CS3230 Design and Analysis of Algorithms

AY 25/26 Sem 1 — github/omgeta

1. Algorithms

Algorithms are sequences of instructions to solve problems.

Runtime analysis for $T(n)$ is usually done for worst-case (maximum time for any input) or average-case (expected time over all inputs)

Correctness Proofs

Iterative Algorithms:

i. Loop Invariant: define I for every call or loop.
ii. Initialization: show I holds before first iteration.
iii. Maintenance: assuming I true at start, show I holds true at start of next iteration.
iv. Termination: show when the loop exits, I together with the exit condition implies correctness.

Recursive Algorithms:

i. Base Case: show algorithm is correct for base cases
ii. Inductive Step: assuming correctness for input smaller than n, show algorithm is correct for any input of size n

Mathematical Properties

Logarithm Rules:

i. $a = b^{\log_b a}$ (Inverse)
ii. $\log(ab) = \log a + \log b$ (Product)
iii. $\log(\frac{a}{b}) = \log a - \log b, \quad \log(\frac{1}{a}) = -\log a$ (Quotient)
iv. $\log_b a = \frac{\log_c a}{\log_c b}, \quad \log_b a = \frac{1}{\log_a b}$ (Change of Base)
v. $\log(a^k) = k \log a$ (Power)

Exponential Rules:

i. $a^m a^n = a^{m+n}$ (Product)
ii. $\frac{a^m}{a^n} = a^{m-n}$ (Quotient)
iii. $(a^m)^n = a^{mn} = (a^n)^m$ (Power)

2. Asymptotic Analysis

Asymptotic bounds for functions $f(n), g(n)$:

i. $f(n) \in O(g(n)) \iff f(n) \leq cg(n)$, where $\exists c, n_0, \forall n \geq n_0$ (Upper)
ii. $f(n) \in \Omega(g(n)) \iff f(n) \geq cg(n)$, where $\exists c, n_0, \forall n \geq n_0$ (Lower)
iii. $f(n) \in \Theta(g(n)) \iff c_1 g(n) \leq f(n) \leq c_2 g(n)$, where $\exists c_1, c_2, n_0, \forall n \geq n_0$ (Tight)
iv. $f(n) \in o(g(n)) \iff f(n) < cg(n)$, where $\forall c > 0, \exists n_0, \forall n \geq n_0$ (Strict Upper)
v. $f(n) \in \omega(g(n)) \iff f(n) > cg(n)$, where $\forall c > 0, \exists n_0, \forall n \geq n_0$ (Strict Lower)

Limits assuming $f(n), g(n) > 0$:

i. $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \implies f(n) \in O(g(n))$
ii. $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0 \implies f(n) \in \Omega(g(n))$
iii. $0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \implies f(n) \in \Theta(g(n))$
iv. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \implies f(n) \in o(g(n))$
v. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \implies f(n) \in \omega(g(n))$

Properties:

i. Transitivity - for $O, \Omega, \Theta, o, \omega$:
   - $f(n) = O(g(n)) \land g(n) = O(h(n)) \implies f(n) = O(h(n))$
ii. Reflexivity - for O, Ω, Θ:
   - $f(n) = O(f(n))$
iii. Symmetry - for Θ:
   - $f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n))$
iv. Complementarity - for O, Ω, o, ω:
   - $f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$
   - $f(n) = o(g(n)) \iff g(n) = \omega(f(n))$

Order of growth is:
$O(1) < O(\frac{1}{k}^n) < O(\log\log n) < O(\log n) < O(\log^k n) < O(n^{\frac{1}{k}}) < O(n) < O(n\log n) < O(n^k) < O(k^n) < O(n!)$

3. Divide & Conquer

Divide & Conquer involves:

i. Divide: split problem into smaller subproblems
ii. Conquer: solve subproblems recursively
iii. Combine: merge subresults to form a total solution

Problems

MergeSort divides the array into halves, recursively sorts each half, and merges the sorted halves.

i. Locally sorted prefix in power of 2
ii. $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$
iii. Out-of-place and stable

Exponentiation by squaring computes x^n.
If n even: compute $(x^{n/2})^2$; if n odd: return $x \cdot (x^{\lfloor n/2 \rfloor})^2$.

i. $T(n) = T(\lfloor n/2 \rfloor) + \Theta(1) = \Theta(\log n)$.

Fibonacci finds $F(n)$ uses $F(2k) = F(k)\big(2F(k+1) - F(k)\big)$ and $F(2k+1) = F(k+1)^2 + F(k)^2$:

i. $T(n) = T(\lfloor n/2 \rfloor) + \Theta(1) = \Theta(\log n)$
ii. Naïve: $T(n) = T(n-1) + T(n-2) + \Theta(1) = \Theta(\varphi^n)$
iii. Stair Climbing: $G(n) = F(n+1)$

Matrix Multiplication partitions input A, B into quadrants of size $n/2$ aand recombine via 8 submatrix products and 4 additions into C:

i. $T(n) = 8T(n/2) + \Theta(n^2) = \Theta(n^3)$
ii. Pad non-square input matrices with zeroes

Strassen's Algorithm computes 7 block products of input A, B and recombines via linear combinations to form C:

i. $T(n) = 7T(n/2) + \Theta(n^2) = \Theta\big(n^{\log 7}\big) \approx \Theta(n^{2.807})$.

4. Recurrences

Common recurrences:

i. $T(n) = T(n/2) + O(1) = O(\log n)$
ii. $T(n) = T(n/2) + O(n) = O(n)$
iii. $T(n) = 2T(n/2) + O(1) = O(n)$
iv. $T(n) = 2T(n/2) + O(n) = O(n \log n)$
v. $T(n) = T(n-1) + O(1) = O(n)$
vi. $T(n) = T(n-1) + O(n) = O(n^2)$
vii. $T(n) = 2T(n-1) + O(1) = O(2^n)$

Telescoping Method

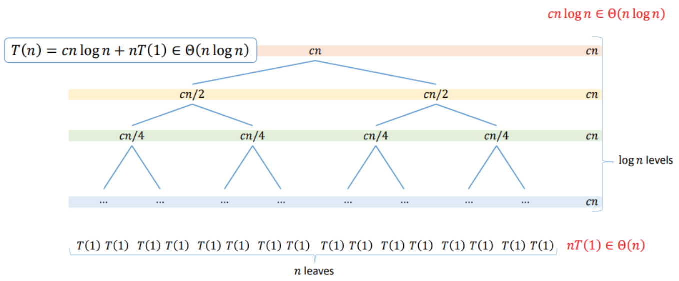
Given recurrence $T(n) = aT(\frac{n}{b}) + f(n)$, divide expression to get $\frac{T(n)}{g(n)} = \frac{T(n/b)}{g(n/b)} + \frac{f(n)}{g(n)}$

i. Cancel out common terms to solve for $T(n)$

Recursion Tree

Given recurrence, draw recurrence tree:

i. Total = depth $\times$ work done per level



Master Theorem

Dividing function $T(n) = aT(n/b) + f(n)$ where $a > 1, b > 1$ has cases:

i. $f(n) \in O(n^{d-\varepsilon}) \implies T(n) = \Theta(n^{\log_b a})$
ii. $f(n) \in \Theta(n^d \log^p n)$
$\wedge\, p > -1 \implies T(n) = \Theta(n^k \log^{p+1} n)$
$\wedge\, p = -1 \implies T(n) = \Theta(n^k \log\log n)$
$\wedge\, p < -1 \implies T(n) = \Theta(n^k)$
iii. $f(n) \in \Omega(n^{d+\varepsilon}) \wedge\ af(n/b) \leq cf(n)$, for $c < 1$
$\implies T(n) = \Theta(f(n))$

Decreasing function $T(n) = aT(n-b) + f(n)$ where $a > 0, b > 0$ has cases:

i. $a < 1 \implies T(n) = O(f(n))$
ii. $a = 1 \implies T(n) = O(n \cdot f(n))$
iii. $a > 1 \implies T(n) = O(a^{n/b} \cdot f(n))$

Substitution Method

Guess $T(n) = O(f(n))$ and verify by induction:

i. Choose values of $c > 0, n_0$ from recurrence
ii. Base case: verify $T(n_0) \leq cf(n)$
iii. Inductive step: assuming $T(k) \leq cf(k)$ for $n > k \geq n_0$, prove $T(n) \leq cf(n)$ by subbing $T(k)$

Example

Prove $T(n) = 4T(n/2) + n \in O(n^2)$:

1. Induction hypothesis: $T(n) \leq (c+1)n^2 - n$
2. Base case: If $n = 1, T(n) = c \leq (c+1)n^2 - n$
3. Inductive step:
   3.1. By strong induction, assume $T(k) \leq (c+1)k^2 - k$ for all $n > k \geq 1$
   3.2. $T(n) = 4T(n/2) + n$
   3.3. $\leq 4(c+1)(n/2)^2 - 4(n/2) + n$
   3.4. $= (c+1)n^2 - n$
4. $\therefore T(n) \in O(n^2)$

5. Probabilistic Analysis

Average-case runtime $A(n)$ is the expected running time over the distribution of possible inputs (uniformly over $n!$ permutations for a uniform random permutation).

QuickSort rearranges the array by $\leq$ and $>$ a chosen pivot in $\Theta(n)$, then recurses on both partitions.

i. Elements before pivot are $\leq$, after pivot are $>$
ii. $T(n) = T(j-1) + T(n-j) + \Theta(n)$, where pivot is j^{th} smallest element.
iii. Worst: $T(n) = T(0) + T(n-1) + cn = \Theta(n^2)$, when $j = 1 \vee j = n$
iv. Average: $A(n) = \frac{1}{n}\sum_{j=1}^{n}[A(j-1) + A(n-j) + cn]$
$= cn + \frac{2}{n}\sum_{j=0}^{n-1} A(j) = O(n \log n)$ (by telescoping)
v. In-place but unstable

6. Randomized Algorithms

Randomized algorithms are dependent on random bits:

i. Las Vegas: always correct, running time is random
ii. Monte Carlo: may be wrong, running time is finite

Tools

Union Bound can upper bound probability that bad event $\varepsilon = \varepsilon_1 \cup \cdots \cup \varepsilon_n$ occurs: $\Pr[\varepsilon] \leq \Pr[\varepsilon_1] + \cdots + \Pr[\varepsilon_n]$

i. $\therefore \Pr[\varepsilon_i] \leq \frac{f}{n}, \forall i \in [n] \implies \Pr[\varepsilon] \leq f$

Markov Inequality can turn a Las Vegas into Monte Carlo: $X \geq 0 \wedge a > 0 \Rightarrow \Pr[X \geq a\mathbb{E}[X]] \leq \frac{1}{a}$

Indicator Random Variable $\mathbf{1}_\varepsilon$ is a binary RV for event ε:

i. $\mathbb{E}[\mathbf{1}_\varepsilon] = \Pr(\varepsilon)$

Linearity of Expectation: $\mathbb{E}[A + B] = \mathbb{E}[A] + \mathbb{E}[B]$

Problems

Randomized QuickSort chooses a random pivot:

i. Average: $A(n) = \mathbb{E}[T(n)] = O(n \log n)$ (LoE)

Approximate Median is a pivot selection algorithm. Pick a random pivot with probability $\geq \frac{1}{2}$ to be an approximate median of rank $[\frac{n}{4}, \frac{3n}{4}]$, repeating k times for boosting.

i. $A(n) = O(n)$ (QuickSelect)
ii. Error Rate: 2^{-k}, less than $\frac{1}{n^2}$ if $k = 1 + 10 \log n$

Freivalds' Algorithm verifies if matrices $AB = C$ by picking random bit vector $\vec{r}$ and check if $AB\vec{r} - C\vec{r} == \vec{0}$, repeating k times with accepting if all successes:

i. $A(n) = O(kn^2)$ (3 matrix-vector multiplications)
ii. Error Rate: 2^{-k} when $AB \neq C$

```
FREIVALDS(A, B, C, k)
  n = rows(A)
  for i = 1 to k
    r = RANDOM-01-VECTOR(n)
    ABr = DOT(A, DOT(B, r))
    Cr = DOT(C, r)
    if ABr != Cr then
      return FALSE
  return TRUE
```

Balls-and-Bins deals with placing m balls into n bins:

i. For a fixed bin, $\Pr[\text{empty}] = (1 - \frac{1}{n})^m \leq e^{-m/n}$ (using $1 + x \leq e^x$)
ii. $\Pr[\exists \text{ empty bin}] \leq n\, e^{-m/n}$ (union bound)
iii. Taking $m \geq 2n \ln n$ makes $\Pr[\exists \text{ empty bin}] \leq \frac{1}{n}$ (so all bins nonempty with probability $\geq 1 - \frac{1}{n}$).

7. Dynamic Programming (DP)

Dynamic Programming involves:

i. Divide: split problem into smaller overlapping subproblems
ii. Conquer: solve subproblems recursively with memoization/bottom-up approach
iii. Combine: merge subresults to form a total solution
iv. Optimal Substructure: optimal solution can be constructed from optimal solutions of subproblems
   - Cut-and-paste Proof: Suppose not, i.e. exists "optimal" solution S with suboptimal subsolution S', if replacement by optimal subsolution S'_{opt} strictly improves solution to S_{opt} $\rightarrow$ contradicts optimality of S, justifying optimal substructure.

Problems

Longest Common Subsequence (LCS) of $A[:i]$ and $B[:j]$:

i. $LCS(i,j) = \begin{cases} 0, & i = 0 \lor j = 0 \\ LCS(i-1, j-1) + 1, & A[i] = B[j] \\ \max\{LCS(i-1,j), LCS(i,j-1)\}, & A[i] \neq B[j] \end{cases}$
ii. $T(n) = \Theta(nm)$
iii. Longest Palindromic Subsequence (LPS) in $A[i:j]$ is just LCS of A and $B = reversed(A)$

```
LCS(A, B)
 m = |A|
 n = |B|
 alloc 2D array L[0..m][0..n] = 0
 for i = 1 to m
   for j = 1 to n
     if A[i] = B[j] then
      L[i][j] = L[i-1][j-1] + 1
     else
      L[i][j] = max(L[i-1][j],L[i][j-1])
 return L[m][n]
```

Knapsack finds maximum value v achievable given $items[:i]$ with (v_i, w_i) and maximum weight W:

i. $dp(i,j) = \begin{cases} 0, & i = 0 \lor j = 0 \\ \max\left\{dp(i-1,j), dp(i-1,j-w_i)+v_i\right\}, & w_i \leq j \\ dp(i-1,j), & \text{otherwise} \end{cases}$
ii. $T(n) = \Theta(nW)$
iii. If item counts are unbounded, loop through items $dp(j) = \max_{i:w_i \leq j}\{\, dp(j - w_i) + v_i \,\}$

```
KNAPSACK-01(v, w, W)
 n = |v|
 alloc table DP[0..n][0..W] = 0
 for i = 1 to n
   for j = 0 to W
     if w[i] <= j then
       DP[i][j] = max(DP[i-1][j],
          DP[i-1][j-w[i]] + v[i])
     else
       DP[i][j] = DP[i-1][j]
 return DP[n][W]
```

Coin Change finds fewest coins to make n using k coins d_i:

i. $dp(n) = \begin{cases} 0, & n = 0 \\ 1 + \min_{i \in [k]}\{dp(j - d_i)\} \end{cases}$
ii. $T(n) = O(nk)$

```
COIN-CHANGE(D, N)
 alloc array C[0..N] = INFINITY
 C[0] = 0
 for v = 1 to N
   for coin in D
     if coin <= v
   C[v] = min(C[v], C[v-coin] + 1)
 return C[N]
```

8. Greedy Algorithms

Greedy algorithms solve only one subproblem at each step, a locally optimal choice hoping it's globally optimal. It outperforms DP, and D&C when it works. It involves:

i. Optimal Substructure

ii. Greedy Choice Property: a locally optimal choice must be globally optimal

- Exchange Argument: Suppose exists optimal solution S, if exchange of local choice before reduction with greedy choice does not hurt optimality, new solution S_{opt} stays optimal, justifying greedy choice.

Problems

Fractional Knapsack allows taking fractions of a item:

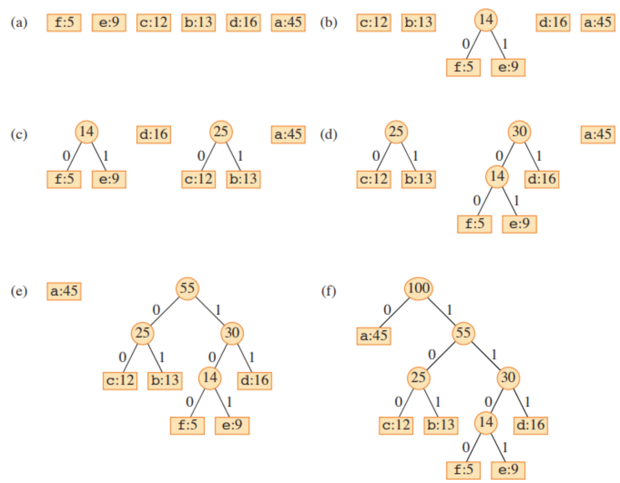
i. Greedy by sorted maximum value/kg v_i, w_i, fill capacity, then take next item

ii. $T(n) = O(n \log n)$

iii. Greedy Choice Property: if j^* is item with maximum value/kg v_j/w_j, there exists optimal knapsack containing $\min\{w_h, W\}$ kg of item h^*

```
FRACTIONAL-KNAPSACK(I, W)
  sort I by (v_i/ w_i) desending
  totalValue = 0
  for each (v, w) in I
      if W = 0
          return totalValue
      take = min(w, W)
      totalValue += take * (v / w)
      W -= take
  return totalValue
```

Huffman Code is an optimal variable length prefix coding, where no prefix $\gamma(x)$ is prefix of $\gamma(y)$, and average bit length $ABL(\gamma) = \sum_{x \in A} freq(x)|\gamma(x)|$ is minimised:



i. Using a min-heap, repeatedly extract two least frequent a, b, create parent z with weight $freq(a) + freq(b)$, and reinsert z. Final tree is an optimal code, where left branch adds 0, and right branch adds 1 to the code.

ii. $T(n) = O(n \log n)$ (using heap)

iii. Greedy Choice Property: any merge not using two smallest nodes increases cost no less than merging the two smallest at every step

```
HUFFMAN(C)
  n = |C|
  Q = C
  for i = 1 to n - 1
    allocate a new node z
    x = EXTRACT-MIN(Q)
    y = EXTRACT-MIN(Q)
    z.left = x
    z.right = y
    z.freq = x.freq + y.freq
    INSERT(Q, z)
  return EXTRACT-MIN(Q)
```

9. Amortized Analysis

Amortized analysis is used over sequence of operations to show that average cost per operation is small, even if some operations are expensive.

i. Guarantee average cost over k operations $\leq kT(n)$ in the worst-case

Methods:

i. Aggregate: $\frac{1}{k}\sum_{i=1}^{k} t(i)$; sum of costs divided by k

ii. Accounting: Overcharge cheap operations to "pay" for expensive operations later

iii. Potential: pick $\phi(0) = 0$, $\phi(i) \geq 0$;
am. cost= $t(i) + \phi(i) - \phi(i-1)$ such that
$\sum$ am. cost of op $i \geq$ actual cost of n ops $= \sum t(i)$

Problems

Increment $k-$bit Binary Counter, counting bit flips:

i. Aggregate: bit j flips $\frac{n}{2^j} \Rightarrow T(n) < 2n \Rightarrow$ am. $O(1)$

ii. Accounting: charge \$2 per $0 \rightarrow 1$, use saved \$1 per $1 \rightarrow 0$ reset; banked $=$ no. of 1s $\geq 0 \Rightarrow$ am. $O(1)$

iii. Potential: let $\phi(i) = \#$ of 1s after op i;
am. cost $= (\ell_i + 1) + (-\ell_i + 1) = 2 = O(1)$

Dynamic Table, when full, alloc size $2n$ and copy n items:

i. Aggregate: $T(n) \leq n + \sum_{j=0}^{\log(n-1)} 2^j \leq 3n$
$\Rightarrow$ am. $O(1)$

ii. Accounting: charge \$2/insert, use saved \$1/copy per item; bank never negative $\Longrightarrow$ am. $O(1)$

iii. Potential: let $\phi(i) = 2i - \text{size}(T)$;
full case am. $= i + (3 - i) = 3 = O(1)$;
not-full case am. $= 1 + 2 = 3 = O(1)$

10. Problem Reduction

Problem A reduces to problem B if we can:

1. convert an instance α of A to an instance β of B, where an instance denotes input
2. solve β with B to obtain solution $B(\beta)$,
3. convert $B(\beta)$ back into a solution $A(\alpha)$ for α.

$p(n)$-time Reduction

Polynomial-time reduction from A to B, $A \leq_P B$, exists if:

i. Converting $\alpha \to \beta$, $B(\beta) \to A(\alpha)$ take $\leq p(n)$ time, and new β takes $\leq p(n)$ size, where $p(n) \in O(n^c)$

ii. Consequence:
   - B is easy ($p(n)$-time) $\implies A$ is easy
   - A is hard (no $p(n)$-time) $\implies B$ is hard

iii. Length encoding: $n = |\alpha|$ is measured in length of bits, e.g. $\ell = \lceil \log n \rceil$ bits for numeric inputs.

iv. Pseudo-polynomial time: if running time is $p(n)$ in numeric value (e.g. $O(n)$ iFib, $O(nW)$ knapsack) but exponential in the bit-length (e.g. $O(2^\ell)$).

Running-time Composition:

i. If $A \leq_P B$ and B can be solved in $T(n)$ time, then A can be solved in $(T(p(n)) + O(p(n)))$ time.

Problems

Longest Palindromic Subsequence $\leq_P$ LCS:

i. $\text{LPS}(\alpha) = \text{LCS}(\alpha, reversed(\alpha))$

ii. Reverse and call LCS in $O(|\alpha|) \implies$ polynomial

Matrix Square $\leq_P$ Matrix Multiplication:

i. $\text{MAT-SQR}(C) = \text{MAT-MULTI}(C, C)$

ii. Copying C in $O(n^2)$ + one mult $\implies$ polynomial

Matrix Multiplication $\leq_P$ Matrix Square:

i. $\text{MAT-MULTI}(A, B) = \text{MAT-SQR}(\begin{bmatrix} 0 & A \\ B & 0 \end{bmatrix})_{1,1}$

ii. Build M and find result in $O(n^2) \implies$ polynomial.

11. NP-Completeness

Decision problems map an instance space I into the solution set {YES, NO}.

i. Given decision problems $A \leq_P B$, α is YES-instance for $A \Leftrightarrow \beta$ is YES-instance for B

Complexity classes for decision problems in a deterministic Turing Machine:

i. P: solvable in $p(n)$ time; $P \subseteq NP$

ii. NP: certificate verifiable in $p(n)$ time
   - Proof: give $p(n)$-time verifier

iii. NP-Hard: if for every problem $X \in$ NP, $X \leq_P A$
   - Proof: can show $p(n)$-time reduction to known NP-Complete problem

iv. NP-Complete: in NP and NP-Hard
   - Proof: prove both NP and NP-Hard

NP-Complete Problems

CSAT checks if DAG with AND, OR, NOT gate nodes and n binary inputs can output 1.

i NP: evaluate certificate with DAG in $O(n)$

ii NP-Hard: any $p(n)$-time NP verifier Q can be built into circuit C in $\Theta(p(n)^2)$ and solved by CSAT, hence all NP problems $\leq_P$ CSAT

CNF-SAT checks satisfiability of CNF (product-of-sums):

i NP-Hard: convert each gate of circuit C to a new var + $O(1)$ clauses. Hence, CSAT $\leq_P$ CNF-SAT

3-SAT checks satisfiability of CNF with 3 literals/clause:

i NP-Hard: for any unrestricted CNF formula ϕ, split every clause with > 3 literals into a chain of 3-literal clauses with auxiliary variables in $O(n)$. Hence, CNF-SAT $\leq_P$ 3-SAT

Independent Set (IS) checks if $\leq k$ nodes in graph $G = (V, E)$ can share no edges:

i. $\text{IS}(G, k) = \text{VC}(G, |V| - k) \implies$ IS $\leq_P$ VC

ii. NP: reject if any edge has both ends in X in $O(|E|)$

iii. NP-Hard: given a 3-SAT formula, create a graph with one node per literal, connecting pairs in the same clause and pairs of complementary literals. Hence, 3-SAT $\leq_P$ IS

Vertex Cover (VC) checks if $\leq k$ nodes in graph $G = (V, E)$ can cover every edge:

i. NP: ensure all edges are adjacent to X in $\Theta(|E|)$

ii. NP-Hard: $\text{VC}(G, k) = \text{IS}(G, |V| - k) \Rightarrow$ VC $\leq_P$ IS

Hitting-Set (HS) checks if set H of $\leq k$ elements can have non-empty intersection with all S_i in $S = \{S_1, \cdots, S_n\}$:

i NP: $\forall S_i$ ensure $H \cap S_i \neq \varnothing$ in $\Theta\big(\sum_i |S_i|\big)$.

ii NP-Hard: $\text{VC}(G, k) = \text{HS}(S = \{\{e\} : e \in E\}, k)$, where reduction is $\Theta(n)$. Hence, VC $\leq_P$ HS.

CLIQUE checks if $\geq k$ nodes in $G = (V, E)$ can form clique:

i. NP: check all nodes are adjacent in $\Theta(k^2)$

ii. NP-Hard: $\text{CLIQUE}(G, k) = \text{IS}(\overline{G}, k)$, where reduction is $\Theta(|E|)$. Hence, CLIQUE $\leq_P$ IS

Additional Information

Stirling Approximation: $n! \sim \sqrt{2\pi n}(\frac{n}{e})^n$

Arithmetic Series:

i. $a_n = a_1 + (n-1)d$

ii. $S_n = \frac{n}{2}(2a_1 + (n-1)d) = \frac{n \times (a_1 + a_n)}{2} \in \Theta(n^2)$

Geometric Series:

i. $g_n = g_1 \times r^{n-1}$

ii. $S_n = \frac{a(1-r^n)}{1-r}$

iii. $S_\infty = \frac{a}{1-r}$ when $|x| < 1$

Harmonic Series:

i. $H_n = 1 + \frac{1}{2}\frac{1}{3} + \cdots + \frac{1}{n} = \ln n + O(1)$