Definition of Algorithm:

* A precise statement to solve a problem on a computer
* A sequence of definite instructions to do a certain job

Pseudo-code consisting of:

* Variables
* Assignments
* Arrays
* Reading/Writing data
* Loops
* Switch/Case
* Function/Procedure
* Begin/End

How to do it? (What is the model?)

* Random access memory model
* Math operations take constant time
* Read/write operations take constant time

Some tips when analyzing algorithms:

* When analyzing an if then else condition, consider the arm that takes the longest time
* When considering a loop, take the sum
* When considering a nested loop, …

Sums of series

* Arithmetic Progression: 1 + 2 + 3 + … + n
* Geometric Progress: 1 + 3 + 9 + … 3k
* AGP: 1 + 2.3 + 3.9 + … (k+1) 3k
* Others: 12 + 22 + 32 + … + n2, Harmonic Series

Useful links:

* <http://en.wikipedia.org/wiki/Arithmetic_progression>
* <http://en.wikipedia.org/wiki/Geometric_series>
* <https://www.boundless.com/algebra/textbooks/boundless-algebra-textbook/sequences-series-and-combinatorics-8/sequences-and-series-53/introduction-to-sequences-224-5904/>
* <http://en.wikipedia.org/wiki/L%27H%C3%B4pital%27s_rule>
* <https://www.youtube.com/watch?v=PdSzruR5OeE>

Big O notation (<=)

* f(n) = O(g(n)) if there exist constants n0 and c such that f(n) ≤ c g(n) for all n ≥ n0.

Big Omega notation (>=)

* f(n) = Ω(g(n)) if there exist constants n0 and c such that f(n) ≥ c g(n) for all n ≥ n0.

Small o notation (<)

* f(n) = o(g(n)) if for any constant c > 0, there exists n0 such that 0 ≤ f(n) < c g(n) for all n ≥ n0.

Small omega (ω) notation (>)

* f(n) = ω(g(n)) if for any constant c > 0, there exists n0 such that f(n) ≥ c g(n) ≥ 0, for all n ≥ n0

Note:

f(n) = O(g(n)) if and only if g(n) = Ω(f(n))

If f(n) = O(g(n)) and g(n) = O(f(n)), then f(n) = Θ(g(n))

f(n) = o(g(n)) if and only if g(n) = ω(f(n))

f(n) = o(g(n)) implies limn→∞f(n)/g(n) = 0

In some cases, we need to use the L’Hopital’s rule to prove the small oh notation. L’Hopital’s rule states that assuming certain conditions hold, \lim_{x\to c}\frac{f(x)}{g(x)} = \lim_{x\to c}\frac{f'(x)}{g'(x)}

PS: 2^n = omega (n^k)

Main Algorithmic Techniques:

* Divide and Conquer
* Greedy Method
* Dynamic Programming
* Graph search methods
* Backtracking
* Branch and bound

Data Structure Definition:

* A data structure is a structure to hold the data, that allows several interesting operations to be performed on the data set.
* The data structure is designed with those specific operations in mind.

Main Data Structures:

* Stack: Last In First Out (LIFO)

Allows 3 operations:

* + Push (a)
  + Pop()
  + Top()

Use an array or linked list to implement stack (use a pointer to point to the top of the stack)

* Queue: First In First Out (FIFO)

Allows 2 operations:

* + dequeue(): Returns the first element
  + enqueue(a): Adds an element a to the end of the queue

Use an array or a linked list to implement queue (keep head and tail indexes or pointers)

Record Struct Object Class Template:

* A record is a built-in structure data type, that allows the packaging of several elements (called fields)
* Every high level language allows the user to define customized records.
  + In C#/Java, this is called “class”.
  + In C, this is called “struct”.

Linked List:

* Singly Linked
  + A special pointer called “first” has the reference to the first record
* Doubly Linked
  + Special pointers called “first” and “last” with references to the first and the last records

Graph:

* A graph G=(V,E) consists of a finite set V, which is the set of vertices, and set E, which is the set of edges. Each edge in E connects two vertices v1 and v2, which are in V.
* Can be directed or undirected
* If (x,y) is an edge, then x is said to be adjacent to y, and y is adjacent from x.
* In the case of undirected graphs, if (x,y) is an edge, we just say that x and y are adjacent (or x is adjacent to y, or y is adjacent to x). Also, we say that x is the neighbor of y.
* The indegree of a node x is the number of nodes adjacent to x
* The outdegree of a node x is the number of nodes adjacent from x
* The degree of a node x in an undirected graph is the number of neighbors of x
* A path from a node x to a node y in a graph is a sequence of node x, x1,x2,...,xn,y, such that x is adjacent to x1, x1 is adjacent to x2, ..., and xn is adjacent to y.
* The length of a path is the number of its edges.
* A cycle is a path that begins and ends at the same node
* The distance from node x to node y is the **length of the shortest path** from x to y.

For Graph Presentation:

If the graph is sparse (very few edges), then adjacency list may be a more efficient choice. Otherwise, a matrix might be a better choice

Tree:

* A tree is a connected acyclic graph (i.e., it has no cycles)
* Rooted tree: A tree in which one node is designated as a root (the top node)
* Leaf is a node that has no children
* Ancestors of a node x are all the nodes on the path from x to the root, including x and the root
* Subtree rooted at x is the tree consisting of x, its children and their children, and so on and so forth all the way down
* Height of a tree is the maximum distance from the root to any node

Binary Tree:

* A tree where every node has at most two children

Binary Search Tree:

* BST is a binary search tree where every node contains a value, and for every node x, all the nodes of the left subtree of x have values <= x, and all nodes in the right subtree of x have values >= x.
* BST supports 3 operations: insert(x), delete(x) and search(x)

Height Balanced BST:

* Red Black and AVL trees are interesting implementations of height balanced BSTs.

Heaps(Priority Queue):

* Very efficient data structure to enforce priority, although do not enforce complete sorting
* Can be max heap or min heap
* Commonly represented using a heap tree (although, can also be a forest)

BTree/2-3 Tree:

* Flexible data structure, where a node has a variable number of children (say between 2 and 4, both including, or between 50 and 100 both including)
* This variable number allows us to leave some “holes” in the tree to fill as insertions happen, thereby allowing insertions without changing the structure of the tree entirely.
* The variable number also allows us to treat deletions without changing the structure.
* 2-3 tree is a specific kind of BTree where each node can have 2 or 3 children.

Union Find (Or Disjoint Set):

* How to maintain sets dynamically – sets can be merged (union), and we want to see which set a particular element is in.
* find(x) 🡪 Identifies the set that element x belongs to
* Union (S1, S2) 🡪 Combines these two sets
* Each set is marked by a leader
* When calling “find” on a set’s member, it returns the leader
* Leader maintains a rank (or height)
* When doing a union, make the tree with smaller height (or rank) to be a child of the tree with the larger height
* Note that this is NOT a binary tree.
* When doing a find, follow that up by compressing the path to the root, by making every node (along the way) point to the root.
* This is not easy to prove, but Union Find with Path compression, when starting with *n* nodes and *m* operations, takes *O(m log\*(n))* time instead of *O(m log n)* time, where the *log\** function is the iterated logarithm (also called the super logarithm) and is an **extremely** slow growing function.
* *log\*(n)* is defined as follows:
  + *0*, if *n <= 1*
  + *1 + log\*(log n)* if *n > 1*
* Application: MTS
  + Using 2 Find operations to check if adding an edge will create a cycle or not.
  + When adding an edge, use a Union Operation
* Divide and Conquer
  + <http://www.cs.cmu.edu/afs/cs/academic/class/15210-f11/www/lectures/03/lecture03.pdf>
  + <http://en.wikipedia.org/wiki/Divide_and_conquer_algorithm>
* Recursive Algorithm  
  <http://en.wikipedia.org/wiki/Recursion_(computer_science>)
* Tail Recursion  
  <http://en.wikipedia.org/wiki/Tail_call>
* Recurrence Relations  
  <http://en.wikipedia.org/wiki/Recurrence_relation>

Divide and Conquer:

* A technique to solve complex problems by breaking into smaller instances of the problem and combining the results
  + Recursive methodology – Smaller instances of the same *type* of problem
* Typically used accompaniments
  + Induction for proving correctness
  + Recurrence relation solving for computing time (and/or space) complexity

A recursive algorithm is an algorithm that calls itself on smaller input.

Algorithm sort (Array a)

Begin

sort (subarray consisting of first half of a)

sort (subarray consisting of second half of a)

do\_something\_else();

End

Recurrence Relation is a recursive formula, commonly used to analyze the time complexity of recursive algorithms

For example

* + T(n) = T(n/2) + T(n/2) + n2
  + T(n) = a T(n/b) + f(n)

D&C Template:

**divide\_conquer(input J)**

**{**

**// Base Case**

**if (size of input is small enough) {**

**solve directly and return**

**}**

**// Divide Step**

**divide J into two or more parts J1, J2,...**

**// Recursive Calls**

**call divide\_conquer(J1) to get a subsolution S1**

**call divide\_conquer(J2) to get a subsolution S2**

**...**

**// Merge Step**

**Merge the subsolutions S1, S2,...into a global solution S**

**return S**

**}**

T(n) = a T(n/b) + f(n)  
Here “a” branches, each with size “n/b”, and f(n) time spent in dividing and merging

D&C Examples:

Binary Search; Merge Sort; Quick Sort

Binary Search: T(n) = T(n/2) + 1

T(n) = O(log n)

Merge Sort:

Algorithm MergeSort (input: A,i,j) {  
 // Divide portion  
 if (j – i < THRESHOLD) {  
 InsertionSort(A,i,j)  
 Return

}  
 int k=(i+j)/2

// Recursive Calls  
 MergeSort(A,i,k)  
 MergeSort(A,k+1,j)

// Merge Calls  
 Merge(A,i,k,k+1,j)

}

T(n) = 2T(n/2) + Θ(n)

T(n) = Θ(n log n)

3 General Approaches:

* Substitution method (Guess and Prove)
* Recursion tree method (unfold and reach a pattern)
* Master theorem

Substitution Method:

* First “guess” the solution
* Then substitute the one in the formula and see

e.g.: Given T(n) = 2 T(n/2) + cn

We first “guess” that the solution is O(n log n)

To prove this using induction, we first assume T(m) <= km log m for all m < n

Then T(n) = 2 T(n/2) + cn <= 2 kn/2 log (n/2) + cn

= kn log n – (k – c)n // log (n/2) = log n – 1

<= k n log n, as long as k >= c

Master Theorem for solving Recurrence Relations:

**Only applies to Recurrence Relations of following type   
   
 *T(n) = aT(n/b) + f(n)***

* **Case 1.** If *f(n) = O(nc)* where *c < logb a*, then *T(n) = θ(n^logb a)*
* **Case 2.** If it is true, for some constant *k* ≥ 0, that *f(n) = θ(nc logk n)* where *c = logb a*, then *T(n) = θ(nc logk+1 n)*
* **Case 3.** If it is true that *f(n) = Ω(nc) w*here *c > logb a*, then *T(n) = θ(f(n))*

T(n) = n^(logba) + f(n) + af(n/b) + … + ak f(n/bk)

We observe that:

* + If f(n) is very small, say a constant, then the first term dominates
  + If f(n) = Θ (n^(logba)), then the T(n) = f(n) log n.

// The log n factor arises because there are ~ log n terms

* + If f(n) is too large, then f(n) terms dominate

QuickSort:

* Select a “partition” element
* Partition the array into “left” and “right” portions (not necessarily equal) based on the partition element
* Sort the left and right sides
* An inverted view of mergesort – spend time upfront (partition), no need to merge later.

Pseudo code:

* quicksort(A,p,r)

if (p < r) {  
 q = partition (A,p,r)  
 quicksort(A,p,q-1)  
 quicksort(A,q+1,r)  
}

T(n) = T(n1) + T(n2) + O(n)

Where n1 + n2 = n – 1

The main Complication in QuickSort is to find a good partition

Two ways to find a good partition:

* Use probability to find a good partition by trying repeatedly
* Find a good partition deterministically

So it all depends upon the kind of the split, and split will likely not be the same each time.

* Worst case – very bad split: O(n2)
* Best case – good split: O(n log n)

**Selection (A,k)**

* **Partition the array A**
* **Suppose, the partition element lands at location k’**
* **If (k == k’) {**
* **return x // Great, we really got lucky.**
* **}**
* **If (k < k’) {**
* **return Selection (A’,k) // Recursive call. Discard A”**
* **}**
* **If (k > k’) {**
* **return Selection (A”,k-k’) // Recursive call. Discard A’**
* **}**

QuickSelect Algorithm- O(n) time algorithm for selecting k-th largest element in an unsorted array.

PS: The constant is rather large, use this algorithm with caution

* Divide the array into groups of 5 (or 7)
* Sort the small groups
* Find the median of the medians
* Partition the array on that median
* Suppose x “lands” at location k’, where 3n/10 <= k’ <= 7n/10
* Depending upon the value of k, either the left side of the partition or the right side will be discarded. In either case, we eliminate at least 30% of data.
* We note that there are 2 recursive calls – 1st call to find the median of medians, and 2nd call after removing 25% of elements
* T(n) = O(n)

Closest pair of points – An interesting O(n log n) time recursive algorithm for finding the closest pair of points.

1. Split the set of points into two equal-sized subsets by finding the median of the x-dimensions of the points.
2. Solve the problem recursively for subsets to the left and right subsets. Thus, there are two recursive calls.
3. Find the minimal distance among the pair of points in which one point lies on the left of the dividing vertical and the second point lies to the right.
4. T(n) = 2T(n/2) + cn + f(n)
   1. If f(n) = O(n2), then, T(n) = O(n2)
   2. If f(n) = O(n), then T(n) = O(n log n)

Greedy Method:

* A technique to build a complete solution by making a sequence of “best selection” steps
* Selection depends upon actual problem
* Focus is simply on “what is best step from this point”
* Applications:
  + Sorting
  + Merging sorted lists
    - Remove the two smallest arrays
    - Add a larger array
    - Keep doing this until we have one array
      * Implement using heap
      * Build the original heap – O(n) time
      * For i = 1 to n-1
        + Remove two smallest elements: 2 log (n)
        + Add a new element log(n) time
      * Total time: O(n log n)
  + Knapsack
  + Minimum Spanning Tree (MST)
    - A spanning tree of a graph is a tree that has all nodes in the graph, and all edges come from the graph
    - Weight of tree = Sum of weights of edges in the tree
    - Selection Policy with Greedy Algorithm: Minimum weighted edge that does NOT create a cycle.
  + Hoffman Encoding

Two Basic Properties of Optimal Greedy Algorithm

* **Optimal Substructure Property**: A problem has optimal substructure if an optimal solution to the problem contains within it optimal solutions to its sub problems.
* **Greedy Choice Property**: If a local greedy choice is made, then an optimal solution including this choice is possible.

When not to use Greedy Algorithm:

You shouldn’t use this algorithm unless you can prove that the solution is optimal

Dynamic Programming:

* A computation/optimization technique
* Computes its solution bottom up by synthesizing them from smaller subsolutions
* Stores the results of subsolutions to avoid recomputation of sub problems to avoid recomputation
* Very useful technique when:
  + Optimal substructure
  + Overlapping subproblems

Conclusion: Dynamic Programming is a methodology of building an optimal solution by first creating subsolutions to subproblems and storing them, potentially losing some space complexity in exchange for much improved time complexity

Comparison of D&C and D&P:

D&C:

* Recursive algorithms
* Top calls Down
* Intermediate results are not saved in explicit storage (e.g., array)
* Only calls subproblems which are actually needed

D&P:

* Also recursive
* Not invoked top down – bottom is called programmatically, before the top is called.
* Intermediate results are saved in explicit storage (e.g., array)
* Solves subproblems that may or may not be needed

Dynamic Programming Template:

**4 Basic Steps:**

1. Develop a mathematical notation that can express any solution and any sub-solution for the problem at hand.
2. Prove that the Optimal Substructure (Principle of Optimality) holds.
3. Develop a recurrence relation that relates a solution to its subsolutions, using the math notation of step 1.
4. Write an algorithm to compute the recurrence relation

E.G.: Matrix Multiplication <https://en.wikipedia.org/wiki/Matrix_chain_multiplication#A_dynamic_programming_algorithm>

* Let Mij denote the cost of multiplying Ai...Aj, (measured in the number of scalar multiplications)
* M(i,i)=0 for all I
* M(1,n) is what we are looking for
* Mij = cost(T) = cost(L)+cost(R)+cost(BC) = Mik + Mk+1,j + rickcj
* We can condition over k to find the “best” value of k – the one that minimizes Mij
* Mij=min{Mik + Mk+1,j + rickcj | i <= k <= j-1}

Algorithm for Matrix Multiplication/Matrix Chain Problem

* Set array M[i,j]
  + Only the upper diagonal values will be used
* Initialize M[i,i] = 0, for all i
* Initialize M[i,i+1] = r[i] \* c[i] \* c[i+1], for all i
* Compute for j = 2 to n-i
  + M[i,i+j] = mink {M[i,k] + r[i]\*c[k]\*c[j] + M[k+1,j]}
* Time complexity of algorithm is O(n3), where n is the number of matrices given
* **The algorithm simply finds the sequence in which to multiply the matrices – not to be confused with matrix multiplication algorithm itself.**

All pairs shortest path (APSP):

* Input: A weighted graph, represented by its weight matrix W.
* Problem: Find the distance between every pair of nodes
* The nodes are numbered 1..n
* D(k)(i,j) = Length of the shortest path from node i to node j using nodes {1..k} as intermediate nodes
* Note that this does not say you use k intermediate nodes.
* Rather, this says, you are allowed to use only the set of {1..k} nodes as intermediate nodes.
* You may use 1 or 2 or any number of intermediate nodes, but that is the *only* set of nodes that you are allowed to use as intermediate nodes.
* D(0)(i,j) = W[i,j]
* D(k)(i,j) either uses the node k or does not
* D(k)(i,j)=min{D(k-1)(i,j), D(k-1)(i,k) + D(k-1)(k,j)}
* Algorithm:
  + for i=1 to n do  
     for j=1 to n do  
     D(0)(i,j) := W[i,j]
  + for k=1 to n do  
     for i=1 to n do  
     for j=1 to n do  
     D(k)(i,j)=min{D(k-1)(i,j), D(k-1)(i,k) + D(k-1)(k,j)}

What we can learn from this implementation:

* Time complexity is O(n3)
* We observe that algorithm is “clean” to implement, in the sense that D(0) can be easily initialized, and computation of D(k) only depends upon D(k-1) array.
* We also observe that once D(k) has been computed, there is no need for D(k-1)
* We can save space by not keeping the old values

Maximum Value Contiguous Subsequence ; Longest Increasing Subsequence (LIS); Coin Change

Graph Traversal Techniques:

A graph search (or traversal) technique visits every node exactly once in a systematic fashion

* BFS
* DFS

Edges of input graph G = (V,E) can be classified in context of the forest G’ produced by the traversal of G

* + Tree edges (aka Discovery Edge): Edge (u,v) if v first discovered by exploring edge (u,v)
  + Back edges: Edge (u,v) connecting a vertex u to an ancestor v. Self loops are also back edges.
  + Forward edge: Edge (u,v) connecting a vertex u to a descendent v.
  + Cross edges: All other edges
* DFS follows the following rules: Select an unvisited node s, visit it, and treat as the current node
* Find an unvisited neighbor of the current node, visit it, and make it the new current node;
* If the current node has no unvisited neighbors, backtrack to the its parent, and make that the new current node  
  Repeat the steps 2 and 3 until no more nodes can be visited.
* If there are still unvisited nodes, repeat from step 1.

[🡪Use of backtracking suggests that a stack is a good data structure for DFS implementation]

* Every node is visited once. Also, every edge (x,y) is "crossed" twice: one time when node y is checked from x to see if it is visited (if not visited, then y would be visited from x), and another time, when we back track from y to x.
* Therefore, the time of DFS is O(n+|E|), or O(n+m)
* If the graph is connected, the time is O(m) because the graph has at least n-1 edges, and so n+m <= 2m +1, implying that n+m is O(m).

Use case: Parenthesis Teorem

Theorem: In DFS, every edge of undirected graph G is either a tree edge or a back edge. (In other words, no forward or cross edges exist in G’ produced by DFS traversal of G).

DFS Applications: Connectivity, Minimum Spanning Trees in Uniformly Weighted Graphs (Time: O(|E|), Biconnectivity(A non-root node x is an articulation point if and only if x has a subtree from which no backward edge originates and ends at a proper ancestor of x; A root node is an articulation point if it has 2 children that are not interconnected)

* Biconnectivity:
  + Each node i will have two new labels: DFN[i] and L[i].
  + DFN[i] ::= sequence in which i is visited. Thus, the first node visited (i.e., the root) has its DFN = 1. The second node visited has a DFN = 2, and so on.
  + L[i] ::= Lowest DFN number of node which can be reached from node i using zero or more tree edges, and then a single back edge; or DFN[i], whichever is lower
  + The DFNs are easy to compute using a simple counter.
  + We note that   
    L[x]=min{   
     DFN[x],   
     {DFN[y] | (x,y) is a back edge},   
     {L[w] | for each child w of x}  
    }

Pseudo Code for Biconnectivity (Graph Traversal PPT Page 24)