

# 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:

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

Recursive Algorithms:

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

## Mathematical Properties

Logarithm Rules:

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

Exponential Rules:

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

## 2. Asymptotic Analysis

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

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

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

- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty \implies f(n) \in O(g(n))$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} > 0 \implies f(n) \in \Omega(g(n))$
- $0 < \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty \implies f(n) \in \Theta(g(n))$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \implies f(n) \in o(g(n))$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty \implies f(n) \in \omega(g(n))$

Properties:

- Transitivity - for  $O, \Omega, \Theta, o, \omega$ :
  - $f(n) = O(g(n)) \wedge g(n) = O(h(n)) \implies f(n) = O(h(n))$
- Reflexivity - for  $O, \Omega, \Theta$ :
  - $f(n) = O(f(n))$
- Symmetry - for  $\Theta$ :
  - $f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n))$
- Complementarity - for  $O, \Omega, o, \omega$ :
  - $f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$
  - $f(n) = o(g(n)) \iff g(n) = \omega(f(n))$

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:

- Divide: split problem into smaller subproblems
- Conquer: solve subproblems recursively
- Combine: merge subresults to form a total solution

## Problems

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

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

Exponentiation by squaring computes  $x^n$ .

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

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

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

- $T(n) = T(\lfloor n/2 \rfloor) + \Theta(1) = \Theta(\log n)$
- Naïve:  $T(n) = T(n-1) + T(n-2) + \Theta(1) = \Theta(\varphi^n)$

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

- $T(n) = 8T(n/2) + \Theta(n^2) = \Theta(n^3)$
- 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  $C$ :

- $T(n) = 7T(n/2) + \Theta(n^2) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.807})$ .

## 4. Recurrences

Common recurrences:

- $T(n) = T(n/2) + O(1) = O(\log n)$
- $T(n) = T(n/2) + O(n) = O(n)$
- $T(n) = 2T(n/2) + O(1) = O(n)$
- $T(n) = 2T(n/2) + O(n) = O(n \log n)$
- $T(n) = T(n-1) + O(1) = O(n)$
- $T(n) = T(n-1) + O(n) = O(n^2)$
- $T(n) = 2T(n-1) + O(1) = O(2^n)$

### Telescoping Method

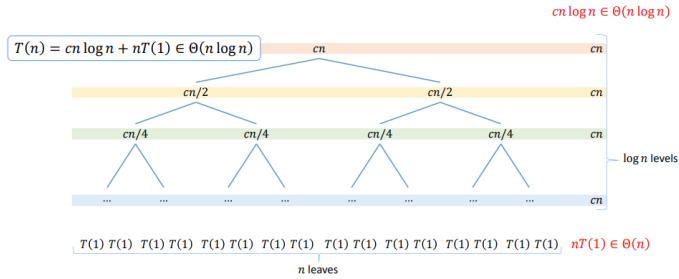
Given recurrence  $T(n) = aT(\frac{n}{b}) + f(n)$ , divide expression to get  $\frac{T(n)}{g(n)} = \frac{T(n/b)}{g(n/b)} + \frac{f(n)}{g(n)}$

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

### Recursion Tree

Given recurrence, draw recurrence tree:

- 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:

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

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

### Substitution Method

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

- Choose values of  $c > 0, n_0$  from recurrence
- Base case: verify  $T(n_0) \leq cf(n)$
- 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  $T(n) = 4T(n/2) + n \in O(n^2)$ :

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

## 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.

- Elements before pivot are  $\leq$ , after pivot are  $>$
- $T(n) = T(j-1) + T(n-j) + \Theta(n)$ , where pivot is  $j^{th}$  smallest element.
- Worst:  $T(n) = T(0) + T(n-1) + cn = \Theta(n^2)$ , when  $j = 1 \vee j = n$
- 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)
- In-place but unstable

## 6. Randomized Algorithms

Randomized algorithms are dependent on random bits:

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

### Tools

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

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

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

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

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

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

## Problems

Randomized QuickSort chooses a random pivot:

- 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.

- $A(n) = O(n)$  (QuickSelect)
- 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:

- $A(n) = O(kn^2)$  (3 matrix-vector multiplications)
- 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:

- For a fixed bin,  $\Pr[\text{empty}] = (1 - \frac{1}{n})^m \leq e^{-m/n}$  (using  $1 + x \leq e^x$ )
- $\Pr[\exists \text{ empty bin}] \leq n e^{-m/n}$  (union bound)
- 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:

- Divide: split problem into smaller overlapping subproblems
- Conquer: solve subproblems recursively with memoization/bottom-up approach
- Combine: merge subresults to form a total solution
- Optimal Substructure: optimal solution can be constructed from optimal solutions of subproblems

Cut-and-paste Proof: If "optimal" solution has suboptimal part, replacing it with an optimal subsolution strictly improves the objective  $\rightarrow$  contradiction, justifying DP.

### Problems

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

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

```

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  $W$ :

- $dp(i, j) = \begin{cases} 0, & i = 0 \vee j = 0 \\ \max\{dp(i-1, j), dp(i-1, j-w_i) + v_i\}, & w_i \leq j \\ dp(i-1, j), & \text{otherwise} \end{cases}$
- $T(n) = \Theta(nW)$
- 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  $d_i$ :

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

```

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:

- Greedy Choice Property: a locally optimal choice must be globally optimal
- Optimal Substructure

### Problems

Fractional Knapsack allows taking fractions of a item:

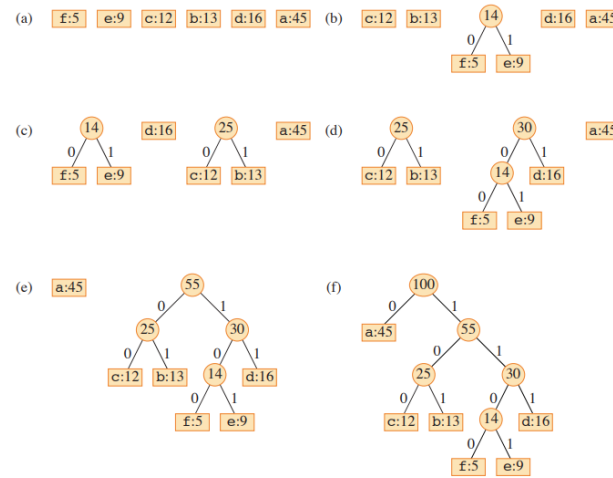
- Greedy by sorted maximum value/kg  $v_i, w_i$ , fill capacity, then take next item
- $T(n) = O(n \log n)$
- Greedy Choice Property: if  $j^*$  is item with maximum value/kg  $v_j/w_j$ , there exists optimal knapsack containing  $\min\{w_h, W\}$  kg of item  $h^*$

```

FRACTIONAL-KNAPSACK(I, W)
  sort I by (v_i / w_i) descending
  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:



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

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

Methods:

- Aggregate:  $\frac{1}{k} \sum_{i=1}^k t(i)$ ; sum of costs divided by  $k$
- Accounting: Overcharge cheap operations to "pay" for expensive operations later
- Potential: pick  $\phi(0) = 0, \phi(i) \geq 0$ ;  
am. cost =  $t(i) + \phi(i) - \phi(i-1)$  such that  
 $\sum \text{am. cost of op } i \geq \text{actual cost of } n \text{ ops} = \sum t(i)$

### Problems

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

- Aggregate: bit  $j$  flips  $\frac{n}{2^j} \Rightarrow T(n) < 2n \Rightarrow \text{am. } O(1)$
- Accounting: charge \$2 per  $0 \rightarrow 1$ , use saved \$1 per  $1 \rightarrow 0$  reset; banked = no. of 1s  $\geq 0 \Rightarrow \text{am. } O(1)$
- 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:

- Aggregate:  $T(n) \leq n + \sum_{j=0}^{\log(n-1)} 2^j \leq 3n \Rightarrow \text{am. } O(1)$
- Accounting: charge \$2/insert, use saved \$1/copy per item; bank never negative  $\Rightarrow \text{am. } O(1)$
- 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  $\alpha$  of  $A$  to an instance  $\beta$  of  $B$ , where an instance denotes input
2. solve  $\beta$  with  $B$  to obtain solution  $B(\beta)$ ,
3. convert  $B(\beta)$  back into a solution  $A(\alpha)$  for  $\alpha$ .

### $p(n)$ -time Reduction

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

- i. Converting  $\alpha \rightarrow \beta$ ,  $B(\beta) \rightarrow A(\alpha)$  take  $\leq p(n)$  time, and new  $\beta$  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.  $LPS(\alpha) = LCS(\alpha, reversed(\alpha))$
- ii. Reverse and call LCS in  $O(|\alpha|) \implies$  polynomial

Matrix Square  $\leq_P$  Matrix Multiplication:

- i.  $MAT-SQR(C) = MAT-MULTI(C, C)$
- ii. Copying  $C$  in  $O(n^2)$  + one mult  $\implies$  polynomial

Matrix Multiplication  $\leq_P$  Matrix Square:

- i.  $MAT-MULTI(A, B) = MAT-SQR(\begin{bmatrix} 0 & A \\ B & 0 \end{bmatrix})_{1,1}$
- 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  $\{\text{YES}, \text{NO}\}$ .

- i. Given decision problems  $A \leq_P B$ ,  $\alpha$  is YES-instance for  $A \iff \beta$  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  $\phi$ , 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.  $IS(G, k) = VC(G, |V| - k) \implies IS \leq_P VC$
- 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:  $VC(G, k) = IS(G, |V| - k) \implies VC \leq_P IS$

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

- i NP:  $\forall S_i$  ensure  $H \cap S_i \neq \emptyset$  in  $\Theta(\sum_i |S_i|)$ .
- ii NP-Hard:  $VC(G, k) = HS(S = \{\{e\} : e \in E\}, k)$ , 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:  $CLIQUE(G, k) = IS(\overline{G}, k)$ , where reduction is  $\Theta(|E|)$ . Hence,  $CLIQUE \leq_P IS$