# CS3230 Cheatsheet AY24/25 —— @JasonYapzx

## 01. Asymptotic Analysis

### Asymptotic Notations

| Definition | If $\exists c, c_1, c_2, n_0 > 0$ s.t. $\forall n \geq n_0$ |
|---|---|
| $f(n) \in O(g(n))$ | $0 \leq f(n) \leq c \cdot g(n)$ |
| $f(n) \in \Omega(g(n))$ | $0 \leq c \cdot g(n) \leq f(n)$ |
| $f(n) \in \Theta(g(n))$ | $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ |

- $f(n) \in O(g(n)) \rightarrow g$ is an **upper** bound on $f$
- $f(n) \in \Omega(g(n)) \rightarrow g$ is an **lower** bound on $f$
- $f(n) \in \Theta(g(n)) \rightarrow g$ is a **tight** bound on $f$

| Definition | For all $c > 0$, $\exists n_0 > 0$ such that $\forall n \geq n_0$ |
|---|---|
| $f(n) \in o(g(n))$ | $0 \leq f(n) < c \cdot g(n)$ |
| $f(n) \in \omega(g(n))$ | $0 < c \cdot g(n) < f(n)$ |

- $f(n) \in o(g(n)) \rightarrow g$ is an **strict upper** bound on $f$
- $f(n) \in \omega(g(n)) \rightarrow g$ is an **strict lower** bound on $f$

### Limits

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

### Properties

$\boxed{\star\ \Theta(g(n)) = O(g(n)) \cap \Omega(g(n))}$

- **transitivity:** applies for $O, \Theta, \Omega, o, \omega$
  $f(n) = O(g(n)) \wedge g(n) = O(h(n)) \Rightarrow f(n) \in O(h(n))$
- **reflexivity:** for $O, \Theta, \Omega$, $f(n) \in O(f(n))$
- **symmetry:** $f(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(f(n))$
- **complementarity:**
  - $f(n) \in O(g(n))$ iff $g(n) \in \Omega(f(n))$
  - $f(n) \in o(g(n))$ iff $g(n) \in \omega(f(n))$
- **misc**
  - if $f(n) \in \omega(g(n))$ then $f(n) \in \Omega(g(n))$
  - if $f(n) \in o(g(n))$ then $f(n) \in O(g(n))$
  - insertion sort: $O(n^2)$ with worst case $\Theta(n^2)$

$\boxed{\log\log n < \log n < (\log n)^k < n^k < k^n}$

## 02. Recurrences

for $a$ sub-problems of size $\frac{n}{b}$ where $f(n)$ is the time to divide/combine

$$T(n) = aT(\frac{n}{b}) + f(n)$$

**Remarks:** e.g. Merge sort: $T(n) = 2T(\frac{n}{2}) + \Theta(n)$
$\star$ If $f(n) \in O(n)$, $\exists$ 2 constants $c > 0, n_0 > 0$ s.t. $f(n) \leq cn$ if $n \geq n_0$. For upper bound calculation, we can replace $\Theta(n)$ with $cn$. $\star$ Similarly for lower bound calculations if $f(n) \in \Omega(n)$, s.t. $f(n) \leq cn$ replace $\Theta(n)$ with $cn$

### Telescoping

- **Goal:** Solve the recurrence $T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + n & \text{if } n > 1 \end{cases}$

$$\frac{T(n)}{n} = \frac{2T(\frac{n}{2}) + n}{n}$$

$$\frac{T(n)}{n} = \frac{2}{n} T(\frac{n}{2}) + 1$$

$$\frac{T(n)}{n} = \frac{T(\frac{n}{2})}{n/2} + 1$$

$$\vdots$$

$$\frac{T(n)}{n} = \frac{T(1)}{1} + \underbrace{1+1+1+\ldots}_{\log n}$$

$$\frac{T(n)}{n} = T(1) + \log n$$

$$T(n) \in \Theta(n\log n)$$

$$\log n \left\{ \begin{aligned} \frac{T(n)}{n} &= \frac{T(\frac{n}{2})}{\frac{n}{2}} + 1 \\ \frac{T(\frac{n}{2})}{\frac{n}{2}} &= \frac{T(\frac{n}{4})}{\frac{n}{4}} + 1 \\ \frac{T(\frac{n}{4})}{\frac{n}{4}} &= \frac{T(\frac{n}{8})}{\frac{n}{8}} + 1 \\ &\vdots \\ \frac{T(2)}{2} &= \frac{T(1)}{1} + 1 \end{aligned} \right.$$

### Substitution

1. guess that $T(n) = O(f(n))$.
2. verify by induction:
   (a) to show that for $n \geq n_0$, $T(n) \leq c \cdot f(n)$
   (b) set $c = \max\{2, q\}$ and $n_0 = 1$
   (c) verify base case(s): $T(n_0) = q$
   (d) recursive case ($n > n_0$):
   - by strong induction, assume $T(k) \leq c \cdot f(k)$ for $n > k \geq n_0$
   - T(n) = $\langle$recurrence$\rangle$ ... $\leq c \cdot f(n)$

   (e) hence $T(n) = O(f(n))$.

**Example**

*Proof* $T(n) = \begin{cases} c & if\ n \leq 1 \\ 4T(\lfloor\frac{n}{2}\rfloor) + n & if\ n > 0 \end{cases}$

**Induction Hypothesis:** $T(n) \leq (c+1)n^2 - n$
**Base case:** $n = 1$
· If $n = 1$, then $T(n) = c = (c+1)n^2 - n$
**Inductive step:** $n \geq 2$

$$\begin{aligned} T(n) &= 4T(\lfloor\tfrac{n}{2}\rfloor) + n \\ &\leq 4(c+1)\lfloor\tfrac{n}{2}\rfloor^2 - 4\lfloor\tfrac{n}{2}\rfloor + n \\ &\leq 4(c+1)(\tfrac{n}{2})^2 - 4(\tfrac{n}{2}) + n \\ &= (c+1)n^2 + n \\ \therefore T(n) &\in O(n^2) \end{aligned}$$

### Recursion Tree

total = height $\times$ number of leaves

- each node represents cost of single subproblem
- height of tree = longest path from root to leaf

### Master Theorem

$a \geq 1, b > 1$, and $f$ is asymptotically positive

$T(n) = aT(\frac{n}{b}) + f(n) =$

$$\begin{cases} \Theta(n^{\log_b a}) & \text{if } f(n) < n^{\log_b a} \text{ polynomially} \\ \Theta(n^{\log_b a}\log^{k+1} n) & \text{if } f(n) = n^{\log_b a} \\ \Theta(f(n)) & \text{if } f(n) > n^{\log_b a} \text{ polynomially} \end{cases}$$

**three common cases**

1. If $f(n) \in O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$,
   - $f(n)$ grows polynomially slower than $n^{\log_b a}$ by $n^\epsilon$ factor.
   - then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) \in \Theta(n^{\log_b a}\log^k n)$ 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 $f(n) \in \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$,
   - 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 $n^\epsilon$ factor
   - then $T(n) = \Theta(f(n))$.

## 03. Iteration Recursion Divide and Conquer

### Iterative Algorithms

- *iterative* $\rightarrow$ loop(s), sequentially processing input elements
- **loop invariant** implies correctness if
  - *initialisation* - true before the first iteration of the loop
  - *maintenance* - if true before an iteration, it remains true at the beginning of the next iteration
  - *termination* - true when the algorithm terminates

**examples**

- **insertionSort**: with loop variable as $j$, $A[1..J-1]$ is sorted.
- **selectionSort**: with loop variable as $j$, the array $A[1..j-1]$ is sorted and contains the $j-1$ smallest elements of $A$.

### Divide-and-Conquer

**powering a number**

*problem:* compute $f(n, m) = a^n \pmod m$ for all $n, m \in \mathbb{Z}$

- observation: $f(x+y, m) = f(x, m) * f(y, m) \pmod m$
- **naive solution**: recursively compute and combine $f(n-1, m) * f(1, m) \pmod m$
  - $T(n) = T(n-1) + T(1) + \Theta(1) \Rightarrow T(n) = \Theta(n)$
- **better solution**: divide and conquer
  - divide: trivial
  - conquer: recursively compute $f(\lfloor n/2\rfloor, m)$
  - combine:
    - $f(n, m) = f(\lfloor n/2\rfloor, m)^2 \pmod m$ if n is even
    - $f(n, m) = f(1, m) * f(\lfloor n/2\rfloor, m)^2 \pmod m$ if odd
  - $T(n) = T(n/2) + \Theta(1) \Rightarrow \Theta(\log n)$

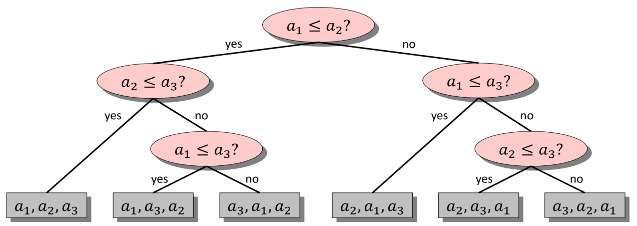
## 04. Comparison-Based Sorting

Comparison-based algorithms, elements can only be compared with one another: $<, \leq, =, >, \geq$. No other information can be used.

1. Insertion sort: $O(n^2)$
2. Selection sort: $O(n^2)$
3. Merge sort: $O(nlogn)$
4. Heap sort: $O(nlogn)$
5. Quick sort: $O(n^2)$

### Decision Trees

Worst-case running time $\geq$ worst-case number of comparisons = height of the tree



Worst-case time complexity of any comparison-based sorting algorithm: $\Omega(nlogn)$.

- Modelled as decision tree, each permutation = possible answer
- $n!$ leaves, height of binary tree is $\geq \log(n!)$
- Stirlings' approximation:
  $\log(n!) \in nlogn - nloge + O(logn) \subseteq \Omega(nlogn)$

### Quick Sort Average Case Analysis

Worst case $T(j-1) + T(n-j)$ time if **pivot** is the $j^{th}$ smallest element

- **Goal:** $T(n) \leq \max\limits_{j\in[n]}\{T(j-1) + T(n-j) + cn\} \triangleright T(n) \in \Theta(n^2)$
- Guess $T(r) \leq c_1 r^2$ and prove by induction
- **Base case:** $T(0) = 0$
- **Inductive step:** $n \geq 1$
  - $T(n) \leq \max\limits_{j\in[n]}\{T(j-1)T(n-j) + cn\}$
  - $T(n) \leq \max\limits_{j\in[n]}\{j^2 - 2j + 1 + n^2 + 2nj + j^2 + cn\}$
  - $T(n) \leq \max\limits_{j\in[n]}\{c_1(n^2 + 1 - 2j(n+1-j)) + cn\}$
  - $T(n) \leq c_1(n^2 - 2n + 1) + cn = c_1 n^2 + cn - c_1(2n-1) \leq c_1 n^2$

Average-case analysis

**Observation:** $A(n)$ is also the expected running time when the permutation $\pi$ is chosen uniformly at random.

- The **average-case** running time $A(n)$ is the average running time over all inputs of size $n$.
  Each permutation is chosen with a probability of $\frac{1}{n!}$.

$$A(n) = \sum_\pi \frac{1}{n!} \cdot (\text{running time of quick sort on } \pi)$$

The summation is over all permutations $\pi$ of $(a_1, a_2, \ldots, a_n)$.

The execution of the quick sort algorithm:
- It depends only on $\pi$.
- It is independent of the actual values of $(a_1, a_2, \ldots, a_n)$.

**Uniformity**

Suppose input permutation $\pi$ of $a_1, a_2, \ldots, a_n$ is uniformly random.

- **Observation 1: pivot** is selected uniformly at random
  - $\forall(j \in [n]), \Pr(\textbf{pivot} = a_j) = \frac{1}{n}$
- **Observation 2:** Permuations for both recursive calls are uniformly random

- Recursive call on $A_S$: Each permuation of $(a_1, a_2, \dots, a_{j-1})$ appears with equal probability
- Recursive call on $A_L$: Each permuation of $(a_{j+1}, a_{j+2}, \dots, a_n)$ appears with equal probability

- **Reason:** if **pivot** $= a_j$ then partition algorithm never compares any 2 elements $(a_1, a_2, \dots, a_{j-1})$
- At start, input $\pi$ restricted to $(a_1, a_2, \dots, a_{j-1})$ is uniformly random $\triangleright$ at the end permuation of $(a_1, a_2, \dots, a_{j-1})$ is still uniformly random

**Recurrence**

- $A(n) = \frac{1}{n} \cdot \sum_{j=1}^{n} [A(j-1) + A(n-j) + cn] = cn + \frac{2}{n} \cdot \sum_{j=0}^{n-1} A(j)$

$\downarrow$ $A(n) \to n \cdot A(n) = cn^2 + 2\Sigma_{j=0}^{n-1} A(j)$
$\cdot$ $(n-1) \cdot A(n-1) = c(n-1)^2 + 2\Sigma_{j=0}^{n-1} A(j)$

$\downarrow$ $n \cdot A(n) - (n-1) \cdot A(n-1) = c(2n-1) + 2A(n-1)$
$\cdot$ $n \cdot A(n) - (n+1) \cdot A(n-1) =$
$(n \cdot A(n) - (n-1) \cdot (A(n-1)) - 2A(n-1)) = c(2n-1)$

$\downarrow$ dividing by $n(n+1)$: $\frac{A(n)}{n+1} - \frac{A(n-1)}{n} = \frac{c(2n-1)}{n(n+1)} < \frac{c(2n+2)}{n(n+1)} = \frac{2c}{n}$
Can also be solved by taking area under integration $\approx O(n \log n)$

- after telescoping: $\frac{A(n)}{n+1} < 2c \cdot (\frac{1}{n} + \frac{1}{n-1} + \dots + \frac{1}{2}) + \frac{A(1)}{2}$
- $\therefore A(n) \in O(n \log n)$

**Types of sorting**

- **Stable:** element of equal values, original orderign preserved (insertion)
- **In-place:** uses *little extra memory* besides input array

## 5. Randomized Algorithms

Utilize randomization to develop algorithms that are more efficient or simpler than deterministic counterparts $\to$ cost of allowing **small error probability**

### Freivald's Algorithm

- Choose $v = \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix}$ to be a uniformly random column vector from $0, 1^n$
- If $ABc = Cv$ then output $AB = C$ otherwise they are not equal

**Freivald's Algorithm — when $AB \neq C$**

- $C^* = \begin{pmatrix} c^*_{1,1} & \dots & c^*_{1,n} \\ \vdots & \ddots & \vdots \\ c^*_{n,1} & \dots & c^*_{n,n} \end{pmatrix}$
- $u = \begin{pmatrix} u_1 \\ \vdots \\ u_n \end{pmatrix} = C^* v$
- Since $AB \neq C$, there exists $(i, j)$ s.t. $c^{i,j} \neq 0$
- $u_1 = c^*_{i,1} v_1 + c^*_{i,2} v_2 + \dots c^*_{i,j} v_k + \dots c^*_{i,n} v_n = (\dots) + c^*_{i,j} v_j$
- Reveal random numbers $\{v_1, v_2, \dots, v_n\} \setminus \{v_j\}$, this term is **fixed**
- After fixing this term, there is **exactly one choice of** $v_j$ that makes $u_i = 0$
- $\therefore \Pr[u_i \neq 0] \geq \frac{1}{2} \Rightarrow \Pr[successful\ algorithm] \geq \frac{1}{2}$

**Principle of deffered decision**

- If we can show that $\Pr[\epsilon | X = x] \geq p$ for every $x$, then $\Pr[\epsilon] \geq p$
- $\Pr[\epsilon] = \Sigma_x \Pr[\epsilon | X = x] \cdot \Pr[X = x] \geq p \cdot \Sigma_x \Pr[X = x] = p$

**Success Probability Amplification**

We only show that Freivalds' algorithm is **incorrect** with probability $\leq \frac{1}{2}$

- **Claim:** error probability can be reduced to **at most f** by repeating the algorithm for $t = \lceil \log \frac{1}{f} \rceil$ times
  - If all $t$ outputs are $AB = C$ return $AB = C$, else return $AB \neq C$
    - If $AB = C$, Freivalds' algorithm always answers $AB = C$ correctly
    - If $AB \neq C$ probability Freivald's algorithm answers $AB = C$ for all $t = \lceil \log \frac{1}{f} \rceil$ iterations is at most $\frac{1}{2^t} \leq f$

### Balls and Bins

Throw $m$ balls into $n$ bins randomly + independently, what is probability every bin contains $\geq 1$ ball. $\to$ Pr. that bin contains zero balls is $(1 - \frac{1}{n})^m \leq e^{-\frac{m}{n}}$

- **Union bound:** Pr. $\geq 1$ bin contains 0 balls is **at most** $(1 - \frac{1}{n})^m \leq ne^{-\frac{m}{n}}$
- **Useful inequality:** $1 + x \leq e^x$ — Pr. is at most $\frac{1}{n}$ if $m \geq 2n \lceil \ln n \rceil$

**$n$ balls uniformly and independently to $n$ bins**

- Expected fraction of bins with exactly 3 balls converge to as $n \to \infty$
- $\binom{n}{c} (\frac{1}{n})^3 (\frac{n-1}{n})^{n-3} = \frac{n(n-2)}{6(n-1)^2} \cdot (1 - \frac{1}{n})^n$
- $\lim_{n \to \infty} \frac{n(n-2)}{6(n-1)^2} = \frac{1}{6} \& \lim_{n \to \infty} (1 - \frac{1}{n})^2 = \frac{1}{e}$

**Coupon Collector Problem**

*$n$ types of coupon put into a box, randomly drawn with replacement. What is the expected number of draws needed to collect $\geq 1$ of each type of coupon?*

- let $T_i$ be time to collect $i$-th coupon after $i - 1$ coupon has been collected.
  - Probability of collecting a new coupon, $p_i = \frac{(n-(i-1))}{n}$
  - $T_i$ has a **geometric distribution**, $E[T_i] = 1/p_i$
- total number of draws, $T = \sum_{i=1}^{n} T_i$, $E[T] = E[\sum_{i=1}^{n} T_i] = \sum_{i=1}^{n} E[T_i]$ by linearity of expectation $= \sum_{i=1}^{n} \frac{n}{n-(i-1)} = n \cdot \sum_{i=1}^{n} \frac{1}{i} = \Theta(n \lg n)$

**Techniques**

- **Markov Inequality:** $X$ is a non-negative random variable and $a > 0$ then $\Pr[X \geq a \cdot \mathbb{E}[X]] \leq \frac{1}{a}$
- **Linearity of Expecation:** if $A + B$ then $\mathbb{E}[X] = \mathbb{E}[A] + \mathbb{E}[B]$ generally — if $X = \Sigma_{i=1}^{n} X_i$ then $\mathbb{E}[X] = \Sigma_{i=1}^{n} \mathbb{E}[X_i]$
- **Indicator random variables:** Let $\epsilon$ be event, indicator that $\mathbb{1}_\epsilon$ for $\epsilon$ defined:
  - $\mathbb{1}_{\mathcal{E}} = \begin{cases} 1, & \text{if } \mathcal{E} \text{ occurs,} \\ 0, & \text{otherwise.} \end{cases}$

**Hashing**

Hash table $A$ is an array of length $n$, with $h$ as a mapping from some universe $U$ to indices in array $\{1, 2, \cdots, n\}$

- `insert(v)` if `v` not in `A[h(v)]`, store `v` in `A[h(v)]` — `search(v)` check if `v` is in `A[h(v)]` — `delete(v)` if `v` is in `A[h(v)]` remove `v` from `A[h(v)]`

**Randomised quicksort**

Before a number in $a_i, \dots a_j$ is selected as a pivot, number in $a_i, \dots a_j$ must belong to the same array. First number chosen as a pivot in $a_i, \dots a_j$

1. $a_i$ or $a_j$: algorithm will compare the pivot with all other numbers in the current array $\to \epsilon_{i,j}$ occurs
2. Not $a_i$ or $a_j$: $a_i$ and $a_j$ belong to different arrays in recursive calls $\to \epsilon_{i,j}$ does not occur
3. A comparison is made between $a_i$ and $a_j$ *iff* first number chosen as pivot $(a_i, \dots, a_j)$ is $a_i$ or $a_j$

- Claim: For any $1 \leq i \leq j \leq n$, $\Pr[\epsilon_{i,j}] = \frac{2}{j-i+1}$
- where $\epsilon_{i,j}$ is a comparison that is made between $a_i$ and $a_j$
- $\mathbb{E}[\text{number of comparisons}] = \sum_{1 \leq i < j \leq n} \mathbb{E}[X_{i,j}] = \sum_{1 \leq i < j \leq n} \frac{2}{j-i+1} = 2 \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{1}{j-i+1} = 2 \sum_{i=1}^{n} \left( \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n-i+1} \right) \in \mathcal{O}(n \log n)$
- By applying *Markov inequality*, the expected running time can be concluded that the randomised quicksort finishes in $P(n \log n)$ time with $\Pr \geq 0.99$

**Types of Randomised Algorithms**

- randomised **Las Vegas** algorithms
  - output is always correct
  - runtime is a *random variable*
  - e.g. randomised quicksort/select
- randomised **Monte Carlo** algorithms
  - output may be incorrect with some small probability
  - runtime is *deterministic*

## 06. Dynamic Programming

- **cut-and-paste proof** 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)
- **overlapping subproblems** - recursive solution contains a small number of distinct subproblems repeated many times

### Longest Common Subsequence

- for sequence $A : a_1, a_2, \dots, a_n$ stored in array
- $C$ is a **subsequence** of $A \to$ if we can obtain $C$ by removing zero 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 $B$.

**Brute force solution**

- check *all* possible subsequences of $A$ to see if it is also a subsequence of $B$, then output the longest one — analysis: $O(m2^n)$
  - checking each subsequence takes $O(m) \to 2^n$ possible subsequences

**Recursive solution**

let $LCS(i, j)$: longest common subsequence of $A[1..i]$ and $B[1..j]$

- base case: $LCS(i, 0) = \emptyset$ for all $i$, $LCS(0, j) = \emptyset$ for all $j$
- general case:
  - if last characters of $A, B$ are $a_n = b_m$, then $LCS(n, m)$ must terminate with $a_n = b_m$
    - the optimal solution will match $a_n$ with $b_m$
  - if $a_n \neq b_m$, then either $a_n$ or $b_m$ is not the last symbol
- **optimal substructure**: (general case)
  - if $a_n = b_m$, $LCS(n, m) = LCS(n-1, m-1) :: a_n$
  - if $a_n \neq b_m$, $LCS(n, m) = LCS(n-1, m) \,||\, LCS(n, m-1)$
- **simplified problem**:
  - $L(n, m) = 0$ if $n = 0$ or $m = 0$
  - if $a_n = b_m$, then $L(n, m) = L(n-1, m-1) + 1$
  - if $a_n \neq b_m$, then $L(n, m) = \max(L(n, m-1), L(n-1, m))$

  **analysis**
  - number of distinct subproblems $= (n+1) \times (m+1)$
  - to use $O(\min\{m, n\})$ space: bottom-up approach, column by column
  - memoize for DP $\Rightarrow$ makes it $O(mn)$ instead of exponential time

### Knapsack Problem

- input: $(w_1, v_1), (w_2, v_2), \dots, (w_n, v_n)$ and capacity $W$
- output: subset $S \subseteq \{1, 2, \dots, n\}$ that maximises $\sum_{i \in S} v_i$ such that $\sum_{i \in S} w_i \leq W$
- $2^n$ subsets $\Rightarrow$ naive algorithm is costly
- **recursive solution**:
  - let $m[i, j]$ be the maximum value that can be obtained using a subset of items $\{1, 2, \dots, i\}$ with total weight no more than $j$.
  - $m[i, j] = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0 \\ \max\{m[i-1, j-w_i] + v_i, m[i-1, j]\}, & \text{if } w_i \leq j \\ m[i-1, j], & \text{otherwise} \end{cases}$
- **analysis**: $O(nW)$
  - $O(nW)$ is **not** a polynomial time algorithm
  - not polynomial in input bitsize
    - $W$ can be represented in $O(\lg W)$ bits
    - $n$ can be represented in $O(\lg n)$ bits
  - polynomial time is strictly in terms of the number of bits for the input

### Changing Coins

**problem**: use the fewest number of coins to make up $n$ cents using denominations $d_1, d_2, \dots, d_n$. Let $M[j]$ be the fewest number of coins needed to change $j$ cents.

- **optimal substructure**:
  - $M[j] = \begin{cases} 1 + \min\limits_{i \in [k]} M[j - d_i], & j > 0 \\ 0, & j = 0 \\ \infty, & j < 0 \end{cases}$

  *Proof.* Suppose $M[j] = t$, meaning $j = d_{i_1} + d_{i_2} + \dots + d_{i_t}$ for some $i_1, \dots, i_t \in \{1, \dots, k\}$.
  Then, if $j' = d_{i_1} + d_{i_2} + \dots + d_{i_t - 1}$, $M[j'] = t - 1$, because otherwise if $M[j'] < t - 1$, by **cut-and-paste** argument, $M[j] < t$.

- runtime: $O(nk)$ for $n$ cents, $k$ denominations

## 07. GREEDY ALGORITHMS

- solve only one subproblem at each step
- beats DP and divide-and-conquer when it works
- **greedy-choice property** $\to$ a locally optimal choice is globally optimal

**Fractional Knapsack** — $O(n \log n)$

- **greedy-choice property**: let $j^*$ be item with *maximum* value/kg, $v_j/w_i$, then $\exists$ an optimal knapsack containing $\min(w_{j^*}, W)$ kg of item $j^*$.
- **optimal substructure**: if we remove $w$ kg of item $j$ from the optimal knapsack, then the remaining load must be the optimal knapsack weighing at most $W - w$ kgs that one can take from $n-1$ original items and $w_j - w$ kg of item $j$.

*Proof.* cut-and-paste argument

Suppose the remaining load after removing $w$ kgs of item $j$ was *not* the optimal knapsack weighing ...

Then there is a knapsack of value $> X - v_j \cdot \frac{w}{w_j}$ with weight ...

Combining this knapsack with $w$ kg of item $j$ gives a knapsack of value $> X \Rightarrow$ contradiction!

**Minimum Spanning Trees**

for a connected, undirected graph $G = (V, E)$, find a spanning tree $T$ that connects all vertices with minimum weight. Weight of spanning tree $T$, $w(T) = \sum_{(u,v)\in T} w(u, v)$.

- **optimal substructure**: let $T$ be a MST. remove any edge $(u, v) \in T$. then $T$ is partitioned into $T_1, T_2$ which are MSTs of $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$.

  *Proof.* cut-and-paste: $w(T) = w(u, v) + w(T_1) + w(T_2)$

  if $w(T_1') < w(T_1)$ for $G_1$, then $T' = \{(u, v)\} \cup T_1' \cup T_2$ would be a lower-weight spanning tree than $T$ for $G$.

  $\Rightarrow$ contradiction, $T$ is the MST

- **Prim's algorithm** - at each step, add the least-weight edge from the tree to some vertex outside the tree
- **Kruskal's algorithm** - at each step, add the least-weight edge that does *not* cause a cycle to form

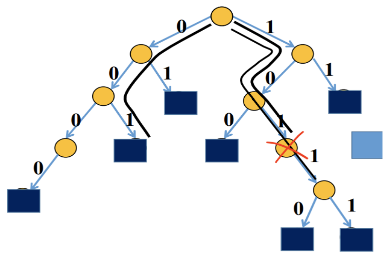
## Binary Coding

Given an alphabet set $A : \{a_1, a_2, \dots, a_n\}$ and a text file $F$ (sequence of alphabets), how many bits are needed to encode a text file with $m$ characters?

- **fixed length encoding**: $m \cdot \lceil \log_2 n \rceil$
  - encode each alphabet to unique binary string of length $\lceil \log_2 n \rceil$
  - total bits needed for $m$ characters $= m \cdot \lceil \log_2 n \rceil$
- **variable length encoding**
  - different characters occur with different frequency $\Rightarrow$ use fewer bits for *more frequent* alphabets
  - average bit length, $ABL(\gamma) = \sum_{x\in A} f(x) \cdot |\gamma(x)|$
  - BUT overlapping prefixes cause indistinguishable characters

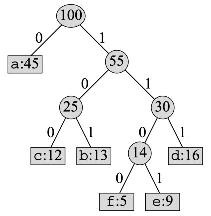
**Prefix encoding**

- a coding $\gamma(A)$ is a **prefix coding** if $\nexists x, y \in A$ s.t. $\gamma(x)$ is a prefix of $\gamma(y)$.
- **labelled binary tree**: $\gamma(A)$ = label of path from root



- for each prefix code $A$ of $n$ alphabets, there exists a binary tree $T$ on $n$ leaves s.t. there is a **bijective mapping** between the alphabets and the leaves
- $OPT_{ABL}(A) = OPT_{ABL}(A') + f(a_1) + f(a_2)$
- $ABL(\gamma) = \sum_{x\in A} f(x) \cdot |\gamma(x)| = \sum_{x\in A} f(x) \cdot |depth_T(x)|$
- binary tree which is an *optimal* prefix coding must be **full binary tree**.
  - every internal node has degree exactly 2
  - multiple possible optimal trees - most optimal depends on alphabet frequencies
- accounting for alphabet **frequencies**:
  - let $a_1, a_2, \dots, a_n$ be alphabets of $A$ in non-decreasing order of freq.
  - $a_1$ must be a leaf node; $a_2$ *can* be a sibling of $a_1$
  - there exists an optimal prefix coding in which $a_1$ and $a_2$ are siblings
- derivation of optimal prefix coding: **Huffman's algorithm**
  - keep merging the two least frequent items to create a new alphabet $a'$
  - remove $a_1$ $a_2$ and replace with $a'$

```
Huffman(C):
  Q = new PriorityQueue(C)
  while Q:
    allocate a new node z
    z.left = x = extractMin(Q)
    z.right = y = extractMin(Q)
    z.val = x.val + y.val
    Q.add(z)
  return extractMin(Q) // root
```



# 08. Amortized Analysis

- **amortized analysis** $\rightarrow$ guarantees the *average* performance of each operation in the *worst case*.
- total amortized cost provides an *upper bound* on the total true cost
- For a sequence of $n$ operations $o_1, o_2, \dots, o_n$,
  - let $t(i)$ be the time complexity of the $i$-th operation $o_i$
  - let $f(n)$ be the *worst-case* time complexity for *any* of the $n$ operations
  - let $T(n)$ be the time complexity of all $n$ operations

$$T(n) = \sum_{i=1}^{n} t(i) = nf(n)$$

## Types of Amortized Analysis

**Aggregate method**

- look at the whole sequence, sum up the cost of operations and take the average - simpler but less precise
- e.g. binary counter - amortized $O(1)$
- e.g. queues (with `INSERT` and `EMPTY`) - amortized $O(1)$

**Accounting method**

- charge the $i$-th operation a fictitious amortized cost $c(i)$
  - **amortized cost** $c(i)$ is a fixed cost for each operation
  - **true cost** $t(i)$ depends on when the operation is called
- amortized cost $c(i)$ must satisfy:

$$\sum_{i=1}^{n} t(i) \le \sum_{i=1}^{n} c(i) \text{ for all } n$$

- take the extra amount for cheap operations early on as "credit" paid in advance for expensive operations
  - **invariant**: bank balance never drops below 0
- the total amortized cost provides an **upper bound** on the total true cost

**Potential method**

- $\phi$ : potential function associated with the algo/DS
- $\phi(i)$: potential at the end of the $i$-th operation
- $c_i$ : amortized cost of the $i$-th operation
- $t_i$ : true cost of the $i$-th operation

$$c_i = t_i + \phi(i) - \phi(i-1)$$
$$\sum_{i=1}^{n} c_i = \phi(n) - \phi(0) + \sum_{i=1}^{n} t_i$$

- hence as long as $\phi(n) \ge 0$, then amortized cost is an upper bound of the true cost. Generally in expensive operation, we want to see if there is a quantity *decreasing* during this operation

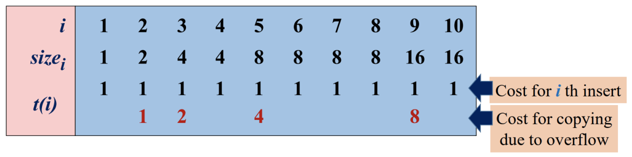
$$\sum_{i=1}^{n} c_i \ge \sum_{i=1}^{n} t_i$$

- usually take $\phi(0) = 0$
- **e.g.** for queue:
  - let $\phi(i) = \#$ of elements in queue after the $i$-th operation
  - amortized cost for insert: $c_i = t_i + \phi(i) - \phi(i-1) = 1 + 1 = 2$
  - amortized cost for empty (for $k$ elements): $c_i = t_i + \phi(i) - \phi(i-1) = k + 0 - k = 0$
- try to keep $c(i)$ small: using $c(i) = t(i) + \Delta\phi_i$
  - if $t(i)$ is small, we want $\Delta\phi_i$ to be positive and small
  - if $t(i)$ is large, we want $\Delta\phi_i$ to be negative and large

**e.g. Dynamic Table (insertion only)**

**Aggregate method**

cost of $n$ insertions $= \sum_{i=1}^{n} t(i) \le n + \sum_{j=1}^{\lfloor \log(n-1) \rfloor} 2^j \le 3n$



| *i* | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| *size<sub>i</sub>* | 1 | 2 | 4 | 4 | 8 | 8 | 8 | 8 | 16 | 16 |
| *t(i)* | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
|  |  | 1 | 2 |  | 4 |  |  |  | 8 |  |

Cost for $i$ th insert

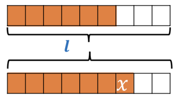
Cost for copying due to overflow

**Accounting method**

- charge \$3 per insertion
  - \$1 for insertion itself
  - \$1 for moving itself when the table expands
  - \$1 for moving one of the existing items when the table expands

**Potential method**

let $\phi(i) = 2i - size(T)$



2(i-1) - (i-1) =

$\phi(i\text{-}1) = 2(i-1) - l$

Before **Insert**($x$)

2i - 2(i-1) =

$\phi(i) = 2i - l$

After **Insert**($x$)

| Operation **Insert**($x$) | Actual Cost | $\Delta\phi_i$ | Amortized Cost |
|---|---|---|---|
| **Case 1**: when table is not full | 1 | **2** | **3** |
| **Case 2**: when table is already full | $i$ | $3 - i$ | **3** |

Amortized cost of n insertions $= 3n = O(n)$

Actual cost of n insertions $= O(n)$

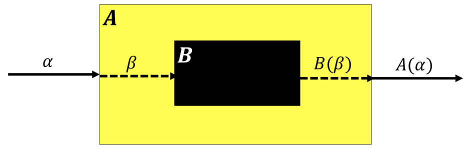
- show that SUM of amortized cost $\ge$ SUM of actual cost
- conclude that sum of amortized cost is $O(f(n)) \Rightarrow$ sum of actual cost is $O(f(n))$

# 9. REDUCTIONS & INTRACTABILITY

## Reduction

Consider two problems $A$ and $B$, $A$ can be solved as follows:

1. convert instance $\alpha$ of $A$ to an instance of $\beta$ in $B$
2. solve $\beta$ to obtain a solution
3. based on the solution of $\beta$, obtain the solution of $\alpha$.
4. $\Rightarrow$ then we say $A$ **reduces** $B$.



**instance** $\rightarrow$ another word for input

**e.g. Matrix Multiplication & Squaring**

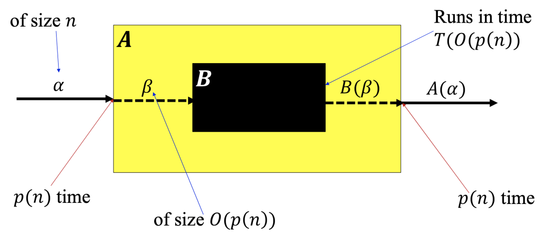
- Mat-Multi: matrix multiplication
  - *input*: two $N \times N$ matrices $A$ and $B$.
  - *output*: $A \times B$
- Mat-Sqr: matrix squaring
  - *input*: one $N \times N$ matrix $C$. *output*: $C \times C$
- Mat-Sqr can be reduced to Mat-Multi
  - *Proof.* Given input matrix $C$ for Mat-Sqr, let $A = C$ and $B = C$ be inputs for Mat-Multi. Then $AB = C^2$.
- Mat-Multi can also be reduced to Mat-Sqr!
  - *Proof.* let $C = \begin{bmatrix} 0 & A \\ B & 0 \end{bmatrix} \Rightarrow C^2 = \begin{bmatrix} 0 & A \\ B & 0 \end{bmatrix}\begin{bmatrix} 0 & A \\ B & 0 \end{bmatrix} = \begin{bmatrix} AB & 0 \\ 0 & BA \end{bmatrix}$

**T-Sum**

- 0-Sum: given array $A$, output $i, j \in (1, n)$ such that $A[i] + A[j] = 0$
- T-Sum: given array $B$, output $i, j \in (1, n)$ such that $B[i] + B[j] = T$
- **reduce T-Sum to 0-Sum**:
  - given array $B$, define array $A$ s.t. $A[i] = B[i] - T/2$.
  - if $i, j$ satisfy $A[i] + A[j] = 0$, then $B[i] + B[j] = T$.

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

- **$p(n)$-time Reduction** $\rightarrow$ if for any instance $\alpha$ of problem $A$ of size $n$,
  - an instance $\beta$ for $B$ can be constructed in $p(n)$ time
  - a solution to problem $A$ for input $\alpha$ can be recovered from a solution to problem $B$ for input $\beta$ in time $p(n)$.
- $n$ **is in bits!**
- if there is a $p(n)$-time reduction from problem $A$ to $B$ and a $T(n)$-time algorithm to solve problem $B$, then there is a $T(O(p(n))) + O(p(n))$ time algorithm to solve $A$.



- $A \leq_P B$ $\rightarrow$ if there is a $p(n)$-time reduction from $A$ to $B$ for some polynomial function $p(n) = O(n^c)$ for some constant $c$. ("$A$ is a special case of $B$")
  - if $B$ has a polynomial time algorithm, then so does $A$
  - "polynomial time" $\approx$ reasonably efficient
- $A \leq_P B, B \leq_P C \Rightarrow A \leq_P C$

## Polynomial Time

- **polynomial time** $\rightarrow$ runtime is polynomial in the **length of the encoding** of the problem instance
- **"standard" encodings**
  - binary encoding of integers
  - list of parameters enclosed in braces (graphs/matrices)
- **pseudo-polynomial** algorithm $\rightarrow$ runs in time polynomial in the **numeric value** if the input but is **exponential** in the **length** of the input
  - e.g. DP algo for Knapsack since $W$ is in numeric value
- Knapsack is NOT polynomial time: $O(nW \log M)$ but $W$ is not the number of bits
- Fractional Knapsack is polynomial time: $O(n \log n \log W \log M)$

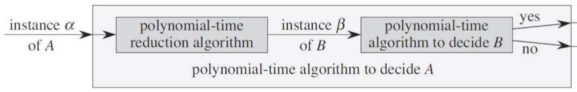
## Decision Problems

- **decision problem** $\rightarrow$ a function that maps an instance space $I$ to the solution set $\{YES, NO\}$
- decision vs optimisation problem:
  - **decision problem**: given a directed graph $G$, *is there* a path from vertex $u$ to $v$ of length $\leq k$?
  - **optimisation problem**: given ..., what is *length* of the shortest path ... ?
  - convert from **decision** $\rightarrow$ **optimisation**: given an instance of the optimisation problem and a number $k$, is there a solution with value $\leq k$?
- the decision problem is *no harder than* the optimisation problem.
  - given the optimal solution, check that it is $\leq k$.
  - if we cannot solve the decision problem quickly $\Rightarrow$ then we cannot solve the optimisation problem quickly
- decision $\leq_P$ optimisation

## Reductions between Decision Problems

**Karp reduction:** 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 $\alpha$ of $A$ and $\beta$ of $B$ such that

1. $\alpha$ is a *YES*-instance of $A$ $\iff$ $(iff)$ $\beta$ is a *YES*-instance of $B$
2. the transformation takes polynomial time in the size of $\alpha$



instance $\alpha$ of $A$
polynomial-time reduction algorithm
instance $\beta$ of $B$
polynomial-time algorithm to decide $B$
yes
no
yes
no
polynomial-time algorithm to decide $A$

**Examples**

- Independent-Set: given a graph $G = (V, E)$ and an integer $k$, is there a subset of $\leq k$ 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 to *at least one* vertex in this subset?
- Independent-Set $\leq_P$ Vertex-Cover
  - *Reduction*: to check whether $G$ has an independent set of size $k$, we check whether $G$ has vertex cover of size $n - k$.

*Proof.* If Independent-Set, then Vertex-Cover.

Suppose $(G, k)$ is a *YES*-instance of Indep-Set. Then there is subset $S$ of size $\geq k$ that is an independent set.

$V - S$ is a vertex cover of size $\leq n - k$. Proof: Let $(u, v) \in E$. Then $u \notin S$ or $v \notin S$.

So either $u$ or $v$ is in $V - S$, the vertex cover.

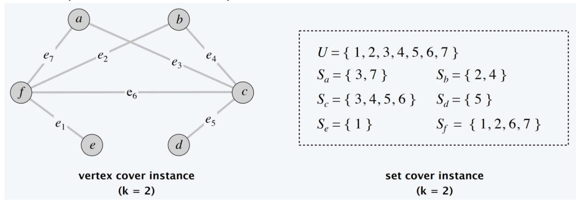
*Proof.* If Vertex-Cover, then Independent-Set.

Same as above, but flip IS and VC

**e.g. Set-Cover**

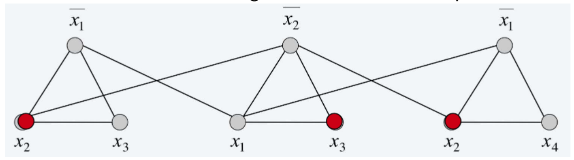
Given integers $k$ and $n$, and collection $\mathcal{S}$ of subsets of $\{1, \ldots, n\}$, are there $\leq k$ of these subsets whose union equals $\{1, \ldots, n\}$?
Claim: **Vertex-Cover** $\leq_P$ **Set-Cover**
*Reduction*: given $(G, k)$ instance of Vertex-Cover, generate an instance $(n, k', \mathcal{S})$ of Set-Cover.

*Proof.* For each node $v$ in $G$, construct a set $S_v$ containing all its outgoing edges. (Number each edge)



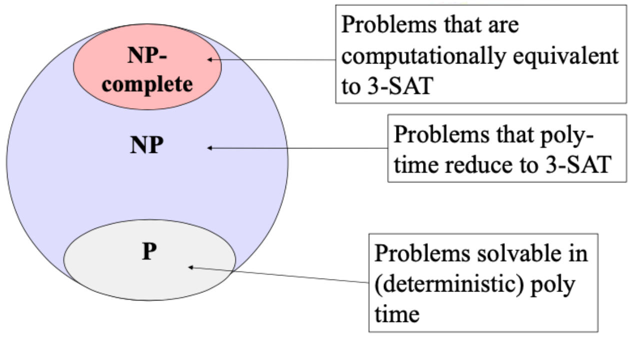
**e.g. 3-SAT**

- **SAT**: given a CNF formula $\Phi$, does it have a satisfying truth assignment?
  - literal: a boolean variable or its negation $x, \bar{x}$
  - clause: a disjunction (OR) of literals
  - conjunctive normal form (CNF): formula $\Phi$ that is a conjunction (AND) of clauses
- **3-SAT** $\rightarrow$ SAT where each clause contains exactly 3 literals
- **3-SAT** $\leq_P$ **Independent-Set**
  - *Reduction*: Construct an instance $(G, k)$ of Indep-Set s.t. $G$ has an independent set of size $k$ $\iff$ $\Phi$ is satisfiable
    - node: each literal term
    - edge: connect 3 literals in a clause in a triangle
    - edge: connect literal to all its negations
    - reduction runs in polynomial time
  - $\Rightarrow$ for $k$ clauses, connecting $k$ vertices form an independent set in $G$.



# 10. NP-COMPLETENESS

- **P** $\rightarrow$ the class of *decision* problems solvable in (deterministic) polynomial time
- **NP** $\rightarrow$ the class of *decision* problems for which polynomial-time verifiable **certificates** of YES-instances exist.
  - aka *non-deterministic polynomial*
  - i.e. no poly-time algo, but verification can be poly-time
  - **certificate** $\rightarrow$ result that can be checked in poly-time to verify correctness
- $P \subseteq NP$: any problem in **P** is in **NP**.
  - if $P = NP$, then all these algos can be solved in poly time



NP-complete
NP
P
Problems that are computationally equivalent to 3-SAT
Problems that poly-time reduce to 3-SAT
Problems solvable in (deterministic) poly time

## NP-Hard and NP-Complete

- a problem $A$ is said to be **NP-Hard** if for *every* problem $B \in NP$, $B \leq_P A$.
  - aka $A$ 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**
  - aka the hardest problems in NP.
- **Cook-Levin Theorem** $\rightarrow$ every problem in NP-Hard can be poly-time *reduced* to 3-SAT. Hence, **3-SAT is NP-Hard and NP-Complete**.
- NP-Complete problems can still be approximated in poly-time! (e.g. greedy algorithm gives a 2-approximation for Vertex-Cover)

## showing NP-Completeness

1. show that $X$ is in NP. $\Rightarrow$ a YES-instance has a certificate that can be verified in polynomial time
2. show that $X$ is NP-hard
   - by giving a poly-time reduction from another NP-hard problem $A$ to $X$. $\Rightarrow$ $X$ is at least as hard as $A$
   - reduction should *not* depend on whether the instance of $A$ is a YES- or NO-instance
3. show that the reduction is valid
   (a) reduction runs in poly time
   (b) if the instance of $A$ is a YES-instance, then the instance of $X$ is also a YES-instance
   (c) if the instance of $A$ is a NO-instance, then the instance of $X$ is also a NO-instance

```
def INDEPENDENT-SET(G, k) -> bool:
    G', k' = reduction(G, k)
    yes_or_no: bool = CLIQUE(G', k') # magically given
    return yes_or_no
```

What to show for a correct reduction:

- `(G, k)` is YES-instance → `(G', k')` is also a YES-instance
- `(G', k')` is YES-instance → `(G, k)` is also a YES-instance
- The transformation takes polynomial time in the size of `(G, k)`

## showing NP-HARD

1. take any **NP-Complete** problem $A$
2. show that $A \leq_P X$

## 11. Linear Time Sorting and Selection

### Direct Addressing Table

```
C = [0]*(k+1) # create the frequency DAT [0..k], O(k)
for Ai in A: # O(n), at the end, C[i] = frequency of i
  C[Ai] += 1
for i in range(k+1): # O(k+n)
  print(*[i] * C[i], end'= ') # set format yourself
```

- No comparisons between element $\rightarrow$ not comparison based sorting
- Trade-off *memory* (DAT) to break $\Omega(n \log n)$ lower bound
- If $k \in O(n)$ then $O(n+k)$ counting sort takes $O(n)$ time
- It requires **big memory** for the frequency counter DAT C. $\Rightarrow$ If k is big, we may not be able to create a DAT C to cover [0..k]
- We want to set k big enough (vs the available working memory), but not too big that we cannot even run the algorithm.

### Counting Sort Stable Version

```
C = [0]*(k+1) # create the frequency DAT [0..k], O(k)
for Ai in A: # O(n), at the end, C[i] = freq. of i
  C[Ai] += 1
for i in range(1, k+1): # O(k), C[i] = num ints <= i
  C[i] += C[i-1] # aka the 'prefix 'sum
  B = [0] * n # output (sorted) array
for i in range(n-1, -1, -1): # O(n), from B2F
  C[A[i]] -= 1 # decrease first (0-based indexing)
  B[C[A[i]]] = A[i] # A[i] is placed at C[A[i]] in B
print(*B) # B is the sorted version of A, and stable
```

### Radix Sort - Iterated Counting Sort

```
for i <- 1 to d # note: LSD is from right to left
  sort by the i-th LSD #(using stable Counting Sort)
```

- We can use **proof by induction**: When we have sorted the $i^{th}$ LSD, then the integers are sorted according to their values on the $i^{th}$ LSD.
- $P(1)$ holds, as Radix Sort first pass will correctly sort the LSD (we are sure the stable Counting Sort sub-routine is correct).
- Suppose $P(i-1)$ holds, we can show $P(i)$ holds too:
  - The integers are sorted based on the $i^{th}$ LSD on the $i^{th}$ pass.
  - Now, due to the stable sort subroutine used, within the group of integers that have the same $i^{th}$ LSD, the stable sort algorithm does not change their relative position! $\Rightarrow$ Thus, these integers are correctly sorted!

### Radix Sort - Analysis

- If we have $d$ digits, Radix Sort runs $d$ iterations of $O(n+k)$ Counting Sort, thus the overall time complexity in $O(d \cdot (n+k))$.
- Setting $k = 10$ (digit $[0..9]$, or base $10$/Decimal), is often **not** the best setup.
- If $b$-bit word is broken in to $\frac{b}{r}$ groups of $r$-bit words:
  - There are only $d = \frac{b}{r}$ passes now
  - Each pass takes $O(n+2^r)$ as $k = 2^r$
  - Total time is thus $O(\frac{b}{r}(n+2^r))$, choose $r$ to minimize $O(\frac{b}{r}(n+2^r))$
  - Optimal $r \approx \log_2 n$ so that $(n+2^r)$ balances
  - Total time is thus $O(\frac{b}{\log_2 n} \cdot (n + 2^{\log_2 n})) = O(\frac{b \cdot n}{\log_2 n})$
  - Now if the integers are in the range of $[0..n^d]$, then $b = \log_2 n^d$ bits or $b = d \cdot \log_2 n$ bits
  - Thus Radix Sort runs in time $O(\frac{d \cdot \log_2 n \cdot n}{\log_2 n}) = O(d \cdot n)$ — linear time

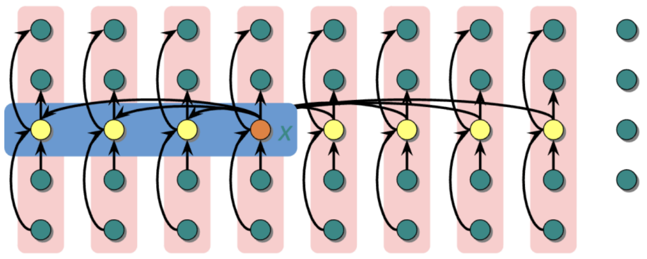
### Quickselect - Pseudocode

```
function QuickSelect(A, l, r, i)
  if l == r, return l
  k = Randomized Partition(A, l, r) // random pivot
  if i == k, then return k // found
  else if i < k, then return QuickSelect(A, l, k-1, i)
  else, then return QuickSelect(A, k+1, r, i)
```

- Best case for QuickSelect is $O(n)$ i.e. one pass of `Randomized_partition`, we find that pivot has rank $k = i$
- If the participants sub-routine is *not randomized*, worst case for QuickSelect is similar to worst case of QuickSort $\rightarrow O(n^2)$

### Worse-case Linear-Time Select - Pseudocode

```
function Select(A, n, i)
  # step 1 - Theta(n)
  # divide A into ceil(n/5) groups of 5 elements each
  # find the median of each 5-elements group in O(1)
  # step 2, T(n/5)
  # Let B be ceil(n/5) medians of each groups
  Select(B, |B|, |B|/2)) # find median of medians
  # step 3 - Theta(n)
  k = Partition(A, x) # use this specific pivot x
  # Let 'A / ''A be elements < x / > x, respectively
  # step 4-5-6, similar as Quickselect - T(7n/10)
  if i == k, then return k # found
  else if i < k, then return Select('A, k-1, i)
  else, return Select(''A, n-k, i-k)
```



- Divide the n elements into groups of 5. There are $\lfloor \frac{n}{5} \rfloor$ such groups
- Find median of each group in $O(1) \rightarrow$ sort and get index $2$, sorting $5$ elements is $O(1)$, there are $\lfloor \frac{n}{5} \rfloor$ such medians
- Recursively Select the median x of the $\lfloor \frac{n}{5} \rfloor$ group medians. This $x$, the **median of medians**, will be the pivot.
- At least half of the group of $5$ medians are $\leq x$, which is at least $\lfloor \lfloor \frac{n}{5}/2 \rfloor \rfloor = \lfloor \frac{n}{10} \rfloor$ elements.
- $\therefore \geq 3 \cdot \lfloor \frac{n}{10} \rfloor$ elements are $\leq x$ and $\geq x$
- For large n, since at least $\lfloor \frac{n}{10} \rfloor$ elements are $leqx$ (or $\geq x$), after partitioning $A$ around $x$, the recursive call to Select (step 5-6) is executed recursively on at most $n - \frac{3n}{10} = \frac{7n}{10}$ elements.
- Thus, the recurrence for running time T(n) takes time of not more than $T(n)$ takes time of not more than $T(\frac{n}{5}) + T(\frac{7n}{10}) + \Theta(n)$in the worst case. For small $n$, we can compute $T(n) \in \Theta(1)$.

## A. Misc. formulas

### A1. Definition of limits

$\lim_{n\to\infty}(\frac{f(n)}{g(n)}) = z \Rightarrow \forall \epsilon > 0,\ \exists n_0 > 0,$ s.t. $\forall n \geq n_0$: $\boxed{|\frac{f(n)}{g(n)} - z| < \epsilon}$

### A2. Logarithm/Exponential rules

1. $\log(M \cdot N) = \log M + \log N \Rightarrow \log(M/N) = \log M - \log N$
2. $\log(M^k) = k \cdot \log(M) \Rightarrow b^{\log_b(k)} = k$
3. $\log(1) = 0 \ ||\ \log_b(b) = 1 \ ||\ \log_b(b^k) = k$

---

a. $a^x \times a^y = a^{x+y} \Rightarrow a^x \div a^y = a^{x-y} \ ||\ (a^x)^y = a^{x \cdot y}$

b. For any $a^b$, $a^b = e^{b \ln(a)}$

### A3. Helpful approximations

1. **stirling's approximation:**
   - $T(n) = \sum_{i=0}^{n} \log(n-i) = \log \prod_{i=0}^{n}(n-i) = \Theta(n \log n)$
   - OR $n! \approx (\frac{n}{e})^n \sqrt{2\pi n}$
2. **harmonic number:**
   $H_n = \sum_{k=1}^{n} \frac{1}{k} = \Theta(\lg n)$
3. **basel problem:**
   $\sum_{n=1}^{N} \frac{1}{n^2} \leq 2 - \frac{1}{N} \xrightarrow{N\to\infty} 2$ because $\sum_{n=1}^{N} \frac{1}{N^2} \leq$
   $1 + \sum_{x=2}^{\log_3 n} \frac{1}{(x-1)x} = 1 + \sum_{n=2}^{N}(\frac{1}{n-1} - \frac{1}{n}) = 1 + 1 - \frac{1}{N} = 2 - \frac{1}{N}$
4. **number of primes in range:** $\{1, \ldots, K\}$ is $> \frac{K}{\ln K}$
5. **L'Hôpital's rule:** $\lim_{x\to c} \frac{f(x)}{g(x)} = \lim_{x\to c} \frac{f'(x)}{g'(x)}$

### A4. Asymptotic bounds

$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < 2^n < 2^{2n}$
$\log_a n < n^a < a^n < n^n < n!$
for any $a, b > 0$, $\log_a n < n^b$

### A5. Multiple parameters

for two functions $f(m,n)$ and $g(m,n)$, we say that $f(m,n) = O(g(m,n))$ if there exists constants $c, m_0, n_0$ such that $0 \leq f(m,n) \leq c \cdot g(m,n)$ for all $m \geq m_0$ or $n \geq n_0$.

### A6. Example proofs

*Proof.* that $2n^2 = O(n^3)$
let $f(n) = 2n^2$. then $f(n) = 2n^2 \leq n^3$ when $n \geq 2$.
set $c = 1$ and $n_0 = 2$.
we have $f(n) = 2n^2 \leq c \cdot n^3$ for $n \geq n_0$.

*Proof.* $n = o(n^2)$
For any $c > 0$, use $n_0 = 2/c$.

*Proof.* $n^2 - n = \omega(n)$
For any $c > 0$, use $n_0 = 2(c+1)$.

*Example.* let $f(n) = n$ and $g(n) = n^{1+\sin(n)}$.
Because of the oscillating behaviour of the sine function, there is no $n_0$ for which $f$ dominates $g$ or vice versa.
Hence, we cannot compare $f$ and $g$ using asymptotic notation.

*Example.* let $f(n) = n$ and $g(n) = n(2+\sin(n))$.
Since $\frac{1}{3}g(n) \leq f(n) \leq g(n)$ for all $n \geq 0$, then $f(n) = \Theta(g(n))$.
(note that limit rules will not work here)

### A7. Formulas

#### Geometric Series

- $S_n = \frac{a(1-r^n)}{1-r}$ — $S_\infty = \frac{a}{1-r}$
  - where $a$ is the first term, $r$ is the common ratio, $n$ is number of terms

#### Arithmetic Series

- $S_n = n(\frac{a_1+a_n}{2})$, $n$ is number of terms, $a_1$ is first term, $a_n$ is $n^{th}$ term

#### Ratio Test

$L = \lim_{n\to\infty} \frac{a_{n+1}}{a_n}$
$$\begin{cases} L < 1 & \textbf{absolutely convergent} \\ L > 1 \text{ or } L = \infty & \textbf{divergent} \\ L = 1 & \textbf{inconclusive} \end{cases}$$

## B. Leetcode pseudo-code

### B1. Coin Change I and II

- Fewest number of coins needed to make target amount

```
def coinChange(coins: List[int], amount: int):
  dp = [0] + [float('inf')] * amount
  for coin in coins:
    for target in range(coin, amount + 1):
      dp[target] = min(dp[target],
                       1 + dp[target - coin])
  if dp[-1] == float('inf'):
    return -1
  else:
    return dp[-1]
```

- Number of combinations that make up target amount

```python
def change(amount: int, coins: List[int]):
  dp = [0] * (amount + 1)
  dp[0] = 1
  for coin in coins:
    for target in range(coin, amount + 1):
      dp[target] += dp[target - coin]
  return dp[-1]
```

### B2. Best time to buy sell stock

- No cooldown

```python
def maxProfit(self, prices: List[int]) -> int:
  if len(prices) < 2:
    return 0
  l, r = 0, 1
  res = 0
  while r < len(prices):
    profit = prices[r] - prices[l]
    if profit >= 0:
      res = max(res, profit)
    else:
      l = r
    r += 1
  return res
```

- With cooldown

```python
def maxProfit(self, prices: List[int]) -> int:
  if not prices:
    return 0
  dp = [[0,0,0] for _ in range(len(prices))]
  # state [buy, sell, cooldown]
  dp[0][0] = -prices[0]
  dp[0][1] = 0
  dp[0][2] = 0
  for i in range(1, len(prices)):
    dp[i][0] = max(dp[i-1][0], dp[i-1][2]
                 - prices[i])
    dp[i][1] = dp[i-1][0] + prices[i]
    dp[i][2] = max(dp[i-1][1], dp[i-1][2])
  return max(dp[-1][1], dp[-1][2])
```

### B3. Subsequence

- Longest Increasing subsequence

```python
def lengthOfLIS(self, nums: List[int]) -> int:
  n = len(nums)
  dp = [1] * n
  for i in range(n - 1, -1, -1):
    for j in range(i + 1, n):
      if nums[i] < nums[j]:
        dp[i] = max(dp[i], 1 + dp[j])
  return max(dp)
```

- Longest Common subsequence

```python
def longestCommonSubsequence(self, text1: str, ↩
     text2: str) -> int:
  dp = [[0] * (len(text2) + 1) for _ in range(len(↩
     text1) + 1)]
  for i in range(len(text1) - 1, -1, -1):
    for j in range(len(text2) - 1, -1, -1):
      if text1[i] == text2[j]:
        dp[i][j] = 1 + dp[i+1][j+1]
      else:
        dp[i][j] = max(dp[i+1][j], dp[i][j+1])
  return dp[0][0]
```

### B4. Distinct Subsequence

If subsequence has last element that is repeated with current element $x_j = x_i$, where $j < i$, there is an additional subsequence that is repeated. Otherwise, there can be 2 subsequences that end with either $x_j$ or $x_i$.

```python
def distinctSubseqII(self, S):
  res, end = 0, collections.Counter()
  for c in S:
    res, end[c] = res * 2 + 1 - end[c], res + 1
  return res
```

### B5. Jump Game II

```python
def jump(self, nums: List[int]) -> int:
 if len(nums) <= 1: return 0
 l, r = 0, nums[0]
 res = 1
 while r < len(nums) - 1:
  res += 1
  nxt = max(i + nums[i] for i in range(l, r + 1))
  l, r = r, nxt
 return res
```

### B6. Container with most water

```python
def maxArea(self, height: List[int]) -> int:
 l, r = 0, len(height) - 1
 res = 0
 while l <= r:
  width = r - l
  length = min(height[l], height[r])
  res = max(res, width * length)
  if height[l] < height[r]:
   l += 1
  else:
   r -= 1
 return res
```

### B7. Maximum subarray

```python
def maxSubArray(self, nums: List[int]) -> int:
  ans, curr = 0, 0
  for i in range(len(nums)):
      if curr < 0:
          curr = 0
      curr += nums[i]
      ans = max(ans, curr)
  return ans
```

### B8. Gas Station

```python
def canCompleteCircuit(gas: List[int], cost: List[int↩
    ]) -> int:
  if (sum(gas) - sum(cost) < 0):
      return -1
  gas_tank, start_index = 0, 0
  for i in range(len(gas)):
      gas_tank += gas[i] - cost[i]
      if gas_tank < 0:
          start_index = i+1
          gas_tank = 0

  return start_index
```

## C. Recurrences

### C1. $T(n) = 9T(\frac{n}{3}) + n^2/lgn$

$$T(n) = 9T(\tfrac{n}{3}) + \frac{n^2}{\lg n}$$

$$\frac{T(n)}{n^2} = \frac{T(\frac{n}{3})}{\frac{n}{9}} + \frac{n^2}{\lg n}$$

$$\|$$

$$\cancel{\frac{T(n)}{n^2} = \frac{T(\frac{n}{3})}{(\frac{n}{3})^2}} + \frac{1}{\lg n}$$

$$\cancel{\frac{T(\frac{n}{3})}{(\frac{n}{3})^2} = \frac{T(\frac{n}{9})}{(\frac{n}{9})^2}} + \frac{1}{\lg(\frac{n}{3})}$$

$$\vdots$$

$$\cancel{T(1)} = \frac{T(\frac{1}{3})}{\frac{1}{9}} + \frac{1}{\lg(1)}$$

$$\therefore \frac{T(n)}{n^2} = \frac{1}{\lg n} + \frac{1}{\lg(\frac{n}{3})} + \cdots \frac{1}{\lg 3} + 9T(\tfrac{1}{3})$$

$$\frac{T(n)}{n^2} = \frac{1}{\lg 3}\left[\frac{1}{\log_3 n} + \cdots + \frac{1}{3} + \frac{1}{2} + \frac{1}{1}\right]$$

$$\frac{1}{\lg 9} \Rightarrow \frac{1}{\lg 3^2} \Rightarrow \frac{1}{2\lg 3}$$

$$\Rightarrow \int_1^{\log_3 n} \frac{1}{x} = \log\log n - \log\log 1$$
$$= \log\log n$$

$$\frac{T(n)}{n^2} \in \Theta(\lg\lg n)$$

$$T(n) \in \Theta(n^2 \lg\lg n)$$

### C2. $T(2^n) = T(2^{n-1}) + \cdots + T(2^1) + T(2^0)$

1. $T(2^n) - T(2^{n-1}) = T(2^{n-1}) + n^2 - (n-1)^2$
2. $T(2^n) = 2T(2^{n-1}) + 2n - 1 || 2^n = N \Rightarrow N = 2^n \Rightarrow lgN = nlg2 \approx n$
3. $T(N) = 2T(N/2) + lgN + 1 \Rightarrow$ by M.T, $T(N) = N \Rightarrow T(2^n) = \Theta(2^n)$

## D. Greedy Algorithms Proof:

### D1. Scheduling

- **Optimal Substructure:**
  - Suppose an optimal scheduling $S$ contains activity $a_j$, let:
    - $Before_j = \{a_i : f_i \le s_j\}$ (finish before activity $a_j$ starts)
    - $After_j = \{a_i : s_i \ge f_j\}$ (start after activity $a_j$ finishes)
    - $\therefore S$ contains optimal scheduling for $Before_j$ and $After_j$
    - Proof by contradiction
      - Assume $S$ is an optimal schedule for the entire set of activities but does not contain optimal schedules for $Before_j$ or $After_j$
      - Replace the suboptimal schedule in $S$ with the optimal scheduling for $Before_j$ or $After_j$
      - $\therefore S$ now has more activities, contradicting the assumption that $S$ is optimal
- **Proving Greedy Choice Property:**
  - Let $a^*$ be activity that finishes the earliest among all activities in the set
  - Assume $S$ is an optimal schedule and $a$ is the first activity in $S$
    - If $a = a^*$, the greedy choice is already part of the optimal solution
    - If $a \neq a^*$, replace $a$ in $S$ with $a^*$
    - Since $a^*$ finishes earlier than $a$, it leaves at least as much room for subsequent activities
    - The new schedule remains compatible and contains the same number of activities, making it optimal
  - By induction, repeating this process ensures the greedy choice (selecting the activity that finishes earliest) always leads to an optimal solution

## E. Reductions

| | |
|---|---|
| **P** | ... you can **implement a polynomial time algorithm** to solve this |
| **NP** | 1. ... you can **outline (NOT solve!) a certificate** as a **proof for YES**<br>2. ... you can **verify the certificate in polynomial time** |
| **NP-Hard** | **Option A**: ... you can **prove all NP problems can reduced to** this<br>**Option B**: ... you can **find an NP-Hard problem X**, and **prove X $\le_p$ this** |
| **NP-Complete** | 1. ... you can **prove that this problem is NP**<br>2. ... you can **prove that this problem is NP-Hard** |

### E1. Reduce from HC $\le_q$ TSP

- **Show the transformation algorithm.**
  - Let $G = (V, E)$ be an instance $\alpha$ of HC. We build an instance $\beta$ of TSP as follows:
    - Create a complete graph $G'$ on the same vertex set $V$.
    - For each pair $u, v \in V$:
      - If $u, v \in E$, then $w(u, v) = 1$.
      - Otherwise, $w(u, v) = 2$ (or any value greater than 1).
  - **Theorem:** $G$ has a Hamiltonian cycle iff. $G'$ has a TSP tour of cost $\le n$.
- **Show the transformation algorithm runs in polynomial time.**
  - A complete graph $G'$ on $n$ vertices has $\binom{n}{2} = \frac{n(n-1)}{2}$ edges.
  - For each edge, we determine if $(u, v) \in E$ in $O(1)$ time (using an adjacency matrix or hash table).

- Assigning weights for all edges takes $O(n^2)$ total.
- **Show a YES answer to HC instance implies YES to TSP instance.**
  - **Theorem ($\rightarrow$):** If $G$ has a Hamiltonian cycle, then $G'$ has a TSP tour of cost at most $n$.
    - Let $C$ be a Hamiltonian cycle in $G$.
    - $G$ is subgraph of complete graph $G'$, $\therefore C$ must also be present in $G'$.
    - $C$ is a tour since each vertex appears exactly once in $C$.
    - Cost of each edge in $C$ is 1, as each edge of $C$ also present in $G$.
    - Hence, the total cost of the tour $C$ in $G'$ is $n$.
  - Therefore, $G'$ has a TSP tour of cost at most $n$.
- **Show a YES answer to TSP instance implies YES to HC instance.**
  - **Theorem ($\leftarrow$):** If $G'$ has a TSP tour of cost $\leq n$, then $G$ has a HC.
    - Let $C$ be a TSP tour of cost at most $n$ in $G'$.
    - The cost of each edge in $G'$ is at least 1.
    - There are $n$ edges in $C$, so each edge of $C$ must have weight exactly 1.
    - Therefore, each edge of $C$ is present in $G$ as well.
    - Since $C$ visits each vertex exactly once, $C$ is a Hamiltonian cycle.
  - Hence, $G$ has a Hamiltonian cycle.

## E2. Prove IS is NP-Complete

1. Show that $IS \in NP$
   - Certificate is in independent set
   - We can check if the size $\geq k$ in $O(1)$ hence polynomial time
   - We can also check all edges $(u, v) \in E$ so that at most $u$ or $v$ but not both are in the IS (e.g. by storing data in Hash Table). This check can be done in $O(E)$ which is also polynomial time.
2. Pick a problem $A$ ($3SAT$) known to be NP-Complete, then show that $A(3SAT) \leq_p X(IS)$

**Solution:**

Let $\phi = C_1 \wedge C_2 \wedge \ldots C_m$ be the input formula to a 3SAT problem, where each clause $C_c$ has three literals chosen from $\{x_i, \bar{x}_i \| 1 \leq i \leq n\}$. We construct a DONUT problem $\langle G, p, k \rangle$ as follows.

The vertices $V$ of $G$ are $\{v_{c,j} \| 1 \leq c \leq m, 1 \leq j \leq 3\}$, where $v_{c,j}$ corresponds to literal $j$ in clause $C_c$. We label each vertex $v_{c,j}$ with $x_i$ or $\bar{x}_i$, whichever appears in position $j$ of clause $C_c$. The edges $E$ of $G$ are of two types:

- For each clause $C_c$, an edge between each pair of vertices corresponding to literals in clause $C_c$, that is, between $v_{c,j_1}$ and $v_{c,j_2}$ for $j_1 \neq j_2$.
- For each $i$, an edge between each pair of vertices for which one is labeled by $x_i$ and the other by $\bar{x}_i$.

The function $p$ maps all vertices to 1. The threshold $k$ is equal to $m$. We claim that $\phi$ is satisfiable iff the total profit in the DONUT problem $\langle G, p, k \rangle$ can be at least $k$.

First, suppose that $\phi$ is satisfiable. Then there is some truth assignment $A$ mapping the variables to $\{true, false\}$. $A$ must make at least one literal per clause true; for each clause, select the vertex corresponding to one such literal to be in the set $U$. Since there are $m$ clauses, this yields exactly $m = k$ vertices, so the total profit is $k$. Moreover, we claim that $U$ cannot contain two neighboring vertices in $G$. Suppose for contradiction that $u, v \in U$ and $(u, v) \in E$. Then the edge $(u, v)$ must be of one of the two types above. But $u$ and $v$ cannot correspond to literals in the same clause because we selected only one vertex for each clause. And $u$ and $v$ cannot be labeled by $x_i$ and $\bar{x}_i$ for the same $i$, because $A$ cannot make both a variable and its negation true. Since neither possibility can hold, $U$ cannot contain two neighboring vertices. $U$ achieves a total profit of $k$ for the DONUT problem $\langle G, p, k \rangle$.

Conversely, suppose that there exists $U \subseteq V$, $|U| \geq k = m$ containing no two neighbors in $G$. Since $U$ does not contain neighbors, it cannot contain two vertices from the same clause. Therefore, we must have $|U| = m$, with exactly one vertex from each clause. Now define a truth assignment $A$ for the variables: $A(x_i) = true$ if some vertex with label $x_i$ is in $U$, and $A(x_i) = false$ if some vertex with label $\bar{x}_i$ is in $U$. For other variables the truth value can be arbitrary. Also since $U$ does not contain neighbors, $U$ cannot contain two vertices with contradictory labels, so assignment $A$ is well-defined. $A$ satisfies all clauses by making one literal corresponding to a vertex in $U$ true in each clause. Therefore, $A$ satisfies $\phi$.

For the last question, suppose that there is a polynomial-time algorithm to solve the original problem, i.e., to output a subset $U$ that maximizes the total profit. Then this algorithm can be easily adapted to a polynomial-time algorithm for DONUT: for any $\langle G, p, k \rangle$, simply run the assumed algorithm and obtain an optimal subset $U$. Then output $true$ if $k \leq |U|$, and $false$ otherwise. Since we have already shown that DONUT is NP-hard, this implies that P= NP.

## E3. Red Blue Colouring

Show problem is in `NP`

The certificate would be an $n$-length sequence $f_1, f_2, \ldots, f_n$ where each $f_j \in red, blue$. A certifier algorithm should verify whether the above sequence is a valid $redblue$ respectful coloring or not, which can easily be done by making a pass over color-preferences of each person. Clearly, the certificate is polynomial in size, and the verification can be done in polynomial time.

1. For providing the correct (or almost correct) polysize certificate + 1.5m
2. For providing justification on how to verify in polytime + 1.5m

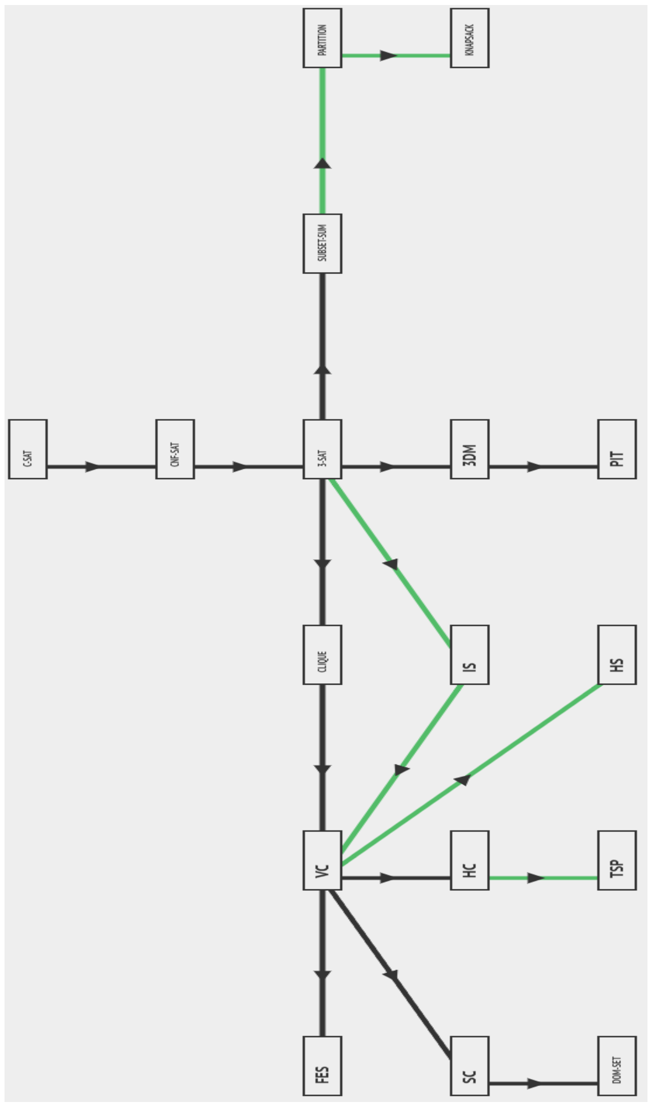
Show problem is in `NP-Hard`

Consider a 3-SAT instance with $n$ variables $x_1, \ldots, x_n$ and $k$ clauses $C_1, \ldots, C_k$. Now, create an instance of the Red-Blue Respectful Coloring problem as follows: For each variable $x_j$, consider a ball $B_j$, and for each clause $C_i$, consider a person $P_i$. The create color preference list for a person $P_i$ as: If $x_j$ appears in $C_i$ as a positive literal, then set $c_{ij} = blue$, else if $x_j$ appears in $C_i$ as a negative literal then set $c_{ij} = red$, otherwise, set $c_{ij}$ to be an arbitrary color from $green, yellow, pink$.
Clearly, the above reduction takes polynomial time. Next, we argue that the 3-SAT instance is satisfiable if and only if the reduced Red-Blue Respectful Coloring problem instance is a YES instance.
To show 3-SAT instance is satisfiable implies the reduced Red-Blue Respectful Coloring problem instance is a YES instance, consider a satisfying assignment $\sigma$ and build a red-blue coloring by setting $f_j = blue$ if $x_j = 1$ in $\sigma$; otherwise, $f_j = red$. Since $\sigma$ satisfies each clause $C_i$, by the construction, for each $P_i$ there exists a ball $B_j$ such that $f_j = c_{ij}$.
For the converse, consider a valid $redblue$ respectful coloring $f_1, \ldots, f_n$. Then create an assignment $\alpha$ by setting $\alpha(x_j) = 1$ if $f_j = blue; \alpha(x_j) = 0$ otherwise. Again, by an argument similar to that in the last paragraph, $\alpha$ is a satisfying assignment.

1. For providing correct/almost correct transformation function from 3-SAT to respectful coloring +2 marks
2. For providing a clear proof on "if and only if" part of the reduction +4 marks
3. For providing some justification (but not a complete one, e.g. only one side) on 'if and only' if part of the reduction +2 marks

## E4. All NP-Complete Problems

- **SAT Problems**
  - **C-SAT**: Determination if there is an input to a boolean circuit that makes the output true.
  - **CNF-SAT**: Boolean satisfiability where the formula is in CNF (conjunctive normal form). Standardized way of expressing boolean formulas in logic where the formula is an AND of one or more clauses where a clause is an OR of literals. A literal is either a bool or its negation.
  - **3-SAT**: CNF-SAT restricted to clauses of exactly 3 literals.
- **Graph Problems**
  - **Vertex Cover**: Smallest set of vertices that touches all edges in the graph.
  - **Clique**: Largest complete subgraph.
  - **Independent Set**: Largest set of vertices with no edges between them.
  - **Hamiltonian Cycle**: A cycle that visits each vertex exactly once.
  - **TSP**: Shortest tour visiting all cities.
  - **DOM-Set**: Smallest subset of vertices where every vertex is either in the set or adjacent to a vertex in the set.
- **Subset and Partition Problems**
  - **Subset Sum**: Given a set of integers, determine if there is a subset whose sum equals a target.
  - **Partition**: Determine if a set of integers can be split into two subsets with equal sums.
  - **Knapsack**: Maximize value subject to weight constraints in a knapsack.
- **Miscellaneous Problems**
  - **3 Dimension Matching**: Match elements from three sets such that pairs form a complete matching.
  - **PIT/Perfect Matching in Bipartite Graph**: Find a matching that covers every vertex exactly once.


PARTITION
KNAPSACK
SUBSET-SUM
C-SAT
CNF-SAT
3-SAT
3DM
PIT
CLIQUE
IS
HS
VC
HC
TSP
FES
SC
DOM-SET