# Algorithms: Design & Analysis Notes

What is an algorithm?
→ Well defined (unambiguous) series of steps/instructions
→ Executes in a finite amount of time.

Computational Problem: → defines a specified relationship between the input & output of the problem instance.
→ A problem instance consists of the input needed to compute a solution to the problem.

Sorting: Input: sequence of n numbers: $\langle a_1, a_2, \dots, a_n \rangle$ [n! permutations]
Output: a permutation $\langle a_1', a_2', \dots, a_n' \rangle$ of the input sequence such that $a_1' \le a_2' \le \dots \le a_n'$.

* RAM → Random Access Machine Model of Computation

Insertion Sort: Best Case: $T(n) = an + b$
Worst Case: $T(n) = an^2 + bn + c$

O-Notation: → upper bound on the asymptotic behaviour of a function
$O(n^c)$ c ≥ some value. $7n^3$ $O(n^3)$, $O(n^4)$, $O(n^5)$
Big-O ⇒ $\exists c, n_0 \; \forall n \ge n_0 \; f(n) \le cg(n) \iff f(n) = O(g(n))$

Ω-Notation: → lower bound on the asymptotic behavior of a function
→ grows at least as fast as a certain rate
$\Omega(n^c)$ c ≤ some value. $7n^3 + 100n^2$ -- $\Omega(n^3)$, $\Omega(n^2)$, $\Omega(n)$
Big-Ω ⇒ ~~$\exists c, n_0 \; \forall n \ge n_0 \; f(n) \ge cg(n)$~~
⇔ $\exists c, n_0 \; \forall n \ge n_0, \; cg(n) \le f(n) \iff f(n) = \Omega(g(n))$

Θ-Notation: → tight bound on the asymptotic behaviour
b grows precisely at a certain rate.
If a function is both $O(f(n))$ & $\Omega(f(n))$, it is $\Theta(f(n))$
$7n^3 + \dots$ $O(n^3)$ & $\Omega(n^3)$, hence $\Theta(n^3)$
Big-Θ ⇒ $\exists c_1, c_2, n_0 \; \forall n \ge n_0, \; c_1 g(n) \le f(n) \le c_2 g(n)$

Small-o: $\forall c > 0, \exists n_0 \ni n_0 \ge n_0, \; 0 \le f(n) < cg(n) \iff f(n) = o(g(n))$
$f(n) = o(g(n)) \iff \lim_{n \to \infty} f(n)/g(n) = 0$

Small ω: $\forall c > 0, \exists n_0 \ni n \ge n_0, \; 0 \le cg(n) < f(n) \iff f(n) = \omega(g(n))$
$f(n) = \omega(g(n)) \iff \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$.
~~$f(n) = o(g(n)) \iff \lim$~~

$T(n) = 10n^3 + 100n^2 + 55$. Show that $T(n) = O(n^3)$

By the definition, $0 \le 10n^3 + 100n^2 + 55 \le cn^3$

$0 \le 10 + \frac{100}{n} + \frac{55}{n^3} \le c$

Let $c = 11$.

Then we need $n$ such that $\frac{100}{n} + \frac{55}{n^3} < 1$.

Let $n = 101$. Then $\frac{100}{n} + \frac{55}{n^3} < 1$.

Then $n_0 \ge 101$, $c \ge 11$.

$10(101)^3 + 100(101)^2 + 55 \le 11(101)^3$ is True. Hence proved

$f(n) = 2n^2 + 3n + 1$ $\quad g(n) = n^2$ $\quad f(n) = O(g(n))$

$0 \le f(n) \le cg(n)$ → By definition

$2n^2 + 3n + 1 \le cn^2$

$2 + \frac{3}{n} + \frac{1}{n^2} \le c$

$< 1$

$\frac{3}{n} + \frac{1}{n^2} < 1$

$\frac{3}{n} + \frac{1}{16} < 1$ $\quad \exists n$

$\Rightarrow \frac{3}{n} + \frac{1}{16} \ge 0.8125 < 1$.

$c \ge 2 + 0.8125 \Rightarrow c \ge 2.8125$ $\quad n \ge 4$

Comparison Sort → compares elements to gain info:

$\exists a_i, a_j$, $i \le j$, one of the tests is performed:

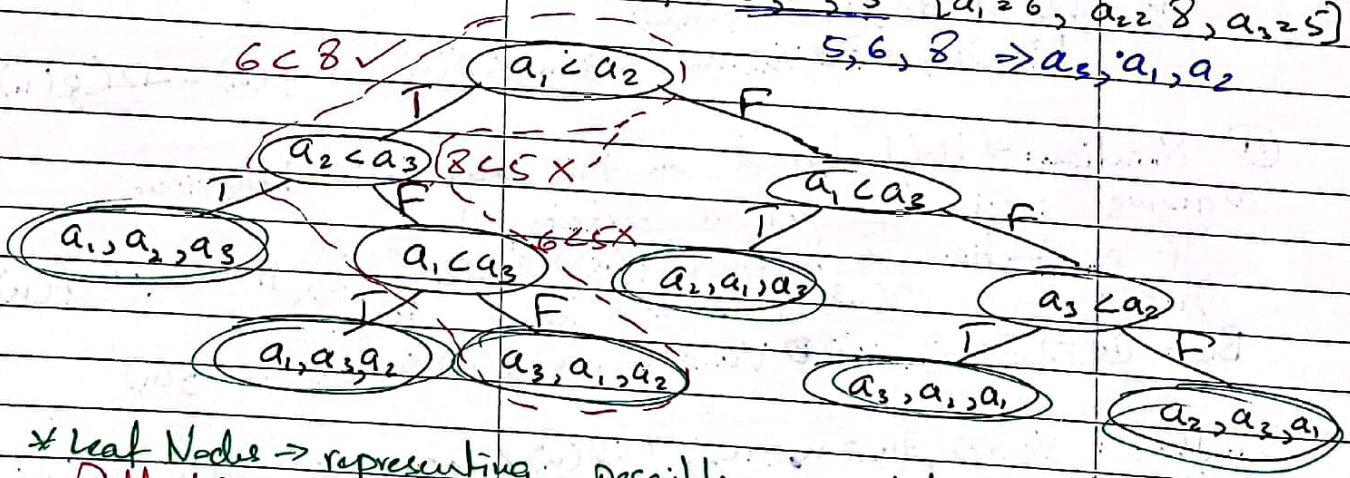
$a_i < a_j$, $a_i \le a_j$, $a_i = a_j$, $a_i \ge a_j$, $a_i > a_j$

Can view comparison sorts abstractly as decision Trees.

↳ Decision Trees: Full Binary Trees, ~~with each node~~ representing comparisons performed by an alg on an input.

Consider Input: $a_1, a_2, a_3$ for 6, 8, 5 $[a_1 = 6, a_2 \ge 8, a_3 \ge 5]$

5, 6, 8 $\Rightarrow a_3, a_1, a_2$




* Leaf Nodes → representing possible permutations / order.

* Path / Decision Route for inputs 6, 8, 5 (T, F, F)

* Length of the longest simple path from root to any of its reachable leaves presents the worst case.

* Consequently, the height of the tree gives the worst-case number of comparisons for a decision tree.

**Theorem 8.1:**

Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

↳ A lower bound for the worst case.

**Proof:** Consider a decision tree of height 'h', with 'ℓ' reachable leaves corresponding to a comparison sort on 'n' elements. We have $n!$ permutations, so $n! \leq \ell$.

* A binary tree of height 'h' has no more than $2^h$ leaves.

Then: $n! \leq \ell \leq 2^h$

$\Rightarrow n! \leq 2^h$

$\Rightarrow h \geq \lg(n!)$ (By taking logs)

$\Rightarrow h \geq \lg\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1 + \frac{k}{n}\right)\right)$ (k is a constant, $k \geq 1/12$)

$\Rightarrow h \geq \frac{1}{2}\lg(2\pi n) + n\lg(n) - n\lg(e) + \lg\left(1 + \frac{k}{n}\right)$

$\Rightarrow h \geq \frac{1}{2}\lg(2\pi) + \frac{1}{2}\lg(n) + n\lg(n) - n\lg(e) + \lg\left(1 + \frac{k}{n}\right)$

$\Rightarrow h \geq n\lg n + \frac{1}{2}[\lg(2\pi) + \lg(n)] - n\lg(e) + \lg\left(1 + \frac{k}{n}\right)$

* ∵ since $n \lg n$ is monotonically increasing, it is the highest order term.

$\Rightarrow h = \Omega(n \lg n)$ QED!

* No comparison sorting algorithm can have a worst case scenario as best as $\Omega(n \lg n)$.

# Recurrance Relations.

Divide & Conquer Algorithm → Divide into smaller parts/pro
↳ Conquer → solve the problem recursively.

→ Usually, $T(n) = D(n) + aT(\frac{n}{b}) + c(n)$

$aT(\frac{n}{b})$ → conquer n size prob into $\frac{n}{b}$ & worked on 'a' of them

$F(n) = F(n-1) + F(n-2) = O(2^n)$

$T(n) = nT(n-1)$ $\quad T(n) = \Theta(n!)$

Fib(n): Base case
return Fib(n-1) + Fib(n-2)

$F(2) = F(1) + F(0) = 1 + 1 = 2.$

$F(n) = F(n-1) + F(n-2)$

Consider: $F(n) = F(n-1) + F(n-1) + 1$ } Difference Equation
$= 2F(n-1) + 1$

$\Rightarrow \nabla F(n) = F_n - F_{n-1}$ ——→

Unroll:

$F(n) = 2F(n-1) + 1 = 2[2F(n-2)+1] + 1$
$= 2^2[F(n-2)] + 2 + 1$
$= 2^2[2F(n-3) + 1] + 2 + 1$
$= 2^3 F(n-3) + 2^2 + 2 + 1$
$= 2^3 F(n-3) + 2^2 + 2^1 + 2^0$
$= 2^n F(0) + 2^{n-1} + 2^{n-2} + \cdots + 2^2 + 2^1 + 2^0$
$F(0) = 0$
$= 2^{n-1} + \cdots + 2^2 + 2^1 + 2^0$
$= \sum_{j=1}^{n-1} 2^j = 2^n - 1$

$F(n) = bF(n-1) + f(n)$
$= b[bF(n-2) + f(n)] + f(n)$
$= b^2 F(n-2) + b^1 f(n) + b^0 f(n)$
$= b^2[bF(n-3) + f(n)] + b^1 f(n) + b^0 f(n)$
$= b^3 F(n-3) + b^2 f(n) + b^1 f(n) + b^0 f(n)$
$= b^n F(0) + b^{n-1}(fn) + \cdots + b^1 f(n) + b^0 f(n)$ $[F(0)=c]$
$= cb^n + \sum_{i=0}^{n-1} b^i f(n)$

Linear Recurrance Relation of order i with constant coefficients:

$a_0 A_n + a_1 A_{n-1} + a_2 A_{n-2} + \cdots + a_i A_{n-i} = 0$

given $A_n = r^n \quad (r^n > 0)$

$a_0 r^n + a_1 r^{n-1} + \cdots + a_i r^{n-i} = 0$

$r^{n-i}[a_0 r^i + a_1 r^{i-1} + \cdots + a_i] = 0$

$a_0 r^i + a_1 r^{i-1} + \cdots + a_i = 0$ } → Characteristic Equation

Characteristic Polynomial

Let $r_1, r_2, \cdots, r_i$ be i distinct roots of our C.P

Then the general solution is:

$A_n = \alpha_1 r_1^n + \alpha_2 r_2^n + \cdots + \alpha_i r_i^n$

Solving Fibonacci Now:

$F_n = F_{n-1} + F_{n-2}$

$F_n - F_{n-1} - F_{n-2} = 0$

Let $F_n = r^n$

$r^n - r^{n-1} - r^{n-2} = 0$

$r^{n-2}[r^2 - r - 1] = 0$

$r^2 - r - 1 = 0 \quad : r_1 = 1+\sqrt{5}/2, \; r_2 = 1-\sqrt{5}/2$

$\varphi$ [Golden Ratio] $\quad \hat{\varphi}$

$\Rightarrow F_n = \varphi^n \Rightarrow F_n = \alpha_1 \varphi^n + \alpha_2 \hat{\varphi}^n = O(2^n)$

$F_0: \; \alpha_1 \varphi^0 + \alpha_2 \hat{\varphi}^0 = 0 \Rightarrow \alpha_1 + \alpha_2 = 0 \Rightarrow \alpha_1 = -\alpha_2$

$F_1: \; \alpha_1 \varphi + \alpha_2 \hat{\varphi} = 1$

$-\alpha_2 \varphi + \alpha_2 \hat{\varphi} = 1$

$\alpha_2 (\hat{\varphi} - \varphi) = 1$

$\Rightarrow \alpha_2 = 1/\hat{\varphi} - \varphi \qquad \alpha_1 = 1/\varphi - \hat{\varphi}$

$\alpha_2 = \dfrac{1}{\frac{1-\sqrt{5}}{2} - \left(\frac{1+\sqrt{5}}{2}\right)} = \dfrac{1}{\frac{1-\sqrt{5}-1-\sqrt{5}}{2}} = \dfrac{1}{\frac{-2\sqrt{5}}{2}}$

$\alpha_2 = -1/\sqrt{5}$

$\alpha_1 = \dfrac{1}{\frac{1+\sqrt{5}}{2} - \left(\frac{1-\sqrt{5}}{2}\right)} = \dfrac{1}{\frac{1+\sqrt{5}-1+\sqrt{5}}{2}} = \dfrac{1}{2\sqrt{5}/2}$

$\alpha_1 = 1/\sqrt{5}$

$A_n - 3A_{n-1} + 2A_{n-2} = 0$

Let $A_n = q^n$

$q^n - 3q^{n-1} + 2q^{n-2} = 0$

$r^2 - 3r + 2r = 0$

$r^2 - 2r - r + 2 = 0$

$r(r-2) - 1(r-2) = 0$

$(r-2)(r-1) = 0$ ⇒ $r_1 = 1, r_2 = 2$.

$A_n = \alpha_1(1)^n + \alpha_2(2)^n$

$A_n = \alpha_1 + \alpha_2(2^n)$ Assympt: $\alpha_1$ doesn't matter

$\Rightarrow A_n = \Theta(2^n)$

**Divide & Conquer** - Divide the problem into smaller problems.
- Conquer smaller problems recursively.
- Combine the sub-problems into your solution.

Let $T(n)$ be the running time with a base case $n \leq c$
then $T(n) = \Theta(1)$ → base case.

Suppose we split the problem into some 'n' any tree,
- subproblem [$a \geq 1$] → branching factor.
- each of the subproblem is of size by a dividing factor [$b > 1$] → dividing factor $[n/b]$

Then: $T(n) = a \cdot T(n/b) + f(n)$ [$f(n) = D(n) + C(n)$]
(Conquer; Cost of Div + Cost of combine.)

$T(n) = T(n/2) + \Theta(1)$ → Binary Search

$T(n) = 2T(n/2) + \Theta(n)$ → Merge Sort
$T(n) = n \lg n$.

[* Substitution using mathematical induction
↳ Read from book]

$T(n) = 2T(n/2) + \Theta(n)$ Assuming 'n' is in the exact powers of 2.
$T(n) = n \lg n$ ∴ $T(n) = \Theta(n \lg n)$.

$T(n) = 2T(n/2) + \Theta(n)$ [$D(n) = 1$ $C(n) = n$]

* $n$ is in the exact powers of 2.

Recurrance: $T(n) = \begin{cases} \Theta(1), & n = 2 \\ 2T(\frac{n}{2}) + \Theta(n) & \text{otherwise, } n > 1, n = 2^k \end{cases}$ base case

$T(n=2) = 2 \cdot \lg_2 2 = 2$

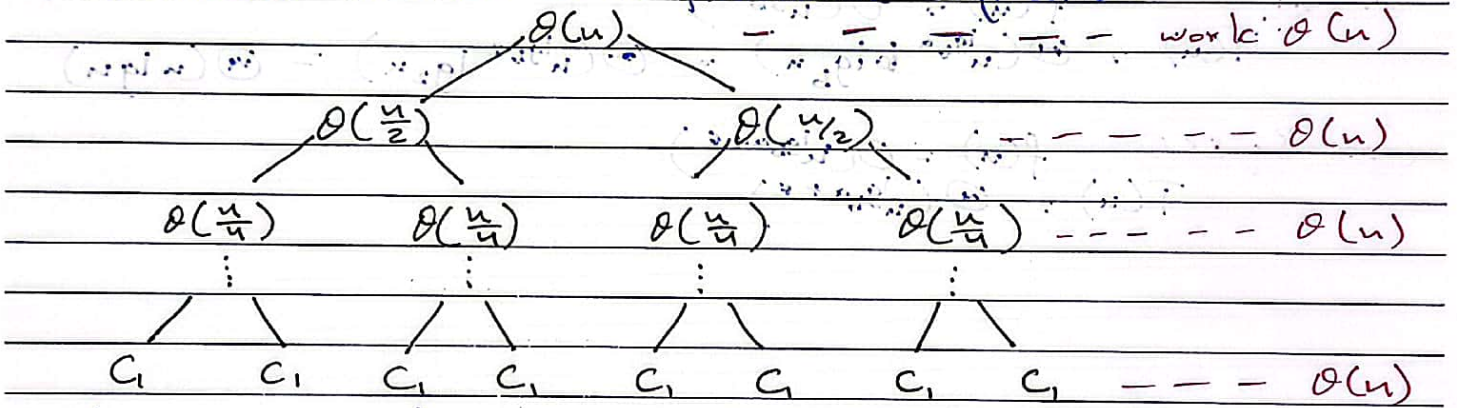
$T(2n) = 2n \lg 2n$

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

$= 2(n \lg \frac{2n}{2}) + 2n = 2n[\lg 2n - \lg_2 2] + 2n$

$T(2n) = 2n[\lg_2 2n - 1] + 2n = 2n \lg_2 2n$

* This type of equation is called a recurrence equation, recurrance relation or a functional equation.




* Assuming $n$ to be some power of 2, then would be $n$ leaves, Then the total time would be:

$T(n) = c_2 n \lg(n) + c_1 n$

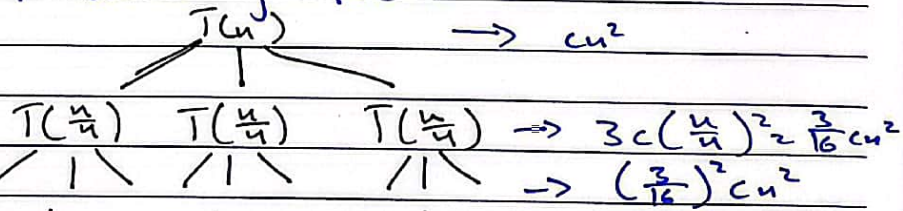
$T(n) \Rightarrow \Theta(n \lg n)$.

What if 'a' & 'b' are not same? $a \neq b$.

↳ Different branching & dividing factors.

$T(n) = 3T(\frac{n}{4}) + cn^2$

$\therefore a = 3, \; b = 4$




$h = \lg_4 n$ (height)

leaves $= 3^h = 3^{\lg_4 n}$

$= n^{\lg_4 3}$

work done by leaves: $\Theta(n^{\lg_4 3})$

$T(n) = cn^2 + \frac{3}{16}cn^2 + (\frac{3}{16})^2 cn^2 + \cdots + (\frac{3}{16})^{\lg_4 n - 1} cn^2 + \Theta(n^{\lg_4 3})$

↳ Summation of Geometric Series

$T(n) = \sum_{j=0}^{\lg_4 n - 1} (\frac{3}{16})^j cn^2 + \Theta(n^{\lg_4 3}) = \frac{1}{1 - \frac{3}{16}cn^2} + \Theta(n^{\lg_4 3})$

$\Rightarrow T(n) = \Theta(n^2)$

**Master Theorem:** If $T(n) = aT(\lceil n/b \rceil) + O(n^d)$ for some constants $a > 0$, $b > 1$ & $d \geq 0$, then

$$T(n) = \begin{cases} O(n^d) & \text{---} \; d > \log_b a \\ O(n^d \log(n)) & \text{---} \; d = \log_b a \\ O(n^{\log_b a}) & \text{---} \; d < \log_b a \end{cases}$$

$$T(n) = \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) + \Theta(n^{\log_b a})$$

**Case 1:** $\varepsilon > 0$ $\quad f(n) = O(n^{\log_b a - \varepsilon})$, $\quad \varepsilon > 0$

$T(n) = \Theta(n^{\log_b a})$

**Case 2:** $\varepsilon = 0$ $\quad f(n) = O(n^{\log_b a})$

$T(n) = \Theta(n^{\log_b a} \log_b n) = \Theta(n^{\log_b 2} \lg_2 n) = \Theta(n \lg n)$

**Case 3:** $\varepsilon < 0$ $\quad f(n) = O(n^{\log_b a + \varepsilon})$

$T(n) = \Theta(n^{\log_b a + \varepsilon})$

## Maximum Sub-Array Problem:

* Array of size $n$, find the contiguous subarray with the largest sum.

arr = [2, -3, 7, -3, 4, 6, -10]

~~Largest~~ Max Sub array.

* Brute force takes $O(n^2)$
  - ↳ Total sub arrays are $n(n+1)/2$
* Divide & Conquer ⇒ similar to merge sort.
  - ↳ $O(n \lg n)$ time.
  - → Split array into two parts.
    - ⇒ compute max on left
    - ⇒ compute max on right
    - ⇒ compute max through middle.

## ~~Karatsuba~~ Multiplying Long Integers:

Karatsuba's Algorithm:

↳ Multiplying two $n$-digit numbers takes $n^2$ time.

↳ ~~3 multiplications~~ op.

↳ Divide & Conquer Approach → $n^{\lg 3}$ time.

* Assume $n$ is in the powers of 2.

$X$ : |A|B| $\quad X = A \cdot 2^{n/2} + B$

$Y$ : |C|D| $\quad Y = C \cdot 2^{n/2} + D$

($n$-bit)

eg. $X = 9724 = \underbrace{97}_{X_H} \times \underbrace{10^2}_{r^{n/2}} + \underbrace{24}_{X_L}$

$\Rightarrow X = X_H r^{n/2} + X_L$

$Y = Y_H r^{n/2} + Y_L$

Then $XY = (X_H r^{n/2} + X_L)(Y_H r^{n/2} + Y_L)$

$= X_H Y_H r^n + (X_H Y_L + X_L Y_H) r^{n/2} + X_L Y_L$

* 4 Mults, 3 additions.

$T(n) = 4(n/2) + O(n) \qquad T(1) = 1$

$\Rightarrow T(n) = O(n^2)$

Gauss's Trick: $bc + ad = (a+b)(c+d) - ac - bd$

*Karatsuba found & realised that instead of "four Multis" we can do it in 3.

Then: $XY = (X_H Y_H) r^n + (X_H Y_L + X_L Y_H) r^{n/2} + X_L Y_L$

$a$ $e$ $d$

3 Sub Probs: $XY = ar^n + er^{n/2} + d$

$\Rightarrow e = [(X_H + X_L)(Y_H + Y_L) - a - d]$

$\because T(n) = 3T(n/2) + O(n)$

$\Rightarrow T(n) = O(n^{1.584})$

for $n \geq 500$ bits

# Graphs & Graph Algorithms.

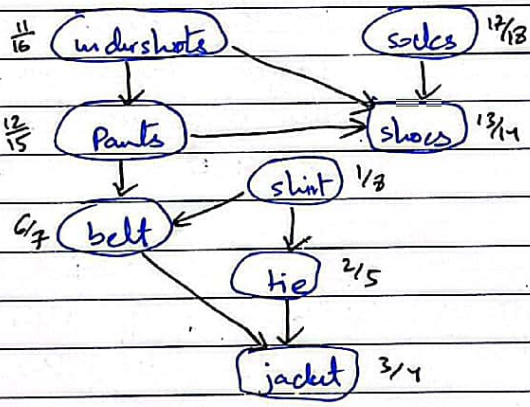
→ Graph is a pair of set of vertices & edges.
$G = (V, E)$ where, $V = \{v_1, v_2, v_3, \dots, v_n\}$
$E = \{(v_i, v_j) \text{ b.c. } i, j \le n, i \ne j\}$

→ Path in an undirected graph is a sequence $v_1, v_2, \dots, v_k$ of nodes s.t. $\{v_i, v_{i+1}\} \in E$, $i = 1, 2, \dots, k-1$
* A path is a cycle of $k \ge 2$ & first $(k-1)$ nodes are disjoint.

Directed Acyclic Graph (DAG)
↳ a directed graph G is acyclic if & only if a dfs of G yields no back edges.
↳ Has no cycles.

Topological Sort → A TS of a dag $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge $(u, v)$, then u appears before v in the ordering.
→ only applicable on dag

Eg:




Output: socks undershorts → pants → shoes watch shirt → belt tie → jacket

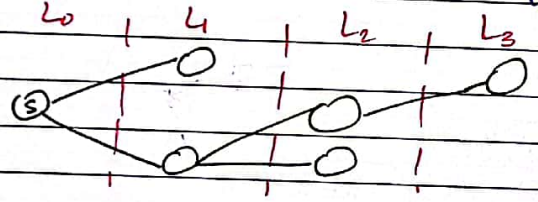
**Connectivity:** → A graph is called connected if all nodes belong to the same component.

Directed Graph — Weakly Connected Components
Strongly Connected Components (SCC)

A DG is Strongly Connected if for every two nodes $u$ & $v$, $u \rightsquigarrow v \wedge v \rightsquigarrow u$ [a path exists from $u$ to $v$ & vice versa]

Graph Traversal { DFS, BFS } Runtime is $O(V+E)$ or $O(n+m)$ ie linear time in graphs.

* Let $S$ be a node of an undirected graph $G = (V, E)$.
* BFS $(s)$ finds all nodes $t$, $t \in V \setminus S$ that are reachable from $S$.
* BFS constructs a tree from $G$, where all reachable nodes are divided into layers.




→ $L_0$ contains only $S$
→ $L_{i+1}$ contains all nodes $v$ s.t they dont belong to layers $L_0, \dots, L_i$ & that the distance [shortest] from $S$ to $V$ is $i$

Finding Connected Components in an Undirected Graph $G = (V, E)$

1- Start with an arbitrary node $S$, & run BFS$(u)$ to construct a tree '$T_1$', if $|T_1| = n$, then the graph is connected. [# of comps = 1]
2- Otherwise pick a node $u \notin T_1$, & run BFS$(u)$ to construct '$T_2$'. IF $|T_1| + |T_2| = n$, stop. Else repeat until all nodes in $G$ are visited.

# Finding Strongly Connected Components (SCC)

↳ in a directed graph $DG = (V, E)$

* Given two nodes $u, v \in V$, in a directed graph $DG = (V, E)$, define an equivalence relation 'SCC'.

SCC is an equivalence relation.

i) Reflexive, $s \rightsquigarrow s$ $(s \in V)$

ii) Symmetric, $s \rightsquigarrow t \wedge t \rightsquigarrow s$ $(s, t \in V)$

iii) Transitive, if $s \rightsquigarrow t \wedge t \rightsquigarrow u \Rightarrow s \rightsquigarrow u$ $(s, t, u \in V)$

* An equivalence relation partitions $s$ into disjoint sets called equivalence sinks.

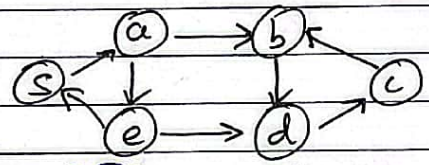
* In case of a directed graph $DG = (V, E)$, the node set $V$ is partitioned into disjoint sets (equivalence relations) or SCC by this relation.

i.e, Given two ~~sets~~ nodes, $u, v \in V$, either $u, v$ are in the same SCC or in different SCCs.

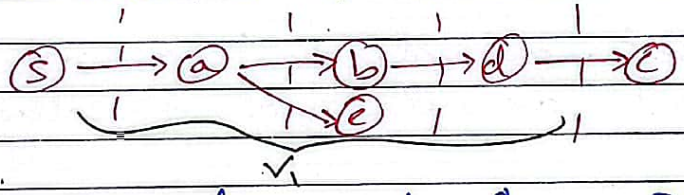
Finding SCCs :- BFS & DFS both can be used.

* Using BFS to find SCCs

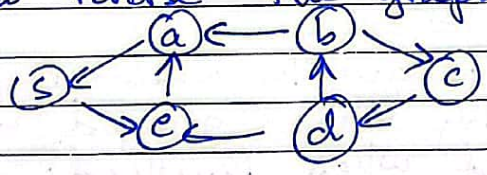
Given a $DG = (V, E)$ $DG =$

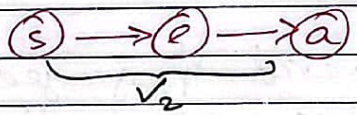



* Pick a node 's' & run BFS on it.




* Now reverse the graph, & run BFS again by 's'.




BFS: 


$V_1 \cap V_2 = \{s, e, a\}$ ⇒ they belong to one SCC.

Likewise, Run BFS(B) on DG & DG reverse.

DG: (b) → (d) → (c) } $W_1$

DG rev: (b) → (a) → (s); (b) → (c) → (d) → (e) } $W_2$

$W_1 \cap W_2 = \{b, c, d\}$ → belong to one SCC.

Then we can draw the actual directed graph as:

{s, e, a} → {b, d, c} { Metanodes in a meta graph / condensed graph.

* A DAG (Directed Acyclic Graph) is a directed graph (meta) of SCCs of a DG. e.g.;

a → b; a → c } Every node in a dag is a metanode & a section of its own.

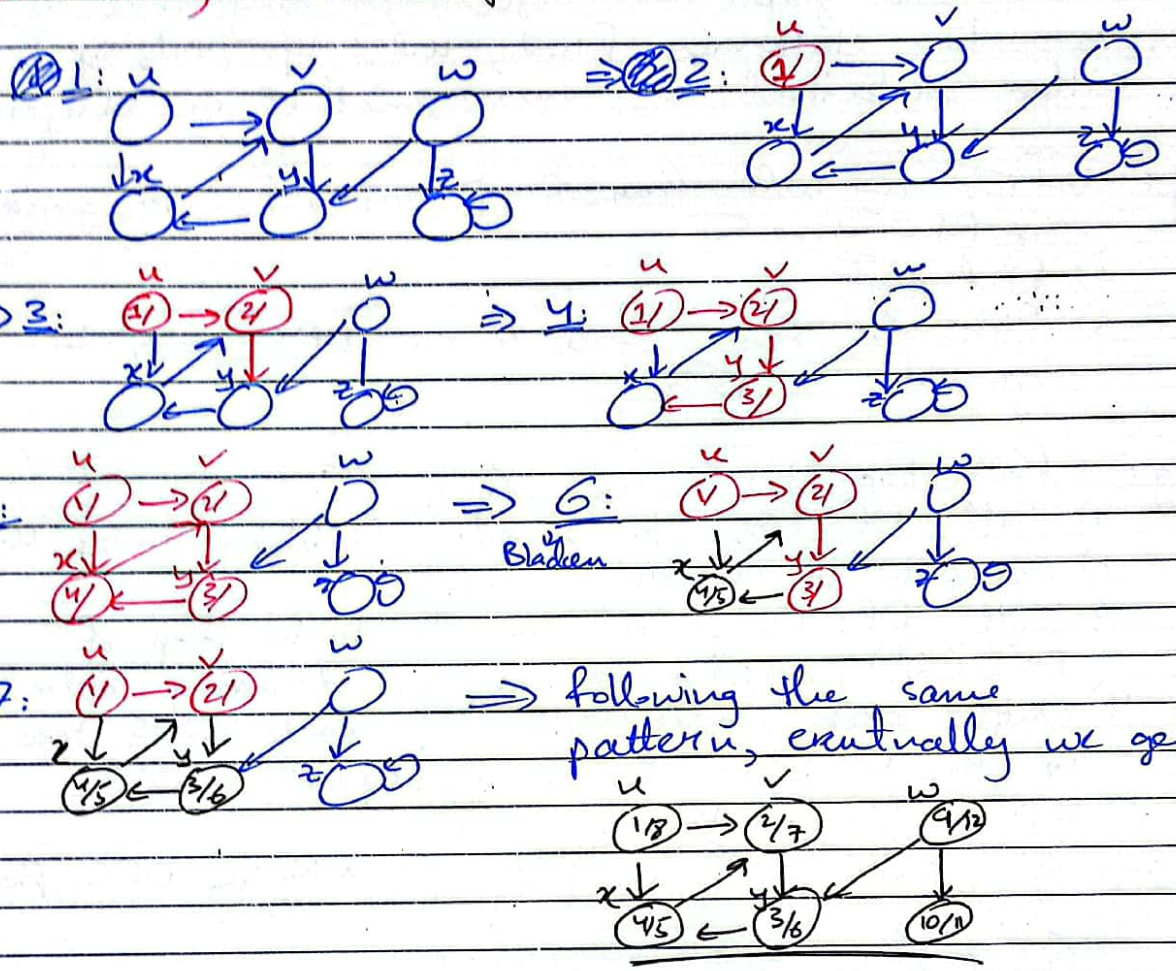
* I.e., given a dag, $G = (V, E)$, every node is a SCC.
* There is at least one node that is called a sink
* There is at least one node that is called a source

## Depth - First - Search (DFS)

* Searches 'deep' in the graph whenever possible.
* Each vertex $v$, is initially white, grayed when discovered, & blackend when finished.
* Uses two timestamps
  - ↳ first when is discovered & grayed
  - ↳ Second when is finishes examination & back tracks.
* Timestamps are b/w 1 & 2|v|.

→ Example.

In the example, blue is initially undiscovered, red is discovered, black is finished.

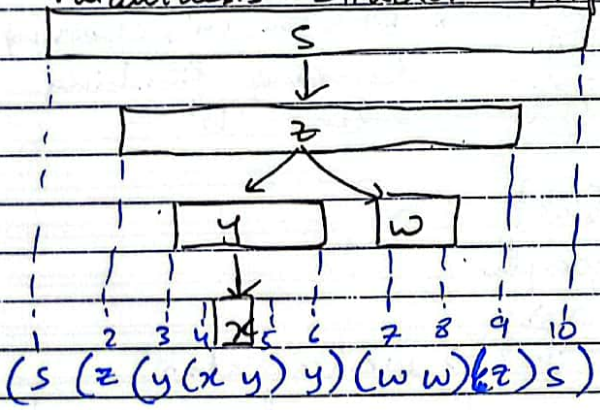



⇒ following the same pattern, eventually we get:




DFS ⇒ $\Theta(V+E)$

Parenthesis Structure Property ⇒ Interesting



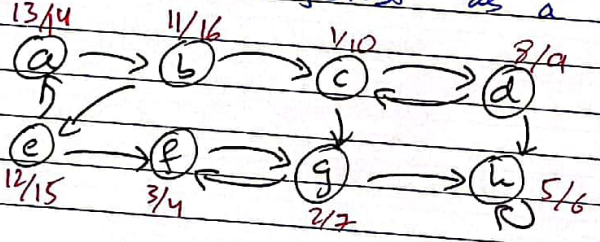
# Using DFS to find SCC.

Given a directed graph $DG = (V, E)$, for any nodes $u, v \in$ there are two intervals $[u.d, u.f]$ & $[v.d, v.f]$, are either contained in one another or disjoint.

~~Alg: 1) Pick any node from $G$ & run DFS to generate a BFS Tree~~
~~2) Find $G^T$~~
~~3) Pick the same node & run DFS on $G^T$ to generate a BFS~~

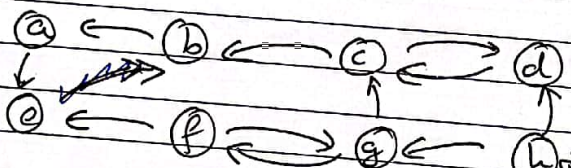
Alg: Run two DFS on $G$ & $G^T$.
1) Call DFS($G$) to compute finishing times $u.f$
2) Compute $G^T$ (Reversed Graph)
3) Call DFS($G^T$) but in the main loop of DFS($G^T$), consider vertices in decreasing order of $u.f$.
4) Output nodes/vertices of each DFS tree in the DFS forest as a seperate SCC.

eg: $G =$



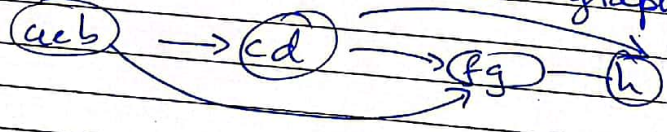

* Runing DFS from 'c'.

$G^T =$




* DFS on $G^T$, choosing decreasing finishing order. 'b'

$F = \{\{b, a, e\}, \{c, d\}, \{f, g\}, \{h\}\}$
↳ $|F| = 4$ → SCCs in our graph




→ we've also linearized our graph.

This linearization leads to Topological Sorting as discussed before.

Date:

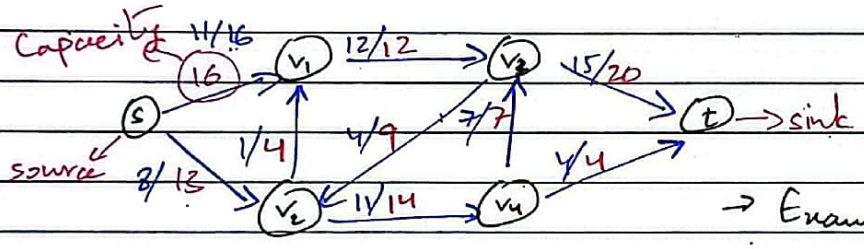
# Maximum Flows.

**Flow Network:** * A flow network $G = (V, E)$ is a directed graph in which each edge $(u, v) \in E$ has a non negative capacity. $c(u, v) \geq 0$. If an edge from $u$ to $v$ exists, an edge from $v$ to $u$ does not.

~~* If $(u, v)$ does~~

* Each flow has 2 distinguished nodes: a source 's' & a sink 't'.

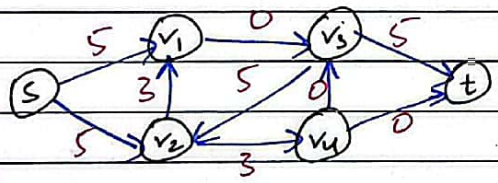
* $\forall u, v \in V$, $0 \leq f(u, v) \leq c(u, v)$ → Capacity Constraint.
$\forall u \in V - \{s, t\}$. $\sum_{u \in V} f(u, v) = \sum_{v \in V} f(v, u)$.
Flow Conservation (in-f = out-f).



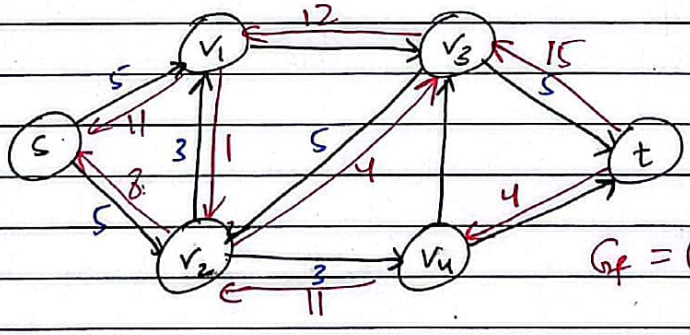

$|f| = 19$

→ Example Graph.

Residual Network of above Graph. $c_f(u, v) = \begin{cases} c(u, v) - f(u, v) \\ f(v, u) \\ 0 \end{cases}$




⇒




$G_f = (V, E_f)$

* Now how can we find the maximum flow across the network graph?

Date:

# Ford-Fulkerson Method

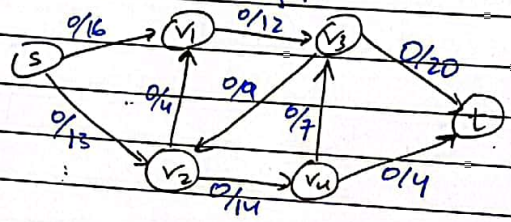
→ Finding a max-flow is an optimization problem.

Ford-Fulkerson Method (G, s, t):

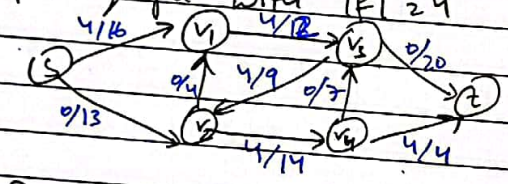
1. Initialize flow to 0
2. while there exists an augmenting path 'p' in the residual network
3. augment flow f along p
4. return f.

* An augmenting path p is a simple path from s to t in the residual network $G_f$.

$|f| = 0$

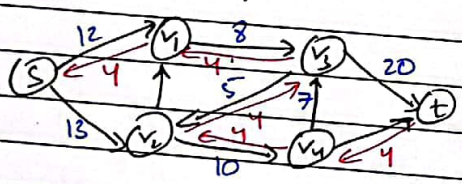



Flow graph with $|f| = 4$

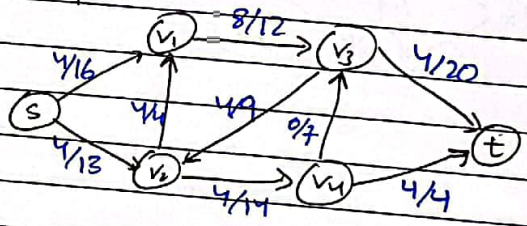



* Consider an augmenting path
$s \to v_1 \to v_3 \to v_2 \to v_4 \to t$
$\text{Min}(c(s, v_1), c(v_1, v_3), \ldots) = 4$

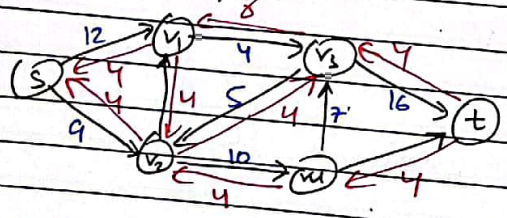
Residual Graph $G_f$:



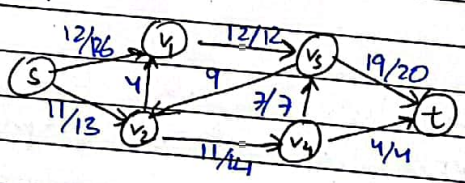

$|f| = 8$




Residual Graph for $|f| = 8$




→ Continuing this approach, our max flow is of $|f| = 23$. Its residual graph is shown below.



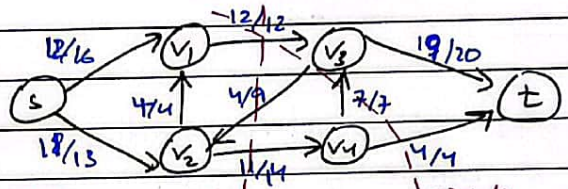
Date:

## Max-flow, Min-Cut Theorem [FFM Correctness]

↳ Max value of an s-t flow is equal to the minimum capacity over all s-t cuts.

* A cut $(S, T)$ of $G(V, E)$ is a partition of $V$ into $S$ & $T = V - S$ such that $s \in S$ & $t \in T$ (disjoint sets).
* A Min-Cut is a cut of ~~a network~~ whose capacity is minimum over all cuts of the network.

For our example graph:




$|f| = 12 + 11 = 23$

Min-Cut $= 12 + 7 + 4 = 23$

Not a Min-Cut

$\rightarrow$ Min-Cut

* To find the min-cut, check whether any edge runs a flow at full capacity. If cutting it reduces the capacity as the flow can't find a way, then cut it.
* Repeat until all such paths are blocked / cut.

* Min-Cut Capacity = Max-Flow.

* ~~Finding perfect Bipartite Matching~~


### The Min-Cut, Max-Flow Theorem:

If the total flow is $|f|$ in a flow network $G = (V, E, s, t)$, then the following three statements are equivalent:

(1) $|f|$ is a maximum flow in $G$
(2) The residual network $G_f$ contains no augmenting paths ($s \not\rightarrow t$)
(3) The total flow $|f| = c(u, v)$ $u \in S$, $v \in T$, for some cut (minimum capacity cut) $(S, T)$ of $G$

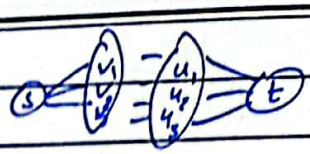
### Proof of the above:

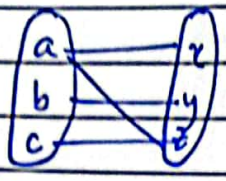
$1 \Rightarrow 2$ Suppose (1) is true, but an augmenting path 'p' exists. Then the flow found by augmenting 'p' ~~some path~~ by some $f_p$ is a flow in $G$ but with a value which is strictly greater than $|f|$. i.e., $|f| + |f_p| > f_{max}$

which is a contradiction. Hence (2) must be true.

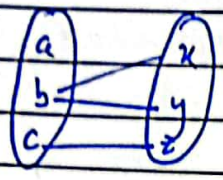
[Make similar reasonings for $2 \Rightarrow 3$ etc].

# Finding Perfect Bipartite Matching

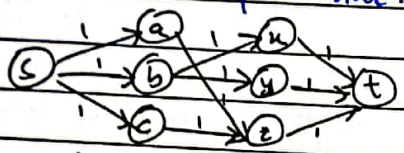


Take an eg:  what is the maximum matching in the graph on the left?

3

Take another eg:  the max possible matching in this case is 2.

We can convert the above into a flow graph by adding two nodes, a source & a sink.

$G_2$ 

Then we can solve this to get max flow as 2, which is also the maximum matching for the bipartite.

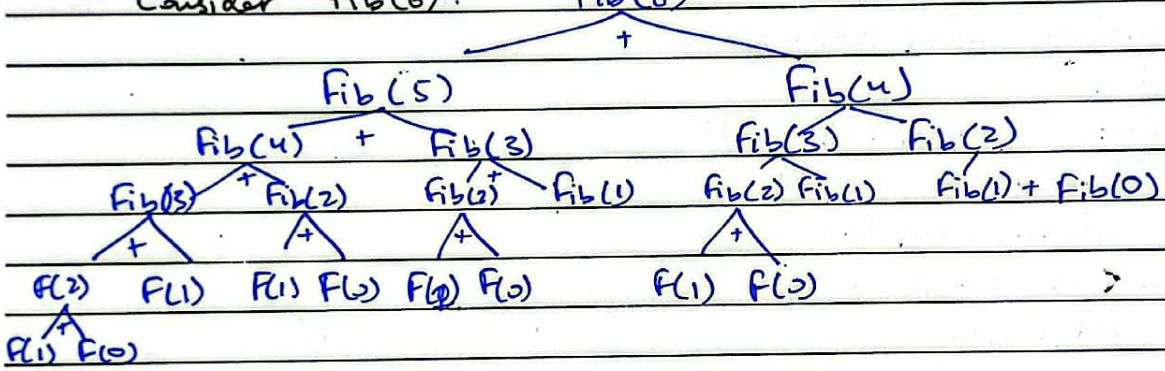
Date:

# Dynamic Programming

* Similar to D&C, divides the problem into subproblems, & recursively solves the subproblems.
* DP applies when subproblems overlap (are common.
* DP solves each subproblem only once, & stores its answer or solution [hash table / array / table], thus avoiding recomputation.
* DP → optimization problems.
* Solves linear recurrences.

Fibonacci Sequence : $Fib_n = Fib_{n-1} + Fib_{n-2}$

Consider fib(6):

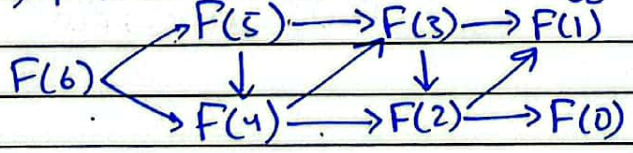



* Overlapping subproblems
* Optimal sub structure. * We can solve only the left most branch to get the whole solution.
→ These 2 properties make it suitable for DP.

Recursively this takes exponential time.
We can solve using DP efficiently in 2 ways → (1) Bottom-Up (2) → Top-Down / Memoization

~~Solving through Bottom~~ Solving through Top-Down Approach.

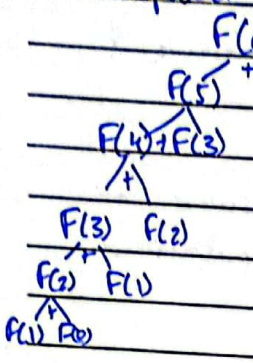
* Interestingly, Fib(6) can be represented via a dag as well, & we can solve the dag.




* This dag shows a top-down approach, since we start at the top, & solve to down until base case, then recursively solve each subprob-

Date:

*In Memoization, a memo [map] of solutions of already computed subproblems is used.




Memo:
F(0) = 0
F(1) = 1
F(2) = 1
F(3) = 2
F(4) = 3
F(5) = 5
F(6) = 8

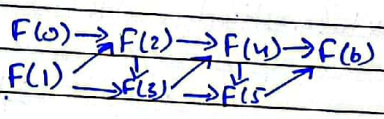
Aspo Then we can keep a memo, parse continuously over the left branch solving each subproblem, thus recursively come up, thus reducing computations.

Pseudocode:

```
fibHelper(n: int, memo: list[int])
    if memo[n] is not None
        return memo[n]
    if n <= 2: (n is less than 2)
        fib = n
    else:
        fib = fibHelper(n-1, memo) + fibHelper(n-2, memo)
    memo[n] = fib
    return fib

fib_TopDown(n: int):
    memo = [None for _ in range(n)]
    return fibHelper(n-1, memo)
```

Bottom Up Approach: Simply keep a list! Dag:




```
fib_BottomUp(n: int):
    fib = [0,1] + [0]*(n-1)
    for i <- 2 to n+1:
        fib[i] = fib[i-1] + fib[i-2]
    return fib[n]
```

So when does DP (Tabulation Method) apply? →

Date:

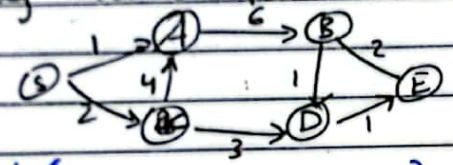
DP → When to use tabulation / bottom-up?

1) recursive approach & optimization
2) Problem can be broken into a combination of subproblems
3) Use solutions (optimal) to solve (of subprobs) the og prob.
4) Recurrence relation is linear, or has only a slight difference. DnC: $T(n) = aT(n/b) + f(n)$
   - ↳ $F(n) = F(n-1) + F(n-2)$
   - ↳ To get $F_{100}$, we need $F_{99}$ & $F_{98}$ which are only slightly smaller.
5) Overlapping subproblems.

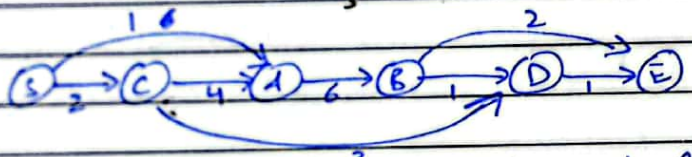
Solving DAGs.

↳ Finding Shortest Path in a ~~Dag~~ DAG (Basis of DP)

Dag:




* Run DFS on G to get the linearized, topologically sorted dag.




Claim: to find the shortest path from a source node to any node, it is sufficient to linearize the dag.

1) Smallest optimal subproblem: $dist(S,S) = 0$
   - ↳ $dist(S) = 0$ which is our base case.
2) Start building the solution to bigger problems using optimal solutions to subproblems.

We get: (↱ init by ∞)

$$dist(v) = \min\{dist(v),\ dist(u) + cost(u,v)\}$$

($dist(v)$: from source; ↳ $(u,v) \in E$)

recursive definition

1) $dist(S) = 0$ ← base case.
2) $dist(C) = dist(S) + c(S,C) = 0 + 2 = 2$ ∵ $c(u,v) = cost(u,v)$
3) $dist(A) = \min\{dist(S) + c(S,A),\ dist(C) + c(C,A)\}$
   $= \min\{0+1,\ 2+4\} = \min\{1,6\} = 1$
4) $dist(B) = dist(A) + c(A,B) = 1 + 6 = 7$
5) $dist(D) = \min\{dist(B) + c(B,D),\ dist(C) + c(C,D)\}$
   $= \min\{7+1,\ 2+3\} = \min\{8,5\} = 5$
6) $dist(E) = \min\{dist(D) + c(D,E),\ dist(B) + c(B,E)\} = \min\{5+1,\ 7+2\}$
   $= \min\{6,9\} = 6$

Date: ....

⇒ Bellman's idea went beyond just finding the shortest path in a dag. Imagine that the nodes are subproblems of larger problem, and finding the shortest path from the source [base-case] to every node the subproblem.

⇒ All Dynamic Programming problems can be reduced to the shortest path problem in a dag

Finding the Longest Common Subsequence (LCS)

· LCS (X, Y) where X & Y are strings.

eg. Let: $X = \langle A, B, C, B, D, B, BA \rangle$

$Y = \langle B, D, C, A, B, A \rangle$

Then $Z = \langle B, C, B, A \rangle$ is a common subsequence which is also one of the longest common subsequences.

$Z' = \langle B, D, B, A \rangle$ is also an LCS

* Brute force results in exponential time ~~subsequence~~ solution! $O(2^{n+m})$ when $n = |X|$ & $m = |Y|$ since all sequences of X are compared with all sequences of Y.

Theorem: Let $X = \langle x_1, x_2, \dots, x_m \rangle$ & $Y = \langle y_1, y_2, \dots, y_n \rangle$

Then let $Z = \langle z_1, z_2, \dots, z_k \rangle$ be any LCS of X & Y.

Then: 1) If $x_m = y_n$, then $z_k = x_m = y_n$

& $z_{k-1} = LCS(X_{m-1}, Y_{n-1})$

2) Else if, $x_m \neq y_n$ & $z_k \neq y_n$

then, $z_k = LCS(X_m, Y_{n-1})$

3) Else if, $x_m \neq y_n$ & $z_k \neq x_m$

then, $z_k = LCS(X_{m-1}, Y_n)$

~~How Suppose, $X = \langle A, B, C, B, D, A, B \rangle$ & $Y =$~~

Then intuitively we get the following cases:

$z_k = (X_0, Y_0) = \emptyset$ } Base cases

$z_k = (X_m, Y_0) = \emptyset$ } for our

$z_k = (X_0, Y_m) = \emptyset$ } recursion.

* From the above we can deduce that LCS (X, Y) contains within the LCS of prefixes of 2 sequences. Hence we have an optimal sub structure.

Date:

## Overlapping Subproblems:

Let $X_2 \langle A, X, Y, T \rangle$ & $Y_2 \langle A, Y, Z, X \rangle$

L("AXYT", "AYZX")

- L("AXY", "AYZX")
  - L("AX", "AYZX")
  - L("AXY", "AYZ")
- L("AXYT", "AYZ")
  - L("AXY", "AYZ")
  - L("AXYT", "AY")

* We already have overlapping subproblems. *
* Further expanding the recursion tree would show more overlapping subproblems. Hence we can use DP.

* Let $C[i,j]$ be the length of $LCS(X_i, Y_j)$
if $i=0$ or $j=0$, then $C[i,j]=0$ (base-case)

or

$$C[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ c[i-1,j-1]+1 & \text{if } i,j>0 \text{ \& } x_i = y_i \\ \max\{c[i,j-1], c[i-1,j]\} & \text{if } i,j>0 \text{ \& } x_i \neq y_j \end{cases}$$

⇒ Instead of recursion, we fill up a table → tabulation, approach [bottom-up].

Consider, $X_2$ "ABCB" & $Y_2$ "BDCAB"

filling the table:

1) Fill the initial cases first. Call trivial subprobs / base cases)
2) Dx any a row major / column major order.

$C[m,n] = 3$ (length of LCS)

& LCS (X, Y) = BCB.

The table is as follows:

| | j | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|
| i | | $y_i$ | B | D | C | A | B |
| 0 | $x_i$ | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | A | 0 | | | | | |
| 2 | B | 0 | | | | | |
| 3 | C | 0 | | | | | |
| 4 | B | 0 | | | | | |

① Small sub probs with optimal solutions.

We move row wise, since A has ← no matches, we put 0, then on the match, we put 1 & carry that value forwards until the next match.

② →

| | j | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|
| i | | $y_i$ | B | D | C | A | B |
| 0 | $x_i$ | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | A | 0 | 0 | 0 | 0 | 1 | 1 |
| 2 | B | 0 | | | | | |
| 3 | C | 0 | | | | | |
| 4 | B | 0 | | | | | |

Date:

(3)

| | | j | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|---|
| i | | $y_i$ | B | D | C | A | B | |
| 0 | $x_i$ | 0 | 0 | 0 | 0 | 0 | 0 | |
| 1 | A | 0 | 0 | 0 | 0 | 1 | 1 | |
| 2 | B | 0 | 1 | 1 | 1 | 1 | 2 | |
| 3 | C | 0 | | | | | | |
| 4 | B | 0 | | | | | | |

→ We start on the second row then with 'B', & on the match set the val to 1. It gets carried till another match that "B" = 2. Bor

Now the interesting thing happens!

(4)

| | | i | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|---|
| i | | $y_i$ | B | D | C | A | B | |
| 0 | $x_i$ | 0 | 0 | 0 | 0 | 0 | 0 | |
| 1 | A | 0 | 0 | 0 | 0 | 1 | 1 | |
| 2 | B | 0 | 1 | 1 | 1 | 1 | 2 | |
| 3 | C | 0 | 1 | 1 | 2 | 2 | 2 | |
| 4 | B | 0 | | | | | | |

* We follow the rules, & since i,j > 0 & $x_i \neq y_j$, we set c[i,j] to max{c[i,j-1], c[i-1,j]} which was 1 which gets carried the same way until our second match of 'C' when the value gets updated & then carried, showing LCS to be 2 yet ["BC"].

(5)

| | | j | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|---|
| i | | $y_i$ | B | D | C | A | B | |
| 0 | $x_i$ | 0 | 0 | 0 | 0 | 0 | 0 | |
| 1 | A | 0 | 0 | 0 | 0 | 1 | 1 | |
| 2 | B | 0 | 1 | 1 | 1 | 1 | 2 | |
| 3 | C | 0 | 1 | 1 | 2 | 2 | 2 | |
| 4 | B | 0 | 1 | 1 | 2 | 2 | 3 | |

* Finally our last string "B", which starts from '1' by c[i-1,j], then gets the value 2 similarly, finally being incremented on match with the last 'B'. Hence, the final value is our LCS which is 3. LCS(X,Y) = 3 ["BCB"].

~~* This shows how~~ * We back track on the matched indices to get our longest subsequence "BCB".

* This shows how we can use DP to solve LCS in O(m x n) from $O(2^{m+n})$. Quite convenient & efficient!

Pseudocode:

```
Lcs_tabulation(X, Y, m, n): → Returns only the length
    table = [[None]*(n+1) for i in range(m+1)]
    for i ← 0 to m+1:
        for j ← 0 to n+1:
            if i==0 or j==0: table[i][j] = 0
            elif X[i-1] == Y[j-1]:
                table[i][j] = 1 + table[i-1][j-1]
            else tab[i][j] = max{table[i-1][j], table[i][j-1]}
    return table[m][n]
```

Date:

# The 0/1 Knapsack Problem

No fractions

* Each item represented by a pair <value, weight>
* Knapsack can accomodate items with a total weight of at most capacity W.

Our question is:
"There are 'n' items to pick from, of weight $w_1, w_2, \dots, w_n$ & value $v_1, v_2, v_3, \dots, v_n$. What's the most valuable combination of items we can accomodate in our Knapsack?"

⇒ Let, I be a vector of length n, [set of items],
⇒ Let, V be a vector of length n to indicate whether the item ends up in the Knapsack.
if $V[i]=1$, then $I[i] \in$ Knapsack.

⇒ Maximize $K(W) = \sum_{i=0}^{n-1} V[i] \times I[i] \leq W$ [W = capacity]

With Brute Force: * Enumerate all possible combinations
↳ Exponential time.
* Guarantees optimal solution but in exponential time.

2 Versions of Knapsack: 1) Items added with repetition
2) Items added without repetition.

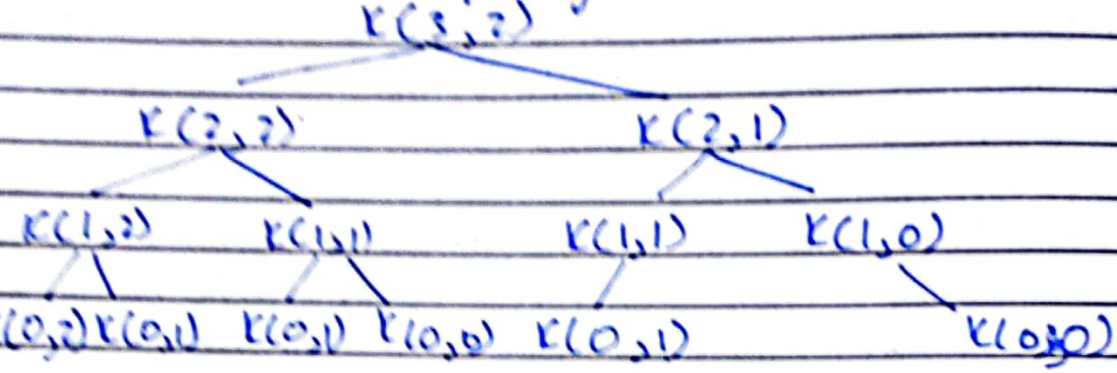
| Item | Weight | Value |
|---|---|---|
| 1 | 6 | 30 |
| 2 | 3 | 14 |
| 3 | 4 | 16 |
| 4 | 2 | 9 |

1) With Repetition
'a' & 2 'd's.
$\sum w_i = a_{wt} + d_{wt} + d_{wt}$
$= 6 + 2 + 2 = 10$
$K(W) = 30 + 9 + 9 = 48$

2) Without Repetition
Pick 'a' & 'c'.
$\sum w_i = a_{wt} + c_{wt}$
$= 6 + 4 = 10$
$K(W) = 30 + 16 = 46$

Recursive Optimal Structure.

Consider 3 items each of weight 1 & some value $v_i$




* We can already see overlapping subproblems. Hence DP is applicable!

Recursively: * Let i be an optimum item in the optimal solution, K(w)

* Removing 'i' still results in the Knapsack having optimal solution for weight/capacity W-Wi & total value K(W-Wi).

* Then, $K(W) = K(W-W_i) + V_i$ for some i

(K(W): optimal sol; K(W-Wi): optimal sol for the subproblem; Vi: value of i)

But which i to remove? Try all possibilities.

* $K(W) = \max_{i:\, W_i \le W} \{K(W-W_i) + V_i\}$ [Base Case: K(0) = 0]

with repetition, size by cap only, 1-D table we get.

| w | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| K(w) | 0 | 0 | 9 | 14 | 15 | 23 | 30 | 32 | 39 | 44 | 48 |

Pseudocode for Recursive

```
Knapsack(W, weights, values, n)
    if n is 0 or W is 0: Returns
        return 0
    if weights[n-1] > W:
        return Knapsack(W, weights, values, n-1)
    else:                                   W-weights[n-1]
without repetition → return max{values[n-1] + Knapsack(W, weights, values, n-1),
                                Knapsack(W, weights, values, n-1)}
with repetition → return max{values[n-1] + Knapsack(W-weights[n-1], weights, values, n),
                             Knapsack(W, weights, values, n-1)}
```

Date:

# Matrix-Chain Multiplication

* Matrix Multiplication is not Commutative, however, it is Associative.
  i. $A \times B \neq B \times A$
  ii. $(A \times B) \times C = A \times (B \times C)$.
* Associativity allows us to find different ways to compute the product of a series of matrices.
* Then given a chain of matrices $\langle A_1, A_2, \ldots, A_n \rangle$ where we have 'n' matrices (not necessarily square) what is the most efficient way to compute their product?

Consider:

$A \times B \times C \times D$

$50 \times 20 \quad 20 \times 1 \quad 1 \times 10 \quad 10 \times 100$

| Parenthesization | Cost Computation | Cost |
|---|---|---|
| $A \times ((B \times C) \times D)$ | $20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100$ | 120,200 |
| $(A \times (B \times C)) \times D$ | $20 \cdot 1 \cdot 10 + 50 \cdot 20 \cdot 10 + 50 \cdot 10 \cdot 100$ | 60,200 |
| $(A \times B) \times (C \times D)$ | $50 \cdot 20 \cdot 1 + 1 \cdot 10 \cdot 100 + 50 \cdot 1 \cdot 100$ | 7,000 |

* Order makes a big difference in the cost!

For 2 Matrices $A_1, A_2$ only 1 Parenthesization.

For 3 Matrices $A_1, A_2, A_3$:

1. $A_1 \times (A_2 \times A_3)$
2. $(A_1 \times A_2) \times A_3$

For 4 Matrices $A_1, A_2, A_3, A_4$:

1. $A_1 (A_2 (A_3 A_4))$
2. $A_1 ((A_2 A_3) A_4)$
3. $(A_1 A_2)(A_3 A_4)$
4. $(A_1 (A_2 A_3)) A_4$
5. $((A_1 A_2) A_3) A_4$

* For 5 Mats, we get 14 ways
* For 6 Mats, 42 ways!
* For 7 Mats, 132 ways!

Following this pattern, we get a really nice & interesting series: 1, 2, 5, 14, 42, 132, 479, 1430, 4862 --- Given by the formula:

$C_n = \frac{1}{n+1} \binom{2n}{n}$ → The Catalan Numbers

Date: .....

## Solving Knapsack Using Dynamic Programming

* We need to keep track of K(W), but also whether an item j ($1 \le j \le n$) from n items is already in the optimal solution or not.
* 2-D problem of size $(n+1) \times (W+1)$.
* Decision Problem
  * Keep a memo to track filled in values, → top down
  * Keep a 2D table to track → bottom up.

$$K(W, j) = \max \begin{cases} K(W - w_j, j-1) + v_j & ; w_j \le W \\ K(W, j-1) \end{cases}$$

Pseudocode → Top-Down Approach

```
memo = [[-1 for _ in range(W+1)] for _ in range(n+1)]
Knap_Memo(W, weights, vals, n):
    if n is 0 or W is 0:
        return
    if memo[n][W] != -1:
        return memo[n][W]
    if weights[n-1] <= W:
        memo[n][W] = max { Vals(n-1) + Knap_Memo(W-weights[n-1], weights, vals, n-1)
                         { Knap_Memo(W, weights, vals, n-1)
        return memo[n][W]
    else:
        memo[n][W] = Knap_Memo(W, weights, vals, n-1)
        return memo[n][W]
```

n for-loop repetition

Pseudocode → Bottom Up Approach.

```
Knap_Bup(W, wts, vals, n):
    K = [[0 for i in range(W+1)] for _ in range(n+1)]
    for i in range(n+1):
        for w in range(W+1):
            if i is 0 or w is 0: K[i][w] = 0
            elseif wts[i-1] <= w:
                K[i][w] = max { vals[i-1] + K[i-1][w - wts[i-1]],
                              { K[i-1][w] }
            else K[i][w] = K[i-1][w]
    return K[n][W]
```

Date:

* Then for any number of Matrices 'n', the possible ways to parenthesize them are:

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

* Interestingly, the Catalan Numbers also follow the recurrance relation:

$$C_{n+1} = C_0 C_n + C_1 C_{n-1} + \cdots + C_n C_0 = \sum_{k=0}^{n} C_k C_{n-k}$$

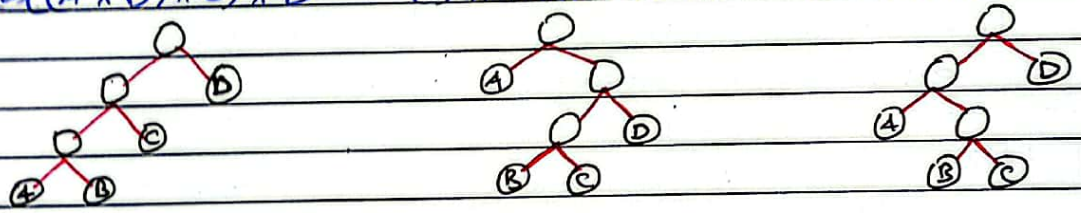
which implies that we are going to be having overlapping subproblems!

Solving the Problem - Binary Trees

→ Every particular parenthesization involves 2 matrices, hence can be represented by a binary tree.

* Consider A, B, C, D.

(a) $(A \times B) \times C) \times D$ : (b) $A \times ((B \times C) \times D)$ (c) $(A \times (B \times C)) \times D$




* We get full binary trees with 'n' leaves which grow exponentially. Thus Brute Force is bad.

* For a tree to be optimal, its subtrees must also be optimal. Since we want to minimize our cost, we have a minimization problem.

$C(i,j)$ = min cost of prod of $A_i \times A_{i+1} \times \ldots \times A_j$

$1 \le i \le j \le n$.

* Each subprob of size $|j-i| \rightarrow$ # mat mults
* Base Case when $i=j \Rightarrow C(i,j)=0$
* For $j>i$, consider optimal subtree for $C(i,j)$. The first branch splits into $A_i \times \ldots \times A_k$ & $A_{k+1} \times \cdots \times A_j$ $\exists k$ st. $i \le k < j$
* Then cost = $C(i,k) + C(k+1,j) + m_{i-1} \cdot m_k \cdot m_j$ & we need to find the smallest splitting point 'k'. st.

$$C(i,j) = \min_{i \le k < j} \{C(i,k) + C(k+1,j) + m_{i-1} \cdot m_k \cdot m_j\}$$

Date:

Pseudocode - Recursive / Brute Force

```
MatChainMult(chain, i, j)
    if i == j:
        return 0
    min ← -inf
    for k ← i - j:
        count = MatChainMult(chain, i, k) + MatChainMult(chain, k+1, j)
                + chain[i-1] * chain[k] * chain[j]
        min = min{min, count}
    return min
```

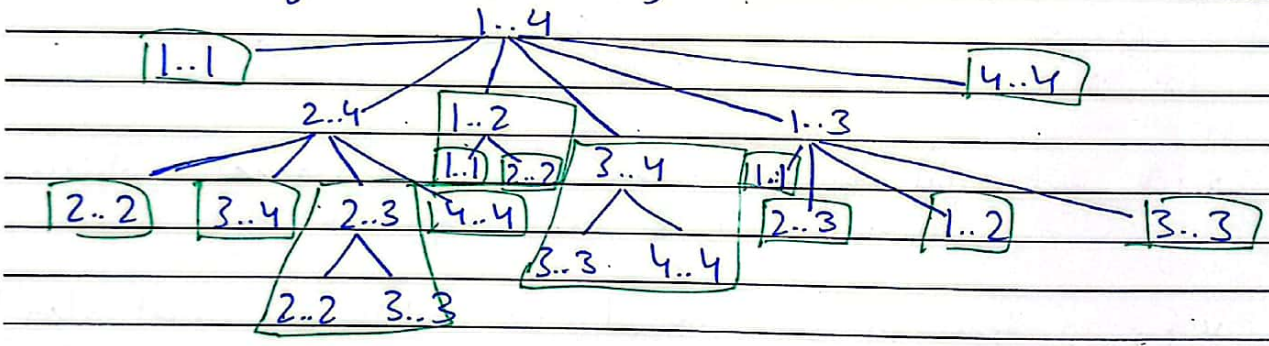
Solving Using Dynamic Programming

1) Memoization

Consider 4 Matrices & Dimensions:

1×2, 2×3, 3×4, 4×3. [arr = [1, 2, 3, 4, 3]]

Then we get the following Recursion Tree




We can already see so many overlapping subproblems.

* Keep a memo of size n*n

```
memo = [[-1 for - in range(n)] for - in range(n)]
def MatChainMemo(chain, i, j):
    return 0 if i == j
    return memo[i][j] if memo[i][j] != -1

    memo[i][j] ← -inf
    for k ← i - j:
        count = MatChainMemo(chain, i, k) + MatChainMemo(chain[k+1:], k+1, j)
                + chain[i-1] * chain[k] * chain[j]
        memo[i][j] = min{memo[i][j], count}
    return memo[i][j]
```

Date:

## 2) Tabular - Bottom Up Approach.

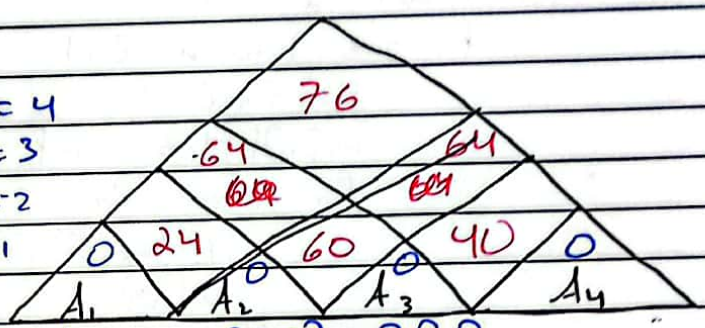
Consider 4 mats: $A_1, A_2, A_3, A_4$

$A_1$ ($P_0 \times P_1$) $2 \times 3$ × $A_2$ ($P_1 \times P_2$) $3 \times 4$ × $A_3$ ($P_2 \times P_3$) $4 \times 5$ × $A_4$ ($P_3 \times P_4$) $5 \times 2$

Recall: $m[i,j] = \begin{cases} 0 & i = j \\ \min\{m[i,k] + m[k+1,j] + P_{i-1} P_k P_j\} & i < j \end{cases}$

Create a table!

| i \ j | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 24 | 64 | 76 |
| 2 | | 0 | 60 | 64 |
| 3 | | | 0 | 24 |
| 4 | | | | 0 |




$A_1 : A_2 := m[1,2] + P_0 P_1 P_2$

$= 0 + 2 \cdot 3 \cdot 4 = 24$

$: m[1,2]$

$m[1,1] + m[2,2]$

$0 + 0$

$A_2 : A_3 = m[2,3] + P_1 P_2 P_3 = 0 + 3 \cdot 4 \cdot 5 = 60$

$m[2,2]$ $m[3,3]$

$0$ $0$

$A_3 : A_4 = m[3,4] + P_2 P_3 P_4 = 0 + 4 \cdot 5 \cdot 2 = 40$

$m[3,3]$ $m[4,4]$

For $u = 3$:

$A_1 A_2 A_3$ → $A_1 : A_2 A_3$ or $A_1 A_2 : A_3$

$= \min \begin{cases} m[1,1] + m[2,3] + P_0 P_1 P_3 \\ m[1,2] + m[3,3] + P_0 P_2 P_3 \end{cases}$

$= \min\{0 + 60 + 2 \cdot 3 \cdot 5,\ 24 + 0 + 2 \cdot 4 \cdot 5\}$

$= \min\{60 + 30,\ 24 + 40\}$

$= \min\{90, 64\} = 64$

fill the rest of table in the $\Delta$

Date:

Pseudocode for Tabular / Bottom Up.

```
MatChainBup(chain):
    n = length(chain)
    m = [[0 for _ in range(n)] for _ in range(n)]
    for i ← 1 to n:
        m[i][i] = 0
    for l ← 2 to n:
        for i ← 1 to n-l+1:
            j = i+l-1
            m[i][j] ← -inf
            for k ← i to j:
                q = m[i][k] + m[k+1][j] + chain[i-1]*chain[k]*chain[j]
                m[i][j] = min{q, m[i][j]}
    return m[1][n-1]
```

* We get a 2-D table, each of whose entries takes $O(n^0)$ time to compute.
* $O(n^2)$ for table & $O(n)$ for each entry.
* Overall runtime = $O(n^3)$!

Interesting things ~~resulting~~ related to Catalan Numbers [Optional]:

Pascals Triangle → Binomial Theorem

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
1   5   10  10  5   1
1 6  15  20  15  6  1
```

Sierpenski's Triangle

```
          1
        1   1
      1   0   1
    1   1   1   1
  1   0   0   0   1
1   1   0   0   1   1
1 0  1   0   1  0  1
```

* Replace even numbers with 0s, replace odd numbers with 1s.