# Quick Refresher: Asymptotic Analysis

# Recall: Asymptotic Analysis

- How do costs scale as input sizes become arbitrarily large?
  - Ignore constant factors and small-order terms
- Run time analysis: count “steps”
  - A step is any amount of constant-time work
- Express # steps as function of input size, *n*
- Simplify using asymptotic notation
  - $O, \Theta, \Omega$

# Example

```
for (int i = 0; i < 2*n; ++i) {
  for (int j = 0; j < n+1; ++j) {
    ++sum; Θ(1)
  }
}
```

Inner loop total: $\Theta(n) \times \Theta(1) = \Theta(n)$

Outer loop total: $\Theta(n) \times \Theta(n) = \Theta(n^2)$

Most precise

Loop: $2n = \Theta(n)$ times

Loop: $n + 1 = \Theta(n)$ times

Also true:

$O(n^2)$ "no worse than"

$\Omega(n^2)$ "no better than"

$O(n^{31})$ "no worse than"

$\Omega(n)$ "no better than"

# Example

```
for (int i = 0; i < n; ++i) {
  for (int j = 0; j < i+1; ++j) {
    ++sum; Θ(1)
  }
}
```

Loop: $n$ times

Loop: $i + 1$ times

Important: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Outer loop total: $\Theta(1) \times (1 + 2 + \cdots + n) = \Theta(1) \sum_{i=1}^{n} i$

$$= \Theta(1) \frac{n(n+1)}{2} = \Theta(1) \times \Theta(n^2) = \Theta(n^2)$$

# Summary

- Just a quick refresher
  - Remember how to analyze a snippet of code
- $O, \Theta, \Omega$
- Common shortcuts
  - Analyze from the inside-out
  - Often multiply # loop iterations by iteration complexity
  - Sometimes iteration complexity depends on loop variable
    - Need to sum
    - Important formula: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

# Coming Up…

What if runtime is affected by things other than *n*?

# Best- and Worst-Case Analysis

(Linear Search)

# Searching an Array

Determine whether size *n* array *arr* contains item *x*

```
LINEAR-SEARCH (arr, n, x)
  for each value of i from 0 to n-1
    if arr[i] == x, return true
  if loop completes, return false
```

If you get an array of size *n*, how many steps will it perform?

It Depends!
Is x in there?
If so, where is it?

# Best- And Worst-Case Analysis

- Best-case complexity:
  - The smallest number of steps the algorithm could take on an input of size *n*
- Best case:
  - The situation in which the best case complexity occurs
- Worst-case complexity:
  - The largest number of steps the algorithm could take on an input of size *n*
- Worst-Case
  - The situation in which the worst-case complexity occurs

# Searching an Array

Determine whether size *n* array *arr* contains item *x*

```
LINEAR-SEARCH (arr, n, x)
  for each value of i from 0 to n-1
    if arr[i] == x, return true
  if loop completes, return false
```

Best case?

*x* is the first item

Best-case complexity?

$\Theta(1)$

Worst case?

*x* is not present

Worst-case complexity?

$\Theta(n)$

# Using $O$, $\Theta$, $\Omega$

- The best-case complexity is $\Theta(1)$
- The worst-case complexity is $\Theta(n)$
- The complexity is $\Omega(1)$
- The complexity is $O(n)$
- The worst-case complexity is $\Omega(n)$
- The worst-case complexity is $O(n)$

- With no traffic, it will take 30 min.
- At rush hour, it will take 2 hrs.
- It will take at least 30 min.
- It will take at most 2 hrs.
- At rush hour, it will take at least 2 hrs.
- At rush hour, it will take at most 2 hrs.

Accurate, but imprecise

# Summary

- Sometimes complexity depends on more than *n*
- Best-case complexity
  - The smallest number of steps for an input of size *n*
- Best case
  - The situation which causes the best-case complexity
- Worst-case complexity
  - The largest number of steps for an input of size *n*
- Worse case
  - The situation which cases the worst-case complexity
- Using $O$, $\Theta$, $\Omega$
  - Translate into “is at most”, “is”, “is at least”
  - Try to say the most precise thing you can accurately say

# Coming Up…

Another example

# Best- and Worst-Case Analysis

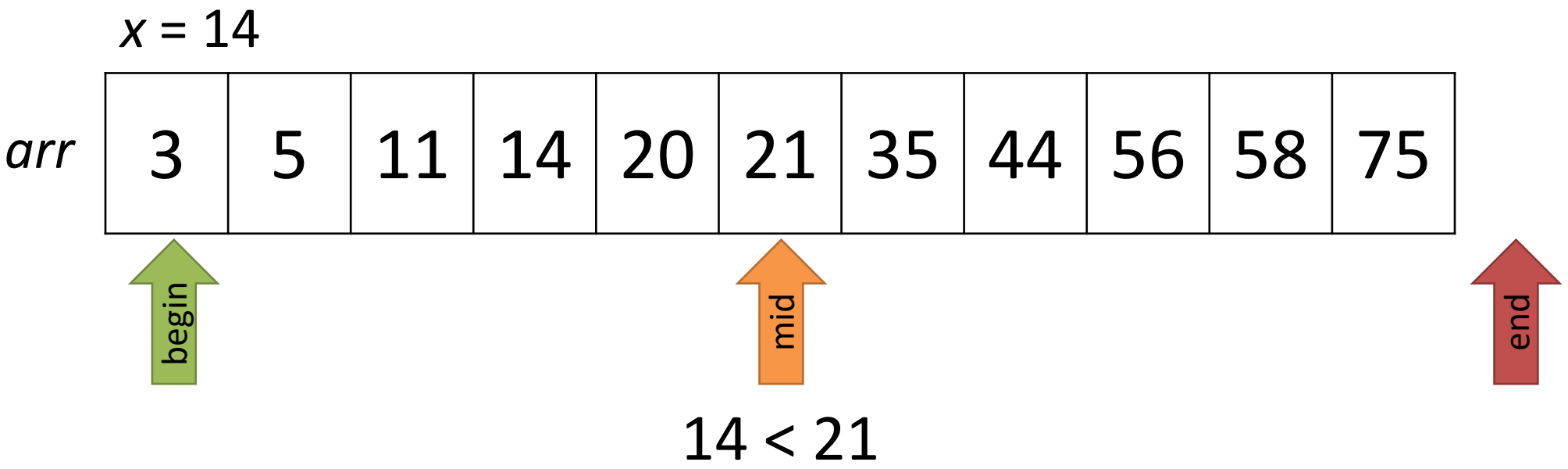
(Binary Search)

# Searching a *Sorted* Array

Determine whether size *n* sorted array *arr* contains item *x*

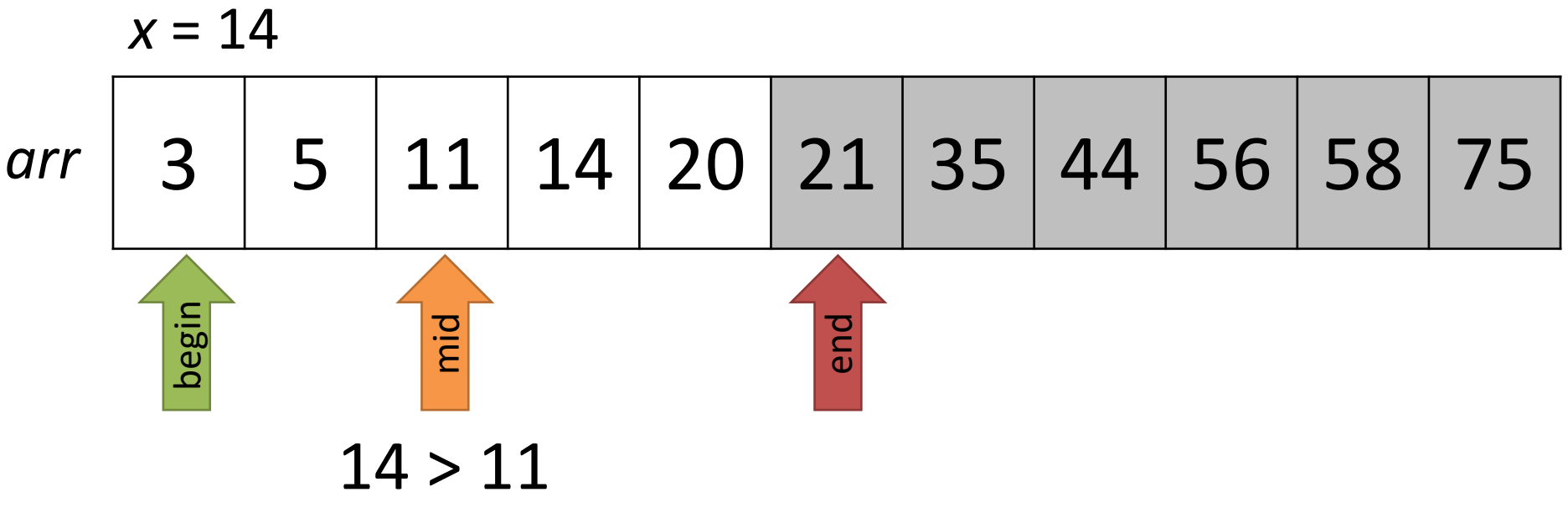
# Searching a *Sorted* Array

Determine whether size *n* sorted array *arr* contains item *x*



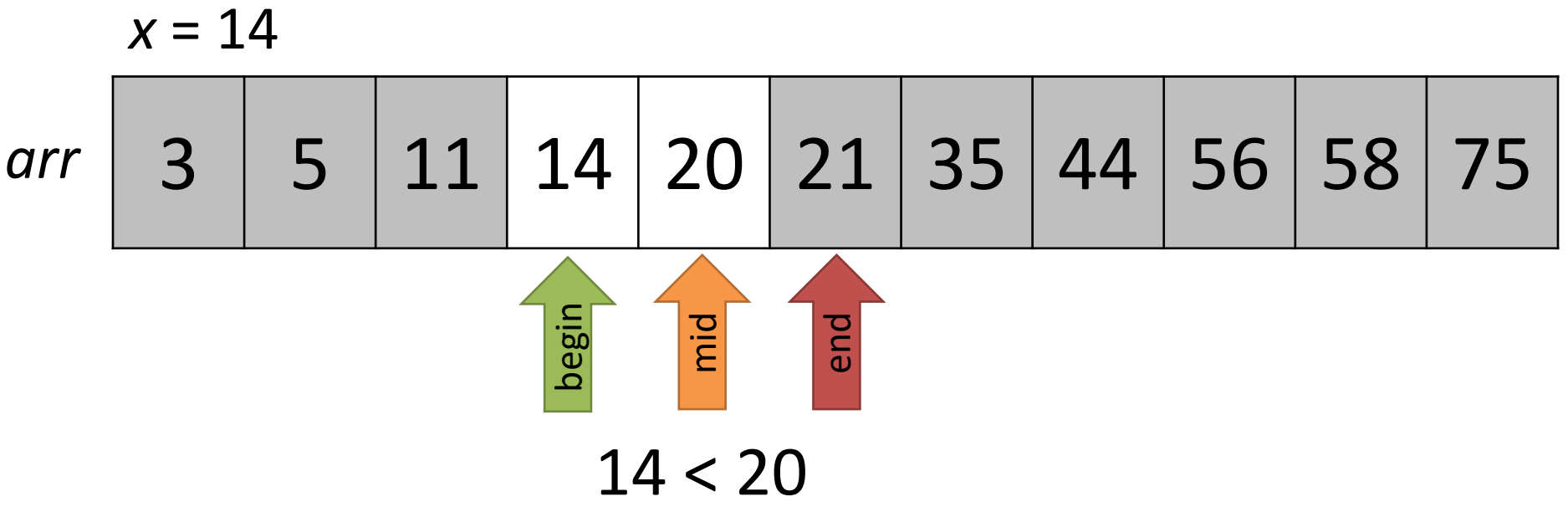
# Searching a *Sorted* Array

Determine whether size *n* sorted array *arr* contains item *x*



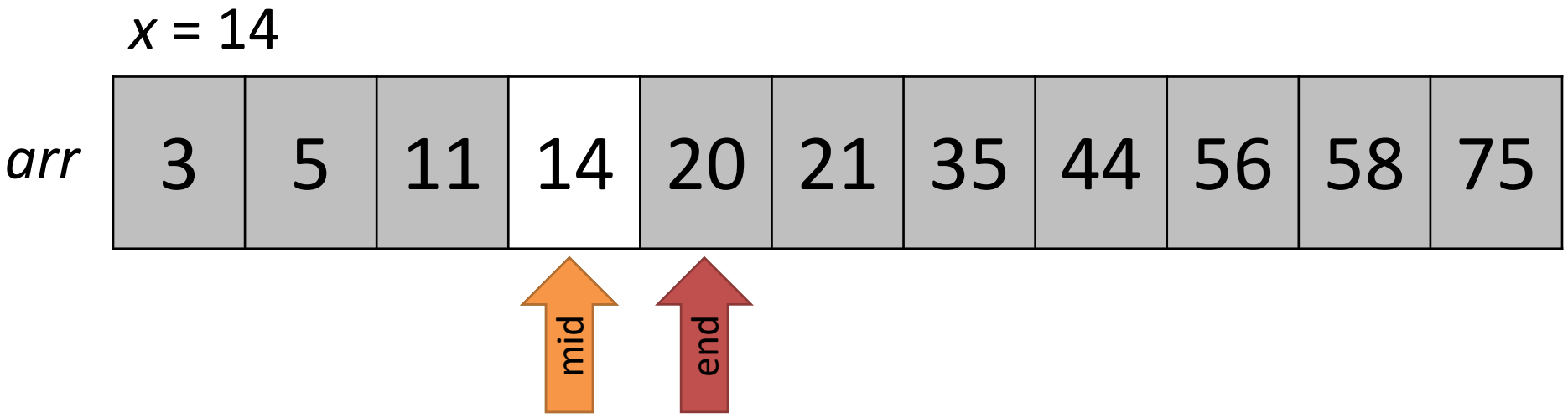
# Searching a *Sorted* Array

Determine whether size *n* sorted array *arr* contains item *x*



# Searching a *Sorted* Array

Determine whether size *n* sorted array *arr* contains item *x*

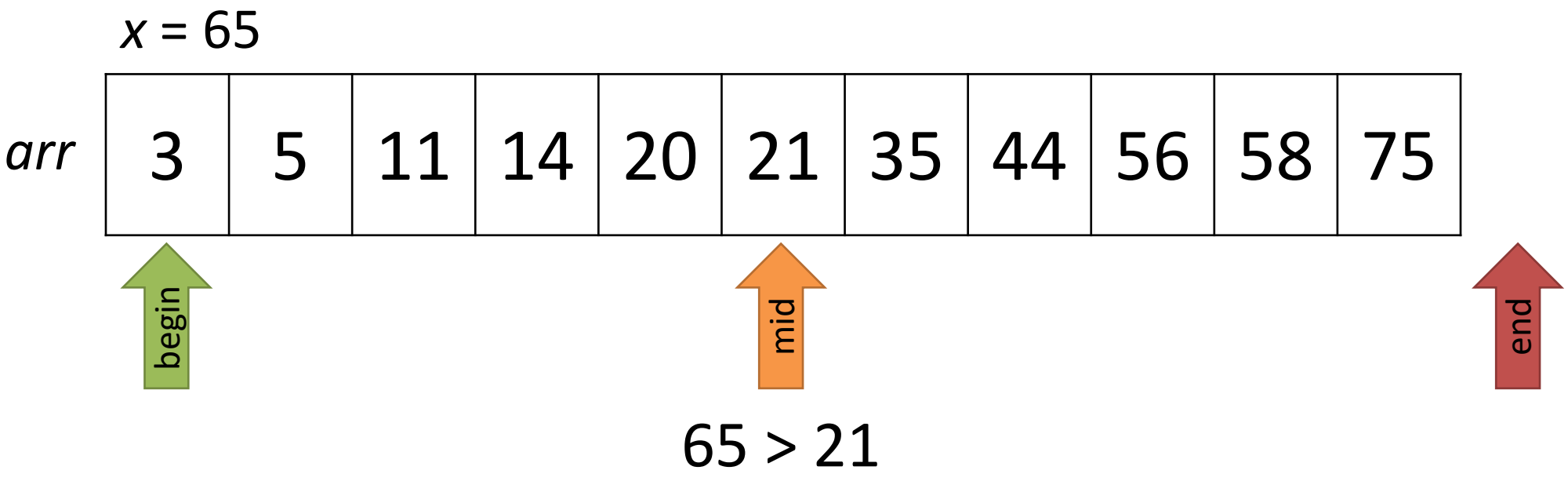



14 == 14

Return true!

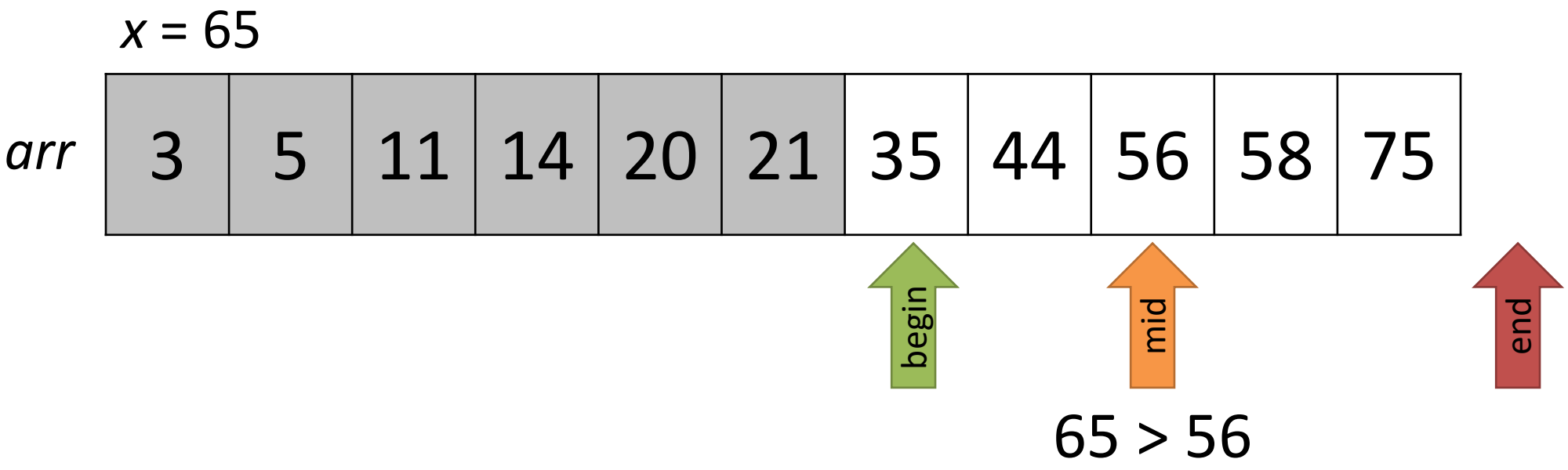
# Searching a *Sorted* Array

Determine whether size *n* sorted array *arr* contains item *x*



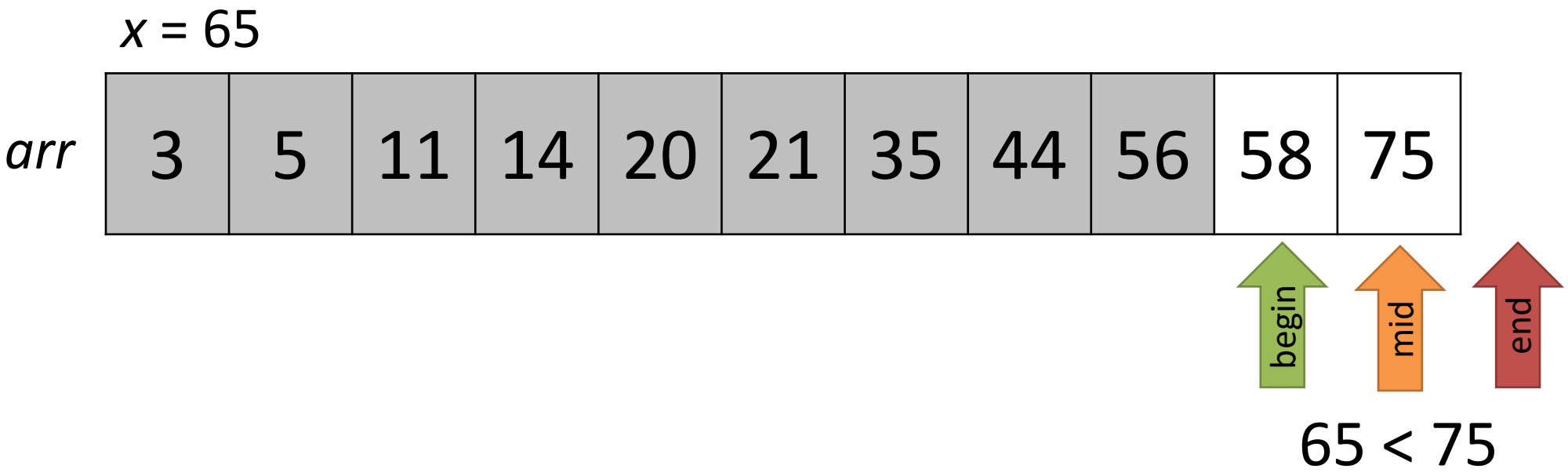
# Searching a *Sorted* Array

Determine whether size *n* sorted array *arr* contains item *x*



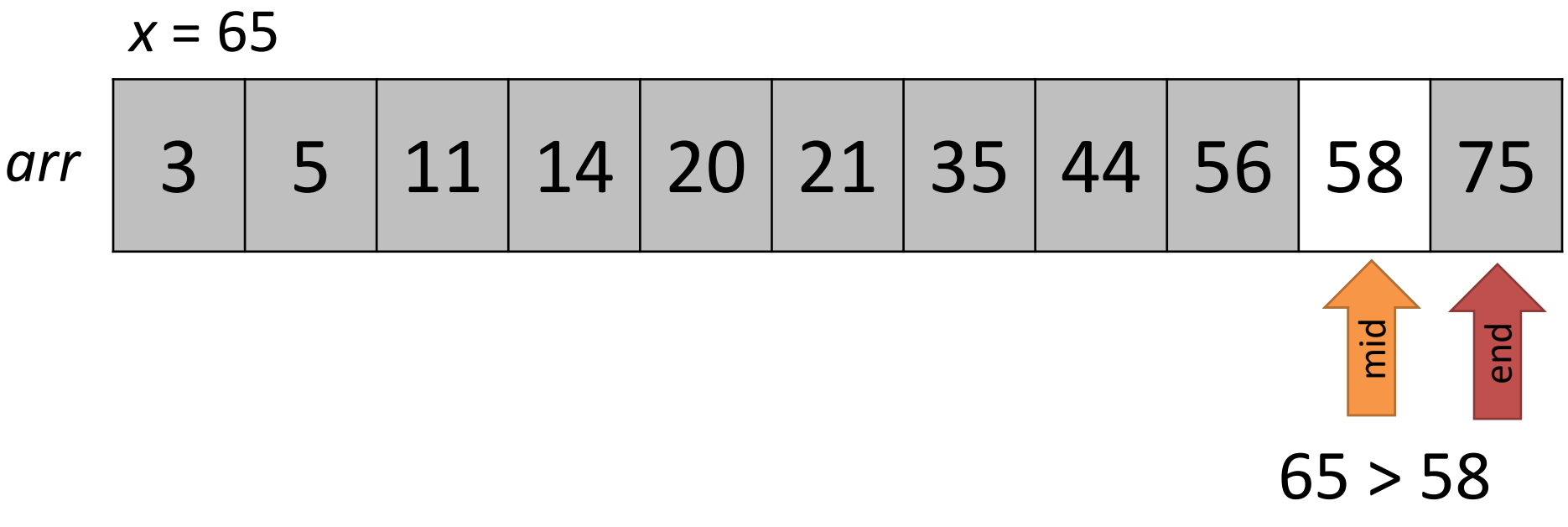
# Searching a *Sorted* Array

Determine whether size *n* sorted array *arr* contains item *x*



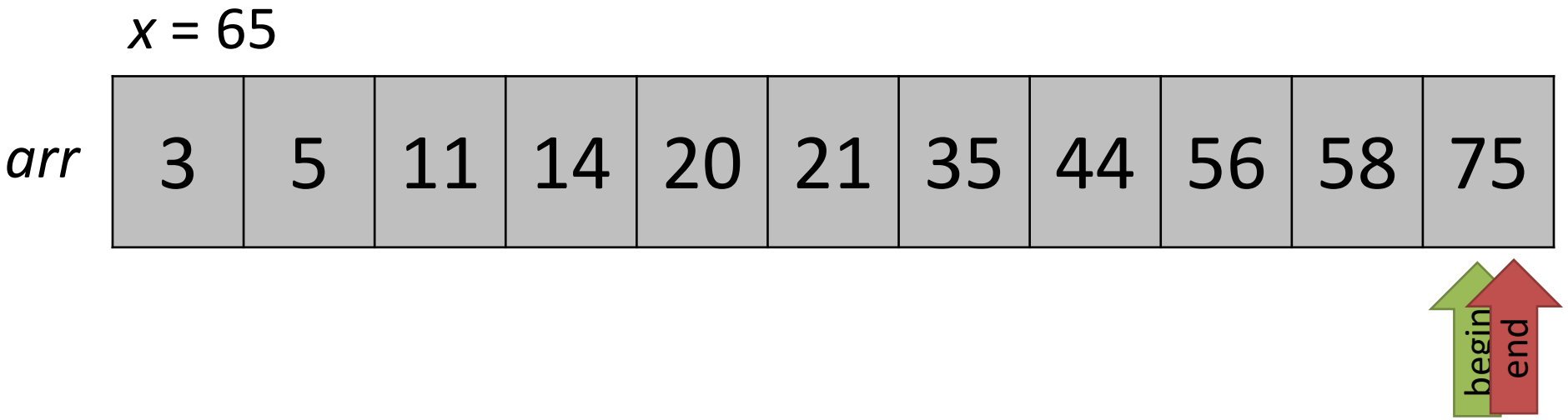
# Searching a *Sorted* Array

Determine whether size *n* sorted array *arr* contains item *x*



# Searching a *Sorted* Array

Determine whether size *n* sorted array *arr* contains item *x*




begin == end

Return false

# Searching a *Sorted* Array

Determine whether size *n* sorted array *arr* contains item *x*

```
BINARY-SEARCH (arr, n, x)
  BINARY-SEARCH-HELPER (arr, 0, n, x)

BINARY-SEARCH-HELPER (arr, begin, end, x)
  if begin == end, return false
  mid = (begin + end)/2 (rounded down)
  if arr[mid] == x, return true
  else if arr[mid] > x,
    return BINARY-SEARCH-HELPER (arr, begin, mid, x)
  else if arr[mid] < x,
    return BINARY-SEARCH-HELPER (arr, mid + 1, end, x)
```

Best case?

*x* is the middle item

Best-case complexity?

$\Theta(1)$

Worst case?

*x* is not present

Worst-case complexity?

$\Theta(\log n)$

# Reasonable things to say

- The best-case complexity is $\Theta(1)$
- The worst-case complexity is $\Theta(\log n)$
- The complexity is $\Omega(1)$
- The complexity is $O(\log n)$

# Summary

- Binary Search
  - Search for an element of a sorted array
  - Consider the middle element
  - If it's too big, search the left half
  - If it's too small, search the right half
  - If it's just right, we're done!
- Complexity: $O(\log n)$

# Coming Up…

Another example

# Best- and Worst-Case Analysis

(Merge Sort)

# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order

Merge Sort:

- Sort the left half
- Sort the right half
- Merge the two sorted lists

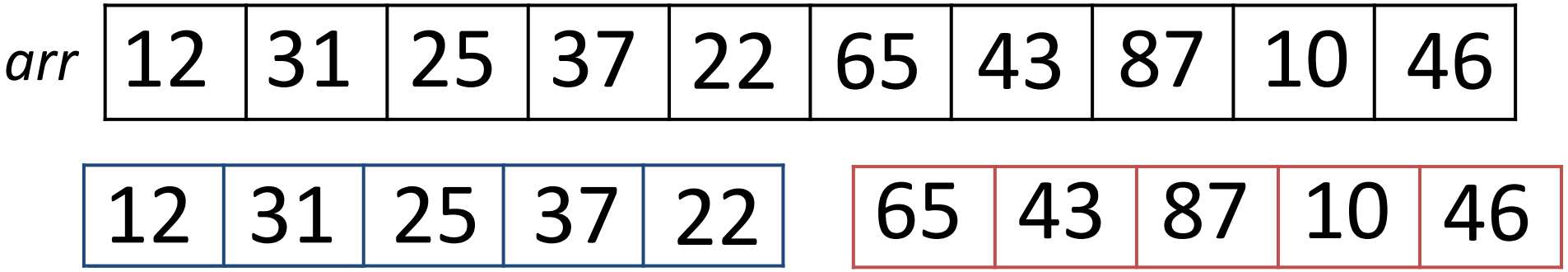
# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order

| *arr* | 12 | 31 | 25 | 37 | 22 | 65 | 43 | 87 | 10 | 46 |
|---|---|---|---|---|---|---|---|---|---|---|

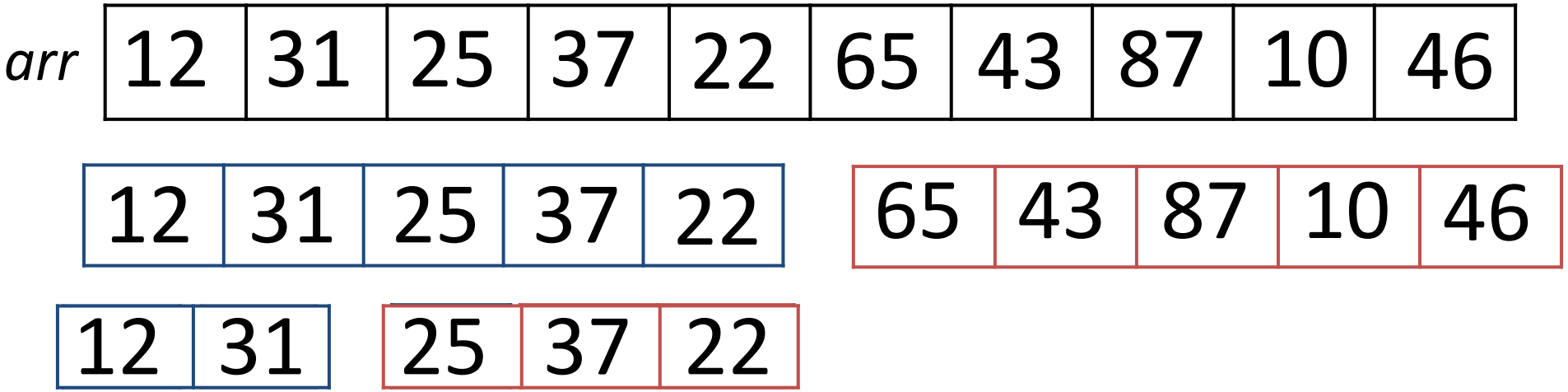
# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order



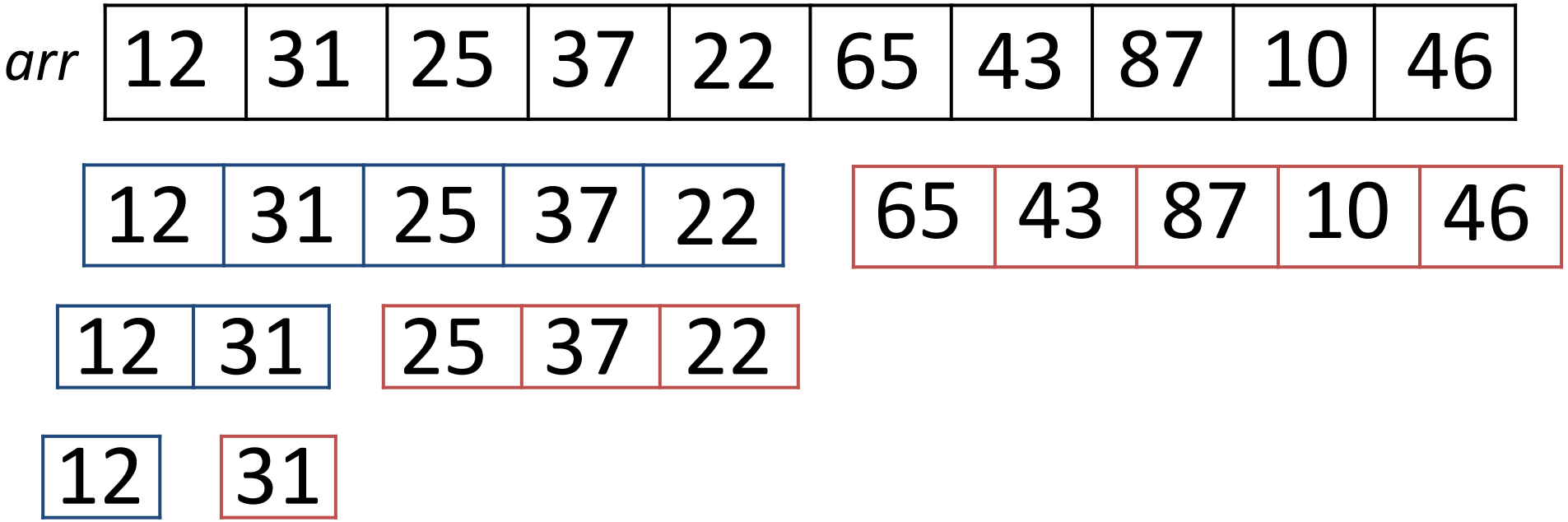
# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order



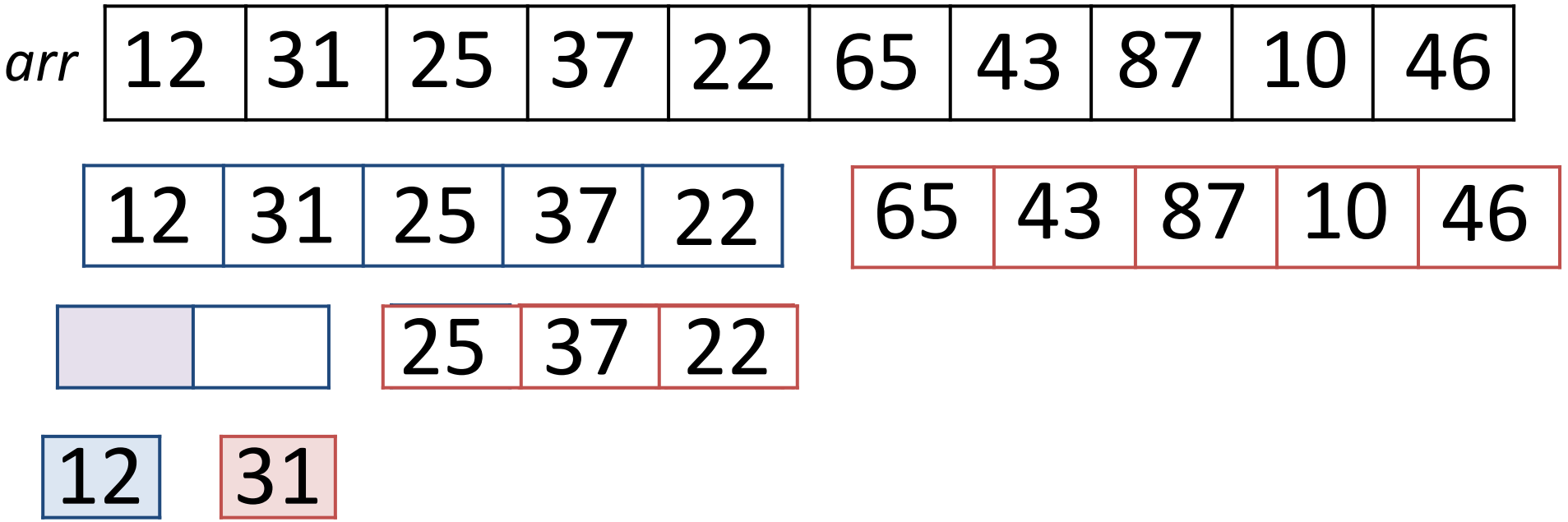
# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order



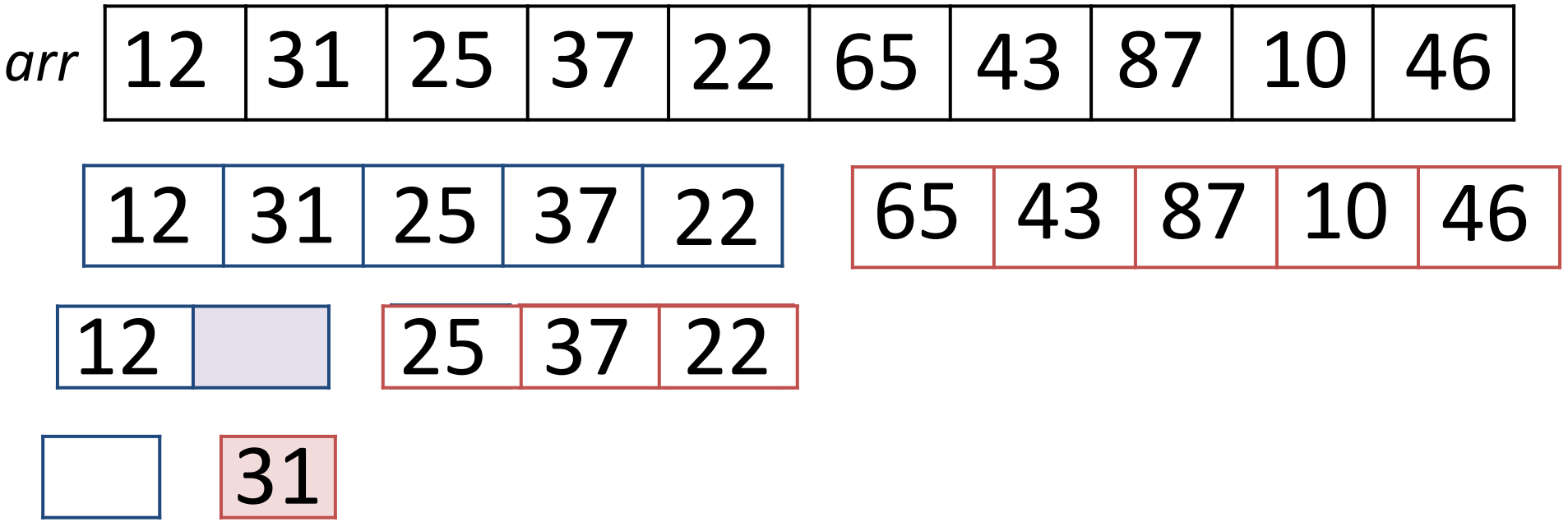
# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order



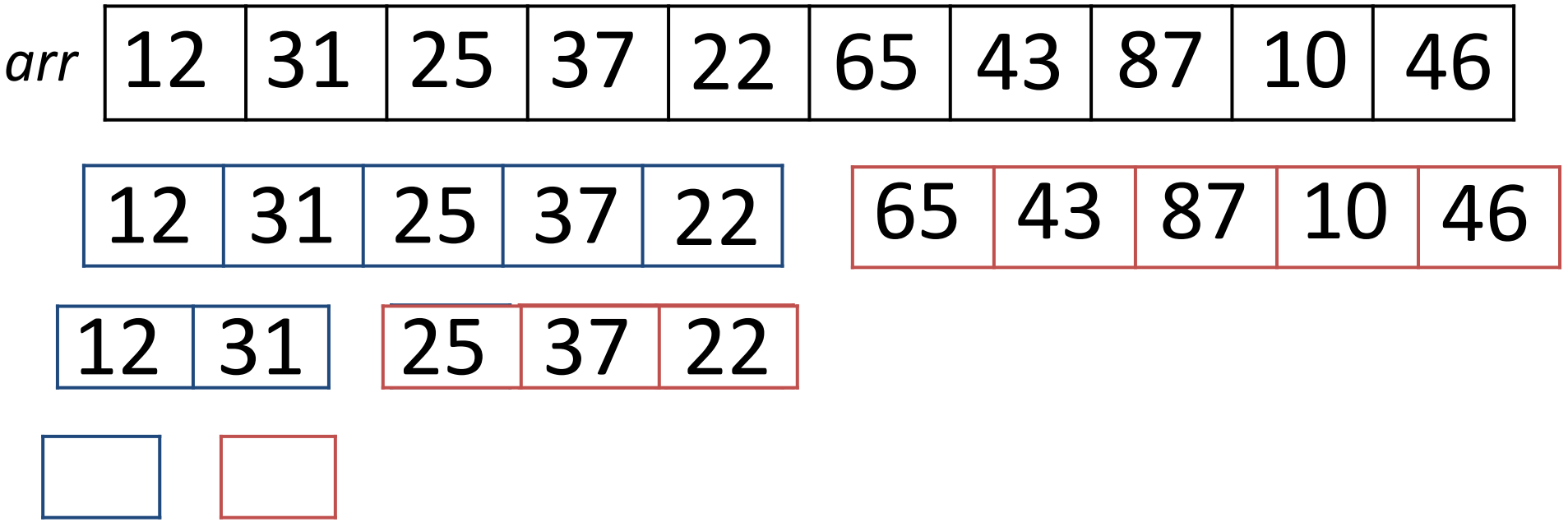
# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order



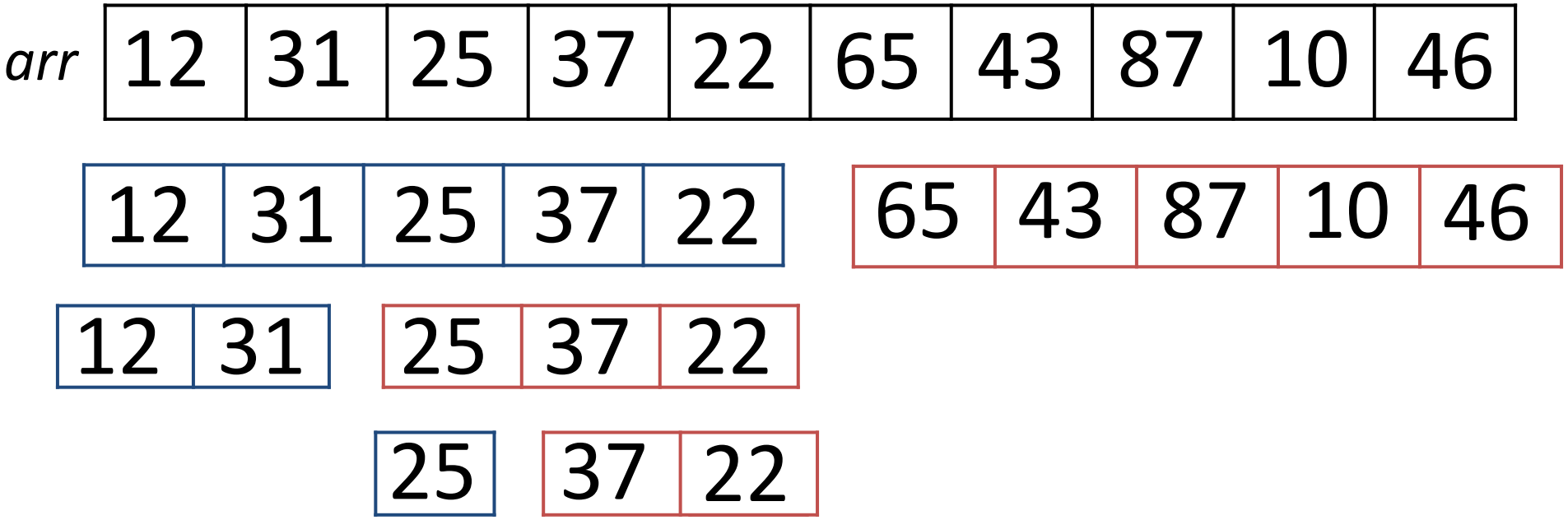
# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order



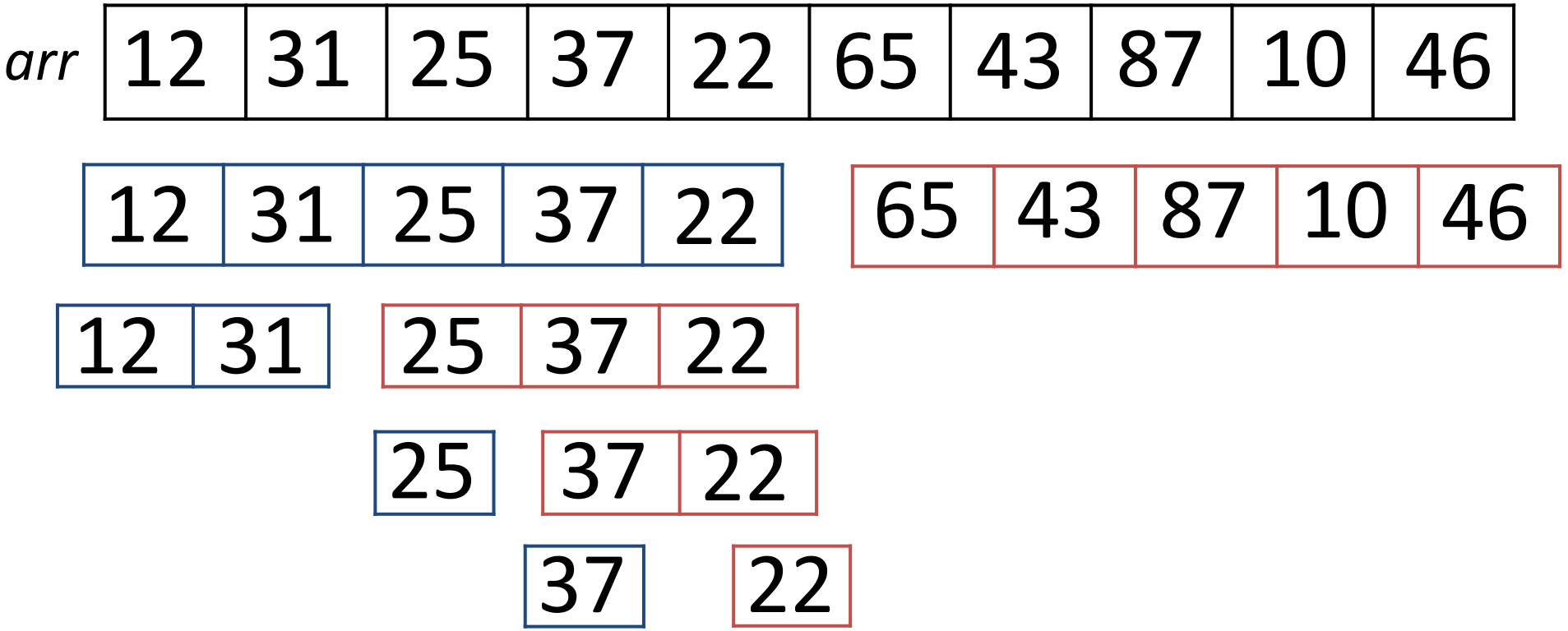
# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order



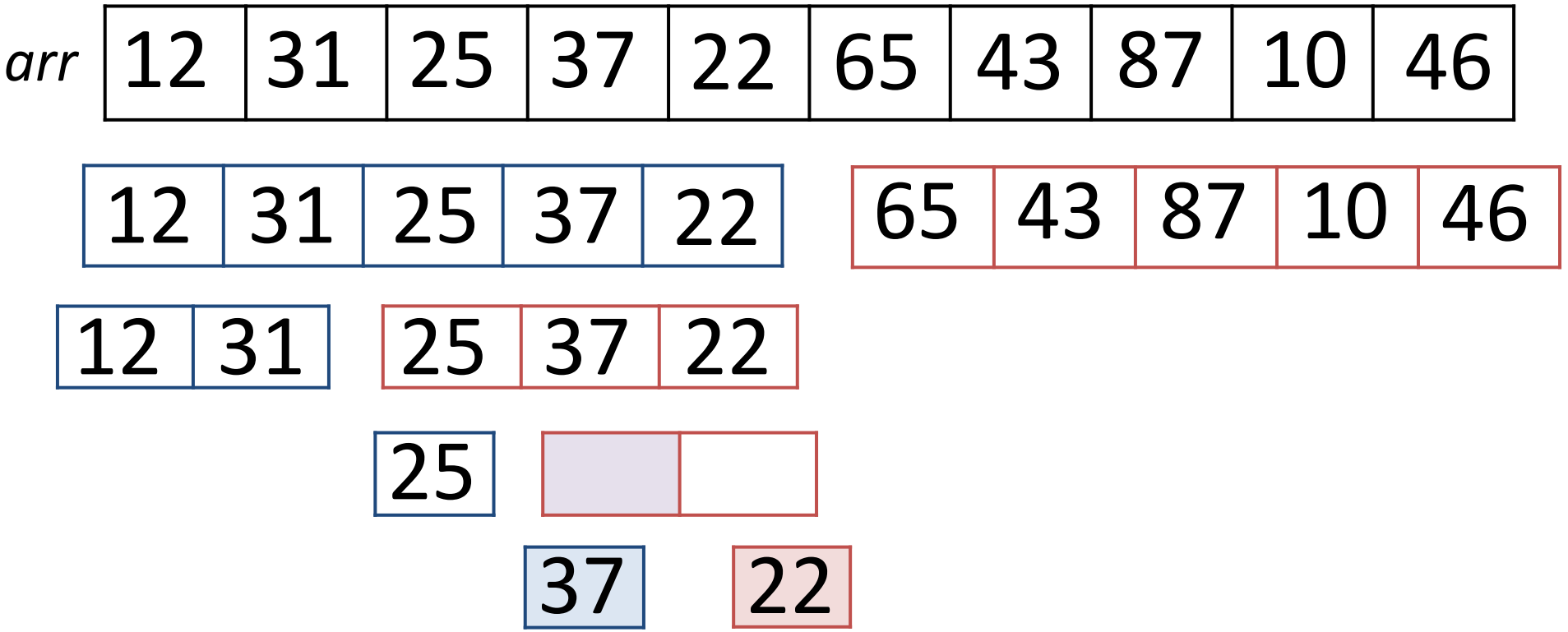
# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order



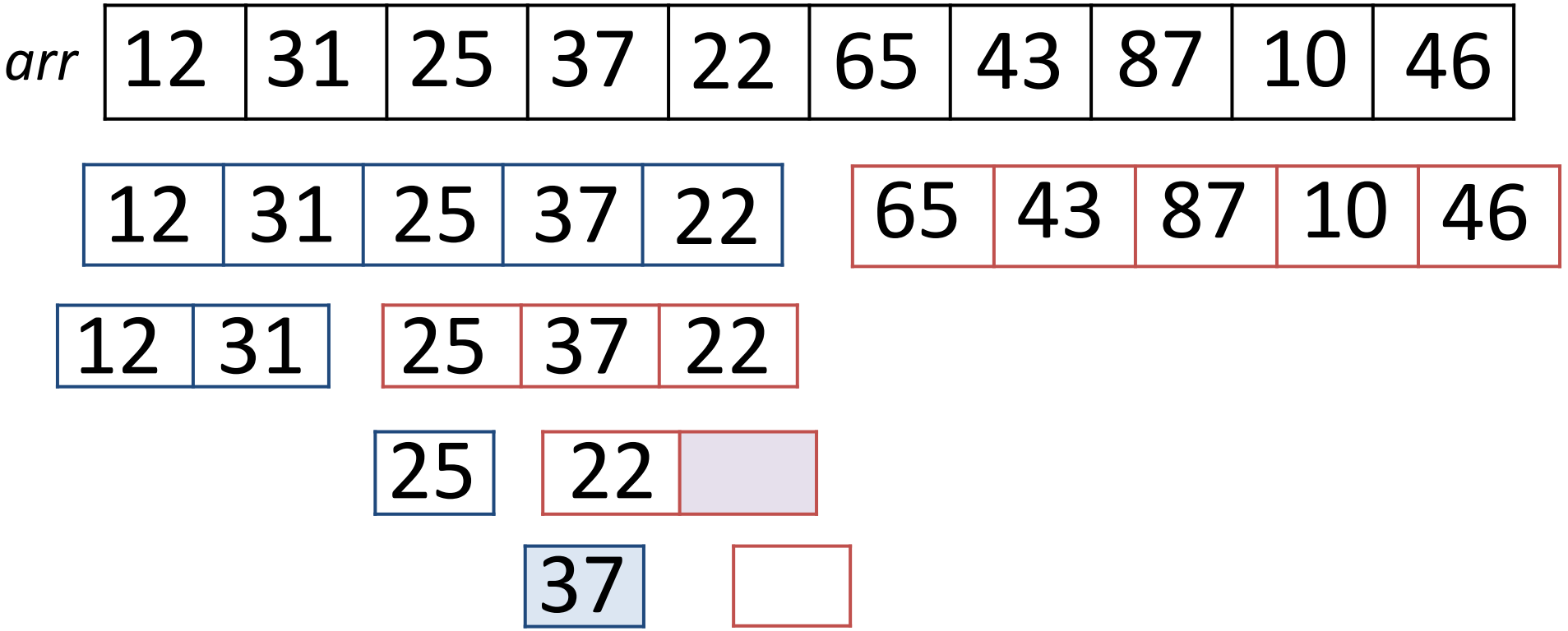
# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order



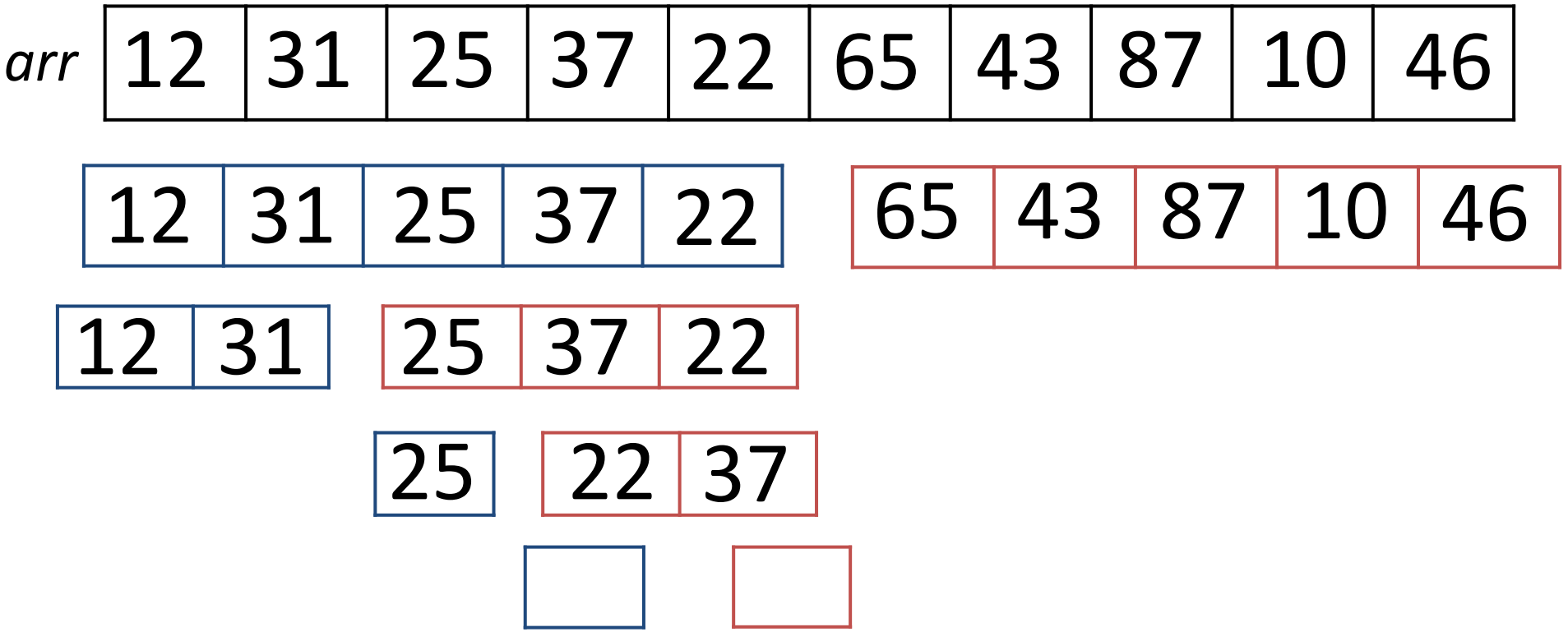
# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order



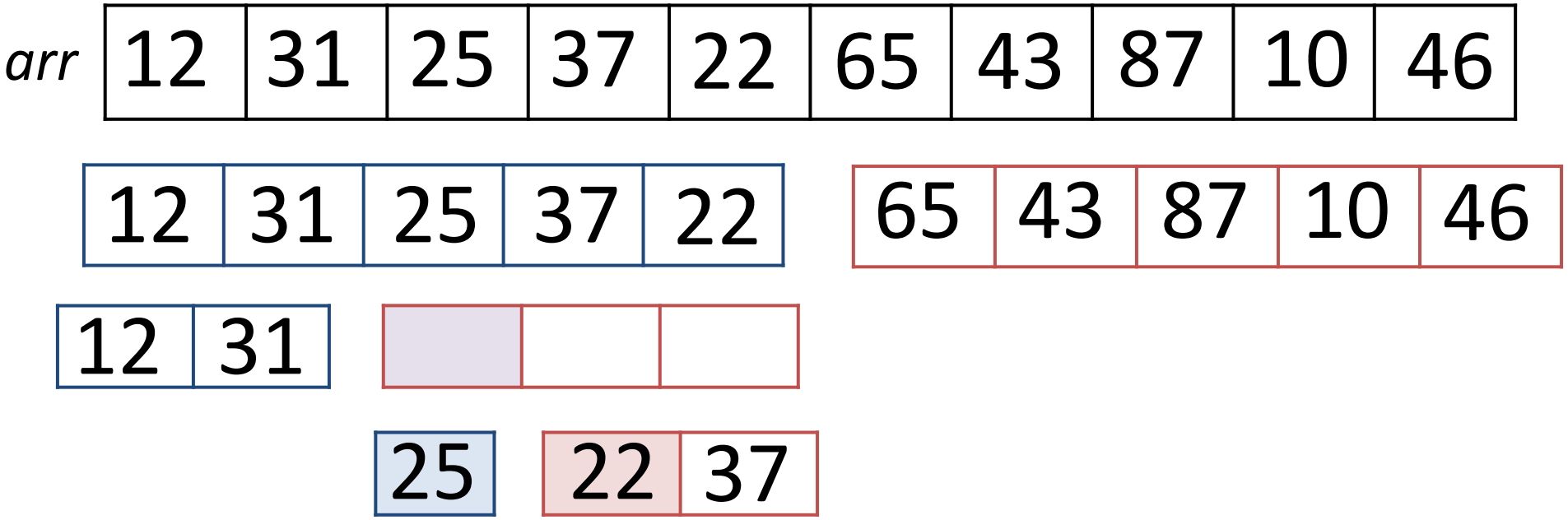
# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order



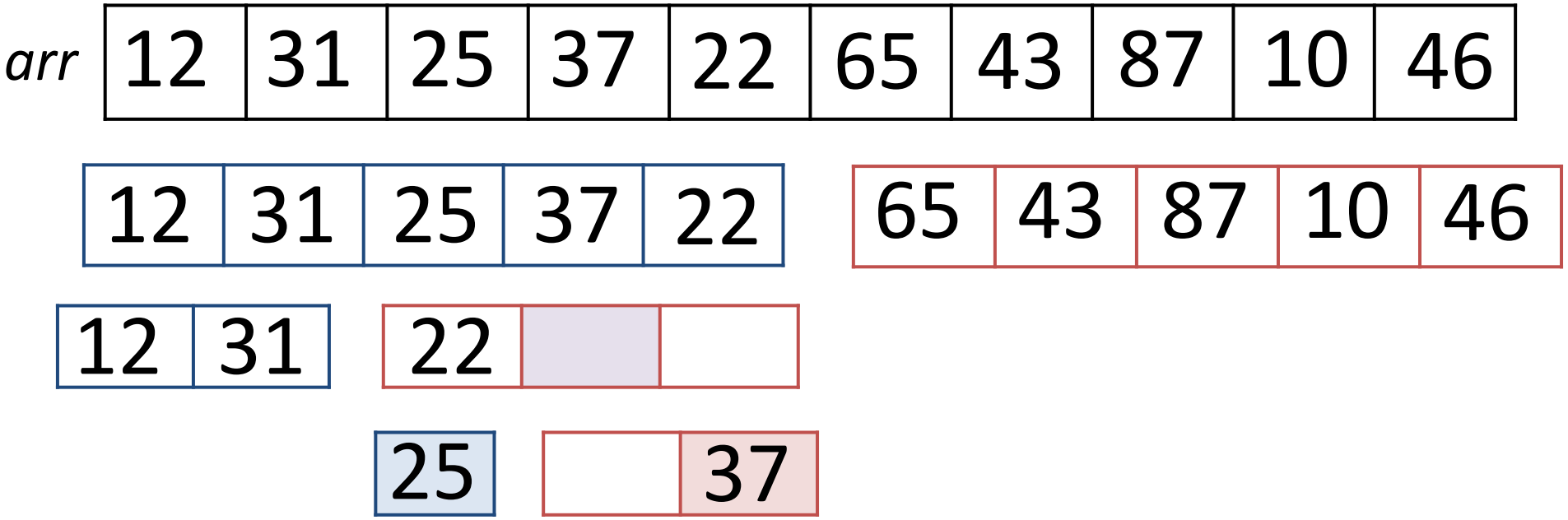
# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order



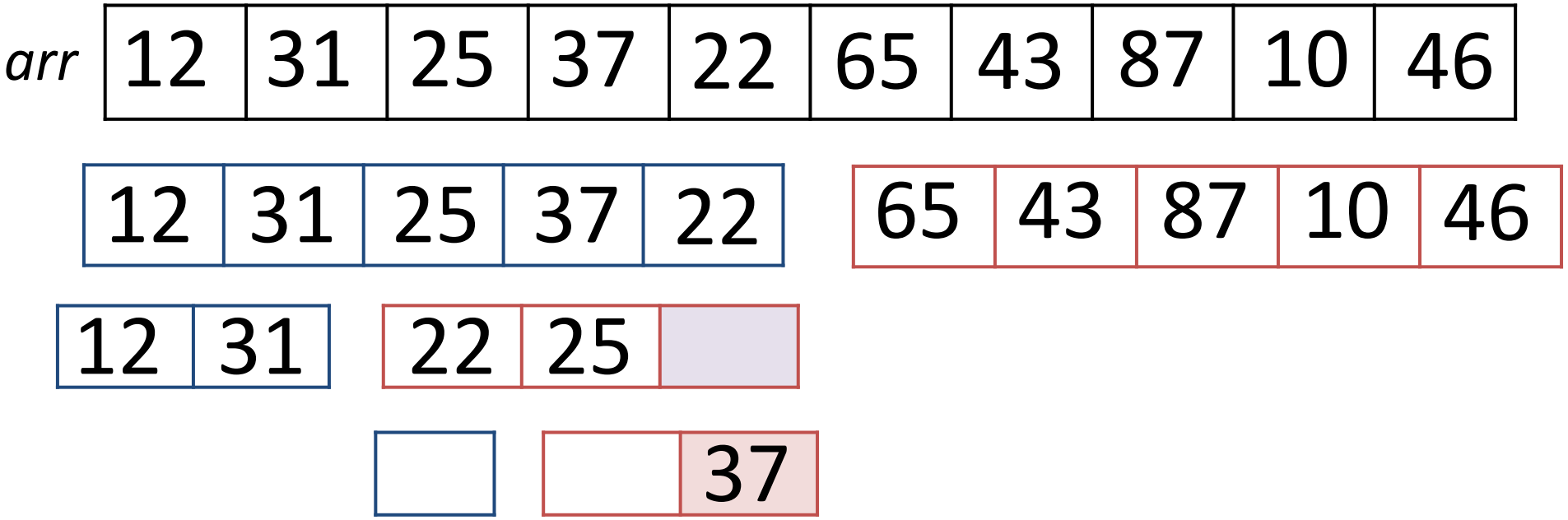
# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order



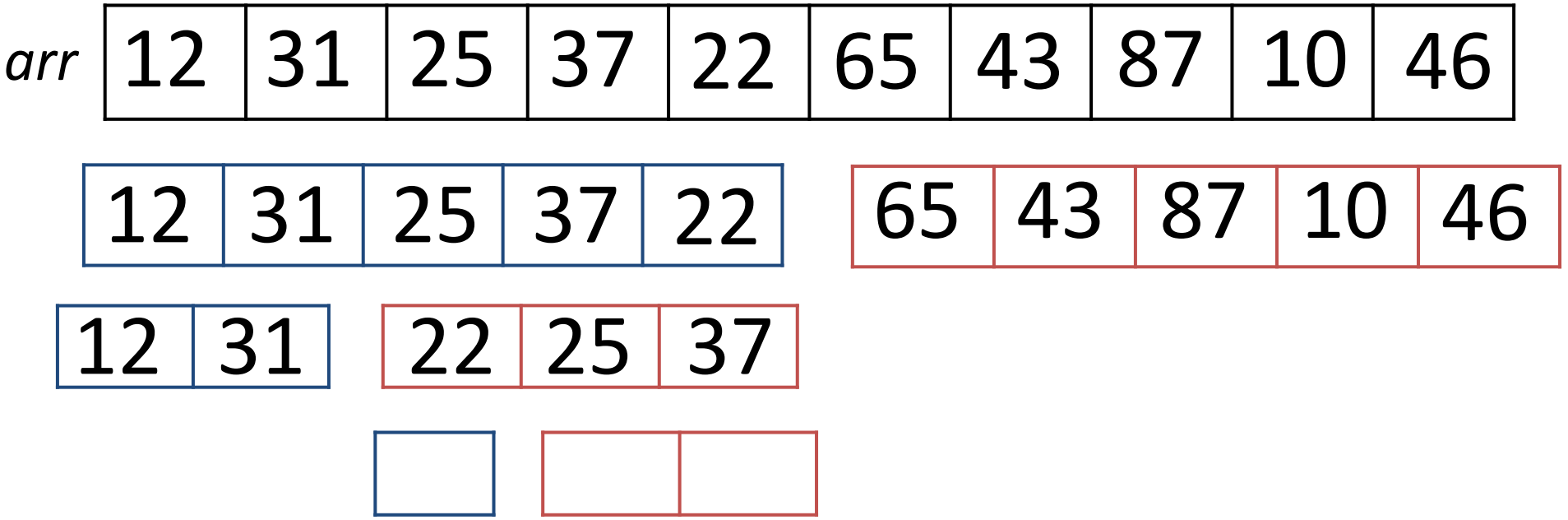
# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order



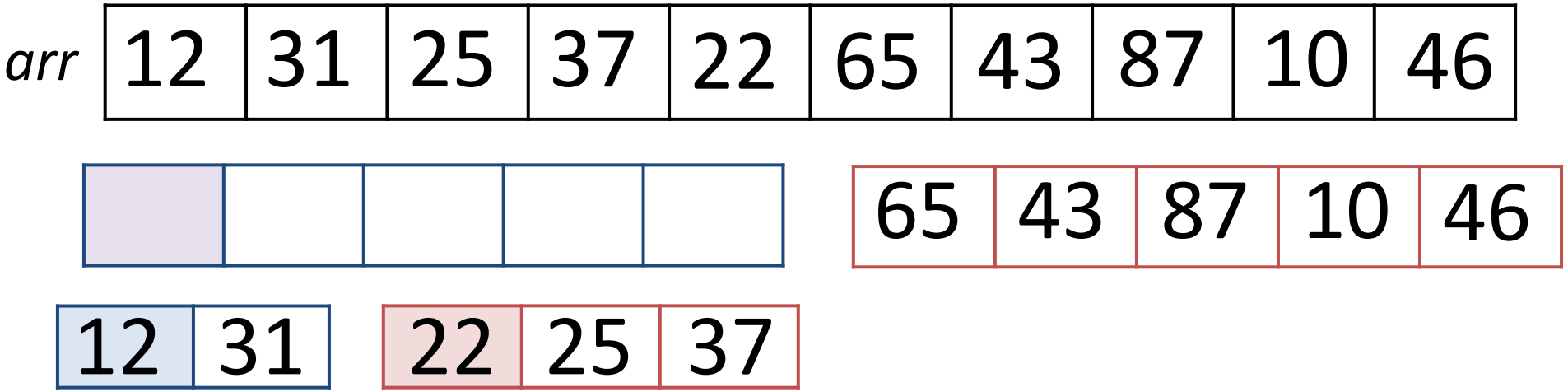
# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order



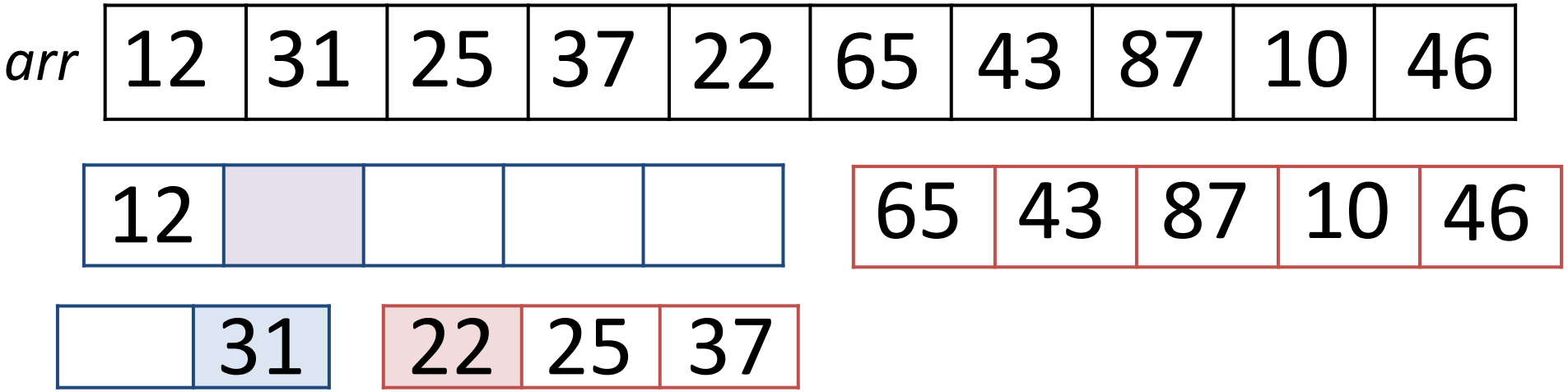
# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order



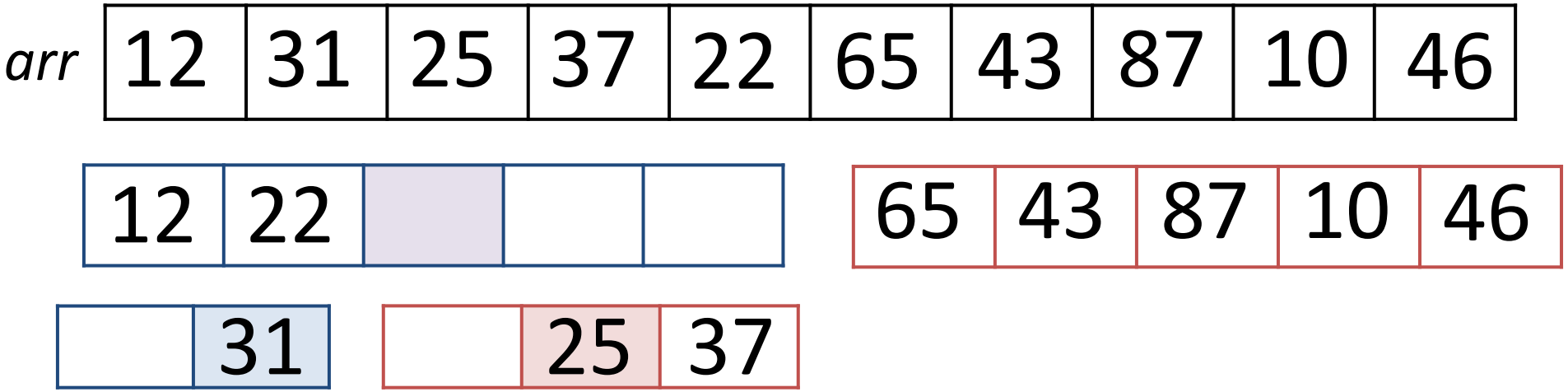
# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order



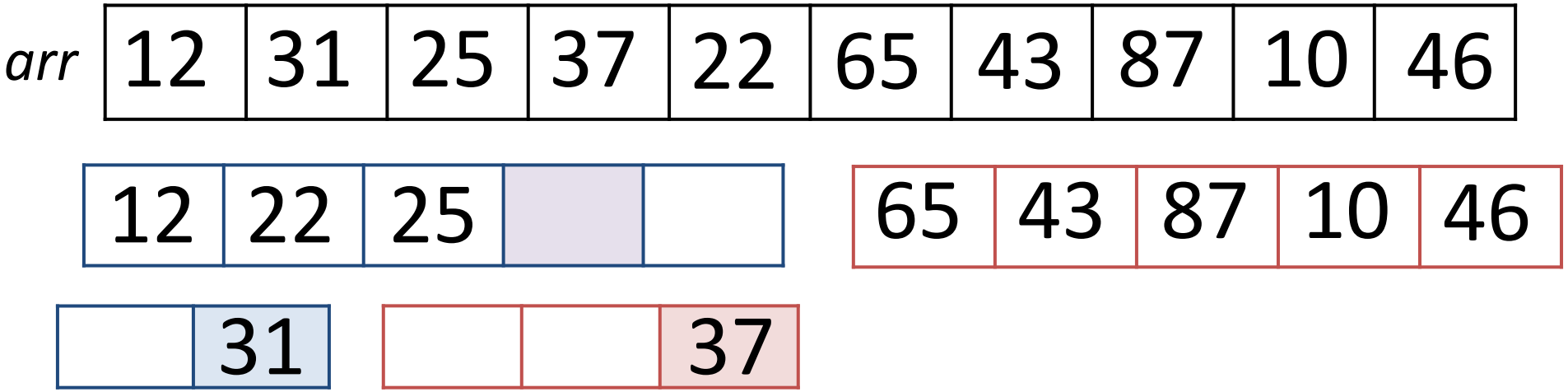
# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order



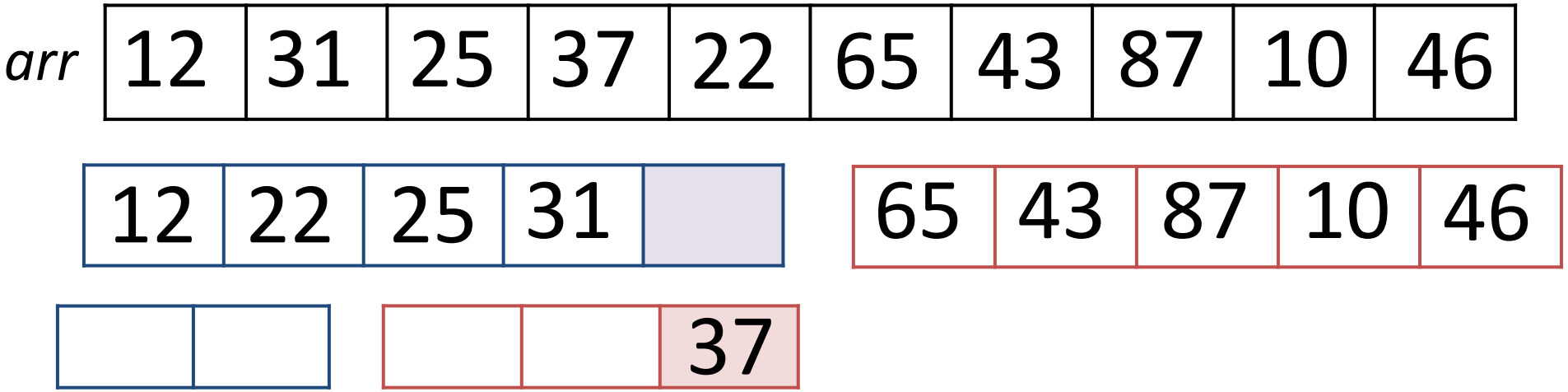
# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order



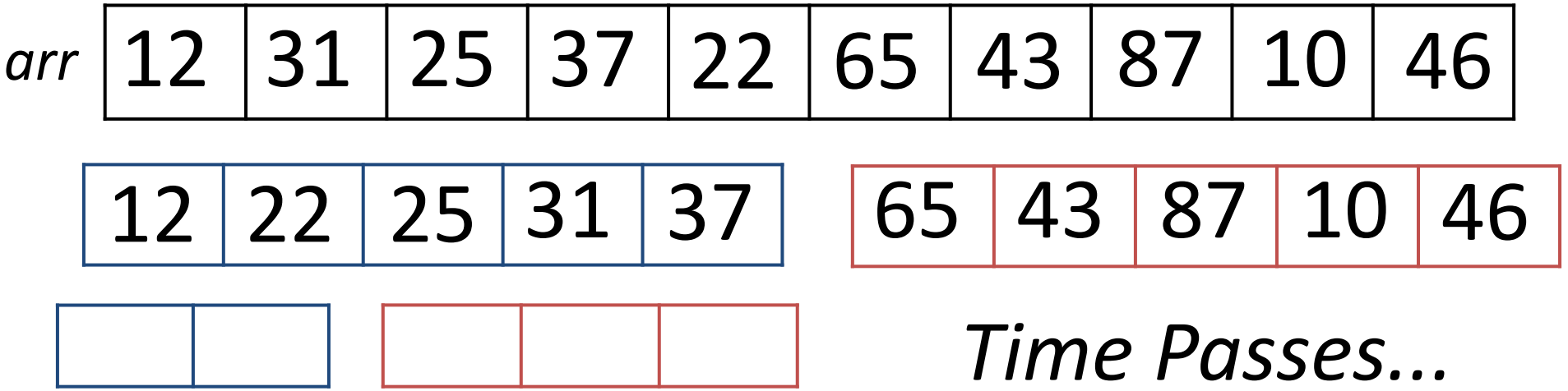
# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order



# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order



# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order

| *arr* | 12 | 31 | 25 | 37 | 22 | 65 | 43 | 87 | 10 | 46 |
|---|---|---|---|---|---|---|---|---|---|---|

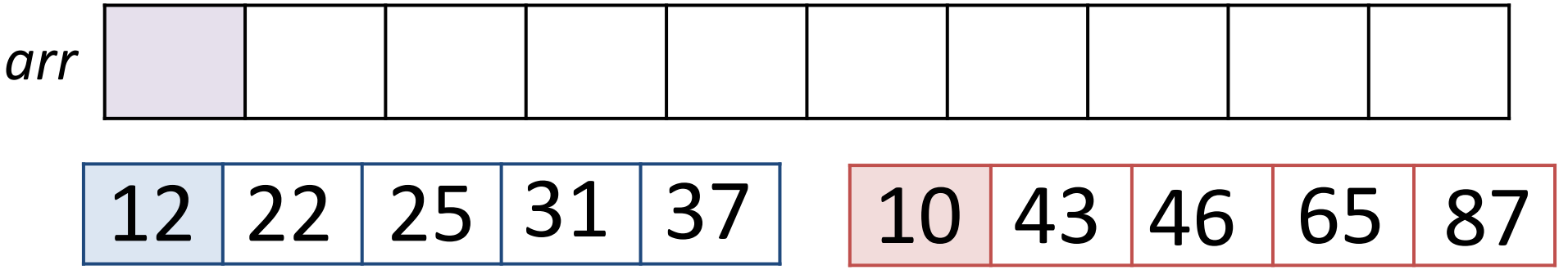
| 12 | 22 | 25 | 31 | 37 |
|---|---|---|---|---|

| 10 | 43 | 46 | 65 | 87 |
|---|---|---|---|---|

*Time Passes...*

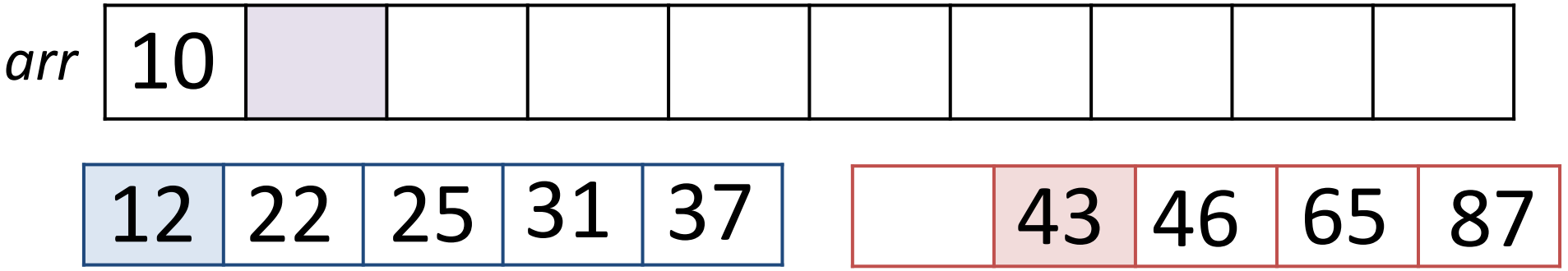
# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order



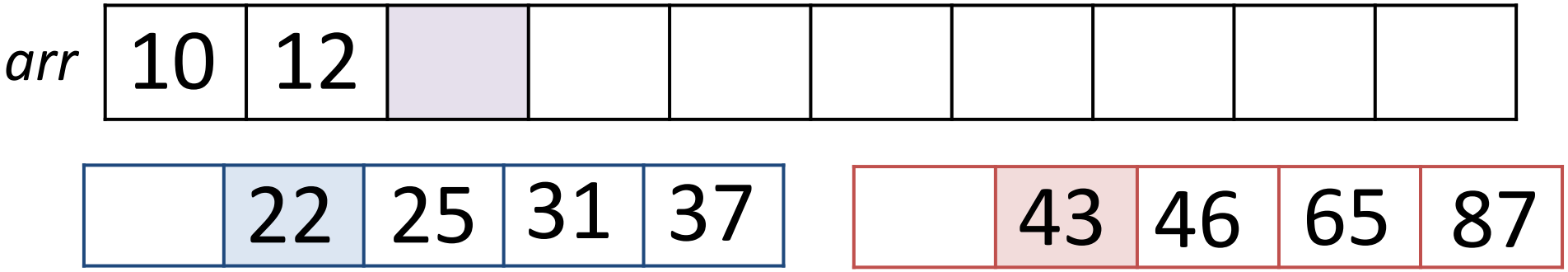
# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order



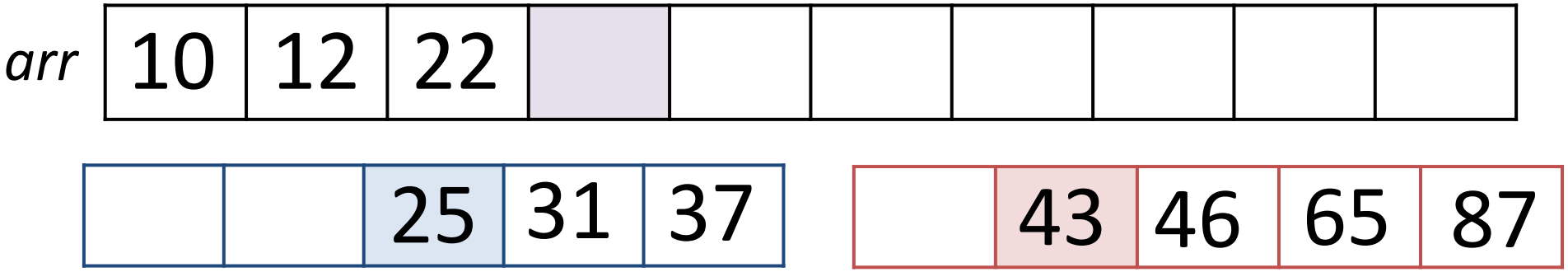
# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order



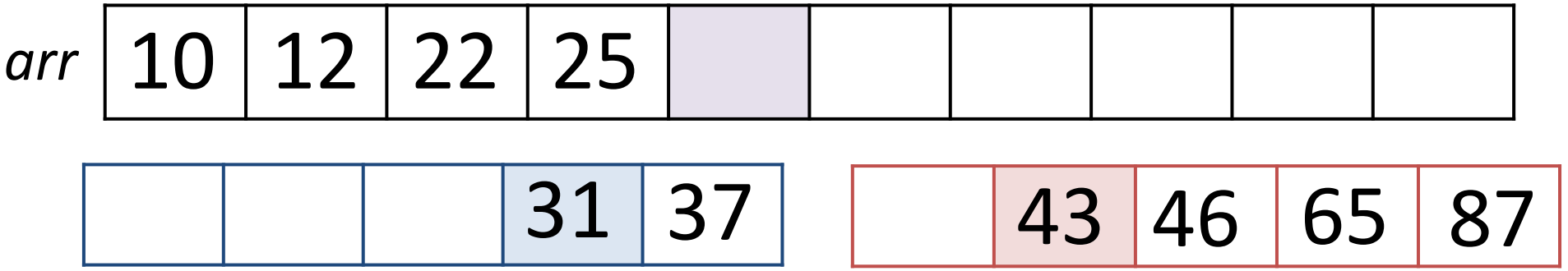
# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order



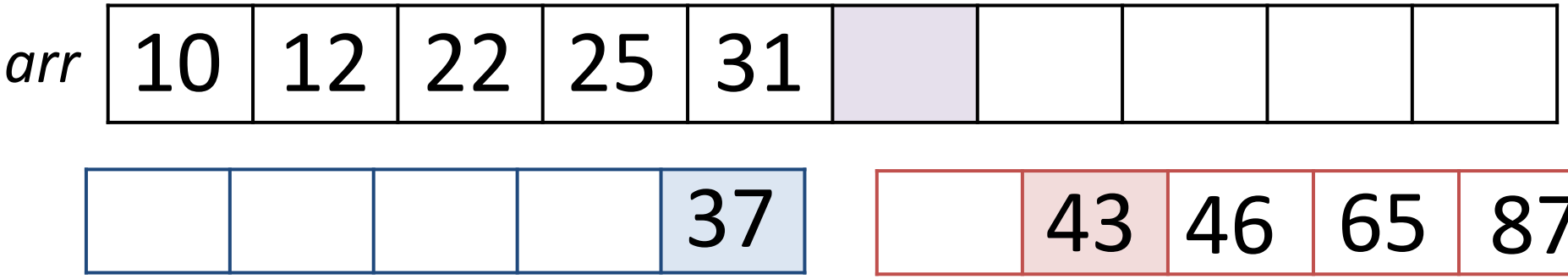
# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order



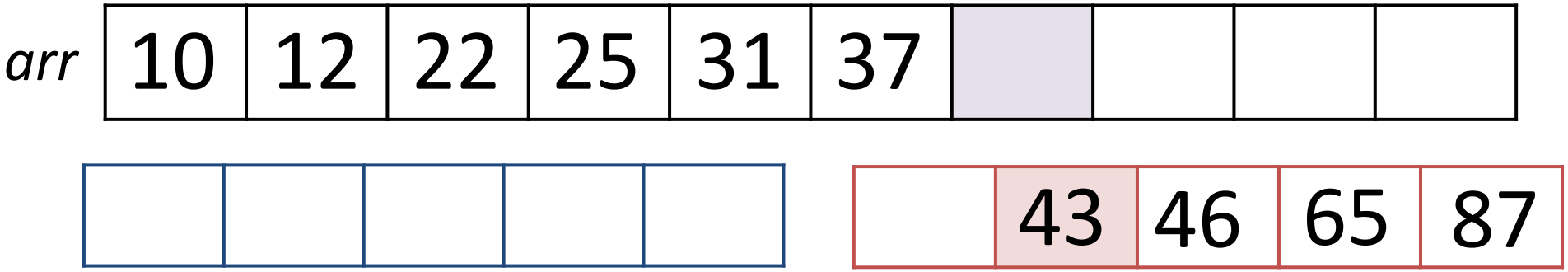
# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order



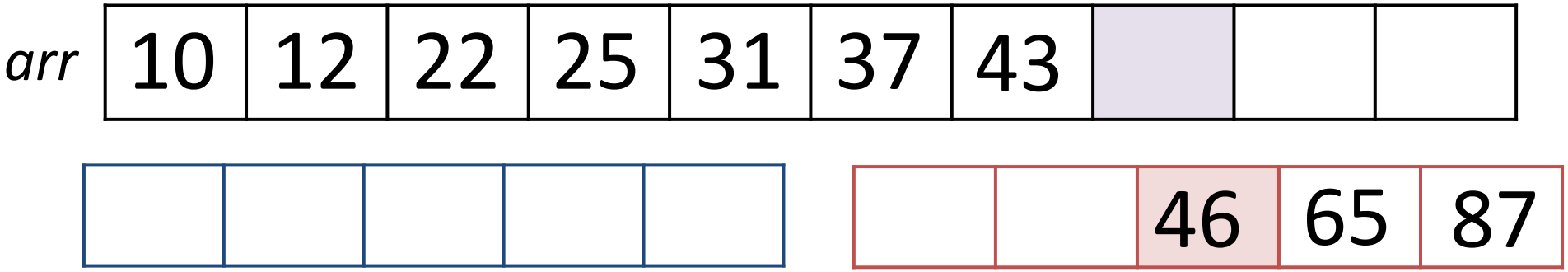
# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order



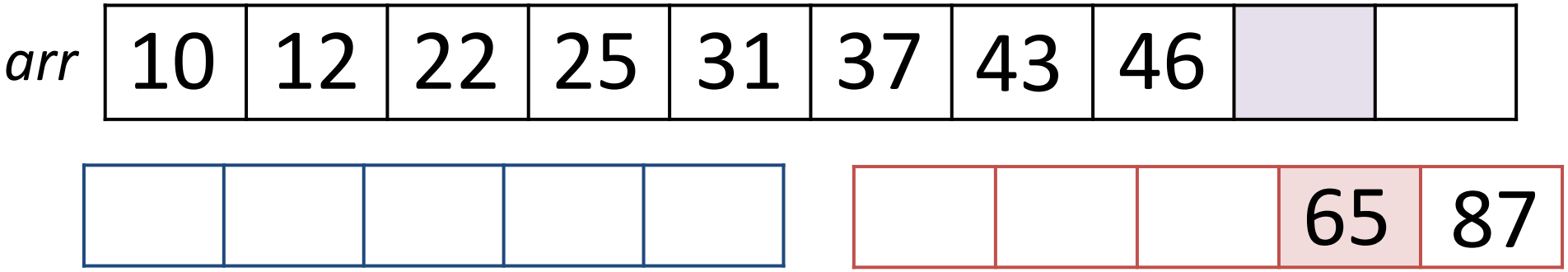
# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order



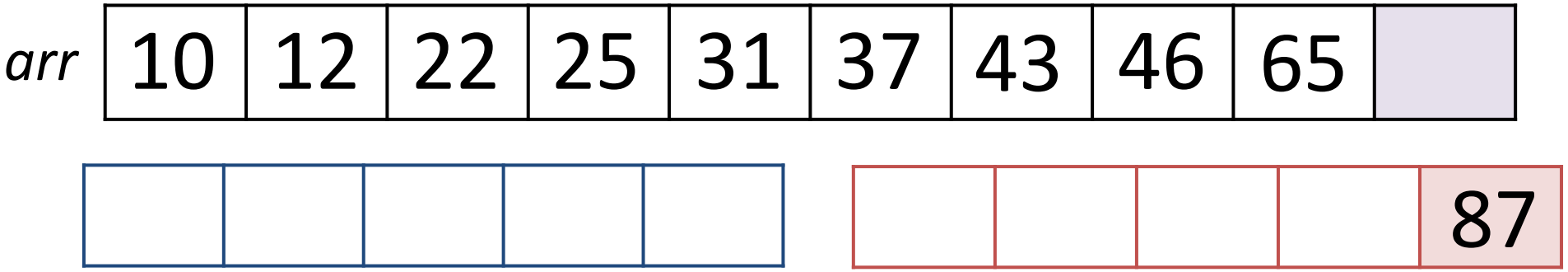
# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order



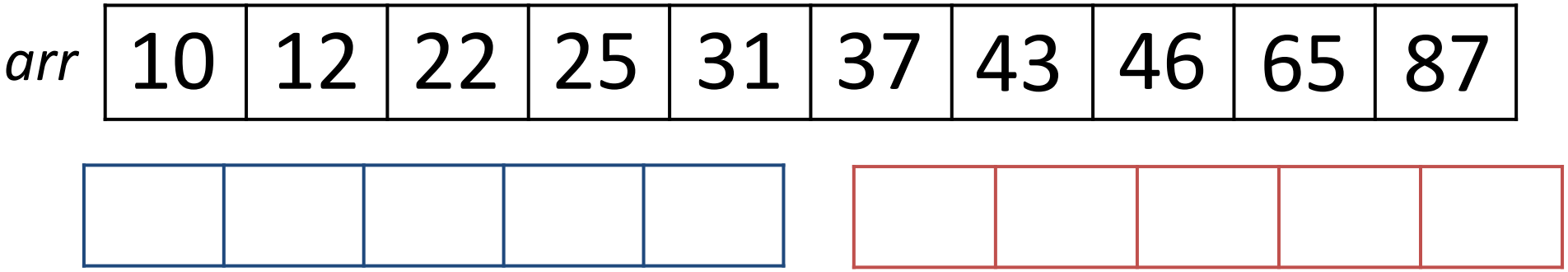
# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order



# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order



# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order

Merge Sort:

- Merge sort the left half
- Merge sort the right half
- Merge the two sorted lists

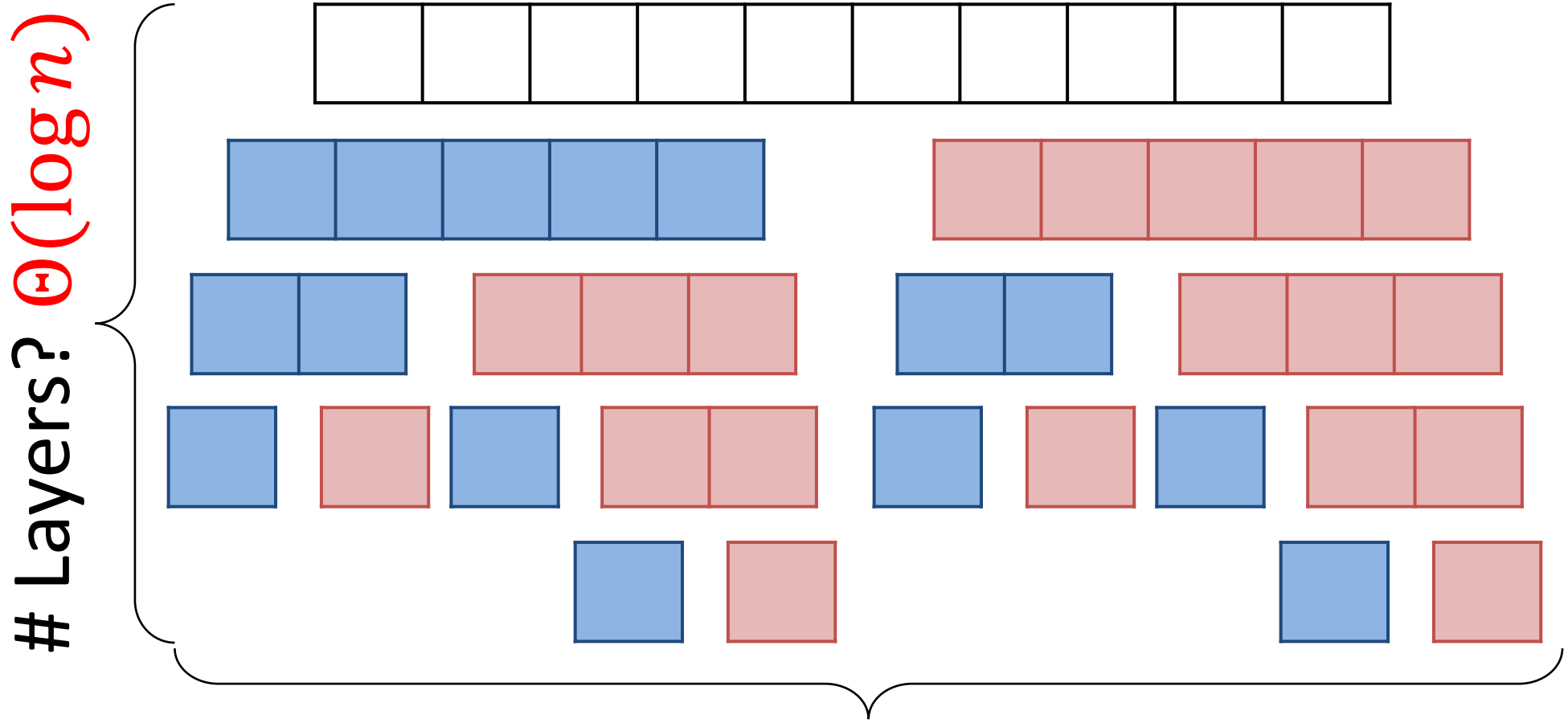
Best case?

Best-case complexity?

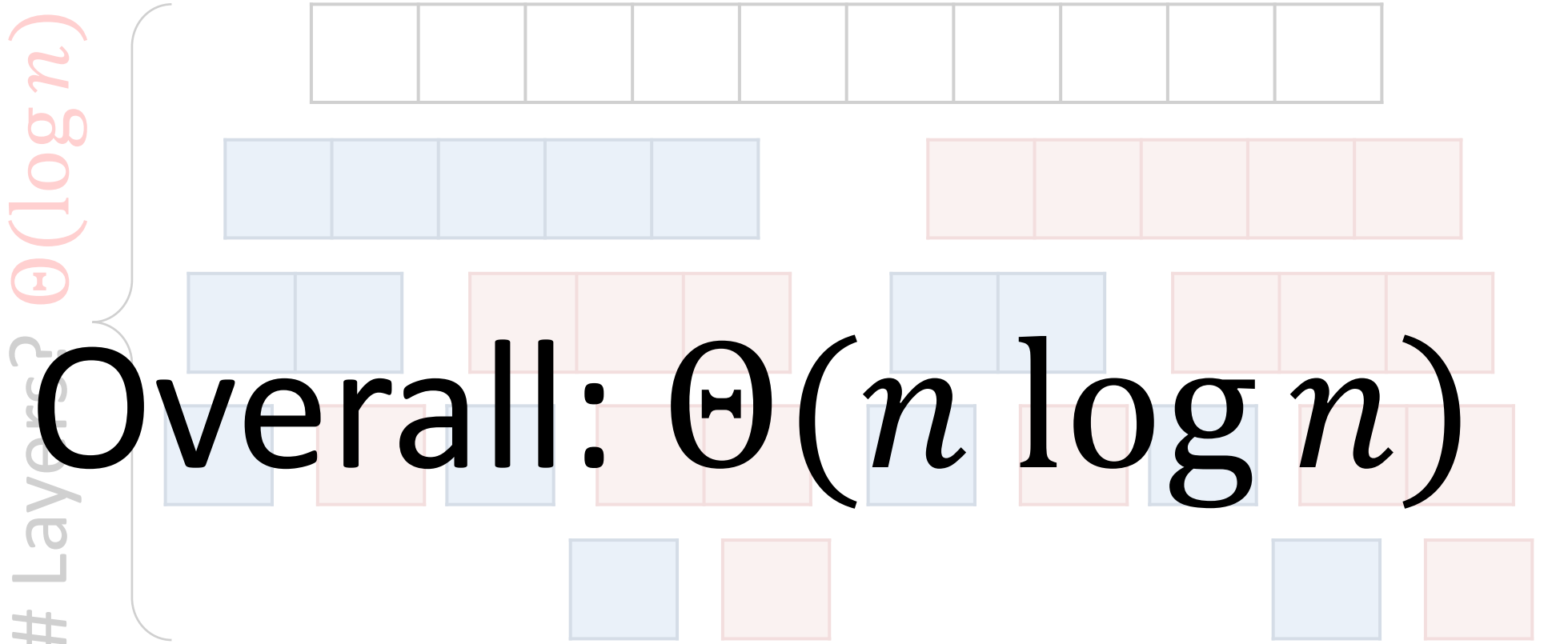
Worst case?

Worst-case complexity?

# Merge Sort: Analysis



# Merge Sort: Analysis




Overall: $\Theta(n \log n)$

# Sorting an Array

Arrange elements of size *n* array *arr* in increasing order

Merge Sort:

- Merge sort the left half
- Merge sort the right half
- Merge the two sorted lists

Best case?

All cases

Best-case complexity?

$\Theta(n \log n)$

Worst case?

All cases

Worst-case complexity?

$\Theta(n \log n)$

# Describing the Complexity

- The complexity is $\Theta(n \log n)$
- The best-case complexity is $\Theta(n \log n)$
- The worst-case complexity is $\Theta(n \log n)$
- The complexity is $O(n \log n)$
- The complexity is $\Omega(n \log n)$

- No matter what, it will take 30 min.
- With no traffic, it will take 30 min.
- At rush hour, it will take 30 min.
- It will take at most 30 min.
- It will take at least 30 min.

# Summary

- Merge Sort
  - Arrange the elements of an array in sorted order
  - Merge sort the left half
  - Merge sort the right half
  - Merge the two sorted arrays
- Complexity: $\Theta(n \log n)$

# Coming Up…

Average case analysis

# Average Case Analysis

# What's the Problem?

- Best-case analysis

  Too optimistic! Maybe the best case is rare.
  Need to plan for the worst.

- Worst-case analysis

  Too pessimistic?
  Maybe the worst case is rare!

# Average Case Analysis

- Goal: get some sense of *typical* complexity

- Assume a probability distribution over all possible cases (with input size *n*)
  - Assign a probability to each case → Make them up (a reasonable model)
    Empirical measurements
    Known randomness in algorithm
- Find the *expected* complexity based on those probabilities

# Expected Value

- Say $X$ can take the values $x_1, x_2, x_3, \ldots$ with some probabilities
- The *expected value* of $X$: a weighted average of the possible values of $X$
  - Each value is weighted by its probability
  - Common values count more
  - Rare values count less
- $\mathrm{E}[X] = \Pr(X = x_1) \cdot x_1 + \Pr(X = x_2) \cdot x_2 + \cdots$
- Average a bunch of random values of X
  - The more random values you use…
  - the closer the average will be to the expected value

# Average Case Analysis

- Goal: get some sense of *typical* complexity
- Assume a probability distribution over all possible cases (with input size *n*)
  - Assign a probability to each case
- Find the *expected* complexity based on those probabilities
  - Say $T(case_i)$ is the number of steps used for $case_i$
  - $\mathrm{E}[T(case_i)] = \Pr(case_1) \cdot T(case_1) + \Pr(case_2) \cdot T(case_2) + \cdots$

# Searching an Array

Determine whether size *n* array *arr* contains item *x*

```
LINEAR-SEARCH (arr, n, x)
  for each value of i from 0 to n-1
    if arr[i] == x, return true
  if loop completes, return false
```

Best case complexity:

$\Theta(1)$

Worst case complexity:

$\Theta(n)$

Average case complexity:

# Searching an Array: Average Case

- Probability model: Let's assume…
  - The item is in the array
  - Every location is equally likely
- Cases:
  - Index 0
    - Probability: $\frac{1}{n}$
    - Number of steps: 1
  - Index 1
    - Probability: $\frac{1}{n}$
    - Number of steps: 2
  - …

Expected Complexity

$$\frac{1}{n}(1) + \frac{1}{n}(2) + \cdots + \frac{1}{n}(n)$$

$$= \sum_{i=1}^{n} \frac{1}{n}(i) = \frac{1}{n}\sum_{i=1}^{n}(i)$$

$\frac{n(n+1)}{2}$

$$= \frac{1}{\cancel{n}}\frac{\cancel{n}(n+1)}{2} = \frac{(n+1)}{2} = \Theta(n)$$

Same as the worst case!

# Searching an Array

Determine whether size *n* array *arr* contains item *x*

```
LINEAR-SEARCH (arr, n, x)
  for each value of i from 0 to n-1
    if arr[i] == x, return true
  if loop completes, return false
```

Best case complexity:

$\Theta(1)$

Worst case complexity:

$\Theta(n)$

Average case complexity:

$\Theta(n)$

# Searching a Binary Search Tree

- Binary Search Tree
  - Best-Case Complexity: $\Theta(1)$
  - Worst-Case Complexity: $\Theta(n)$
  - Average-Case Complexity: $\Theta(\log n)$
    - All insert orders equally likely
    - All item positions equally likely

$O(n)$

- BST w/ Randomized Insert
  - Best-Case Complexity: $\Theta(1)$
  - Worst-Case Complexity: $\Theta(n)$
  - Average-Case Complexity: $\Theta(\log n)$
    - No assumptions about insert order!
    - All item positions equally likely

- Red-Black Tree
  - Best-Case Complexity: $\Theta(1)$
  - Worst-Case Complexity: $\Theta(\log n)$
  - Average-Case Complexity: $\Theta(\log n)$
    - All item positions equally likely

Downsides?
- More space overhead
- More time overhead
- More complicated

# Sorting an Array

- Merge Sort
  - Best-Case Complexity: $\Theta(n \log n)$
  - Worst-Case Complexity: $\Theta(n \log n)$
  - Average-Case Complexity: $\Theta(n \log n)$

- Quicksort
  - Best-Case Complexity: $\Theta(n \log n)$
  - Worst-Case Complexity: $\Theta(n^2)$
  - Average-Case Complexity: $\Theta(n \log n)$
    - All initial orders equally likely

Worst when array is already sorted!

More common in practice!
- Can pre-jumble to satisfy probabilistic assumption
- Less overhead

# Summary

- Average Case Analysis
  - Find the “typical” complexity
    - Account for how common/rare various cases are
  - Find the expected complexity, based on some probabilities for various cases
- Linear search
  - Average case complexity: $\Theta(n)$
- BST Comparisons
  - Randomized insert gives $\Theta(\log n)$ search no matter how items are inserted
  - Red-Black Trees give $O(\log n)$ search
- Sorting Algorithms
  - Merge Sort guarantees $\Theta(n \log n)$ time
  - Quicksort gives $\Theta(n \log n)$ on average, is usually faster in practice

# Coming Up…

Amortized analysis