COSC1285/2123: Algorithms & Analysis

Divide and Conquer

Jeffrey Chan

RMIT University
Email : jeffrey.chan@rmit.edu.au

Lecture 5

Overview

Levitin – The design and analysis of algorithms

This week we will be covering the material from Chapter 5.

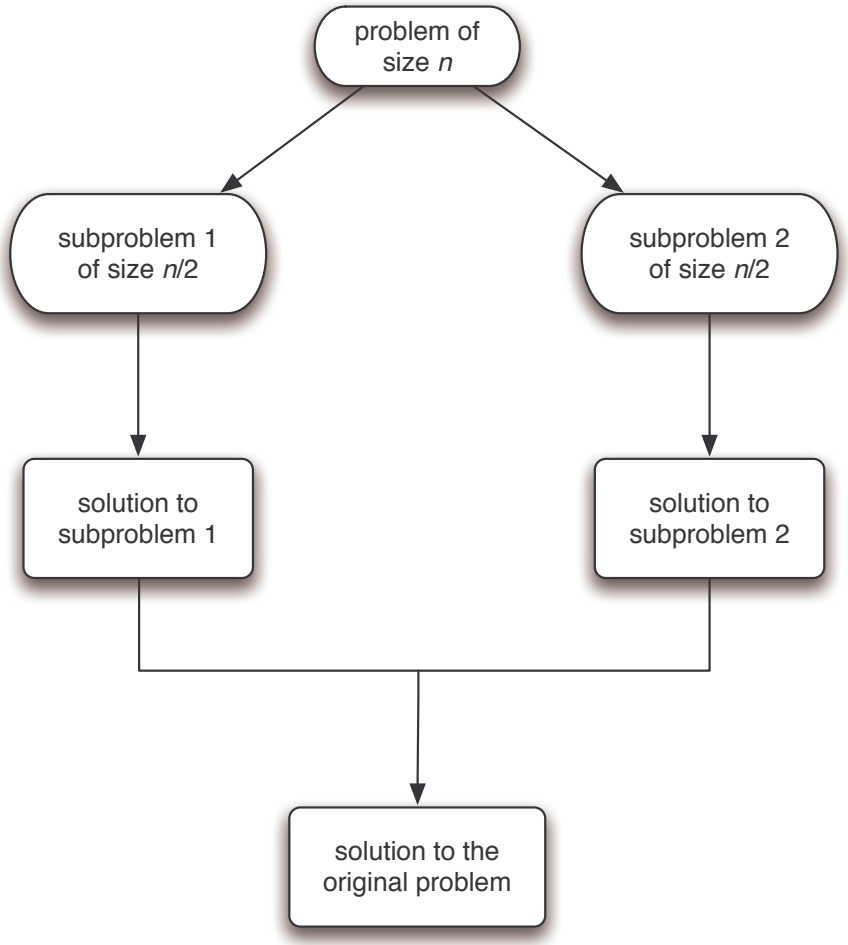
Learning outcomes:

- Understand the *Divide-and-conquer* algorithmic approach.
- Understand and apply merge and quick sort.
- Understand and apply height and traveral operations for BST.

Outline

Overview

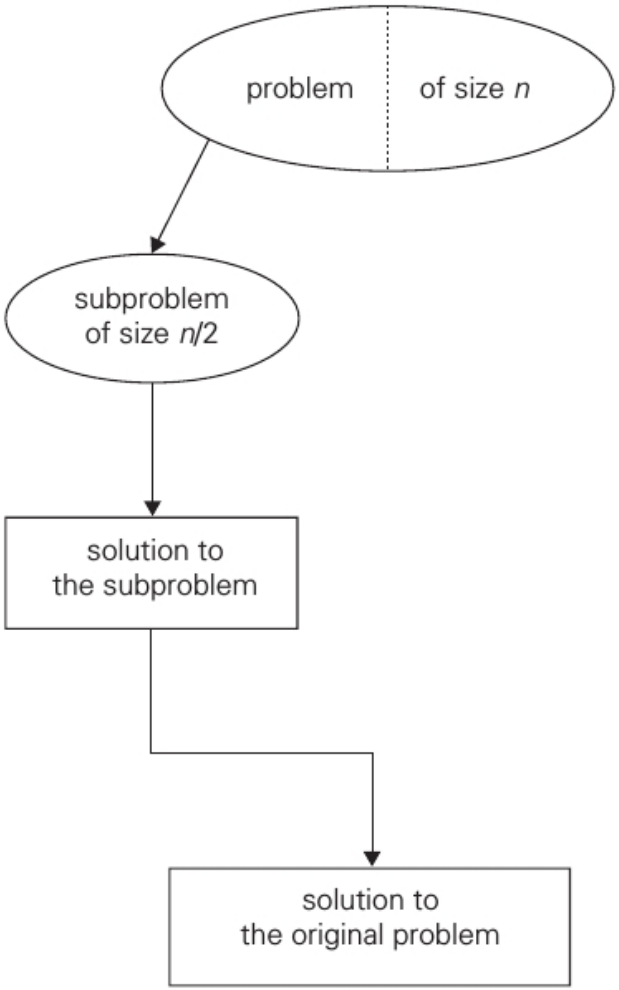
Divide and Conquer



Strategy:

1. Divide the problem instance into smaller subproblems.
2. Solve each subproblem (recursively).
3. Combine smaller solutions to solve the original instance.

Compare with Decrease-by-a-constant-factor



Strategy:

1. Decrease-by-a-constant-factor algorithms.

Overview

Merge Sort

- Idea:
  - Imagine we recursively divided an array (we wanted to sort) into halves, until we reach single element partitions.

Merge Sort

- Idea:
  - Imagine we recursively divided an array (we wanted to sort) into halves, until we reach single element partitions.
  - We then recursively merge the partitions, where we have a process that maintains sorting after partitions are merged.

Merge Sort

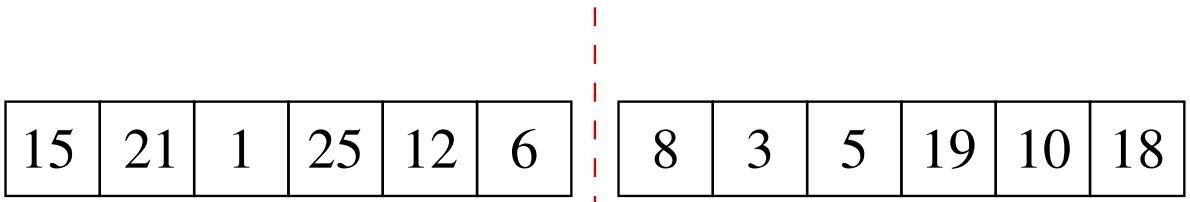
- Idea:
  - Imagine we recursively divided an array (we wanted to sort) into halves, until we reach single element partitions.
  - We then recursively merge the partitions, where we have a process that maintains sorting after partitions are merged.
  - When we finally merge the last two partitions, we have a sorted array.

Merge Sort Example

15	21	1	25	12	6	8	3	5	19	10	18

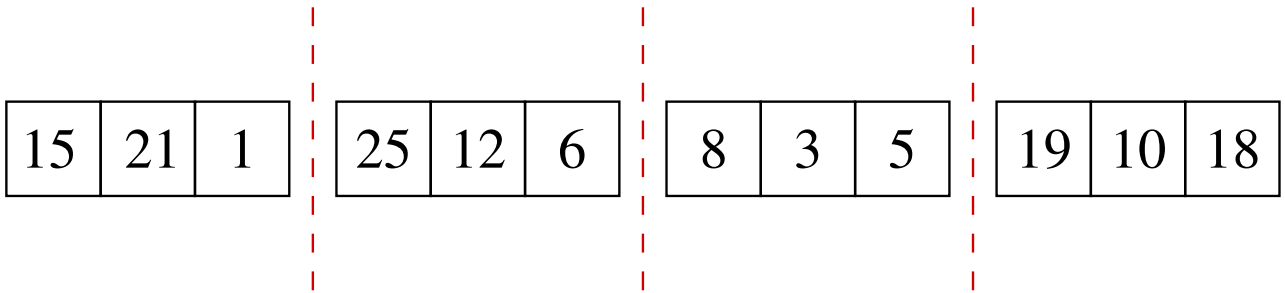
COMPARES | 0 |

Merge Sort Example

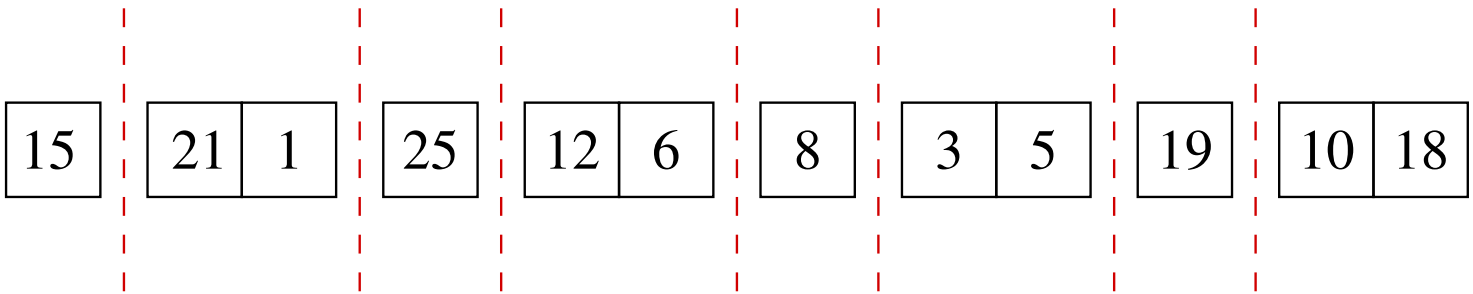


COMPARES 0

Merge Sort Example

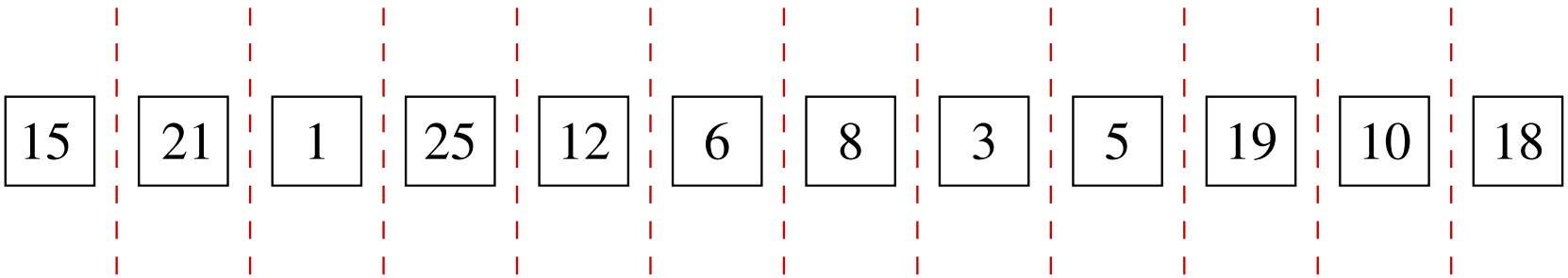


Merge Sort Example



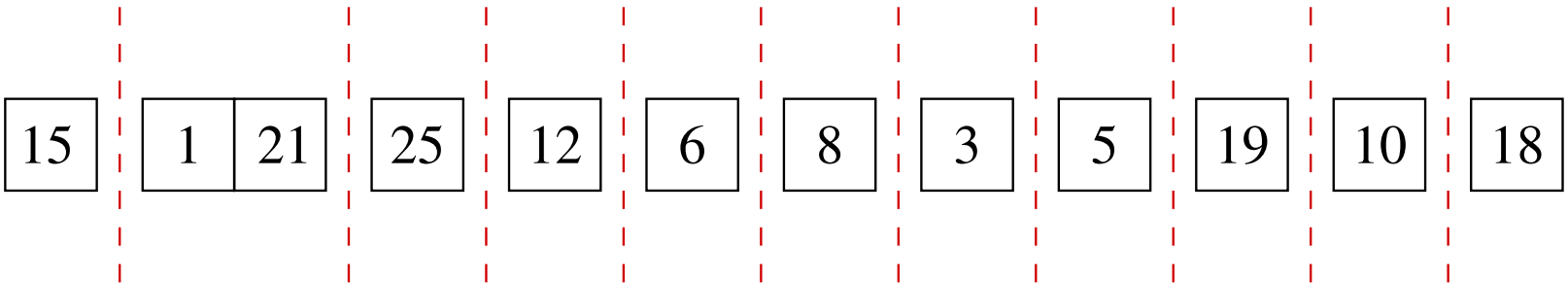
COMPARES 0

Merge Sort Example



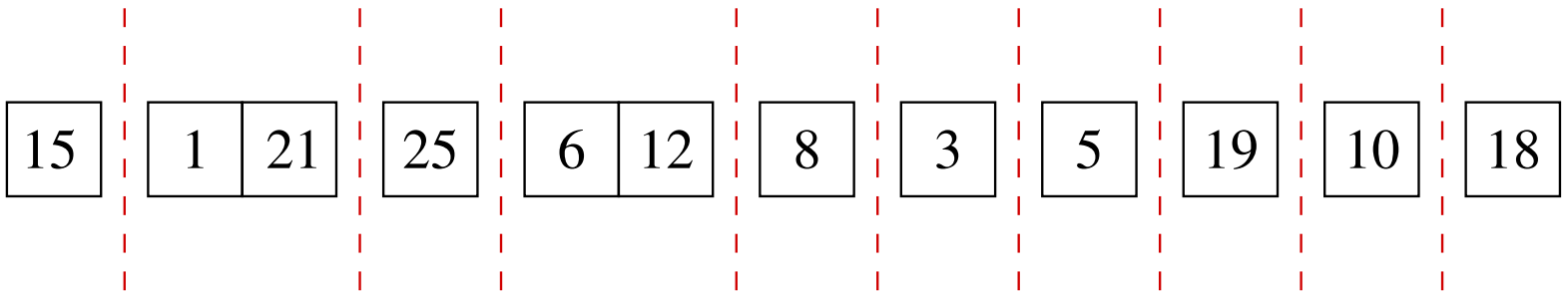
COMPARES 0

Merge Sort Example



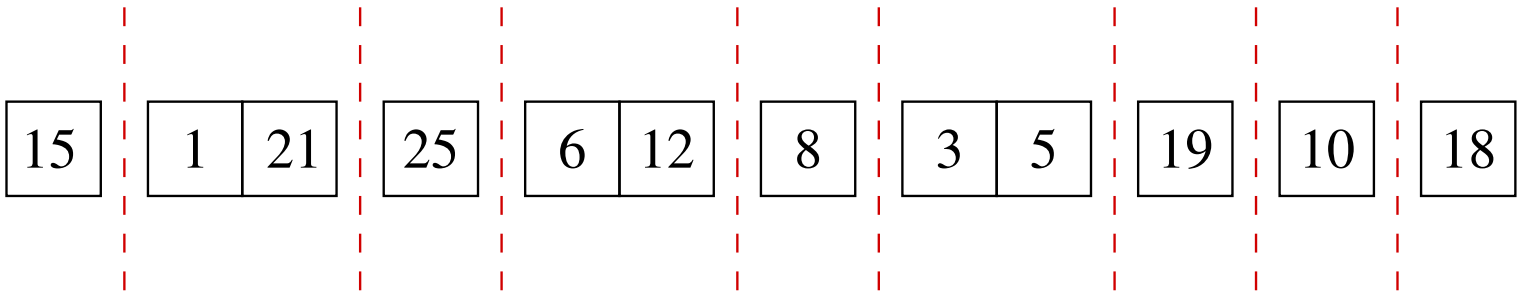
COMPARES 1

Merge Sort Example



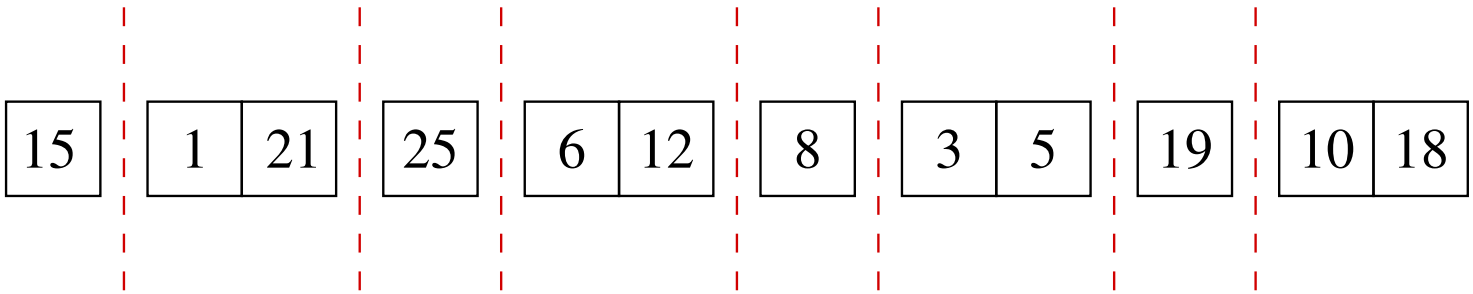
COMPARES 2

Merge Sort Example



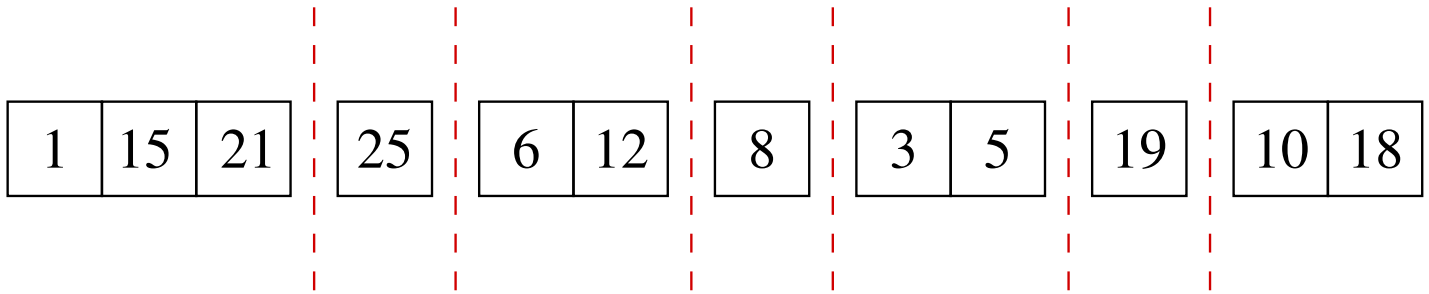
COMPARES 3

Merge Sort Example



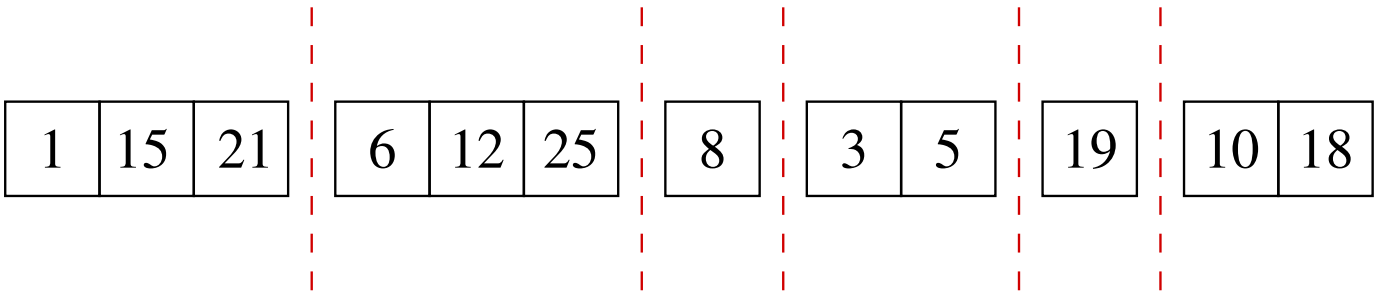
COMPARES 4

Merge Sort Example

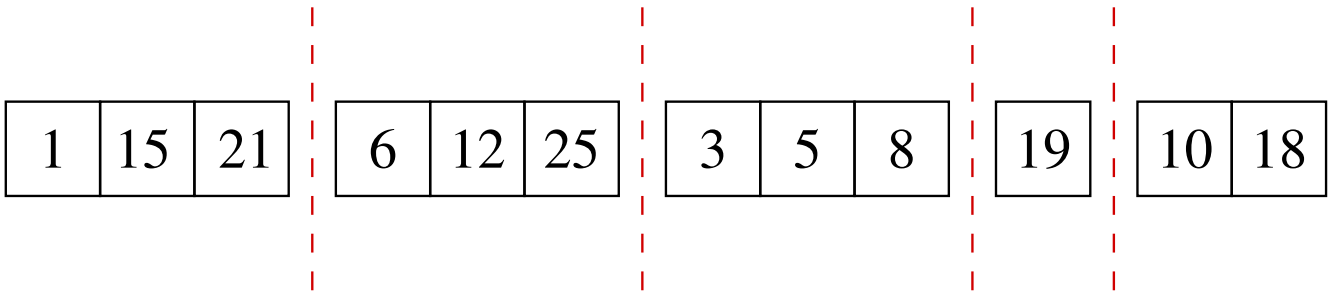


COMPARES 6

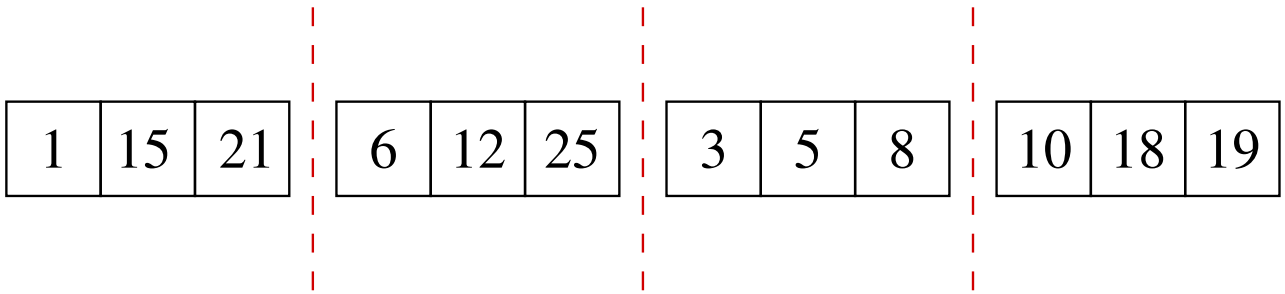
Merge Sort Example



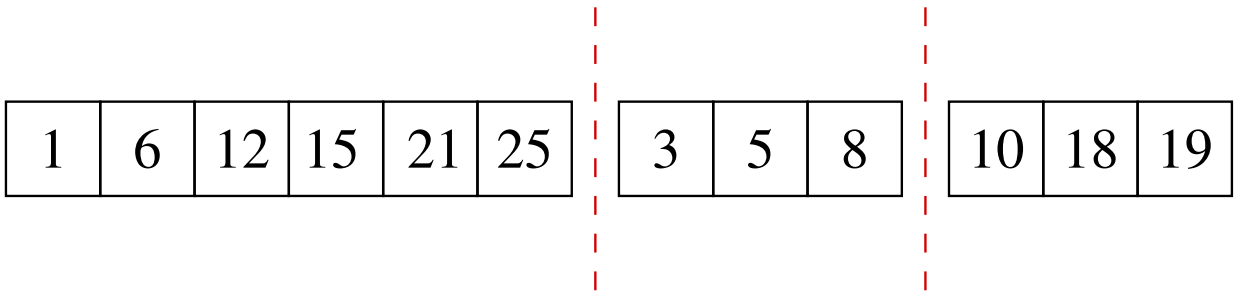
Merge Sort Example



Merge Sort Example

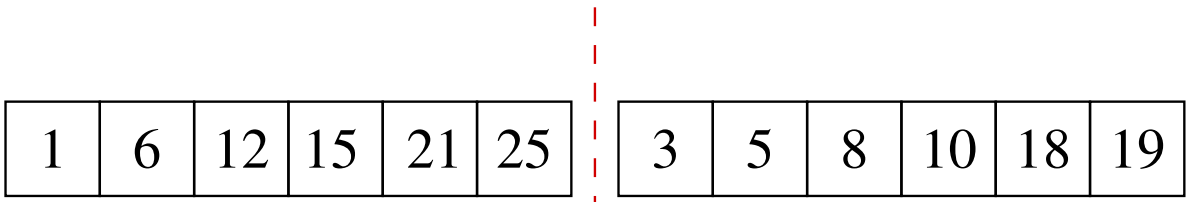


Merge Sort Example



COMPARES 17

Merge Sort Example



COMPARES 20

Merge Sort Example

1	3	5	6	8	10	12	15	18	19	21	25

COMPARES | 30 |

Merge Sort Algorithm

ALGORITHM **MergeSort** $(A[0 \ldots n-1])$
/* Sort an array using a divide-and-conquer merge sort. */
/* INPUT : An array $A[0 \ldots n-1]$ of orderable elements. */
/* OUTPUT : An array $A[0 \ldots n-1]$ sorted in ascending order. */

1: **if** $n > 1$ **then**
2: $B = A[0 \ldots \lfloor n/2 \rfloor - 1]$ */* B is first half of A */*
3: $C = A[\lfloor n/2 \rfloor \ldots n-1]$ */* C is second half of A */*
4: **MergeSort** (B)
5: **MergeSort** (C)
6: **Merge** (B, C, A) */* Merge B and C to help sort A */*
7: **end if**

Merge in Mergesort

Given two sorted subarrays B, C, want to merge them together to form a sorted A.

Merge in Mergesort

Given two sorted subarrays B, C, want to merge them together to form a sorted A.

Idea:

1. Consider first element of each subarray, i.e, B[0] and C[0]. Compare them. Whichever one is smaller, copy to A[0], and increment current pointer of subarrays that has smaller element and A.
2. Repeat until one of subarrays is empty. Then copy the rest of subarray to A.

Comments on Merge Sort

- Guarantees $\mathcal{O}(n \log n)$ time complexity, regardless of the original distribution of data – this sorting method is insensitive to the data distribution.

Comments on Merge Sort

- Guarantees $\mathcal{O}(n \log n)$ time complexity, regardless of the original distribution of data – this sorting method is insensitive to the data distribution.
- The main drawback in this method is the extra space required for merging two partitions/sub-arrays, e.g. B and C from pseudo-code.

Comments on Merge Sort

- Guarantees $\mathcal{O}(n \log n)$ time complexity, regardless of the original distribution of data – this sorting method is insensitive to the data distribution.
- The main drawback in this method is the extra space required for merging two partitions/sub-arrays, e.g. B and C from pseudo-code.
- Merge sort is a stable sorting method.

Merge Sort Analysis

Lets set up the recurrence relationship.

$$C(n) = 2C(n/2) + C_{merge}(n), \text{ for } n > 1, C(1) = 0.$$

$$C_{merge}(n) = n - 1$$

$$C(n) = 2C(n/2) + n - 1 \text{ for } n > 1, C(1) = 0.$$

Merge Sort Analysis

Lets set up the recurrence relationship.

$$C(n) = 2C(n/2) + C_{merge}(n), \text{ for } n > 1, C(1) = 0.$$

$$C_{merge}(n) = n - 1$$

$$C(n) = 2C(n/2) + n - 1 \text{ for } n > 1, C(1) = 0.$$

How do we solve this recursion? There are several ways:

- using backward substitution method.
- using recursion tree (please read textbook, not examinable).
- using the Master Theorem (please read textbook, not examinable).

Merge Sort Analysis

Lets set up the recurrence relationship.

$$C(n) = 2C(n/2) + C_{merge}(n), \text{ for } n > 1, C(1) = 0.$$

$$C_{merge}(n) = n - 1$$

$$C(n) = 2C(n/2) + n - 1 \text{ for } n > 1, C(1) = 0.$$

How do we solve this recursion? There are several ways:

- using backward substitution method.
- using recursion tree (please read textbook, not examinable).
- using the Master Theorem (please read textbook, not examinable).

$C(n) \in \mathcal{O}(n \log_2(n))$

Overview

Quick Sort

Motivation:

- Mergesort has consistent behaviour for all inputs – what if we seek an algorithm that is fast for the average case?
- Quicksort is such a sorting algorithm, often the best practical choice in terms of efficiency because of its good performance on the average case.
- Quick Sort is a *divide and conquer* algorithm.

Quick Sort

Idea:

1. Select an element from the array for which, we hope, about half the elements will come before and half after in a sorted array. Call this element the pivot.

Quick Sort

Idea:

1. Select an element from the array for which, we hope, about half the elements will come before and half after in a sorted array. Call this element the pivot.
2. Partition the array so that all elements with a value less than the pivot are in one subarray, and larger elements come in the other subarray.

Quick Sort

Idea:

1. Select an element from the array for which, we hope, about half the elements will come before and half after in a sorted array. Call this element the pivot.
2. Partition the array so that all elements with a value less than the pivot are in one subarray, and larger elements come in the other subarray.
3. Swap pivot into position of array that is between the partitions.

Quick Sort

Idea:

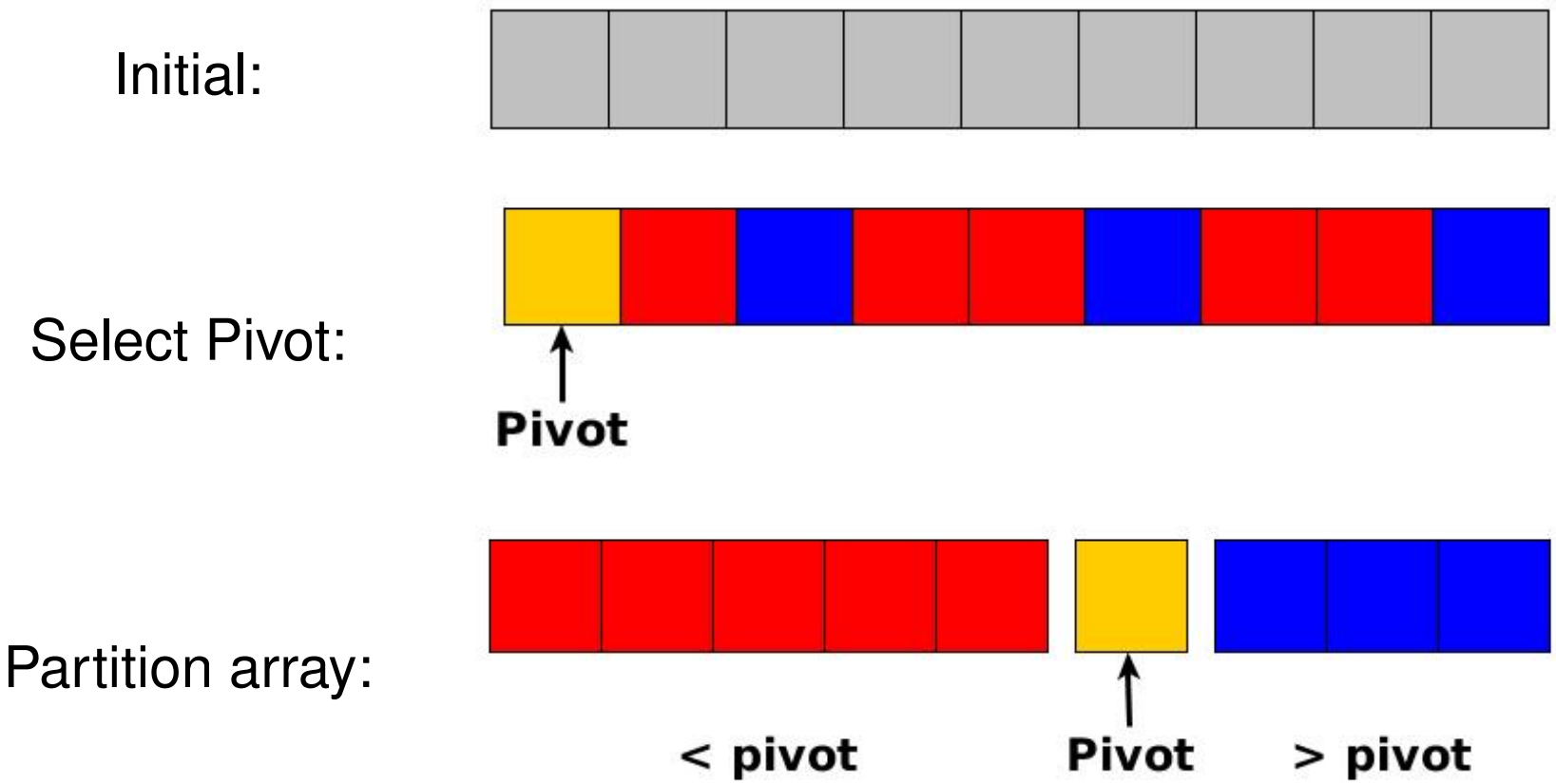
1. Select an element from the array for which, we hope, about half the elements will come before and half after in a sorted array. Call this element the pivot.
2. Partition the array so that all elements with a value less than the pivot are in one subarray, and larger elements come in the other subarray.
3. Swap pivot into position of array that is between the partitions.
4. Recursively apply the same procedure on the two subarrays separately.

Quick Sort

Idea:

1. Select an element from the array for which, we hope, about half the elements will come before and half after in a sorted array. Call this element the pivot.
2. Partition the array so that all elements with a value less than the pivot are in one subarray, and larger elements come in the other subarray.
3. Swap pivot into position of array that is between the partitions.
4. Recursively apply the same procedure on the two subarrays separately.
5. Terminate when only subarrays are of one element.
6. When terminate, because we do things in-place, the resulting array is sorted.

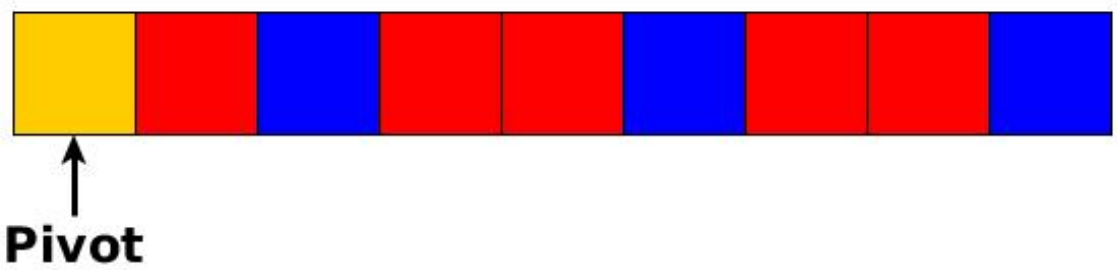
Quick Sort


Initial:
Select Pivot:
Pivot
Partition array:
< pivot
Pivot
> pivot

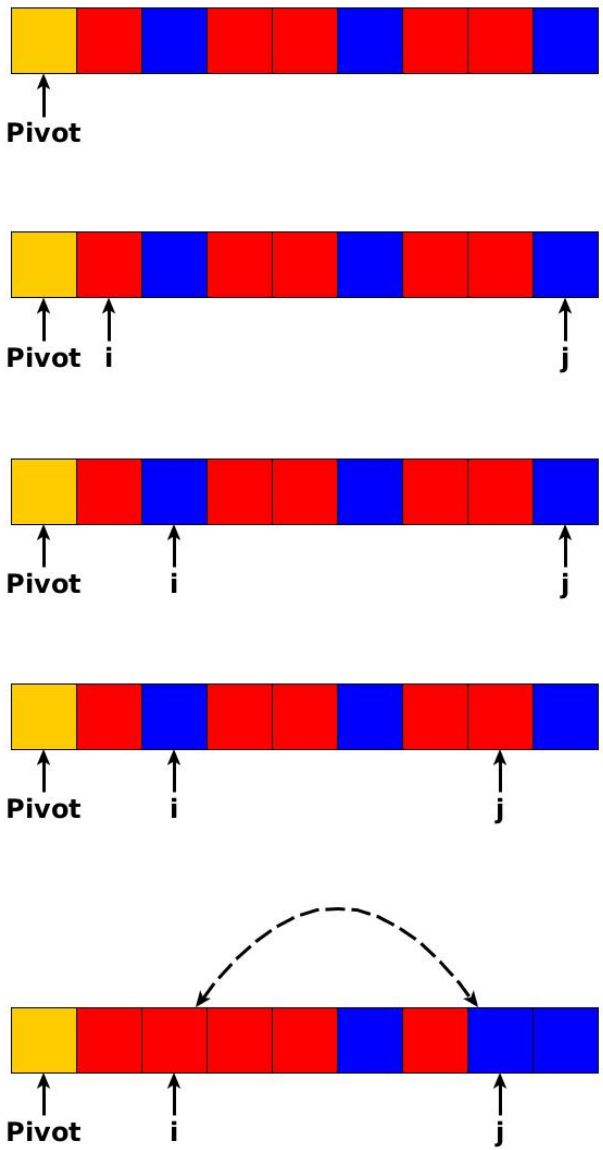
Quick Sort (Hoare partition scheme)

How to efficiently partition the elements less than pivot on one side, greater than partition on the other?

We basically run two linear scans from the front and back of the array, and swap elements that are out of place (i.e., will be on the wrong side of the eventual place of pivot)



Quick Sort (Hoare partition scheme)



Quick Sort

ALGORITHM **QuickSort** $(A[\ell \ldots r])$
/* Sort a subarray using by quicksort. */
/* INPUT : A subarray $A[\ell \ldots r]$ of $A[0 \ldots n-1]$, defined by its left and right indices ℓ and r. */
/* OUTPUT : A subarray $A[\ell \ldots r]$ sorted in ascending order. */

```
1: if ℓ < r then
2:     /* s is the index to split array. */
3:     s = QPartition(A[ℓ ... r])
4:     QuickSort(A[ℓ ... s − 1])
5:     QuickSort(A[s + 1 ... r])
6: end if
```

Quick Sort

ALGORITHM **QPartition** ($A[\ell \ldots r]$)
/* Partition a subarray using the first element as a pivot. */
/* INPUT : A subarray $A[\ell \ldots r]$ of $A[0 \ldots n-1]$, defined by its left and right indices ℓ and r, where $(\ell < r)$. */
/* OUTPUT : A partition of $A[\ell \ldots r]$, along with the index of the split position. */

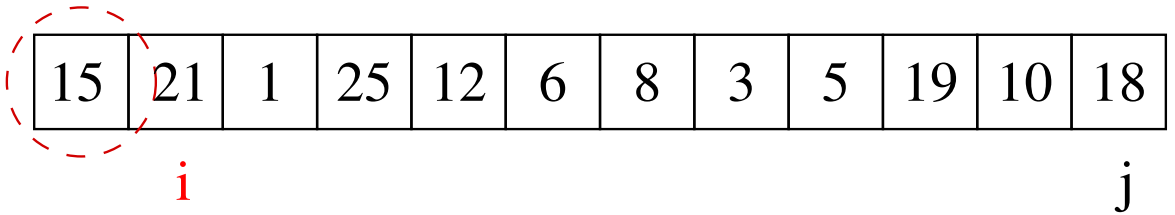
```
1:  p = A[ℓ]
2:  i = ℓ; j = r + 1
3:  repeat
4:      do
5:          i = i + 1
6:      while A[i] < p and i < r
7:      do
8:          j = j − 1
9:      while A[j] > p
10:      swap(A[i], A[j])
11: until i ≥ j
12: swap(A[i], A[j]) // we need to reverse the last swap, which is incorrect when i and j could
    cross each other
13: swap(A[ℓ], A[j])
14: return j
```

Quick Sort Example

15	21	1	25	12	6	8	3	5	19	10	18

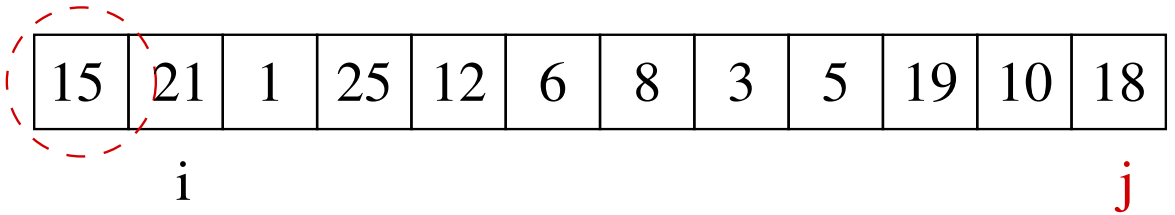
COMPARES 0 SWAPS 0

Quick Sort Example



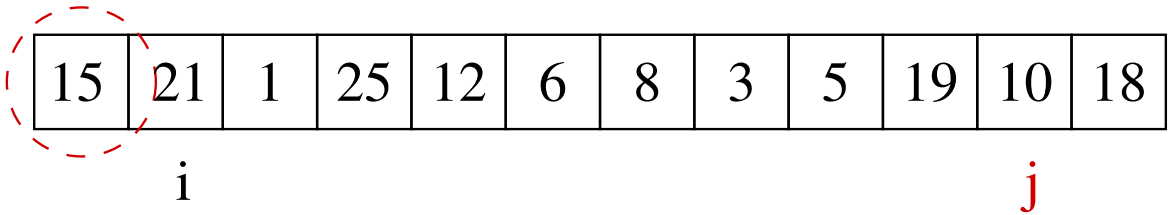
COMPARES 1 SWAPS 0

Quick Sort Example



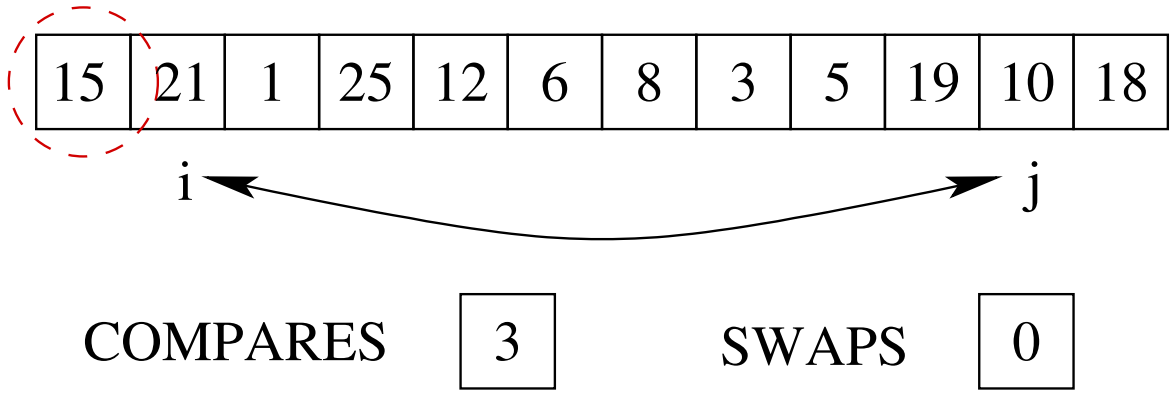
COMPARES 2 SWAPS 0

Quick Sort Example

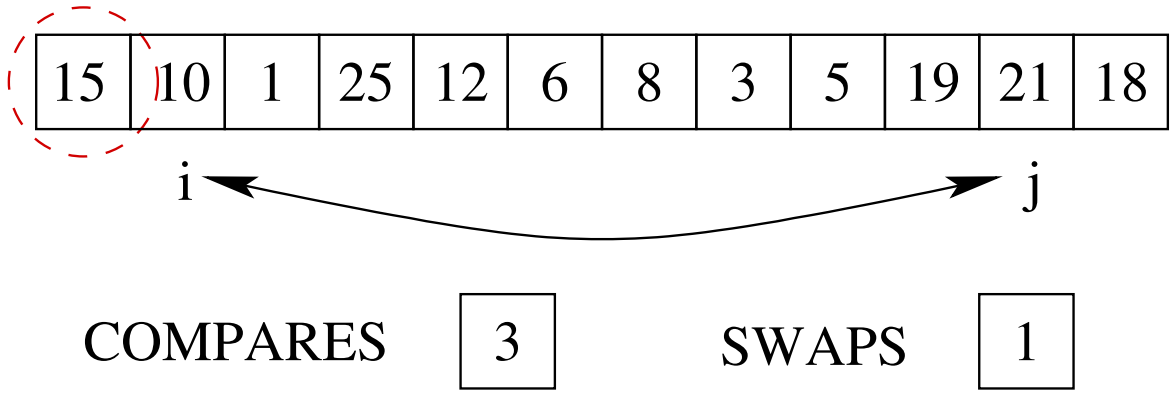


COMPARES 3 SWAPS 0

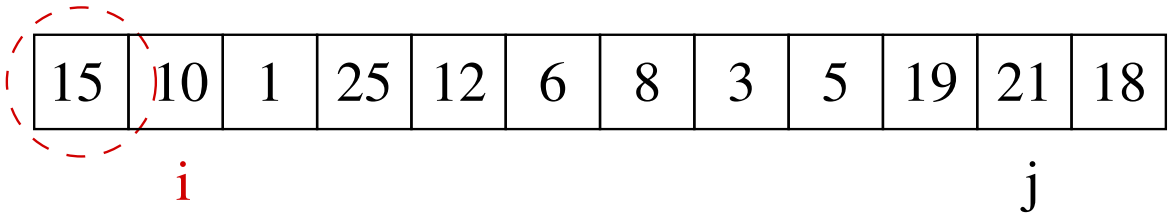
Quick Sort Example



Quick Sort Example

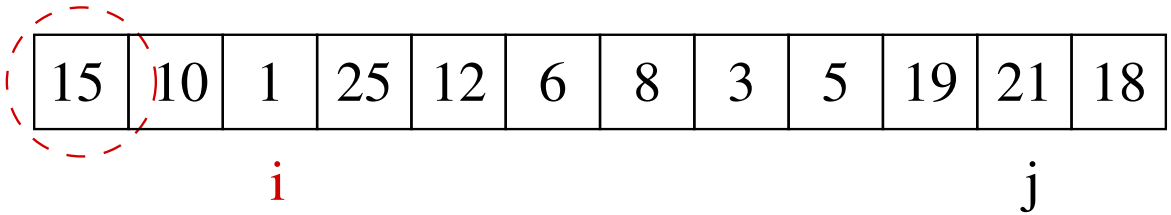


Quick Sort Example



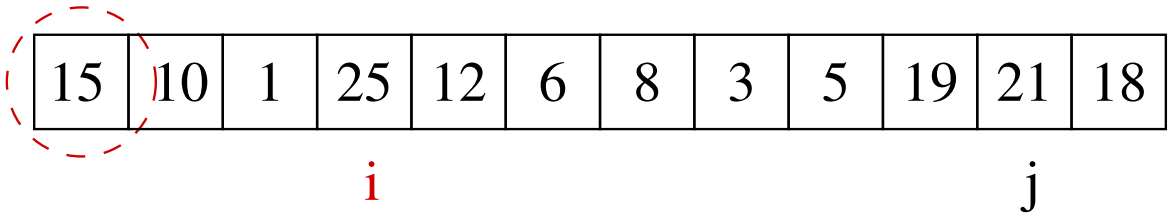
COMPARES 4 SWAPS 1

Quick Sort Example



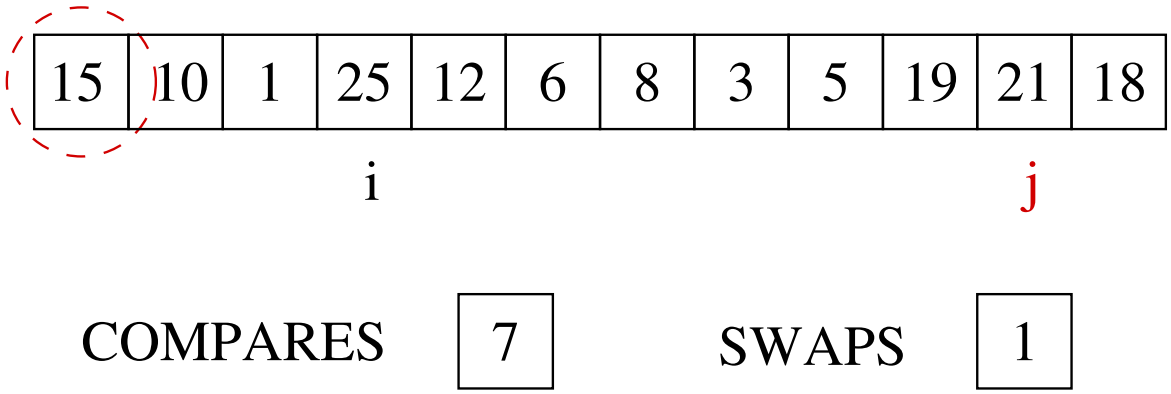
COMPARES 5 SWAPS 1

Quick Sort Example

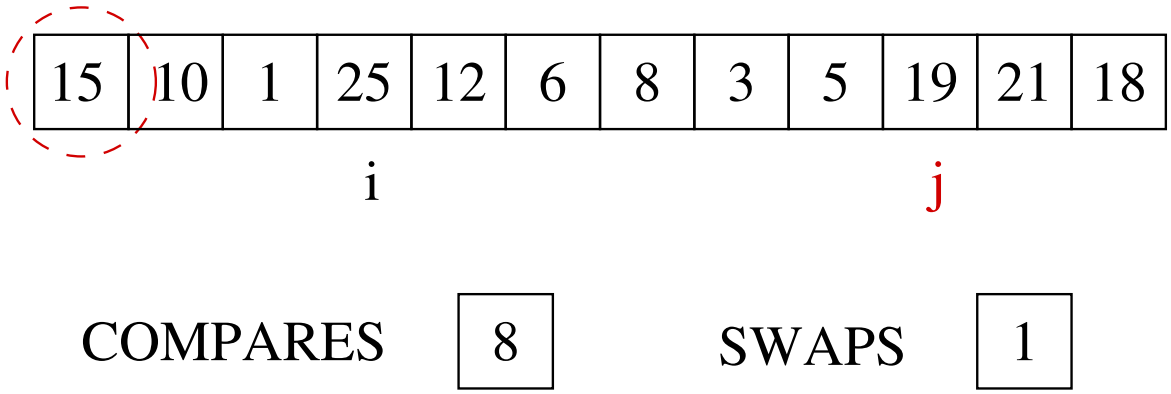


COMPARES 6 SWAPS 1

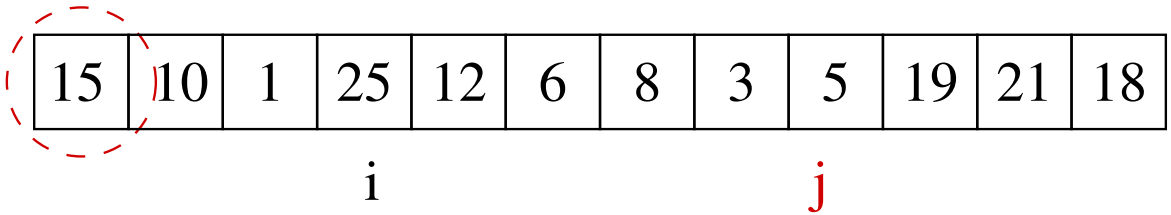
Quick Sort Example



Quick Sort Example

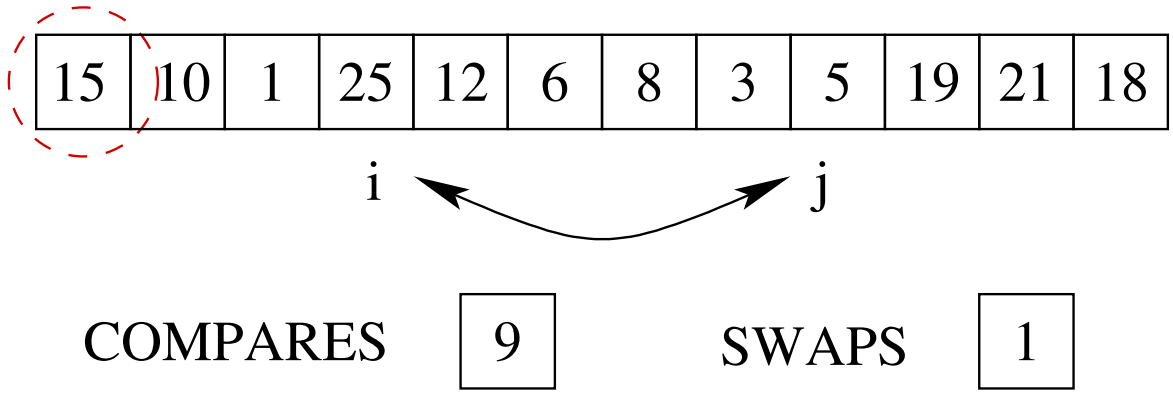


Quick Sort Example

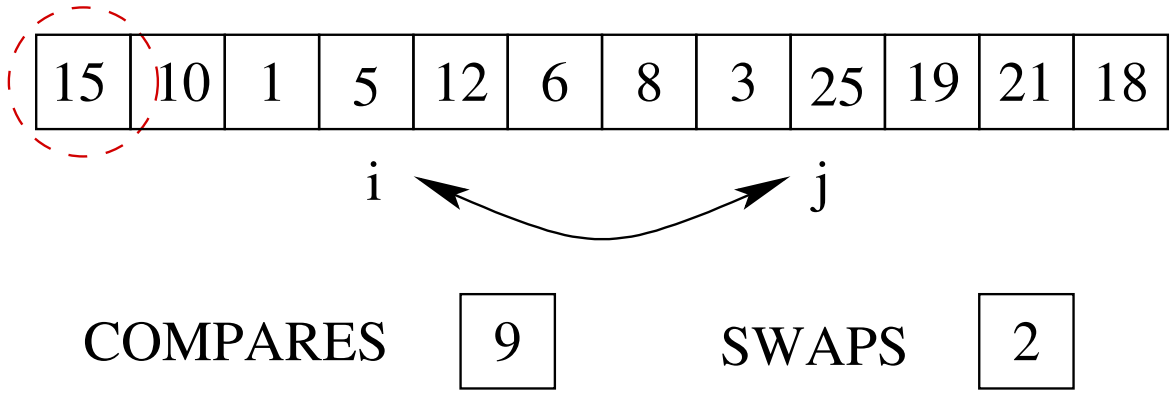


COMPARES 9 SWAPS 1

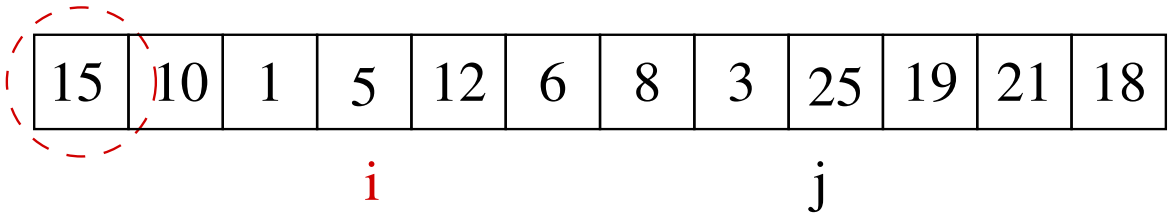
Quick Sort Example



Quick Sort Example

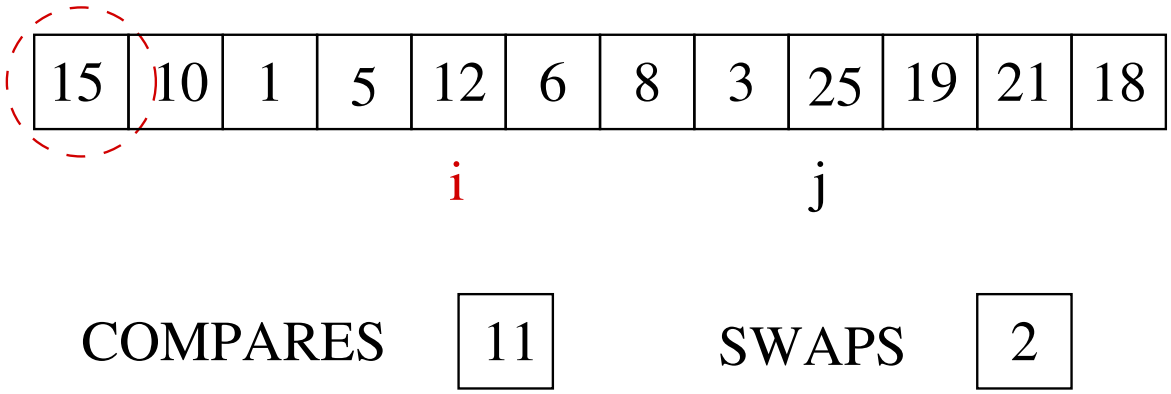


Quick Sort Example

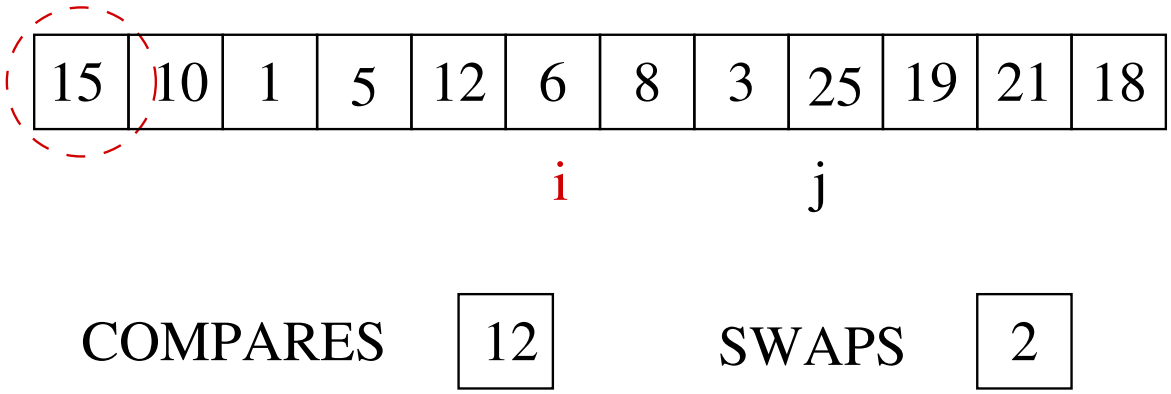


COMPARES 10 SWAPS 2

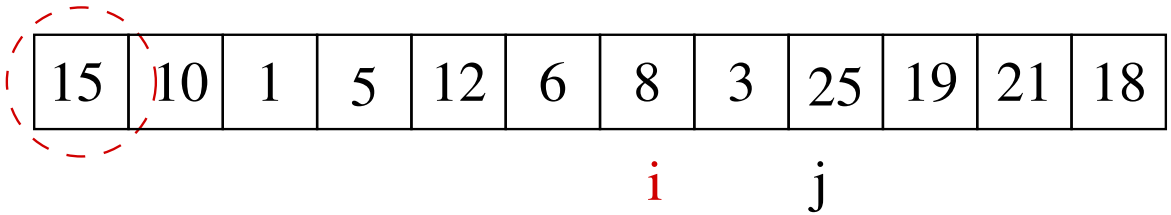
Quick Sort Example



Quick Sort Example

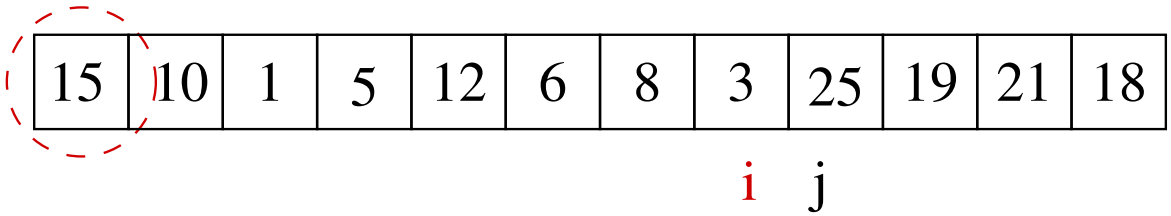


Quick Sort Example



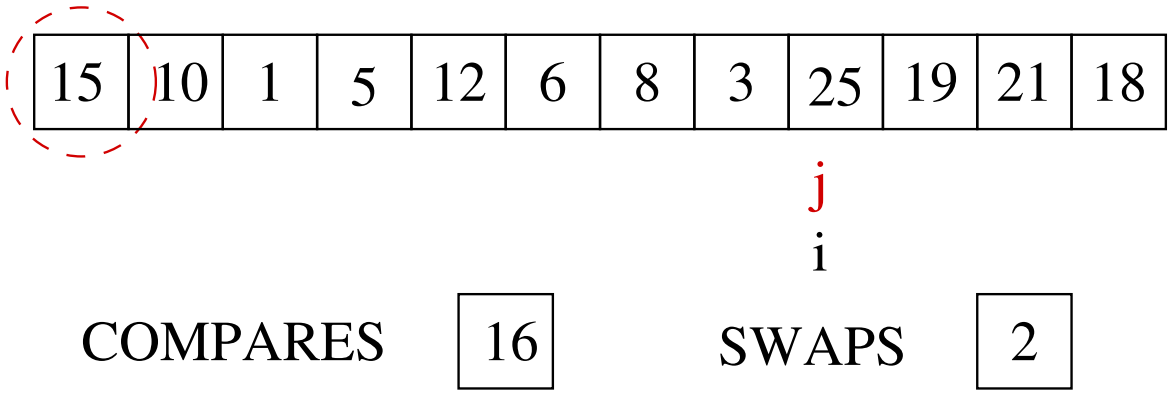
COMPARES 13 SWAPS 2

Quick Sort Example

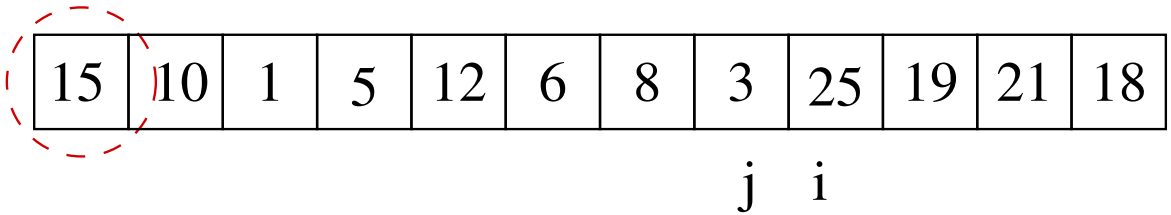


COMPARES 14 SWAPS 2

Quick Sort Example

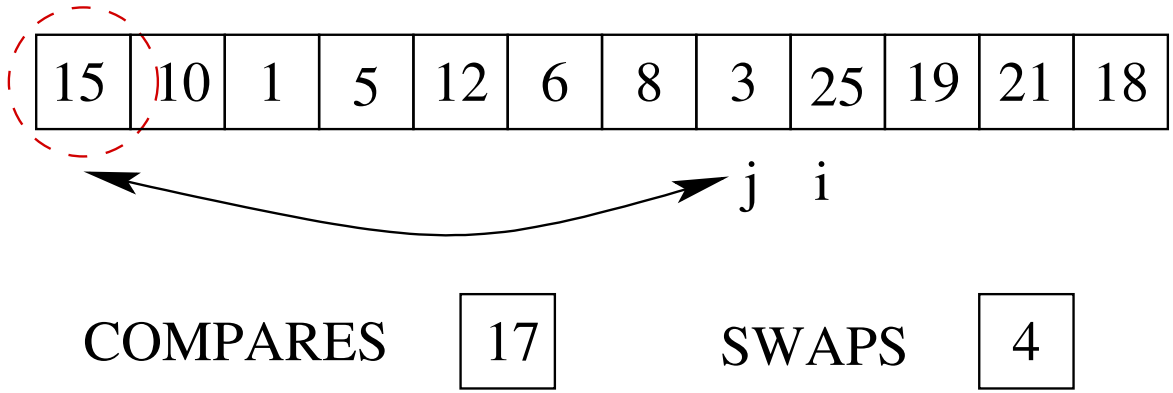


Quick Sort Example

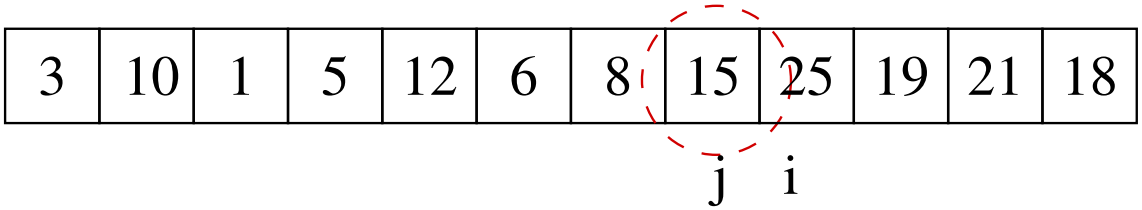


COMPARES 17 SWAPS 2

Quick Sort Example

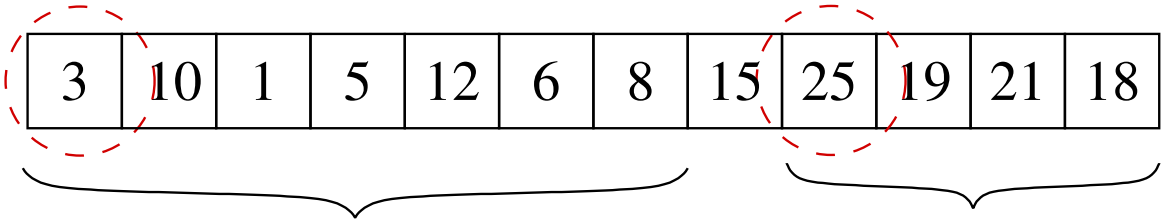


Quick Sort Example



COMPARES 17 SWAPS 5

Quick Sort Example



COMPARES 17 SWAPS 5

Quick Sort Complexity

1. Best Case:
   - Occurs when the pivot repeatedly splits the dataset into two equal sized subsets.
   - The complexity is $\mathcal{O}(n \log_2 n)$.

Quick Sort Complexity

1. Best Case:
   - Occurs when the pivot repeatedly splits the dataset into two equal sized subsets.
   - The complexity is $\mathcal{O}(n \log_2 n)$.
2. Worst Case:
   - If the pivot is chosen poorly, one of the partitions may be empty, and the other reduced by only one element.
   - Then the quick sort is slower than brute-force sorting (due to partitioning overheads).
   - The complexity is $n + (n-1) + (n-2) + \ldots + 1 \approx n^2/2 \in \mathcal{O}(n^2)$.
   - Occurs when array is already sorted or reverse order sorted.

Quick Sort Complexity

1. Best Case:
   - Occurs when the pivot repeatedly splits the dataset into two equal sized subsets.
   - The complexity is $\mathcal{O}(n \log_2 n)$.
2. Worst Case:
   - If the pivot is chosen poorly, one of the partitions may be empty, and the other reduced by only one element.
   - Then the quick sort is slower than brute-force sorting (due to partitioning overheads).
   - The complexity is $n + (n-1) + (n-2) + \ldots + 1 \approx n^2/2 \in \mathcal{O}(n^2)$.
   - Occurs when array is already sorted or reverse order sorted.
3. Average case:
   - Number of comparisons is $\approx 1.39 n \log n$.
   - 39% more comparisons than merge sort.
   - But faster than merge sort in practice because of lower cost of other high-frequency operations, and uses considerably less space (no need to copy to temporary arrays).

Quick Sort – Pivots

Choosing a pivot:

- First or last element: worst case appears for already sorted or reverse sorted arrays (as we saw last slide).

Quick Sort – Pivots

Choosing a pivot:

- First or last element: worst case appears for already sorted or reverse sorted arrays (as we saw last slide).
- Median of three: requires extra compares but generally avoids worst case.

Quick Sort – Pivots

Choosing a pivot:

- First or last element: worst case appears for already sorted or reverse sorted arrays (as we saw last slide).
- Median of three: requires extra compares but generally avoids worst case.
- Random element: Poor cases are very unlikely, but efficient implementations can be non-trivial.

Quick Sort – Pivots

Choosing a pivot:

- First or last element: worst case appears for already sorted or reverse sorted arrays (as we saw last slide).
- Median of three: requires extra compares but generally avoids worst case.
- Random element: Poor cases are very unlikely, but efficient implementations can be non-trivial.
- As long as selected pivot is not always the worst case, Quick sort on average performs well.

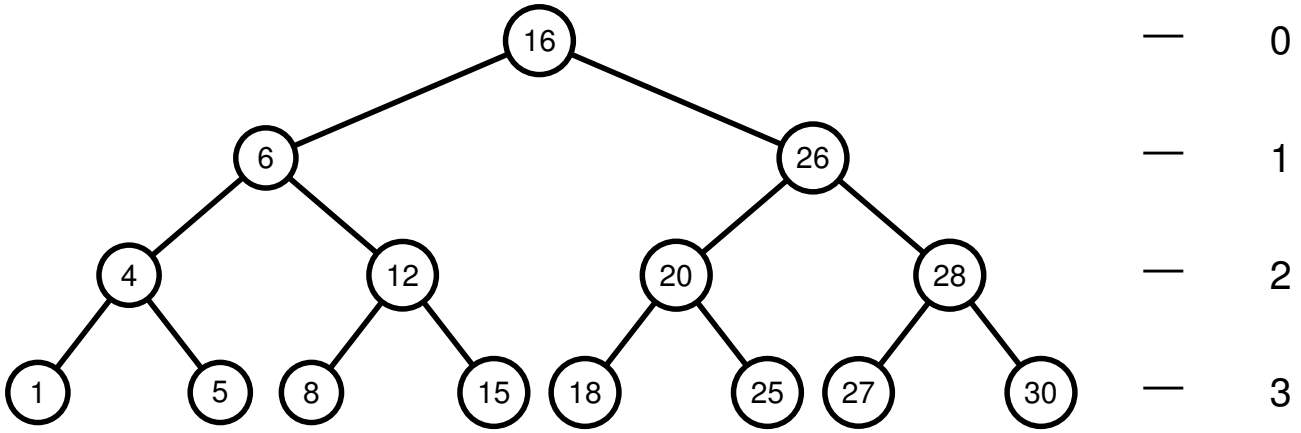
For the curious: Watch the Google Talk: 'Three Beautiful Quicksorts' by Jon Bentley (after lectures)

Quick Sort Properties

- Quick sort does sorting in place (don't need additional copying and temporary arrays).
- Quick Sort is *not* a stable sorting method.

Overview

BST Height



The height of a tree is the length of the path from the root to the deepest node in the tree. A rooted tree with only 1 node has a height of zero.

BST Height

ALGORITHM **BSTHeight** (T)
// Recursively compute the height of a binary search tree.
// INPUT : A binary tree T.
// OUTPUT : Height of T.

1: **if** $T = \emptyset$ **then**
2: **return** -1
3: **else**
4: **return** max (**BSTHeight** (T_L), **BSTHeight** (T_R)) $+1$
5: **end if**

BST Height

ALGORITHM **BSTHeight** (T)
// Recursively compute the height of a binary search tree.
// INPUT : A binary tree T.
// OUTPUT : Height of T.

```
1: if T = ∅ then
2:     return −1
3: else
4:     return max (BSTHeight (T_L), BSTHeight (T_R)) +1
5: end if
```

The worst case complexity to calculate the height is $\Theta(n)$. Why?

BST Height

ALGORITHM **BSTHeight** (T)
// Recursively compute the height of a binary search tree.
// INPUT : A binary tree T.
// OUTPUT : Height of T.

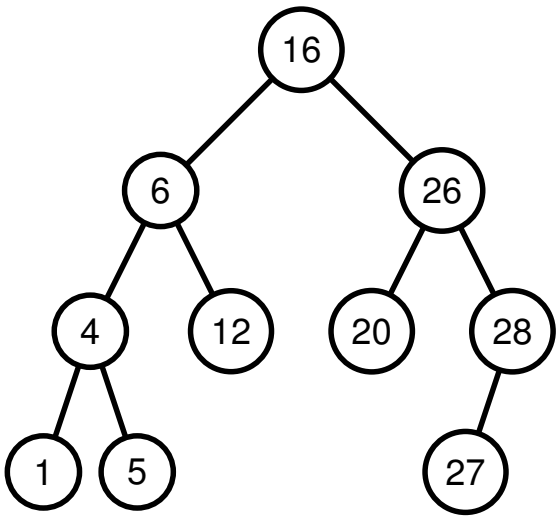
```
1: if T = ∅ then
2:     return −1
3: else
4:     return max (BSTHeight (T_L), BSTHeight (T_R)) +1
5: end if
```

The worst case complexity to calculate the height is $\Theta(n)$. Why?
What is the best case?

BST Traversal

- Given a tree, systematically process every node in that tree.
- For binary trees, we have up to two children for each node, and therefore we have three basic orders in which we can process the tree.
- preorder: The root is visited before the left and right subtrees are visited (in that order).
- inorder: The root is visited after visiting its left subtree but before visiting the right subtree.
- postorder: The root is visited after visiting the left and right subtrees (in that order).
- All of these methods are $\Theta(n)$.

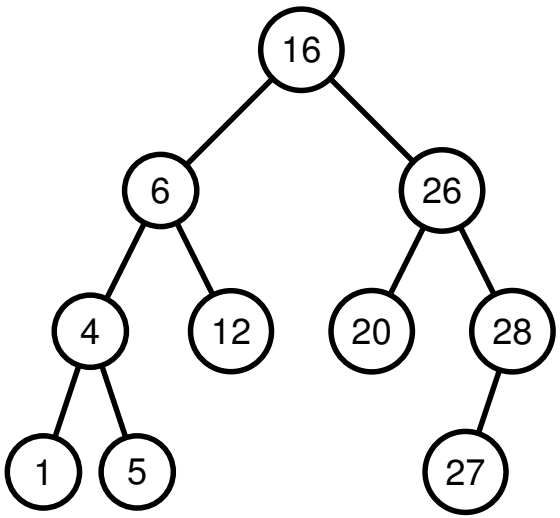
Inorder BST Traversal



ALGORITHM **Inorder**(T)
// An In-order traversal of a binary tree.
// INPUT : A binary tree T (with labeled vertices).
// OUTPUT : Node labels listed in-order.

1: **if** $T \neq \emptyset$ **then**
2: **Inorder**(T_L) // T_L is the left sub-tree.
3: **process** *node label*
4: **Inorder**(T_R) // T_R is the right sub-tree.
5: **end if**

Inorder BST Traversal

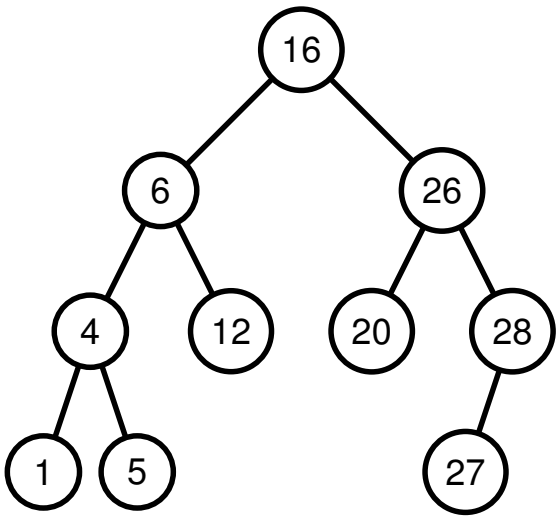


ALGORITHM **Inorder**(T)
// An In-order traversal of a binary tree.
// INPUT : A binary tree T (with labeled vertices).
// OUTPUT : Node labels listed in-order.

```
1: if T ≠ ∅ then
2:     Inorder(T_L) // T_L is the left sub-tree.
3:     process node label
4:     Inorder(T_R) // T_R is the right sub-tree.
5: end if
```

OUTPUT : $1, 4, 5, 6, 12, 16, 20, 26, 27, 28$

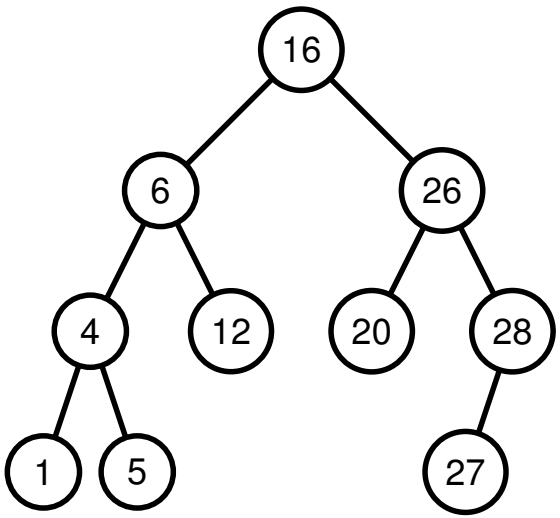
Pre-order BST Traversal



ALGORITHM **Preorder**(T)
// A Pre-order traversal of a binary tree.
// INPUT : A binary tree T (with labeled vertices).
// OUTPUT : Node labels listed in preorder.

1: **if** $T \neq \emptyset$ **then**
2: **process** *node label*
3: **Preorder**(T_L) // T_L is the left sub-tree.
4: **Preorder**(T_R) // T_R is the right sub-tree.
5: **end if**

Pre-order BST Traversal

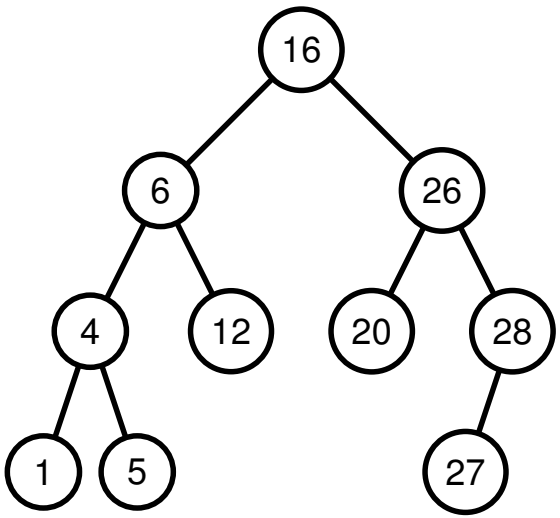


ALGORITHM **Preorder**(T)
// A Pre-order traversal of a binary tree.
// INPUT : A binary tree T (with labeled vertices).
// OUTPUT : Node labels listed in preorder.

1: **if** $T \neq \emptyset$ **then**
2: **process** *node label*
3: **Preorder**(T_L) // T_L is the left sub-tree.
4: **Preorder**(T_R) // T_R is the right sub-tree.
5: **end if**

OUTPUT : $16, 6, 4, 1, 5, 12, 26, 20, 28, 27$

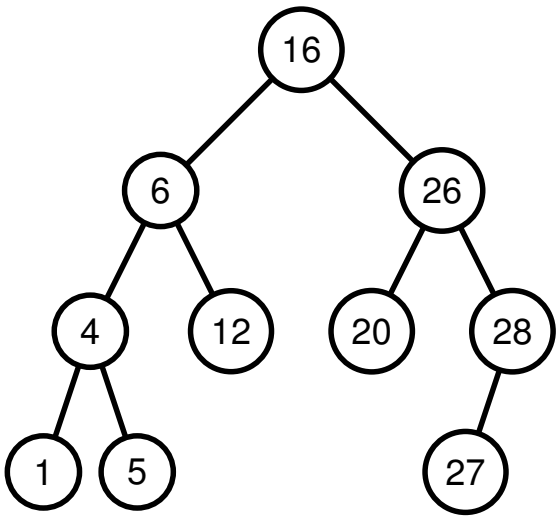
Postorder BST Traversal



ALGORITHM **Postorder**(T)
// A Post-order traversal of a binary tree.
// INPUT : A binary tree T (with labeled vertices).
// OUTPUT : Node labels listed in postorder.

1: **if** $T \neq \emptyset$ **then**
2: **Postorder**(T_L) // T_L is the left sub-tree.
3: **Postorder**(T_R) // T_R is the right sub-tree.
4: **process** *node label*
5: **end if**

Postorder BST Traversal



ALGORITHM **Postorder**(T)
// A Post-order traversal of a binary tree.
// INPUT : A binary tree T (with labeled vertices).
// OUTPUT : Node labels listed in postorder.

```
1: if T ≠ ∅ then
2:     Postorder(T_L) // T_L is the left sub-tree.
3:     Postorder(T_R) // T_R is the right sub-tree.
4:     process node label
5: end if
```

OUTPUT : $1, 5, 4, 12, 6, 20, 27, 28, 26, 16$

Overview

Case Study - Problem

Case Study Problem

XYZ Pty. Ltd. manufactures and maintains a set of golf buggies around the world. Each buggy is identified by a unique ID that encodes when it was made and what batch it came from, and buggies that have IDs in a particular range are from made in the same period and have same features in them. XYZ wants to build a custom database that maintains what is the current set of active buggies they are servicing around the world. They want to quickly retrieve buggies by their IDs and also quickly retrieve all buggies in a particular ID range. They ask for your help.

Case Study - Mapping the Problem to a Known Data Structure

The problem has two requirements:

- Retrieval of buggy by ID
- Retrieval of a set of buggies that have the queries ID range

Which data structure to use?

Case Study - Mapping the Problem to a Known Data Structure

The problem has two requirements:

- Retrieval of buggy by ID
- Retrieval of a set of buggies that have the queries ID range

Which data structure to use?
Binary search tree!

Case Study - Solving the Problem

Binary search tree:

- Use ID as the key.
- Use BST search to locate individual key.

Range queries?

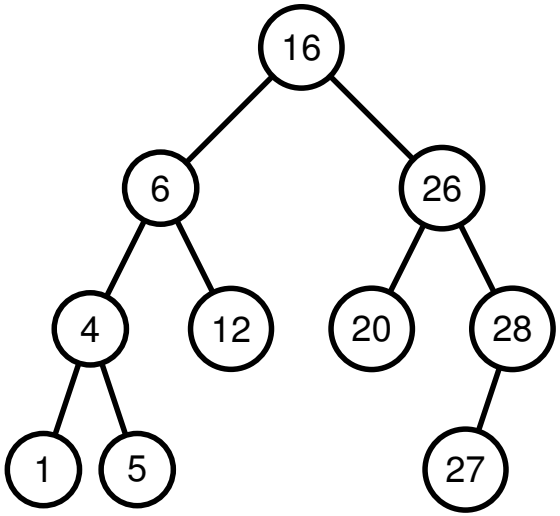
Case Study - Solving the Problem

Binary search tree:

- Use ID as the key.
- Use BST search to locate individual key.

Range queries?
Variation of in-order traversal!

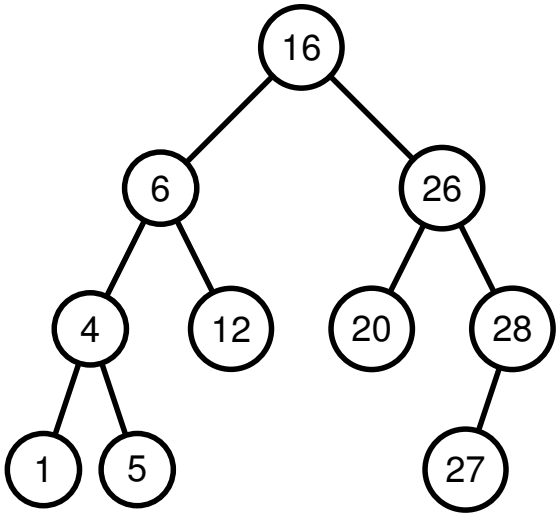
Case Study - Solving the Problem



ALGORITHM **RangeQ**(T, lr, ur)
// An range query of a binary search tree.
// INPUT : A search binary tree T (with labeled vertices), range [lr,ur].
// OUTPUT : Node labels that are within the range.

1: **if** $T \neq \emptyset$ **then**
2: If *node label* > *lr* then **RangeQ**(T_L, lr, ur)
3: If *lr* < *node label* < *ur* then **print** *node label*
4: If *node label* < *ur* then **RangeQ**(T_R, lr, ur)
5: **end if**

Case Study - Solving the Problem



ALGORITHM **RangeQ**(T, lr, ur)
// An range query of a binary search tree.
// INPUT : A search binary tree T (with labeled vertices), range [lr,ur].
// OUTPUT : Node labels that are within the range.

1: **if** $T \neq \emptyset$ **then**
2: If *node label* > *lr* then **RangeQ**(T_L, lr, ur)
3: If *lr* < *node label* < *ur* then **print** *node label*
4: If *node label* < *ur* then **RangeQ**(T_R, lr, ur)
5: **end if**

QUERY : [10,23]
OUTPUT : 12, 16, 20

Overview

Summary

- Introduced the *Divide-and-conquer* algorithmic approach.
- Sorting - merge and quick sort.
- Height and tree traversal for BST.
- Case study.