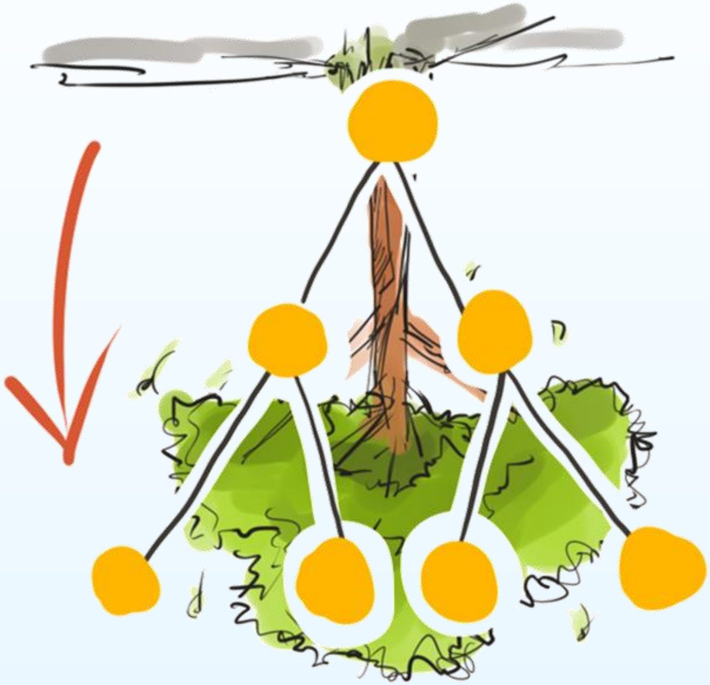


Lecture 12: Searching and Sorting Algorithms

01204212 Abstract Data Types and Problem Solving

Department of Computer Engineering
Faculty of Engineering, Kasetsart University
Bangkok, Thailand.

Department of Computer Engineering Kasetsart University

MIKE LABORATORY

Outline

- Searching Algorithms
  - Linear Search
  - Binary Search

- Sorting Algorithms
  - Selection Sort
  - Insertion Sort
  - Bubble Sort
  - Merge Sort
  - Quick Sort
  - …

Searching and Sorting

- Fundamental problems in computer science and programming
- Sorting done to make searching easier
- Multiple different algorithms to solve the same problem
  - How do we know which algorithm is better?
- Examples will use arrays of integers to illustrate algorithms
- We will look at searching first!

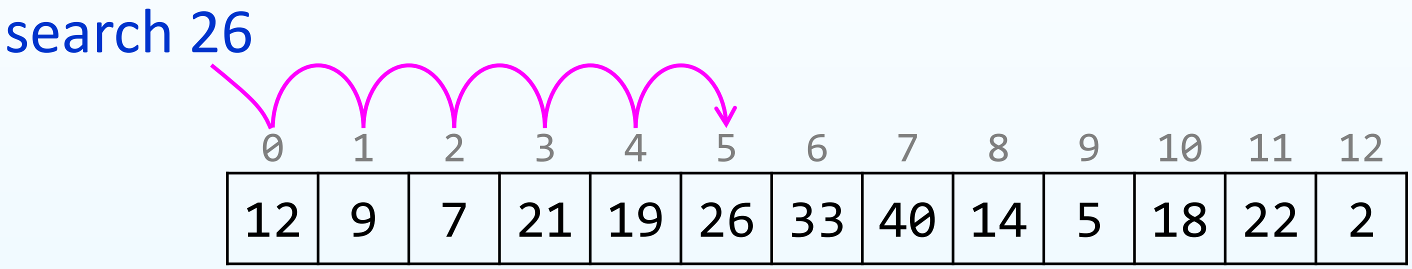
Searching

- We have learned ADTs that provide a search operation on different data structures such as linked lists, arrays, BSTs/AVL trees, and hash tables

- However, in this lecture, the search scheme is operated on an array of integers:
  - Given a list of data, searching is to find the location of a particular value or report that value is not present
  - Only focus on linear search and binary search

Linear Search

Linear search is a method that sequentially checks each element of the list until a match is found or the whole list has been searched



search 26

0	1	2	3	4	5	6	7	8	9	10	11	12
12	9	7	21	19	26	33	40	14	5	18	22	2

```
int linear_search(int arr[], int n, int value) {
  int i;
  for (i=0; i<n; i++)
    if (arr[i] == value)
      return i;
  return -1;
}
```

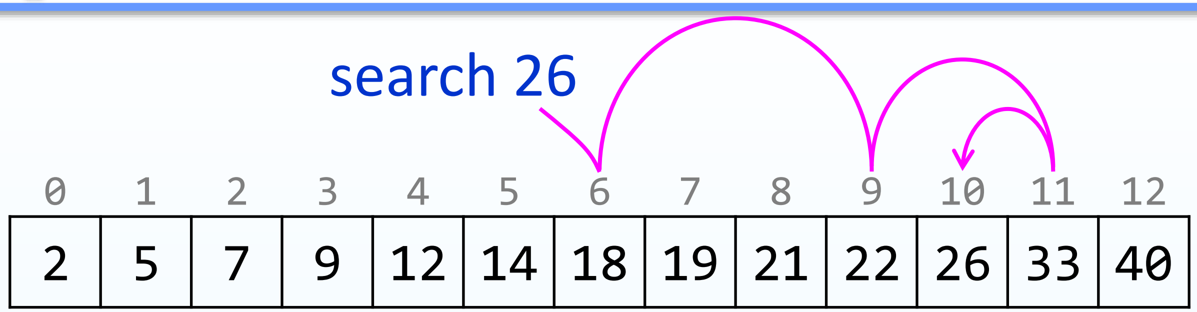
Running time = $O(n)$

Binary Search

Binary search is a method that finds the position of a target value within a sorted array

1. Start at the middle
2. Check that element:
   2.1 If it is match; return that position
   2.2 If the value looking for is less than; recursively check at the middle of the left-half interval
   2.3 If the value looking for is greater than; recursively check at the middle of the right-half interval
3. Terminate when the value is found, or the interval is empty

Binary Search


search 26

0	1	2	3	4	5	6	7	8	9	10	11	12
2	5	7	9	12	14	18	19	21	22	26	33	40

```
int binary_search(int arr[], int l, int r, int value) {
  int m;
  if (l <= r) {
    m = l + (r-l)/2; //same as (l+r)/2 but avoid overflow
    if (arr[m] == value)
      return m;
    if (arr[m] > value)
      return binary_search(arr, l, m-1, value); //left
    return binary_search(arr, m+1, r, value);   //right
  }
  return -1;
}
```

Running time

$$T(n) = T\left(\frac{n}{2}\right) + 1 = O(\log n)$$

Outline

- Searching Algorithms
  - Linear Search
  - Binary Search

- Sorting Algorithms
  - Selection Sort
  - Insertion Sort
  - Bubble Sort
  - Merge Sort
  - Quick Sort
  - …

Sorting

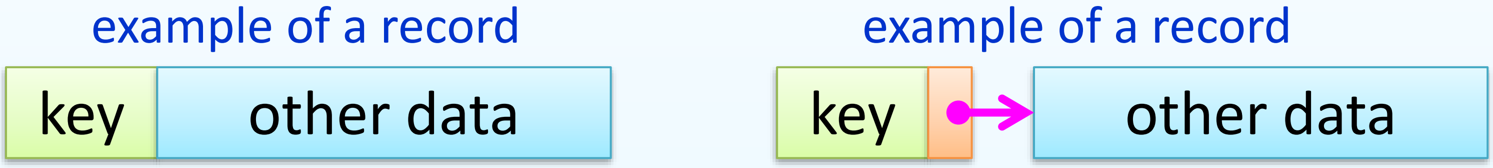
- Sorting is a process that organizes a collection of data into either ascending or descending order

- Formally,
  - Input: A sequence of n numbers $\langle a_1, a_2, \dots, a_n \rangle$
  - Output: A permutation (reordering) $\langle a_1', a_2', \dots, a_n' \rangle$ of the input sequence such that $a_1' \le a_2' \le \cdots \le a_n'$

- Example:
  - Given an input $\langle 6, 3, 1, 7 \rangle$, the algorithm should produce $\langle 1, 3, 6, 7 \rangle$

Structure of Data

- We rarely sort separated values
- Usually, the numbers to be sorted are part of a collection of data called a record
- Each record contains a key that is the value to be sorted



- Note that when the keys are rearranged, the data associated with the keys must also be rearranged (time consuming !!)
- Pointers can be used instead (space consuming !!)

Structure of Data

- We will sort a number of records based on a key:

19991532	Stevenson	Monica	3 Glendridge Ave.
19990253	Redpath	Ruth	53 Belton Blvd.
19985832	Kilji	Islam	37 Masterson Ave.
20003541	Groskurth	Ken	12 Marsdale Ave.
19981932	Carol	Ann	81 Oakridge Ave.
20003287	Redpath	David	5 Glendale Ave.

Numerically by ID Number

19981932	Carol	Ann	81 Oakridge Ave.
19985832	Khilji	Islam	37 Masterson Ave.
19990253	Redpath	Ruth	53 Belton Blvd.
19991532	Stevenson	Monica	3 Glendridge Ave.
20003287	Redpath	David	5 Glendale Ave.
20003541	Groskurth	Ken	12 Marsdale Ave.

Lexicographically by surname, then given name

19981932	Carol	Ann	81 Oakridge Ave.
20003541	Groskurth	Ken	12 Marsdale Ave.
19985832	Kilji	Islam	37 Masterson Ave.
20003287	Redpath	David	5 Glendale Ave.
19990253	Redpath	Ruth	53 Belton Blvd.
19991532	Stevenson	Monica	3 Glendridge Ave.

Why Study Sorting?

- There are a variety of situations that we can encounter:
  - Do we have randomly ordered keys?
  - Are all keys distinct?
  - Need guaranteed performance?
- Examples:
  - Sorting price from lowest to highest
  - Sorting flights from earliest to latest
  - Sorting grades from highest to lowest
  - Sorting songs based on artist, album, playlist, etc.
- Various algorithms are better suited to some of these situations

Some Definitions

- Internal sort
  - The data to be sorted is all stored in the main memory of computer
- External sort
  - Some of the data to be sorted might be stored in some external, slower, device
- In-place sort
  - The amount of extra space required to sort the data is at most $\Theta(1)$ (i.e., fixed number of local variables)

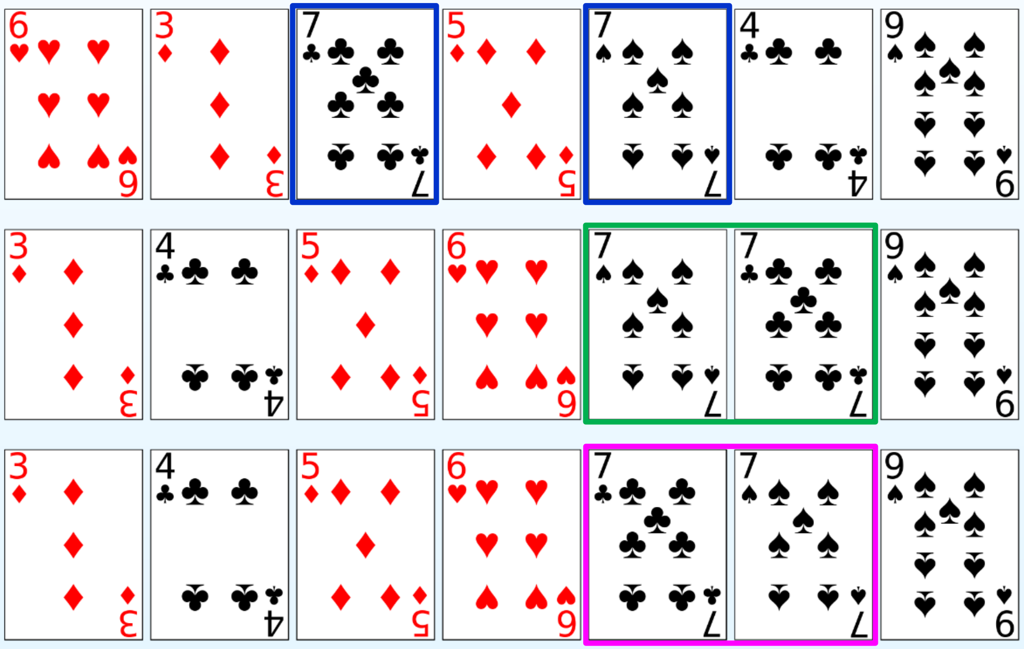
Stability

- A sorting algorithm is stable if whenever there are two records R and S with the same key and with R appearing before S in the original list
  - Preserve relative order of record with equal keys

Original list

Sorted list
without stability

Sorted list
with stability



Some Common Sorting Algorithms

- Selection sort
- Insertion sort
- Bubble sort
- Merge sort
- Quick sort
- Heap sort
- Counting sort
- Radix sort
- Bucket sort

Classification of Sorting Algorithm

Sorting algorithms can be categorized based on various criterions:

- Based on the number of swaps
  - Selection sort requires the minimum number of swaps
- Based on the running time
  - Require $O(n^2)$: selection, insertion, bubble sorts
  - Require $O(n \log n)$: merge, quick, heap sorts
  - Require $O(n)$: counting, radix, bucket sorts
- Based on stability
  - With stability: insertion, bubble, merge sorts
  - Without stability: quick, heap sorts
- Based on extra space requirement
  - In place: selection, insertion, bubble, quick sorts
  - Out place: merge sort

Outline

- Searching Algorithms
  - Linear Search
  - Binary Search

- Sorting Algorithms
  - Selection Sort
  - Insertion Sort
  - Bubble Sort
  - Merge Sort
  - Quick Sort
  - …

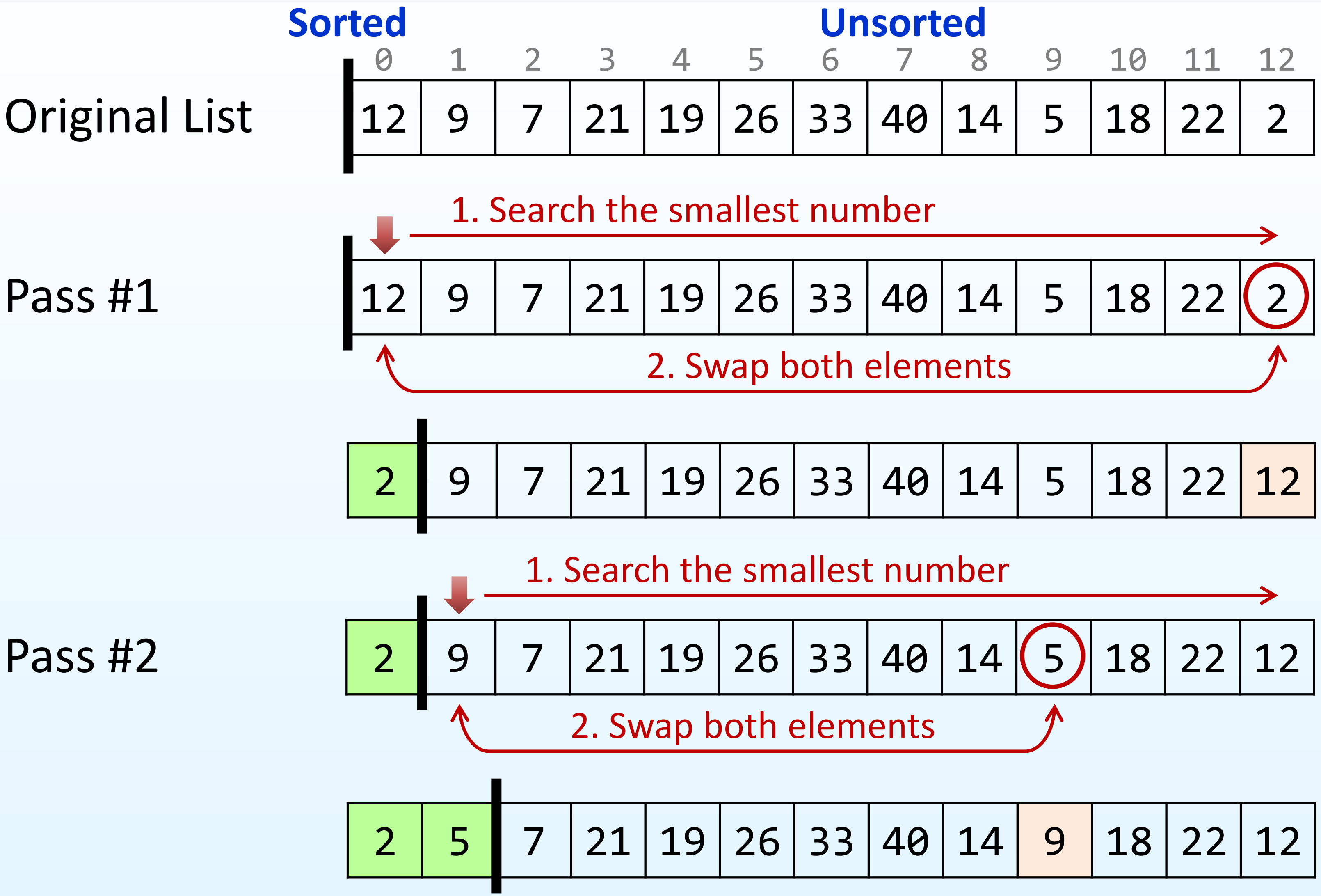
Selection Sort

Selection sort is one of the easiest approaches to sorting

Idea:

- Partition the input list of n elements into a sorted and unsorted part (initially sorted part is empty)
- Select the smallest element and swap it with the first element of the unsorted part
- Increase the size of the sorted part by one
- Repeat this $n - 1$ times to sort the list

Example



Sorted

Unsorted

Original List

0	1	2	3	4	5	6	7	8	9	10	11	12
12	9	7	21	19	26	33	40	14	5	18	22	2

1. Search the smallest number

Pass #1

12	9	7	21	19	26	33	40	14	5	18	22	2

2. Swap both elements

2	9	7	21	19	26	33	40	14	5	18	22	12

1. Search the smallest number

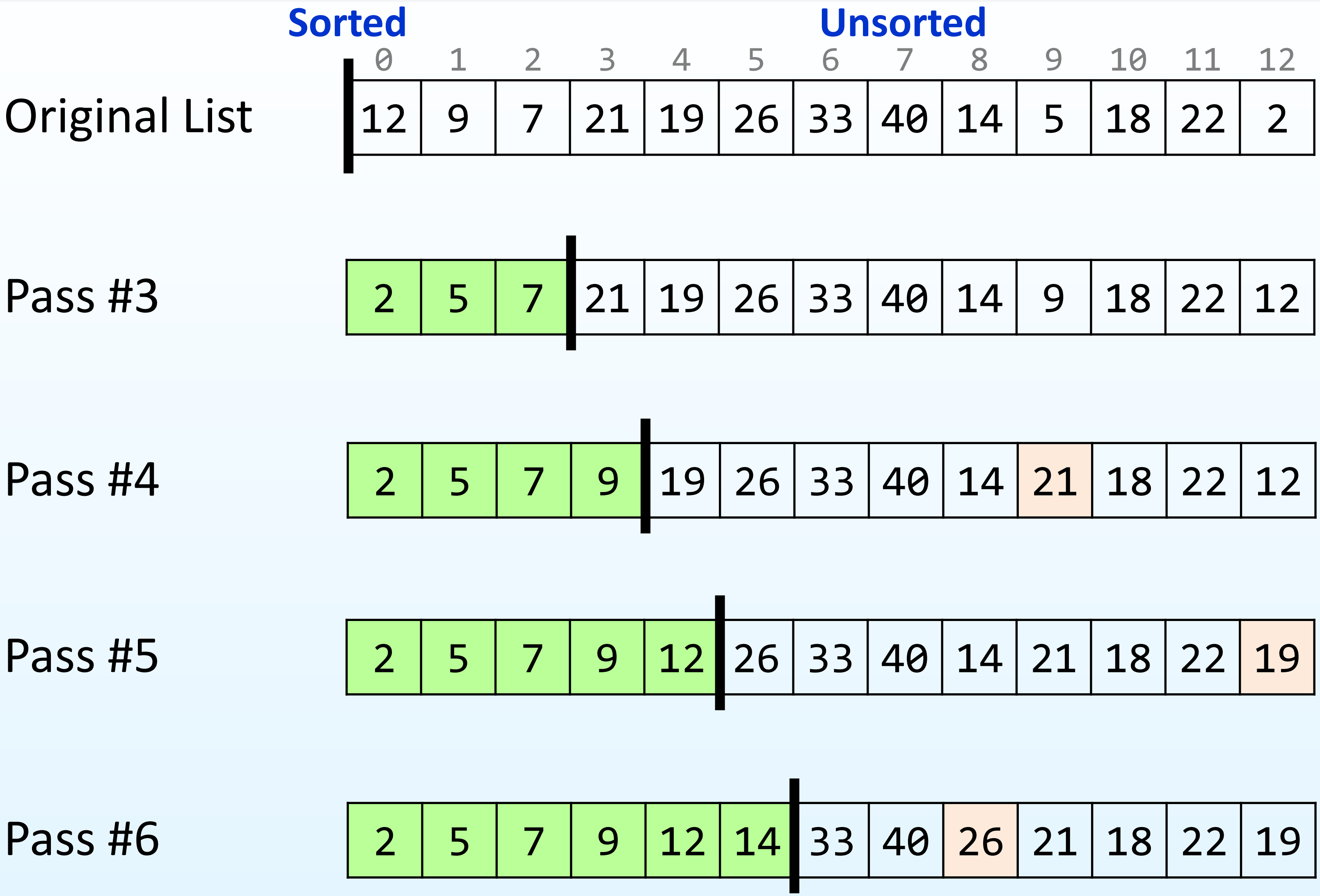
Pass #2

2	9	7	21	19	26	33	40	14	5	18	22	12

2. Swap both elements

2	5	7	21	19	26	33	40	14	9	18	22	12

Example



Sorted

Unsorted

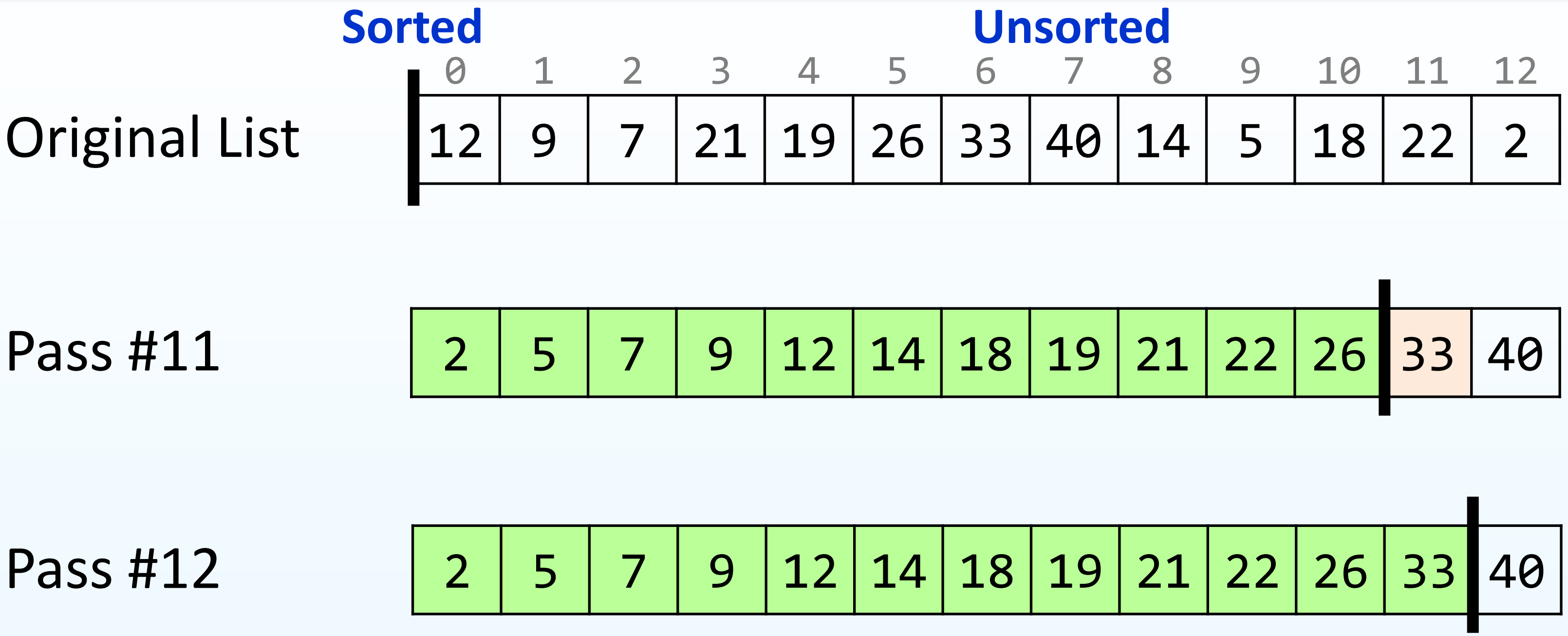
	0	1	2	3	4	5	6	7	8	9	10	11	12
Original List	12	9	7	21	19	26	33	40	14	5	18	22	2
Pass #3	2	5	7	21	19	26	33	40	14	9	18	22	12
Pass #4	2	5	7	9	19	26	33	40	14	21	18	22	12
Pass #5	2	5	7	9	12	26	33	40	14	21	18	22	19
Pass #6	2	5	7	9	12	14	33	40	26	21	18	22	19

Example

Sorted | Unsorted

	0	1	2	3	4	5	6	7	8	9	10	11	12
Original List	12	9	7	21	19	26	33	40	14	5	18	22	2
Pass #7	2	5	7	9	12	14	18	40	26	21	33	22	19
Pass #8	2	5	7	9	12	14	18	19	26	21	33	22	40
Pass #9	2	5	7	9	12	14	18	19	21	26	33	22	40
Pass #10	2	5	7	9	12	14	18	19	21	22	33	26	40

Example



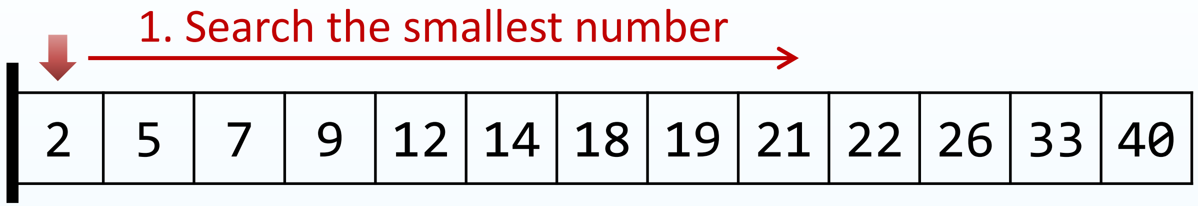
Sorted

Unsorted

	0	1	2	3	4	5	6	7	8	9	10	11	12
Original List	12	9	7	21	19	26	33	40	14	5	18	22	2
Pass #11	2	5	7	9	12	14	18	19	21	22	26	33	40
Pass #12	2	5	7	9	12	14	18	19	21	22	26	33	40

Time Complexity Analysis

- Best case

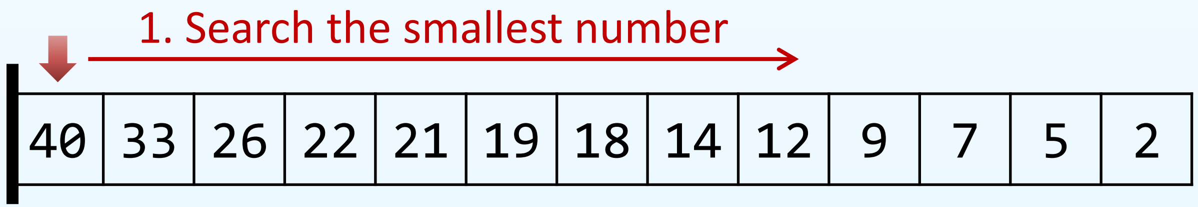


1. Search the smallest number

2	5	7	9	12	14	18	19	21	22	26	33	40

$$T(n) = n + (n-1) + (n-2) + \cdots + 3 + 2 \ = \frac{n(n+1)}{2} - 1 \ = \Omega(n^2)$$

- Worst case



1. Search the smallest number

40	33	26	22	21	19	18	14	12	9	7	5	2

$$T(n) = n + (n-1) + (n-2) + \cdots + 3 + 2 \ = \frac{n(n+1)}{2} - 1 \ = O(n^2)$$

- Average case

$$T(n) = \Theta(n^2)$$

Extra Space Requirement

- Selection sort is an in-place algorithm

- It performs all computation in the original array and no other array is used

- Hence, the extra space works out to be $O(1)$

Important Notes

- Selection sort is not a very efficient algorithm when data set are large
  - This indicated by the average and worst case complexities
- However, selection sort uses minimum number of swap operations $O(n)$ among all the sorting algorithms
- Traditional selection sort is not a stable algorithm, but we can modify for stable one

Outline

- Searching Algorithms
  - Linear Search
  - Binary Search

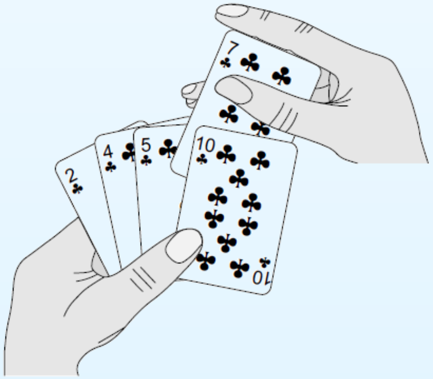
- Sorting Algorithms
  - Selection Sort
  - Insertion Sort
  - Bubble Sort
  - Merge Sort
  - Quick Sort
  - …

Insertion Sort

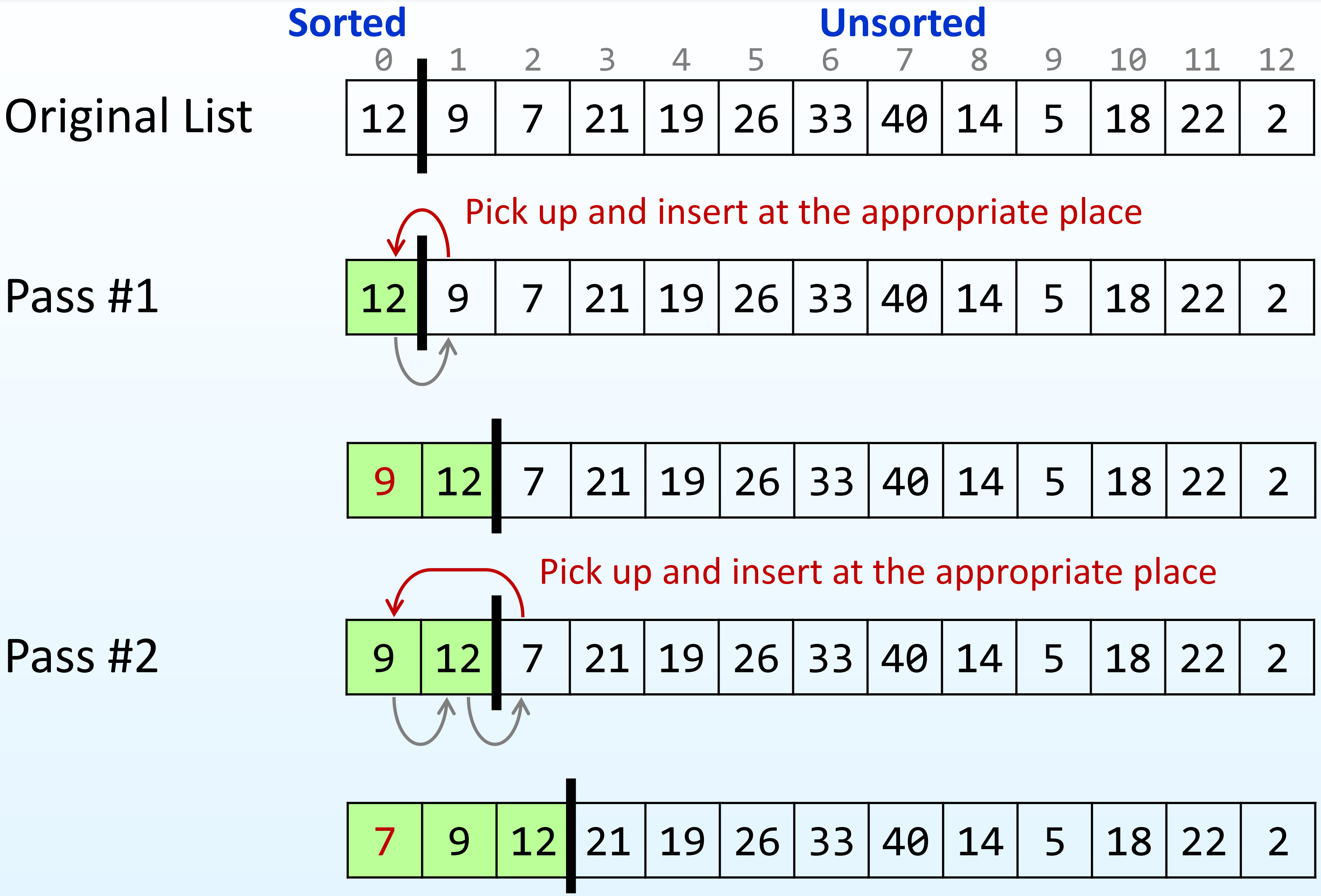
Insertion sort is the most common sorting technique used by card players

Idea:

- Partition the input list of n elements into a sorted and unsorted part
  - Initial sorted part with the first element of the list
- In each pass, the first element of the unsorted part is picked up, transferred to the sorted sub-list, and inserted at the appropriate place
- Repeat at most $n - 1$ passes to sort the list



Example



Sorted Unsorted

	0	1	2	3	4	5	6	7	8	9	10	11	12
Original List	12	9	7	21	19	26	33	40	14	5	18	22	2

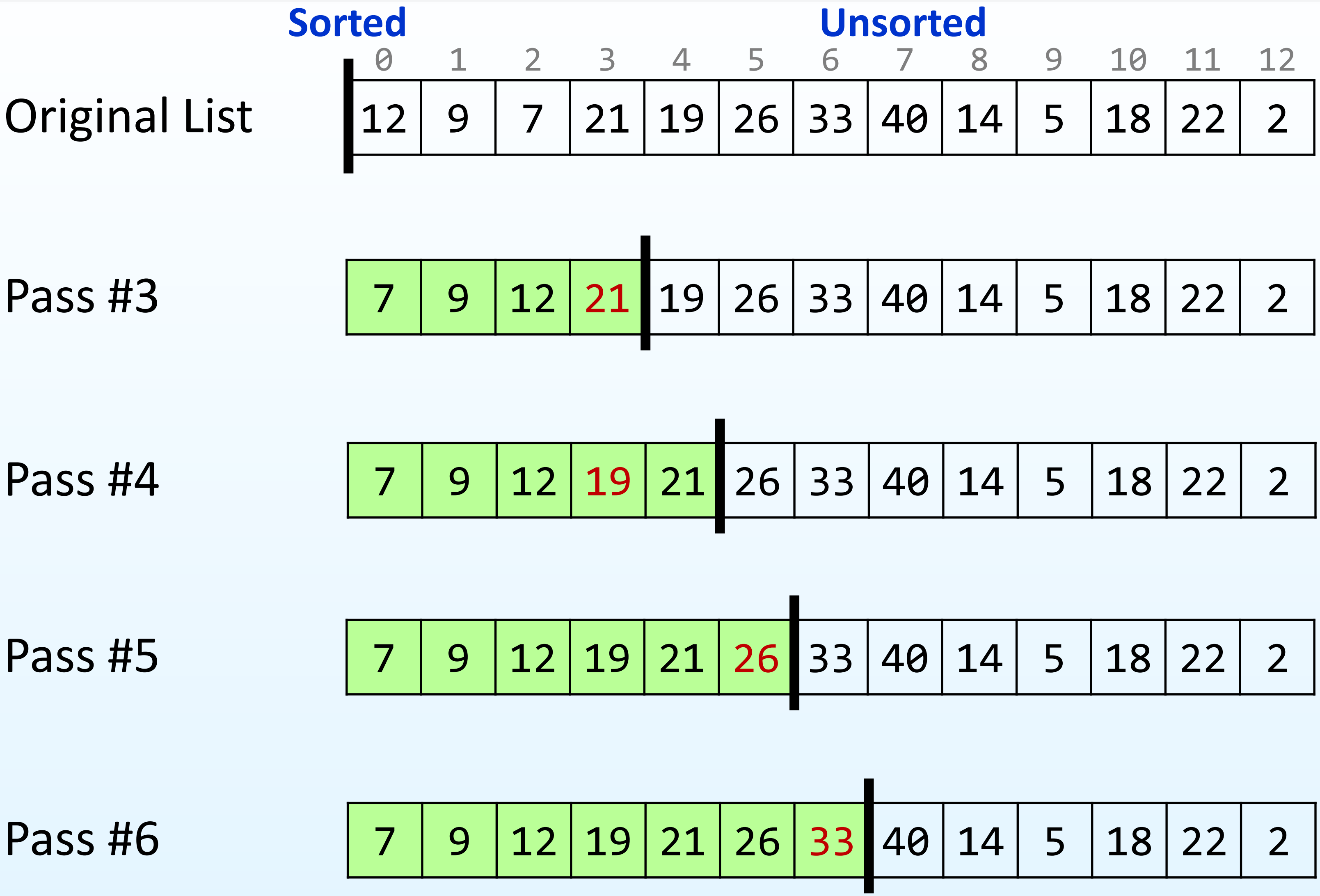
Pick up and insert at the appropriate place

Pass #1	12	9	7	21	19	26	33	40	14	5	18	22	2
	9	12	7	21	19	26	33	40	14	5	18	22	2

Pick up and insert at the appropriate place

Pass #2	9	12	7	21	19	26	33	40	14	5	18	22	2
	7	9	12	21	19	26	33	40	14	5	18	22	2

Example



Sorted

Unsorted

	0	1	2	3	4	5	6	7	8	9	10	11	12
Original List	12	9	7	21	19	26	33	40	14	5	18	22	2
Pass #3	7	9	12	21	19	26	33	40	14	5	18	22	2
Pass #4	7	9	12	19	21	26	33	40	14	5	18	22	2
Pass #5	7	9	12	19	21	26	33	40	14	5	18	22	2
Pass #6	7	9	12	19	21	26	33	40	14	5	18	22	2

Example

Sorted

Unsorted

	0	1	2	3	4	5	6	7	8	9	10	11	12
Original List	12	9	7	21	19	26	33	40	14	5	18	22	2
Pass #7	7	9	12	19	21	26	33	40	14	5	18	22	2
Pass #8	7	9	12	14	19	21	26	33	40	5	18	22	2
Pass #9	5	7	9	12	14	19	21	26	33	40	18	22	2
Pass #10	5	7	9	12	14	18	19	21	26	33	40	22	2

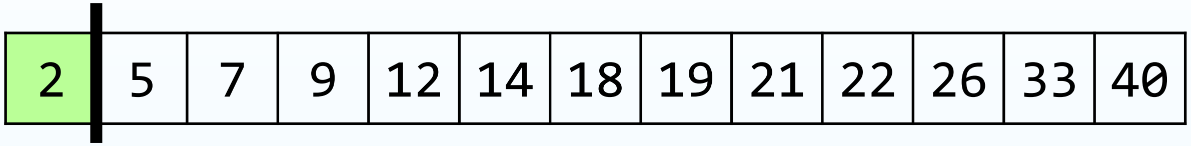
Example

Sorted Unsorted

	0	1	2	3	4	5	6	7	8	9	10	11	12
Original List	12	9	7	21	19	26	33	40	14	5	18	22	2
Pass #11	5	7	9	12	14	18	19	21	22	26	33	40	2
Pass #12	2	5	7	9	12	14	18	19	21	22	26	33	40

Time Complexity Analysis

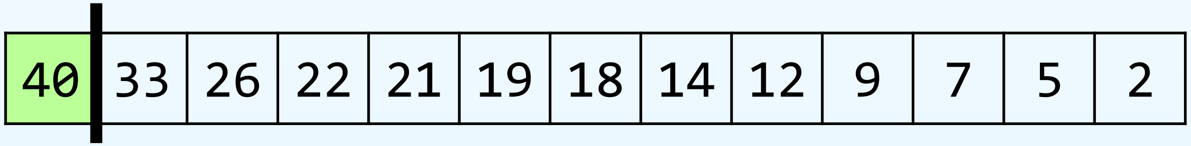
- Best case



2	5	7	9	12	14	18	19	21	22	26	33	40

$$T(n) = 1 + 1 + 1 + \cdots + 1 + 1 \quad = n - 1 \quad = \Omega(n)$$

- Worst case

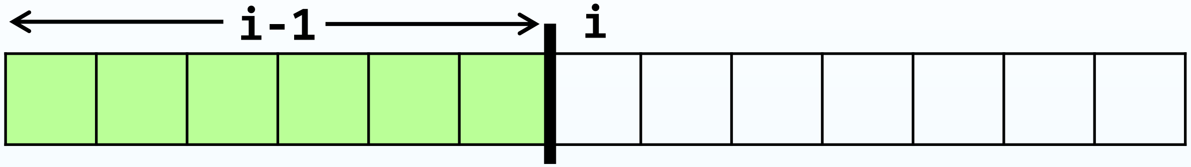


40	33	26	22	21	19	18	14	12	9	7	5	2

$$T(n) = 1 + 2 + 3 + \cdots + (n-2) + (n-1) \quad = \frac{(n-1)n}{2} \quad = O(n^2)$$

Time Complexity Analysis

- Average case



Probability of placing the i^{th} element at each position 0 to $i-1$ is $\frac{1}{i}$

Then, the running time of placing the i^{th} element is

$$1 \cdot \frac{1}{i} + 2 \cdot \frac{1}{i} + 3 \cdot \frac{1}{i} + \cdots + i \cdot \frac{1}{i} = \frac{1}{i} \sum_{j=1}^{i} j$$

Therefore, the average running time for n elements of the list is

$$T(n) = \sum_{i=1}^{n} \frac{1}{i} \sum_{j=1}^{i} j = \sum_{i=1}^{n} \frac{1}{i} \cdot \frac{i(i+1)}{2} = \frac{1}{2}\left(\frac{n(n+1)}{2} + n\right) = \Theta(n^2)$$

Extra Space Requirement

- Insertion sort is an in-place algorithm

- It performs all computation in the original array and no other array is used

- Hence, the extra space works out to be $O(1)$

Important Notes

- Insertion sort is not a very efficient algorithm when data set are large
  - This indicated by the average and worst case complexities

- However, insertion sort is adaptive, and number of comparisons are less if array is partially sorted

Outline

- Searching Algorithms
  - Linear Search
  - Binary Search

- Sorting Algorithms
  - Selection Sort
  - Insertion Sort
  - Bubble Sort
  - Merge Sort
  - Quick Sort
  - …

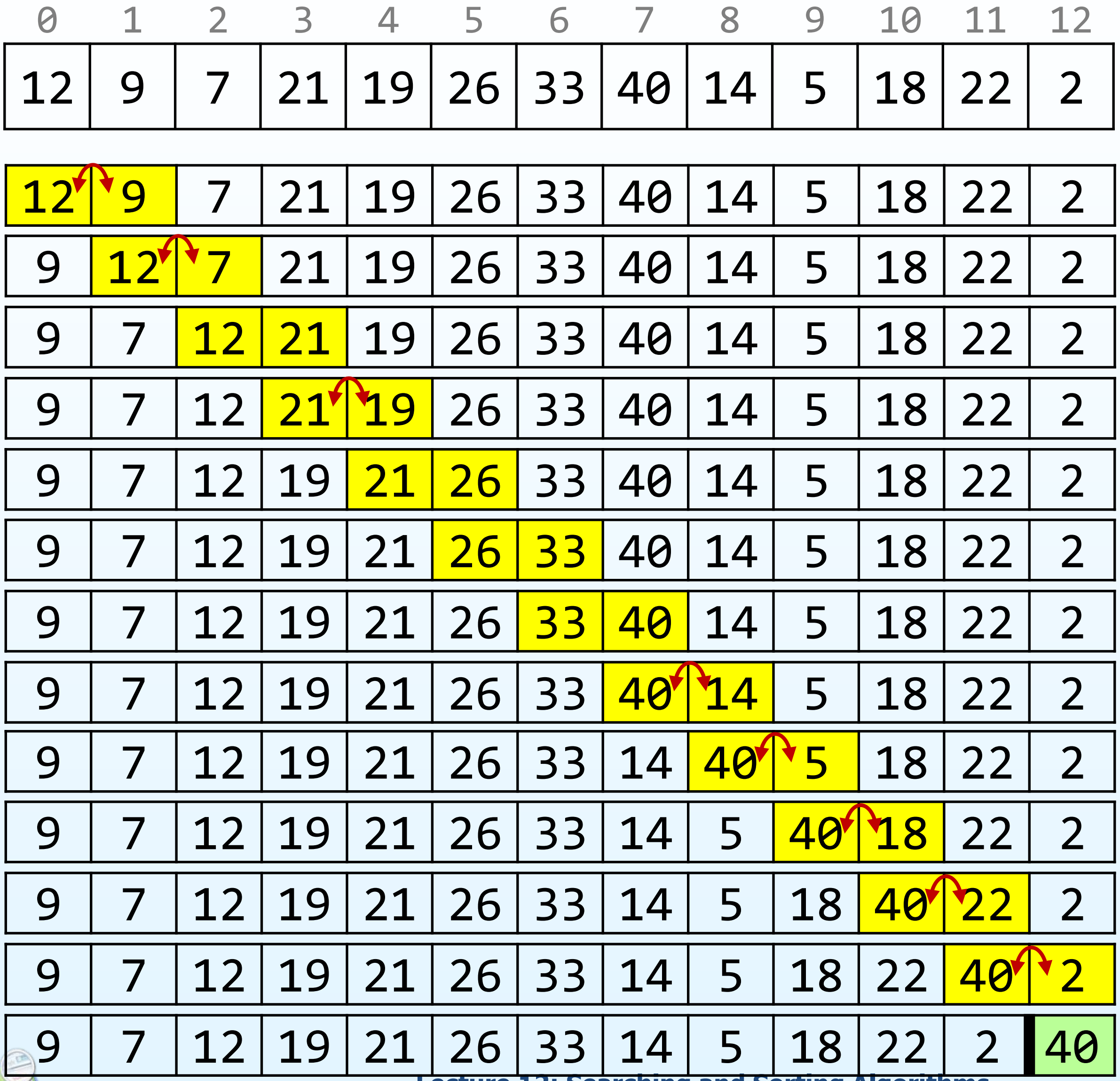
Bubble Sort

Bubble sort is the easiest sorting algorithm; it inspired by observing the behavior of air bubbles over foam

Idea:

- Use $n - 1$ passes through a list
- In each pass,
  - Compare the adjacent elements of the list
  - Swap the two elements if they are in the wrong order
  - Place the next largest element to its proper position

Example



	0	1	2	3	4	5	6	7	8	9	10	11	12
Original List	12	9	7	21	19	26	33	40	14	5	18	22	2
Pass #1	12	9	7	21	19	26	33	40	14	5	18	22	2
	9	12	7	21	19	26	33	40	14	5	18	22	2
	9	7	12	21	19	26	33	40	14	5	18	22	2
	9	7	12	21	19	26	33	40	14	5	18	22	2
	9	7	12	19	21	26	33	40	14	5	18	22	2
	9	7	12	19	21	26	33	40	14	5	18	22	2
	9	7	12	19	21	26	33	40	14	5	18	22	2
	9	7	12	19	21	26	33	40	14	5	18	22	2
	9	7	12	19	21	26	33	14	40	5	18	22	2
	9	7	12	19	21	26	33	14	5	40	18	22	2
	9	7	12	19	21	26	33	14	5	18	40	22	2
	9	7	12	19	21	26	33	14	5	18	22	40	2
	9	7	12	19	21	26	33	14	5	18	22	2	40

Example

	0	1	2	3	4	5	6	7	8	9	10	11	12
Original List	12	9	7	21	19	26	33	40	14	5	18	22	2
Pass #2	9	7	12	19	21	26	33	14	5	18	22	2	40
	7	9	12	19	21	26	33	14	5	18	22	2	40
	7	9	12	19	21	26	33	14	5	18	22	2	40
	7	9	12	19	21	26	33	14	5	18	22	2	40
	7	9	12	19	21	26	33	14	5	18	22	2	40
	7	9	12	19	21	26	33	14	5	18	22	2	40
	7	9	12	19	21	26	33	14	5	18	22	2	40
	7	9	12	19	21	26	14	33	5	18	22	2	40
	7	9	12	19	21	26	14	5	33	18	22	2	40
	7	9	12	19	21	26	14	5	18	33	22	2	40
	7	9	12	19	21	26	14	5	18	22	33	2	40
	7	9	12	19	21	26	14	5	18	22	2	33	40

Example

	0	1	2	3	4	5	6	7	8	9	10	11	12
Original List	12	9	7	21	19	26	33	40	14	5	18	22	2
Pass #3	7	9	12	19	21	26	14	5	18	22	2	33	40
	7	9	12	19	21	26	14	5	18	22	2	33	40
	7	9	12	19	21	26	14	5	18	22	2	33	40
	7	9	12	19	21	26	14	5	18	22	2	33	40
	7	9	12	19	21	26	14	5	18	22	2	33	40
	7	9	12	19	21	26	14	5	18	22	2	33	40
	7	9	12	19	21	14	26	5	18	22	2	33	40
	7	9	12	19	21	14	5	26	18	22	2	33	40
	7	9	12	19	21	14	5	18	26	22	2	33	40
	7	9	12	19	21	14	5	18	22	26	2	33	40
	7	9	12	19	21	14	5	18	22	2	26	33	40

Example

	0	1	2	3	4	5	6	7	8	9	10	11	12
Original List	12	9	7	21	19	26	33	40	14	5	18	22	2
Pass #4	7	9	12	19	14	5	18	21	2	22	26	33	40
Pass #5	7	9	12	14	5	18	19	2	21	22	26	33	40
Pass #6	7	9	12	5	14	18	2	19	21	22	26	33	40
Pass #7	7	9	5	12	14	2	18	19	21	22	26	33	40
Pass #8	7	5	9	12	2	14	18	19	21	22	26	33	40
Pass #9	5	7	9	2	12	14	18	19	21	22	26	33	40
Pass #10	5	7	2	9	12	14	18	19	21	22	26	33	40
Pass #11	5	2	7	9	12	14	18	19	21	22	26	33	40
Pass #12	2	5	7	9	12	14	18	19	21	22	26	33	40

Time Complexity Analysis

- Best case

2	5	7	9	12	14	18	19	21	22	26	33	40

$$T(n) = (n-1) + (n-2) + \cdots + 3 + 2 + 1 = \frac{(n-1)n}{2} = \Omega(n^2)$$

  - Better implementation

```
for pass ← 1 to n-1 do
    sorted = true
    for i ← 1 to n-pass do
        if (arr[i-1] > arr[i]) then
            swap(arr[i-1], arr[i])
            sorted = false
    if sorted then
        break
```

$$T(n) = (n-1) = \Omega(n)$$

Time Complexity Analysis

- Worst case

40	33	26	22	21	19	18	14	12	9	7	5	2

$$T(n) = (n-1) + (n-2) + \cdots + 3 + 2 + 1 = \frac{(n-1)n}{2} = O(n^2)$$

- Average case

$$T(n) = \Theta(n^2)$$

Extra Space Requirement

- Bubble sort is an in-place algorithm

- It performs all computation in the original array and no other array is used

- Hence, the extra space works out to be $O(1)$

Important Notes

- Bubble sort is beneficial when
  - Array elements are less, and
  - The array is nearly sorted

Outline

- Searching Algorithms
  - Linear Search
  - Binary Search
- Sorting Algorithms
  - Selection Sort
  - Insertion Sort
  - Bubble Sort
  - Merge Sort
  - Quick Sort
  - …

Merge Sort

Merge sort uses a divide and conquer paradigm for sorting

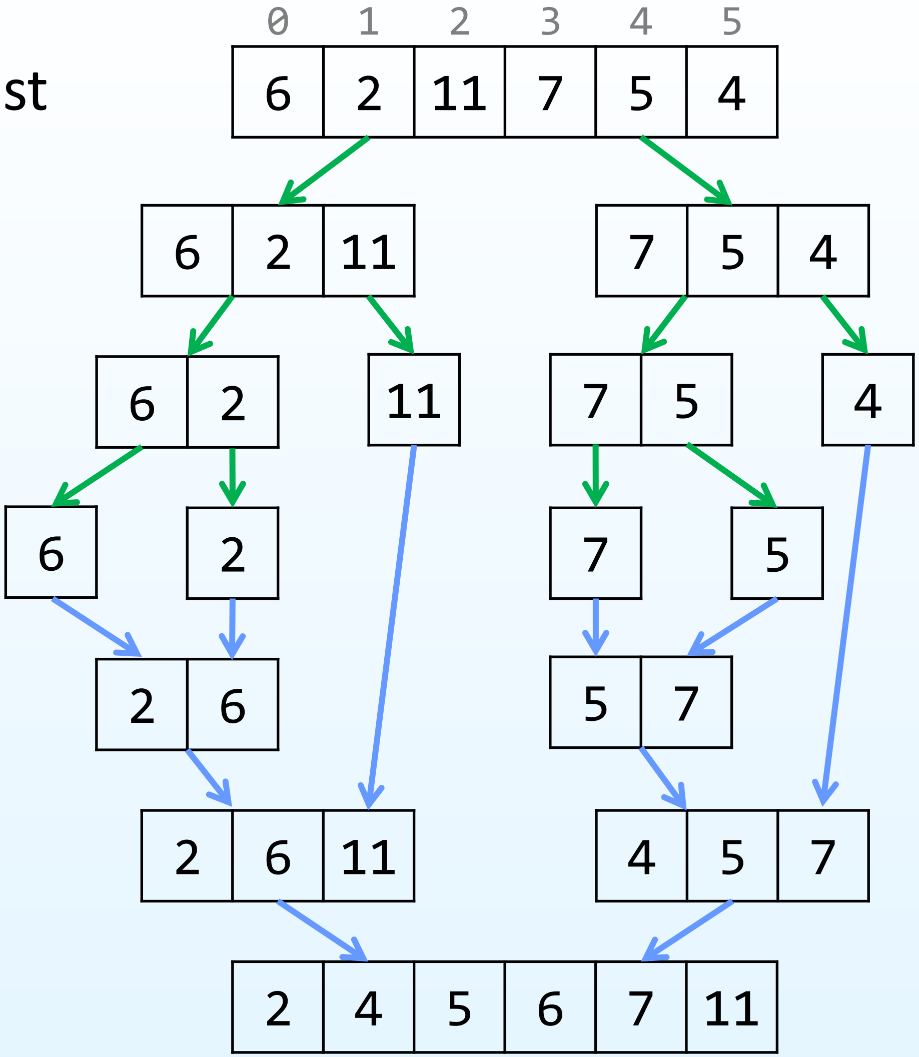
Idea:

The algorithm is defined recursively:

- If the list is of size 1, it is sorted—we are done
- Otherwise,
  1. Divide an unsorted list into two sub-lists, and sort each sub-list recursively using merge sort
  2. Merge the two sorted sub-lists into a single sorted list

Example

Original List



(Divide)

(Divide)

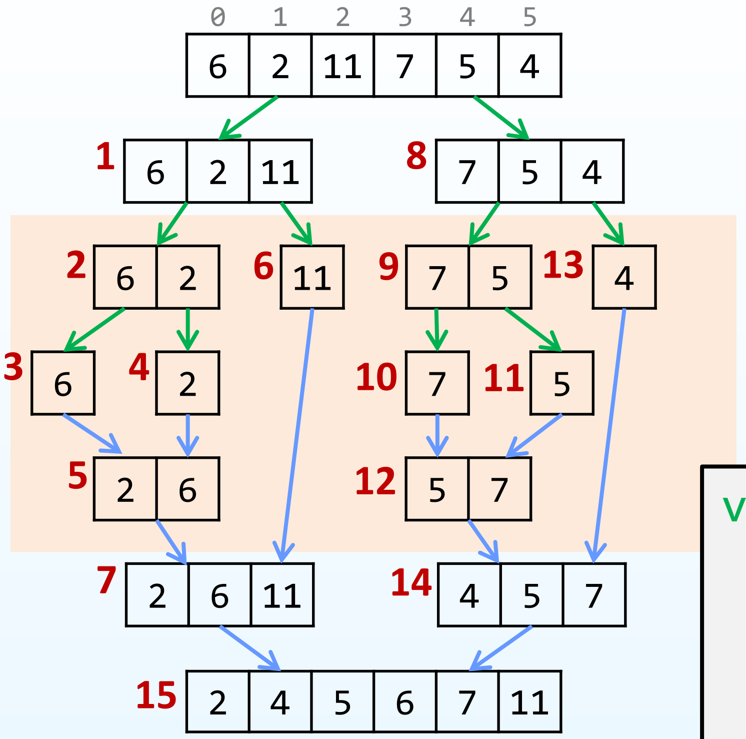
(Sorted trivially)

(Merge)

(Merge)

Sorted List

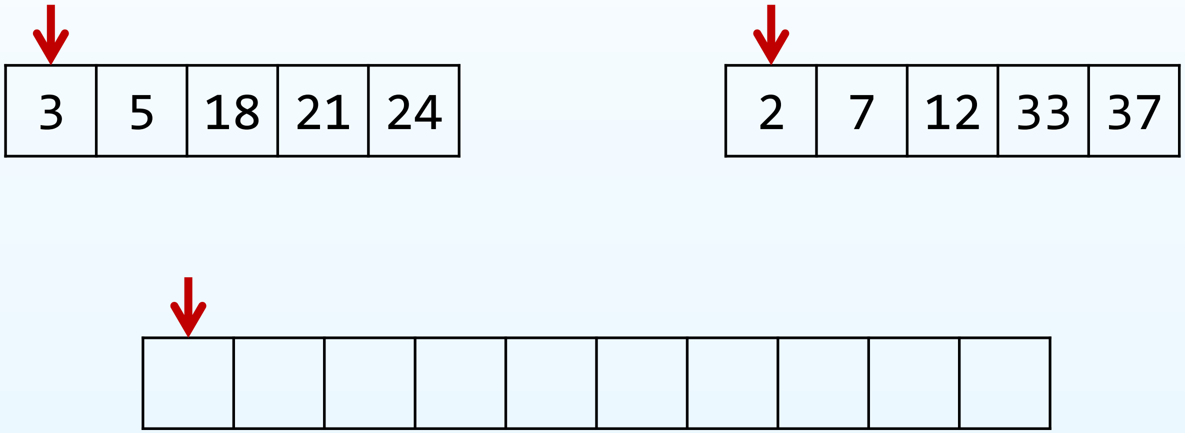
Implementation



```
void merge_sort(int arr[], int l, int r) {
  int m;
  if (l < r) {
    m = l + (r-l)/2;
    merge_sort(arr, l, m);
    merge_sort(arr, m+1, r);
    merge(arr, l, m, r);
  }
}
```

Merging Two Lists

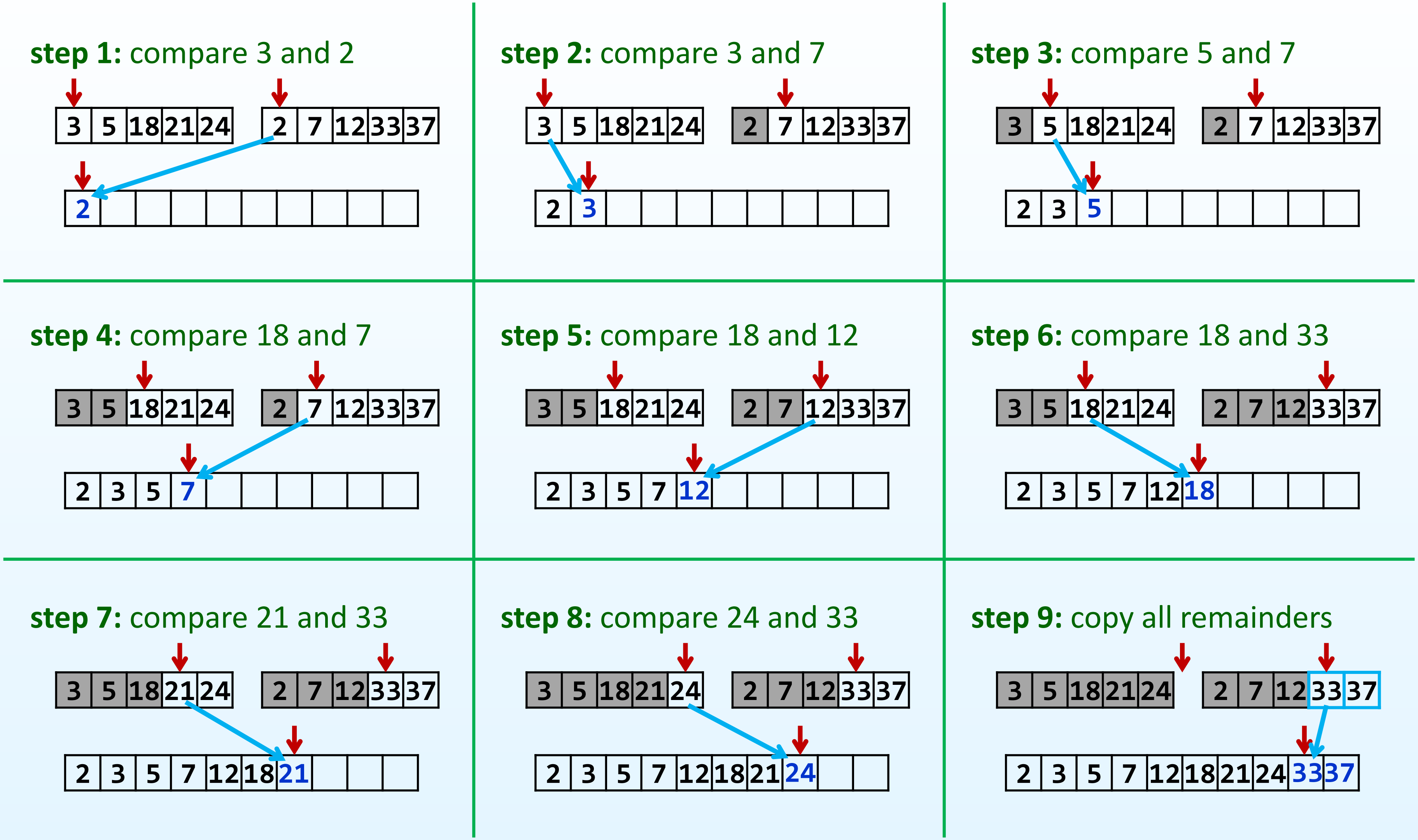
- Consider the two sorted arrays and an empty array
- Define three indices at the start of each array
- **Method:** compare and copy the least value



3	5	18	21	24

2	7	12	33	37

Merging Two Lists



step 1: compare 3 and 2

3 5 18 21 24 | 2 7 12 33 37

2

step 2: compare 3 and 7

3 5 18 21 24 | 2 7 12 33 37

2 3

step 3: compare 5 and 7

3 5 18 21 24 | 2 7 12 33 37

2 3 5

step 4: compare 18 and 7

3 5 18 21 24 | 2 7 12 33 37

2 3 5 7

step 5: compare 18 and 12

3 5 18 21 24 | 2 7 12 33 37

2 3 5 7 12

step 6: compare 18 and 33

3 5 18 21 24 | 2 7 12 33 37

2 3 5 7 12 18

step 7: compare 21 and 33

3 5 18 21 24 | 2 7 12 33 37

2 3 5 7 12 18 21

step 8: compare 24 and 33

3 5 18 21 24 | 2 7 12 33 37

2 3 5 7 12 18 21 24

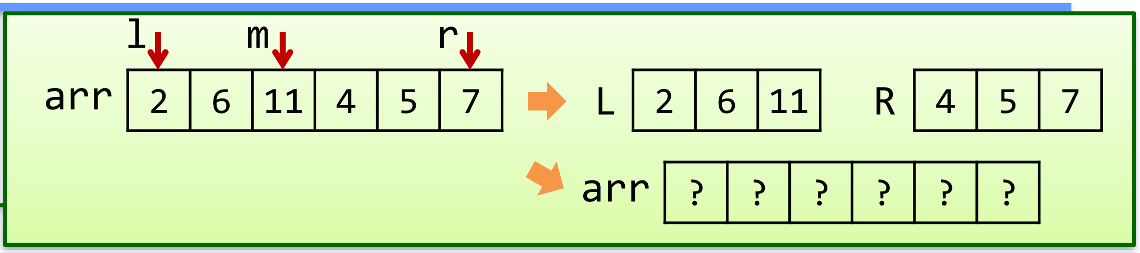
step 9: copy all remainders

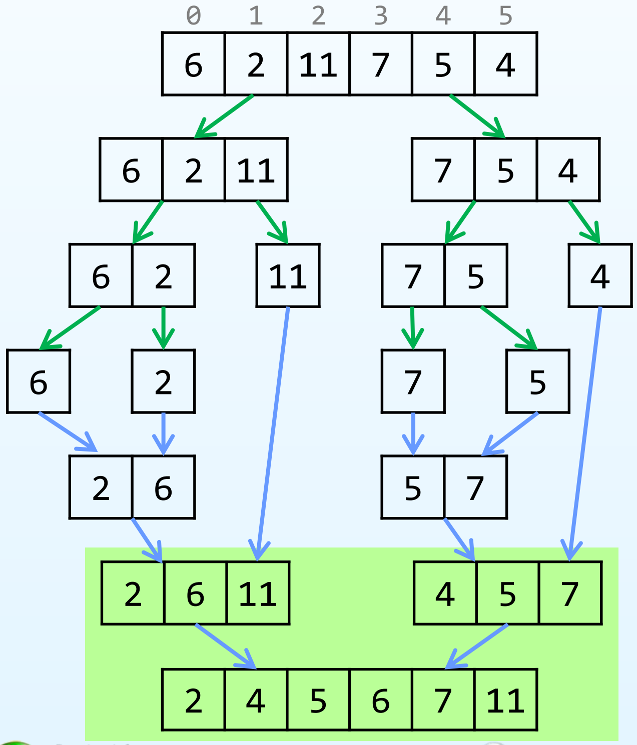
3 5 18 21 24 | 2 7 12 33 37

2 3 5 7 12 18 21 24 33 37

The merge() Function

```
void merge_sort(int arr[], int l, int r) {
  int m;
  if (l < r) {
    m = l + (r-l)/2;
    merge_sort(arr, l, m);
    merge_sort(arr, m+1, r);
    merge(arr, l, m, r);
  }
}
```





```
void merge(int arr[], int l, int m, int r) {
  int i, j, k;
  int nl = m-l+1, L[nl];
  int nr = r-m,   R[nr];
  // Copy data to temporary L and R arrays
  for (i=0; i<nl; i++)
    L[i] = arr[l+i];
  for (j=0; j<nr; j++)
    R[j] = arr[m+1+j];
  // Merge the L and R arrays back into arr
  i = 0; j = 0; k = l;
  while (i < nl && j < nr)
    arr[k++] = (L[i]<=R[j])? L[i++] : R[j++];
  // Copy the remaining elements, if any
  while (i < nl)
    arr[k++] = L[i++];
  while (j < nr)
    arr[k++] = R[j++];
}
```

Analysis of Merging

```
void merge(int arr[], int l, int m, int r) {
  int i, j, k;
  int nl = m-l+1, L[nl];
  int nr = r-m,   R[nr];
  // Copy data to temporary L and R arrays
  for (i=0; i<nl; i++)
    L[i] = arr[l+i];
  for (j=0; j<nr; j++)
    R[j] = arr[m+1+j];
  // Merge the L and R arrays back into arr
  i = 0; j = 0; k = l;
  while (i < nl && j < nr)
    arr[k++] = (L[i]<=R[j])? L[i++] : R[j++];
  // Copy the remaining elements, if any
  while (i < nl)
    arr[k++] = L[i++];
  while (j < nr)
    arr[k++] = R[j++];
}
```

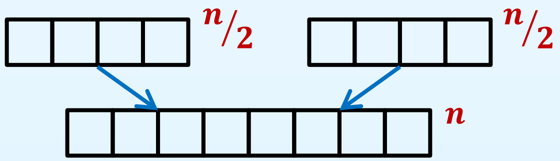
$\Theta(m - l + 1)$

$\Theta(r - m)$

$\Theta(r - l + 1)$

Merging n elements takes $\Theta(n)$

Memory requirements are also $\Theta(n)$



Time Complexity Analysis

```
void merge_sort(int arr[], int l, int r) {
  int m;
  if (l < r) {
    m = l + (r-l)/2;
    merge_sort(arr, l, m);
    merge_sort(arr, m+1, r);
    merge(arr, l, m, r);
  }
}
```

$T(n/2)$

$T(n/2)$

$\Theta(n)$

Recurrence relation:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

$$= \Theta(n \log n)$$ for all best, worst, and average cases

Extra Space Requirement

- Merge sort use additional memory for left and right sub-arrays

- Hence, the extra space works out to be $\Theta(n)$

Important Notes

- Merge sort uses a divide and conquer paradigm
- Merge sort is a recursive sorting algorithm
- Merge sort is a stable sorting algorithm
- Merge sort is not an in-place sorting algorithm

Outline

- Searching Algorithms
  - Linear Search
  - Binary Search

- Sorting Algorithms
  - Selection Sort
  - Insertion Sort
  - Bubble Sort
  - Merge Sort
  - Quick Sort
  - …

Quick Sort

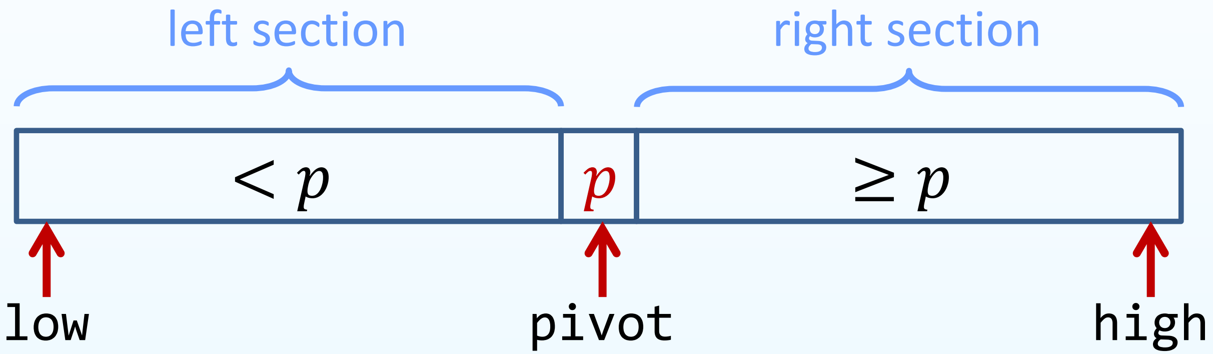
Quick sort uses a divide and conquer paradigm for sorting

Idea:

1. First, select a pivot element
2. Partition the list into two parts (elements smaller than and greater than or equal to the pivot)
3. Then, sort each part independently (recursively)
4. Finally, combine the sorted subsequences by a simple concatenation

Partition

- Partitioning places the pivot in its correct position within the sorted list



- Arranging the elements around the pivot p generates two smaller sorting problems:
  - Sort the left section and the right section
  - When these two smaller sorting problems are solved recursively, our bigger sorting problem is solved

Partition

For example, given

80	38	95	84	66	10	79	44	26	87	96	12	43	81	3

we can select the middle entry (44) as pivot, and sort the remaining entries into two sections:

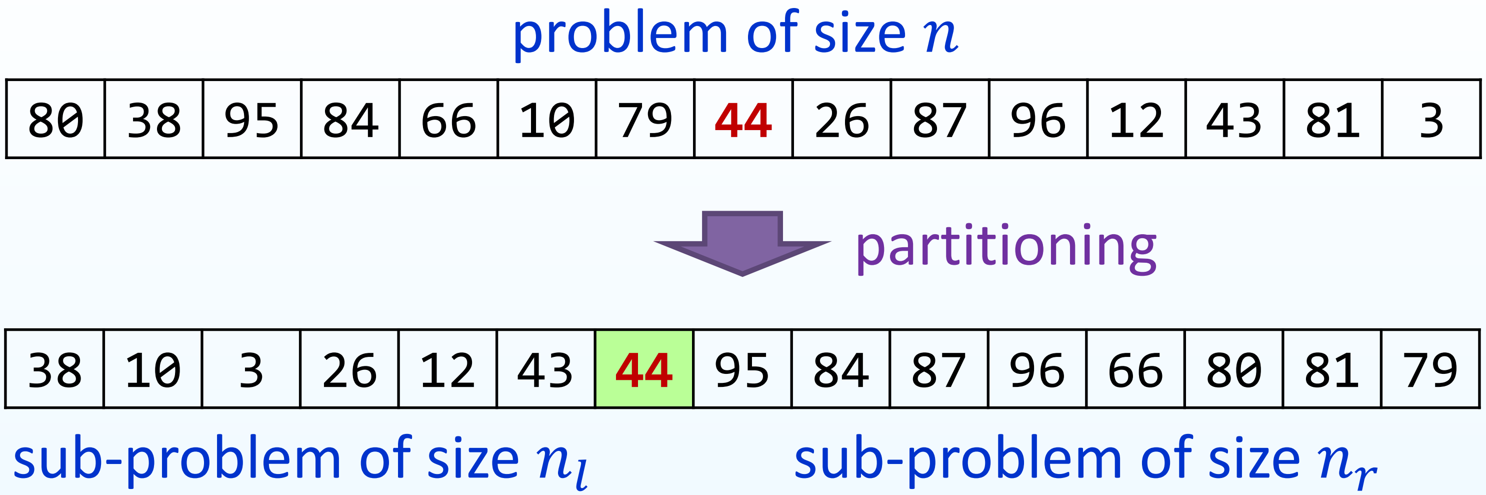
← less than 44 → ← greater than 44 →

38	10	3	26	12	43	44	95	84	87	96	66	80	81	79

Notice that 44 is now in the correct location if the list was sorted

- Proceed by applying the sorting algorithm recursively to the left and right sections independently

The quick_sort() Function



problem of size n

80	38	95	84	66	10	79	**44**	26	87	96	12	43	81	3

partitioning

38	10	3	26	12	43	**44**	95	84	87	96	66	80	81	79

sub-problem of size n_l

sub-problem of size n_r

```
void quick_sort(int arr[], int low, int high) {
  int pivot;
  if (low < high) {
    pivot = partition(arr, low, high);
    quick_sort(arr, low, pivot-1);
    quick_sort(arr, pivot+1, high);
  }
}
```

Pivot Selection

For example, given

80	38	95	84	66	10	79	44	26	87	96	12	43	81	3

- If we select 44 (the middle element) as pivot, we get:

38	10	3	26	12	43	**44**	94	84	84	96	66	80	81	79

- If we select 10 (randomly) as pivot, we get:

3	**10**	95	84	66	80	79	44	26	87	96	12	43	81	38

- If we select 66 (randomly) as pivot, we get:

38	3	10	44	26	12	43	**66**	80	87	96	95	79	81	84

Pivot Selection

Somehow, we have to select a pivot, and we hope that we will get a good partitioning:

- We can choose a pivot randomly, or
- We can choose the first element as the pivot, or
- We can choose the middle element as the pivot, or
- We can choose the last element as the pivot, or
- We can use a combination of the above criterions, or
- …

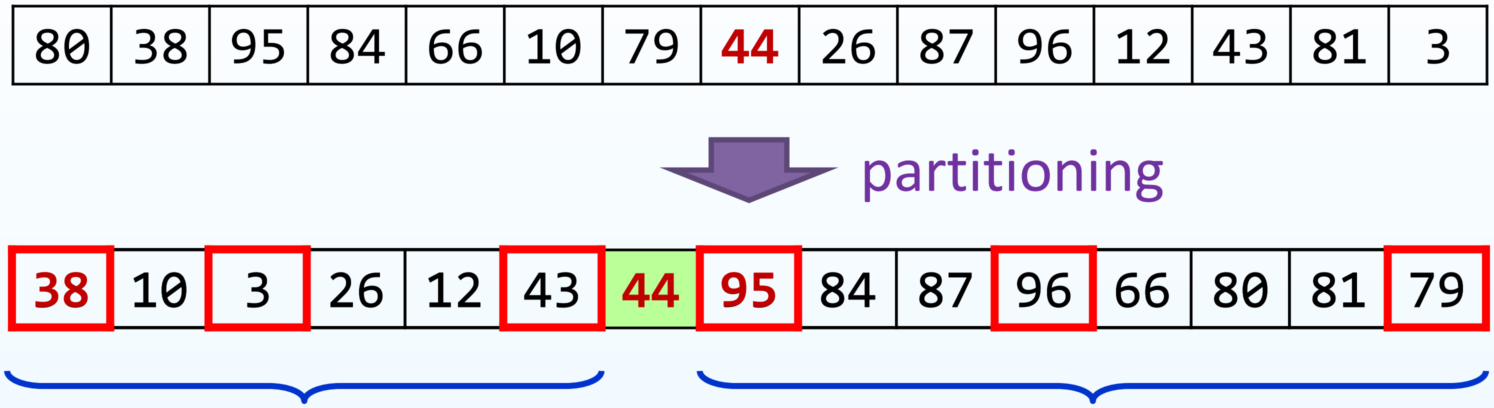
Pivot Selection: Median-of-Three

- If we know the median of the elements, we will get the perfect partition
- However, it is difficult to find the median
- So consider another strategy:
  - Choose the median of the first, middle, and last elements
- This will usually give a better approximation of the actual median

80	38	95	84	66	10	79	**44**	26	87	96	12	43	81	3

- 44 is selected since it is the median of 80, 44, and 3

Pivot Selection: Median-of-Three



80	38	95	84	66	10	79	44	26	87	96	12	43	81	3

partitioning

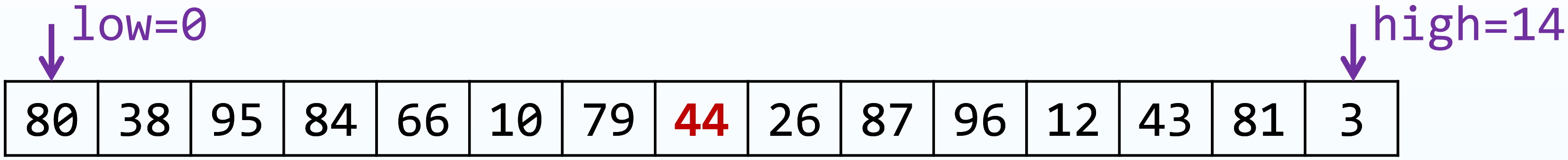
38	10	3	26	12	43	44	95	84	87	96	66	80	81	79

Select 38 for partitioning
the left sub-list

Select 95 for partitioning
the right sub-list

Partitioning Example

We call `quick_sort(arr, 0, 14)`



low=0

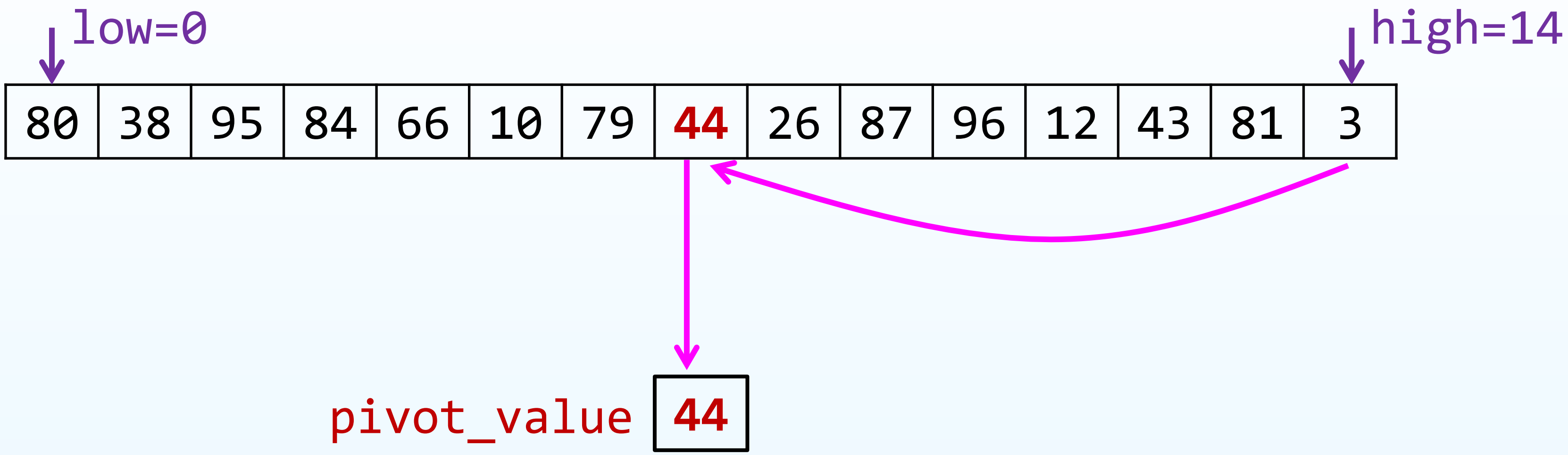
high=14

80	38	95	84	66	10	79	**44**	26	87	96	12	43	81	3

- First, find a pivot
  - Using the median-of-three method, we get 44

Partitioning Example

We call `quick_sort(arr, 0, 14)`



low=0

high=14

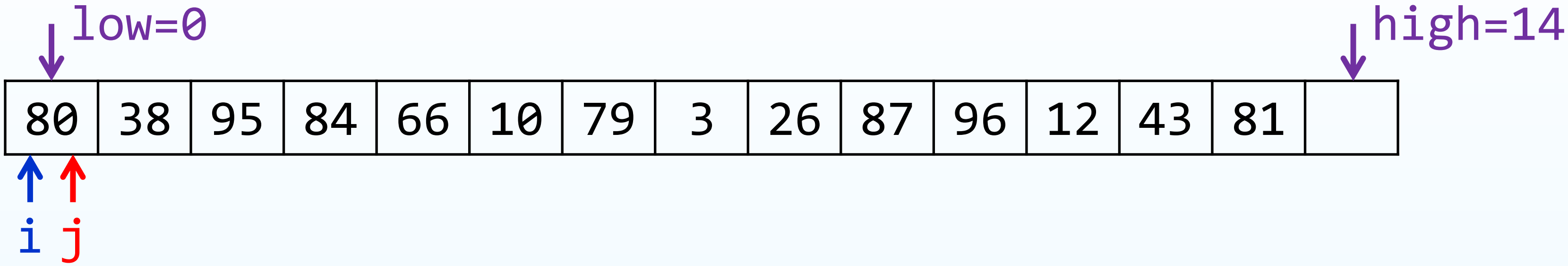
80	38	95	84	66	10	79	44	26	87	96	12	43	81	3

pivot_value 44

- Copy the pivot value to a temporary memory
- Replace the pivot with the last element

Partitioning Example

We call `quick_sort(arr, 0, 14)`



low=0

high=14

80	38	95	84	66	10	79	3	26	87	96	12	43	81	

i j

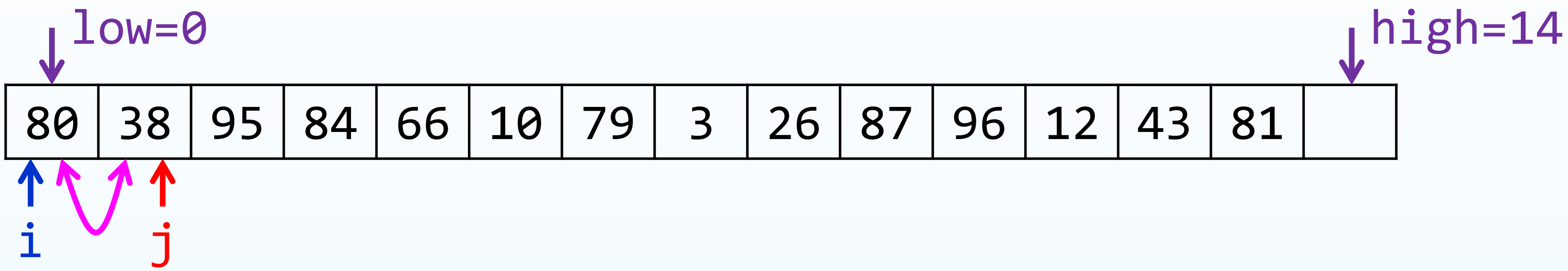
pivot_value **44**

We define the blue `i` and red `j` indices to indicate elements that are greater than and less than the pivot, respectively

- Start `i` and `j` at `low`

Partitioning Example

We call `quick_sort(arr, 0, 14)`



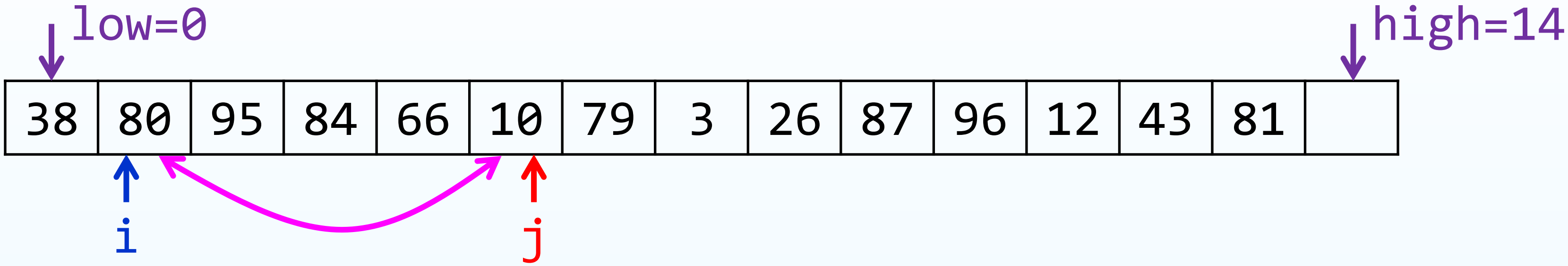
80	38	95	84	66	10	79	3	26	87	96	12	43	81	

`pivot_value` **44**

- Move `j` up to until finding the next element that is less than the pivot
- Swap the elements pointed by `i` and `j`
  - Then, move `i` up by one

Partitioning Example

We call `quick_sort(arr, 0, 14)`



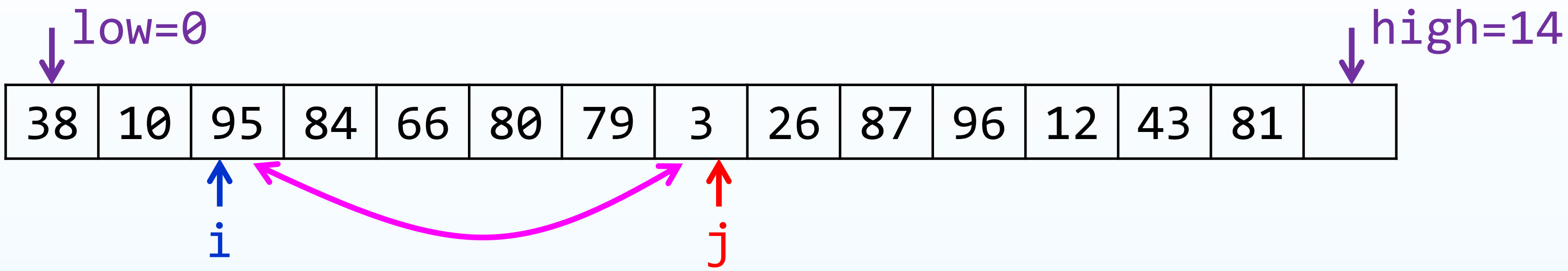
38	80	95	84	66	10	79	3	26	87	96	12	43	81	

`pivot_value` **44**

- Move `j` up to until finding the next element that is less than the pivot
- Swap the elements pointed by `i` and `j`
  - Then, move `i` up by one

Partitioning Example

We call `quick_sort(arr, 0, 14)`



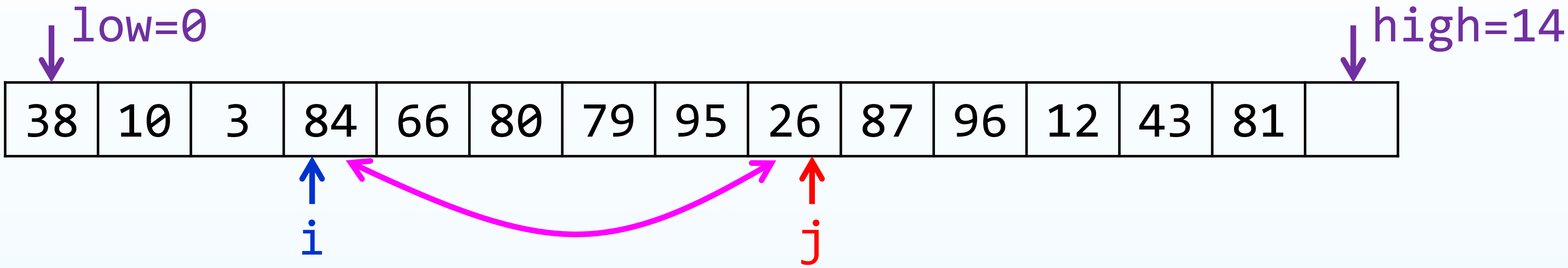
38	10	95	84	66	80	79	3	26	87	96	12	43	81	

pivot_value **44**

- Move `j` up to until finding the next element that is less than the pivot
- Swap the elements pointed by `i` and `j`
  - Then, move `i` up by one

Partitioning Example

We call `quick_sort(arr, 0, 14)`



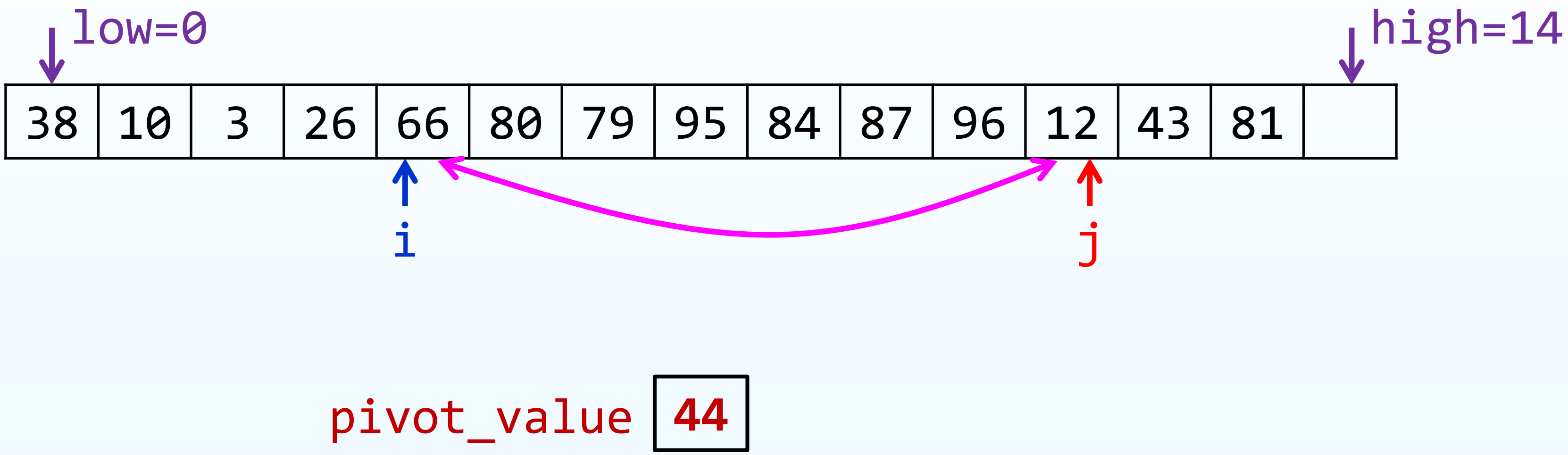
38	10	3	84	66	80	79	95	26	87	96	12	43	81	

`pivot_value` **44**

- Move `j` up to until finding the next element that is less than the pivot
- Swap the elements pointed by `i` and `j`
  - Then, move `i` up by one

Partitioning Example

We call `quick_sort(arr, 0, 14)`



low=0

high=14

38	10	3	26	66	80	79	95	84	87	96	12	43	81	

i

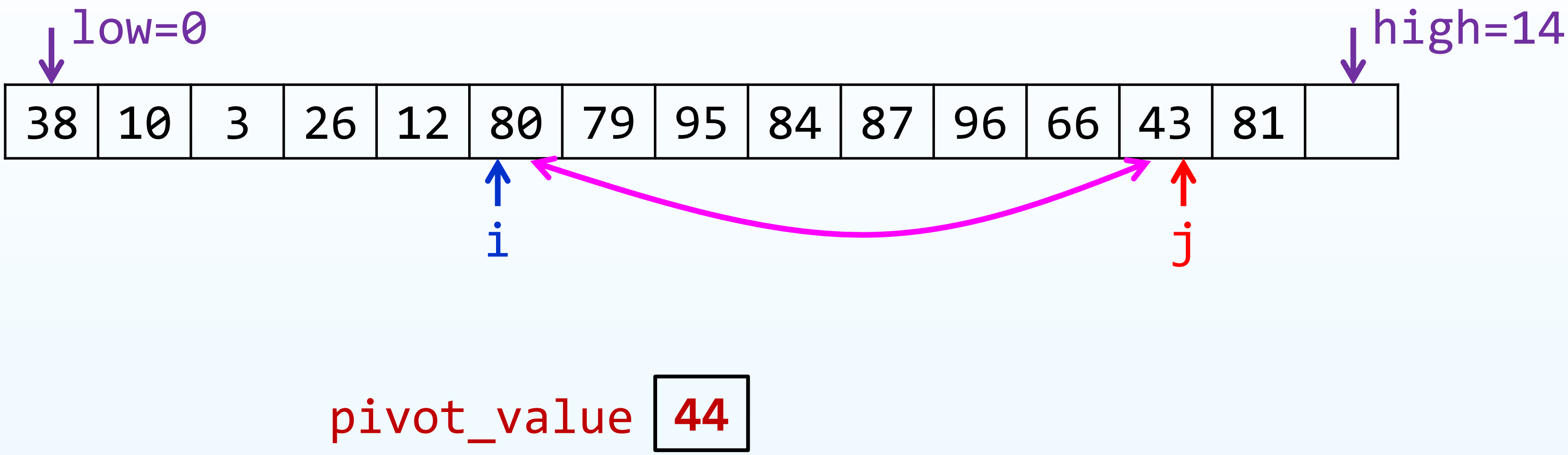
j

pivot_value 44

- Move j up to until finding the next element that is less than the pivot
- Swap the elements pointed by i and j
  - Then, move i up by one

Partitioning Example

We call `quick_sort(arr, 0, 14)`

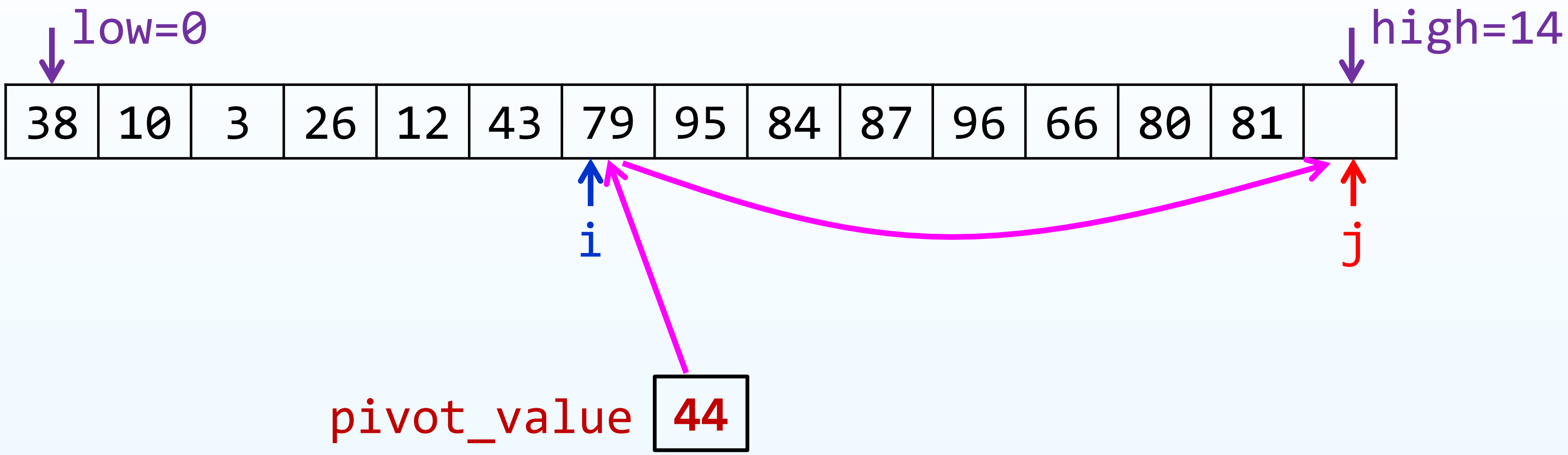


38	10	3	26	12	80	79	95	84	87	96	66	43	81	

- Move `j` up to until finding the next element that is less than the pivot
- Swap the elements pointed by `i` and `j`
  - Then, move `i` up by one

Partitioning Example

We call `quick_sort(arr, 0, 14)`

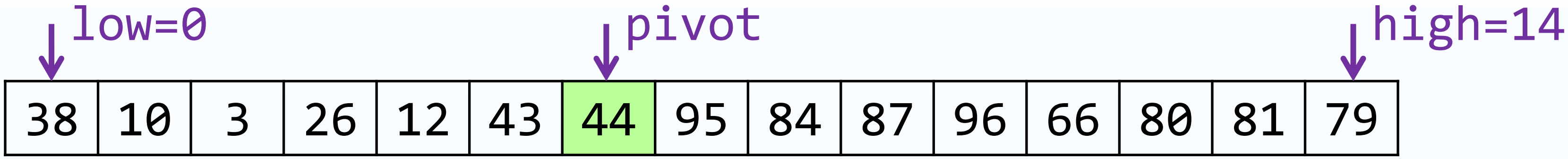


38	10	3	26	12	43	79	95	84	87	96	66	80	81	

- Move `j` up to until finding the next element that is less than the pivot
  - However, the iteration will be terminated when `j` reaches `high`
- Afterwards, move the element pointed by `i` to the end
- Finally, copy the pivot value to locate at `i`

Partitioning Example

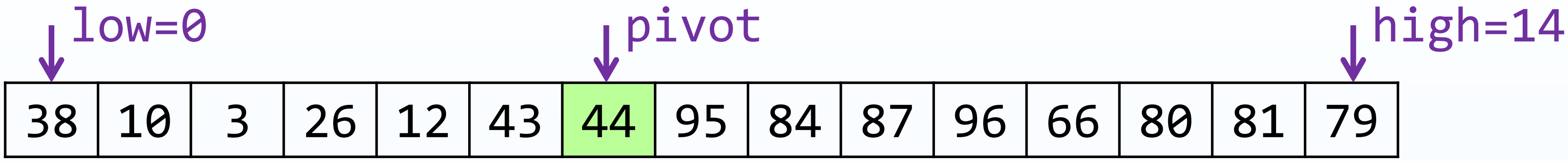
We call `quick_sort(arr, 0, 14)`



38	10	3	26	12	43	44	95	84	87	96	66	80	81	79

- We get the correct location of the pivot in the sorted list
  - Partitioning returns `pivot`
  - The list is divided into the left and right sub-lists
- We then call `quick_sort(arr, low, pivot-1)` and `quick_sort(arr, pivot+1, high)` to sort each partition separately

The partition() Function



low=0

pivot

high=14

38	10	3	26	12	43	44	95	84	87	96	66	80	81	79

```
int partition(int arr[], int low, int high) {
  // Select a pivot (may use the median-of-three method)
  int pivot = find_pivot(arr, low, high);
  int i = low, j = low;
  // Temporarily store the pivot value at the end of the array
  swap(&arr[pivot], &arr[high]);
  for (j=low; j<high; j++) {
    // If found smaller element, swap it with the current greater element
    if (arr[j] < arr[high])
      swap(&arr[i++], &arr[j]); // Increment i by one after swapping
  }
  // Move the pivot value back to the correct position, return that index
  swap(&arr[i], &arr[high]);
  return i; // Return the pivot index
}
```

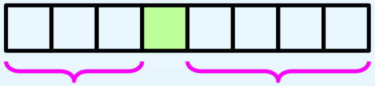
Analysis of Partitioning

```
int partition(int arr[], int low, int high) {
  // Select a pivot (may use the median-of-three method)
  int pivot = find_pivot(arr, low, high);
  int i = low, j = low;
  // Temporarily store the pivot value at the end of the array
  swap(&arr[pivot], &arr[high]);
  for (j=low; j<high; j++) {
    // If found smaller element, swap it with the current greater element
    if (arr[j] < &arr[high])
      swap(&arr[i++], &arr[j]); // Increment i by one after swapping
  }
  // Move the pivot value back to the correct position, return that index
  swap(&arr[i], &arr[high]);
  return i; // Return the pivot index
}
```

$\Theta(1)$

$\Theta(high - low + 1)$

Partitioning n elements takes $\Theta(n)$



the sum is $\Theta(n)$

Time Complexity Analysis

```
void quick_sort(int arr[], int first, int last) {
  int pivot;
  if (first < last) {
    pivot = partition(arr, first, last)
    quick_sort(arr, first, pivot-1);
    quick_sort(arr, pivot+1, last);
  }
}
```

- Best case

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \quad = \Omega(n \log n)$$

- Worst case

$$T(n) = T(0) + T(n-1) + \Theta(n) \quad = O(n^2)$$

- Average case

$$T(n) = \Theta(n \log n)$$

Any Question?