Stable Matching Problem

- n set of men and women, find a matching that is best for all.
- **Perfect Matching**: All men m and women w each appear in at most one pair of the matching
- **Unstable Pair**: Perfect matching where m prefers w and w prefers m to their current partners.

STRONG/WEAK INSTABILITY

Strong Instability - w & m both prefer each other over current partner
Weak Instability - w prefers m over current partner and m prefers w or is indifferent b/w the two, or m prefers w over current partner and w prefers m or is indifferent b/w the two

GALE-SHAPELY ALGORITHM

```
initially all m and w are free
while an unmatched man m hasn't proposed to every
    ↪ woman:
    w <- highest ranked woman in m's list that m
        ↪ hasn't proposed to
    if w is free:
        (m,w) is engaged
    else:
        if w prefers m' to m then:
            m is still free
        else w prefers m to m' then:
            (m,w) is engaged
            m' is set to free
```

Only n^2 proposals possible, thus $O(n^2)$

Algorithm Analysis

- **Big-O**: there exists constants $c > 0$ and $n_0 \geq 0$ s.t. $T(n) \leq c \cdot f(n)$ for all $n \geq n_0$
- **Big-Omega**: there exists constants $c > 0$ and $n_0 \geq 0$ s.t. $T(n) \geq c \cdot f(n)$ for all $n \geq n_0$
- **Big-Theta**: there exists constants $c_1, c_2 > 0$ and $n_0 \geq 0$ s.t. $c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$ for all $n \geq n_0$

Using limit theorem

- $\lim \frac{f(n)}{g(n)} = c > 0$, $f(n)$ is $\Theta(g(n))$
- $\lim \frac{f(n)}{g(n)} = c = 0$, $f(n)$ is $O(g(n))$

Can prove using limit that there is no constant c that acts as upper bound to show that $f(n) \neq O(g(n))$

MASTER THEOREM

$T(n) = aT(n/b) + f(n)$

- $a \geq 1$: is the number of subproblems
- $b > 0$: is the factor by which the subproblem size decreases
- **f(n)**: work to divide/merge subproblems

Given the recurrence relation
$T(n) = aT(n/b) + f(n)$, k = $\log_b a$

- **Case 1**: If $f(n) = O(n^{k-\epsilon})$ for $\epsilon > 0$, then $T(n) = \Theta(n^k)$
- **Case 2**: If $f(n) = \Theta(n^k)$, then T(n) = $\Theta(n^k \log n)$
- **Case 3**: If $f(n) = \Omega(n^{k+\epsilon})$ for $\epsilon > 0$ and $a \cdot f(n/b) \leq c \cdot f(n)$ for $c < 1$, then $T(n) = \Theta(f(n))$.

Note that $a \cdot f(n/b) \leq c \cdot f(n)$ holds if $f(n) = \theta(n^{k+\epsilon})$

Analysis of Master Theorem

Compare f(n) with $n^{log_b a}$

- **Case 1**: $n^{log_b a}$ is larger, hence T(n) = $\Theta(n^{log_b a})$
- **Case 3**: f(n) is larger, hence T(n) = Θ(f(n))
- **Case 2**: f(n) and $n^{log_b a}$ are of the same size, so T(n) = $\Theta(n^{log_b a} logn)$

Master Theorem Fails

Use iteration technique – Given $T(n) = T(x) + f(n)$, substitute n with x until a pattern is found, then generalize it to find the solution.
Special Cases: $T(n) = aT(n-b) + n^k$

- If $a < 1$, then $T(n) = O(n^k)$.
- If $a = 1$, then $T(n) = O(n^{k+1})$.
- If $a > 1$, then $T(n) = O(n^k \cdot a^{n/b})$.

Greedy Algorithms

Make the best choice at that time, locally optimal choice each time will lead to globally optimal solution.

- **Cashier Algorithm:** Pick the largest coin denomination possible to use the fewest number of coins. Optimal for 1,5,10,25,100, but can be suboptimal.
- **Interval Scheduling:** Given jobs, find max subset of non-overlapping jobs. Use Earliest Finish Time template (sorting required -> $O(nlogn)$)
- **Interval Partitioning:** Given lectures, find min number of rooms s.t. no 2 lectures are at the same time in the same rooms. Use Earliest Start Time template, allocate the room if no conflict, otherwise add a new classroom (sorting -> $O(nlogn)$)

HUFFMAN ENCODING / HUFFMAN TREE

Create trees by combining lowest frequencies first (usually left 0 & right 1), then create a table with the codes. Use the codes and frequencies to determine the avrg code length. ($\sum_i len(c_i) p_i$)

CACHING

FIFO: Add cache items in order of first in and first out
LIFO: Remove the last item in the cache when new items are added
LRU: Remove the cache item that was least recently used / added offline
LFD / FIF: Remove the cache item that will be used furthest in the future the optimal offline page replacement algorithm, OPT = LFD
FWF: Remove ALL cache items when the cache is full and a new item is being added.
Randomized Marking: whenever you add or request an item, it will be marked. When we want to add a new item, we randomly remove an unmarked item. If all items are marked and we are adding a new item, we unmark all of the cache items and randomly remove one.

PROOFS OF OPTIMALITY

- **Greedy Stays Ahead**: After each step, greedy solution is at least as good as another algorithm (an optimal algorithm).
- **Structural**: Find a structural bound that a solution must get, then show that greedy gets this bound.
- **Exchange Argument**: Transform another solution one step at a time to the greedy solution without hurting its quality.
- **Contradiction**: Assume greedy is NOT optimal, then find a contradiction using the optimal solution/algorithm.

Dynamic Programming

UNDERSTANDING RECUR & HOW TO PRODUCE THEM

Similar to divide and conquer, break any given $OPT(j)$ into an equation made up of smaller subproblems, using strictly smaller values for $OPT()$. Use multiple cases to represent base cases.

Weighted Interval Scheduling

Find max weight of jobs that are not overlapping. Cannot use earliest finish time since weights matter, so we need to take into account the weights.
Define $p(j)$ = largest index $i < j$ such that job i compatible with j.
Binary Choice: Job n is in the optimal solution or NOT in optimal solution
Define $OPT(j)$ = sum of the weights of all jobs that are optimal up to job j

Recurrence Relation & Algorithm

$$OPT(j) = \begin{cases} v_j + OPT(p(j)) & j \in OPT \\ OPT(j-1) & j \notin OPT \end{cases}$$

```
Weighted-Interval-Scheduling(jobs) -- BU
    Sort jobs by earliest finish time
    Compute p(1), ..., p(n)
    M[0] = 0
    for j = 1 to n
        M[j] = max(v[j] + M[p(j)], M[j-1])
    return M[n]
Find-Solution(j)
    if j = 0: return {}
    else if (v[j] + M[p(j)] > M[j-1])
        return {j} U Find-Solution(p[j])
    else:
        return Find-Solution(j-1)
```

Finding optimal cost takes $O(nlogn)$ due to initial sorting. To find the set of jobs, we do a second pass, taking $O(n)$.

KNAPSACK PROBLEM

Given n objects with weights and values. We want to fill a knapsack of max weight W s.t. it has the max value. Define $OPT(i, w)$ = max profit for items 1, .., i with weight limit w.

$$OPT(i, w) = \begin{cases} 0 & i = 0 \\ OPT(i-1, w) & w_i > w \\ max(OPT(i-1, w), \\ v_i + OPT(i-1, w - w_i) & \text{o.w.} \end{cases}$$

```
Knapsack(items, W)
    for w = 0 to W: M[0,w] = 0
    for i = 1 to n:
        for w = 1 to W:
            if (w[i] > w):
                M[i,w] = M[i-1,w]
            else:
                M[i,w] = max(m[i-1,w], v[i] + M[i
                    ↪ -1, w-w[i]])
    return M[n,W]
```

COIN CHANGE PROBLEM

Given an array of coin values,
$V = \{C_1, C_2, ..., C_m\}$ Cases are coin is not taken $solution[i-1][j]$, or is taken $solution[i][j - v[i]]$.

$$solution[i][j] = solution[i-1][j] + solution[i][j - v[i]]$$

SEGMENTED LEAST SQUARES

Example of a multiway choice DP
Find a set of $f(x)$ that fits the points the best with not too many lines.
Define $OPT(j)$ = min cost for $p_1, ..., p_j$
Define $e(i, j)$ = min sum of squares for $p_i, ..., p_j$

$$OPT(j) = \begin{cases} 0 & j = 0 \\ min_{1 \leq i \leq j}(e(i,j) + c + M[i-1]) & \text{o.w.} \end{cases}$$

```
Segmented-Least-Squares(jobs)
    for j = i to n
        for i = 1 to j
            Compute e(i,j)
    M[0] = 0
    for j = 1 to n
        M[j] = min(e(i,j) + c + M[i-1]) for all i,
            ↪ j >= 1
    return M[n]
Find-Segments(j)
    if j = 0: return {}
    else:
        Find i,j for min(e(i,j) + c + M[i-1])
        return the segment and the result of Find-
            ↪ Segments(i-1)
```

RNA SECONDARY STRUCTURE

Given RNA molecule $B = b_1 ... b_n$, find max base pairs of secondary structures.
Secondary Structure Criteria:

- Watson-Crick: A-U, U-A, C-G, G-C
- No sharp turn: Separated by at least 4 bases $(b_i, b_j) \in S \rightarrow i < j - 4$
- Non-crossing: $(b_i, b_j), (b_k, b_j l) \in S$ means $i < k < j < l$ not allowed.

Define $OPT(i, j)$ =max number of base pairs in substring $b_i ... b_j$

$$OPT(i, j) = \begin{cases} 0 & i \geq j - 4 \\ OPT(i, j-1) & b_j \notin S \\ 1 + max_t(OPT(i, t-1) \\ +OPT(t+1, j-1)) & (b_t, b_j) \in S \end{cases}$$

```
RNA(molecule B)
    for k = 5 to n-1
        for i = 1 to n-k
            j = i + k
            M[i,j] = max( M[i,j-1], 1+max_t(M[i,t
                ↪ -1] + M[t+1,j-1]) )
    return M[1,n]
```

Note that we take max t (aka. max value computed from using all possible t where $i \leq t < j - 4$) such that there are no sharp turns and (b_t, b_j) are Watson-Crick complements
RNA Secondary Structure is an example of dynamic programming over an interval, time complexity $O(n^3)$ and space complexity $O(n^2)$

TOP-DOWN VS BOTTOM-UP

top down: calculate all the needed values.
bottom-up: create a table, might not need all the table values.

Divide and Conquer

CLOSEST PAIRS PROBLEM

Given n points, find a pair of points with smallest euclidian distance. Brute force takes $\theta(n^2)$ calculations.

Divide & Conquer: $O(nlog^2 n)$, reduced with merging pre-sorted list to $O(nlogn)$

```
Closest-Pair(List of Pairs)
    Find line L such that it separates the points
        ↪ into exactly 2 halves.
    d1 = Closest-Pair(points left of L)
    d2 = Closest-Pair(points right of L)
    d = min(d1,d2)
    Delete all points further than d from L
    Sort/Merge remaining points by y-coord
    Compare if any of these remaining points is
        ↪ less than d
    return d
```

KARATSUBA TRICK

$m = \lceil n/2 \rceil$ – Divide into 2 subproblems
B = number base, usually base 10 or base 2.
a, b – first half & second half of number x
c, d – first half & second half of number y
$xy =$
$B^{2m}(ac) + B^m((ac + bd) - ((a - b)(c - d))) + bd$
Only needs 3 recursive calls, some additions and shifts. $T(n) = 3T(n/2) + \theta(n) \rightarrow \theta(n^{log_2 3})$

MERGE SORT

Divide list into 2 until there is only 1 item left, so sorted. Merge the two sorted lists. Runs in $O(logn)$

```
Mergesort(list)
    if |list| == 1: return list
    l1 = Mergesort(list[0:half])
    l2 = Mergesort(list[half:end])
    mergedlist = []
    Compare values of each item in l1 and l2 and
        ↪ add one item to merged list per
        ↪ iteration (depends if increasing or
        ↪  decreasing). If one is empty, then
        ↪  just add the non-empty list to
        ↪ mergedlist.
    return mergedlist
```

Network Flow

FORD-FULKERSON

High-level overview

1. Given a residual graph, "push" the maximum amount of flow possible through one path
2. Update residual graph with successfully pushed flows subtracted from positive and added to negative direction
3. Push more flow through paths with remaining positive flow in the direction needed

```
Ford-Fulkerson(G,s,t)
    foreach edge e: flow(e) = 0
        G_flow = residual graph
        while there is an augmenting path P in
            ↪ G_flow:
            flow = augment(flow,P)
            update G_flow
        return flow
Augment(flow, P)
    b = bottleneck capacity of path P
    foreach edge e in P:
        if e is a "real" edge: flow(e) += b
        else e is residual edge: flow(e) -= b
    return flow
```

The Ford-Fulkerson algorithm runs in $O(|E| val(f*))$, where $val(f*)$ is the value of the maximum flow

Edmonds-Karp Algorithm

We want to choose paths with fewest number of edges. Thus, we can use breadth first search in the residual graph to find the shortest path from s to t

```
Edmonds-Karp(G,s,t)
    foreach edge e: flow(e) = 0
        G_flow = residual graph
        while there is P in G_flow:
            P = BFS(G_flow,s,t)
            flow = augment(flow,P)
            update G_flow
        return flow
```

Runs in $O(m^2 n)$ due to good path choice

MAX-FLOW / MIN-CUT

- **Min Cut**: Find a cut (partition) of the vertices such that the sum of the capacities of the edges is minimal.
- **Max Flow**: Find the flow with a maximum value for the entire graph. Each flow must not exceed each edge's capacity and the flow going into a vertex must be equal to the flow out.
- **Theorem**: Max Flow value = Min Cut capacity
- **Lemma**: Let f be any flow and let (A, B) be any cut. Then, the net flow across (A, B) equals the value of f: $\sum_{e out of A} f(e) - \sum_{e into A} f(e) = v(f)$
- **Used in Ford-Fulkerson**: In the Ford-Fulkerson algorithm, by reaching the max flow value, we are ensuring that: Let f be any flow and (A, B) be any cut. Then, $v(f) = cap(A, B)$

BIPARTITE MATCHING

- Given bipartite graph w/ nodes that can be partitioned to L and R & edges that has one end in L and another in R, find the max cardinality matching.
- Create digraph $G' = (L \cup R \cup \{s, t\}, E')$
- Make all edges from L to R infinity.
- Add source s to all nodes in L w/ capacity of 1
- Add sink t from all nodes in R w/ capacity of 1
- Running max flow algorithm will find the max number of matching

PERFECT MATCHING

- Given bipartite graph, a perfect matching happens when each node appears in exactly one edge in $M \subseteq E$
- **Hall's Theorem**: Bipartite graph with $|L| = |R|$ has perfect matching iff $|N(S)| \geq |S|$ for all $S \subseteq L$.
- Note that $N(S)$ is the vertex in R that is connected to S by an edge in M

CIRCULATIONS WITH DEMANDS

Multiple sources and multiple sinks, each sink wants to get a certain amount of flow, and each source has a certain amount of flow to give. Reduction into max flow is adding a "root" node for source and one for sink, with capacity of edges as the values.

AIRLINE SCHEDULING

- Produce efficient schedule for airline operation activities.
- Given a set of flight k, where each flight i leaves origin o_i at time s_i and arrives at dest d_i and time f_i.
- Goal: Minimize flight crews
- For each flight i, add node u_i, v_i and edge (u_i, v_i) with lower bound & capacity 1.
- Add source s w/ demand $-c$ & edges to u_i w/ capacity 1.
- Add sink t w/ demand c & edges from v_i w/ capacity 1.
- If flight j doesn't conflict (time & location) with i, add edge (v_i, u_j) w/ capacity 1

BASEBALL ELIMINATION

First, calculate the maximum possible games the target team we are querying can win, say, m. Next, construct a flow graph, with the following sets of nodes: s as the source, $u_{xy}y$, for matches that need to be played among pairs of other teams, v_x, for teams that are not the target, and t for the sink. We build edges between s and u_{xy}, with edge weights of the number of remaining games for each pair, edges between u_{xy} to teams v_x and v_y, since only one team can win each match, and edges between each team x and sink with capacity $m - w_x$. If there is a max flow equal to $g*$, the total number of games left between all pairs of teams excluding the target, then it is possible for the target team to win or tie. Else, it is not possible.

PROJECT SELECTION

Model a set of P projects, each with revenue p_i, as a DAG representing dependencies between projects. (Edge (i, j) indicates i can only be selected if j is as well).
We reduce it to minimum-cut on a new graph G'. To construct G', add root source and root sink. For each node with $p_i > 0$, add edge (i, t) with capacity p_i, and for each node with $p_i < 0$, add edge (i, t) with capacity $-p_i$.
Add precedence constraints, give each edge in G an infinite capacity in G'.
Compute min cut (A', B') in G' and declare $A' - \{s\}$ to be optimal set of projects.

NP Problems

SHOW A PROBLEM IS NP

NP (verifier definition): Problems that are verifiable in polynomial time ($O(n^k)$).
Therefore, to show a problem is in NP, create a verifier for the problem that runs in polynomial, and show that it correctly verifies the result.
NP (nondeterministic algorithm definition): Problems that are solvable in polynomial time by nondeterministic algorithms.
The two definitions are interchangable.

Show a problem is as hard as another NP-Complete problem

Definition: Problem is NPC if problem is in class NP, and as "hard" as any problem in NP.
Formally, if X is NP-complete, $X \in \mathbf{NP} \wedge X \leq_P Y$, then Y is NP-complete.

Polynomial-time Reductions

If we have a procedure that transforms any instance of X into an instance of Y, such that the process takes polynomial time, and the answers to the problem are the same, then we have reduced X to Y. $X \leq_P Y$ means X is reduced to Y.
Use **polynomial time reductions** in the opposite way: If $X \leq_P Y$, and X is not polynomial-time, then Y is not polynomial-time.

Independent Set/ Vertex Cover / Set Cover

- **Independent Set**: Given a graph G = (V, E) and an integer k, is there a subset of vertices $S \subseteq V$ such that $|S| \geq k$, and for each edge at most one of its endpoints is in S?
- **Vertex Set**: Given a graph G = (V, E) and an integer k, is there a subset of vertices $S \subseteq V$ such that $|S| \leq k$, and for each edge, at least one of its endpoints is in S?
- **Set Cover**: Given a set U of elements, a collection $S_1, S_2, ..., S_m$ of subsets of U, and an integer k, does there exist a collection of $\leq k$ of those sets whose union is equal to U?

NPC Algo.s & Reductions from Class:
Packing/Covering: 3-SAT $\leq_p$ Independent Set $\leq_p$ Vertex Cover $\equiv_p$ Set Cover
Sequencing: 3-SAT $\leq_p$ Directed Hamiltonian Cycle $\leq_p$ Hamiltonian Cycle $\leq_p$ Longest Path $\leq_p$ Travelling Salesperson
Partitioning: 3-SAT $\leq_p$ Colorability Problem $\leq_p$ Register Allocation Problem
Numerical: 3-SAT $\leq_p$ Subset Sum Problem $\leq_p$ Partition Problem $\leq_p$ Interval Scheduling with Release Time Problem
3-SAT $\leq_p$ Independent Set: G contains 3 nodes for each clause, one for each literal. Connect 3 literals in a clause in a triangle, Connect literal to each of its negations.
G contains Independent Set of size $k = |\Phi|$ iff Φ is satisfiable.
Proof $\Rightarrow$: Let S be independent set of size k. S must contain exactly one node in each triangle. Set these literals to true (and remaining variables consistently). Truth assignment is consistent and all clauses are satisfied.
Proof $\Leftarrow$: Given satisfying assignment, select one true literal from each triangle. This is an independent set of size k.
Complexity of Reduction: Constructing k triangles is $O(k)$. Connecting literals to their negations is also $O(k)$. Hence reduction is polynomial time.
Independent Set Problem $\leq_p$ Vertex Cover Problem: We run Vertex-Cover($G, n-k$), and we get $V - S$ a vertex cover of size $n-k$. S is of size k. Consider two nodes $u \in S$ and $v \in S$. Observe that $(u, v) \notin E$ since $V - S$ is a vertex cover. Thus, no two notes in S are joined by an edge, which implies S is an independent set.
Vertex Cover Problem $\leq_p$ Independent Set Problem: Let S be an independent set of size k. $V - S$ is of size $n-k$. Consider an arbitrary edge (u, v). S being independent implies either, $u \notin S$ or $v \notin s$ (or both), $u \in V - S$ or $v \in V - S$ (or both). Thus, $V - S$ covers (u, v).
Vertex Cover Problem $\leq_p$ Set Cover Problem: Universe $U = E$. Include one set of each node $v \in V$: $S_v = \{e \in E : e \text{ incident to } v\}$. $G = (V, E)$ contains a vertex cover of size k iff (U, S) contains a set cover problem of size k.
Directed Hamiltonian Cycle $\leq_p$ Hamiltonian Cycle: Given a Digraph $G = (V, E)$, construct an undirected graph G' with $3n$ nodes. Where for each node $i = 0 \ldots n$ we create i_{in} which connects to i all nodes pointing to i, i which connected i to i_{in} and i_{out}, i_{out} which connects to i and all nodes pointing out of i.

Hamilton Cycle Problem $\leq_p$ Travelling Salesman Problem Given an instance $G = (V, E)$ of Hamiltonian Cycle Problem, create n cities with distance function
$d(u, v) = \begin{cases} 1 & \text{if}(u, v) \in E \\ 2 & \text{if}(u, v) \notin E \end{cases}$ If there is a Hamiltonian path, we can travel to all the cities within length (of exactly) n! Otherwise, we cannot do so without travelling a adding a +2 to our trip, ensuring our tour length would be over n. TSP instance has tour of length $\leq n$ iff G has Hamiltonian Cycle.

Colorability Problem $\leq_p$ Register Allocation Problem Given a Register Allocation Problem we can create an interference graph where nodes are program variables, edge between u and v if there exists an operation where both u and v are "live" at the same time. Observe that we can solve the Register Allocation Problem problem iff the interference graph is k-Colorable for any constant $k \geq 3$.

Vertex-Cover $\leq_p$ Hitting-Set Construct $G = (V, E)$ in the following way. Let $B_1, B_2, \ldots, B_m$ be sets of size 2 such that $\{u, v\} = B_i$ iff $(u, v) \in E$ for $i = 1, \ldots, m$ where $m = |E|$. Let $V = A$. Then, the hitting set $H \subseteq A = V$ will be a subset of vertices $|H| \leq k$ where for each edge $(u, v) \Rightarrow \{u, v\} = B_i$ for some i, at least one of its endpoints is in H because $H \cap B_i = H \cap \{u, v\} \neq \emptyset$. As such H is a valid solution to the vertex cover problem.

3-SAT

SAT: Given a CNF formula ϕ, does it have a satisfying truth assignment?
3-SAT: SAT where each clause contains exactly 3 literals (and each literal corresponds to a different variable).
3-SAT $\leq_P$ Set Cover:

Subset-Sum

- Given natural numbers and integer W, is there a subset that adds up to W
- **Theorem** 3-SAT $\leq_P$ Subset-Sum
- Thus, it is also NP-Complete

Hamilton Cycle

- Given undirected graph, is there a cycle that passes through all vertices

Hamilton Path

- Given graph, is there a path from s to t that passes through all vertices

Directed Hamilton Cycle

- Given digraph, is there a directed cycle that passes through all vertices
- **Theorem** Dir-Ham-Cyc $\leq_P$ Ham-Cyc

Since we can create an instance of Dir-Ham-Cyc from an instance of 3-SAT s.t. there is a hamiltonian cycle iff ϕ is satisfiable, we know Dir-Ham-Cyc is NP-complete.
Thus, 3-SAT $\leq_P$ Dir-Ham-Cyc $\leq_P$ Ham-Cyc $\leq_P$ Ham-Path
So all above are NP-complete

Finding Small Vertex Covers

- $O(2^k kn)$ time algorithm.

```
Vertex-Cover(G,k)
    if (G contains no edge) return true
    if (G contains >= kn edges) return
        ↪ false
    let (u,v) be any edge of G
    a = Vertex-Cover(G - {u}, k-1)
    b = Vertex-Cover(G - {v}, k-1)
    return a or b
```

- Each invocation takes $O(kn)$ time. A vertex cover of size k has at most $k(n-1)$ edges since each vertex covers at most $n-1$ edges.

```
Independent-Set-In-A-Forest(F){
    S <- emptyset
    while (F has at least one edge){
        Let e = (u, v) be an edge in v such that v
            ↪ is a laf
        Add v to S
        Delete from F nodes u and v, and all edges
            ↪ incident to them
    }
    return S
}
```

Vertex Cover in Bipartite Graph

The max cardinality of a matching is equal to the min cardinality of a vertex cover.

Travelling Salesman

- Given a set of n cities and distance $d(u, v)$ between cities, is there a tour of length $\leq D$
- **Theorem** Ham-Cyc $\leq_P$ TSP
- Construct an instance of TSP from an instance of Ham-Cyc s.t. the distance is 1 if $(u, v) \in E$ or 2 if $(u, v) \notin E$
- Thus, is NP-Complete

3-Colorability

- Given undirected graph, can the vertices be coloured red, blue, green and no two adjacent vertices have the same colour
- **Theorem** 3-SAT $\leq_P$ 3-Color
- Construct a graph instance of 3-color from the instance of 3-SAT s.t. all literals are vertices.
- Add 3 vertices, Base, True, False & connect to each other.
- Connect negation of each literal with each other and connect each literal to the base.
- The graph is 3 colorable iff it is 3 satisfiable, thus it is also NP-Complete

Backtracking

RAT IN THE MAZE ALGORITHM

WhereIsDoor:

1. Look North, if there is unused door, use it, otherwise goto 2.
2. Look East, if there is unused door, use it, otherwise goto 3.
3. Look South, if there is unused door, use it, otherwise goto 4.
4. Look West, if there is unused door, use it, otherwise goto 5.
5. Unused door does not exist, go back through the door you entered.

IHaveBeenThere: Mark the room you have entered.
IHaveUsedThisDoor: Mark the door you have used
DoorBetweenRooms:
MyImportantPath: sequence: room1, door1, room2, ..., roomk, doork, roomk+1

And rat has been in each room exactly once except the room k+1, where it might be for the second time, and each door was used exactly once, except door k.

WasIThere?: Returns YES if the room is entered for the second time.
RatAlgorithm:

1. WhereIsDoor;
2. If WasIThere? = YES, go back through the door you entered and modify MyImportantPath by popping stack twice, and goto 1.
3. Otherwise, modify IHaveBeenThere, IHaveUsedThisDoor, add door used and room current to MyImportantPath, and goto 1.

Has time complexity O(size of maze) and space complexity O(size of maze) too.

Approximation Algorithms

LOAD BALANCING

- **Greedy List Scheduling Algorithm** considers n jobs in a fixed order and we assign job j to the machine that has the smallest load so far.
- A load on a machine is the sum of the subset of jobs assigned to a specific machine. The makespan is the max load on any machine.
- $O(n \log m)$ where n is the amount of jobs and m is the amount of machines. This is a 2-approximation.
- **Greedy with Longest Processing Time Algorithm** sorts n jobs in descending order of processing time, and then run list scheduling algorithm (as above). This is a 4/3-approximation.
- Complexity is $O(n \log n)$ due to sorting.

DOMINATING SET

Given a graph, a dominating set contains a set of nodes where every node in graph is neighbour of a node in dominating set.
E.g. given n transmitters, each with d edges, we show for some constant c, a set of $\frac{cn \log n}{d+1}$ random nodes is very likely to be a dominating set. ($c \log n$ times larger than optimal solution, but quickly solvable)

There is a $\frac{d+1}{n}$ chance of picking a node t that covers v, therefore, the probability that every node picked we fail to dominate v is $\prod_{t=1}^{k} Pr[fail(v, t)] = (1 - \frac{d+1}{n})^k$. For our case of $k = \frac{cn \log n}{d+1}$, $Pr[fail(v)] \leq 1/n^c$.

Randomization

Rabin-Miller Algorithm

Probablistic primality test

```
Miller-Rabin(n, k)
if n == 2
    return TRUE
if Is-Even(n)
    return FALSE
a = Random-Positive-Int()
if a^(n-1) ≢ 1 mod n
    return FALSE
else
    Find s, h such that s is odd
    and n - 1 = s2^h
    Compute sequence a^(s·2^0),
    a^(s·2^1), a^(s·2^2),
    . . . , a^(s·2^h) mod n
    if all elements in sequence are 1
        return TRUE
    elseif the last element different from 1
    is -1
        return TRUE
    else
        return FALSE
```

Randomized Divide and Conquer: Finding the Median

```
Select(S, K)
Choose a splitter a_i ∈ S
for each element a_j ∈ S
    Put a_j in S^- if a_j < a_i
    Put a_j in S^+ if a_j > a_i
if |S^-| = k - 1
    return a_i
elseif |S^-| ≥ k
    // kth largest element is in S^-
    return Select(S^-, k)
else |S^-| = ℓ < k - 1
    // kth largest element is in S^+
    return Select(S^+, k - 1 - ℓ)
```

LOCAL SEARCH

- Find a local optimum rather than the global optimum.
- Every iteration should make a choice that is going to improve optimality.
- If no more improvement can be made, then the local optimum is found
- Sequentially move from a current solution to a neighbour solution that has better cost (gradient descent)

Local Search Vertex Cover

- Start with $S = V$, if there is a neighbor S' that is also a vertex cover with $|S'| < |S|$, replace S with S'
- This is done by deleting/adding nodes.
- Terminates after at most n steps

HOPFIELD NEURAL NETWORKS

State-flipping algo: Repeatedly flip the state of any unsatisfied node. Terminates with stable config after at most $W = \sum_e |w_e|$ iterations.

State: $s_{node} = \pm 1$. Stable: all nodes are satisfied. Satisfied node: weight of incident good edges $\leq$ weight of incident bad edges, $\sum_{v:e=(u,v)\in E} w_e s_u s_v \leq 0$. Good edge: for edge $e = (u, v)$, $w_e \times s_u \times s_v < 0$. Note: Decision problem is always yes, no poly-time algo for search problem.

Exact Exponential Algorithms

EXACT 3-SAT ALGORITHM

```
3-Sat(p):
    If p is empty return true.
    (l1 or l2 or l3) and p' <- p.
    If 3-Sat(p' | l1 = true) return true.
    If 3-Sat(p' | l2 = true) return true.
    If 3-Sat(p' | l3 = true) return true.
    return false.
```

Takes $O(poly(n)3^n)$ time.

EXACT HAMILTONIAN CYCLE ALGORITHM

Dynamic Programming Solution: Let $c(s, v, X)$ be cost of cheapest path between s and v that visits every node in X exactly once.
$OPT = min_{v \neq s} c(s, v, V) + c(v, s)$ Therefore

$$c(s, v, X) =$$

$$\begin{cases} c(s, v) & \text{if } |X| = 2 \\ \min_{u \in X \setminus \{s,v\}} c(s, u, X \setminus \{v\}) \\ \quad + c(u, v) & \text{if } |X| > 2 \end{cases}$$

Other

ROD CUTTING PROBLEM

Given a rod of length n, with varying prices per length of rod, maximize the total amount of money gained.
We can construct the recurrence relation $r_n = \max_{1 \leq i \leq n}(p_i + r_{n-i}$ We compute for smallest to largest rod lengths, final result stored in $r[n]$.

SEARCHING IN SORTED ARRAYS

Binary Search: $O(\log n)$ time complexity, $O(\log n)$ space complexity if using recursion, $O(1)$ otherwise.

QUICKSORT

Average case complexity $O(n \log(n))$ because of random pivot. Worst case $O(n^2)$.

```
Quicksort(S)
if |S| ≤ 3
    return Insertion-Sort(S)
else
    pivot = Random-Element(S)
    for x ∈ S
        if x < pivot
            Append(S-, x)
        if x > pivot
            Append(S+, x)
    S- = Quicksort(S-)
    S+ = Quicksort(S+)
    return Concat(S-, [x], S+)
```

```
Partition(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        Swap(A[i], A[j])
Swap(A[i + 1], A[r])
return i + 1
```

Summation Rules

$\sum_{i=m}^{n} a_i = \sum_{i=m+k}^{n+k} a_{i-k}$.
$\sum_{i=m}^{n} a_i = \sum_{i=m-k}^{n-k} a_{i+k}$
Quadratic: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$,
$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$,
$\sum_{i=1}^{n} i^3 = (\frac{n(n+1)}{2})^2$
Geometric: For
$|r| < 1, \sum_{i=1}^{n} ar^{i-1} = \frac{a(1-r^n)}{1-r}$.
$|r| > 1, \sum_{i=1}^{n} ar^{i-1} = \frac{a(r^n - 1)}{r-1}$

Stats

- $A \subset S$, $P(A) = \sum_{x \in A} P(x)$
- $P(\emptyset) = 0$, $P(S) = 1$
- $A \cap B = \emptyset \implies P(A \cup B) = P(A) + P(B)$
- $P(A \cup B) = P(A) + P(B) - p(A \cap B)$
- Mutually exclusive $P(A \cap B) = \emptyset$
- Independent $P(A \cap B) = P(A)P(B)$
- Conditional $P(A|B) = \frac{P(A \cap B)}{P(B)}$
- Random Variable X = usually the frequency of occurrence of something
- Expected Value $E(X) = \sum x_i P(x_i)$
- Linearity of Expected Value $E(\sum X_i) = \sum E(X_i)$, $E(X+Y) = E(X) + E(Y)$, $E(cX) = cE(X)$