Brute Force

A straightforward approach, usually based directly on the problem's statement and definitions of the concepts involved

Simple Examples:

1. Computing a^n ($a > 0$, n a nonnegative integer)
2. Computing n!
3. Multiplying two matrices
4. Searching for a given value in a list
5. Naïve sorting algorithms
   e.g. Selection Sort

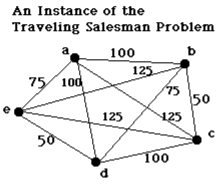
Exhaustive Search

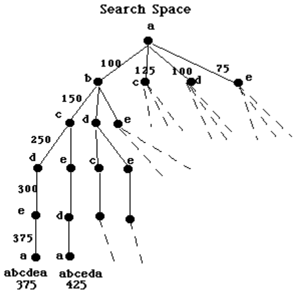
- A brute force solution to a problem involving search for an element with a special property.

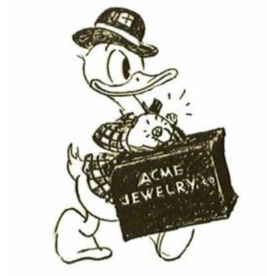
- Method:
  - Generate a list of all potential solutions to the problem in a systematic manner.
  - Evaluate potential solutions one by one disqualifying infeasible ones and, for an optimization problem, keeping track of the best one found so far.
  - When search ends, announce the solution(s) found.

Example 1: Traveling Salesman Problem

- Given *n* cities with known distances between each pair, find the shortest tour that passes through all the cities exactly once before returning to the starting city

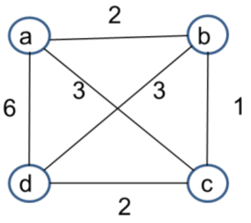






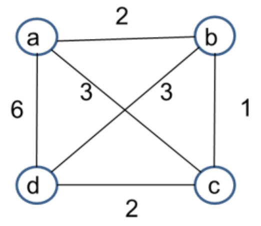
Example 1: Traveling Salesman Problem

- Given *n* cities with known distances between each pair, find the shortest tour that passes through all the cities exactly once before returning to the starting city
- Alternatively: Find shortest *Hamiltonian circuit* in a weighted connected graph
- Example:



Example 1: Traveling Salesman Problem

- Nearest Neighbor (greedy) does not guarantee an optimal solution!



TSP by Exhaustive Search

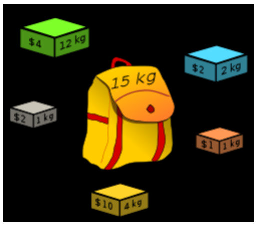
Tour	Cost
a→b→c→d→a	2+3+7+5 = 17
a→b→d→c→a	2+4+7+8 = 21
a→c→b→d→a	8+3+4+5 = 20
a→c→d→b→a	8+7+4+2 = 21
a→d→b→c→a	5+4+3+8 = 20
a→d→c→b→a	5+7+3+2 = 17

How many **distinct** tours? 5 cities? N cities?

Example 2: 0-1 Knapsack Problem

- Given n items:
  - weights: w1 w2 … wn
  - values: v1 v2 … vn
  - a knapsack of capacity W
- Find most valuable subset of the items that fit into the knapsack
- Example: Knapsack capacity W=16

item	weight	value
1	2	$20
2	5	$30
3	10	$50
4	5	$10



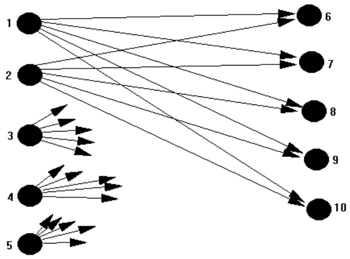
Knapsack Problem by Exhaustive Search

Subset	Total weight	Total value
{1}	2	$20
{2}	5	$30
{3}	10	$50
{4}	5	$10
{1,2}	7	$50
{1,3}	12	$70
{1,4}	7	$30
{2,3}	15	$80
{2,4}	10	$40
{3,4}	15	$60
{1,2,3}	17	not feasible
{1,2,4}	12	$60
{1,3,4}	17	not feasible
{2,3,4}	20	not feasible
{1,2,3,4}	22	not feasible

Example 3: The Assignment Problem

There are n people who need to be assigned to n jobs, one person per job. The cost of assigning person i to job j is C[i,j]. Find an assignment that minimizes the total cost.

	Job 0	Job 1	Job 2	Job 3
Person 0	9	2	7	8
Person 1	6	4	3	7
Person 2	5	8	1	8
Person 3	7	6	9	4



How many assignments are there?

Assignment Problem by Exhaustive Search

$$C = \begin{matrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{matrix}$$

Assignment (col.#s)	Total Cost
1, 2, 3, 4	9+4+1+4=18
1, 2, 4, 3	9+4+8+9=30
1, 3, 2, 4	9+3+8+4=24
1, 3, 4, 2	9+3+8+6=26
1, 4, 2, 3	9+7+8+9=33
1, 4, 3, 2	9+7+1+6=23
	etc.

Final Comments on Exhaustive Search

- Exhaustive-search algorithms run in a realistic amount of time only on very small instances
- In some cases, there are much better alternatives!
  - Euler circuits
  - shortest paths
  - minimum spanning tree
  - assignment problem
- In many cases, exhaustive search or its variation is the only known way to get exact solution

Combinatorial Objects: Generating Permutations and Subsets

- Conceptually the easiest way to think about these objects is recursively
- This leads to relatively straightforward implementations.

Generating Permutations

- Think top down: Suppose you can generate permutations of any set of (n-1)-elements.
- The given a set of n-elements use the black box on the n-1 elements.
- Given {a,b,c} how would this work to generate the permutations in lexicographic order? Write the pseudo code.

Generating Subsets - Bitstrings

- Think top down: Suppose you can generate all bitstrings of length (n-1).
- Use the black box on the bitstrings of length (n-1) to get the bitstrings of length n.
- How? Write the pseudo code.

Final Comments on Exhaustive Search

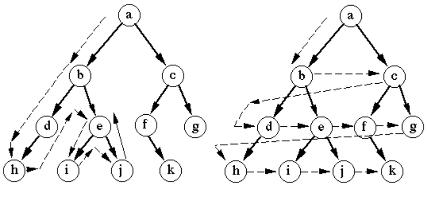
- Exhaustive-search algorithms run in a realistic amount of time only on very small instances
- In some cases, there are much better alternatives!
  - Euler circuits
  - shortest paths
  - minimum spanning tree
  - assignment problem
- In many cases, exhaustive search or its variation is the only known way to get exact solution

Graphs

- G=(V,E) V= Vertex Set; E= Edge Set
- Directed: edges ordered pairs undirected: unordered pairs
- Path: sequence of vertices v_0, v_1, …, v_k
  where for i=1,…k, v_i-1 is adjacent to v_i (or sequence of edges)
- Simple path: no vertex or edge repeated except possibly the first
- Cycle: path where $v_0 = v_k$ k>1
- G is connected if there is a path between every pair of vertices
- Connected Component of G is a subgraph of G that is connected
- Adjacency Matrix → entry (i,j) = 1 if there is an edge=(i,j) else 0
- Adjacency List → each vertex has a list of the adjacent vertices

Graph Traversal Algorithms

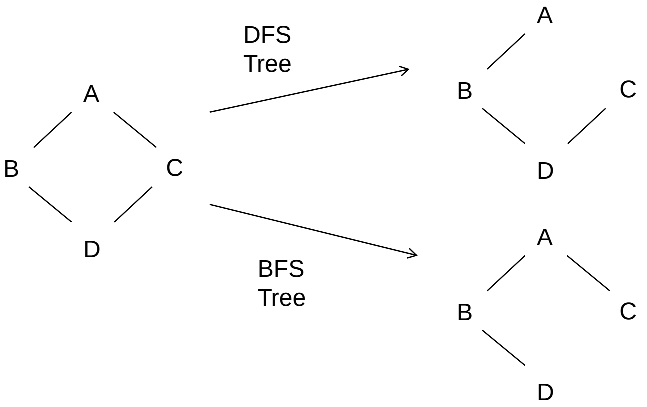
- Many problems require processing all graph vertices (and edges) in systematic fashion.
- Computer representation of graphs contains little direct information about the overall structure.
- Graph traversal algorithms:
  - Depth-first search (DFS)
    - explore deep first
    - Stack -- LIFO
  - Breadth-first search (BFS)
    - explore all “neighbors” first
    - Queue – FIFO
- If G is a tree then preorder, inorder, and postorder are all depth first search algorithms!



Depth-first search

Breadth-first search

Example: DFS vs BFS “spanning trees”



Depth-First Search (DFS)

- Visits graph’s vertices by always moving away from last visited vertex to unvisited one, backtracks if no adjacent unvisited vertex is available.
- Uses a stack (explicitly or implicitly)
  - 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 is no adjacent unvisited vertex
- “Redraws” graph in tree-like fashion (with tree edges and back edges for undirected graph)
  - *Tree edge – an edge that is traversed by the algorithm*
  - *Back edge – an edge in the original graph that is not part of the tree that is traversed in visiting all the nodes of the graph. Edges that lead to vertices that have already been visited!*

Depth First Search Algorithm

- Recursive marking algorithm
- Initially every vertex is unmarked

```
DFS(i: vertex)
  mark i as visited;
  for each j adjacent to i do
    if j is unmarked then DFS(j)
end{DFS}
```

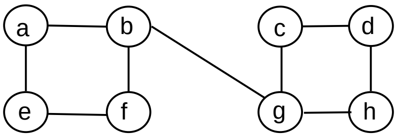
Marks all vertices reachable from i

DFS

```
DFS(G)
        Initialize cnt = 0 and visited(v) = false
        for all v ∊ V
                if not visited(v) then dfs(v)
dfs(v)
        visited(v) = true
        previsit(v); cnt = cnt + 1; pre(v)= cnt;
        for each edge (v,u) ∊ E
                if not visited(u) then dfs(u)
        postvisit(v); cnt = cnt + 1; post(v) = cnt
```

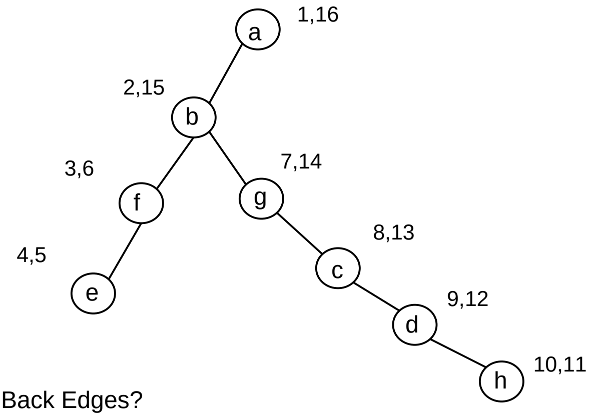
- The order of edges is usually determined by the data structure.
- Our convention when doing by hand, order is by the order of the symbols, e.g. alphabetical!

Example: DFS traversal of undirected graph



DFS (a)

Example: DFS traversal of undirected graph

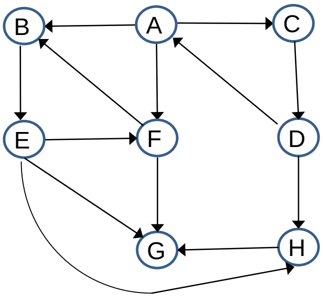


Back Edges?

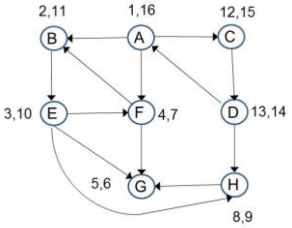
Directed Graphs – things to know

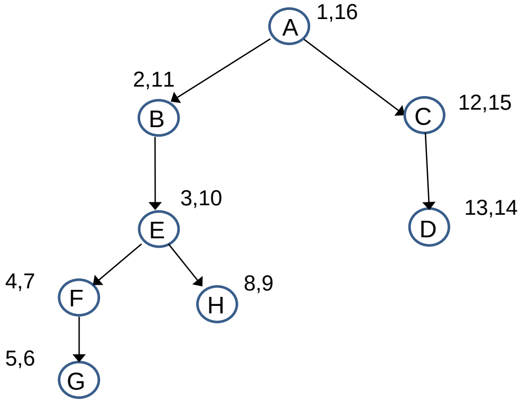
- Directed Graph or Digraph: every edge has a direction.
  - Adjacency matrix – may not be symmetric. Adjacency list – only put edge in one not two of the edge lists
- When constructing the depth-first search *forest* you can get 4 different types of edges
  - tree edges
  - non tree edges of the following types
    - forward edges – to descendants other than children
    - back edges – to ancestors
    - cross edges – to anywhere else

Example: DFS traversal of directed graph



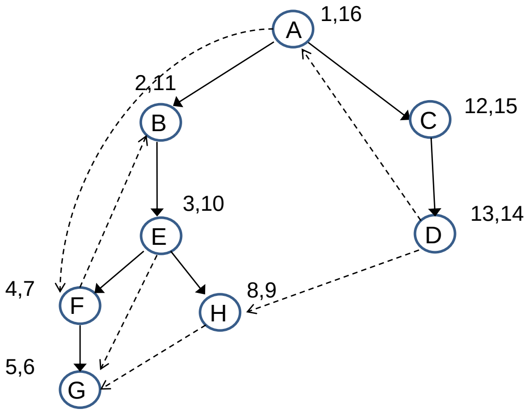
Example: DFS traversal of directed graph





Draw in the Back (2), Forward(2), and Cross Edges(2) – labeling appropriately.

Example: DFS traversal of directed graph



Draw in the Back (2), Forward(2), and Cross Edges(2) – labeling appropriately.

Classification of edges by a DFS of a directed graph

- Tree edges – part of the DFS forest - edges that involve visiting a vertex for the first time
- Forward edges – edges from a node to a non-child descendent in a DFS tree – the descendent has already been visited
- Backward edges – edges to an ancestor in the DFS tree
- Cross edges – edges to a vertex that is neither an ancestor or a descendent

Pre and Post numbers

Let (u,v) be an edge from u to v. Let u_{pre}, u_{post} and v_{pre}, v_{post} denote the pre and post numbers for vertex u and v respectively

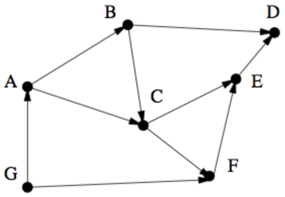
- $[u_{pre}, u_{post}]$ $[v_{pre}, v_{post}]$: the interval for v is contained entirely in the interval for u → the edge from u to v is either a *tree* or a *forward* edge, said another way v is a *descendent* of u. u_{pre}, v_{pre}, v_{post}, u_{post}
- $[u_{pre}, u_{post}]$ $[v_{pre}, v_{post}]$: the intervals are entirely disjoint → the edge from u to v is a cross edge, neither is a descendent of the other. v_{pre}, v_{post}, u_{pre}, u_{post},
- $[u_{pre}, u_{post}]$ $[v_{pre}, v_{post}]$: the interval for u is contained entirely in the interval for v → the edge from u to v is a *backward* edge, v is a *ancestor* of u. v_{pre}, u_{pre}, u_{post}, v_{post},

Notes on DFS

- DFS can be implemented with graphs represented as:
  - adjacency matrices: $\Theta(V^2)$
  - adjacency lists: $\Theta(|V|+|E|)$
- 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 (strong and weak)
  - checking acyclicity
  - finding articulation points and biconnected components
  - searching state-space of problems for solution (AI)

Directed Acyclic Graphs: Topological Sorting

- A *DAG*: a directed acyclic graph, i.e. a directed graph with no (directed) cycles   (Constraint modeling)
- Topological Sorting: Vertices of a DAG can be linearly ordered so that for every edge its starting vertex is listed before its ending vertex



Applications of Topological Sorting

- Instruction scheduling
- Ordering of formula cell evaluation when recomputing formula values in spreadsheets
- Determining the order of compilation tasks to perform in makefiles
- Data serialization and resolving symbol dependencies in linkers
- Determine order to load tables with foreign keys in databases

- Project Evaluation and Review Technique, commonly abbreviated PERT, is a statistical tool, used in project management, which was designed to analyze and represent the tasks involved in completing a given project.
- Critical Path Method

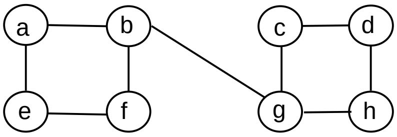
DFS-based Algorithm for Topological Sort

- Perform DFS traversal, noting the order vertices are popped off the traversal stack
- Reverse order (list vertices with highest post number first) solves topological sorting problem
- Back edges encountered?→ NOT a dag!

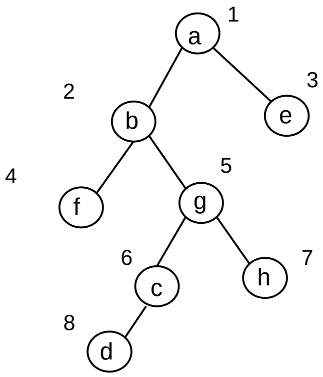
Breadth-first search (BFS)

- Visits graph vertices by moving across to all the neighbors of last visited vertex
- Instead of a stack, BFS uses a queue
- Similar to level-by-level tree traversal
- “Redraws” graph in tree-like fashion (with tree edges and cross edges for undirected graph)

Example of BFS traversal of undirected graph

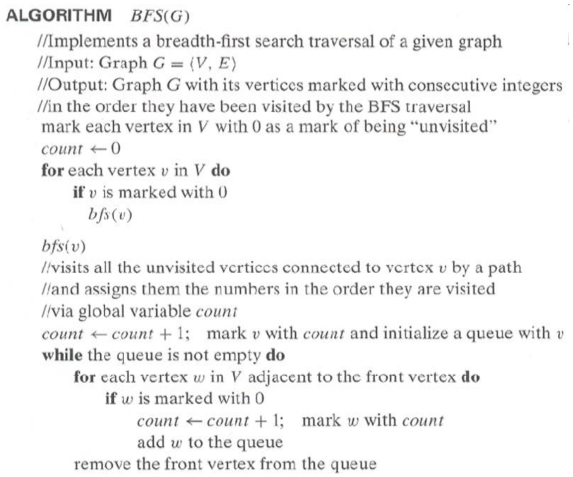


Example: BFS traversal of undirected graph



Cross Edges ?

Pseudocode of BFS



```
ALGORITHM  BFS(G)
    //Implements a breadth-first search traversal of a given graph
    //Input: Graph G = ⟨V, E⟩
    //Output: Graph G with its vertices marked with consecutive integers
    //in the order they have been visited by the BFS traversal
    mark each vertex in V with 0 as a mark of being “unvisited”
    count ← 0
    for each vertex v in V do
        if v is marked with 0
            bfs(v)

    bfs(v)
    //visits all the unvisited vertices connected to vertex v by a path
    //and assigns them the numbers in the order they are visited
    //via global variable count
    count ← count + 1;   mark v with count and initialize a queue with v
    while the queue is not empty do
        for each vertex w in V adjacent to the front vertex do
            if w is marked with 0
                count ← count + 1;   mark w with count
                add w to the queue
        remove the front vertex from the queue
```

Notes on BFS

- BFS has same efficiency as DFS and can be implemented with graphs represented as:
  - adjacency matrices: Θ(V^2)
  - adjacency lists: Θ(|V|+|E|)
- Yields single ordering of vertices (order added/deleted from queue is the same)
- Applications: same as DFS, but can also find paths from a vertex to all other vertices with the smallest number of edges

DFS compared to BFS

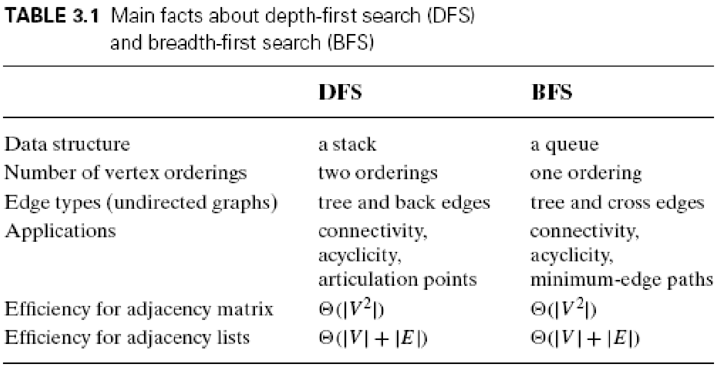


TABLE 3.1 Main facts about depth-first search (DFS) and breadth-first search (BFS)

	DFS	BFS
Data structure	a stack	a queue
Number of vertex orderings	two orderings	one ordering
Edge types (undirected graphs)	tree and back edges	tree and cross edges
Applications	connectivity, acyclicity, articulation points	connectivity, acyclicity, minimum-edge paths
Efficiency for adjacency matrix	$\Theta(\|V^2\|)$	$\Theta(\|V^2\|)$
Efficiency for adjacency lists	$\Theta(\|V\| + \|E\|)$	$\Theta(\|V\| + \|E\|)$

Decrease-and-Conquer
(Considered by many as part of Divide and Conquer)

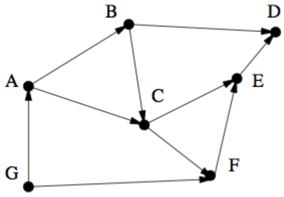
- Reduce problem instance to smaller instance of the same problem
- Solve smaller instance
- Extend solution of smaller instance to obtain solution to original instance

Types of Decrease and Conquer

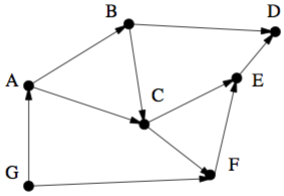
- Decrease by a constant (usually by 1):
  - » insertion sort
  - » topological sorting
  - » algorithms for generating permutations, subsets
- Decrease by a constant factor (usually by half)
  - » binary search and bisection method
  - » exponentiation by squaring
  - » multiplication à la russe
  - » Destructive testing
- Variable-size decrease
  - » Euclid’s algorithm
  - » selection by partition
  - » Nim-like games

Recall the Topological Sorting Problem in DAG

- A DAG: a directed acyclic graph, i.e. a directed graph with no (directed) cycles
- Topological Sorting: Vertices of a DAG can be linearly ordered so that for every edge its starting vertex is listed before its ending vertex



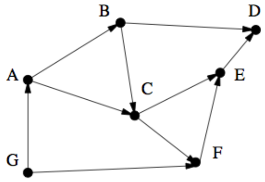
A different approach to Topological Sorting - 1



- What must be true of any vertex that is the first vertex in a topological sort?

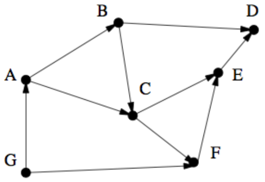
- If that vertex is removed (e.g. task completed), then what must be true of the vertex that comes next?

A different approach to Topological Sorting - 2



- What must be true of any vertex that is the first vertex in a topological sort?
  - It must have no incoming edges --- its in-degree = 0
- If that vertex is removed (e.g. task completed), then what must be true of the vertex that comes next?
  - Again it must have no incoming edges after the first vertex and its outgoing edges are removed from the graph!

A different approach to Topological Sorting - 3

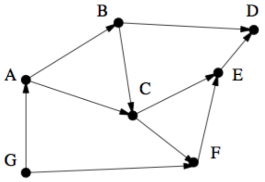


- This gives us a pretty good idea of an iterative algorithm that can be used to produce a topological sort. Assuming we can compute and keep track of the in-degree of each vertex efficiently.
- How can we efficiently compute the in-degree of a directed graph?

CAL POLY
SAN LUIS OBISPO

Computer Science Department

A different approach to Topological Sorting - 4

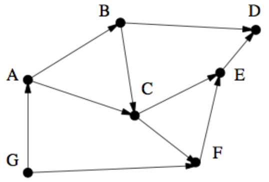


- How can we efficiently compute the in-degree of each vertex of a directed graph?
  - For each vertex have a field that will keep the count of incoming edges.
  - Traverse the adjacency list (or matrix) and increment the target vertex in-degree field for each edge.

CAL POLY
SAN LUIS OBISPO

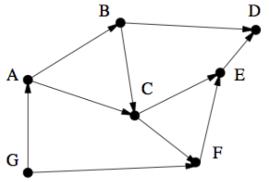
Computer Science Department

A different approach to Topological Sorting - 5



- How can we efficiently update in-degree of the remaining directed graph after a vertex has been removed?
  - Traverse the adjacency list of the vertex being removed and decrement the target vertex in-degree field for each edge.

A different approach to Topological Sorting - 6



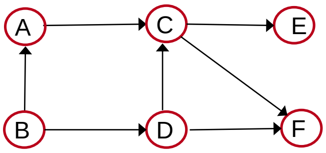
		remove G	remove A	remove B	remove C	remove F	remove E	remove D
A	1	→0	*order*					
B	1	1	→0	*order*				
C	2	2	→1	→0	*order*			
D	1	1	1	→0	1	1	→0	*order*
E	2	2	2	2	→1	→0	*order*	
F	2	→1	1	1	→0	*order*		
G	0	*order*						

Source Removal Algorithm

```
Compute in-degree of all vertices
Repeat
   - identify a source vertex (in-degree = 0)
      (a vertex with no incoming edges)
   - remove the source and all the edges from
     it, update in-degrees of target vertices
Until either
   - no vertex is left-all vertices marked done
    (problem is solved) or
   - no source among remaining vertices (not a dag)
```

- Efficiency: same as efficiency of the DFS-based algorithm

Source Removal Algorithm Example



- One Source, Two Sinks, 4 possible linearizations (topological sorts)
- Every DAG has at least one source and a sink

Euclid’s Algorithm

- Euclid’s algorithm is based on repeated application of equality gcd(m, n) = gcd(n, m mod n)
  - Ex.: gcd(80,44) = gcd(44,36) = gcd(36, 12) = gcd(12,0) = 12
- One can prove that the second number, decreases at least by half after two consecutive iterations.
- Recurrence relation: $T(n) = T(n/2) + 2$ // basic op is mod
- Hence, $T(n) \in \Theta(\log n)$. However as you will see later this is not a log n algorithm since n is not the correct measure of the size of the problem

Priority Queue

- A priority queue is the ADT of a set of elements with numerical priorities and the following operations:
  - find element with highest priority
  - delete element with highest priority
  - insert element with assigned priority
- Heap is a very efficient way for implementing priority queues
- Applications determine what is a priority ordering!
  Sometimes want largest, sometimes smallest element to be found/deleted.
- Many implementation options : list, sorted list, …

A Priority Queue is an ADT

- Useful in many contexts: process scheduling, shortest paths,…
- Many implementation options : list, sorted list, …
- With following operations:
  - find element with highest priority
  - delete element with highest priority
  - insert element with assigned priority
- Enhance with
  - Delete a given element
  - Change key for a given element
  - The essential idea is to be able to find the element in the heap in constant time! Maintain an additional array for the objects in the heap that contains their position in the (heap) priority queue (handle)

Heaps Efficiently Implement Priority Queues

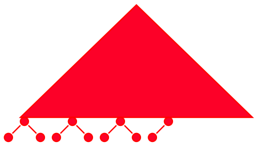
- Heap is a very efficient way of supporting an algorithm that requires repeatedly finding the maximum (or minimum) of a data set.

- In particular, heaps support
  - Remove max – O(log n) time
  - Insert – O(log n) time

- Examples of problems where Heaps are useful
  - **Priority Queues**
  - Simulations
  - Sorting

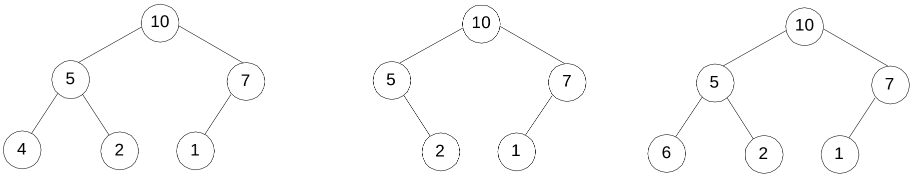
Max Heaps (almost identical for Min Heaps)

- Definition A heap is a binary tree with keys at its nodes (one key per node) such that:
- It is *essentially complete*,
  - i.e., all its levels are full except possibly the last level, where only rightmost keys may be missing – **shape property**.



- The key at each node is ≥ keys at its children (MaxHeap) – **heap property** (sometimes called parental dominance)

Illustration of the heap's definition



a heap **not a heap** **not a heap**

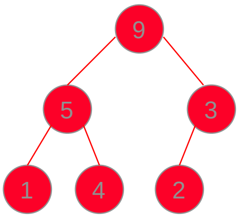
Note: Heap's elements are ordered along any path down from its root, but they are not ordered left to right

Some Important Properties of a Heap (MaxHeap)

- Given *n,* there exists a unique binary tree with *n* nodes that is essentially complete, with $h = \lfloor \log_2 n \rfloor$
- The root contains the largest key
- The subtree rooted at any node of a heap is also a heap
- A heap can be efficiently implemented using an array

Heap’s Array Representation

- Store heap’s elements in an array (whose elements indexed, for convenience, 1 to n) in top-down left-to-right order
- Example:



Index	1	2	3	4	5	6
Value	9	5	3	1	4	2

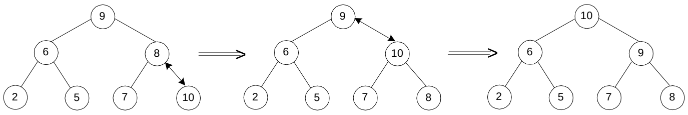
- Left child of node j is at 2j   Right child of node j is at 2j+1
- Parent of node j is at $\lfloor j/2 \rfloor$
- Parental nodes are represented in the first $\lfloor n/2 \rfloor$ locations

Insertion of an element into a heap

- Insert the new element at last position in heap. This maintains structure property
- Compare it with its parent and, if it violates heap condition, exchange them (Drift up)
- Continue comparing the new element with nodes up the tree until the heap condition is satisfied

Example: Insert key 10

Efficiency: O(log *n*) since the height of the tree is log n



Insertion into heap: (Drift up or Percolate up)

- Set aside first open position to make room for new element.
- Drift the hole up until find place for new element that maintains the heap property

```
A is array
Insert( Item x)
    check heap capacity, increase if necessary
    increase heap size by 1 make space for additional item
    int hole = heapSize
    while ((hole>1) && x>heap[hole/2]))
       A[hole]=A[hole/2]    // move parent down
       hole = hole/2        // move open position up
    a[hole]=x               // put x where it belongs
```

Return and remove max from heap

```
Store first entry (max) for return
Move the last entry in heap to first entry in heap
Reduce heapsize by 1
Drift-down the new first element until the heap property is
       restored
Return the original root of the heap
```

Note: There are only two basic “moves” in the heap.

- *Drift-down* – used in bottom up construction and removal
- *Drift-up* – used in insertion and top-down construction

Remove max uses:
Drift down or Percolate down

```
a is array containing the heap
driftDown( int pos)
    tmp=array[pos]  // save value want to drift down
    while(pos * 2 <= heapSize)  // pos is parent node
        child = pos * 2
        if( child!=heapSize && (a[child + 1]>a[child]))
             child++
        if array[ child ] > tmp  )  //array[child] bigger
             a [ pos ] = a [ child ] // move it up
             pos = child            // drift down pos
        else
             break
    a [pos] = tmp
```

Heap Construction 1: Top-down heap construction

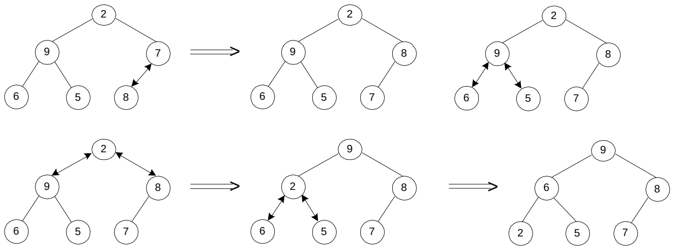
- Start with empty heap and repeatedly insert elements
- Worst case performance = ???
  - Each drift down of the final n/2 elements takes log (n/2)
  - There are n/2 of these elements

Heap Construction 2 Bottom-up

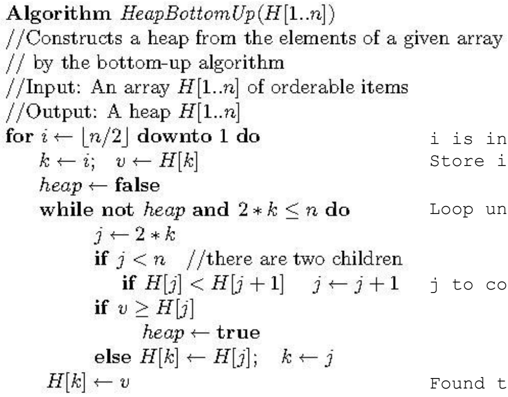
- Initialize the array structure with keys in the order given
  - this ensures the structure property
- Loop: from the last node with children *downto* root
  - if it node doesn't satisfy the heap condition:
    Drift Down
- This will enforce the Heap condition on each parent node

Example of Bottom-up Heap Construction

Construct a maxheap for the list 2, 9, 7, 6, 5, 8



Detailed pseudocode (different version of drift down) of bottom-up heap construction



```
Algorithm HeapBottomUp(H[1..n])
//Constructs a heap from the elements of a given array
// by the bottom-up algorithm
//Input: An array H[1..n] of orderable items
//Output: A heap H[1..n]
for $i \leftarrow \lfloor n/2 \rfloor$ downto 1 do                i is index of node to drift down
    $k \leftarrow i$;   $v \leftarrow H[k]$                        Store i and H[i] to swap later
    heap ← false
    while not heap and $2 * k \leq n$ do                  Loop until find spot for H[i]
        $j \leftarrow 2 * k$
        if $j < n$   //there are two children
            if $H[j] < H[j+1]$   $j \leftarrow j + 1$          j to contain biggest child
        if $v \geq H[j]$
                heap ← true
        else $H[k] \leftarrow H[j]$;   $k \leftarrow j$
    $H[k] \leftarrow v$                                   Found the correct spot for H[i]
```