Divide and Conquer:

Break problems into subparts that are themselves smaller instances of the same type of problem. Recursively evaluate these problems. Combine their result correctly.

Multiplication: Normal way to multiply is $(a+bi)(c+di) = ac-bd+(bc+ad)i$ which takes 4 multiplications. Can we do it in three? Yes, since $bc+ad = (a+b)(c+d) - ac - bd$. This improvement significantly speeds up multiplication when applied recursively.

Algorithm: x can be split into $x_L B^m + x_R$ and y can be split into $y_L B^m + y_R$. Recursively calculate $x_L y_L = z_0$, $x_R y_R = z_1$, $z_2 = (x_R + x_L)(y_R + y_L)$, and $z_3 = z_2 - z_1 - z_0$. Return $z_1 * 2^n + z_4 * 2^{n/2} + z_2$. The runtime is $3T(n/2) + O(n)$ which is $O(n^{1.58})$ if $B=2, m = n/2$.

Merge sort: If $n > 1$: return merge(mergesort($a[1, \ldots, n/2]$), mergesort($a[n/2, \ldots, n]$)) else return a. We can prove this by induction, trivially the base case is correct, and we can safely assume that the returned lists by the recursive subcalls returns the sorted list. Therefore, as long as the merge routine is correct, our algorithm is perfect! Runtime: $2T(n/2) + O(n)$ which is $O(n \log n)$ by master's theorem (since we split the problem into two part, recurse on both and merge)

Median finding and k-finding: Random pivot select: Consider any number $v \in S$. Split S into S_L, S_R, and S_v where S_L has values less than v, S_R has values less than v, and S_v has values equal to v. We know that the k^{th} element is in S_L if $|S_L| \geq k$ where the element is still the k^{th} greatest in S_L, in S_R if $|S_L| + |S_v| < k$ where the k^{th} element is the $(k - |S_L| - |S_v|)$ element in S_R. If $|S_L| < k \leq |S_L| + |S_v|$, return v as that is our k^{th} element in S. Runtime: If we always pick the median the runtime is $T(n) = T(n/2) + O(n) = O(n)$. However, if we always chose the worst v value, our runtime blows up to $\Theta(n^2)$. However, the amortized runtime is still $O(n)$ since the worst case scenario is unlikely.

Matrix Multiplication: $Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}$ but this takes $O(n^3)$ time. Is there a faster method?

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix}\begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{bmatrix}$$ which can be done in 8 multiplications which is still $O(n^3)$

Can we can solve this faster? We can actually solve the above in 7 multiplications.

$P_1 = A(F-H)$, $P_2 = (A+B)H$, $P_3 = (C+D)E$, $P_4 = (G-E)D$, $P_5 = (A+D)(E+H)$, $P_6 = (B-D)(G+H)$, $P_7 = (A-C)(E+F)$

and, $XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix}\begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} P_5+P_4-P_2+P_6 & P_1+P_2 \\ P_3+P_4 & P_5+P_1-P_3+P_7 \end{bmatrix}$; $T(n) = 7T(n/2) + O(n^2) = O(n^{2.82})$ } Strassen's Alg.

Fast Fourier Transform: If we have polynomial $A(x) = \sum_{i=0}^{d} a_i x^i$ and $B(x) = \sum_{i=0}^{d} b_i x^i$ then the k^{th} coefficient $c_k = \sum_{i=0}^{k} a_i b_{k-i}$ therefore finding all coefficients takes $\Theta(d^2)$. Can we do better? Yes, Divide and Conquer.

FFT matrix looks like:

$$M_n(\omega) = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{(n-1)} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & & & \vdots \\ 1 & \omega^j & \omega^{2j} & \cdots & \omega^{j(n-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega^{(n-1)} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{pmatrix} \quad \omega = e^{-\frac{2\pi i}{n}}$$

and $(M_n(\omega))^{-1} = \frac{1}{n} M_n(\omega^{-1})$!

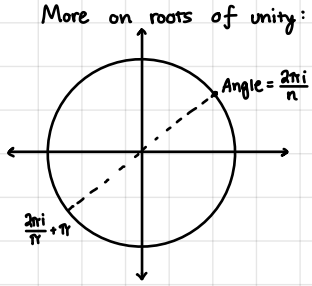
The FFT algorithm breaks $M_n(\omega)$ into a product of simple matrices!

FFT algorithm:

```
FFT(a, ω):
    if ω = 1 : return a
    (s_0, s_1, ..., s_{n/2-1}) = FFT((a_0, a_2, ..., a_{n-2}), ω^2)
    (s'_0, s'_1, ..., s'_{n/2-1}) = FFT((a_1, a_3, .., a_{n-1}), ω^2)
    For j=0 to n/2-1:
        r_j = s_j + ω^j s'_j
        r_{j+n/2} = s_j - ω^j s'_j
    return (r_0, r_1, ..., r_{n-1})
```

$iFFT(a, \omega) = \frac{1}{n} \cdot FFT(a, \omega^{-1})$

More on roots of unity:



Solutions to equation $z^n = 1$. For even n, solutions are plus minus paired

Applications of Fast Fourier Transform:

K-sum: All problems of this type involve computing the number of ways to achieve some s given a collection of elements. Strategy: Construct polynomials by encoding elements as the exponents to yield a product whose terms ax^b encode the following information, a is the number of ways to combine elements to achieve sum b.

Convolution: Observe that polynomial multiplication at its core is a convolution. To see why consider multiplying $p(x) = \sum_{i=0}^{m} a_i x_i$ and $q(x) = \sum_{j=0}^{n} b_j x^j$ which is

$r(x) = \left(\sum_{i=0}^{m} a_i x^i\right)\left(\sum_{j=0}^{n} b_j x^j\right) = \sum_{i=0}^{m}\sum_{j=0}^{n} a_i b_j x^i x^j = \sum_{i=0}^{m}\sum_{j=0}^{n} a_i b_{k-i} x^k$

$= \sum_{k=0}^{m+n}\left(\sum_{i=0}^{k} a_i b_{k-j}\right)x^k$ where the inside of the parantheses is the discrete convolution

Strategy: Construct two polynomial by filling their coefficients in with useful information from the problem. While doing this you want to think about the things you want to multiply, then sum together. Or think about what a and b should represent in the summation $\sum_{i=0}^{k} a_i b_{k-i}$.

Shifted Dot Products / Cross correlation: Cross correlation is often reffered to as a "sliding dot product" operation on two vectors a and b. Each coefficient in the output is of the form $\sum_{i=0}^{k} a_i b_{j+i}$ where k is the length of one of a or b. The steps to do this are:

Reverse b to yield b^R → Generate polynomials A and B^R → Use FFT to compute the product product $C(x) = (A \cdot B^R)(x)$.

→ The coefficients of C are the crosscorrelation of a and b

Proof of Correctness: $A(x) = a_0 + a_1 x^1 + \cdots + a_m x^m$ and let $B^R(x) = b_n + b_{n-1} x^1 + \cdots + b_0 x^n$. If we use the discrete convolution formula previously mentioned then

$$\sum_{j=0}^{k} a_j b_{n-(k-j)} = \sum_{j=0}^{k} a_j b_{(n-k)+j}$$

Which is the cross correlation between a and b at shift $n-k$. We use this strategy whenever we want some kind of sliding window that computes a dot product at each possible starting location.

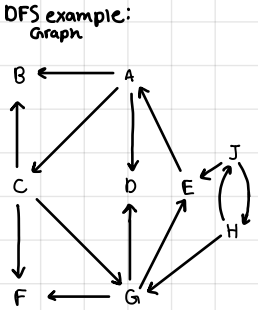
Graph Theory

```
def explore(G,v):
    visited = true
    previsit(v)
        for each {v,u} ∈ E:
            if not visited(u):
                explore(u)
        postvisit(v)
def dfs(G):
        for all v ∈ V:
            if not visited(v):
                explore(v)
def bfs(G,s):
    for all u in V:
        dist(u) = ∞
    dist(s) = 0
    Q = [s]
    while Q is not empty:
        v = Q.eject()
        for each (v,u) ∈ E
            if dist(u) == ∞:
                Q.inject(u)
                dist(u) = dist(v)+1
```

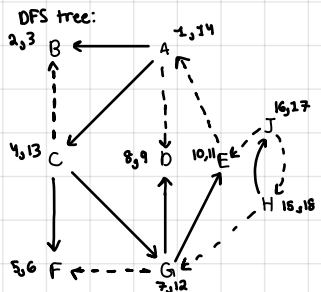
```
Dijkstra's(G,l,s):
    for all u ∈ V:
        dist(u) = ∞, prev(u) = nil
    dist(s) = 0
    H = makequeue(V)
    while H is not empty:
        u = deletemin(H)
        for all (u,v) in E:
            if dist(v) > dist(u) + l(u,v)
                dist(v) = dist(u) + l(u,v)
                prev(v) = u
def bellman-ford(G,l,s):
    for all u in V:
        dist(u) = ∞, prev(u) = nil
    dist(s) = 0
    repeat |V|-1 times:
        for all e in E: update(e)
update(e=(u,v)): d(v) = min(d(v), d(u)+l(u,v))
def dag-shortest path(G,l,s):
    Linearize G
        for each u ∈ V in linear order:
            for all (u,v) ∈ E: update(u,v)
```

DFS example:

Graph



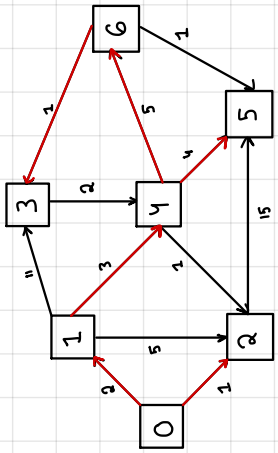
DFS tree:



An edge $(u,v) \in E$ is a tree/Forward edge if $pre(u) < pre(v) < post(v) < post(u)$ [( )]; a back edge if $pre(v) < pre(u) < post(u) < post(v)$ [( )]; and a cross edge if $pre(v) < post(v) < pre(u) < post(u)$ [ ] ( ).

SCC's and topological sort: Vertex with largest post order is a source. Perform DFS on source vertex to get topological sort. SCC's is a maximal subset of strongly connected vertices. The highest post visit number in the reversed graph is a sink.

SCC detection algorithm: Kosaraju's algorithm



Djikstra's Queue:

1	2	3	4	5	6
2	①	∞	∞	∞	∞
②	①	∞	∞	16	∞
②	①	13	⑤	16	∞
②	①	13	⑤	⑨	10
②	①	13	⑤	⑨	⑩
②	①	⑪	⑤	⑨	⑩

Kruskals: Kruskal's MST algorithm starts with the empty graph and then selects edges from E according to the rule: "Repeatedly add the next edge that doesn't produce a cycle."

Cut property: Suppose edges X are a part of some MST of $G=(V,E)$. Pick any subset of nodes S for which X does not between S and V-S, and let e be the lightest edge across this partition. Then $X \cup \{e\}$ is a part of some MST.

```
Kruskals(G,w):
    for all u ∈ V: makeset({u})
    X = {} and then Sort the edges E by weight
    for all edges {u,v} ∈ E: if find(u) ≠ find(v): add edge {u,v} to X and union(u,v)
```

makeset: $\pi(x) = x$; $rank(x) = 0$

find: while $x \neq \pi(x)$: $x = \pi(x)$; return x

union: $r_x = find(x)$, $r_y = find(y)$; if $r_x = r_y$: return; if $rank(r_x) > rank(r_y)$: $\pi(r_y) = r_x$; else: $\pi(r_x) = r_y$; if $rank(r_x) = rank(r_y)$: $rank(r_y) = rank(r_y) + 1$

Rank is height of the subtree rooted at that node

With path compression func find(x) → if $x \neq \pi(x)$: $\pi(x) = find(\pi(x))$ makes the amortized cost of find and union operations to be barely above O(1). In fact, its $O(\log^* n)$.

```
Prims(G,w):
    for all u ∈ V:
        cost(u) = ∞, prev(u) = nil
    cost(u_0) = 0
    H = makequeue(V)
    while H is not empty:
        v = deletemin(H)
        for each {v,z} in E:
            if cost(z) > w(v,w)
                cost(z) = w(v,z)
                prev(z) = v
                decreasekey(H,z)
```

Asymptotics:

Limit Rule: If $f(n), g(n) \geq 0$ then

- if $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$, then $f(n) = O(g(n))$
- if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$, for some $c > 0$, then $f(n) = \Theta(g(n))$
- if $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$, then $f(n) = \Omega(g(n))$

If the recurrence relation is of the form $T(n) = aT(\frac{n}{b}) + O(n^d)$ for some constants $a > 0$, $b > 1$, and $d \geq 0$, then

$$T(n) \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

1. "Unraveling" the recurrance relation: Recursively keep plugging in smaller subproblems to T(n) to find a pattern
2. Draw a tree! enough said
3. Change of variables: We can make a change of variable to mold our recurrence into a nicer form to work with.

   $T(n) = 3T(n^{1/3}) + O(\log n)$

   $n = e^x$ Then $T(e^x) = 3T(e^{x/3}) + O(x)$

   If $S(x) = T(e^x) = T(n)$, then $S(x) = 3S(x/3) + O(x)$.

   By master's theorem this is $S(x) = O(x \log x)$

   $= O(\log n \log\log n)$.
4. Squeeze or Squeeze + Guess & Check

Greedy Algorithms: greedy algorithms make the locally optimal choice at each step. 0 on left, 1 on right

Huffman Encoding: Given characters/frequencies $\{c_i, f_i\}$ encode each character in binary so that the final encoding is maximally efficient. Main idea: Find the two symbols with the smallest frequency, lets say i and j, and make them children of a new node that has frequency $f_i + f_j$ and repeat until one node remains. Cost (average bit/character): $\sum_{i=1}^{n} f_i l_i$, where l_i is the number of bits to encode the i^{th} character.

Horn satisfiability: Greedy algorithm for finding a set of Boolean assignment s.t all clauses are satisfied.

Tips on how to approach: Try to think step by step, not long term. What is the best thing I can do at this step? Be dumb for greedy algorithms.

Exchange Argument / Proof by Contradiction

Assume Optimal solution is O with sequence of choices $\{o_1, o_2, \ldots\}$. Assume greedy solution is G with sequence of choices $\{g_1, g_2, \ldots\}$.

WLOG, consider first point of discrepancy between G and O at i. Prove g_i is a better/equivalent choice to o_i making it optimal. Then apply this generally.

Linear programming

LP basics

1. It can either be a minimization or maximization problem
2. Its constraints can either be equations and/or inequalities
3. The variables are often restricted to be nonegative but can be unrestricted in sign

LP reductions and conversions

Primal: $\max c^T x$, $Ax \le b$, $x \ge 0$ $\Longrightarrow$ Dual: $\min y^T b$, $y^T A \ge c^T$, $y \ge 0$

Changing objective: $\max c^T x = \min -c^T x$; $\min c^T x = \max -c^T x$

Inequality to Equality: $ax \le b \longrightarrow ax + s = b$, $s \ge 0$

Equality to Inequality: $ax = b \longrightarrow ax \le b$, $ax \ge b$

Unrestricted variable: $x \in \mathbb{R} \longrightarrow x = x_+ - x_-$, $x_+, x_- \ge 0$

Zero-Sum Games

Given a 2D matrix where the rows are player A's moves, the columns are player B's moves, and the values are A's rewards for each move. Whichever player goes first has to announce their strategy first. Strategy is the vector of probabilities of playing specific moves.

If player A goes first, player B will pick the move that minimizes A's reward. So A's best strategy will be the maximum of the minimum of player B's moves. If player B goes first, player A will pick the move that maximizes A's reward. So B's best move will be the minimum of A's moves. These two equations will be the dual of each other.

Basic example: A's strategy is $[x_1, x_2]$; B's strategy is $[y_1, y_2]$

	B1	B2
A1	a	c
A2	b	d

A picks x_1, x_2 to maximize the value $\min\{ax_1 + bx_2, cx_1 + dx_2\}$

B picks y_1, y_2 to minimize the value $\max\{ay_1 + cy_2, by_1 + dy_2\}$

More examples (LP construction):

		Friend: rock	paper	scissors
	rock	-10	3	3
You:	paper	4	-1	-3
	scissors	6	-9	2

my optimal strat: r, s, p is probability that I play rock, scissors, and paper respectively; let z be expected payoff if opp plays well.

$\max z$

$s.t.\ -10r + 3p + 3s \ge z$ (opp choose r)

$4r - p - 3s \ge z$ (opp choose p)

$6r - 9p + 2s \ge z$ (opp choose s)

Opps Optimals strat:

$\min z$

$s.t.\ -10r + 3p + 3s \le z$ (I choose rock)

$4r - p - 3s \le z$ (I choose paper)

$6r - 9p + 2s \le z$ (I choose scissors)

Duality:

Weak Duality: Opt. Primal $\le$ Opt. Dual

Strong Duality: Opt. Primal = Opt. Dual

Simplex Algorithm:

Start at vertex x^* → Find neighbor y^* with maximal value

If value(y^*) > value(x^*) move to y^* else stop. → Repeat

Network Flow

A flow f from s to t in a directed graph G whose edges have capacities $c_e \ge 0$ has the following properties:

- flow on each edge f_e must satisfy: $0 \le f_e \le c_e$.
- Conservation of flow: flow entering vertex = flow leaving vertex

Goal: Find maximum flow $val(f^*)$ where $val(f) = \sum_{(s,i) \in E} f_{(s,i)}$

Ford-Fulkerson Algorithm

Residual graph: A directed graph R whose edge weight is denoted as the residual capacity $r(u,v)$.

$r(u,v) = c(u,v) - f(u,v)$ if $(u,v) \in E$ and $f(u,v) < c(u,v)$

$r(u,v) = c(u,v) + f(v,u)$ if $(v,u) \in E$ and $f(v,u) > 0$

where $c(u,v)$ is the capacity of (u,v) on G, $f(u,v)$ is flow on (u,v).

For example: If $c(u,v) = 3$, $c(v,u) = 4$ and $f(u,v) = 2$ then $r(u,v) = 1$ and $r(v,u) = 6$.

Algorithm:

1. find a path from s to t and route max flow through this path
2. Update capacities by updating the residual graph
3. Repeat 1 and 2 until algorithm halts (no path s to t is found)

Explanation behind algorithm:

- Intuition: The back edges in residual graph can undo suboptimal flows in some edges for future iterations
- Runtime: $O(|V| \cdot |E|^2)$ if using BFS to find the paths

Max-flow Min-cut Theorem

An (s,t)-cut partitions V into two subsets S, T, where $s \in S, t \in T$

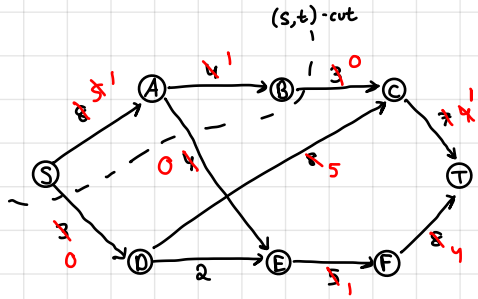
Capacity of (s,t)-cut C: $val(C) = \sum_{e \in \{(u,v) | u \in S, v \in T\}} c_e$

Theorem: $val(f^*) = val(C^*)$. Maximum flow is equal to the minimum cut. In the residual graph, the set of vertices, X, reachable from s, yields the cut $(X, V \setminus X)$ whose capacity equals the max flow.

When considering (s,t)-cuts' weights, only consider edges going from the partition containing s to the partition containing t. (Any edge going the opposite direction must have zero flow.

Naive upperbound on iteration: If edges in the original network have integer capacities $\le C$, then the maximum flow is at max $C \cdot |E|$ and the number of iterations is also, therefore, $C \cdot |E|$.

Example:



Iterations:

1: S → A → B → C → T

2: S → A → E → F → T

3: S → D → C → T

Final max-flow: 10

Dynamic Programming Paradigms

Notes:

Edit distance:

The edit distance between two words is by how much two words differ. Another way to think about it is the best possible alignment between two words (or the minimum number of insertions deletions, and substitution of characters in order to transform the first word to the other)

Subproblem: let $E(i,j)$ be the edit distance between the first i letters of the first word and first j letters of the second word.

Recurrance Relation:

$$E(i,j) = \min \begin{cases} 1 + E(i-1,j) \\ 1 + E(i,j-1) \\ diff(i,j) + E(i-1,j-1) \end{cases}$$

Knapsack Problems

Given a knapsack with max capacity W and n items of weight $w_1, \dots, w_n$ and dollar value $v_1, \dots, v_n$ what is the most valuable combination of unique items to fit in this knapsack.

Subproblem: $f(i,w)$ is the max value acheiveable with a knapsack of capacity w and items $1, \dots, i$.

Recurrance Relation: $f(i,w) = \max\{f(i-1,w), f(i-1,w-w_i) + v_i\}$

With repetition? subproblem: $K(w)$ = maximum value acheiveable with a knapsack of capacity w

Recurrance Relation: $K(w) = \max_{i:w_i \le w} \{K(w-w_i) + v_i\}$

Matrix Multiplication: Matrix multiplication is associative so what if we wanted to minimize to cost of multiplying matrices (w.r.t multiplications)?

Subproblem: let $C(i,j)$ = minimum cost of multiplying $A_i \times \dots \times A_j$

Recurrance Relation: $C(i,j) = \min_{i \le k < j} \{C(i,k) + C(k+1,j) + m_{i-1} \cdot m_k \cdot m_j\}$

Shortest DAG Path

Subproblem: let $dist(D)$ be the shortest path to D.

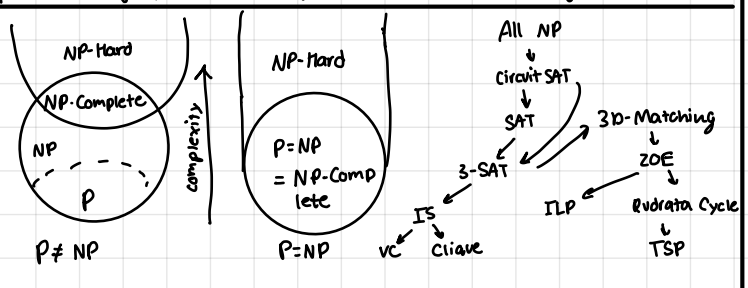
Recurrance Relation: $dist(v) = \min_{(u,v) \in E} \{dist(u) + l(u,v)\}$ (solve in topological order)

Longest Increasing Subsequence:

$L(j)$ = length of longest increasing subsequence ending at j

Recurance Relation: $L(j) = 1 + \max_{(i,j) \in E} \{L(i)\}$

Tips: Define an appropriate subproblem with relevant parameters, ensuring that parameters fully determine the subproblem → Given access to all sub-problems, develop an appropriate relation to solve the current subproblem. → test using a small proof or small cases. You should only need previous (slightly smaller) subproblems to solve the larger one



Mechanical potpourri

NP-Complete problem: Problems in NP s.t every problem in NP reduce to them.

How to Prove A is NP-Complete:

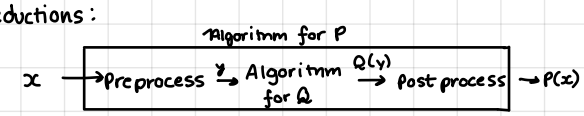
1. Show A is NP-Hard by designing a verification algorithm
2. Choose an NP-Complete problem B
3. Show that A is NP-Hard by reducing B to A
4. Since $A \in NP$ and $A \in$ NP-Hard, $A \in$ NP-Complete

TLDR: Prove $A \in NP$ and prove $B \leq_p A$ (A is at least as hard as B) where B is any NP-Complete problem.

In scope NP-Complete Problems:

- 3-SAT: Given a set of clauses each containing 1 to 3 literals, (for example $(\bar{x} \vee y \vee \bar{z})(x \vee \bar{y} \vee z)(\bar{x} \vee \bar{y})$), find a satisfying truth assignment
- Independant Set: Given a graph and a number g, find the set of g pairwise non-adjacent vertices.
- Vertex Cover: Given a graph and a budget b, find b vertices s.t an endpoint of every edge is in the cover set.
- Integer Linear Programming: Given a system of linear inequalities, find a feasible integer solution.
- Rudrata Path: Given a graph g and two vertices s and t, find a path starting at s and ending at t.
- Set Cover: Given a set of elements E and several subsets of E $(S_1, \ldots, S_m)$ with budget b, select b of these subsets s.t their union is E.
- The Traveling Saleman Problem: We are given n vertices and all $n(n-1)/2$ distances between them as well as a budget b. We are asked to find a tour, a cycle that passes through every vertex once, with a total cost b or less.

Reductions:



Reducing from P to Q $\equiv P \to Q \equiv P \leq_p Q$

Both pre and post processing take polynomial time.

Implication: If there exists an efficient algorithm for problem Q, then there exists an efficient algorithm for P. If there's no efficient algorithm for P, then there is no efficient algorithm for Q

Reductions:

3-SAT → IS: Construct edges within clauses and between x and $\bar{x}$

If independant set of size m then satisfyable.

IS → VC: Suppose we have $G=(V,E)$. Find the independant set, then just take set of other vertices to get vertex cover.

IS → Clique: Indep. set S of graph G is a clique C of graph $\overline{G}$.

Exam Example Problem:

Show that Far-Away Points is NP-Complete.

Given an instance of the Independant Set problem we will construct an instance of the far-away points problem (desribe a reduction converting an IS problem to FAP). Prove reduction by proving that a solution to FAP => IS and solution to IS => FAP.

Coping with NP-Completeness (Approximations)

An approx. Algorithm with an approx. ratio α to a problem A means that the solution found by this algorithm is between OPT(A) and $\alpha \cdot OPT(A)$. $\alpha > 1$ for minimization problems and $\alpha < 1$ for maximization problems.

2-Approximation for Vertex Cover:

- While there are edge: select any edge e and delete all edges that share an end point with e
- Approximated vertex cover is the endpoints of the selected edges.
- In order to cover all the edges, the optimal vertex cover must be at least the number of edges selected, which is atleast half of the number of endpoints from the approximation algo. above
- Therefore the result has at most twice as much as the optimal solution and achieves an approx ratio of $\alpha = 2$.

2-Approximation for Traveling Salesman Problem (Graph must satisfy Δ prop.)

- Build an MST from the graph G.
- Run dfs on the MST
- Delete any repeating vertices in the pre-order visit

ILP integer relaxation: We get this by removing integrality constraint. Now to get a valid approximation, we must round the results of the relaxed LP. The manner in which we round must be consistent with the constraints of the original LP.

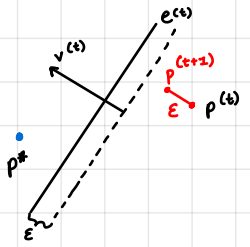
Example:

Gradient Descent

Consider a game between Alice and Bob (in 2D). And there is a point $p^* = (x^*, y^*)$ on the 2-d plane.

Algorithm:
In the i^{th} iteration,
1. Bob proposes a point $p^{(t)} = (x^{(t)}, y^{(t)})$
2. Alice finds a seperating line $\ell^{(t)}$ that seperates Bob's current point $p^{(t)}$ from the destination p^*. Formally, p^* is on one side of the line $\ell^{(t)}$, but $p^{(t)}$ is atleast ϵ-away from the line on the opposite side

Alice loses when she cannot find a seperating line.



Let $v^{(t)}$ denote the unit vector perpendicular to Alice's seperating line $\ell^{(t)}$. Consider the following strategy for Bob: $p^{(t+1)} = p^{(t)} + \epsilon v^{(t)}$ (i.e. Bob moves directly towards the seperating line)
With this strategy, Bob gets closer and closer to p^*.
In fact, in every iteration, the squared distance of Bob's point $p^{(t)}$ to p^* decreases by ϵ^2
$\Rightarrow$ Bob's squared distance decreases by ϵ^2 every iteration. $\Rightarrow$ If Bob plays this strategy, then Alice will fail to find the seperating line in $\left\lfloor \frac{\|p^{(0)} - p^*\|^2}{\epsilon^2} \right\rfloor$ iterations.

A **HYPERPLANE** is the set of points $x \in \mathbb{R}^d$ that satisfy an equation of the form $\langle v, x \rangle - \theta = 0$.

Algorithm: In the i^{th} iteration,
1. Bob proposes a point $p^{(t)} \in \mathbb{R}^{(d)}$
2. Alice finds a seperating hyperplane that seperates Bob's current point $p^{(t)}$ from the destination p^*. Formally, Alice finds $v^{(t)}, \theta^{(t)}$ s.t v is a unit vector and $\langle v^{(t)}, p^* \rangle - \theta^{(t)} \geq 0$ while $\langle v^{(t)}, p^{(t)} \rangle - \theta^{(t)} \leq \epsilon$.
Alice loses when there is no seperating hyperplane.

A successful strategy for Bob is therefore $p^{(t+1)} = p^{(t)} + \eta v^{(t)}$.

Feasibility for linear programs

Find a point $x \in \mathbb{R}^d$ s.t
$\langle a_1, x \rangle \geq b_1$
$\langle a_2, x \rangle \geq b_2$
$\vdots$
$\langle a_m, x \rangle \geq b_m$

The point pursuit game can be used to solve this LP efficiently. In each iteration Bob proposes a solution $x^{(t)}$. Alice looks for a constraint violated by Bob's solution, i.e., a constraint for which $\langle a_i, x^{(t)} \rangle \leq b_i - \epsilon$

(Assuming that $\|a_i\| = 1$ else we can divide by a scaling factor)

LP via Gradient Descent

$x^0 \leftarrow 1$
for $t = 0$ to T
- (Find violated constraint) Find a constraint $\langle a_i, x \rangle \geq b_i$ violated by the $x^{(t)}$ with error at least ϵ, i.e., $\langle a_i, x^{(t)} \rangle \leq b_i - \epsilon$.
If no constraints are violated, return $x^{(t)}$
- Set $x^{(t+1)} \leftarrow x^{(t)} + \eta \cdot a_i$

Return "no feasible solution within distance $\epsilon\sqrt{T}$ of the starting point

We can use this algorithm to even solve LPs with possibly exponential constraints.

Half space: A halfspace is a set of points on one side of a hyperplane, formally, for some $w \in \mathbb{R}^d$ and real number θ; $H = \{x \in \mathbb{R}^d \mid \langle w, x \rangle - \theta \geq 0\}$

A convex set is an intersection of a (possibly infinite) family of halfspaces.

Fact. If x is a point that is not in a convex set S then there exists a halfspace H such that $S \subseteq H$ but $x \notin H$. Additionally, a set $S \in \mathbb{R}^d$ is a convex set if for every pair of points $x, y \in S$, the line segment joining them is also in S. $x, y \in S \Rightarrow \lambda x + (1-\lambda) y \in S$ for all $\lambda \in [0, 1]$

A seperation oracle O for a convex set S is an algorithm that, given a point not in S, will find a halfspace seperating x from the set S. In our previous game, Alice was playing the role of a seperation oracle for the convex set that is the feasible region of an LP.

An ϵ-seperation oracle O for a convex set S is an algorithm that solves the following problem: Input; a point x that is atleast ϵ-away from the convex set S. Output; A halfspace $H = \{y \mid \langle w, y \rangle - \theta \geq 0\}$ such that $S \subseteq H$ but the point x is ϵ-away from H if $\|w\| = 1$ and $\langle w, x \rangle - \theta \leq -\epsilon$. Given an ϵ-seperating oracle O for a non-empty set S, the following algorithm finds a point x that is at most ϵ-away from the set S using $T = \Omega(D^2/\epsilon^2)$ - calls to the oracle O. D is the distance from the initial point $x^{(0)}$ from S.

Input: A seperation oracle O for some convex set S; Output: A paint x at most ϵ-away from S
for $t = 0$ to T:
Use the oracle O to find a halfspace that seperates the current point $x^{(t)}$ from the convex set S, i.e., $\langle w, x^{(t)} \rangle \leq -\epsilon$ while, $\langle w, x \rangle - \theta \geq 0$ for all $x \in S$. If no such seperating halfspace, rv $x^{(t)}$
else set $x^{(t+1)} \leftarrow x^{(t)} + \eta \cdot w$

A function $f : \mathbb{R}^d \to \mathbb{R}$ is convex if $\forall x, y \quad f(\frac{x+y}{2}) \leq (f(x) + f(y))/2$

Algorithm: Gradient descent for convex function

Input: A convex function $f : \mathbb{R}^d \to \mathbb{R}$ s.t $\|\nabla f(x)\| \leq B \ \forall x$, and a start point $x^{(0)}$

for $t = 0$ to T:
Set $x^{(t+1)} \leftarrow x^{(t)} - \epsilon/B \cdot \frac{\nabla f(x^{(t)})}{\|\nabla f(x^{(t)})\|}$

This algorithm essentially walks 'downhill' from the point x to x^* in the opposite direction of the gradient.

return "$x^{(t)}$ among $t = 0, \ldots, T$ with smallest value for $f(x^{(t)})$"

Discussion and Homework problems

Fair Allocation

There are N dollars to be allocated among n employees. Based on work experience, employee i is entitled to at least e_i dollars.
An allocation is fair if no subset of $n/10$ employees or fewer recieves more than half the dollars.
Find a fair allocation if possible.

Let x_i be the dollars employee i gets. Then the constraints are
$x_i \geq e_i \quad \forall i$
$\sum_{i \in S} x_i < \frac{N}{2} \quad \forall$ subsets S at most $n/10$ employees

Seperation Oracle:

We can verify the first set of constraints in $O(n)$ time by just checking each individual constraint.

To determine if any fairness constraints are broken, simply check whether the $n/10$ highest paid employees are allocated more than $N/2$ dollars. If they are, then we have found a broken constraint else no fairness constraint is broken.

Setting tolls

Input: An undirected unweighted graph $G_1 = (V, E)$, a pair of nodes s and t, and a real number B.

Output: A set of weight $w : E \to \mathbb{R}^+$ s.t:
1. For every s-t path P in the graph G_1, the total weight on the path is atleast 1
2. The total weight of all edges in the graph is at most B

Define an LP with the following constraints.

$\sum_{e \in p} w_e \geq 1 \quad \forall p$ that is an s-t path
$\sum_{e \in E} w_e \leq B$

We can verify the first constraint using djikstra's algorithm and check whether the total weight of the path is less than 1. If it is, then the constraint is violated, else none of the constraints in the first set are violated. We can check the second constraint easily by summing over the edge weights

Disk interior problem

Input: n circular disks on the 2d plane specified by $\{(x_i, y_i, R_i) \mid i = 1, \ldots, n\}$ where (x_i, y_i) is the center of a a circle with radius $= R_i$. Return a point approximately in all disks.

Seperation Oracle: