1 Dynamic Programming

- Typically built on the **divide-and-conquer** paradigm where we divide the input instance into 2 or more parts (of equal sizes), solve the same problem for each smaller instance recursively and combine the solutions to get solution of original instance
- Fibonacci is commonly used to demonstrate DP as the recursive algorithm takes $2^{(n-2)/2} \approx \phi^n$ time to run if using unoptimized recursion and takes only $5n$ time to run using iterative algorithm
- DP uses memoization technique to prune away recursion tree to reduce the number of repeated calculation

1.1 Main Concepts

- **cut-and-paste proof** $\rightarrow$ proof by contradiction - suppose you have an optimal solution. Replacing ("cut") subproblem solutions with this subproblem solution ("paste") in) should improve the solution. If the solution doesn't improve, then it's not optimal (contradiction)
- **optimal substructure** - optimal substructure to a problem (instance) contains optimal solutions to subproblems
- **Overlapping subproblems**: recursive solution contains a "small" number of distinct subproblems repeated many times
- **Tips**: look at the prefix of the problem!! DP usually generates solutions based on what is already calculated. It generally also involves some sort of decision you have to make (e.g. either you take or you don't take)

1.2 Longest Common Subsequence

- Definition of Subsequence: for a sequence $A : a_1, a_2, ..., a_n$ stored in an array, C is a subsequence of A if we can obtain C by removing 0 or more elements from A
- **Problem**: given two sequences $A[1..n]$ and $B[1..m]$, compute the longest sequence C such that C is a subsequence of A and B

1.2.1 Brute Force Approach

- Check all possible subsequence of A to see if it is also a subsequence of B, then output longest one
- Runtime will be $O(m2^n)$, where m is the time taken to check whether a subsequence of A is a subsequence of B and there are 2^n possible subsequence since each char in A we can choose to either take or we don't

1.2.2 Recursive Formulation

- Let $LCS(i, j)$ be the LCS of $A[1..i]$ and $B[1..j]$
- **Base case**: $LCS(i, 0) = \emptyset, \forall i$, $LCS(0, j) = \emptyset, \forall j$
- **General case** (optimal substructure):
  - if $a_n = b_m \rightarrow LCS(n, m) = LCS(n-1, m-1) :: a_n$, where $:: a_n$ mean that we concatenate a_n to the subsequence in $LCS(n-1, m-1)$
  - **Cut-and-paste argument**: Suppose S returned by $LCS(n-1, m-1)$ is optimal solution for $A[1..n-1], B[1..m-1]$, we could just append a_n to S to get a subsequence with a length increased by 1, thus S could not have been the largest subsequence
  - if $a_n \neq b_m \rightarrow LCS(n, m) = max(LCS(n-1, m), LCS(n, m-1))$
- **Simplified Problem**:
  - $L(n, m)$: Length of LCS of $A[1..n]$ and $B[1..m]$
  - $L(n, m) = 0$ if n or $m = 0$
  - if $a_n = b_m \rightarrow L(n, m) = L(n-1, m-1) + 1$
  - if $a_n \neq b_m \rightarrow L(n, m) = max(L(n, m-1), L(n-1, m))$
- **Analysis**:
  - $T(n, m) = T(n-1, m) + T(n, m-1) + \Theta(1) \rightarrow 2^n$
  - However, there is effectively only $(n+1) \times (m+1)$ subproblems
  - Could be solved in $O(nm)$ using bottom up approach with extra space of $O(min(n, m))$ to store intermediate results

1.3 Knapsack Problem

- **Input**: $(w_1, v_1), (w_2, v_2), ..., (w_n, v_n)$ and capacity W
- **Output**: subset $S \subseteq \{1, 2, ..., n\}$ that maximises $\sum_{i \in S} v_i, s.t. \sum_{i \in S} w_i \leq W$
- **Naive solution**: runs in 2^n time since there are 2^n possible combinations

1.3.1 Recursive Solution

- Let $m(i, j)$ be the maximum value that can be obtained using the first $\{1, 2, ..., i\}$ items with total weight $\leq j$
- $$m(i,j) = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0 \\ max(m(i-1, j-w_i) + v_i, m(i-1, j)), & \text{if } w_i \leq j \\ m(i-1, j), & \text{otherwise} \end{cases}$$
- **Analysis**: $O(nW)$, which is a pseudo-polynomial algorithm as W can be represented in $O(\lg W)$ bits and n can be represented in $O(\lg n)$ bits.
- Also a suboptimal solution as it depends on W

1.4 Changing Coins

- **Problem**: use the fewest number of coins to make up n cents using denominations $d_1, d_2, ..., d_n$. Let $M[j]$ be the fewest number of coins needed to change j cents.
- **Optimal Substructure**: $$M[j] = \begin{cases} 1 + \min_{i \in [k]} M[j - d_i] & j > 0 \\ 0, & j = 0 \\ \infty, & j < 0 \end{cases}$$
- **Proof**: Suppose $M[j] = t$, meaning $j = d_{i_1} + d_{i_2} + ... + d_{i_t}$ for some $i_1, ..., i_t \in \{1, ..., k\}$. Then if $j' = d_{i_1} + d_{i_2} + ... + d_{i_{t-1}}$, $M[j'] = t - 1$ because otherwise if $M[j'] < t - 1$, by cut-and-paste argument, $M[j] < t$
- **Runtime**: $O(nk)$, for n cents and k denominations

2 Greedy Algorithms

- In DP, there are often choices to be made which leads to multiple subproblems (branching)
- In Greedy Algorithms, we **only solve 1 subproblem** at each step which could result in greedy algorithms beating divide-and-conquer and DP **when it works**
- To solve Greedy Algorithm problems, look out for the **Optimal Substructure** (cut-and-paste argument) and **Greedy-Choice property** $\rightarrow$ locally optimal solution is globally optimal

2.1 Fractional Knapsack

- **Input**: $(w_1, v_1), (w_2, v_2), ..., (w_n), v_n$ and capacity W
- **Output**: Weights $x_1, ..., x_n$ that maximizes $\sum_i x_i \cdot \frac{x_i}{w_i}$, subject to $\sum_i x_i \leq W$ and $0 \leq x_j \leq w_j, \forall j \in [n]$

2.1.1 Optimal Substructure property

- If we remove w kgs of one item j from the optimal knapsack, then the remaining load must be the optimal knapsack weighing at most $W - w$ kgs that one ca ntake from the $n - 1$ original items and $w_j - w$ kgs of item j
- **Cut-and-paste proof**: Suppose some $(y_1, ..., y_i, ..., y_n)$ weighed $\leq (W - w)$ and had better value than $(x_i, ..., x_i - w, ..., x_n)$ (optimal solution), then $(y_1, ..., y_i + w, ..., y_n)$ weighs $\leq$ W and has better value than $(x_1, ..., x_i, ..., x_n) \rightarrow$ Contradiction!

2.1.2 Greedy-choice Property

- Let j^* be the item with the maximum value/kg, v_j/w_j. Then there exists an optimal solution that contains $min(w_{j^*}, W)$ kgs of item j^*
- **"Exchange argument"**: Suppose an optimal knapsack contains $(x_1 + x_2 + ... + x_n) = min(w_{j^*}, W)$ of items $(1, ..., n)$, we could replace this with $min(w_{j^*}, W)$ of item j^* which does not change the weight but value will also not decrease since j^* has max weight/value ratio $\rightarrow$ knapsack stays optimal

2.1.3 Analysis

- Algorithm will be to sort items in terms of their weight/value ratio and keep choosing the item with highest weight/value ratio until we reach the target weight
- Sorting takes $O(n \lg n)$ so greedy fractional knapsack takes $O(n \lg n)$

2.2 Minimum Spanning Tree

2.2.1 Graph Review

- In any graph, $|E| = O(|V|^2)$
- Degree of vertex is the number of edges containing v
- **Handshaking Lemma**: In any graph, $\sum_{v \in V} \deg(v) = 2 \cdot |E| \rightarrow$ total number of degree = 2 $\cdot$ number of edges

2.2.2 Problem Definition

- **Input**: Connected graph $G = (V, E)$ and weight function w (weight is on edges)
- **Output**: spanning tree of G with minimum weight

2.2.3 Optimal Substructure

- Let T be a MST. Remove any edge $(u, v) \in T$, then T is partitioned into 2 subsets T_1 and T_2 which are both MSTs
- **Cut-and-paste argument**: If T_1' has a lower MST weight than T_1, then we could create a MST $T' = \{u, v\} \cup T_1' \cup T_2$ which has a lower weight than T for $G \rightarrow$ Contradiction!

2.2.4 Greedy Choice Property

- Let T be the MST of $G = (V, E)$, and let $A \subseteq V$. Supposed that $(u, v) \in E$ is the least weight edge connecting A to $V - A$, then $(u, v) \in T$

2.3 Prim's Algorithm

- Maintain $V - A$ as a priority queue Q. At each step, add the least-weight edge from the tree to some vertex outside the tree then relax

Time = $\Theta(V) \cdot T_{\text{EXTRACT-MIN}} + \Theta(E) \cdot T_{\text{DECREASE-KEY}}$

Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$
Fibonacci heap	$O(\lg V)$ amortized	$O(1)$ amortized	$O(E + V \lg V)$ worst case

2.4 Kruskals Algorithm

- Basic idea is to add least-weight edge that does not cause cycle to form.
- Running time of $O(E \lg v)$, works better for sparse trees where $|E| \approx |V|$

3 Incremental Algorithms

3.1 Flow Networks

- Directed graph $G = (V, E)$ in which each edge $(u, v) \in E$ has a capacity $c(u, v) \geq 0$, if $(u, v) \notin E, c(u, v) = 0$
- Assumes no self-loops and not anti-parallel edges
- Flow is a function f that assigns a number $f(u, v)$ to each edge - the flow from u to v
- Constraints:
  1. **Capacity**: each edge has a capacity constraint that states that flow cannot exceed capacity
  2. **Flow conservation**: For every vertex - Flow into vertex must equal flow out of vertex
- Value of flow, $|f|$ is the total flow from source to target

3.2 Max Flow Problem

- Given a flow network $G = (V, E)$ and capacities c, find a flow that maximizes $|f|$
- Solved using Ford-Fulkerson Algorithm (won't return non-integer values)
  1. Start with $f(u, v) = 0$ for all (u, v)
  2. While there is a path p from s to t in G_f:
     - (a) Let $m = min_{(u,v) \in p} c_f(u, v)$
     - (b) For each $(u, v) \in p$
       - i. If $(u, v) \in E$: increment $f(u, v)$ by m (increment if edge in G)
       - ii. If $(v, u \in E)$: decrement $f(v, u)$ by m (decrement if edge not in G)
- **Runtime**: $O(|E| \cdot |f_{max}|)$ if we use depth-first search, where f_{max} is the maximal flow. $O(|V||E|^2)$ if we use BFS (Edmonds-Karp Algorithm), $O(|E||V|^2)$ if we use Dinic's
- To prove correctness, need to show:
  - If f is a valid flow (satisfies capacity constraint and flow conservation), it remains a valid flow after the changes made in step 2(b)
  - If there is no path p from s to t in G_f, then f is maximal flow

Partition of V into *any* S and T such that $s \in S$ and $t \in T$ is called a *cut*

Capacity of a cut is the total capacity of all edges going from S to T

Any cut is a bottleneck for any flow

So for any flow f,
$|f| \leq$ capacity of minimum cut

When algorithm terminates,
S – vertices reachable from s in G_f
T – vertices that can reach t in G_f

Observe: For $u \in S, v \in T$:
- If $(u, v) \in E: f(u, v) = c(u, v)$
- If $(v, u) \in E: f(u, v) = 0$

$|f_{FF}|$ = total capacity of edges going from S to T

$|f_{FF}| \geq$ capacity of this cut

Put together,
$|f_{FF}|$ = capacity of min cut = value of max flow

3.2.1 Application: Bipartite Matching

- Problem is the find the largest number of edges a matching can have
- Problem could be modeled by connecting the bipartite graph with a source node that points to each of the "left" nodes and a sink node that all "right" nodes point to
- Any **integer** flow corresponds to some matching

4 Linear Programming

Linear programming is a mathematical modeling technique in which a **linear function (objective) is maximized or minimized** when subjected to various constraints. The optimal solution can be found in $poly(n, m)$ time and is quite efficient to solve in practice.

4.1 "Standard form" of LP

Objective is to be *maximised* (rather than minimised)
All variables are required to be non-negative
All constraints are less-than-or-equal-to

Maximise: $\sum_{j \in [n]} c_j \cdot x_j$

Subject to:

$x_1, ..., x_n \geq 0$

$a_{11}x_1 + \cdots + a_{1n}x_n \leq b_1$
$a_{21}x_1 + \cdots + a_{2n}x_n \leq b_2$
$\cdots$
$a_{m1}x_1 + \cdots + a_{mn}x_n \leq b_m$

Convenient form for discussing LP algorithms and properties

Any LP can be converted to this standard form (while effectively maintaining the optimal objective value)

4.1.1 Converting to Standard Form

~~Objective is asked to be minimised~~

P_1: Minimise: $(x_1 + 4x_2)$

x_2 is not required to be non-negative

$x_1 \geq 0$

$x_1 - 2x_2 \geq 10$

$3x_1 + 5x_2 = 20$

~~Constraints are not less-than-or-equal-to~~

P_3: Maximise: $(-x_1 - 4(x_2' - x_2''))$

$x_1, x_2', x_2'' \geq 0$ Replace x_2 with $(x_2' - x_2'')$

$-x_1 + 2(x_2' - x_2'') \leq -10$

$3x_1 + 5(x_2' - x_2'') \leq 20$
$-3x_1 - 5(x_2' - x_2'') \leq -20$

Any value x_2 can take, $(x_2' - x_2'')$ can also take, and vice versa

$OPT(P_3) = OPT(P_2) = -OPT(P_1)$

4.2 Example: Max Flow

- Variables defined as $f(u, v)$ for every $(u, v) \in E$
- Maximize: $\sum_{v \in Neigh(s)} f(s, v)$
- Subjected to:
  - $\forall (u, v) \in E : f(u, v) \geq 0$
  - $\forall (u, v) \in E : f(u, v) \leq c(u, v)$ (capacity constraint)
  - $\forall u \in V \backslash \{s, t\} : \sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$ (flow conservation)
- Can solve Max Flow problem using LP in $poly(|V|, |E|)$ which is not as fast as dedicated algorithms for Max Flow

4.3 Example: Min-Cost Flow

Directed graph $G = (V, E)$ in which each edge $(u, v) \in E$ has a *capacity* $c(u, v) \geq 0$ and *cost* $a(u, v)$

Sending flow $f(u, v)$ along edge (u, v) costs $a(u, v) \cdot f(u, v)$

Same capacity and flow conservation apply

Total cost of flow: $\sum_{(u,v) \in E} a(u, v) \cdot f(u, v)$

What is the minimum cost of a flow of size d from s to t?

Variables: $f(u, v)$ for every $u, v \in E$

Minimise: $\sum_{(u,v) \in E} a(u, v) \cdot f(u, v)$

Subject to:

$\forall (u, v) \in E: f(u, v) \geq 0$

$\forall (u, v) \in E: f(u, v) \leq c(u, v)$

$\forall u \in V \setminus \{s, t\}: \sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$

$\sum_{v \in Neigh(s)} f(s, v) = d$

4.4 Example: Multicommodity Flow

Directed graph $G = (V, E)$ in which each edge $(u, v) \in E$ has a *capacity* $c(u, v) \geq 0$

k different "commodities", each with its own source s_i, target t_i, desired flow d_i

Each commodity has its own flow $f_i(u, v)$

1. Capacity: $\forall (u, v) \in E: \sum_{i \in [k]} f_i(u, v) \leq c(u, v)$

2. Flow conservation: $\forall i \in [k]$: $\forall u \in V \setminus \{s_i, t_i\}$:
$\sum_{v \in V} f_i(v, u) = \sum_{v \in V} f_i(u, v)$

Does there exist a set of flows where for all $i \in [k]$:
$\sum_{v \in Neigh(s_i)} f_i(s_i, v) = d_i$

Variables: $f_i(u, v)$ for every $u, v \in E, i \in [k]$

Minimise: 0

Subject to:

$\forall i: \forall (u, v) \in E: f_i(u, v) \geq 0$

$\forall (u, v) \in E: \sum_i f_i(u, v) \leq c(u, v)$

$\forall i: \forall u \in V \setminus \{s_i, t_i\}: \sum_{v \in V} f_i(v, u) = \sum_{v \in V} f_i(u, v)$

$\forall i: \sum_{v \in Neigh(s_i)} f(s_i, v) = d_i$

* Note that we are optimizing" a constant function here as this is a decision problem

4.5 Simplex Algorithm

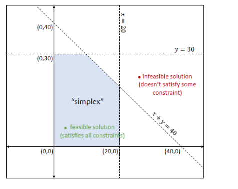
Linear programs can be represented as a graph

x – no. of units of A produced
y – no. of units of B produced

Maximise: $(x + 2y)$
$x, y \geq 0$
$x \leq 20$
$y \leq 30$
$x + y \leq 40$

All constraints are linear inequalities

Objective is a linear function



- If an optimal solution exist, it will always be at a vertex
- There could be situations where linear programs are only constraint in 1 axis $\rightarrow$ feasible solution exists but no optimum, LP is **unbounded**
- Simplex could be roughly described as:
  1. Start at a feasible vertex, and compute its objective value
  2. Find a neighbouring vertex with larger objective value
     - If none exists, output current vertex as optimum
     - Else, move to that vertex and repeat
- If optimal solution is **parallel with vertex**, any point on the line will be optimal solution
- If there are n variables, then simplex will be a n-dimensional convex polyhedron
- Correctness follows from convexity - any local optimum is a global optimum

4.6 Comparison of LP algorithms

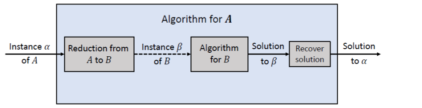
Algorithms	Comments
Simplex Algorithm	Efficient in practice Exponential running time in worst case
Ellipsoid Algorithm	First polynomial-time algorithm to solve LP Slow in practice
Interior Point method	Efficient in practice Polynomial running time in worst case Many recent improvements to running time

5 Reductions

Reductions between computational problem is a fundamental idea in algorithm design. It is useful in giving us a way to compare the **hardness** of 2 problems

5.1 Definition of Reduction

- Consider 2 problems A and B. If A can be solved as follows:
  - **Input**: Instance α of A
    1. Convert α to an instance β of B
    2. Solve β and obtain a solution
    3. Based on solution of β, obtain solution of α
  - Then, we say A reduces to $B \rightarrow$ Start with A, convert to and solve B to obtain solution for A



5.2 Example: Palindrome and LCS

- Problem A: Given string x, find its **Longest Palindromic Subsequence**
- Problem B: Given pair of strings (x, y), find the **Longest Common Subsequence**
- Idea: The LPS of a string is the LCS of the string and its reverse. i.e. LPS $\leq_p$ LCS

5.3 Example: T-Sum

- Problem A: Given Array A of length n, Output (i, j) such that $A[i] + A[j] = 0$
- Problem B: Given Array B of length n, and a number T, output (i, j) such that $B[i] + B[j] = T$
- Want to show T-Sum $\leq_p$ 0-sum
- Idea: Given array B, define array A such that $A[i] = B[i] - T/2$. If (i, j) satisfy $A[i] + A[j] = 0$ then $B[i] + B[j] = T$

5.4 Bounded-Time Reductions

Same definition as normal reduction just that the reduction should only take $p(n)$ time and we should be able to obtain the solution of α from β in $p(n)$ time as well

5.4.1 Encoding of Input

- In the Palindrome and LCS reduction, it is an $O(n)$ time reduction since a string x will take $O(n)$ bits to represent and reducing it to $(x, rev(x))$ could easily be done in $O(n)$ time where n is the length of string x
- In Matrix Squaring to Matrix Multiplication reduction, input for squaring is the $N \times N$ matrix and the input for the multiplication is also the same $N \times N$ matrix $\rightarrow O(n)$ reduction where $n = N \times N$
- In general n is the length of the encoding of the problem instance
- Integers can be encoded using binary $\rightarrow O(\lg n)$, objects, arrays, matrices are encoded using a list of parameters enclosed in brackets, separated by comma
- Note that in the matrices example, the technically correct bits used should be $O(N^2 \lg m)$ but we can assume that each element in the matrix is "not too large" i.e. ≤ 32 bits so can be treated as constant

5.4.2 Running Time Composition

- Claim that if there is a $p(n)$ time reduction from problem A to problem B and there is a $T(n)$-time algorithm to solve problem B on instances of size n, then there is a $$T(p(n)) + O(p(n))$$ time algorithm to solve problem A on instance of size n

5.4.3 Polynomial-Time Reduction

- $A \leq_p B$ if there is a $p(n)$-time reduction from A to B for some **polynomial function** $p(n) \rightarrow$ implies that B is at least as hard as A
- If B has a poly time algorithm, then B so does A
- If $A \leq_p B$, we can infer that if A cannot be solved in poly time then neither can B
- Poly-time algorithms refers to any algorithm that runs in $\leq O(n^5)$

5.5 Example: Factor

- Given 2 positive integers m and t, with $t < m$, output whether m is divisible by f such that $1 < f \leq t$
- Not a poly-time algo!!

- What is the size of the problem encoding?
  - $\Theta(\lg m)$ bits to encode m
  - $\Theta(\lg t)$ bits to encode t where $\lg t < \lg m$.
  - Total encoding length is $\ell = \Theta(\lg m)$.

```
for $f$ = 2 to $t$:
    if $m \% f == 0$:
        return true
return false
```

- Runtime is potentially $\Omega(m) = \Omega(c^\ell)$ for some constant c, exponential in ℓ. So, not polynomial time in encoding of input!!

5.6 Knapsack and Fractional Knapsack

Ans: (c) No for KNAPSACK, yes for FRACTIONAL KNAPSACK

Knapsack DP solution is a pseudo-polynomial algorithm but Greedy Fractional knapsack is a polynomial algorithm

The input for both problems is a list $(v_1, w_1), \ldots, (v_n, w_n), T$.

Encoding for T

The input size is $O(n \log M + \log T)$ where M is an upper-bound on the v_i's and w_i's.

List of N inputs each of size logM

Does nT number of arithmetic operations

Running time for KNAPSACK is $O(nT \log M)$. Running time for FRACTIONAL KNAPSACK is $O(n \log n \, (\log M)^2)$.

Sorting takes nlgn

Comparing the weight/value ratio. Multiplication assume to take (lgM)^2 time

5.7 Pseudo-Polynomial Time Algorithm

- **Definition**: An algorithm that runs in time polynomial in the ***numeric value of the input*** but is exponential in the ***length of the representation*** (length in bits) of the input is called a pseudo-polynomial time algorithm
- Both factor and knapsack are pseudo-poly time algorithm
- Show polynomial in terms of value of input but exponential in terms of length of input!!!

5.8 Decision Problems

- A decision problem is a function that maps each instance to either "YES" or "NO"
- Decision vs Optimization (aka search problems)
  - **Decision Problem**: Given a directed graph G, is there a path from vertex u to v of length $\leq k$?
  - **Optimization problem**: Given ..., what is the length of the shortest path from u to v?

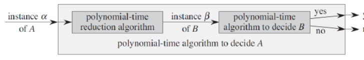
5.8.1 Reducing Decision to Optimization Problem

- **decision $\longrightarrow$ optimization**: given an instance of the optimization problem and a number k, is there a solution with value $\leq k$?
- Decision problem is no harder than the optimization problem $\longrightarrow$ given optimal solution, can easily check if $< k$
- If we cannot solve optimization problem problem quickly then we cannot solve decision problem quickly $\longrightarrow$ decision $\leq_p$ optimization
- We **could also reduce decision problem to Optimization problem** by either sequentially searching for the min/max value which decision problem fails or using binary search

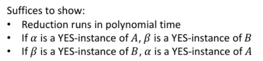
5.8.2 Reduction between Decision Problems

Given two decision problems A and B , a polynomial-time reduction from A to B denoted $A \leq_p B$ is a transformation from instances α of A and β of B such that

1. α is a YES-instance of $A \Leftrightarrow \beta$ is a YES-instance from B. Could alternatively show YES-instance of $A \longrightarrow$ YES-instance of B AND NO-instance of $A \longrightarrow$ NO-instance of B
2. The transformation takes polynomial time in the size of α

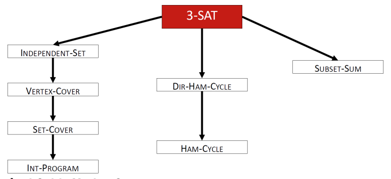


instance α of A
polynomial-time reduction algorithm
instance β of B
polynomial-time algorithm to decide B
yes
no
yes
no
polynomial-time algorithm to decide A



Suffices to show:
- Reduction runs in polynomial time
- If α is a YES-instance of A, β is a YES-instance of B
- If β is a YES-instance of B, α is a YES-instance of A

6 NP-Completeness

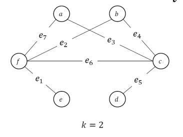


6.1 Independent Set to Vertex Cover

- **Independent-Set**: Given a graph $G = (V, E)$ and an integer k, is there a subset of $\geq k$ vertices such that no 2 are adjacent
- **Vertex-Cover**: Given a graph $G = (V, E)$, and an integer k, is there a subset of $\leq k$ vertices such that each edge is incident ot at least 1 vertex in the subset
- **Reduction**: Given an instance of (G, k) of independent set, output $(G, n-k)$ as instance of vertex-cover $\longrightarrow$ reduction is clearly polynomial since we only change k to $n-k$
- **YES-instance to YES-instance**
  - Suppose G has a set of vertices S of size $\geq k$ such that no 2 are connected by an edge
  - **Claim**: $V \backslash S$ is a vertex cover of size at most $(n-k)$
  - Let $(u, v) \in E$, then either $u \notin S$ or $v \notin S$. So either u or v will be in $V \backslash S$
- **NO-instance to NO-instance**
  - Suppose $(G, n-k)$ is a YES-instance of Vertex-Cover. That is, there is a subset of S of $\leq (n-k)$ vertices that cover all edges
  - **Claim**: $V \backslash S$ is an independent set of size at least k
  - Let $(u, v) \in E$, if both $(u, v) \in V \backslash S$, then the edge (u, v) is not covered by S
  - So for any $u \in V \backslash S$, none of its neighbors are in $V \backslash S$, so it is an independent set

6.2 Vertex-Cover to Set-Cover

- **Set-Cover**: Given integers k, n and a collection $\mathcal{S}$ of subsets of $\{1, \ldots, n\}$, are there $\leq k$ of these subsets who union equals $\{1, \ldots, n\}$
- **Reduction**: Given instance $(G = (V, E), k)$ of Vertex-Cover, we generate an instance $(n, k', \mathcal{S})$ of Set-Cover as follows:
  - Set $n = |E|$ and $k' = k$
  - Order the edges of G arbitrarily: $e_1, \ldots, e_n$
  - For each $v \in V$, define subset $S_v = \{i : e_i$ incident on $v\}$
  - $\mathcal{S}$ is the collection of all such sets S_v



$k = 2$

Vertex-Cover

$n = |E| = 7$

$k' = k = 2$

$S_a = \{3,7\}$ $S_b = \{2,4\}$
$S_c = \{3,4,5,6\}$ $S_d = \{5\}$
$S_e = \{1\}$ $S_f = \{1,2,6,7\}$

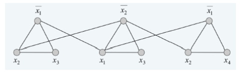
Set-Cover

- **Running time**:
  - Polynomial in $|E|$ to order edges
  - For each $v \in V$, polynomial time in $|E|$ to write down S_v
  - Total polynomial time in $|E|$ which is polynomial in the input size
- **YES-instance to YES-instance**
  - Suppose G has a vertex-cover T of size k, that is every edge in E is adjacent to some $v \in T$
  - Each i is contained in S_v if e_i is adjacent to v. So the union of S_v's for $v \in T$ contain all $i \in \{1, \ldots, n\}$
  - So there is a set of k sets in $\mathcal{S}$ that cover $\{1, \ldots, n\}$, so $(n, k', \mathcal{S})$ is a YES-instance of Set-Cover
- **NO-instance to NO-instance**
  - Suppose $(n, k', \mathcal{S})$ is a YES-instance of Set-Cover, that is there is a set T of k' sets from $\mathcal{S}$ that cover $\{1, \ldots, n\}$
  - If i is contained in S_v then e_i is adjacent to v. So the set of vertices v corresponding to the sets $S_v \in T$ cover all edges $\{e_1, \ldots, e_n\}$
  - So G has a vertex cover of size k, so (G, k) is a YES-instance of Vertex-Cover

6.3 3-SAT to Independent-Set

- **SAT**: given a CNF formula ϕ, does it have a satisfying assignment
  - Literal: boolean variable and its negation - $x_i, \bar{x_i}$
  - Clause: disjunction (OR) of literals - $C_j = x_1 \vee \bar{x_2} \vee x_3$
  - Conjunctive Normal Form (CNF): a formula ϕ that is a conjunction (AND) of clauses - $\phi = C_1 \wedge C_2 \wedge C_3$
- 3-SAT: SAT where each clause is given in the given formula **contains exactly 3 literals** corresponding to different variables. Max number of clauses is 2^n where n is the number of variables (x_i)
  - Fastest algorithm known for 3-SAT is $1.308^n \longrightarrow$ exponential running time
- **Reduction**:
  - G contains 3 vertices for each clause - one for each literal
  - Connect the 3 literals in each clause in a triangle
  - Connect each literal to each of its negation
  - Set k = number of clauses

$(\overline{x_1} \vee x_2 \vee x_3)$
$\wedge (x_1 \vee \overline{x_2} \vee x_3)$
$\wedge (\overline{x_1} \vee x_2 \vee x_4)$



- **Runtime**: Polynomial in size of ϕ
- **YES-instance to YES-instance**
  - If ϕ is satisfiable, take any satisfying assignment and, for each clause, some literal that is set to be true by this assignment
  - The vertices corresponding to those literals form an independent set of size $k \longrightarrow$ the vertices are from different clauses so wont be connected
- **NO-instance to NO-instance**:
  - Suppose (G, k) is a YES-instance and has independent set S of size k
  - Each of the k triangles must contain exactly 1 vertex from S
  - Set those literals to true and all clauses will be satisfied, so ϕ is satisfiable

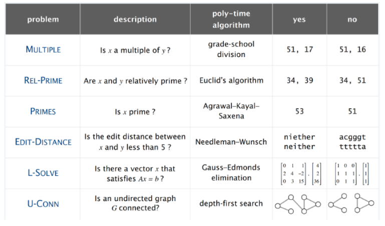
6.4 Importance of 3-SAT

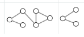
- Since reductions are transitive, if 3-SAT is shown not to have a poly time algorithm, then none of the algorithms will have poly time algorithm

6.5 P vs NP

- **P**: the class of decision problems solvable in (*deterministic*) polynomial time
- **NP**: !the class of problems for which polynomial-time verifiable certificates of YES-instances exist
  - aka *non-deterministic polynomial*, no known polynomial time algorithm that solves it but verification can be done in polynomial time
  - Verification algorithm is defined as a function $V(x, y)$ that takes an instance x and certificate y with $|y| = poly(|x|)$ such that: there exist a y for which $V(x, y) = 1$ iff x is a YES-instance

6.5.1 Problems in P



problem	description	poly-time algorithm	yes	no
MULTIPLE	Is x a multiple of y?	grade-school division	51, 17	51, 16
REL-PRIME	Are x and y relatively prime?	Euclid's algorithm	34, 39	34, 51
PRIMES	Is x prime?	Agrawal–Kayal–Saxena	53	51
EDIT-DISTANCE	Is the edit distance between x and y less than 5?	Needleman–Wunsch	niether neither	acgggt ttttta
L-SOLVE	Is there a vector x that satisfies $Ax = b$?	Gauss–Edmonds elimination	$\begin{bmatrix} 0 & 1 & 1 \\ 2 & 4 & -2 \\ 0 & 3 & 15 \end{bmatrix}, \begin{bmatrix} 4 \\ 2 \\ 36 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$
U-CONN	Is an undirected graph G connected?	depth-first search		

6.5.2 Verification Example: Subset-Sum

- In Subset-Sum, given a list of integers S and target t, problem is to decide if there is $S' \subseteq S$ that sums up to t
- Certificate is the subset S' itself. Verifier checks whether the sum of elements in S' is t in polynomial time
- Hence Subset-Sum is in NP.

6.5.3 Verification Example: Ham-Cycle

- In Ham-Cycle, given a graph G, problem is to decide whether there is a simple cycle that visits each vertex once
- Certificate is the cycle itself. Verifier checks in polynomial time whether it is a cycle and visits each vertex once
- Hence, HAM-CYCLE is in NP

6.5.4 Verification Example: Factor

- Given numbers m and $t < m$, is there an $1 < f \leq t$ that divides m
- Certificate is the number f itself. Verifier checks in poly time if it is $\leq t$ and divides m
- Hence, FACTOR is in NP

6.5.5 P $\subseteq$ NP

- All problem in P is in NP!
- The certificate can be anything. The verifier $V(x, .)$ can solve for the instance x by itself and check if it is a YES-instance

6.5.6 co-NP

- A problem is in co-NP if polynomial time verifiable certificates ("counterexamples") of NO instances exist
- Complement of any NP problem is in co-NP
- Factor is also in co-NP, where the certificate for the NO-instance is the prime factorization of m. Verifier checks that all factors are more then t and also verifies that they are prime using the AKS primality test

6.5.7 NP-Hard & NP-Complete

- a problem A is said to be **NP-Hard** if for every problem $B \in NP$, $B \leq_p A \longrightarrow A$ is at least as hard as every problem in NP
- a problem A is said to be **NP-Complete** if it is in **NP** and is also **NP-hard**
- **Cook-Levin-Theorem** $\longrightarrow$ every problem in NP-Hard can be poly-time reduced to 3-SAT. Hence, 3-SAT is NP-Hard and NP-Complete.
- To show a problem is NP-Hard:
  1. Show that X is in NP $\longrightarrow$ YES-instance has a certificate that can be verified in poly time
  2. Show that X is in NP-Hard $\longrightarrow$ show poly-time reduction from another NP-Hard problem A to X
  3. Show that reduction is valid $\longrightarrow$ YES-instance to YES-instance, NO-instance to NO-instance, poly time reduction

7 Approximation Algorithms

Previously shown that there are many NP-Complete problems that cannot be solved in poly time but yet they show up often in practice. To deal with them:

1. Use exponential time algorithm (brute force) if a correct answer is required $\longrightarrow$ Can be used for small instances
2. See if given instance has any special features that can make it easier to solve $\longrightarrow$ e.g. knapsack with small capacity can be solved using the pseudopoly DP solution
3. Find an approximation to the problem $\longrightarrow$ applicable to **optimization problems**

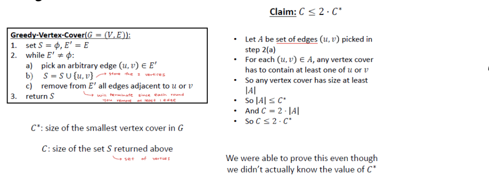
7.1 Optimization Problems

- Vertex-Cover (decision) - asks if there exist a subset of $\leq k$ vertices such that each edge is incident to at least 1 vertex in subset
- Vertex-Cover (optimization) - want to find the smallest subset of vertices ...
- If optimization can be solved then decision can be solved $\longrightarrow$ just check whether optimal solution is less than constraint in decision problem!
- If P $\neq$ NP, optimization problem cannot be solved in poly time

7.2 Approximating Optimization Problem

- **Minimization Problem**: Given an instance, find a solution that has minimum cost
  - C^*: cost of optimal solution
  - C: cost of solution found by approximation algorithm
  - Approximation Ratio: $\frac{C}{C^*} \longrightarrow$ always larger than 1. Good approximation algorithm will have small approximation ratio and is ideally a constant
- **Maximization Problem**:
  - Approximation ratio will be $\frac{C^*}{C}$ instead

7.3 Approximating Vertex Cover



Claim: $C \leq 2 \cdot C^*$

```
Greedy-Vertex-Cover(G = (V, E)):
1.  set S = ϕ, E' = E
2.  while E' ≠ ϕ:
      a) pick an arbitrary edge (u, v) ∈ E'
      b) S = S ∪ {u, v}   ↪ store the 2 vertices
      c) remove from E' all edges adjacent to u or v
3.  return S   ↪ will terminate since each round you remove at least 1 edge
```

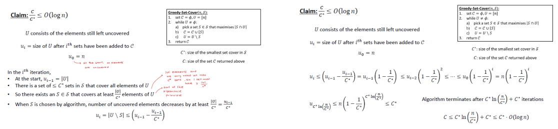
C^*: size of the smallest vertex cover in G

C: size of the set S returned above ↪ set of vertices

- Let A be set of edges (u, v) picked in step 2(a)
- For each $(u, v) \in A$, any vertex cover has to contain at least one of u or v
- So any vertex cover has size at least $|A|$
- So $|A| \leq C^*$
- And $C = 2 \cdot |A|$
- So $C \leq 2 \cdot C^*$

We were able to prove this even though we didn't actually know the value of C^*

7.4 Approximating Set Cover



Claim: $\frac{C}{C^*} \leq O(\log n)$

U consists of the elements still left uncovered

u_i = size of U after i^{th} sets have been added to $\mathcal{C}$

$u_0 = n$ ↪ at the start, all elements are uncovered

```
Greedy-Set-Cover(n, 𝒮):
1.  set 𝒞 = ϕ, U = [n]
2.  while U ≠ ϕ:
      a) pick a set S ∈ 𝒮 that maximises |S ∩ U|
      b) 𝒞 = 𝒞 ∪ {S}
      c) U = U \ S
3.  return 𝒞
```

C^*: size of the smallest set cover in $\mathcal{S}$

C: size of the set $\mathcal{C}$ returned above

In the i^{th} iteration,
- At the start, $u_{i-1} = |U|$
- There is a set of $\leq C^*$ sets in $\mathcal{S}$ that cover all elements of U — |U| elements and we only need at max C^* sets, so 1 set must have $\geq \frac{|U|}{C^*}$
- So there exists an $S \in \mathcal{S}$ that covers at least $\frac{|U|}{C^*}$ elements of U — sort of like pigeonhole principle
- When S is chosen by algorithm, number of uncovered elements decreases by at least $\frac{|U|}{C^*} = \frac{u_{i-1}}{C^*}$

$$u_i = |U \setminus S| \leq \left(u_{i-1} - \frac{u_{i-1}}{C^*}\right)$$

Claim: $\frac{C}{C^*} \leq O(\log n)$

U consists of the elements still left uncovered

u_i = size of U after i^{th} sets have been added to $\mathcal{C}$

$u_0 = n$

```
Greedy-Set-Cover(n, 𝒮):
1.  set 𝒞 = ϕ, U = [n]
2.  while U ≠ ϕ:
      a) pick a set S ∈ 𝒮 that maximises |S ∩ U|
      b) 𝒞 = 𝒞 ∪ {S}
      c) U = U \ S
3.  return 𝒞
```

C^*: size of the smallest set cover in $\mathcal{S}$

C: size of the set $\mathcal{C}$ returned above

$$u_i \leq \left(u_{i-1} - \frac{u_{i-1}}{C^*}\right) = u_{i-1}\left(1 - \frac{1}{C^*}\right) \leq u_{i-2}\left(1 - \frac{1}{C^*}\right)^2 \leq \cdots \leq u_0\left(1 - \frac{1}{C^*}\right)^i = n\left(1 - \frac{1}{C^*}\right)^i$$

$$u_{C^* \ln\left(\frac{n}{C^*}\right)} \leq n\left(1 - \frac{1}{C^*}\right)^{C^* \ln\left(\frac{n}{C^*}\right)} \leq C^*$$

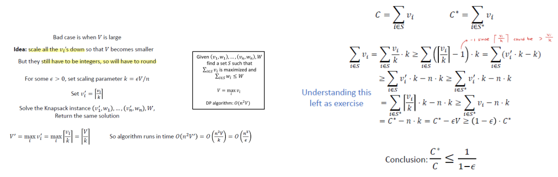
Algorithm terminates after $C^* \ln\left(\frac{n}{C^*}\right) + C^*$ iterations

$$C \leq C^* \ln\left(\frac{n}{C^*}\right) + C^* \leq C^* \cdot O(\log n)$$

7.5 Approximating Knapsack

Earlier, we used DP to solve Knapsack in pseudo poly time of $O(nW)$. In the approximation algorithm, we use another DP algorithm that solves Knapsack in $O(n^2V)$ where V is the max value of any item

- Need to choose a scaling parameter $k = \epsilon V/n$ for some $\epsilon > 0$
- S which is the solution given by the approximation algorithm still satisfies the weight constraint since we did not change the weight at all $\longrightarrow$ Approximation is $\approx$ optimal!



Bad case is when V is large

Idea: scale all the v_i's down so that V becomes smaller

But they still have to be integers, so will have to round

For some $\epsilon > 0$, set scaling parameter $k = \epsilon V/n$

Set $v_i' = \left\lceil \frac{v_i}{k} \right\rceil$

Solve the Knapsack instance $(v_1', w_1), \ldots, (v_n', w_n), W$,
Return the same solution

$V' = \max_i v_i' = \max_i \left\lceil \frac{v_i}{k} \right\rceil = \left\lceil \frac{V}{k} \right\rceil$

So algorithm runs in time $O(n^2V') = O\left(\frac{n^2V}{k}\right) = O\left(\frac{n^3}{\epsilon}\right)$

Given $(v_1, w_1), \ldots, (v_n, w_n), W$
find a set S such that
$\sum_{i \in S} v_i$ is maximized and
$\sum_{i \in S} w_i \leq W$

$V = \max v_i$

DP algorithm: $O(n^2V)$

$$C = \sum_{i \in S} v_i \qquad C^* = \sum_{i \in S^*} v_i$$

Understanding this left as exercise

-1 since $\left\lceil \frac{v_i}{k} \right\rceil$ could be $> \frac{v_i}{k}$

$$\begin{aligned}\sum_{i \in S} v_i &= \sum_{i \in S} \frac{v_i}{k} \cdot k \geq \sum_{i \in S} \left(\left\lceil \frac{v_i}{k} \right\rceil - 1\right) \cdot k = \sum_{i \in S} (v_i' \cdot k - k) \\ &\geq \sum_{i \in S} v_i' \cdot k - n \cdot k \geq \sum_{i \in S^*} v_i' \cdot k - n \cdot k \\ &= \sum_{i \in S^*} \left\lceil \frac{v_i}{k} \right\rceil \cdot k - n \cdot k \geq \sum_{i \in S^*} v_i - n \cdot k \\ &= C^* - n \cdot k = C^* - \epsilon V \geq (1 - \epsilon) \cdot C^*\end{aligned}$$

Conclusion: $\frac{C^*}{C} \leq \frac{1}{1-\epsilon}$

7.6 Approximation Schemes

- A Polynomial-Time Approximation Scheme (PTAS) for a problem is an algorithm that given an instance and an $\epsilon > 0$, runs in time $poly(n)f(\epsilon)$ for some function f and has approximation ratio $(1 + \epsilon)$
  - Note that $f(\epsilon)$ could be exponential!!
- A Fully Polynomial-Time Approximation Scheme (FPTAS) for a problem is an algorithm that given an instance and an $\epsilon > 0$, runs in time $poly(n, 1/\epsilon)$ and has approximation ratio $(1 + \epsilon)$
- Approximation algorithm for Knapsack is a FPTAS

8 Asymptotic Notations & Master Theorem

O-notation [upper bound ($\leq$)]: $f(n) = O(g(n))$

$$\boxed{0 \leq f(n) \leq cg(n)}$$

Ω-notation [lower bound ($\geq$)]: $f(n) = \Omega(g(n))$

$$\boxed{0 \leq cg(n) \leq f(n)}$$

Θ-notation [tight bound]: $f(n) = \Theta(g(n))$

$$\boxed{0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)}$$

o-notation ($<$): $f(n) = o(g(n))$

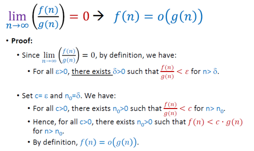
$$\boxed{0 \leq f(n) < cg(n)}$$

ω-notation ($>$): $f(n) = \omega(g(n))$

$$\boxed{0 \leq cg(n) < f(n)}$$

Note: $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$

8.1 Limits



$\lim_{n \to \infty} \left(\frac{f(n)}{g(n)}\right) = 0 \rightarrow f(n) = o(g(n))$

- **Proof:**
  - Since $\lim_{n \to \infty} \left(\frac{f(n)}{g(n)}\right) = 0$, by definition, we have:
    - For all ε>0, there exists δ>0 such that $\frac{f(n)}{g(n)} < \varepsilon$ for n> δ.
  - Set c= ε and n₀=δ. We have:
    - For all c>0, there exists n₀>0 such that $\frac{f(n)}{g(n)} < c$ for n> n₀.
    - Hence, for all c>0, there exists n₀>0 such that $f(n) < c \cdot g(n)$ for n> n₀.
    - By definition, $f(n) = o(g(n))$.

Assume $f(n), g(n) > 0$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \qquad \Rightarrow f(n) = o(g(n))$$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty \qquad \Rightarrow f(n) = O(g(n))$$

$$0 < \lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty \qquad \Rightarrow f(n) = \Theta(g(n))$$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} > 0 \qquad \Rightarrow f(n) = \Omega(g(n))$$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty \qquad \Rightarrow f(n) = \omega(g(n))$$

8.2 Common useful facts

- $1 < \log\log n < \log n < (\log n)^k < \sqrt{n} < n < n^k < a^n < n! < n^n$
- $2^{2 \cdot 2^{\lg\lg n}} < n^2 \lg\lg n < n^3 \equiv \sum_{i=2}^{n} \frac{n^3}{i(i-1)} < n^{\lg n} < 2^n < (\lg n)^n < n!$

8.3 Master Method

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } f(n) < n^{\log_b a} \text{ polynomially} \\ \Theta(n^{\log_b a} \log n) & \text{if } f(n) = n^{\log_b a} \\ \Theta(f(n)) & \text{if } f(n) > n^{\log_b a} \text{ polynomially} \end{cases}$$ where

$a \geq 1, b > 1$ and f is asymptotically positive **3 common cases of Master Method**

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$,
   - $f(n)$ grows polynomially slower than $n^{\log_b a}$ by n^{ε} factor.
   - then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a} \log^k n)$ for some $k \geq 0$,
   - $f(n)$ and $n^{\log_b a}$ grow at similar rates.
   - then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$,
   - and $f(n)$ satisfies the **regularity condition**
     - $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n
     - this guarantees that the sum of subproblems is smaller than $f(n)$.
   - $f(n)$ grows polynomially faster than $n^{\log_b a}$ by n^{ε} factor
   - then $T(n) = \Theta(f(n))$.

9 Miscellaneous

9.1 Useful Facts

- $e^x \geq 1 + x$
- $(1 - \frac{1}{n})^n \approx \frac{1}{e}$
- $\sum_{j=0}^{\lg(n-1)} 2^j = 2^{\lg(n-1)+1} - 1 \leq 2(n-1)$
- $a = b^{\log_b a}$
- $\log_c(ab) = \log_c a + \log_c b$
- $\sum_{i=0}^{\lg n} \frac{1}{\lg n/a^i} = \lg\lg n$
- $(\lg n)! \leq \lg n^{\lg n} = 2^{\lg n \lg\lg n}$
- $\sum_{i=1}^{\lg n} \lg\lg \frac{n}{i} = \lg n \lg\lg n$
- $\log_b a^n = n \log_b a$
- $\log_b a = \frac{\log_c a}{\log_c b}$
- $\log_b(1/a) = -\log_b a$
- $\log_b a = \frac{1}{\log_a b}$
- $a^{\log_b c} = c^{\log_b a}$
- $\frac{d}{dn} \lg\lg n = \frac{1}{n \lg n}$
- $n^2 \log_n n! = n^2 \times \frac{\lg n!}{\lg n} = n^3$

9.2 Approximations and Series

- Stirling's Approximation:
  - $n! = \sqrt{2\pi n}(\frac{n}{e})^n(1 + \Theta(\frac{1}{n}))$
  - $\log(n!) = \Theta(n \log n)$
- Arithmetic Series: $\sum_{k=1}^{n} = 1 + 2 + 3 + \cdots + n = \frac{1}{2}n(n+1) = \Theta(n^2)$
- Harmonic Series: $\sum_{k=1}^{n} \frac{1}{k} = \Theta(\lg(n))$
- Geometric Series: $\sum_{k=0}^{n} x^k = 1 + x + x^2 + \cdots + x^n = \frac{x^{n+1} - 1}{x - 1}$
- Geometric Series: $\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$ when $|x| < 1$
- L'Hopital's Rule: $\lim_{x \to \infty} \frac{f(x)}{g(x)} = \lim_{x \to \infty} \frac{f'(x)}{g'(x)}$

9.2.1 Arithmetic and Geometric Series

	Arithmetic	Geometric
a_n	$a_n = a_1 + d(n-1)$	$a_n = a_1 \cdot r^{n-1}$
S_n	$S_n = \frac{n}{2}(a_1 + a_n)$	$S_n = a_1\left(\frac{1-r^n}{1-r}\right)$
Infinite Series		$S_\infty = \frac{a_1}{1-r}, \lvert r \rvert < 1$

9.3 Permutations and Combinations

- nPr = n! / (n-r)!
- nCr = n! / (n-r)!r!
- $\binom{2n}{n} \approx 2^{2n}/\sqrt{n}$