

COR-IS1702: COMPUTATIONAL THINKING

WEEK 13: REVIEW

Road Map

Algorithm Design and Analysis

(Weeks 1 - 5)

Fundamental Data Structures

(Weeks 6 - 9)

Computational Intractability and Heuristic
Reasoning

♦ Week 10: Heuristics

♦ Week 11: Limits of Computation

This week → ♦ **Week 13: Review**

Grand Revision

What have we covered so far?

Algorithm Design and Analysis

- ♦ 1: Introduction, Counting
- ♦ 2: Python, Algorithms
- ♦ 3: Complexity
- ♦ 4: Iteration & Decomposition
- ♦ 5: Recursion

Data Structures

- ♦ 6: Linear data structure (stack, queue, priority)
- ♦ 7: Non-linear (binary tree, binary search tree)
- ♦ 9: Non-linear (graph)

Heuristic Reasoning

- ♦ 10: Heuristics
- ♦ 11: Limits of Computing
- ♦ 13: Review

Algorithm Design & Analysis

Key Considerations

- ♦ What is the most straightforward way to solve the problem?
 - ❖ brute force: try all possibilities
- ♦ Is this efficient?
 - ❖ What is the total number of steps in the worst case?
 - ❖ How does it grow with the input size?
- ♦ Can we do it more efficiently?
 - ❖ by decomposing it into smaller problems that are easier to solve?
 - ❖ by stating the problem recursively?
 - ❖ by organizing the data in a smarter way?
- ♦ Is it absolutely necessary to get the true optimal answer?
 - ❖ How do we get a “good enough” answer in a shorter amount of time?

Algorithm Analysis

- ♦ Is it correct?
 - ❖ produces the right output given the input
- ♦ Is it efficient?
 - ❖ runs as fast as possible (efficient in time)
 - ❖ worst-case complexity: Big O notation

A more efficient algorithm has a slower growth rate in running time as the input size increases.

Big O Notation

- ♦ Algorithmic efficiency is measured in terms of growth of computation with respect to input size
- ♦ Big O notation
 - ❖ represents the concept of “upper bound”
 - ❖ deals with worst-case complexity (unless otherwise specified)
 - ❖ e.g., $O(n^2)$ means an algorithm is of the order n^2
- ♦ Steps to derive the Big O of an algorithm:
 - ❖ figure out what the worst case is
 - ❖ figure out the number of computations in terms of input size n
 - ❖ reduce it to the “dominant” terms

Dominance Rules for Big O Notation

- ♦ If there are additive terms, keep only the most dominant term
 - ❖ If number of operations is $2^n + n^2$, complexity is $O(2^n)$
 - ❖ If number of operations is $n^2 + \log n$, complexity is $O(n^2)$
- ♦ Higher order dominates lower order
 - ❖ If number of operations is $n^3 + n^2 + n$, the complexity is $O(n^3)$
 - ❖ If number of operations is $5^n + 2^n$, the complexity is $O(5^n)$
- ♦ Ignore multiplicative constants in the highest-order term
 - ❖ If number of operations is $3n^2$, the complexity is $O(n^2)$
- ♦ **Watch out:** do not drop multiplicative terms
 - ❖ If number of operations is $2^n \times n^2$, the complexity is $O(2^n \times n^2)$
- ♦ **Watch out:** if it involves terms that are not necessarily comparable, keep them
 - ❖ If number of operations is $m \times n$, the complexity is $O(m \times n)$
 - ❖ If number of operations is $2^m + 2^n$, the complexity is $O(2^m + 2^n)$

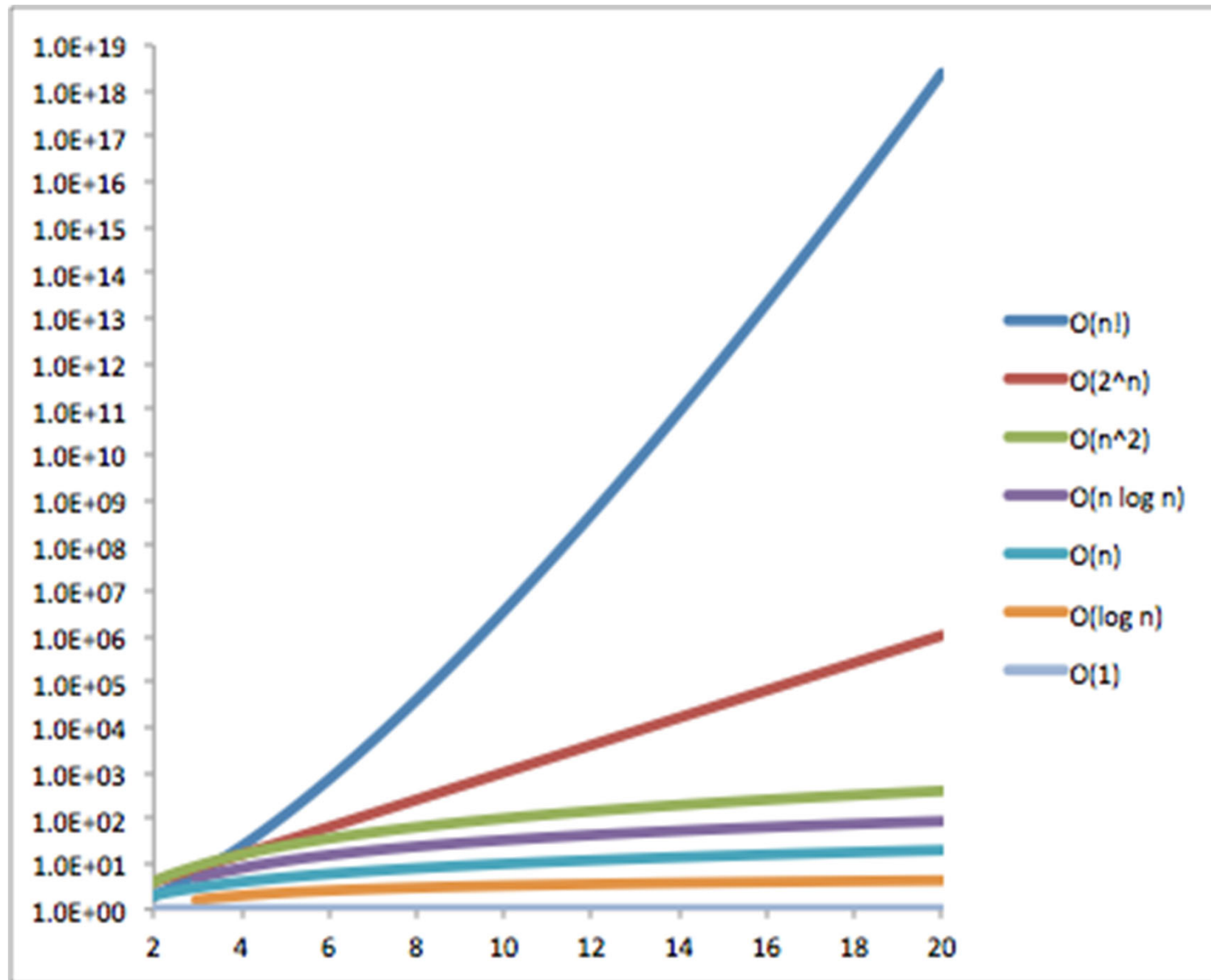
Common Orders of Complexity

increasing order of complexity
↓

	Big O	Remarks
Constant	$O(1)$	not affected by input size n
Logarithmic	$O(\log n)$	we will see this during decomposition/recursion
Linear	$O(n)$	roughly proportional to input size n
Linearithmic	$O(n \log n)$	we will see this during decomposition/recursion
Polynomial	$O(n^k)$	k is some constant, e.g., $k = 1$ is linear, $k = 2$ is quadratic, $k = 3$ is cubic
Exponential	$O(k^n)$	k is some constant
Factorial	$O(n!)$	often considered to be within exponential family

Complexities are not limited to these only

Log
scale



factorial
exponential
polynomial
linearithmic
linear
logarithmic
constant

Example: Complexity

```
def f(n):  
    sum = 0  
    for i in range(1, n):  
        sum = sum + i  
    return sum  
  
def g(n):  
    sum = 0  
    for i in range(1, n):  
        sum = sum + i + f(i)  
    return sum
```

Algorithm Design

- ✦ Brute Force (aka Exhaustive Search)
 - ❖ exhaustively try all the reasonable possibilities
 - ❖ simple, but may be computationally expensive
- ✦ Decomposition
 - ❖ breaking a large problem into smaller sub-problems
 - ❖ Divide and Conquer
 - ❖ can be implemented as iteration, or as recursion
- ✦ Heuristics Reasoning
 - ❖ trading off quality of solution (good enough) for more efficient runtime
 - ❖ greedy algorithm
 - ❖ local search

Example Recursion

Write a recursive algorithm `repeatstring(a, n)` that returns a concatenation of n copies of the string `a`.

What is the base case?

`n == 1`

What is the reduction step?

`a + repeatstring(a, n-1)`

```
def repeatstring(a, n):  
    if n == 1:  
        return a  
    return a + repeatstring(a, n-1)
```

Worst-case complexity: $O(n)$

Iterative vs. Recursive for Factorial

Iterative

```
def factorial(n):  
    f = 1  
    i = n  
    while i > 0:  
        f = f * i  
        i = i - 1  
    return f
```

Recursive

```
def factorial(n):  
    if n == 1:  
        return 1  
    else:  
        return n * factorial(n-1)
```

Worst-case complexity is $O(n)$ for both iterative and recursive algorithms.

Recursive Algorithm for Fibonacci

♦ Reduction:

$$\diamond \text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2)$$

♦ Base cases:

$$\diamond \text{fibonacci}(0) = 0, \text{fibonacci}(1) = 1$$

```
def fibonacci(n):  
    if n == 0:  
        return 0  
    elif n == 1:  
        return 1  
    else:  
        return fibonacci(n-1) + fibonacci(n-2)
```

Recursive version of
fibonacci

every invocation leads to two
recursive calls with similar problem
size $\sim n$

Recursion Not Necessarily More Efficient

Iterative version of fibonacci(n)

```
def fibonacci(n):  
    if n == 0:  
        return 0  
    x = 0                #base case fibonacci(0)  
    y = 1                #base case fibonacci(1)  
    for i in range(2, n+1): #fibonacci(2) and so on  
        z = x + y        #sum the previous two numbers  
        x = y            #shift x, y to most recent 2 numbers  
        y = z  
    return y
```

The iterative algorithm has a single `for` loop, running $n-1$ times.
Worst-case complexity is $O(n)$, much less than the recursive version.

Searching Problem

- ♦ Given a collection of numbers, find a specific number
 - ❖ $a = [29, 14, 50, 2, 24, 71, 27]$
- ♦ What is the most straightforward way to solve the problem?
 - ❖ brute force: linear search
- ♦ What is the complexity?
- ♦ Can we do it more efficiently?

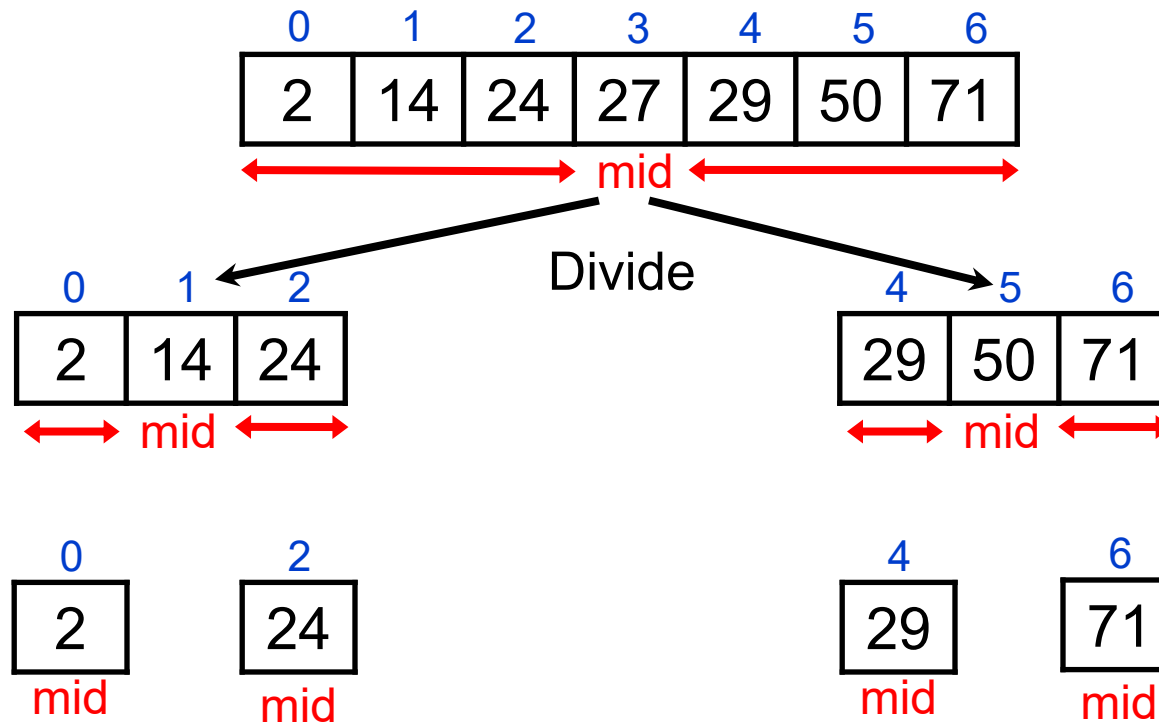
Searching Algorithm

- ♦ Brute force approach: $O(n)$ complexity
 - ❖ if linearly-organized, e.g., lists or arrays, use linear search
 - ▶ scope of search is all elements
 - ❖ if a binary tree, use traversal (preorder, inorder, postorder)
 - ▶ scope of search is dependent on root of subtree
 - ❖ if a graph, use traversal (depth first search, breadth first search)
 - ▶ scope of search is dependent on the source
- ♦ Decomposition (Divide-and-Conquer): $O(\log n)$ complexity
 - ❖ organize into sorted array
 - ▶ binary search on sorted list/array
 - ▶ binary search can be written iteratively or recursively
 - ❖ organize into binary search tree (assumed balanced tree)

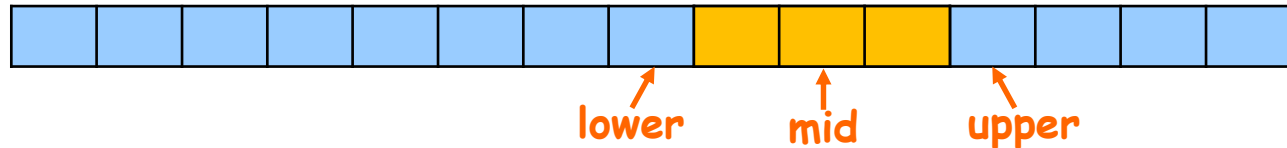
Using Decomposition

- ♦ Binary search
 - ❖ array must first be sorted: $a = [2, 14, 24, 27, 29, 50, 71]$
- ♦ To search a list of n items, first look at the item in location $n/2$
 - ❖ then search either the region from 0 to $n/2-1$ or from $n/2+1$ to $n-1$
- ♦ How many comparisons are needed?
- ♦ What is the complexity?

Binary Search Divide & Conquer Strategy



Binary Search (Iterative version)



```
def bSearch(array, target):  
    lower = -1  
    upper = len(array)  
    while not (lower + 1 == upper):  
        mid = (lower + upper) // 2  
        if target == array[mid]:  
            return mid  
        elif target < array[mid]:  
            upper = mid  
        else:  
            lower = mid  
    return -1
```

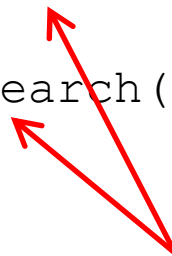
#fail if region empty
#find middle of region
#succeed if k is at mid

#search lower region

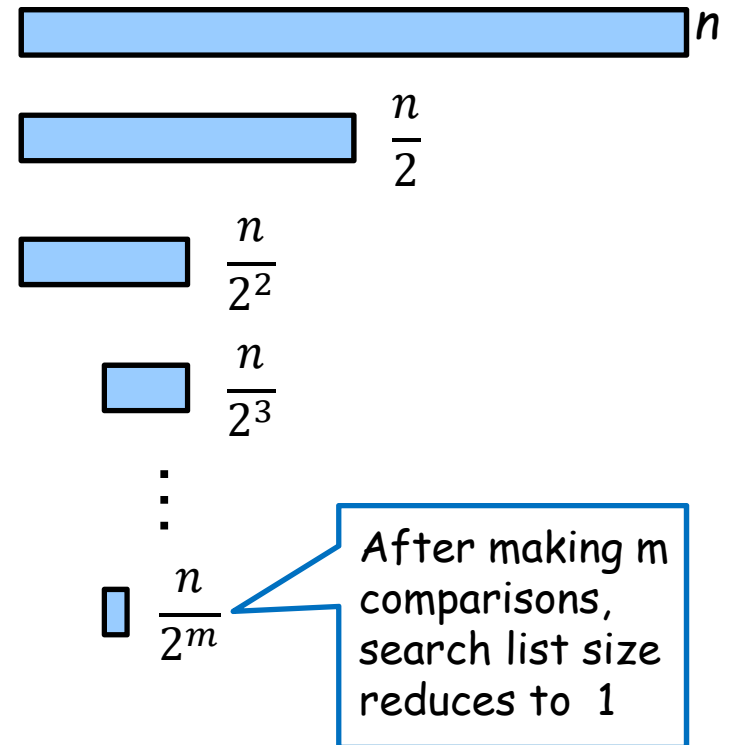
#search upper region
#not found

Binary Search (Recursive version)

```
01 def rbsearch(array, target, lower=None, upper=None):
02     if lower == None:
03         lower = -1
04         upper = len(array)
05
06     if lower + 1 == upper:      # base case
07         return -1              # not found
08     mid = (lower + upper) // 2
09     if array[mid] == target:    # success
10         return mid
11     elif array[mid] < target:   # search upper region
12         return rbsearch(array, target, mid, upper)
13     else:                     # search lower region
14         return rbsearch(array, target, lower, mid)
```

 while loop replaced
by recursive calls

Complexity of Binary Search



$$\frac{n}{2^m} = 1 \rightarrow n = 2^m \rightarrow m = \log_2 n$$

Worst case: reduce original search area of size n to a search area of size 1, and perform the final comparison to conclude search failure.

$$\text{\# of comparison} = (\log_2 n + 1)$$

Sorting Problem

- ♦ Given n elements, sort them in increasing order
- ♦ Brute force algorithm:
 - ❖ for every possible permutation of the n elements
 - ▶ check whether the elements are in sorted order
- ♦ What is the complexity?
- ♦ There must be a better way

Insertion Sort

Incremental approach

- ♦ At any point in the algorithm
left part of the array is already sorted

```
def iSort(array):
```

```
    i = 1
```

```
    while i < len(array):
```

```
        move_left(array, i)
```

```
        i += 1
```

```
    return array
```

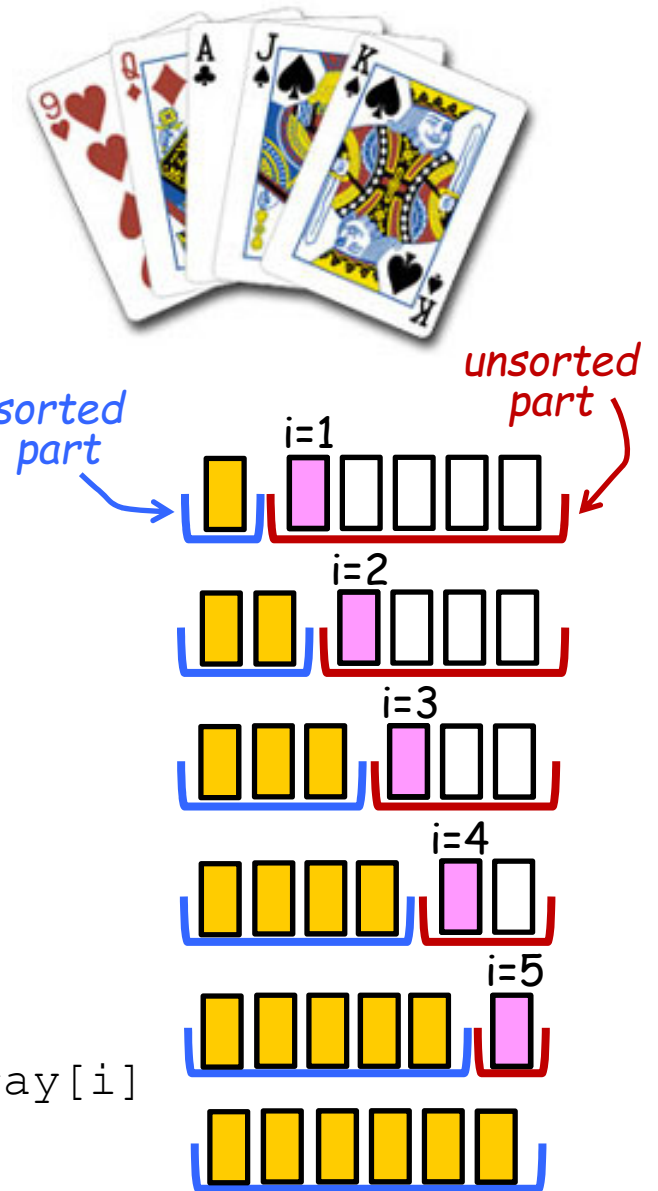
```
def move_left(array, i):
```

```
    while i > 0 and array[i] < array[i-1]:
```

```
        # swap elements at i-1 and i
```

```
        array[i], array[i-1] = array[i-1], array[i]
```

```
        i -= 1 # move left
```



Complexity of Insertion Sort

- ◆ In general, for an array with n items, the potential number of comparisons for $i = 1 \dots n-1$:

- ❖ $i = 1$, 1 comparison
- ❖ $i = 2$, 2 comparisons
- ❖ ...
- ❖ $i = n-1$, $(n-1)$ comparisons

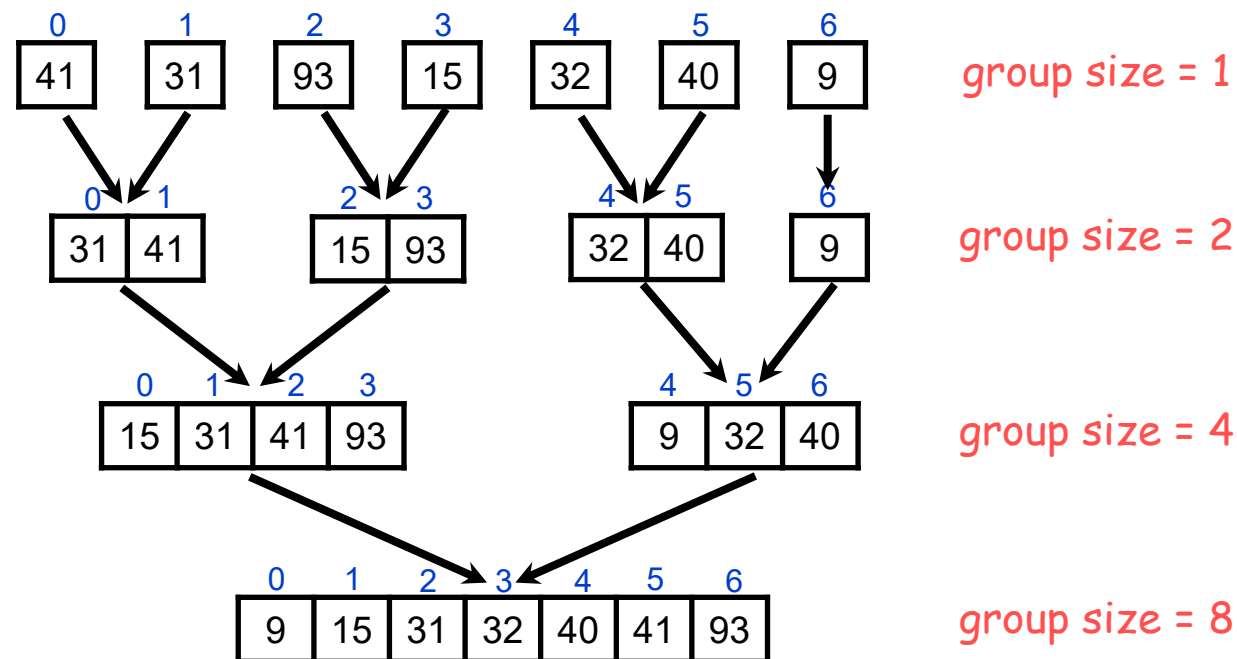
$$\text{Total \# of comparisons : } \frac{n \times (n - 1)}{2}$$

	0	1	2	3	4	5
0						
1	•					
2	•	•				
3	•	•	•			
4	•	•	•	•		
5	•	•	•	•	•	

- ◆ This worst case happens when the array is originally sorted in reverse order.
- ◆ Best case happens when the array is originally in the desired order.

Merge Sort (Iterative version / Bottom-up Approach)

- ◆ The iterative merge sort algorithm works from “the bottom up”
 - ❖ start by solving the smallest pieces of the main problem
 - ❖ keep combining their results into larger solutions
 - ❖ eventually the original problem will be solved



a = [41, 31, 93, 15, 32, 40, 9]

mSort(a) → merge_groups(a, 1) → a[0:2] = merge(a, 0, 1)
 a[2:4] = merge(a, 2, 1)
 a[4:6] = merge(a, 4, 1)
 a[6:8] = merge(a, 6, 1)

merge_groups(a, 2) → a[0:4] = merge(a, 0, 2)
 a[4:8] = merge(a, 4, 2)

merge_groups(a, 4) → a[0:8] = merge(a, 0, 4)

0	1	2	3	4	5	6
9	15	31	32	40	41	93

0	1	2	3
15	31	41	93

4	5	6
9	32	40

0	1
31	41

2	3
15	93

4	5
32	40

6
9

0	1
41	31

2	3
93	15

4	5
32	40

6
9

```

1: def mSort(array):
2:   groupsizes = 1
3:   while groupsizes < len(array):
4:     merge_groups(array, groupsizes)
5:     groupsizes *= 2
6:   return array
  
```

```

1: def merge_groups(a, gs)
2:   i = 0
3:   while i < len(a):
4:     j = i + 2*gs
5:     a[i:j] = merge(a, i, gs)
6:     i += 2*gs
  
```

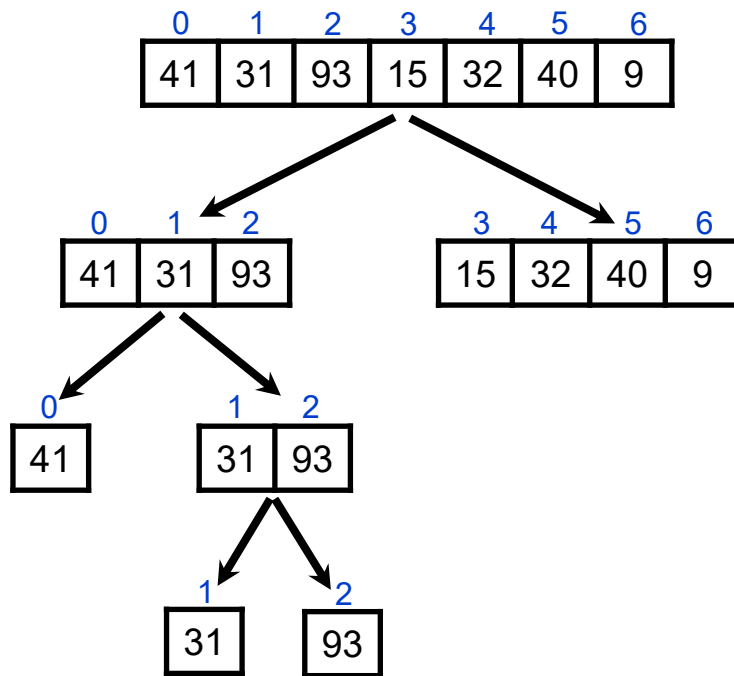
```

def merge(array, i, groupsizes):
  r = []
  firstGroup = array[i:i+groupsizes]
  secondGroup = array[i+groupsizes:i+groupsizes*2]

  while (len(firstGroup) != 0 or len(secondGroup) != 0):
    if (len(firstGroup) == 0):
      while (len(secondGroup) != 0):
        r.append(secondGroup.pop(0))
    elif (len(secondGroup) == 0):
      while (len(firstGroup) != 0):
        r.append(firstGroup.pop(0))
    else:
      if (firstGroup[0] > secondGroup[0]):
        r.append(secondGroup.pop(0))
      else:
        r.append(firstGroup.pop(0))
  return r
  
```

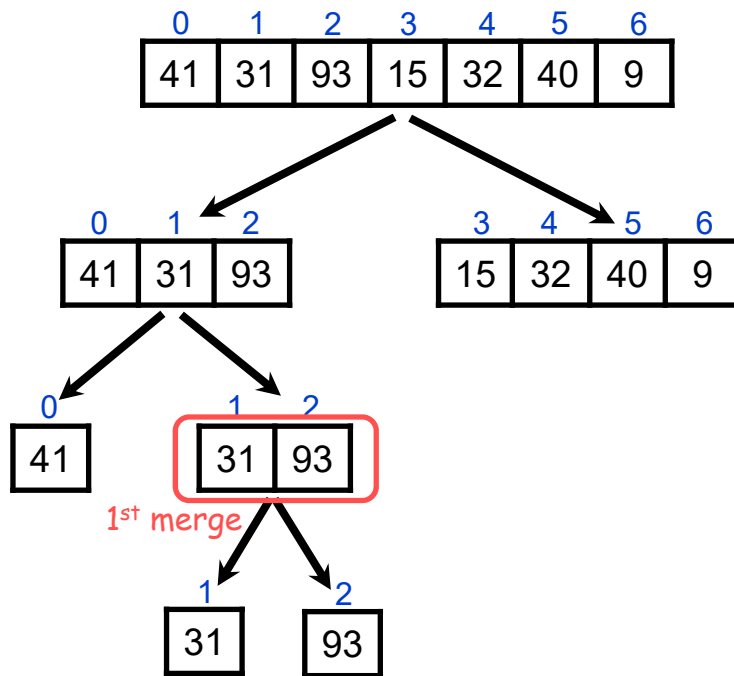
Merge Sort (Recursive version / Top-down Approach)

- ♦ The recursive merge sort is based on top-down strategy
 - ❖ Begin with the full array to be sorted
 - ❖ Divide the array into shorter arrays to be sorted
 - ❖ Sort (conquer) the shorter arrays recursively



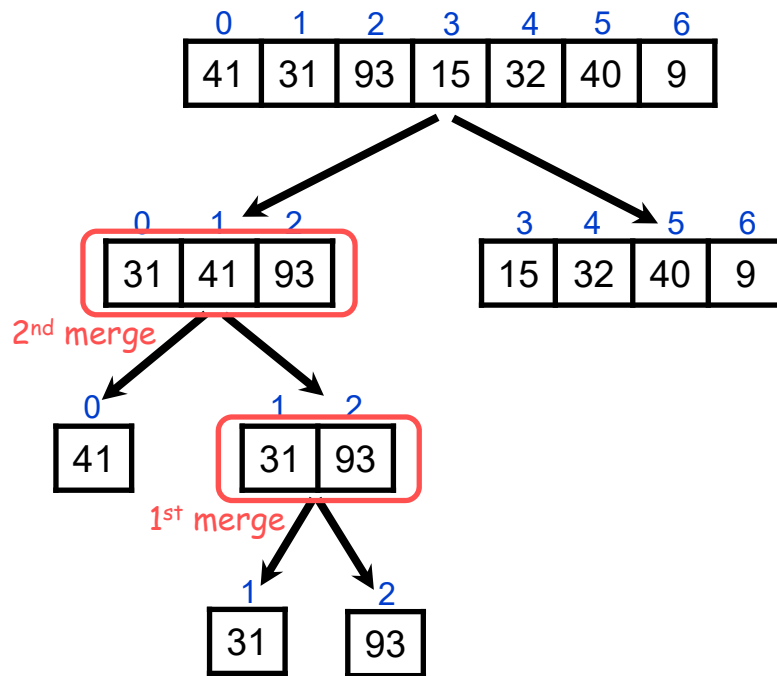
Merge Sort (Recursive version / Top-down Approach)

- ◆ The recursive merge sort is based on top-down strategy
 - ❖ Begin with the full array to be sorted
 - ❖ Divide the array into shorter arrays to be sorted
 - ❖ Sort (conquer) the shorter arrays recursively



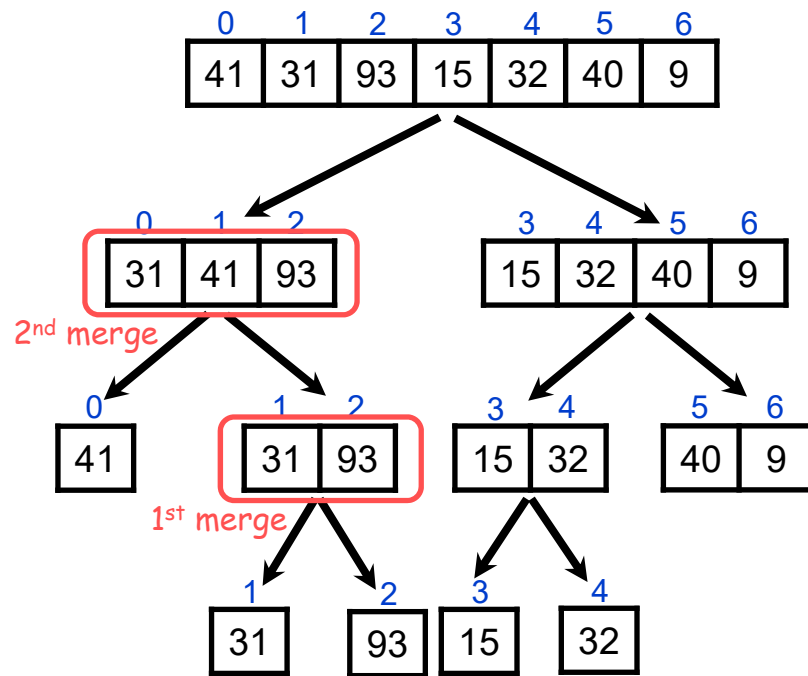
Merge Sort (Recursive version / Top-down Approach)

- ◆ The recursive merge sort is based on top-down strategy
 - ❖ Begin with the full array to be sorted
 - ❖ Divide the array into shorter arrays to be sorted
 - ❖ Sort (conquer) the shorter arrays recursively



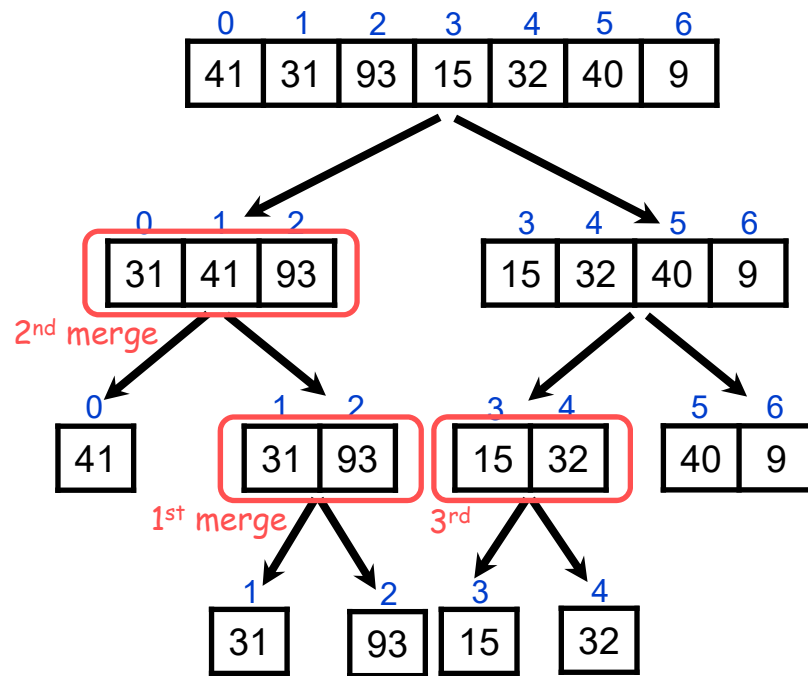
Merge Sort (Recursive version / Top-down Approach)

- ♦ The recursive merge sort is based on top-down strategy
 - ❖ Begin with the full array to be sorted
 - ❖ Divide the array into shorter arrays to be sorted
 - ❖ Sort (conquer) the shorter arrays recursively



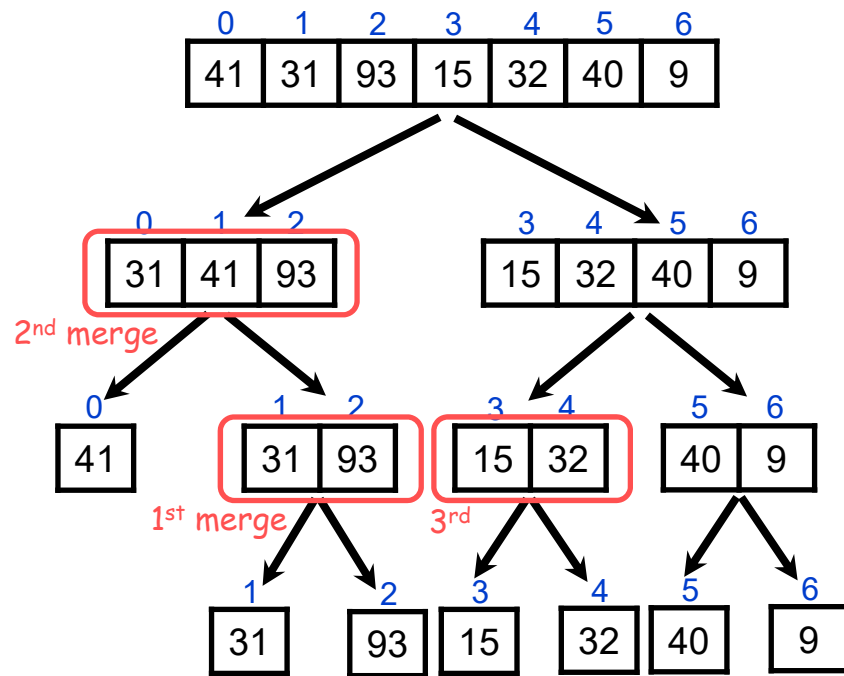
Merge Sort (Recursive version / Top-down Approach)

- ◆ The recursive merge sort is based on top-down strategy
 - ❖ Begin with the full array to be sorted
 - ❖ Divide the array into shorter arrays to be sorted
 - ❖ Sort (conquer) the shorter arrays recursively



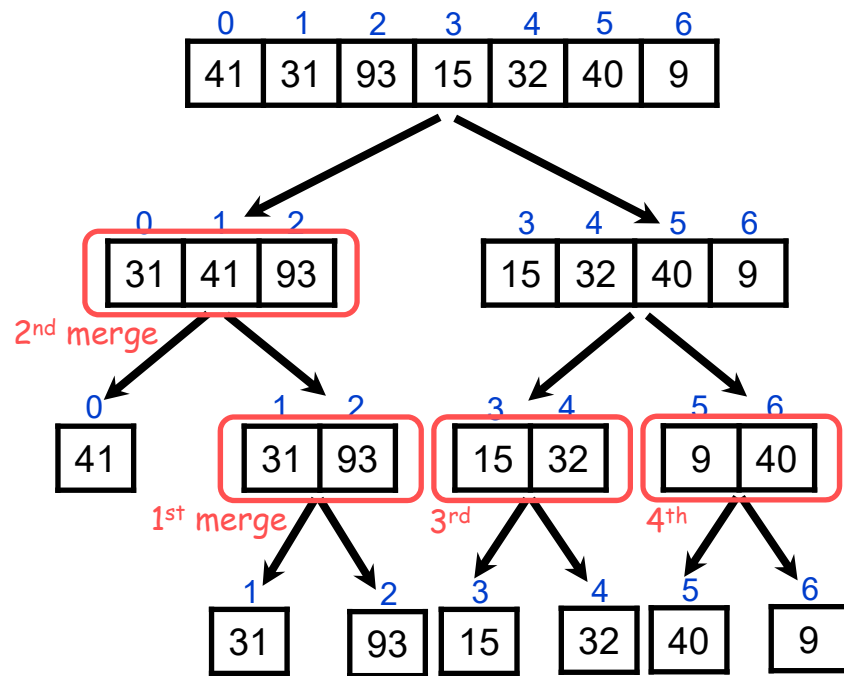
Merge Sort (Recursive version / Top-down Approach)

- ◆ The recursive merge sort is based on top-down strategy
 - ❖ Begin with the full array to be sorted
 - ❖ Divide the array into shorter arrays to be sorted
 - ❖ Sort (conquer) the shorter arrays recursively



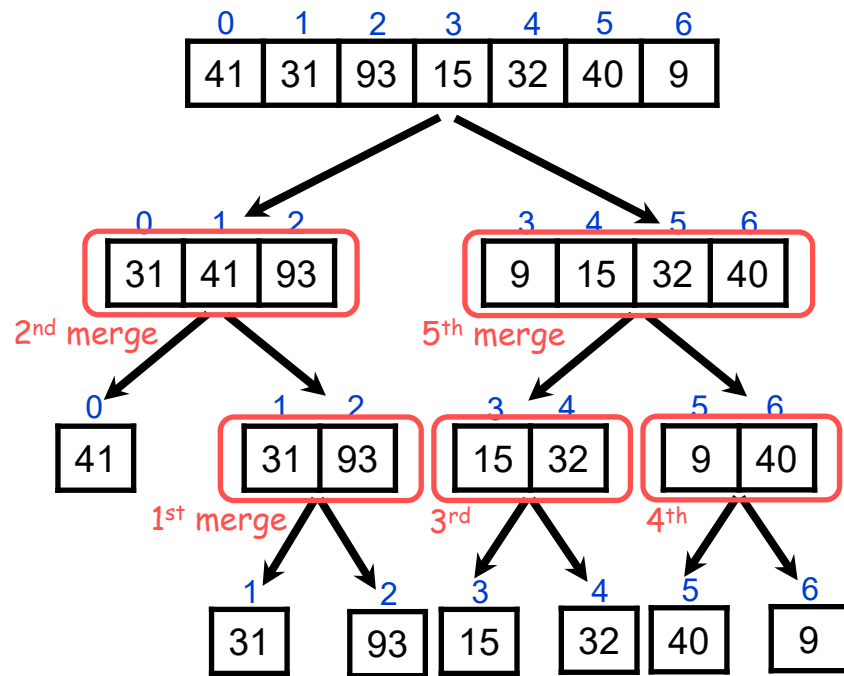
Merge Sort (Recursive version / Top-down Approach)

- ◆ The recursive merge sort is based on top-down strategy
 - ❖ Begin with the full array to be sorted
 - ❖ Divide the array into shorter arrays to be sorted
 - ❖ Sort (conquer) the shorter arrays recursively



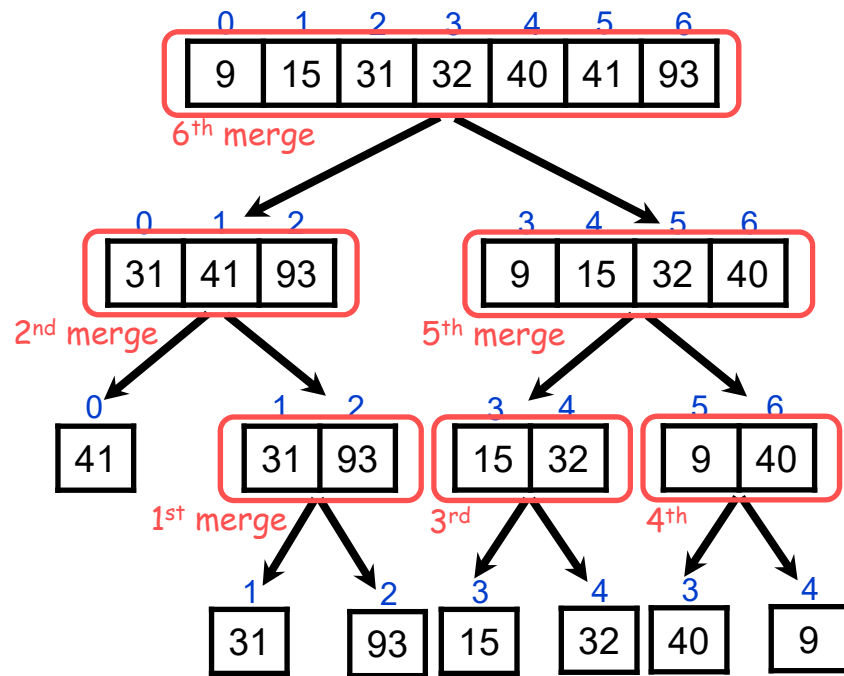
Merge Sort (Recursive version / Top-down Approach)

- ◆ The recursive merge sort is based on top-down strategy
 - ❖ Begin with the full array to be sorted
 - ❖ Divide the array into shorter arrays to be sorted
 - ❖ Sort (conquer) the shorter arrays recursively

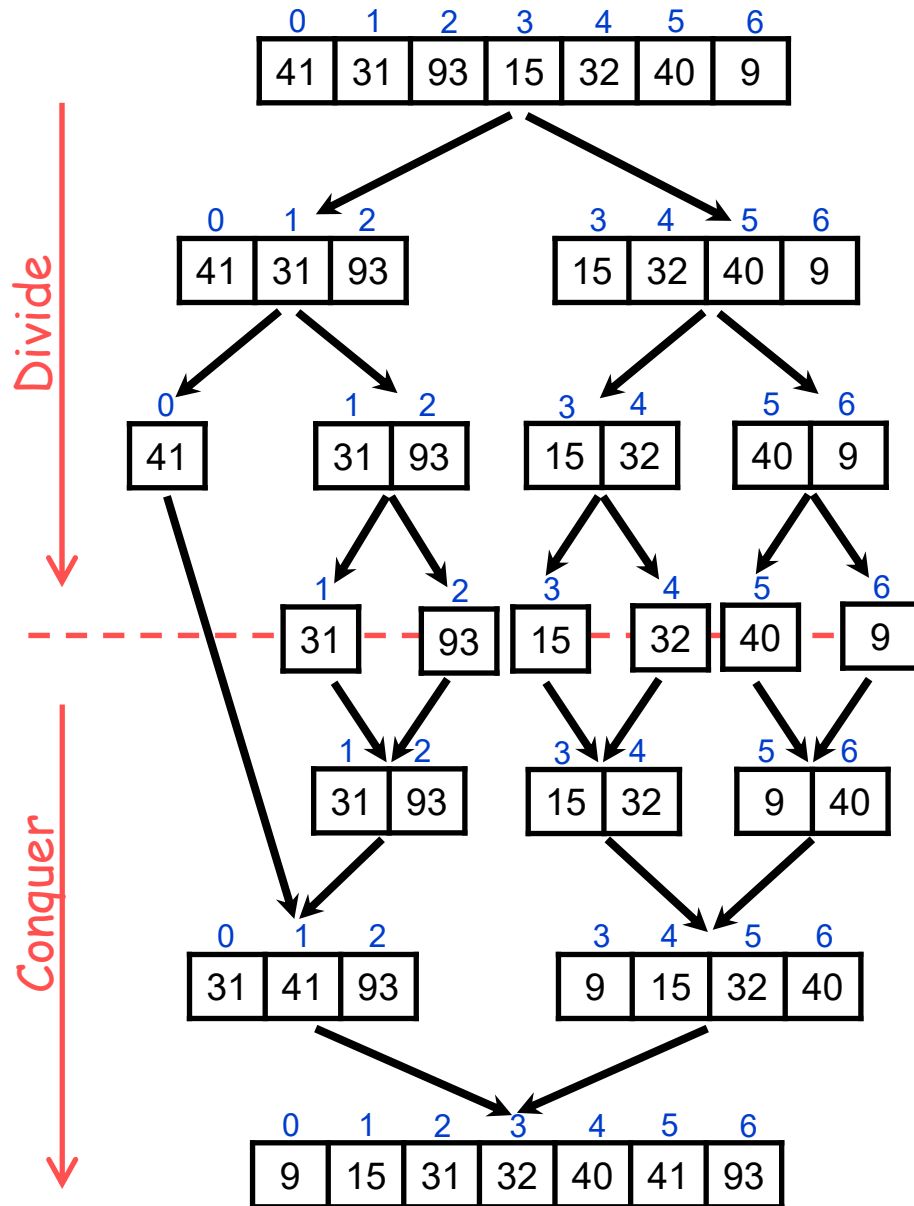


Merge Sort (Recursive version / Top-down Approach)

- ◆ The recursive merge sort is based on top-down strategy
 - ❖ Begin with the full array to be sorted
 - ❖ Divide the array into shorter arrays to be sorted
 - ❖ Sort (conquer) the shorter arrays recursively



Tracing of Recursive Merge Sort



```
def rmSort(a):
```

```
    if len(a) == 1:
```

```
        return a
```

```
    mid = len(a)//2
```

```
    a1 = rmSort(a[0:mid])
```

```
    a2 = rmSort(a[mid:len(a)])
```

```
    return merge2(a1, a2)
```

```
def merge2(a1, a2):
```

```
    i = 0
```

```
    j = 0
```

```
    ret = []
```

```
    while i < len(a1) or j < len(a2):
```

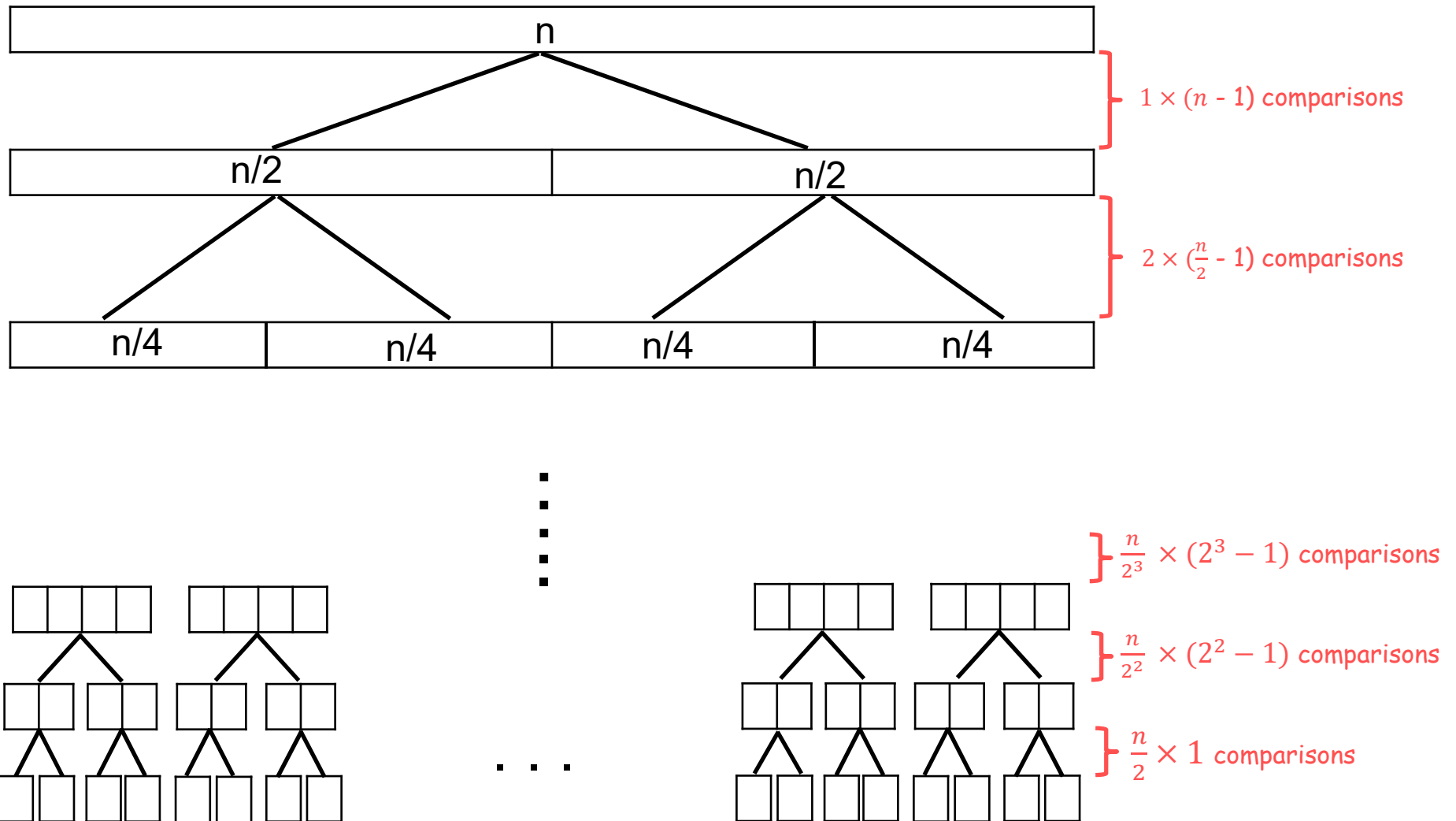
```
        if (j == len(a2)) or
           (i < len(a1) and a1[i] < a2[j]):
            ret.append(a1[i]) #pick item fr a1
            i += 1
```

```
        else:
```

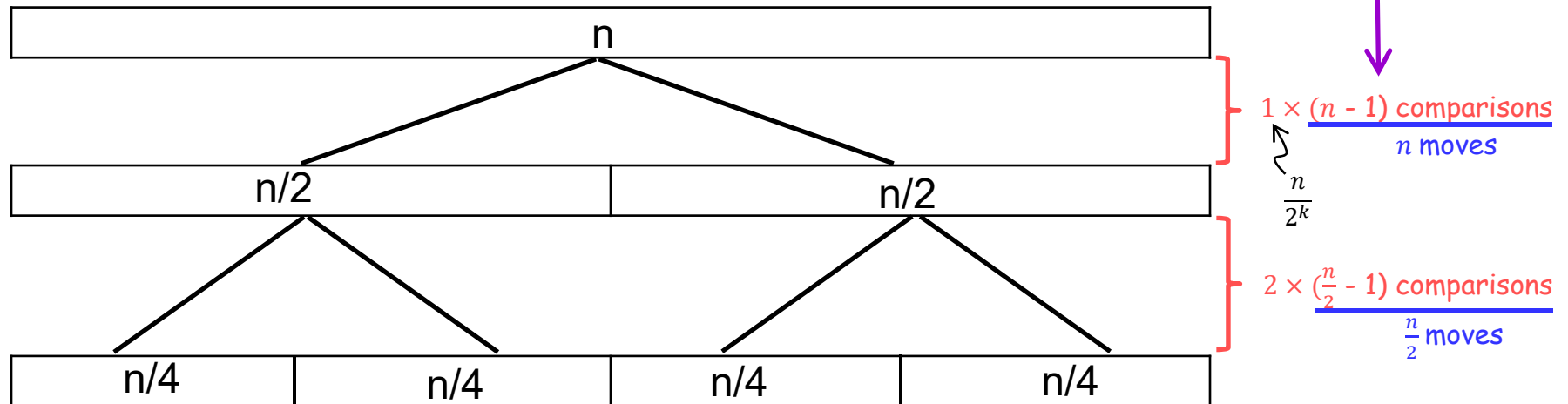
```
            ret.append(a2[j]) #pick item fr a2
            j += 1
```

```
    return ret
```

Complexity of Merge Sort



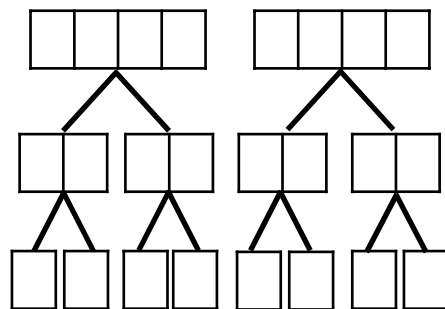
Complexity of Merge Sort



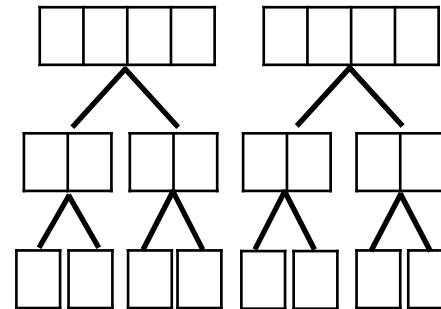
$$\begin{aligned} n/2^k &= 1 \\ \Rightarrow n &= 2^k \\ \Rightarrow k &= \log_2 n \end{aligned}$$

Thus, there are $\log_2 n$ levels of n moves each.

Total # of moves = $n \log_2 n$



...



Level 3 to 4: $\frac{n}{2^3} \times \frac{(2^3-1) \text{ comparisons}}{2^3 \text{ moves}}$

Level 2 to 3: $\frac{n}{2^2} \times \frac{(2^2-1) \text{ comparisons}}{2^2 \text{ moves}}$

Level 1 to 2: $\frac{n}{2} \times \frac{1 \text{ comparisons}}{2 \text{ moves}}$

Comparison of Sorting Algorithms

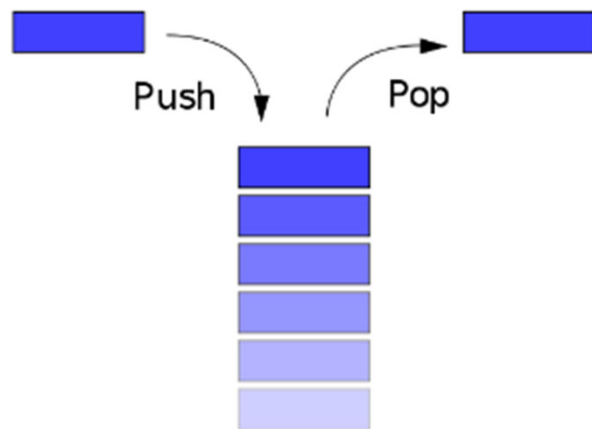
	Insertion Sort	Merge Sort
<i>Worst case</i>	$O(n^2)$, when array in reverse	$O(n \log n)$
<i>Average case</i>	$O(n^2)$	$O(n \log n)$
<i>Best case</i>	$O(n)$, when array already sorted	$O(n \log n)$
<i>Pros</i>	very fast for small arrays	near optimal no. of comparisons
<i>Cons</i>	very slow for large arrays	requires temporary storage

Data Structures

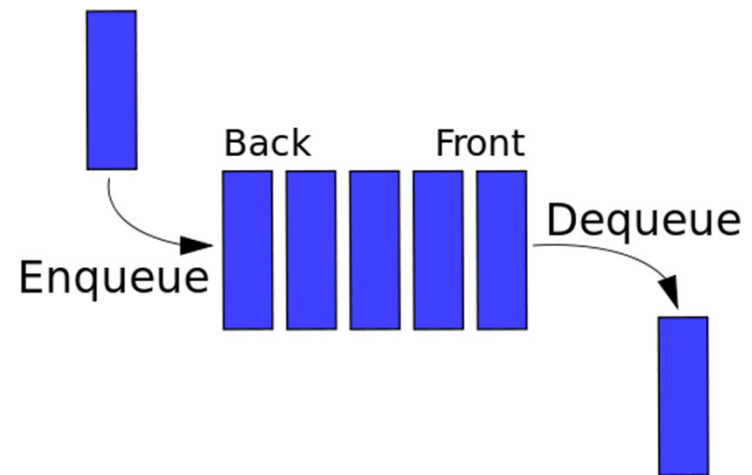
Linear Data Structures

- ♦ Only one data element is accessible at any point of time
- ♦ Simplifies the client's programming logic
 - ❖ no need to keep track of indices
- ♦ The key question is **which** data element is accessible
 - ❖ stack vs. queue vs. priority queue

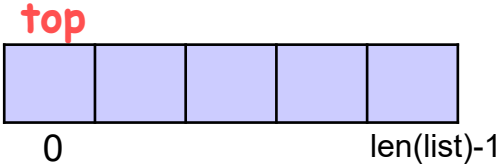
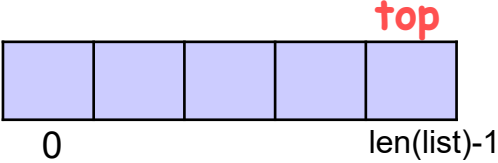
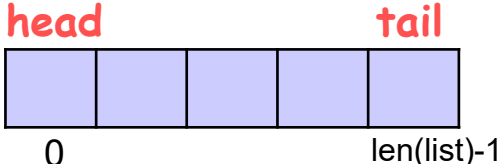
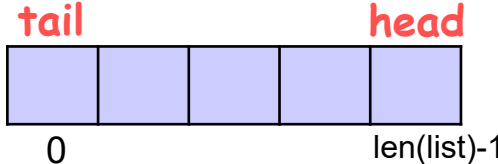
Stack: Last-In First-Out (LIFO)



Queue: First-In First-Out (FIFO)



List-based Implementation of Stack / Queue

Stack	Queue
<p>❖ Top is at position 0</p> <ul style="list-style-type: none">▪ push(x) is $O(n)$▪ pop() is $O(n)$  <p>0 len(list)-1</p> <p>❖ Top is at position len(list)-1</p> <ul style="list-style-type: none">▪ push(x) is $O(1)$▪ pop() is $O(1)$  <p>0 len(list)-1</p>	<p>❖ Head at position 0, Tail at position len(list)-1</p> <ul style="list-style-type: none">▪ enqueue(x) is $O(1)$▪ dequeue() is $O(n)$  <p>0 len(list)-1</p> <p>❖ Head at position len(list)-1, Tail at position 0</p> <ul style="list-style-type: none">▪ enqueue(x) is $O(n)$▪ dequeue() is $O(1)$  <p>0 len(list)-1</p>

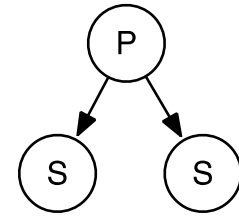
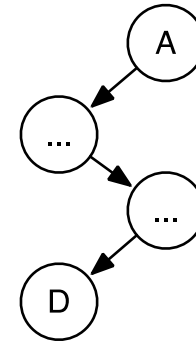
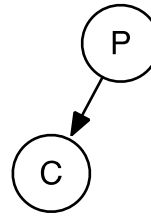
Comparison of Priority Queue Implementations

	Unsorted List	Sorted List	Binary Search Tree (worst case)	Binary Search Tree (average case)
enqueue	$O(1)$	$O(n)$	$O(n)$	$O(\log n)$
peek	$O(1)$	$O(1)$	$O(1)$	$O(1)$
dequeue	$O(n)$	$O(1)$	$O(n)$	$O(\log n)$

Non-linear Data Structures – Binary Trees

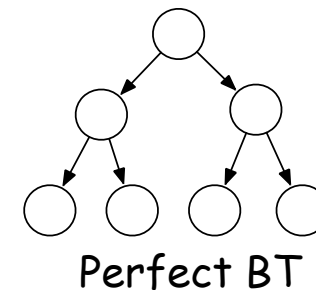
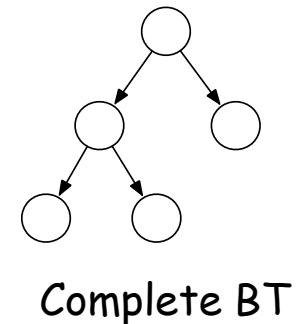
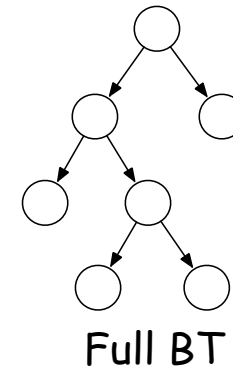
◆ Relationship

- ❖ Parent and child
- ❖ Ancestor and descendant
- ❖ Siblings



◆ Types of Binary Trees

- ❖ Full – each node has 0 or 2 children
- ❖ Complete – every level is filled except the last
 - last level is filled from left
- ❖ Perfect – all leaves are at the same depth
 - all internal nodes have 2 children



Traversals of a Binary Tree

- ♦ Pre-order **c L R**
 - ❖ visiting a parent first, before visiting the left child and the right child
- ♦ In-order **L c R**
 - ❖ visiting the parent after the left child, but before the right child
- ♦ Post-order **L R c**
 - ❖ visiting the parent after visiting the left and right children

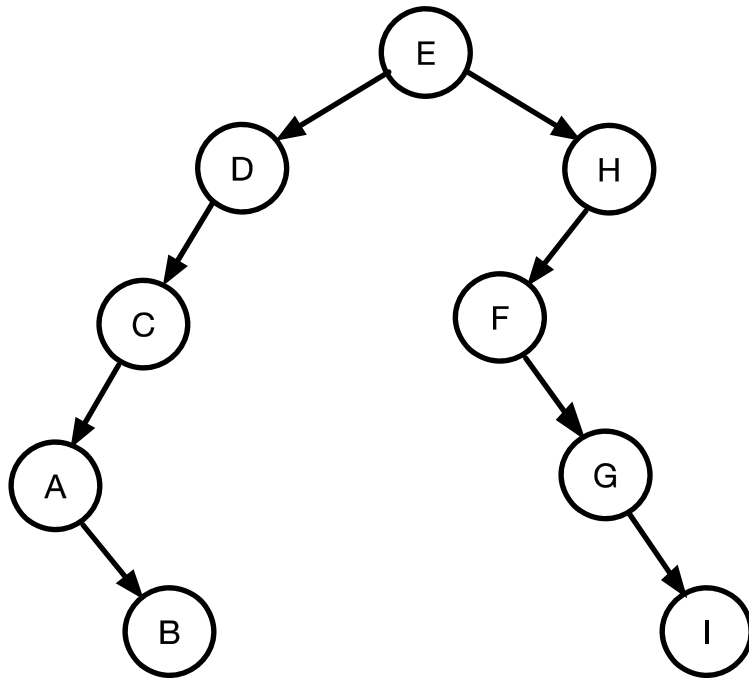
```
def preorder_traverse(node):  
    if node != None:  
        c visit(node)  
        L preorder_traverse(node.left)  
        R preorder_traverse(node.right)
```

```
def postorder_traverse(node):  
    if node != None:  
        L postorder_traverse(node.left)  
        R postorder_traverse(node.right)  
        c visit(node)
```

```
def inorder_traverse(node):  
    if node != None:  
        L inorder_traverse(node.left)  
        c visit(node)  
        R inorder_traverse(node.right)
```

Each node is visited once, therefore
time of complexity of the traversal
algorithms is $O(n)$

Practice Traversal

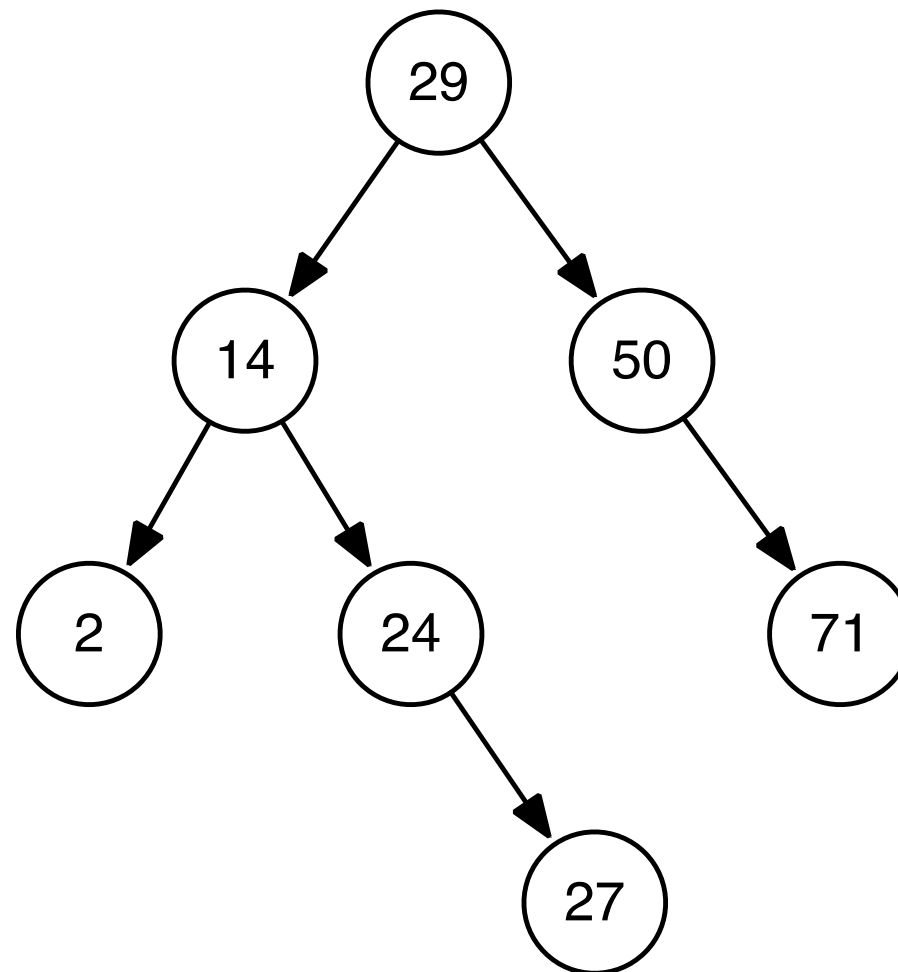


Pre-order traversal:

In-order traversal:

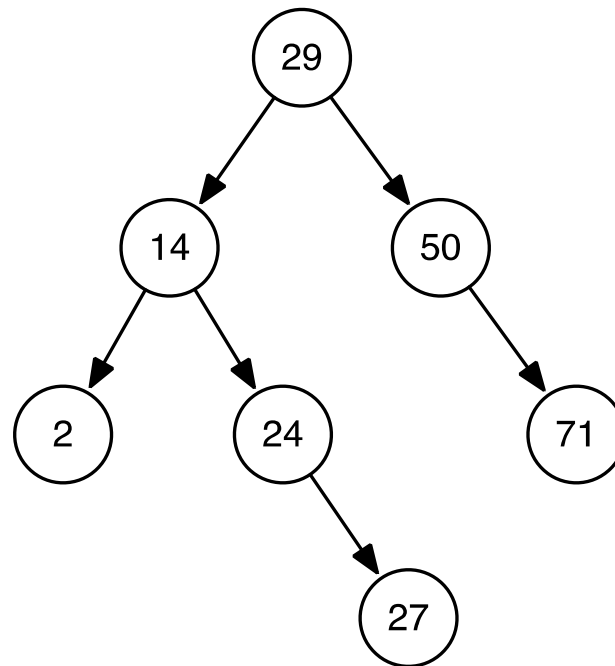
Post-order traversal:

Binary Search Tree

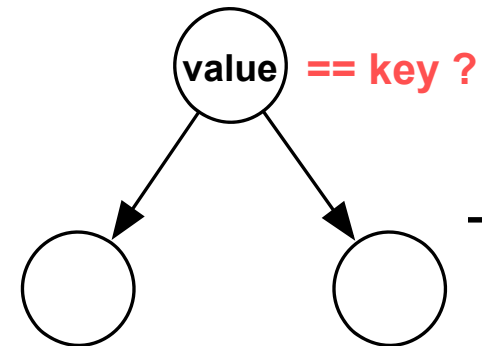


Using Data Structure: Binary Search Tree

- ♦ Start from the root node
 - ❖ if the root node has the `value == key`, return the root node
 - ❖ if the root node has `value > key`, recursively search the left subtree
 - ❖ if the root node has `value < key`, recursively search the right subtree
 - ❖ else, return `None`



Helper Method searchbst



```
def searchbst(root, key):  
    if root.value == key:  
        return root  
    elif root.left != None and root.value > key:  
        return searchbst(root.left, key)  
    elif root.right != None and root.value < key:  
        return searchbst(root.right, key)  
    else:  
        return None
```

Complexity?

For a balanced tree, time complexity will be $O(\log n)$

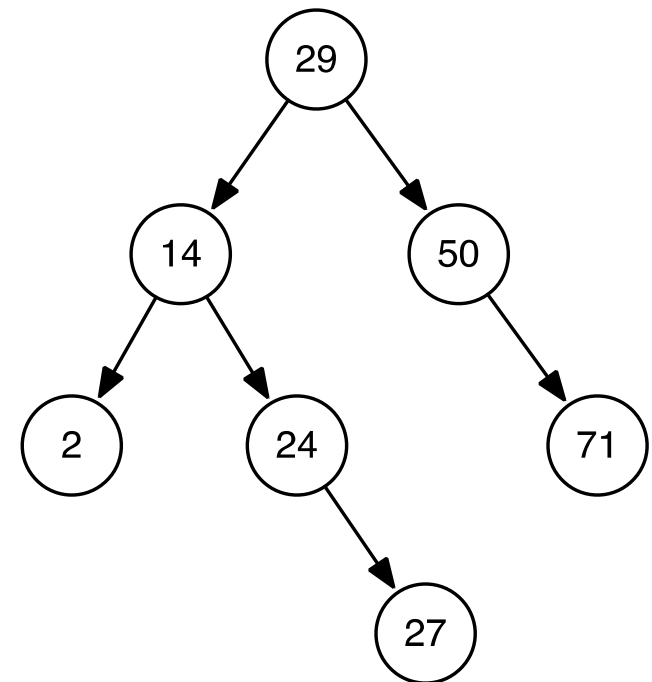
For an unbalanced tree, time complexity will be $O(n)$

Inserting a Node into a BST

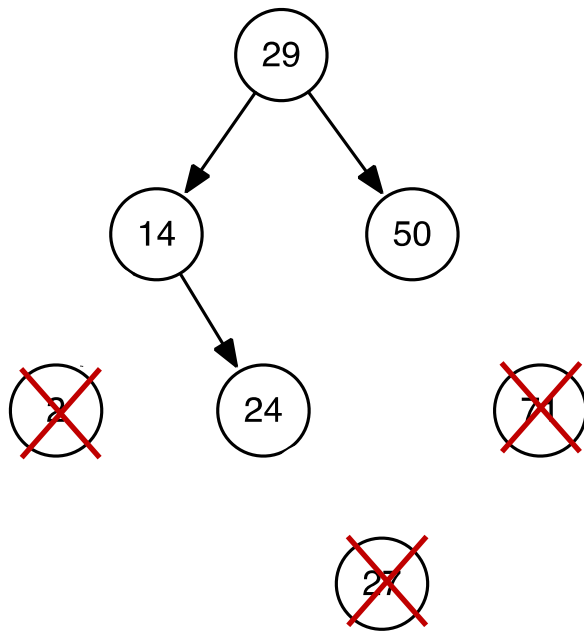
```
def insertbst(root, key):  
    if root.value == key:  
        return root  
    elif root.value > key:  
        if root.left != None:  
            return insertbst(root.left, key)  
        else:  
            node = Node(key)  
            root.setLeft(node)  
            return node  
    else:  
        if root.right != None:  
            return insertbst(root.right, key)  
        else:  
            node = Node(key)  
            root.setRight(node)  
            return node
```

Removing a Node from BST

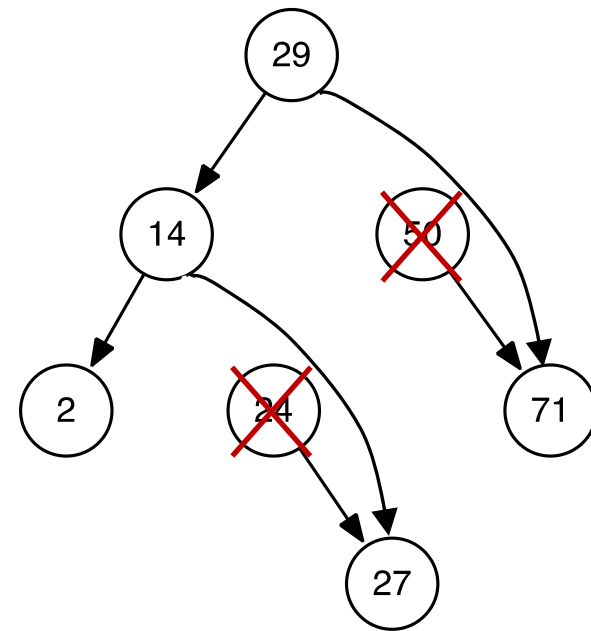
- ♦ Take care to keep the resulting BST in proper order
- ♦ Traverse to find the node
- ♦ Need to consider three scenarios:
 - a) the node to be removed is a leaf node
 - b) the node to be removed has one child
 - c) the node to be removed has two children



◆ Removing a leaf node



◆ Removing a node with just one child



- ◆ Removing a node n with two children
 - ❖ choose the left-most node of the right-subtree of n
 - ❖ choose the right-most node of the left-subtree of n

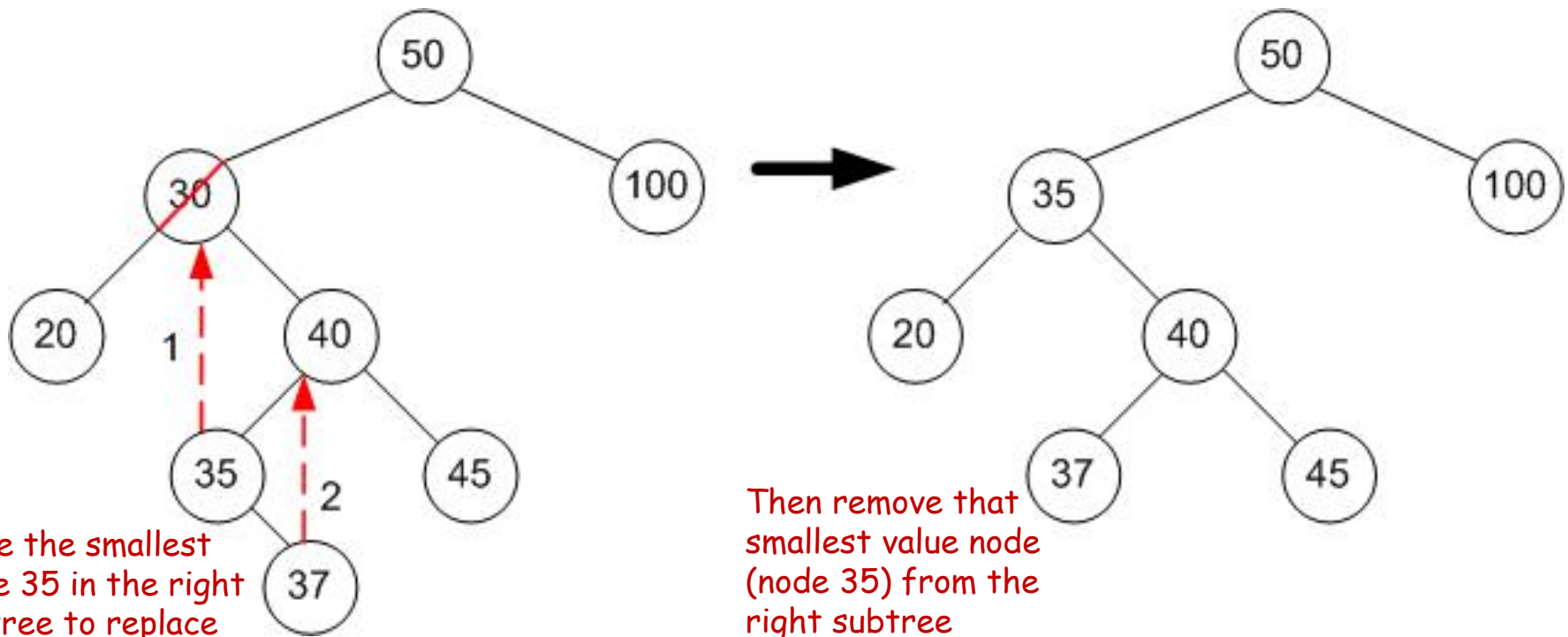
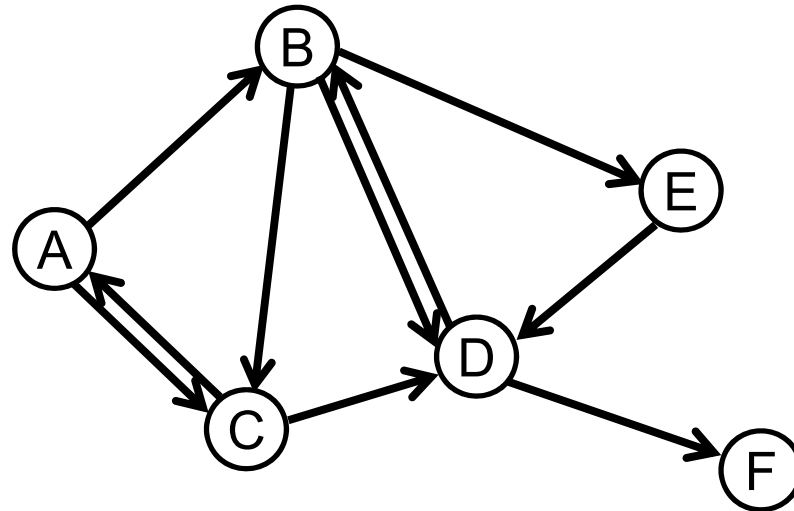


Diagram by Linuxdude at English Wikibooks - Transferred from en.wikibooks to Commons., Public Domain,
<https://commons.wikimedia.org/w/index.php?curid=61794591>

Non-linear Data Structures – Graphs

Directed Graph



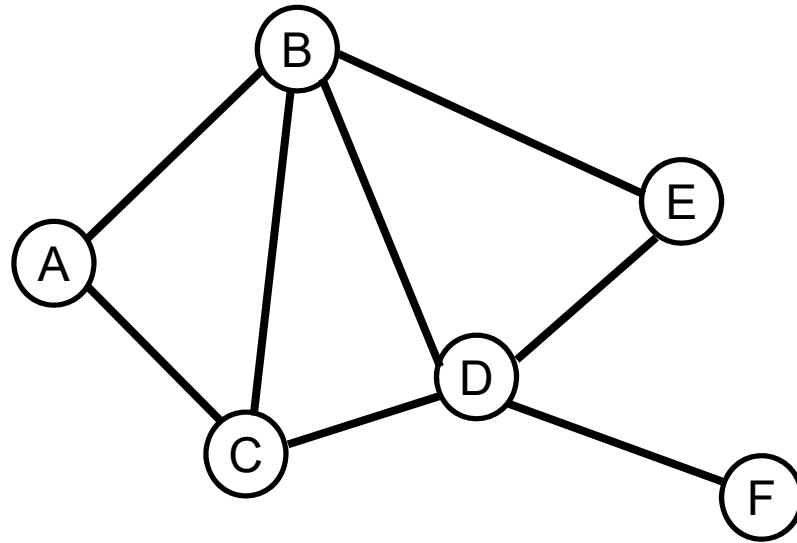
Adjacency Matrix

	A	B	C	D	E	F
from	0	1	1	0	0	0
B	0	0	1	1	1	0
C	1	0	0	1	0	0
D	0	1	0	0	0	1
E	0	0	0	1	0	0
F	0	0	0	0	0	0

Adjacency List

A	B	C	
B	C	D	E
C	D	A	
D	B	F	
E	D		
F			

Undirected Graph



Adjacency Matrix

	A	B	C	D	E	F
A	0	1	1	0	0	0
B	1	0	1	1	1	0
C	1	1	0	1	0	0
D	0	1	1	0	1	1
E	0	1	0	1	0	0
F	0	0	0	1	0	0

Adjacency List

A	B	C		
B	A	C	D	E
C	D	A	B	
D	B	E	F	C
E	D	B		
F	D			

Traversals of a Graph

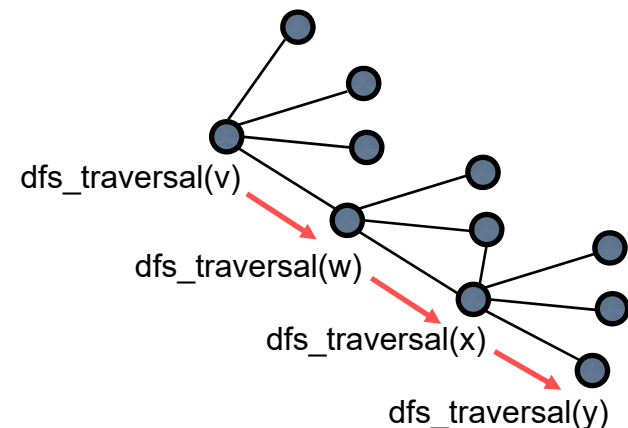
- ◆ “Walk through” the graph by visiting each vertex that can be reached from the starting vertex
- ◆ Depth-first search (DFS) - *Go as 'deep' as possible*
 - ❖ prioritizes following edges along a path
- ◆ Breadth-first search (BFS) - *Go as 'wide' as possible*
 - ❖ prioritizes following edges closest to the starting vertex

DFS vs BFS

```
def dfs_traversal(vertex):  
    visit(vertex)  
    for i in range(len(vertex.adjList)):  
        neighbor = vertex.adjList[i]  
        if not neighbor.isVisited():  
            dfs_traversal(neighbor)
```

```
def bfs_traversal(vertex):  
    q = Queue()  
    visit(vertex)  
    q.enqueue(vertex)  
    while !q.isEmpty():  
        v = q.dequeue()  
        for i in range(len(v.adjList)):  
            neighbor = v.adjList[i]  
            if not neighbor.isVisited():  
                visit(neighbor)  
                q.enqueue(neighbor)
```

```
def visit(vertex):  
    vertex.visited = True  
    print(vertex)
```




DFS vs BFS

```
def dfs_traversal(vertex):  
    visit(vertex)  
    for i in range(len(vertex.adjList)):  
        neighbor = vertex.adjList[i]  
        if not neighbor.isVisited():  
            dfs_traversal(neighbor)
```

```
def bfs_traversal(vertex):  
    q = Queue()  
    [visit(vertex)  
    q.enqueue(vertex)  
    while !q.isEmpty():  
        v = q.dequeue()  
        for i in range(len(v.adjList)):  
            neighbor = v.adjList[i]  
            if not neighbor.isVisited():  
                [visit(neighbor)  
                q.enqueue(neighbor)]
```

```
def visit(vertex):  
    vertex.visited = True  
    print(vertex)
```

q: v


bfs_traversal(v)

DFS vs BFS

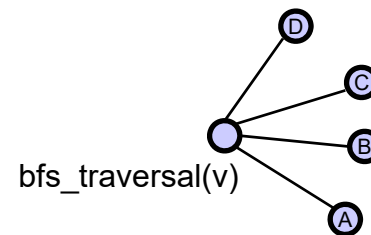
```
def dfs_traversal(vertex):  
    visit(vertex)  
    for i in range(len(vertex.adjList)):  
        neighbor = vertex.adjList[i]  
        if not neighbor.isVisited():  
            dfs_traversal(neighbor)
```

```
def bfs_traversal(vertex):  
    q = Queue()  
    [visit(vertex)  
     q.enqueue(vertex)  
     while !q.isEmpty():  
         v = q.dequeue()  
         for i in range(len(v.adjList)):  
             neighbor = v.adjList[i]  
             if not neighbor.isVisited():  
                 [visit(neighbor)  
                  q.enqueue(neighbor)]
```

```
def visit(vertex):  
    vertex.visited = True  
    print(vertex)
```

q:

A	B	C	D
---	---	---	---



DFS vs BFS

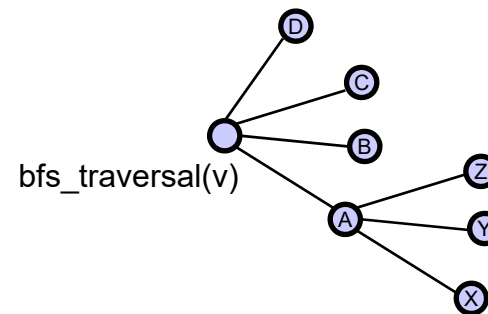
```
def dfs_traversal(vertex):  
    visit(vertex)  
    for i in range(len(vertex.adjList)):  
        neighbor = vertex.adjList[i]  
        if not neighbor.isVisited():  
            dfs_traversal(neighbor)
```

```
def bfs_traversal(vertex):  
    q = Queue()  
    [visit(vertex)  
    q.enqueue(vertex)  
    while !q.isEmpty():  
        v = q.dequeue()  
        for i in range(len(v.adjList)):  
            neighbor = v.adjList[i]  
            if not neighbor.isVisited():  
                [visit(neighbor)  
                q.enqueue(neighbor)]
```

```
def visit(vertex):  
    vertex.visited = True  
    print(vertex)
```

q:

B	C	D	X	Y	Z
---	---	---	---	---	---



DFS vs BFS

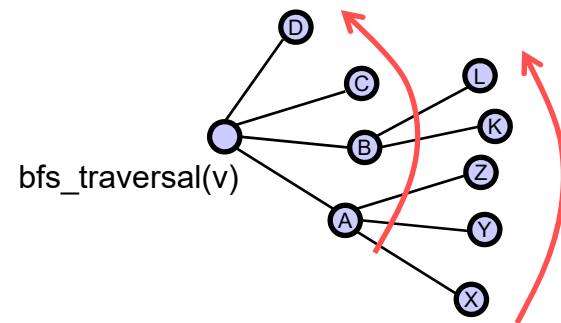
```
def dfs_traversal(vertex):  
    visit(vertex)  
    for i in range(len(vertex.adjList)):  
        neighbor = vertex.adjList[i]  
        if not neighbor.isVisited():  
            dfs_traversal(neighbor)
```

```
def bfs_traversal(vertex):  
    q = Queue()  
    [visit(vertex)  
     q.enqueue(vertex)  
     while !q.isEmpty():  
         v = q.dequeue()  
         for i in range(len(v.adjList)):  
             neighbor = v.adjList[i]  
             if not neighbor.isVisited():  
                 [visit(neighbor)  
                  q.enqueue(neighbor)]
```

```
def visit(vertex):  
    vertex.visited = True  
    print(vertex)
```

q:

C	D	X	Y	Z	K	L
---	---	---	---	---	---	---

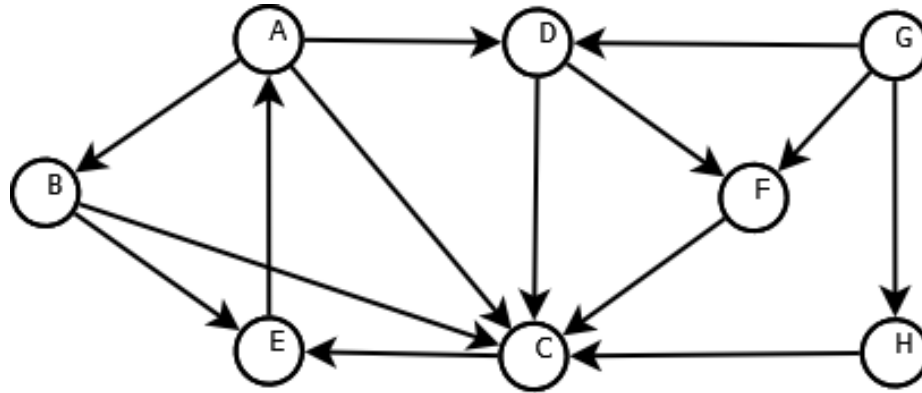


DFS (Non-recursive with Stack)

```
def dfs(vertex):  
    ① s = Stack()  
    ② s.push(vertex)  
    ③ while s.count() > 0:  
        ④ v = s.pop()  
        if not v.isVisited():  
            visit(v)  
        ⑥ for i in range(len(v.adjList)):  
            n = v.adjList[len(v.adjList) - 1 - i]  
            if not n.isVisited():  
                s.push(n)
```

To pop according to alphabetic order.

Practice DFS Traversal



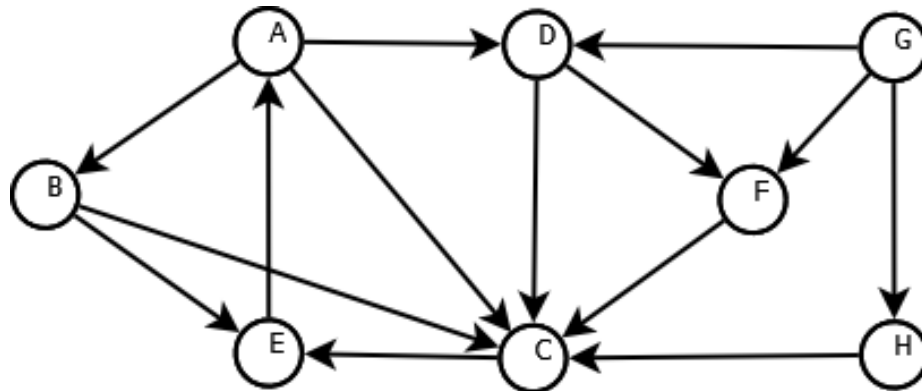
DFS traversal
starting from A:

DFS traversal
starting from G:

Adjacency List

A	B	C	D
B	C	E	
C	E		
D	C	F	
E	A		
F	C		
G	D	F	H
H	C		

Practice BFS Traversal



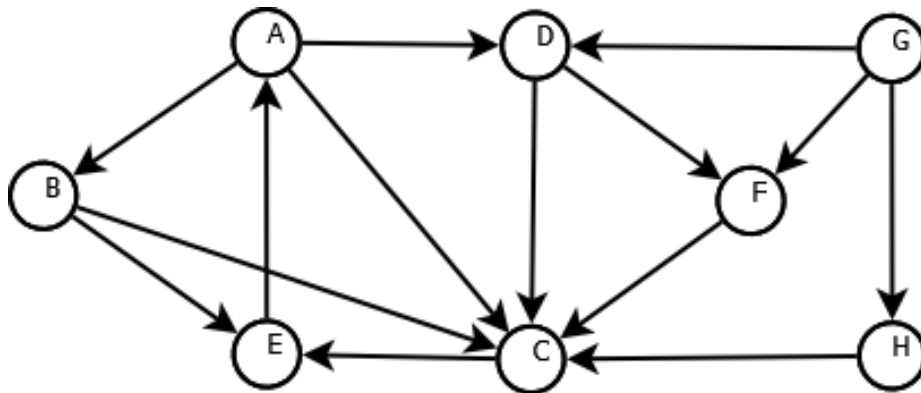
BFS traversal starting from G:

step	dequeued	visited/enqueued	queue: [head.... tail]
0			
1			
2			
3			
4			
5			
6			
7			
8			

Adjacency List

A	B	C	D
B	C	E	
C	E		
D	C	F	
E	A		
F	C		
G	D	F	H
H	C		

Practice DFS (Non-recursive with Stack) Traversal



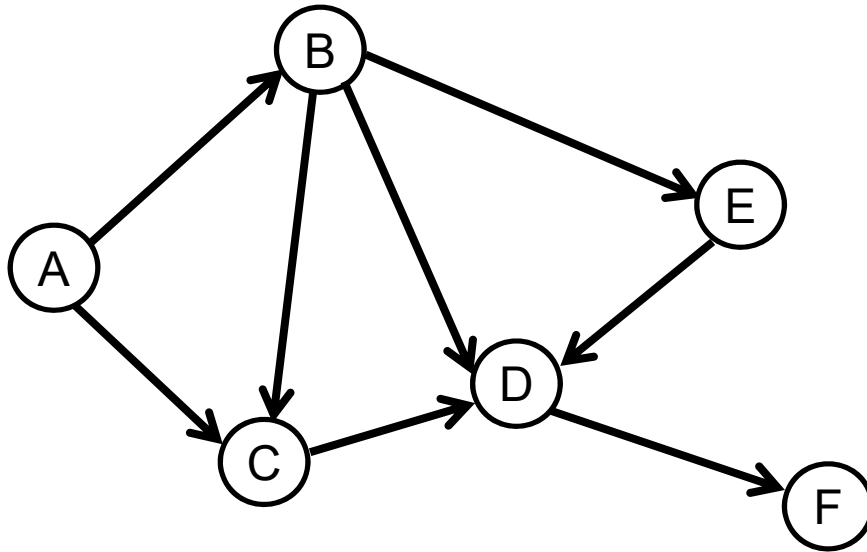
Adjacency List

A	B	C	D
B	C	E	
C	E		
D	C	F	
E	A		
F	C		
G	D	F	H
H	C		

DFS traversal starting from G:

step	popped	visited	pushed	stack: [B..T]
0				
1				
2				
3				
4				
5				
6				
7				
8				
9				

Directed Acyclic Graph (DAG)



Adjacency List

A	B	C		
B		C	D	E
C			D	
D				F
E			D	
F				

DAG does not contain any cycle.

Topological Ordering

Figure 14-14
A DAG

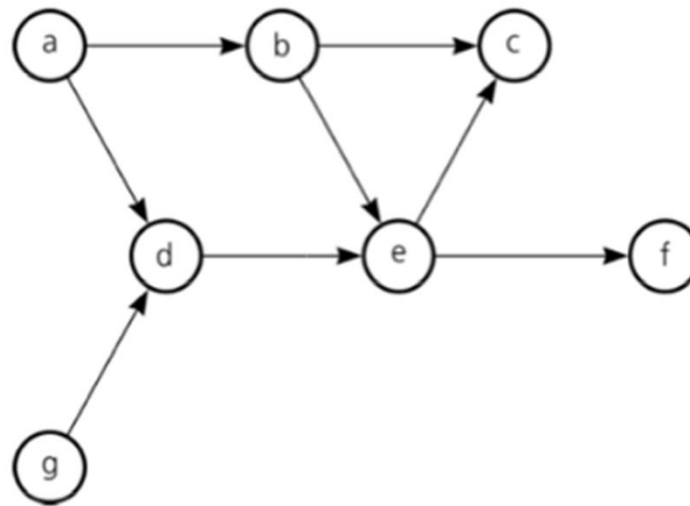
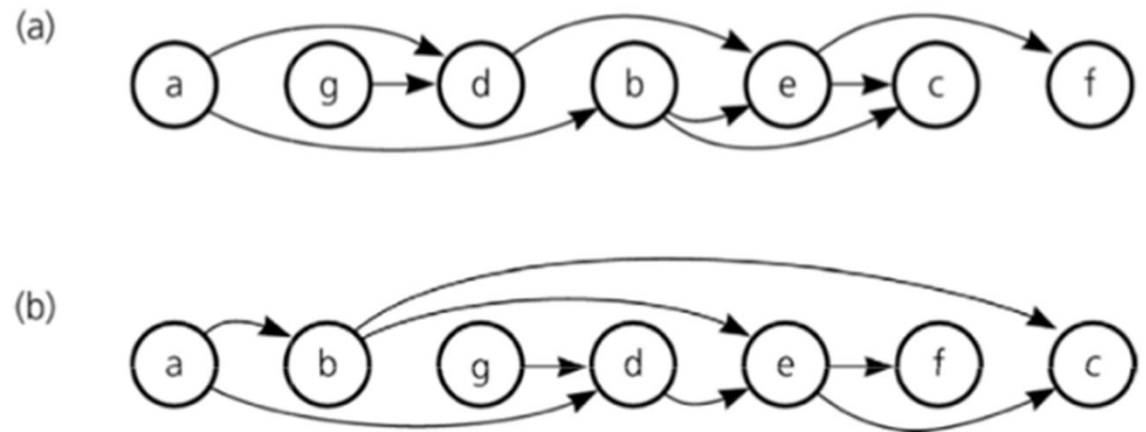


Figure 14-15

The graph in Figure 14-14 arranged according to the topological orders:

- a) *a, g, d, b, e, c, f* and
- b) *a, b, g, d, e, f, c*



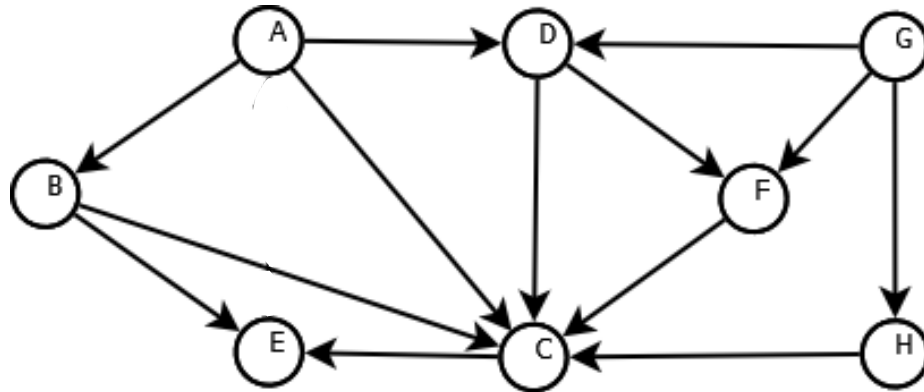
Topological Sort Algorithm

```
def topsort(graph) :  
    s = Stack()  
    for i in range(len(graph.vertices)) :  
        vi = graph.vertices[i]  
        if not vi.isVisited():  
            topsort_dfs(vi, s)  
    return s
```

```
def topsort_dfs(vertex, stack) :  
    visit(vertex)  
    for i in range(len(vertex.adjList)) :  
        neighbor = vertex.adjList[i]  
        if not neighbor.isVisited():  
            topsort_dfs(neighbor, stack)  
    stack.push(vertex)
```

- ◆ Run DFS from every vertex to ensure that all vertices are included in a topological ordering.
- ◆ Add vertex into a stack at the end of DFS traversal on this vertex. Vertices at the end of a path will be pushed first into the stack.
- ◆ When a vertex is pushed into the stack, all vertices depending on it are already in the stack.

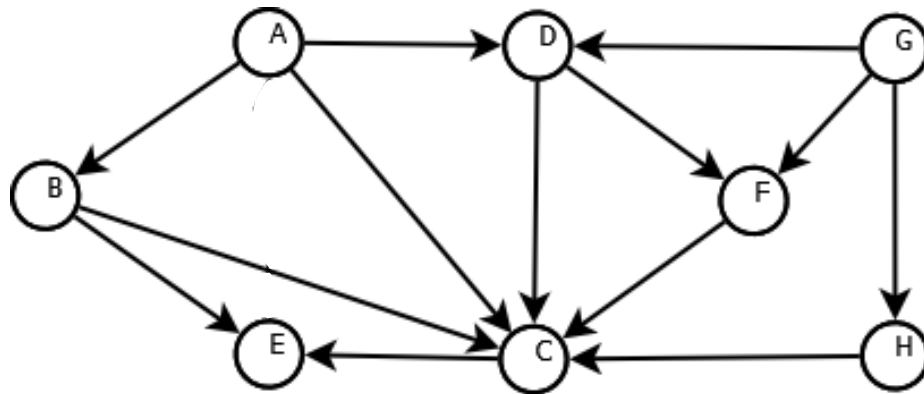
Practice Topological Sorting



Adjacency List

A	B	C	D
B	C	E	
C	E		
D	C	F	
E			
F	C		
G	D	F	H
H	C		

Practice Topological Sorting



DFS traversal
starting from A:

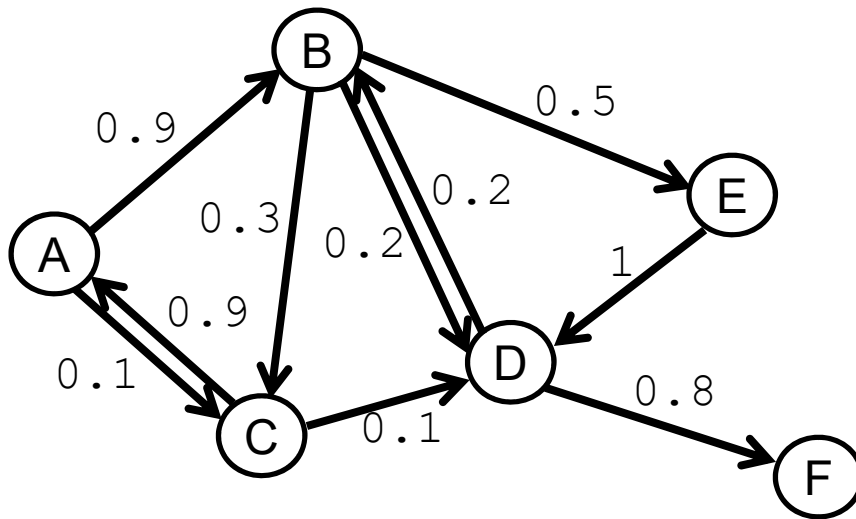
DFS traversal
starting from G:

stack

Adjacency List

A	B	C	D
B	C	E	
C	E		
D	C	F	
E			
F	C		
G	D	F	H
H	C		

Weighted Graph & Shortest Path Finding



Adjacency List

A	(B, 0.9)	(C, 0.1)
B	(C, 0.3)	(D, 0.2) (E, 0.5)
C	(D, 0.1)	(A, 0.9)
D	(B, 0.2)	(F, 0.8)
E	(D, 1)	
F		

Adjacency Matrix

	A	B	C	D	E	F
A	∞	0.9	0.1	∞	∞	∞
B	∞	∞	0.3	0.2	0.5	∞
C	0.9	∞	∞	0.1	∞	∞
D	∞	0.2	∞	∞	∞	0.8
E	∞	∞	∞	1	∞	∞
F	∞	∞	∞	∞	∞	∞

Complexity of DFS/BFS

Adjacency List

A	B	C			
B	C	D	E		
C	D	A			
D	B	F			
E	D				
F					

For directed graph,
 $O(V) + O(E)$
 $= O(V+E)$

For undirected graph,
 $O(V) + O(2E)$
 $= O(V+E)$

<https://www.quora.com/Why-is-the-complexity-of-DFS-O-V+E>

Quora Home Answer Spaces Notifications Search

Originally Answered: Why is the complexity of DFS $O(V+E)$?

Say, you have a connected graph with V nodes and E edges.

In DFS, you traverse each node exactly once. Therefore, the time complexity of DFS is at least $O(V)$.

Now, any additional complexity comes from how you discover all the outgoing paths or edges for each node which, in turn, is dependent on the way your graph is implemented. If an edge leads you to a node that has already been traversed, you skip it and check the next. Typical DFS implementations use a hash table to maintain the list of traversed nodes so that you could find out if a node has been encountered before in $O(1)$ time (constant time).

- If your graph is implemented as an adjacency matrix (a $V \times V$ array), then, for each node, you have to traverse an entire row of length V in the matrix to discover all its outgoing edges. Please note that each row in an adjacency matrix corresponds to a node in the graph, and the said row stores information about edges stemming from the node. So, the complexity of DFS is $O(V * V) = O(V^2)$.
- If your graph is implemented using adjacency lists, wherein each node maintains a list of all its adjacent edges, then, for each node, you could discover all its neighbors by traversing its adjacency list just once in linear time. For a directed graph, the sum of the sizes of the adjacency lists of all the nodes is E (total number of edges). So, the complexity of DFS is $O(V) + O(E) = O(V + E)$.
 - For an undirected graph, each edge will appear twice. Once in the adjacency list of either end of the edge. So, the overall complexity will be $O(V) + O(2E) \sim O(V + E)$.

Adjacency Matrix

	A	B	C	D	E	F
A	0	1	1	0	0	0
B	0	0	1	1	1	0
C	1	0	0	1	0	0
D	0	1	0	0	0	1
E	0	0	0	1	0	0
F	0	0	0	0	0	0

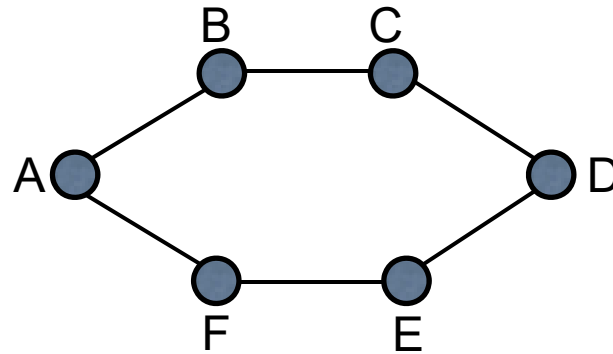
For both directed and undirected graph,
 $O(V * V)$
 $= O(V^2)$

Heuristic Reasoning

Optimization Problem

- ♦ An optimization problem is associated with an objective function
 - ❖ For any solution, the objective function evaluates a *value*
 - ❖ The value indicates the goodness of this solution
 - ❖ The goal is to obtain the solution with as optimal value as possible
- ♦ Example: Traveling Salesman Problem
 - ❖ Objective function computes the distance covered by a route
 - ❖ The goal is to find a route with as short distance as possible

Strategies



A B C D E F A
B C D E F A B
C D E F A B C
D E F A B C D
E F A B C D E
F A B C D E F

} same
tour

♦ Exhaustive search

- ❖ consider all possible tours, resulting in true optimal value
- ❖ total number of tours to examine: $(n - 1)! / 2$
- ❖ complexity is $O(n!)$

A B C D E F A
A F E D C B A

} same
tour

♦ Random search

- ❖ consider some number k of randomly chosen tours
- ❖ complexity is $O(kn)$
- ❖ depending on chance, the results may get close or far from the optimal value

Heuristic Methods

- ♦ Main idea:
 - ❖ make a series of decisions in sequential steps
 - ❖ at every step, make the best decision for the current step
 - ❖ note that the local decisions do not guarantee best global solution
- ♦ **Greedy algorithm** - *Greedy 1 algorithm, Greedy 2 algorithm*
 - ❖ construct the solution step by step
- ♦ **Local search algorithm** - *2-Opt algorithm*
 - ❖ start with a feasible solution
 - ❖ at every step, make a small modification to the solution
- ♦ Proof of performance bound (how close we get to the optimal value) is important in the theory of computer science, but is not the focus of this course.

1st Greedy Algorithm for TSP

♦ Pseudocode:

- ❖ let S be set of cities to be included in the tour
- ❖ let T be the initial tour of 1 city picked randomly
- ❖ remove this city from S
- ❖ while there are still cities in S :
 - ▶ find the city x in S with the smallest distance to the last selected city y
 - ▶ remove x from S

2nd Greedy Algorithm for TSP

♦ Pseudocode:

1. let S be set of cities to be included in the tour
2. let T be the initial tour of 2 cities with closest distance
3. remove these 2 cities from S
4. while there are still cities in S :
 - 4a. find the city x in S with the smallest distance to any city y in the current tour T
 - 4b. choose the one with lower overall distance:
 - insert x between y and its next destination z in T
 - insert x between y and its previous destination u in T
 - 4c. remove x from S

Practice 1st and 2nd Greedy algorithm

Distance Matrix

	A	B	C	D	E	F
A	0	16	47	72	77	79
B	16	0	37	57	65	66
C	47	37	0	40	30	35
D	72	57	40	0	31	23
E	77	65	30	31	0	10
F	79	66	35	23	10	0

1st Greedy algorithm with starting vertex A:

2nd Greedy algorithm:

Step	Visited	Unvisited	Tour	Distance
0				
1				
2				
3				
4				

Algorithm Efficiency & Problem Tractability

- ♦ Efficient vs. inefficient algorithm
 - ❖ An algorithm is considered “efficient” if it has polynomial time complexity or lower, i.e., $O(n^k)$
 - ❖ An algorithm with exponential $O(k^n)$ or factorial $O(n!)$ complexity is considered inefficient
- ♦ Tractable vs. intractable problems
 - ❖ A problem is considered tractable or solvable if there is a known solution with polynomial complexity or lower.
 - ❖ Otherwise, the problem is considered intractable (NP-hard)
 - ❖ Searching for “Needle in a haystack” (recall M. Sipser’s video)

Classes of Problems

♦ P

- ❖ has a solution that runs within polynomial time

♦ NP

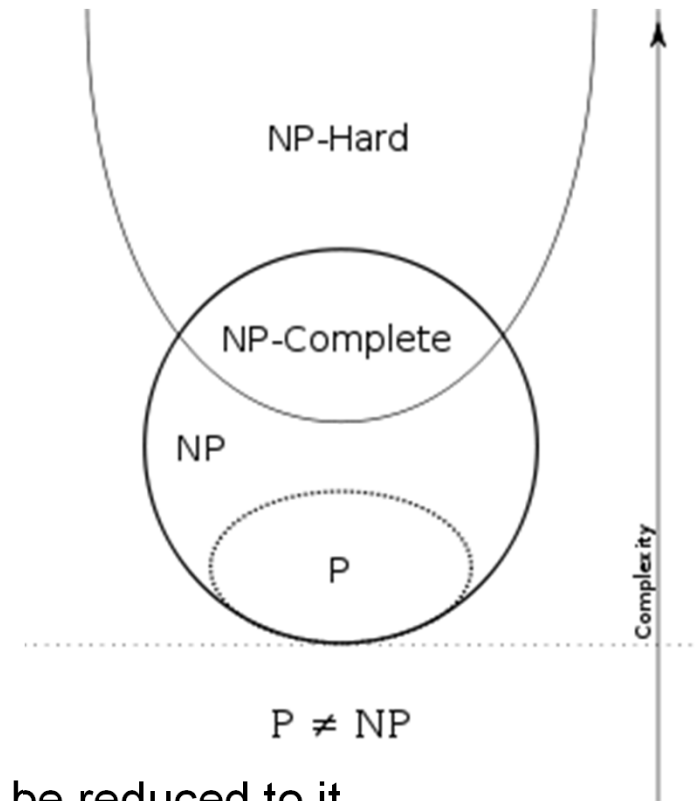
- ❖ can be verified within polynomial time

♦ NP-Hard

- ❖ at least as hard as NP, i.e. all NP problems can be reduced to it
- ❖ may not be verifiable within polynomial time

♦ NP-Complete

- ❖ NP-Hard problems that in NP, i.e. verifiable within polynomial time
- ❖ hardest problems in NP



Why is P vs. NP important?

♦ If $P = NP$

- ❖ Problems that we used to think are hard suddenly seem “solvable”.
- ❖ Good:
 - ▶ There is hope that we can obtain optimal solutions to traveling salesman and other problems in polynomial time.
- ❖ Bad:
 - ▶ Some cryptography depends on the assumption that solving some mathematical operations are extremely hard, while checking them are easy.

♦ If $P \neq NP$

- ❖ Focus on finding heuristic solutions.
- ♦ Most computer scientists “believe” that $P \neq NP$, but it has not been conclusively proven yet.

Where Do We Go from Here?

- ♦ SIS Courses:
 - ❖ **CS426: Agent-based Modeling & Simulation (by Cheng Shih-Fen), Term 2**
 - ▶ agent-based modeling and simulation as a methodology for modeling and analyzing complex business environments and operations.
 - ❖ **CS423 : Heuristic Search & Optimization (by Arunesh), Term 2**
 - ▶ advanced heuristics algorithms
 - ❖ **CS201 : Data Structures & Algorithms (by Hady), Term 1**
 - ▶ follow-up course from CT with more depth in most topics
- ♦ Recommended free online courses (from Princeton University):
 - ❖ <https://www.coursera.org/learn/algorithms-part1>
 - ❖ <https://www.coursera.org/learn/algorithms-part2>