Formal Definitions:

O-Notation: Set of functions bounded above (cg(n) grows faster and is higher on the graph than f(n))… g(n) is an asymptotic upper bound for f(n)

O(g(n)) = { f(n): there exist positive constants c and n0 such that 0 ≤ f(n) ≤ cg(n) for all n ≥ n0 }.

Ω-Notation: Bounded below (f(n) is higher on the graph than cg(n)), g(n) is an asymptotic lower bound for f(n)

Ω(g(n)) = { f(n): there exist positive constants c and n0 such that 0 ≤ cg(n) ≤ f(n) for all n ≥ n0 }.

Θ-Notation: Bounded above and below, O and Ω

Θ(g(n)) = { f(n): there exist positive constants c1, c2, and n0 such that 0 ≤ c1g(n) ≤ f(n) ≤ c2g(n) for all n ≥ n0 }.

Lgn = O(√n): lgn is bounded above by square root of n

Limit method of finding O, Ω, Θ:

If we have functions f(n) and g(n), we set up a limit quotient between f and g:

{ 0 Then f(n) = O(g(n))

Limn->infinity f(n)/g(n) = { c>0 Then f(n) = Θ(g(n))

{ +inf Then f(n) = Ω(g(n))

Log rules:

Logn3 = 3logn

Log3n = (logn)3

How to do log2 on calc: Log(value being calculated) THEN Ans/log(2)

Divide and Conquer: Divide, Conquer, Combine

Example is mergeSort(S, c) with a runtime of T(n) = 2T(n/2) + Θ(n) where 2 is the number of subproblems, n/2 is subproblem size, and Θ(n) is the work dividing and combining

Function mergeSort(S, c):

Input: sequence S with n elements, comparator c

Output: sequence S sorted according to c

If S.size() > 1

(S1, S2) 🡨 partition(S, n/2)

mergeSort(S1, c)

mergeSort(S2, c)

S 🡨 merge(S1, S2)

Binary Search is also divide and conquer with runtime of T(n) = 1T(n/2) + Θ(1) = Θ(lgn)

MinMax Divide & conquer:

**MIN-MAX(A)**

If |A|=1 then return min=max=A[0]

Divide A into two equal subsets A1 and A2

(min1, max1) = MIN-MAX(A1)

(min2, max2) = MIN-MAX(A2)

If min1 <= min2 then min = min1

Else min = min2

If max1 >= max2 then max = max1

Else max = max2

Return (min,max)

**Recurrence** T(n) = 2T(n/2) + 2

**Solution:** (n)

Master Method:  
Master Method applies to recurrences of the form T(n) = a T(n/b) + f(n)

Where a ≥ 1, b > 1, and f is asymptotically positive

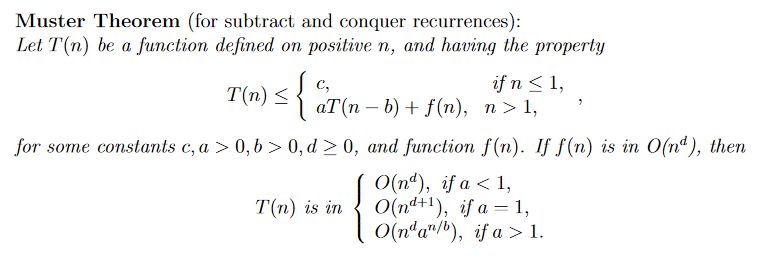
**case 1:** if f(n) = O(nlogba -ε) for some ε > 0, then: T(n) = Θ(nlogba)

**case 2:** if f(n) = Θ(nlogba), then: T(n) = Θ(nlogba lgn)

**case 3:** if f(n) = Ω(nlogba +ε) for some ε > 0, and if

af(n/b) ≤ cf(n) for some c < 1 and all sufficiently large n, then:

T(n) = Θ(f(n))



Iteration Example for Binary Search:

T(n) = c + T(n/2) T(n/2) = c + T(n/4)

= c + c + T(n/4) T(n/4) = c+ T(n/8)

= c + c + T(n/8)

Stop when n/2i = 1 🡪 I = lgn

T(n) = c + c + c + … + c + T(1)

= clgn + T(1)

= Θ(lgn)

Substitution Example for Binary Search:

Guess: T(n) = O(lgn)

Induction goal: T(n) ≤ clgn, for some c and n ≥ n0

Induction hypothesis: T(n/2) ≤ c lg(n/2)

Proof of induction goal:

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

≤ c lg(n/2) + k

≤ c (lg(n)-lg2) + k

≤ c lgn **– c + k**

≤ c lgn – (c – k)

if: c - k ≥ 0 => c ≥ k

T(n) ≤ c lgn

T(n) = O(lgn)

Substitution: T(n) = T(n-1) + n

Guess: T(n) = O(n2)

–Induction goal: T(n) ≤ c n2, for some c and n ≥ n0

–Induction hypothesis: T(n-1) ≤ c(n-1)2 for all n < n

•Proof of induction goal:

T(n) = T(n-1) + n ≤ c (n-1)2 + n

T(n) ≤ cn2 – 2cn + c + n

≤ cn2 – (2cn – c - n)

if: 2cn – c – n ≥ 0 <-> c ≥ n/(2n-1) <-> c ≥ 1/(2 – 1/n)

T(n) ≤ cn2

–For n ≥ 1  2 – 1/n ≥ 1 🡪 any c ≥ 1 will work

Therefore: T(n) = O(n2)

Substitution Example 4

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

•Guess: T(n) = O(nlgn)

–Induction goal: T(n) ≤ cn lgn, for some c and n ≥ n0

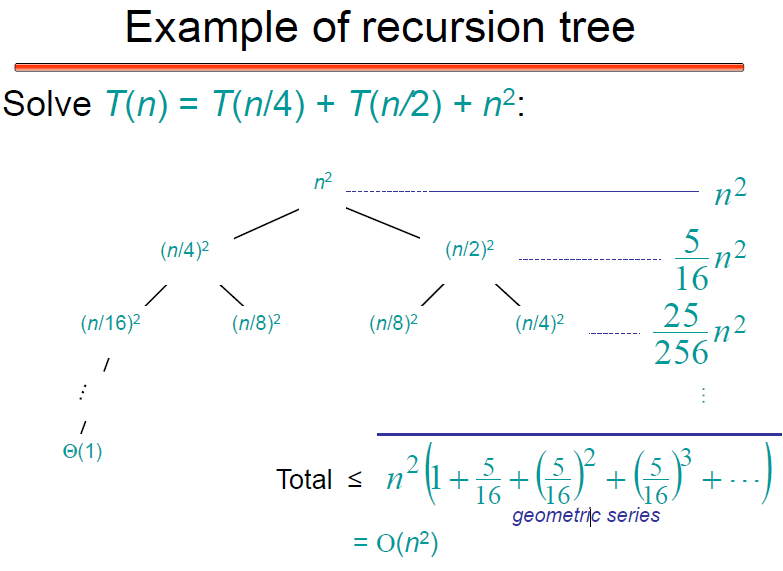
–Induction hypothesis: T(n/2) ≤ cn/2 lg(n/2)

•Proof of induction goal:

T(n) = 2T(n/2) + n ≤ 2c (n/2)lg(n/2) + n

= cn lgn – cn + n ≤ cn lgn

if: - cn + n ≤ 0 🡪 c ≥ 1



Dynamic Programming:

Like divide and conquer, DP solves problems by combining solutions to subproblems, but unlike D&C, subproblems are reused and are not independent

5 steps: Define subproblems, guess part of the solution, relate subproblem solutions, recurse + memorize or build a DP bottom-up table, solve original problem

Common DP examples: Fibonacci, binomial coefficients, longest common subsequence, longest increasing subsequence, knapsack, shortest path, chain matrix multiplication, edit distance, rod cutting, optimal BST

DP Example for Fibonacci: Runtime is linear Θ(n) because all the values only need to be calculated once then memoized

Memo = {}

Fib(n) {

If (n in memo) { return memo[n] }

If (n <= 1) {

F = n;

} else {

F = fib(n-1) + fib(n-2)

}

Memo[n] = f;

Return f;

}

Bottom Up DP example for Fibonacci: Stores every value in the table rather than just those used

Fib = { }

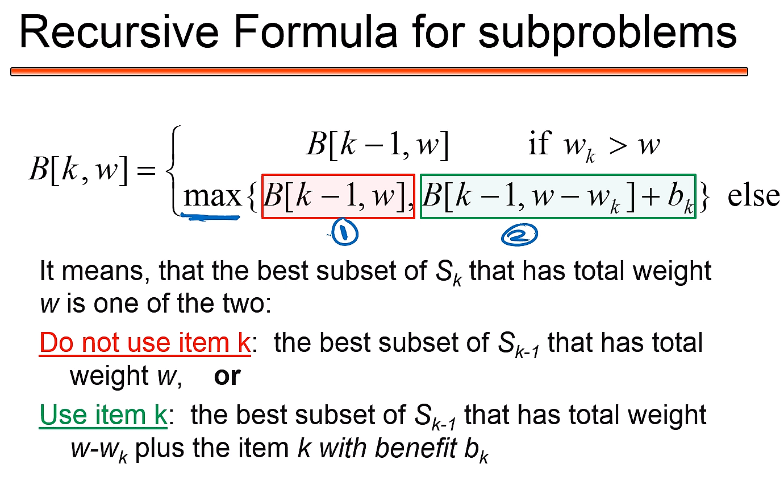
Fib[0] = 0;

Fib[1] = 1;

For k = 2 to n

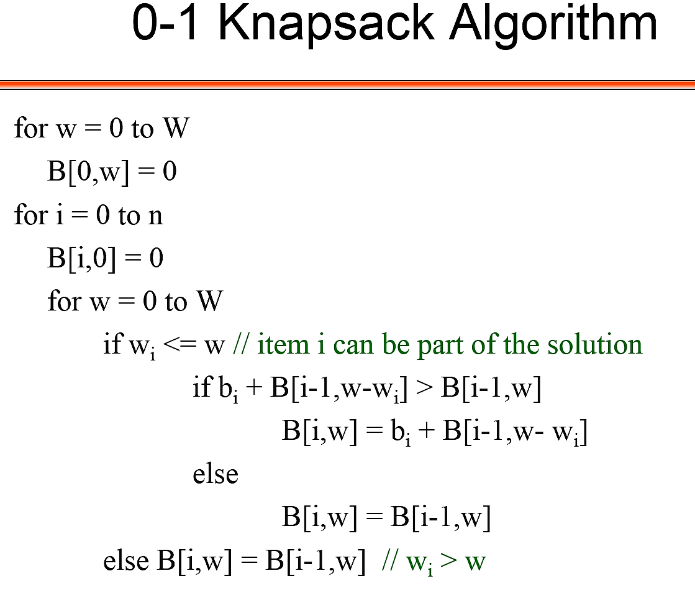
Fib[k] = fib[k-1] + fib[k-2];

Return fib[n];

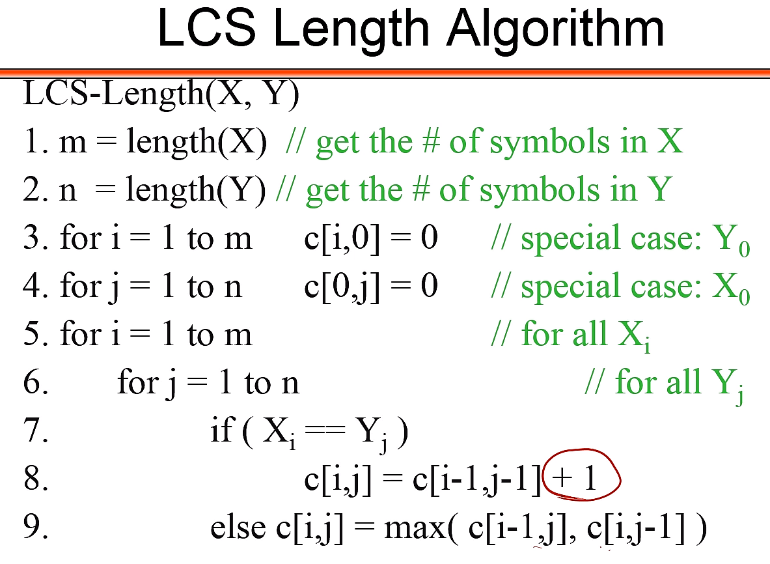


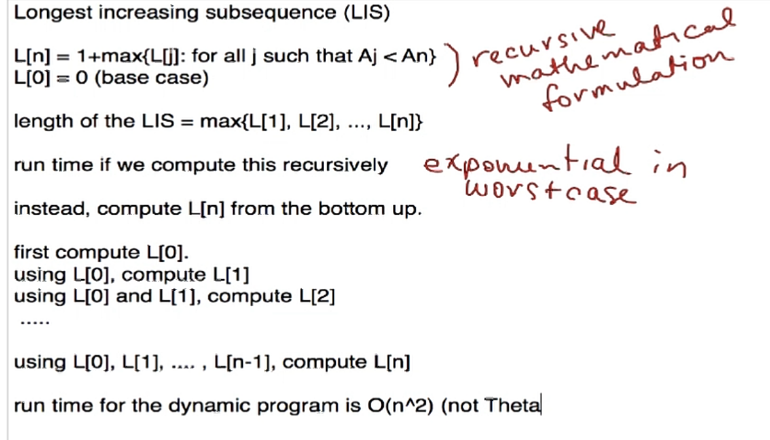
Knapsack DP example: Find best benefit per weight (W)

items for a bag, runtime is O(nW) pseudo-polynomial



Longest common subsequence example: Runtime is Θ(m+n)





Function changedp(coins,amount)

minCoins = [0] \* (amount + 1)

coinsUsed = [0] \* (amount + 1)

numCoins = [0] \* length(coins)

for each changeSubproblem in (amount + 1)

coinsNeeded = changeSubproblem

if changeSubproblem == 0

lastCoinUsed = 0

else:

lastCoinUsed = 1

For each coin in coins

if coin <= changeSubproblem

if (1 + minCoins[changeSubproblem - coin] < coinsNeeded)

coinsNeeded = 1 + minCoins[changeSubproblem - coin]

lastCoinUsed = coin

minCoins[changeSubproblem] = coinsNeeded

coinsUsed[changeSubproblem] = lastCoinUsed

coin = amount

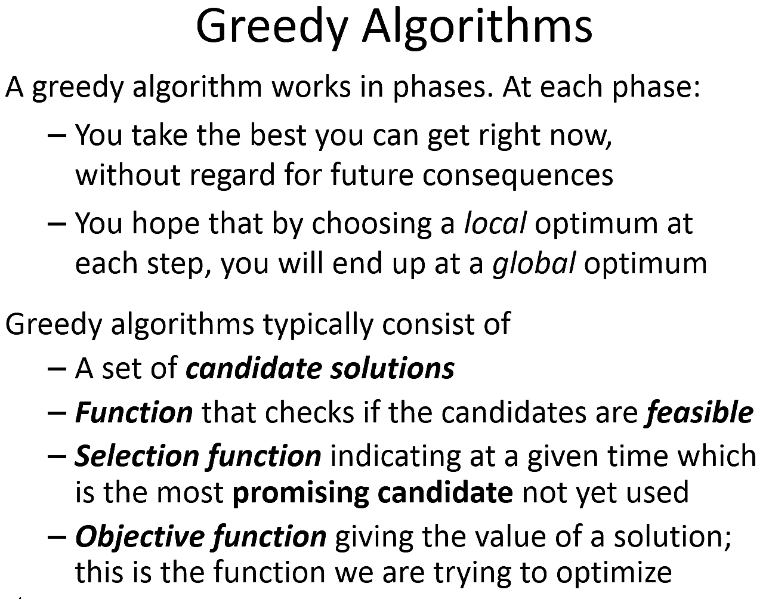
while coin > 0

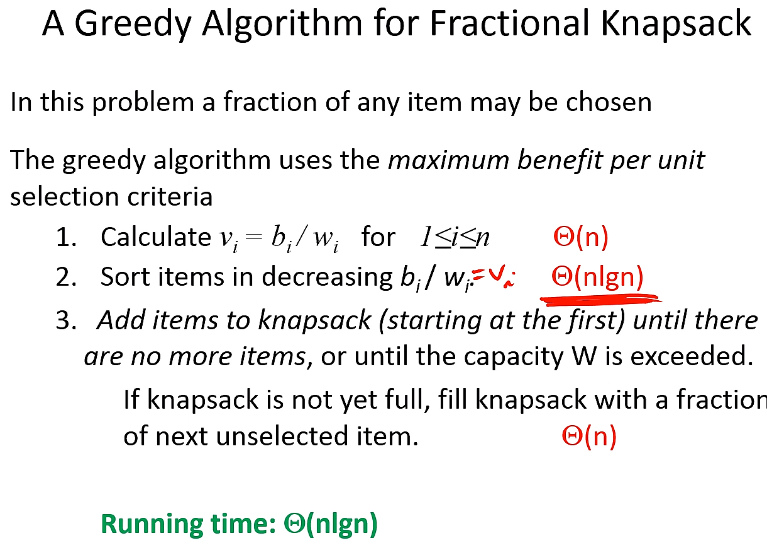
coinUsed = coinsUsed[coin]

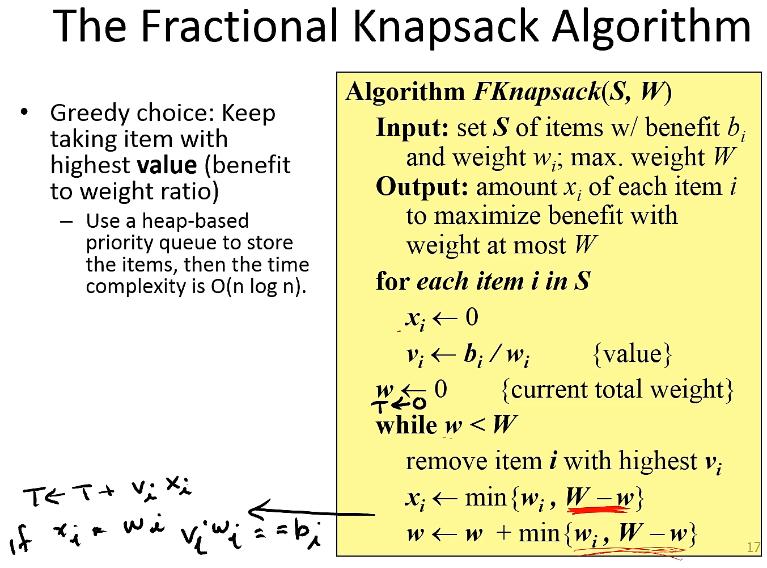
numCoins[coins.index(coinUsed)] += 1

coin -= coinUsed

return (numCoins, minCoins[amount])







Function changegreedy(coins, amount)

maxIndex ← coins.length - 1

coinCount ← zero filled array of size coins.length

numCoins ← 0

while amount > 0

if coins[maxIndex] <= amount

coinCount[maxIndex] ← amount / coins[maxIndex]

amount ← amount - (coins[maxIndex] \* coinCount[maxIndex])

numCoins ← numCoins + coinCount[maxIndex]

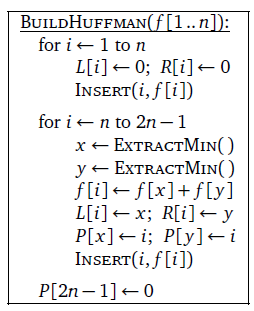
maxIndex ← maxIndex - 1

return (coinCount, numCoins)

Huffman code is O(n logn)

To actually implement Huffman codes efficiently, we keep the characters in a min-heap, where the priority of each character is its frequency. We can construct the code tree by keeping three arrays of indices, listing the left and right children and the parent of each node. The root of the tree is the node with index 2*n* - 1.

The algorithm performs *O*(*n*) min-heap operations. If we use a balanced binary tree as the heap, each operation requires *O*(log *n*) time, so the total running time of BuildHuffman is *O*(*n* log *n*).



Solutions to common recurrences:

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

Sequential Search: T(n) = T(n-1) + O(1) 🡪 O(n)

Tree-traversal: T(n) = 2T(n/2) + O(1) 🡪 O(n)

Quicksort: T(n) = 2T(n/2) + O(n) 🡪 O(n logn)

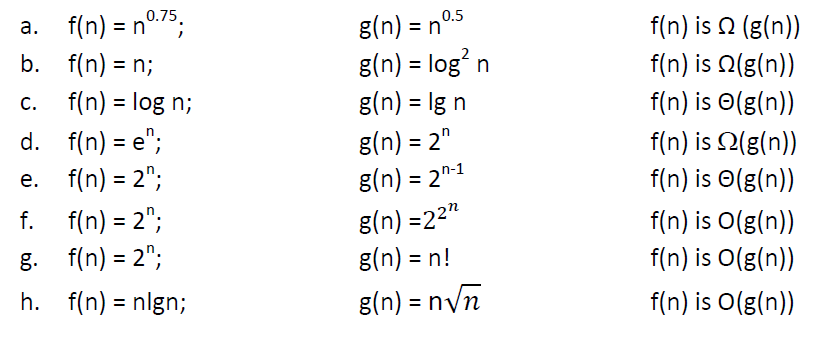
Selection sort: T(n) = T(n-1) + O(n) 🡪 O(n2)

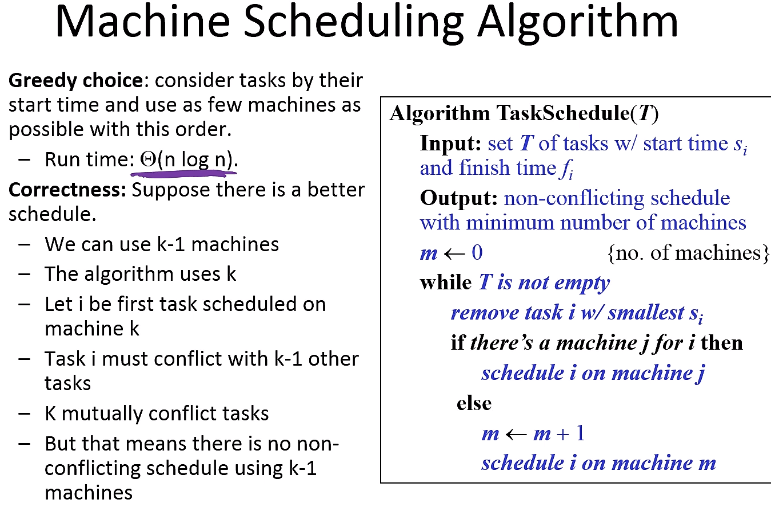
Towers of Hanoi: T(n) = 2T(n-1) + 1 🡪 O(2n)

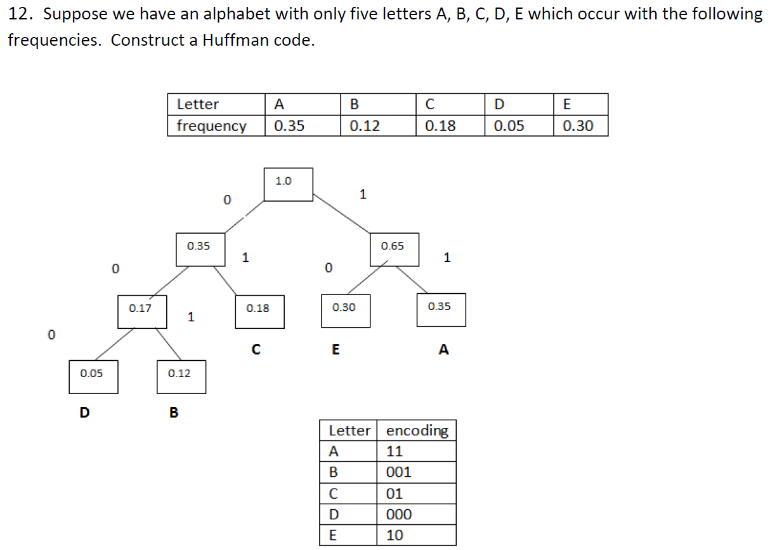
Insertion sort: O(n2)

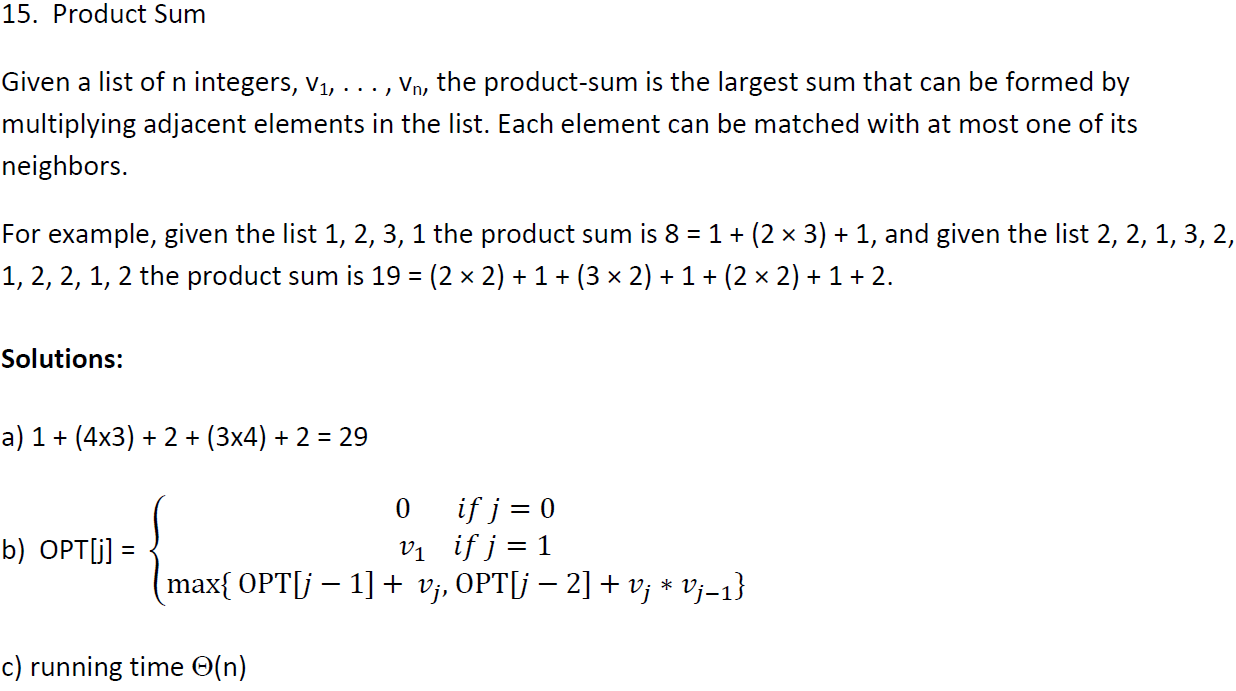
Merge sort: O(nlgn)

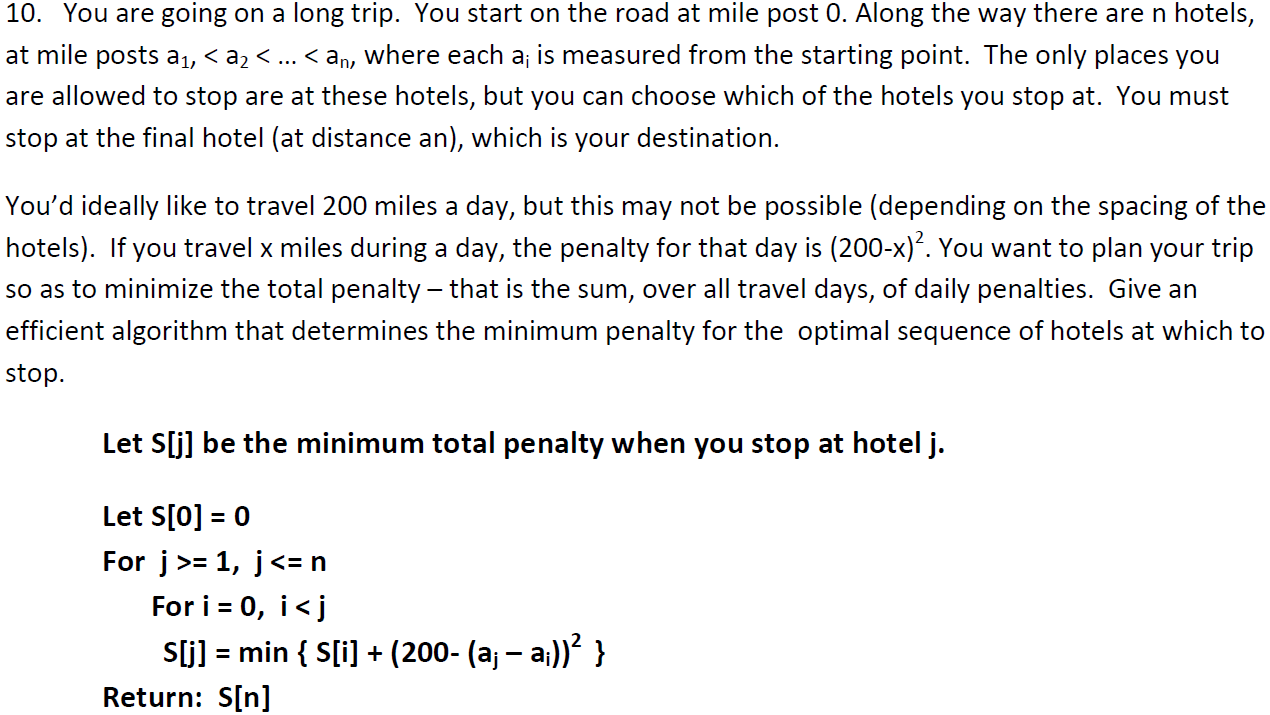
Naïve sort: O(n3)







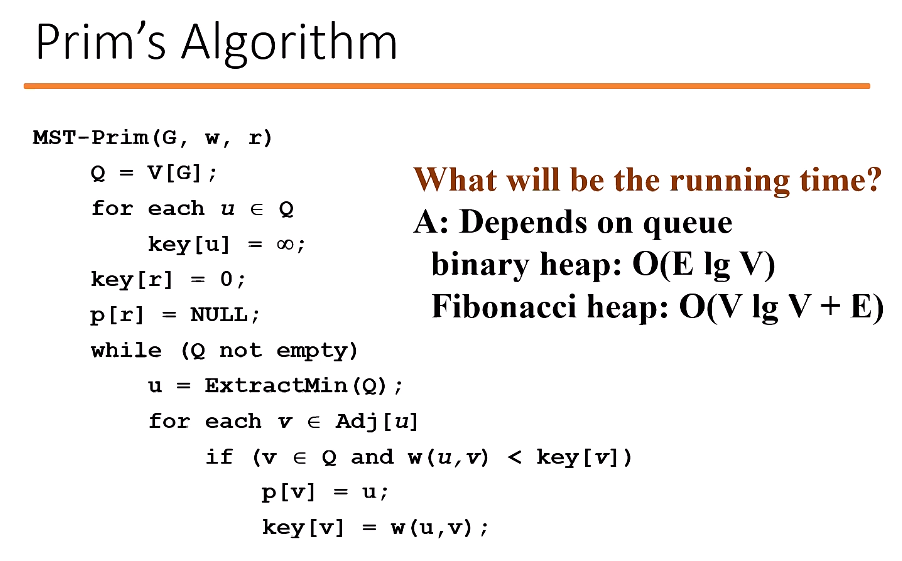




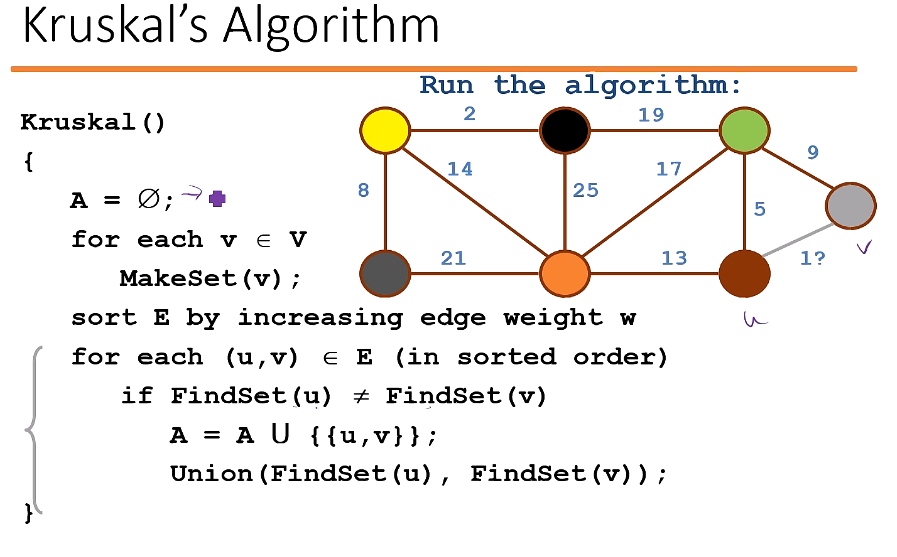
**Graph algorithms:**

**MST**: A tree that connects all vertices and minimizes weight, used on weighted graphs. Includes kruskal’s algorithm, prim’s algorithm. Prim extends a tree by including the cheapest outgoing edge, and Kruskal adds the cheapest edge that joins disjoint components.

Prim’s algorithm grows one tree T one vertex at a time.



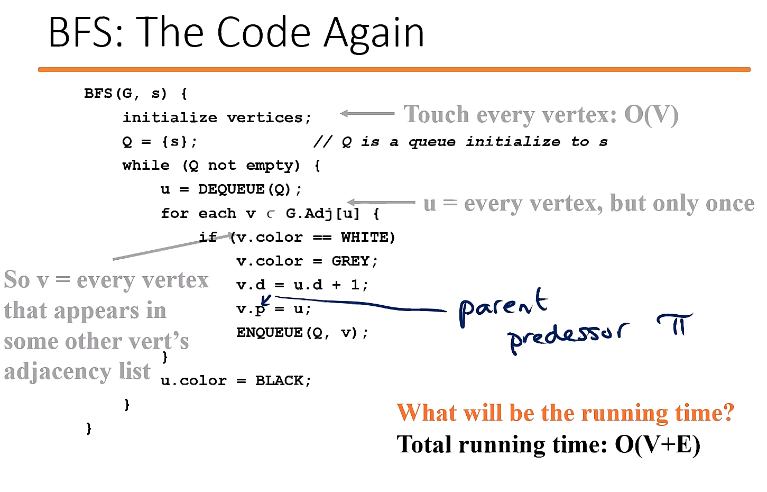
Kruskal’s algorithm is an edge based algorithm, adding edges one at a time in increasing weight order. Maintains a forest of trees. Running time of O(E lg V)



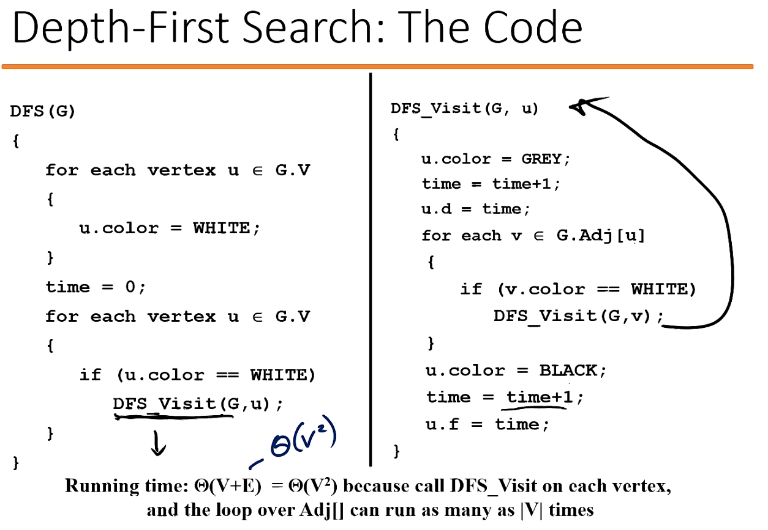
**BFS**: Starts at a vertex s, and discovers every vertex reachable from s. Produces a BFS tree with root s and all reachable vertices. Discovers vertices in increasing order of distance from s (distance between v and s is the minimum number of edges from s to v).

Does this by storing vertex colors. Initially all white, when discovered: grey, when processed: black.

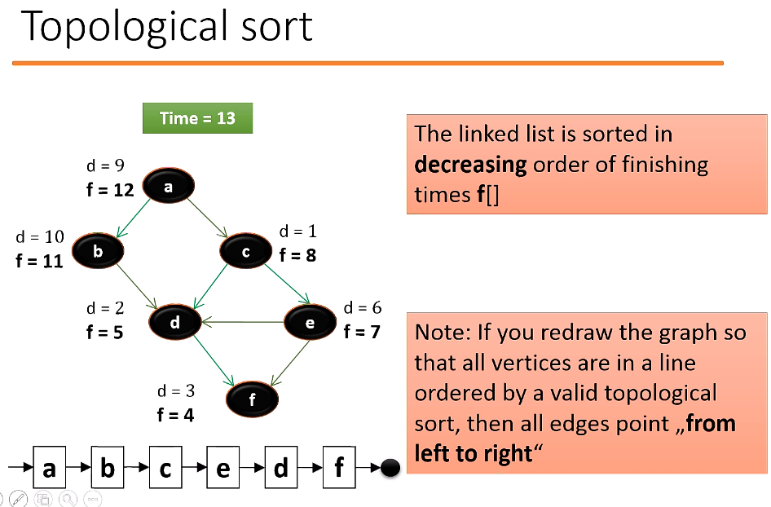
Calculates shortest path distance to source node.

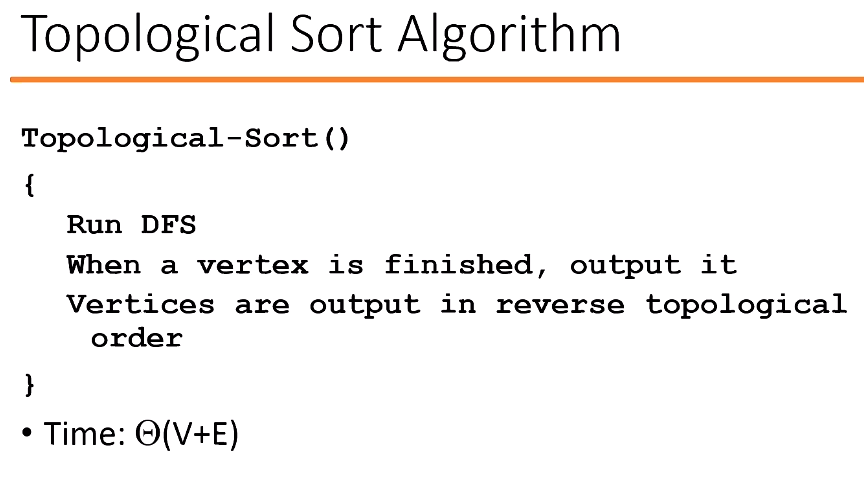


**DFS**: Explore deeper whenever possible. Edges are explored out of the most recently discovered vertex v that still has unexplored edges. When all of v’s edges have been explored, backtrack to the vertex from which v was discovered. Recursive algorithm that has white nodes as undiscovered, gray as discovered but unfinished, and black as finished. Tree edge: encounter new white vertex. Back edge: from descendent to ancestor.



**Topological Sort of a DAG**: Linear ordering of all vertices in a graph G such that vertex u comes before vertex v if edge (u,v) is in G. Precedence relations: an edge from x to y means you need to finish x before y. Intuition: can only schedule ones all subtasks have been scheduled.

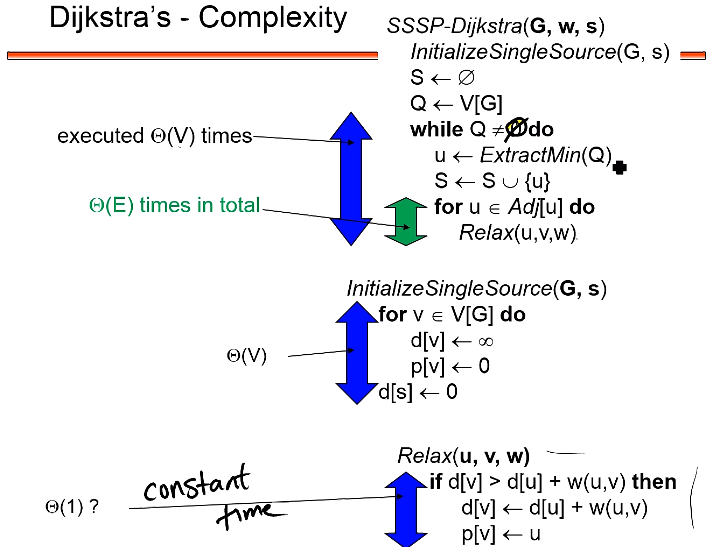


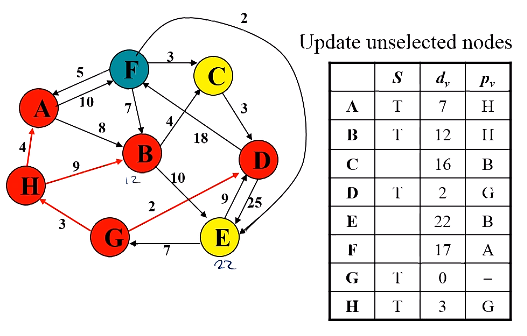


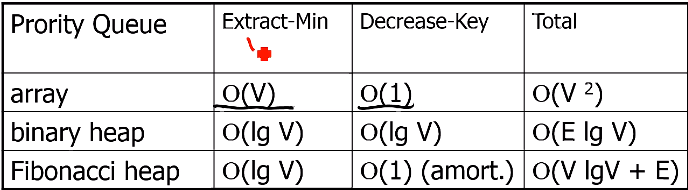
**Shortest Path:** Single source, single destination. The shortest path is a path of the minimum weight. All shortest path algorithms have relaxation:  
Relax (u,v,w)

Dijkstra’s algorithm works on directed and undirected graphs. All edges must have nonnegative weights. It is similar to BFS.

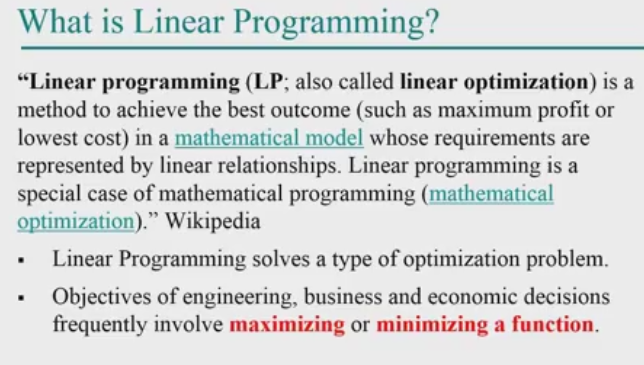
If (d[v] > d[u]+w) then d[v] = d[u] + w;

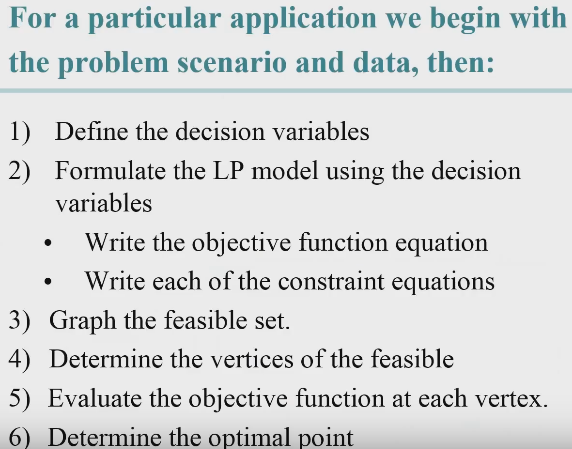




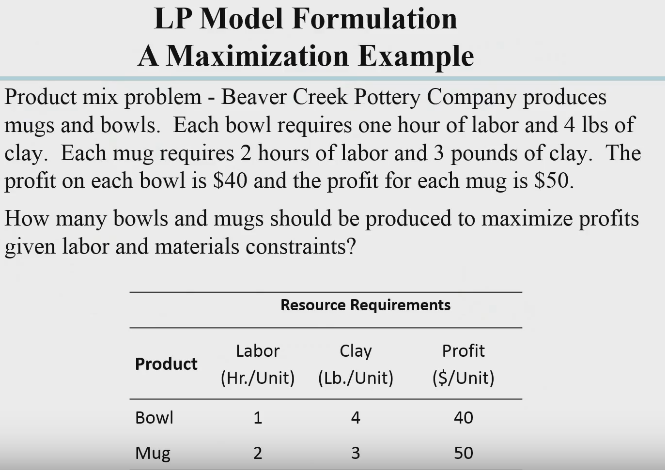


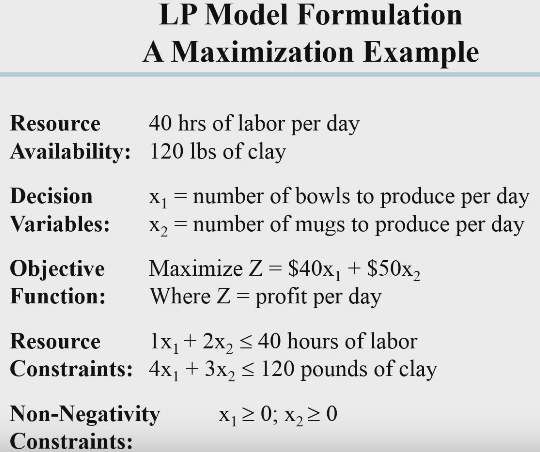
**Linear Programming**:



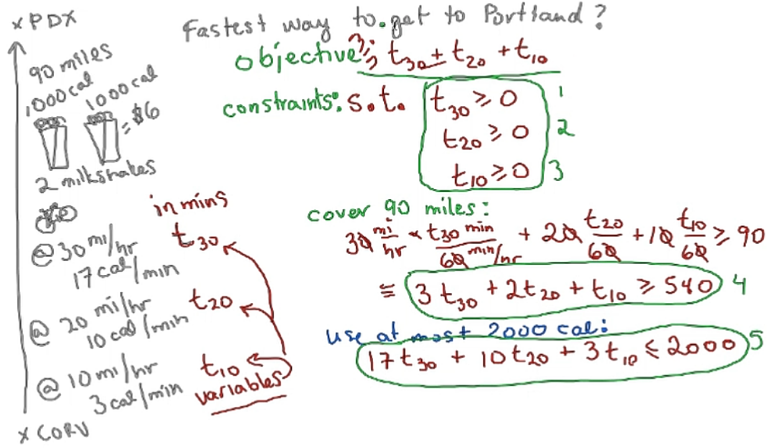


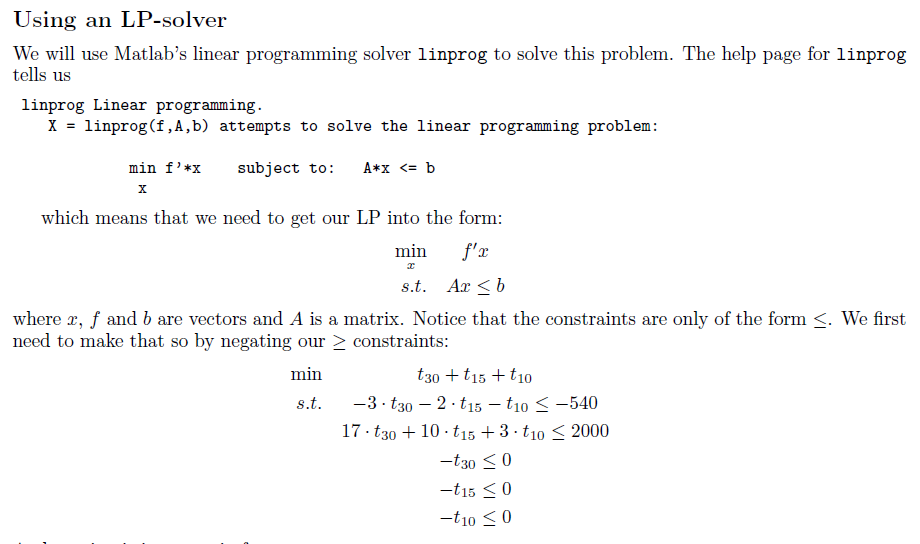
**Product Mix**: Finding combinations of items given resource constraints



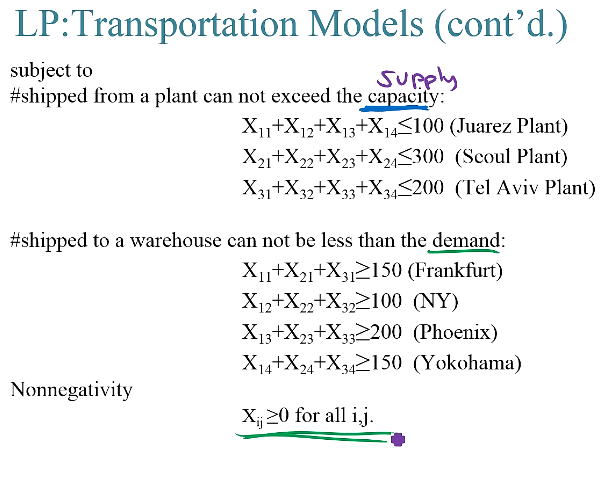


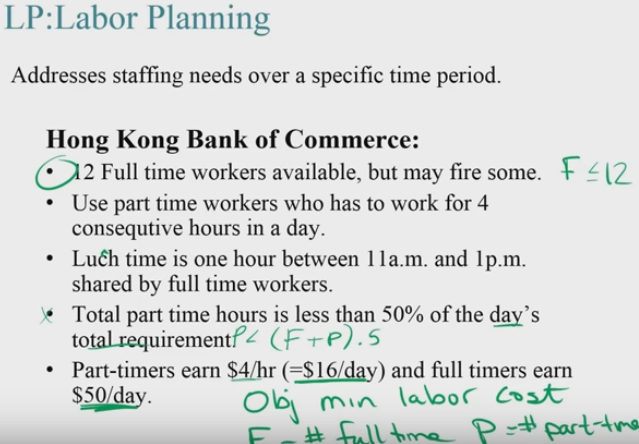
Feasible solutions violate no constraints. Infeasible solutions violate constraints.

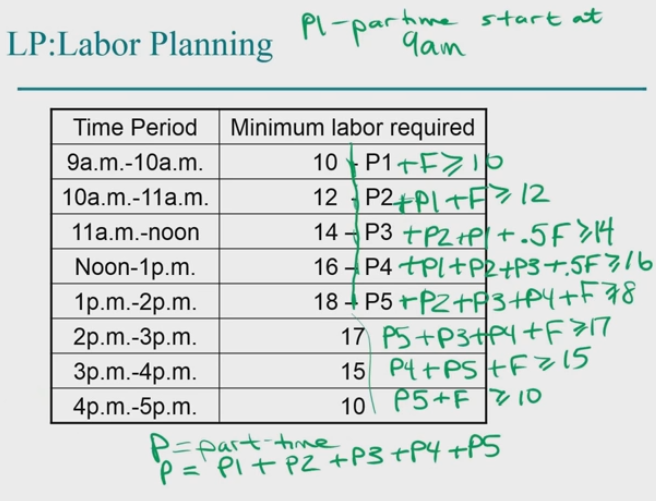


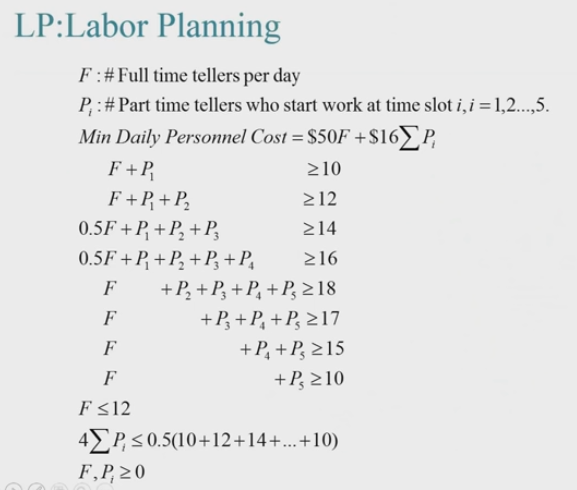


**Transportation Problems:**



**Scheduling**:





**Complexity classes:**

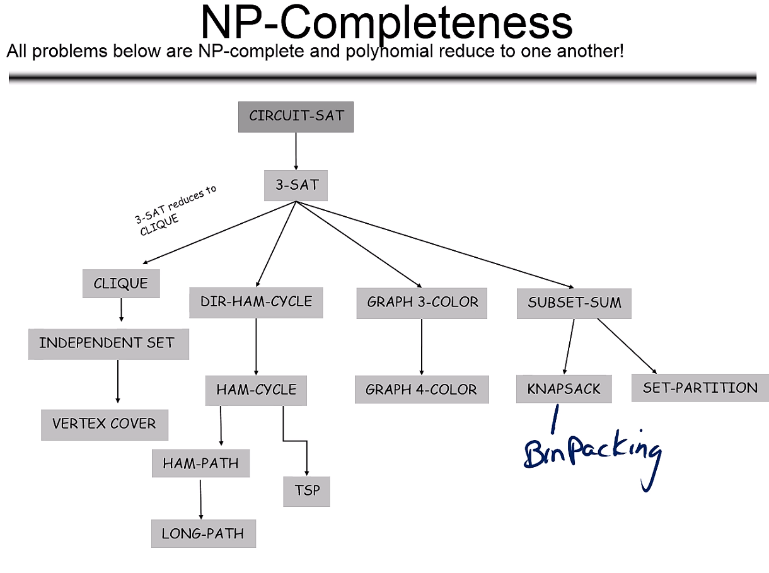
**P**: Polynomial, solvable in polynomial time

**NP**: Nondeterministic polynomial, solution is verifiable (not solvable) in polynomial time

**NP-Complete**: Intractable problems that as they grow large, they can’t be solved in a reasonable amount of time (cannot be solved in polynomial time). Problems in NP-Complete are in NP and NP-hard.

**NP-Hard:** At least as hard as the problems in NP. A problem is NP hard when every problem in NP can reduced in polynomial time to the problem.

Reductions: Reduction is a way of saying that one problem is “easier” than another. If A is no harder than B, then A<=B. A problem A can be reduced to another problem B if any instance of A can be “easily rephrased” as an instance of B, the solution to which provides a solution to the instance of A



**NP-Complete Problems:** Can be “solved” through approximation (almost optimal solution), randomization, restriction (restrict size of input to increase speed), parameterization, and heuristics (an algorithm that works reasonably well for most cases but has no proof that it works every time).

**CIRCUIT-SAT:** the decision problem of determining whether a given Boolean circuit has an assignment of its inputs that makes the output true

**3-CNF-SAT:** A Boolean formula that is an AND of clauses, each of which is an OR of exactly 3 distinct literals.



**CLIQUE**: computational problems of finding cliques (subsets of all vertices, all adjacent to each other, also called complete subgraphs) in a graph

**VERTEX-COVER**: a set of vertices such that each edge of the graph is incident to at least one vertex of the set

**HAM-CYCLE**: a closed loop through a graph that visits each node exactly once

**TSP**: find the shortest route that passes through each of a set of points once and only once then returns to starting vertex

**SUBSET-SUM**: given a set of integers, is there a non-empty subset whose sum is zero? (example: {-7, -3, -2, 5, 8}, the answer is yes because {-3, -2, 5})

**INDEPENDENT SET**: Set of vertices in a graph where no two vertices are adjacent

**LONG-PATH**: problem of finding a simple path of maximum length in a given graph (no repeated vertices)

**HAM-PATH**: problem of finding whether a Hamiltonian path exists in a graph. A ham path is a path in an undirected or directed graph that visits each vertex exactly once.

**K-COLOR**: a vertex coloring of a graph that is an assignment of one of k possible colors to each vertex of G such that no two adjacent vertices receive the same color.

**KNAPSACK**: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

**Bin Packing**: objects of different volumes must be packed into a finite number of bins or containers each of volume *V* in a way that minimizes the number of bins used.

**Approximation algorithms**: Algorithms used to find approximate solutions to optimization problems. Often used for NP-hard problems.

Analyzing experimental data:

**Linear Regression**: Finding the best fitting line through the points (y=mx+b)