of array A[1:1].

  - Induction step: Suppose MAX is maximum of array A[1:j]; we prove for A[1:j+1]

    - At iteration i=j+1, we set MAX to be maximum(MAX,A[j+1]).

    - Maximum of array A[1:j+1] is maximum(max of A[1:j] , A[j+1])

    - So MAX is maximum of A[1:j+1].

    - By induction, the statement is true.

Recursion and Recursive Algorithms

Recursion

- Recursion is “algorithmic induction”
  - Induction: if you can prove it for n=k, you can also prove it for n=k+1
  - Recursion: if you can solve it for n=k, you can also solve it for n=k+1
- Recursive algorithm:
  - If the problem is simple enough, solve it directly
  - Otherwise, “break” the problem into smaller pieces, solve each one using the same algorithm

Recursion

- Recursion is “algorithmic induction”
  - Induction: if you can prove it fo… …e it for n=k+1
  - Recursion: if you can solve it for … …solve it for n=k+1

Base case

- Recursive algorithm:
  - If the problem is simple enough, solve it directly
  - Otherwise, “break” the problem into smaller pieces, solve each one using the same algorithm

Recursion

- Recursion is “algorithmic induction”
  - Induction: if you can prove it for n=k, you can also prove it for n=k+1
  - Recursion: if you can solve it for n=k, you can also solve it for n=k+1
- Recursive algorithm:
  - If the problem is simple enough, solve it directly
  - Otherwise, “break” the problem into **smaller** pieces, solve each one using the same algorithm

Example: Finding Maximum

- Algorithm **Find-Max** on an array A[1:n]:
  - If n=1, return A[1]
  - Let temp = **Find-Max**(A[1:n-1]). Return max of temp and A[n].

Example: Finding Maximum

Base case

- Algorithm **Find-Max** on an array A[1
  - If n=1, return A[1]
  - Let temp = **Find-Max**(A[1:n-1]). Return max of temp and A[n].

Example: Finding Maximum

- Algorithm **Find-Max** on an array A[1:n]:
  - If n=1, return A[1]
  - Let temp = **Find-Max**(A[1:n-1]). Return max of temp and A[n].
- Proof of correctness? postponed
- Runtime analysis?

Example: Finding Maximum

- Runtime analysis?
  - Define the function T(n) as the worst-case runtime of **Find-Max** on an array of length n
  - We know
    - $T(1) = \Theta(1)$
    - $T(n) \leq T(n-1) + \Theta(1)$
  - Can we write T(n) in a more familiar way?
  - $T(n) \leq T(n-1) + \Theta(1) \leq T(n-2) + \Theta(1) + \Theta(1) \leq \cdots \leq \sum_{i=1}^{n} \Theta(1) = \Theta(n)$

Example: Finding Maximum

- Runtime analysis?
  - Define the function T(n) as the worst-case runtime of **Find-Max** on an array of length n
  - We know
    - $T(1) = \Theta(1)$
    - $T(n) \leq T(n-1) + \Theta(1)$

This is called a recurrence or a recurrence formula

  - Can we write T(n) in a more familiar way?
  - $T(n) \leq T(n-1) + \Theta(1) \leq T(n-2) + \Theta(1) + \Theta(1) \leq \cdots \leq \sum_{i=1}^{n} \Theta(1) = \Theta(n)$

Example: Finding Maximum

- Runtime analysis?
  - Define the function T(n) as the worst-case runtime of **Find-Max** on an array of length n
  - We know
    - $T(1) = \Theta(1)$
    - $T(n) \leq T(n-1) +$
  - Can we write T(n) i a more familiar way?
  - $T(n) \leq T(n-1) + \Theta(1) \leq T(n-2) + \Theta(1) + \Theta(1) \leq \cdots \leq \sum_{i=1}^{n} \Theta(1) = \Theta(n)$

We will review solving recurrences more carefully next week

Example: Binary Search

- Algorithm **Binary-Search** on a sorted array A[1:n] and element x:

Example: Binary Search

- Algorithm **Binary-Search** on a <u>sorted</u> array A[1:n] and element x:

Does x belong to A?

Example: Binary Search

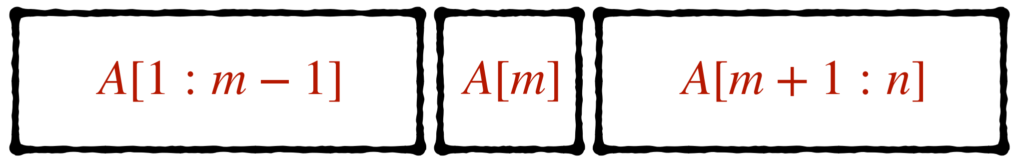
- Algorithm **Binary-Search** on a sorted array A[1:n] and element x:
- **Binary-Search**(A[1:n],x):
  - If $n=0$, return `NO'.
  - Let $m = \lfloor \frac{n}{2} \rfloor$:
    - If A[m]=x, return `Yes'
    - If A[m] > x, return **Binary-Search**(A[1:m-1],x)
    - If A[m] < x, return **Binary-Search**(A[m+1:n],x)

Proof of Correctness

- Statement: **Binary-Search** works correctly.
- Proof by induction:
  - Base case: $n = 0$
    - An empty array contains no element
  - Induction step: Suppose the statement is true for all arrays of length $n \leq k$ and we prove it for $n = k + 1$

Proof of Correctness

- Induction step: Suppose the statement is true for all arrays of length $n \leq k$ and we prove it for $n = k + 1$



- If A[m]=x, answer is correct
- If A[m]>x: then all of A[m+1:n] > x also by sortedness
- So x can only belong to A[1:m-1]
- By induction hypothesis the answer is correct on A[1:m-1]
- The A[m] < x case is symmetric

Runtime Analysis

- Define T(n) as the worst-case runtime of **Binary-Search** on an array of length n
- We know
  - $T(1) = \Theta(1)$
  - $T(n) = T(n/2) + \Theta(1)$
- We can write T(n) in a more friendly way as:
- $T(n) \leq T(n/2) + \Theta(1) \leq T(n/4) + \Theta(1) + \Theta(1) \leq \cdots \leq \underbrace{\Theta(1) + \cdots + \Theta(1)}_{\log n} = \Theta(\log n)$

Summary of Recursion

- Recursive algorithms:
  - Break the problem into smaller pieces, call the algorithm on each smaller piece recursively
- Proof of correctness: Induction
- Runtime analysis: Writing a recurrence formula and solving it

Divide and Conquer

Divide and Conquer

- A simple family of recursive algorithms:
  - Divide the input into two or more instances of the same problem
  - Solve the problem recursively on each part
  - Combine the answers on the parts to get the final answer
  - If an instance is small enough, solve it by brute force instead
- Proof of Correctness: Induction
- Runtime analysis: Recurrence

Sorting

- Sorting:
  - Problem: Given an array of integers $A[1:n]$, sort them in increasing (non-decreasing) order of their values
  - Algorithm:
    - Insertion sort, selection sort, bubble sort, …
    - Merge sort, quick sort, heap sort, …
    - Count sort, radix sort, bucket sort, …

Sorting

- Sorting:
  - Problem: Given an array of integers $A[1:n]$, sort them in increasing (non-decreasing) order of their values
  - Algorithm:
    - Insertion sort, selection sort, bubble sort, …
    - **Merge sort**, quick sort, heap sort, …
    - Count sort, radix sort, bucket sort, …

Merge Sort

- **Merge-Sort**(A[1:n])

1. If n=0 or 1, return A.
2. Run **Merge-Sort**($A[1:\frac{n}{2}]$) and **Merge-Sort**($A[\frac{n}{2}+1,n]$)
3. Combine the first and second half using the **Merge** algorithm

Merge Sort

- **Merge-Sort**(A[1:n])

  1. If n=0 or 1, return A.

  2. Run **Merge-Sort**($A[1:\frac{n}{2}]$) and **Merge-Sort**($A[\frac{n}{2}+1,n]$)

  3. Combine the first and second half using the **Merge** algorithm

- **Merge**(B[1:m],C[1:k])

  - What is the problem it solves?

  - Given two sorted arrays, return the sorted array D[1:m+k] of their combination

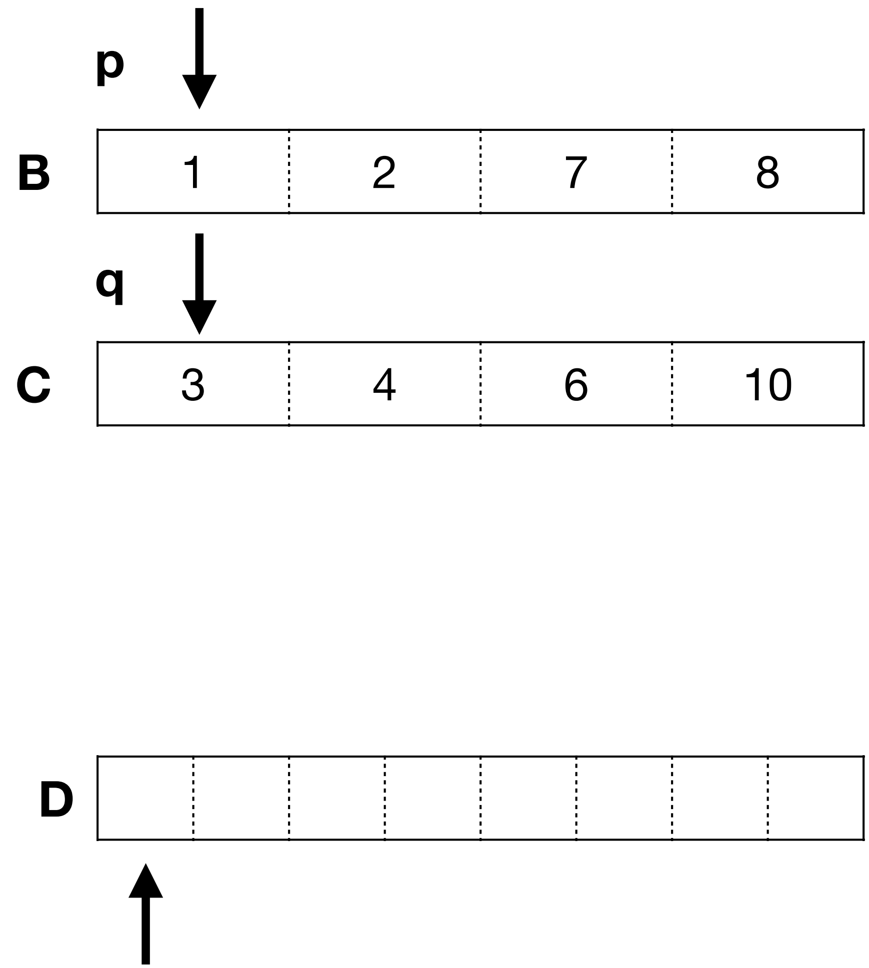
Merge

- **Merge**(B[1:m],C[1:k])
  - Create an empty array D[1:m+k] with pointers p=1 and q=1
  - Add $+\infty$ to the end of both arrays B and C
  - For i=1 to m+k:
    - If B[p] < C[q],
      - let D[i] = B[p] and set p = p+1
    - Else let D[i] = C[q] and set q=q+1
  - Return D

Merge — Example

Merge(B[1:m],C[1:k])

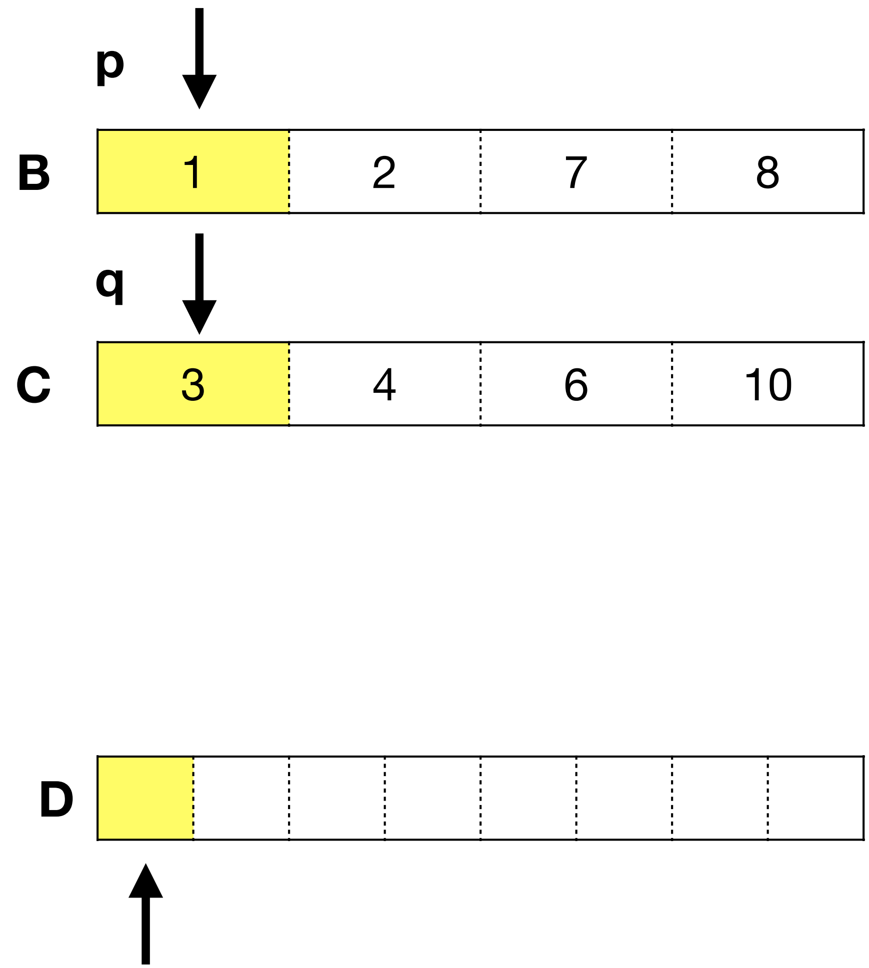
1. Create an empty array D[1:m+k] with pointers p=1 and q=1
2. Add $+\infty$ to the end of both arrays B and C
3. For i=1 to m+k:
   - If B[p] < C[q],
     - let D[i] = B[p] and set p = p+1
   - Else let D[i] = C[q] and set q=q+1
4. Return D



Merge — Example

Merge(B[1:m],C[1:k])

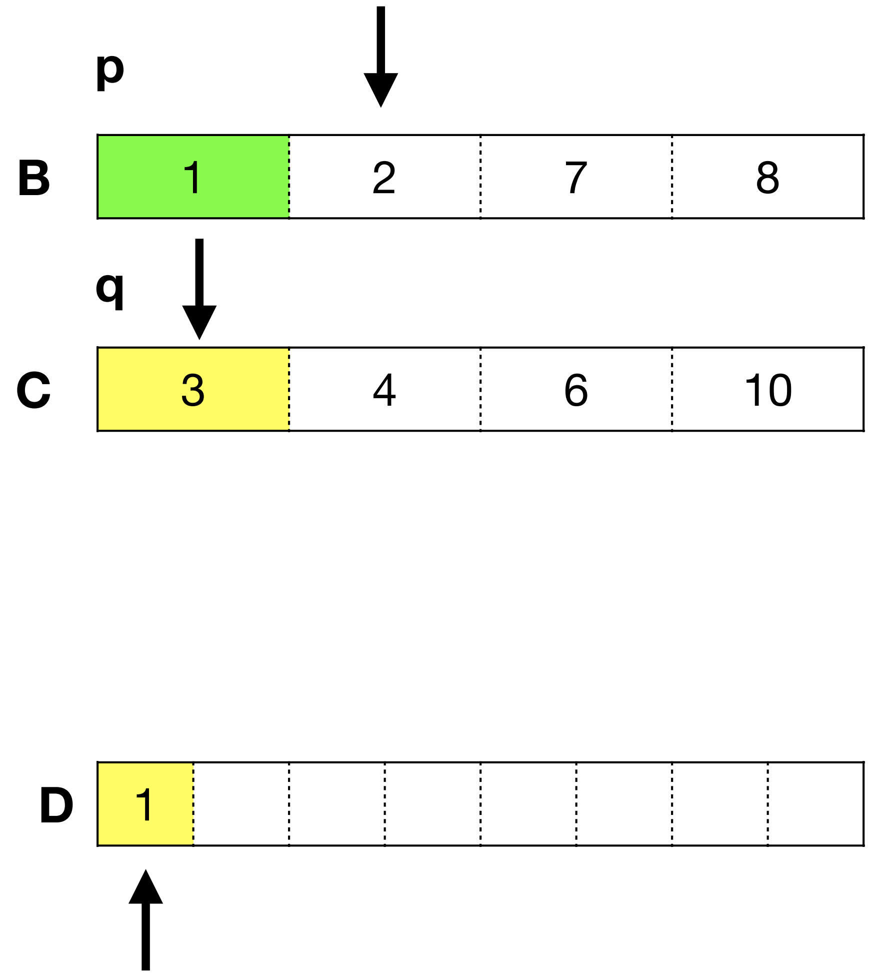
1. Create an empty array D[1:m+k] with pointers p=1 and q=1
2. Add $+\infty$ to the end of both arrays B and C
3. For i=1 to m+k:
    - If B[p] < C[q],
        - let D[i] = B[p] and set p = p+1
    - Else let D[i] = C[q] and set q=q+1
4. Return D



Merge — Example

Merge(B[1:m],C[1:k])

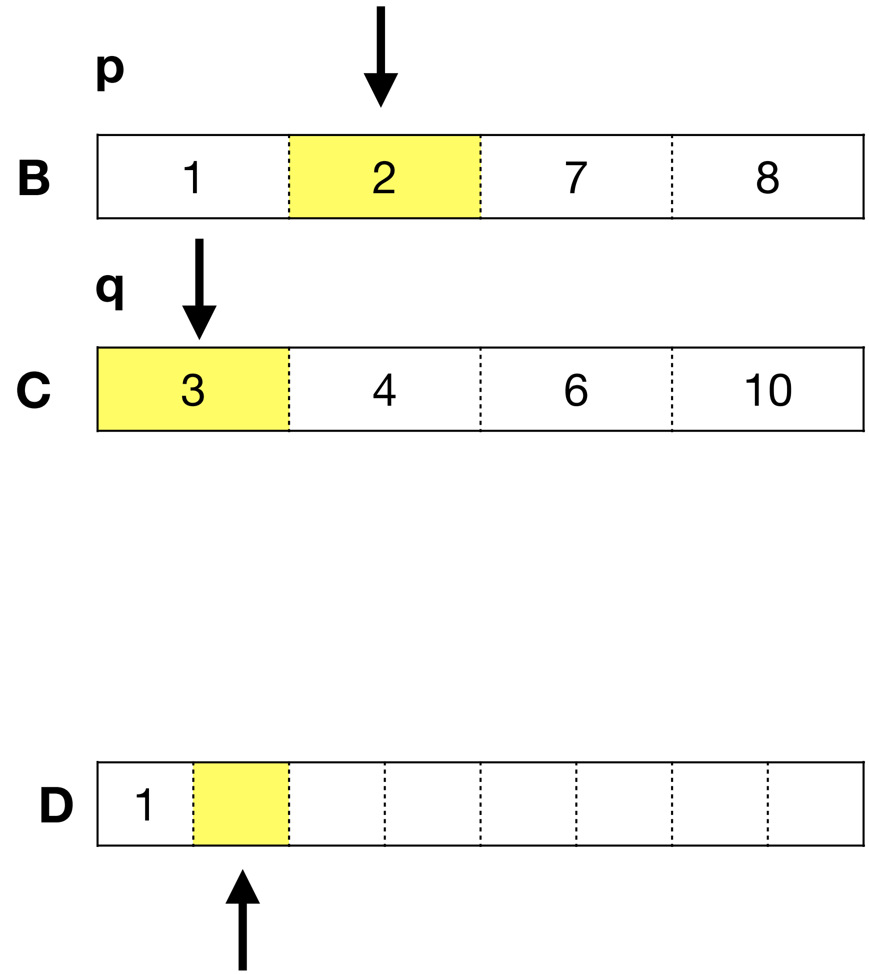
1. Create an empty array D[1:m+k] with pointers p=1 and q=1
2. Add $+\infty$ to the end of both arrays B and C
3. For i=1 to m+k:
   - If B[p] < C[q],
     - let D[i] = B[p] and set p = p+1
   - Else let D[i] = C[q] and set q=q+1
4. Return D



Merge — Example

Merge(B[1:m],C[1:k])

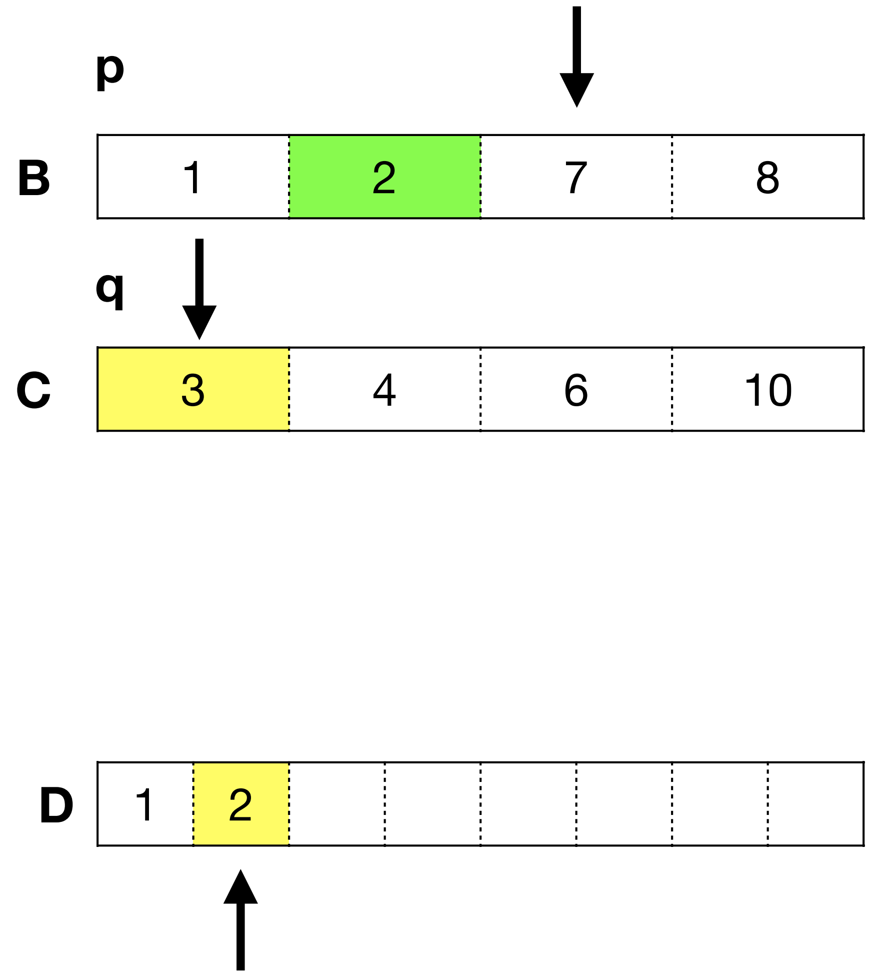
1. Create an empty array D[1:m+k] with pointers p=1 and q=1
2. Add $+\infty$ to the end of both arrays B and C
3. For i=1 to m+k:
   - If B[p] < C[q],
     - let D[i] = B[p] and set p = p+1
   - Else let D[i] = C[q] and set q=q+1
4. Return D



Merge — Example

Merge(B[1:m],C[1:k])

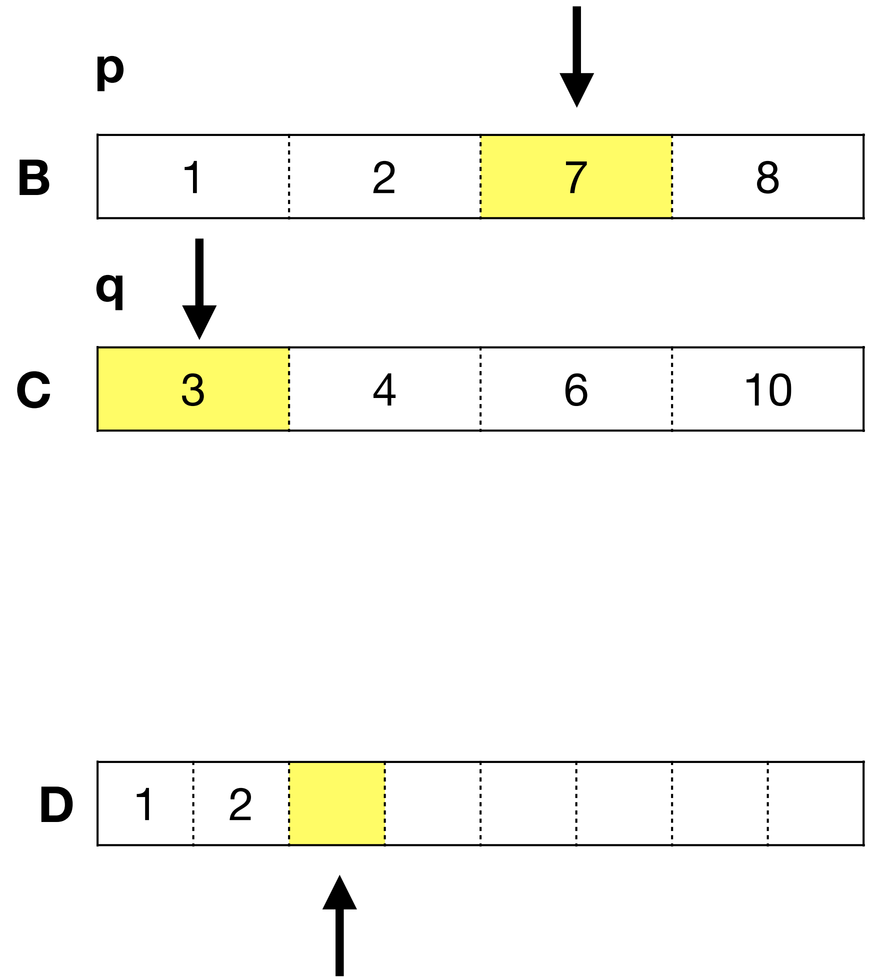
1. Create an empty array D[1:m+k] with pointers p=1 and q=1
2. Add $+\infty$ to the end of both arrays B and C
3. For i=1 to m+k:
   - If B[p] < C[q],
     - let D[i] = B[p] and set p = p+1
   - Else let D[i] = C[q] and set q=q+1
4. Return D



Merge — Example

Merge(B[1:m],C[1:k])

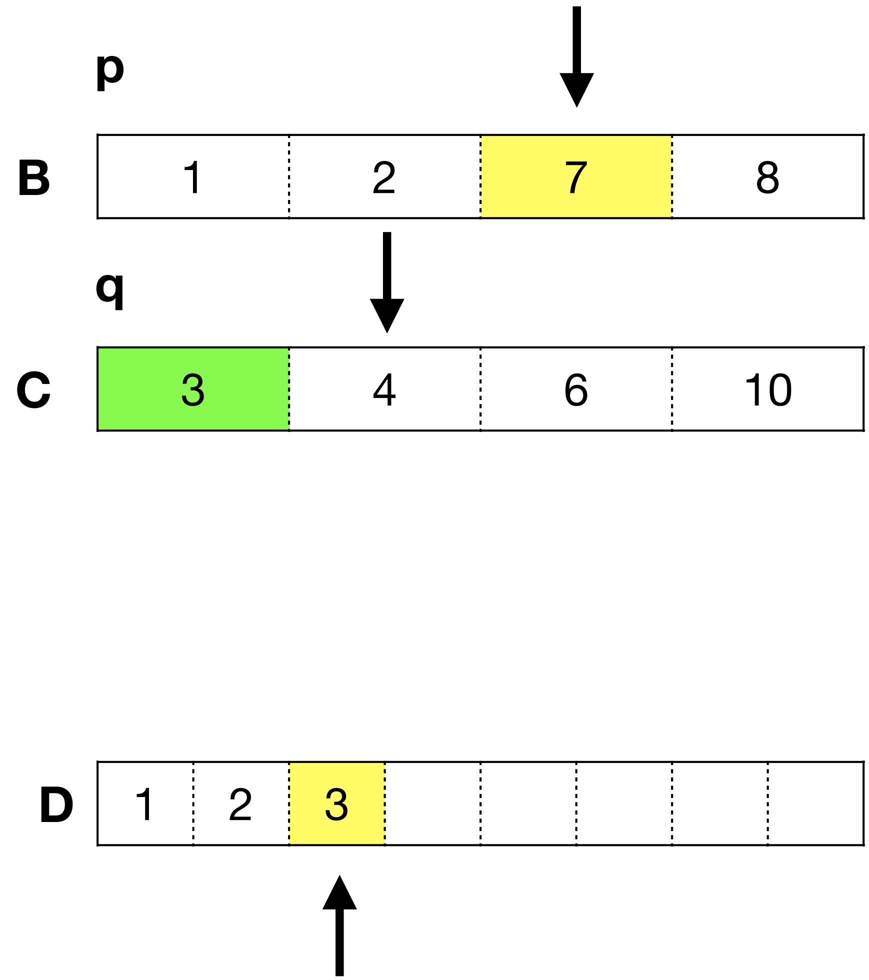
1. Create an empty array D[1:m+k] with pointers p=1 and q=1
2. Add $+\infty$ to the end of both arrays B and C
3. For i=1 to m+k:
    - If B[p] < C[q],
        - let D[i] = B[p] and set p = p+1
    - Else let D[i] = C[q] and set q=q+1
4. Return D



Merge — Example

Merge(B[1:m],C[1:k])

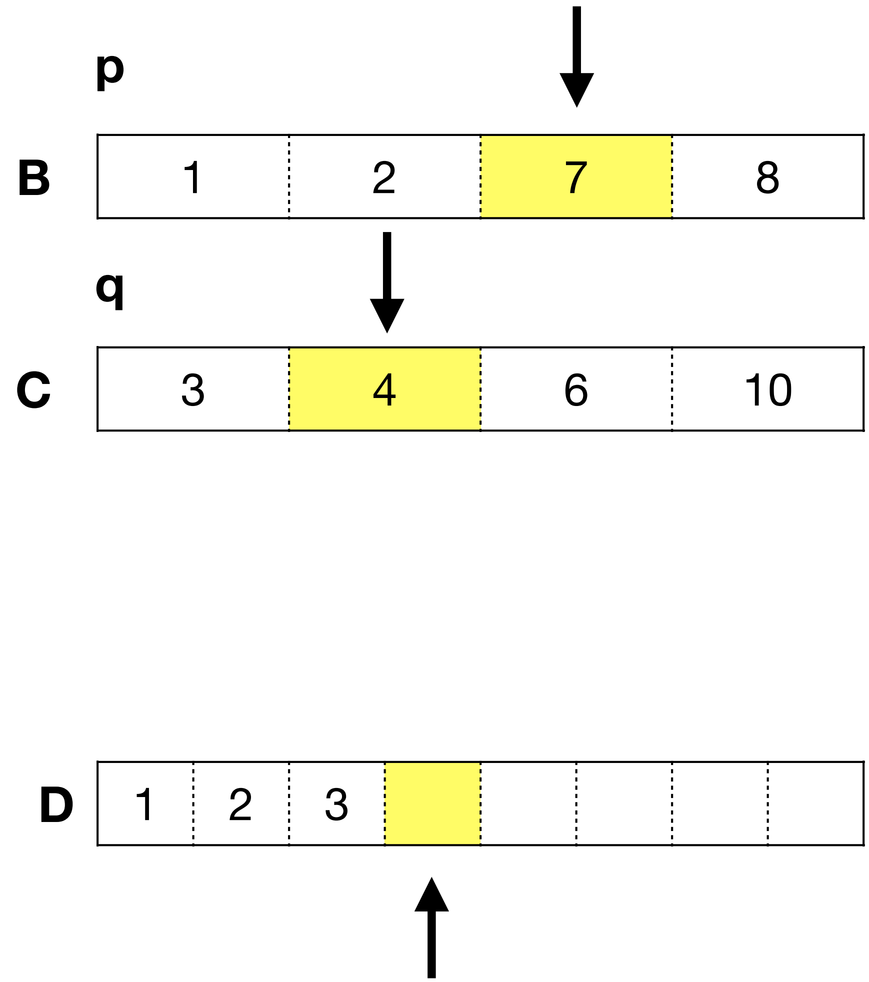
1. Create an empty array D[1:m+k] with pointers p=1 and q=1
2. Add $+\infty$ to the end of both arrays B and C
3. For i=1 to m+k:
   - If B[p] < C[q],
     - let D[i] = B[p] and set p = p+1
   - Else let D[i] = C[q] and set q=q+1
4. Return D



Merge — Example

Merge(B[1:m],C[1:k])

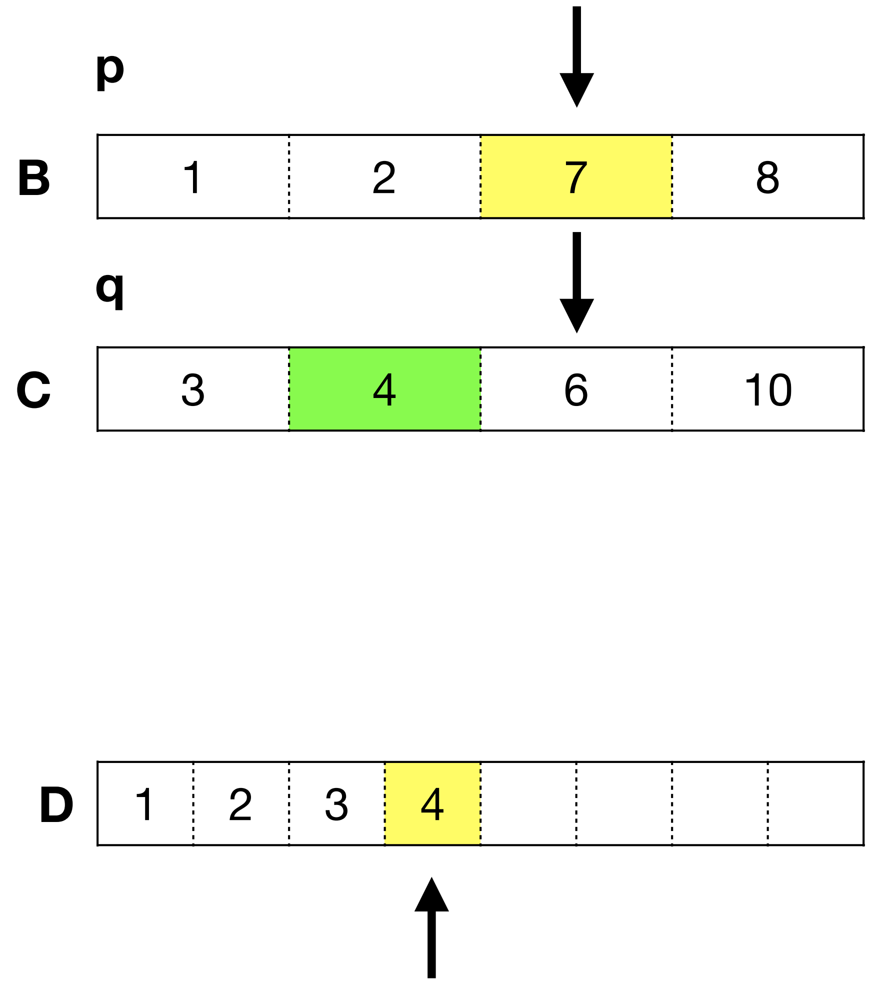
1. Create an empty array D[1:m+k] with pointers p=1 and q=1
2. Add $+\infty$ to the end of both arrays B and C
3. For i=1 to m+k:
    - If B[p] < C[q],
        - let D[i] = B[p] and set p = p+1
    - Else let D[i] = C[q] and set q=q+1
4. Return D



Merge — Example

Merge(B[1:m],C[1:k])

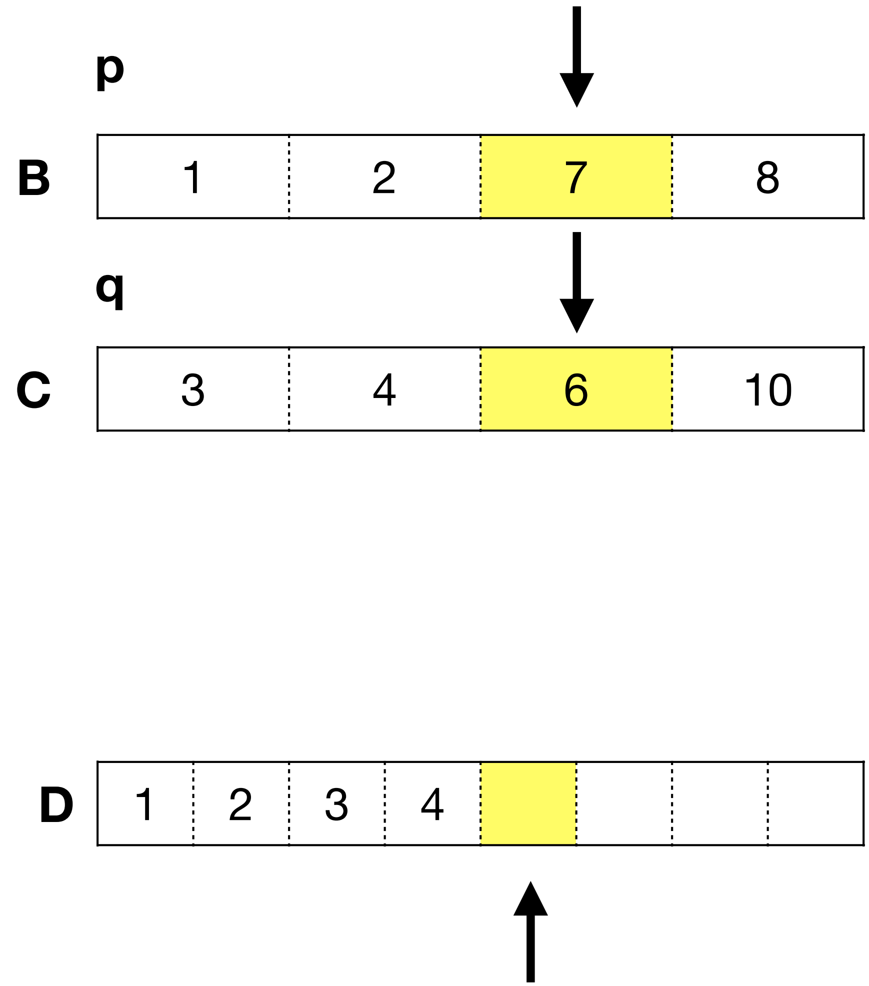
1. Create an empty array D[1:m+k] with pointers p=1 and q=1
2. Add $+\infty$ to the end of both arrays B and C
3. For i=1 to m+k:
   - If B[p] < C[q],
     - let D[i] = B[p] and set p = p+1
   - Else let D[i] = C[q] and set q=q+1
4. Return D



Merge — Example

Merge(B[1:m],C[1:k])

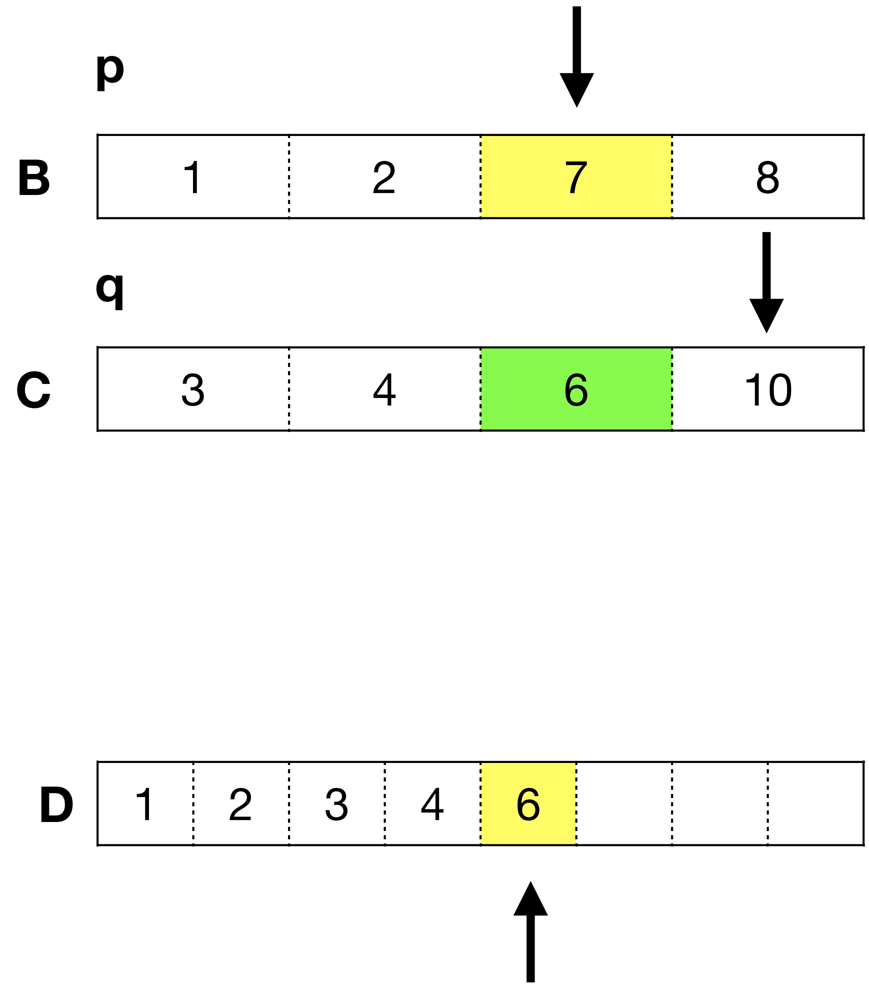
1. Create an empty array D[1:m+k] with pointers p=1 and q=1
2. Add $+\infty$ to the end of both arrays B and C
3. For i=1 to m+k:
   - If B[p] < C[q],
     - let D[i] = B[p] and set p = p+1
   - Else let D[i] = C[q] and set q=q+1
4. Return D



Merge — Example

Merge(B[1:m],C[1:k])

1. Create an empty array D[1:m+k] with pointers p=1 and q=1
2. Add $+\infty$ to the end of both arrays B and C
3. For i=1 to m+k:
    - If B[p] < C[q],
        - let D[i] = B[p] and set p = p+1
    - Else let D[i] = C[q] and set q=q+1
4. Return D



Merge — Example

Merge(B[1:m],C[1:k])

1. Create an empty array D[1:m+k] with pointers p=1 and q=1

2. Add $+\infty$ to the end of both arrays B and C

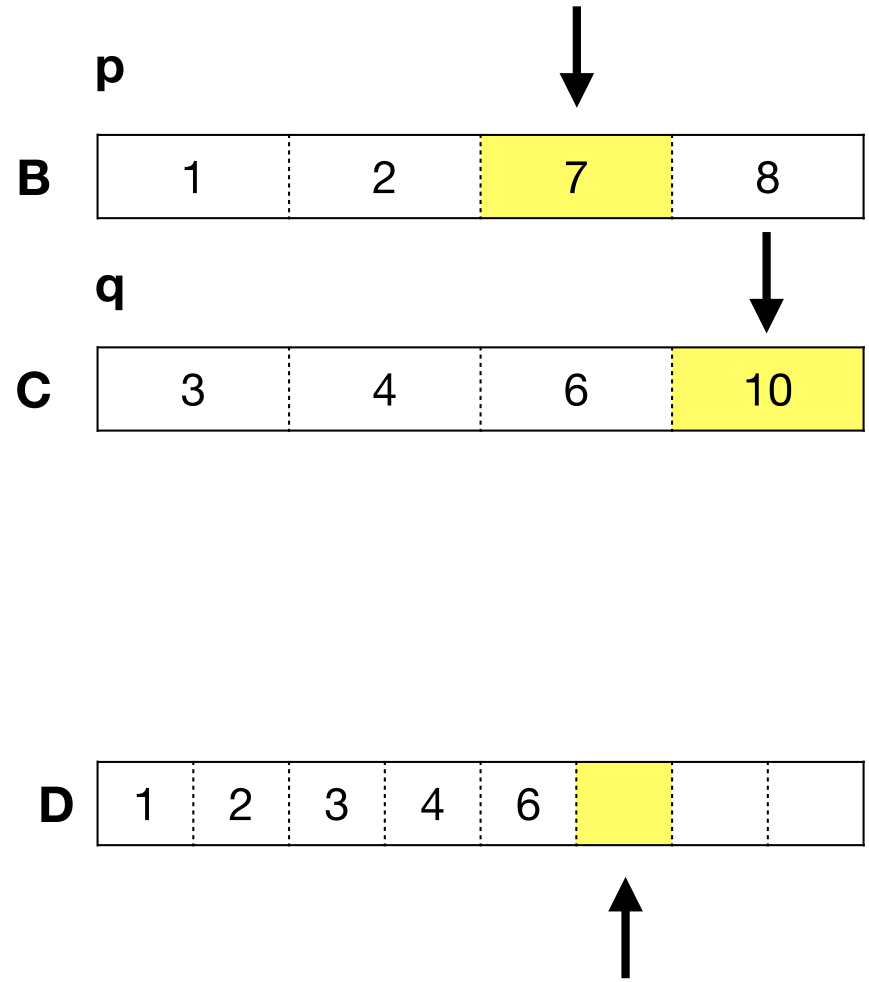
3. For i=1 to m+k:

   - If B[p] < C[q],

     - let D[i] = B[p] and set p = p+1

   - Else let D[i] = C[q] and set q=q+1

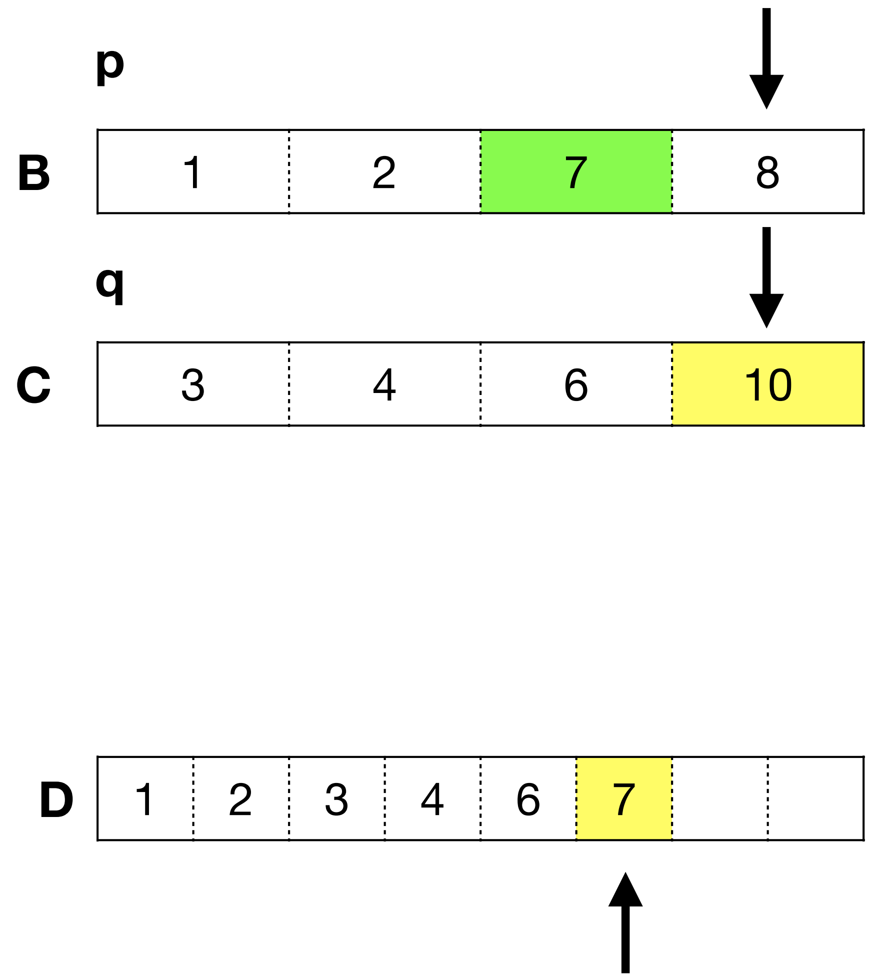
4. Return D



Merge — Example

Merge(B[1:m],C[1:k])

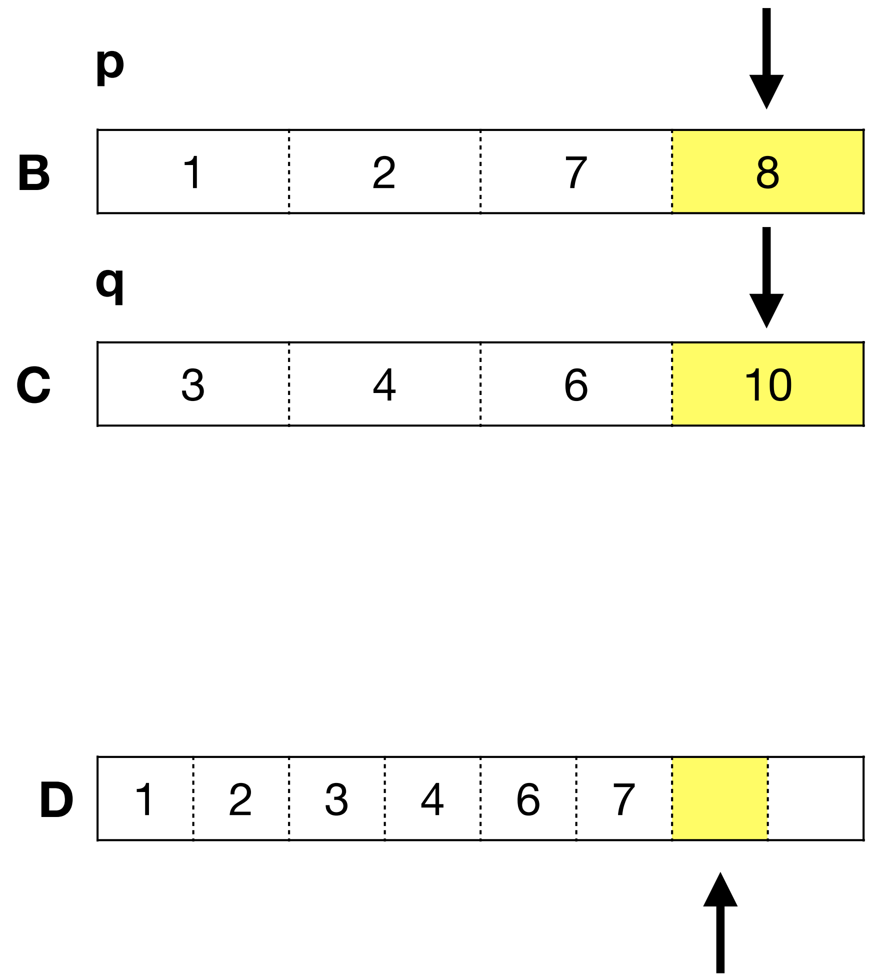
1. Create an empty array D[1:m+k] with pointers p=1 and q=1
2. Add $+\infty$ to the end of both arrays B and C
3. For i=1 to m+k:
   - If B[p] < C[q],
     - let D[i] = B[p] and set p = p+1
   - Else let D[i] = C[q] and set q=q+1
4. Return D



Merge — Example

Merge(B[1:m],C[1:k])

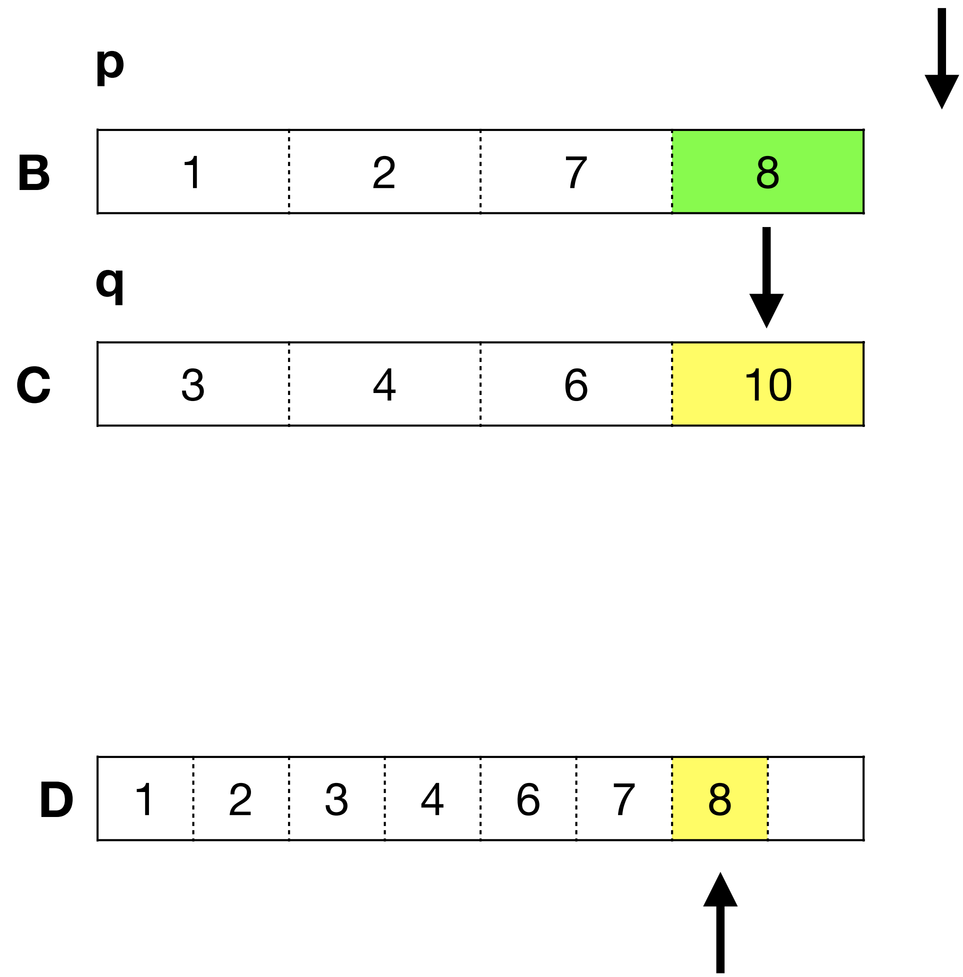
1. Create an empty array D[1:m+k] with pointers p=1 and q=1
2. Add $+\infty$ to the end of both arrays B and C
3. For i=1 to m+k:
   - If B[p] < C[q],
     - let D[i] = B[p] and set p = p+1
   - Else let D[i] = C[q] and set q=q+1
4. Return D



Merge — Example

Merge(B[1:m],C[1:k])

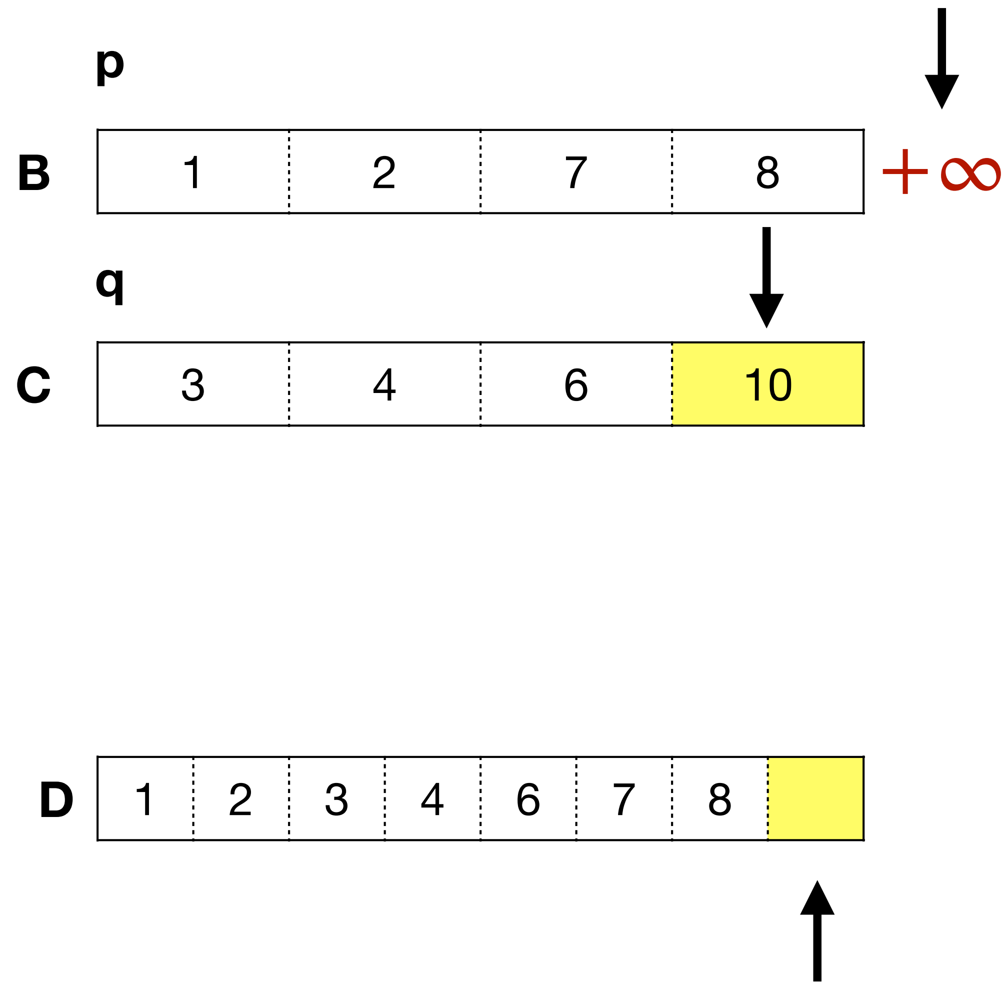
1. Create an empty array D[1:m+k] with pointers p=1 and q=1
2. Add $+\infty$ to the end of both arrays B and C
3. For i=1 to m+k:
    - If B[p] < C[q],
        - let D[i] = B[p] and set p = p+1
    - Else let D[i] = C[q] and set q=q+1
4. Return D



Merge — Example

Merge(B[1:m],C[1:k])

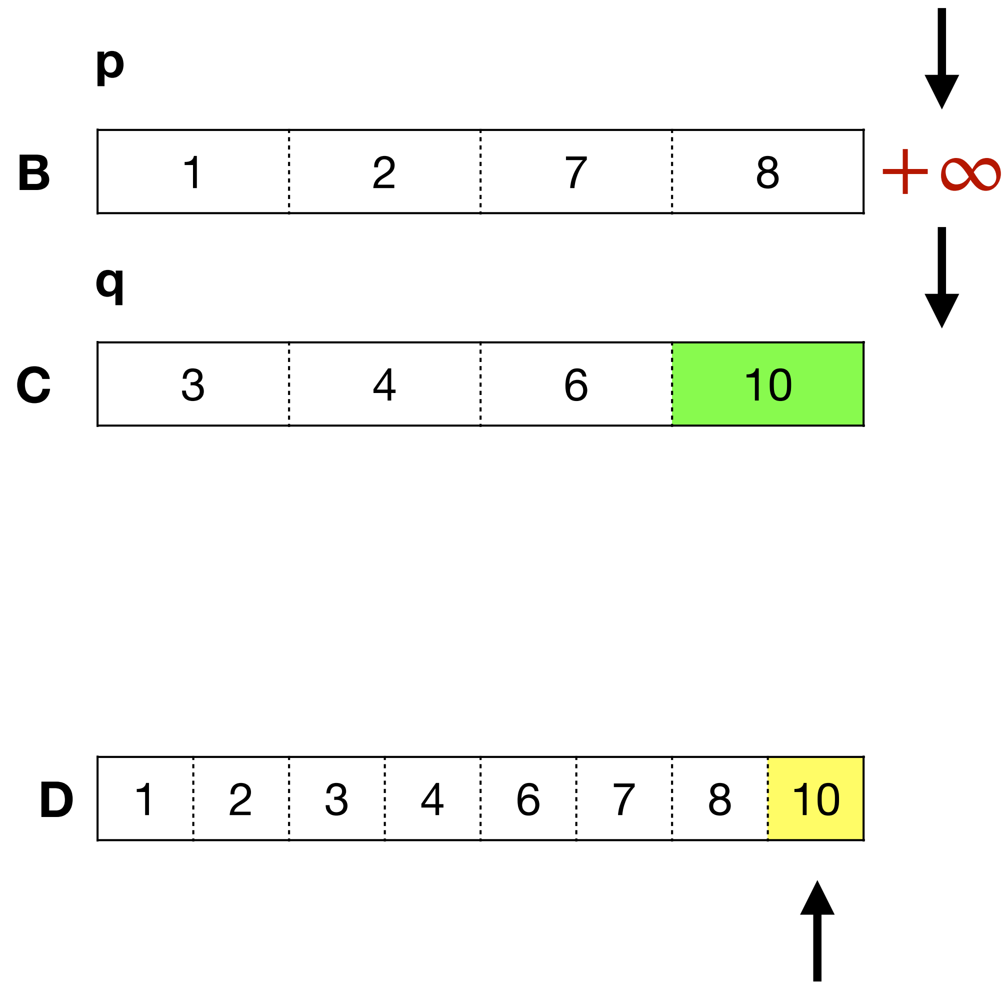
1. Create an empty array D[1:m+k] with pointers p=1 and q=1
2. Add $+\infty$ to the end of both arrays B and C
3. For i=1 to m+k:
   - If B[p] < C[q],
     - let D[i] = B[p] and set p = p+1
   - Else let D[i] = C[q] and set q=q+1
4. Return D



Merge — Example

Merge(B[1:m],C[1:k])

1. Create an empty array D[1:m+k] with pointers p=1 and q=1
2. Add $+\infty$ to the end of both arrays B and C
3. For i=1 to m+k:
    - If B[p] < C[q],
        - let D[i] = B[p] and set p = p+1
    - Else let D[i] = C[q] and set q=q+1
4. Return D



Proof of Correctness: Merge

Merge(B[1:m],C[1:k])

1. Create an empty array D[1:m+k] with pointers p=1 and q=1
2. Add $+\infty$ to the end of both arrays B and C
3. For i=1 to m+k:
    - If B[p] < C[q],
        - let D[i] = B[p] and set p = p+1
    - Else let D[i] = C[q] and set q=q+1
4. Return D

- Statement: For every $i \leq m + k$, after iteration i, D[1:i] contains the smallest elements of B + C in the sorted order.

Proof of Correctness: Merge

Merge(B[1:m],C[1:k])

1. Create an empty array D[1:m+k] with pointers p=1 and q=1
2. Add $+\infty$ to the end of both arrays B and C
3. For i=1 to m+k:
    - If B[p] < C[q],
        - let D[i] = B[p] and set p = p+1
    - Else let D[i] = C[q] and set q=q+1
4. Return D

- Statement: For every $i \leq m+k$, after iteration i, D[1:i] contains the smallest elements of B + C in the sorted order.
- Proof by induction:
- Induction Base:
- For i=1, D[1] = min(B[1],C[1]) which is the smallest number (B and C are sorted)

Proof of Correctness: Merge

Merge(B[1:m],C[1:k])

1. Create an empty array D[1:m+k] with pointers p=1 and q=1
2. Add $+\infty$ to the end of both arrays B and C
3. For i=1 to m+k:
   - If B[p] < C[q],
     - let D[i] = B[p] and set p = p+1
   - Else let D[i] = C[q] and set q=q+1
4. Return D

- Induction step:
- Suppose it is true for i=j and we prove it for i=j+1
- Elements of B[1:p-1] and C[1:q-1] are already placed in D[1:j]
- D[j+1] = min(B[p],C[q])
- By induction hypothesis, D[1:j] contains smallest j elements
- So either B[p] or C[q] is the smallest remaining number (B and C are sorted)

Proof of Correctness: Merge-Sort

- **Merge-Sort**(A[1:n])

1. If n=0 or 1, return A.

2. Run **Merge-Sort**($A[1:\frac{n}{2}]$)

   and **Merge-Sort**($A[\frac{n}{2}+1,n]$

3. Combine first and second halves using the **Merge** algorithm

- Statement: **Merge-Sort** is correct for every array
- Proof by induction
- Induction base:
- n=0 or 1, A is already sorted
- Induction step:
- Suppose it is true for all integers $n \leq m$ and we prove it for n=m+1

Proof of Correctness: Merge-Sort

- **Merge-Sort**(A[1:n])

1. If n=0 or 1, return A.
2. Run **Merge-Sort**($A[1:\frac{n}{2}]$) and **Merge-Sort**($A[\frac{n}{2}+1,n]$
3. Combine first and second halves using the **Merge** algorithm

- By induction hypothesis, after line 2, both $A[1:\frac{n}{2}]$ and $A[\frac{n}{2}+1:n]$ are sorted correctly
- By correctness of **Merge**, their combined array will be sorted correctly
- All elements of A belong to the combined array

Runtime Analysis: Merge

Merge(B[1:m],C[1:k])

1. Create an empty array D[1:m+k] with pointers p=1 and q=1
2. Add $+\infty$ to the end of both arrays B and C
3. For i=1 to m+k:
    - If B[p] < C[q],
        - let D[i] = B[p] and set p = p+1
    - Else let D[i] = C[q] and set q=q+1
4. Return D

- A single for-loop with $(m+k)$ iterations each taking $\Theta(1)$ time
- $\Theta(m+k)$ time in total

Proof of Correctness: Merge-Sort

- **Merge-Sort**(A[1:n])

1. If n=0 or 1, return A.
2. Run **Merge-Sort**($A[1 : \frac{n}{2}]$)

   and **Merge-Sort**($A[\frac{n}{2} + 1, n]$
3. Combine first and second halves using the **Merge** algorithm

- We write a recurrence
- $T(n) \leq 2 \cdot T(n/2) + O(n)$
- Next week, we will see how to solve such a recurrence
- For now, let us just mention that $T(n) = O(n \log n)$
- So we got ourself a very fast algorithm for sorting!

A Fun Question to Think About

Finding Local Minimum

- Suppose we have an array $A[1:n]$ with unique numbers
- Find an entry of this array which is smaller than its neighbors (left and right whenever they exist):
  - Neighbor of $A[1]$ is $A[2]$
  - Neighbor of $A[n]$ is $A[n-1]$
  - Neighbors of $A[i]$ for any other i is $A[i-1]$ and $A[i+1]$

Finding Local Minimum

- Suppose we have an array A[1:n] with unique numbers
- Find an entry of this array which is smaller than its neighbors (left and right whenever they exist):
  - Neighbor of A[1] is A[2]
  - Neighbor of A[n] is A[n-1]
  - Neighbors of A[i] for any other i is A[i-1] and A[i+1]
- **Hint:** this question has a solution as fast as binary search even though A is NOT sorted here

Gaining Intuition

- Draw a couple of examples and figure out their answers

1	2	3	4	5	6	7	8

2	5	4	6	7	8	3	1

8	7	6	5	1	2	3	4

4	5	1	7	3	2	6	8

Gaining Intuition

- Draw a couple of examples and figure out their answers

1	2	3	4	5	6	7	8

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

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

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