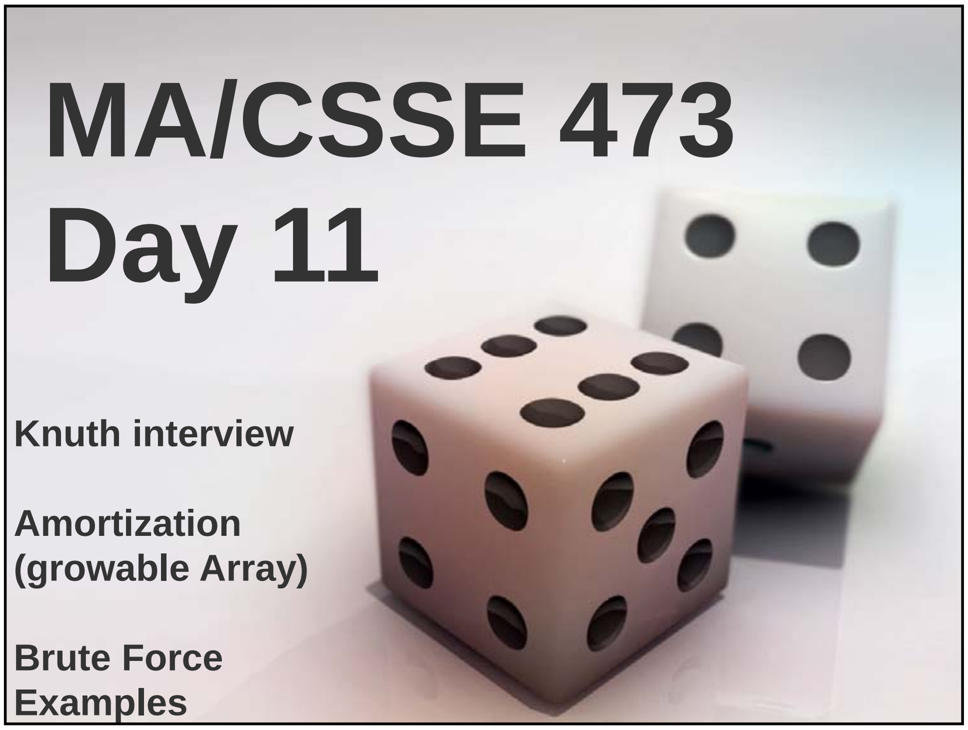
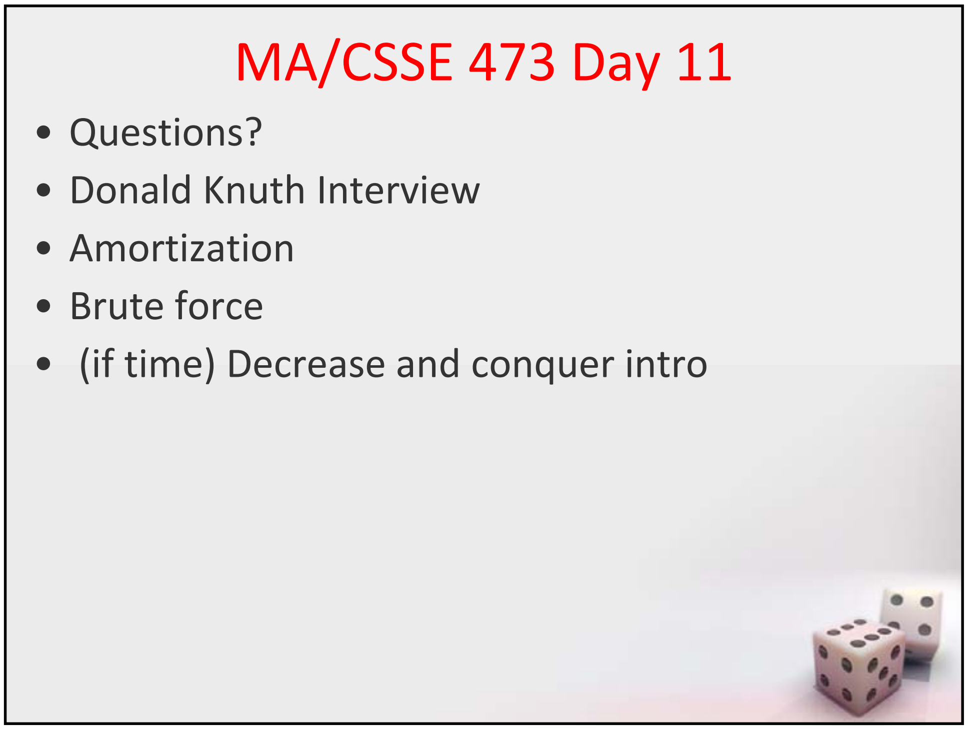


MA/CSSE 473 Day 11

Knuth interview

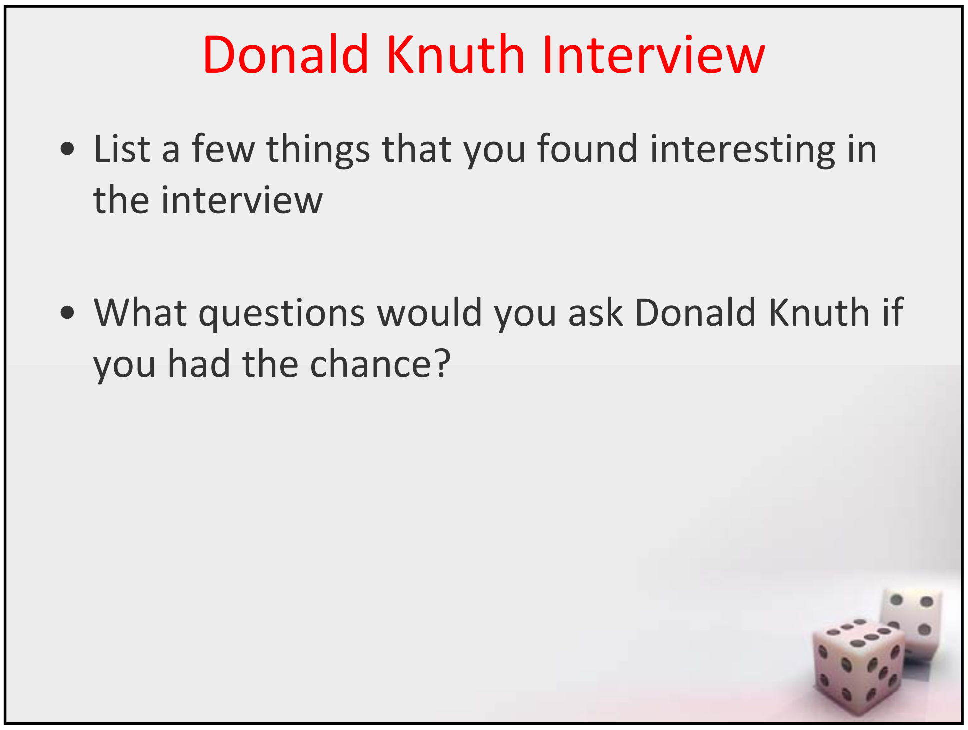
Amortization (growable Array)

Brute Force Examples



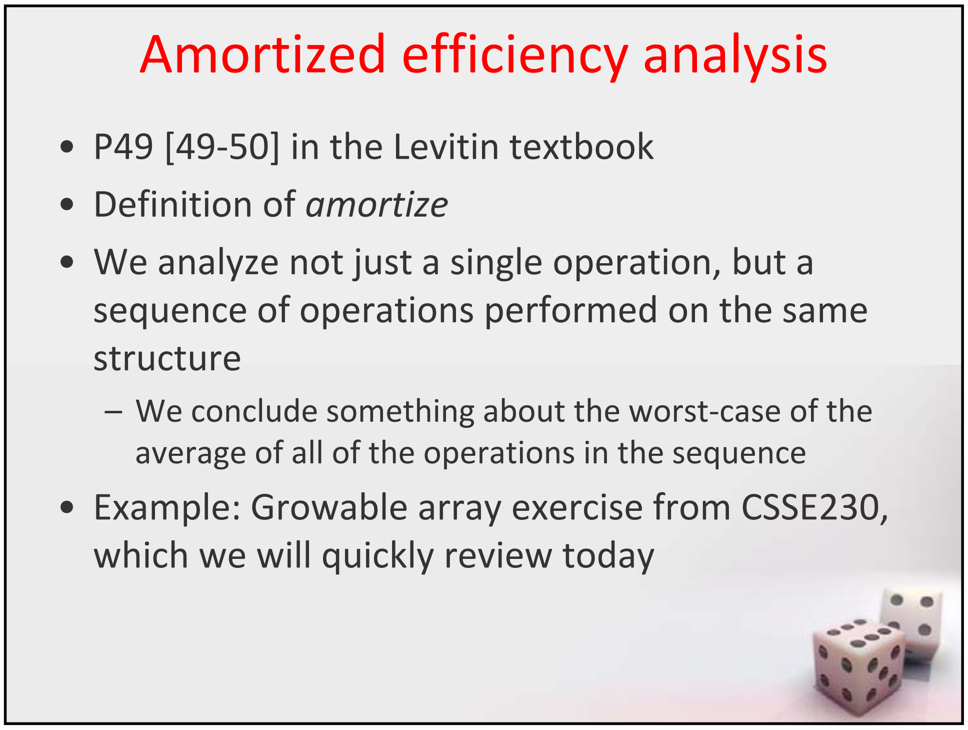
MA/CSSE 473 Day 11

- Questions?
- Donald Knuth Interview
- Amortization
- Brute force
- (if time) Decrease and conquer intro



Donald Knuth Interview

- List a few things that you found interesting in the interview
- What questions would you ask Donald Knuth if you had the chance?



Amortized efficiency analysis

- P49 [49-50] in the Levitin textbook
- Definition of *amortize*
- We analyze not just a single operation, but a sequence of operations performed on the same structure
  - We conclude something about the worst-case of the average of all of the operations in the sequence
- Example: Growable array exercise from CSSE230, which we will quickly review today

Growable Array (implement ArrayList)

- An ArrayList has a *size* and a *capacity*
- The capacity is the length of the fixed-size array currently allocated to hold the list elements
- For definiteness, we start with *size*=0 and *capacity*=5
- We add a total of N items (N is not known in advance), one at a time, each to the end of the structure
- When there is no room in the array (i.e. capacity=size and we need to add another element)
  - Allocate a new, larger array
  - copy the *size* existing elements to the new array
  - add the new element to the new array

Growable Array (implement ArrayList)

- When there is no room in the array (i.e. capacity=size and we need to add another element)
  - Allocate a new, larger array
  - copy the *size* existing elements to the new array
  - add the new element to the new array
- What is the total/average overhead (due to element copying) if
  a. we add one to the array capacity each time we have to grow it?
  b. we double the array capacity each time we have to grow it?
- Note in the second case that the amortized worst-case cost is asymptotically less than the worst case for a single element
- Every time we have to enlarge the capacity, we make it so we do not have to enlarge again soon.

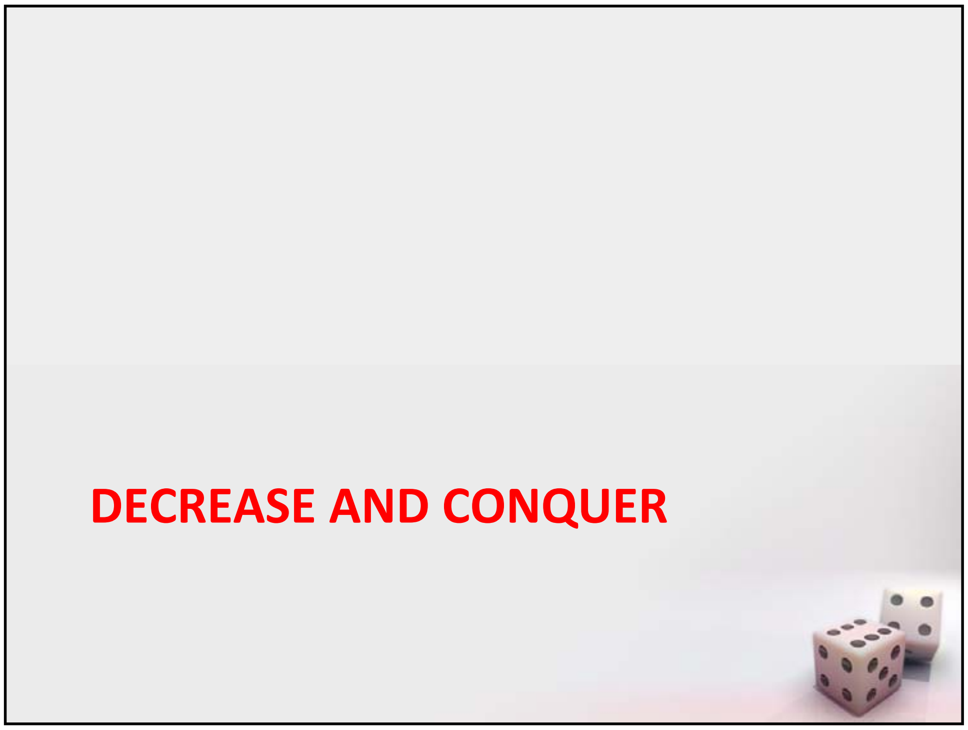
Brute Force Algorithms

- Straightforward, simple, not subtle, usually a simple application of the problem definition.
- Often not very efficient
- Easy to implement, so often the best choice if you know you'll only apply it to small input sizes

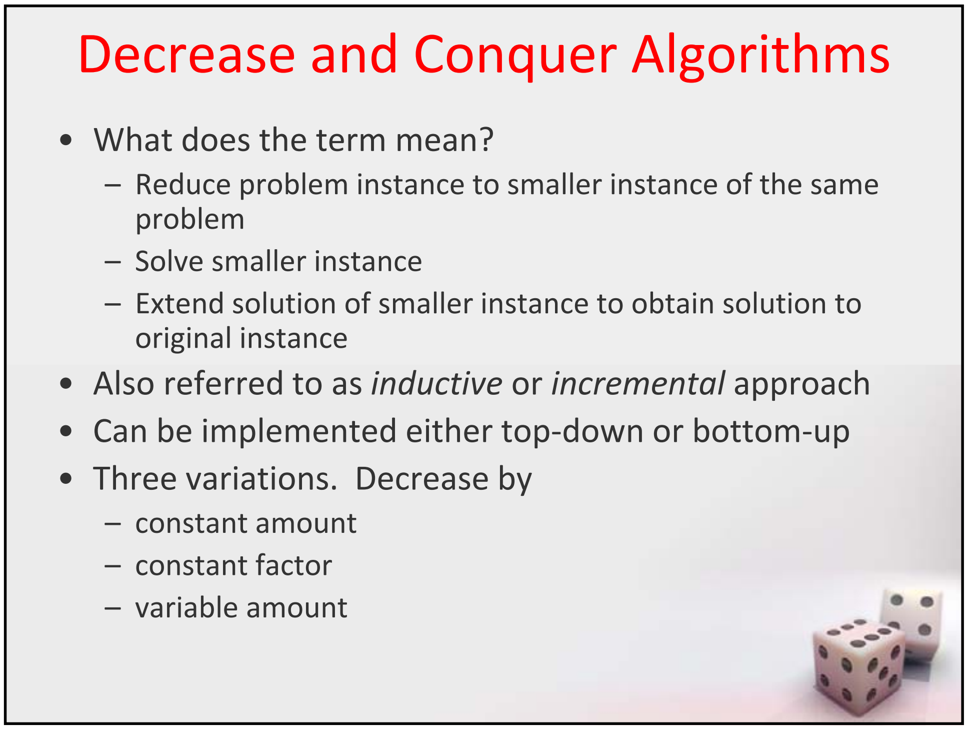
3-4

What is a brute force approach to

1. Calculate the n^{th} Fibonacci number?
2. Compute the n^{th} power of an integer?
3. Search for a particular value in a sorted array?
4. Sort an array?
5. Search for a substring of a string?
6. Find the maximum contiguous subsequence in an array of integers?
7. Find the largest element in a Binary Search Tree?
8. Find the two closest points among N points in the plane?
9. Find the convex hull of a set of points in the plane?
10. Find the shortest path from vertex A to vertex B in a weighted graph?
11. Solve the traveling salesman problem?
12. Solve the knapsack problem?
13. Solve the assignment problem?
14. Solve the $n \times n$ non-attacking chess queens problem?
15. Other problems that you can think of?



DECREASE AND CONQUER



Decrease and Conquer Algorithms

- What does the term mean?
  - Reduce problem instance to smaller instance of the same problem
  - Solve smaller instance
  - Extend solution of smaller instance to obtain solution to original instance
- Also referred to as *inductive* or *incremental* approach
- Can be implemented either top-down or bottom-up
- Three variations. Decrease by
  - constant amount
  - constant factor
  - variable amount

Decrease by constant *vs* by half

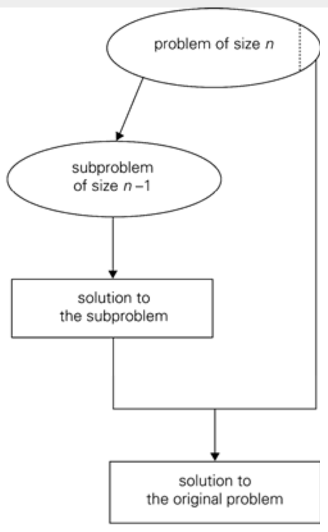


FIGURE 5.1 Decrease (by one)-and-conquer technique

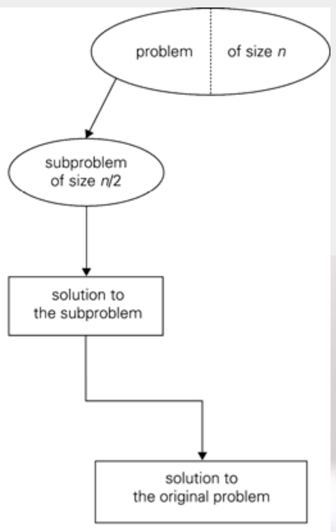


FIGURE 5.2 Decrease (by half)-and-conquer technique

One Problem, Four approaches

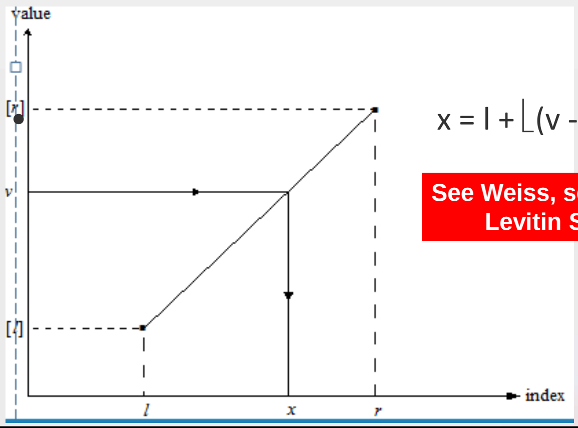
- Recall the problem of integer exponentiation: Compute a^n*, where n is a power of 2:*
  - Brute Force: $a^n = a*a*a*a*...*a$
  - Divide and conquer: $a^n = a^{n/2} * a^{n/2}$
  - Decrease by one: $a^n = a^{n-1} * a$
  - Decrease by constant factor: $a^n = (a^{n/2})^2$

Variable Decrease Examples

- Euclid's algorithm
  - **b** and **a % b** are smaller than **a** and **b**, but not by a constant amount or constant factor
- Interpolation search
  - The two sides of the partitioning element are smaller than n, but can be anything from 0 to n-1.

Interpolation Search

- Searches a sorted array similar to binary search but estimates location of the search key in A[l..r] by using its value v.
- Specifically, the values of the array’s elements are assumed to increase linearly from A[l] to A[r]
- Location of v is estimated as the x-coordinate of the point on the straight line through (l, A[l]) and (r, A[r]) whose y-coordinate is v:



$$x = l + \lfloor (v - A[l])(r - l)/(A[r] - A[l]) \rfloor$$

See Weiss, section 5.6.3
Levitin Section 4.5 [5.6]

Interpolation Search Running time

- Average case: $\Theta(\log(\log n))$ Worst: $\Theta(n)$
- What can lead to worst-case behavior?
  - Social Security numbers of US residents
  - Phone book (Wilkes-Barre)
  - CSSE department employees*, 1984-2017

adkins	**defoe**	**mutchler**
anderson	dalkolic	oexmann
ardis	**degler**	**rupakheti**
azhar	**fisher**	sengupta
bagert	**galluzzi**	shillingford
baker	**hays**	srivastava
bohner	jeschke	**stamm**
bowman	kaczmarczyk	**stouder**
boutell	kinley	sullivan
chenette	**laxer**	surendran
chenoweth	lo	**taylor**
chidanandan	mcleish	**wilkin**
clifton	**mellor**	**wollowski**
criss	merkle	young
cultur	**mohan**	
curry	**mouck**	
19/46: A-D	29/46: A-D, K-M	36/46: A-D, K-M, S

***Red** and **blue** are current employees

Some "decrease by one" algorithms

- Insertion sort
- Selection Sort
- Depth-first search of a graph
- Breadth-first search of a graph

Review: Analysis of Insertion Sort

- Time efficiency

  $C_{worst}(n) = n(n\text{-}1)/2 \in \Theta(n^2)$

  $C_{avg}(n) \approx n^2/4 \in \Theta(n^2)$

  $C_{best}(n) = n - 1 \in \Theta(n)$

  (also fast on almost-sorted arrays)
- Space efficiency: in-place

  (constant extra storage)
- Stable: yes
- Binary insertion sort
  - use Binary search, then move elements to make room for inserted element

Graph Traversal

Many problems require processing all graph vertices (and edges) in systematic fashion

Most common Graph traversal algorithms:

- Depth-first search (DFS)
- Breadth-first search (BFS)

Depth-First Search (DFS)

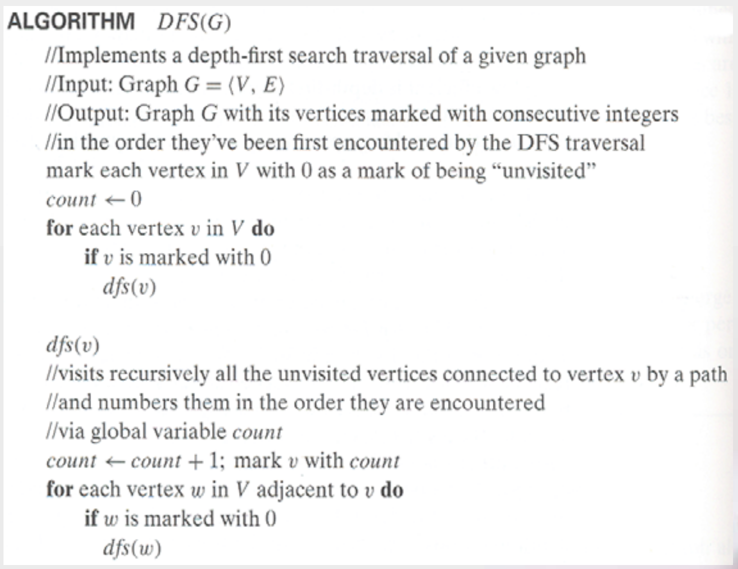
- Visits a graph’s vertices by always moving from last visited vertex to unvisited one, backtracks if there is no adjacent unvisited vertex is available
- Uses a stack (or could use recursion)
  - a vertex is pushed onto the stack when it’s reached for the first time
  - a vertex is popped off the stack when it becomes a dead end, i.e., when there are no adjacent unvisited vertices
- “Redraws” graph in tree-like fashion (with tree edges and back edges for undirected graph)

  –A back edge is an edge of the graph that goes from the current vertex to a previously visited vertex (other than the current vertex's parent in the tree).

Notes on DFS

- DFS can be implemented with graphs represented as:
  - adjacency matrix: $\Theta(V^2)$
  - adjacency list: $\Theta(|V|+|E|)$
- Yields two distinct ordering of vertices:
  - order in which vertices are first encountered (pushed onto stack)
  - order in which vertices become dead-ends (popped off stack)
- Applications:
  - checking connectivity, finding connected components
  - checking acyclicity
  - finding articulation points
  - searching state-space of problems for solution (AI)

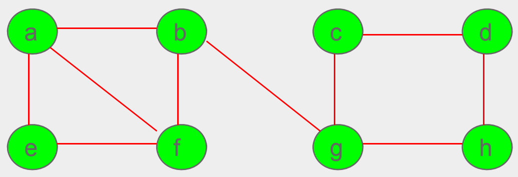
Pseudocode for DFS



```
ALGORITHM  DFS(G)
    //Implements a depth-first search traversal of a given graph
    //Input: Graph $G = \langle V, E \rangle$
    //Output: Graph $G$ with its vertices marked with consecutive integers
    //in the order they've been first encountered by the DFS traversal
    mark each vertex in $V$ with 0 as a mark of being "unvisited"
    $count \leftarrow 0$
    for each vertex $v$ in $V$ do
        if $v$ is marked with 0
            $dfs(v)$

    $dfs(v)$
    //visits recursively all the unvisited vertices connected to vertex $v$ by a path
    //and numbers them in the order they are encountered
    //via global variable $count$
    $count \leftarrow count + 1$; mark $v$ with $count$
    for each vertex $w$ in $V$ adjacent to $v$ do
        if $w$ is marked with 0
            $dfs(w)$
```

Example: DFS traversal of undirected graph



DFS traversal stack:

DFS tree: