CPSC 320 Final

Types of Problems

Boolean Satisfiability (SAT) Problem: NP Complete.

Given a series of truth assignments, is there a way to make the entire statement true?

$X_1 \vee \overline{X}_2 \vee X_3 \vee X_4$
X_5
$\overline{X}_1$
$X_2 \vee \overline{X}_3 \vee \overline{X}_5$
$\overline{X}_2 \vee X_3$

X1 = F
X5 = T
X2 = T
X3 = T
X4 = T

This satisfies the clause.
Certifying if it's correct
Plug in the values.
is polynomial, meaning it it's in **class NP.**
Solving the algorithm
Brute force:
1. Make list of variables $X_1, \ldots, X_n$
2. Try every assignment of truth values to these variables
3. Return YES if satisfies, NO otherwise.
4. Repeat for 2^n different assignments in worst case.

Worst case run time: $\Omega(n2^n)$

To convert non-decision problem to decision , add a random variable (k in the case of steiner).

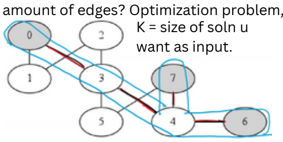
3-SAT Problem: Every clause must be exactly length 3.

Better for NP-completeness reductions, since it is constrained. (We will reduce from 3-SAT to problem we want to prove is NP-complete).

Steiner Problem (ST): Optimization Problem, NP Complete

Given the graph below, how do we connect the shaded parts (S) with least amount of edges? Optimization problem, minimizing edges connecting.

K = size of soln u want as input.



Decision Problem: Output can be either a yes or no. For Steiner, Yes is when it connects all shaded nodes, with edge amount < k. If u specify edge amount < any other solution, is NOT a decision problem.

An Instance of ST: $G = (V, E),\ S \subseteq V$, and k, where $1 \leq k \leq n$.

ST is good: Edges of E' (soln) connect all vertices in S, and |E'| < = K
ST is optimal: Edges of E' (soln) connect all vertices in S, |E'| < = the size of any other solution for the instance.

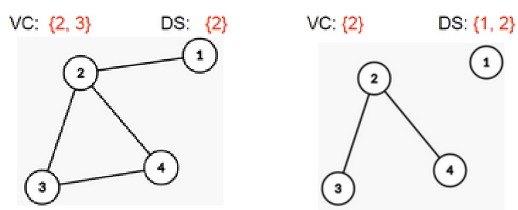
Certifying the algorithm
(1) Check that edges in E' have size at most k, (2) the edges connect all shaded vertices with a DFS on the subgraph of G formed by deleting all edges but the ones in E', starting from any shaded nodes. During DFS, we "check off" all shaded nodes. If |E'| <= K and all shaded notes are checked off, E' is a good solution (Yes). is in **CLASS NP.** **If all nodes are shaded, is minimum spanning tree algorithm.**

Vertex Cover Problem (NP-COMPLETE)

Given a graph **G = (V, E)** and an **integer K**, is there a vertex cover with at most **K** vertices in **G**? A vertex cover is a subset **W** of **V** such that |**W** | ≤ **K**, such that every edge in **E** has at least one endpoint in **W**. In other words, **ALL EDGES TOUCH A VERTEX IN W.**

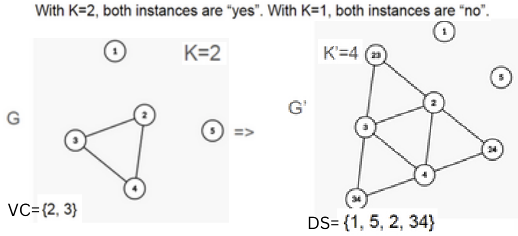
Dominating Set Problem (NP-COMPLETE)

Given a graph **G = (V, E)** and an **integer K**, is there a dominating set with at most **K** vertices in **G**? A dominating set is a subset **W** of **V** such that |**W**| ≤ **K**, such that every element of **V** − **W** is joined by an edge to an element of **W**.
EVERY VERTEX NOT IN THE SUBSET, IS CONNECTED WITH AN EDGE TO ANY ELEMENT IN THE SUBSET.



Vertex Cover -> Dominating Set Reduction

Recognize that VC cares about 'covered edges', and DS cares about 'dominated nodes'. Add a node to **G'** for each edge in **G,** and attach the endpoints to it.



the edge in 3 -> 4 becomes 3 -> 34, 34->4, with an edge in between.
Isolated nodes don't have an edge, so they don't need to be part of vertex cover, but need to be part of dominating set. Therefore
K' = K + #(isolated nodes in G).

Reduction: Given an instance (G, K) of Vertex Cover, construct an instance (G', K') of Dominating Set by adding one new node v_e for each edge $e \in E$, and connecting this node to the endpoints of e. That is, if $e = \{x, y\}$ then we add the edges $\{v_e, x\}$ and $\{v_e, y\}$.

Choose K' to be K plus the number of isolated nodes of G.

Runtime: The time to generate the new nodes and edges is $O(m)$.

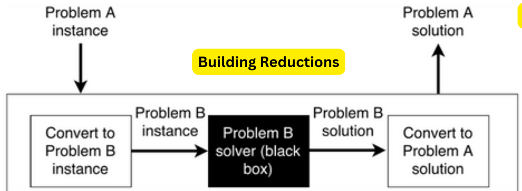
Correctness: The following is a formal proof of this algorithm's correctness, using the same "if and only if" style proof as assignment 1 question 4.3 and 4.4:

Suppose that G has a vertex cover W of size at most K. Since W is a vertex cover of G, (i) all non-isolated nodes in $V - W$ are joined by an edge to the nodes of W and (ii) all nodes v_e are also joined by an edge to the nodes of W. So the set W' which contains all nodes of W, plus the isolated nodes of G, is a dominating set of G' and has size at most K'.

Conversely, let W' be a dominating set of G'. Note that W' must contain all isolated nodes of G'. Let W be obtained from W' by discarding isolated nodes, and replacing any v_e node in W' by one of e's endpoints. The nodes of W are all nodes of G, and have an edge to all of the v_e nodes. This means that every edge of G has at least one endpoint in W, and so W must be a vertex cover of G. [Note: Without the extra v_e nodes, this part of the reduction correctness would not hold.]

INTER: OUTSIDE (INTER-GALACTIC), OUTSIDE OF SUBSET
INTRA: INSIDE (INSIDE THE SUBSET)

Building Reductions



Building a reduction from SAT to 3-SAT, to prove SAT is NP-Complete.

(convert to problem b instance)
Converting SAT instance (Prob. A) to 3-SAT instance (Prob. B)
Instance I of SAT. Instance I' of 3-SAT. For reduction to work, I is satisfiable only if I' is.

1. Imagine I has clause with two literals, $(\overline{X}_2 \vee X_3)$. To obtain I' from I, we replace this clause with $(\overline{X}_2 \vee X_3 \vee Y)$ and $(\overline{X}_2 \vee X_3 \vee \overline{Y})$. A truth assignment that satisfies the original clause, satisfies this one. Y is useless, since it is true in first and false in the other.
2. Imagine I has clause with 1 literal, (X_5). Replace with $(X_5 \vee \overline{Y} \vee \overline{Z})$, $(X_5 \vee \overline{Y} \vee Z)$, $(X_5 \vee Y \vee \overline{Z})$ and $(X_5 \vee Y \vee Z)$.
3. Imagine I has 4 literals, $(X_1 \vee \overline{X}_2 \vee X_3 \vee X_4)$. Replace with: $(X_1 \vee \overline{X}_2 \vee Y) \wedge (\overline{Y} \vee X_3 \vee X_4)$.
If one of the first two literals (i.e., X_1 or $\overline{X}_2$) is true, first new clause is satisfied. Must make Y false.
4. To generalize, convert any clause with k > 3 literals $(l_1 \vee l_2 \vee \ldots \vee l_k)$ Into 3 literals. Generalization:
$$(l_1 \vee l_2 \vee X_{i+1}) \wedge (\bar{X}_{i+1} \vee l_3 \vee X_{i+2}) \wedge (\bar{X}_{i+2} \vee l_4 \vee X_{i+3}) \wedge \ldots$$
$$\wedge\ (\bar{X}_{i+(j-2)} \vee l_j \vee X_{i+(j-1)})\ \wedge \ldots$$
$$\ldots \wedge\ (\bar{X}_{i+(k-4)} \vee l_{k-2} \vee X_{i+(k-3)}) \wedge (\bar{X}_{i+(k-3)} \vee l_{k-1} \vee l_k)$$
There are $k-2$ new clauses in total, using $k-3$ new variables $X_{i+1}, X_{i+1}, \ldots, X_{i+(k-3)}$ to "link" the clauses together. Overall, an algorithm to do this takes time $\Theta(k)$.
5. **TRANFORMATION INSTANCE ALGORITHM:** Use name TRANSFORM-CLAUSE to refer to algorithm. The algorithm converts an instance I of SAT to an instance I' of 3-SAT, working clause by clause to call TRANSFORM-CLAUSE for K > 3, and for K < 3 calling on step 1 and 2.

SAT instance		3SAT instance
$X_3 \vee \overline{X_5}$	⇒	$X_3 \vee \overline{X_5} \vee Y_1$ $X_3 \vee \overline{X_5} \vee \overline{Y_1}$
$\overline{X_1} \vee X_2 \vee X_4 \vee X_5$	⇒	$\overline{X_1} \vee X_2 \vee Y_2$ $\overline{Y_2} \vee X_4 \vee X_5$
X_4	⇒	$X_4 \vee \overline{Y_3} \vee \overline{Y_4}$ $X_4 \vee \overline{Y_3} \vee Y_4$ $X_4 \vee Y_3 \vee \overline{Y_4}$ $X_4 \vee Y_3 \vee Y_4$
$X_1 \vee \overline{X_2} \vee \overline{X_3} \vee X_4 \vee \overline{X_5} \vee \overline{X_6}$	⇒	$X_1 \vee \overline{X_2} \vee Y_5$ $\overline{Y_5} \vee \overline{X_3} \vee Y_6$ $\overline{Y_6} \vee X_4 \vee Y_7$ $\overline{Y_7} \vee \overline{X_5} \vee \overline{X_6}$

(convert B solution to problem A Solution)
Transform solution algorithm: This algorithm to transform a solution (YES or NO) to the 3-SAT instance I' back to a solution for I simply uses the exact same solution (YES OR NO).

Algorithm SAT-to-3-SAT(I) takes time linear in the length of I, and the algorithm to translate the solution back takes $O(1)$ time.

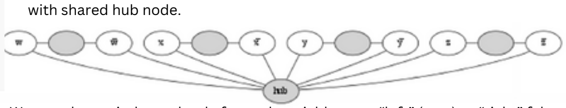
What do reductions tell us:

1. If our reduction's algo takes **O(f(n))** time, and black box solver for B also takes **O(f(n))** time, **runtime to solve problem A** is also **O(f(n))**.
2. If our reduction takes O(g(n)) time, and there is no algorithm for problem A that runs in O(g(n)) time, there **CANNOT be an SOLVING algorithm** that runs in O(g(n)) time. Remember that $P \neq NP$, so THIS IS HOW WE USE REDUCTIONS IN AN NP-COMPLETENESS PROOF.
3. Our reduction from SAT to 3-SAT tells us that if 3-SAT can be solved in polynomial time, then ANY problem in NP can be solved in polynomial time.

Building a Reduction from 3-SAT to ST, to show that ST is NP-Complete

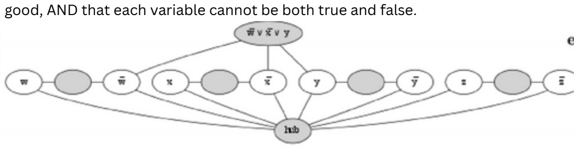
Converting 3-SAT instance (Prob. A) to ST instance (Prob. B)
If we reduce from 3-SAT to ST in polynomial time, we can "chain" together a reduction from any problem to 3-SAT with the reduction from 3-SAT to ST, to get a larger polynomial reduction from that problem in NP to ST. Going from ST -> 3SAT isn't interesting, since all it does is say that ST can be reduced into 3SAT.
(convert to problem b instance)

1. Given a 3-SAT instance with four variable gadgets (w, x, y, z), turn into graph, with shared hub node.



We can choose independently for each variable to go "left" (true) or "right" false. We make k=8 so there's JUST ENOUGH edges to enforce you choosing one edge or the other.

2. Given a clause $(\overline{w} \vee \overline{x} \vee y)$, enforce it so that the tree is good if the clause is good, AND that each variable cannot be both true and false.



For each variable, we need to connect the shaded node between the + and -. Whichever way we connect it is what we choose for the truth value.

3. Complete reduction from 3-SAT to ST such that ST instance is YES if and only if 3-SAT instance is YES also.

Nodes: We first create a single hub node. Then, for each variable x_i of instance I, we create a node labeled x_i and one labeled $\bar{x}_i$, and one additional "pin" node p_i (the unlabeled node). Then for each clause $c_l = (l_1 \vee l_2 \vee l_3)$, we create a node labeled c. (Recall that each l_j is either some variable x_i or its negation $\bar{x}_i$.) So, with n variables and c clauses, that's one hub, n pins, n variable nodes, n negated variable nodes, and c clause nodes, for a total of $3n + c + 1$ nodes.

Edges: We put edges between the hub and each of x_i and $\bar{x}_i$, as well as edges between p_i and each of x_i and $\bar{x}_i$. Also, we connect each clause node c_l to each of its literals l_1, l_2, and l_3. The total number of edges is $4n + 3c$. These nodes and edges comprise our graph G.

We let the set of "shaded" node S be the hub, the n pin nodes and the c clause nodes. So, there are $n + c + 1$ shaded nodes in total.

Finally, we let $k = 2n + c$. Make sure to state why it is polynomial as well.

Continued on Next Page...

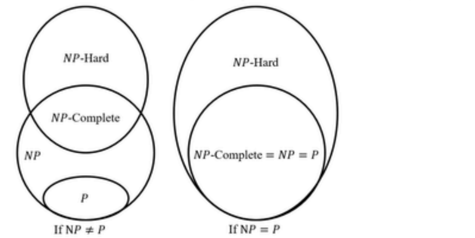
Theory

P: Can create algorithm to solve in polynomial time.
NP: Can be verified in polynomial time.
NP-complete: A problem A that's **NP,** in which you can reduce any other **NP** problem B to A polynomially. It's possible to reduce any NP problem into 3-SAT.
NP-hard: A problem is as hard as every other problem in NP. NP hard problems can possibly not be verified in polynomial time.
P != NP No one has ever been able to solve an NP problem polynomially. If someone successfully does so, then ALL NP problems are solvable polynomially.

For a problem A to be NP-complete:
1. prove A is in NP => outline steps to verify solution in polynomial time
2. prove A is in NP-hard => reduce B to A where B is NP-complete/np-hard
A <= B (A can be reduced to B), 3SAT <= Subset SUms

- P = {all problems for which a *known* polynomial algorithm exists}
- NP = {all problems for which an *efficient certifier* exists}
- NP-Hard = { class of decision problems to which all NP problems could be reduced to in polynomial time }
- NP-Complete = NP ⋂ NP-Hard

Another way to write the definition of NP-Hard is $\{X : Y \leq X,\ \forall Y \in \text{NP}\}$.
Another way to write the definition of NP-Complete is $\{X : X \in \text{NP and } Y \leq X,\ \forall Y \in \text{NP}\}$



	P	NP	NP-Complete	NP-Hard
Solvable in Polynomial Time	✓			
Solution Verifiable in Polynomial Time	✓	✓	✓	
Reduces any NP Problem in Polynomial Time			✓	✓

More Types of Problems

ALL NP PROBLEMS ARE DECISION PROBLEMS.

Counting Inversions (P) problem, not a decision one.

Input: Array A[1...n]
Task: Count the number of pairs where (i < j) and A[i] > A[j]. In other words, find the number of pairs where the smaller number is ahead of the larger number.

For array [1,4,3,2], the NUMBERS (4,3), (4, 2), and (3,2) are inversions. 3 inversions total.

Brute Force Algorithm is O(n^2). Can count maximum of n*(n-1)/2 inversions. Can solve faster with CountInversions Divide and Conquer.

```
function CountInversions(A[1..n])
    inversions ← 0
    for i ∈ [1..n] do
        for j ∈ [(i + 1)..n] do
            if A[i] > A[j] then
                inversions ← inversions + 1
            end if
        end for
    end for
    return inversions
end function
```

Counting Inversions (Divide and Conquer)

HELPER FUNCTION CALLED COUNTAPPENDEDINVERSIONS()
Input: Two arrays A[1...n], B[1...m]
Task: Count the inversions of [A[...], B[...]] in O(n+m) time.

```
function CountAppendedInversions(A[1..n], B[1..m])
    inversions ← 0, i ← 1, j ← 1
    while i ≤ n and j ≤ m do
        if A[i] ≤ B[j] then ▷ A[i] is at least as small as any remaining element of B
            i ← i + 1
        else                ▷ B[j] is smaller than every remaining element of A
            inversions ← inversions + (n − i + 1)
            j ← j + 1
        end if
    end while
    return inversions
end function
```

Given A=

2	4	10	11	14	...

B=

6	8	16	20	...

If (10, 6) is an inversion, everything right of 10 is also an inversion of 6, since arrays are sorted!

We can use CountAppendedInversions() for a divide and conquer algorithm for counting inversions in nlogn time.

```
function CountInversions&Sort(A[1..n])
    if n ≤ 1 then
        return (0, A)
    end if
    mid ← ⌊n/2⌋
    inv_L, Left ← CountInversions&Sort(A[1..mid])
    inv_R, Right ← CountInversions&Sort(A[(mid + 1)..n])
    inv_combine ← CountAppendedInversions(Left, Right)
    SortedArr ← merge(Left, Right)
    return (inv_L + inv_R + inv_combine, SortedArr)
end function
```

Split array in half, combine the two arrays with our helper, then merge them back together, returning the combined inversions of the left, right, and combined together.

$T(n) = 2T(n/2) + cn$

Master Theorem Case 2:
$a=2,\ b=2,\ \log_2 2 = 1$
$f(n)=cn \in \Theta(n^1)$

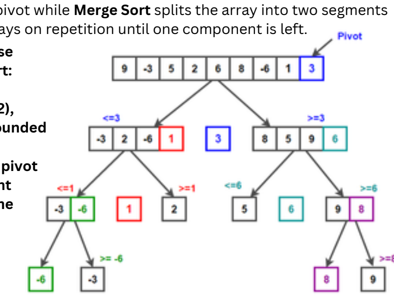
$T(n) = \Theta(n\log n)$

Quicksort vs Merge Sort

worst case merge sort is nlogn

Quick Sort sorts the components by comparing each component with the pivot while **Merge Sort** splits the array into two segments or subarrays on repetition until one component is left.

worst case quick sort: O(n^2), theta(n^2), under bounded nlogn if smallest pivot is element every time
Average: O(nlogn)



Proving Reductions are Correct

To prove Steiner reduction is correct, show that instance I is a Yes-Instance of 3-SAT **IF AND ONLY IF** $I' = (G, S, k)$ is a YES-Instance of ST.

To prove a reduction is correct, you need to prove:

1. **"If" (Solution a -> Solution B)**
2. **"Only if" (Solution B -> Solution A).**

If I is a Yes-instance of 3-SAT then I' is a Yes-instance of ST

What is a good solution for I?

Since I is a Yes-Instance of 3-SAT, there must be some assignment of truth values that makes at least one literal true in every clause.

What is a good solution for I'?

The solution to an ST problem is of at most k edges that connect all the shaded nodes.

Given a Yes solution of I, how can you choose I' so that it's Yes also?

SOLUTION: We designed our reduction so that one "variable gadget" represents the truth (or falsehood) of each variable. Thus, in each "variable gadget", we'll use two edges to connect the "pin" to the hub via the variable node x_i if x_i is set to true, or via $\bar{x}_i$ if x_i is set to false. This is possible using $2n$ edges in total.

Then, pick the first literal that is true in each clause c_j, and choose the edge connecting c_j to that literal. Note that this literal node must already be connected to the variable's hub and pin, since we ensured we always connect in the node corresponding to the variable's truth value. We use c additional edges here.

Finish the proof to show that I' is actually working, and is a soln to instance.

SOLUTION: Above we built a candidate solution using exactly $2n + c$ edges. Does it actually connect all shaded nodes? Every variable's pin node is connected to the hub, as discussed above. Further, every clause is connected to a true literal, which must in turn be connected to the hub given the way we chose the variable gadgets' edges. Thus, our ST "solution" is a valid certificate showing the answer to the ST instance is YES.

If I' is a Yes-instance of ST then I is a Yes-instance of 3-SAT

SOLUTION: Since I' is a Yes-instance of ST, there is a set of at most $k = 2n+c$ edges that connects up all shaded nodes of G. By part 1 above, this set of edges can connect exactly $2n + c + 1$ nodes into a tree.

There are $n + c + 1$ shaded nodes that **must** be connected; so, n of the unshaded nodes can also be connected. For every variable gadget, at least one of its "variable" nodes (positive or negated) **must** be connected (because there's no other way "out" of the variable's pin, which is why we called it a "pin"). Since this consumes all remaining available nodes, we must choose exactly one of each positive/negated variable pair. Call these connected variable nodes the "chosen" nodes. Furthermore, each clause c_i must be connected to at least one of its literals, and moreover, should be connected to a "chosen" node.

(Note that it can be possible to connect all $2n+c+1$ of the shaded and chosen nodes with $k = 2n+c$ edges, where some clause node has **more than one** incident edge; this does not pose a problem in our reasoning.)

Now consider the truth assignment corresponding to the "chosen" nodes, that is, set x_i to be true if x_i is a chosen node, and set x_i to be false if $\bar{x}_i$ is a chosen node. Since every clause node has an edge to at least one chosen node, all clauses in our 3SAT instance are satisfied by this truth assignment, and so the instance's answer is **Yes**.

You've now shown that ST is in NP and NP-hard (because 3-SAT—which is known to be NP-hard—is polynomial-time-reducible to ST). Therefore, ST is NP-complete!

Greedy Clustering

maximize the minimum intra-category edge similarity.

function CLUSTERING-GREEDY(n, E, c)
▷ $n \geq 1$ is the number of photos
▷ E is a set of edges of the form (p, p', s), where s is the similarity of p and p'
▷ c is the number of categories, $1 \leq c \leq n$
create a list of the edges of E, in decreasing order by similarity
let $\mathcal{C}$ be the categorization with each photo in its own category
Num-$\mathcal{C} \leftarrow n$ ▷ Initial number of categories
while Num-$\mathcal{C} > c$ **do**
remove the highest-similarity edge (p, p', s) from the list
if p and p' are in different categories of $\mathcal{C}$ **then**
"merge" the categories containing p and p'
Num-$\mathcal{C} \leftarrow$ Num-$\mathcal{C} - 1$
return $\mathcal{C}$

AS BEFORE: We're given a complete, weighted, undirected graph $G = (V, E)$ represented as an adjacency list, where the weights are all between 0 and 1 and represent similarities—the higher the more similar—and a desired number $1 \leq k \leq |V|$ of categories.

We define the similarity between two categories C_1 and C_2 to be the maximum similarity between any pair of nodes $p_1 \in C_1$ and $p_2 \in C_2$. We must produce the categorization—partition into k (non-empty) sets—that minimizes the maximum similarity between categories.

Now, we'll prove this greedy approach optimal.

1. Sort a list of the edges E in decreasing order by similarity.
2. Initialize each node as its own category.
3. Initialize the category count to $|V|$.
4. While we have more than k categories:
   (a) Remove the highest similarity edge (u, v) from the list.
   (b) If u and v are not in the same category: Merge u's and v's categories, and reduce the category count by 1.

k-Clique: The input consists of a graph $G = (V, E)$ and a positive integer k. The k-clique problem asks: does the graph contain a clique of size k? That is, we want to know if there is a subset W of k vertices in V with the property that every two elements in W are joined by an edge.

Independent Set: The input consists of a graph $G = (V, E)$ and a positive integer k. The Independent Set problem asks: does the graph contain an independent set of size k? That is, we want to know if there is a subset W of k vertices in V with the property that no two elements in W are joined by an edge.

SAT: The input is a collection of m *clauses* over n boolean variables $X_1, X_2, \ldots X_n$. Each clause is a disjunction of some of the variables or their complements.

The problem consists in answering the question "Is there a way to assign truth values to each variable that makes **every** clause of the instance TRUE?"

QuickSelect Recurrence

$$T_{QS}(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ T_{QS}(3n/4) + cn & \text{otherwise} \end{cases}$$

QuickSelect

function QUICKSELECT(A[1..n], k) // returns the element of rank k in an array A of n distinct numbers, where $1 \leq k \leq n$
if $n > 1$ **then**
Choose pivot element $p = A[1]$
Let Lesser be an array of all elements from A less than p
Let Greater be an array of all elements from A greater than p
if |Lesser| = k - 1 **then**
return p
else
if |Lesser| > k - 1 **then** // all smaller elts are in Lesser; k is unchanged
return QuickSelect(Lesser, k)
else// |Lesser| $< k - 1$
// subtract from k the number of smaller elts removed
return QuickSelect(Greater, $k-$ |Lesser| -1)
else
return $A[1]$

Best case: O(n), worst case O:(n^2)

Common Algorithms and their uses

Breadth first search - Shortest Path Algorithm, Level Order Traversal
Kruskal's Algorithm - Minimum Spanning Tree
Prim's Algorithm - Minimum Spamming Tree
Dijkstra's Algorithm - Shortest Path Algorithm for graphs with weighted edges
Topological sort - Order vertices of directed acyclic graph
Greedy Clustering Algorithm - Maximize similarity within cluster, Minimize outside of cluster.

Graph Vocabulary

Complete: Every node is connected to very node in the graph.
Directed: Edges have directions (arrows).
Weighted: Edges have weightings signifying difficulty.

Knapsack with Structured Weights (Tutorial 6)

There are n items. Given a positive integer array 1 ..n of **V** representing values, and another array **W** of positive integers 1 .. n representing weights,

Select a subset $S \subseteq \{1, \ldots, n\}$ that maximizes the total value of S,

but keeps the total weight below C.

The regular knapsack problem cannot be solved polynomially.

Structured weights constraint: Assume that C=1. Assume there is an unlimited of 1/16, 1/8, 1/4, 1/2 items, all with value 0. Weights can only be 1 of those 4.

Solving the algorithm **(GREEDY)**

1. Given an instance I of problem create a new instance I'.
2. Sort the 1/16 weights of array **W** in decreasing order. Pair them up.
3. Combine the weights and values of them, put into I'. w_i + w_j, v_i + v_j
   a. W = [1, 2]
   b. V = [1/16, 1/16]
   c. This becomes W=[3], V=[1/8]
4. Once all 1/16 weights are eliminated, repeat for 1/8, 1/4, and 1/2.
5. At the end, there will be multiple 1/2 items. Select the two items with the highest value. That is the solution.

Knapsack Exchange Argument

Use an exchange argument to prove that if an optimal solution uses c items of weight 1/16, it uses c of the highest value 1/16-weight objects.

Imagine that **Solution S** (subset of {1, 2, .. n}) uses **c** items with weight 1/16, but NOT the **c** highest value 1/16 weight items. You can always swap in a higher-value 1/16-weight item to get a better solution, so S could not have been optimal. The same reasoning is applied for 1/8, 1/4, 1/2. We have unlimited supply of dummy items, so this works, and is optimal because of the exchange argument approach.

Proving that the optimal Knapsack Solution uses even number of weights

Suppose **not**, that you have an optimal solution that uses an odd number of items. If you add up all the weights in the solution, then the numerator has to be odd also. Since C = 1 = 16/16, we can always add at least one more 1/16 weight item. This is why we need an unlimited supply of 0 value items of 1/16, 1/8, ... 1/2

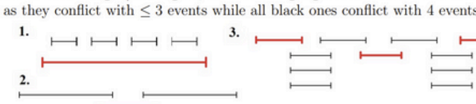
4.1 Interval Scheduling Problem

THE PROBLEM. Suppose we have a set E of n events. Each event i has a start time of $s(i)$ and an end time of $f(i)$. Two events are *compatible* if they do not overlap in time. We would like to find a subset $A \subseteq E$ of compatible events that is the largest in size.

THE ALGORITHM. To solve this problem, our greedy algorithm, at each step, would pick some event i from set E, add it to set A, and then remove all events from E that are not compatible with i. It remains to determine which criterion we should use to pick the event i at each step. There are several possible choices here:

(1) pick the event that starts first
(2) pick the event with the shortest duration
(3) pick the event that is the most compatible
(4) pick the event that ends first

It turns out that the first three choices do not lead to an optimal solution. The image below illustrates counter-examples for each of the three cases. The solution produced by each of the potential greedy algorithms is highlighted in red, while an optimal solution takes up the top row in each image (in image 3, all events highlighted in red are the most compatible ones as they conflict with ≤ 3 events while all black ones conflict with 4 events each).



Coin Change Problem

SOLUTION:

function SOLN'(i)
▷ Note: It would be handy if Soln had 0 and negative entries.
▷ We use this function SOLN' to simulate this.
if $i < 0$ **then return** infinity
else if i = 0 **then return** 0
else return Soln[i]

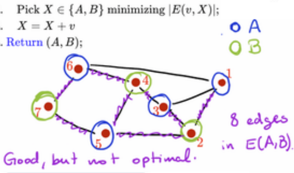
function DP-CHANGE(n)
if $n \leq 0$ **then return** SOLN'(n)
else
▷ Assumes $n > 0$; otherwise, just run SOLN'
create a new array Soln[1..n]
for i from 1 to n **do**
Soln[i] $\leftarrow$ the minimum of
SOLN'$(i - 25) + 1$,
SOLN'$(i - 10) + 1$, and
SOLN'$(i - 1) + 1$
return Soln[n]

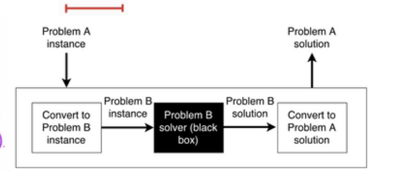
Max Cut Greedy

(a) (b)

$Y \in$ NP}.

1. $A = \emptyset,\ B = \emptyset$;
2. **For** $v \in V$:
3. Pick $X \in \{A, B\}$ minimizing $|E(v, X)|$;
4. $X = X + v$
5. **Return** (A, B);





Deterministic Select

has a better worst case runtime then quickselect

Deterministic Select: Worst Case runtime: $\Theta(n)$. By choosing median of medians as pivot where $\delta \in (0,1)$ ex. $\delta = (\frac{3}{4})$
Median $= \lceil \frac{n}{2} \rceil$ $\frac{3n}{10}$ elements are smaller than m (median of group medians) $\frac{3n}{10}$ also larger than m
For QuickSelect, largest possible size of one of the Lesser or Greater lists $\max\{|LESSER|, |GREATER|\} < \frac{7n}{10}$
↳ since our pivot is between ~~___~~ pivot (smaller & greater than the pivot)
Recurrence Relation for worst-case running time of DeterministicSelect
with recursion tree Total Work $\leq cn \sum \left(\frac{9}{10}\right)^i = O(n), \Theta(n)$ since $T(n) = T(\frac{n}{5}) + T(\frac{7}{10}n) + cn$
$\Omega(n)$ work at first level

Deterministic Select. Group elements into groups of 5 and find medians of each group. We get floor of n/5 groups. To select **pivot**, group A into groups of 5 and find medians of median using a recursive call to deterministic quickselect.

1. Choose a Problem *B* to reduce to.
   a. Let's reduce the Resident Hospital Problem to Stable Matching Prob.

2. Transform a small instance of RHP into an instance of B.

```
r1: h1 h2        h1 (1): r2 r1 r3    r1: h1 h2_1 h2_2      h1:   r2 r1 r3
r2: h2 h1        h2 (2): r1 r2 r3    r2: h2_1 h2_2 h1      h2_1: r1 r2 r3
r3: h1 h2                            r3: h1 h2_1 h2_2      h2_2: r1 r2 r3
```

3. Transform a solution from *B* instance into a solution to the RHP instance.

SOLUTION: Running Gale-Shapley gives this solution: $\{(h_1, r_1), (h_{2_1}, r_2), (h_{2_2}, r_3)\}$.

That's already very close to the solution we found by hand of $\{(h_1, r_1), (h_2, r_2), (h_2, r_3)\}$. It looks like we just need to erase the subscripts on the hospitals, since hospital-slots are no longer separate.

4. Design an algorithm to transform any instance I of RHP to an instance of *B*

Clone hospitals, with all having same preference list. Once this is done, is SMP instance.

5. Design an algorithm to transform a solution of B into a solution for RHP.

The only thing different about M from a possibly-stable RHP solution would be the subscripts on the hospital-slots. If we erase those, then since each hospital-slot had one match and each hospital had s(h) hospital-slots, each hospital in the RHP solution will now have s(h) matches, as we expect.

Stock Market

Divide & Conquer
The Stock Market: 2 halves, right maximal is maximal in P. If maximal in left, we must check maximal points from LHS recursion. $\Theta(n)$: For each LHS maximal pt p. Check each point in RHS points if they dominate p
Maximal Points: $\Theta(n\log n)$: LeftMaximal & RightMaximal. Scan RHS set to find highest y-val point q — sort by x-values
Output (RightMaximal) ∪ NonDominated (Left Maximal, q) $T(n) = 2T(\frac{n}{2}) + f(n)$ $\log_2(2) = 1$
Prune & Search — Left Maximal — ⊙ highest y-val in RHS — $f(n) \in \Theta(n^1 \log^0 n)$ case 2
QuickSelect (med): Use quicksort. Have recursion tree $T(n) = \Theta(n\log n)$

```
Algorithm MaximalPoints(P)
  sort P by increasing x-coordinate
  return MaximalPointsHelper(P, 0, length[P]-1)

Algorithm MaximalPointsHelper(P, first, last)
  if first = last then
    return { P[first] {

  mid ← ⌊(first + last)/2⌋
  Sl ← MaximalPointsHelper(P, first, mid)
  Sr ← MaximalPointsHelper(P, mid+1, last)

  q ← point of Sr with largest y-coordinate
  return Sr union UndominatedPoints(Sl, q)
```

SMP

SMP: Brute Force: $\Omega(n!\ n^2)$
make all permutations, n^2 to check for instabilities w/ all employers & applicants
Gale Shapley: $O(n^2)$ → n is number of elements in each set

RHP

Reduction
RHP: m is hospitals, n is residents. To test if a solution form is valid & g... (naively check each pair), $O(|H||R|) = O(mn)$
How many subsets of size k can be selected from n elements
$\binom{n}{k} = \frac{n(n-1)\cdots(n-k+1)}{k!}$ transform instance & solution

Graphs

Articulation Point: Vertex whose removal increases the # of connected components in the graph undirected
Diameter: the smallest # of edges on any path b/w 2 nodes (these are diametric nodes) undirected, unweighted
FindDIAM: Brute force - Use BFS to compute all the distances, if dist(i,j) > diam, diam = dist(i,j) → longest shortest path
$|\text{Valid\_Solns}| = \binom{n}{2} + n$ Runtime: $O(n^2(n+m))$ = $O(n^3 + n^2m)$ depends on dense or sparse — for loops — BFS — $m \approx n^2$ if dense
Optimal - Do BFS first, then let u_i be any node occurring on last layer $L_{d(i)}$. Then check $d(i) >$ Diam
Overall Work: $O(n(n+m)) = O(n^2 + mn)$ — cn work at each

True, False, or Open Q's? For NP-COMPLETE (Tutorial 10)

If $X \leq_p X'$, X is in P and X' is in NP, then X' is in P

"if X can be reduced to X', X is P, and X' is NP, then is X' in P?

ANS: Open question. depends if P=NP or P!=NP. If P!=NP, then it is false because X' could be NP complete, and not P. If P=NP however, then it could be possible for X to be in P.

Imagine X and X' are decision problems, where both problems have Yes and No Instances.

If $X \leq_p X'$ and X is *not* in NP, then X' is not in NP.

"if X can be reduced to X', and X is NOT in NP, then X' is not in NP.

SOLUTION: True. This is a hard one and requires proof. We will prove contrapositive, that if X' were in NP, then X must also be in NP.

Let f be a polynomial-time reduction from instances of X to instances of X'. Since we assume that X' is in NP, there must be an efficient certifier for X', let's call this algorithm Certifier-X'.

To show that X is in NP, we construct an efficient certifier for X. This certifier takes an instance I and a certificate, say C', computes $f(I)$, and then runs algorithm Certifier-X' on $(f(I), C')$. If I is a Yes-instance of X, then I' is a Yes-instance of X', and so there is some polynomial-size certificate that causes Certifier-X' to output Yes. Also, if I is a No-Instance of X, then $f(I)$ is a No-instance of X', and Certifier-X' outputs No, no matter what the certificate is.

If an NP-complete problem "subset sums" is a special case of Partition, then Partition is NP-complete, not the other way around.

The decision problem Partition takes a set of integers S, and returns yes if they can be 'partitioned' into two disjoint subsets whose union is S, such that the two subsets sum to the same value (half the sum of elements in S). Partition is in NP.

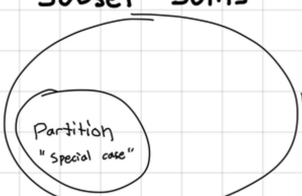
Partition is a special case of another problem, Subset Sums. There is a very nice reduction that proves that 3-SAT $\leq_p$ Subset Sums. Again assume P!=NP

Statements This is sufficient evidence that Partition is NP-Complete.

SOLUTION: False. This special case argument is backwards. If an NP-Complete problem A is a special case of another problem B, then we can conclude B is NP-complete, not the other way around.

It turns out Partition actually is NP-Complete, but you need to do a reduction to prove that.

Subset Sums



Subset Sums being NP Complete does NOT mean Partition is.

IF Partition is NP-Complete, then Subset Sums is also.

If a problem is in NP, and P!=NP, is the problem NP-complete?

The decision problem Integer Factorization (IF) takes two numbers, n and k, and returns yes if n has an integer factor less than k other than 1, and no otherwise. This problem is in NP. Despite researchers looking for one for a very long time, there is currently no known algorithm that solves this problem in polynomial time.

Statement: Assume P$\neq$NP. Based on the above facts, we can conclude that IF is NP-Complete.

Answer: NO, you must do a reduction to prove that it is NP-hard, and thus NP-complete.

Can SAT be solved polynomially? No, depends on if P=NP

Statement: There is no positive integer c such that boolean satisfiability (SAT) is in $O(n^c)$.

SOLUTION: Open Question. If P=NP, then every problem in NP has a polynomial time algorithm. SAT is in NP, therefore there is an algorithm that solves SAT in polynomial time.

Conversely, if P$\neq$NP, then there is at least some problem in NP that does not have a polynomial time algorithm. Since SAT is NP-Complete, every problem in NP can be reduced to it in polynomial time. So, if SAT could be solved in polynomial time, then all problems in NP can be solved in polynomial time, a contradiction to our assumption that P$\neq$NP.

Master Theorem

Long Version

1. If $f(n) \in O(n^c)$ where $c < \log_b a$ then $T(n) \in \Theta(n^{\log_b a})$. **#1**
2. If for some constant $k \geq 0$, $f(n) \in \Theta(n^c(\log n)^k)$ where $c = \log_b a$, then $T(n) \in \Theta(n^c(\log n)^{k+1})$. **#2**
3. If $f(n) \in \Omega(n^c)$ where $c > \log_b a$ and $af(\frac{n}{b}) \leq kf(n)$ for some constant $k < 1$ and sufficiently large n, then $T(n) \in \Theta(f(n))$. **#3**

Short Version given a, b, and n^k

$$\begin{aligned} T(n) &= aT(n/b) + cn^k \\ T(1) &= c, \\ T(n) &\in \Theta(n^k) \text{ if } a < b^k \quad \textbf{\#3} \\ T(n) &\in \Theta(n^k \log n) \text{ if } a = b^k \quad \textbf{\#2} \\ T(n) &\in \Theta(n^{\log_b a}) \text{ if } a > b^k \quad \textbf{\#1} \end{aligned}$$

Flipped Version

If $T(n) = aT(n/b) + O(n^d)$ for constants $a > 0$, $b > 1$, $d \geq 0$, then

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \quad \textbf{\#3} \\ O(n^d \log n) & \text{if } d = \log_b a \quad \textbf{\#2} \\ O(n^{\log_b a}) & \text{if } d < \log_b a \quad \textbf{\#1} \end{cases} \qquad T(n) = \begin{cases} aT(n/b) + f(n) & \text{if } n \geq n_0 \\ \Theta(1) & \text{if } n < n_0 \end{cases}$$

The Master Theorem: Let $a \geq 1, b > 1$ be real constants, and let $T(n)$ be defined by:

Case 1. If $f(n) \in O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$ then $T(n) \in \Theta(n^{\log_b a})$.

Case 2. If $f(n) \in \Theta(n^{\log_b a} \log^k n)$ for some $k \geq 0$ then $T(n) \in \Theta(n^{\log_b a} \log^{k+1} n)$.

Case 3. If $f(n) \in \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$ and $af(n/b) < \delta f(n)$ for some $0 < \delta < 1$ and all n large enough, then $T(n) \in \Theta(f(n))$.

Divide & Conquer, Master Theorem

To solve a recurrence:

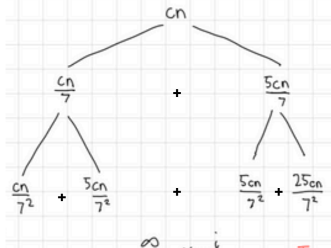
1. **Draw a recursion tree,** sum work done at all nodes at all levels
2. **The Master Theorem,** only works for specific forms of recurrence

Drawing the recursion tree

$$T_7(n) \leq \begin{cases} T_7(\lfloor \frac{n}{7} \rfloor) + T_7(\frac{5n}{7} + 3) + cn & \text{if } n \geq 7 \\ \Theta(1) & \text{if } n < 7 \end{cases}$$

CASE 1: INFINITE

work at each level < 1



cn

Work at level 1 $\frac{6}{7}cn$

Because work at each level gradually decreases, it's a SUM OF INFINITE SERIES.

Work at level 2 no $\frac{36}{49}cn = \left(\frac{6}{7}\right)^2 cn$ work at each level

$cn \sum_{i=0}^{\infty} \left(\frac{6}{7}\right)^i$ Is sum of infinite series because $\frac{6}{7} < 1$

sum of infinite geo series $= \frac{a}{1-r}$

It's O(n) because if the sum of the infinite geo series is constant, and it occurs at every level n, it's c*n which is O(n)

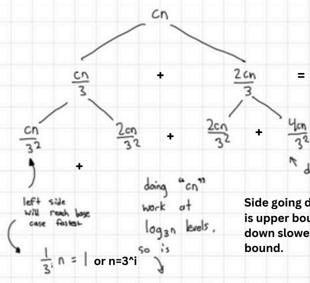
r is work at each level

$$\sum_{k=0}^{\infty} ar^k = \frac{a}{1-r}, \text{ for } |r| < 1. \qquad = \frac{1}{1-\frac{6}{7}} \sim c \qquad O(n) \quad cn$$

$$T_3(n) \leq \begin{cases} T_3(\lfloor \frac{n}{3} \rfloor) + T_3(\frac{2n}{3} + 1) + cn & \text{if } n \geq 3 \\ \Theta(1) & \text{if } n < 3 \end{cases}$$

CASE 2: FINITE

work at each level $>= 1$



$\frac{3}{3}cn = cn$

$\frac{1+2+2+4}{9} = \frac{9}{9} = cn$ is cn for each level

Side going down fastest (1/3) is upper bound, side going down slowest (2/3) is lower bound.

$\frac{1}{3^i} n = 1$ or n=3^i

$i = \log_3 n \qquad \Omega(n \log_3 n)$

n=3^i

log_3(n)=i

$\left(\frac{2}{3}\right)^i n = 1$

$n = \left(\frac{3}{2}\right)^i$

$\log n = i \log \frac{3}{2}$

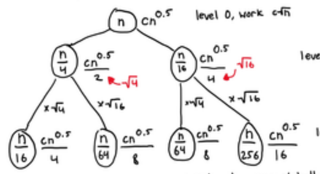
$i = \log_{\frac{3}{2}} n$

$O(n \log_{\frac{3}{2}} n) \qquad n \log n$

$\Theta(n \log n)$

$$T(n) = \begin{cases} c & \text{if } n \leq 16 \\ T(\frac{n}{4}) + T(\frac{n}{16}) + c\sqrt{n} & \text{if } n > 16 \end{cases}$$

CASE 3: SQRT



level 0, work $c\sqrt{n}$

level 1, $\frac{3cn^{0.5}}{4} = \left(\frac{3}{4}\right)^1 cn^{0.5}$

level 2, $\frac{9cn^{0.5}}{16}$

work is geometrically decreasing at each level.

$c\sqrt{n} \sum_{i=0}^{\infty} ar^i = \frac{a}{1-r} = \frac{1}{1-\frac{3}{4}} = 4c\sqrt{n} : O(\sqrt{n})$

The Stock Market, dividends and risks

```
Algorithm UndominatedPoints(P, q)
  S ← ∅
  for each point p of P do
    if q.x < p.x or q.y < p.y then
      add p to S
  return S
```

Each company is (x,y) (expected annual dividends, -10 ≤ y ≤ 10 risk, (-) riskier). Dominates when above and to the right. q = (q.x,q.y) dominates p = (p.x,p.y) if q.x ≥ p.x and q.y ≥ p.y.

Sort by increasing x then y. Split ½.

Right Subproblem: if a point is maximal here, must be maximal in P.

Left Subproblem: if a point is maximal here, not automatically maximal in P. If a point in RSP dominates p in LSP, then a point q in RSP with largest y guaranteed to dominate p

4. $T(n) = 3T(n/9) + \sqrt{n} \log_2 n$

SOLUTION: Here $a = 3$ and $b = 9$, which means $\log_b a = \log_9 3 = 0.5$, and $f(n) = \sqrt{n} \log_2 n = n^{0.5} \log_2 n$. We are therefore in case 2 of the Master theorem and $T(n) \in \Theta(\sqrt{n} \log^2 n)$.

5. $T(n) = \sqrt{n} T(n/3) + n^2$

SOLUTION: The Master Theorem can not be used because a is not a constant (it is $\sqrt{n}$).

6. $T(n) = 9T(n/3) + n \log n$

SOLUTION: Here $a = 9$ and $b = 3$, and so $\log_b a = \log_3 9 = 2$. Moreover, $f(n) = n \log n \in O(n^{2-0.5})$ so we are in case 1. This implies $T(n) \in \Theta(n^{\log_b a}) = \Theta(n^2)$.

7. $T(n) = 2T(n/2) + n/\log n$

SOLUTION: The Master Theorem can not be used: $a = 2$ and $b = 2$, and so $\log_b a = \log_2 2 = 1$. However there is no $\varepsilon > 0$ such that $n/\log n \in O(n^{1-\varepsilon})$.

Master Theorem Constants

$T(n) = 5T(\sqrt{n}) + n$

No, b is not a constant. Is sqrt(n)

$T(n) = T(n/2) + 1$

a = 1 and b = 2 which means log_b(a) = log_2(1) = 0, and f(n) = 1 = n^0. **in case 2** of the Master theorefore, T (n) ∈ Θ(log n).

$T(n) = T(n/4) + n$

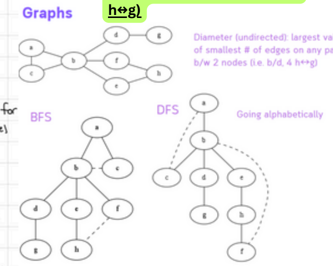
a = 1, b = 4, d=1 means log_b(a) = log_4(1) = 0. since d > log_b(a) = 1 > 0, case 3 applies.

To check regularity condition: a*f(n/b) = f(n/4) = n/4, and n/4 < δn for any δ in the range 0.4 < δ < 1. So case 3 applies, and T (n) ∈ Θ(n).

Articulation point: Two connected components c_1 and c_2 n G where the only path from c_1 to C_2 goes through V. T contains a subset of edges in G, so it cannot move any additional path apart from the one through V. Therefore, V is an articulation point.

diameter: largest value of smallest # of edges on any path b/w 2 nodes (i.e. b/d, 4 h↔g)

Graphs



Stock market Answer

SOLUTION: All that we need to do is check each point of P one at a time.

```
Algorithm UndominatedPoints(P, q)
  S ← ∅
  for each point p of P do
    if q.x < p.x or q.y < p.y then
      add p to S
  return S
```

Assignment

Mr Plow Greedy Exchange Argument

1. [5 points] Use an exchange argument to show that this greedy strategy yields an optimal schedule.

Let $\mathcal{O}$ denote an optimal solution, with ordered weights given by $o_1, o_2, \ldots, o_n$. The weighted sum of satisfaction for $\mathcal{O}$ is $S(\mathcal{O}) = \sum_{i=1}^{n}(n-i)o_i$.

Let p, q, where $p < q$, denote indices in $\mathcal{O}$ that define an inversion, relative to the order of the greedy ordering $\mathcal{G}$. That is, this is a pair of jobs that appear in a different order than they do in $\mathcal{G}$. Because the greedy strategy is to order jobs by decreasing weight, we must have $o_p \leq o_q$. The weighted sum of satisfaction for $\mathcal{O}$ is

$$S(\mathcal{O}) = (n-1)o_1 + (n-2)o_2 + \ldots + (n-p)o_p + \ldots + (n-q)o_q + \ldots + (n-n)o_n.$$

Let $\mathcal{O}'$ denote the solution obtained after we swap o_p with o_q in $\mathcal{O}$ (i.e., we swap the order so these two elements appear in the same order as they do in $\mathcal{G}$). Then

$$\begin{aligned} S(\mathcal{O}') &= (n-1)o_1 + (n-2)o_2 + \ldots + (n-p)o_q + \ldots + (n-q)o_p + \ldots + (n-n)o_n \\ &= S(\mathcal{O}) + (n-p)(o_q - o_p) + (n-q)(o_p - o_q) \\ &= S(\mathcal{O}) + (q-p)(o_q - o_p) \geq S(\mathcal{O}), \end{aligned}$$

because $q > p$ and $o_q \geq o_p$. Therefore, the weighted satisfaction of $\mathcal{O}'$ is at least as high as the weighted satisfaction of $\mathcal{O}$.

We can proceed in this way, repeatedly swapping inversions between the optimal schedule and the greedy schedule, until the optimal schedule is transformed into the greedy schedule. The weighted satisfaction will not decrease with each swap. Therefore, $S(\mathcal{G}) \geq S(\mathcal{O})$ and the greedy schedule is in fact optimal.

P vs Z

$$P(m,n) = \begin{cases} \alpha P(m, n-1) + (1-\alpha)P(m-1, n+1) & \text{if } m > 0, n > 0; \\ 0 & \text{if } m = 0, n > 0; \\ 1 & \text{if } m > 0, n = 0. \end{cases}$$

```
Algorithm ProbVictory(m, n):
  Initialize an (m+1)x(n+m+1) 2D array P (with zero-based indexing)

  % fill in base cases
  for i=1 to m:
    P[i, 0] ← 1
  for j=1 to n+m:
    P[0, j] ← 0

  % fill in the rest of the table
  for i=1 to m:
    for j=1 to (n+m-i):
      P[i, j] ← α*P[i, j-1] + (1-α)*P[i-1, j+1]

  return P[m, n]
```

Memory and Runtime: Θ(mn)+Θ(m^2)

Divide and Conquer

The Key Stages

1. ***Divide.*** We recursively divide the main problem into two or more smaller pieces tha usually are of equal size. Division is done based on some criteria and that criteria coul depend on step 3.
2. ***Base Cases.*** Once we have divided the main problem into small enough pieces, w carry out the computations to solve each of those subproblems. Those computation are usually just brute force.
3. ***Combine.*** We recursively combine the results of the two adjacent subproblems. I doing so, we compare their results and use some criteria to build up the overall result Generally, there are three cases to consider [with iii being the most typical one]:
   - i. the result from the first subproblem is the one we want
   - ii. the result from the second subproblem is the one we want
   - iii. the result that could be formed using elements from *both* subproblems is the on we want

   The work in this step should usually be done in at most *linear* time and often this ste is the most challenging one to design.

Quicksort

```
function QuickSort(A[1..n])        ▷ returns the sorted array A of n distinct numbers
  if n > 1 then                                                        ▷ Θ(1)
    Choose pivot element p = A[1]                                      ▷ Θ(1)
    Let Lesser be an array of all elements from A less than p          ▷ Θ(n)
    Let Greater be an array of all elements from A greater than p      ▷ Θ(n)
    LesserSorted ← QuickSort(Lesser)                                   ▷ T_Q(⌈n/4⌉ − 1)
    GreaterSorted ← QuickSort(Greater)                                 ▷ T_Q(⌊3n/4⌋)
    return the concatenation of LesserSorted, [p], and GreaterSorted   ▷ O(1)
  else
    return A                                                           ▷ Θ(1)
```

Recurrence relation for the runtime of QuickSort:

$$T_Q(n) = \begin{cases} c, & \text{if } n = 0 \text{ or } n = 1 \\ T_Q(\lceil \frac{n}{4} \rceil - 1) + T_Q(\lfloor \frac{3n}{4} \rfloor) + cn, & \text{otherwise.} \end{cases}$$

Ignoring floors, ceilings, constants for recurrence

Divide & Conquer

Overall Work: $O(n(n+m)) = O(n^2 + mn)$

Divide & Conquer

QuickSort: pick a pivot, lesser as array of all elements from A less than p. greater as array of all elements from A greater than p. } Sort } Sort $\theta(n)$ to split

If choose $\lceil \frac{n}{4} \rceil$-th smallest element as its pivot, $T_Q(n) = T_Q(\frac{n}{4}) + T_Q(\frac{3n}{4}) + cn$

Find base case when problem (n/c) has size $\le 1 \rightarrow (\frac{1}{4})^c n \le 1$ $\log_4 n \le i$ $\log_4 n$ "full levels"

Deepest leaf: $(\frac{3}{4})^i n \ge 1 \Leftrightarrow n \ge (\frac{4}{3})^i \Leftrightarrow \log_{\frac{4}{3}} n \ge i$ Shallowest leaf $n \le 4^i$

good for randomized data

cn work at each level

We must do cn work at each level from 0 to shallowest $\Omega(n \log_4 n)$ At most $O(n)$ at each level Total: $O(n \log_{\frac{4}{3}} n)$ $\theta(n \log n)$

Interval Scheduling

Set of requests {1,2,...,n}; i-th request corresponds to an interval of time starting at s(i) and finishing at f(i).

Subset of requests:

Compatible: no two of them overlap in time, and goal is to maximize compatible subset

Optimal: compatible sets of maximum size

Select a first request i1, once accepted, reject all requests not compatible with i1. Select next request i2 and repeat until out of requests to select from.

Solution: Accept the request that finishes first (request i for which f(i) is as small as possible).

Find Position Divide & Conquer

It's O(n) because if the sum of the infinite geo series is constant, and it occurs at every level n, it's c*n which is O(n)

Given a graph $G = (V, E)$ and integer $k \ge 1$, does G contain an independent matching of size k? That is, is there a subset M of E with $|M| = k$ such that M is an independent matching?

Independent Matching Problem

Here we give an incomplete reduction from IMP to SAT (see Page 3 for a formal definition of SAT). Given a graph $G = (V, E)$ with m edges, and a value k, we define the variable $X_{i,j}$ for $i = 1$ to m and for $j = 1$ to k, which represents whether the edge e_i is the jth element of the independent matching M. We define clauses as follows:

(A) For each integer a from 1 to k, add the clause:

$$X_{1,a} \vee X_{2,a} \vee \ldots \vee X_{m,a}$$

The clauses in (A) ensure that at exactly one edge is in each position of the clique.

and for every 2 distinct edges e_i, e_j also add the clause:

$$\overline{X}_{i,a} \vee \overline{X}_{j,a}.$$

The clauses in (B) ensure that each edge is assigned to at most one position of the independent matching.

(B) For any edge e_p and for every two distinct integers a, b from 1 to k, add the clause:

$$\overline{X}_{p,a} \vee \overline{X}_{p,b}.$$

The clauses in (C) ensure that the selected edges form a matching. Specifically, these make sure that if e_p, e_q have a common endpoint, then at most one can be selected in the matching.

(C) For each pair of distinct edges e_p and e_q which share an endpoint, and for every pair of integers a, b in 1 to k, add the clause:

$$\overline{X}_{p,a} \vee \overline{X}_{q,b}.$$

The clauses in (D) ensure that for any 2 edges in the selected matching, there is no edge which connects the endpoint of one to an endpoint of the other.

(D) For any path of length three, induced by consecutive edges e_i, e_j, e_q, and for every two integers a, b from 1 to k, add the clause:

$$\boxed{\overline{X}_{i,a} \vee \overline{X}_{q,b}}$$

(To be completed.)

$$P(m,n) = \begin{cases} 0 & \text{if } m = 0 \\ 1 & \text{if } n = 0 \\ \alpha P(m, n-1) + (1-\alpha) P(m-1, n+1) & \text{if } m > 0, n > 0 \end{cases}$$

SAT

True

Start by introducing nk variables $X_{i,j}$ where $n = |V|$ and $X_{i,j}$ denotes whether vertex i is the j^{th} element of the clique. Clique of size k.

A vertex must be in each of k positions: For each integer a from 1 to k, add the clause $X_{1,a} \vee X_{2,a} \vee \ldots \vee X_{n,a}$

No more than 1 vertex can occupy any position of clique: For each pair of vertices v_p & v_q and for every a in 1 to k, add: $\overline{X}_{p,a} \vee \overline{X}_{q,a}$

A single vertex can't occupy more than one position of the clique: For each vertex v_v and for every 2 distinct integers a, b from 1 to k, add the clause $\overline{X}_{p,a} \vee \overline{X}_{p,b}$

Every vertex in position of clique must share an edge w/ every other vertex in any position of clique For all pairs of vertices v_p and v_q that don't share an edge, add the clause: $\overline{X}_{p,a} \vee \overline{X}_{q,b}$ for every a & b b/w 1 & k

$$T(n) = aT(\lceil \tfrac{n}{b} \rceil) + O(n^d) \quad a > 0, b > 1, d \ge 0 \quad \text{then: } T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

2-DP Longest Common Substring Problem

The Longest Common Subsequence of two strings A and B is the longest string whose letters appear in order (but not necessarily consecutively) within both A and B.

3. Given two strings A and B of length $n > 0$ and $m > 0$, we will denote the length of the LCS of A and B by LLCS(A[1…n], B[1…m]). Describe LLCS(A[1…n], B[1…m]) as a recurrence relation over smaller instances. Use and generalize your work in the previous problems!

LLCS(A[1…n], B[1…m])

$$\begin{cases} 1 + \text{LLCS}(A[1\ldots n-1], B[1\ldots m-1]) & \text{if } A[n] = B[m] \\ \max \begin{cases} \text{LLCS}(A[1\ldots n-1], B[1\ldots m]) \\ \text{LLCS}(A[1\ldots n], B[1\ldots m-1]) \end{cases} & \text{if } A[n] \ne B[m] \end{cases}$$

either delete from A or B.

```
procedure Memo-LLCS(A[1..n], B[1..m])
  create a 2-dimensional array Soln[0..n][0..m]
  initialize all elements of Soln to −1
  Memo-Helper(A[1..n]B[1..m])

procedure Memo-Helper(A[1..i], B[1..j])
  ▷ As always with memoization, we wrap the recurrence with a check
  ▷ to see if the Soln array already contains the answer. If not, compute
  ▷ the answer via the recurrence and store it. If so, just return the answer
  if Soln[i][j] is −1 then
    if i == 0 or i == 0 then
      Soln[i][j] ← 0
    else if A[i] == B[j] then
      Soln[i][j] ← Memo-Helper(A[1..i − 1], B[1..j − 1]) + 1
    else
      Soln[i][j] ←
        max{Memo-Helper(A[1..i-1],B[1..j]), Memo-Helper(A[1..i],B[1..j-1])}
  return Soln[i][j]
```

	ε	c	co	cou	coun	count	countr	country
ε	0	0	0	0	0	0	0	0
t	0	0	0	0	0	1	1	1
ty	0	0	0	0	0	1	1	2
tyc	0	1	1	1	1	1	1	2
tyco	0	1	2	2	2	2	2	2
tycoo	0	1	2	2	2	2	2	2
tycoon	0	1	2	2	3	3	3	3

```
procedure Explain-LCS(A[1..n], B[1..m], Soln)
  ▷ Note: Soln[0..n][0..m] is a filled-in LLCS memoization table for A and B

  if n == 0 or m == 0 then ▷ base case
    return ""
  else
    if A[n] == B[m] then
      ▷ the final letters match, so, we add a letter to the LCS
      return Explain-LCS(A[1..n − 1], B[1..m − 1], Soln) +A[n]
    else
      ▷ which recursive call yielded the max?
      if Soln[n − 1][m] ≥ Soln[n][m − 1] then
        ▷ we don't use the last letter of A in the solution
        return Explain-LCS(A[1..n − 1], B[1..m], Soln)
      else
        ▷ we don't use the last letter of B in the solution
        return Explain-LCS(A[1..n], B[1..m − 1], Soln)
```

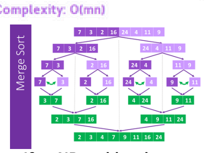
Memo

```
procedure DP-LLCS(A[1..n], B[1..m])
  create a 2-dimensional array Soln[0..n][0..m]

  ▷ fill in the base cases
  for i from 0 to n do
    Soln[i][0] ← 0
  for j from 1 to m do
    Soln[0][j] ← 0
  ▷ fill in the recursive cases column-by-column, top-to-bottom
  for i from 1 to n do
    for j from 1 to m do
      if A[i] = B[j] then
        Soln[i][j] ← Soln[i − 1][j − 1] + 1
      else
        Soln[i][j] ← max{ Soln[i − 1][j], Soln[i][j − 1] }
  return Soln[n][m]
```

DP

Runtime = O(mn) m is first word length, n length of second
#subproblems for memoized and DP is O(mn)
Space Complexity: O(mn)



Step 1: Split sub-lists i two until you reach pair of values.

Step 3: Sort/swap pai of values if needed.

Step 4: Merge and so sub-lists and repeat proces till you merge the full list.

If an NP problem has a polynomial number of possible certificates, then it is in P. It can be solved in polynomial time because you can just check every single certificate to verify it working. Polynomial num of certificates * polynomial time to check each.

Hamiltonian Cycle: cycle that visits each vertex exactly once. A Hamiltonian path that starts and ends at adjacent vertices can be completed by adding one more edge to form a Hamiltonian cycle, and removing any edge from a Hamiltonian cycle produces a Hamiltonian path.

Travelling Salesman: Given a set of cities and the distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point

The Hamiltonian cycle problem is a special case of the travelling salesman problem, obtained by setting the distance between two cities to one if they are adjacent and two otherwise, and verifying that the total distance travelled is equal to n. If so, the route is a Hamiltonian cycle.

Vertex Cover: A set of vertices that includes at least one endpoint of every edge of the graph.

Dominating Set: For graph G, is a subset D of its vertices, such that any vertex of G is in D, or has a neighbor in D. The domination number γ(G) is the number of vertices in a smallest dominating set for G

Independent Matching Problem: Given an undirected graph G = (V, E), a matching is a subset M of the edges E such that each node of V is an endpoint of at most one edge in M . A matching is independent if there is no edge (u, v) in G where u and v are the endpoints of two different edges of the matching M.

Max Cut Problem: Input: G(V,E), output is a partition of V that maximizes the edges crossing

Greedy algo: Easy to do O(n^2), Can be done O(n*m)

NP Complete Problems.

1. ***Independent Set.*** For a graph $G = (V, E)$, a subset of vertices $S \subseteq V$ is *independent* if no two vertices in S that are joined by an edge. Given a graph G and $k \in \mathbb{N}$, does G contain an independent set of size k or larger?
2. ***Vertex Cover.*** For a graph $G = (V, E)$, a subset of vertices $S \subseteq V$ is a *vertex cover* if every edge $e \in E$ has at least one of its endpoints in S. Given a graph G and $k \in \mathbb{N}$, does G contain a vertex cover of size k or smaller?
3. ***Set Packing.*** Given an n-element set U, a collection of subsets $\{S_1, S_2, \ldots, S_m\} \subset U$ and a number $k \in \mathbb{N}$, does there exists a collection of at least k of those sets with the property that no two of them intersect?
4. ***Set Cover.*** Given an n-element set U, a collection of subsets $\{S_1, S_2, \ldots, S_m\} \subset U$ and a number $k \in \mathbb{N}$, does there exists a collection of at most k of those sets whose union is equal to U?
5. ***Clique.*** For a graph $G = (V, E)$, a subset of vertices $S \subseteq V$ is a *clique* if every pair of vertices in V is joined by an edge. Given a graph G and $k \in \mathbb{N}$, does G contain a clique of size k or larger?
6. ***Subset Sum.*** Given a set of n integers $V = \{v_1, v_2, \ldots, v_n\}$, is there a subset $U \subseteq V$ such that $\sum_{u_i \in U} u_i = k$?
7. ***Set Partition.*** Given a set of n integers $V = \{v_1, v_2, \ldots, v_n\}$, can the elements of V be partitioned into two sets U and $V - U$ such that $\sum_{u_i \in U} u_i = \sum_{u_i \in V-U} u_i$?
8. ***Graph Coloring.*** A graph $G = (V, E)$ is said to be k-colorable if the endpoints of any edge (u, v) could be colored using different colors when there is total of k available colors. Given a graph G and $k \in \mathbb{N}$, does G have a k-coloring?
9. ***3D Matching.*** Given disjoint sets X, Y, Z each of size n, and given a set $T \subseteq X \times Y \times Z$ of ordered triples, does there exist a subset of n triples in T such that each element of $X \cup Y \cup Z$ is contained in *exactly one* of those triples?
10. ***SAT (Satisfiability).***
    - Let $X = \{x_1, \ldots, x_n\}$ be a set of n Boolean variables [i.e. each x_i is 0 or 1].
    - A term t_i over X is either one of the variables x_i or its negation $\overline{x}_i$.
    - A clause C_j of length l is a disjunction of distinct terms: $C_j = t_1 \vee t_2 \vee \cdots \vee t_l$.
    - A collection of clauses is the conjuction $C_1 \wedge C_2 \wedge \cdots \wedge C_k$.

    A truth assignment is some assignment of values of 0 or 1 to each $x_i \in X$. In other words, a truth assignment is a function $f : X \to [0, 1]$. The collection of clauses $C_1 \wedge C_2 \wedge \cdots \wedge C_k$ is *satisfiable* if there exists a truth assignment that evaluates the collection to 1.

    A problem is called l-SAT if the length of all of its clauses C_j is exactly l.

Example 4. Master theorem doesn't apply in the following cases:

- $T(n) = 2^n T\left(\frac{n}{2}\right) + 1$ [$a = 2^n$ is not a constant]
- $T(n) = \frac{1}{2} T\left(\frac{n}{2}\right) + 1$ [$a = \frac{1}{2}$ is not greater or equal to 1]
- $T(n) = 2T(n) + 1$ [$b = 1$ is not greater than 1]
- $T(n) = T\left(\frac{n}{2}\right) - n \log n$ [$f(n) = -n \log n$ is not positive]
- $T(n) = T(n-1) + 1$ [recurrence is not in the right form]
- $T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + 1$ [recurrence is not in the right form]

	Best	Average	Worst	Worst Space
Quick Sort:	O(nlogn)	O(nlogn)	O(n²)	O(logn)
Merge Sort:	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Selection:	O(n²)	O(n²)	O(n²)	O(1)
Insertion:	O(n)	O(n²)	O(n²)	O(1)
Bubble:	O(n)	O(n²)	O(n²)	O(1)

Consider a class of the change-making problem for which the greedy algorithm returns an optimal solution (that is, the one that uses the fewest coins). What can we say about the dynamic programming solution to this problem?

◉ The dynamic programming solution will be optimal, but generally slower than the greedy solution

$$R(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ \max_{k=j \text{ to } n} \{S[k-j+1] + R(i-1,k)\} & \text{if } 1 \le i \le m \text{ and } 1 \le j \le n. \end{cases}$$

```
function IterativeMystery:
    // initialize table; we choose to keep the zero rows/columns
    initialize a zero-indexed (m+1)*(n+1) array R

    // handle base cases
    Set R[i][0] = 0 for all i and R[0][j] = 0 for all j

    for i=1 to m:
        for j=n to 1:
            maxSoFar = -inf
            for k=j to n:
                val = S[k-j+1] + R[i-1][k]
                if val > maxSoFar:
                    maxSoFar = val
            R[i][j] = maxSoFar

    return R[m][n]
```

Runtime is theta((n^2)*m) time, and theta(n*m) space.

Q3.1 Greedy Algorithm
3 Points

Briefly describe a simple greedy strategy to determine a distribution of exam bundles. You must provide a plain English description of your greedy strategy, and should only provide pseudocode if you feel it is necessary to clarify details of your algorithm. (We suspect that if you need pseudocode, your approach is too complicated.)

Pick the bundles from left to right, starting from the first ta and giving the current TA the same bundle until giving the bundle to the TA will cause it to exceed m. If this happens, then move onto the next TA.

Q3.
1 Po
The
solu

```
function DeterministicSelect(A[1..n], k) // returns the element of rank k in an array A of n
distinct numbers, where 1 ≤ k ≤ n
  if n ≥ 5 then
    Choose pivot element p using the procedure described above
    Let Lesser be an array of all elements from A less than p
    Let Greater be an array of all elements from A greater than p
    if |Lesser| = k - 1 then
      return p
    else
      if |Lesser| > k - 1 then // all smaller elts are in Lesser; k is unchanged
        return DeterministicSelect(Lesser, k)
      else// |Lesser| < k − 1
        // subtract from k the number of smaller elts removed
        return DeterministicSelect(Greater, k− |Lesser| −1)
  else
    sort A and return A[k − 1]
```

Deterministic Select

Question 3.3: Give a reasonable "greedy stays ahead" lemma that you could use to prove your algorithm is optimal. (You do not need to prove the lemma.)

Solution: The first k TAs in the greedy solution will be carrying at least as many total bundles as the first k TAs in the optimal solution.