# Algorithms Course Notes

## Motivations

**Algorithm**: Algorithms are well defined set of rules for solving computational problems. Algorithms play a key role in modern technological innovation.

* Sorting
* Shortest Path
* Scheduling
* Cryptography
* Computer Graphics
* Networks
* Computational Biology

(Delete when done) Prioritize:

* Definitions/Properties
* Proofs
* Algorithmic Tricks

**Lemma:** A helper theorem used to prove a larger theorem.

**Corollary:** A bonus theorem that follows directly from another theorem. No or little additional proof required.

For any new or existing algorithms, we need to ask ourselves and prove:

* Is it correct? (Correctness)
* Can we do better? (Efficiency)

## Part One

Reference: <https://youtu.be/NqKkxQamroo?si=v1LLAxohiUZRTu0s>

### Multiplication

**Integer Multiplication**

Correctness: Given two n-digit numbers, is the algorithm to compute their product correct?

Efficiency: Given two n-digit numbers, how long does it take to get a result?

* **Elementary Multiplication**
  + Correctness:
    - 
    - <https://pages.cs.wisc.edu/~cs240-1/readings/07_Program_Correctness.pdf>
  + Efficiency: O(N^2)
* **Karatsuba Multiplication**
  + Correctness:
    - 
    - <https://people.cs.uchicago.edu/~laci/HANDOUTS/karatsuba.pdf>
  + Efficiency:
    - T(n) = 3 \* T(n / 2) + O(n) 🡺 a > b^d 🡺 n^(log(a)) 🡺 n^(log(3))
    - O(n): additions and shifts have linear complexity.
    - Better for numbers with less than 1000 digits.
  + **Algorithm Notes:**
    - Karatsuba Algorithm Split
      * **x = a \* 10^(n/2) + b**
      * **y = c \* 10^(n/2) + d**
      * **x \* y = (a \* c) \* 10^n + 10^(n / 2) \* (ad + cb) + bd**
    - Clever Grouping
      * The middle term ( **10^(n / 2) \* (ad + cb)** ) is cleverly calculated by subtracting the already computer terms: **ac** and **bd.**
      * **(a + b) \* (c + d) = ac + ad + bc + bd**
      * **(ad + cb) = (a + b) \* (c + d) – ac – bd 🡺 Karatsuba((a + b), (c + d)) – ac - bd**
    - Karatsuba Recursive Calls
      * n == Max (length of x and y).
      * karatsuba(12, 56) Output: (5 \* 100) + (33 - 5 - 12) \* 10 + 12 = 672
        + a = 1, b = 2, c = 5, d = 6
      * karatsuba(1234, 5678)
        + a = 12, b = 34, c = 56, d = 78
      * karatsuba (46, 134)
        + x = (12 + 34), y = (56 + 78)
        + a = 4, b = 6, (c = 13, d = 4) ??? what to do with 3 digit numbers ??? Round up first piece, round down last piece
      * Base Case: One or both numbers have length == 1.
    - **Why This Saves Effort:**
      * We calculate three smaller multiplications (ac, bd, (a + b) \* (c + d)) instead of four.
      * The subtraction and the multiplications by powers of ten involve simple shifts and additions, which are less computationally expensive than direct multiplication.
      * Karatsuba realized we could do something clever with the distributive property to reduce the number of calculations.
    - **The essence:**
      * Karatsuba is a clever rearrangement of the standard multiplication that lets us calculate a large multiplication using smaller multiplications and some simple additions and subtractions.
* **Fast Fourier Transform**
  + Correctness:
    - Playlist: <https://www.youtube.com/watch?v=wmCIrpLBFds&list=PLHXZ9OQGMqxdhXcPyNciLdpvfmAjS82hR>
  + Efficiency:
    - O(N\*log(N))
    - Where N is the number of digits (or the degree of the polynomial).\
    - The overhead manipulations make this algorithm the most optimal when dealing with numbers such that N > 1000.
  + **Algorithm Notes:**
    - The FFT is a speed-up trick for a mathematical operation known as Discrete Fourier Transform (DFT). DFT breaks a signal S(n) down into the frequencies that make it up. But DFT takes a lot of time for larger datasets.
    - FFT works faster (especially on datasets that have sizes equal to a poser of 2).
    - Can be used to multiply number represented as polynomials.
    - Convert Multiplication of Polynomials 🡺 Multiplication in Frequency Domain 🡺 Normal Multiplication.
    - Steps:
      * Represent polynomial as a list of its coefficients.
        + Pad lists with 0 to make them the same length.
      * Apply FFT to transform coefficients from Time Domain 🡺 Frequency Domain.
      * Multiply Frequency Domain lists.
      * Apply inverse FFT to get result in Time Domain. Final Answer.

### Sorting

**Sorting**

Correctness: Given a list of n comparable objects, does the algorithm return the list in sorted order?

Efficiency: Given the list of n comparable objects, how long does it take to sort them and return a result?

* **Merge Sort**
  + Correctness:
    - ****
    - <https://www.cs.mcgill.ca/~dprecup/courses/IntroCS/Lectures/comp250-lecture16.pdf>
  + Efficiency:
    - O(N log N)
    - T(N) = 2 \* T(N / 2) + O(N)
  + **Algorithm Notes:**
    - Merge Sort is a sorting algorithm that recursively sorts a list by splitting the size in two and recursively calling itself on the smaller subproblem. When the problem is small enough (i.e. the length of the list is 2 or 1) the algorithm sorts the smaller subproblem and returns it. The algorithm combines the two recursive calls by linearly scanning through the two, now sorted, subarrays and adding their elements to a larger array in sorted order based on simpler comparisons for the beginning of each list.
* **QuickSort**
  + Correctness
    - ****
    - <https://www.cs.cmu.edu/afs/cs/academic/class/15451-s07/www/lecture_notes/lect0123.pdf>
  + Efficiency:
    - O(N\* Log(N))

**Mentionable:**

* Insertion Sort
  + Split the array into two parts, sorted and unsorted. Take the first element in the unsorted array and insert it into its correct position in the sorted array until all elements in the unsorted array have been placed.
* Selection Sort
  + Iteratively search for the smallest element in the unsorted array and place it at the end of the sorted half of the array.
* Bubble Sort
  + Repeatedly compare adjacent elements to see if they are in the correct order. If they are not, swap their positions and continue. Larger elements will bubble down, and smaller elements will bubble up.

### Asymptotic Runtime

Given f(n) and g(n), pick c and n0:

* Big-O f(n) = O(g(n)):
  + f(n) is bounded above by a constant multiple of g(n).
  + | f(n) | <= c \* | g(n) | for some c and all n >= n0
* Big-Omega f(n) = Omega(g(n)):
  + f(n) is lower bounded by a constant multiple of g(n).
  + | f(n) | >= c \* | g(n) | for some c and all n >= n0
* Big-Theta (Both Big-O and Big-Omega):
  + f(n) is bounded above and below by two constant multiples of g(n).
  + c1 \* | g(n) | <= | f(n) | <= c2 \* | g(n) | for some c1, c2 and all n >= n0
* **The Master Method**
  + Correctness
    - ****
    - <https://www.khoury.northeastern.edu/home/jaa/CSG713.04F/Information/Handouts/master.pdf>
  + Efficiency
    - A math equations and formulas

      Description automatically generated with medium confidence
    - **T(n) = a \* T(n / b) + O(n^d)**
* **Stirling’s Approximation**
  + Correctness
    - (My Notes)
    - Notes link

**Guiding Principles for Algorithm Design**

* Think of your Algorithm’s runtime in terms of Worse-Case, Average-Case, and Best-Case terms.
* Don’t consider constant time factors / constants, unless you are looking for minor optimizations.
* Focus on running time for large input sizes.
* A Fast Algorithm is a algorithm whose worse case runtime grows slowly as the input size increases.

### Divide & Conquer

This technique splits the problem in half (Divide), solves those problems recursively (Conquer), and combines the solution in the caller function (Combine).

**Counting the Number of Inversions**

By using the logic for Merge Sort, but with some additional logic, we can **count the number of inversions on the left side** of the array, **count the number of inversions on the right side** of the array, and **count the number of split inversions** (numbers on the left that are strictly greater than numbers on the right).

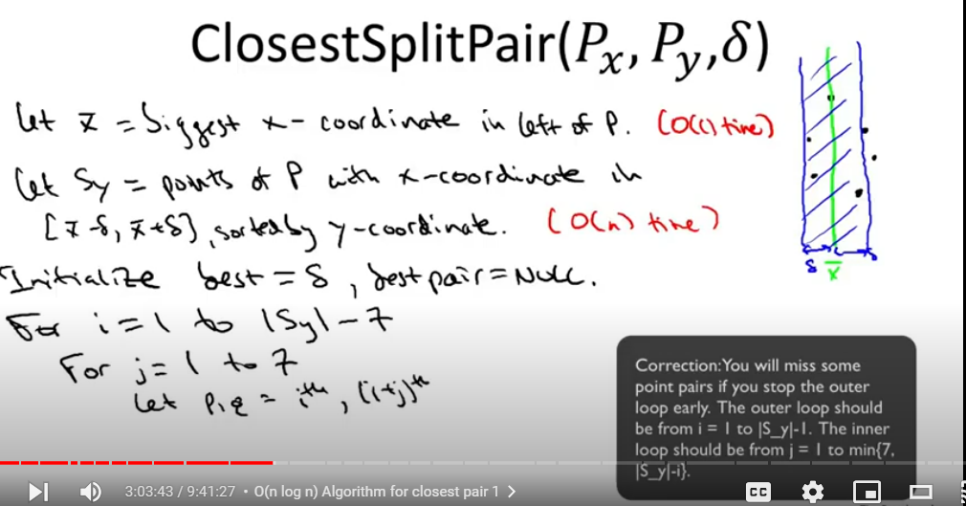
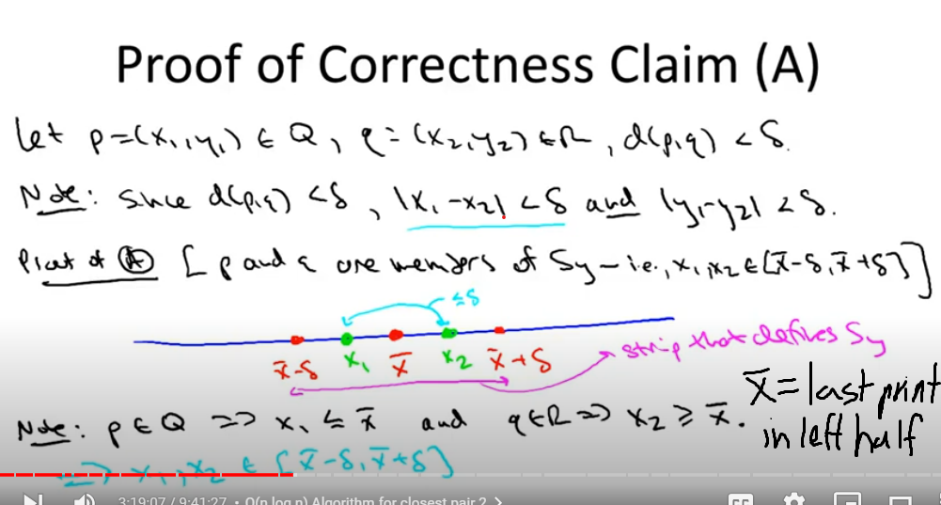
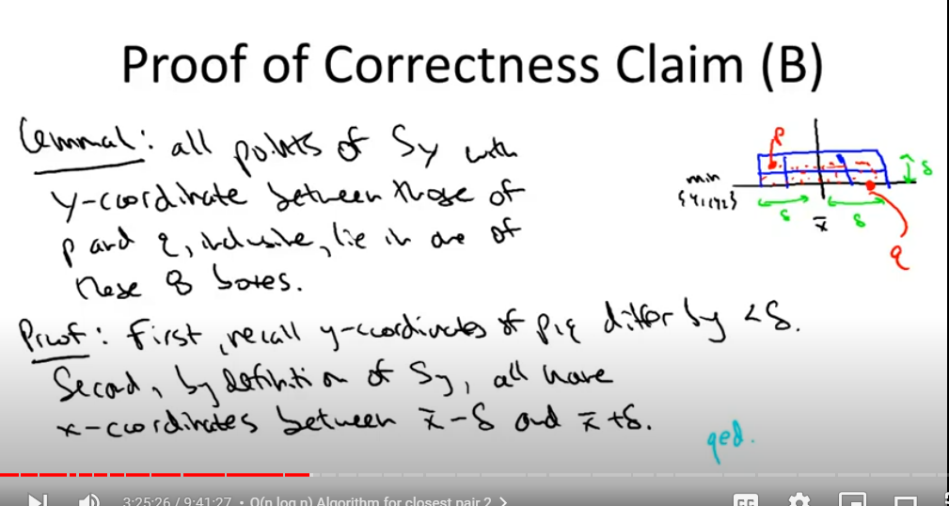
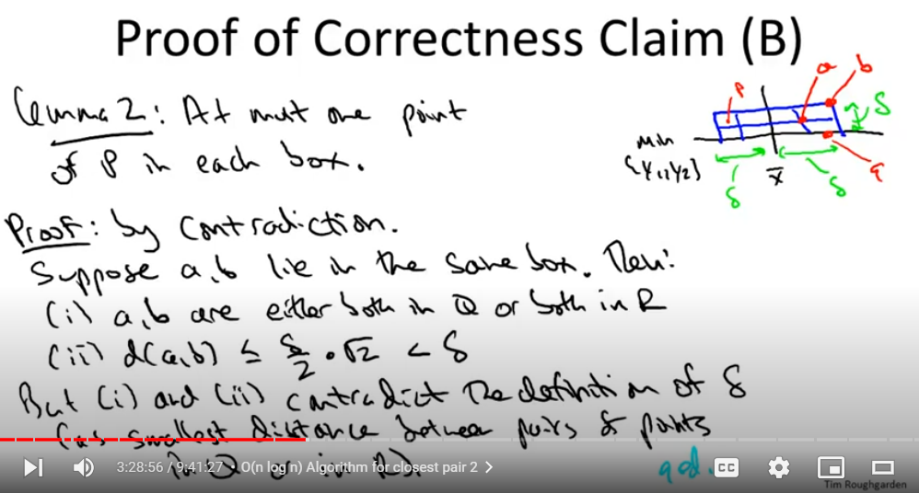
To accomplish this, when we make the recursive call on both sides, we also sort them (essentially performing Merge Sort). There are some cases to consider:

1. **Base Case**: Length of the array is 0 or 1. **Return the array.**
2. **Case 1:** When merging back, the current element on the right half of the array at index j is smaller than the rest of the elements on the left side starting at index i.
   1. **This implies that the value at index j on the right is inverted with all the values on the left that have not been processed yet.**
   2. i = 2, j = 2, l = [1, 2, 6], r = [4, 5, 7]
      1. Here, we take 6 from the left, but we know that we took 2 values from the right before we took 6, so that counts two inversions.
3. **Case 2:** When merging back, the first element of the right side of the array is not processed until all elements on the left side of the array are processed.
   1. **This implies that there are no inversions.**

* **Closest Pair Problem** (Computational Geometry)

Continuing to utilize the Divide and Conquer method, we can now tackle the problem of finding the pair of points that are the closest to each other in a given set.

To do this, we will split the problem in half again and solve for the pair of closest points in these splits. **The challenge comes** when **deciding how to split these pairs, how to sort these pairs, and what to do if the closest pair of points are on two different sides of the split**.

* + Correctness
    - 
    - <https://www21.in.tum.de/~nipkow/pubs/ijcar20-closest.pdf>
  + Efficiency
    - O(N\*Log(N))
  + 
  + 
  + 
  + 

## Part Two

Reference: <https://youtu.be/ahvrc4tZbTE?si=1S-2hYyvTbHwSzA2>

### Graphs

V 🡺 Vertex, Vertices, Nodes, (n/N)

E 🡺 Edge, Edges, (SPARSE🡺MIN = V – 1, DENSE🡺MAX = V\*(V - 1) / 2), (m/M)

Edge Types 🡺 Directed, Undirected

Graph Representations 🡺 Adjacency List,(O(M + N) Space) Adjacency Matrix (O(N^2) Space)

**Cuts In Graphs**

A cut in a graph is a partitioning of the vertices in graph G such that G is split into two, non-empty sets. Not all edges must cross the cut and connected vertices can be in the same set. Crossing the cut implies that the tail (end of a directed edge) is in set B and the head (the end of the arrow) is in set A.

**The Minimum Cut Problem**

Given graph G, return the minimum set of edges that cross the cut between the two non-empty sets.

**Karger’s Contraction Algorithm**

* Correctness
  + 
  + <https://courses.cs.washington.edu/courses/cse521/16sp/521-lecture-1.pdf>
* Efficiency
  + This algorithm has a high failure rate, but can be used as a trial run algorithm, meaning that the more it is run, the more likely it is that the smallest we’ve seen so far is the correct algorithm.
  + O(N^4 \* log(N)) algorithm’s runtime times the number of repeated trials needed.

**Breadth First Search**

This algorithm uses a queue to process the nodes in a graph in a layered fashion.

Applications:

1. Exploring nodes in layers
2. Computing shortest paths
3. Computing connected components in undirected graphs

* Correctness
  + 
  + <https://www.cs.dartmouth.edu/~deepc/Courses/S19/lecs/lec13.pdf>
* Efficiency
  + O(V + E)

**Depth First Search**

This algorithm uses a stack to process the nodes in a graph by fully traversing each path from a node before returning to the source.

Applications:

1. Maze-like searching
2. Computing topological orderings
3. Connected components for directed graphs

* Correctness
  + 
  + <https://www.cs.dartmouth.edu/~deepc/Courses/S19/lecs/lec11.pdf>
* Efficiency
  + O(V + E)

**Dijkstra’s Algorithm**

A graph search algorithm to find the shortest path in a graph from a source to a target. The algorithm will only work correctly on a graph with positive edge weights.

* Correctness
  + 
  + <https://web.engr.oregonstate.edu/~glencora/wiki/uploads/dijkstra-proof.pdf>
* Efficiency
  + O((V + E) \* log(V)) 🡺O(E \* log(V))

**Topological Sort**

Topological sorting is a way to order the vertices in a DAG such that for any edge (u, v) the task of u will come before the task of v.

* Correctness
  + 
  + <https://tildesites.bowdoin.edu/~ltoma/teaching/cs231/fall14/Lectures/11-topsort/topsort.pdf>
* Efficiency
  + O(V + E)

**Tarjan’s Algorithm**

This algorithm uses DFS to find and label all Strongly Connected Components (SCCs) in a graph.

* Correctness
  + ****
  + <https://activities.tjhsst.edu/sct/lectures/1516/SCT_Tarjans_Algorithm.pdf>
* Efficiency
  + O(V + E)

**Kosaraju’s Algorithm**

This algorithm correctly finds the nodes in the graph where running DFS will produce the Strongly Connected Components in the graph.

By reversing the edges in G to get G’ and ordering the second DFS by finishing times of the first DFS in decreasing order (topological sort) we are identifying bidirectional paths in the graph. If a SCC exists in G then it also exists in G’.

By ordering the second DFS traversal in G by the DFS finishing times in G’, we are iteratively identifying SCC sinks in G.

Good DFS calls to find SCCs will make calls to nodes in a sink SCC in the order that they appear. I.e. a form of traversing the reverse topological sort.

1. Reverse the graph and run DFS
2. Order the nodes by DFS completions of the reverse graph.
3. Run DFS on this decreasing order in the normal graph.

* Correctness
  + 
  + [SCCcorrect.dvi (mcgill.ca)](https://www.cs.mcgill.ca/~pnguyen/251F09/SCCcorrect.pdf)
* Efficiency
  + O(V + E)

**Heaps**

Heaps are a data structure represented by an array, but thought of as a tree. The operations for a Heap (Bubble Up, Bubble Down, Insert, Delete) offer quick flexibility for implementing a priority queue that can constantly track the min/max in a set and quickly remove it from the queue.

* Correctness
  + 
  + [recitation10-8.pdf (mit.edu)](https://courses.csail.mit.edu/6.006/fall10/handouts/recitation10-8.pdf)
* Efficiency
  + Log(N) for adding, getting the max/min
  + N \* Log(N) for sorting
  + Practice? <https://courses.csail.mit.edu/6.006/fall10/handouts/recitation10-8.pdf>

**Deleting from a BST**

1. Case one is deleting a note with no children
   1. Just delete the node.
2. Case two is deleting a note with one child
   1. The single child takes the place of the deleted node.
3. Case three is deleting a note with two children
   1. In the case of deleting a node with two children, the node being deleted should be replaced with its immediate predecessor (which will always be a leaf or a node with one child), reducing the problem to deleting a node with one child.

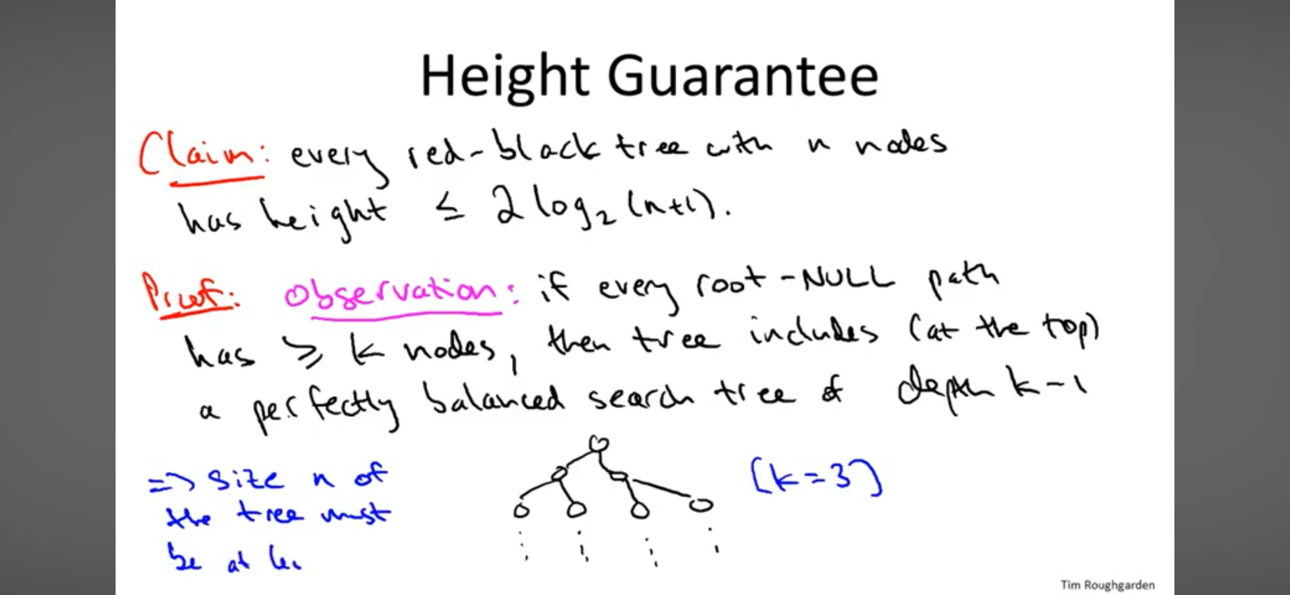
**Balanced BST rotations**

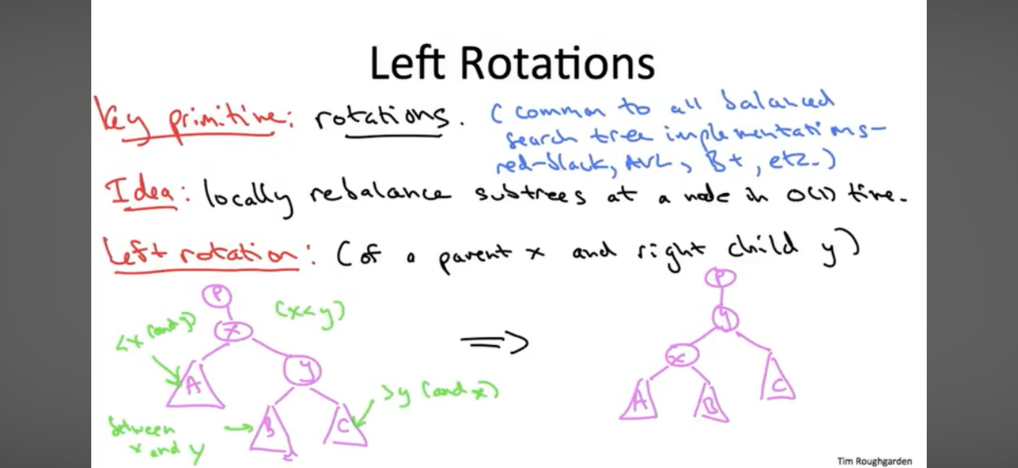
1. BST rotations are a parent child pair operation.
   1. In the case of a left rotation, the right child’s left child becomes the parent's new right child.
   2. In the case of a right rotation, the left child’s right child becomes the parent's new left child.
   3. In both cases, the parent becomes the new child of the old child.

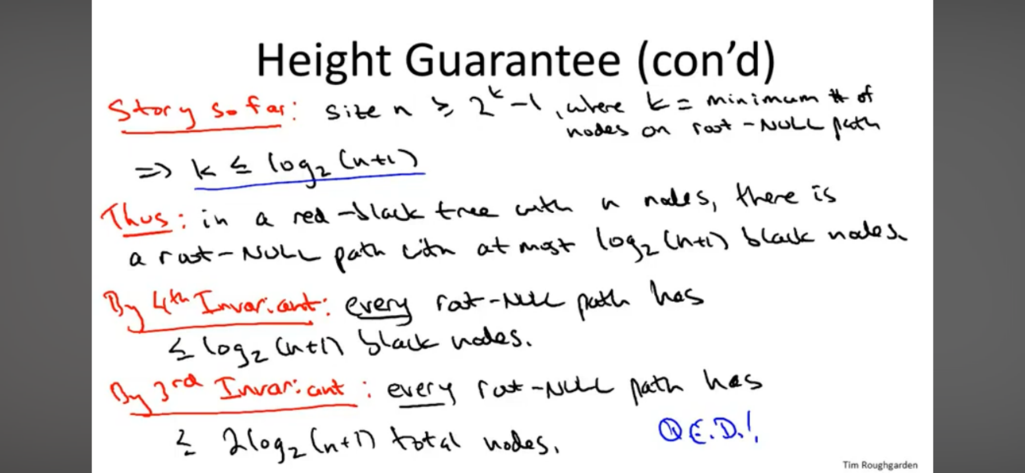
**Red-Black Trees**

Red-Black Trees are a type of balanced binary search trees that maintain a maximum height of Log(N) + 1 to keep operations like insert, search, delete, etc. fast. By using a balanced BST, users do not need to worry about the order in which data is added to the trees. RB Trees maintain the following invariants:

1. Each node is Red or Black
2. Root is always Black (Newly added nodes are always red) (A tree with two nodes and a red root would violate #4)
3. No two reds can be in a row. (If a node is red, both its children are black)
4. Every path from root to leaf has the same number of black nodes.

* Efficiency
  + O(Log(N)) for insert/delete/search operations.
* Correctness



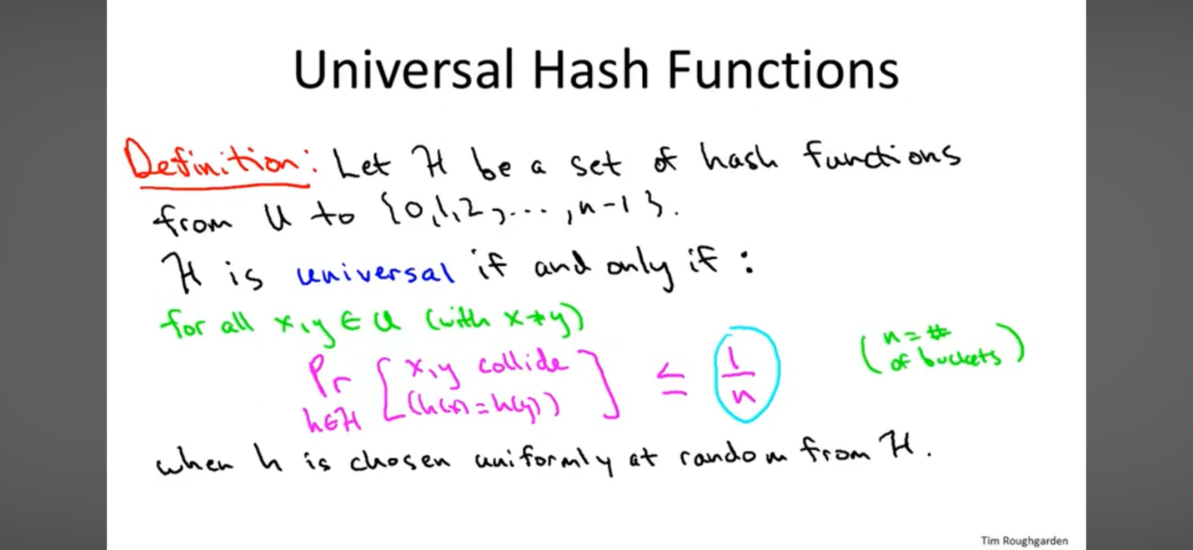


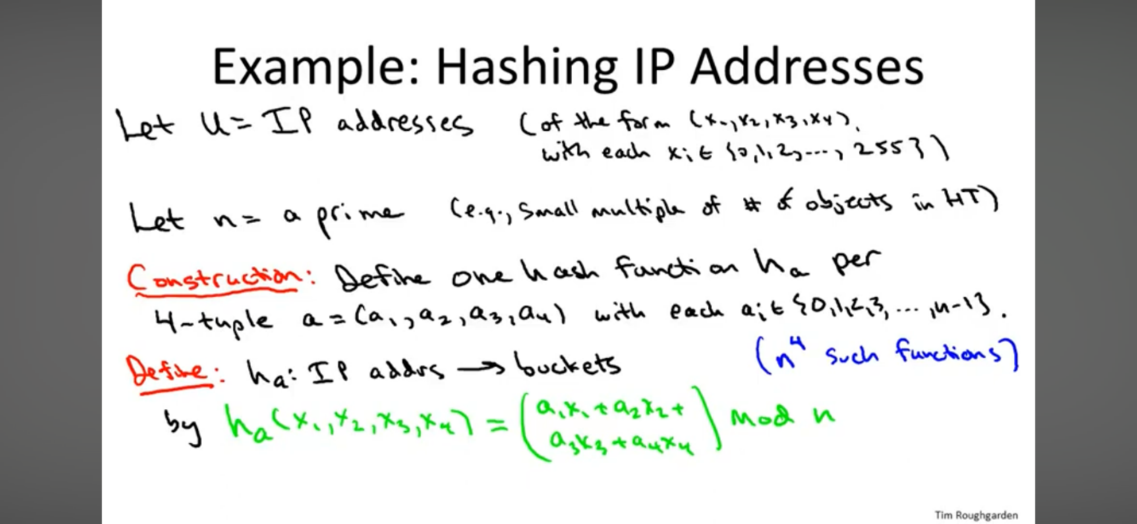
**HashTables**

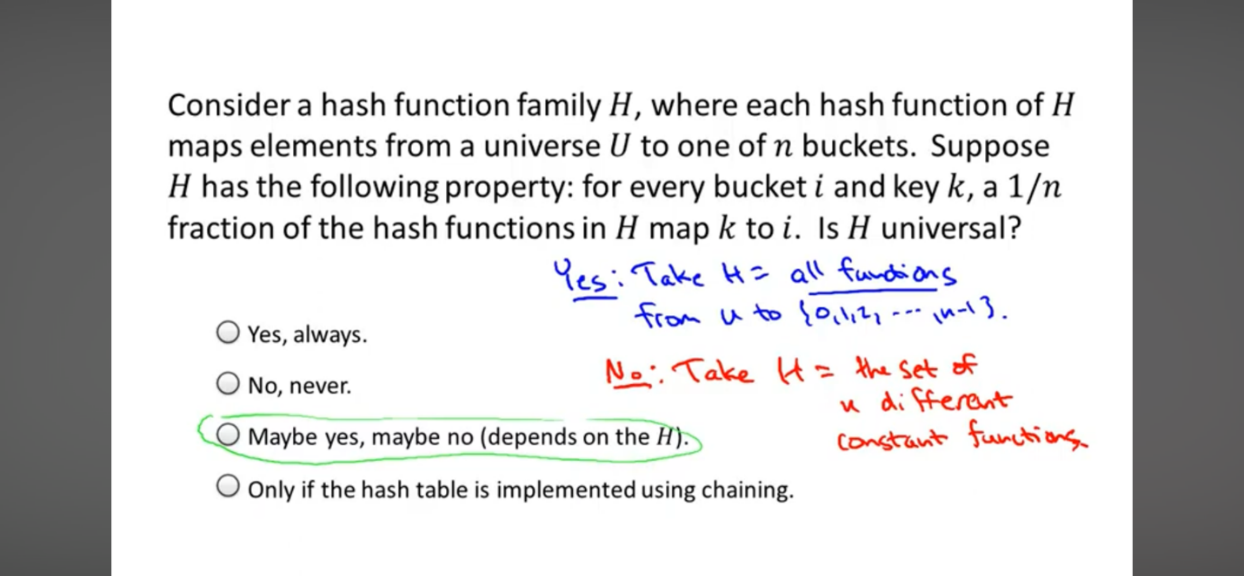
A hash table can be thought of as an array where the contents of the array at a specific index can be quickly found based on the contents there. For example, if you’re looking for a name in an array of size 26 and each letter represents an index, you’ll know without linear search that names that start with Z are at the end of the array. Using a linked list, several items can map to several positions in the array. Hash Tables are great for existence checks.

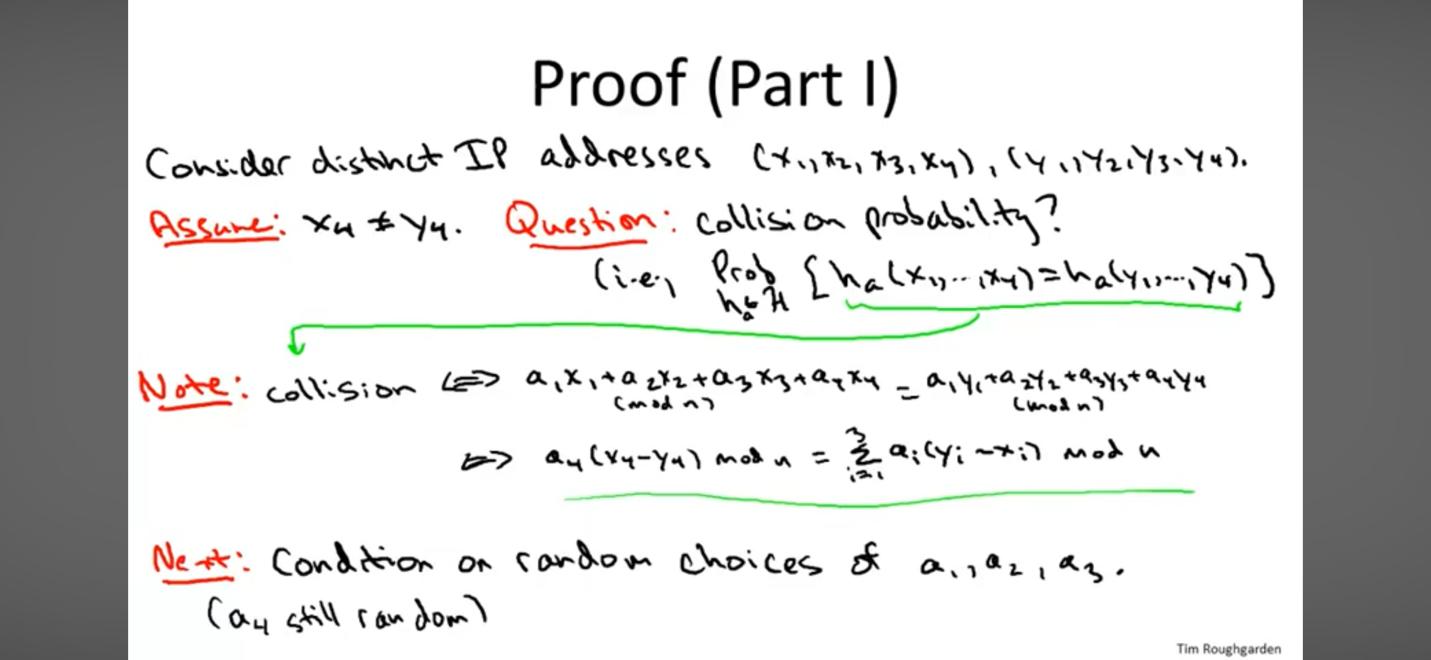
**Operations**: Insert, Delete, Search

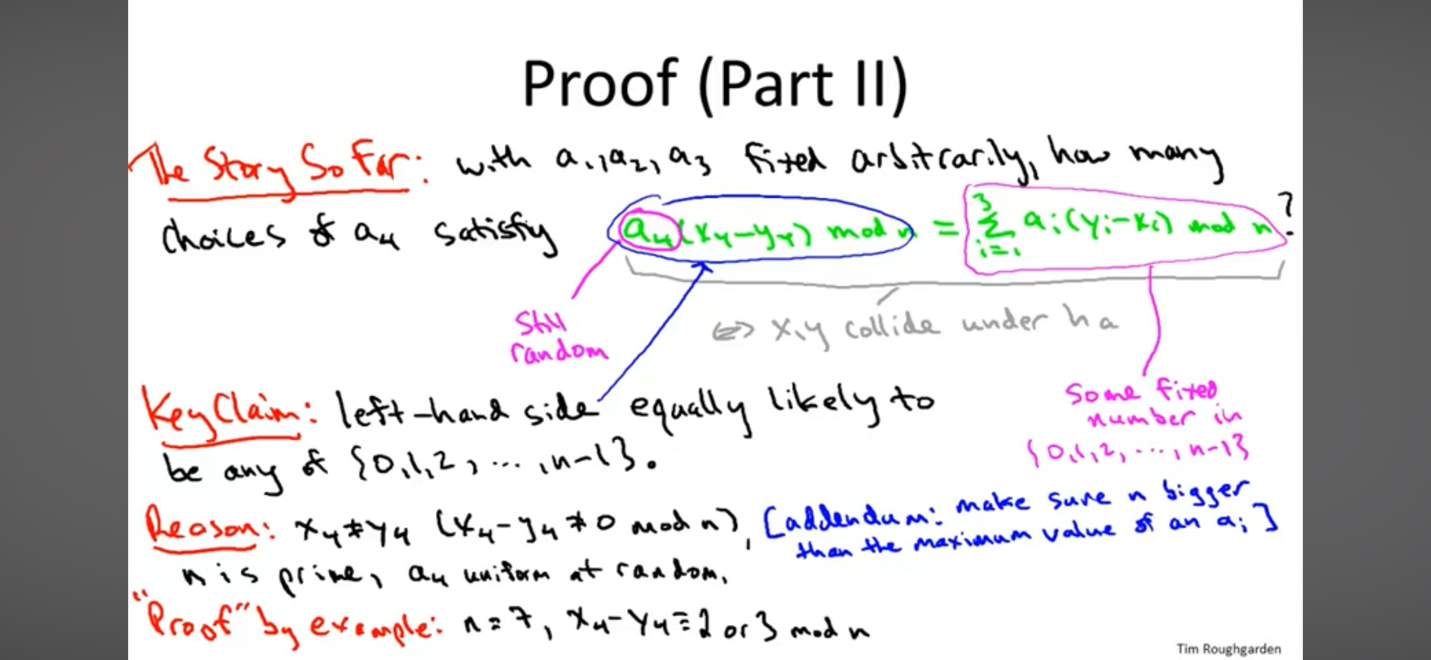
* Correctness (in terms of Hashing)











* Efficiency
  + Constant to Linear Search O(1) to O(N)
  + Constant to Linear Insert O(1) to O(N)
  + Constant to Linear Delete O(1) to O(N)

Additional Proof to work on

* Universal Hashing

## Part Three

Reference: <https://youtu.be/nUIbHblMop4?si=uzZqLwlOt7Rs7Jln>

Greedy Algorithms, Dynamic Programming, Programming Paradigms and Applications

#### Greedy Algorithms

***Topics to Revisit: (Delete when done)***

* Hopcroft-Ullman 6:57:00
* Tarjan’s Bound 7:17:00

**Scheduling**

The Exchange Argument

Why must there be two consecutive values in the processing order of the algorithm (I, j) where I < j? Because there is only one permutation where these consecutive values are strictly increasing. All other permutations must have at least one inversion. The most inversions possible are N choose 2.

There is a point in the two differing solutions where they begin to diverge. Why would they diverge? What aspect of the algorithm would cause this to happen?

* Correctness
  + (See Proofs folder in Algorithms Github Repository)
  + https://www.cs.cornell.edu/courses/cs482/2007su/exchange.pdf

**Weighted Scheduling**

* Correctness

* + 
* Efficiency
  + O(N\*Log(N))
* Outstanding questions
  + In the proof, why is finding a better path and proving that the path can be improved enough to prove the correctness of the greedy algorithm?
  + Why are the switched jobs consecutive?

**Minimum Spanning Trees**

To prove the correctness of a greedy, MST algorithm it must be proved that:

1. Induction: The current set of edges in the spanning tree span all of the vertices in the spanning tree.
2. The output is a Spanning Tree (although it may not be minimum)
   1. Contradiction: This is done by using the steps of the algorithm to prove that the graph is connected, then the algorithm will produce a spanning tree. (Empty Cut Lemma)
   2. Logical: It also needs to be established that the graph does not produce a cycle (Lonely Cut Corollary)
3. The Spanning Tree produced is the minimum (Exchange Argument)
   1. Recall that spanning trees are not unique
   2. (By The Cut Property the cheapest edge along the cut must be in the spanning tree)
   3. Contradiction: Assume e is the edge chosen originally and there’s an edge e’ that if added to the MST will produce a cycle. Replace e with e’

**The Cut Property**

The Cut Property Proof

Contradiction: If there are two solutions, choose the first point at which G and O differ. What are the cases that would cause the solutions to diverge based on the algorithm?

* Correctness
  + 
  + <https://www.eecis.udel.edu/~saunders/courses/320/10s/cut-property.pdf>

**Prim’s Algorithm**

Algorithm #1: Store edges crossing the cut in the heap (at most V – 1) of them

Algorithm #2: Store the vertices in the heap with the cheapest edge incident to it as the key.

1. Requires deletion and insertion for the UpdateKey when a new vertex is absorbed, and a cheaper path may be possible to a vertex already crossing the cut.
2. Produces cheaper constants.

* Correctness
  + 
  + <https://home.uncg.edu/cmp/faculty/srtate/330.f16/primsproof.pdf>
* Efficiency
  + O(E \* Log(V)) or O(E \* V)

**Kruskal’s Algorithm**

**For Correctness, we need to show:**

1. T is connected 🡺 spanning tree exists.
2. The spanning tree has no cycles
3. Fix a cut and use the Cut Property to show that an edge along the MST must cross a cut.

* Correctness
  + 
  + <http://people.qc.cuny.edu/faculty/christopher.hanusa/courses/634sp12/Documents/KruskalProof.pdf>
* Efficiency
  + O(E \* Log(V)) with Union Find, O(V \* E) otherwise

Why M\*log(N) and not M\*log(M)?

N^2 <= M 🡺 log(M^2) 🡺 2 \* log(N)

**Disjoint Set / Union Find Algorithm**

**Motivation:** How can we quickly find out if there is a cycle in a graph?

**Operations:**

Find(x) = Find the set that x belongs to by returning its representative.

Rank(x) = The maximum number of hops from any leaf to x. Once a root becomes the child of another set, its rank will never be updated again. (Any union will go directly to the root)

Union(C1, C2) = Change all parent pointers of C1 and C2 to point to the same set representative.

Find( with Path Compression) = After a successful find, change the node’s parent pointer to point directly to the set’s overall parent pointer. (This includes all members traversed on the way to the set representative) Calls to find will update the parent pointer but not the rank.

Lazy Union(x, y) (Union by Rank) = Only update the parent pointer of the member with the lowest rank. If they are equal, pick at random.

**Rank Properties**

1. Find(x) operations do not affect the rank.
2. Rank(x), if it is not a root, is strictly less than the rank of x’s parent.
3. Only ranks of the root can increase over time (the ranks of the children are frozen in time)
4. There are at most n/(2^r) members in a set with rank r.

**Rank Lemma:**

Given an arbitrary sequence of unions and finds, for every integer r, there are at most n/(2^r) objects with rank r. r is bounded above by Log(N)

**Claim 1:** If x and y have the same rank, then their subtrees are disjoint.

* **Proof by Contrapositive:** If the sub trees are not disjoint 🡺x and y do not have the same rank
  + **Ranks along this path are strictly increasing.**

**Claim 2:** The subtree of a rank r object has size at least 2^r. Size is bounded below by 2^r

* **Induction:** On the number of union operations.

**Result:** There are at most n/(2^r) such groups with rank r.

**Claim 1 + Claim 2 = Rank Lemma**

**Log\*(N) =** the number of times you have to apply Log to N before the result is less than or equal to 1.

* Correctness
  + 
  + <https://www.cs.cmu.edu/~avrim/451f13/lectures/lect0912.pdf>
* Efficiency
  + Find O(1), O(log(n)) 🡺 O(log\*(N)) with Path Compression on N Union and Find operations.
  + Rank O(1) 🡺 metadata
  + Union O(L1 + L2), O(1) with Union by Rank

**Huffman Encoding**

Generating variable length codes for messages based on the frequency of the character that appears. Codes should be easy to translate and should not be confused with other possible messages (I.e. it’s hard to tell where one code ends and the other begins)

**Properties:**

* **Prefix Free Codes:** for every pair of encodings, neither is a prefix of the other.
* **Variable Length Encoding:** Allows us to take advantage of non-uniform frequencies.
* **Preservation of Objective Function Value:** The encoding of any two values A and B as siblings in a tree instead of as a single node in a tree preserves the average encoding calculation up to a constant which is the sum of their probabilities. (i.e. independent of depth.)
  + There’s a one-to-one correspondence between feasible solutions between trees for the smaller subproblem and feasible solutions to the original problem where A and B are siblings.
  + This tree correspondence preserves the encoding length up to a constant factor.
  + Minimizing all average encoding lengths for feasible solutions where A and B are siblings.

**Building an Encoding Tree**

With a right heavy tree, left branches represent a 0 and right branches represent a 1. Building from the bottom up (I. e from leaves to root) is a much more efficient way order the codes, to merge the subtrees, and build the tree.

**Average Length**

If L(T) = the average length of a single character in the tree then L(T) = the sum of p[a character appears] \* the number of bits used to represent that character.

* Correctness
  + 
  + <https://www.cs.umd.edu/class/fall2017/cmsc451-0101/Lects/lect06-greedy-huffman.pdf>
* Efficiency
  + O(N\* Log(N)) for priority queue operations

**Network Flow - Edmonds-Karp / Ford-Fulkerson Algorithm (Just for Fun)**

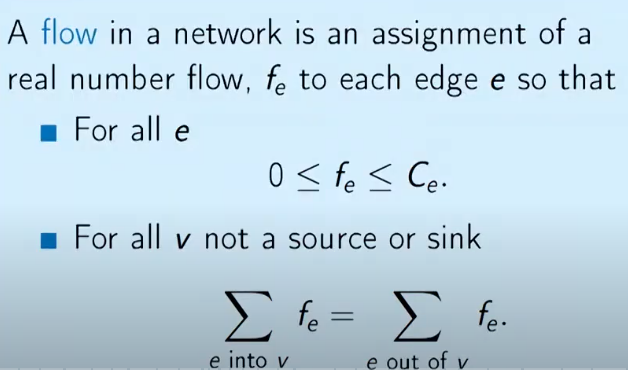
A specific implementation of the Ford-Fulkerson algorithm for finding the maximum flow in a flow network. It uses **breadth-first search (BFS)** to find augmenting paths.

**Input:** A DAG with edge weights.

**Output:** The maximum flow |f|

**Properties**

1. Non-negative flow: The flow from one vertex to the next is non-negative (i.e. non-negative edge weights)
2. Conservation of Flow: For all non-sink & non-source vertices, all flow going into a vertex must leave it (i.e. it does not accumulate)
3. Size of Flow: Sum of the flow of the incoming edge(s) – Sum of the total flow going to outgoing edges.
   1. This is 0 for vertices that are not sources or sinks, negative for sources, and equal to the max flow for the sum over the sinks.
4. Residual Graph: If there is a path from source to sink then there is more flow that can be added to the existing flow.

****

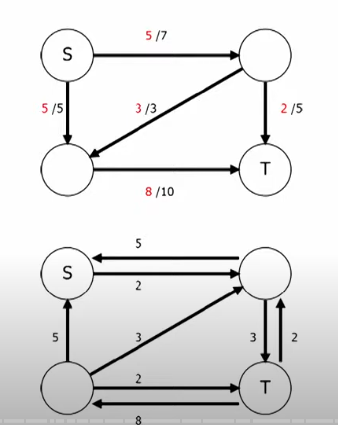
**A close-up of a math problem

Description automatically generated**

**Residual Graph**

This is an augmented version of the original graph after sending some flow down the network. The original edges are not their original capacity minus the amount of flow we were able to send along that edge from the source to the sink. The pair of vertices involved now also have an edge going in the reverse direction that has weight equal to the amount of flow that successfully traveled along that edge so far.

Flows in the residual graph on backwards edges should be subtracted and flows going in the forward direction should be added.



Optimality can be proven by finding the bottleneck of the flow. (i.e. Min Cut)

FF uses DFS to find the next path.

EK uses BFS to find the next path.

* Correctness
  + ???
  + https://david-kempe.com/teaching/edmonds-karp.pdf
* Efficiency
  + EK: O(E^2 \* V) 🡺 Distance between s and t can only increase V times. Each iteration can have at most E new saturated edges. Upper bound of E \* V saturated paths where each path takes at most E time to traverse and update.
  + FF: O(E \* |f|) 🡺 E is the time to compute the residual graph and finding new path using DFS

#### **Dynamic Programming**

If we had an optimal solution to the problem, what would it look like? What can we say about it?

1. Identify subproblems
2. Express the larger problem as a combination of the solutions to the smaller subproblems.

**Weight Independent Set in Path Graphs**

Given a graph with non-negative weights on vertices (not edges), output an independent set such that no two vertices are adjacent, avoiding consecutive pairs of vertices.

* Correctness
  + 
  + <https://www.cs.cornell.edu/courses/cs4820/2024sp/notes/08_WIS-reading.pdf>
* Efficiency
  + O(V)

**Knapsack Problem**

Given a sequence of items with values and weights, find the maximum total value that can be obtained with a maximum capacity of weight W.

* Correctness
  + 
  + <https://courses.cs.duke.edu/fall17/compsci330/lecture5note.pdf>
* Efficiency
  + O(nW)

## Part Four

Reference: <https://youtu.be/M5JzG8oDicA?si=OtUu9DIzKS2HGT0W>

***Topics to Revisit: (Delete when done)***

* Knapsack Heuristic 4:33:00

**Bellman Ford**

This algorithm can compute the shortest paths in a graph with negative edge weights, assuming that there is no cycle. This algorithm will find a cycle if one exists. This contrasts with Dijkstra’s Algorithm.

S(u, v) = The shortest path from u to v

Pi(x) = The parent of x that creates the shortest path

D[v] = The current distance of v from s.

**Properties**

1. **Upper Bound property:** d[v] >= S(u, v) and if d[v] == S(u, v) then d[v] will not change again (unless there is a negative weight cycle). At the beginning of the algorithm, d[v] = 0 and S(u, v) = negative infinity
2. **Sub path Theorem:** Intermediate paths between s and v are themselves shortest paths.
3. **Triangle Inequality:** For x, y, z in V, S(x, z) <= S(x, y) + S(y, z)
4. **Path Relaxation:** Any sequence of calls to RELAX produces S(v0, vk).

* Correctness
  + ****
  + <https://people.csail.mit.edu/alinush/6.006-spring-2014/mit-fall-2010-bellman-ford.pdf>
* Efficiency
  + O(V\*E)

**Floyd-Warshall – All Pairs Shortest Paths**

This is the most efficient all-pairs shortest path algorithm to date. With this algorithm, the graph may have negative edges and APSPs can be computed in cubic time. To perform the same computation with Bellman Ford would take O(V^4) in the worst case. Dijkstra’s would fail on a graph with negative edge weights, and it would run in O(V^3 \* Log(V)) time in the worst case.

Floyd-Warshall uses dynamic programming to find any path from u to v that goes through any of the other vertices in the graph.

To terminate the algorithm, it must be that (1) the algorithm has computed all pairs of shortest paths for all (u,v) pairs or (2) the algorithm reports that there is a negative weight cycle (sum of all edges in the cycle are negative).

* Correctness
  + 
  + <http://www.cs.toronto.edu/~lalla/373s16/notes/APSP.pdf>
* Efficiency
  + O(V^3)

**Polynomial-Time, Solvable Problems vs Non-deterministic Polynomial Time**

Polynomial-Time Solvable 🡺 Given a problem with input size n, there is a polynomial time algorithm that solves the problem in O(n^k) time where k is some constant.

P = the set of all polynomial time solvable problems.

Computational Tractability 🡺 The litmus test for this involves checking the problem’s membership in P.

**The Traveling Salesperson Problem**

**Input**: A complete graph with non-negative edge weight costs.

**Output**: An Eulerian Tour (visits every vertex exactly once).

Edmonds Conjecture 🡺 There is no polynomial-time algorithm to solve the TSP. (i.e. P != NP)

**Reductions & Completeness**

**Reductions:** Problem A reduces to Problem B iff given a polynomial time solution to Problem B, we can solve Problem A efficiently.

{ex.

Detecting a cycle REDUCES to DFS

All-Pairs Shortest Paths REDUCES to Single Source Shortest Paths

}

How can it be determined if a problem is NP-Complete before trying to come up with an algorithm for it.

**NOTE:** If we believe that it is not possible to solve Problem A is polynomial-time, then it is not possible to solve Problem B in polynomial-time either.

**Completeness** (the contrapositive of Reductions): If Problem B is not in P then Problem A is not in P. If we don’t believe we can solve Problem A efficiently and Problem A reduces to Problem B, then we don’t believe that we can solve Problem B efficiently either.

Problem A reduces to Problem B 🡺 B allows you to solve A 🡺 A <= B

**C-Complete**

Problem A is C-Complete if A is in C and every problem in C reduces to Problem A (i.e. given an efficient algorithm for Problem A, all problems in C can be solved efficiently. Problem A is the hardest problem in C)

To prove that TSP is C-Complete, we need to define the class C of problems of which TSP is the hardest.

C = all of the problems that are brute force solvable, including constraint problems like 3-SAT (because TSP is solvable in O(V!) time)

**NP-Complete**

A problem is in NP if:

* Solutions have length polynomial in the input size
* Solutions can be verified in polynomial time.

To prove that a problem is computationally intractable, prove that it is a part of the NP Class (i.e. by solving your problem efficiently, you would be able to solve a problem in the NP Class efficiently and therefore all problems in the NP Class efficiently.)

**Strategies for Solving NP-Complete Problems (as efficiently as possible)**

1. Focus on solving the computationally tractable special cases.
   1. WIS Path Graphs (we know how to solve) when trying to solve Maximum WIS in general graphs.
   2. Knapsack with small W.
   3. 2-SAT instead of 3-SAT
2. Heuristics
   1. Not always correct
   2. A\*, Karger’s Contraction Algorithm
3. Solve in exponential time with smart optimizations
   1. Do this if you are unwilling to compromise on correctness.
   2. TSP in O(2^n) instead of O(n!)

### Practice

1. Discrete Math book – general proof practice
2. Knuth’s Algorithms book – reading + practice problems
3. Practice problems running the algorithms above (see the internet)
   1. Edmonds-Karp
   2. Weighted Scheduling
   3. Max Weight Independent Set
   4. Knapsack
   5. APSP
4. Revisit proofs as necessary
5. Write an exchange argument for the greedy Weighted Scheduling algorithm that uses the difference instead of the ratio.
6. Max Flow – Residual Graphs
   1. Find a situation where flow would be taken back.
   2. A screenshot of a cell phone

      Description automatically generated
7. Ackerman and Inverse Ackerman to revisit Trajan and Ullman bounds + Fredman Saks

## Additional Algorithm Design Notes

1. **Paradigms (And how to prove their correctness)**
   1. Divide and Conquer
      1. Induction
         1. Assume subproblems are correct
         2. Explore the two or more recursive calls and their implications on the solution to the current problem. (Explicitly define how the solution to smaller subproblems is a solution to the main subproblem)
   2. Greedy Algorithms
      1. Induction
      2. Exchange Arguments 🡺 Take Greedy solution G and some hypothetical better solution O such that:
         1. There is an element in G that is not in O (and/or?) there is an element in O that is not in G. 🡺Turn O into G with a polynomial number of switche… OR…
         2. There are two consecutive elements in O that are in a different order than they are in G (they are the same up until a certain point) 🡺 Contradiction if O can be improved.
   3. Randomized Algorithms
   4. Dynamic Programming
      1. Induction
   5. Brute Force
   6. Backtracking
2. **Algorithmic Primitives (4)**
   1. Sorting
      1. If the best algorithm without sorting is quadratic or worse, sorting is essentially free and gives an opportunity to bring the runtime down to N\*Log(N).
   2. Graph Search (DFS/BFS) + MST + Disjoint Sets for cycle detection
   3. Data Structures
3. **Big O Notation**
   1. Understand it: Big O notation lets you analyze an algorithm's time and space complexity (how resource usage scales with input size). Focus on the dominant terms (e.g., O(n^2) is worse than O(n log n)).
   2. Aim for lower complexity: Favor linear time (O(n)) or logarithmic time (O(log n)) where possible.
   3. Common complexities: Get familiar with the time complexities of common operations and data structures (access, search, insertion, etc.).
4. **Data Structures**
   1. Pick the right tool: Choosing the right data structure is crucial.
   2. Arrays: Fast for random access, but resizing is expensive.
   3. Linked Lists: Efficient for insertions and deletions, less so for random access.
   4. Hash Tables: Often provide near-constant time lookup (O(1) on average).
   5. Trees: Enable logarithmic searches, useful for ordering.
   6. Heaps: Offer quick access to the minimum or maximum element.
   7. Know their strengths: Understanding the time complexity of operations within each data structure will guide your decisions.
5. **Algorithmic Techniques**
   1. Divide and Conquer: Break the problem into smaller, similar subproblems, solving them recursively. Examples: Merge Sort, Quick Sort.
   2. Greedy Algorithms: Make locally optimal choices at each step, hoping for a global optimum. Not always the best, but can be good for certain problems.
   3. Dynamic Programming: Store solutions to subproblems to avoid recalculations, trading space for time. Great for problems with overlapping subproblems.
      1. Memoization: Specific kind of dynamic programming where results are cached.
      2. Backtracking: Explore possible solutions, pruning branches when they can't lead to a solution.
6. **Optimization Techniques**
   1. Profiling: Use a profiler to pinpoint bottlenecks in your code, focus your optimization efforts accordingly.
   2. Caching: Store frequently used results for quicker access.
   3. Pre-calculation: Compute expensive results beforehand if they'll be reused multiple times.
   4. Avoid nested loops: Nested loops often lead to quadratic (or worse) complexity. Find optimizations where possible.
   5. Bitwise operations: For certain tasks, bit manipulation can be faster than arithmetic operations.
7. **Space-Time Tradeoffs**
   1. Be aware of tradeoffs: Sometimes you can improve time efficiency at the cost of using more memory (and vice-versa).
   2. Consider problem constraints: If you have ample memory, a hash table might be better. If memory is scarce, you might optimize for space even with a slightly less time-efficient algorithm.