Searching and Sorting

Dr Govindha R Yeluripati

Outcomes

- At the end of this lecture students will be able to

1) Understand and analyse commonly used search algorithms
2) Implement the binary search algorithm
3) Understand and analyze important sorting algorithms
4) Implement Merge sort and Quick sort algorithms
5) Apply searching and sorting techniques as appropriate to practical applications

References

- Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser, Data Structures and Algorithms in Java, 6th Edition, John Wiley and Sons (Section 5.1.3 for Binary Search)
- Weiss, M.A (2012). Data Structures and Algorithm Analysis in Java, Third Edition, Pearson (Section 2.4.4 for Binary Search)
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, Introduction to Algorithms (Exponential search, Quick Sort, Merge sort)
- Algorithms and Data Structures by Robert Sedgewick and Kevin Wayne (2019) (Interpolation search)
- The Algorithm Design Manual by Steven S. Skiena (2013) (interpolation search, exponential search)

Searching and Sorting: What are they?

- Searching is finding the location of a specific item among a collection of items or data structure. These items could be any data, such as numbers, letters, words, or more complex data items.

- Sorting involves arranging data items in a certain order (ascending or descending). This is usually required to improve the efficiency of operations like searching or merging datasets.

Searching and Sorting: The need

- Storing and retrieving information is very common in computing
- Searching and sorting are two of the most fundamental operations in computer science and have a wide variety of applications.
  - **Databases**: to efficiently find and retrieve data
  - **Search Engines**: use sorting algorithms to rank the results of a search query (e.g., most relevant first)
  - **Compilers**: use sorting algorithms to optimize the code
  - **Operating Systems**: to manage files, processes, and memory
  - **Scientific Computing**: data analysis, machine learning, and artificial intelligence

Searching and Sorting: The need

- Benefits and some more real-world applications:
  - Efficiency in Data Retrieval (Searching)
    - e.g., a transaction in banking
  - Organizing Data (Sorting):
    - e.g., library books, dictionary
  - Optimization of Further Operations:
    - e.g., data compression, graphics rendering, and network routing
  - Improving User Experience:
    - e.g., list of emails sorted by date, a leaderboard in a game sorted by high scores, products in an online store sorted by price or ratings
  - Data Integrity and Consistency:
    - In database management: elimination of duplicate records, merging data sets, identifying inconsistencies
  - Analytical Efficiency and Decision Making:
    - In data analytics: segmenting and filtering data, identifying trends, and making data-driven decisions.
- So, searching and sorting are critical for efficiently managing and processing data.

Searching Algorithms

- Problem: Finding a particular item in a collection of items
- Input: A search item/key/target (e.g., name, ID)
- Output/Result: An answer whether the search item is found or not, along with the position/location (e.g., URL, array index, page number) of the item found.

Searching Algorithms

Commonly used techniques:

- Linear Search
- Binary Search

Other techniques:

- Jump Search
- Interpolation Search
- Exponential Search
- Fibonacci Search
- Ternary Search
- Quad Search

Linear Search

- Basic and simple search algorithm
- Searches for an element or value in the given sequence in a sequential order. Hence, also called as sequential search.
- Search element is compared with all the elements given in the list and all the matching positions or indexes are returned
- Applied on the unsorted or unordered list when there are fewer elements in a list.

Linear Search: Algorithm

linear_search(*data_structure*, *target_element*):

1. **for** each *element* in *data_structure*:
2. **if** *element* == *target_element*:
3. **return** 'Found' and the *element*'s index
4. **return** 'Not Found'

- Example:

Index	0	1	2	3	4	5	6	7	8	9
A	**51**	**82**	**12**	**-5**	**-5**	**22**	**21**	**89**	**-5**	**18**

Search item = -5

Output : Found at 3

Search item = 99

Output : Not Found

Linear Search: Time complexity

linear_search(*data_structure*, *target_element*):

1. **for** each *element* in *data_structure*: → **1Unit for initialization, *n+1* times comparison, *n* times increment**
2. **if** *element* == *target_element*: → **1Unit , *n* times**
3. **return** 'Found' and the *element*'s index → **1Unit , 0 to *n* times based on the values in the sequence (we can ignore return statements)**
4. **return** 'Not Found' → **1Unit**

$$T(n) = (1 + n + 1 + n) + n + 1 + 1 = 3n + 4$$

Hence, $\boldsymbol{T(n) = O(n)}$

Linear(Sequential) Search

Use-Cases:

- Can be used on any list, whether it's sorted or not.
- Best for *small lists* or lists that are not sorted and cannot be sorted.

Time Complexity:

- Worst-case performance: O(n)
- Best-case performance: O(1)

Binary Search

- Prerequisite: Given elements should be sorted (ascending or descending as required)
- Check whether the item to be searched (key) is the middle element and if so output the middle index
- If the key is smaller than the middle element, repeat the procedure in the left sub-array
- If the key is greater than the middle element, repeat the procedure in the right sub-array

Binary Search: Example

- Sorted array: A = [1, 3, 4, 6, 8, 9, 11]

  Let the search value, *key* = 4

1. Compare key with 6 (mid)
2. key is smaller. Repeat with A = [1, 3, 4]
3. Compare key with 3 (mid)
4. key is larger. Repeat with A = [4].
5. Compare key with 4 (mid, only element)
6. Key = 4, so the index/position of 4 is returned.

Binary Search

$T(n) = 1+1+ \lfloor \log_2(n) \rfloor +1 + (\lfloor \log_2(n) \rfloor)x7+1 = 7\lfloor \log_2(n) \rfloor +4$

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

Pseudocode:

binarySearch (*array*, *target*):

1. *low* = 0 → 1 Unit
2. *high* = length(*array*) - 1 → 1 Unit
3. **while** *low* <= *high*: → 1 Unit, $\lfloor \log_2(n) \rfloor +1$ times
4. *mid* = (low + high) / 2 // integer division → 3 Units
5. **if** *array*[*mid*] == *target*: → 1 Unit
6. **return** *mid* → 1 Unit
7. **else if** *array*[*mid*] < *target*: → 1 Unit
8. *low* = *mid* + 1 → 2 Units
9. **else**:
10. *high* = *mid* - 1 → 2 Units
11. **return** -1 // if the target is not found → 1 Unit

Maximum of these will be considered

$\lfloor \log_2(n) \rfloor$ times

Binary Search

Use-Cases:

- The array must be sorted before using binary search.
- Useful for larger lists because it's significantly faster than linear search.

Time Complexity:

- Worst-case performance: O(log n)
- Best-case performance: O(1)

Jump Search

- Also called **block search**, and works on sorted arrays.
- Basic idea:
- To check fewer elements by jumping ahead by fixed steps, instead of traversing the whole list.
- Once we find an interval where the element may reside, perform a linear search.

- **Use-Cases**:
- The array must be sorted.
- Useful when a list is somewhat larger, as it can be more efficient than linear search but simpler than a full-fledged binary search.

Jump Search: Algorithm

Algorithm **JumpSearch**(array, target):

```
n = length(array)
block = sqrt(n)          // Block size to be jumped
// Finding the block where the element is present (if it is in the array)
prev = 0
while array[min(block, n)-1] < target:
    prev = block
    block += sqrt(n)
    if prev >= n:
        return -1            // Target is not present in the array
while array[prev] < target:    // Performing linear search in the block
        prev += 1
        if prev == min(block, n):   // If reached next block / end of array,
            return -1     //element is not present
if array[prev] == target:
     return prev      // If the element is found
return -1         // Unsuccessful search
```

Worst-case performance: O(sqrt(n))
Best-case performance: O(1)

Exponential Search

- Also known as **expanding intervals search**
- Used on a sorted array
- Basic Idea:
- First find a range where the target value resides and then performing a binary search within that range.

- Exponential search is efficient for unbounded searches, where the size or the end of the list is unknown or infinite.

Exponential Search: Major functionality

1) Start with a bound: Begin at the first element and check if it's the target. If not, proceed to the next step.

2) Expand Exponentially: Expand the search interval exponentially (1, 2, 4, 8, 16, ...) until an interval is found where the upper bound is greater than or equal to the target, or you have reached the end of the list.

3) Binary Search: Once an interval is identified where the target might be, perform a binary search within that range.

Exponential Search: Algorithm

- **exponentialSearch**(sortedArray, target):

```
1.    n = length(sortedArray)    //Optional step for unbounded structure
2.    if sortedArray[0] == target:        // target is present at first location
3.        return 0
4.    i = 1  // Initialize the range for binary search by repeated doubling
5.    while i < n and sortedArray[i] <= target:   //Check for out of bounds
                                                  error/exception for the stuctures
6.        i = i * 2                               where length is unknown
7.    return binarySearch(sortedArray, i/2, min(i, n-1), target)
      // Perform binary search for the target in the range found
```

Exponential Search: Time Complexity

- The exponential search portion has a time complexity of O(log i), where 'i' is the position of the target value in the array.
- The binary search portion has a time complexity of O(log n).
- Hence, the overall time complexity is O(log i) + O(log n), which is effective for sorted and unbounded sequences.

Interpolation Search

- An improved variant of binary search for instances where the values in the sorted array are uniformly distributed.
- Involves calculating an estimate of where the target value is likely to be, based on the low, high, and target values.

- **Use-Cases**:
- The array must be sorted and the values are distributed uniformly.
- More efficient than binary search when the elements are uniformly distributed.

Interpolation Search: Algorithm

- **interpolationSearch**(array, target):

```
1.  low = 0
2.  high = length(array) - 1
3.  while low <= high and target >= array[low] and target <= array[high]:
4.      if low == high:
5.          if array[low] == target:
6.              return low
7.          return -1
8.   // Calculate the probe position. Using the formula for interpolation
9.  pos = low + ((high - low) / (array[high] - array[low])) * (target -
    array[low])
10.  if array[pos] == target:          // Target found
11.          return pos
12.  if array[pos] < target:           // Target is in upper part
13.          low = pos + 1
14.  else:                             // Target is in lower part
15.          high = pos - 1
16.  return -1
```

Interpolation Search

- Worst-case performance: O(n)
- Best-case performance: O(1)
- Average performance depends on how uniformly distributed the values are. It can be much better than O(log n) if the values are very uniform.

Data structures we study that support efficient searching

- Hash Table (Hashing)
- Binary Search Tree

Sorting Algorithms

Commonly used/discussed:

- Bubble sort
- Insertion sort
- Selection sort
- Shell sort
- Merge sort
- Quick sort

Other algorithms:

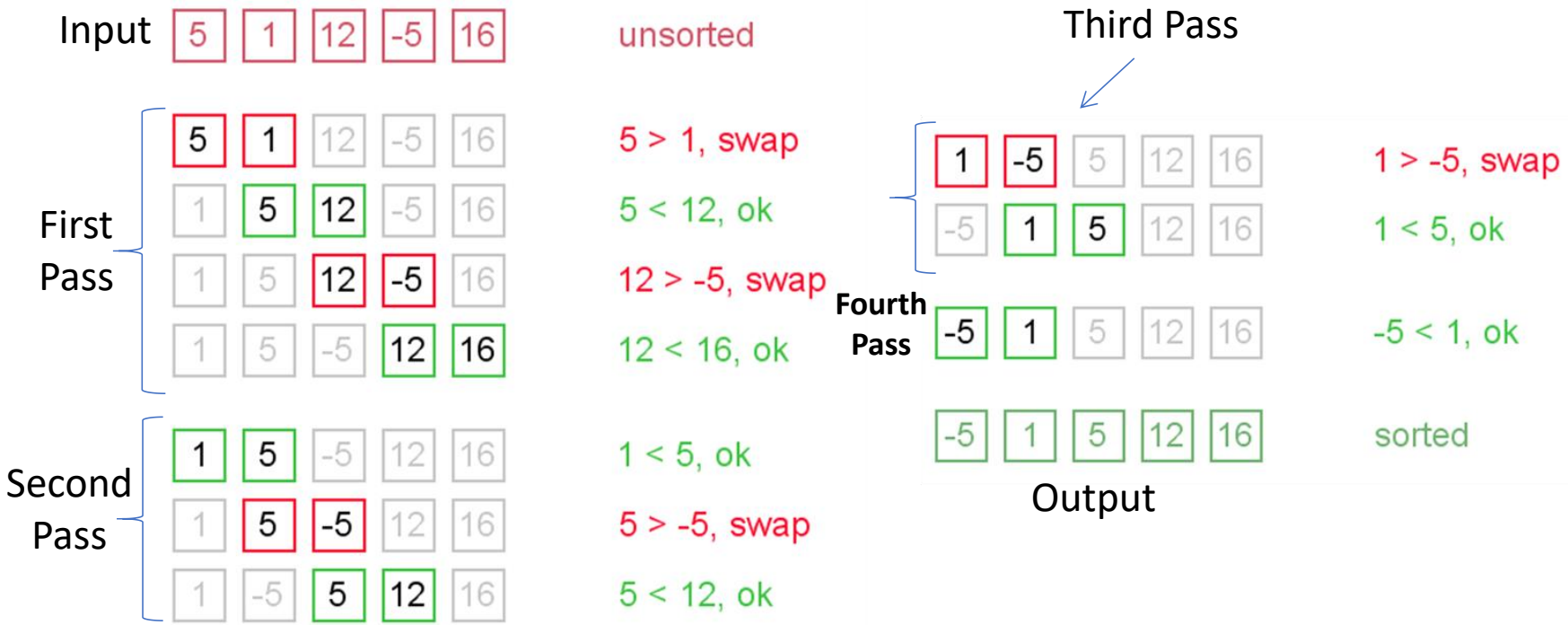
- Heap sort
- Radix sort
- Counting sort
- Bucket sort

Bubble Sort

- Compare each pair of adjacent elements from the beginning of an array and, if they are in reversed order, swap them.

- If at least one swap has been done, repeat above step.

Bubble Sort: Example



Bubble Sort: Algorithm

- BubbleSort (A)

1. *swapped* = true → 1Unit
2. **while**(*swapped* == true) → 1 Unit, Maximum *n*-1 times
3. *swapped* = false → 1Unit
4. **for** i = 1 to *n*-1 → 1Unit for initialization, *n* times comparison, *n*-1 times increment
5. **if** A[*i*] > A[*i*+1] → 1Unit
6. *temp* = A[i] → 1Unit
7. A[*i*] = A[i+1] → 1Unit
8. A[*i*+1] = *temp* → 1Unit
9. *swapped* = true → 1Unit

n-1 *times*

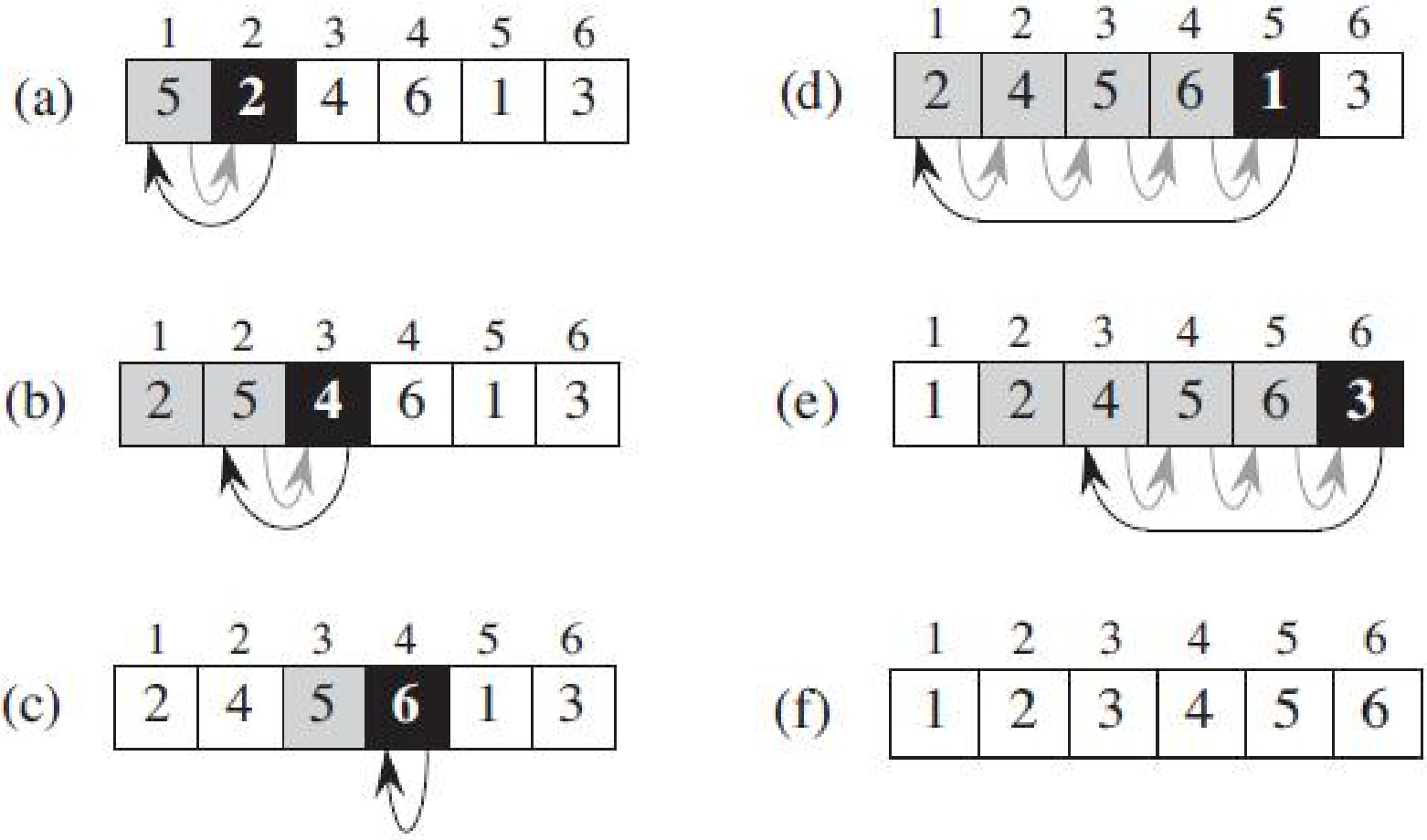
$T(n) = 7n^2-5n+1$

$O(n^2)$

Insertion Sort

- Array is imagined as two parts sorted part and unsorted part.
- At the beginning, sorted part contains first element of the array and unsorted part contains the remaining.
- At every step, algorithm takes first element in the unsorted part and inserts it to the right place of the sorted part
- When unsorted part becomes empty, algorithm stops.

Insertion Sort: Example



Insertion Sort: Time complexity

For each ***j*** = 2, 3, …, *n*, number of times while loop test in line 5 is executed will be ***j***

```
INSERTION-SORT(A)
1  for j = 2 to n                          → 1Unit for initialization, n times comparison, n - 1times increment
2      key = A[j]                          → 1 Units, Maximum n-1 times in worst case
3      // Insert A[j] into the sorted      → 0 Units, because, its comment line :)
           sequence A[1 .. j - 1].
4      i = j - 1                           → 2 Units, Maximum n-1 times in worst case
5      while i > 0 and A[i] > key          → 2 Units, Maximum j times in worst case
6          A[i + 1] = A[i]                 → 2 Units, Maximum j-1 times in worst case
7          i = i - 1                       → 2 Units, Maximum j-1 times in worst case
8      A[i + 1] = key                      → 2 Units, Maximum n-1 times in worst case
```

Insertion Sort: Time complexity

We know that, $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$ and $\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$

Therefore,

$$T(n) = (1 + n + n - 1) + (n - 1) + 2(n - 1) + 2\left[\frac{n\,(n+1)}{2} - 1\right] +$$
$$2.\frac{n\,(n-1)}{2} + 2.\frac{n\,(n-1)}{2} + 2(n - 1)$$

$$= 2n + 3n - 3 + n^2 + n - 2 + n^2 - n + n^2 - n + 2n - 2$$

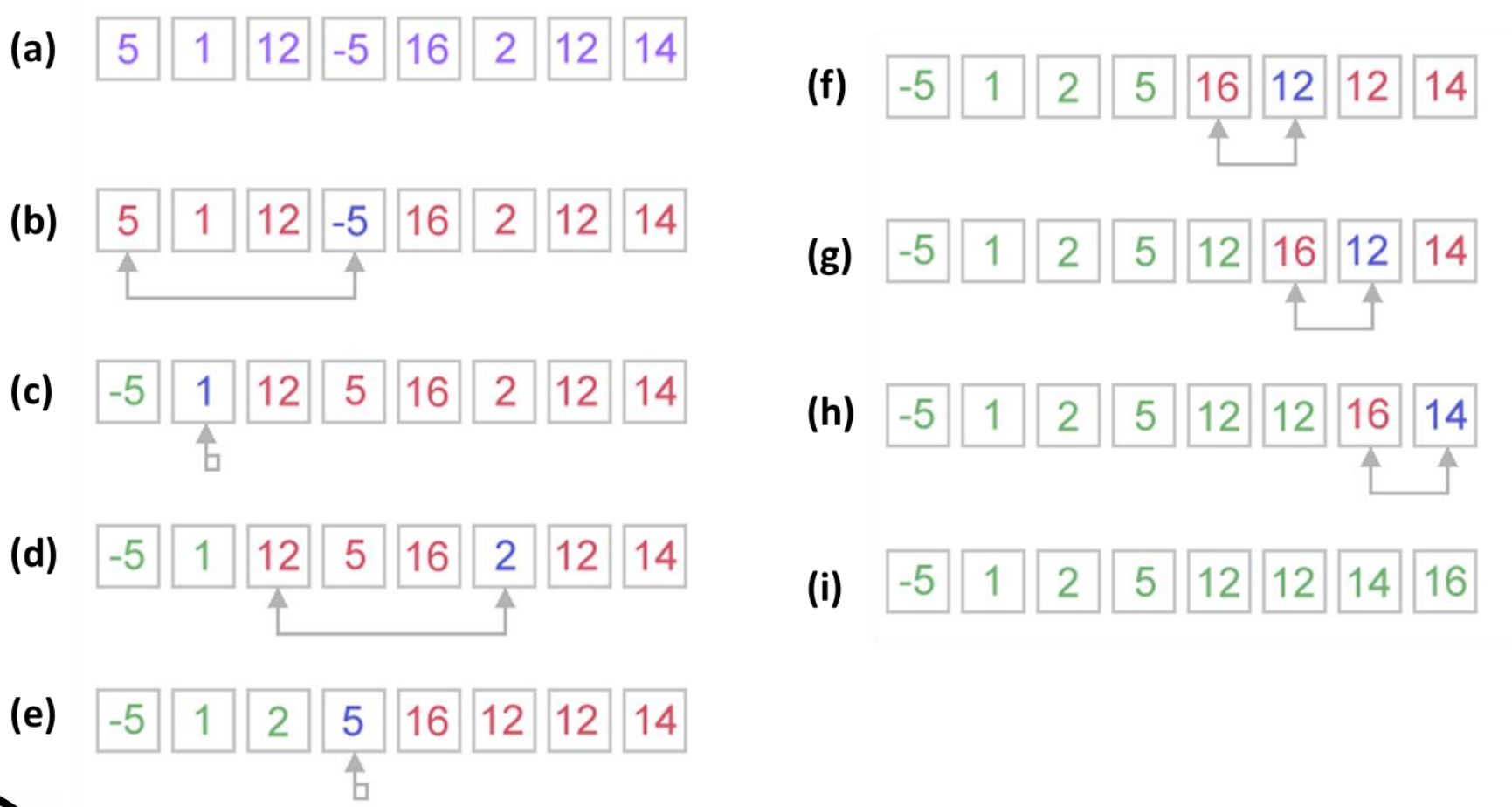
$$= 3n^2 + 6n - 7$$

Hence, $\boldsymbol{T(n) = O(n^2)}$

Selection Sort

- Like insertion sort, array is imagined as two parts - sorted part and unsorted part.
- At the beginning, sorted part is empty, while unsorted one contains whole array.
- At every step, algorithm finds smallest element in the unsorted part and adds it to the end of the sorted part.
- When unsorted part becomes empty, algorithm stops.

Selection Sort: Example



Selection Sort: Time complexity

- SelectionSort (A)

1. **for** *i* = 1 to *n* → 1Unit for initialization, *n+1* times comparison, *n* times increment
2. *min* = *i* → 1Unit, *n* times
3. **for** *j* = *i* + 1 to n → 2Units for initialization, (*n-i*)+1 times comparison, (*n-i*) times increment (times *n, due to outer loop*)
4. **if** (A[*j*] < A[*min*]) → 1Unit, (*n-i*) times (times *n, due to outer loop*)
5. *min* = *j* → 1Unit, (*n-i*) times (times *n, due to outer loop*)
6. **if** (*min* != *i*) → 1Unit, *n* times
7. *temp* = A[*i*] → 1Unit, *n* times (*n*/2 for a reversed array ?)
8. A[*i*] = A[*min*] → 1Unit, *n* times (*n*/2 for a reversed array ?)
9. A[*min*] = *temp* → 1Unit, *n* times (*n*/2 for a reversed array ?)

Selection Sort: Time complexity

$$T(n) = (1 + n + 1 + n) + n + 2n + \sum_{i=1}^{n} (n - i + 1) + \sum_{i=1}^{n} (n - i) + \sum_{i=1}^{n} (n - i)$$

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

$$= 2 + 5n + \frac{n(n+1)}{2} + \frac{n(n-1)}{2} + \frac{n(n-1)}{2} + \frac{n(n-1)}{2} + 4n$$

$$= 2n^2 + 8n + 2$$

Hence, $\boldsymbol{T(n) = O(n^2)}$

Searching and Sorting Algorithms Animation

- https://www.toptal.com/developers/sorting-algorithms
- Bubble Sort
- Insertion Sort
- Selection Sort
- Merge Sort
- Quick Sort
- Jump Search

Merge Sort

- Merge Sort is a divide-and-conquer algorithm
- Works by recursively splitting a list in half.
- Once a single-element list is obtained, merge them back together in sorted order.

- Three Major operations:

1) **Divide**: The list is split into two halves.
2) **Conquer**: Recursively sort both halves.
3) **Merge**: Merge (combine) the sorted halves to produce a single sorted list.

Merge Sort

1. Given a sequence (array) of *n* elements,
   $a[1], a[2], \ldots \ldots, a[n]$

2. Split the array into two sets
   $a[1], \ldots \ldots, a[\lfloor \frac{n}{2} \rfloor]$ and $a[\lfloor \frac{n}{2} \rfloor + 1], \ldots \ldots, a[n]$

3. Each of these individual arrays is sorted and

4. The resulting sorted arrays are merged to produce a single sorted array of *n* elements

Merge Sort: Algorithm

MergeSort(*low*, *high*)

//*low* and *high* are 1st and last positions in the given list

1. **if** (*low* < *high*) then
2. *mid* = ⌊(*low* + *high*)/2⌋ //computing *mid* index
3. MergeSort(*low*, *mid*)
4. MergeSort(*mid* +1, *high*)
5. Merge(*low*, *mid*, *high*)

Merge Sort: Algorithm

Merge Procedure in Mergesort:

Merge(*low*, *mid*, *high*)

1. $h = low,\quad i = low,\quad j = mid + 1$
2. **while** ($h \leq mid$ and $j \leq high$)
3. **if** ($a[h] \leq a[\,j\,]$)
4. $b[i] = a[h]$ //*b* is a new array
5. $h = h + 1$
6. **else**
7. $b[i] = a[\,j\,]$
8. $j = j + 1$
9. $i = i + 1$

This is the reason for which Merge sort is 'not in-place' algorithm (or 'out-of-place' algorithm)

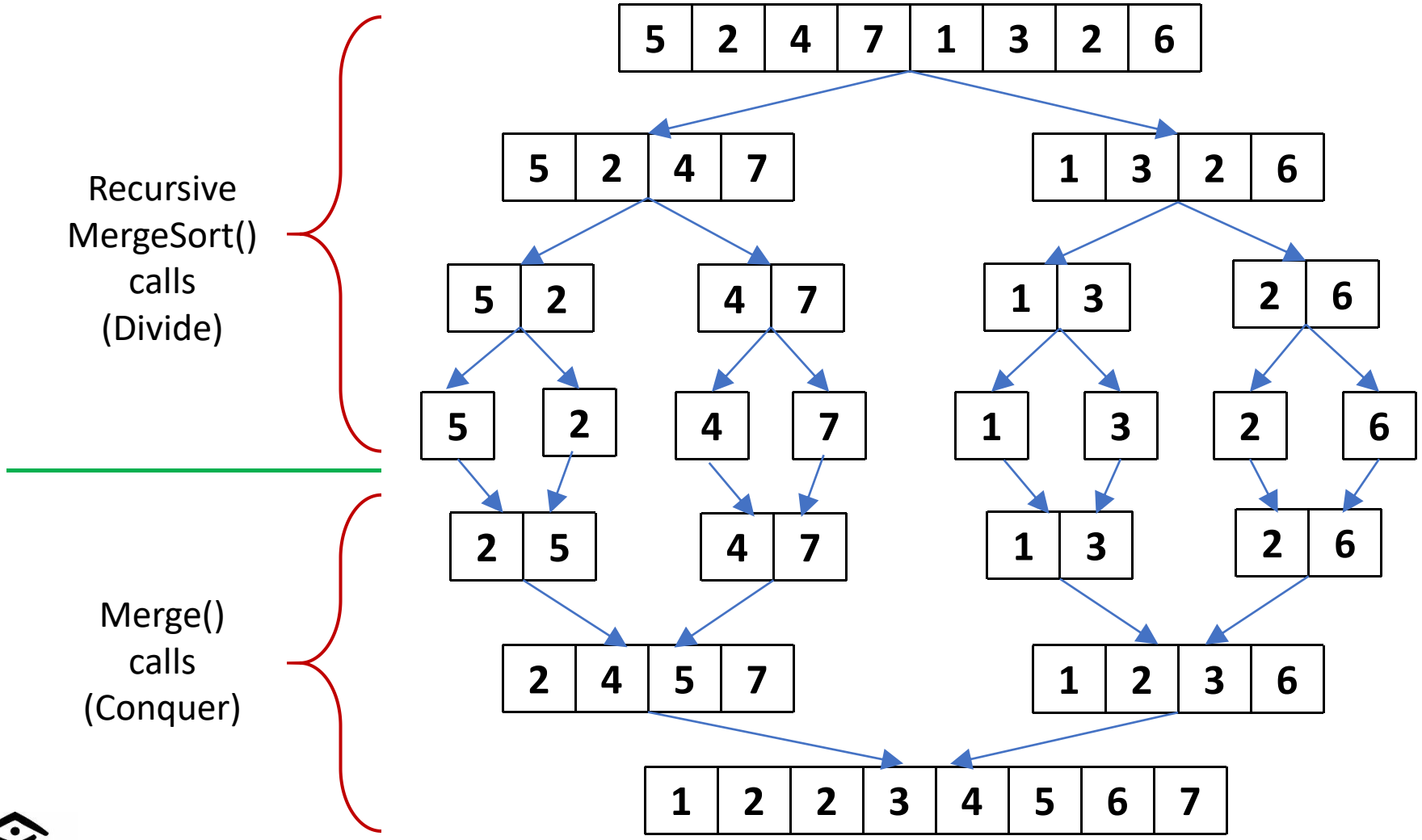
(contd..on next slide)

Merge Sort: Algorithm

Merge Procedure in Mergesort : (contd…)

```
10. if (h > mid )
11.         for k = j to high
12.                 b[i] = a[k]
13.                 i = i +1
14.   else
15.         for k = h to mid
16.                 b[i] = a[k]
17.                 i = i +1
18.   for k = low to high
19.                 a[k] = b[k]
```

Merge Sort: Example 1



Merge Sort: Tracing out the execution sequence

```
MergeSort( low, high)                (1, 8)
    if(low < high)                   (1 < 8 - TRUE)
        mid = ⌊(low + high) / 2⌋      ( 4 )
        MergeSort(low, mid)          (1, 4)
        MergeSort(mid+1, high)
        Merge(low, mid, high)
```

1

Merge Sort: Tracing out the execution sequence

```
MergeSort( low, high)                (1, 4)
    if(low < high)                   (1 < 4 - TRUE)
        mid = ⌊(low + high) / 2⌋      ( 2 )
        MergeSort(low, mid)          (1, 2)
        MergeSort(mid+1, high)
        Merge(low, mid, high)
```

2

```
MergeSort( low, high)                (1, 8)
    if(low < high)                   (1 < 8 - TRUE)
        mid = ⌊(low + high) / 2⌋      ( 4 )
        MergeSort(low, mid)          (1, 4)
        MergeSort(mid+1, high)
        Merge(low, mid, high)
```

1

Merge Sort: Tracing out the execution sequence

```
MergeSort( low, high)                    (1, 2)
    if(low < high)                  (1 < 2 - TRUE)
        mid = ⌊(low + high) / 2⌋          ( 1 )
        MergeSort(low, mid)              (1, 1)
        MergeSort(mid+1, high)
        Merge(low, mid, high)
```

```
MergeSort( low, high)                    (1, 4)
    if(low < high)                  (1 < 4 - TRUE)
        mid = ⌊(low + high) / 2⌋          ( 2 )
        MergeSort(low, mid)              (1, 2)
        MergeSort(mid+1, high)
        Merge(low, mid, high)
```

```
MergeSort( low, high)                    (1, 8)
    if(low < high)                  (1 < 8 - TRUE)
        mid = ⌊(low + high) / 2⌋          ( 4 )
        MergeSort(low, mid)              (1, 4)
        MergeSort(mid+1, high)
        Merge(low, mid, high)
```

3

2

1

Merge Sort: Tracing out the execution sequence

```
MergeSort( low, high)                    (1, 1)
   if(low < high)                  (1 < 1 - FALSE)
      mid = ⌊(low + high) / 2⌋
      MergeSort(low, mid)
      MergeSort(mid+1, high)
      Merge(low, mid, high)
```

4

```
MergeSort( low, high)                    (1, 2)
   if(low < high)                  (1 < 2 - TRUE)
      mid = ⌊(low + high) / 2⌋          ( 1 )
      MergeSort(low, mid)              (1, 1)
      MergeSort(mid+1, high)
      Merge(low, mid, high)
```

3

```
MergeSort( low, high)                    (1, 4)
   if(low < high)                  (1 < 4 - TRUE)
      mid = ⌊(low + high) / 2⌋          ( 2 )
      MergeSort(low, mid)              (1, 2)
      MergeSort(mid+1, high)
      Merge(low, mid, high)
```

2

```
MergeSort( low, high)                    (1, 8)
   if(low < high)                  (1 < 8 - TRUE)
      mid = ⌊(low + high) / 2⌋          ( 4 )
      MergeSort(low, mid)              (1, 4)
      MergeSort(mid+1, high)
      Merge(low, mid, high)
```

1

Merge Sort: Tracing out the execution sequence

```
MergeSort( low, high)                    (1, 2)
    if(low < high)                  (1 < 2 - TRUE)
        mid = ⌊(low + high) / 2⌋          ( 1 )
        MergeSort(low, mid)              (1, 1)
        MergeSort(mid+1, high)           (2, 2)
        Merge(low, mid, high)
```

3

```
MergeSort( low, high)                    (1, 4)
    if(low < high)                  (1 < 4 - TRUE)
        mid = ⌊(low + high) / 2⌋          ( 2 )
        MergeSort(low, mid)              (1, 2)
        MergeSort(mid+1, high)
        Merge(low, mid, high)
```

2

```
MergeSort( low, high)                    (1, 8)
    if(low < high)                  (1 < 8 - TRUE)
        mid = ⌊(low + high) / 2⌋          ( 4 )
        MergeSort(low, mid)              (1, 4)
        MergeSort(mid+1, high)
        Merge(low, mid, high)
```

1

Merge Sort: Tracing out the execution sequence

```
MergeSort( low, high)                    (2, 2)
    if(low < high)                  (2 < 2 - FALSE)
        mid = ⌊(low + high) / 2⌋
        MergeSort(low, mid)
        MergeSort(mid+1, high)
        Merge(low, mid, high)
```

4

```
MergeSort( low, high)                    (1, 2)
    if(low < high)                  (1 < 2 - TRUE)
        mid = ⌊(low + high) / 2⌋          ( 1 )
        MergeSort(low, mid)              (1, 1)
        MergeSort(mid+1, high)           (2, 2)
        Merge(low, mid, high)
```

3

```
MergeSort( low, high)                    (1, 4)
    if(low < high)                  (1 < 4 - TRUE)
        mid = ⌊(low + high) / 2⌋          ( 2 )
        MergeSort(low, mid)              (1, 2)
        MergeSort(mid+1, high)
        Merge(low, mid, high)
```

2

```
MergeSort( low, high)                    (1, 8)
    if(low < high)                  (1 < 8 - TRUE)
        mid = ⌊(low + high) / 2⌋          ( 4 )
        MergeSort(low, mid)              (1, 4)
        MergeSort(mid+1, high)
        Merge(low, mid, high)
```

1

Merge Sort: Tracing out the execution sequence

```
MergeSort( low, high)                    (1, 2)
    if(low < high)                  (1 < 2 - TRUE)
        mid = ⌊(low + high) / 2 ⌋          ( 1 )
        MergeSort(low, mid)             (1, 1)
        MergeSort(mid+1, high)          (2, 2)
        Merge(low, mid, high)           (1, 1, 2)
```

3

```
MergeSort( low, high)                    (1, 4)
    if(low < high)                  (1 < 4 - TRUE)
        mid = ⌊(low + high) / 2⌋           ( 2 )
        MergeSort(low, mid)             (1, 2)
        MergeSort(mid+1, high)
        Merge(low, mid, high)
```

2

```
MergeSort( low, high)                    (1, 8)
    if(low < high)                  (1 < 8 - TRUE)
        mid = ⌊(low + high) / 2 ⌋          ( 4 )
        MergeSort(low, mid)             (1, 4)
        MergeSort(mid+1, high)
        Merge(low, mid, high)
```

1

Merge Sort: Tracing out the execution sequence

```
MergeSort( low, high)                (1, 4)
    if(low < high)                   (1 < 4 - TRUE)
        mid = ⌊(low + high) / 2⌋      ( 2 )
        MergeSort(low, mid)          (1, 2)
        MergeSort(mid+1, high)       (3, 4)
        Merge(low, mid, high)
```

2

```
MergeSort( low, high)                (1, 8)
    if(low < high)                   (1 < 8 - TRUE)
        mid = ⌊(low + high) / 2⌋      ( 4 )
        MergeSort(low, mid)          (1, 4)
        MergeSort(mid+1, high)
        Merge(low, mid, high)
```

1

Merge Sort: Tracing out the execution sequence

```
MergeSort( low, high)                (3, 4)
    if(low < high)                   (3 < 4 - TRUE)
        mid = ⌊(low + high) / 2 ⌋     (3)
        MergeSort(low, mid)          (3, 3)
        MergeSort(mid+1, high)
        Merge(low, mid, high)
```

3

```
MergeSort( low, high)                (1, 4)
    if(low < high)                   (1 < 4 - TRUE)
        mid = ⌊(low + high) / 2⌋      ( 2 )
        MergeSort(low, mid)          (1, 2)
        MergeSort(mid+1, high)       (3, 4)
        Merge(low, mid, high)
```

2

```
MergeSort( low, high)                (1, 8)
    if(low < high)                   (1 < 8 - TRUE)
        mid = ⌊(low + high) / 2 ⌋     ( 4 )
        MergeSort(low, mid)          (1, 4)
        MergeSort(mid+1, high)
        Merge(low, mid, high)
```

1

Merge Sort: Tracing out the execution sequence

```
MergeSort( low, high)                (3, 3)
    if(low < high)                   (3 < 3 - FALSE)
        mid = ⌊(low + high) / 2 ⌋
        MergeSort(low, mid)
        MergeSort(mid+1, high)
        Merge(low, mid, high)
```

4

```
MergeSort( low, high)                (3, 4)
    if(low < high)                   (3 < 4 - TRUE)
        mid = ⌊(low + high) / 2 ⌋     (3)
        MergeSort(low, mid)          (3, 3)
        MergeSort(mid+1, high)
        Merge(low, mid, high)
```

3

```
MergeSort( low, high)                (1, 4)
    if(low < high)                   (1 < 4 - TRUE)
        mid = ⌊(low + high) / 2⌋     ( 2 )
        MergeSort(low, mid)          (1, 2)
        MergeSort(mid+1, high)       (3, 4)
        Merge(low, mid, high)
```

2

```
MergeSort( low, high)                (1, 8)
    if(low < high)                   (1 < 8 - TRUE)
        mid = ⌊(low + high) / 2 ⌋    ( 4 )
        MergeSort(low, mid)          (1, 4)
        MergeSort(mid+1, high)
        Merge(low, mid, high)
```

1

Merge Sort: Tracing out the execution sequence

```
MergeSort( low, high)                (3, 4)
    if(low < high)                   (3 < 4 - TRUE)
        mid = ⌊(low + high) / 2 ⌋     (3)
        MergeSort(low, mid)          (3, 3)
        MergeSort(mid+1, high)       (4, 4)
        Merge(low, mid, high)
```

3

```
MergeSort( low, high)                (1, 4)
    if(low < high)                   (1 < 4 - TRUE)
        mid = ⌊(low + high) / 2⌋      ( 2 )
        MergeSort(low, mid)          (1, 2)
        MergeSort(mid+1, high)       (3, 4)
        Merge(low, mid, high)
```

2

```
MergeSort( low, high)                (1, 8)
    if(low < high)                   (1 < 8 - TRUE)
        mid = ⌊(low + high) / 2 ⌋     ( 4 )
        MergeSort(low, mid)          (1, 4)
        MergeSort(mid+1, high)
        Merge(low, mid, high)
```

1

Merge Sort: Tracing out the execution sequence

```
MergeSort( low, high)                    (4, 4)
    if(low < high)                  (4 < 4 - FALSE)
        mid = ⌊(low + high) / 2 ⌋
        MergeSort(low, mid)
        MergeSort(mid+1, high)
        Merge(low, mid, high)
```

4

```
MergeSort( low, high)                    (3, 4)
    if(low < high)                  (3 < 4 - TRUE)
        mid = ⌊(low + high) / 2 ⌋            (3)
        MergeSort(low, mid)              (3, 3)
        MergeSort(mid+1, high)           (4, 4)
        Merge(low, mid, high)
```

3

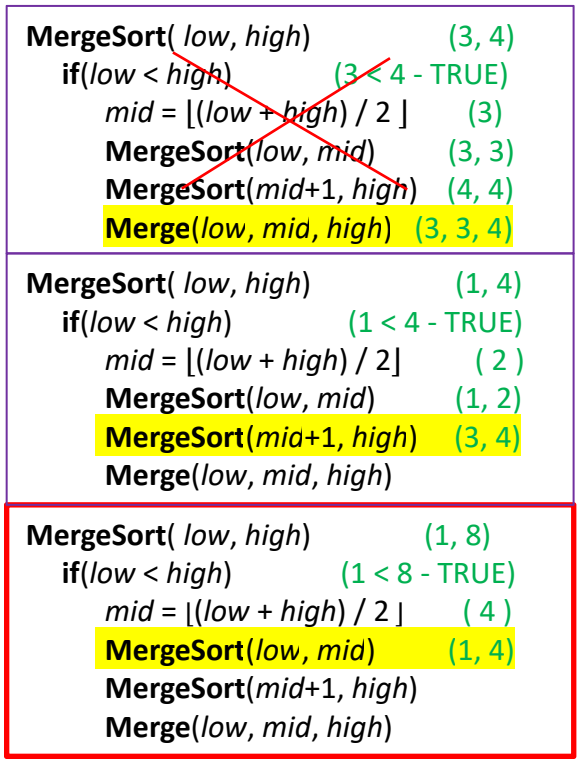
```
MergeSort( low, high)                    (1, 4)
    if(low < high)                  (1 < 4 - TRUE)
        mid = ⌊(low + high) / 2⌋             ( 2 )
        MergeSort(low, mid)              (1, 2)
        MergeSort(mid+1, high)           (3, 4)
        Merge(low, mid, high)
```

2

```
MergeSort( low, high)                    (1, 8)
    if(low < high)                  (1 < 8 - TRUE)
        mid = ⌊(low + high) / 2 ⌋            ( 4 )
        MergeSort(low, mid)              (1, 4)
        MergeSort(mid+1, high)
        Merge(low, mid, high)
```

1

Merge Sort: Tracing out the execution sequence



3

2

1

Merge Sort: Tracing out the execution sequence

```
MergeSort( low, high)                (1, 4)
    if(low < high)                   (1 < 4 - TRUE)
        mid = ⌊(low + high) / 2⌋      ( 2 )
        MergeSort(low, mid)          (1, 2)
        MergeSort(mid+1, high)       (3, 4)
        Merge(low, mid, high)        (1, 2, 4)
```

2

```
MergeSort( low, high)                (1, 8)
    if(low < high)                   (1 < 8 - TRUE)
        mid = ⌊(low + high) / 2⌋      ( 4 )
        MergeSort(low, mid)          (1, 4)
        MergeSort(mid+1, high)
        Merge(low, mid, high)
```

1

Merge Sort: Tracing out the execution sequence

```
MergeSort( low, high)                (1, 8)
    if(low < high)                   (1 < 8 - TRUE)
        mid = ⌊(low + high) / 2⌋     ( 4 )
        MergeSort(low, mid)          (1, 4)
        MergeSort(mid+1, high)       (5, 8)
        Merge(low, mid, high)
```

1

Merge Sort: Tracing out the execution sequence

```
MergeSort( low, high)                (5, 8)
    if(low < high)                   (5 < 8 - TRUE)
        mid = ⌊(low + high) / 2⌋      ( 6 )
        MergeSort(low, mid)          (5, 6)
        MergeSort(mid+1, high)
        Merge(low, mid, high)
```

2

```
MergeSort( low, high)                (1, 8)
    if(low < high)                   (1 < 8 - TRUE)
        mid = ⌊(low + high) / 2⌋      ( 4 )
        MergeSort(low, mid)          (1, 4)
        MergeSort(mid+1, high)       (5, 8)
        Merge(low, mid, high)
```

1

Merge Sort: Tracing out the execution sequence

```
MergeSort( low, high)                    (5, 6)
    if(low < high)                  (5 < 6 - TRUE)
        mid = ⌊(low + high) / 2⌋          ( 5 )
        MergeSort(low, mid)              (5, 5)
        MergeSort(mid+1, high)
        Merge(low, mid, high)
```

3

```
MergeSort( low, high)                    (5, 8)
    if(low < high)                  (5 < 8 - TRUE)
        mid = ⌊(low + high) / 2⌋          ( 6 )
        MergeSort(low, mid)              (5, 6)
        MergeSort(mid+1, high)
        Merge(low, mid, high)
```

2

```
MergeSort( low, high)                    (1, 8)
    if(low < high)                  (1 < 8 - TRUE)
        mid = ⌊(low + high) / 2⌋          ( 4 )
        MergeSort(low, mid)              (1, 4)
        MergeSort(mid+1, high)           (5, 8)
        Merge(low, mid, high)
```

1

Merge Sort: Tracing out the execution sequence

```
MergeSort( low, high)                    (5, 5)
   if(low < high)                (5 < 5 - FALSE)
      mid = ⌊(low + high) / 2 ⌋
      MergeSort(low, mid)
      MergeSort(mid+1, high)
      Merge(low, mid, high)
```

4

```
MergeSort( low, high)                    (5, 6)
   if(low < high)                (5 < 6 - TRUE)
      mid = ⌊(low + high) / 2 ⌋          ( 5 )
      MergeSort(low, mid)            (5, 5)
      MergeSort(mid+1, high)
      Merge(low, mid, high)
```

3

```
MergeSort( low, high)                    (5, 8)
   if(low < high)                (5 < 8 - TRUE)
      mid = ⌊(low + high) / 2⌋           ( 6 )
      MergeSort(low, mid)            (5, 6)
      MergeSort(mid+1, high)
      Merge(low, mid, high)
```

2

```
MergeSort( low, high)                    (1, 8)
   if(low < high)                (1 < 8 - TRUE)
      mid = ⌊(low + high) / 2 ⌋          ( 4 )
      MergeSort(low, mid)            (1, 4)
      MergeSort(mid+1, high)         (5, 8)
      Merge(low, mid, high)
```

1

Merge Sort: Tracing out the execution sequence

```
MergeSort( low, high)                    (5, 6)
    if(low < high)                  (5 < 6 - TRUE)
        mid = ⌊(low + high) / 2 ⌋          ( 5 )
        MergeSort(low, mid)             (5, 5)
        MergeSort(mid+1, high)          (6, 6)
        Merge(low, mid, high)
```

3

```
MergeSort( low, high)                    (5, 8)
    if(low < high)                  (5 < 8 - TRUE)
        mid = ⌊(low + high) / 2⌋           ( 6 )
        MergeSort(low, mid)             (5, 6)
        MergeSort(mid+1, high)
        Merge(low, mid, high)
```

2

```
MergeSort( low, high)                    (1, 8)
    if(low < high)                  (1 < 8 - TRUE)
        mid = ⌊(low + high) / 2 ⌋          ( 4 )
        MergeSort(low, mid)             (1, 4)
        MergeSort(mid+1, high)          (5, 8)
        Merge(low, mid, high)
```

1

Merge Sort: Tracing out the execution sequence

```
MergeSort( low, high)                (6, 6)
   if(low < high)                    (6 < 6 - FALSE)
      mid = ⌊(low + high) / 2 ⌋
      MergeSort(low, mid)
      MergeSort(mid+1, high)
      Merge(low, mid, high)
```

4

```
MergeSort( low, high)                (5, 6)
   if(low < high)                    (5 < 6 - TRUE)
      mid = ⌊(low + high) / 2 ⌋       ( 5 )
      MergeSort(low, mid)            (5, 5)
      MergeSort(mid+1, high)         (6, 6)
      Merge(low, mid, high)
```

3

```
MergeSort( low, high)                (5, 8)
   if(low < high)                    (5 < 8 - TRUE)
      mid = ⌊(low + high) / 2⌋        ( 6 )
      MergeSort(low, mid)            (5, 6)
      MergeSort(mid+1, high)
      Merge(low, mid, high)
```

2

```
MergeSort( low, high)                (1, 8)
   if(low < high)                    (1 < 8 - TRUE)
      mid = ⌊(low + high) / 2 ⌋       ( 4 )
      MergeSort(low, mid)            (1, 4)
      MergeSort(mid+1, high)         (5, 8)
      Merge(low, mid, high)
```

1

Merge Sort: Tracing out the execution sequence

```
MergeSort( low, high)                    (5, 6)
    if(low < high)                  (5 < 6 - TRUE)
        mid = ⌊(low + high) / 2 ⌋          ( 5 )
        MergeSort(low, mid)             (5, 5)
        MergeSort(mid+1, high)          (6, 6)
        Merge(low, mid, high)           (5, 5, 6)
```

3

```
MergeSort( low, high)                    (5, 8)
    if(low < high)                  (5 < 8 - TRUE)
        mid = ⌊(low + high) / 2⌋           ( 6 )
        MergeSort(low, mid)             (5, 6)
        MergeSort(mid+1, high)
        Merge(low, mid, high)
```

2

```
MergeSort( low, high)                    (1, 8)
    if(low < high)                  (1 < 8 - TRUE)
        mid = ⌊(low + high) / 2 ⌋          ( 4 )
        MergeSort(low, mid)             (1, 4)
        MergeSort(mid+1, high)          (5, 8)
        Merge(low, mid, high)
```

1

Merge Sort: Tracing out the execution sequence

```
MergeSort( low, high)                (5, 8)
    if(low < high)                   (5 < 8 - TRUE)
        mid = ⌊(low + high) / 2⌋      ( 6 )
        MergeSort(low, mid)          (5, 6)
        MergeSort(mid+1, high)       (7, 8)
        Merge(low, mid, high)
```

2

```
MergeSort( low, high)                (1, 8)
    if(low < high)                   (1 < 8 - TRUE)
        mid = ⌊(low + high) / 2⌋      ( 4 )
        MergeSort(low, mid)          (1, 4)
        MergeSort(mid+1, high)       (5, 8)
        Merge(low, mid, high)
```

1

Merge Sort: Tracing out the execution sequence

```
MergeSort( low, high)                (7, 8)
    if(low < high)                   (7 < 8 - TRUE)
        mid = ⌊(low + high) / 2 ⌋    (7)
        MergeSort(low, mid)          (7, 7)
        MergeSort(mid+1, high)
        Merge(low, mid, high)
```

3

```
MergeSort( low, high)                (5, 8)
    if(low < high)                   (5 < 8 - TRUE)
        mid = ⌊(low + high) / 2⌋     ( 6 )
        MergeSort(low, mid)          (5, 6)
        MergeSort(mid+1, high)       (7, 8)
        Merge(low, mid, high)
```

2

```
MergeSort( low, high)                (1, 8)
    if(low < high)                   (1 < 8 - TRUE)
        mid = ⌊(low + high) / 2 ⌋    ( 4 )
        MergeSort(low, mid)          (1, 4)
        MergeSort(mid+1, high)       (5, 8)
        Merge(low, mid, high)
```

1

Merge Sort: Tracing out the execution sequence

```
MergeSort( low, high)                (7, 7)
    if(low < high)                   (7 < 7 - FALSE)
        mid = ⌊(low + high) / 2 ⌋
        MergeSort(low, mid)
        MergeSort(mid+1, high)
        Merge(low, mid, high)
```

4

```
MergeSort( low, high)                (7, 8)
    if(low < high)                   (7 < 8 - TRUE)
        mid = ⌊(low + high) / 2 ⌋     (7)
        MergeSort(low, mid)          (7, 7)
        MergeSort(mid+1, high)
        Merge(low, mid, high)
```

3

```
MergeSort( low, high)                (5, 8)
    if(low < high)                   (5 < 8 - TRUE)
        mid = ⌊(low + high) / 2⌋     ( 6 )
        MergeSort(low, mid)          (5, 6)
        MergeSort(mid+1, high)       (7, 8)
        Merge(low, mid, high)
```

2

```
MergeSort( low, high)                (1, 8)
    if(low < high)                   (1 < 8 - TRUE)
        mid = ⌊(low + high) / 2 ⌋    ( 4 )
        MergeSort(low, mid)          (1, 4)
        MergeSort(mid+1, high)       (5, 8)
        Merge(low, mid, high)
```

1

Merge Sort: Tracing out the execution sequence

```
MergeSort( low, high)                (7, 8)
    if(low < high)                   (7 < 8 - TRUE)
        mid = ⌊(low + high) / 2 ⌋    (7)
        MergeSort(low, mid)          (7, 7)
        MergeSort(mid+1, high)       (8, 8)
        Merge(low, mid, high)
```

3

```
MergeSort( low, high)                (5, 8)
    if(low < high)                   (5 < 8 - TRUE)
        mid = ⌊(low + high) / 2⌋     ( 6 )
        MergeSort(low, mid)          (5, 6)
        MergeSort(mid+1, high)       (7, 8)
        Merge(low, mid, high)
```

2

```
MergeSort( low, high)                (1, 8)
    if(low < high)                   (1 < 8 - TRUE)
        mid = ⌊(low + high) / 2 ⌋    ( 4 )
        MergeSort(low, mid)          (1, 4)
        MergeSort(mid+1, high)       (5, 8)
        Merge(low, mid, high)
```

1

Merge Sort: Tracing out the execution sequence

```
MergeSort( low, high)                (8, 8)
   if(low < high)                    (8 < 8 - FALSE)
      mid = ⌊(low + high) / 2 ⌋
      MergeSort(low, mid)
      MergeSort(mid+1, high)
      Merge(low, mid, high)
```

4

```
MergeSort( low, high)                (7, 8)
   if(low < high)                    (7 < 8 - TRUE)
      mid = ⌊(low + high) / 2 ⌋      (7)
      MergeSort(low, mid)            (7, 7)
      MergeSort(mid+1, high)         (8, 8)
      Merge(low, mid, high)
```

3

```
MergeSort( low, high)                (5, 8)
   if(low < high)                    (5 < 8 - TRUE)
      mid = ⌊(low + high) / 2⌋       ( 6 )
      MergeSort(low, mid)            (5, 6)
      MergeSort(mid+1, high)         (7, 8)
      Merge(low, mid, high)
```

2

```
MergeSort( low, high)                (1, 8)
   if(low < high)                    (1 < 8 - TRUE)
      mid = ⌊(low + high) / 2 ⌋      ( 4 )
      MergeSort(low, mid)            (1, 4)
      MergeSort(mid+1, high)         (5, 8)
      Merge(low, mid, high)
```

1

Merge Sort: Tracing out the execution sequence

```
MergeSort( low, high)                (7, 8)
    if(low < high)                   (7 < 8 - TRUE)
        mid = ⌊(low + high) / 2 ⌋    (7)
        MergeSort(low, mid)          (7, 7)
        MergeSort(mid+1, high)       (8, 8)
        Merge(low, mid, high)        (7, 7, 8)
```

3

```
MergeSort( low, high)                (5, 8)
    if(low < high)                   (5 < 8 - TRUE)
        mid = ⌊(low + high) / 2⌋     ( 6 )
        MergeSort(low, mid)          (5, 6)
        MergeSort(mid+1, high)       (7, 8)
        Merge(low, mid, high)
```

2

```
MergeSort( low, high)                (1, 8)
    if(low < high)                   (1 < 8 - TRUE)
        mid = ⌊(low + high) / 2 ⌋    ( 4 )
        MergeSort(low, mid)          (1, 4)
        MergeSort(mid+1, high)       (5, 8)
        Merge(low, mid, high)
```

1

Merge Sort: Tracing out the execution sequence

```
MergeSort( low, high)                    (5, 8)
    if(low < high)                  (5 < 8 - TRUE)
        mid = ⌊(low + high) / 2⌋          ( 6 )
        MergeSort(low, mid)              (5, 6)
        MergeSort(mid+1, high)           (7, 8)
        Merge(low, mid, high)            (5, 7, 8)
```

2

```
MergeSort( low, high)                    (1, 8)
    if(low < high)                  (1 < 8 - TRUE)
        mid = ⌊(low + high) / 2⌋          ( 4 )
        MergeSort(low, mid)              (1, 4)
        MergeSort(mid+1, high)           (5, 8)
        Merge(low, mid, high)
```

1

Merge Sort: Tracing out the execution sequence

```
MergeSort( low, high)                (1, 8)
    if(low < high)                   (1 < 8 - TRUE)
        mid = ⌊(low + high) / 2⌋      ( 4 )
        MergeSort(low, mid)          (1, 4)
        MergeSort(mid+1, high)       (5, 8)
        Merge(low, mid, high)        (1, 4,  8)
```

1

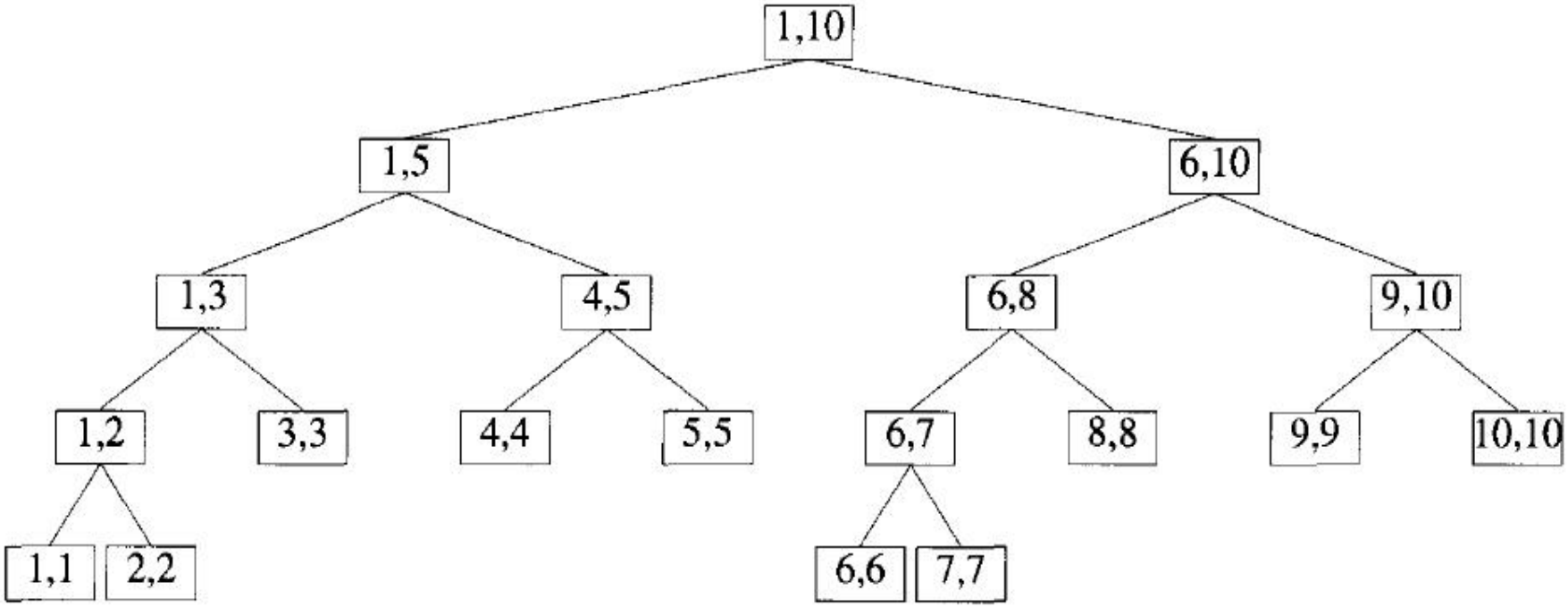
Merge Sort: Tracing out the execution sequence

Done

Merge Sort: Example 2

a = (310, 285,179, 652,351,423,861, 254, 450,520)

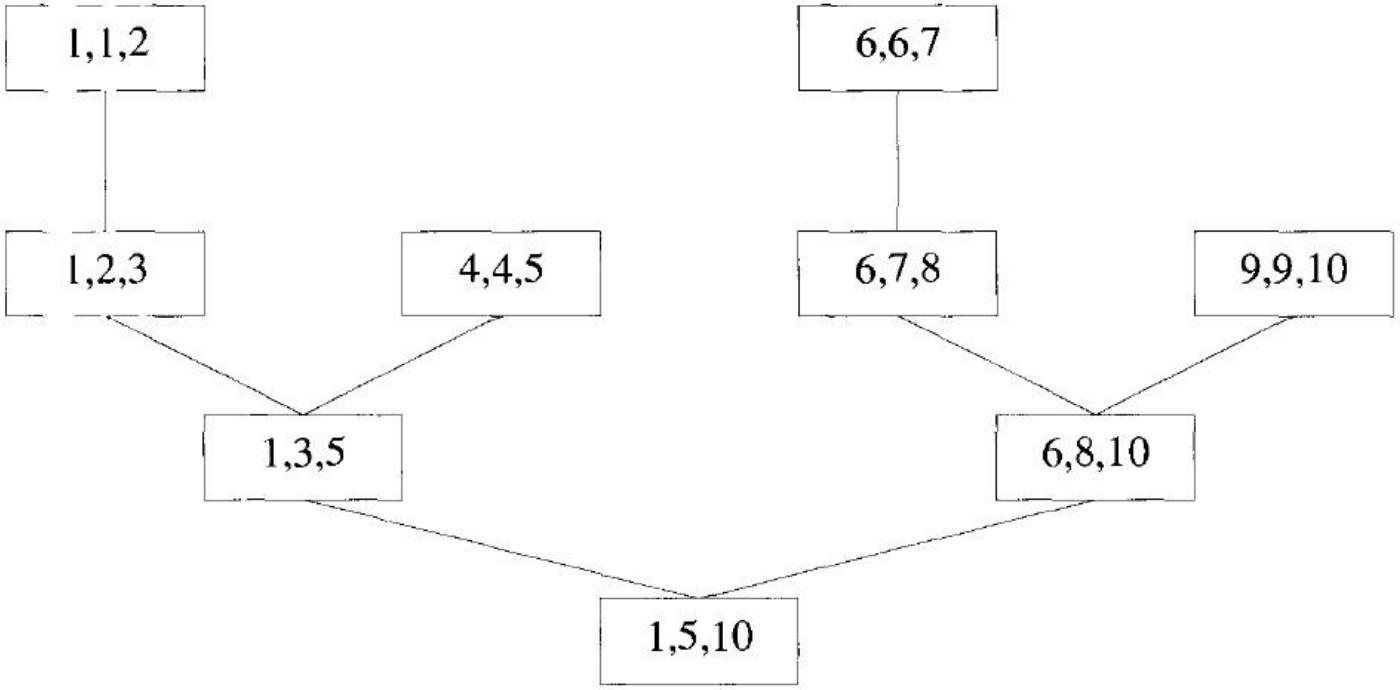
Tree of Calls of MergeSort



Merge Sort: Example 2

a = (310, 285,179, 652,351,423,861, 254, 450,520)

Tree of Calls of Merge Procedure



Merge Sort: Example 2

Position of array after the first recursive call

(310| 285| 179| 652,351| 423,861,254, 450,520)

a[1] merged with *a*[2]

(285,310| 179| 652,351| 423,861,254, 450,520)

a[1], *a*[2] merged with *a*[3]

(179,285,310| 652,351| 423,861,254, 450,520)

a[4] merged with *a*[5]

(179,285,310| 351,652| 423,861,254, 450,520)

a[1], *a*[2], *a*[3] merged with *a*[4], *a*[5]

(179,285,310,351,652| 423,861,254, 450,520)

Merge Sort: Example 2

Position of array after the second recursive call

(179,285,310,351,652| 423 | 861| 254 | 450,520)

a[6] merged with *a*[7]

(179,285,310,351,652| 423 , 861| 254 | 450,520)

a[6], *a*[7] merged with *a*[8]

(179,285,310,351,652| 254, 423,861| 450,520)

a[9] merged with *a*[10]

(179,285,310| 351,652| 254, 423,861| 450,520)

a[6], *a*[7], *a*[8] merged with *a*[9], *a*[10]

(179,285,310,351,652| 254, 423,450,520,861)

Final Merge

(179, 254, 285, 310, 351, 423, 450, 520, 652, 861)

Merge Sort: Time Complexity

- The time for ***merge*** operation is proportional to *n*
- The computing time for Mergesort is described by

$$T(n) = \begin{cases} a & n = 1,\ a \text{ is constant} \\ 2T(\frac{n}{2}) + cn, & n > 1\ c \text{ is constant} \end{cases}$$

We assume *n* is a power of 2 for simplicity,

i.e., $n = 2^k$

We can solve the equation by successive substitutions

Merge Sort: Time Complexity

$$
\begin{aligned}
T(n) &= 2(2T(n/4) + cn/2) + cn \\
&= 4T(n/4) + 2cn \\
&= 4(2T(n/8) + cn/4) + 2cn \\
&\vdots \\
&= 2^k T(1) + kcn \\
&= an + cn \log n
\end{aligned}
$$

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

Quick Sort

- Fastest known sorting algorithm in practice, but, not **stable**

Means elements of equal value in the input array may or may not be in the same order after sorting

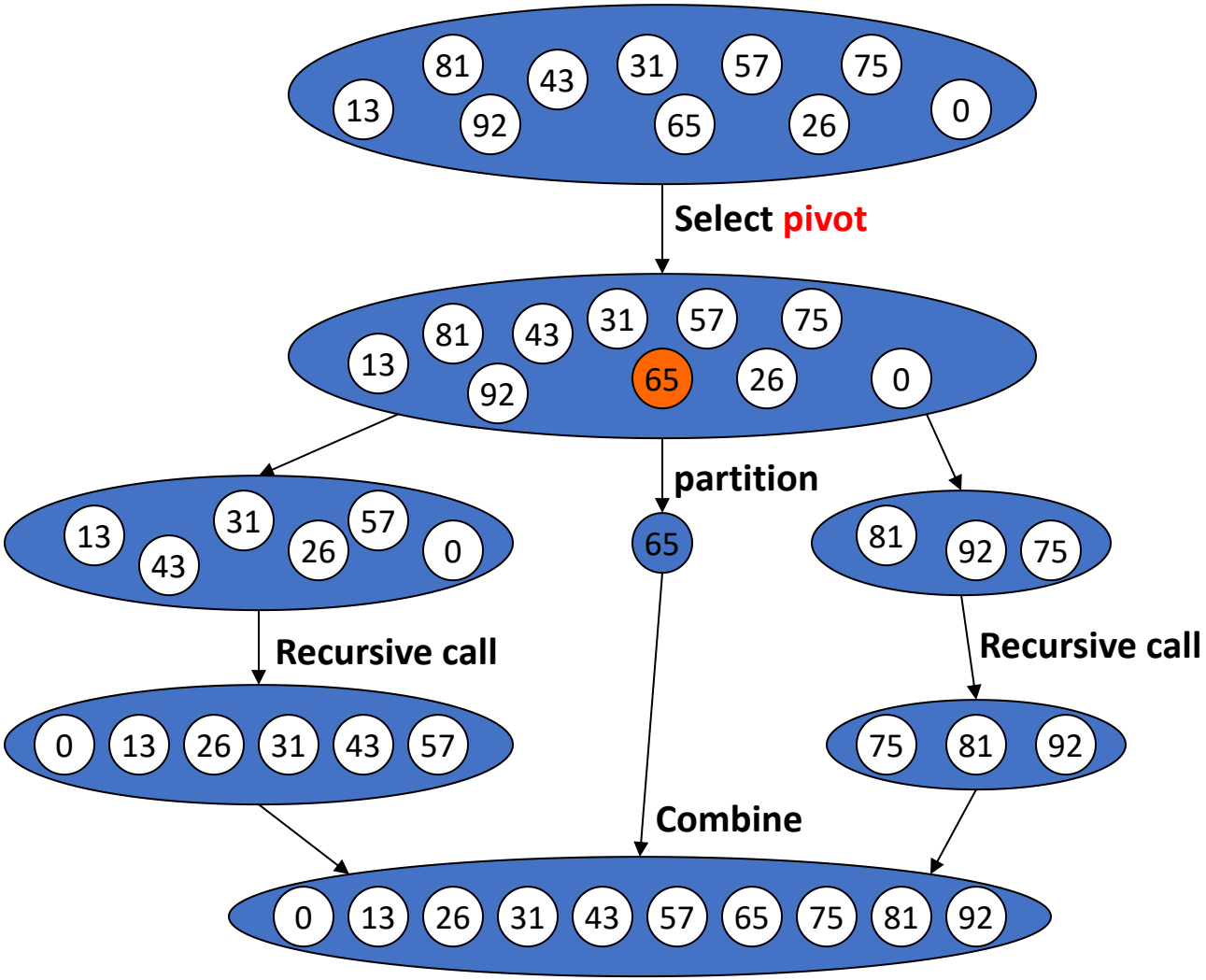
- Average case time complexity is O(n log n), with small constant factors
- Worst case time complexity is O(n^2), but, rarely happens, if coded correctly
- Sorts almost in place. That means, does not require any additional array as in MergeSort

Quick Sort: The concept

- Given an array A or list to be sorted,

1) Partition (**divide**) A in to two sub-arrays, such that, elements in the first part are less than or equal to the elements in the second part
2) Recursively QuickSort the two sub-arrays (**conquer**)
3) Combine the sorted sub-arrays . This is trivial because, in place sorting and use of **partition** method gives sorted sub-arrays

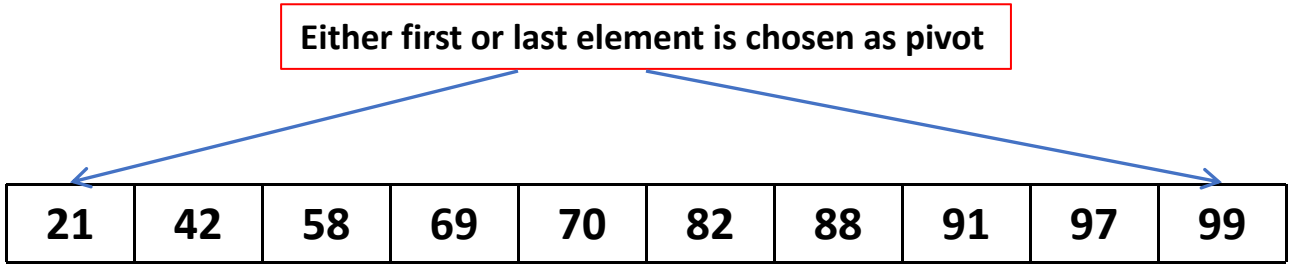
Quick Sort: Example



Quick Sort: Methods to Select Pivot Element

Strategy 1: Choose the first element (or last element)

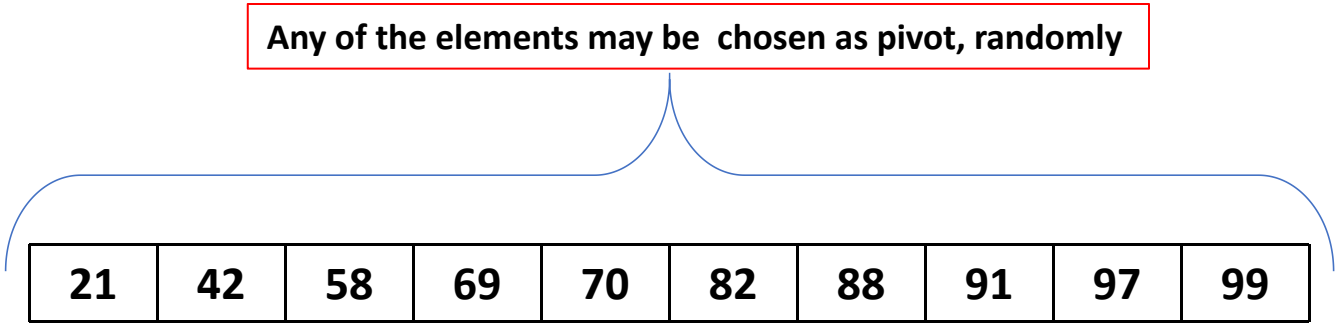
- Works well only if input is random
- If input array is sorted (either ascending or descending), every recursive call will create two sub-arrays such that, one is having zero elements and the other one with all the remaining elements.
- This gives worst performance (i.e., inefficient)



Quick Sort: Methods to Select Pivot Element

Strategy 2: Pick the pivot element randomly

- Usually works well, even for mostly sorted input
- However, random number generation is an expensive operation



Quick Sort: Methods to Select Pivot Element

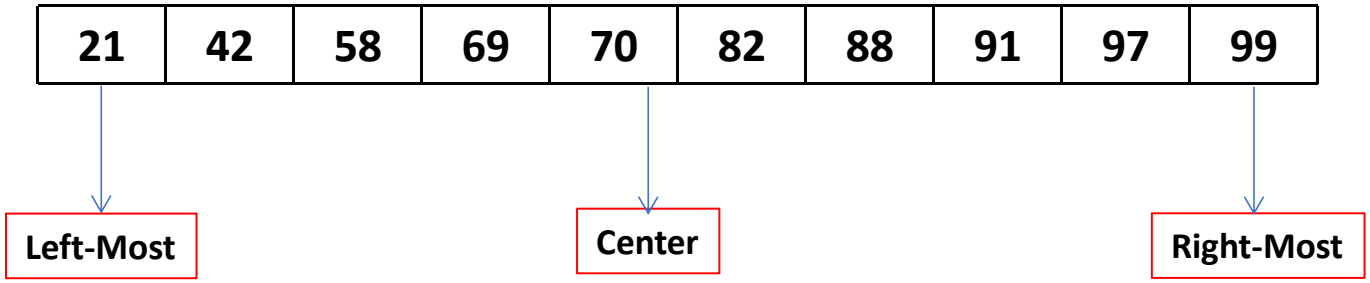
Strategy 3: **Median-of-three** Partitioning

- Choose the pivot as the median of the input array element
- **Median** = element in the middle of the sorted array
- The input is divided into two almost equal partitions
- However, its hard to calculate the median quickly, without sorting the given array first
- So we find the approximate median
- **Pivot** = median of the left-most, right-most and mid element of the given array
- Solves the problem of sorted input

Quick Sort: Methods to Select Pivot Element

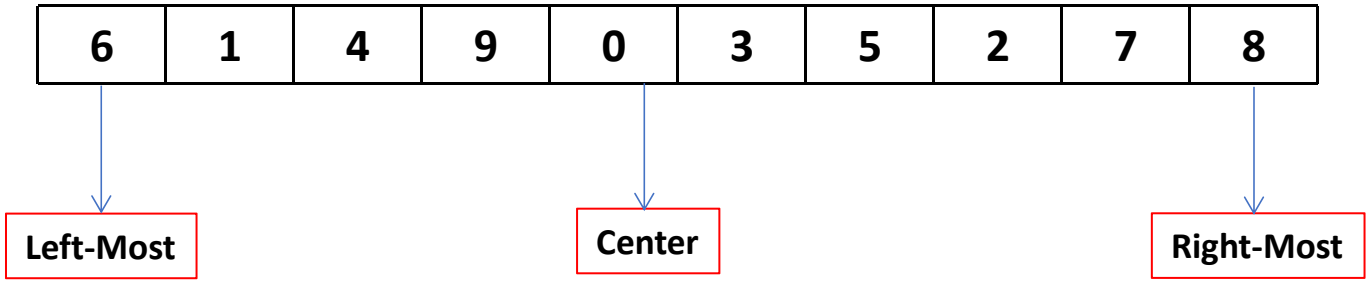
Strategy 3: **Median-of-three** Partitioning

Example1:



Median of 21, 70, and 99 is : 70, So Pivot = 5th element = **70**

Example2:



Median of 6, 0, and 8 is : 6, So Pivot = 1st element = **6**

Quick Sort: Algorithms

- With **pivot as last** element

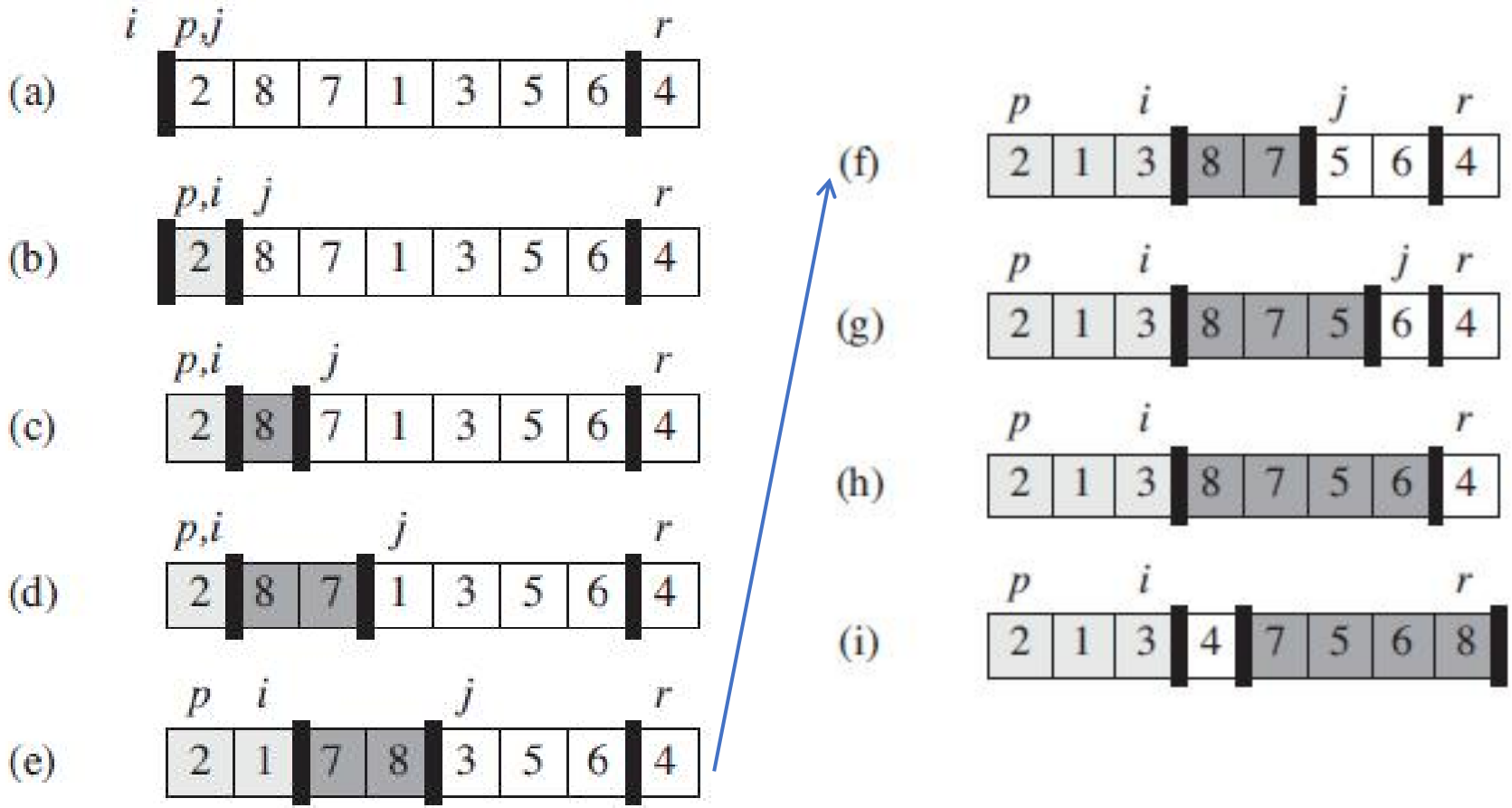
```
QuickSort (A, p, r)
1.    if (p < r )
2.          q = Partition (A, p, r)
3.          QuickSort (A, p, q-1)
4.          QuickSort (A, q+1, r)
```

```
Partition(A, p, r)
1.        x = A[r]
2.        i = p-1
3.        for j = p to r -1
4.            if ( A[ j ] ≤ x )
5.                i = i +1
6.                swap A[ i ] with A[ j ]
7.        swap A[i +1] with A[r ]
8.        return (i + 1)
```

Legend:
A is the array
p is the start index
r is the end index
q is the pivot index

Quick Sort: Algorithms

Partition - Example:



Quick Sort: Time Complexity

- Depends on partitioning
- If partitioning is **balanced**,QuickSort runs as fast as MergeSort, asymptotically. (i.e., $O(n \log n)$)
- If partitioning is **unbalanced** QuickSort runs asymptotically as slow as InsertionSort (i.e., $O(n^2)$)

Quick Sort: Time Complexity

- **Worst-Case Partitioning**:
  - At every stage, after the partitioning, one sub-array contains n -1 elements and the other one contains 0 elements.
  - This occurs when a sorted array is taken as input
  - The running time in this case can be given by the following recurrence

$$T(n) = \begin{cases} O(1) & \text{when } n \leq 1 \\ T(n-1) + O(n) & \text{when } n > 1 \end{cases}$$

Solving by substitution method, we get

$$\mathbf{T(n) = O(n^2)}$$

Quick Sort: Time Complexity

- **Best-Case Partitioning**:
  - Occurs when the sub-arrays are completely balanced every time
  - Each sub-array has $\leq n/2$ elements
  - The running time in this case is given by the recurrence

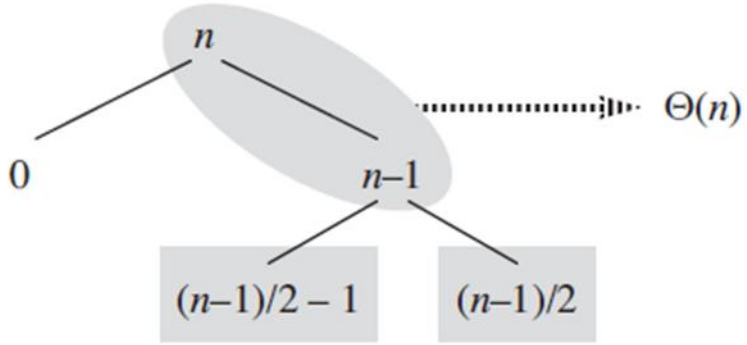
$$T(n) = \begin{cases} O(1) & \text{when } n \leq 1 \\ 2T(\frac{n}{2}) + O(n) & \text{when } n > 1 \end{cases}$$

Solving by substitution method, we get

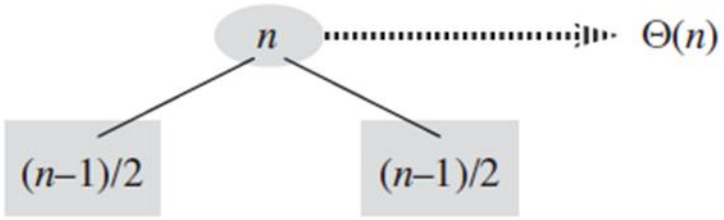
$$T(n) = O(n \log n)$$

Quick Sort: Time Complexity

- **Balanced Partitioning (Average case):**
  - Average case running time is much closer to the best case
  - Split in the recursion tree will not be always constant (i.e., Mix of good and bad splits at different levels)



(a)



(b)

Quick Sort: Time Complexity

- **Balanced Partitioning (Average case):**
  - The combination of the bad split followed by the good split (Figure (a) in previous slide) produces three sub-arrays of sizes 0, $(n-1)/2$ -1 and $(n-1)/2$
  - The combined partitioning cost of this mixed partitioning is $O(n) + O(n\text{-}1) = O(n)$
  - This time is almost same as that of balanced case (Figure (b) in previous slide) except by a slightly larger constant factor
  - Hence, running time of quicksort, when levels alternate between good and bad splits, is like the running time for good splits alone: $T(n) = O(n \log n)$

Exercise

- When the average running time of Merge sort and quick sort is the same asymptotically (i.e., O(*n*log*n*), why is Quick sort considered much faster?
- What are the in-built sorting algorithms used by Java and Python?

- Implement both merge sort and quick sort algorithms and measure the actual running time.

Questions and Discussion….

Thank you