Wed homework, due next Wed

Pen and paper HW 20%

Midterm Mon 7th week 30%

Final 50%

Scoring: maximum between hw and the weighted?

Office hours MW 2-3

Lec 1

Problem 1: binary expansion; input: n; find A0...Am

Sigma(i=0...m, Ai \* 2^i) = n

Induction proof of recursive algorithms (the recursion is the proof)

Lec 2

**Exponential algorithms** are beyond the topic of this course

We discuss **polynomial time algorithm**

Repeated divided by two algorithm

Graph G(V, E), minimum spanning tree problem.

In this class, we deal with the worst case.

In this class, we deal with deterministic algorithms.

**Sorting**

O(n^2), essentially bubble sort, replication of the binary expansion method 1 (what that is?)

O(nlog(n)), essentially merge sort, replication of binary expansion method 2 (repeatedly divide by 2)

Flip coin merge sort to decide pivot

Average analysis

Np ?= p

**Homework requirements**

Idea of the algorithm, pseudocode, simple clear English

Don’t do brute force

TA session 1: time analysis, factors

# of operations

Size of input

Time per operation

f = O(g), f = Omega(g) ⇒ f = Theta(g)

Theta: tight bond

Elementary school multiplication: Theta(n^2)

Polynomial time: If T(n) = O(n^c) for some c, then T(n) is polynomial time algorithm

Exponential time: T(n) = O(c^n)

Example problems (recursion):

a^n ( = a^(n/2) + a^(n/2), better than = a \* a^(n-1) as the result of the first case need only be calculated once)

The tower of Hanoi

### Lec 3

**Induction**; example

* binary representation;
* **merge sorting**

Cut to two halves, sort each; two pointers looking at the start of each half and merge by moving pointers

Complexity: T(n) = T(n/2) + cn; On each level, there’s the merge cost of cn in total; number of level is log(n), so the merge sort complexity is O(nlog(n))

(In this class, details like n is odd or even does not matter, can assume n is 2^m)

Whatever you do, if it’s more complex than nlogn, you can assume the input is sorted (weakest link)

**Example: # of inversions** (Two rankings of movies, how close these two are?) (How many pairs whose orders are inverted in two ordered sequences; range: 0 ~ n(n-1)/2 )

Trivial algorithm: go over all the pairs

One way to do it: 2-3 tree (Just a B-tree with order 3)

The other way’s merge sort based (divide-and-conquer); the “sorted to compare against” for each subarray assumption (basis: transform two arrays into one array; essentially using one as the standard, as the items can be arbitrary sorted numbers); just change merge to counting, complexity’s still nlog(n).

**Binary search trees** (degree 2)

Extend: B+ tree (not necessarily, considering whether the intermediate nodes carry value or just index; so if the degree 3 tree in example’s not a B+ tree, what’s the “push-up” mechanism for this tree?)

Insert: log(n)

Example: Manhattan Skyline

Given (si, ti, hi) for building i (overlapping), want to find silhouette ( f(x) = max(hi) of i, s.t. si<=x<=ti ) (piece-wise flat, meaning?) (The problem should be rephrased as intending to build the skyline function (curve), essentially, keep the next highest when at an ei)

At most 2n break points in our “skyline”

Can be solved by divide and conquer (take the first half, take the second half, find the silhouette)

With this example, introducing the **Heap**, with the intention being that insert, delete and find max will be log(n); almost balanced, only right side of leaf level having potential blanks

Assuming all buildings have different heights. We keep the height for each node, parent node height bigger than that of the children. Array representation of heap: a[i] has left child a[2i], right child a[2i+1].

Deletion: hole created; take the last element and fill the hole, then sift down (flip with the bigger among the two children), (then sift up?) (local adjustment)

(Is this one heap between each pair of breakpoints?)

This is nlog(n), as we have O(n) break points, and each operation’s log(n)

### Lec 4: graphs

G = (V, E)

**Undirected**, when E = a set of sets of two nodes (set, as the sequence does not matter) ({u, v | u, v belong to V})

**Directed**, when E = a set of pairs of two nodes (pair, as the sequence does matter) ({(u, v) | u, v belong to V}) (pairs, or tuples, as they have sequence as well)

**Path**, a set (continuation) of edges

**Simple path**, where nodes in it are only visited once

**Trail**, allows the same node to appear twice

**Connected**, where there always exists a simple path from one node to another

G’ is the underlying undirected version of directed graph G if we ignore directions

**Tree**, an undirected connected acyclic graph G is a tree

The **degree** of the node, how many edges have the node

In a tree, a node with degree one is a leaf

A tree may not a have a leaf, if the tree has only one node; every tree that has at least two nodes has at least two leaves (proof for one leaf: start from a node, move until the resulting node is degree one; this process will terminate, as we are dealing with finite number of nodes, and the tree is acyclic) (proof for two leaves, as we have a leaf, start from that leaf, must reach another leaf; my proof by oddity, of each edge increases of total number of degrees by 2, is actually not a good proof because it could be 1+3)

The **diameter** of a tree: the longest path between any two nodes (diameter should start and end with leaves; proof)

Rooted tree is directed; we don’t talk about root / children when the tree is undirected

**Graph representation**

* Incidence matrix; 1 where the edge’s there; undirected: symmetric; Inefficient when the matrix is sparse
* Adjacency list: array position i has a list of nodes that are connected to node i; directed graph use outgoing adjacency list (or use incoming adjacency list). len(list[i]): degree of node i (undirected)

**How to find the diameter of a tree**

Brute force: find all pairs of nodes, calculate the length between each of them;

Recursively, remove all the leaves, we still have a tree, that’s where the induction “magic” comes in, each step length +2; if end up with nothing, +1; if end up with one node, +0

Implementation**:** Iterative, adjacency list based, O(n), n is the number of edges? (Would it be different from the number of nodes?)

(Phase by phase is O(n \* p), where p is number of phases; instead we store the nodes that become degree one because of this phase in another list; this is a common implementation trick)

**Celebrity problem**

Description: a celebrity is someone that everyone knows (assuming everyone doesn’t know themselves), but doesn’t know anyone; Given an incident matrix, look for if a celebrity exists in this matrix

Idea: there is at most one celebrity. Algorithm: judiciously take away a person from the table: among a pair of person, at most one can be celebrity; take out the one that’s definitely not a celebrity

### Week2 TA session

[Master theorem](https://en.wikipedia.org/wiki/Master_theorem)

Divide and conquer algorithm complexity analysis; For a form of T(n), three cases are given, it’s also possible that master theorem does not apply

In the case of master theorem not applicable, you can use the definition of T(n) itself

For example, nlogn = Omega(n), this is true

For any epsilon > 0, nlogn = Omega(n^(1+epsilon)) cannot hold

For example, T(n) = 2T(n/2) + nlogn cannot use master theorem, expand it step by step

T(n) = nT(1) + n(logn + log(n/2) + … + log(n(1/2)^logn)) = nT(1) + n(logn + log(n/2) + … + log1) < n(logn)^2

(logn)^2 = log^2n? Is this true?

Two simple applications of master theorem:

For quicksort, worst case is T(n) = T(n-1) + T(1) + theta(n), and we have O(n^2)

For mergesort, T(n) = 2T(n/2) + n, and we have O(nlogn)

Hw1. pb3. element school division should output the remainder as well

Pb3. should do proof of correctness

To formally prove the complexity of a divide and conquer algorithm, do the T(n) = f(n) stuff

Hw1. pb2. (formal) proof of correctness; proof by contradiction, suppose there is a different longer path p2, there will be a node n where p1 and p2 diverge. By the algorithm there exists a recursive call on this node, and for that call we have height(n.p1) > height(n.p2), which contradicts with the assumption

Topological sort of a graph

example

Courses and prerequisites, find if we can take all the courses: decide if the directed graph’s acyclic

Idea: each time remove the nodes whose in-degree is 0, the complexity would be O(n) (or max(n, e), whatever it takes to find in-degree 0s given the initial (v, E)), initial scan is n, and as we remove, we keep the newly removable nodes in a queue

There is a topological sort iff there are no cycles in the directed graph (what about undirected graph?)

### Lec 5: graph traversal

BFS (mainly applied in undirected graphs); DFS (mainly applied in directed graphs)

Topological sort (mainly applied in DAGs, directed, acyclic)

**BFS**

BFS example, for finding out if graph connected or not;

BFS idea: start with any node, take all its neighbors, clump them into one node (can work for parallel edges as well), until I exhaust the entire graph; we build a BFS tree along the process (n-1 edges picked, where n is the number of nodes)

Implementation: (FCFS) queue-based implementation (mark when enqueue instead of dequeue, only difference being when there are parallel edges? Not really, if nodes on the same “level” connect to each other;)

Complexity: O(|E|) (actually O(|E| + |N|, as we assume connected graph, |N| is ignored)), as we’ll look at all edges in the graph

Properties:

A node we found at phase >= (i+2) cannot be connected to a node we found at phase i

Phase i nodes can only be connected to phase i-1 and i+1 nodes; with this we divide the graph into layers

We won’t traverse inter-phase edges during a BFS

**Theorem**: an undirected connected graph with n nodes and n-1 edges has to be a tree

For a graph without inter-phase edges during a BFS, the cycles are going to have even lengths; and we have a bipartite graph (for all (w, u) in E, w in X, u in Y, where X and Y are two disjoint sets of nodes)

**Theorem**: Iff BFS starting from any nodes does not have inter-phase edge (=> all cycles are even length), the graph is bipartite

**DFS**

Stack-based implementation: in a directed context, take a node, check who I can reach from this node. (Recursively) go over my adjacency list, find the first node, suspend, think of that node as the one we started from (essentially a recursion with side effects: we mark the node as visited, which is the side effect as we are not starting “completely” fresh afterwards)

Also, mark when push rather than when pop (we have three states, unmarked, marked (saw it), finished; is the finished state necessary?)

We build a DFS tree with this traversal

Definition: strongly connected (directed graph): between every pair of nodes, there is a directed path; O(E) algorithm for finding out if a directed graph (represented as an outgoing or incoming adjacency list) is strongly connected: two DFSs, one following the direction, one countering the direction (Since, it’s O(E) to get an incoming adjacency list if an outgoing adjacency list is given)

Pre-order traversal, allocating numbers to nodes; every time encounter a new node, give it a number; And post-order, when we are done with a node, we number it

To be in a strongly connected graph, the relation between set of nodes is reflexive (by definition), symmetric and transitive; we can partition based on equivalence relation;

**Pbs:**

* Take a directed graph, we can partition it into strongly connected components. (O(E) algorithm for outputting this)
* Take a directed graph, is there a directed cycle in the graph? (Based on DFS)

**Topological sort**

Given a DAG, we want to number the nodes, such that start (tail) will have smaller number than the end (head, interesting definition)

Special nodes in DAG: there must be sources, who do not have in edges; and there must be sinks, who do not have out edges. (Proof, assuming no sources or sinks, we’ll run into loops unless the graph is “unlimited”)

O(N+E) algorithm as given in TA session slides; Idea: I’ll not give a number to a node, until I give numbers to all nodes outgoing to this; actually similar with BFS, “level-by-level”

### Lec 6: Greedy Algorithm

Algorithm: each step, commit, make progress, (greedy’s trying to make as much progress)

**Example: Sorting**

* Divide and conquer sort (such as merge sort, in which there’s no commitment at any point of time)
* Bucket sorting (scan, find a minimum, scan the rest until end; in which commitment’s made in the each round)

**Activity selection problem**

Description: group i submit applications (si, ei) (start and end time), i = 1..N. Want to schedule a maximum number of groups, such that no groups’ applications overlap

Solution: always schedule the group that ends earliest (this is greedy, as we’ve a criteria for selection, and after each round we commit)

Proof of correctness: contradiction (suppose there exists an optimal solution which does not contain the interval of the first termination time; case analysis on the first choice of the supposed optimal solution: <t1 (that produced by this algorithm), contradiction with how ti’s picked; =t1 contradiction with assumption; >t1, would result in inferior, as total events smaller than or equal to that found by the algorithm)

Implementation: with this thinking, it’s O(n^2) as it’s O(n\*(n+1)/2)

Implementation optimization: sort by end time, O(nlogn), pick the first group and look at the next non-overlapping group, which is O(n) in total. This is O(nlogn)

**Job, deadline scheduling problem**

Description: given a set of jobs each with a deadline and a processing time, want to schedule jobs so that the maximum lateness is minimized

Solution: order the jobs according to deadline, and do jobs in the same order

Proof of correctness: contradiction (Suppose there exists an optimal schedule that starts with a different job. Find the same job in the optimal schedule as given by the algorithm. Consider the job and its predecessor, switching the two would result in decrease of maximum lateness; until we switch it to the first position, etc)

Implementation: O(nlogn)

**Dijkstra’s algorithm (greedy)**

Description: shortest path from one node to another in a (directed) positive-weight graph

Side note: why is positive-weight significant?

* (With negative, you can’t guarantee that the next shortest path is attained from what we already have picked)
* <http://stackoverflow.com/questions/13159337/why-doesnt-dijkstras-algorithm-work-for-negative-weight-edges>

Idea

Step 1, picks the smallest adjacent node, combine the picked and source as one node

Each step, update the reachable nodes with shortest path, and continue from the shortest node

Implementation

* For one implementation; each edge looked at only once, and in each phase we potentially need to update the label and choose the minimum label, this gives us O(|E| + |V|^2), |E| can be O(|V|^2), so for a dense graph, you can’t do better than this
* For another implementation; If the |E| is not O(n^2) (we have a not so dense graph), to reduce the complexity: keep a heap of the nodes (minimal-label-node at the top), when we update a label, we do heap sift-up and sift-down if there’s a violation. This is O(|E|log|V|), as for each edge, we might do height-of-the-heap operations.

Question: remind me of the same-weight handling? And the end condition (all edges visited, or smallest weight is the destination?)?

Hw problem: based on the description of Dijkstra, instead of l(i1,i3) = l(i1,i2) + l(i2,i3), we have l(i1,i3) = l(i1,i2)^2 + l(i2,i3)^2; Set 2 to arbitrarily high, the length would be dominated by the longest path (the bottleneck edge); To solve this problem, the only thing that needs to change from Dijkstra is the node weight calculation metric

### TA session

**Activity selection problem**

Follow-up from in class;

Generalized proof for all greedy algorithm: two-step method

Let a1, …, an be the choices, such that f1 <= f2 <= … <= fn (where fn is the finish time of n)

* Greedy choice property: there is an optimal solution B such that it contains the 1st greedy choice, ie, a1
* Optimal substructure property: let B be an optimal solution for S, then B’ = B - {a1} is optimal for S’ = S - {all activities not compatible with a1}

For the first property:

Let B be an optimal solution for S, if a1 is in B, we are done; if a1 is not in B, let ak be the activity in B that has minimal f time, f1 <= fk, sj >= fk (sj: start time of j) for any aj in {B - {ak}}

So, {B-{ak}} U {a1} is a feasible solution with the same size as B

For the second property:

Suppose B’ is not an optimal solution for S’; there exists B’’ is an optimal solution for S’. |B’’| > |B’|, B’’ U {a1} is a solution for S, |B’’ U {a1}| > |B|, contradicts with B being optimal solution for S.

Claim: there exists an optimal solution B for S that contains the first K greedy choices (k >= 1), given those two properties.

If this claim is true, we can say that an algorithm which holds those two properties is correct

Proof by induction on k: base case k=1, directly apply GCP, base case holds

Induction hypothesis: assume claim holds for k=m, for m>=1

Induction step: we want to show claim for k=m+1; let B be an optimal solution for S such that it contains the first m greedy choices; let B’ = B - {the first m greedy choices}, S’ = {all activities of S that are compatible with the first m greedy choices}, by OSP, we have B’ is an optimal solution for S’ (or S - S’?) (OSP only says about one choice, how we extended it to multiple?). The (m+1)th choice would be the first greedy choice for the subproblem S’ (or S - S’), by GCP, B contains the choice m+1, thus the claim holds for m+1

**Example**

Problem: given n integers, a1...an, we want t1..tn ti in [1,n], such that at1, at2…atn (concatenated) is maximized

(for example, numbers 12, 34, 56 should result in the maximum: 563412; 12, 123 case and 12, 121 case)

Make an array a1, a2, …, an such that a1ai >= aia1 for all i (sorted array by a new type of comparison), and this array would result in the maximized concatenation.

Proof of correctness:

Its GCP: there is an optimal solution B=at1...atn such that t1 = 1

Let B be such an optimal solution, if t1=1, then we are done; if t1=/=1, then we have t1=k=/=1, and solution B’ = akat1at2...ati-1ati+1...atn >= B, where ti=1 (reason of B’ >= B is that we can move a1 step by step from B, and each time we result in a concatenation larger than or equal to B); given that B is optimal (B >= B’), we have B’ = B, B’ is also optimal

Its OSP: let B be an optimal solution, B = at1at2...atn such that t1=1, we want to prove B’=at2...atn is an optimal solution for a2...an

Can prove by contradiction: if there’s a better B’’, then there will be an a1B’’ > (or >=?) B, contradicts with B being optimal

**Huffman tree**

Description: given alphabet A={a1, …, an}. frequency f(x), for all x in A, compute a full binary tree T such that the cost sigma(f(x) \* d(x)) is minimized (d: depth)

Proof by the GCP and OSP:

* GCP: we want to prove that “let x and y be the smallest and 2nd smallest frequency characters, then there is an optimal tree in which x and y are siblings” (side note, by being siblings, they are guaranteed to be on the leaf level)

Proof: let T be an optimal tree; Assume x y are not siblings, and a b are deepest leaves of T and a b are siblings. Suppose f(a) <= f(b), we have f(x) <= f(a), f(y) <= f(b); Consider the tree T’ with a and x swapped; T’ - T = f(a) \* d\_T’(a) + f(x) \* d\_T’(x) - f(a) \* d\_T(a) - f(x) \* d\_T(x) = f(a) \* d\_T(x) + f(x) \* d\_T(a) - f(a) \* d\_T(a) - f(x) \* d\_T(x) = (f(a) - f(x))(d\_T(x) - d\_T(a)) <= 0, similarly, based on T’, we have T’’ = T’ with b and y swapped, and we have T’’ <= T’, since T is optimal,T = T’’ which have x y as siblings

* OSP: let A = {...x, y...}, A’ = {...z…}, F = {...f(x), f(y)...}, F’ = {...f(z) = f(x) + f(y)...}. Let T be an optimal tree for (A, F), prove that T’ is optimal for (A’, F’)

Proof (usually by contradiction, if T’ is not optimal for the subproblem, T won’t be optimal for the original): T = sigma(f(c) \* d\_T(c)), c in A - {x,y} + f(x) \* d\_T(x) + f(y) \* d\_T(y) = sigma(f(c) \* d\_T(c)) + f(z) \* d\_T’(z) + f(x) + f(y) = T’ + f(x) + f(y); Since we have T = T’ + Constant, there can’t be a better solution T’’, such that T’’ + Constant yields better result than T (verify this statement?)

Random thoughts on hw3:

Seems that post-order DFS iterative is not that straightforward to implement, should implement it at some point

Lec 7:

**Minimum spanning tree**

Description: undirected G(V, E), w(e) is the weight (called for undirected as opposed to length in directed) of edge e. W(T) = sigma(w(e)), where e are all edges in the tree T. We want to find the min(W(T)).

Definition cut(x,y), where x U y = V, x intersect y = empty set, and the edges (w,q) st w in x, q in y; A partition (cut) has 1-to-1 correspondence with the binary string 2^|V|. (each node either in x, or in y).

Lemma: for any cut(x,y), there exists a MST that contains the min weight edge in the cut. (What’s the conclusion for same-weight edges?) (the smallest edge in every cut must be in the MST: use this logic to justify Prim’s evaluation)  
Proof for lemma: Assume that the minimum edge is not in the cut; let’s put that edge in the tree, there will be at least a cycle (~~the number of cycles that goes through the cut is even?~~). Take any edge in the cut and cycle and cut it, we still have a tree (every node who uses the removed edge to be reachable can use the new edge that we put in), and that tree has a smaller W(T)

Prim’s algorithm: Dijkstra with a different evaluation (cost of a path is the maximum edge in the path?); in other words, think of a cut with one node and the rest of the graph, and the smallest edge from that node must be in the tree. Do this recursively (would there be a a->b b->a different choice case? No)

Complexity: same as Dijkstra

Recursive Prim: the MST from one part and the MST from another, plus the smallest edge that connects the two parts

Under what condition will the edge with the maximum weight be in the MST?

When that edge is a bridge

Kruskal's algorithm (MST)

The edge with minimum weight is in the tree, what about the second smallest one? Should be in; if the third one close the loop, don’t take it, otherwise take it

The O(|V|) optimization?

What problem are we solving again? Seems to be the Union-Find implementation of Kruskal’s algorithm; should double check (Sec 4.6 of book)

Union, find? Joining sets and naming them? Change the name of the smaller one to the bigger one? (Point the root of the smaller one to the root of the bigger one) Same level, just connect one to the other (maintain a set as the rooted tree, set name is the root of tree)

Set will have level l implies that the set contains at least 2^l nodes

Whole work is O(nlog\*n)

Regarding proving a certain tree is MST in hw, I find the top answer here helpful

<http://math.stackexchange.com/questions/352163/show-that-theres-a-minimum-spanning-tree-if-all-edges-have-different-costs>

### Lec 8

(The assumptions, Dijkatra’s non-negative-length edges; in MST, we assume all weights are distinct, thus there’s only one MST)

**Minimum distance between points on a plane (divide and conquer)**

Description: N points in a plane; Pi = (xi, yi); Distance d(pi, pj) = sqrt((xi-xj)^2 + (yi-yj)^2); want to find the pair of nodes that has minimum distance (assume no two pairs with the same distance)

Naive algorithm: O(n^2)

Algorithm: divide and conquer; divide to two groups each with (n/2) points. Solve the problem on both sides, we have d\_left and d\_right; with both sides merged, we want to do something like

T(n) = 2T(n/2) + ?

Ignore points that are outside the strip of (y1-d/2, y1+d/2), where y=y1 is the line we used to divide

One thought: divide on x too, can we argue that the points we are interested in always falls within (x1-d/2, y1-d/2), (x1+d/2, y1+d/2); Seems that we can.

This makes the remaining problem trivial, as the remaining square has at most 6 points in one hemisphere (because the min distance on either hemisphere is d) (still holds if d for x division and d for y division are different); with one step we reduce the problem to O(n) (one traversal to find the points in the square)

So, this approach makes the problem, T(n) = 4T(n/2) + O(n)

The idea: starting from the top most point in the strip, check the points on the opposite side, we have at most 4 to check in “the box” (let the node we choose be (xi, yi), the box is (xi-d/2, yi-d/2) (xi, yi+d/2)); as we go down the sorted points (by y) on one side, we can eliminate nodes on the other side as they fall out of the boxes of the current node on our chosen side. This makes the algorithm

T(n) = 2T(n/2) + O(n), which is O(nlogn); the initial sorting we did is also O(nlogn)

As we divide, to make sure we don’t sort again in the subproblem, whose cost is not in the above notation; we pass in the sorted list to the subproblem. (Here with O(n) figuring out if the original point’s in the left or right would be enough)

So we need a sorting of x, a sorting of y, and then the recursive algorithm

**Huffman coding tree (greedy)**

Side note: consider the job scheduling function again, what about min(sum(Li))

**Job scheduling problem with weights (dynamic programming)**

Description: n jobs with (s\_i, t\_i, w\_i) (start, end, weight); want max(sigma(w\_k)), k in 1..n, where (s\_k, t\_k) is disjoint of any other jobs

Idea: start from a recursion opt that takes the list of jobs.

Without loss of generality, jobs are ordered according to ti (at least O(nlogn)), we have two cases (assume there’s a single best solution)

* Job n is not in the best solution: opt(jobs exclude job n)
* Job n is in the best solution: w\_n + opt(jobs that are not conflicting with n)

Let p(j) = the job i, st t\_i <= s\_j, and t\_i is max among those satisfy; this is O(logn), and doing this for all the jobs (O(nlogn)) is free (since we already sort for O(nlogn))

So,

opt(1) = w(1), ...

opt(n) = max(opt(n-1), w(n) + opt(p(n)))

With this description, we have an exponential solution, O(2^n); verify?

Problem is, we’ll calculate the same opt(k) many times

With DP, going up the tree, the max(...) takes O(1), doing this for n jobs is O(n), the real work’s in the sorting and p(n) for each n, which makes the overall algorithm O(nlogn)

To output the chosen jobs, keep track if larger is w(n) + opt(p(n))

An interpretation for DP, each time we solve a subproblem, we maintain the results for later calculation

Another interpretation for DP, when you face a recursion whose number of subproblems is polynomial, we compute it iteratively from bottom-up (and save the results for each subproblem)

Side note, remind me of the DP for Fibonacci; more specifically, how numbers are “stored”?

Side note, with side effects free and Haskell’s call evaluation (call by name, iirc?), you get DP for free?

### TA session

**Selection problem (divide and conquer) - BFRPT algorithm**

Description: given set S of n reals and an integer k (1<=k<=n), want to find k-th smallest in S

Solution (Theta(nlogn)): sort, and find k-th

Solution (Theta(n)): (one observation, hardest is when k is around n/2: the median);

Select (S, k)

* If |S| <= 5, then compute k-th smallest by brute force (takes 4+3+2 steps at most)
* Divide S into n/5 groups of 5 elements each; and find the median of each group by brute force
* Let M be the set of group medians, by calling select(M, |M|/2), we have the median of the group of medians; let m\* be the median of group of medians
* Let S1 = {elements < m\*}, S2 = {elements > m\*} (note: when finding median, this pivot cuts off 3/10 elements at least)
* If k = |S1| + 1, then return m\*, else if k <= |S1|, return select(S1, k), else return select(S2, k - |S1| - 1)

Analysis: step 0 is const (9), step 1 is 9n/5, step 2 is C(n/5), step 3 is n, step 4 is C(7n/10) (since m\* may not divide the list equally, only approximately a quarter (n/5 (groups) \* ½ (half of the groups satisfy) \* 3 (elements per group)) of the numbers are guaranteed larger or smaller than m\*), where C is the cost of the algorithm

So we have a guess, (with 14n/5 not considered)

C(n) = 9, if n <=5;

C(n/5) + C(7n/10) + 14n/5, if otherwise

Expand recursively, C(n) = n + 9n/10 + (9/10)^2 \* n + … + (9/10)^k \* n <= (sigma(9/10)^i) \* n (i=1..infinity) = 10n

Proof by induction: claim C(n) <= d.n for some const d>=0 to be determined

Base case: n=1...5, C(n) <= 9, choose d = 9, conclusion holds

IH: assume C(n) <= dn for all n < m where m >= 5 (strong induction hypothesis)

Induction case: want to prove C(m+1) <= d(m+1), C(m+1) = C((m+1)/5) + C (7(m+1)/10) + 14(m+1)/5 <= d(m+1)/5 ((m+1)/5 <= m) + 7d(m+1)/10 + 14(m+1)/5 = (9d/10 + 14/5)(m+1), and we want to prove (9d/10 + 14/5)(m+1) > d(m+1), which holds for d >= 28

So we have a linear time solution (constrained by 28n).

Why we choose 5: 5 is the smallest number that works, can prove

(Actually, for a C(n) = p/q(C(n)) + kn, if p < q, this should be O(n), nq/p)

Another to think about: quick select (where average case is linear time), this is similar to quick select plus m\*

### Lec 9

Examples of dynamic programming

**Shortest path - Bellman Ford algorithm (Dynamic programming)**

Description: consider the shortest path without restriction of non-negative weights, but with the restriction of non-negative length cycles (thus, shortest path is the shortest simple path);

(Note: if negative cycles exist in the path from s to e, we consider the shortest path undefined. (So, we are looking at the shortest simple path, the algorithm yields undefined if shortest path is undefined); hw: reasoning about this behavior)  
Idea: d(v) = min(d(w\_i) + l(w\_i,v)), where (w\_i, v) in E;

If there exists a loop, then we could be going in circles; therefore, we insert the parameter d(v,i), which is the distance of shortest path from s to v under the condition of not using path of more than i edges; if we can get d(v,n-1), we have the solution of the problem (essentially, each time we do recursion, we know that the hop count of path reduces by 1)

Thus we have d(v, i) = min( min( (d(w\_j, i-1) + l(w\_j, v)) ), d(v, i-1) )

We have the inside min, so that we don’t enforce having i number of edges in the path (which corresponds with the definition we gave for d(v,i): “not using more than i edges”)

This description as is, is exponential; bottom-up dynamic programming (so, from d(v,0), we calculate all d(v,1), etc) reduces the complexity to

Complexity: O(|V||E|), we do |E| work in |V| rounds, this could be O(|V|^3) at worst

Edge relaxation:

Forget about the hop count i, d(v) = min(d(w\_i) + l(w\_i,v)), where (w\_i, v) in E;

Hw: prove that edge relaxation can work

Negative cycle check:

After |V|-1 steps, If no label changes from d(v,|V|-1) to d(v,|V|), we know that the algorithm concludes (no negative cycles)), since if there are no updates, from d(v,|V|) to d(v,|V|+1) would face the exact same situation of no updates

**Floyd Warshall algorithm (shortest path with node set constraint)**

Description: define d\_k(w,v) as shortest path from w to v where only nodes from the set {1..k} are allowed inside the path

d\_0(w,v) = l(w, v) if (w, v) in E, otherwise infinity

Idea: d\_k(w, v) = min( d\_k-1(w,v), d\_k-1(w,k) + d\_k-1(k,v) ) (node k on the path or not; we have k-1 in the second part as we still have the assumption of no negative cycles)

Complexity: the recursion as-is is exponential; however, the number of subproblems is n^3 at most, as w, v and k are from 1..n. (each time, we produce a n^2 array from a n^2 array, and we do this n times)

We store the results of all computations, time complexity is O(n^3)

It’s better to do Floyd Warshall than to do Bellman Ford for each source, under what scenarios (if you want the multiple combinations anyway?)?

To use Floyd Warshall algorithm algorithm for a generic shortest path algorithm, we start with intermediate set (the constraint) being an empty set, and each iteration we solve the problem with node k added to the intermediate set, we have the decision of going through k vs not.

[Pseudo code implementation with path reconstruction](https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm)

While the original implementation takes d\_k(x,y), consider the implementation without the k (constraint) included in the d\_(x,y) (which is what the wikipedia solution does), this algorithm still works. Reasoning for this?

**Minimum edit distance (Gene difference problem)**

Description: from one word to another, you can delete a letter, insert a letter, or change a letter; Find the edit distance. ( define this min edit distance function as E, E(a,b) = E(b,a) )

Idea:

Consider the E(i,j) being the min edit distance function between w\_1|i, w\_2|j, (word 1 limited to i, word 2 limited to j); E(0,j) = j = E(j,0)

Lemma: think of aligning, the optimal solution should align the x\_m with the y\_n if x\_m = y\_n, where x\_m and y\_n are the last of each word; and the solution would be E(m, n) = E(m-1, n-1)

Otherwise: if x\_m != y\_n, the solution E(m, n) = 1 + min(E(m, n-1), E(m-1, n), E(m-1, n-1)); corresponding with the three operations; this recursion as-is is exponential.

Complexity: however, the number of subproblems (combinations from bottom up) is (max(m,n))^2; (Or just O(m,n)?)

Bottom up: take the m by n table, fill in the first column and row, then fill in the second,etc

With a filled table, to solve (i,j), we need 3 look ups each time;

The table can be thought of as a directed graph, with each edge (i,j) → (l,m) being the way we reach (l,m); this table is a DAG. If we keep only one path the shortest paths, the DAG with direction removed is a tree; The horizontal and vertical edges have length 1, but the diagonal edges have length 0 or 1, the editing cost is the length of the path; This process of looking for shortest path (with different weight for different actions for example) in the table is no different from our DP problem. We also don’t need all the table items for this algorithm, we need only 2 \* min(n,m) space (if we don’t want to trace) (essentially, each time we compute row i from i-1).

If we want to trace, a divide and conquer idea: consider the row n/2, entries of nodes on that row denote the shortest path from r to that entries; the segments of the shortest path is a shortest path (Verify). There must be at least one entry on row n/2 that’s on the shortest path from r to (m,n), and we reduce the tracing problem to two sub problems each with the size of a quarter of the table. And the total time plus tracing would be (mn (find the shortest path) + mn (find the two paths to the node on n/2) + mn/2 + mn/4 + …) = O(mn); this way we don’t increase the space complexity, as compared to the space complexity without tracing

(Similarly for tracing a path in DFS?)

Hw finding: prim’s algorithm does not work on directed graphs with negative weights (does it work on directed graph with non-negative weights?)

Summarize: the pattern for proving a tree is MST

We didn’t define trees with directed edges in the first place, it seems

### Lec 10

Continue on DP

**Knapsack problem**

Description: given n knapsacks each with a volume v\_i and utility u\_i

Maximize Sigma(u\_i), subject to Sigma(v\_i) <= V

Idea: ideally, we want to take the one with highest u\_i/v\_i (greedy), however this solution may not be optimal (does not satisfy greedy choice property, the optimal solution may not contain the greediest pick)

K(i,w): optimal solution for the problem with V=W, and just the first i item

The recursion:

* item i is not in: solve original problem with one less item; K(i,w) = K(i-1,w)
* item i is in: solve problem with one less item, and the constraint of V - v\_i; K(i,w) = K(i-1,V-v\_i) + u\_i

Complexity: this is exponential as-is; however, the number of different subproblems is at most n\*V

So do it bottom up, we’ll build the V by n table (for example, V being the row); to solve (i,w) the position we’ll be looking at will be (i-1,w) and (i-1,w-u\_i), and we can solve the table row by row

Since producing K(i,W) takes constant number of operations, the algorithm’s O(nV)

Up to now the problems we talked about have polynomial-in-the-size-of-input solutions

When we think of it as: we need logV size to represent V, so, for this solution, we are polynomial in the value of the input, but exponential in the bits of the input

For example, in sorting, there’s no such differentiation above, we don’t care about the value of the numbers we are sorting (so, we can prove that sorting is at least O(nlogn))

We call our O(nV) algorithm pseudo polynomial

Hw maximum latency to sort problem, no general solution; his conceptual reasoning for not being able to extract sorting?

### TA session

Dynamic programming

The core concept is an optimal substructure

**Matrix-chain multiplication (MCM)**

Description: Given n matrices, A\_1, A\_2...A\_n of reals; each matrix A\_i is a P\_(i-1) \* P\_i matrix 1 <= i <= n

Goal: compute A\_1 \* A\_2 \* … \* A\_n using the minimal number of scalar multiplications;( if A\_i \* A\_(i+1) takes P\_(i-1) P\_i P\_(i+1) scalar multiplications)

(For example, A1 A2 A3 A4, P0 = 10, P1 = 20, P2=50, P3=1, P4=100, ((A1 A2)(A3 A4)) gets you 65K operations, (A1(A2A3)A4) gets you 2.2K operations; we want the minimal number of operations)

Idea: we need to break the chain multiplication into groups (A\_1...A\_k) \* (A\_(k+1) … A\_n), we have optimal substructure property for each of the part (if a group’s not a substructure, we can find a better solution than the entire array; formally describe)

A\_(ij) = A\_i \* A\_(i+1)... \* A\_j

The cost of A\_(ij) = A\_(ik) + A\_((k+1)j) + P\_(i-1) \* P\_k \* P\_j

Let m\_ij be min(the cost needed to compute A\_ij)

m\_ij = 0 if i = j

m\_ij = min{i<=k<j}{m\_ik + m\_(k+1)j + P\_(i-1) \* P\_k \* P\_j}

Implementation: we can do a top-down recursive expensive implementation

Instead, we do bottom-up iterative dynamic programming implementation

We have an n by n table, and we go up from the diagonal, until we reach the spot m\_1n

In each iteration, we increase (j-i) till it reaches (n-1)

For i = 1..n, m\_ii = 0

For j-i = 1..n-1

For i = 1..n - (j-i)

M\_ij = infinity

For k = i..j-1

Tmp = m\_ik + m\_(k+1)j + P\_(i-1) P\_k P\_j

If tmp < m\_ij

m\_ij = tmp

s\_ij = k

To keep track of the ks we picked, we use s\_ij; to print the solution, we can use the following

Parens(i,j)

If i = j print Ai, return

print(“(”)

parens(i, s\_ij)

parens(s\_ij + 1, j)

print(“)”)

**Longest simple path problem (where optimal substructure does not hold)**

Description: given a graph G, compute for every ordered pair (i,j), compute a *longest* simple path from i to j.

We don’t have optimal substructure, as the path from source S to the intermediate node Q we find may contain nodes in the path from Q to the destination D (give a counterexample)

This problem is solved by brute force

### Lec 11

Sorting: comparison sorting (what we’ve covered so far; answer’s always greater or less)

Alternatively, the program (the execution of the algorithm) being a binary tree, each step a query (node in the tree)’s made, and yes/no answers go down the tree

Sorting takes n(n-1)/2 pairs at most, using transitivity reduces it to NlogN; Verify if this is an effect of transitivity?

Mergesort is smart in this sense; Mergesort results in one such balanced binary tree: (every path is NlogN deep)

Bubble sort results in a skewed tree, where the longest path is n(n-1)/2

**Prove: comparison sorting cannot get better than O(nlogn) (lower bound proof)**

Idea: each different input (arrangement, where the relationship of numbers in positions i and j matters and the absolute value does not; or, in the permutation sense) takes a different path in the tree (Otherwise the inputs won’t be different arrangements)

Consider the number of distinct arrangements (permutations) we can have: n! (n factorial); best way to accommodate the distinct paths would be a balanced binary tree; and the depth of this binary tree is log\_2(n!) > (n/2)logn (proof)

( log\_2(n!) > log\_2( n \* (n-1) \* … \* (n/2) ) > log\_2( (n/2)^(n/2) ) = (n/2)(logn - 1) ) (take the first n/2 element); note that here we want the “lower (>)” bound

Remind me of n = ?n^2

Can we do better if we treat the numbers as numbers (rather than permutation)? If the range is unbounded, cannot do better than nlogn (Ramsey’s theorem related?); but if not, as an example of bounded range:

**Bucket sorting**

Description: given m numbers whose max is smaller than n (for example, range (1,m+1)), return a sorted array.

Idea: insert to corresponding bucket (or increment, if the given numbers are not distinct) in an array of size n, copy back to the return array (use number as pointers)

We have O(n); (though would it make sense to express it with n instead of m)

Extension: what if we may have same keys, but different “content”; we want to sort by key, but maintain the original order in the input if multiple entries share the same key (stable sorting)

Idea: instead of keeping and incrementing the count in the bucket, keep a list of the actual entry (which maintains the order of input) in each bucket

**Radix sort**

Description: given n numbers and the range of 1..n^m, sort

Idea: starting from the least significant bit, we can apply the bucket extension (with same key and different “content” case); this corresponds with a recursion: assume we’ve the result of sorting by k-1 least significant bits, we can sort for k least significant bits.

Complexity would be O(mn)

**Independent set property (King of the Tournament)**

Theorem: given a directed graph G = (V, E); there exists I(G) who is an [independent set](https://en.wikipedia.org/wiki/Independent_set_(graph_theory)) in G(‘s underlying undirected graph) st for any v in V, there exists a w in I(G) st w → v or w → u → v (given any node, I can reach it within two directed hops from a node within the independent set)

Definition - independent set: for an undirected graph G, its independent set is a set of nodes I belong to V, such that for any w, v in I, (w,v) is not in E.

The problem of finding the largest independent set is NP-complete

Example: the tournament’s a complete graph, so the conclusion would be: either I beat you, or I beat someone who beats you

Example: ambulance allocation in an independent set, and ambulances from a node can reach any node in two hops

Proof of theorem by induction:

Base case: graph with one node, it can reach itself

The induction idea: take a node n in G, remove it and nodes that it can reach within one hop, and we have G’

Induction hypothesis: for G’, there exists I(G’) for G’ st for any v in V’, there exists a w in I(G’) st w → v or w → u → v

Induction case: we have I(G’) is an independent set in G. We have two cases:

* I(G’) can reach any nodes in G within two hops, we are done
* I(G’) cannot reach all nodes in G within two hops. Since I(G’) can reach all nodes in G’ in two hops, we have the nodes in G that we cannot reach from I(G’) within 2 hops belong to the nodes we removed. We have two cases:
  + If some of nodes incoming to n is in I(G’), we have the conclusion;
  + If none of the nodes incoming to n is in I(G’), we can build I(G) = I(G’) U n, which satisfies the conclusion

Hw problem: find king of tournament without recursion

When you do induction, the hypothesis can be anything as long as it helps deliver to the next stage

**Longest increasing subsequence**

Description: for a series of distinct numbers i\_1...i\_n, find its largest subsequence i\_{w\_1}...i\_{w\_k} which is increasing.

Induction hypothesis: for i\_1...i\_m, we know for each k = 1...m, let d\_k be the value of the smallest possible last element of all increasing subsequences of length k (d\_k is denoted as infinity if no such element exists for k); we fill an array A of size m such that position j has d\_j;

Lemma: array A is an increasing subsequence (proof by contradiction) till infinity

Induction: for i\_1...i\_{m+1}, find where i\_{m+1} would fit in A such that A\_p < i\_{m+1} < A\_p+1}, replace A\_{p+1} with i\_{m+1} (we’ve something better for position p+1, but not position p+2 since it’s the last element)

When we get to n, the length of A till infinity is the answer

Complexity: O(nlogn), n phases with logn executions to fit i\_{m+1} in each phase.

(Modifications based on the description [here](http://www.geeksforgeeks.org/longest-monotonically-increasing-subsequence-size-n-log-n/) (we are following a slightly different description))

**Geometric algorithm**

**Convex Hull Algorithm**

Convex combination: the plane as defined by sigma\_{i=1}^{n}{alpha\_i \* p\_i}, where 0 <= alpha\_i <= 1 and sigma\_{i=1}^{n}{alpha\_i} = 1, for points p\_1...p\_n. (Any point is superfluous if it can be written as a convex combination of other points.)

(Convex hull: a set of points in the plane, you exclude any points by moving a line to the points until reaching a point in the convex hull) (A convex polygon is a polygon which is a convex hull for all points inside; or, defined as all angles are less than 180) (Check out mathematical desc of these)

Given two lines, we know if they intersect or not by checking if the intersection’s on either line.

Problem description: given a (not necessarily convex) simple polygon which divides the plane into inside and outside. We want to test what point’s inside and what point’s outside

Solution: given a point, draw a ray and see how many edges it crosses; even outside, odd inside

Complexity: O(n) where n is the number of edges, since intersection checking is constant

Problem description: find the convex hull given a set of points

Solution: sort points by x, draw a line between max and min x, the two points are in the boundary of the convex hull; divide the set of points into two sets above and below the line; for all the points in the subset, starting from the min x, draw a line between each two x-adjacent points, if the formed angle’s larger than 180, we backtrack

Complexity: O(n) after the sorting, O(nlogn) with the sorting, where n is the number of points. The later part’s O(n) as even though we do backtrack, since each node once removed is not going to be visited again, and each visit / deciding the angle is constant time

Reducing complexity: the complexity of convex hull algorithm cannot get better than O(nlogn), as given a convex hull algorithm, we can build a sorting algorithm (for array i\_1...i\_n, apply the convex point algorithm for (i\_1, i\_1^2), …, (i\_n, i\_n^2), which is supposed to output a circular list of points, point\_i connects to point\_{i+1})

### TA session

Topic: DP, midterm

**Hw4 Pb2 (MST defined as Dijkstra with max as evaluation criteria)**

1. Given solution does MST => bottleneck smallest holds; combined with bottleneck smallest uniqueness to prove bottleneck smallest => MST. (Remind me if my proof is also in => direction? Appears not)
2. Prove algorithm gives us MST: if path s → u is bottleneck smallest, then for some node i on the path s → u, i → u is also bottleneck smallest; assume that there exists a path (u,v) in the MST that does not hold the property; consider the path (s, w, u) and (s,w, v); without loss of generality, we can have a path (s, w, u, v) which is bottleneck smaller than (s, w, v), contradiction
3. Prove algorithm’s the same as Prim: (complexity wise, DijkstraMST should be log(n) time as expensive as Prim’s algorithm, because we are comparing bottleneck smaller array each time; verify this);

Midterm (one piece cheat sheet)

1. Divide and conquer (may not need to prove correctness)
2. Greedy (give algorithm and prove)
3. DP (write recursion, may not need to prove correctness)
4. General problem (any/combination of 1-3 or graph) (harder)

If not asked to prove correctness, usually trivial and no need to prove

Can draw conclusion from class, if not asked directly about the conclusion itself

**Optimal binary search tree (DP)**

Description: given S={k\_1 < … < k\_n} of reals, is a query key a in S?

Successful search: a in S; unsuccessful search: a not in S;

Let Pi = probability of a = k\_i (1 <= i <= n)

Let Q0 = probability of a < k\_1

Let Qi = probability of k\_i < a < k\_{i+1} (1 <= i < n)

Let Qn = probability of a > k\_n

We have sigma(Pi) + sigma(Qi) = 1

For each leaf node, we add two children, and define the node d\_i as the corresponding gap

Cost of search ending at k\_i (d\_i) is 1 + D\_{T}(k\_i) (or 1 + D\_{T}(d\_i)) (D\_T: depth of node in the tree)

Expectation of cost of search E(T) = sigma\_{i=1..n}( P\_i \* (1 + D\_{T}(k\_i) ) ) + sigma\_{i=0...n}( P\_i \* (1 + D\_{T}(d\_i) ) )

Goal is to find a BST st E(T) is minimized

Idea: with k\_r being the root, on the left subtree, we have k\_1...k\_{r-1} and d\_0...d\_{r-1}; on the right subtree we have k\_{r+1}...k\_n and d\_r...d\_n; we have optimal substructure for this problem (verify);

E(T) = P\_r + E(left sub tree) + E(right sub tree) = P\_r + sigma\_{i=1...r-1}{ P\_i \* (1 + D\_{T}(k\_i)) } + sigma\_{i=0...r-1}{ Q\_i \* (1 + D\_{T}(d\_i)) } + sigma\_{i=r...n}{ P\_i \* (1 + D\_{T}(k\_i)) } + sigma\_{i=r...n}{ Q\_i \* (1 + D\_{T}(d\_i)) } = sigma\_{i=1...n}{P\_i} + sigma\_{i=0...n}{Q\_i} + E(T\_1) + E(T\_2)

Let T\_{ij} be an optimal BST on k\_i...k\_j

Let e\_ij = E(T\_ij), e\_ij = sigma\_{k=i...j}{P\_k} + sigma\_{k=i...j}{Q\_k} + E(left) + E(right); Call the first two sigmas w\_ij; and the recursion is

e\_ij = min{ w\_ij + e\_i{r-1} + e\_{r+1}j } for r in (i, j)

**Midterm-hardness problems**

Description: given n, how many structurally unique BST that store the value 1...n; e.g. n=3, we have 5 unique BSTs;

Solution: DP: the recursion: B(n) = sigma\_{r=1...n}{B(r-1) \* B(n-r)}; B(0) = 1, B(1) = 1, where B(n) is the number of structurally different binary trees for 1..n

Description: Given an unsorted array of ints, find the length of longest increasing subsequence

(Brute force: O(n^3) (go through every subarray; how is brute force n^3)

Solution: L\_j = max{ L\_i + 1 } such that key(i) < key(j), i < j; this is O(n^2); (essentially annotating the 1st, 2nd... number by length of longest increasing subarray of the array till the marked element)

Midterm problems

Interleaving string, my iterative version seems problematic; counter example:

A = “XXY”, string B = “XXZ” and string C = “XXZXXY”

With my removal, gets “ZXX”, and returns false; which is wrong

Remind me, if 4.1 is asking for tour with recurring edge, or no?

The 2 \* MST traversal method I mentioned is actually a DFS

4.3 could do with simple DFS?

### Lec 12

**Problem of median**

Description: find the median value of an array

Idea: pick a random element and go through the array, split by that element; O(n^2) at worst

We want to pick a good pivot to split the array with, expectation-wise, picking twice before deciding the pivot will on average give us an n algorithm, though worst case does not change

O(n) idea: mentioned in **TA session,** [**BFRPT algorithm**](https://en.wikipedia.org/wiki/Median_of_medians)(the key, as Gafni mentioned, is that the pivot gets rid a reasonable amount of nodes; didn’t go into details, and provided a not 100% correct analysis)

**Quicksort**

Take pivot, split (worst case: O(n^2), average is good)

Remind me its implementation

**Max Flow**

Description: ship liquid through a network of pipelines: weighted directed graph with two distinct nodes (source s and destination t); each edge having “capacity” c(e) > 0; how much we can ship from s to t at maximum?

Define the actual flow that goes through e, 0 =< f(e) =< c(e);

Constraint of the flow (conservation equations): whatever comes in must go out; Sigma\_{goes in v}f(e) = Sigma\_{goes out of v}f(e), v being all nodes except s and t

Without loss of generality, suppose no pipes going out of t, or in s; (and there’s no direct edge from s to t)

Problem, in other words, is max( Sigma\_{goes in t}f(e) ) or max( Sigma\_{goes out of s}f(s) )

Idea for a greedy algorithm: do DFS, find first path from s to t, and ship the value of bottleneck on it. Reduce each used edge by that value, and do it again.

This does not guarantee the optimal result. The conceptual mistake is that you can “push back” by not greedily allocating with the bottleneck of one path (no greedy choice property)

Idea: at the beginning, flow = 0; construct a “residual” graph G\_f, while there is an “augmenting” path in G\_f, push flow and update the flow f = f + f\_{G\_u}, and construct a new G\_f, until we cannot push flow any more. (“residual graph” having a push back?)

Algorithm: [Ford Fulkerson](https://en.wikipedia.org/wiki/Ford%E2%80%93Fulkerson_algorithm)

f(e) ← 0 for all e

G\_f ← G

Construct the residual graph G\_f, where exists an st path in G\_f (the path can be chosen by BFS with minimal number of edges), add the path to G\_f and construct a new G\_f, until there are no such st path (if we use an integer unit, the algorithm will terminate as each time the flow grows by at least 1: integral theorem)

Residual graph construction: adds a reverse path with bottleneck weight for each edge in the (s, t) path that we’ve taken, and update the edges in the path by reducing weight with bottleneck weight; (Conceptually, at all times in the residual graph, (u, v) + (v, u) = capacity(u, v), and weight(u, v) means remaining, weight(v, u) means taken)

Why the algorithm calculates maxflow: there are no s, t path in the final residual graph; take all reachable nodes from s in the final residual graph, call them A, call the rest B; and all the edges from A to B are saturated, and all edges from B to A are at 0 flow (otherwise there will be edge from A to B in the residual graph); so when we get stuck, we identified a cut in which flows goes from only A to B, and all such edges are saturated, so the value of the flow equals the cut of (A, B)

Complexity: pseudo polynomial, depends on the value of the volume; for this problem, we have polynomial algorithms, too, scaling algorithm: the idea is that we don’t mingle large capacity edges with small capacity edges (do maxflow for only edges within the range (maxEdge / 2, maxEdge) ), and we’ll have log(maxEdge) number of iterations.

**Min s, t cut**

Description: a partition of the nodes of a graph to two parts A and B. s in A, t in B. cut(A, B) is a set of directed edges from A to B; C(A, B) = Sigma\_{e in cut(A, B)}{l(e)}; (C: capacity)

Its relationship with maxflow is maxflow <= min cut (you can’t do better than the capacity of the cut, proof with the Conservation Equations); this relationship is called weak duality.

Define flow over the cut(A, B) = Sigma\_{e.tail=A, e.head=B} f(e) - Sigma\_{e.head=A, e.tail=B} f(e) (those going out - those going in)

Proof of weak duality: sigma of the flow outgoing from s ( Sigma\_{e outgoing from s} f(e) ) = Sigma\_{e outgoing from s} f(e) + Sigma\_{v in A, v =\= s} (Sigma\_{e outgoing from v} f(e) - Sigma\_{e incoming to v} f(e)) (flows cancel each other, on one end it’s incoming, other end it’s outgoing) = Sigma\_{e.tail=A, e.head=B} f(e) - Sigma\_{e.head=A, e.tail=B} f(e) <= C(A, B) (since the first part is smaller than or equal to C(A, B), and the latter part is positive)

Applications of MaxFlow:

**Max matching in a bipartite graph**

Description: given a graph G = (V,E), a matching M in G is a set of pairwise non-adjacent edges; that is, no two edges share a common vertex. We want to maximize the number of edges in a matching. (the one-one marriage analogy) This is hard for any graph, we consider it for a bipartite graph (the analogy of heterosexual marriage only).

Idea: transform the bipartite graph by adding a vertex s that connects to all the nodes in one part of the bipartite graph, and another node t that connects to all the nodes in the other part. capacity(s, u) = 1, capacity(v, t) = 1, capacity(u, v) = infinity;

Extension: for each matching, there’s a corresponding flow (integral theorem, you can’t allocate half-half, two edges cannot share a node by doing a half-half weight)

**Menger theorem**

Origin: arises from network reliability (telephone call routing, for example): what is the minimum number of edges that you can take away to break connectivity between s and t

And a similar problem: how many calls can I route from s to t, given that calls cannot share the same wire (number of edge disjoint paths connecting s and t)

Description: the numbers for the above two problems are equal (max flow = max number of edge disjoint paths)

Idea: every collection of k edge disjoint paths, on each path can route 1; we can find the max flow, which is min cut, which is the number of edges we can take away to disconnect s and t

**Frobenius Hall Theorem (Perfect Matching)**

Definition: neighborhood N(S): given a node set S belonging to V, denote the set of one-hop neighbors of S as N(S),

Description (online): Given a bipartite graph G = (V, E), V partitioned into (L, R), there exists a matching of size |L| if and only if for every node set S belonging to L, |S| <= |N(S)|

( If all nodes in S and N(S) have degree k, we have |S| < |N(S)|, (and a perfect matching can be found, verify?); (N(S) could have other incoming edges whose heads are not in S) )

Description (in class): Given a bipartite graph G = (L, R, E), |L| = |R| = n, there exists a perfect matching iff for all S belonging to L, |N(S)| >= |S|

Proof (this is just FYI): assume that for all S belonging to L, |N(S)| >= |S| holds but there are no perfect matchings. (We find the perfect matching by adding s, t nodes and doing MaxFlow)

We’ve no perfect matching, so by MaxFlow, given an A, B cut, C(A, B) <= |L|; let L\_A = L intersect A, L\_B = L intersect B, similarly R\_A, R\_B; we have C(A, B) = L\_B + R\_A

(since there cannot be an edge in the original graph crossing the cut, as those edges are infinity; so, only edges in the cut are the ones from s to B or the ones from A to t, where s is in A, t is in B)

So, the capacity of the cut equals with the sum of the two intersections

We have N(L\_A) is a subset of R\_A (the nodes on the right side reachable from A are a subset of nodes in intersection, as if you can reach the L\_A nodes, you can reach the R\_A nodes as L to R edges have infinite capacity (essentially, you have to put them in A as otherwise there’ll be infinity crossing the A-B cut)), so we have |N(L\_A)| <= |R\_A| = C(A, B) - |L\_B| < |L| - |L\_B| = |L\_A|; so we have |L\_A| > |N(L\_A)|, which contradicts with the assumption.

(Should prove the other way round, too, verify?)

Found an interesting set of graph theory problems [here](http://www.math.cmu.edu/~lohp/docs/math/mop2009/graph-theory-more.pdf)

**Circulation problem**

Description: suppose we’ve a network of pipes, insert some flows into the network at node s (supply), and take the output from the network at node t (demand); value of supply = value of demand, want to know if we can ship d amount of flow from s to t (if the network can accommodate / if there’s a circulation);

Idea: this is a direct application of MaxFlow

Extension: assume that we’ve two demand nodes t1 and t2, and want to know if we can ship d1+d2 amount to t1 and t2 correspondingly;

Idea: add another super node t connected to both d1 and d2, capacities being d1 and d2 correspondingly, do MaxFlow

Extension2: having multiple supplies s1 and s2 and multiple demands

Idea: add another super node s connected to both s1 and s2...similar with multiple demands.

Extension3: some pipes have a minimum capacity in addition to the max capacity (min, max), want to solve circulation

Idea: reduce to the general circulation problem, by replacing the edge e having min\_e capacity with an edge e’={m, n} having capacity (max\_e - min\_e), and putting a supply of e\_min at m, and a demand of e\_min at n

### TA session

3. 2) may can or cannot reduce to LCS, should think more about this (Verified cannot)

**MST**

We have Prim, Kruskal, Sollin algorithms

A generalized algorithm: let A be a subset of edges of an MST of graph G,(Prim, Kruskal and Sollin are all derived from this generic algorithm)

def an edge e not in A is safe for A if AU{e} is also a subset of edges of an MST of G

We have the algorithm:

Generic\_MST(G):

Input: a connected, undirected, weighted graph G = (V, E, w)

Output: G’s MST

A ← empty

While (|A| < |V| - 1)

Find an edge e not in A that is safe for A, A ← A U {e}

Return (T={V, A})

(Loop invariant: A belongs to some MST of G)

The following part’s covered in class as well

Def: a cut in G is any partition of V into set S and {V - S}, ( cut = (S, V - S) )

Def: a cut respect A if for each edge in A , *both* *endpoints are* in S or in (V - S) (A doesn’t have the full set)

Def: any edge of G with one endpoint in S and one endpoint in (V - S) is a crossing edge

The smallest weight crossing edge is called a light edge crossing the cut

Key theorem: let G be a connected undirected weighted graph, let A be any subset of edges such that A is contained in some MST of G, let a cut (S, V - S) be any cut respecting A, let e = (u, v) be a light edge crossing the cut; then e is safe for A. (i.e. A U {e} is in some MST of G)

(Essentially, the same description as in class)

Proof: let T be an MST of G, suppose T containing A,

* if e is in T then done;
* if e is not in T, then T U {e} contains a unique cycle C, walk from u to v using edges of C - {e}; since e is a crossing, u and v are in opposite sides of the cut; so there exists an edge e’ in this walk that is a crossing; so the graph T’=(T - e’) U e is a spanning tree and the w(T’) <= w(T), which contradicts with T being MST; note that both e and e’ are not in A since the cut respects A, so A is a subset of T’, e is also a member of T’, so A U e is a subset of T’; so e is safe for A

For Prim algorithm, consider it from the generalized algorithm: each time the crossing is the (the nodes you’ve included) vs (the nodes you’ve not included)

**Amortized analysis**

Usually, we have worst case, average case, randomized case analysis

Amortized analysis is a type of worst case analysis (essentially, do worst case on a series of operations and analyze the average, rather than worst case analyze one and times the number of operations)

**Example of amortized analysis with a stack**

Description: consider a stack S, we have operations push(S, x) and mpop(S, k): pop min{|S|, k} amount of elements out of S; push costs 1, and pop costs min(|S|, k)

Question: what’s the total time to execute a sequence of n push and mpop ops in total on an initially empty stack?

# of individual pops is <= # of pushes <= n

So the range of total number of ops is from 0 to 2n

So the amortized cost per op is 2n/n, which is 2.

In the normal worst case analysis, the cost per op is O(n) (because of the possibility of a mpop with near n elements)

**Binary counter (inf): Increment(A)**

Worst case of cost of increment(A) is O(n), where n is the number of bits (find the first “0” bit to flip)

If we have a series of increments, the total cost is n + (n/2) + (n/4) … for the series of increments; which makes the average constant time rather than linear time

If there exists an amortized analysis which yields better results than the ordinary worst case analysis, you should likely use it.

**Proof of amortized analysis**

* Aggregate method
* Accounting method (credits method)
* Potential method

**Accounting method**

The Stack (S) example;

Invariant: at all time during the execution of the sequence Delta, every item on S has 1 credit on it; (Prove that we’ll never run into deficit? What does that mean?)

Base case: this invariant is true initially since S is empty to begin with

Induction hypothesis: assume it holds before the i-th operation of Delta is executed

Induction case:

* i-th operation is a push, use 1 credit to do the push, and store other credits on x; clearly invariant holds after the i-th operation
* I-th operation is a mpop, by invariant, every item of S has a credit; for each of the pop, we’ll use the credit on it to pay for it, and again the invariant holds.

So, amortized credit cost wise, push is 2 and mpop is 0.

So, any sequence can be executed on an initially empty stack without running a deficit, and the total cost is < 2n

**Splay tree** (a different BST from red-black tree) featuring amortized analysis

### Lec 13

(Continuation on MaxFlow, notes followed that of last time’s)

### Lec 14

(Continuation on MaxFlow, starts NP completeness)

**Decision problem** (As opposed to optimization problem)

Description: given a language X = {word1, word2, ...} answer if a word is in the language or not For example, given a language of primes, answer if a number is prime or not

An algorithm A(S) solves X iff it outputs yes if S is in X, otherwise outputs no

Another example, X = {MST decision problem}, where each element is encoded as (G, k), the algorithm S outputs yes if sum(E) < k;

Use S to build an MST (optimization) algorithm: give some k, if S(G, k), test S(G, k/2); otherwise S(G, 2k)

Back to the original description, A(S) is polynomial in |S|

Polynomial is a subset of Nondeterministic Polynomial, which is a subset of exponential time

P = NP? is unproved

In exponential time of the size of the input, we can solve NP; There exists a C, such that for all s, there exists t such that C(S, t) is yes iff s is in X, (|t| is polynomial in |S|) (for example, the Hamiltonian cycle problem? Essentially exhausting the results?) (Clarify X?)

**Language of SAT (Satisfiability)**

Conjunctive normal form: forms like (x\_1 or x\_2) and (x\_3 or x\_4) and ...

Given a CNF, consider the x\_1..x\_n being boolean, we have 2^n assignments, for each of the assignment, the form evaluates to true or false.

Satisfiable: there exists an assignment that the CNF evaluates to true

Checking if a form is satisfiable is NP-complete

The language is the set of assignments that can make the form evaluate to true; the decision problem of checking an assignment is exponential

(How we introduced NP-completeness with this notion?)

### TA session

**Midterm DP interleaving problem**

3.2

L\_ij =

false, if a\_i not equal to c\_i+j and b\_j not equal to c\_i+j;

l\_(i-1)j, if a\_i = c\_i+j not equal to b\_j;

l\_i(j-1), if b\_j = c\_i+j not equal to a\_i;

l\_i(j-1) or l\_(i-1)j, if a\_i = b\_j = c\_i+j

3.3

LPS(A) = LCS(A, rev(A)) in length, but not necessarily in sequence:

Counter example: abcdebca acbedcba; the algorithm finds abdca, which has correct length but not correct sequence (abcba)

Proof: first, we have LCS(A, rev(A)) >= LPS(A); (if it’s LPS, it has to be in LCS)

Then, we want to show LCS(A, rev(A)) <= LPS(A)

Consider the two strings A and A’, find their LCS as L (or LPS, verify?), center of L being k, find the k in A and A’ , let them be i and j; i and j separates A and A’ to A\_R, A\_L, A’\_R, A’\_L;

Case 1: if A\_R belongs to rev(A’\_L), then A’\_R belongs to rev(A\_L); L\_R = LCS(A\_R, A’\_R) <= LCS(rev(A’\_L), rev(A\_L)); Rev(L\_R) <= LCS(A’\_L, A\_L); the string Rev(L\_R) k L\_R, it’s a PS of A, LCS = Rev(L\_R) k L\_R <= LPS (Verify the proof)

To get the correct sequence, you can do LCS(A, rev(A)), take the 1st half and reverse to replace the 2nd half

**Max flow extension: Circulation with demands**

Description: same problem as discussed in class; (Goal is to find a flow f such that

* Capacity constraint: any e, 0 <= f(e) <= C(e)
* Demand constraint: any v, f^in(v) - f^out(v) = d(v), where d(v) is the demand at v
* Let S = {set of supply}, T = {set of demand}; Sigma\_{s in S} (-d\_s) = Sigma\_{t in T} d\_t (sources being negative demand)

(Can sources have incoming edges? They should be able to)

Solution: as discussed in class, add extra node connecting to all sources; add extra node connecting to all destinations; newly added edges have the capacity of abs(d\_s) (or abs(d\_t)); solve the MaxFlow between the two newly added edges;

There exists a circulation iff there exists a flow of value Sigma\_{t in T} d\_t.

Extension: some of the edges have a lower bound

* Extension to capacity constraint: for some edges e, we’ve l(e) <= f(e) <= C(e)

Solution: as discussed in class, reduce it to the unextended circulation;

f\_0(e) = l(e); Def L\_v = f\_0^in(v) - f\_0^out(v), then we have new graph with capacity constraint: 0 <= f(e) <= C(e) - l(e); and new demand constraints: f^in(v) - f^out(v) = d\_v - L\_v (these specific definitions do not seem to help, best just refer to notes / idea in class)

### Lec 15

Class of problems in NP: problem that if the answer (to decision problem) is yes can be proven in polynomial time. (The proving Hamilton circle is polynomial example?)

Proving NP completeness: a problem is NP complete if an NP complete problem can be “polynomially reduced” to it;

Polynomially reduced: the stuff we do in SAT => 3-SAT problem is a polynomial reduction

**Between SAT and Ham: reducing a problem to show NP completeness**

Given a SAT algorithm, can build a hamiltonian circle algorithm that given a graph, after polynomial time transformation, pass to SAT, and SAT can show if a hamiltonian circle exists or not. (Ham is reducible to SAT)

This suggests that Ham is polynomially easier than SAT (polynomial + SAT solves Ham)

(The problem reduced to (in the inner rectangle) is the polynomially harder one)

Given a problem S that is NP, if an NP complete problem T (for example, SAT) is polynomially easier than S, than S is NP complete.

(This is contrary to the naive thought of reducing the S to T; to show that a problem is NP complete, you show that an NP complete problem can be reduced to it; this logic could be tested in final)

**3-SAT**

Description: every clause in SAT has exactly 3 literals

The 1-SAT and 2-SAT problems are polynomial

We want to prove that 3-SAT is NP complete (by showing that SAT can be reduced to 3-SAT: using polynomial time input processing + 3-SAT to solve SAT)

Idea: to turn a single phrase into phrases of 3: add an element y that’s independent from the x\_1 to x\_n;

For example, converting A = (x\_1 or x\_2 or x\_3 or x\_4) to B = (x\_1 or x\_2 or y) and (x\_3 or x\_4 or ¬y): we want to show that A is satisfiable => B is satisfiable; and B is satisfiable => A is satisfiable (the latter is equivalent with A is not satisfiable => B is not satisfiable, contrapositive)

If we’ve two elements (x\_1 or x\_2) converts to (x\_1 or x\_2 or y) and (x\_1 or x\_2 or ¬y)

If we’ve one element, (x\_1) converts to (x\_1 or y) and (x\_1 or ¬y)

If we’ve m, m > 3 elements, we split into two groups of (m - 2) and 2; each time the size becomes strictly smaller, and the number of reductions we do is linear to the size of the clause;

With the above, we’ve a polynomial addon that can reduce SAT to 3-SAT, and 3-SAT is NP complete.

Alternative idea (TA session): given a CNF of literals x\_i, build a tree where parent nodes are relationships (negate, implies, and, or) between its children; denote each parent node as y\_i, the final expression is equal with ( y\_top and (y\_top ⇔ (how y\_top is built from its children)) and (y\_top’s children ⇔ how y\_top’s children’s built from its children)...)

For each (y ⇔ (how y is built from its children)), we can use at most three literals (y and its two children at most) expressed in CNF to represent its truth table. Thus the original expression can be broken down into CNFs of 3 literals (I find this explanation to be more vague than the one in class, should verify)

**Independent set**

Description: given undirected graph G such that I belongs to V and nodes in I are not adjacent on the same edge (independent set definition); Want to find a maximum independent set.

The decision version of the problem is given a (G, k), want to know if there is an |I| >= k.

Idea: we need to first show that the problem is in NP (by showing that given a set, we can check if it’s an independent set and if its size is >= k in polynomial time);

Then we show how 3-SAT can be reduced to independent set problem. We construct a G by associating each clause with a triangle (each literal with a node in the triangle), put an edge between each literal and its inverse; we want to show that “3-SAT true => G has independent set >= k and the other way round.”

“3-SAT true => G has independent set >= k (actually, n)”: since 3-SAT is true, if we choose one literal that’s evaluated to true in each phrase, and include the corresponding node in G in the set, we have an independent set >= n (n >= k always holds)

“G has independent set >= k (actually, n) => 3-SAT true”: the largest possible independent set in G is of size n (can never pick two nodes in one triangle), so the independent set is of size n, and there’s one corresponding assignment that satisfies 3-SAT

Thus independent set is NP complete

**Vertex cover**

Description: given an undirected graph G, a Vertex Cover VC belonging to V is if for all e=(v, w), either v is in VC or w is in VC or both. We want to find a minimum VC; (decision problem: whether exists an (G, k) such that |VC| <= k). Prove that this problem is NP-complete

Idea: first show problem is NP (Given a set, can check with polynomial)

Then show independent set can be reduced to vertex cover.

We do it by passing the same G, and |V| - k to the VC problem, and show both directions.

“Independent set >= k => vertex cover <= n - k (n = |V|)”: complement of an independent set is a vertex cover (for an edge to not be covered, both nodes have to be in the independent set, contradicts).

“vertex cover <= n - k => Independent set >= k”: complement of a vertex cover is an independent set (if it’s not an independent set, two nodes of the same edge have to be in the set; those two nodes have to be not in the complement (which is the VC), and their edge won’t be covered by the vertex cover, contradicts)

Conclusion: vertex cover is NP complete

Alternative idea (TA session, just mentioned): reduce clique to it

**Set cover**

Description: a collection of sets Gamma = s\_1...s\_n belonging to U, C\_Gamma is a cover of U if for all s\_i, there exists u in C\_Gamma, (if) u in s\_i. Find the min|C\_Gamma| is NP complete. (decision problem: |C\_Gamma| <= k)

Idea: it’s NP and we can reduce the vertex cover to this problem. The reduction is to consider each edge being the set of two elements

### Lec 16

More examples on proving NP completeness

**Clique**

Description: given an undirected graph G = (V, E), a subset C of V is a clique in G if for all v, w in C, there exists e = (v, w) in E; want to find the maximum clique. (Informally, find the largest complete subgraph) The decision problem: given (G, k), if there exists Clique C in G, |C| > k

Idea: Obvious it’s in NP, as given a set, the decision is polynomial. Key is finding the NP complete problem that gets reduced to it.

Reduce Independent Set to Clique; the relationship between Independent Set’s input (G, k) and Clique’s input (G’, k’): G’ = ¬G (where we don’t have an edge before, we have edge now), k’ = k

Alternative idea (TA session): reduce 3-SAT to clique. For each clause (l\_1, l\_2, l\_3) of 3-SAT, we place three nodes (v\_1, v\_2, v\_3) in V; add edge between (v\_i, v\_j) if v\_i and v\_j are corresponding with literals in different clauses but the corresponding literals are consistent (which means, v\_i’s corresponding literal is not a negation of v\_j’s (both can be true at the same time)).

If 3-SAT is satisfiable, then at least 1 literal in each clause is true, then we can take these k vertices from the k clauses which creates a complete graph.

If the graph has clique of size k, since we cannot take two nodes from the same clause, and we have k clauses, we have to take one node from each clause; assign true to the nodes we take, and the 3-SAT expression evaluates to true.

**Dominating Set**

Description: a node dominates its neighbors; given an undirected graph G, find a dominating set S, such that for any node in G, either it’s in S, or it’s dominated by a node in S. Want to find the minimum dominating set (similar as vertex cover, except considering all the vertices; difference, for example, in a triangle)

Idea: obviously NP;

Reduce Vertex Cover (input (G, k)) to it (input (G’, k’)). For each edge e = (v, w) in G, add a node x, and edge (v, x), (w, x) in G’;

Want to prove this reduction works: specifically, |VC| >= k => |DS| >= k and |DS| >= k => |VC| >= k

First part’s obvious (VC is DS); second part: given a DS consisting of some newly added nodes, we can swap the newly added node in DS with a node it connects to in the original graph, and won’t be covering less nodes (or edges? verify)

(Side note, there seems to be limited patterns in node, edge transformation)

**Directed Hamiltonian Cycle**

Description: given a directed graph G, does there exist a simple cycle that goes through all the nodes

Idea: it obviously is in NP, as we can check if it’s a Ham cycle in polynomial time.

To show it’s NP complete, we reduce 3-SAT to it: if there’s Hamilton cycle, 3-SAT outputs yes;

We construct the graph by having each expression x\_i in 3-SAT represent a line, if the nodes on the line are traversed from left to right true; otherwise false; to connect the lines and form the graph, we put in a node for each x\_i, such that if x\_i evaluates to true, this node can be included in the Ham cycle passing through x\_i; for each pair of (x\_i, x\_j), add this node connected to two nodes in (x\_i, x\_j): assuming that x\_i has nodes n\_1, n\_2, …, this new node can first connect to n\_1 and n\_2; then skipping n\_3, the next new node can connect to n\_4 and n\_5…

This construction is the proof… (Verify the construction process)

**Undirected Hamiltonian Cycle**

Prove NP completeness: obviously NP. Want to reduce directed Hamilton Cycle to it. Just removing the directions wouldn’t work, as there may be a cycle in undirected but not the directed version; Instead, given the input G which is a directed graph, we split each node into 3 nodes, one connecting incoming, one connecting outgoing, and one connecting the two, and then remove the directions, to form the G’; this way all solutions for undirected Ham cycle in G’ will be solution for the corresponding directed G

**Graph coloring**

Description: the coloring of G is an assignment of colors to nodes such that no neighbors have the same color; want to have as few colors as possible to cover G: given a G and k colors, can the graph be colored? The 3-color problem is NP-complete (n-color is also NP-complete as 3-color can be reduced to it)

Idea: obvious in NP;

Reduce 3-SAT to it; each triangle (T, F, B) corresponds with an expression; for example, given a clause (x1, x2, x3), we build the structure with edges: (B, x1), (B, x2), (B, x3), (x1, x11), (x11, x12), (x2, x21), (x21, x22), (x3, x31), (x31, x32), (T, x12), (x12, x22), (x22, x32), (x32, F), (T, x11), (T, x21), (T, x31), (for the expression ¬xi, connect B to the node representing ¬xi)

So that

* if all expressions are false, we cannot three color the graph (x11, x21, x31) has to be (B, B, B), (x32) has to be T, (x22) has to be F, (x12) has to be T, which connects also with T and violates three coloring
* If one expression is true, we can three color the graph described above

(Seems that we connect all the clauses together by having them all share the B node: verify the complete equivalence between the two problems)

**Subset sum problem**

Find if such a subset exists such that the sum of their elements equals with a given M

**Max three dimensional matching problem**

The max matching problem but each matching consists of three disjoint nodes; two - polynomial, three - NP complete

### TA session P, NP, NP complete

Polynomial time (p-time): O(n^k) where n is the input size and k is a const

Alphabet Sigma = finite set of symbols

Language L over Sigma is any subset of strings in Sigma\*

For example, let Sigma = {0, 1}; Sigma\* could be {0, 1, 10, 11, ...}

L = {10, 11, 101, 111, 1011, ...} is language of primes;

A decision problem has a yes/no answer.

Any decision problem Q can be viewed as language : L = {x ← {0, 1}\* : Q(x) = 1};

Q decides L: every string in L accepted by Q ( Q(x) =1 ); every string not in L is rejected

For example, a decision problem of whether there exists a path:

The language Path = {given <G, u, v, k>: G = (V, E) is an undirected graph, u, v in V, k >= 0 is an integer, and find whether there exists a path from u to v in G with <= edges}

And the corresponding optimization problem is the shortest path between u and v

The problem P = {for L belonging to {0, 1}\*, there exists an algorithm A that decides L in p-time}

Algorithm A verifies language L if L = {for x in {0, 1}\* : there exists y in (0, 1)\* such that A(x, y) = 1} (essentially, given a specific solution, algorithm A tests if the solution satisfies)

NP: Let A be a p-time algorithm, and k is a const; NP = {L in {0, 1}\*: there exists a certificate y, |y| = O(|x|^k), and an algorithm A st A(x, y) = 1} (informally, a problem is NP if it can be verified in P; P belongs to NP, whether P = NP is undecided)

NP complete problems are the “hardest part” in NP: if any NP complete problem is p-time solvable, then all NP complete problem is p-time solvable

Reductions (reduce language L\_1 to L\_2 via function f):

* Convert input to L\_1 to instance f(x) of L\_2
* Apply decision algorithm for L\_2 to f(x)

Then L\_1 <= L\_2 (L\_1 can be reduced to L\_2, L\_1’s not necessarily easier than L\_2, your conversion could be exponential)

If we want to show L\_1 is easier, you want to reduce L\_1 to some easy problems; if we want to show L\_1 is hard, you want to reduce a hard problem to L\_1 (Given the definition of reduction, the direction matters; think about the reduction direction and its conceptual meaning again).

P-time reductions (L\_1 is p-time reducible to L\_2)

If there exists a p-time computable function f: {0, 1}\* → {0, 1}\* such that for all x in {0, 1}\*, x in L\_1 iff f(x) in L\_2

A language L belonging to {0, 1}\* is NP complete if

* L is in NP
* L’ <=\_p L for every L’ in NP (so it’s hardest in the sense that everything can be reduced to it)

If only the 2nd condition is satisfied, it’s called NP-hard

So, relationship wise: P (solvable in P) is in NP (verifiable in NP), NP intersect NP-hard (may not be in NP and some NP-complete problem can reduce to it) is NP-complete (in NP and some NP-complete problem can reduce to it)

Lemma if L is a language st L’ <=\_p L where L’ is in NPC, then L is NP-hard, and if L is in NP, then L is in NPC. (so to prove NP-completeness, you no longer need to reduce from every problem in P, but instead only reduce from a known NP-complete problem)

So, to prove NPC:

1. Prove L in NP
2. Select L’ in NPC
3. Describe algorithm to compute f mapping every input X to L’ to input f(x) of L
4. Prove that f satisfies x in L’ iff f(x) in L for all x in {0, 1}\*
5. Prove computing f takes p-time

To probe L is NPH, remove step 1; 4 is usually the work, 5 is usually trivial

**SAT problem**

SAT = { <Phi> : Phi is a satisfiable boolean formula }

Theorem: SAT is in NPC (proof skipped)

(Theorem: if any NPC problem is p-time solvable, then P = NP)

For the final, know how to do reduction would be enough

Every problem can be encoded into a Turing machine, which can be encoded into 0 and 1s (Turing completeness)

Check and verify hw scores