# Analyzing Algorithms

## Running Time Analysis

* **Worst-case**: Maximum time of algorithm on any input of size n
* **Average-case**: Expected time of algorithm over all inputs of size n
* **Best-case**: Fastest time on some inputs

## Asymptotic Analysis

* Ignore machine-dependent constants
* Analyze growth of T(n) as n approaches ∞
  + Drop lower order terms and ignore leading constants

## Asymptotic Notation

### O-notation: Upper Bound

There exists positive constants and such that for all

* Classifies algorithms by how they respond to input size changes
* Characterizes functions by their growth rate

### Ω-notation: Lower Bound

There exists positive constants and such that for all

* Demonstrates that dominates as n increases

### Θ-notation: Tight Bound

There exists positive constants, , and such that for all

* The growth of and are the same as n increases

**Exercise:** Prove the following:

# Solving Recurrences

**Recurrence**: an equation/inequality that describes a function in terms of its own value on smaller inputs

## Recursion Tree Method

* Draw a tree, with the execution times of each node and the number of children of each node
* Analyze total execution time of each level
* Add up each level to get total execution time

## Repeated Substitution

* Substitute for n until you find a pattern
* Write a formula in terms of n and i (the number of substitutions)
* Choose i so that it references the base case
* Solve the resulting summation

**Exercise:** Solve the following recurrences:

1. Linear search in an array:
2. Binary search:

## Master Theorem

Can be applied to recurrences in the form:

where and is asymptotically positive

1. Find , , and
2. Calculate
3. Compare and asymptotically
4. Apply appropriate case

### Case 1

for some

* grows slower than by a factor of
* Running time dominated by leaves

### Case 2

* and grow at similar rates
* Running time spread evenly between root and leaves

### Case 3

for some and satisfies **regularity condition** for some

* grows faster than by a factor of
* Running time dominated by root

# Elementary Data Structures (Stacks/Queues)

## Definitions

* **Data type**: set of allowed values for a variable
* **Data structure**: systematic way of organizing/accessing data
* **Abstract data type**: set of elements + set of operations on these elements
  + Serves as specification of requirements when building solutions to algorithm problems
  + Encapsulates data structures and algorithms that implement them

## Data storage for ADTs

### Contiguous storage (Array)

* Stores n objects into a contiguous (continuous) space of memory
* If more memory is required, need to copy all existing information into new memory

### Node-based storage (Linked List)

* Each item being stored holds the object (data) itself and a reference to the next item
* Allows for dynamic allocation, with objects arranged in linear order
* Doubly linked lists allow for all operations to run in O(1) constant time

## Stacks

* Objects are inserted/removed according to the last-in-first-out (LIFO) principle
* Array implementation with internal array S of size N, and pointer t to top of stack:
  + All methods run in O(1) time
  + push(x) – inserts x on top of S

if size() = N

return Error

t=t+1

S[t] = x

* + pop() – removes top object from S

if size() = N

return Error

top = S[t]

S[t] = null

t = t-1

return top

* + top() – returns top object of S

if isEmpty()

return Error

return S[t]

* + size() – returns current number of objects in stack

return t+1

## Queues

* Objects are inserted/removed according to the first-in-first-out (FIFO) principle
* Array implementation with internal array Q of size N, and pointers f, r for the front and rear elements:
  + All methods run in O(1) time
  + enqueue(x) – inserts x at back of Q

if size() = N

return Error

Q[r] = x

r = (r+1) mod N

* + dequeue() – removes front object of Q

front = Q[f]

Q[f] = null

f = (f+1) mod N

return front

* + size() – returns current number of objects in Q

return (N-f+r) mod N

* + front() – returns object at the front of Q

if isEmpty()

return Error

return Q[f]

# Hash Tables

**Hash function**: used to map a set of unique keys into an array

* Collisions occur if a slot is already occupied

## Collision Resolution

### Chaining

* Array of links (a table) indexed by keys, to lists of items with the same key
* **Simple Uniform Hashing**: Assume each key is equally likely to be hashed into any slot of the table, independent of where other keys are hashed
* **Load factor**: Given a hash table with slots holding elements, the load factor is – the average number of keys per slot
* Search, insertion, and deletion all take O(1) time when lists are doubly linked

### Open Addressing

* All elements are stored in the hash table
* Probe the table until an empty slot is found
* Hash function takes in probe number

### Linear Probing

* If the current location is filled, try the next location
  + h(k,i) = (h(k) + i) mod m
* Uses less memory than chaining – don’t need to store links to lists
* Slower than chaining – primary clustering means may need to walk along table for a long time

### Double Hashing

* Use two hash functions
  + h(k,i) = (h1(k) + i\*h2(k)) mod m
* Distributes keys more evenly than linear probing
* h2(k) must always be co-prime to m to guarantee that probe sequences will return all slots from 0 to m-1
  + If m is prime, have 1 < h2(k) < m
  + Let m = 2j and have h2(k) always return an odd number

## Hash Function

* A good hash function should distribute keys uniformly and compute quickly
* **Division method**
  + h(k) = k mod m
  + k = key, m = array size
  + Choose m to be prime to help ensure even distribution
* **Multiplication method**
  + h(k) = floor( m \* (k\*A) mod 1 )
  + k = key, m = array size, 0 < A < 1 (a constant)
  + Typically, choose m = 2j

# Trees

A collection of nodes

* **Parent/Child**: every node except the root has one parent; a node can have an arbitrary number of children
* **Leaves**: nodes with no children
* **Siblings**: nodes with the same parent

Terminology

* **Path**: a sequence of nodes where each node is the parent of its successive node
* **Length**: the number of edges in a path
* **Depth of a node**: the length of the path from root to the node
  + Depth of tree = depth of deepest leaf
* **Height of a node**: length of the longest path from a node to a leaf
  + All leaves have height 0
  + Height of tree = height of root

## Binary Tree

A tree where each node doesn’t have more than two children

ADT

Accessors

* key()
* parent()
* left()
* right()

Mutators

* setKey()
* setParent()
* setLeft()
* setRight()

## Tree Traversal

* **Pre-order traversal**: Node, Left, Right
* **In-order traversal**: Left, Node, Right
* **Post-order traversal**: Left, Right, Node

# Binary Search Trees

A binary tree where each key in the left subtree of a node is smaller than the node’s key, and each key in the right subtree is larger than the node’s key

## Searching

**Iterative**

Search(T, k)

x = T

while (x != null)

if (x.key() == k) return x

if (x.key() < k) x = x.right()

else x = x.left()

return null

**Recursive**

Search(T, k)

if (T == null) return null

if (T.key() == k) return T

if (T.key() < k) return Search(T.right(), k)

return Search(T.left(), k)

* The running time on BST of height h is O(h)
* Worst-case running time on a BST with n items is O(n) – straight BST

## Minimum/Maximum

TreeMin(x)

while x.left() != null

x = x.left()

return x

TreeMax(x)

while x.right() != null

x = x.right()

return x

* The running time of search on BST of height h is O(h)

## Successor/Predecessor

**Case 1: right subtree of x is non-empty**

* Successor is the leftmost node in the right subtree of x
* Return TreeMin(x.right())

**Case 2: right subtree of x is empty**

* Successor is the lowest ancestor whose left child is also an ancestor of x

Successor(x)

if (x.right() != null) return TreeMin(x.right())

p = x.parent()

while (p != null && p.right() == x)

x = p

p = p.parent()

return p

* The running time of search on BST of height h is O(h)

## Insertion

* Find where z belongs
* Insert z by setting parent and left/right pointers

TreeInsert(T, z)

x = T

y = null

while (x != null)

y = x

if (x.key() < z.key()) x = x.right()

else x = x.left()

z.setParent(y)

if (y != null)

if (y.key() < z.key()) y.setRight(z)

else y.setLeft(z)

else T = z

## In-Order Tree Traversal

* Useful for sorting – prints out keys in order from lowest to highest

InOrderTreeWalk(T)

if (T == null) return

InOrderTreeWalk(T.left())

print T.key()

InOrderTreeWalk(T.right())

## Deletion

**Case 1: No children**

* Remove x

**Case 2: One child**

* Set x.parent() to point to x’s child
* Remove x

**Case 3: Two children**

* Find x’s predecessor/successor y
* Delete y
* Replace x with y

# AVL Trees

## Balanced BSTs

* The height of a binary tree is at least Θ(log n)
  + Worst-case execution time of insertion and deletion in BST is Θ(n)
  + Balanced binary search trees guarantee small height
* AVL trees are BSTs where for every node, the height of the left and right subtrees differ by at most 1
  + Height of subtree = maximum number of edges to a leaf
  + Height of an empty subtree = -1
  + Height of one node = 0

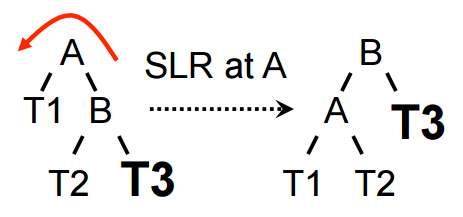
## AVL Tree Insertion

* Insert z as in a regular BST
* Trace path from z to root; at each node, check if node is balanced
  + Only nodes on the path from insertion point to root may be violated
  + Must rebalance the deepest violated node
* Insertion is always O(log n)
  + Insert z = O(log n)
  + Trace path from leaf to root = O(log n)
  + Check height difference/Rotations = O(1)

Let A be the node that needs to be rebalanced:

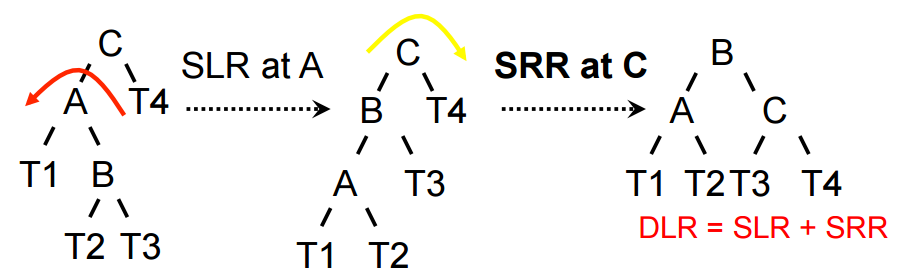
### Case 1: z is inserted into left subtree of left child of A

Single Left Rotation (SLR)



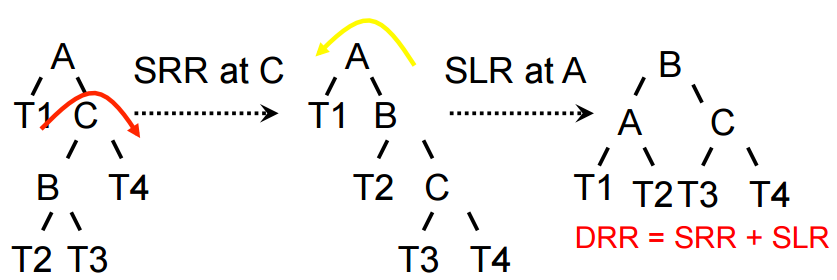
### Case 2: z is inserted into right subtree of left child of A

Double Left Rotation (DLR) = SLR + SRR



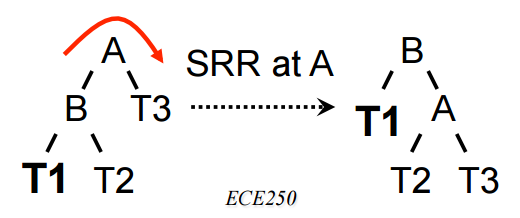
### Case 3: z is inserted into left subtree of right child of A

Double Right Rotation (DRR) = SRR + SLR



### Case 4: z is inserted into right subtree of right child of A

Single Right Rotation (SRR)

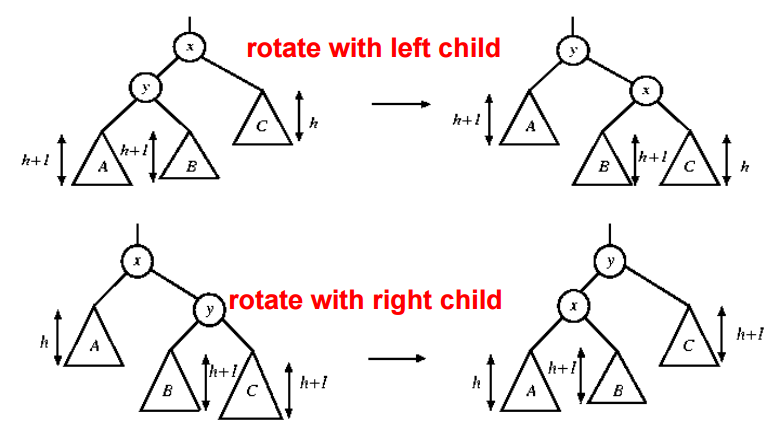


## AVL Tree Deletion

* Delete z like in a normal BST
* Trace path from new leaf to root; at each node, check if node is balanced
* If not, perform rotation
* Keep checking until we reach root

### New Case: two subtrees of y are the same height

Single Left/Right Rotation will suffice



# B-Trees

* **Disk-based data structure**: a search tree that is secondary storage enabled (e.g. hard disks, magnetic disks, etc.)
  + Running time of disk-based algorithms depends on CPU and number of disk accesses
* Size of B-tree nodes depends on page size – one node = one page
* Height is logarithmic – e.g. B-tree of height 2 contains over 1 billion keys, if root node has 1000 keys

## B-Tree Definitions

* Every leaf has the same depth – the tree’s height *h*
* In a B-Tree of **degree** t:
  + Every node except the root has **at least t-1 keys**;thus, every node except the root has **at least t children**
  + Every node contains **at most** **2t-1 keys**; therefore every node has **at most 2t children**
  + The root node has between 0 and 2t-1 keys (0 and 2t children)
* For a B-tree of height *h*, containing *n* (>= 1) keys and degree *t* (>= 2):
* **Searching**: typical binary search tree search algorithm, with added disk reading
* **Creating an empty tree**: create a root and write it to the disk

## Splitting Nodes

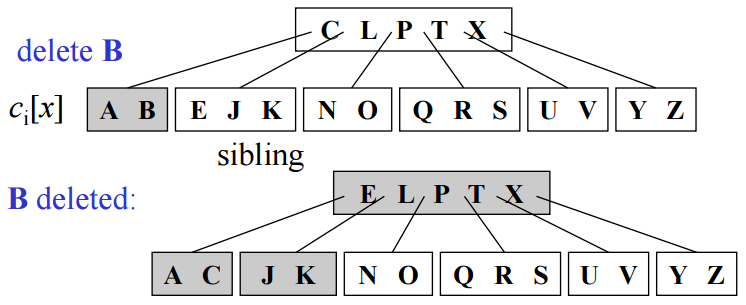
* When nodes reach their maximum capacity 2t-1, each full node must be split before insertion of new keys
* One (middle) key of node moves up to the parent, leaving two nodes each with t-1 keys
* Insertion is only done in leaf nodes
* Splitting the root node requires creating a new node – tree grows at the top instead of bottom
* Running time:

## Inserting Keys

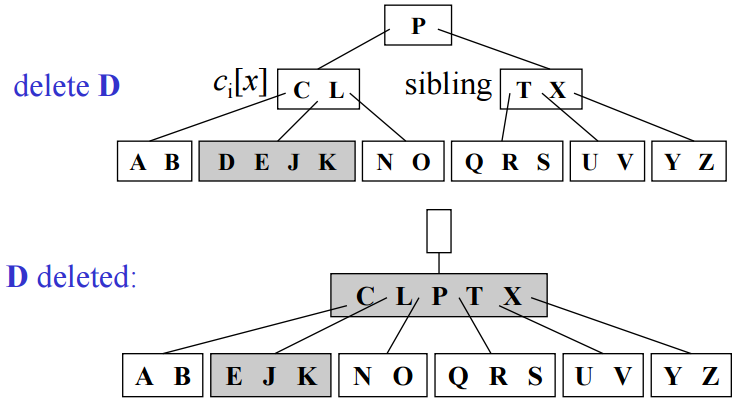
* Starting from root, recursively traverse down the tree to leaf level
* Ensure that every node has < 2t-1 keys, splitting when necessary
* Insert key into appropriate node
* Running time:
  + Disk I/O:
  + CPU:

## Deleting Keys

* Starting from root, recursively traverse down the tree to leaf level
* Ensure that every node has >= t keys
* Three cases (*k* = key to delete):
  + **Case 1: *k* in leaf node *x* and *x* has >= t keys** 
    - Just delete *k* from *x*
  + **Case 2: *k* in internal node *x***
    - If the child preceding *k* has >= t keys:
      * Find the predecessor of *k, j*
      * Recursively delete *j*
      * Replace *k* with *j* in *x*
    - If both children preceding/succeeding *k* have t-1 keys:
      * Merge the two children and delete *k* from *x*
  + **Case 3: *k* suspected in lower level node ci[*x*]**
    - Distribution: If ci[*x*] only has t-1 leys but one of its immediate siblings has >= t keys, move a key from *x* to ci[*x*], and move a key from ci[*x*]’s immediate sibling up to *x*



* + - Merging: If ci[*x*] and its immediate siblings only have t-1 leys, merge ci[*x*] with one sibling by moving a key from *x* down to the merged node, as the median key of that node



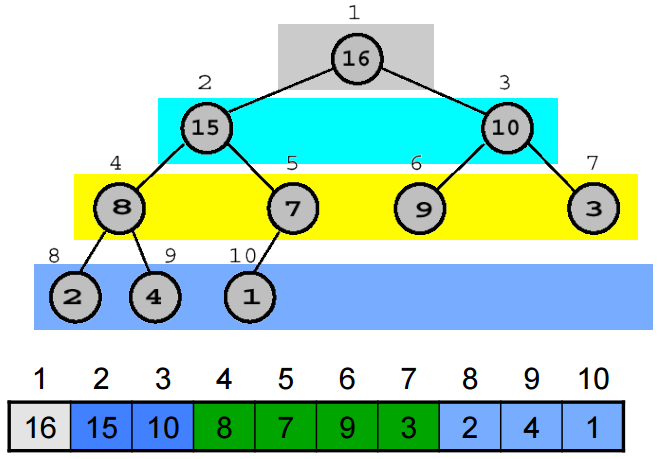
* Running time:
  + Disk I/O:
  + CPU:

# Heaps, Heapsort, and Priority Queues

## Binary Heaps

* Nearly complete binary tree – all levels except lowest ones are completely filled
* The key in the root is greater than the keys of all its children; left and right subtrees are again binary heaps

## Max Heap



* Parent(i) = floor(i/2)
* Left(i) = 2i
* Right(i) = 2i + 1
* A[Parent(i)] >= A[i]

## Heapify

* Heapify(A, i) makes A into a heap by moving A[i] down the heap until the heap property A[Parent(i)] >= A[i] is satisfied again
  + Compares A[i] with both of its children (e.g. A[j] and A[k])
  + If a larger child is found (e.g. A[j]), swap A[i] and A[j]
  + Call Heapify(A, j)
* Running time: O(log n)

## Building a Heap

* Given an array A[1 … n], convert it into a heap
* The elements in the second half of the array are leaf nodes, which are 1-element heaps
* Hence, only need to call Heapify(A, i) on i = floor(n/2) down to i = 1
* Running time: O(n)

## Heapsort

Running time: O(n) + O(n log n) = O(n log n)

Heapsort(A):

BuildHeap(A);

for (int i = A.length; i > 1; i++) {

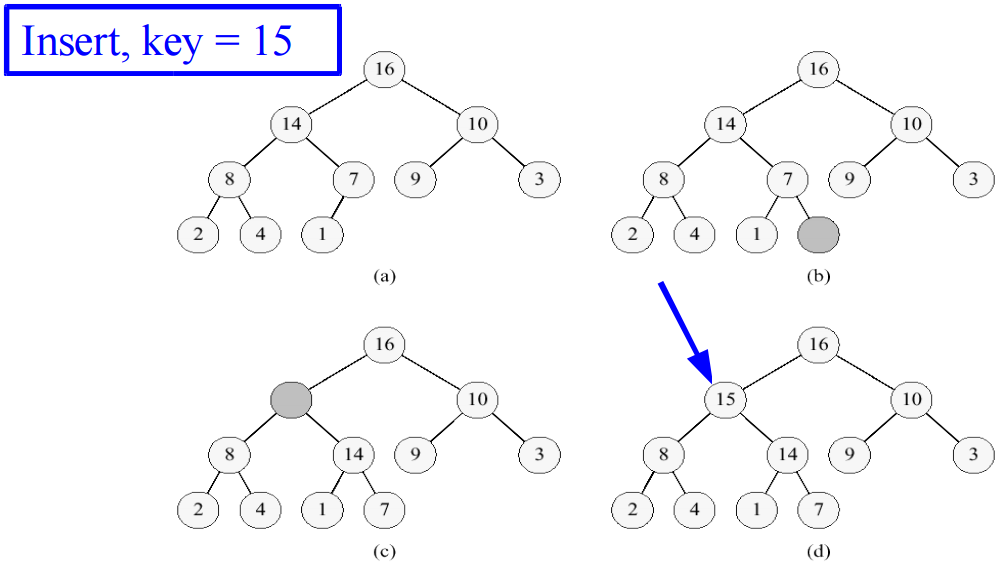
Swap(A[i], A[1]);

Heapify(A, 1);

}

## Priority Queue

* A data structure for maintaining a set of elements that each have an associated key
* A max heap can be used to implement a max PQ with the following operations:
  + **Insert(S, x)**: insert element x into set S
    - Add empty spot in PQ
    - Traverse up the parents while the parent’s key < x’s key
    - Insert x in its position
    - Running time: O(log n)



* + **Maximum(S)**: returns the element in S with the largest key
  + **ExtractMax(S)**: returns and removes the element in S with the largest key
    - max = A[1]
    - Swap A[1] and A[n]
    - n -= 1
    - Heapify(A, 1) to remake the heap
    - return max
    - Running time: O(log n)

# Quicksort

* In-place sort – doesn’t require additional array
* Divide and conquer algorithm:
  + Partition array into 2 subarrays, such that elements in the lower part <= elements in the higher part
    - Select a pivot element x around which to partition
    - One variable i starts from beginning of array, the other j starts from end
      * Increment i until you find an element >= x
      * Decrement j until you find an element <= x
      * If i < j, swap elements and continue incrementing/decrementing
      * Otherwise, you are done, so return j
  + Recursively sort the 2 subarrays
  + Combine the sorted subarrays (trivial, since already sorted)
* Running time: depends on the distribution of partitions
  + Average (balanced) case: O(n log n)
  + Worst case: O(n2) - input is already sorted or reverse sorted

# Sorting in Linear Time

* **Comparison sorts**: use comparisons to determine the relative order of elements – worst-case running time is minimum O(n log n)
  + Every comparison sort can be modeled by a decision tree
    - Algorithm splits whenever it compares two elements
    - Tree contains the comparisons along all possible instruction traces
    - Running time of the algorithm = length of path taken, worst case time = height of tree
* **Counting sorts**: doesn’t rely on comparisons between elements
  + For each input x, determine how many elements are less than x
  + Use this info to place x in the correct position of the output array
  + Stable sort: preserves input order of equal elements

CountingSort(A, k)

C = new int[k]; // aux array; k = max number

B = new int[n]; // output array; n = A.length

for (int i = 0; i < n; i++) { // count the number of an element

C[A[i]]++;

}

for (int i = 1; i < k; i++) { // add previous number to current

C[i] = C[i] + C[i-1];

}

for (int i = n-1; i >= 0; i--) { // put elements in output array

B[C[A[i]]] = A[i];

C[A[i]]--;

}

return B;

# Radix Sort

* Sorts each input from least significant to most significant digit with a **stable sort**
* Given *n* *d*-digit numbers where each digit takes up to *k* possible values, radix sort correctly sorts these number in O(d(n + k)) time if the stable sort that it uses takes O(n + k) time
* Given *n* *b*-digit numbers and positive integers *r* <= *b*, radix correctly sorts these numbers in O((b/r)(n + 2r)) time if the stable sort that it uses takes O(n + k) time
  + If we’re sorting *n* words with *b* bits each, each word can be seen having *b/r* base-2r digits
  + Example: 32-bit word
    - If r = 8, b/r = 4 passes of counting sort on base-28 digits
    - If r = 16, b/r = 2 passes of counting sort on base-216 digits
  + Want to choose r to minimize total running time
    - Choosing r = log n implies T = O(bn/log n)
    - For numbers in the range 0 to nd – 1, b = d log n so radix sort runs in O(dn) time
* Is radix sort preferable to comparison sorts?
  + If b [number of digits] = O(log n [number of elements]), then if we choose r as close as possible to log n, radix sort’s running time will be O(n), better than comparison sorts

# Dynamic Programming

* Remember solution to solved sub-problems; build complete solution bottom-up

## Optimization Problems

* Choose most optimal solution of many – one with the max/min value
* Solution exhibits a structure – a string of the most optimal choices

Steps:

1. Show optimal substructure – an optimal solution to the problem contains optimal solutions to sub-problems
2. Write recurrence for the value of an optimal solution
3. Compute the value of an optimal solution in a bottom-up fashion, so that necessary sub-results are always computed
4. Construct optimal solution from computed information (make sequence of choices leading to optimal solution)

## Longest Common Subsequence (LCS) Problem

Let Xm = x1 x2 … xm and Yn = y1 y2 … yn, and Z be the LCS of Xm and Yn

**Step 1: Optimal Substructure**

* If xm = yn, append to beginning of Z and find LCS(xm-1, yn-1)
* If xm != yn, skip either a letter from X or a letter from Y by taking max { LCS(xm, yn-1), LCS(xm-1, yn) }

**Step 2: Recurrence**

LCS(Xi, Yj) =

* 0 if i = 0 or j = 0
* LCS(Xi-1, Yj-1) + 1 if i, j > 0 and xi = yj
* max { LCS(Xi, Yj-1), LCS(Xi-1, Yj) } if i, j > 0 and xi != yj

# Elementary Graph Algorithms

## Graph Terminology

A graph G = (V, E) is composed of V – a set of vertices, and E – a set of edges connecting the vertices

* An edge e = (u, v) is an edge from u to v (assume directed graphs)
* Two vertices u and v are **adjacent** if and only if (u, v) exist in E
* The **degree** of a vertex = the number of adjacent vertices
* **Path**: a sequence of adjacent vertices
  + **Simple path**: a path with no repeated vertices
  + **Cycle**: a simple path that ends on the first vertex
* **Subgraph**: a subset of vertices and edges forming a graph
  + **Connected component**: the maximum subgraph where every two vertices are connected by a path
* **Connected graph**: a graph where every two vertices are connected by a path
  + **Tree**: connected graph without cycles
  + **Forest**: collection of trees

## Adjacency List

* For a vertex v, a sequence of vertices adjacent to v
* Space required: O(V + sum of degree(V)) = O(V + E)

## Adjacency Matrix

* Matrix M with entries for all pair of vertices
  + M[i, j] = true – there is an edge (i, j) in the graph
  + M[i, j] = false – there isn’t an edge (i, j) in the graph
* Space = O(V2)

## Breadth-First Search

* Traverse a connected component of a graph, defining a spanning tree
* Looks at all vertices on the same level before going to the next
  + Initialize all vertices with distance = infinity, color = white
  + The starting vertex s is assigned distance = 0, color = gray, and added to the **queue**
  + While the queue isn’t empty:
    - Increment distance
    - Get front of queue u
    - For each of u’s adjacent vertices, if its color = white, set distance, color = gray, and enqueue it
    - Dequeue u and set its color to black
  + At the end, the distance of a vertex v corresponds to the length of the shortest path from s to v
* Running time: O(V + E)
  + Discovers all reachable vertices from starting vertex s
  + Computes shortest distance from s to each vertex
  + Computes a breadth-first tree containing all reachable vertices

## Depth-First Search

* Looks at all the descendants of a vertex before backtracking to other vertices on the same level
  + Initialize all vertices with distance = infinity, color = white
  + For each vertex u in the graph’s vertices, if u’s color = white, call DFSVisit(u)
    - Increment distance
    - Set u’s distance, color = gray
    - For each vertex v adjacent to u, if v’s color = white, call DFSVisit(v)
    - Set u’s color = black
* Running time: O(V + E)

### DFS Edge Classifications

* Gray -> white = **Tree edge**: edge from ancestor -> descendant in a depth-first forest
* Gray -> gray = **Back edge:** descendant -> ancestor in depth-first tree
* Gray -> black = **Forward edge**: non-tree edge from ancestor -> descendant in depth-first tree

**Cross edge**: edges between vertices of same depth, between trees or subtrees

# MST/Prim’s Algorithm

* **Spanning tree**: a subgraph of a graph G that is a tree and contains all vertices of G
  + For a graph with V vertices, there are V-1 edges in its spanning tree
* For a graph G with weight function w that assigns cost to all edges, the **minimum spanning tree** is a spanning tree that minimizes the sum of w(u, v) for all edges
  + Must make V-1 choices to arrive at optimization goal (one choice per edge)
  + After selecting each edge, the problem becomes a sub-problem with one less vertex

## Growing an MST

* Build a set of edges, A while maintaining a set S of vertices in A
* Each edge (u, v) added to A must be a “safe edge” that maintains A as a subset of some MST
  + Add u and v to S
  + Look for light edge – edge connecting a vertex in S to a vertex in V – S with minimum weight

## Prim-Jarnik Algorithm

* Vertex-based algorithm – grows a tree T, one vertex at a time
* Set A covers the portion of T already computed
* For each vertex v outside of A, set v.key() to the minimum weight of an edge connecting v to an edge in A (v.key() = infinity if no edge exists)
* Running time: O(V \* T(extractMin) + E \* T(modifyKey))
  + If Q is an array, O(V \* O(V) + E \* O(1)) = O(V2)
  + If Q is a max heap, O(V \* O(log V) + E \* O(log V)) = O(E log V)

PrimsMST(G, s)

for each (Vertex v in G.V()) {

v.setKey(infinity);

v.setParent(null);

}

s.setKey(0);

Q.init(G.V());

while (!Q.empty()) {

u = Q.extractMin();

for each (Vertex v in u.adjacent()) {

if (v in Q and G.w(u, v) < v.key()) {

v.setKey(G.w(u, v));

Q.modifyKey(v);

v.setParent(u);

}

}

}

## Kruskal’s Algorithm

* Edge-based algorithm – adds edges one at a time in increasing weight order
* Maintains a forest of trees (a set of sets) S, and a set A covering edges in the MST
* Sort edges by weight
* Add minimum weight safe edge (u, v) to A, and merge sets containing u and v; an edge is “safe” if it connects vertices of distinct trees (within separate sets in S)
* Running time: O(E log V)

KruskalsMST(G)

A = null

S.init()

for each (Vertex v in G.V()) {

S.makeSet(v);

}

Sort edges of G.E() in increasing G.w(u, v);

for each (Edge (u, v) in G.E()) {

if (S.findSet(u) != S.findSet(v)) {

A.add((u,v));

S.union(u, v);

}

}

return A;

# Single-Source Shortest Path: Dijkstra’s Algorithm

## Shortest Path Problems

* **Single-source/Single-destination**: find the shortest path from a given vertex to all other vertices
* **Single pair**: find the shortest path between two given vertices (single-source solves this one)
* **All pairs shortest path**: find the shortest path for very pair of vertices (dynamic programming)
* Sub-paths of shortest paths are shortest paths
* Shortest paths don’t contain any cycles (or we could optimize by removing the cycle)

## Dijkstra’s Algorithm

* Like BFS, but considers edge weights
* Can’t handle negative weight edges
* Basic idea:
  + Maintain set S of solved vertices
  + At each step, look for closest vertex u, add u to S, and relax all edges from u
* Running time: O(V \* T(extractMin) + E\*T(modifyKey)) 🡨 just like Prim’s
  + If Q is an array: O(V2)
  + If Q is a max heap: O(E log V)

Dijkstra(G, s)

for each (Vertex v in G.V()) {

v.setd(infinity);

v.setParent(null);

}

s.setd(0);

Q.init(G.V());

while (!Q.empty()) {

u = Q.extractMin();

for each (Vertex v in u.adjacent()) {

Relax(u, v, G);

Q.modifyKey(v);

}

}

Relax(u, v, G)

if (v.d() > u.d() + G.w(u, v)) {

v.setd(u.d() + G.w(u, v));

v.setParent(u);

}

# Single-Source Shortest Path: Bellman-Ford Algorithm

* Dijkstra’s algorithm doesn’t work when there are negative weight edges
* Bellman-Ford algorithm detects negative cycles (returns false) or returns the shortest-path tree
* Running time: O(VE)

BellmanFord(G, s)

for each (Vertex v in G.V()) {

v.setd(infinity);

v.setParent(null);

}

s.setd(0);

for (i = 1 to i = G.V().length - 1) {

for each (Edge (u, v) in G.E()) {

Relax(u, v, G);

}

}

for each (Edge (u, v) in G.E()) {

if (v.d() > u.d() + G.w(u, v)) {

return false;

}

}

return true;

## Shortest Path in a DAG

### Directed Acyclic Graph (DAG)

* A directed graph with no cycles
* A directed graph is acyclic (a DAG) if and only if its DFS has no back edges (gray -> gray)

### Topological Sort

* A linear ordering of the vertices in a DAG, such that for any edge (u, v), u appears before v in the ordering
* Vertices are arranged left to right in the order of decreasing finishing time
  + Call DFS to compute finishing time for each vertex
  + When each vertex is completed, insert it into the front of a linked list
  + Return the linked list
* Running time: O(V + E)

DAGShortestPath(G, s)

for each (Vertex v in G.V()) {

v.setd(infinity);

v.setParent(null);

}

s.setd(0);

topologically sort G

for each (Vertex u in topological order) {

for each (Vertex v in u.adjacent()) {

Relax(u, v, G);

}

}

# All-Pairs Shortest Path

* Compute a table giving the length of the shortest path between any two vertices, and the shortest path themselves
  + Call Dijkstra on every vertex: doesn’t work with negative edges
  + Call Bellman-Ford on every vertex: O(V2E)

## Floyd-Warshall Algorithm

* A dynamic programming algorithm
* Considers the intermediate vertices of a path
* Sub-problem: find the shortest path through a subset of vertices
* Running time: O(n3)

For a directed weighted graph, let W be the adjacency matrix, where each edge (u, v) is either its weight if it exists, or infinity if it doesn’t

* dij(n) in D(n) stores the length of the shortest path between two vertices i and j
* πij(n) in Π (n) stores the predecessor (vertex before j in the shortest path from i to j)

FloydWarshall(W)

n = W.rows().length; // number of vertices

D(0) = W;

Π(0) = { null if i = j or dij(0) = ∞

{ i if i != j and dij(0) != ∞

for (int k = 1; k <= n; k++) {

for (int i = 1; i <= n; i++) {

for (int j = 1; j <= n; j++) {

dij(k) = min{ dij(k-1), dik(k-1) + dkj(k-1) };

πij(k) = { πij(k-1) if dij(k) = dij(k-1)

{ πkj(k-1)  if dij(k) = dik(k-1) + dkj(k-1)

}

}

}

return D(n);

# NP Completeness

## Towers of Hanoi

* Recurrence: T(n) = 2T(n-1) + 1
* Running time: O(2n) 🡨 exponential

## Decision Problems

* Optimization problems can generally be transformed into true/false decision problems
  + For an optimization problem O, with a related decision problem D, if we show that D is hard to solve, O must also be hard to solve

## Reasonable vs. Unreasonable Algorithms

* **Tractable problems**: algorithms bound by a polynomial function nk 🡨 reasonable algorithm
* **Intractable problems**: algorithms with running times above nk 🡨 unreasonable algorithm

## P Complexity Class

* Set of all decision problems that can be solved in polynomial time on real computers
* Examples include: shortest path, relative prime, LCS

## NP (Non-deterministic Polynomial time)

* Set of all decision problems that can be solved in polynomial time on non-deterministic computers
* Problems with efficient verification algorithms, but hard to find solutions

### NP-Hard

* A problem that, if it can be solved efficiently, can be used (as a subroutine) to solve any other NP problem efficiently

### NP-Complete

* NP problems that are NP-hard – if solved efficiently, can solve any other NP problem efficiently
  + Finding a polynomial time algorithm for one NP-complete problem would automatically yield a polynomial time algorithm for all NP problems
  + Proving that one NP-complete problem has an exponential lower bound would automatically prove that all NP-complete problems have exponential lower bounds

## Polynomial Time Reduction

* Suppose we know how to solve a decision problem B in polynomial time (PT)
* Consider a decision problem A, that we want to solve in PT
* If we can transform any instance a of A to an instance b of B, such that:
  + The transformation takes PT
  + The answer for a is “yes” if and only if the answer to b is “yes”
* Then A can be solved in polynomial time – A can be “reduced” to B

A language L1 is **PT reducible** to a language L2, i.e. L1 <p= L2, if there exists a PT computable function f(x) over a range {0, 1}\* such that for all x in {0, 1}\*, x exists in L1 if and only if f(x) exists in L2

### NP-Completeness

* PT reduction shows that one problem is at least as hard as another, within a PT factor
  + If L1 <p= L2, then L1 is no more than a polynomial factor harder than L2
* A language L, subset of {0, 1}\* is **NP-complete** if:

1. L exists in NP
2. L’ <p= L for every L’ that exists in NP

* A language L is **NP-hard** if 2 is satisfied but not necessarily 1
* If any NP-complete problem is PT-solvable, then **P = NP** and any problem in P is NP-complete
  + Equivalently, if any NP-complete problem is not PT-solvable, then no NP-complete problem is PT-solvable
* If **P != NP**, then NP-complete problems cannot be in P