**How class is structured/How to Study**

*Knowledge Check Quiz UNGRADED/unlimited attempts*

*Practice Quizzes UNGRADED/unlimited attempts use this to prepare for the graded quizzes*

Graded Homework Quiz 1 attempt lasting 2 hours Do not submit homework, but can use it to do quiz

Graded Quiz 1 attempt lasting 2 hours Solutions provided after due date

Midterm 1: Wednesday 2/26/2020 at 9am-10:15am, Room: CDN 60

Midterm 2: Wednesday 3/25/2020 at 9:00am-10:15am, Room: CDN 60

Final exam: Friday 5/1/2020 at 3:05pm-4:20pm, Room: CDN 60

**Unit 0: Prior Knowledge Needed**

**Asymptotic Notation**

Definition of big-O: f(n) є big-O g(n) iff ∃ constants a > 0 such that 0 ≤ f(n) ≤ a \* g(n) ⩝n ≥ n0

Definition of big-Θ: f(n) є big-Θ g(n) iff ∃ constants a1, a2 > 0 such that 0 ≤ a1 \* g(n) ≤ f(n) ≤ a2 \* g(n) ⩝n ≥ n0

Definition of big-Ω: f(n) є big- Ω g(n) iff ∃ constants a > 0 such that 0 ≤ a \* g(n) ≤ f(n), ⩝n ≥ n0

Reflexivity f is O(f)

Constants if f is O(g) and c > 0, then cf is O(g)

Products if f1 is O(g1) and f2 is O(g2), then f1f2 is O(g1g2)

Sums if f1 is O(g1) and f2 is O(g2), then f1 + f2 is O(max{g1, g2})

Transitivity if f is O(g) and g is O(h), then f is O(h)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| NAME | COMPLEX | RUNNING TIME | EXAMPLES |  |
| constant time |  | *O*(1) | 10 | Finding the median value in a sorted [array](https://en.wikipedia.org/wiki/Array_data_structure) of numbers  Calculating (−1)*n* |
| [inverse Ackermann](https://en.wikipedia.org/wiki/Inverse_Ackermann_function) time |  | *O*(*α*(*n*)) |  | [Amortized time](https://en.wikipedia.org/wiki/Amortized_time) per operation using a [disjoint set](https://en.wikipedia.org/wiki/Disjoint_set_data_structure) |
| [iterated logarithmic](https://en.wikipedia.org/wiki/Iterated_logarithm) time |  | *O*([log\*](https://en.wikipedia.org/wiki/Iterated_logarithm) *n*) |  | [Distributed coloring of cycles](https://en.wikipedia.org/wiki/Cole-Vishkin_algorithm) |
| log-logarithmic |  | *O*(log log *n*) |  | Amortized time per operation using a bounded [priority queue](https://en.wikipedia.org/wiki/Priority_queue)[[2]](https://en.wikipedia.org/wiki/Time_complexity#cite_note-2) |
| logarithmic time | [DLOGTIME](https://en.wikipedia.org/wiki/DLOGTIME) | *O*(log *n*) | log *n*, log(*n*2) | [Binary search](https://en.wikipedia.org/wiki/Binary_search) |
| polylogarithmic time |  | poly(log *n*) | (log *n*)2 |  |
| fractional power |  | *O*(*n*c) where 0 < c < 1 | *n*1/2, *n*2/3 | Searching in a [kd-tree](https://en.wikipedia.org/wiki/Kd-tree) |
| linear time |  | *O*(*n*) | *n*, *2n + 5* | Finding the smallest or largest item in an unsorted [array](https://en.wikipedia.org/wiki/Array_data_structure), [Kadane's algorithm](https://en.wikipedia.org/wiki/Kadane%27s_algorithm) |
| "n log-star n" time |  | *O*(*n* [log\*](https://en.wikipedia.org/wiki/Iterated_logarithm) *n*) |  | [Seidel](https://en.wikipedia.org/wiki/Raimund_Seidel)'s [polygon triangulation](https://en.wikipedia.org/wiki/Polygon_triangulation) algorithm. |
| linearithmic time |  | *O*(*n* log *n*) | *n* log *n*, log *n*! | Fastest possible [comparison sort](https://en.wikipedia.org/wiki/Comparison_sort); [Fast Fourier transform](https://en.wikipedia.org/wiki/Fast_Fourier_transform). |
| quasilinear time |  | *n* poly(log *n*) |  |  |
| quadratic time |  | *O*(*n*2) | *n*2 | [Bubble sort](https://en.wikipedia.org/wiki/Bubble_sort); [Insertion sort](https://en.wikipedia.org/wiki/Insertion_sort); [Direct convolution](https://en.wikipedia.org/wiki/Convolution_theorem); Closest pair of points on a plane; Stable match |
| cubic time |  | *O*(*n*3) | *n*3 | Naive multiplication of two *n*×*n* matrices. Calculating [partial correlation](https://en.wikipedia.org/wiki/Partial_correlation). |
| polynomial time | [P](https://en.wikipedia.org/wiki/P_(complexity)) | 2*O*(log*n*) = poly(*n*) | *n*2 + *n*, *n*10 | [Karmarkar's algorithm](https://en.wikipedia.org/wiki/Karmarkar%27s_algorithm) for [linear programming](https://en.wikipedia.org/wiki/Linear_programming); [AKS primality test](https://en.wikipedia.org/wiki/AKS_primality_test); when input doubles slow down by constant |
| quasi-polynomial time | QP | 2poly(log*n*) | *n*log log*n*, *n*log*n* | Best-known O(log2 *n*)-[approximation algorithm](https://en.wikipedia.org/wiki/Approximation_algorithm) for the directed [Steiner tree problem](https://en.wikipedia.org/wiki/Steiner_tree_problem). |
| sub-exponential time (first definition) | SUBEXP | *O*(2*nε*) for all *ε* > 0 | *O*(2log *n*log log *n*) | Contains [BPP](https://en.wikipedia.org/wiki/Bounded-error_probabilistic_polynomial) unless EXPTIME (see below) equals [MA](https://en.wikipedia.org/wiki/Arthur%E2%80%93Merlin_protocol#MA).[[5]](https://en.wikipedia.org/wiki/Time_complexity#cite_note-bpp-5) |
| sub-exponential time (second definition) |  | 2*o*(*n*) | 2*n*1/3 | Best-known algorithm for [integer factorization](https://en.wikipedia.org/wiki/Integer_factorization); formerly-best algorithm for [graph isomorphism](https://en.wikipedia.org/wiki/Graph_isomorphism_problem) |
| exponential time (with linear exponent) | [E](https://en.wikipedia.org/wiki/E_(complexity)) | 2*O*(*n*) | 1.1*n*, 10*n* | Solving the [traveling salesman problem](https://en.wikipedia.org/wiki/Traveling_salesman_problem) using [dynamic programming](https://en.wikipedia.org/wiki/Dynamic_programming) |
| exponential time | [EXPTIME](https://en.wikipedia.org/wiki/EXPTIME) | 2poly(*n*) | 2*n*, 2*n*2 | Solving [matrix chain multiplication](https://en.wikipedia.org/wiki/Matrix_chain_multiplication) via [brute-force search](https://en.wikipedia.org/wiki/Brute-force_search) |
| factorial time |  | *O*(*n*!) | *n*! | Solving the [traveling salesman problem](https://en.wikipedia.org/wiki/Travelling_salesman_problem) via [brute-force search](https://en.wikipedia.org/wiki/Brute-force_search) |
| double exponential time | [2-EXPTIME](https://en.wikipedia.org/wiki/2-EXPTIME) | 22poly(*n*) | 22*n* | Deciding the truth of a given statement in [Presburger arithmetic](https://en.wikipedia.org/wiki/Presburger_arithmetic) |

Recurrence Relations = function is determined in terms of itself, i.e. subsequent values depend on previous values

Traditional example is Fibonacci Sequence: F(n) =

Some examples have easy equations:

Geometric Progressions: if A0 = k and An – dAn-1 then An = k(dn)

*e.g*. (3, 6, 12, 24, 48, . . .) = 3(2n)

if An – An-1 = k and A0 = c then An = c + which reduces to n(n + 1) if A0 = 0

e.g. (0, 2, 6, 12, 20, 30, 42, . . .) = n(n + 1) see <https://www.youtube.com/watch?v=eAaP4XaB8hM> at 9:24

Homogenous Recurrence Relations: have 4-step solution process:

1. Create characteristic polynomial:

e.g. An – An-1 – 6An-2 = 0 where A0 = 1 and A1 = 8 create characteristic polynomial

rn – rn-1 – 6rn-2 = 0 then dive each term by the lowest power of n => n-2

r2 – r – 6 = 0 then factor

(r – 3)(r + 2) then An = the solutions raised to n with coefficients – if both roots same

You have to multiply one by n as follows: An = α(3n) + βn(3n)

An = α(3n) + β(-2n) then solve for coefficients

A0 = 1 = α(30) + β(-20) = α + β => 2 = 2α + 2β

A1 = 8 = α(31) + β(-21) = 3α + 2β => 8 = 3α + 2β

10 = 5α => α = 2 and β = -1 therefore

An = 2(3n) + (-1)(-2n)

1. Factor

(r – 3)(r + 2) then An = the solutions raised to n with coefficients – if both roots same

you have to multiply one by n as follows: An = α(3n) + βn(3n)

1. Determine form of An

An = α(3n) + β(-2n) then solve for coefficients

1. Solve for coefficients

A0 = 1 = α(30) + β(-20) = α + β => 2 = 2α + 2β

A1 = 8 = α(31) + β(-21) = 3α + 2β => 8 = 3α + 2β

10 = 5α => α = 2 and β = -1 therefore

An = 2(3n) + (-1)(-2n)

Proofs

Recursion

MASTER METHOD: k = power on n, p = power on logn in f(n) if form T(n) = aT(n/b) + f(nk logpn):

**CASE 1:** if f(n) ≤ (nlogba-∊)then T(n) = Θ(nlogba) EX: T(n) = 9T(n/3) + f(n), nlogba = Θ(n2) so f(n) = *O*(n2-∊) so T(n) = Θ(n2).

**Example:** T(n) = 4T(n/2) + n here log24 = 2 which is larger than the exponent on n1 so **Case 1** = Θ(nlog24) = Θ(n2)

**CASE 2**: if logba = k then: **a**) if p > -1 then Θ(nk logp+1n); **b**) if p = -1 then Θ([nk or nlogba] logn); **c**) if p < -1 then Θ(nk log n).

**Example:** T(n) = 4T(n/2) + n2 here log24 = 2 which is = to the exponent on n2 so **Case 2** = Θ(nlogba log n) = Θ(n2 log n)

**CASE 3**: if logba < k or the traditional way to write it is if f(n) = O(nlogba + ɛ) then: **a**) if p ≥ 0 then Θ(nk logpn); **b**) if p < 0 Θ(nk)

**Example:** T(n) = 4T(n/2) + n3 here log24 = 2 which is < the exponent on n3 so **Case 3** = Θ(n3)

**CASE 3**: if f(n) ≥ (nlogba+∊) [*i.e.* Ω(nlogba+∊) ] **AND** if af(n/b) ≤ cf(n) for c < 1 then T(n) = Θ(f(n))

EX: T(n) = 3T(n/4) + n logn. f(n) = ≥ nlog43+∊ where ∊ ≈ 0.2 so f(n) = Θ(n) **AND** 3f(n/4)\*log((n/4) ≤ (¾) n logn so cf(n) says c = ¾ < 1 so T(n) = Θ(f(n)) which is Θ(n log n).

Inadmissible (Master Method does not work) if: A ≠ constant; A < 1; f(n) is negative; Case 3 but fails regularity EX: f(n) = n(2 – cosn); f(n) – n/logn = Case2b

Worst-Case Analysis

Discrete Math:

Sets

Functions

Logic

**Graphs:**

G = (V, E) where V is the set of nodes and E is the set of edges between pairs of nodes

Size parameters: n = |V|, m = |E|

Adjacency Matrix = n x n matrix with two representations of each edge showing a 1 if (u, v) is an edge

Checking if (u, v) is an edge takes Θ(1) and identifying all edges takes Θ(n2)

Adjacency List = linked-list-like representation with space proportional to m + n w/2 reps of each edge

Checking if (u, v) is an edge takes O(deg(u)) and identifying all edges takes Θ(m + n)

Path in undirected graph = sequence P of nodes with property each consecutive vi, vi+1 is joined by an edge

Path is simple of all nodes are distinct

Undirected graph is connected if for every pair of nodes u and v there is a path between u and v

Cycle is a path in which v1 = vk, k > 2 and the first k-1 nodes are all distinct (does not use all nodes)

Tree is an undirected graph with no cycles – thus it has n – 1 edges

Rooted Tree is tree with each edge oriented away from a root node r

e.g. Phylogeny Tree showing evolutionary history, GUI Containment Hierarchy

**Graph Traversal:**

Depth-First Search (DFS) – arbitrarily goes as far at it can (when it has no edge to follow that does not lead to a node that has already been visited then back-tracks

Used to find connected components – see <https://www.youtube.com/watch?v=7fujbpJ0LB4> at 5:09

Psuedocode:

n = number of nodes in graphs

g = adjacency list showing what edges end at what nodes

visited[ false, false, . . . , false] // number of nodes

start\_node = 0

dfs(start\_node): // argument = at so dfs(at): and we send it start\_node = 0

if visited[at]: return visited[at] = true // base case of recursion

neighbors = graph[at]

for next in neighbors:

dfs(next) // recursive call

Breadth First Search = one layer at a time until all nodes visited to find shortest path on unweighted graphs

Algorithm: uses a queue with start node on bottom and each visited node enqueued on top of that

Finding shortest path takes O(V + E)

Psuedocode:

n = number of nodes

g = adjacency list

s = start node

e = end node bfs(s, e) = subfunctions prev = solve(s) and reconstructPath(s, e, prev)

solve(s)

q = queue data structure

q.enqueue(s)

bool array visited with all false values for each node but mark starting node as true

array of nodes indicating who parent of each node is with all starting at NULL

while !q.isEmpty(): => node = q.dequeue() => neighbors = g.get(node)

for(next : neighbors):

if !visited[next]:

q.enqueue(next)

visited[next] = true

prev{next] = node

return prev

reconstructPath(s, e, prev) // by going backwards from e to s

path = []

for(at = e; at != null; at = prev[at])

path.add(at)

path.reverse() // because path is e-to-s and we want s-to-e

if(path[0] == s // check if path connects s to e

return path

Bipartite Graphs = undirected graph where nodes can be colored red or blue such that every edge has one red end and one blue end – applications include stable marriage and scheduling of machines

If a graph is bipartite it cannot contain an odd length cycle

Looking at the layers of graph created by a BFS, exactly one of the following holds:

1. No edge joins two nodes of the same layer, thus the graph is bipartite; or
   1. Results in blue nodes being on even layers and red nodes being on odd layers
2. An edge joins two nodes of the same layer, thus the graph contains an odd cycle/is not bipartite

Connectivity in Directed Graphs

Directed Reachability = given a node, find all nodes reachable from that node

Directed s-t Shortest Path Problem = given two nodes what is shortest path between them

Graph Search – BS extends to directed graphs

Strong Connectivity = node u and v are mutually reachable if there is a path between in both directions

Graph is Strongly Connected if every pair of nodes is mutually reachable, i.e. if you choose node s, G is strongly connected iff every node is reachable from s and s is reachable from every node

Algorithm for strong connectivity runs in O(m + n)

Pick any node s

Run BFS from s

Run BFS from s in reverse

Return true if all nodes reachable in both BFS executions

Directed Acyclic Graphs (DAG) = directed graph that contains no directed cycles, e.g. precedence constraints

If G is a DAG than G has a node with no incoming edges

Topological Order of a directed graph = in an ordering of nodes v1, v2, . . ., vn every edge (vi, vj) has i < j

If G has a topological order then G is a DAG, and if G is a DAG it has a topological order

Algorithm for finding topological ordering:

Find node v with no incoming edges and order it first and delete from G

Recursively compute a topological ordering of G-{v} and append this order after v

Basic Data Structures and Algorithms

Hash Tables

Hash Function = maps a key to a whole number in a fixed range

Separate Chaining = using data structure (usually a linked-list) for each slot that has collisions

Open Addressing = collision solution based on Load Factor = (items in table)/(size of the table) being below 80%

When there is a collision a Probing Sequence is used to find an open slot for the hash value – 4 kinds:

1. Linear Probing: P(x) = ax + b where a, b are constants |
   1. Need to pick a and N (table size) that are relatively prime to each |

other which means the greatest common denominator is one |

* 1. Start using the probing function with x = 0 if there is no collision |
  2. Each time you use P(x) and do NOT find an open slot you ++x |

1. Quadratic Probing: P(x) = ax2 + bx + c where a, b, c are constants |
   1. Most randomly chosen QP functions will create cycles – 3 solutions: | each subject to cycles
      1. Let P(x) = x2 and table size = prime > 3 with α < 50% |
      2. Let P(x) = (x2 + x)/2 and table size a power of 2 |
      3. Let P(x) = (-1)x \* x2 and table size a prime N ≡ 3 mod 4 |
2. Double Hashing: P(k, x) = x \* H2(k) where H2(k) is a secondary hash function |
   1. There are well-known secondary hash functions for each data type |
   2. If delta evey calculates to 0, change it to one |
3. Pseudo Random Number Generator: P(k, x) = x \* RNG(H(k), x)

Binary search Trees – has height Θ(log n) as each node has no more than 2 children

**treeSearch**(x, k): where x = root of tree or subtree to be searched and k = value searched for

if(x = null or k = x.key):

return x

else if(k < x.key):

return treeSearch(x.left, k)

else return treeSearch(x.right, k)

**treeInsert**(T, z): where T = tree and z = value to be inserted – runs in O(height)

y = null

x = T.root

while (x != null) do:

y = x

if(z.key < y.key)

x = x.left

else x = x.right

z.parent = y

if(y = null)

T.root = z

else if(z.key < y.key)

y.left = z

else y.right = z

**treeTransplant**(T, u, v) : = replace subtree rooted at u with another tree rooted at v

if(u.parent = null) // u is the root

T.root = v

Else if(u == u.parent.left) // u is the left child

u.parent.left = v

else

u.parent.right = v // u is the right child

if(v != nill)

v.parent = u.parent

**treeInsert**(T, z): where T = tree and z = value to be deleted – runs in O(height)

see BST slides 9-10

if z.left = nil then // z has no left child

TRANSPLANT(T,z,z.right)

elseif z.right=nil then // z has no right child

TRANSPLANT(T,z,z.left)

else // z has two children

y=TREE-MINIMUM(z.right) // find successor of node z

if y.parent != z then

TRANSPLANT(T,y,y.right)

y.right=z.right

y.right.parent =y

TRANSPLANT(T,z,y)

y.left = z.left

y.left.parent = y

if(z.left = nil or z.right = nil) // choose node to splice-out

y = z // z has only one child

else y = TREE-SUCCESSOR(z) // z has two children. We will/\* slice out its successor y which

if(y.left nil) // has at most 1 child

z = y.left

else // x is set to non-nil

x = y.right // child of y

if(x != nil)

x.parent = y.parent

if(y.parent = nil)

T.root = x

else

if(y = (y.parent).left)

(y.parent).left = x

else (y.parent).right = x

if(y != z) z.key = y.key

return y

Heaps = balanced binary tree but does not follow BST rules and is stored as an array w/o a node class

min (elements all smaller than their children so root smallest) or max (elements are all larger than their children)

root = 0 element and its left child = 1 and its right child = 2 etc so heap tree can be stored as an array

Red-Black Trees = approximately balanced BST with height at most Θ(log n) and the following properties:

1. Every node is either red of black
2. Root node is black
3. Every leaf node is black
4. If a node is red both of its children are black
5. Every simple path from a node to a descendant leaf contains the same number of black nodes

All basic BST operations can be performed in Θ(log n) time

A RB-tree with n internal nodes has a height at most height at most 2log(n + 1)

Greedy Algorithms = at each step of the algorithm choose the option that is best at the moment

Divide-and-Conquer = continually cutting search in half

Dynamic Programming

Minimum Spanning Trees (MST) uses (Kruskal’s and Prim’s algorithms) to connect all nodes of undirected graph at minimal cost using union-find

Kruskal’s Algorithm: (greedy – runs in O(m log n) where m is union-find ops and n is number of nodes)

Sort edges by ascending edge weight

Walk through sorted edges to see what nodes they belong to and if already unified do not include edge

Here you create groups and check if a new node is already in a group: if it is leave it and if it is not join it to the group that the other node of the edge was matched up to in the sorted edges using union-find

Terminate when every edge has been processed or all nodes have been unified

Prim’s Algorithm: (greedy and faster than Kruskal’s on dense graphs – does not unify minimum spanning forests)

Lazy version runs O(E log(E)) – uses priority queue filled with a poll like a breadth first search

Initiate minimum Priority Queue (PQ) that sorts edges based on cost

Adjacency List or Adjacency Matrix for referral to determine what nodes are connected by edges

Start algorithm at any node s and mark s as visited and iterates over all edges of s and adds them to PQ

While PQ is not empty, dequeue the next cheapest edge and add to the MST but skip if node at one end already visited – repeat

Psuedocode: lazyPrims(s = 0);

m = n-1

edgeCount, mstCount = 0, 0

mstEdges = [null, . . . , null] // up to size m

addEdge(s)

while(!pq.isEmpty() and edgeCount !=m):

edge – pq.dequeue()

nodeIndex – edge.to

if(visited[nodeIndex]:

continue

mstEdges[edgeCount++] = edge

mstCost += edge.cost

addEdges(nodeIndex)

if(edgeCount != m):

return(null, null) // no MST exists

return (mstCost, mstEdges)

Eager version runs O(E log(V)) where V is vertices/nodes and using Indexed Priority Queue

Initiate Indexed Priority Queue (IPQ) to store node index, edge object pairs

Adjacency List or Adjacency Matrix for referral to determine what nodes are connected by edges

Initiate visited Boolean array that tracks if nodes have been visited

Psuedocode: lazyPrims(s = 0);

m = n-1

edgeCount, mstCount = 0, 0

mstEdges = [null, . . . , null] // up to size m

relaxEdgeAtNode(s) // <https://www.youtube.com/watch?v=xq3ABa-px_g> at 12:00

while(!ipq.isEmpty() and edgeCount !=m):

// extract the next best (node index, edge object) pair from ipq

destNodeIndex, edge – ipq.dequeue()

mstEdges[edgeCount++] = edge

mstCost += edge.cost

relaxEdgesAtNode(destNodeIndex)

if(edgeCount != m):

return(null, null) // no MST exists

return (mstCost, mstEdges)

Shortest-Paths (Dijkstra’s algorithm) – based on Breadth First Search (BFS) finds path from source to all other nodes

All edges must have non-negative weight => this allows Dijkstra’s to act in a greedy manner (proof page 14 infra)

Maintain distance array where distance to start node = 0 and distance to all other nodes starts at infinite

This distance array is continually updated to keep shorter distances and delete longer distances to a node

Maintain priority queue of key value pairs which tells what node to visit next based on sorted minimum value

Insert (s, 0) into PQ and loop while PQ is no empty each time pulling out most promising node/distance pair

Iterate over all edges outwards from current node and relax each edge appending a new key-value pair to PQ

Pseudocode:

**dijkstra**(g, n, s):

vis = [false, false, . . . , false] // for the number size n

dist = [∞, ∞, . . . , ∞, ∞] // for the number size n

pq = empty priority queue // this would change to indexed priority queue for eager version

pq.insert((s, 0)) // this would change to indexed priority queue for eager version w/no object

while pq.size() != 0:

index, monValue = pq.poll()

vis{index] = true

if(dist[index] < minValue: continue // this is an optional optimization

for(edge : g[index]):

if vis[edge.to]: continue

newDist = dist[index] + edge.cost

if newDist < dist[edge.t]:

dist[edge.to] = newDist

pq.insert((edge.to, newDist)) // would change to indexed PQ if/else for eager version

return dist

**findShortestPath**(g, n, s, e):

dist, prev = Dijkstra(g, n, s)

path = []

if(dist[e] == ∞) return path

for(at = e; at != null; at = prev[at])

path.add(at)

path.reverse()

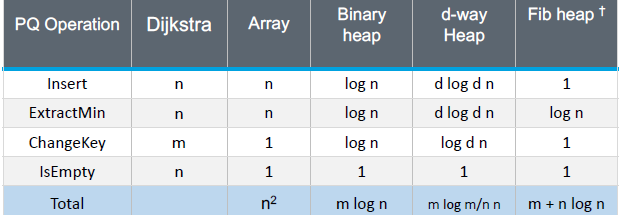
return path

Can optimize Dijkstra’s by not finding all shortest paths if we know the destination node by checking if current node index is the current node and if so aborting

“Eager” version of Dijkstra’s eliminates stale node-distance pares in the priority queue by using an Indexed Priority Queue (IPQ)

Using D-ary Heap instead of indexed heap would be an optimization as it speeds node distances/decreas key operations while slowing less numerous removal operations – use |edges|/|nodes| for D-ary size

Use of Fibonacci Heap for priority queue of unexplored nodes can reduce time complexity to O(m + n log n)



**Unit 1: Stable Matching (page 1)**

Module 1: Solving Stable Matching Problem with Algorithms

1. **Marriage Problem** = n number of men matched up to n number of women
   1. Stable if there is no pair not matched together would prefer to be matched
      1. Stable Matching = perfect matching with no unstable pairs
   2. Unstable if any UNMATCHED pair prefer each other to current partners

Gale-Shapley Propose-And-Reject Algorithm: finds one of possibly many stable matchings

WHAT DATA STRUCTURE FOR PREFERNCE MATRIX?

**Initialize each person to be free.**

**while (some man is free and hasn't proposed to every woman) {** // loop O(n2) as n men propose to n women

**Choose such a man m** // maintain list of free men in a queue to choose, thus O(1) naïve O(n)

**w = 1st woman on m's list to whom m has not yet proposed** // maintain array count[m] that counts

// proposals made by m and use pointer to

// point at last proposal, thus O(1) naïve O(n)

**if (w is free)** // if else then O(1) if careful by using arrays that show women are

// engaged to what man, and what men are engaged to what women

**assign m and w to be engaged** // maintain 2 arrays wife[m] and husband[w] initiated to 0

// if m matched to w then wife[m] = w and husband[w] = m

**else if (w prefers m to her fiancé m')** // create inverse preference index for each woman (take linear time

// to build this preprocessed data structure using for i = 1 to n in Pref inverse[pref[i]] = i) =then we have constant time O(n) naïve O(n)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Woman | 1st | 2nd | 3rd | 4th | 5th | 6th | 7th | 8th |
| Pref | 8 | 3 | 7 | 1 | 4 | 5 | 6 | 2 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Woman | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Inverse | 4th | 8th | 2nd | 5th | 6th | 7th | 3rd | 1st |

// you look at two values in inverse and compare them

**assign m and w to be engaged, and m' to be free**

**else**

**w rejects m**

**}**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Men’s Preference Profile** | | | | | |
| **V** | **~~B~~** | **A** | **D** | **E** | **C** |
| **W** | **D** | **B** | **A** | **C** | **E** |
| **X** | **B** | **E** | **C** | **D** | **A** |
| **Y** | **A** | **D** | **C** | **B** | **E** |
| **Z** | **B** | **D** | **A** | **E** | **C** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Women’s Preference Profil** | | | | | |
| **A** | **Z** | **V** | **W** | **Y** | **X** |
| **B** | **X** | **W** | **Y** | **~~V~~** | **Z** |
| **C** | **W** | **X** | **Y** | **Z** | **V** |
| **D** | **V** | **Z** | **~~Y~~** | **X** | **~~W~~** |
| **E** | **Y** | **W** | **Z** | **X** | **V** |

* 1. Proof of Correctness Gale-Shapley
     1. Termination:
        1. Observe: men propose to women in decreasing order of preference
        2. Observe: once a woman is matched, she never becomes unmatched; she only “trades-up”
        3. Claim: algorithm terminates after at most n2 iterations of while loop.
           1. Proof: each time through the while loop a man proposes to a new woman, so there are only n2 possible proposals => n(n – 1) + 1 proposals required maximum
     2. Perfection:
        1. Claim: all men and women get matched
           1. Proof (by Contradiction)

Suppose that Z is not matched

Then some woman is not matched

This means that some woman was never proposed to, but

Z proposed to everyone since he ended up unmatched => thus a contradiction

* + 1. Stability:
       1. Claim: there are no unstable pairs
          1. Proof (by Contradiction)

Suppose A-Z is an **unstable pair: each prefers each other to their matched partner**

Case 1: Z never proposed to A => Z prefers his matched partner to A => A-Z is stable

Case 2: Z proposed to A => A rejected Z => A prefers her matched partner to Z => A-Z OK

In either case A-Z is stable thus a contradiction

* 1. Asymptotic Running Time of Gale-Shapley = O(n2) if done efficiently as noted on pseudocode above but could be O(n3) if you built the algorithm naively
  2. Algorithm is Man Optimal = each man receives best valid partner – when men propose
     1. Claim: GS matching S\* is man-optimal
        1. Proof (by Contradiction)
           1. Suppose some man is paired with someone other than best partner. Men propose in decreasing order of preference => some man is rejected by a valid partner
           2. Let Y be first such man, and let A be first valid woman that rejects Y
           3. Let S be a stable matching where A and Y are patched
           4. When Y is rejected A forms (or reaffirms) engagement with a man, say Z, she prefers to Y
           5. Let B be Z’s partner in S
           6. Z not rejected by any valid partner at the point when Y is rejected by A, thus, Z prefers A to b
           7. But A prefers Z to y
           8. Thus A-Z is instable in s – thus a contradiction
  3. Man Optimal necessarily means it is women pessimal = each women receives the worst valid partner
     1. Claim: GS finds woman-pessimal stable matching S\*
        1. Proof:
           1. Suppose A-Z matched in S\* but Z is not worst valid partner for A
           2. There exists stable matching S in which A is paired with a man, say Y, whom she likes less than Z
           3. Let B be Z’s partner in S
           4. Z prefers A to B (from man optimality)
           5. Thus A-Z is an unstable pair in S

1. **Matching Residents to Hospitals** = self-reinforcing admission process = **NOTE that since the 1990s the residents have been the proposers in the real system, but Prof at Video 1:20 and the TA says we should think of the hospitals as the proposers**
   1. Stable if there is no pair not matched together would prefer to be matched
      1. Individual self-interest will prevent any applicant/hospital side-deal from being made
   2. Unstable if:
      1. At first she describes unstability like this:
         1. r prefers h to its assigned hospital
         2. h prefers r to one of its admitted residents
      2. Then later she describes unstability like this:
         1. h and r are acceptable to each other; and
         2. either r is unmatched or r prefers h to her assigned hospital; and
         3. either h does not have all its places filled, or h prefers r to at least one of its assigned residents
   3. VARIANT 1: some participants declare others as unacceptable
   4. VARIANT 2: unequal number of participants
   5. VARIANT 3: limited polygamy (hospital has multiple slots to fill)
2. **Roommate Problem – Irving’s Algorithm**
   1. unlike Gale-Shapley:
      1. creates pairs from a single set not two separate sets; and
      2. there might NOT be a stable matching
   2. Stable if no pair exists where both members prefers another as roommate over their current partner
   3. Unstable if any unmatched pair prefer each other as roommates over their current partner

Pseudocode: Squares = proposals Circles = acceptances Xs = rejections

PHASE 1: Run through proposals and acceptances making sure to cross-cancel rejections until each roommate ends up with both a proposal and acceptance

while there are unmatched people do

Let i be the smallest value such that ai is unmatched

ai proposes to his favorite roommate aj who has not previously rejected him

If aj has not received a proposal before then

aj accepts ai

else if aj prefers ai over his current match ak then

aj accepts ai

aj rejects ak symmetrically

else

aj rejects ai symmetrically

PHASE 2: reject worst choices by crossing out all options to the right of roommates’ acceptance and removing them and any proposal and acceptance notation so all you have is potential matches

For all accepted proposals (ai, aj) do

Reject all (aj, ak) where aj prefers ai over ak

PHASE 3: a) choose first choice with > 1 potential match and put above a line and put its 2nd best choice below the line; b) above the line place the worst choice of the option last placed below the line until a cycle is identified; c) cross cancel diagonally above and below line and symmetrically notate on diagram; d) continue this until each roommate only has one match

For all cycles (p1, . . . , p­) and associated second preferences (q1, . . . , qn) such that

qi is the second preference of pi

pi+1 is the last preference of qi

pn є { pc, . . . , pn-1 } do

for i = 1 . . . n-1 do

qi rejects pi+1

* 1. Asymptotic Running Time of Irving’s Algorithm for roommate matching = O(n2) + O(n2) + O(n2) = **O(n2)**
     1. Initializing various lists of preferences takes O(n2)
     2. Phase 1 and phase 2 together can have at most n2 rejects so they run in O(n2)
     3. Phase 3 runs in O(n2)
     4. Number of possible proposals:
        1. There are (n – 1) choices for everyone in their preference list. The worst case would be to have to read through the entire preference list of each roommate which would lead to n(n – 1) = n2 – n operations
        2. Therefore, in question where there are 2n roommates the worst case = 2n(2n – 1) = 4n2 – 2n

Module 2: Well-known Problems in Algorithms (page 12)

1. **Interval Scheduling Problem** = **greedy** = O(n log n)
   1. Input is set of jobs with start and finish times (looks like a Gantt Chart)
   2. Goal is to find the maximum cardinality subset of mutually compatible jobs
2. **Weighted Interval Scheduling** = **dynamic** = O(n log n) = generally maximize weight of jobs scheduled
   1. Solved by dynamic programming not greedy
3. **Bipartite Matching** = **max-flow** = O(nk) = matches two partitioned sets with no edges between nodes on one side of partition
   1. Solution is no 2 edges share an endpoint – so 1:1 matching
   2. Solved by max flows methodology
4. **Independent Set** = find maximum cardinality of nodes that make independent set (edges share no nodes in set)
   1. No efficient/polynomial time algorithm
   2. **NP-complete** problem
5. **Competitive Facility Location** = 2 competing players alternate in selecting nodes and cannot select a node if any of its neighbors have been selected – goal is to pick subset with maximum weight
   1. Input is a graph with a weight for each node – in a linear format
   2. Cannot be solved as **PSPACE-complete** problem which might not even be solvable in exponential time

**====================**

**Unit 2: Greedy Algorithms (page 115) – “never look back”**

**Module 1:** Interval Scheduling and Interval Partitioning (page 116)

1. Goal is to find maximum subset of mutually compatible jobs, i.e. jobs do not overlap in time
   1. She clearly says that number of jobs (cardinality of the set of jobs chosen) is a priority over utilization of resource (sometimes these are equal)
   2. Job j starts at sj and finishes at fj
2. Greedy Algorithms are used to solve Interval Scheduling problems: 4 ways to order jobs – take if compatible:
   1. Earliest Start Time = consider jobs in ascending order of start time sj
      1. Not Always going to get optimal solutions
   2. **Earliest Finish Time** = consider jobs in ascending order of finish time fj
      1. Only of these 4 algorithms that survives her optimality analysis
      2. Always pick the job with the earliest finish time that does not conflict with already chosen job
      3. **Psuedocode – increasing order of finish time: // this sort is O(n log n) so dominates algorithm**

**Sort jobs by finish time**

**A = set of jobs selected // could be queue or even array**

**For j = 1 to n { // this loop is O(n) so dominated by sort O(n log n)**

**if (job j compatible with A)**

**A <- A U { j }**

**}**

**return A**

* + 1. Asymptotic behavior = O(n log n)
       1. Remember job j\* that was added last to A
       2. Job j is compatible with A if sj ≥ fj
  1. Shortest Interval = consider jobs in ascending order of interval length
     1. Not Always going to get optimal solutions
  2. Fewest Conflicts = for each job, count the number of conflicting jobs cj and schedule in ascending order or conflicts cj
     1. Not Always going to get optimal solutions

1. Greedy earliest finish time is optimal **Proof** (by contradiction):
   1. Assume greedy is not optimal
   2. Let i1, i2, . . . , ik denote a set of jobs selected by greedy algorithm
   3. Let j1, j2, . . . , jm denote a set of jobs in an optimal solution with i1 = j2, i2 = j2, . . . , ir = jr for the largest possible value of r, i.e. the greedy and optimal are the same up until r then differ at r + 1
   4. The way she described proof is that she assumed an arbitrary optimal solution that had one chosen job different than (her example had the optimal picking a job with a later finishing time) the earliest finishing time algorithm. But when she swaps the different jobs, it is shown that the earliest finishing time set of jobs is also an optimal result, thus you have a contradiction
2. Interval Partitioning = goal is to find minimum number of classrooms to schedule all lectures – so difference from Interval Scheduling above is that instead of trying to run the maximum amount of jobs, here you must run all jobs and are finding the minimum number of resources needed to run those jobs
   1. Depth of a set of open intervals is the maximum number that contain any given time, i.e. the maximum number of intervals that overlap at the same time
   2. Number of classrooms needed ≥ depth
   3. Asymptotic behavior = O(n log n) // keep classrooms in priority queue
   4. **Psuedocode – interval partitioning (greedy):**

**Sort intervals by starting time so the s1 ≤ s2 ≤ s3 . . . , ≤ sn // Sort runs in O(n log n)**

**d <- 0 // Number of classrooms initiates at 0**

**for j = 1 to n { // Loop runs in O(n) but + log n each iteration**

**if (lecture j is compatible with some classroom k) // Constant time if set of lectures in queue e.g.**

**schedule lecture j in classroom k // maintain finish time of last job added which**

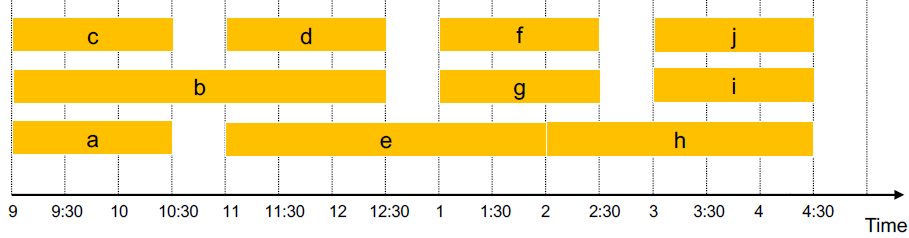
**else // can be a field in the queue**

**allocate a new classroom d + 1 // BUT you need to save earliest, latest finish**

**schedule lecture j in classroom d + 1 // time in additional priority queue that is a**

**d <- d + 1 // min heap data structure which takes log n to**

**} // update each for loop**



* 1. Asymptotic running time = O(n log n)
  2. Greedy is optimal **Proof**:
     1. Let d = number of classrooms that greedy algorithm allocates
     2. Classroom d is opened because we need to schedule a job, say j, that is incompatible with all d – 1 classrooms
     3. Since we sorted by start time, all these incompatibilities are caused by lectures that start no later than sj
     4. Thus we have d lectures overlapping at time sj + ɛ
     5. Key observation => all schedules use ≥ d classrooms

**Module 2:** Scheduling to Minimize Lateness (page 125, Slide 26) = schedule all jobs to minimize lateness total of all jobs

1. Job j requires tj units of processing time and is due at time dj, so j starts at time sj it finishes at time fj = sj + tj
2. Greedy algorithms for minimizing lateness: consider the jobs in some order
   1. Shortest processing time first = consider jobs in ascending order of tj
   2. Earliest deadline first = consider jobs in ascending order of deadline dj

**Psuedocode Earliest Deadline First:**

**Sort n jobs by deadline so that d1 ≤ d2 ≤ . . . ≤ dn // runs with O(n log n) like all sorts**

**t <- 0**

**for j = 1 to n { // loop runs in linear time O(n)**

**assign job j to interval [t, t + tj]**

**sj <- t // everything inside loop runs in constant time O(1)**

**fj <- t = tj**

**output intervals [sj, fj]**

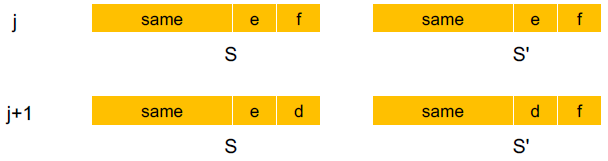
**} // overall run time = O(n log n)**

* 1. There exists an optimal schedule with no idle time and the greedy schedule has no idle time
  2. Smallest slack = consider jobs in ascending order of slack dj – tj

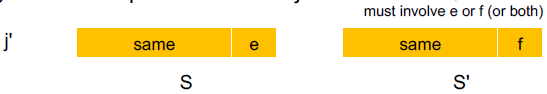
1. Analysis of Greedy Algorithm – Greedy schedule S is optimal – Proof:
   1. Define S\* to be an optimal schedule that has fewest number of inversions and analyze:
      1. Can assume S\* has no idle time
      2. If S\* has no inversions then S = S\*
      3. If S\* has an inversion let i-j be an adjacent inversion
         1. Swapping i and j does not increase the max lateness and strictly decreases number of inversions
         2. This contradicts definition of S\*
2. Minimizing Lateness: **Inversions** = in schedule S an inversion is a pair of jobs i and j such that the **deadline** of i < the deadline if j, but j is scheduled before i
   1. **Greedy schedule has no inversions**
   2. If a schedule with no idle time has inversion it has one with a pair of inverted jobs scheduled consecutively
   3. Swapping two adjacent, inverted jobs reduces the number of inversions by 1 but does not increase lateness
      1. Proof
3. Greedy Analysis Strategies
   1. Greedy algorithm stays ahead = show that after each step of the greedy algorithm its solution is at least as good as any other algorithm’s
   2. Exchange argument = gradually transform any solution to the one found by greedy w/o hurting quality
   3. Structural = discover a simple “structural” bound asserting that every possible solution must have a certain value. Then show that your algorithm always achieves this bound

**Module 3:** Optimal Offline Caching (page 131)

1. Caching:
   1. Cache with a capacity to store k items
   2. Sequence of m item requests d1, d2, . . . , dm
   3. Cache hit = item already in cache when requested
   4. Cache miss = item not already in cache when requested but must be brought into cache while evicting something else in cache
   5. GOAL is an eviction schedule that minimizes cache misses
   6. Farthest-In-Future = evict the item in cache that is not requested until farthest in the future = Optimal **offline** caching per Bélády in the 1960s and serves as a guideline for optimal, BUT **only works in Offline case where you know the sequence of up-coming cache requests** **– runs in O(n)**
   7. Proof (by induction on number of requests j): **S** = optimal reduced schedule (one always exists) that matches Sff up to j requests; **Sff** = eviction schedule output by our algorithm; **S’** is what be are building based on S that satisfies the invariant and will match Sff up to j + 1 requests
      1. Invariant: there exists an optimal reduced schedule S that makes the same eviction schedule as Sff through the first certain number of requests (here j + 1)
      2. Let S be reduced schedule that satisfies invariant through j requests. We produce S’ that satisfies invariant after j + 1 requests (this is simple a certain number of requests):
         1. Consider (j + 1)st request d = dj+1
         2. Since S and Sff have agreed up until now they have same cache contents before request j + 1
         3. **Case 1:** (d is already in the cache) S’ = S satisfies invariant
         4. **Case 2:** (d is not in the cache and S and Sff evict same elements) S’ = S satisfies invariant
         5. **Case 3:** (d is not in the cache of either S or Sff as they have same contents up until now; Sff evicts e; S evicts f ≠ e) – *[S and Sff had done same thing prior]*
            1. Begin construction of S’ from s be evicting e instead of f

 here S’ = Sff

* + - * 1. Now S’ agrees with Sff on the first j + 1 requests; we show that having element f in cache is no worse than having element e and can still lead to an optimal schedule
      1. Let j’ be the first time after j + 1 that S and S’ take a different action, and let g be item requested at time j’

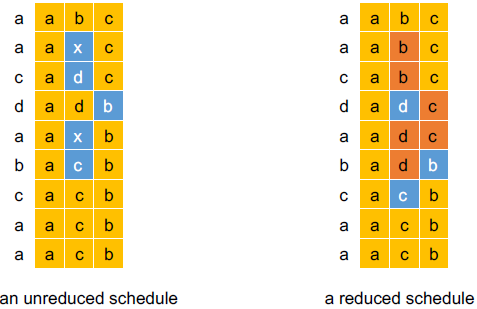
 S’ mimics what Sff does

What forces S’ to take a different action than S is when a request g is in S but not S’ such as e being in S and f being in S’ thus the comment “must involve e or f (or both)”

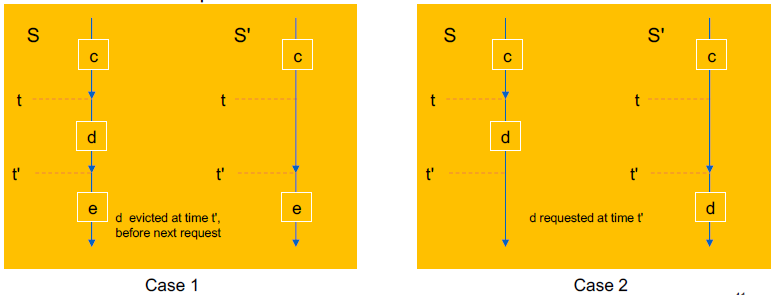
* + - 1. **Case 3a:** g = e. Cannot happen with Farthest-In-Future as there must be a request for f before e, *i.e*. if e was to be accessed sooner than f, F-I-F would never have kept f and put e out (contradiction)
      2. **Case 3b**: g = f. Element f cannot be in cache of S so let e’ be the element that S evicts
         1. If e’ = e, S’ accesses f from cache; now save S and S’ have same cache
         2. If e’ ≠ e, S’ evicts e’ and brings e into the cache; now S and S’ have same cache (note S’ is no longer reduced, but can be transformed into a reduced schedule that agrees with Sff though step j + 1)
         3. S’ is no longer reduced, but can be transformed into a reduced schedule that agrees with Sff through step j + 1
      3. **Case 3c:** g ≠ e, f. S must evict e. Make S’ evict f; now S and S’ have same cache



1. Reduced Eviction Schedules = a schedule that only inserts an item into the cache in a step in which that item is requested – intuition says you can transform an unreduced schedule into a reduced one with no more cache misses:

 for reduced if hit do nothing

* 1. Claim (by intuition) – given an unreduced schedule S, you can transform it into a reduced schedule S’ with no more cache misses than were in S
  2. Proof (by induction on number of unreduced items, *i.e*. items that do not enter cache at requested time):
     1. Suppose S brings d into cache at time t without a request (this is an “unreduced move”)
        1. S = unreduced and S’ = reduced
     2. Let c be the item S evicts when it brings d into the cache
     3. Case 1: d evicted at time t’ before next request for d = easy case as unreduced has no effect



* + 1. Case 2: d requested at time t’ before d is evicted = d stayed in cache long enough that it had an effect
       1. Here the video ran out before she finished

1. Caching Perspective:
   1. Online vs. offline algorithms
      1. Offline: full sequence of requests is known a priori
      2. Online (reality): requests are not known in advance – approximation will be best we can do
      3. Caching is among the most fundamental online problems in Computer Science
   2. LIFO (Last In First Out) = evict page brought in most recently
   3. LRU (Least Recently Used) = evict page whose **most recent access was earliest**
      1. This is ff with direction of time reversed
   4. Theorem: ff is optimal offline eviction algorithm
      1. Provides basis for understanding and analyzing online algorithms
      2. LRU is k-competitive (Section 13.8)
      3. LIFO is arbitrarily bad

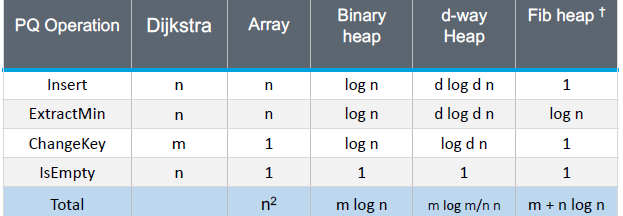
**Module 4:** Further Examples of Greedy Algorithms

1. Shortest Path – on directed graph (V, E) – does not NEED to be directed, just weighted, BUT NOT negative
2. Dijkstra’s (*see* page 7 *supra*) – **finds shortest path** to every node in graph not just target – short description:
   1. Can take negative edge weights
   2. Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from S to u
   3. Initialize S = { s }, d(s) = 0
   4. Repeatedly choose unexplored node v which minimizes shortest path π(v) and add v to S and set d(v) = π
   5. Invariant: for each node u є S, d(u) is the length of the shortest s-u path
   6. Proof (by induction on |S|):
      1. Base case: |S| = 1 is trivial (and shows algorithm is recursive)
      2. Inductive hypothesis: assume true for |S| = k ≥ 1
         1. Let v be the next node added to S and let u-v be the chosen edge
         2. The shortest s-u path path plus (u, v) is an s-v path of length π(v)
         3. Consider any s-v path P and see it is no shorter than π(v)
         4. Let x-y be the first edge in P that leaves S and let P’ be the subpath to x
         5. P is already too long as soon as it leaves S

λ (P) ≥ λ(P’) = λ(x, y) ≥ d(x) = λ(x, y) ≥ π(y) ≥ π(v)

Noneg weights inductive hypoth def of π(y) Dijkstra chose v instead of y

* 1. Implementation – page 7 *supra*, slide 55 in Greedy Algos - ᵻ signifies individual ops are amortized bounds



1. Minimum Spanning Tree – subset of edges that connect all nodes with minimum weight – negative edge weights allowed if definition changed slightly
   1. Cayley’s Theorem = there are nn-2 spanning trees of Kn (no poly time algo/brute force not best approach)
   2. Direct Applications for MST: network design, approximation algorithms for NP-hard problems such as travelling salesperson and Steiner tree
   3. Indirect application for MST: max bottleneck paths, Low Density Parity Check (LDPC) code for error correction, image restoration with Renyi entropy, learning salient features for real-time face verification, reducing data storage in sequencing amino acids in a protein, model locality of particle interactions in turbulent fluid flows, autoconfig protocol for Ethernet bridging to avoid cycles in a network
   4. Greedy Algorithms that all **produce a minimum spanning tree**:
      1. Kruskal’s = start with t = φ and consider edges in ascending order of cost, i.e. sort them first. Insert edge e in T unless doing so would create cycle
         1. Cannot take negative edge weights (theoretically might work, but need Bellman-Ford instead)
         2. Runs in O(m log n) for sorting and O(m α (m, n)) for union find
         3. **Pseudocode:**

**Kruskal (G, c) {**

**Sort edges by weights so that c1 ≤ c2 ≤ . . . ≤ cm // sort time controls run time**

**for each (u є V) make a set containing singleton u**

**for i = 1 to m**

**(u, v) = e1**

**If (u and v are in different sets) { // sees if u and v in diff connected components**

**T <- T U { e1 } // this means do not choose if creates a cycle**

**Merge the sets containing u and v // merge two components**

**}**

**return T**

**}**

* + 1. Reverse-Delete Algorithm = start with T = E and consider edges in descending order. Delete edge e from T unless doing so would disconnect T
       1. Reverse of Kruskal you start with ALL the edges in the graph and delete edges while deleting an edge does not disconnect set T
       2. Rather than start by considering lowest weight edge you start with highest weight edge
          1. You cut edges due to cycle property and keep edges due to cut property
    2. Prim’s (like Dijkstra’s and BFS)= start with some root node s and greedily grow a single tree T from s outward (compared to Kruskal’s multiple trees). At each step apply cut property to s and add the cheapest edge e to T that has exactly one endpoint in T. Maintain set of explored nodes S. For each unexplored node v, maintain attachment cost a[v] = cost of cheapest edge form v to a node in S
       1. Can take negative edge weights
       2. Runs in O(n2) with an array and O(m log n) with a binary heap (m = n – 1 which is full MST)
       3. **Psuedocode:**

**Prim(G, c) {**

**for each (v є V) a[v] <- ∞**

**Initialize an empty priority queue Q**

**for each (v є V) insert v into Q**

**Initialize set of explored nodes S <- φ**

**while (Q is not empty) {**

**u <- delete min element from Q**

**S <- S U { u ]**

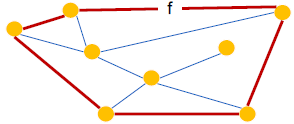
**for each (edge e = (u, v) incident to u)**

**If ((v /є S) and (ce < a[v])) // difference with Dijsktra’s is what you use for key of**

**Decrease priority a[v] to ce // priority queue – here just least cost edge – and DFS**

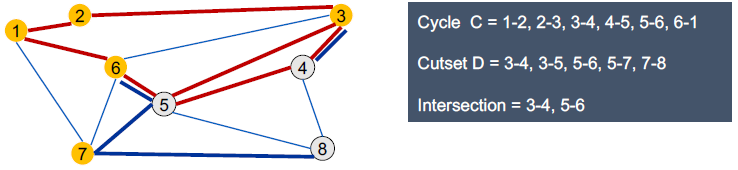
**} // does not need priority queue, just a queue**

1. Cycles and Cuts
   1. Cut = subset S of nodes and we are interested in edges with only one endpoint in the subset
      1. Edges have one endpoint in S and one endpoint in the subset V - S
   2. Cut Property: let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then MST contains e – see below for proof
2. Cycle = set of edges that form a loop
   1. **Cycle property:** let C be any cycle in G and let f be the max cost edge belonging to C. Then the MST T\* does not contain f

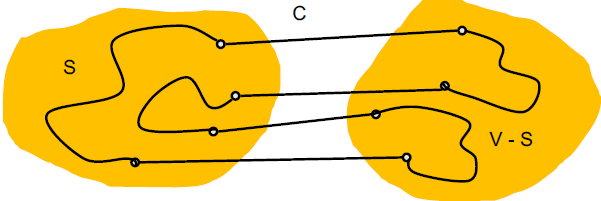


* 1. Proof (exchange argument):
     1. Suppose f belongs to T\* and see what happens
     2. Deleting f from T\* creates a cut in S in T\*
     3. Edge f is both in the cycle C and in the cutset D corresponding to S => there exists another edge e that is in both C and D
     4. T’ = T\* U {e} – {f} is also a spanning tree
     5. Since ce < cf cost(T’) < cost(T\*) which is a contradiction

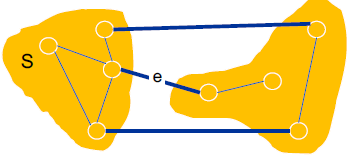
1. Cutset: cut = subset of nodes and corresponding cutset is the subset of edges that has exactly one endpoint in S
   1. A cycle and a cutset intersect in an even number of edges because you are moving from cut S to V – S then you have to come back again



Proof by picture:



* 1. **Cut Property:** let S be any subset of nodes and let e be the minimum cost edge with exactly one endpoint in S then the MST T\* contains e



* 1. Proof (exchange argument):
     1. Suppose e does not belong to T\* (a contradiction) and see what happens
     2. Adding e to T\* creates a cycle in C in T\*
     3. Edge e is both in the cycle C and in the cutset D corresponding to S => there exists another edge f that is in both C and D
     4. T’ = T\* U {e} – {f} is also a spanning tree
     5. Since ce < cf cost(T’) < cost(T\*) which is a contradiction

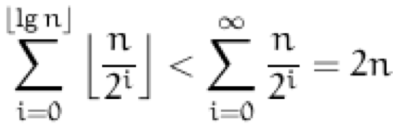
1. Clustering = given a set U of n objects labeled p1, p2, . . . , pn, classify into coherent groups
   1. Distance Function = numeric value specifying “closeness” of two objects
   2. Fundamental problem = divide into clusters so that points in different cluster are far apart – used in: routing in mobile ad hoc networks, identifying patterns in gene expression, document categorization for web search, similarity searching in medical images databases, Skycat = cluster 109 sky objects into stars, quasars, galaxies
   3. Clustering of maximum spacing
      1. K-clustering = divide objects into k non-empty groups, *i.e*. fix the number of clusters to be found
      2. Distance function – assume it satisfies several natural properties
         1. d(pi, pj) = 0 iff pi = pj identity of indiscernibles
         2. d(pi, pj) ≥ 0 nonnegativity
         3. d(pi, pj) = d(pj, pi) symmetry
      3. Spacing = minimum distance between any pair of points in different clusters = found by comparing every pair of nodes where one node is in one cluster and the second node is in the other cluster and choosing the pair that has the shortest distance (she later mentions that as part of k-clustering algorithm you do find all these pairs and their distances)
      4. Clustering of maximum spacing = given an integer k, find a k-clustering of maximum spacing
2. Greedy Clustering Algorithm
   1. Single-link k-clustering algorithm:
      1. Form a graph on the vertex set U, corresponding to n clusters
      2. Find the closest pair of objects such that each object is in a different cluster, and add an edge between them
      3. Repeat n-k times until there are exactly k clusters
   2. Key observation: this procedure is precisely Kruskal’s algorithm except we stop when there are k connected components, *i.e*. you stop when n – k edges have been inserted into the set
      1. k-cluster is formed by taking MST and deleting k – 1 weightiest edges; or it can be said
      2. to cluster run Kruskal’s but stop when n – k edges inserted into spanning tree
   3. Remark: equivalent to finding an MST and deleting the k – 1 most expensive edges
   4. Theorem: let C\* denote the clustering C\*1, . . . , C\*k formed by deleting the k – 1 most expensive edges of a MST. C\* is a k-clustering of max spacing.
   5. Proof: let C denote some other clustering c1, . . . , Ck
      1. The spacing of C\* is the length d\* of the (k – 1)st most expensive edge
      2. Let pi, pj be in the same cluster in C\* called C\*r but in different clusters in C called Cs and Ct
      3. Some edge (p, q) on pi – pj path in C\*r spans two different clusters in C
      4. All edges on pi – pj path have length ≤ d\* since Kruskal chose them
      5. Spacing of C is ≤ d\* since p and q are in different clusters
   6. Similarly, if you have a k-clustering and you want to change it to a k – 1 clustering, simply add the minimum weight edge across the 3 components which turns 2 closest together clusters into one cluster

**====================**

**Unit 3: Amortized Analysis and Splay Trees**

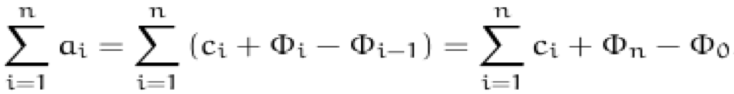
Module 1: Amortized Analysis

1. A binary representation of a number is exactly log n + 1 bits long. For example log232 = 5 + 1 = 6 = 10 0000
   1. Thus an INCREMENT function on an integer between 0 and n takes Θ(n log n) in the worst case which is n times for each 0 - n then log n worst case for each bit flipping operation for each n – BUT the worst case only happens when a lot of the bits are 1, which is not every time – so count exact number of bit flips

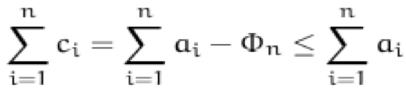


* 1. NOTE here that if the average is 2n and we run n times that is 2n/n so average per call is 2 not n log n

1. Aggregate Method = average case = sum up all operations then divide by n
   1. To find the worst case running time for a sequence of n operations the amortized cost of each operation is T(n)/n
   2. Difficult to apply in some scenarios suggesting use of another method
2. Accounting Method = pay for each of a certain kind of operation and save for another kind of operation
   1. Suppose it costs us a dollar to toggle a bit
   2. Instead of paying for each bit flip when it happens we charge 2 dollars when we want to set a bit from 0 to 1
      1. One of those dollars is spent changing the bit from 0 to 1
      2. The other is stored as credit until we need to reset the same bit back to 0
      3. We always have enough credit to pay for the next INCREMENT
   3. The amortized cost of an INCREMENT is the total charge it incurs which is exactly 2 dollars since each INCREMENT changes just one bit from 0 to 1
3. Potential Method = instead of associating costs or charges with particular operations we consider prepaid work as potential that can be spent on later operations. The potential is a function of the entire data structure
   1. Most powerful method but hardest to use – builds on the physics metaphor of “potential energy”
      1. Example: riding bicycle down and up hills: you work twice as hard to go up hill, but get free ride down
      2. Start data structure with 0 potential energy then operations build or use energy
   2. Instead of associating costs or charges with particular operations we consider prepaid work as potential that can be spent on later operations. The potential is a function of the entire Data Structure
   3. Analysis based on binary counter:
      1. Let Di denote our data structure after i operations and let φi denote its actual potential
         1. φi = the number of bits set to 1 on the binary counter and can never be negative
      2. Let ci denote the actual cost of the ith operation which changes Di-1 int Di
      3. The amortized cost of the ith operation denoted ai is defined as the actual cost plus the change in potential: **ai = ci + φi - φi-1**
      4. The total amortized cost of n operations is the actual total cost plus the total change in potential:



* + 1. Define a potential function so that φ0 = 0 and φi ≥ 0 for all i (starts at 0 and never goes negative)
    2. The total actual cost of any sequence of operations will be less than the total amortized cost



1. **Potential Method explanation from MIT:** <https://www.youtube.com/watch?v=3MpzavN3Mco> @42:00
   1. “Defining Karma in a formal way”
   2. Like the accounting method in that you can think about there being a bank account with some balance, but that balanced is defined as a function of the data structure state
   3. Define potential function φ – mapping data structure configuration – must always be non-negative
      1. Potential function is trying to measure “how bad is the data structure right now”
      2. From the potential function you can compute the Δs, *i.e*. the actual values we paid and spent using the accounting function – “but it is a little bit of black magic to come up with this function”
      3. Amortized cost = actual cost + Δφ where **Δφ = φ(after operation) – φ(before operation)**
         1. **If you add the right side up it telescopes via cancellation for each term with the previous term**
      4. What we care about is the sum of the amortized costs because it is supposed to be an upper bound on the sum of the actual costs
      5. Σamortized costs = Σactual costs + φ(end) – φ(beginning)
         1. You have to pay φ(beginning) so that gets moved to the left side so we end up with Σamortized costs + φ(beginning) = Σactual costs + φ(end)
         2. After assuming start with empty structure at 0, so φ(beginning) = 0
   4. Examples:
      1. Incrementing a Binary Counter: costs 1-2-1-3-1-4 . . .
         1. Increment has clear actual cost: number of trailing 1s + 1
         2. Define φ to be the total number of 1 bits
         3. Increment destroys t 1s and creates one 1 bit
         4. Amortized cost = 1 + t – t + 1 = 2
         5. If you were measuring running time instead of the number of changes bits you would have to make the actual cost O(1 + t) and put a constant in from of φ so that φ = c(number of 1 bits) which if c is big enough c(-t + 1) would annihilate O(i + t) => a = O(1 + t) – ct + 1c = 2
2. Potential Method on Binary Counter – hardest part is picking best function, but no good rules for choosing
   1. The potential function φi after the ith INCREMENT is equal to the number of bits with value 1
   2. Initially all bits are zero so φ0 = 0 and φi ≥ 0 for all i
   3. The actual cost of an INCREMENT and the change in potential then become:
      1. ci = #bits changed from 0 to 1 + #bits changes from 1 to 0 (NOTE right of + is same as next line)
      2. φi - φi-1 = #bits changed from 0 to 1 - #bits changed from 1 to 0 (NOTE right of – is same as line above)
         1. THUS these two terms cancel and you end up with 2 \* the #bits changed from 0 to 1 shown below)
   4. Thus the amortized cost of the ith INCREMENT is: ai = ci + φi - φi-1 = 2 \* (#bits changes from 0 to 1)
   5. The INCREMENT changes only one bit from 0 to 1, hence the amortized cost of INCREMENT is 2

FIRST EXAM ONLY COVERS ABOVE MATERIAL

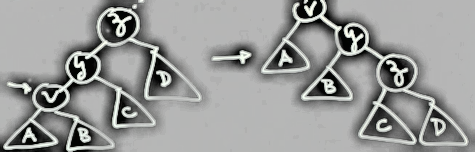
Module 2: Splay Trees = binary search tree (not balanced so height of tree may be > log n) where a target node is the root node – acts as a cache where last = root – beauty of a splay tree is that even though it is **not necessarily balanced**, its average performance is same as a balanced tree w/o overhead of doing balancing

1. Dynamic Binary Search Trees and Balanced Search Trees
   1. Every internal node has exactly 2 children – value of all nodes in left subtree < n and all in right subtree > n
   2. Depth d(v) = distance from root to node v – she counts number of edges on the path
   3. Height h(v) = distance from v to farthest leaf in the subtree rooted at v – count number of edges
   4. Height of the tree is the height of the root – count the number of edges
   5. Size |v| of v is the number of nodes in the subtree rooted at v – count all nodes so **inclusive**
   6. N = size of the whole tree
   7. **Minimum** height of any binary tree is [log n] + 1 – depends on size of n is why 1 added
   8. Balanced search tree: tree of height O(log n) that support search, insertion and deletion in O(log n) worst-case time
2. Rotations – used by a balanced search tree to maintain tree balance – can be performed in constant time
   1. Single rotation adjusts the shape of tree locally aka a Zag
   2. Rotation as node x decreases its depth by one and increases its parent’ depth by one
3. Double Rotations – two kinds:
   1. Roller-Coaster (aka Zig-Zig) consists of a rotation at x’s parent followed by a rotation at x, both in the same direction
      1. Used when x is a left child of a left child which uses two single rotations right – STRAIGHT PATH, or
      2. when x is a right child of a right child which uses two single rotations left– STRAIGHT PATH
      3. for either. FIRST ROTATION is at the parent of x then the second rotation is at x itself
   2. Zig-Zag consists of two rotations at x, in opposite directions
      1. Used when x is a right child of a left child, or
         1. Uses two single rotations: first left at the parent of x, then right at x itself
      2. when x is a left child of a right child
         1. Uses two single rotations: first right at parent of x then left at x itself
   3. for either. FIRST ROTATION is at the parent of x then the second rotation is at x itself
   4. **Each double rotation decreases the depth of x by two**, leaves the depth of its parent unchanged, and increases the depth of its grandparent by either one or two, depending on the type of double rotation
   5. **Either type of double rotation can be performed in constant time** as any rotation can be done in constant
4. Splaying = moving an arbitrary node in the tree up to the root through a series of double rotations, possibly with one single rotation at the end – three ways to initiate: **NOTE: total cost of splay tree operations = O(log n)**
   1. Use standard BST search to find node containing key or its predecessor if key is not present
      1. Splay whichever node is found
   2. Insert a new node using the standard BST insert algorithm then splay that node
   3. Delete – find a node to be deleted then splay that node and delete it
      1. This splits the tree into 2 subtrees: one with keys ≤ x and the other with keys ≥ x
      2. Find node w in left subtree with largest key, i.e. the predecessor of x in the original tree, splay it and finally join it to the right subtree
   4. Splaying a node v requires time proportional to d(v) which is the depth before splaying
   5. Every time you do any operation: search, insert, delete – you splay the tree:
      1. Search: use standard BST search to find node or its predecessor if node not present and splay whatever node is found
      2. Insert: insert node using standard BST insert algorithm, then splay that node
      3. Delete: find node to be deleted, splay it then delete it from root, which splits tree into 2 subtrees
         1. Find node w in left subtree with largest key, i.e. the predecessor of x in the original tree, splay it since it will not have a right child join it to the right subtree with node w as the root of new tree, OR
         2. Find node w in right subtree with largest key, i.e. the successor of x in the original tree, splay it since it will not have a left child join it to the left subtree with node w as the root of new tree
   6. Each search, insertion or deletion consists of constant number of operations of the form: “walk down to a node, and then splay it up to the root.”
      1. Since the walk down is cheaper than the splay all we need to get good amortized bounds for splay trees is derive good amortized bounds for a single splay
      2. We will use the potential method. The **rank of any node v is defined as r(v) = └log |v|┘ which is the log of the number of nodes in the subtree rooted at v inclusive**. We define the potential of a splay tree T at time t to be the sum of the ranks of all its nodes:



* + 1. r(v) = rank of v before a single or double rotation recall r(v) = └log |v|┘ before
    2. r’(v) = rank of v after the rotation is performed recall r’(v) = └log |v|┘ after
    3. Lemma: the amortized cost of a single rotation at v is at most 1 + 3r’(v) - 3r(v) and the amortized cost of a double rotation is at most 3r’(v) – 3r(v)
    4. By adding up the amortized costs of all the rotations we find that the total amortized cost of splaying a node v is at most 1 + 3rfinal(v) - 3rstart(v)
    5. After the splay v becomes the root hence rfinal(v) = └log n┘
    6. Amortized cost of a splay is at most 3 log n + 1 = O(log n) (took floor out as this is worst bound)
    7. Thus every insertion, deletion, or search in a splay tree takes amortized time O(log n) which is optimal

1. Amortization Analysis of Splay Rotations – start with lemma above (see pdf for clean version of below)
   1. amortized cost of a single rotation at v is at most 1 + 3rfinal(v) - 3rstart(v) => 3[r’(v) – r(v)] + 1
   2. amortized cost of a double rotation is at most 3rfinal(v) - 3rstart(v) => 3[r’(v) – r(v)]
   3. Corollary: **Total cost of splay operation = O(log n)** => => 3[rfinalsplay(v) - rstartaplay(v)] + 1
   4. Pick all possible rotations (Roller Coaster/Zig-Zig and Zig-Zag) and single rotation
      1. Let s and r denote the size, rank before a rotation – **rank =** **log of the # of nodes in subtree rooted at v**
      2. Let s’ and r’ denote the size, rank after a rotation – **size = # of modes in subtree rooted at v**
      3. **Case 1: single Rotation**
         1. φ(Tt) = difference in **potential of data structure** which = sum of rank of nodes which = log of the size of the tree under the node, *i.e*. log of the number of nodes in the subtree rooted at that node
         2. v = node to be splayed y = root node to be replaced by v via final rotation c = actual cost
         3. at = ct + φ(Tt) – φ(Tt-1) = 1 + r’(v) + r’(y) – r(v) – r(y) ≤ 1 + r’(v) – r(v) ≤ 1 + 3[r’(v) – r(v)]
            1. potential/rank of nodes in subtrees before and after is same so cancel so actual cost left
      4. **Case 2: Roller Coaster/Zig-Zig** – way she draws tree has y as v’s parent and z as v’s grandparent



* + - 1. at = 2 + [r’(v) + r’(y) + r’(z)] + [r(v) + r(y) + r(z)] NOTE: r’(v) = r(z) so you can cancel terms
      2. ≤ 2 + [r’(y) + r’(z)] – [r(v) + r(y)] => 2 + r’(y) + r’(z) – r(v) - r(y) after distributing – sign
      3. NOTE: r’(v) ≥ r’(y) and r(y) ≥ r(v) so ≤ 2 + r’(v) + r’(z) – 2r(v) ≤ (\*)
         1. (\*) = **assumption that 2r’(v) – r(v) – r’(z) ≥ 2** because

The fact that **if a and b > 0 and a + b ≤ 1 then [log(a) + log(b)] ≤ -2**

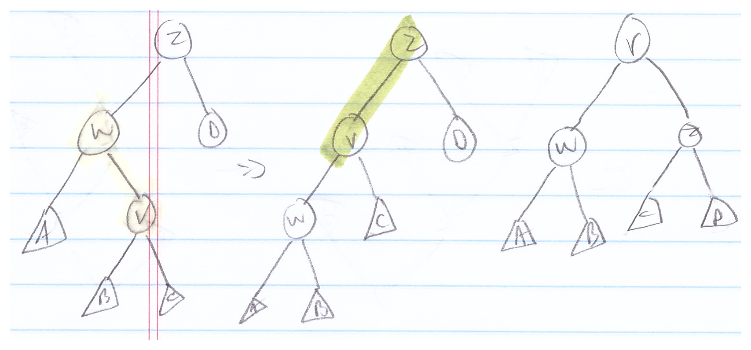
Using rank definition: 2 log s’(v) – log s(v) – log s’(z) NOTE: log(a) – log(b) = log(a/b)

Rewrite negation distributing -2 log s’(v): log [s(v)/s’(v)] + log [s’(z)/s’(v)]

NOTE: s(v) + s’(z) ≤ s’(v) which shows that both logs above log + log ≤ 1 which means that log [s(v)/s’(v)] = to a and log [s’(z)/s’(v)] = b in the **mid-assumption formula**

Therefore: log [s(v)/s’(v)] + log [s’(z)/s’(v)] ≤ -2 which is inverse of **original assumption**

* + - 1. 2 + r’(v) + r’(z) – 2r(v) ≤ 2r’(v) – r(v) – r’(z) + r’(v) + r’(z) – 2r(v) can cancel terms and combine
      2. ≤ 3r’(v) – 3r(v) which corresponds to the bounds we had for double rotation: 3[r’(v) – r(v)]
    1. **Case 3: Zig-Zag** – all ranks cancel but for the target node and its parent and grandparent



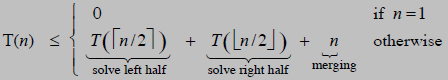
* + - 1. at = 2 + r’(v) + r’(w) + r’(z) – r(v) – r(w) – r(z) NOTE: r’(v) = r(z) and r(w) ≥ r(v) so cancel/combine
      2. ≤ 2 + r’(w) + r’(z) – 2r(v) ≤ just like case 2 we can show that 2r’(v) – r’(w) – r’(z) ≥ 2 so substitute
      3. ≤ 2r’(v) – r’(w) – r’(z) + r’(w) + r’(z) – 2r(v) which means we can cancel so
      4. = 2r’(v) – 2r(v) ≤ 3r’(v) - 3r(v) which matches the lemma 3[r’(v) – r(v)]

====================

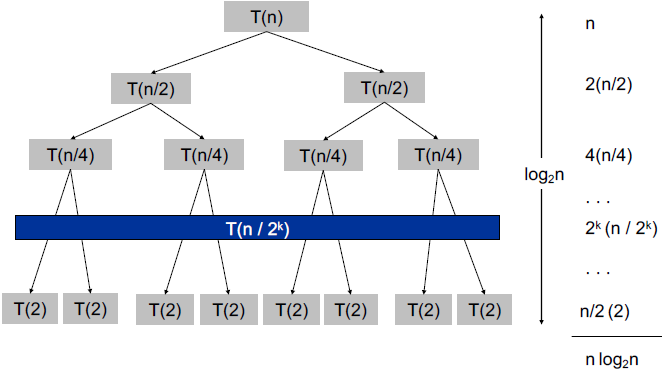
**Unit 4: Divide and Conquer**

Module 1: General Techniques

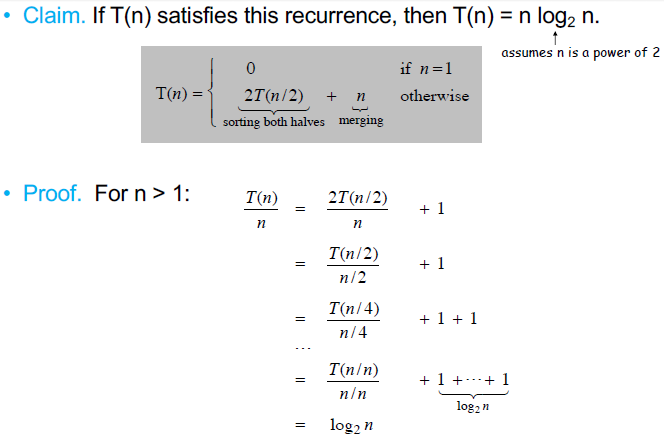
1. Divide in Conquer = break problem into non-overlapping sub problems (compare to dynamic programming where the sub-problems DO overlap) then combine solutions into answer to original
   1. Can bring asymptotic behavior down from O(n2) to O(n log n) – sorting is an example – mergesort
   2. Problems become easier once sorted
      1. Find the median
      2. Find the closest pair
      3. Binary search in a database
      4. Identify statistical outliers
      5. Find duplicates in a mailing list
   3. Obvious sorting applications:
      1. List files in a directory
      2. Organize an MP3 library
      3. List names in a phone book
      4. Display Google PageRank results
   4. Non-Obvious sorting applications:
      1. Data Compression
      2. Computer graphics
      3. Interval scheduling
      4. Computational biology
      5. Minimum spanning tree
      6. Supply chain management
      7. Simulate system of particles
      8. Book recommendations on Amazon
      9. Load balancing on a parallel computer
   5. Mergesort
      1. Divide = O(1)
      2. Sort = 2T(n/2) T(n) = running time of algorithm for input size n
      3. Merge = O(n)
         1. Traverse both sub- arrays from lowest index-to-highest index comparing values of next available elements in the two arrays the choosing/inserting the lowest value to insert in the output array
         2. Recurrence relation: divide step does not cost anything so not shown – merging = max n comparisons – floor and ceiling are there to cover the case that n is not a power of 2:

 T(n) = O(n log2 n)

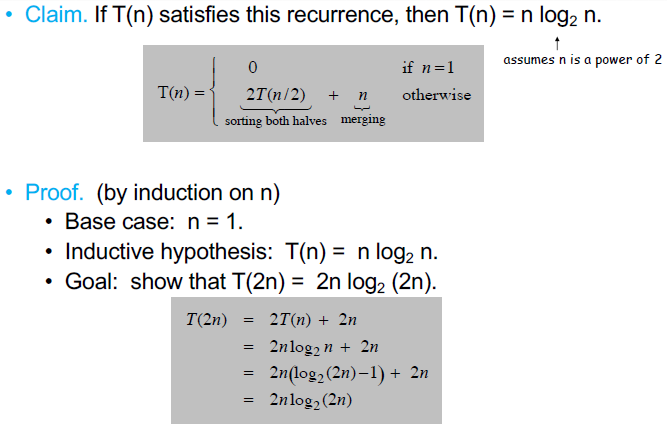
Proof by Recursion Tree: she says that you only have to divide until you get arrays of 2 items to compare



Proof by Telescoping: [T(n/n)/n/n] = 0 so it is the 1 + 1 + . . . + 1 that gives the log2 n



Proof by Induction



**Module 2: Closest Pair of Points** = given n points in a plane, find pair w/smallest Euclidian distance between them

1. Applications: Graphics, computer vision, geographic information systems, molecular modeling, air traffic control
2. Brute Force = check all pairs of points p and q with n(n – 1) comparisons = n(n – 1) or Θ(n2) comparisons
3. 1-D Version = O(n log n) easy if points are on a line as can sort all points and go left-to-right on line
   1. Computing distances only takes linear time as points consecutive, but the sort takes n log n time
4. Can we match that in two dimensions?
   1. Assumption: no two points have same x coordinate – just to make presentation clearer – not needed
   2. Algorithm: first attempt – not great:
      1. sub-divide region into four quadrants
      2. impossible to ensure n/4 points in each piece and if not tough to find closest pair
5. Algorithm:
   1. Divide: draw vertical line L so that roughly n/2 points on each side – sweep across plane until passed n/2 points
   2. Conquer: find closest pair in each side recursively – these two points are CANDIDATES for the closest pair
   3. Combine: find closest pair with one point in each side <- seems like Θ(n2)
   4. Return best of three solutions and the minimum of the distances of the pairs on either side = δ as any close points near the division need to be closer than either of the closest points on either side in order to WIN the closest points competition
      1. assuming distance < δ
      2. Only need to consider points within δ of line L
      3. Sort points in 2δ strip by their y coordinate
      4. Only check the distances of those within next 11 positions of sorted list according to y coordinate!
         1. Why? Number points in 2δ strip by their y coordinates and divide up 2δ strip by (1/2)δ boxes so that you have 12 boxes in three rows of 4 with si in the bottom row
            1. There cannot be 2 points in any of these boxes because if there were they would be closer together than δ which would change the size of the 2δ strip
         2. Definition: let si be the point in the 2δ-strip with the ith smallest y-coordinate
            1. She starts at the lowest y value and always compares it to values above it to assure coverage
         3. Claim: if |i – j| ≥ 12 then distance between si and sj is at least δ
            1. i is the smallest point on left of divide and j is point on right. i is in lower row of 12 (1/2)δ boxes
            2. when she goes through pseudocode she says you only need to compare low y to 11 other points according to the sorted order via y coordinate
      5. Proof:
         1. No two points lie in same 1/2δ-by-1/2δ box because if they were as far apart as possible in the same box they would be on a diagonal and using Pythagoras they would be δ/√2 < δ and this is not possible because each box is on opposite sides of the dividing line and we already found the closest pair on either side of the dividing line and this new pair could not be closer than that previously found pair
         2. Two points at least 2 rows apart have distance ≥ 2(1/2δ) because each row is (1/2)δ tall so 2\*(1/2)δ = δ and we already found a point that is δ so two rows apart does not help us
         3. Since we have 12 boxes and there is only one other point in each box and si is in one of the boxes we only have to make up to 11 comparisons to other points
         4. Fact: still true if we replace 12 with 7 because some of those 11 points are on same side of the dividing line so not helpful and some boxes too far away so can tighten this up to 7 boxes instead of 12)
6. Psuedocode:

**Closest-Pair(p1, …, pn) {**

**If n ≤ 1 return ∞**

**Compute separation line L such that half the points // O(n log n) to sort by x coordinates // O(n) if presorting**

**are on one side and half on the other side.**

**δ 1 = Closest-Pair(left half)**

**δ 2 = Closest-Pair(right half) // 2T(n/2)**

**δ = min(δ 1, δ 2)**

**Let S be the set of points at distance at // can use the already sorted by x coordinates from above**

**most δ from separation line L. // but then she says you could just “sweep” all to find**

**Sort S by y-coordinate. // O(n log n) // O(n) if presorting**

**Scan points in y-order and compare distance**

**between each point and next 11 neighbors. // O(n) comes from 11n comparisons**

**If any of these distances is less than δ, update δ.**

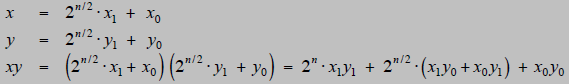
**return δ.**

**}**

1. Running Time: T(n) ≤ 2T(n/2) + O(n log n) => T(n) = O(n log2 n)
   1. Can we achieve O(n log n)?
      1. Yes. Always keep two sorted lists of a set of points P, P\_x and P\_y, sorted by x- and y-coordinates, respectively, *i.e*. do one single sort but save two results: one as to x and one as to y
      2. Before each recursive call, precompute R\_x, R\_y, L\_x, and L\_y (where L and R are the points to the left and right of the separation line, respectively) in O(n) time. How?
      3. Also compute S\_y in O(n) time. How?

**Module 3:** Integer Multiplication and Matrix Multiplication

1. She reviews elementary-school-style binary multiplication shifting over for each multiplicand digit
   1. She divides her 2 binary numbers up and shifts/multiplies the higher position bits and get this:



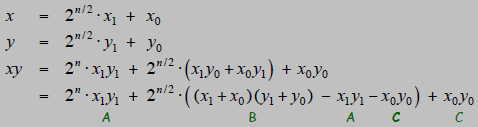
This is 4 multiplications but each only involves n/2 bit numbers

Then assuming n is a power of 2:

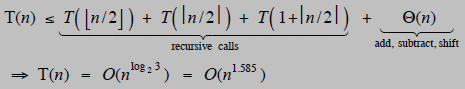


NOTE: this is Master Method Case 1 in that log24 = 2 > the exponent on n in Θ(n1) so result is Θ(nlog24) = Θ(n2)

1. Karatsuba Multiplication: to multiply two n-digit integers:
   1. Add two n/2-digit integers
   2. Multiply three n/2-digit integers
   3. Add, subtract, and shift n/2-digit integers to obtain result

The last line reduces the # of mults on n/2d to 3 and the A and C multiplications used to do the reduction are already done in other parts of the formula so really just ABC

* 1. Theorem [Karatsuba-Ofman, 1962] – can multiply two n-digit integers in Θ(n1.585) bit operations



* 1. Karastuba recursion tree:

1. From YouTube video – description of Karatsuba’s Algorithm using 1234 \* 5678 where a = 56, b = 78, c = 12, d = 34
   1. Compute: a \* c = 56 \* 12 = 672
   2. Compute: b \* d = 78 \* 34 = 2652
   3. Compute: (a + b)(c + d) = (56 + 78)(12 + 34) = 134 \* 46 = 6164
   4. Compute: ③ - ② - ① = 6164 – 2652 – 672 = 2840
   5. Pad ① with four 0s 6720000
   6. Pad ② with no 0s 2652
   7. Pad ④ with two 0s 284000
   8. Total all three 7006652
2. Matrix Multiplication: divide-and-conquer
   1. From YouTube: naïve algorithm involves three loops and that is where O(n3) comes from

**void multiply(int A[][N], intB[][N], int C[][N]) {**

**for (int i = 0; i < N; i++){**

**for (int j = 0; j < N; j++){**

**C[i][j] = 0;**

**for (int k = 0; k < N; k++){**

**C[i][j] += A[i][k] \* B[k][j];**

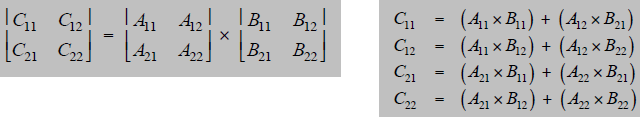
**}**

**}**

**}**

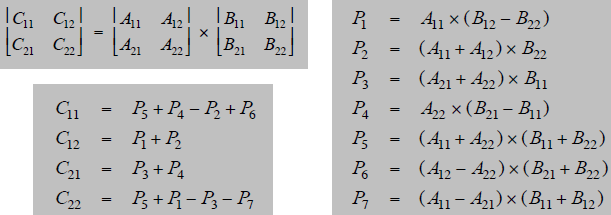
**}**

* 1. Non-Strassen Divide: partition A and B into n/2 x n/2 blocks
     1. NOTE that each of these n/2 sub-matrices are 25% of the original matrix
  2. Non-Strassen Conquer: perform 8 (n/2 x n/2) matrix multiplications recursively so T(n) = 8T(n/2) + O(n2) = O(n3)
  3. Non-Strassen Combine: add appropriate products using 4 matrix additions



Not any better than naïve method!

* 1. Key idea: multiply 2 x 2 block matrices with only 7 multiplications instead of 8

 See e)ii) below

From YouTube a more graphical depiction:

* + 1. 7 multiplications total instead of 8 so we get T(n) = 7T(n/2) + O(n2) = O(nlogba) = O(nlog27) = O(n2.8074)
    2. 18 = 10 + 8 additions or subtractions
  1. Fast Matrix Multiplication (Strassen 1969)
     1. Divide: partition A and B into n/2 x n/2 blocks
     2. Compute: 14 n/2 x n/2 matrices via 10 matrix additions (14 comes from finding B12 – B22 and A11 + A12 shown in above diagram before making recursive calls)
     3. Conquer: perform 7 n/2 x n/2 matrix multiplications recursively (7 recursive calls result in 7 products)
     4. Combine: 7 products into 4 terms using 8 matrix additions
  2. Analysis:
     1. Assume n is a power of 2
     2. T(n) = # arithmetic operations



* 1. Fast Matrix Multiplication in Practice:
     1. Implementation issues:
        1. Sparsity; caching effects; Numerical stability; Odd matrix dimensions; crossover to classical algorithm around n = 128
     2. Controversies:
        1. Is Strassen only a theoretical curiosity?
        2. Advanced Computation Group at Apple reports 8x speedup on G4 Velocity Engine when n = 2,500
        3. Range of instances where it is useful is subject of controversy
     3. Remark
        1. Can “Strassenize” Ax=b, determinant, eigenvalues, other matrix ops
  2. Fast Matrix Multiplication in Theory
     1. Multiply two 2x2 matrices w/only 7 scalar multiplications? Yes via Strassen 1969 - Θ(nlog27) = Θ(n2.81)
     2. Multiply two 2x2 matrices w/only 6 scalar multiplications? No via Hopcroft and Kerr 1971 - Θ(nlog26) = Θ(n2.59)
     3. Multiply two 3x4 matrices with only 21 scalar multiplications? No - Θ(nlog321) = Θ(n2.77)
     4. Multiply two 70x70 matrices w/only 143,540 scalar mults? Yes via Pan 1980 - Θ(nlog70143640) = Θ(n2.80)
     5. Decimal Wars: December 1979 = O(n2.521813) January 1980 = O(n2.521801)
     6. Best known - O(n2.376) by Coppersmith-Winograd 1987
     7. Conjecture - O(n2+ɛ)
     8. Caveat – theoretical improvements to Strassen are progressively less practical

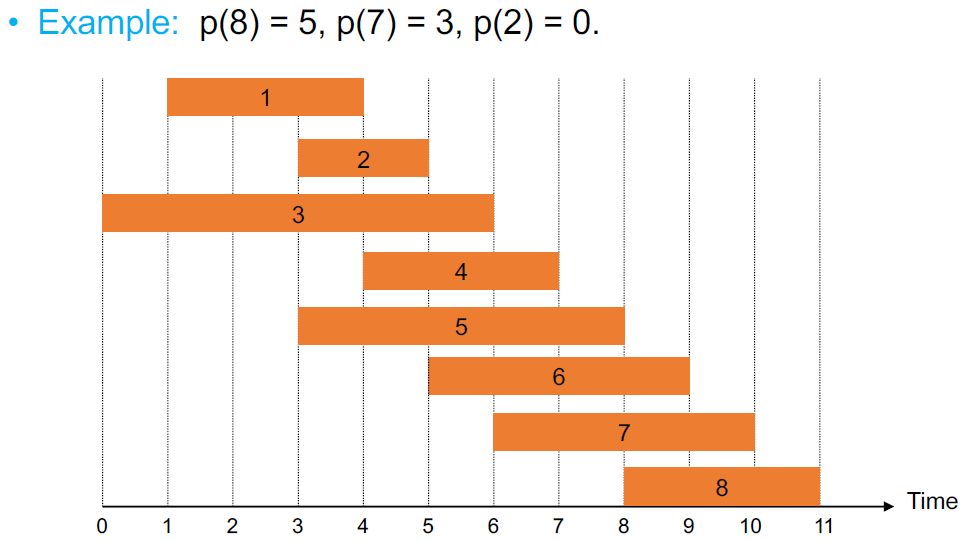
====================

====================

**Unit 5: Dynamic Programming**

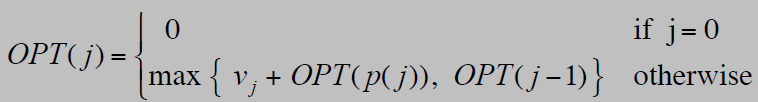
Module 1: General Techniques

1. Algorithmic Paradigms
   1. Greedy = build up a solution incrementally, myopically optimizing some local criterion
      1. Works for weighted if all the job weights are 1, but can fail if arbitrary weights are allowed
   2. Divide-and-Conquer = break a problem up into disjoint sub-problems to form a solution to the original problem
   3. Dynamic Programming = break a problem up into a series of overlapping sub-problems and build up solutions to larger and larger sub-problems
2. Dynamic Programming Applications: bioinformatics, operations research, control theory, info theory, graphics, AI
   1. Famous dynamic programming algorithms
      1. Viterbi for hidden Markov models
      2. Unix diff for comparing two files
      3. Smith-Waterman for sequence alignment
      4. Bellman-Ford for shortest path routing in networks
      5. Cocke-Kasimi-Younger for parsing context free grammars
3. Weighted Interval Scheduling: goal = find max weight subset of mutually compatible jobs
   1. Label job by sorted finishing time: f1 ≤ f2 ≤ . . . ≤ fn
   2. Define p(j) = largest index i < j such that job i is compatible with job j



NOTE that all jobs above the p value are compatible due to the jobs being sorted

1. Dynamic Programming: Binary Choice
   1. Define OPT(j) = value of optimal solution to problem consisting of job requests 1, 2, . . . , j
      1. CASE 1: OPT selects job j
         1. Cannot use incompatible jobs { p(j) + 1, p(j) + 2, . . . , j – 1 }
         2. Must include optimal solution to the problem consisting of remaining compatible jobs 1, 2, . . . , p(j)
      2. CASE 2: OPT does not select job j
         1. Must include optimal solution to the problem consisting of remaining compatible jobs 1, 2, . . . , j – 1



Module 2: Weighted Interval Scheduling

1. Weighted Interval Scheduling Brute Force – **pseudocode: takes(number, start, finish, weight) first Sort jobs by finish // sort take O(n log n)**

**Compute p(1), p(2), . . . , p(n) // the job w/latest but previous finishing time that does not overlap**

**Compute-Opt(n) // she says this is main call to function**

**Compute-Opt(j){**

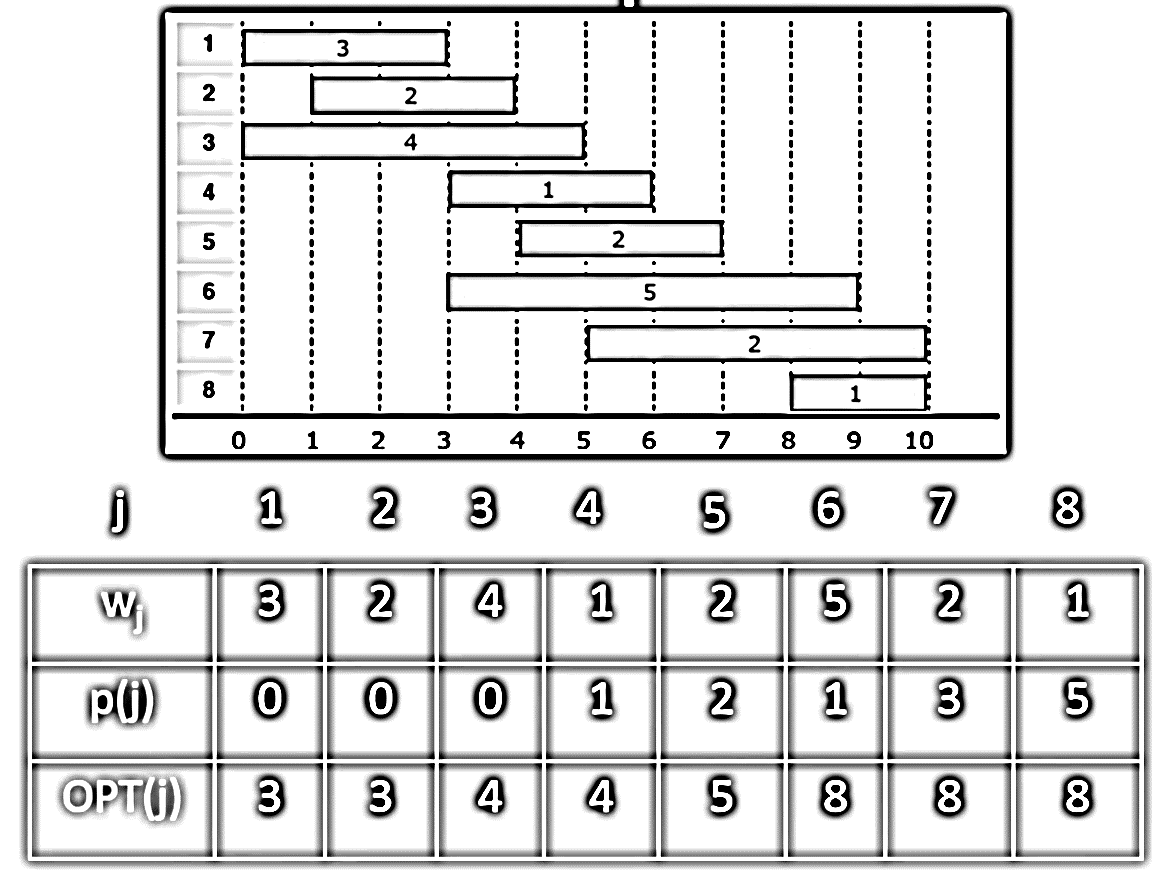
**if(j == 0) // base case for recursion**

**return 0**

**else // compare [weight + p(j) with p(j – 1)] to find Opt(j), e.g. below when**

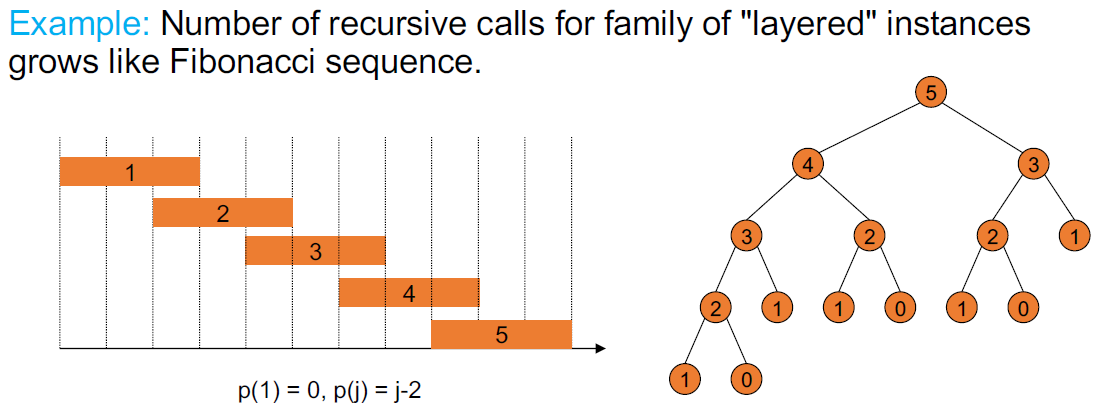
**Return max(wj + Compute-Opt(p(j)), Compute-Opt(j – 1) // j = 2 weight + p(j) = 2 but p(j – 1) = 3 so 3**

**How it works:** wj = weight then Compute-Opt(p(j)) = the table entry for OPT(j) for the p(j) value so for example when j = 5, w5 + Compute-Opt(p(5)) = 2 + {the value of OPT[p(j)]} and here p(j) = 2 so the OPT(2) was 3 so ultimately you add 2 to 3 = 5 by referring back to OPT(1) value and adding that to w5

Once Opt(j) does not grow pick first value – here 6

From: <https://www.youtube.com/watch?v=iIX1YvbLbvc> who says O(n log n) if you use clever binary search but she says you need Memoization in order to get to n log n

* 1. Observation: recursive algorithm fails due to redundant sub-problems => exponential algorithms

Note 3-2-1 gets calculated twice so dynamic candidate

1. Weighted Interval Scheduling Memoization – **pseudocode: takes(number, start, finish, weight) first Sort jobs by finish // sorting take O(n log n) time**

**Compute p(1), p(2), . . . , p(n) // the job w/latest but previous finishing time that does not overlap**

**// computing p requires O(n) time**

**For j = 1 to n { // Difference from brute force is this MATRIX of values that are referred**

**M[j] = empty } // to instead of recalculating the same OPT over and over**

**M[0] = 0 }**

**Compute-Opt(n) // she says this is main call to function**

**Compute-Opt(j){ // Computng OPT(j) takes O(1) time**

**if(M[j} is empty) // if we have not previously solved the OPT before then do so**

**M[j] = max(vj + M-Compute-Opt(p(j)), M-Compute-Opt(j – 1))**

**Return M[j] // if already solved return old solution otherwise return new solution**

**} // overall run time of M-Compute-Opt(n) is O(n)**

1. Weighted Interval Scheduling Running Time = O(n log n) or if jobs are presorted O(n)
   1. M-Compute\_Opt(j) returns existing value M(j) or fills in one new entry M(j) and makes two recursive calls
   2. Progress measure ɸ = # nonempty entries of M[ ]
      1. Initially ɸ = 0, throughout ɸ ≤n
      2. The call to fill in an entry increases ɸ by 1 => at most 2n recursive calls total so M-Compute-Opt(n) = O(n)
      3. She shows presorted example of sort including both start and finish times which allows linear search
2. Automated Memoization – LISP has facility built in but Java and other popular languages do not

Module 2: Weighted Interval Scheduling

1. Weighted Interval Scheduling: Finding a Solution – dynamic programming algorithms compute optimal final value, but do not give you the solution itself. Here is routine for pulling actual solution out – **pseudocode**

**Run M-Computer-Opt(n)**

**Run Find-Solution(n)**

**Find-Solution(j) {**

**if ( j == 0 )**

**Output nothing**

**else if ( vj + M[p(j)] > M[j – 1]**

**Print j**

**Find solution(p(j))**

**else**

**Find-Solution(j – 1)**

**}**

1. Weighted Interval Scheduling Bottom-Up – replaces recursion with for loop – pseudocode takes same input

**Sort jobs by finish // bottom-up = recurses to base case (leaf) then works way up (tree)**

**Compute p(1), p(2), . . . , p(n)**

**Iterative\_Compute-Opt {**

**M[0] = 0**

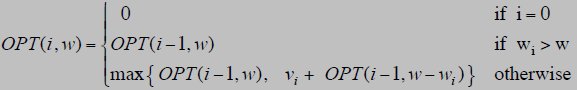
**For j = 1 to n**

**M[j] = max vj + M-Compute-Opt(p(j)), M-Compute-Opt(j – 1))**

**}**

**Module 3: Knapsack**

1. Knapsack Problem
   1. Given n objects and a knapsack
   2. Item i weighs wi > 0 kilograms and has a value vi > o
   3. Knapsack has capacity of W kilograms
   4. Goal: fill knapsack so as to maximize total value
   5. Greedy algorithm is NOT optimal- same with using value-to-weight ratio
2. Dynamic Programming:
   1. False Start: Define OPT(i) = max profit subset of items 1, . . . , i
      1. Case 1: OPT does not select item i, but rather selects best of { 1, 2, . . . , i-1}
      2. Case 2: OPT selects i which does not imply rejecting other items
         1. Without knowing what other problems were selected before item i, we do not know if there is enough room for item i
      3. Conclusion is we need more sub-problems
   2. Adding a new variable: let OPT(i, w) = max profit subset of items I, . . . , i with weight limit w
      1. Case 1: OPT does not select item i, but rather selects best of { 1, 2, . . . , i-1} using weight limit w
      2. Case 2: OPT selects item i
         1. New weight limit = w – wi
         2. OPT selects best of { 1, 2, . . . , i – 1 } using new weight limit



1. Knapsack Problem: Bottom-Up – can be considered as filling up an n-by-W array using following **pseudocode**

**Input: n, W, wi, . . . , wn, vi, . . . , vn  // bottom-up = recurses to base case (leaf) then works way up (tree)**

**for w = 0 to W // O(W) where w = weight count and W = weight maximum**

**M[0, w] = 0**

**for i = 1 to n // i stands for item number // O(n W) for nested**

**for w = 1 to W // iterates 1-to-total not via weights of items**

**if (wi > w) // compare item weight to weight count// O(1)**

**M[i, w] = M[i – 1, w] // M[i – 1, w] = value above // O(1)**

**Else**

**M[i, w] = max {M[i – 1, w], vi + M[i – 1, w – wi]} // w iteration count - wi item weight //up to 2 x O(1) so O(1)**

**return M[n, W] // then you look up value at M[i -1, w iteration count - wi item weight]**

1. Knapsack Algorithm
   1. From YouTube: <https://www.youtube.com/watch?v=CUAolXf8u-U>

if i or w == 0 which means if weight of current i == running total weight, then M[i, w] // diff than above

if wi > w then M[i, w] = M[i-1, w}

else M[i, w] = max {M[i – 1, w], vi + M[i – 1, w – wi]} // w – wi = move left running weight count – weight of i

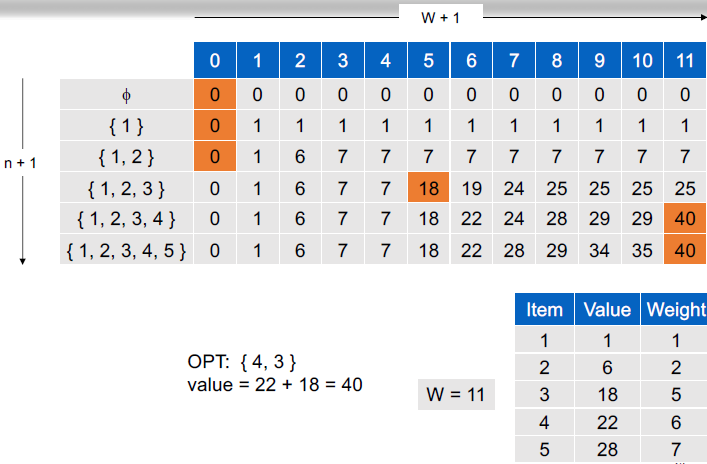
// so you go up 1 level (i – 1) then left w – wi

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | 0 base -> Weight = W = 6 | | | | | | |
| i |  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| wt | val | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 3 | 0 | 1 | 3 | 4 | 4 | 4 | 4 |
| 3 | 4 | 0 | 1 | 3 | 4 | 5 | 7 | 8 |
| 4 | 6 | 0 | 1 | 3 | 4 | 6 | 7 | 9 |

NOTE: in W = 3 and wt = 2 the term vi + M[i – 1, w – wi] => 3 + M[1, 3-2] as w = 3 and wi = 2 so add M[1, 1] = 1

NOTE: in W = 1-2 and wt = 3 the weight of i = 3 so w is less than wi so we just bring M[i-1, w] down for value

* 1. From Slides:



1. Running Time = Θ(n W)
   1. Not polynomial on input size! **Psuedo-polynomial means dependent on the numeric value which is also the length of input in binary** – looks like only primality testing and knapsack are pseudo-polynomial
      1. An algorithm whose worst-case time complexity depends on numeric value of input (**not number of inputs**) is called Pseudo-polynomial algorithm.
      2. E.G. primality testing with this simple algorithm. It looks like O(n) but is really O(2logn) due the length of the inputs because the length of the input in bits is going to grow exponentially:

**for (i = 2; i < n – 1; i++) {**

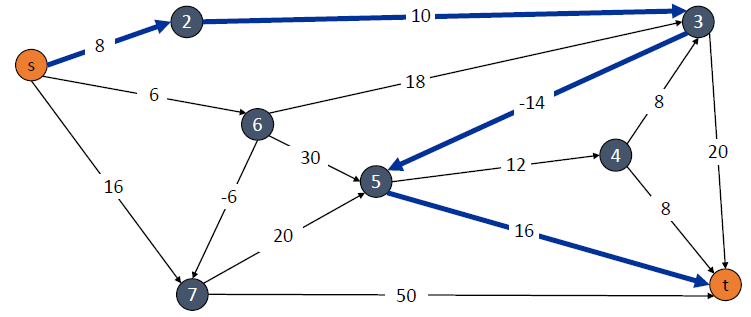
**if n % i == 0 return false**

**} return true**

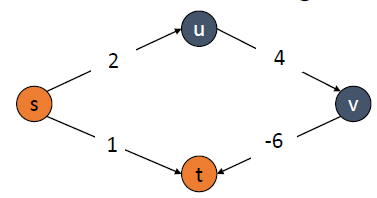
* 1. “Pseudo-polynomial” because max weight W in O(n W) = O(2logW n) is represented by 2logW in bits in binary
     1. W is exponential in the size of W - 2logW = W also Wlog2 = W
  2. Decision version of Knapsack is NP-complete as there is no polynomial algorithm to solve, HOWEVER
     1. If the capacity of the knapsack can be shown to be a constant or multiple of 2 then polynomial available as asymptotically the running times depends on n not W
  3. There exists a knapsack approximation algorithm that produces a feasible solution that has a value within 0.01% of optimum

**Module 4: Shortest Path: Bellman-Ford**

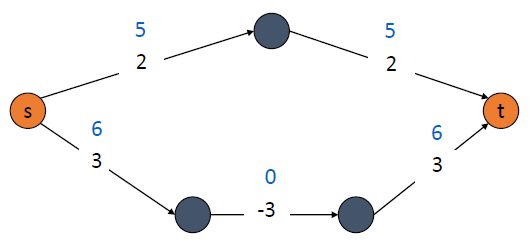
1. Shortest path problem = given a directed graph G = (V, E) w/edge weights cvw find the shortest path from node s to t
2. Example: in following graph, nodes are agents in a financial setting and cvw is transaction cost when we buy from agent v and sell to w. The total minimum cost from node s to t = 20:

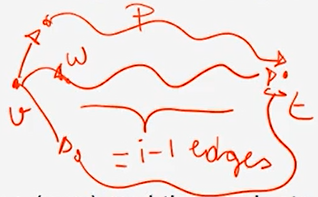


1. Dijkstra’s can fail if there are negative edge costs if it never gets to the negative edge, e.g. here picking s-t not s-u



1. Re-Weighting = adding a constant to every edge. This can change the solution because if there are more edges on the path then more weight will be added to the path than a path with less edges: here 2x v. 3x added weight



1. Negative Cost Cycles: if some path from s to t contains a negative cost cycle there does not exist a shortest s-t path; otherwise, there exists one that is simple
2. Shortest Path – Dynamic Programming: find the path from every node v to the destination node t instead of from the start node to every other node like Dijkstra
   1. OPT(i, v) is the sub-problem that gives shortest v-t path using at most i edges
   2. Case 1: P uses at most i-1 edges thus OPT(i, v) = OPT(i-1, v)
      1. A path with i edges is simply a path with i-1 edges plus one edge. Call that one more edge w
   3. Case 2: P uses exactly i edges thus if (v, w) is the first edge then OPT uses (v, w) and then selects best w-t path using at most i-1 edges

cvw = cost of one edge

|Case 1 |Case 2 – (v,w) ϵ E means try every possible see diagram above|

|try every node adjacent to v as candidate for node w which requires many i – 1

|this second case catches the cases when you are using exactly i edges

* 1. Remark: by previous observation, if there are no negative cycles then OPT(n-1, v) = length of shortest v-t path because if there is a positive weight cycle it will simply not be taken, i.e. simple path dominates path w/cycle
     1. Assuming no negative cycles, once we get to an i that is = to n – 1 we know we have found a shortest path

**Pseudocode for bottom-up: // Prof says come up w/memorized version for practice – see slides 12 and 16**

**Shortest-Path(G, t) { // bottom-up = recurses to base case (leaf) then works its way back up (tree)**

**foreach node v ϵ V**

**M[0, v] <- ∞** **// here she puts 0 in the entry for node t as there IS a path from t to t using 0**

**M[0, t] <- 0 // table M = space complexity n2 = n rows and n columns = can be large**

**for i = 1 to n-1 // look at sub-problems in order of increasing complexity = increasing # edges**

**foreach node v ϵ V // once # of edges fixed go through every node of the graph**

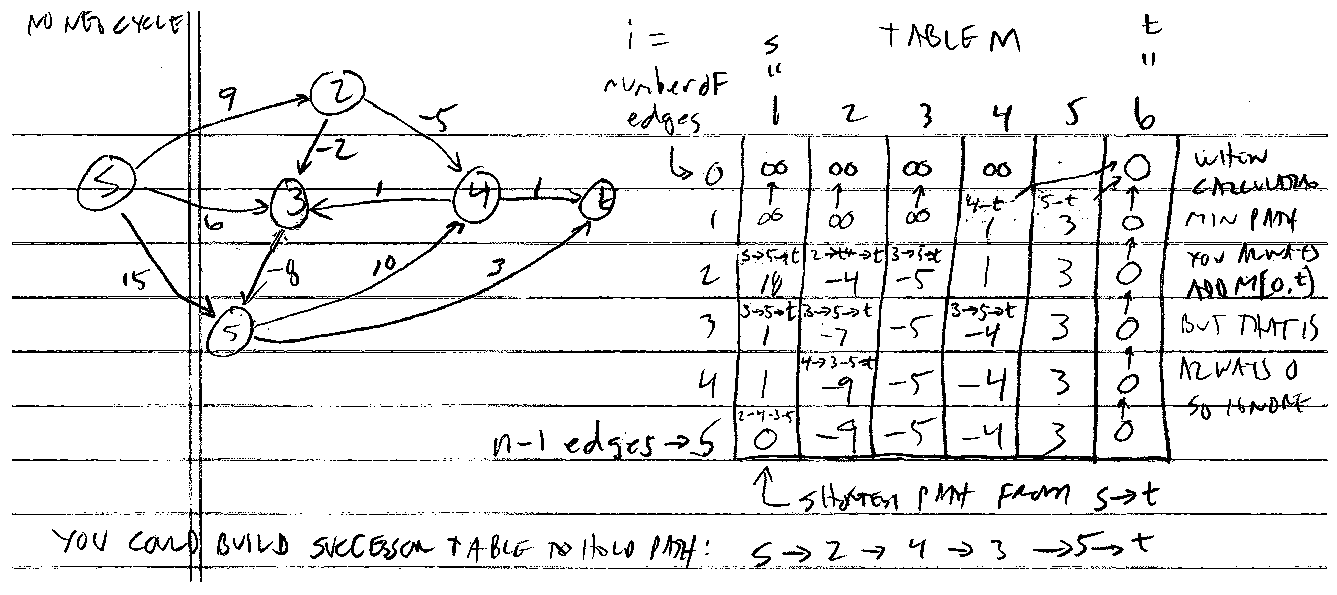
**M[i, v] <- M[i-1, v] // Case 1 – n2 iterations recall m = n – 1 = number of edges**

**foreach edge (v, w) ϵ E // not necessarily following the same order as recurrence relation**

**M[i, v] <- min {M[i, v], M[i-1, w] + cvw } // can Case 2 improve on what we had before (ellvw = cvw)**

**} // may not occur in an organized order – m \* n iterations**

**// M[s, n-1] = the shortest path**

****

* 1. This algorithm runs in Θ(mn) time (which dominates n2 time of nested for loops) and Θ(n2) space
     1. This gets closer to Θ(n2) as the graph is closer to complete, i.e. there are closer to maximum edges
     2. If you do not save the sub-problem solutions for reference it runs in Θ(n3)
     3. You can reduce space to Θ(n) from Θ(n2) by continually overwriting 1D array instead of having 2D array
  2. Finding shortest paths – maintain a successor[i, v] that points to next node on a shortest v-t path using at most i edges – successor table allows tracing back actual edges of path not just the total length of the shortest path
     1. Could also be derived from table M

1. Shortest Paths Practical Improvements
   1. Practical Improvements:
      1. Maintain only one array M[v] = shortest v-t path that we have found so far which is not in matrix M
      2. No need to check of the form (v, w) unless M[w] changed in previous iteration
   2. Theorem: throughout the algorithm, M[v] is length of some v-t path, and after i rounds of updates, the value M[v] is no larger than the length of shortest v-t path using ≤ i edges
      1. She explains that in the line labeled Case 1 in pseudocode why do we need to write M[i – 1, v] into M[i, v] for all then overwrite Case 2 on to same?
      2. She goes further to say that if you removed index i or i – 1 from all M[i, v] and M[i – 1, v] in pseudocode the algorithm would work exactly the same – NOTE this makes the Case 1 foreach redundant
   3. Overall impact:
      1. Memory: O(m + n)
      2. Running time: O(mn) worst case, but substantially faster in practice
2. Bellman-Ford Efficient Implementation – **pseudocode (simply uses single array instead of table)**

**Push-Based-Shortest-Path(G, s, t) {**

**foreach node v ϵ V {**

**M[v] <- ∞ // array M[v] records shortest path so far – algo implicitly keeps track of missing i from M[i, v]**

**successor[v] <- ɸ**

**} // note M[i, v] <- M[i-1, v] missing here as we do not mindlessly write next row to start**

**M[t] = 0 // you could delete all i in M[i, v] in base code and it would run fine**

**for i = 1 to n-1 { // which means first nested for loop missing here**

**foreach node w ϵ V {**

**if (M[w] has been updated in previous iteration) {**

**foreach node v such that (v, w) ϵ E {**

**if (M[v] > M[w] + cvw) {**

**M[v] <- M[w] + cvw**

**successor[v] <- w**

**} } }**

**If no M[w] value changed in iteration i, stop.**

**} }**

**FROM THE TEXTBOOK:**

**Asynchronous-Shortest-Path(*G*, *s*, *t*)**

***n* = number of nodes in *G***

**Array *M*[*V*]**

**Initialize *M*[*t*]= 0 and *M*[*v*]=∞ for all other *v* ∈ *V***

**Declare *t* to be active and all other nodes inactive**

**While there exists an active node**

**Choose an active node *w***

**For all edges *(v*, *w)* in any order**

***M*[*v*]= min*(M*[*v*], *cvw* + *M*[*w*]*)***

**If this changes the value of *M*[*v*], then**

***first*[*v*]= *w***

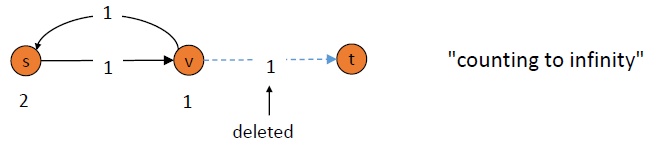
***v* becomes active**

**Endfor**

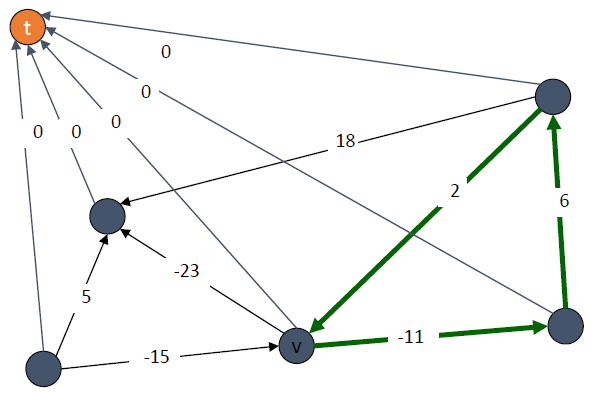
***w* becomes inactive**

**EndWhile**

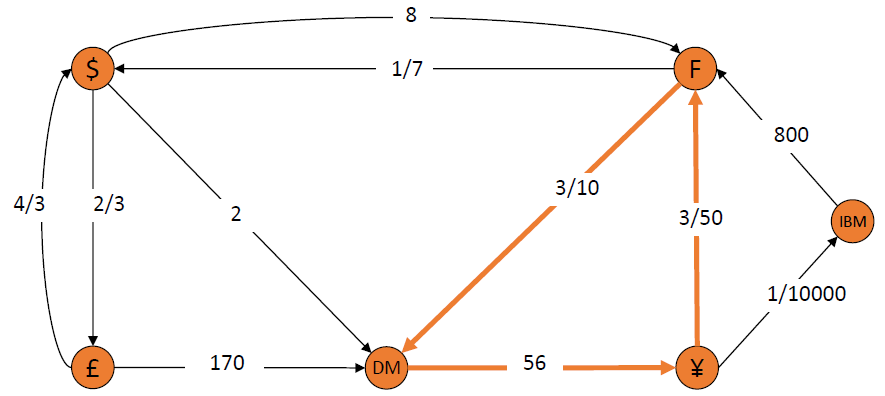
1. Distance Vector Protocol – still no negative weight edges as in network routing no neg weights
   1. We can apply the shortest path problem to routers in a communication network to determine the most efficient path to a destination
      1. Node represent routers
      2. Edges represent direct communication links
      3. Cost of an edge is the delay on the link
   2. We can use Dijkstra’s to solve this problem but it requires global knowledge of the network but no server has this broad of knowledge of the Internet
   3. Bellman-Ford uses only local knowledge of neighboring nodes
      1. We do not expect routers to run in lockstep. The order in which each foreach loop executes is not important. Moreover, algorithm still converges even if updates are asynchronous
   4. Routing by Rumor
      1. Each router maintains a vector of shortest path lengths to every other node (distances) and the first hop on each path (directions)
         1. she says you are actually running n – 1 Bellman-Ford algorithms in parallel here because every entry in vector here is going to be an estimate of the shortest path from v to a given destination t, i.e.
         2. Each router performs n separate computations, one for each potential destination node
         3. Bellman-Ford can continually run in the background and by its nature adapts as node w finds a more efficient path to destination t where Dijkstra’s greedy nature does not allow such flexibility
      2. Examples: Routing Information Protocol (RIP), Xerox XNS RIP, Novell IPX RIP, Cisco Interior Gateway Routing Protocol (IGRP), AppleTalk RTMP
   5. Caveats
      1. Edge costs may change during the algorithm
      2. If edge(v, t) below is deleted, Bellman-Ford will begin to count to infinity
         1. She explains that when v loses the v – t edge its asks its neighbors for help trying to get to t and it finds that the routing table of s says that s knows a 2-hop route (now out of date) to t and gets into a loop where based on s having a 2-hop route v mistakenly tells s it has a route with 3 hops and s updates its routing table and an endless cycle begins – protocols have mechanism to ID this issue and abort



1. Path Vector Protocols = a way to mitigate Distance Vector Protocol and its issues outlined above
   1. To avoid problems of Distance Vector Protocol network designers adopted Path Vector Protocol:
      1. Each router stores the entire path (not just distance and first hop)
      2. Based on Dijkstra’s
      3. Requires significantly more storage – wastes a lot of network resources, so only used when you have a lot of processing power and cannot afford to fail
   2. Examples using Path Vector Protocol:
      1. Border Gateway Protocol BGP)
      2. Open Shortest Path First (OSPF)
2. Negative Cycles in a Graph – we know there is negative weight edges, but not if there is negative weight cycle
   1. Detecting
      1. Lemma: if OPT(n, v) = OPT(n – 1, v) for all v, then there are no negative cycles on paths to node t.
         1. This means that if you are done running Bellman-Ford using n – 1 edges, and you run one more iteration so that you are now up to n edges, if the values do not change then there is no negative cycle
      2. Proof = Bellman-Ford
   2. Application
      1. Lemma: if OPT(n, v) < OPT(n – 1, v) for some node v, then (any) shortest path from v to t contains cycle W. Moreover W has a negative costs.
      2. Proof:
         1. Since OPT(n, v) < OPT(n – 1, v) we know P has exactly n edges
         2. By pigeonhole principle P must contain a directed cycle W
         3. Deleted W yields a v-t path with < n edges => W has negative cost
      3. She says these lemmas set up iff in that you have a negative cycle if and only if running one more iteration of Bellman-Ford sees a decrease of one of the corresponding entries from row n – 1 (see example of B-F run)
   3. Theorem: one can detect negative cost cycle in O(mn) time
      1. Add new (sink) node t and connect all nodes to t with 0-cost edge
      2. Check if OPT(n, v) = OPT(n – 1, v) for all nodes v
         1. If yes then there are no negative cycles
         2. If no then extract from shortest path from v to t

This is more efficient than running B-F for n interations

* 1. Currency conversion: given n currencies and exchange rates between pairs of currencies, is there an arbitrage opportunity? Yes if a cycle is > 1, e.g. 100 Francs on orange path ends up being 100.8 Francs after cycle



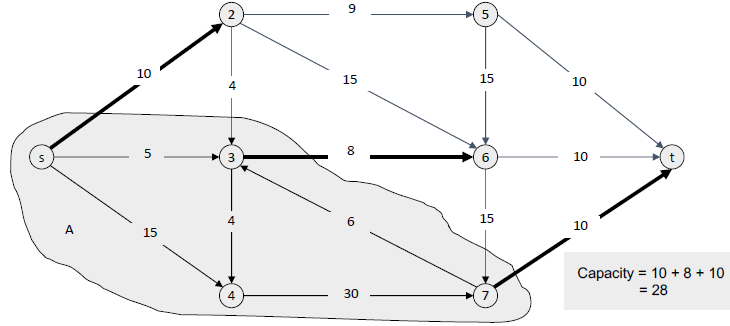
SECOND EXAM ONLY COVERS ABOVE MATERIAL

========================

========================

**Unit 6: Network Flows (page 337)**

1. Maximum Flow = Minimum Cut = cornerstone problems in combinatorial optimization = beautiful math duality
   1. Nontrivial applications/reductions:
      1. Data Mining - Open pit mining
      2. Project Selection - Airline scheduling
      3. Bipartite Matching - Baseball Elimination
      4. Network Connectivity - Network Reliability
      5. Image Segmentation - Egalitarian Stable Matching
      6. Security of Statistical Data - Network Intrusion Detection
      7. Distributed Computing - Multi-Camera Scene Reconstruction - many more
   2. Flow Network = an abstraction for material flowing through the edges of a graph/network
      1. G = (V, E) = directed graph, no parallel edges
      2. Two distinguished nodes: s = source and t = sink
      3. c(e) = capacity of edge e
   3. Cuts
      1. Definition: an s-t cut is a partition (A, B) of V with s ϵ A and t ϵ B
      2. Definition: the capacity of cut (A, B) = cap(A, B) = Σe out of a c(e)
         1. To find capacity of a cut you add up the capacity of all edges that flow from A to B (but not B to A)
         2. Capacity of a cut does NOT consider back flows (whereas flows do consider back flows)
      3. Minimum Cut Problem = find an s-t cut of minimum capacity

 This is a minimum cut

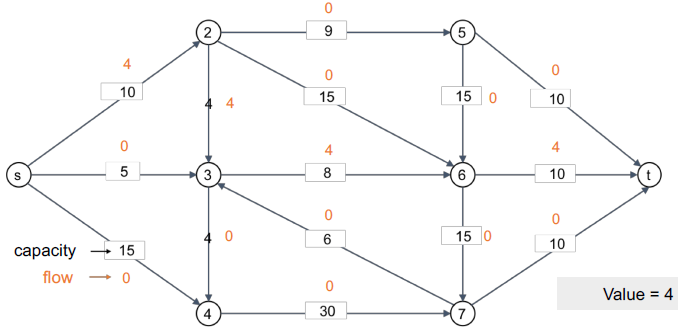
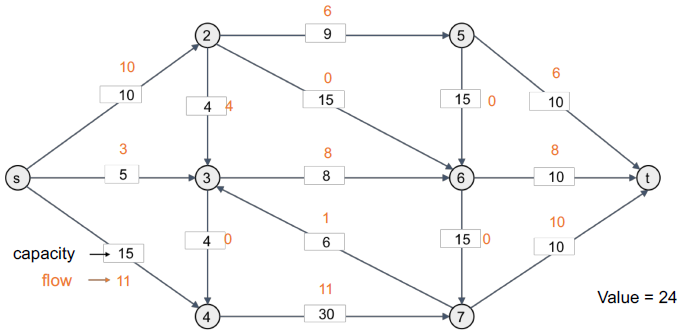
* 1. Flows
     1. Definition: an s-t flow is a function that satisfies: flows DO consider reverse flows unlike capacities
        1. For each e ϵ E: 0 ≤ f(e) ≤ c(e) - capacity (flow is between 0 and capacity inclusive and

flow cannot exceed capacity)

* + - 1. For each v ϵ V – {s, t}: Σe in to v f(e) = Σe out of v f(e) - conservation (whatever goes in node must come out

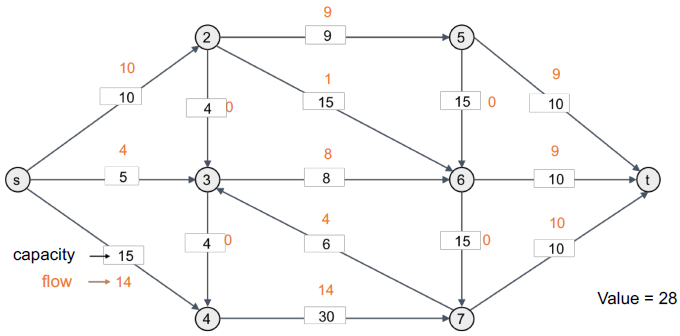
With exception of source and sink nodes thus – {s, t} and note that only place flow starts out of is start node and only place flow ends up is sink node)

* + 1. Definition: the value of a flow f is v(f) = Σe out of s f(e), *i.e*. flow goes from start node to sink node

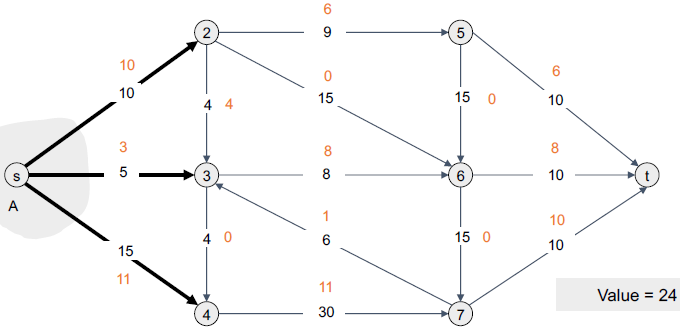
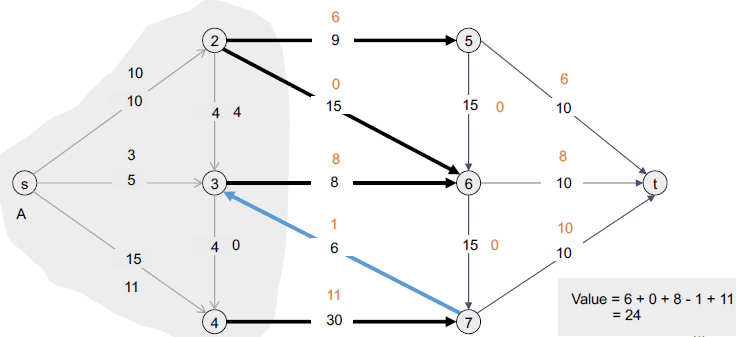
 

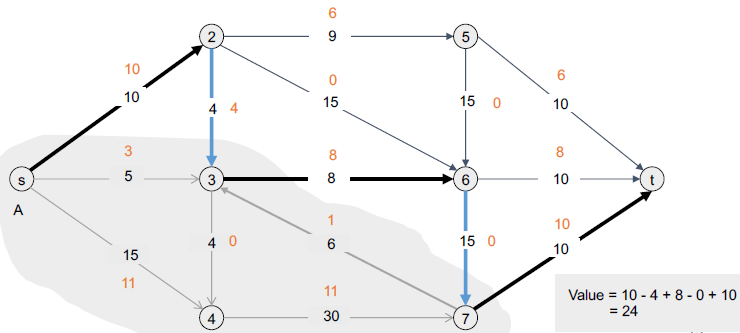
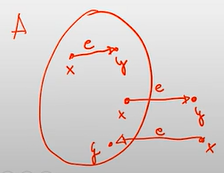
* 1. x

1. Maximum Flow Problem – find s-t flow of maximum value

28 is the maximum amount of flow possible for this network – which as seen a couple of diagrams above is also the minimum cut of this graph as min cut = max flow

1. Flows and Cuts
   1. Flow Value Lemma: let f be any flow, and let (A, B) be any s-t cut. Then, the net flow sent across the cut is equal to the amount leaving s, *i.e*. the net flow leaving s = Σe out to a f(e) – Σe in of a f(e) = v(f)

 +2X/+1X/-1X

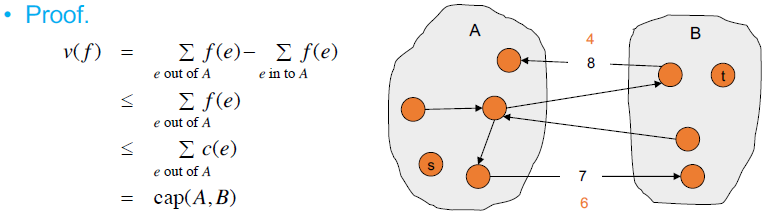
* 1. Proof: v(f) = Σe out of s f(e) - Σe out of s f(e) v(f) = all flow out of s

= Σv ϵ A [Σe out of v f(e) – Σe into v f(e)] by flow conservation, all terms except v = s are 0

|-> <-| must = 0 for Ɐv ϵ A, v ≠ s which means all cancel = lemma

= Σe out of A f(e) – Σe into A f(e) => this is the net flow across the cut

* 1. Weak Duality:
     1. Let f be any flow and let (A, B) be any s-t cut. Then the value of the flow is at most the capacity of the cut, *i.e.* Cut Capacity = 30 => Flow Value ≤ 30
     2. Let f be any flow, then for an s-t cut (A, B) we have v(f) ≤ cap(A, B)



* 1. Certificate of Optimality: let f be any flow and let (A,B) be any cut. If v(f) = cap(A, B) then f is a max flow and (A,B) is a min cut: Value of flow = 28 Cut capacity = 28 => Flow value ≤ 28 – *see* diag on bottom of group of 3 above

1. Towards a Max Flow Algorithm:
   1. Greedy algorithm: (never looks back)
      1. Start with f(e) = 0 for all edges e ϵ E
      2. Find an s-t path P where each edge has f(e) < c(e)
      3. Augment flow along path P
      4. Repeat until you get stuck (local optimality /=> global optimality)
   2. Residual Graph:
      1. Original edge: e = (u, v) ϵ E has Flow f(e), Capacity C(e)
      2. Residual Edge:
         1. “Undo” flow sent
         2. E = (u, v) and eR = (v, u) eR = reverse edge that goes backwards
         3. Residual capacity: for e in E: cf(e) = c(e) – f(e) cf(eR) = f(e) residual cap = original cap - flow
      3. Residual Graph: Gf = (V, Ef)
         1. Residual edges with **positive** residual capacity do not include residual edges with capacity 0
         2. Ef = {e: f(e) < c(e)} U {eR: f(e) **> 0**}
   3. Module 1: Ford-Fulkerson Algorithm
      1. Method: in addition to main graph create a residual network graph which initially looks the same because all the residual edges are 0 so they do not appear
         1. Pick a path, any path, in the residual graph
         2. Find the bottleneck edge on the path as the flow always increases by the value of the bottleneck
         3. Send the bottleneck edge value flow down the path on the ORIGINAL network and indicate flow values
         4. Notate residual graph with reverse flow edges with values = to amount of value assigned to the original graph (note that if all flow is used the reverse edge is simply a reverse of the original edge since the original edge is now 0 and we do not show 0 values on the graph) while reducing the original edges to the value of the capacity – the assigned flow, *i.e*. the reverse edge shows how much is flowing in the path edge (which also shows how much you could DECREASE the flow on that edge in the future of the algorithm) and the original path edge shows how much capacity is left on that edge
         5. Repeat steps 1-4 and adjust flows as needed including “reversing” reverse edges to alter flow, *i.e*. **reduce** the flow on the reverse edge and send it elsewhere
      2. Ultimately done when no more paths possible in residual graph – which = the min s-t cut which includes the nodes that CAN still be reached from the source node = max flow which is the flow of edges crossing cut
   4. Ford-Fulkerson uses Augmenting Path Algorithm
      1. **Psuedocode (hides some details)**

**Augment(f, c, P) { // f = flow; c = capacities of all edges; P = path of edges**

**b <- bottleneck(P) // edge in P w/smallest residual capacity**

**foreach (u, v) ϵ P {**

**if ((u, v) ϵ E) // check if (u, v) is in the original network**

**f((u, v)) <- f((u, v)) + b // if yes is a forward edge and you will increase flow by bottleneck**

**else**

**f((u, v)) <- f((v, u)) – b // if yes is a reverse flow & you decrease flow on corresponding original edge**

**}**

**Return f**

**}**

**Ford-Fulkerson(G, s, t, c) { // G = flow network; s = start; t = sink; c = capacities of all edges**

**foreach e ϵ E f(e) <- 0 // start w/0 flow everywhere // O(n)**

**Gf <- residual graph // build w/same info as main // O(n + m)**

**while (there exists augmenting path P) { // path s-t allows amend – while loop can have up to max flow value of iterations if each +1 and max flow = value of flow of all edges leaving s which ≤ (n – 1) \* C**

**// per iteration running time: O(m + n) BFS/DFS/Dijkstra O(m log n)**

**f <- Augment(f, c, P) // augment current flow on new path // O(n)**

**update Gf // complexity of update hidden // O(n) really O(m)?**

**}**

**return f**

**} // n = number of nodes and m – number of edges (n – 1)**

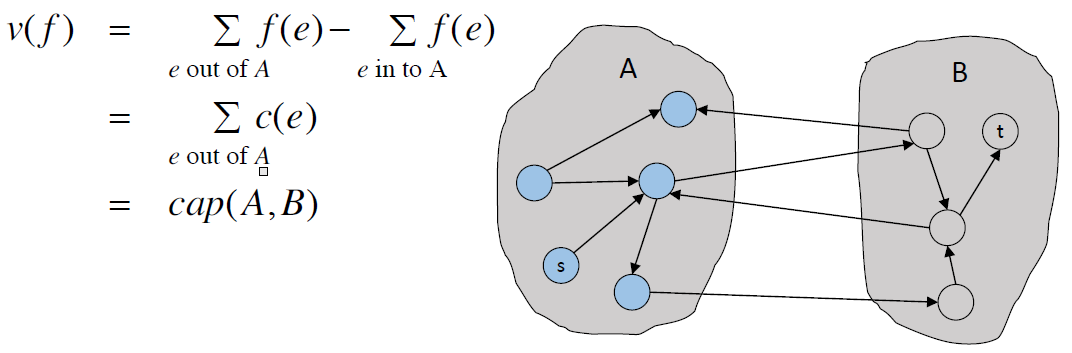
1. Module 2: Max Flow-Min Cut Theorem (page 346) – Ford-Fulkerson (1956) => min cut = max flow
   1. Augmenting path theorem = flow f is a max flow iff there are no augmenting paths
   2. Max-flow min-cut theorem proof (by transitivity): we prove both simultaneously by showing that TFAE:
      1. There exists a cut (A, B) such that v(f) = cap(A, B)
      2. Flow f is a max flow
      3. There is no augmented path relative to f

(i) => (ii) this was a corollary to weak duality lemma

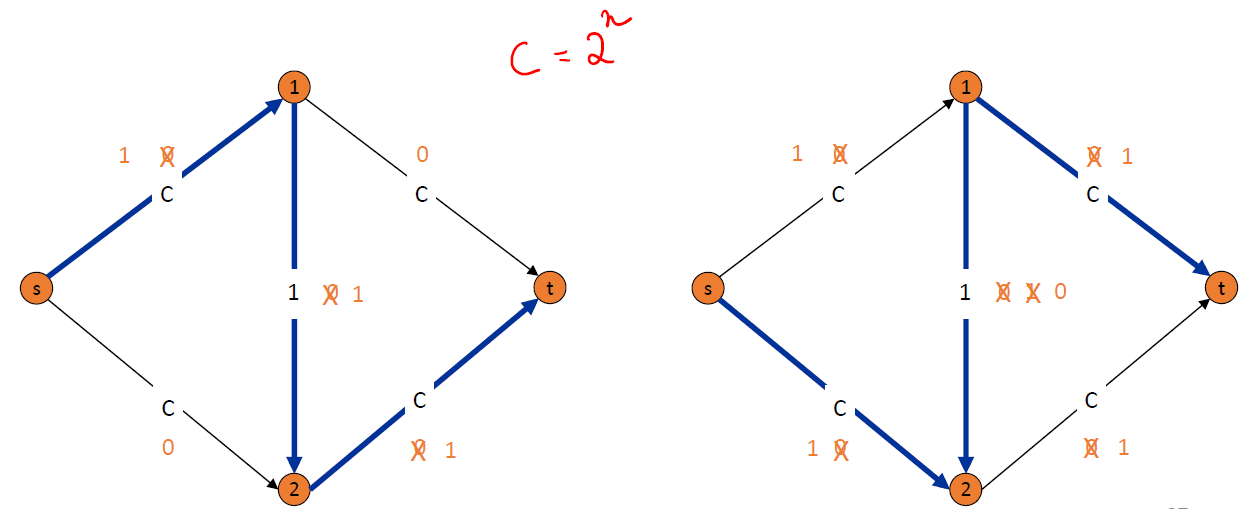
(ii) => (iii) we show contrapositive. Let f be flow. If there exists an augmented path, then we can improve f by sending flow along path contrapositive = ┐(iii) => ┐(ii)

(iii) => (i)

* + - 1. Let f be flow with no augmenting paths
      2. Let A be set of vertices reachable from s in residual graph
      3. By definition of A, s ϵ A
      4. By definition of f, t /ϵ A



1. Running Time (depends on how many times we go through the while loop in the pseudocode)
   1. Assumption: all capacities are integers between 1 and C
   2. Invariant: every flow value f(e) and every residual capacities cf(e) remains an integer throughout algorithm
   3. Theorem: the algorithm terminates in at most v(f\*) ≤ (n – 1) \* C iterations
      1. Proof: each augmentation increases value by at least one
      2. Worst-case running time of Ford-Fulkerson = O(nC(m + n)) = O(nmC) because assuming m ≥ n – 1
   4. Corollary: if C = 1 or any constant, Ford-Fulkerson runs in O(mn) time after assumptions that C = a constant
   5. Integrality theorem: if all capacities are integers, then there exists a max flow f for which every flow value f(e) is an integer
      1. Proof: since algorithm terminates, theorem follows from invariant
2. Ford-Fulkerson: Exponential Number of Augmentations
   1. Question: is generic Ford-Fulkerson algorithm polynomial in input size (m, n, and log C)?
   2. Answer: (like Knapsack – No). if max capacity is C, then algorithm can take C iterations

here each augmentation is + 1 if C = 2n leads to exponential running time because like pseudo-polynomial knapsack in C = 2n then it depends on log n and we end up with exponential running time – **but unlike knapsack we are not “doomed” to NP-Complete here =>**

1. Choosing Good Augmenting Paths
   1. Use care when selecting augmenting paths
      1. Some choices lead to exponential algorithms
      2. Clever choices lead to polynomial algorithms
      3. If capacities are irrational, algorithm not guaranteed to terminate!
   2. Goal: choose augmenting path so that:
      1. Can find augmenting paths efficiently
      2. Few iterations
   3. Choose augmenting path with: [Edmonds-Karp 1972 [= FF using BFS], Dinitz 1970
      1. Max bottleneck capacity – try to do most augmenting in each pass – algorithm a bit tricky
      2. Sufficiently large bottleneck capacity – little easier than max bottleneck = Capacity Scaling – *see* next section
      3. Fewest number of edges (Edmonds-Karp) – try to pick a path with fewest number of edges, BFS does this
2. Capacity Scaling Algorithm
   1. Intuition: choosing path with highest bottleneck capacity increases flow by max possible amount
      1. Don’t worry about finding exact highest bottleneck path
      2. Maintain scaling parameter ∆ - start with a very large delta and will result in bottleneck with large capacity
         1. If no path found from s-t then reduce ∆ and continue, *e.g*. if you have edges all with capacities > 100 but for one with capacity = 1 if you use Gf(100) you will not consider the one small edge
      3. Let Gf(∆) be subparagraph of the residual graph consisting of only arcs with capacity at least ∆

**Psuedocode**

**Scaling-Max-Flow(G, S, T, c) {**

**foreach e ϵ E f(e) <- 0**

**∆ <- smallest power of 2 greater than or equal to C // want to minimize ∆ iterations so power of 2**

**Gf <- residual graph**

**while (∆ ≥ 1) { // Ford-Fulkerson look at Gf(∆) not Gf directly**

**Gf(∆) <- ∆-residual graph**

**While (there exists augmenting path P in Gf(∆)) {**

**F <- augment(f, c, P)**

**Update Gf(∆)**

**}**

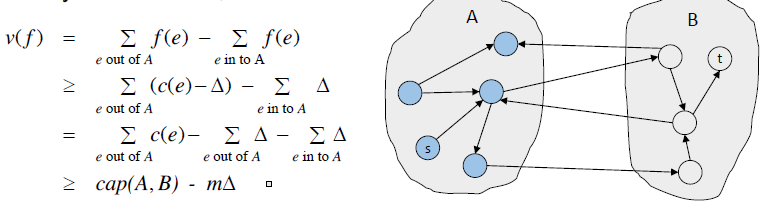
**∆ <- ∆/2 // halve ∆ so look at log(C) different ∆ values**

**}**

**Return f**

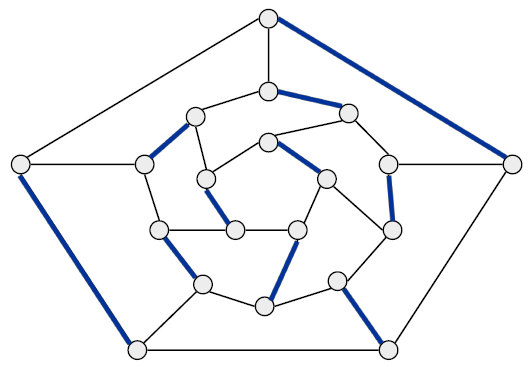
**}**

* 1. Proof of correctness:
     1. Assumption: all edge capacities are integers between 1 and C
     2. Integrality invariant: all flow and residual capacities values are integral
     3. Correctness: if algorithm terminates, then f is a max flow: Proof
        1. By integrality invariant, when ∆ = 1 => Gf(∆) = Gf
        2. Upon termination of ∆ = 1 phase, there are no augmenting paths
  2. Capacity Scaling – Running Time
     1. Lemma 1: the outer while loop repeats 1 + ⌈log2C⌉ times
        1. Proof: initially c ≤ ∆ < 2C. ∆ decreases by a factor of 2 each iteration
     2. Lemma 2: let f be the flow at the end of a ∆-scaling phase. Then the value of the maximum flows is at most v(f) + m ∆
        1. Proof is almost identical to max-flow min-cut theorem
           1. We show that at the end of a ∆-phase, there exists a cut (A, B) such that cap(A, B) ≤ v(f) + m ∆
           2. Choose A to be the set of nodes reachable from s in Gf(∆)
           3. By definition of a s ϵ A
           4. By definition of f, t /ϵ A

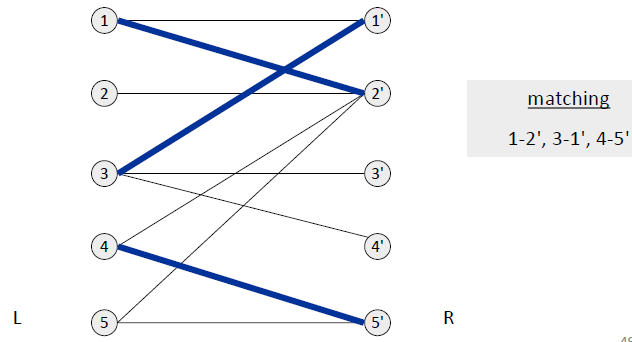
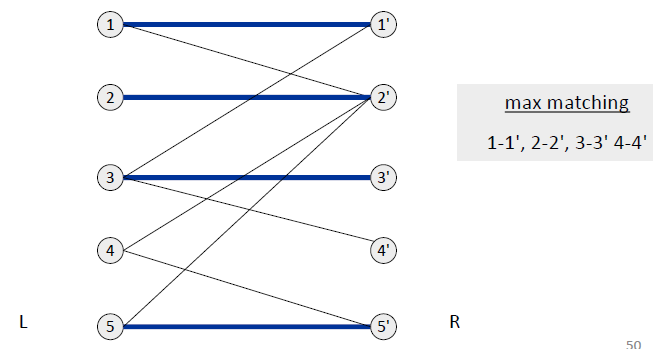


* + 1. Lemma 3: there are at most 2m augmentation per scaling phase (for 2∆). L2 => v(f\*) ≤ m (2∆). Each augmentation in a ∆-phase increases v(f) by at least ∆
    2. Theorem: the scaling max-flow algorithm finds a max flow in O(m log C) augmentations. It can be implemented to run in O(m2 log C) time

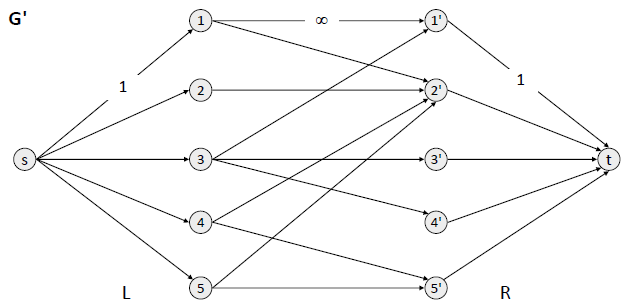
1. Edmonds-Karp/Dinitz Algorithm = Fewest Number of Edges = same Ford-Fulkerson just using BFS to find a path
   1. Augment along shortest augmenting path (in terms of number of edges): use BFS
   2. Running time = O(nm2) from O(mn) augmentations and O(m) per augmentation – **strongly polynomial**
2. Module 4: Bipartite Matching Max Cardinality
   1. Matching:
      1. Input: undirected graph G = (V, E) find subset such that no two edges share an endpoint
      2. M ⊆ E is a matching if each **node** appears in at most one edge in M
      3. Max matching = find a max cardinality matching – blue edges here represent a maximum matching:



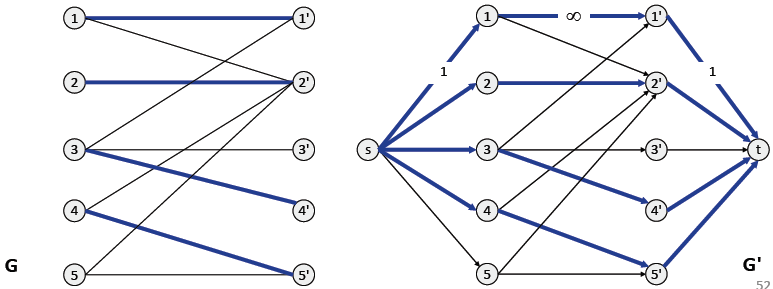
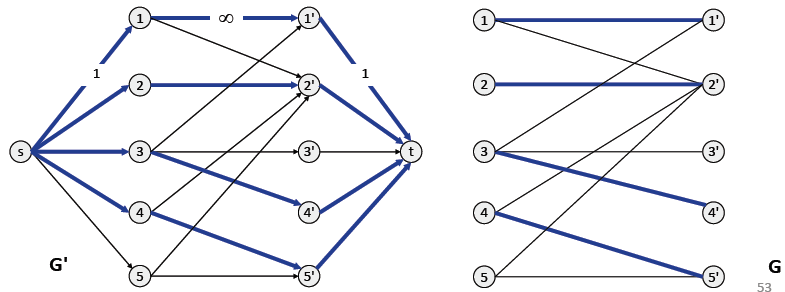
* 1. Bipartite Matching: all edges are between a node in Left subset and Right subset
     1. Input: undirected, bipartite graph G = (L U R, E)
     2. M ⊆ E is a matching if each node appears in at most one edge in M
     3. Max Matching = find a max cardinality matching

* + 1. Max flow formulation: think of edges representing a flow from Left subset to Right subset by adding s and t and maximum flow = max bipartite matching
       1. Create digraph G’ = (L U R U {s, t}, E’)
       2. Direct all edges from L to R, and assign infinite (or unit) capacity
       3. Add source s, and unit capacity edges from s to each node in L
       4. Add sink t, and unit capacity edges from each node in R to t

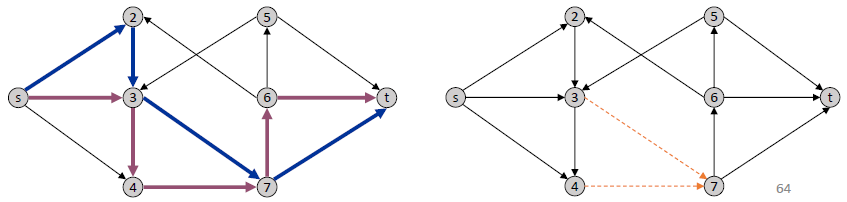
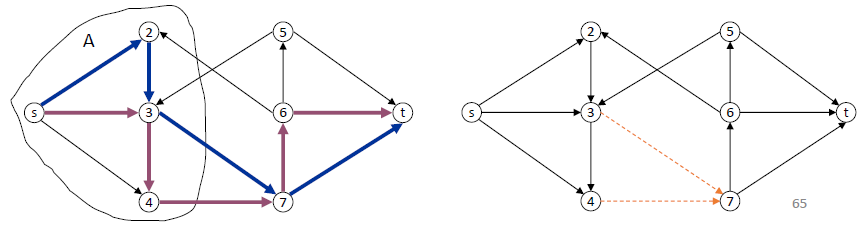
here EACH edge from s and to t has capacity of 1 and capacity of center edges does not matter as long as ≥ 1 as only flow of 1 going into/out of each

* 1. Bipartite Matching Proof of Correctness:
     1. Theorem: max cardinality matching in G = value of max flow in G’
     2. Proof: ≤ (see left diagram below)
        1. Given max matching M of cardinality k
        2. Consider flow f that sends 1 unit along each of k paths
        3. f is a flow, and has cardinality k

* + 1. Proof: ≥ (*see* right diagram above)
       1. Let f be a max flow in G’ of value k
       2. Integrality theorem => k is integral and can assume f is 0-1
       3. Consider M = set of edges from L to R with f(e) = 1
          1. Each node in L and R participates in at most one edge in M
          2. |M| = k: consider cut (L U s, R U t)
  1. Bipartite Matching Running Time
     1. Which max flow algorithm to use for bipartite matching? Does not matter too much
        1. Generic augmenting path: O(m val(f\*)) = O(mn) generic Ford-Fulkerson – max flow ≤ 1 as edges = 1
        2. Capacity scaling: O(m2 log C) = O(m2) slightly higher worst case than generic
        3. Shortest augmenting path: O(m n1/2)
     2. Non-bipartite matching
        1. Structure of non-bipartite graphs is more complicated, but well-understood (Tutte-Berge, Edmonds-Galai) uses idea of “blossoms”
        2. Blossom algorithm: O(n4) (Edmonds 1965)
        3. Best Known: O(m n1/2) (Micali-Vazirani 1980) so matches best time for bipartite

1. Edge Disjoint Paths = 2 paths edge disjoint if they do not share an edge (but can share nodes)
   1. Disjoint path problem: given a digraph G = (V, E) and two nodes s and t, find the max number of edge-disjoint s-t paths
   2. Definition: two paths are edge-disjoint if they have no edge in common (but can share nodes)
   3. Example: communication networks
   4. Max flow formulation: assign unit capacity to every edge assign every edge capacity of one
   5. Theorem: max number edge-disjoint s-t paths equals max flow value
   6. Proof: ≤
      1. Suppose there are k edge-disjoint paths P1, . . . , Pk
      2. Set f(e) = 1 if e participates in some path Pi; else set f(e) = 0
      3. Since paths are edge-disjoint, f is a flow of value k
   7. Proof: ≥
      1. Suppose max flow value is k
      2. Integrality theorem => there exists 0-1 flow f of value k
      3. Consider edge (s, u) with f(s, u) = 1
         1. By conservation, there exists an edge (u, v) with F(u, v) = 1
         2. Continue until reach t, always choosing a new edge
      4. Produces k (not necessarily simple, i.e. can eliminate cycles to get simple paths if desired) edge-disjoint paths
2. Network Connectivity
   1. Network Connectivity = given a digraph G = (E, V) and two nodes s and t, find min number of edges whose removal disconnects t from s
   2. Definition = a set of edges F ⊆ E disconnects to from s if all s-t paths use at least one edge in F
3. Edge Disjoint Paths and Network Connectivity
   1. Theorem (Menger 1927): the maximum number of edge-disjoint s-t paths is equal to the minimum number of edges whose removal disconnects to from s
      1. **which is shown by the minimum cut/maximum flow IFF the flow capacities of each edge is 1**
   2. Proof: ≤ (*see* left diagram below)
      1. Suppose the removal of F ⊆ E disconnects to from s, and |F| = k
      2. All s-t paths use at least one edge of F, hence, the number of edge-disjoint paths is at most k

* 1. Proof: ≥ (*see* right diagram above)
     1. Suppose max number of edge-disjoint paths is k
     2. Then max flow value is k
     3. Max-flow min-cut => cut (A, B) of capacity k
     4. Let F be a set of edges going from A to B
     5. |F| = k and disconnects to from s

**Unit 7: Polynomial Time Reductions and NP-Completeness (page 451)**

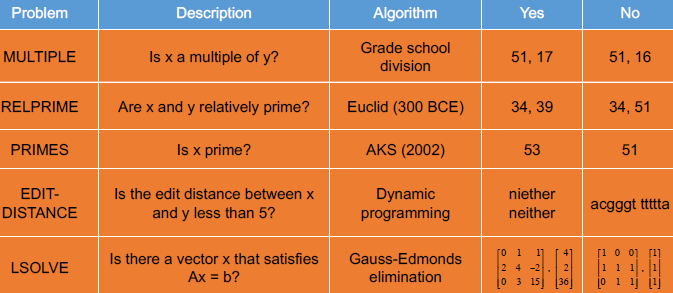
Module 2: Classes P and NP

1. Algorithm Design Patterns and Anti-Patterns
   1. Algorithm Design Patterns Example
      1. Greedy O(n log n) interval scheduling
      2. Divide-and-Conquer O(n log n) FFT
      3. Dynamic programming O(n2) edit distance
      4. Duality O(n3) bipartite matching – BUT previously O(mn), O(m2), and O(m n1/2)
      5. Reductions
      6. Local search
      7. Randomization
   2. Algorithm Design Anti-Patterns – instead of showing how to solve a problem, showing it might be unsolvable
      1. NP-completeness O(nk) algorithm unlikely
      2. PSPACE-completeness O(nk) certification algorithm unlikely
      3. Undecidability No algorithm possible
2. Classify Problems, i.e. which problems will we be able to solve in practice = those with polynomial-time algorithms

|  |  |
| --- | --- |
| YES | Probably NO |
| Shortest Path | Longest Path |
| Matching | 3-D Matching |
| Minimum Cut | Maximum Cut |
| 2-SAT | 3-SAT |
| Planar 4-Color | Planar 3-Color |
| Bipartite Vertex Cover | Vertex Cover |
|  |  |
| Primality Testing | Factoring |

Cobham 1964, Edmonds 1965, Rabin 1966Classes P and NP

* 1. Decision Problems
  2. Polynomial time
  3. Primes – prior to development, calculating primes was exponential on the representation of the input number
  4. Definition of P = decision problems for which there is a poly-time algorithm



* 1. Definition of NP = decision problems for which there exists a poly-time certifier
     1. **NP = nondeterministic polynomial time**
     2. **Some problem in NP – the NP-Complete problems – are those for which no one knows a deterministic polynomial algorithm for solving them, and the implications for finding one such algorithm is HUGE because once you find that, you will have found a deterministic polynomial algorithm for all of them**
     3. Certifier views things from a “managerial” viewpoint
     4. Certifier does not determine whether s ϵ X on its own, but rather it checks a proposed proof t that s ϵ X
     5. Definition: algorithm C(s, t) is a certifier for problem X iff for every string s, s ϵ X, there exists a string t such that C(S, t) = yes

1. Certifiers and Certificates
   1. Composites: given an integer s, is s composite
      1. Certificate: a nontrivial factor to of s. Note that such certificate exists iff s is a composite. Moreover |t| ≤ |s|
      2. Example certifier: **Boolean C(s, t) {**

**if (t ≤ 1 or t ≥ s)**

**return false**

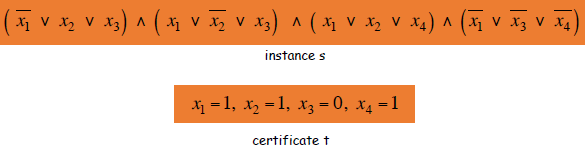
**else if (s is a multiple of t)**

**return true**

**else**

**return false }**

* + 1. Example: s = 437,669 Certificate: t = 541 or 809 as 437,669 = 541 \* 809
    2. Conclusion: Composites is in NP
  1. Satisfiability – SAT = given a CNF formula ɸ, is there a satisfying assignment?
     1. Certificate = an assignment of truth values to the n Boolean variables
     2. Certifier = check that each clause in ɸ has at least one true literal
     3. Example:



* + 1. Conclusion: SAT is in NP; also 3-SAT, etc.
  1. Hamiltonian Cycle: given an undirected graph G = (V, E) does there exists a simple cycle C that visits every node
     1. Certificate = a permutation of the n nodes
     2. Certifier = check that the permutation contains each node in V exactly once, and that there is an edge between each pair of adjacent nodes in permutation
     3. Conclusion: HAM-CYCLE is in NP

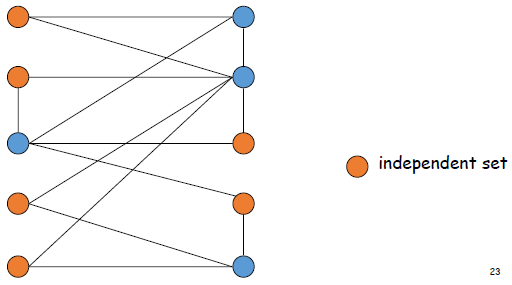
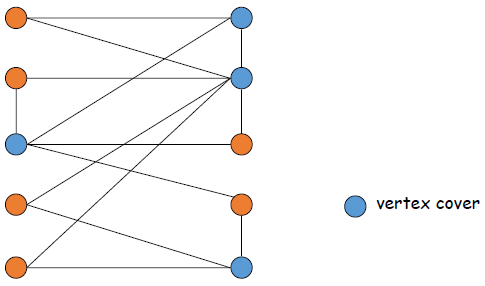
1. P, NP, EXP
   1. **P = there is a poly-time algorithm (all P problems are NP problems)**
   2. **EXP = there is an exponential-time algorithm**
   3. **NP = decision problems for which there is a poly-time certifier and we do not know whether a poly-time algorithm exists for it (there are thousands of NP-Complete problems)**
      1. **A theoretical oracle is what provides the “satisfying assignment” => true answers to be certified with a yes**
      2. **NOTE that there is NO polynomial-time certifier that can tell you the guess is wrong**
      3. **TEXT: A polynomial-time algorithm for any NP-Complete problem would imply the existence of a polynomial-time algorithm for all of them**
   4. **NP-Complete = a NP-complete problem Y has the property that for all X (X can be any class contained in NP) in NP that X ≤p Y (X reduces to Y) and Y is in NP = at least as hard as everything in NP because it is in NP-Hard**
      1. **YouTube said NP-Complete problems are in both NP and NP-Hard**
      2. **All NP-Complete problems are NP-Hard but all NP-Hard are not NP-Complete**
   5. **NP-Hard = at least as hard to solve as any problem in NP**
   6. Claim: P ⊆ NP
      1. Proof: consider any problem X in P:
         1. By definition there exists a poly-time algorithm A(s) that solves X
         2. Certificate t = ɛ, certifier C(s, t) = A(s)
   7. Claim: NP ⊆ EXP
      1. Proof: consider any problem X in NP
         1. By definition there exists a poly-time certifier C(s, t) for X
         2. To solve input s, run C(s, t) on all strings t with |t| ≤ |p(|s|) and there will be an exponential number of such strings, basically b{p(|s|)} where b is the size of your alphabet
         3. Return yes if C(s, t) returns yes for any of these
   8. The main question: P versus NP
      1. Is P = NP? Subject to Clay $1,000,000 prize Consensus opinion is P ≠ NP
         1. If yes: efficient algorithms for 3-color, TSP, Factoring (which would break RSA), SAT, . . .
         2. If no: no efficient algorithms possible for 3-color, TSP, SAT, . . .
         3. Every problem in NP reduces to X so X /ϵ P unless P = NP because if Y ϵ NP – P then Y would have a poly-time algorithm if X had a poly-time algorithm and P = NP

Module 2: **Polynomial-Time Reductions** (page 452) – in this class these are ≡ polynomial transformations

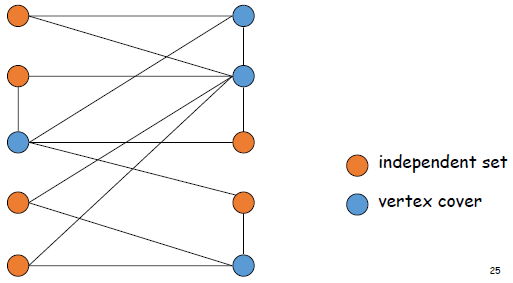
1. Desiderata: suppose we could solve Y in polynomial-time. What else could we solve in Polynomial time?
   1. Reduction from problem A -> problem B = poly-time algorithm converting A inputs ≡ B inputs then runs B in order to find answer to A and if B has a poly-time algorithm then so does A
2. Reduction: problem X polynomially reduces to problem Y if arbitrary instances of problem X can be solved using:
   1. Polynomial number of standard computational steps, plus if you make polynomial number of calls to an
   2. Polynomial number of calls to oracle that solves problem Y algo that runs in poly time then overall = poly
   3. **A polynomial transformation only allowed to make one call to the oracle algorithm not polynomial amount of calls, which she says covers all of the polynomial-time reductions we talk about in this class**
   4. **TEXT: can arbitrary instances of problem X be solved using a polynomial number of standard computational steps, plus a polynomial number of call to a black box that solves problem Y? if yes, then X ≤p Y which is read as either “X is polynomial-time reducible to Y,” or “ Y is at least as hard as X (with respect to polynomial-time).” Note that we still pay for the time to write the input of the black box solving Y, and to read the answer the black provides.**
   5. **AXIOM 8.1: suppose X ≤p Y. If Y can be solved in polynomial-time, then X can be solved in polynomial-time.**
   6. **AXIOM 8.2: suppose X ≤p Y. If X cannot be solved in polynomial-time, then Y cannot be solved in polynomial-time.**
3. Notation for reduction of X to Y: X ≤p Y **NOTE the ≤p effectively points BACKWARDS as X is reducing to Y**
   1. A reduction X ≤p Y means that problem Y is **at least as hard to solve as problem X**
   2. **This means that you can use Y polynomially many times as an “oracle” to solve problem x**
4. Remarks:
   1. We pay for time to write down instances sent to black box => instances of Y must be of polynomial size because if > just to write them down would take > polynomial time
   2. Note: Cook reducibility = polynomial reduction
5. Purpose: classify problems according to relative difficulty
6. Design Algorithms: if X ≤p Y and Y can be solved in polynomial-time then X can also be solved in polynomial time
   1. A reduction X ≤p Y means that problem Y is **at least as hard to solve as problem X**
7. Establish intractability: if X ≤p Y and X cannot be solved in polynomial-time then Y cannot be solved in polynomial-time because if Y could be solved in polynomial time it would infer that X could also be solved in polynomial time which is not true if we can shows that X cannot be solved in polynomial time
8. Establish equivalence (up to cost reduction): if X ≤p Y and Y ≤p X we use notation X ≡p Y
9. **Reduction by simple equivalence** (as opposed to special case to general case or encoding with gadgets)
   1. INDEPENDENT-SET: given a graph G = (V, E) and an integer k, is there a subset of vertices S ⊆ V such that |S| ≥ k, and for each edge at most one of its endpoints is in S? *i.e.* there is no edge in between the nodes of the set S
   2. **TEXT: says finding the largest independent set (optimization version) was addressed in chapter 1, but here we turn it into a yes/no (decision) problem by introducing the constant k. For given a graph *G* on *n***

**nodes, we simply solve the decision version of Independent Set for each *k*; the largest *k* for which the answer is “yes” is the size of the largest independent set in *G*. (And using binary search, we need only solve the decision version for *O(*log *n)* different values of *k*.)**

* + 1. Example: is there an independent set of size ≥ 6? Yes as no edge with two orange end-points
       1. Certifier can issue certificate by finding edges that do not have both endpoints in orange set in poly time
    2. Example: is there an independent set of size ≥ 7? No – no good way to show this

1. **Vertex Cover**: given a graph G = (V, E) and an integer k, is there a subset of vertices S ⊆ V such that |S| ≤ k, and for each edge at most one of its endpoints is in S? **TEXT: vertices do the “covering” and edges are the objects being “covered.” While the entire set V is a cover, the hard part is finding small covers**
   * 1. Example: is there an independent set of size ≤ 4? Yes as every edge in group has one orange node end point
     2. Example: is there an independent set of size ≤ 3? No – no good way to show this
2. VERTEX-COVER ≡p INDEPENDENT-SET **(comes from V-C ≤p I-S and I-S ≤p V-C thus they are equivalent)**
   * 1. Proof: given a graph G(V, E) we show that S is an independent set of size k iff (its complement) V – S is a vertex cover of size k’ = n – k (NOTE: you could either set k to size of independent set or size of vertex cover instance => k’)



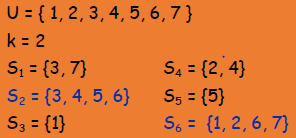
=>

* + - 1. Let S be any independent set
      2. Consider an arbitrary edge (u, v)
      3. S independent => u /ɛ S or v /ɛ S => u ɛ V – S or v ɛ V – S
      4. Thus, V – S covers (u, v)

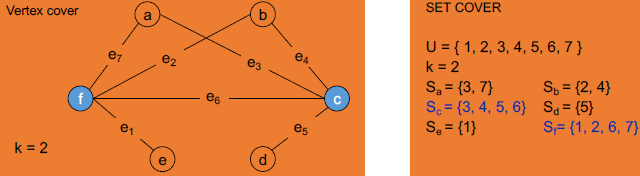
<=

* + - 1. Let V – S be any vertex cover
      2. Consider two nodes u ϵ S and v ϵ S
      3. Observe that (u, v) /ϵ E since V – S is a vertex cover
      4. Thus, no two nodes in S are joined by an edge => S independent set
  1. **TEXT: If we have a black box to solve Vertex Cover, then we can decide whether G has a independent set at least k by asking the black box whether G has a vertex cover of size at most n – k.**
  2. **TEXT: if we have a black box to solve Independent Set, then we can deicide whether G has a vertex cover of size at most k by asking the black box whether G has an independent set of size at least n – k.**

1. **Reduction from special case to general case** (as opposed to Reduction by simple equivalence or encoding with gadgets)
   1. **SET-COVER**: given a set U of elements, a collection S1, S2, . . . , Sm of subsets of U, and an integer k, does there exist a collection of ≤ k of these sets whose union is equal to U?
      1. Sample application:
         1. m available pieces of software – each with certain functions and you want group of functions
         2. Set U of n capabilities that we would like our system to have
         3. The ith piece of software provides the set Si ⊆ U of capabilities
         4. Goal: achieve all n capabilities using fewest pieces of software
      2. Example:

 Union of both blue subsets ≡ the set U

* 1. **Vertex Cover Reduces to Set Cover**: Claim VERTEX-COVER ≤p SET-COVER
     1. VERTEX-COVER is a special instance of SET-COVER, but NO proof for SC ≤p VC
     2. Proof: 0) given VERTEX-COVER instance G = (V, E), k, we construct a set cover instance whose size equals the size of the vertex cover instance (should be able to do this is polynomial time)
        1. Construction: create a set-cover instance: k = k, U = E, Sv = {e ϵ E: e incident to v} **FROM VERTEX-COVER**
        2. Set-cover of size ≤ iff cover of size ≤ k



**TEXT: goal is to cover the edges in E, so we formulate an instance of VERTEX-COVER then model it with an instance of SET-COVER, then assuming we have an oracle/certifier/black-box for SET-COVER we find a vertex-cover of the size k iff the oracle finds s set-cover of the size k**

1. **Reduction by encoding with gadgets** (as opposed to Reduction by simple equivalence or special to general case)
   1. Satisfiability this is how MIT reduced 3-SAT to Super Mario Brothers
      1. Literal: a Boolean variable or its negation
      2. Clause: a disjunction of literals = literals joined by OR so only one literal needs to be true for true
      3. Conjunctive normal form (CNF): a propositional formula ɸ that is the conjunction of clauses
         1. = clauses joined by AND so all clauses have to be true for true
      4. SAT: given CNF formula ɸ does it have a satisfying truth assignment of truth assignment to the variables?
         1. MIT says any algorithm = circuit = Boolean = 3-SAT so 3-SAT is a stand-in for an algorithm, *e.g*. he reduces 3-SAT to Super Mario Brothers
      5. 3-SAT: SAT where each clause contains exactly 3 literals (each corresponding to a different variable)

 note 3 literals given

* 1. 3 Satisfiability Reduces to Independent Set:
     1. Claim 3-SAT ≤p INDEPENDENT-SET ***i.e*. solve 3-SAT with INDEPENDENT-SET black box or oracle**
        1. Proof: given an instance ɸ of 3-SAT, we construct instance (G, k) of INDEPENDENT-SET that has independent set of size k iff ɸ is satisfiable
           1. Construction:

G contains 3 vertices for each clause, one for each literal = create triangle for each clause

Connect 3 literals in a clause in a triangle – Note literals will appear > once perhaps 3 times

Connect literal to each of its negations - *see* diagram below – **encode conflicts by adding edges**

K = number of clauses so since 3 clauses k = 3

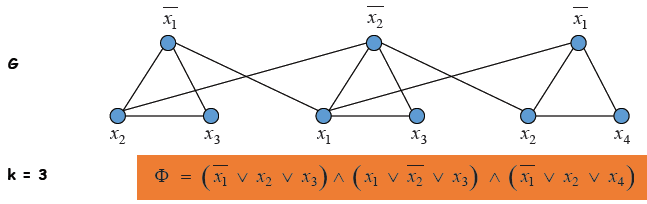
* + 1. Claim: G contains independent set of size k = |ɸ| iff ɸ is satisfiable
    2. **Recall you cannot put 2 nodes in an independent set if there are edges between them so you will never have a variable and its negation as part of the set because there is an edge between that pair of nodes**
       1. Proof:

=>

* + - * 1. Let S be independent set of size k
        2. S must contain exactly on vertex in each triangle
        3. Set these literals to true (and any other variables in a consistent way)
        4. Truth assignment is consistent and all clauses are satisfied

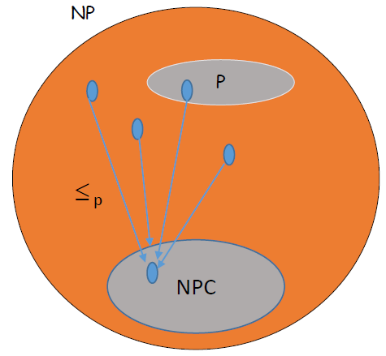
<=

* + - 1. Given satisfying assignment, select one true literal from each triangle. This is an indy set of size k



1. Review
   1. Basic reduction strategies
      1. Simple equivalence: INDEPENDENT-SET ≡p VERTEX-SET
      2. Special case to general case: VERTEX-SET ≤p SET-COVER
      3. Encoding with gadgets: 3-SAT ≤p INDEPENDENT-SET
      4. **NOTE that overall: 3-SAT ≡p INDEPENDENT-SET ≡p VERTEX-COVER ≡p SET-COVER**
   2. Transitivity: if X ≤p Y and Y ≤p Z, then X ≤p  Z
      1. Proof idea: concatenate the two algorithms – one algorithm for X ≤p Y and one algorithm for Y ≤p Z = X ≤p  Z
   3. Example: 3-SAT ≤p INDEPENDENT-SET ≤p VERTEX-COVER ≤p SET-COVER but **these are all really ≡p**

Module 3: **NP-Completeness** (page 466) **= not solvable in poly time, but verifiable in poly time so both NP and NP-Hard**

1. **Arguably the most natural way to define a “hardest” problem *X* is via the following two properties: (i) *X* ∈ NP; and (ii) for all *Y* ∈ NP, *Y* ≤*P X*. In other words, we require that every problem in NP can be reduced to *X*. We will call such an *X* an *NP-complete* problem.**
   1. **Suppose X is an NP-Complete problem. Then X is solvable in polynomial-time iff P = NP. A crucial consequence of this is that if there is ANY problem in NP that cannot be solved in polynomial-time, then no np-Complete problem can be solved in polynomial-time.**
2. Classify Problems
   1. Goal: classify problems according to those that can be solved in polynomial-time and those that cannot
   2. Probably requires exponential-time
      1. Given Turing machine, does it halt in at most k steps? Note: in this class we look higher level than Turing
         1. it takes log(k) bits to represent k so the running time will be exponential based on k or O(nlog(k))
         2. NP-Completeness is defined at the Turing Machine level, but here we are looking at a higher level
      2. Given a board position in an n-by-n generalization of chess, can black guarantee a win?
         1. An exponential-time algorithm exists for answering this, but no polynomial algorithm exists
   3. Frustrating news: huge number of fundamental problems have defied classification for decades – we will show that these problems are “computationally equivalent” and appear to be different manifestations of one really hard problem, i.e. they define the class and if one of them is solved it impacts other NP-complete problems
3. Polynomial Transformation
   1. Definition: problem X polynomially **reduces (Cook)** to problem Y if arbitrary instances of problem X can be solved using:
      1. Polynomial number of standard computational steps, plus
      2. Polynomial number of calls to oracle that solves problem Y
   2. Definition: problem X polynomially **transforms (Karp)** to problem Y if given any input x to X, we can construct an input y in polynomial time such that x is a yes instance of X iff y is a yes instance of Y – (|y| must be poly in |x|)
   3. Note: polynomial transformation is polynomial reduction with just one call to oracle for Y. All reductions we will see in this class are polynomial transformations
   4. Open Question: are these two concepts the same? Abusing notation ≤p and blurring distinction
4. **NP-Complete** 
   1. NP-Complete: **a NP-complete problem Y has the property that for all X (X can be any class contained in NP) in NP that X ≤p Y (X reduces to Y) and Y is in NP and at least as hard as any other problem in NP (because it is in NP-Hard) and because if you can solve Y you could solve any other problem in NP (and if Y could be solved in polynomial time then P = NP see below and note that determining primes was NEVER considered NP-Complete and that someone has found a O(n8) algorithm for finding primes does NOT mean P = NP)**
      1. Any problem in NP (*see* diagram) including one in P, has to be reducible to polynomial be NP-Complete
   2. Theorem: suppose Y is an NP-Complete problem. Then Y is solvable in poly-time iff P = NP
   3. Proof: **this has never actually happened**

<= If P = NP then Y can be solved in poly-time since Y is in NP

=> Suppose Y can be solved in poly-time

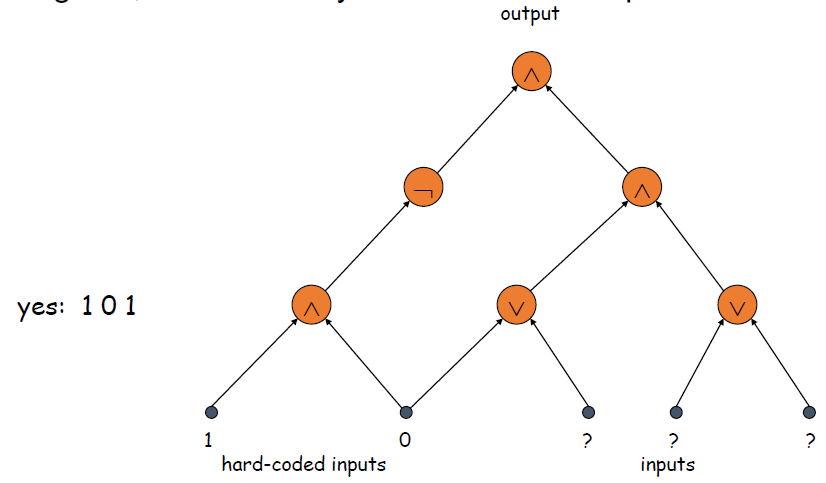
- Let X be any problem in NP. Since X ≤p Y, we can solve X in poly-time. This implies NP ⊆ P

- We already know P ⊆ NP, thus P = NP

**NOTE: many algorithms have been declared NP-Complete but none have been solved in polynomial times (the primes problem that has an O(n8) poly algorithm since 2002 has never been shown to be NP-Complete)**

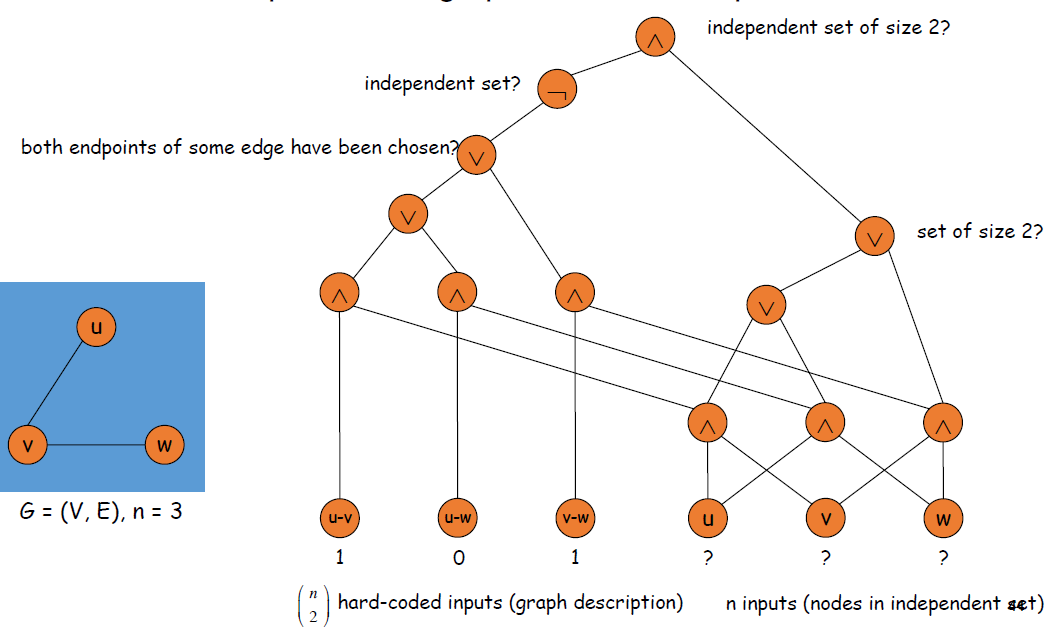
* 1. Fundamental Question: are there “natural” NP-Complete problems?

1. Circuit Satisfiability – a place to start on how to show NP-Completeness – if a problem is in NP we know a certifier exists verifying yes instances to that problem – and that certifier runs in polynomial time given the right certificate – the idea is encapsulate all the steps the certifier would be doing into a set of Boolean clauses that would only be true if you had taken the right steps for the certifier and that for your certifier to work you need the right certificate – if you transform X into an instance of satisfiability and you solve the satisfiability problem that shows you found the satisfying set of variables in the formula that is a satisfying assignment that corresponds to the what the certificate should have been – so if you solve the set problem if you build clauses correctly to mimic what the satisfier is doing for your NP-Complete problem then a satisfying assignment corresponds to the certificate that will guide you toward the right solution for the problem
   1. CIRCUIT-SAT: Given a combinatorial circuit built out of AND, OR, and NOT gages, is there a way to set the circuit inputs so that the output is 1? (She does not seem concerned we know this, just saying to know that CIRCUIT-SAT = SAT)

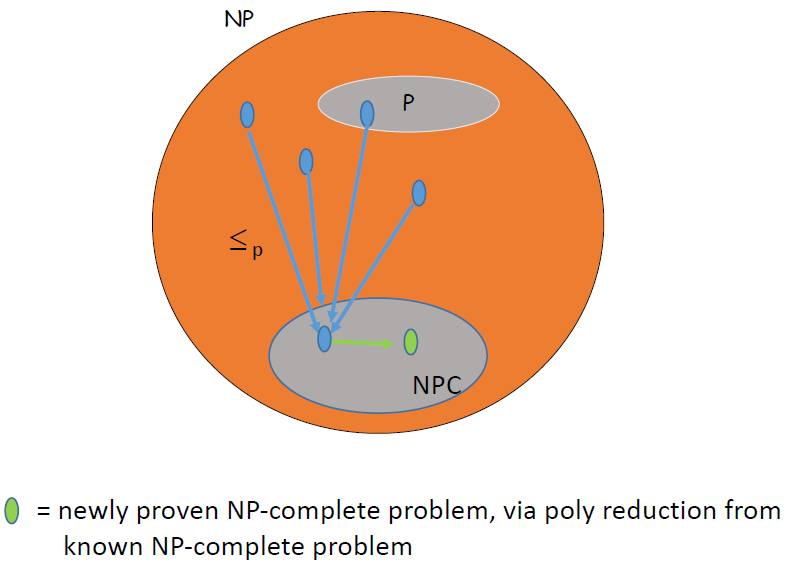
 **Pre-assigned then Assigned Inputs**

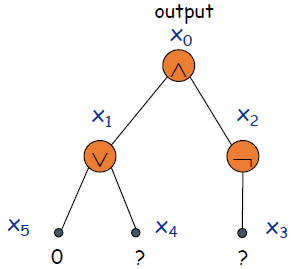
* 1. Note: equivalent to SAT problem, since every Boolean formula can be expressed as a Boolean circuit and vice-versa (will not present reductions here)
  2. **She says that all you need to know is that problem is in NP and that a Boolean circuit has a true value iff you have a solution to the problem, but notion here is that Boolean circuit is generally enough for which you could do this without knowing exactly what problem you had at hand – all you need is that the problem is in NP and a certifier exists to be able to do the reduction**

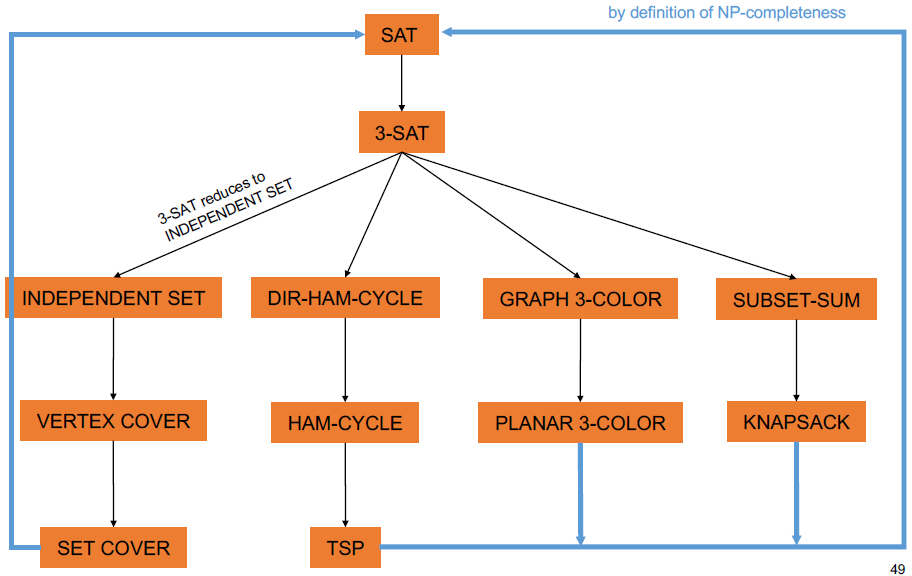
1. The “First” NP-Complete Problem
   1. Theorem: CIRCUIT-SAT is NP-Complete. [Cook 1971, Levin 1973]
   2. **She says once Cook gave us SAT as NP-Complete our life became easier because the transitivity property allows us to show that SAT reduces in polynomial time to another problem then the other problem has been proven to be NP-Complete, because every problem in NP reduces to SAT and you concatenate all of those solutions with the solution to the new problem then you have a polynomial solution to all problems including the new problem. NOTE (HUGE): that we reduce the known NP-Complete problem to a NEW problem because that shows that every other NP problem reduces to the NEW problem, and if we go the reverse direction and show that the NEW problem reduces to the know NP-Complete problem we have shown nothing because we already know the known NP-Complete problem is NP-Complete!**
   3. Proof: (sketch)
      1. (CIRCUIT-SAT is in NP): any algorithm that takes a fixed number of bits n as input and produces a yes/no answer can be represented by such a circuit. Moreover, if algorithm takes poly-time, then circuit is of poly-size.
      2. Sketchy part of proof: fixing the number of bits is important, and reflects basic distinction between algorithms and circuits
      3. (Poly time reduction from any X in NP): consider some problem X in NP. It has a poly-time certifier C(s, t). To determine whether s is in X, need to know if there exists a certificate t of length p(|s|) such that C(s, t) = yes
      4. View C(s, t) as an algorithm on |s| + p(|s|) bits (input s, certificate t) and convert it into a poly-sized circuit K
         1. First |s| bits are hard-coded with s
         2. Remaining p(|s|) bits represent bits of t
      5. Circuit K is satisfiable iff C(s, t) = yes
   4. Example: construction below creates a circuit K whose inputs can be set so that K outputs true iff graph G has an independent set of size 2



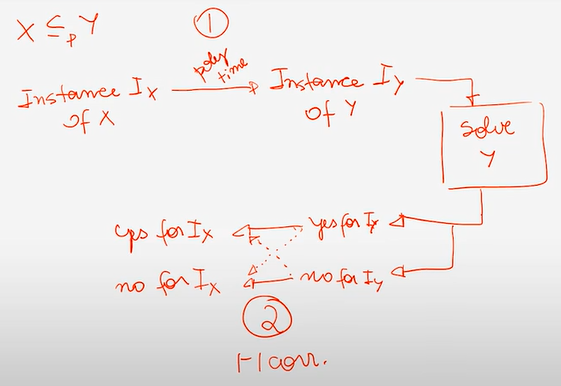
1. Establishing NP-Completeness => **you reduce TO the problem you are trying to prove is NP-Complete**
   1. Remark: once we establish first “natural” NP-complete problem, others fall like dominoes
   2. Recipe to establish NP-Completeness of problem Y:
      1. **Step 1:** show that Y is in NP **If you do not show this but only show Steps 2-3 you have shown NP-Hard**
         1. **You do this by giving a nondeterministic poly-time algorithm or showing Y has a certificate and verifier that must be polynomial in run time**
      2. **Step 2:** choose a known NP-complete problem X
         1. Cook showed that 3-SAT is in NP-Complete so 3-SAT is the choice ultimately – but pick your favorite and we saw that transitivity says 3-SAT ≤p INDEPENDENT-SET ≤p VERTEX-COVER ≤p SET-COVER
      3. **Step 3:** prove that X ≤p Y Direction of reduction is important! If other way around not showing anything!
      4. **Karp Step 3: consider an arbitrary instance sx of problem X, and show how to construct, in polynomial-time, and instance sy of of problem Y that satisfies the following properties:**
         1. **If sx is a “yes” instance of X, then sy is a “yes” instance of Y**
         2. **If sy is a “yes” instance of y, then sx is a “yes” instance of X i.e. sy and sx have the same answer**
   3. Justification: if X is an NP-complete problem, and Y is a problem in NP with the property that X ≤p Y then Y is NP-complete
   4. Proof: let W be any problem in NP. Then W ≤p X ≤p Y W ≤p X by def of NP-complete and X ≤p Y by assumption
      1. By transitivity, W ≤p Y
      2. Hence Y is NP-complete

**NPC problem here via Cook = CIRCUIT-SAT so if we can show that any other problem in NP reduces to CIRCUIT-SAT then we will have shown that problem is NP-Complete**

1. 3-SAT is NP-Complete **Recall that 3-SAT ≡p INDEPENDENT-SET ≡p VERTEX-COVER ≡p SET-COVER so if 3-SAT is shown to be NP-Complete the other two problems will also have been shown to be NP-Complete**
   1. Theorem: 3-SAT is NP-Complete
   2. Proof: Suffices to show that CIRCUIT-SAT ≤p 3-SAT since 3-SAT is in NP
      1. Let k be any circuit
      2. Create a 3-SAT variable xi for each circuit element i
      3. Make circuit compute correct values at each node:
         1. X2 = ┐x3 => add 2 clauses: x2 v x3, ┐x2 v ┐x3
         2. X1 = x4 or x5 => add 3 clauses: x1 v ┐x4, x1 v ┐x5, ┐x1 v x4 v x5
         3. X0 = x1 and x2 => add 3 clauses: ┐x0 v x1, ┐x0 v x2, x0 v ┐x1 v ┐x2
      4. Hard-coded input values and output value
         1. X5 = 0 => add 1 clause: ┐x5
         2. X0 = 1 => ad 1 clause: x0
      5. Final step: turn clauses of length < 3 into clauses of length exactly 3
2. Proving NP-Completeness
   1. Observation: all problems below are NP-complete and polynomially reduce to one another! (could have used CIRCUIT-SAT instead of SAT below)



If you always want to be able to shows that X ≤p Y:

Step 2 needs one 1:1 correspondence so pick either solid or dotted lines but not both

1. Some NP-Complete Problems
   1. Six basic genres of NP-Complete problems and paradigmatic examples:
      1. Packing problems: SET-PACKING, INDEPENDENT SET
      2. Covering problems: SET-COVER, VERTEX-COVER
      3. Constraint satisfaction problems: SAT, 3-SAT
      4. Sequencing problems: HAMILTONIAN-CYLCE, TSP
      5. Partitioning problems: 3D-MATCHING, 3-COLOR
      6. Numerical problems: SUBSET-SUM, KNAPSACK
   2. Practice: most NP problems are either known to be in P or NP-Complete
   3. Notable exceptions: Factoring, graph isomorphism (special case of isomorphism “easier” to solve), Nash equilibrium

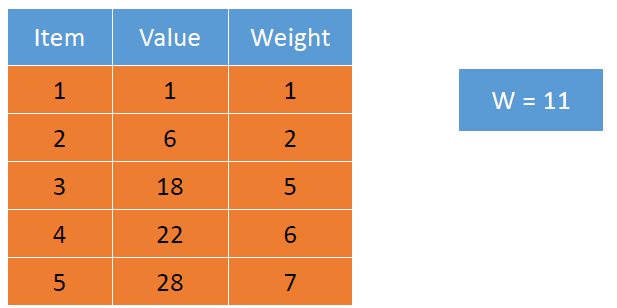
**Unit 8: Introduction to Approximation and Randomized Algorithms**

**Module 1:** **Approximation Algorithms** – for solving an NP-Hard problem where a poly-time algorithm is unlikely

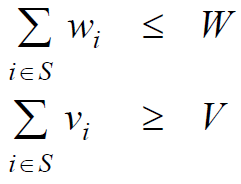
1. Knapsack problem is applicable to cache of memory on a server
   1. If you run exponential algorithm it could take so long that the solution is no longer relevant, so
2. Must sacrifice one of 3 desired features:
   1. Solve problem to optimality
   2. Solve problem in poly-time
   3. Solve arbitrary instances of the problem
3. (ρ = rho) ρ-approximation algorithm – may not get optimal solution, but
   1. Guaranteed to run in poly-time
   2. Guaranteed to solve arbitrary instance of the problem
   3. Guaranteed to find solution within ratio ρ of true optimum – factor has to be between 0 and 1 for maximization but for minimization (*e.g*. VERTEX-COVER) factor could be 2 – formally bound by ρ not just “hand waiving”
4. Challenge = need to prove a solution’s value is close to optimum, without even knowing what optimum value is!
5. We will look at Knapsack as an example, but there are other approximation algorithms, such as:
   1. Load Balancing – she says slides available but not presenting this
   2. Traveling Salesman Problem (TSP) – similar to Hamiltonian Cycle Problem = path touches all nodes w/o cycles
      1. TSP has weighted edges so equivalent to minimum cost Hamiltonian cycle = NP-Complete
      2. Pretty good approximation algorithm based on Minimum Spanning Tree (MST) where we know:
         1. Weight of MST ≤ **weight of TSP** ≤ 2 \* weight of MST because if you follow MST w/2 \* each edge = solve
            1. Weight of MST ≤ weight of TSP because MST is minimum and might be solution, but if you take out the highest weight edge of a TSP cycle you end up with a MST

**Module 2:** **Fully Polynomial Time Approximation Scheme (FPTAS)** for Knapsack Problems (page 644) – *see* PTAS w/o F

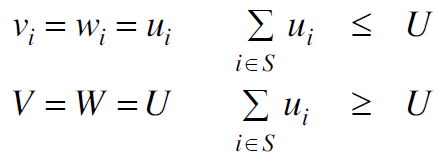
1. (not fully) PTAS = (1 + ɛ) – approximation algorithm for any constant ɛ > 0 (works for many values of ɛ)
   1. Load Balancing (Hochbaum – Shmoys 1987) – admits PTAS but what is presented on slides is not a PTAS
   2. Euclidean TSP (Arora 1996) – admits PTAS but she is not talking about it
   3. Knapsack
2. Consequences – PTAS produce arbitrarily high solution quality, but trades off accuracy for time
   1. You can pick an ɛ such as 0.0001 and get a result that is within 1.0001 of optimal in polynomial time
   2. Whether algorithm relies on ɛ is what makes it PTAS or FPTAS, but we are not delving deep into approximation algorithms in this class, merely touching the surface
3. Knapsack Problem – PTAS via rounding and scaling
   1. Given n objects and a knapsack
   2. Item i has value vi > 0 and weight wi > 0 we assume wi ≤ W
   3. Knapsack can carry weight up to W
   4. Goal: fill knapsack so as to maximize total value and not violate weight capacity of knapsack
   5. Example: { 3, 4 } has value 40



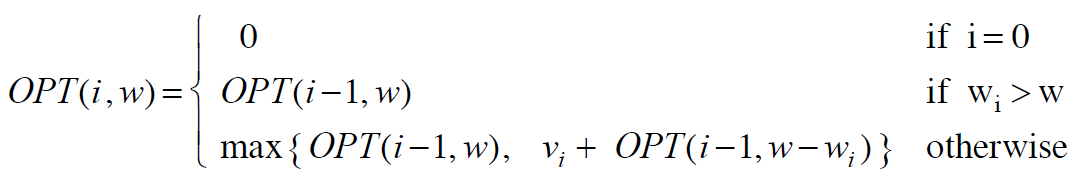
1. Knapsack is NP-Complete – proof is based on reduction of SUBSET-SUM to KNAPSACK
   1. KNAPSACK: given a finite set X, nonnegative weights wi, nonnegative values vi, a weight limit W, and a target value V, is there a subset such that:



* 1. SUBSET-SUM: given a finite set X, nonnegative value ui, and an integer U, is there a subset S ⊆ X whose elements sum up exactly to U?
  2. Claim: SUBSET-SUM ≤p KNAPSACK – proof: given instance (u1, . . . , un, U) of SUBSET-SUM, create KNAPSACK instance:

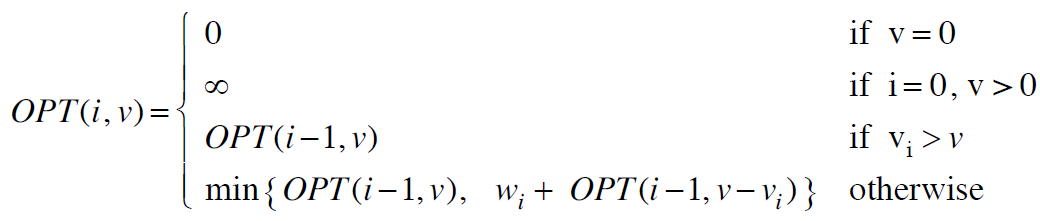
 so they must be equal

1. PTAS (not fully) for Knapsack Dynamic Programming I – **this is the algorithm we saw before for Knapsack**
   1. OPT(i, w) = max value subset of items 1, . . . , i with weight limit w **(based on subset of item WEIGHTS) here we maximize value and weight is a constraint**
      1. Case 1: OPT does not select item i, but rather selects best of 1, . . . , i-1 using up to weight limit w
      2. Case 2: OPT selects i
         1. new weight limit = w – wi
         2. OPT selects best of 1, . . . , i-1 using up to weight limit w



| Case 1 | | Case 2 | row 2 is special case for item weight

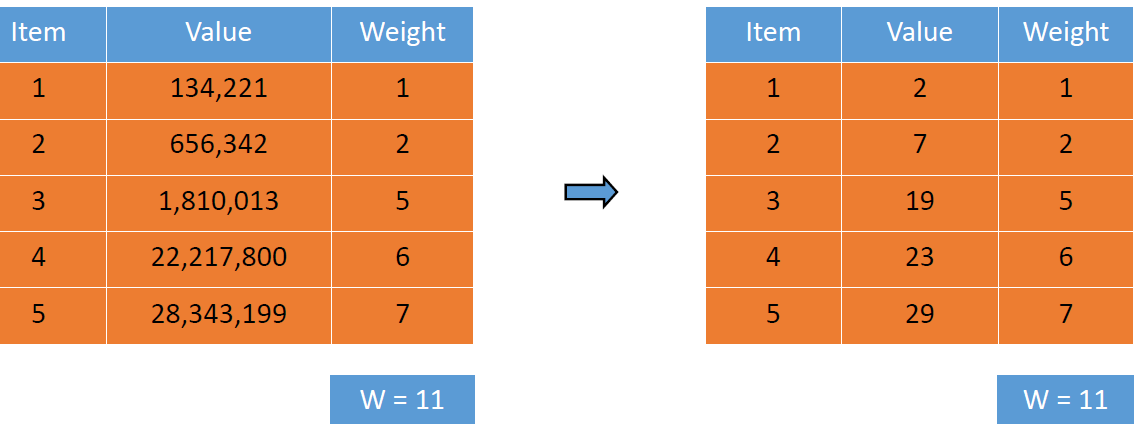
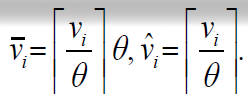
* 1. Running time: O(n W) where W = weight limit
  2. NOT polynomial in input size due to log(W) bits input makes this an exponential function on W not poly-time
  3. ALTERNATIVE DYNAMIC PRGAMMING II FRAMEWORK FOR KNAPSACK = similar but based on value not weight and this tweak ends up being the basis for the polynomial FPTAS algorithm
  4. OPT(i, v) = min weight subset of items 1, . . . i that yields value **exactly** v **(based on subset of item VALUES) we want to find the subset that has the smallest possible weight (as opposed to the highest value above)**
     1. Case 1: OPT does not select item i, but rather OPT selects best of 1, . . . , i – 1 that achieves exactly value v
     2. Case 2: OPT selects item i
        1. Consumes weight wi, new value needed = v – vi
        2. OPT selects best of 1, . . . , i – 1 that achieves exactly value v – vi



| Case 1 | | Case 2 | row 3 is special case for item value

* 1. Running time: O(n V\*) = O(n2 vmax) where V\* = number of columns and V\* ≤ n vmax sub into O(n V\*) = O(n2 vmax)
     1. V\* = optimal value = maximum v such that OPT(n, v) ≤ W NOT polynomial in input size, *e.g*. 2n input
     2. This II algorithm is bad when vmax is very large compared to number of items you have (*see* orange table below) – consider putting values on a line and one has the value 2n then the line is going to be very sparsely populated – whereas if all items had small values this would run in polynomial-time

1. Knapsack FPTAS: intuition for approximation algorithm: recall that ɛ = precision parameter
   1. Round all values up to lie in smaller range in order to run in polynomial-time – involves scaling factor Θ
   2. Run dynamic programming algorithm on rounded instance
   3. Return optimal items in rounded instance – NOTE that large, unrounded values on left table result in a huge number of rows in your dynamic programming table, many of which have ∞ as there will be no solution

| rounding |

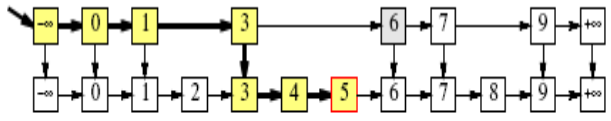
She says vi ≤ vibar ≤ vi + 1

* + 1. vmax = largest value in original instance ɛ = precision parameter Θ = scaling factor
  1. SLIDES 37-38 have Knapsack FPTAS notation and she discusses scaling on video 16:00-20:00
  2. Running time: O(n3 / ɛ)

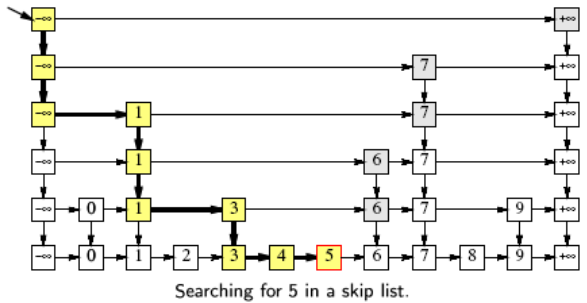
1. Proof FPTAS for Knapsack works – SLIDE 38 – 25:00+ on video
   1. ρ = (1/(1+ɛ)) \* OPT

**Module 3: Randomization:**

1. Skiplists – have the desirable properties of balanced binary search trees, but their structure is simpler to maintain
   1. Shortcuts: at a high level a skip list is just a sorted, singly linked list with some shortcuts
   2. Instead of a search being O(n) for a linked list in the worst case, to speed up the process we make a second-level list that contains roughly half the items for the original list
      1. For each item in the original list, we duplicate it with probability ½
      2. String together all duplicates into a second sorted linked list
      3. Add a pointer from the duplicate back to the original
      4. Just to be safe, we also add sentinel nodes at the beginning and end of both lists
      5. Now we can find a value x in this augmented structure using a two-stage algorithm:
         1. First, we scan for x in the shortcut list, starting at the -∞ sentinel node. If we find x we are done. Otherwise, we reach some value bigger than x and now we know x is not in the shortcut list
            1. Let w be the largest item less than the x in the shortcut list
         2. In the second phase, we scan for x in the original list, starting from w. Again, if we reach a value bigger than x, we know x is not in the data structure
      6. Since each node appears in the shortcut list with a probability ½, the expected number of nodes examined in the first phase is at most n/2.
      7. Only one node examined in the second phase has a duplicate (I think this is w, but not if target not present)
         1. **Jamison said note this is about SECOND phase so this is tied together with expected 2 more node will be touched, so what they mean here is w or 3 in the diagram below is the only node duplicated**
      8. The probability that any node is followed by k nodes without duplicates is 2-k
         1. The expected number of nodes examined in the second phase is at most 1 + Σk>0 2-k = 2 (here 4 and 5):

 Σk>0/∞ 2-k = 1 as standard series

* + - * 1. here that the summation of k starts > 0 so here is ends up being ½ so Σk> ½ 2^-½ which = 1 so 1 + 1 = 2
    1. Thus, by adding these random shortcuts, we have reduced the expected cost of a search from n to (N/2) + 2, roughly a factor of two in savings – going from O(n) to ???
  1. Idea: add shortcuts to the shortcuts, and repeat recursively
     1. For each node in the original list we flip a coin over and over until we get tails
     2. For each heads we make a duplicate of the node. The duplicates are stacked up in level and the nodes on each level are strung together in sorted linked lists
     3. Each node v stores a search key (Key(v)), which is a pointer to its next lower copy (down(v)), and a pointer to the next node in its level (right(v)).
  2. Searching: similar procedure for inserting or deleting
     1. Starting at leftmost node L in the highest level, we scan through each level as far as we can without passing the target value x
     2. The search ends when we either reach a node with search key x or fail to find x on the lowest level
     3. Proceed down to the next level

edges can be shortcuts on top of original list

* 1. Expected performance bounds:
     1. Intuitively, since each level of the skip list has about half the number of nodes as the previous level, the total number of levels should be about O(log n)
     2. Similarly, each time we add another level of random shortcuts to the skip list, we cut the search time in half except for constant overhead
     3. So, after O(log n) level, we should have a search time of O(log n)
  2. Showing depth is O(log n) with high probability
     1. “High Probability” is a technical term that means the probability is at least 1 – 1/nc for some constant c ≥ 1
     2. In order for a key x to appear on the kth level, we must have flipped k heads in a row, so Pr[L(x) ≥ k] = 2-k
     3. In particular (for k = 2 log n), Pr[L(x) ≥ 2 log n] = 1/n2
     4. Since Pr[A V B] ≤ Pr[A] + Pr[B] for any random events A and B, we have Pr[L ≥ 2 log n] ≤ Σ1-n Pr[L(x) ≥ 2 log n] = Σ1-n 1/n2 = 1/n
     5. So w/high probability, a skip list has O(log n) level. That is, Pr[L < 2 log n] ≥ 1 – 1/n which tends to a as n -> ∞
  3. Description of running algorithm in reverse on SLIDES 56-57
  4. Putting it all together:
     1. Running algorithm backwards allows showing that E[L] = Ჲ(log n)
     2. In a series of coin flips where PR[Head] = PR[Tails] = ½ E[# of Heads] = E[# of Tails]
     3. Each time we do v <- up(v), v gets one level closer to L. Thus, (# of Heads) = L, implying that E[# of Heads] = L
     4. Hence, E[# of Tails] = E[# of Heads] = L
     5. Since, E[running time of FLIPWALK] = O(E[#of con flips]) and E[# of coin flips] = E[(# of Heads) + (# of Tails)] = E[#of Heads] + E[# of Tails]
     6. By linearity of expectation, we have: E[running time of FLIPWALK] = 2L = O(log n)

**The End of the Class**