Table of Contents

- Algorithms
  - Asymptotic Run Time
    - Proving asymptotic run times
  - Divide and Conquer
    - Algorithm Design Considerations
    - The Master Method
    - Substitution Method
    - Recursion Trees
  - Randomized Algorithms
    - Algorithm Design Considerations
  - Decision Trees
    - Proving Lower Bound for Comparison Sorting Models
  - Proof Notes
    - Induction
    - Contradiction
    - Proving Correctness
    - Proving Runtime
  - Glossary of Algorithms
    - Binary Search
    - Karatsuba
    - Matrix Multiplication
    - QuickSort
    - RSelect
    - DSelect
    - BucketSort
    - RadixSort
- Data Structures
  - Operations on Data
  - Binary Tree
    - Heaps
    - Binary Search Tree
    - Red-Black Tree
    - Hash Tables
- Graphs
  - Depth First Search
    - Directed Acyclic Graph
  - Breadth First Search
  - Strongly Connected Components
  - Dijkstra's Algorithm
- Dynamic Programming
  - Algorithm Strategies
  - Knapsack
  - BFS as DP
  - Weighted Graphs
- Greedy Algorithms
    - Activity Selection
    - Job Scheduling

- Huffman Codes
- Minimum Spanning Trees
  - Graph Cuts
  - Prim's Algorithm
  - Kruskal's Algorithm
- Min Cut / Max Flow
  - s-t Cuts
  - Maximum Flow
  - Ford-Fulkerson Method
    - Edmonds Karp
  - Maximum Bipartite Matching
  - Global Minimum Cuts - Karger
    - Karger-Stein
  - Stable Matchings
    - Deferred Acceptance / Gale-Shapley
- Review
- Probability Reference

Algorithms

Asymptotic Run Time

- Adding two n digit integers is an O(n) operation since for each position we add at most three digits
- Little o vs Big O - Theta is meant to be asymptotically tight, little o is absolute and a looser bound
- $log(n!) = \Theta(log(n^n)) = \Theta(nlog(n))$

Proving asymptotic run times

- Disproving Big O / Omega. Using contradiction show n can be compared to a constant. If n_0 is then set to be the constant, then $n > n_0$ will always be greater (less) than the constant, showing the contradiction.
- Proving Big O / Omega - simply need to show for some c and n_0 pair that we can bound the function. This works since we only need to show the inequality is true for some c, not all.
- Proving Theta = proving O and Omega separately

Divide and Conquer

Algorithm Design Considerations

- Divide the input into smaller subproblems, conquer the problems recursively, combine the solutions into a solution for the original problem
- Start with a brute force algorithm - this sets an upper bound on the possible run time.
- Break the possibilities into cases - if we are trying to find this property, it could occur in the following ways. Given those ways, set up this portion as possible to do recursively, this portion needs to happen before or after recursion. Then think about run time - we are making every possible comparison, do we know of properties, facts, or goals that would reduce the amount of work we need to perform.
- MergeSort has the clever part hidden in the conquer portion, QuickSort has it hidden before the recursion, in deciding where to pivot. It can also occur in the recursive calls.

- When we are thinking about the size of the elements in an array, think how sorting could be used. When we are thinking about the index or ordering of an array that should not change, consider comparing certain values of specified indices. Is it a global all encompassing solution (MergeSort type), or we care about one element or finding one thing (search, BST)
- After coming up with a brute force recursive algorithm, ask whether any work is duplicated or superfluous. Are there things we are doing that could eliminated like in Karatsuba or can be skirted thanks to the conditions imposed by the invariant.
- Think of for-free primitives -> if we are already running in a certain omega time, adding a primitive in the same time could help with the algorithm without breaking the runtime
- If you can take a non-recursive algorithm into recursive form, is there some duplicated work in the recursion that can now be removed

The Master Method

- $T(n) \leq aT(\frac{n}{b}) + O(n^d)$
- a = number of recursive calls at an iteration of the algorithm (not all recursive calls total)
- b=input size shrinkage factor - how much smaller is the element passed to a recursive call
- d=exponent in running time for the combine step, since $O(n^d)$ is the work done outside of the recursive calls.
- Must be able to compose the algorithm in terms of this standard recurrence. a,b,d are constants, not dependent on n.
- $a = b^d \rightarrow O(n^d log\, n)$
- $a < b^d \rightarrow O(n^d)$
- $a > b^d \rightarrow O(n^{log_b a})$
- More general formulation: for $T(n) = aT(\frac{n}{b}) + f(n)$ for positive $a \geq 1$ and b > 1
    - if $f(n) = O(n^{log_b a - \epsilon})$ for some $\epsilon > 0$, $T(n) = \Theta(n^{log_b a})$
    - If $f(n) = \Theta(n^{log_b a}), T(n) = \Theta(n^{log_b a} logn)$
    - If $f(n) = \Omega(n^{log_b a + \epsilon})$ and if $af(\frac{n}{b}) \leq cf(n)$ for c < 1 and all sufficiently large n, then $T(n) = \Theta(f(n))$
- What this boils down to: a is the number of subproblems at each recurrence. If we perform n^d work at the first level, then each subsequent level performs $(\frac{n}{b})^d$ work. So the question is does the number of subproblems grow faster or slower than the work shrinkage factor of b^d? This is why a recursion tree can be helpful - sketch out how the subproblems spawn and the size of work shrinks.
- Altering recurrence relations to fit the formula: eg $T(n) = 2T(\sqrt{n}) + O(logn)$. 1) Try to find a substitution that might make this a more manageable relation. Here $k = log\, n$ seems reasonable so $n = 2^k$. 2) Reformulate the RR using the new variable: $T(2^k) = 2T(2^{k/2}) + O(k)$. 3) Make a new RR s.t. we have a form $S(k)$ on the LHS. Here we use $S(k) = T(2^k) = 2S(\frac{k}{2}) + O(k)$. 4) Use the master theorem to find the runtime in terms of k. 5) Reverse the original substitution to get the runtime in terms of n.

Substitution Method

- Guess a function f(n) which you suspect satisfies $O(n) \leq O(f(n))$. Prove by induction on n that this is true.

- Fix a positive integer $n \geq 2$ then try to prove $T(n) \leq l \times n$.
- For the base case, we must show that you can pick some d s.t. $T(n_0) \leq d \times g(n_0)$ for our guessed function g(n) and d constant greater than zero. Then assume our guess is correct for everything smaller than n and prove it using the inductive hypothesis. Typically prove the inductive step from the hypothesis and obtain a condition for d.
- Inductive hypothesis we assume our guess is correct for any n < k and prove our guess for k.
- Concretely, we have $T(n) \leq aT(\frac{n}{b}) + O(n)$. Guess O(n log n), meaning prove $T(n) \leq cn\, log\, n$ for an appropriate choice of c. Assume this holds for all positive values m < n, or m = n/2. Substitute m for n in the expression and simplify. May have many terms, but can try to bound with our original limit *given* a certain restriction on c (eg. c > 1). Prove with induction for $n > n_0$ where we get to choose n_0 to avoid tricky boundaries. Plug in constant values of n, see what values we get and pick a c s.t. the bound always holds.
- If you have tricky expressions, bound the expression with a simpler one that eliminates low order terms.
- For DSelect / DQuickSort the idea used is the fractions in the two recurrences sum to less than 1 -> therefore the guess was a constant times n -> O(n) = cn.

Recursion Trees

- Start from level zero where there is no recursion. Split the tree down by the number of recursive calls / subproblems created at each level.
- The levels go from 0 to $log_b n$ where b is the shrinkage factor for the array size. At level j, each subproblem is size $\frac{n}{b^j}$ and the work performed is $c(\frac{n}{b^j})^d$.

Randomized Algorithms

- Probability concepts needed: linearity of expectation, indicator RVs, $\sum_{i=1}^{n-1} f(i) \leq \int_1^n f(x)dx$, expectations of Bernoulli, Binomial, and Geometric RVs.
- For QuickSort, if we always choose the first element on a sorted array, it runs in $O(n^2)$ time. It looks over every element comparing it to successive pivots. If the median were always the pivot, would run in $O(nlog\, n)$ time since the recurrence relation is the same as MergeSort. A uniform random process runs in $O(nlog\, n)$ shown through decomposition. Here the decomp shows that for element i and j to be compared, there are (j - i + 1) picks between them inclusive, 2 of which will compare them and the remaining in which they are split into different arrays and never compared. Therefore the expected number of comparisons is $\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$. This is then expanded as a power series, and bounded by the power series of ln(n). For the double sum, then bounded by 2n ln(n).
- For decomposition 1) identify the random variable Y 2) Express Y as a sum of indicator (0-1) variables $X_1, \ldots, X_m$, $Y = \sum_{l=1}^{m} X_l$. 3) Apply linearity of expectation 4) Compute each $P(X_l = 1)$ and add to find E(Y)
- Consider the running time of the algo of an input as a RV and bound the expectation of the RV
- Geometric series: $1 + r + r^2 ... + r^k = \frac{1-r^{k+1}}{1-r}$
- For r < 1, $1 + r + r^2 ... + r^k \leq \frac{1}{1-r}$

Algorithm Design Considerations

- Guess and Check - randomly guessing something and checking the rest of the elements against it.
- Picking a Pivot - pivot and partitioning and moving along with one side or the other

Decision Trees

- Trying to demonstrate bounds of run time for all types of sorting models
- Compare two items at the top of three, then at the next step, can compare one of the same items to a new item. Transitivity tells us something about all 3 element. The leaves of the tree have the ordering of the elements compared by the upward branches. Internal nodes are boolean answers to comparison, leaf nodes correspond to outputs, all possible orderings of the items
- Size of the tree depends on the size of the array. The elements in the array determine the path of the tree - running the algorithm corresponds to a single path in the tree, ie to the correct sorting of the inputs.

Proving Lower Bound for Comparison Sorting Models

- QuickSort takes a pivot, compares a single number to the pivot, and there is some indicator (say genie) that says whether the item is bigger that the pivot
- Theorem: comparison models are $\Omega(nlogn)$. Deterministic must take this many, randomized in expectation.
- Using a decision tree, we have every comparison down the tree ending in sorted output at the leaves. The runtime is the length of the path from the root to leaf for the correct output of the algorithm. Worst case run time would be the longest path from the root to any leaf.
- A binary tree has n! leaves - it contains every possible answer to the algorithm. Since it is a permutation of all items, this is n!. Then it must have $log(n!)$ depth at minimum, so it longest path must be at least log(n!). Then log(n!) = $\Theta(log(n^n)) = \Theta(nlog(n))$. (This is done via an approximation - but just think of this as the lower bound since every comparison based method must have longest path in the best case of log(n!))

Proof Notes

Induction

- Induction - need to state we will be using induction, declare a base case, inductive hypothesis, inductive step, formally state the demonstrated results. The inductive hypothesis should be not quite the conclusion but the specific thing we will show in the inductive step. For example showing the first element is bigger than the last element, use A[0] > A[i] for any i, not jumping specifically to the first element is bigger than the last.
- For your induction to be valid for arbitrary *n*, you must be able to point to a base case and sequence of inductive steps that show the claim extends to *n*. As an example, when *n*=3 using boilerplate induction, this corresponds to base case of 1, and inductive steps from 1 to 2 and 2 to 3. If you can formalize a way to show that each *n* from 1 to *k* can be expressed in this form, the induction is valid.
- IH: After running for i iterations, we have achieved some goal expected after that iteration.
- Base case: typically trivial, something 0, 1 to show our algorithm holds this
- Inductive step: Know up to the i-1th iteration, want to show for i. Given generic variable inputs with certain invariant properties known after iteration i-1, explain how the algorithm would actually work upon those variables. This may involve laying out cases for different inputs. Once cases addressed, show that the algorithm does the right thing in all cases and returns an output as expected.

Contradiction

- Contradiction - clearly state proof by contradiction. This can be combined with induction if the inductive step is needed to show a contradiction. Always starts with assuming for the sake of argument the complement to the stated claim.

Proving Correctness

Proving Runtime

- First, the master theorem should be attempted. Alternatively the general master theorem but be wary of case 3.
- When children are of different size, eg. $T(n) = T(\frac{n}{2}) + T(\frac{n}{4}) + O(n)$, the substitution method might be best since we can bound and guess.
- Recursion trees are great for actual algorithm analysis, though tend to be useful always

Glossary of Algorithms

Binary Search

- Divide and conquer to find a item in a sorted list. Look in the middle of the book and recurse on the left OR right depending on if value is greater or less than the mid point.

Karatsuba

Matrix Multiplication

QuickSort

- Randomize algorithm for sorting. The random element is the choice of pivot. 1) Choose a pivot element, 2) rearrange the input array around the pivot s.t. all smaller elements go to the left, bigger to the right. Elements are only compared to the pivot, not to every other element.
- Recursive calls occur after partitioning and there is no combine element.

RSelect

- ith order statistic is the ith smallest element in an array. Here we are trying to find that element for some specified i.
- Choose pivot randomly, if we picked the ith order statistic done, otherwise determine if our pivot k > i look in the left portion of array, otherwise right half. If the median is always picked then the recurrence relation is $T(n) \leq T(\frac{n}{2}) + O(n)$ and it runs in $O(n)$. Worst case runs in $O(n^2)$. In expectation runs in linear time $O(n)$ given properties of random pivots likely to be good pivots.

DSelect

- Instead of random pivot, pick pivot that is the median of medians. Divide list in to 5 parts, find the median of each part and find the median of the medians. The median of each sublist, we have a constant number of sublists and performs a constant amount of work per subarray, so computing the median takes constant time. Think of it as a MergeSort on the subarray of length n/5, leading to an O(n) routine.
- Has two recursive calls - one to find the median of medians, one to find the order statistic. The recursive calls play fundamentally different roles and can have different run times.
- Median of medians guarantees a split in the middle 40% of the array. Recurrence relation is then $T(n) \leq T(\frac{n}{5}) + T(\frac{7n}{10}) + O(n)$ which is roughly $T(n) \leq T(\frac{n}{5}) + T(\frac{7n}{10}) + cn$ since it is linear time outside of recursive calls.

BucketSort

- Big idea: Sorting in linear time
- Buckets (linked list that are FIFO) for each value, placed numbers into the correct bucket then concatenate. O(n), but restrictive given possible inputs.
- Uses the fact that we are actually sorting numbers. Have buckets to place numbers into from array, then just concatenate the the buckets. O(1) to put into buckets, then enumerated over each bucket O(n). Need to assume there are not too many values and need to know what values might show up ahead of time.

RadixSort

- Big idea: Sorting in linear time
- For sorting integers up to size M. Idea: BucketSort repeated, start by looking at the least significant digits, BucketSort, then repeat with the next least significant. Only need buckets 0-9. We place the full number in the bucket, just are sorting by a single digit at a time. Performed with linked lists because we need a FIFO system to get a sorted array at the end, iterating over the original and mid arrays from left to right.
- Given n d-digit numbers (in base 10), d iterations, O(n) per iteration, total is O(nd). d turns out to be $log_{10}n + 1$ so leads to O(nlog(n)), same as comparative algorithms.
- We need to change the base of the log. Bigger base means more buckets but fewer digits and fewer iterations. If r becomes very big, then you have BucketSort. For base 100 you have 100 buckets, 00-99.
- For n integers, max size M, and base r: # iterations = # digits, base r; d = $log_r(M) + 1$. Time per iteration is still O(n+r).
- Reasonable choice is r = n. -> $O(n \times (log_M(n) + 1))$. If $M \leq n^c$ for constant c, then this becomes O(n). If M is huge, this is not going to be a good algorithm. Want M to be near n, not orders of magnitude larger.

Data Structures

Operations on Data

- Given some data structures like sorted arrays or linked lists, want to insert, delete
- In a sorted array, search is efficient in O(logn) but insert / delete mean moving things in memory, touching every element for O(n). In a linked list just replace the pointer for insert, but search / delete take a long time.

Binary Tree

- Nodes without children really have NIL children
- Complete - every level (except last) is completely filled. In the last level all nodes are as far left as they can be.

Heaps

- Complete binary tree s.t. every descendent of a node has a larger key
- Two invariants - complete and ordered
- Insert fast and extract-min fast
- Insert - got to first empty node, plop new value there, this is O(1). Make sure we still have a complete

binary tree, then make sure that the keys are sorted correctly for a heap. Then “bubble up” if the child is bigger than the parent node until they key is smaller than its children. Each bubble up is constant time, so this takes at most O(logn), the total number of levels.

- Extract-min - Extract root, since this is the min. Set the bottom node (now violating the invariant) to the root, then bubble down using smallest child logic until invariant fixed. Taking the right-most, bottom-most element to the top, then bubble down.
- In either case, swapping up the tree or down the tree with O(1) with each operation. In total, we have the O(height), and since it is a binary tree, this is O(logn).
- Heap is not a unique ordering of the elements.

Binary Search Tree

- Supports all of the operations that a sorted array supports plus insertions and deletions in O(logn) time - dynamic
- Invariant: every left descendent of a node has key less than the node and right has key larger.
- Look up the tree to check if a tree is violating this property. You can make many different BST for a set of values.
- Similar to QuickSort - choose a root (pivot), then sort elements left and right, repeat.
- In order traversal - outputs the elements in sorted order. Do the left subtree first, print the key, then traverse the right. Runs in O(n).
- Search - traverse down the levels of the tree. Right if we are looking for a bigger key than current node, left otherwise. If there is no path towards the direction we need, return None. O(height) / O(logn). To find min or max, we simply traverse a single direction until the end. Note if the value is not in the tree, we may not get the closest value, since if a parent node is the closest but bigger (smaller) than the searched valued, we will perform one more search left (right) and take that value as the terminus.
- Insert - start with search, then insert when we cannot traverse to a closer key.
- Delete - search, delete, but also need to move children to new nodes, but this happens in O(1) time so the search time dominates.
- Height of tree - longest path from root to leaf. Best case is log(n) but the worst case in n-1 if you have a single chain. Balance is important. Therefore important to think of running time in terms of O(height) and consider whether we can guarantee a certain height limit or know the height from the problem.
- Could take O(n) if every node has a single child (poorly built). Instead of starting from scratch if it’s unbalanced, could use local operations to fix it. Turn to Red-Black trees - close enough to balanced that maintains itself
- A single BST corresponds to a single ordering of the elements. There are many BSTs possible but each one is a single ordering.

Red-Black Tree

- Every node is red or black. The root is black. All leaves have NIL children that count as black. The children of every red node are black. For all nodes X, all paths from X to NIL’s have the same number of black nodes (instead of trying to guarantee all paths have the same number of all nodes). Meaning for any NIL, you should have to get through the same number of black nodes when you pick the same starting place. Note if a black higher up a tree only has one child, then it also has a NIL child and is 0 black nodes away from a NIL.

- Invariants (4): Valid BST. The root is black. The children of every red node are black. For all nodes X, all paths from X to NIL's have the same number of black nodes (instead of trying to guarantee all paths have the same number of all nodes).
- This makes the black nodes completely balanced and the red nodes need to be spread out. Easy to maintain these properties with insertion and deletion. Intuitively, red nodes indicate when a path is becoming too long.
- Any valid red-black tree on n nodes (non-NIL) has height ≤ 2 log(n + 1) = O(log n) - one side can only be double the short side by padding every other black node with red nodes.
    - Proof: IH: For subtree of size $\leq k$ (k nodes), $k \geq 2^{b(root)} - 1$
    - Base case: Tree of 1 node
    - Inductive step: k(x) = k(y) + k(z) + 1
    - $k(x) \geq (2^{b(y)} - 1) + (2^{b(z)} - 1) + 1 \geq 2 \times 2^{b(x)(-1)} - 1 \geq 2^{b(x)} - 1$
    - Notice we plug in the inductive hypothesis by plugging in for k(y), z and bounding
- Search only takes O(logn) since that is the height of the tree for sure. All other operations are O(logn) as well, though we won't show exactly why.

Hash Tables

- Hash tables are all about fast lookups for dynamic data sets. O(1) expected time for insert / delete / search. Ideal for when we want to store values but do not care about ordering.
- Given a universe U of size M, for very large M. There are only n elements that will ever show up though, we just don't know which elements in advance.
- For class examples, we are going to set #buckets = #elements = n.
- We have a hash function that maps elements to buckets. We will only consider hashing with chaining in this class. If we get many elements that map to the same bucket, we get a very long chain - could take O(n) time to search through the linked list. Want a hash function that ensures the elements are spread across buckets, ie. O(1) entries per bucket.
- For any fixed choice hash function h, one can always produce a subset of keys S such that all keys in S are mapped to the same location in the hash table.
- Let X be the size of the hash bucket that ki maps to:
    - $$\begin{aligned} E[X] &= \sum_{j=1}^{n} \Pr[h(x_i) = h(x_j)] \\ &= 1 + \sum_{j \neq i} \Pr[h(x_i) = h(x_j)] \\ &= 1 + \frac{n-1}{n} \leq 2 \end{aligned}$$
    - This follows from $\Pr[h(x_i) = h(x_j)] = 1/n$
- We cannot choose a hash function that guarantees that an input will be distributed across buckets - so instead we could randomize our pick of hash function. Say we have hash functions h $\sim U(),\ iid$. If we fix u_i bucket then the chance that u_j gets put in that bucket is $\frac{1}{n}$. Sum over all $j \neq i$. But if we pick a different function for every x, then we would have to store h(x) for every x - terrible. We would need Mlog(n) bits to store, more than the naive storing in M buckets, which only takes M bits.
    - Aside: description length. Set S with s things in it. l = # of bits. # of l-bit strings = $2^l \geq s$ so $l \geq log(s)$

    - We have M choices in the universe, and # of hash functions is $\binom{n}{h(1)}\binom{n}{h(2)}\dots\binom{n}{h(M)} = n^M$.
    - How do we fix this? Pick from a smaller family of hash functions.
- Universal Hash Family: for u_i, u_j in U, with $u_i \neq u_j$, $P(h(u_i) == h(u_j)) \leq \frac{1}{n}$. Point is to mimic the uniform distribution of a random hash function while limiting the space it requires to store.
- Determining if universal - can we generate a pair of numbers that are mapped disproportionately to certain buckets. For example, hash family of mods, every integer + mod will map to the same buckets, so the chance that mapping two values to the same bucket is more than 1/n. The contradiction to universality should be relative to the pair of inputs picked u_i, u_j. If we find two hash functions with the same output for our pair, calculate what fraction of the total buckets this would make up.
- Creating a hash family:
    - Pick prime $P > |U|$
    - For $a, b \in \{0, \dots p-1\}$ consider family $h_{a,b}(x) = ax + b \bmod p \quad \bmod n$
    - Taking mod p maintains there are p different assignments in the first step. Taking mod n ensures there are n buckets and is where the collisions happen, mapping a space of p onto n buckets.
    - We can store this for a choice on p in log(M) bits. A universal hash family of size $O(m^2)$ exists using this method, since p-1 choices for a, p choices for b = p(p-1) = $O(M^2)$
- Good for lookup of pairs - loop through list of numbers and look for pair in the hash table in constant time. Could also think of BFS looking in a hash table to see if it has explored that vertex yet.

Graphs

- G(V, E) - V vertices, E edges. n = # of vertices, m = # of edges. Degree of vertex = # of edges from vertex. Can also specify in-degree / out-degree for directed graphs. Neighbors = connected vertices
- Can store a graph in either: Adjacency matrix - binary whether two vertices share an edge, rows source columns destination. Linked list - pointers to neighbors.
- Linked list is better for sparse graphs, since uses O(n + m) space vs $O(n^2)$ for adjacency matrix. Generally assume linked lists in class. We could use binary search trees to reduce search time to log
- Undirected Graph Connected Components - a maximal set such that for all u,v in S, there exists a path in G from u to v. Connected components can just be found using BFS/DFS.
- **Parentheses theorem**: if v is a descendent of w in a tree, (w.start, v.start, v.finish, w.finish). If neither is a descendent of the other (v.start, v.finish, w.start, w.finish), this is a general claim, not just for DAGs
- Note that a tree is a connected graph with no cycles

Depth First Search

- Keep going deeper until you have no new neighbors to go to, mark node as explored, then backtrack until there is an unexplored neighbor
- Keep track of unvisited, in progress, and all done nodes. Keep track of time we first enter node (start time) and the time we finish and mark it done (finish time)
- Finds all the nodes reachable from the starting point. Connected component - all of the vertices have some path from one vertex to another. DFS finds all connected components. **DFS is really building a tree implicitly and traveling down to a leaf** and then backtracking.
- Run time for DFS? Degree(w) in the for loop, run the for loop for every w: $\sum_w deg(w) + O(1) = 2m + n$. Every edge is plus one to the degree of the 2 nodes it connects, which is why we have 2m. Therefore $O(m + n)$, the n since we do a constant amount of work for every vertex in the recursive call.

- When you cannot reach all vertices, we start again at an unvisited vertex and get multiple DFS trees - DFS forest
- If (u,v) is an edge in an undirected graph and during DFS, $finish(v) < finish(u)$, then u is an ancestor of v in the DFS tree. When we do DFS, we store "Visited" nodes in a stack to keep track of the order in which they were visited. Stacks, by nature, have a "last-in first-out" order, meaning the last node you added into the stack will be popped out before any of the nodes before it.

Directed Acyclic Graph

- Imagine dependencies, need to figure out what relies on what. DFS starts at first node and explores
- **Topological sorting**
    - In a DAG, given an edge $A \rightarrow B$, then $B.finishTime < A.finishTime$. If we order by reverse finish times, we can ensure we follow the dependencies in the right order.
    - If an edge from $A \rightarrow B$, then A appears before B in the sort. If there aren't edges connecting two vertices, there is not a rule in TopoSort about which should come first.
    - As you mark vertices as done, we put it at the beginning of our list, end with a topologically sorted list. If we started at a vertex that isn't the top, we would complete lower vertices but then we would need to start a new DFS from nodes left unexplored - can end up with a DFS forest instead of a tree.
    - Runs in $O(m+n)$ since is just an alteration of DFS

Breadth First Search

- Explore immediate neighbors, then their immediate neighbors, etc.
- Think of it running on a FIFO queue. When travel to a vertex, create an **adjacency list** of the vertices that are its neighbors.
- Outer for loop over all of the distances, for each vertex of distance i add to L_i and look at all of their neighbors, for each neighbor if v is not visited mark as visited and add to L_{i+1}
- Also finds all connected components like DFS. Also runs in $O(m+n)$ - inner loop over the adjacency list for each vertex once, outer for loop over all vertices. Sum of degrees = n, check every edge m
- Application: shortest path, we know by the level of a vertex how many steps it takes to get from the starting vertex to another, found in $O(m)$. The distance is the difference in layers between the source and destination. Just initialize a distance estimate, 0 for source and ∞ for all others, then each distance is updated as +1 compared to the distance of the neighbor from which it was discovered.
- Application: bipartite, can we split graph in 2 s.t. no vertices that share a class have edges between them. Think of students enrolled in classes - students cannot be enrolled in students. Could check this with a tree, but with BFS can do it based on level each vertex is a member of, then check if there is a shared edge with any members of the same color. BFS finds not bipartite for odd cycles - can never have an odd cycle graph that is bipartite. Odd cycle is a subgraph with odd number of vertices / edges - starting from one vertex, how many steps does it take to cycle back to that vertex.

Strongly Connected Components

- Maximal set S subset V s,t, for all u,v in S there exists a path in G from u to v and v to u
- Weakly connected - If we turned directed into undirected, then vertices would be connected components

- Path from every vertex to every other vertex within SCCs.
- SCC Graph - can circle the SCC within a total graph, then each SCC can be contracted into a meta-vertex.
- Lemma 1: our SCC graph is a DAG - it cannot have cycles between SCCs
- Undirected graphs - connected components, find with BFS/DFS. Directed graphs - strongly connected components, find with DFS. There is no overlap here. Do not talk about strongly connected components in undirected graphs
- To find SCCs we need to find a sink SCC. Basically need to find a set of vertices from which we cannot move onto the rest of the graph. This means a reverse topological traversal of SCCs - Kosaraju's Algorithm
- Topological Ordering of SCCs - can take the smallest finish time of a node in an SCC to be the smallest finish time of the SCC
- An O(n+m) algo for SCC
    - Starting place matters - starting from one vertex may get us the SCC from running DFS, while another may just have a single tree. Starting DFS from the last SCC is good, because we can only explore that SCC.
    - Run DFS from any starting vertex, look at the finish times.
    - Reverse all of the edges in the graph - this doesn't change SCCs since they go in both directions
    - Run DFS again, using the latest finish time for the first run to tell us where to start.
    - The SCCs are the different trees in the second DFS forest
- The SCCs of G are the same as those in $G_{reverse}$ - it does not matter whether we run DFS first or reverse the edges first.
- Finish time of an SCC DAG - the largest finish time of any vertex in the SCC. Starting time is the smallest starting time of any vertex
- SCC DAG can now by topologically sorted based on the SCC finish times (from the regular DFS, not reversed)
- **Key Lemma**: In the forward SCC graph - if edge from A to B, then A.finish > B.finish. Then (corollary) in the reverse graph if $B \to A$, A.finish > B.finish in the real graph.

Dijkstra's Algorithm

- Assume all weights are non-negative. In general making some assumptions about your input can make faster algorithm choices
- SSSP $O((m+n)log(n))$, APSP $O(n(m+n)log(n))$
- In BFS, when we put u in L_i, we know that u is exactly i away in the shortest possible path. For B-F, we loop over vertices more than once because we don't know that we have the shortest path to u in the previous iteration.
- Maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. Repeated selects the vertex $u \in V - S$ with the min shortest-path estimate adds u to S, and relaxes all edges leaving u. Maintains the invariant Q = V - S at the start of each while loop.
- The algorithm only extracts vertices from Q and each vertex is added to S once, so loops exactly n times. once a vertex is in S, it is done and the paths leading from it can be considered (not just from the node last added to S). It always chooses the vertex in V-S closest to S. This is a greedy strategy
- Note the algorithm calls insert and extract-min once per vertex and decrease key once per edge.

Dynamic Programming

- Algorithm design paradigm, storing our work to reduce repeated computation
- The idea of dynamic programming is to have a table of solutions of subproblems and fill it out in a particular order (e.g. left to right and top to bottom) so that the contents of any particular table cell only depends on the contents of cells before it.
- Usually used for optimization problems. Big problems break into sub-problems like in divide and conquer.
- Keep a table of solutions to the subproblems, refer to the table instead of recomputing work already performed
- Runtime - (size of cache) x (# of operations each time you add to cache)
- Bottom up
    - Solve the small problems first, then the bigger problems, and the final steps solve the real problem.
    - Often the easier one to write and analyze the running time
    - Think of solving F[0], F[1], then iterating to find $F[2], \ldots, F[n]$
- Top down
    - Like a recursive algorithm, similar to divide and conquer
    - The difference is memoization - when we encounter an already solved subproblem, we just plug in. This means we have to keep track of what we have already solved.
    - Top down fibonacci - have list of computed values. Check if we have value of F[n] and return else run the recursion then return F[n]
    - Make sure the **cache is updated** before returning the value within the recursive function.
- Bottom up often nested for loop, top down is more recursive
- Optimal substructures
    - The optimal substructure is how we express the solution to the big problem as solutions to smaller problems.
    - BF used $d^{i+1}(s,v) = min_u(d^i(s,u) + w(u,v))$
    - FW used $D^{(k)}[u,v] = min(D^{(k-1)}[u,v], D^{(k-1)}[u,k] + D^{(k-1)}[k,v])$
    - The fact that we cared about all of the pairs in FW, but wouldn't work for BF because we aren't just considering distances from the sources to the destination.
    - For FW, its substructure is best since it only has to take the minimum of two things, whereas using BF for this task takes min over all neighbors U. Substructure should be designed to fit the problem.

Algorithm Strategies

- Book Recipe
    - Identify a small collection of subproblems
    - Show how to solve larger problems using solutions to smaller ones
    - Infer the final solution from the solutions to all subproblems
- Class Recipe
    - Identify optimal substructure
    - Recursively define the value of an optimal solution

    - Find the optimal value - write the pseudocode to find the value of our optimal solution
    - Find the optimal solution - how do we modify the code to keep track of what is included in our optimal solution
- Find a recursion that determines how we would find an optimal solution for the jth input given a solution for smaller indices up to j. This typically involves breaking into cases - whether the jth input improves our previous solution or not determines whether it is included in the jth optimal solution
- Jump into the middle of the problem imagining someone has handed us a solution up to the point we are considering
- The recursion helps define the invariant of the problem. The recursion itself would take many redundant calculations, but we use a cache to limit our calculations to one or two recent iterations by looking at the stored values from the last iteration.
- May make sense to work bottom up when the solution to a subproblem depends on the solution to more subproblems - then we are using nested for-loops
- For running times, consider the invariant on inputs - eg. every recursive call is given some G_i already computed, how many such G_i exist at the end. The number of subproblems is a lower bound on the running time of the algorithm. Running time is at most:
    - (# subproblems x time per subproblem) + postprocessing
- Differences in approach from divide and conquer
    - subproblems may be split into multiple types / cases instead of a single invariant
    - subproblems recur across different recursive calls so caching has obvious benefits
    - often working on reducing exponential time algorithms vs polynomial
    - correctness is considered first and foremost over running time
    - subproblems are often just slightly smaller than their parents
- Bottom up: Initialize cache, using the recurrence relation to determine the dimension, ie. how many indices do we need. Define base cases to ensure we have filled in the values reference later in the code. Fill in the cache and return the value desired. Runtime tends to be easy then - often constant time in work, then just size of the cache is the run time.

Knapsack

- Essentially constrained optimization. Various items in a knapsack with values and weights. Want to maximize value for a constrained weight. Can also have an unbounded knapsack, or a 0-1 knapsack. The first has infinite copies of each item and the second only has a single copy of each item.
- Unbounded: Solve the problem for smaller knapsack and work up to a larger knapsack. Runs in $O(nW)$
    - We only care about the weights, not really the items added to the knapsack
    - Optimal substructure: solve problem for a smaller knapsack and increase the capacity. If S is an optimal solution for capacity x, then the optimal substructure for capacity $x - w_i$ is $S - v_i$ - since if we could do better it would be part of the optimal solution S for the larger problem too.
    - Definitions: k[x] is the optimal value for capacity x. Say $k[x] = V$
    - $k[x - w_i] = V - v_i$ the value before we added the last item to fill the knapsack.
    - We take the max over all previous possibilities plus a new item’s value
    - W is the capacity of the backpack, sum of weights that fit

    - Items array keeps track of the items used
    - Looping over capacities x and items i: $K[x] = \max\{K[x], K[x - w_i] + v_i\}$
    - Optimal substructure
        - Define S as optimal solution to problem with $n \geq 1$ items. S is either an optimal solution for the first n-1 items with capacity C or an optimal solution for the first n-1 items with capacity C - s_n supplemented with the last item n. The second case creates an optimal subproblem in which we reserve space for the last item n
        - Therefore for $V_{i,c}$ the maximum total value of a subset of first i items with capacity w, $V_{i,c} = \begin{cases} V_{i-1,c} & \text{if } s_i > c \\ max(V_{i-1,c}, V_{i-1,c-s_i} + v_i) & \text{if } s_i \leq c \end{cases}$
        - Loops over the items i to n, then over the capacities 0 to C
    - Reconstruction
        - Can have the algorithm yield the value of the optimal solution, then reconstruct what items actually are in the optimal solution by traversing the structure created. Input of the array created then checking the neighbors up and to the left to see where the current value came from.
- 0-1 Knapsack
    - Optimal Substructure: 0/1 knapsack with fewer items. Solve the problem with restricted set of items then increase items and size of knapsacks. Indexed by x and j - first j items with capacity x
    - Can only take one or 0 iterations of each item. We will keep track of the items used. Each subproblem requires the memory of what was used before it since we can only use each item once. Therefore we leave out one at a time.
    - 2 Cases: either we use item j or we do not use item j
    - If we don't use j , then we have the same solution as for j-1 items considered: j = $k[x, j-1]$. If we use j, then the optimal solution for j-1 must remove the weight of j and its value: j = $k[x - w_j, j-1] + v_j$ We take the max of previous values when we don't use j, or we take the previous value (less j's weight) and add the value of j.

BFS as DP

- Optimal substructure - shortest path using $\leq i$ edges
- Recursive formulation: $d^{i+1}(s, v) \leq d^i(s, u) + d^i(u, v) = i + 1$ One way to get to s to v, is s to u, then u to v. There might be better paths, so we use $\leq$

Weighted Graphs

- Cost of a path is a sum of the weights along that path. Shortest path has minimum cost
- Assume no negative cycles, since would travel it infinitely. But we can have negative weights.
- **Bellman-Ford**: Using DP to find shortest path. Runs in O(nm)
    - If there is a negative cycle reachable from s, then the Bellman-Ford algorithm detects and reports "Negative Cycles". If there is a path from s to v, then there exists a shortest path from s to v has at most n − 1 edges. Why? If a shortest path has a cycle, the cycle cannot be negative and we can remove it and improve its total distance.
    - Optimal substructure - shortest path using $\leq i$ edges. i is like a budget allowed to a path as the number of vertex hops we make, as we increase i we allow more hops and open up more potential

  - paths. Instead of limiting the input, we are limiting the size of the output produced by the algorithm.
  - Recursive formulation $d^{i+1}(s,v) = min_u(d^i(s,u) + w(u,v))$. Last vertex was some neighbor u, so the distance is the last distance to u plus the weight from u to v.
  - Lemma: A sub-path of a shortest path is also a shortest path
  - Definition: Simple Path in a graph with n vertices has at most n-1 edges in it, ie. simple means that the path has no cycles. There is a shortest path with at most n-1 edges when we have no negative cycles.
  - If no actual path from u to v, could think of it as an infinite weight
  - Stopping criterion: once the output for a recurrence is the same as the previous one, no improvement can be made. With no negative cycles, guaranteed to terminate by the time $i = n$. Bottom up, finding the shortest path from i = 0, working up to the path we want to calculate 0 to n - 2.
  - Could improve by looping over only the paths that updated in the last iteration - since if it did not update, we won't have a new shortest path for it in the next iteration.
  - Can just keep the previous two rows in the table, since rest not used for further computation.
- **Floyd - Warshall**
  - Want to find all pairs of shortest paths (APSP)
  - n by n table of all sources to all destinations with all 0's on diagonals.
  - Running with B-F, now have n more starting points - $O(n^2m)$, could be even worse if $m = n^2$. This is the time to beat
  - Define a substructure of a collection of points s.t. u has one path entering substructure and one leaving to reach v
  - Optimal substructure: For all pairs, u,v, find the cost of the shortest path from u to v, so that all the internal vertices on that path are in $1, \ldots, k-1$.
  - Given the shortest path from u to v using a collection of k-1, how do we find it using a collection of k? Either we use the kth vertex or not. If it doesn't then the shortest path did not change $D^{(k)}[u,v] = D^{(k-1)}[u,v]$. If it does, then we must have already calculated the path from u to k and the distance from k to v, since it follows a subpath using nodes for which we have calculated distances. $D^{(k)}[u,v] = D^{(k-1)}[u,k] + D^{(k-1)}[k,v]$
  - Left with recursive formulation: $\mathrm{D}^{(k)}[\mathrm{u},\mathrm{v}] = \min\left\{\mathrm{D}^{(k-1)}[\mathrm{u},\mathrm{v}], \mathrm{D}^{(\mathrm{k}-1)}[\mathrm{u},\mathrm{k}] + \mathrm{D}^{(\mathrm{k}-1)}[\mathrm{k},\mathrm{v}]\right\}$
  - We have n^3 iterations, so $O(n^3)$. So we saved $\frac{m}{n}$ time from BF. Every vertex has at least one edge and at most n^2 so this is an improvement.

Greedy Algorithms

- Construct solution iteratively with short term decisions, hoping it makes for an optimal solution in the end. It is often possible to construct multiple competing greedy aglortithms for a problem. The main issue is proving correctness - many greedy algorithms are not correct
- Exchange Arguments - prove that every feasible solution can be improved by modifying it to look more like the output of the greedy algorithm. Given an optimal solution, how does flipping two adjacent orderings change the optimality of the algorithm?
- Often can conceptualize a DP problem - but we might be able to solve without solving all subproblems first. Greedy algos typically have a top-down approach - make a choice then solve a subproblem

- Approach:
    - Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve
    - Prove that there is always an optimal solution to the original problem that makes the greedy choice
    - Demonstrate optimal substructure - having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice we have made we arrive at an optimal solution to the original problem
- Recall optimal substructure - an optimal solution to the problem contains within it optimal solutions to subproblems
- Hard to come up with the right algorithm, easier to write the pseudocode and analyze the run time. It often doesn't work but it is worth trying because they tend to be simple and run quickly. Best for when we have especially nice substructure.
- Why does it work? We never rule out an optimal solution when we make a choice.
    - Claim: After t iterations there still exists an optimal solution optimal solution that extends what we have already selected.
    - After 0 iterations, an optimal solution still exists as an option
    - Then at the end of the algorithm, there cannot be another iteration, so the solution itself selected must be optimal
    - Given optimal solution that includes a_j, if our greedy algorithm includes a_k, then we must show that an optimal solution exists with a_k. To show this we swap a_k and a_j. Show that it is a feasible solution (doesn't violate our rules) and that it is optimal
- Optimal substructure is important and used - we need our initial choice to not affect the overall problem. We have just one sub-problem at each step instead of many like in DP.

Activity Selection

- Activities that have certain time spans, want to choose acvities that do not overlap. Activities a_i, starting times s_i, finish times f_i. Want to maximize the number of activities you can do
- Shortest job? Could conflict with more jobs and prevent us from filling schedule. Fewest conflicts first? Could reduce the total number of activies we select. Earliest ending time first? Cannot come up with a counter example, should guide your intuition to try it.
- Pick the activity with the smallest finish time, then continue picking earliest finish time that doesn't conflict with already selected schedule.

Job Scheduling

- Jobs do not have specified time slots, but take certain amount of time to complete
- n tasks, task i takes t_i hours. For every hour that passes until task i is done pay c_i - ie cost is cumulative up to the time when the task is complete.
- This problem breaks up nicely into subproblems - if ABCD is optimal, then BCD is the optimal when just considering jobs B, C, D
    - $Cost(A, B, C, D)$
      $= C_A t_A + C_B(t_A + t_B) + C_C(t_A + t_B + t_C) + C_D(t_A + t_B + t_C + t_D)$
      $= t_A(C_A + C_B + C_C + C_D) + t_B(C_B + C_C + C_D) + t_C(C_C + C_D) + t_D(C_D)$

    - Then $Cost(B, C, D) = t_B(C_B + C_C + C_D) + t_C(C_C + C_D) + t_D(C_D)$
- Take the best job first, then repeat for the subgroup without the first job. Notice above we could break out problem by costs or by time. We need to consider both cost and time - take a ratio to maximize - $\frac{\text{cost of delay}}{\text{time it takes}}$
- Claim: After t iteration the optimal schedule extends the first t jobs we scheduled
    - Swapping our chosen job with the next optimal job - need to simply show that our solution T is better than the optimal solution handed to us T^*. Want to swap adjacent terms only so we otherwise have the same order. Should be able to analyze the costs of two adjacent jobs and determine which is an optimal solution.
    - Or could say assume by contradiction T^* is optimal and doesn't follow our criteria. Then swapping changes anything it can only make it better. If it doesn't change anything then the ratios are the same of the swapped elements.

Huffman Codes

- Variable length code - giving frequent characters short codewords and infrequent characters longer codes.
- Prefix code - no codeword is also a prefix of some other codeword. Create unambiguous codes for variable length codes
- Can be represented in a binary tree in which only leaves are characters - non prefix would have other node characters
- Cost = p(x) d(x) - probability of character showing up times the depth of the character in the tree. Want to minimize the overall cost of the tree.
- Set of C leaves, performs a sequence of C - 1 merging operations to create the final tree. Uses a min priority queue Q keyed on the frequency attribute, ie always merges the least frequent character first to be siblings in the tree. Treat the merged characters as a single character with the sum of their frequencies.

Minimum Spanning Trees

- Given undirected connected graph in whcih each edge has a real-value cost. Goal: compute a spanning tree of the graph with the minimum possible sum of edge costs.
- Spanning tree is a subset $T \subset E$ of edges that satisfies: 1) T is acyclic (ie. a tree) 2) for every pair of vertices, T includes a path between v and w (spanning)

Graph Cuts

- A cut is a partition of the vertices into two parts. Any partition, does not have to be a single connected component, can be disconnected and be in the same set.
- A set of edges S in G. A cut respects set S if no edges in S cross the cut we chose.
- Light edge - the smallest weight of any edge crossing the cut
- Lemma - suppose there is an MST containing S, (u,v) is a light edge. Then there is an MST containing S U (u,v). Breaking this down:
    1. Some cut that respects S
    2. Edge (u,v) is the lightest edge that crosses the cut
    3. Some MST that extends S (or contains S)

    - If all of those conditions are met, then there is an MST that extends S and also (u,v) (ie. the MST contains these edges)
- Therefore if we are building an MST and we have a cut, we know we can add the lightest cut edge to the MST.
- Essentially back into Prim - have a set S containing the nodes we have added to the MST and the cut crosses the boundary to in and out of the spanning tree. Then safely add the lightest edge across the cut, since it does not rule out future optimality. Show that at each choice we meet the conditions from the lemma - respects S, picked the lightest edge, MST extension is the inductive hypothesis

Prim's Algorithm

- Very similar to Dijkstra
- Start by choosing an arbitrary vetex. We will greedily construct a tree one edge at a time, adding the cheapest edge that extends the reach of the tree. We look over all available edges from the tree we have so far and pick the cheapest (even if from the original vertex)
- Only considers edges that cross the frontier from X (already in the tree) to V-X (still outside the tree). After n-1 iterations, all edges are in X and the algorithm halts.
- Runs in $O(nm)$ time - In each iteration searches over all edges to find the cheapest one with endpoints in X and V-X. Runs n-1 times with each iteration taking $O(m)$
- If instead it is implemented with heap, since we are continuously doing a min search, runs in $O((n+m)log(n))$ time with Fib Heap. Have the key of each vertex $w \in V - X$ be the minimum cost of an edge (v,w) for $v \in X$, or ∞ if none exists. Essentially each vertex is storing the minimum distance to the current existing tree. This means we preprocess the minimum cost in creating the heap, then need to update the heap after extract min to recalculate the min for each vertex (direct neighbors to the tree are affected). Vertices connected to the vertex x added to X either maintain the cost they had previously to X or it needs to be updated to reflect the cost of that vertex to x.
- Minimum bottleneck property - For graph with real-valued edge costs, an edge $(v, w) \in E$ satisfied the MBP if it is a minimum bottleneck v-w path, ie iff there is no v-w path consisting solely of edges with cost less than the c_{vw}.
- Every edge chosen by Prim satisfies MBP
- MBP implies MST - if every edge of T satisfies the MBP then T is a MST.
    - When we add an edge to a graph it either adds a cycle (Type C) or fuses two connected components to one (Type F)
    - Spanning trees have exactly n-1 edges
- Difference from Dijkstra - take the min previous distance to the tree and the edge weight to the tree - do not care about distance to the source anymore. So could pick different tree since we may not pick the shortest path from s to t to be in the MST if there are other edges connecting them with lower weights.

Kruskal's Algorithm

- Also greedily constructs a spanning tree one edge at a time, but grows multiple trees in parallel. Chooses the cheapest edge over the whole graph that does not create a cycle
- Total run time $O(mn)$. Sorts edge array of the graph with m entries - $O((m+n)logn)$, contains loop m times to check with edge can be added to the solution with creating a cycle, needs to check for cycles using BFS/DFS, which takes $O(m+n)$, leading to overall run time of $O(mn)$

- We are making a forest of disjoint trees, each step merges two trees when we add an edge. Can use the same graph cutting lemma to show optimality.
- Can be upgraded to $O(mlogn)$ using the union-find data structure.
  - union-find maintains a partition of a static set of objects. Initialize $O(n)$ each object is its own set, and these merge over time. Running Find $O(log(n))$ - given a union-find data structure and an object x in it, return the name of the set that contains x. Union $O(log(n))$- given two objects in the structure, merge the set that contain x and y.
  - Implemented as an array with one position for each object and a parent field that stores the array index of some other object - essentially a directed graph. We initialize with each vertex as its own parent. Find then travels down the parent relations until it cannot any further - WC RT is then the height of the tree. The depth of an object is the number of parent edge traversals performed by Find from x.
  - Union then demotes one parent root and promotes the other. There are several ways to do this, but we add the promoted root as the parent of the demoted root to minimize the increase in depth to the demoted tree vertices to 1. Want to minimize the increase in depth so demote the smaller tree, meaning we need to store the size of the tree as well, updated at each merge. Performs two Finds and some constant work, keeping it $O(logn)$
  - Checking for cycles when an edge is added is equivalent to checking whether the endpoints are already in the same CC - requires two Find operations

Min Cut / Max Flow

s-t Cuts

- Source vertex s, sink vertex t
- Edges only count towards a cut if it goes from s side to t side - cost of a cut if the sum of the capacities of the edges that cross the cut
- Minimum s-t cut - separates with the minimum capacity
- In addition to a capacity, each edge has a flow, constrained by its capacity. At each vertex the incoming flows equal the outgoing flows except for s and t
- The amount of flow = amount flowing out of s = amount flowing into t

Maximum Flow

- A directed graph can be seen as a flow network, with material moving from a source to a sink with each edge possessing a capacity
- Flow conservation - the rate at which material enters a vertex must equal the rate at which it leaves the vertex - no accumulation
- We wish to compute the greatest rate at which we can ship material from the source to the sink without violating any capacity constraints
- **Max-Flow Min-Cut theorem**: The max flow from s to t is equal to the cost of a min s-t cut
  - Lemma 1: For any s-t flow and any s-t cut, the value of the flow is at most the cost of the cut max flow $\leq$ min cut. Cannot have flow that is more that the capacity bridging two areas since all material must bridge the cut divide

- Flow network - G = (V, E) is a directed graph in which each edge(u, v) $\in$ E has a nonnegative capacity c(u,v) ≥ 0. If E contains edge (u,v) then there is no reverse edge (v, u).
- For s source and t sink, a flow in G is a real valued function $f : V \times V \rightarrow \mathbb{R}$ satisfying
    - Capacity constraint: For all u,v in V $0 \leq f(u,v) \leq c(u,v)$. The flow from one vertex to another must be nonnegative and must not exceed capacity
    - Flow conservation: For all u in $V - \{s,t\}$, $\sum_{v \in V} f(v,u) = \sum_{v \in V} f(u,v)$. If (u,v) not in E, the f(u,v) = 0. Flow in equals flow out (except for sink and source).
- The value of the flow $|f| = \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s)$ - the total flow out of the source minus the flow in to the source. Typically flow into source is 0, but included for generalization
- Graphs with antiparallel edges - violate our assumption that if $(v_1, v_2) \in E$ then $(v_2, v_1) \notin E$. Choose one of the two antiparallel edges, split it by adding a new vertex v' and replacing edge (v_1, v_2) with (v_1, v') and (v', v_2) with capacity for both new edges equal to the capacity of the original edge.
- Multiple sources and sinks - can convert to a standard maximum flow problem. Add a supersource s and add directed edge (s, s_i) with capacity $c(s, s_i) = \infty$ for all sources. Add a supersink t with directed edges $(t_i, t), \text{ capacity } c(t_i, t) = \infty$. Supersource provides as much flow as needed for the multiple sources and supersink consumes as much as needed.

Ford-Fulkerson Method

- Iteratively increases the value of the flow. Starting with $f(u,v) = 0$ for all u,v giving an initial flow of 0. At each iteration increase the flow value in G by finding an augmenting path in an associated residual network G_f. The flow monotonically increases with each iteration but individual edges might increase or decrease. Repeats until no more augmenting paths can be found.
- Residual network - given a flow network G and flow f, the residual network G_f consists of edges with capacities that represent how we can change the flow on edges of G. Each edge can admit the capacity - current flow. If that difference is positive then we place that edge in G_f with residual capacity $c_f(u,v) = c(u,v) - f(u,v)$.
- Edges can be added to G_f that are not in G. If we need to reduce the flow to an edge (u,v), we place an edge (v,u) in G_f with residual capacity $c_f(v,u) = f(u,v)$ to cancel out the flow on (u,v). Therefore residual capacity is formally defined as $c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E \\ f(v,u) & \text{if } (v,u) \in E \\ 0 & \text{otherwise} \end{cases}$
- Say we have a directed edge from A to B, with 1 unit of flow and 6 capacity. If we want to route 1 unit of flow from B to A, this is the same as routing -1 from A to B, ie. reducing our flow of A to B to 0. In the residual network, we add the 1 unit of pseudoflow from B to A as well as the residual flow from A to B of 5. The flow from B to A corresponds to $f(v,u)$, ie we can only reverse flow that is already flowing from A to B. (Condition if $(v,u) \in E$ really the same as testing if (u,v) in E). Notice the residual + reverse residual edge is equal to the capacity.
- **In the residual network, t is not reachable from s in G_f iff f is a max flow.** Then need to augment the flow in the original graph. This should cause some residual edges to drop to 0, deleting that edge. At some point we remove a viable path from s to t and we are left with a max flow.
- Augmentation of flow f by f' - $(f \uparrow f')(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) & \text{if } (u,v) \in E \\ 0 & \text{otherwise} \end{cases}$
    - Pushing flow on the reverse edge in the residual network is called cancellation

  - An augmenting path p is a simple path from s to t in the residual network. We may increase the flow on an edge (u,v) of an augmenting path by up to $c_f(u,v)$ without violating the capacity constraint. The residual capacity of p is the maximum amount by which we can increase it - $c_f(p) = min(c_f(u,v) : (u,v) \text{ is on } p)$ Then $f_p(u,v) = \begin{cases} c_f(p) & \text{if } (u,v) \text{ is on } p \\ 0 & \text{otherwise} \end{cases}$
- Augmenting - if we use a forward edge we increase its flow, if we have a backward edge, we decrease its flow.
- **The flow is maximum iff its residual network contains no augmenting path**
- A cut (S, T) of a flow network is a partition of V into S and T = V - S st s in S and t in T. If f is a flow the the net flow f(S, T) across the cut (S, T) is defined as $f(S,T) = \sum_{u\in S}\sum_{v\in T} f(u,v) - \sum_{u\in S}\sum_{v\in T} f(v,u)$. The capacity of the cut (S, T) is $c(S,T) = \sum_{u\in S}\sum_{v\in T} c(u,v)$
- A minimum cut of a network is a cut whose capacity is minimum over all cuts of the network. For capacity we count only capacities of edges going from S to T, while flow nets against the reverse direction.
- The value of any flow f in a flow network G is bounded from above by the capacity of any cut of G
- **Max flow min cut theorem**: The max flow from s to t is equal to the cost of a min s-t cut
  - If f is a flow in a flow network G = (V, E) with source s and sink t then the following conditions are equivalent:
  1. f is a maximum flow in G
  2. The residual network G_f contains no augmenting paths
  3. $|f| = c(S,T)$ for some cut (S,T) of G
- Summarizing: in each iteration we find some augmenting path p and use p to modify the flow f. We replace f by $f \uparrow f_p$ and get new flow of value $|f| + |f_p|$. The run time depends on how we find the augmenting path p. Using BFS, the algorithm runs in polynomial time, $O(E|f^*|)$ for maximum flow in network transformed to have integer capacities f^*. Can be large for large capacity networks

Edmonds Karp

- Find the augmenting path with BFS - choose the augmenting path as a shortest path from s to t in the residual network where each edge has unit weight. Then for all vertices $v \in V - \{s,t\}$ the shortest path distance from s to v in the residual network increases monotonically with each augmentation.
- Not E-K: Fattest path algorithm - pick the path with the biggest capacity. Use F-F with the fattest path rule, finishes in time $O(m^2 log(f))$ for f max flow. Can do binary search over the edge capacities to find the fattest path, then run BFS on the fattest path.
- E-K: Use the shortest path with F-F. Runs in time $O(nm^2)$

Maximum Bipartite Matching

- Given an undirected graph a matching is a subset of edges M st for all vertices v in V at most one edge of M is incident on v. We say that a vertex v is matched by the matching M if some edge in M is incident on v. A maximum matching has the maximum amount of matched vertices.
- We want to construct a flow network where flows correspond to matchings. The corresponding flow network $G' = (V', E')$ for the bipartite graph G: let s and t be new vertices not in V and $V' = V \cup \{s,t\}$. Since vertices are part of bipartite graph $V = L \cup R$, the directed edges of G' are the edges of E directed from L to R along with |V| new directed edges from s to all in L and R to t. We assign unit capacity to each edge in E'.

- Integer valued - f(u,v) is an integer for all (u,v) in V
- If M is a matching in G then there is an integer valued flow f in G' with value |f| = |M|. Conversely if f is an integer valued flow in G' then there is a matching M in G with cardinality |M| = |f|.
- If the capacity function c takes on only integral values then the max flow f produced by the F-F method has integral |f| and for all edges f(u,v) is integral. Therefore the cardinality of a max matching M in a bipartite graph G equals the value of a max flow f in its corresponding flow network G'.
- Given each edge with capacity 1, then each L is matched with 1 R.
- Great for assigment problems

Global Minimum Cuts - Karger

- An s-t cut was defined to be a partition of V into sets A and B such that $s \in A$ and $t \in B$.
- For a cut (A, B) in an undirected graph G, the size of (A, B) is the number of edges with one end in A and the other in B. A global minimum cut is a cut of minimum size. Thus the global min-cut is a natural "robustness" parameter; it is the smallest number of edges whose deletion disconnects the graph.
- Naive: Over n vertices, we have roughly two choices of cuts per vertex, we have 2^n possible cuts to looks for. Another option: look at all pairs of vertices, assigning s and t to each pair, trying s-t min cut; then we have running time $n^2 * t(F - F)$
- Global min cuts useful for clustering, seeing where there are denser connections.
- Karger / Contraction Algorithm - given multigraph, undirected graph allowed to have parallel edges, choose edge in G at random and contract it, ie combine u,v into new node w. Note that the new graph G' could have parallel edges due to vertex contraction. Recurse, creating many supernodes in G'. The algorithm terminates when there are only two supernodes - this is the minimum cut.
- Note in our implementation we use an unweighted, undirected graph, though the algorithm can be extended to include other graph types. Parallel edges become superedges / or remain parallel edges.
- Running time - we contract n-2 edges, naively each contraction takes $O(n)$, since there may be n nodes in the supernodes we are merging. We just have two lists of edges that connect some super nodes, we need to remove those edges when we contract. So the total running time is $O(n^2)$. We could get a better runtime using a union-find data structure - generate a random assortment of edges in $O(m)$ so the whole thing takes $O(m\alpha(n))$
- Karger is not always correct, unlike QuickSort - QuickSort is a Las Vegas Randomized Algo, always right but sometimes slow. Karger is a Monte Carlo randomized algorithm, it is always fast but sometimes wrong.
- It is wrong iff we choose any of the edges that cross the min-cut, then we will never find the min-cut. Probabilistically, Karger is skewed towards selecting smaller cuts compared to the uniformly random cut. The smaller the cut, the lower the probability that we ever ruin it - Karger is more likely to choose cuts that have many edges. Can run Karger multiple times to get a high probability of finding the global minimum.
- We can actually produce a global min-cut with high probability of $\binom{n}{2}^{-1}$. Define p as the probability of success in one trial. P(T trials fail) $= (1 - P)^T$. If we want failure probability at most δ, choose $T = \binom{n}{2} ln(1/\delta)$. (P(Fail in T trials) = $(1 - P)^T \leq e^{-pT} = e^{-ln(1/\delta)} = \delta$)
- So to succeed with probability $1 - \delta$, need to run Karger $\binom{n}{2} ln(\frac{1}{\delta})$ times

- Theorem: Probability that we ever ruin the minimum cut is at most $\frac{1}{\binom{n}{2}}$
    - Suppose min cut is of size k - then all vertices must have degree at least k. This means graph has at least $\frac{nk}{2}$ edges
    - After i-1 merges, at least $\frac{(n-i+1)k}{2}$ edges are alive. So probability the k min edges are alive is at least $1 - \frac{k}{(n-i+1)k/2} = \frac{n-i-1}{n-i+1}$
    - Over all iterations i, get $\frac{n-2}{n}\frac{n-3}{n-1}\ldots\frac{1}{3} = \frac{2}{n(n-1)} = \frac{1}{\binom{n}{2}}$
- An undirected graph G on n nodes has at most $\binom{n}{2}$ global min cuts.

Karger-Stein

- Repeating Karger is expensive - $O(n^4)$. K-S will also get us success probability 0.99 in $O(n^2 log^2(n))$. We pretty much always use K-S since it is always faster but Karger gives us the intuition
- As Karger progresses, the probability of picking an edge on the min cut increases - our choices get riskier. But this means if we are repeating the Karger algorithm, we can skip repeating the early iterations since they were very likely to make good choices.
- After a number of iterations, fork our graph and have branches run Karger in parallel. We will run Karger until there are $\frac{n}{\sqrt{2}}$ supernodes left, which is where there is roughly a 1/2 probability of the min cut still alive.
- Split into two independent copies G_1, G_2, recurse running K-S on first and second copies and return the better cut of the two. Keep splitting at $\frac{\frac{n}{\sqrt{2}}}{\sqrt{2}} = \frac{n}{2}$ for each iteration.
- Karger draws a tree from 1 node of n to 1 leaf and we do this n^2 times. K-S draws 1 tree, starting from n down to n^2 leaves of 1. We save nothing at the bottom with K-S, but we save a lot at the top, just a single root node instead of repeating the linear tree from top to bottom.
- Depth of recursion tree is $log_{\sqrt{2}}n = 2logn$, number of leaves is $2^{2logn} = n^2$. Still need to see at least n^2 cuts like Karger to ensure we see the best one, but we save on computation by skipping the earlier iterations.
- Recurrence relation: $\mathrm{T(n) = 2T(n/\sqrt{2}) + O\left(n^2\right)}$ for a single run. For $\delta = 0.01$, get $O(n^2 log^2(n))$
- The probability of success is $p = \frac{1}{depth+1}$ for depth of tree.

Stable Matchings

- How should we match doctors to residencies? Both have preferences for each other. For simplicity, assume each hospital has 1 slot, n hospitals and n doctors.
- With a bipartite graph with weights expressing preferences for doctors, could look for the maximum weight bipartite matching. Hungarian algorithm solves this max weight matching problem once we have the preferences. Some doctors might misreport their preferences, some matchings made outside of algorithm, etc.
- Blocking pair - given matching M, (i, j) are a blocking pair if they prefer each other to their assignment M. Imagine a doctor and hospital matching outside of the algorithm
- Stable matching - a matching M without blocking pairs
    - For every unmatched (i, j), either doctor i prefers hospital M(i) over hospital j and/or
    - Hospital j prefers doctor j over doctor i
- Resolves the problem of doctors matching outside of algorithm - ie. there is no pair that would prefer to

subvert the outcome of the algorithm and match themselves

- Input: a list of preferences, ie. a permutation of {1,...,n} from each doctor / hospital

Deferred Acceptance / Gale-Shapley

- Looks like a greedy algorithm with one exception - the decisions are revocable since we are possibly ruling out an optimal solution at each step.
- Try to match each doctor to top choice as first iteration matching
- If blocking pair discovered, just switch the matching to get rid of blocking pairs
- While an unmatching doctor exsists, try to match i to next favorite hospital in the list.
- Running time - worst case we run the while loop n times, going through the doctor's full preference list. There are n doctors so total $O(n^2)$
- The matching is doctor-optimal, every doctor is matched to favorite hospital possible in any stable matching. No incentive to misreport their preferences. However, the matching is hospital-worst and hospitals can gain from gaming their preferences.
- The doctor optimal matching is always the hospital worst matching and vice versa.
- Rural hospital theorem - If a hospital doesn't fill all its positions in some stable matching, then it is matched to exactly the same set of doctors in every stable matching.

Review

- B-F, F-W handles negative edge weights - should really only be considering using these algos when this limiting condition holds
- Remember Dijkstra is SSSP meaning from a single node the shortest paths to every other node
- DC, DP, Greedy - consider the substructure of the problems, are they overlapping like in DP? Single like in Greedy? Non-overlapping and multiple like in DC?
- MST - think if you have some network and need a minimum cost. MST often useful a primitive as part of another algorithm
- Kruskal vs. Prim - consider sparsity of graph - double check when to use each
- Be specific about the type of min cut problem - global, s-t
- F-F updating flow is like adding the new values of flow sent through residual to the current flow - with reverse edges being negative
- Fattest path = $O(m^2 log(n) log(|f|))$ E-K $O(nm^2)$. Use how these values compare to each other - if log f is huge. But would have to know the flow to know the run time of fattest path
- Should be able to make a residual graph
- Global min cut is completely separate from s-t. Here we have an undirected graph, find a min cut S, partition of the nodes into two groups st the cut value is minimized.
  - When you have a group of things are need to split into two groups somehow.
- Stable matching - some notion of stability and a blocking pair - maybe expect as a short answer
  - Doctors proposing then hospitals either tentatively accepting or rejecting forever
  - Bipartite settings have preference constraints that make this work well, otherwise cannot assume no cycles etc
- Greedy Example - difference from hotels is there is no cost / reward associated with each hole.

    - Walking to farthest hole doesn’t rule out stopping at another hole between current position and the end
- Transportation example - create an air node and draw edges with ai costs, run mst with roads only and with airports.
- Thief example - s-t cut, cutting off resources from source to a sink. But we need directed edges with weights. Making them bidirectional edges, turn multi-edges into weights.
    - when the thief can move, then the source is essentially changing
    - create DP cache to hold the mincuts for each vertex
    - F-F to get each min cut
    - If can’t blow up some edges, make them weight infinity or collapse to supernodes
    - The tweak of dynamic source node means its somewhat in between all of our algorithms
    - Remember DP, since graph things very often have overlapping subproblems

Probability Reference

- Binomial - n trials of a Bernoulli. Expectation = np
- Bernoulli - indicator with x=1 with probability p, x=0 with probability 1-p. Expectation = p.
- Geometric - number of trials until you see a success, where probability of success in each trial is p. Expectation = $\frac{1}{p}$