Algorithm Analysis + Design (AAD)

Algorithms

It is any well defined computational procedure that takes some value or a set of values as input and produces some value or a set of values as output

Algorithm is a tool for solving well specified computational problem.

eg. we might need to sort a sequence of numbers into nondecreasing order. Here is how we formally define the sorting problem:-

Input: A sequence of n numbers $(a_1, a_2 \ldots a_n)$

Output: A permutation (reordering) $(a_1', a_2', \ldots a_n')$ of input sequence such that

$a_1' \leq a_2' \leq \ldots \leq a_n'$

Instance of a problem:- consists of input needed to compute a solution to the problem.

Scanned by CamScanner

eg. input sequence (31, 41, 59, 26, 41, 58), a sorting algo returns as output the sequence (26, 31, 41, 41, 58, 59). Such an input sequence is called an instance of sorting problem

An algorithm is said to be correct if for every input instance, it halts with the correct output.

An incorrect algorithm might not halt at all on some input instances or it might halt with an answer other than desired one.

Growth of Functions

The order of growth of running time of an algorithm gives a simple characterization of algorithm's efficiency and also allows us to compare the relative performance of alternative algorithms.

Scanned by CamScanner

Asymptotic notation

The notations we use to describe the asymptotic running time of an algo are defined in terms of functions whose domains are set of natural numbers, $N = \{0, 1, 2 - -\}$.

Asymptotic Tight Bound

Θ- notation (Theta notation)

Let $f(n)$ + $g(n)$ be functions with domain $\{1, 2, 3 ___\}$ we write $f(n) = \Theta(g(n))$ and say that $f(n)$ is of order $g(n)$ if there exists positive constant c_1 + c_2 and n_0 such that

$$0 \le c_2 g(n) \le f(n) \le c_1 g(n) \quad \text{for } n \ge n_0$$

eg. $f(n) = 5n^2 + 6n + 7$

$$0 \le 5n^2 \le f(n) \le 18n^2 \quad \text{for } n \ge 1$$

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

Scanned by CamScanner

Asymptotic upper bound:-

Big-Oh Notation:- Let $f(n)$ & $g(n)$ be functions with domain $\{1, 2, 3 \ldots\}$ we write $f(n) = O\,g(n)$ and say that $f(n)$ is of order atmost $g(n)$ if there exists +ve constants c and n_0 s.t.

$$0 \le f(n) \le c_1 g(n) \text{ for } n \ge n_0$$

eg. $f(n) = 5n^2 + 6n + 7$

$f(n) \le 5n^2 + 6n^2 + 7n^2$ for $n \ge 1$

$f(n) \le 18n^2$

$\Rightarrow f(n) = O(n^2)$ $\boxed{c = 18,\ n_0 = 1}$

eg. $f(n) = 5n^2 + 6n + 7$

$f(n) \le 5n^3 + 6n^3 + 7n^3$

$f(n) \le 18n^3$ $(n \ge 1)$

$\Rightarrow f(n) = O(n^3)$ $\boxed{n_0 \ge 1 \quad c \ge 18}$

$f(n) = O(n^2), O(n^3), O(n^4), O(n^5) \ldots$

$f(n) = 16 \forall$

$f(n) = O(1)$ $n_0 = 0$, $c = 16$.

Scanned by CamScanner

$f(n) = 16 * 1$

$f(n) = 16 n^0 \quad n \geq 0$

$f(n) \leq 16$

$f(n) \neq O(n \lg n)$

$f(n) \neq O(n)$

$f(n) \neq O(1)$

Asymptotic Lower bound:-

Omega Notation:- Let $f(n)$ & $g(n)$ be two functions with domain $\{1, 2, 3 \ldots\}$

If $0 \leq c_2 g(n) \leq f(n)$ for $n \geq n_0$ where c_2 and n_0 are +ve constants, then we write $f(n) = \Omega g(n)$ and say that $f(n)$ is of order atleast $g(n)$.

$f(n) = 5n^2 + 6n + 7$

$f(n) \geq 5n^2$

$f(n) = \Omega(n^2) \quad (n \geq 1)$

$f(n) \geq 6n$

$f(n) = \Omega(n)$

$f(n) \geq 7 * 1 \quad n_0 = 0$

$f(n) = \Omega(1) \quad c \geq 7$

Little-Oh-Notation:- Let $f(n)$ & $g(n)$ be 2 functions.
If $0 \le f(n) < c_1 g(n)$ for $n \ge n_0$ where c_1 & n_0 are +ve constants then we write $f(n) = o\,g(n)$ and say little-oh notation for $f(n)$ is $g(n)$.

Little-Omega-Notation:- Let $f(n)$ & $g(n)$ be two functions. If $0 \le c_2 g(n) < f(n)$ for $n \ge n_0$ where c_2 and n_0 are +ve constants, then we write $f(n) = \omega(g(n))$ and say $f(n)$ is of order little omega $g(n)$

eg. $f(n) = 5n^2 + 6n + 7$

$f(n) = O(n^3), O(n^4), O(n^5)$

$f(n) = \omega(n), \omega(1)$

$f(n) \le 5n^2 + 6n^2 + 7n^2$ for $n \ge 1$

$f(n) < 5n^3 + 6n^3 + 7n^3$

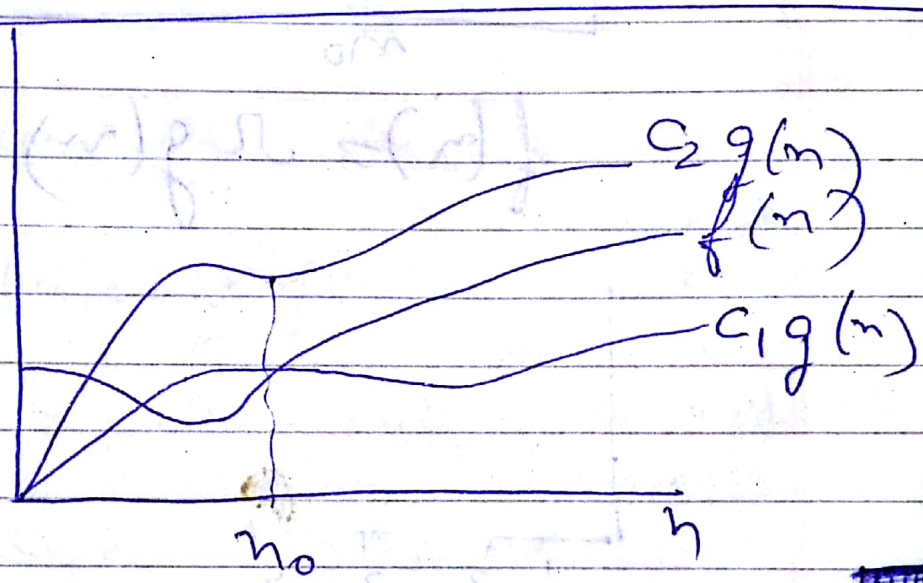
Note:-

(1.) (Big-oh notation of $f(n)$) $\cap$ (Omega notation of $f(n)$) = Theta notation of $f(n)$

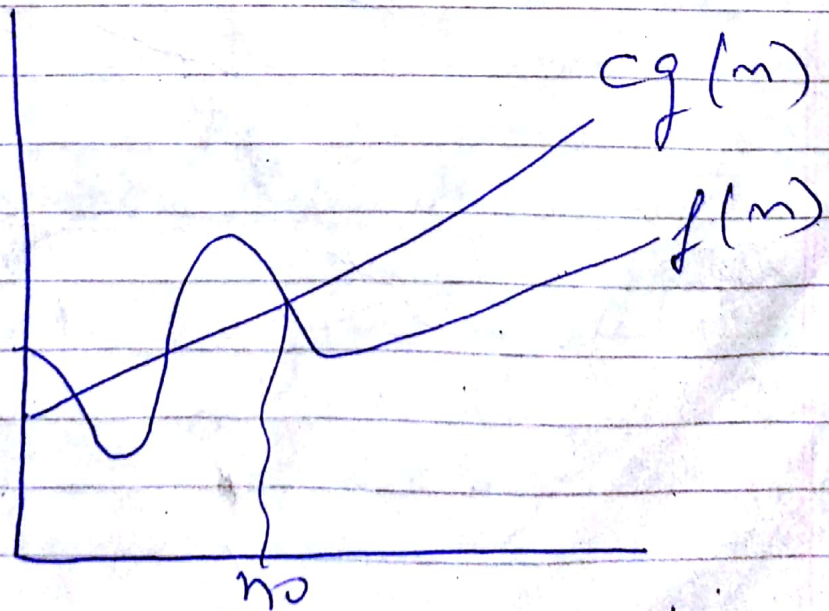
Scanned by CamScanner

(2) (Big-Oh notation) − (Theta notation) = (Little-Oh-notation)

(3) (Omega notation) − (Theta notation) = (Little-omega-notation)

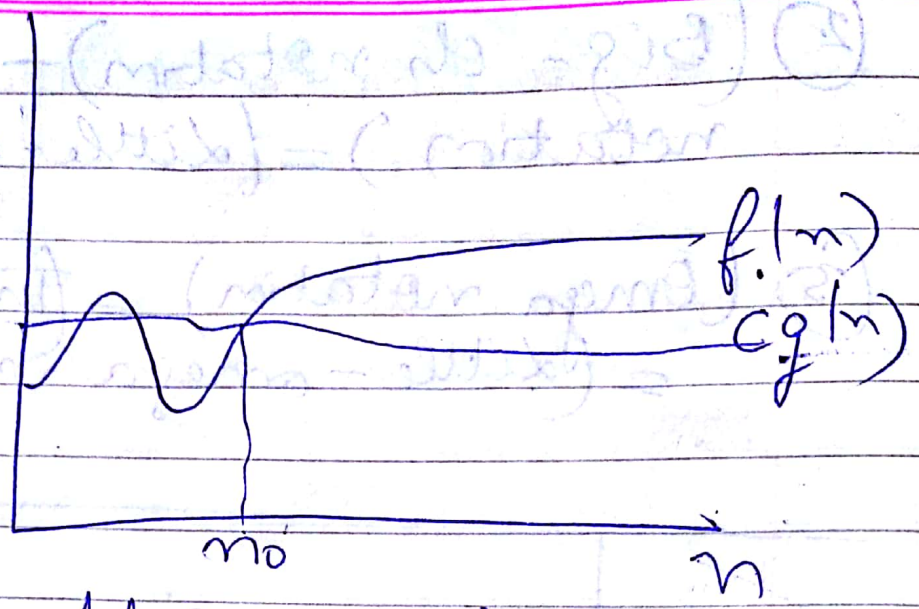


$f(n) = \Theta g(n)$

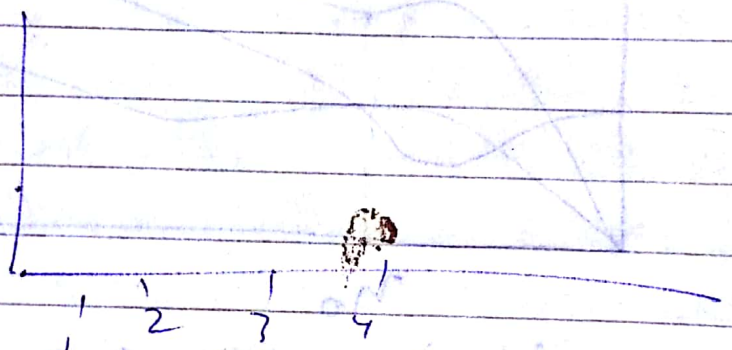


$f(n) = O g(n)$

Scanned by CamScanner



$$f(n) = \Omega\, g(n)$$



$3 \le n \le 3$

Scanned by CamScanner

Comparison of functions

Transitivity

$f(n) = \Theta(g(n))$ + $g(n) = \Theta(h(n))$

$\underset{\Omega}{\overset{O}{\Theta}}$ $f(n) = \Theta(h(n))$

Reflexivity

$f(n) = \underset{\Omega}{\overset{O}{\Theta}} f(n)$

Symmetry

$f(n) = \Theta g(n)$ iff $g f(n) = \Theta(f(n))$

Transpose symmetry

$f(n) = \underset{O}{O} g(n)$ iff $f(n) = \underset{\omega}{\Omega} f(n)$

Scanned by CamScanner

Recurrences

A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs.

Methods of solving recurrences (i.e for obtaining asymptotic bounds on solution):-

(1) Substitution method:- We guess a bound and then use mathematical induction to prove our guess correct.

(2) Recursion tree method:- converts the recurrence into tree whose nodes represents costs incurred at various level of recursion.

(3) Master method:- provides bounds for recurrences of the form:-

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$, $b > 1$ and $f(n)$ is a given function.

Scanned by CamScanner

Note: when we solve recurrence, we omit floors, ceilings + boundary conditions

Go to master method

(2) Substitution method

Substitution method for solving recurrences consists of 2 steps:-

(1) Guess the form of solution.
(2) Use mathematical induction to find constants and show that solution works.

eg. Determine an upper bound on the foll^n recurrence:-

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$

We guess sol^n to be

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

ie $T(n) \le cn \lg n$ for some $c > 0$ and $n \ge n_0$

We will prove it by induction:-

Basic step:-

$T_1 \le c \lg 1 \; T_1 \le 0$

$c = a > 0$

$a\ b = 0$

$b = a > 0$

If $n = 1$

$T(1) \le c \cdot 1 \cdot \lg 1$

≤ 0

If we take $n = 1$, then $\lg 1 = 0$

i.e. $T(1) \le 0$

So, taking $n = 2$

$T(n) = 2T(n/2) + n$

$T(2) = 2T(1) + 2$ Let $T(1) = c_1$

$= 2c_1 + 2$

We have to prove:-

$T(2) \le c\, 2 \lg 2 = 2c$

$2c_1 + 2 \le 2c$

$\Rightarrow c \ge c_1 + 1$

ie for $n = 2$, $T(2) \le c \cdot 2 \lg 2$

for $c \ge c_1 + 1$

Inductive step

Let us assume it true for $n/2$

i.e. $T(n/2) \le c \frac{n}{2} \lg n/2$

Let us check for n ie

we have to prove,-

$T(n) \le c \cdot n \lg n$

$T(n) = 2T(n/2) + n$

$\Rightarrow T(n) \le 2c \frac{n}{2} \lg(n/2) + n$

$\Rightarrow T(n) \le cn \lg(n/2) + n$

$= cn \lg n - cn \lg 2 + n$

$\le cn \lg n - (cn - n) \le cn \lg n - cn + n$

$2 \cdot (n/2) + 1)$

ie $T(n) \le cn \lg n$

if $cn - n \ge 0$

$cn \ge n$

$c \ge 1$

Using M.I, it is proved.

Q: Solve

$T(n) = T(n/2) + 1$

Let us ~~(assume)~~ guess the sol^n to be :- $O(\lg n)$

ie we need to prove

$T(n) \le c \lg n$ for some $c > 0$ & $n \ge n_0$

<u>We will prove it by induction</u>

<u>Basic step</u>

If we take $n = 1$, we need to prove $T(1) \le c \times \lg 1$ ie $T(1) \le 0$ Not true

so take $n = 2$, Let $T(1) = c_1$

~~we need to prove~~

$T(2) = T(1) + 1 = c_1 + 1$

Scanned by CamScanner

We need to prove

$$T(2) \leq c \lg 2$$

$$c_1 + 1 \leq c$$

ie for $n=2$ $T(2) \leq c \lg 2$

for $c \geq c_1 + 1$

Inductive step.

Let us assume it true for $n/2$

ie $T(n/2) \leq c \lg n/2$

Let us check for n

ie we have to prove

$$T(n) \leq c \lg n$$

$$\begin{aligned} T(n) &= T(n/2) + 1 \\ &\leq c \lg n/2 + 1 \\ &= c(\lg n - \lg 2) + 1 \\ &= c \lg n - c + 1 \\ &\leq c \lg n - (c-1) \end{aligned}$$

ie $T(n) \leq c \lg n$

if $c \geq 1$ $\left(\begin{array}{c} c-1 \geq 0 \\ c \geq 1 \end{array} \right)$

So, it is true for n

$\therefore$ using induction, it is proved.

Scanned by CamScanner

$n^{\log_b a}$ $\qquad$ $n^{\log_2 2} = n^0 = 1$

Solving Recurrence relation using Master method

Let $T(n) = aT(n/b) + f(n)$
where $a \geq 1$; $b > 1$
Let $f(n)$ be function

(1) If $f(n)$ is polynomially smaller than $n^{\log_b a}$
i.e. $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$
then $T(n) = \Theta(n^{\log_b a})$

(2) If $f(n) = \Theta(n^{\log_b a})$
then $T(n) = \Theta(n^{\log_b a} \cdot \lg n)$

(3) If $f(n)$ is polynomially larger than $n^{\log_b a}$ i.e.
$f(n) = \Omega(n^{\log_b a + \epsilon})$
for some constant $\epsilon > 0$
and if $[a f(n/b) \leq c f(n)]$
for some constant $c < 1$
and all sufficiently large n, then $(T(n) = \Theta(f(n)))$

Scanned by CamScanner

Master method does not cover all possible cases:-

It is not applicable if :-

(i) $f(n) < n^{\log_b a}$ but not polynomially smaller.

(ii) $f(n) > n^{\log_b a}$ but not polynomially larger.

(iii) Condition $a f(n/b) \leq c f(n)$ for some constant $c < 1$ fails to hold.

eg. $T(n) = 2T(n/2) + \Theta(n)$

$a = 2,\ b = 2,\ f(n) = \Theta(n)$

$n^{\log_b a} = n^{\log_2 2} = n$

~~$f(n) = \Theta(n) = \Theta($~~

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

$\therefore$ 2nd case is applicable

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

eg. $T(n) = 9T(n/3) + n$

$a = 9, \quad b = 3$

$f(n) = n$

$n^{\log_b a} = n^{\log_3 9}$

$= n^{\log_3 3^2}$

$= n^{2\log_3 3}$

$= n^2 \quad (\text{as } \log_3 3 = 1)$

$$\boxed{\begin{aligned} f(n) &= O(n^{2-1}) \\ &= O(n) \quad \text{for } \epsilon = 1 \end{aligned}}$$

As $n < n^2$, so $f(n)$ is smaller than $n^{\log_b a}$

so, 1st case is applicable

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

$af(n/b) \le cf(n)$

$c < 1$

eg. $T(n) = 4T(n/2) + n^3$

$a = 4, \quad b = 2, \quad f(n) = n^3$

$n^{\log_b a} = n^{\log_2 4}$

$= n^{2\log_2 2}$

$= n^2 \quad (\text{as } \log_2 2 = 1)$

$f(n) = n^3 > n^2$ i.e. $f(n) > n^{\log_b a}$,

$\therefore$ 3rd case is applicable.

$\Rightarrow a f(n/b) \le c f(n)$ for some $c < 1$ and $n \ge n_0$

$\Rightarrow 4 f(n/2) = 4\left(\frac{n^3}{8}\right)$

$= \frac{n^3}{2}$

$\Rightarrow \cancel{\frac{n^3}{2} \le c n^3} \quad \Rightarrow \frac{n^3}{2} \le c n^3$

$\Rightarrow \cancel{c \ge \frac{1}{2}}$? for $c = \frac{3}{4}$

then $a f(n/b) \le c f(n)$ for $n \ge 0$

$\therefore T(n) = \Theta(n^3)$

Scanned by CamScanner

eg. $T(n) = 3T(n/4) + n \lg n$

$a = 3$

$b = 4$

$n^{\log_b a} = n^{\log_4 3}$

$\log_4 3 = \frac{\log_{10} 3}{\log_{10} 4} = \frac{0.477}{0.602}$

$= 0.792 < 1$

eg. $n = 2$

$n \lg n = 2$

$\epsilon \simeq 0.2$

$n^{\log_4 3} = 2^{0.792} = 1.731$

eg. $n = 4$

$n \lg n = 2 \times 2 = 4$

$n^{\log_4 3} = 4^{0.792} = 2.99$

$f(n) = n \lg n$ is larger than $n^{\log_4 3}$

So, 3rd case is applicable

$\frac{2}{4}$

$\frac{1+2}{2} = \frac{3}{2}$

$\frac{1}{2} < \frac{3}{2}$

0.5

$\frac{0.42}{1.5}$

$\frac{1.63}{1.5}$

$a f(n/b) \leq c f(n)$ for $c < 1$ and $n \geq n_0$

$\Rightarrow 3 f(n/4)$

$\Rightarrow 3(n/4 \log n/4)$

$\Rightarrow \frac{3n}{4} \lg n - \frac{3n}{4} \lg 4$

$\leq \frac{3n \lg n}{4}$

$\leq c f(n)$

if $c = \frac{3}{4}$

$\therefore T(n) = \Theta(f(n))$

$= \Theta(n \lg n)$

Scanned by CamScanner

eg. $T(n) = 4T(n/2) + n^2/\lg n$

$a = 4,\ b = 2$

$f(n) = n^2/\lg n$

$n^{\log_b a} = n^{\log_2 2^2} = n^2$

$f(n) = \dfrac{n^2}{\lg n}$ is smaller than n^2,

but not polynomially smaller so we cannot apply master method.

Scanned by CamScanner

Iteration method

eg. $T(n) = 2T(n/2) + n$

$\Rightarrow 2\left[2T(n/4) + \frac{n}{2}\right] + n$

$\Rightarrow 4T(n/4) + n + n$

$\Rightarrow 2^2 T(n/2^2) + 2n$

$\Rightarrow 2^2\left[2T(n/2^3) + \frac{n}{4}\right] + 2n$

$\Rightarrow 2^3 T(n/2^3) + 3n$

$\Rightarrow 2^m T(n/2^m) + mn$

$\Rightarrow 2^m T(1) + mn$

$\frac{n}{2^m} = 1$

$n = 2^m$

$m = \log_2 n$

and $\boxed{T(1) = 1}$

$\Rightarrow 2^m + mn$

$\Rightarrow n + \lg n \cdot n$

$\Rightarrow n \lg n + n$

Scanned by CamScanner

$4 \times 2 + 4 = 12 \leq 16$ 23

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

$T(n) \leq n \lg n + n \lg n$

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

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

---

eg. $T(n) = 4T(n/2) + n^2$

$= 4\left[4T(n/4) + \frac{n^2}{4}\right] + n^2$

$= 4^2 T(n/2^2) + 2n^2$

$= 4^2\left[4T(n/2^3) + \frac{n^2}{16}\right] + 2n^2$

$= 4^3 T(n/2^3) + 3n^2$

$= (2^m)^2 T(n/2^m) + mn^2$

$= (2^m)^2 T(1) + mn^2$

$\frac{n}{2^m} = 1$

$n = 2^m$

$m = \log_2 n$

and $T(1) = 1$

Scanned by CamScanner

$\Rightarrow n^2 \cdot 1 + n^2 \lg n$

$$T(n) = n^2 + n^2 \lg n$$
$$\left(\begin{aligned} &\leq n^2 \lg n + n^2 \lg n \\ &\leq 2n^2 \lg n \end{aligned} \right)$$
$$T(n) = O(n^2 \lg n)$$

<u>2nd order recurrence relation</u>

Q: $a_n = \begin{cases} 5a_{n-1} - 6a_{n-2} & n > 1 \\ 16 & n = 1 \\ 7 & n = 0 \end{cases}$

$a_n = 5a_{n-1} - 6a_{n-2}$

Characteristic equation is:-

$x^n - 5x^{n-1} + 6x^{n-2} = 0$

Divide by lowest degree term x^{n-2}

$x^2 - 5x + 6 = 0$

$\Rightarrow x^2 - 3x - 2x + 6 = 0$

$\Rightarrow x(x-3) - 2(x-3) = 0$

$\Rightarrow (x-3)(x-2)$

$\Rightarrow x = 3, 2$

Scanned by CamScanner

Solution is of form

$a_n = \alpha \cdot 3^n + \beta \cdot 2^n$

$n=0$

$\Rightarrow \alpha + \beta = 7 \quad \text{--- (1)}$

$n=1$

$\Rightarrow 3\alpha + 2\beta = 16$

$\Rightarrow 2\alpha + 2\beta = 14$

$\Rightarrow 3\alpha + 2\beta = 16$

$(-), \quad (-), \quad (-)$

$+\alpha = +2$

$\beta = 5$

$a_n = 2 \cdot 3^n + 5 \cdot 2^n$

$\left(\begin{aligned} &\leq 2 \cdot 3^n + 5 \cdot 3^n \\ &\leq 3^n (2+5) \\ &\leq 7 \cdot 3^n \end{aligned} \right)$

$a_n = O(3^n)$

Scanned by CamScanner

$\lg n = m$

$n = 2^m$

Change of Variables

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

Let $n = 2^m$

$\Rightarrow m = \lg n$

$\Rightarrow T(2^m) = 2T(2^{m/2}) + m$

$\Rightarrow$ Let $T(2^m) = S_m$

$$\boxed{\Rightarrow S_m = 2S_{m/2} + m}$$

$\Rightarrow S_m = \Theta(m \lg m)$ (merge sort or master method $\Rightarrow \Theta(n \lg n)$)

$\Rightarrow T(n) = \Theta(\lg n \cdot \lg \lg n)$

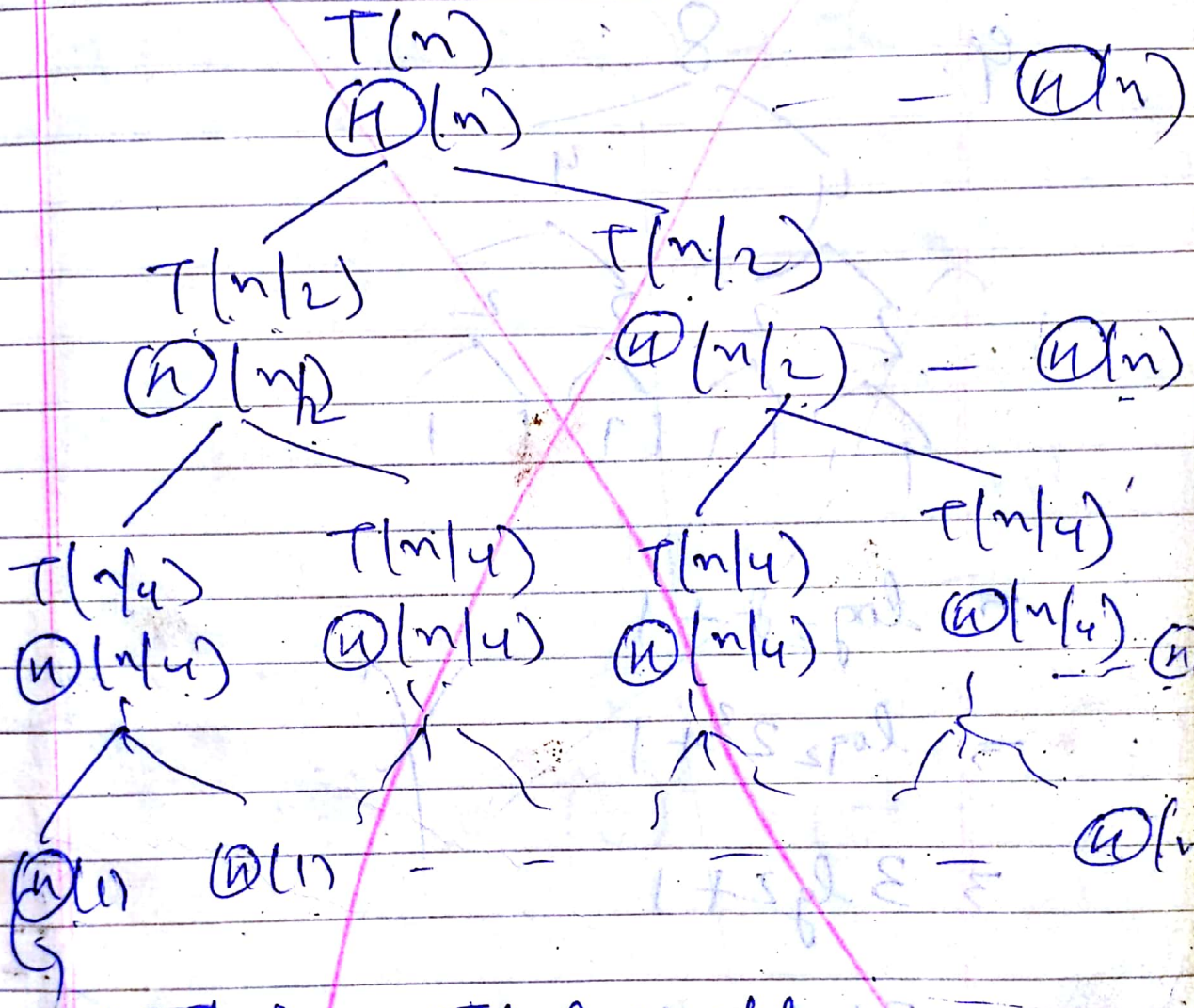
$O(n^2)$

$n^{\log_2 4}$

$n, \frac{n}{2}, \frac{n}{4}$

Recursion tree

$$T(n) = 2T(n/2) + \Theta(n)$$



$T(n)$ = Total no. of levels × running time of each level

$$= (\lg n + 1)\,\Theta(n)$$

$$= \Theta(n \lg n) + \Theta(n)$$

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

For, level it should become $T(1)$

$\frac{n}{2^i} = 1$

$n = 2^i$

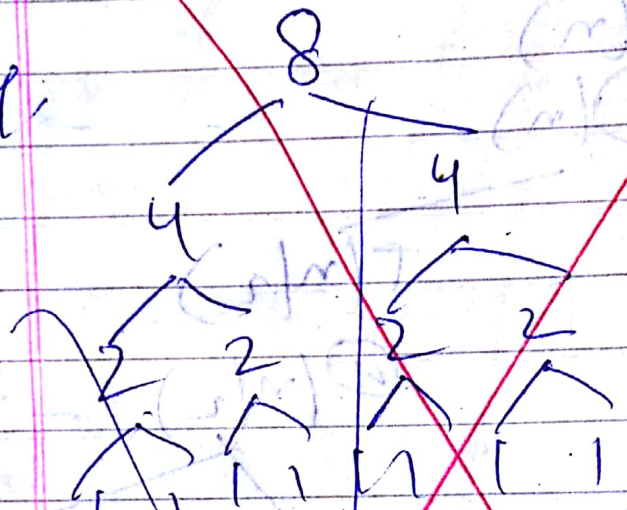
$i = \lg n$

$i = (0, 1, \ldots \lg n) \rightarrow$ Total $= \lg n + 1$ levels

No. of levels for a tree ~~with height~~ n

$= \lg n + 1$

eg.



$= \log_2 8 + 1$

$= \log_2 2^3 + 1$

$= 3 \lg 2 + 1$

$= 3 + 1 = 4$

No. of ~~last~~ level $= 1 + \lg n$

Last level $- T(1)$

$\frac{n}{2^i} = 1$

$n = 2^i$

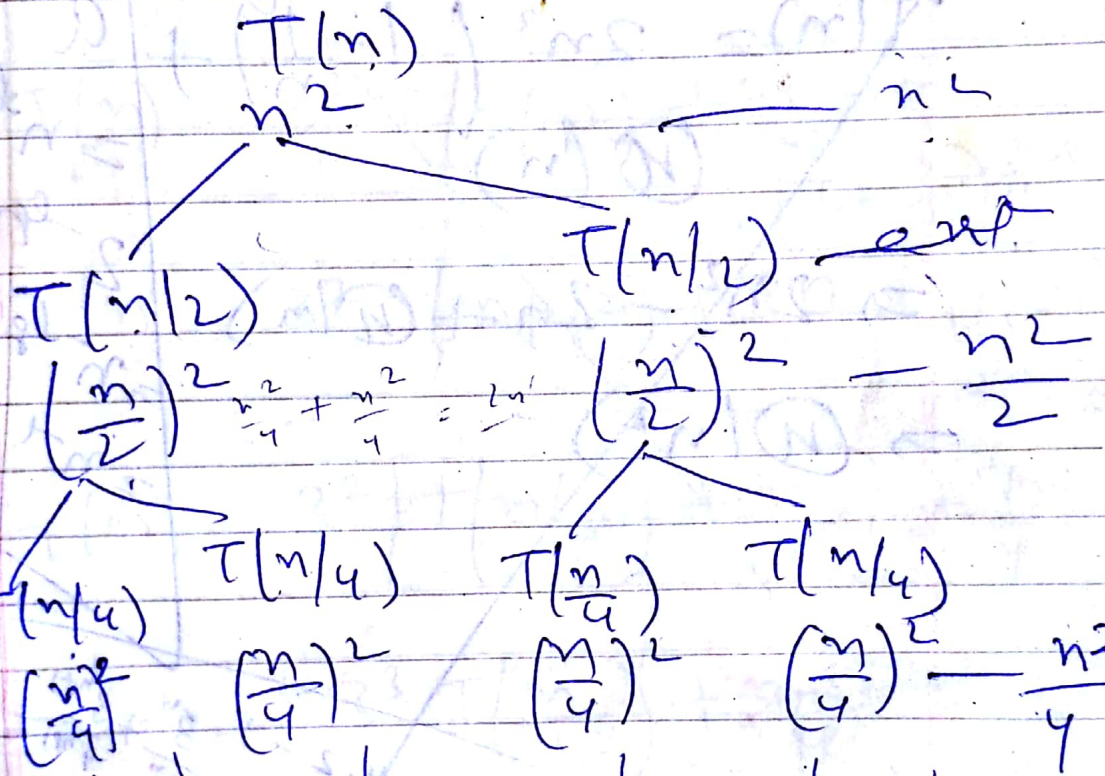
$i = \lg n$

$n^{\log_b a} = n^{\log_2 2} = n$ level

Q: $T(n) = 2T(n/2) + n^2 + \Theta(n)$

$T(1) = \Theta(1)$

$T(n) = n^2 + T(n/2) + T(n/2)$



$T(n) = n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \cdots + \frac{n^2}{2^{\log n - 1}} + \Theta(n)$

$T(n) = n^2\left[1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^{\log n - 1}}\right] + \Theta(n)$

$\frac{n}{2^i}$ $\log n$ ts

i $\log n + 1$, but starts with zero

$\log n - 1$

$\log n + 1$ but starts with zero

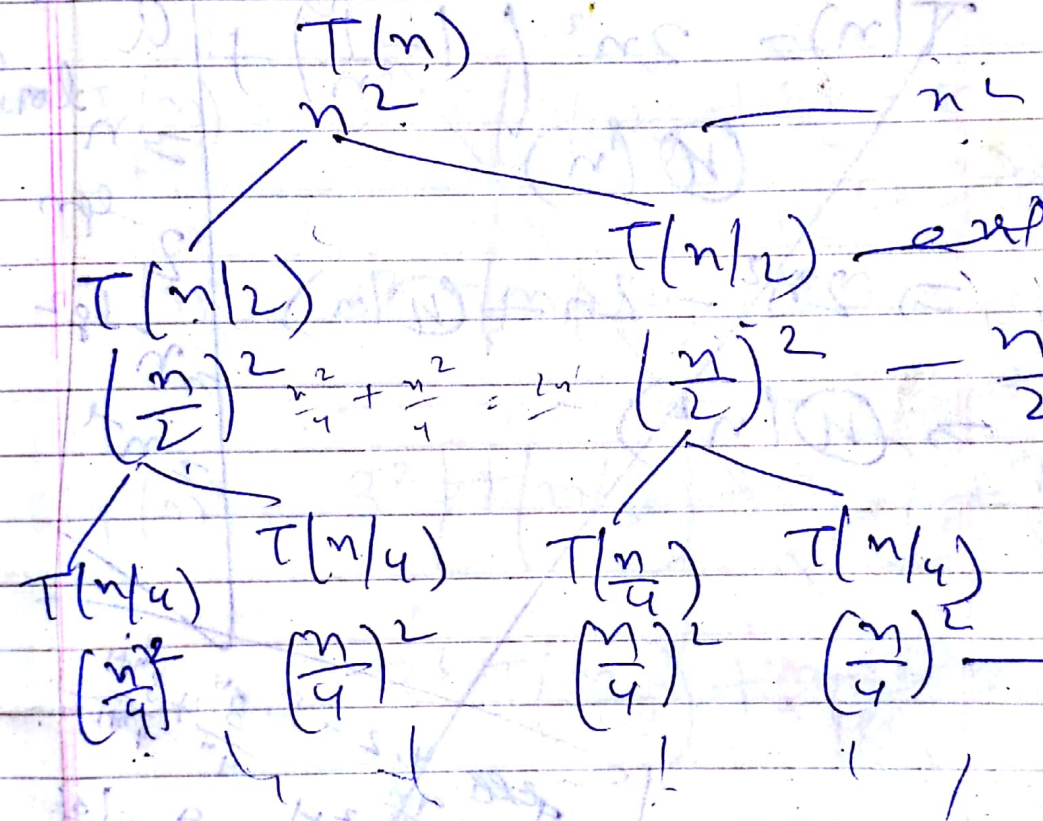
cost of last level

$n^{\log_b a} = n^{\log_2 2} = n$

Q: $T(n) = 2T(n/2) + n^2 \quad \Theta(n)$

$T(1) = \Theta(1)$

$T(n) = n^2 + T(n/2) + T(n/2)$



$T(n) = n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \cdots + \frac{n^2}{2^{\log n - 1}} + \Theta(n)$

$\log n - 1$

$T(n) = n^2\left[1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^{\log n - 1}}\right] + \Theta(n)$

$\frac{n}{2^i}$ $\log n$ i $\log n + 1$, but start with zero

~~$\log n \to \log n + 1$ but an start $\log n$ given~~

$$\left(a^{\log_b n} = n^{\log_b a}\right)$$

$$\frac{a(1-r^n)}{1-r}$$

$$T(n) = n^2 \cdot \frac{1\left(1-\left(\frac{1}{2}\right)^{\log n}\right)}{1-\frac{1}{2}} + \Theta(n)$$

$$T(n) = 2n^2\left(1-\frac{1}{n}\right) + \Theta(n)$$

$$T(n) = 2n^2 - 2n + \Theta(n)$$

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

$$\frac{a(1-r^n)}{1-r}$$

Scanned by CamScanner

$$\left(a^{\log_b n} = n^{\log_b a}\right)$$

$$\Uparrow$$

$$\frac{a(1-r^n)}{1-r}$$

$$T(n) = n^2 \cdot \frac{1\left(1-\left(\frac{1}{2}\right)^{\log n}\right)}{1-\frac{1}{2}} + \Theta(n)$$

$$T(n) = 2n^2\left(1-\frac{1}{n}\right) + \Theta(n)$$

$$T(n) = 2n^2 - 2n + \Theta(n)$$

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

$$\frac{a(1-r^n)}{1-r}$$

Scanned by CamScanner

Iteration method

eg. $T(n) = 3T(n/4) + n$

$\Rightarrow T(n) = 3\left[3T(n/16) + \frac{n}{4}\right] + n$

$\Rightarrow T(n) = 3^2 T\left(\frac{n}{4^2}\right) + n + \frac{3n}{4}$

$\Rightarrow T(n) = 3^2\left[3T\left(\frac{n}{64}\right) + \frac{n}{16}\right] + n + \frac{3n}{4}$

$\Rightarrow T(n) = 3^3 T\left(\frac{n}{4^3}\right) + n + \frac{3n}{4} + \frac{9n}{16} + \cdots$

$\Rightarrow T(n) = 3^3 T\left(\frac{n}{4^3}\right) + n\left[1 + \frac{3}{4} + \frac{3^2}{4^2} + \cdots\right]$

$\Rightarrow T(n) = 3^m T\left(\frac{n}{4^m}\right) + n\left[\frac{1 \cdot \left(1 - \frac{3}{4}\right)^m}{1 - \frac{3}{4}}\right]$

$\Rightarrow T(n) = 3^m T\left(\frac{n}{4^m}\right) + 4n\left(1 - \left(\frac{3}{4}\right)^m\right)$

$\Rightarrow \left(\begin{array}{l} \frac{n}{4^m} = 1 \\ n = 4^m \\ m = \log_4 n \\ \text{and } T(1) = 1 \end{array}\right)$

Scanned by CamScanner

$$\Rightarrow T(n) = 3^{\log_4 n} \cdot 1 + 4n\left[\frac{1 - 3^{\log_4 n}}{1 - 4^{\log_4 n}}\right]$$

As $n^{\log_b a} = a^{\log_b n}$

$$\Rightarrow T(n) = n^{\log_4 3} + 4n\left[1 - \frac{n^{\log_4 3}}{n^{\log_4 4}}\right]$$

$$\Rightarrow T(n) = n^{\log_4 3} + 4\cancel{n}\left[\frac{n - n^{\log_4 3}}{\cancel{n}}\right]$$

$$\Rightarrow T(n) = n^{\log_4 3} + 4n - 4n^{\log_4 3}$$

$$\Rightarrow T(n) = 4n - 3n^{\log_4 3} = \frac{\log 3}{\log 4}$$

$$\Rightarrow T(n) \leq 4n$$

$$\Rightarrow T(n) = O(n).$$

Scanned by CamScanner

Q: $t_n = \begin{cases} n & \text{if } n = 0, 1, 2 \\ 5t_{n-1} - 8t_{n-2} + 4t_{n-3} & \text{otherwise} \end{cases}$

$t_n - 5t_{n-1} + 8t_{n-2} - 4t_{n-3} = 0$

$x^n - 5x^{n-1} + 8x^{n-2} - 4x^{n-3} = 0$

$x^3 - 5x^2 + 8x - 4 = 0$

$\Rightarrow (x-1)(x^2 - 4x + 4)$

$\Rightarrow x = 1, 2, 2$

$$
\begin{array}{r|l}
 & x^2 - 4x + 4 \\
\hline
x-1 & x^3 - 5x^2 + 8x - 4 \\
 & \underline{x^3 - x^2} \\
 & (-) \quad (+) \\
 & -4x^2 + 8x - 4 \\
 & \underline{-4x^2 + 4x} \\
 & (+) \quad (-) \\
 & 4x - 4 \\
 & \underline{4x - 4} \\
 & (-) \quad (+) \\
 & \times
\end{array}
$$

Scanned by CamScanner

Solution is of the form:-
(if 2 roots are same)

$t_n = \alpha \cdot 1^n + \beta \cdot 2^n + r \cdot n \cdot 2^n$

Note: If 3 roots are same,
eg. if 2, 2, 2

$t_n = \alpha \cdot 2^n + \beta \cdot n \cdot 2^n + r \cdot n^2 2^n$

$\Rightarrow t_0 = \alpha + \beta = 0$

$\Rightarrow t_1 = \alpha + 2\beta + 2r = 1$

$\Rightarrow t_2 = \alpha + 4\beta + 8r = 2$

$(-\beta) + 2\beta + 2r = 1$

$\beta + 2r = 1 \;] \times 3$

$3\beta + 8r = 2$

$\Rightarrow 3\beta + 6r = 3$

$\Rightarrow 3\beta + 8r = 2$

$(-) \quad (-) \quad (-)$

$-2r = 1$

$r = \frac{-1}{2}$

$\alpha = -\beta$

$\alpha = -2$

$\beta + 2r = 1$

$\beta + 2\left(\frac{-1}{2}\right) = 1$

$\beta - 1 = 1$

$\beta = 2$

Scanned by CamScanner

$\alpha = -2, \beta = 2, \gamma = -\frac{1}{2}$ 35

Solution is:

$t_n = -2 \cdot 1^n + 2 \cdot 2^n - \frac{1}{2} \cdot n \cdot 2^n$

$t_n = (4)(n \cdot 2^n)$

Q: $t_n - 2t_{n-1} = 3^n$ — (1)

Multiply equation (1) by (3)

$\Rightarrow 3t_n - 6t_{n-1} = 3^{n+1}$

Put $n = n-1$

$\Rightarrow 3t_{n-1} - 6t_{n-2} = 3^n$ — (2)

From (1) & (2)

$\Rightarrow t_n - 2t_{n-1} = 3t_{n-1} - 6t_{n-2}$

$\Rightarrow t_n - 5t_{n-1} + 6t_{n-2} = 0$

$\Rightarrow x^n - 5x^{n-1} + 6x^{n-2} = 0$

Scanned by CamScanner

$\Rightarrow x^2 - 5x + 6 = 0$

$\Rightarrow x^2 - 3x - 2x + 6 = 0$

$\Rightarrow x(x-3) - 2(x-3) = 0$

$\Rightarrow (x-3)(x-2)$

$\Rightarrow x = 2, 3$

$t_n = \alpha \cdot 2^n + \beta \cdot 3^n$

From (1)

$\Rightarrow t_1 - 2t_0 = 3$

$\Rightarrow t_1 = 3 + 2t_0$

$\Rightarrow t_0 = \alpha + \beta$

$\Rightarrow t_1 = 2\alpha + 3\beta = 3 + 2t_0$

$\Rightarrow t_0 - \alpha = \beta$

$\Rightarrow 2\alpha + 3(t_0 - \alpha) = 3 + 2t_0$

$\Rightarrow 2\alpha + 3t_0 - 3\alpha = 3 + 2t_0$

$\Rightarrow 3t_0 - \alpha = 3 + 2t_0$

$\Rightarrow -\alpha = 3 + 2t_0 - 3t_0$

$\Rightarrow -\alpha = 3 - t_0$

$\Rightarrow \alpha = t_0 - 3$

$\Rightarrow \alpha + \beta = t_0$

$\Rightarrow \cancel{t_0} - 3 + \beta = \cancel{t_0}$

$\Rightarrow \beta = 3$

Scanned by CamScanner

$\alpha = t_0 - 3$

$\beta = 3$

$\Rightarrow t_n = (t_0 - 3) \cdot 2^n + 3^{n+1}$

$\Rightarrow t_n = \Theta(3^n)$

Scanned by CamScanner

Divide & Conquer Technique

In this technique:

1. Divide the problem into a no. of subproblems.
2. Conquer the subproblems by solving them recursively till we get a straight forward solution to subproblem.
3. Combine the solutions to the subproblems into the solution for the original problem.

eg. Merge sort

1. In this algorithm, divide the n-element sequence into two subsequences of $n/2$ elements each.
2. Sort the 2 subsequences recursively using merge sort.
3. Combine (merge) the two sorted subsequences to produce the sorted list.

Scanned by CamScanner

Algo for merge sort

Merge sort (A, p, r)

1. if $p < r$
2. then $q \leftarrow \lfloor (p+r)/2 \rfloor$
3. Merge sort (A, p, q)
4. Merge sort $(A, q+1, r)$
5. Merge (A, p, q, r) $\rightarrow \Theta(n)$

Constant time $\Theta(1)$ executed only once

$T(n/2)$ each if merge sort takes $T(n)$ time

Index of 1st element $\rightarrow p$

" " last " $\rightarrow r$

" " middle " $\rightarrow q$

Input: $A[p \ldots r]$

$A[p \ldots q]$ ---- $A[q+1 \ldots r]$

$A[p \ldots r]$

Scanned by CamScanner

Algo for merge procedure

```
Merge (A, p, q, r)
▷ C[p ... r] stores sorted list.
  i ← p
  j ← q+1
  k ← p
▷ copy smaller of A[i] and A[j]
  while i ≤ q and j ≤ r
  do if A[i] < A[j]
  then C[k] ← A[i]
       i ← i+1, k ← k+1
  else C[k] ← A[j]
       j ← j+1, k ← k+1

▷ copy remainder of 1st sequence
  while i ≤ q
  do C[k] ← A[i]
     i ← i+1
     k ← k+1

▷ copy remainder of second sequence
  while j ≤ r
  do C[k] ← A[j]
     j ← j+1
     k ← k+1
```

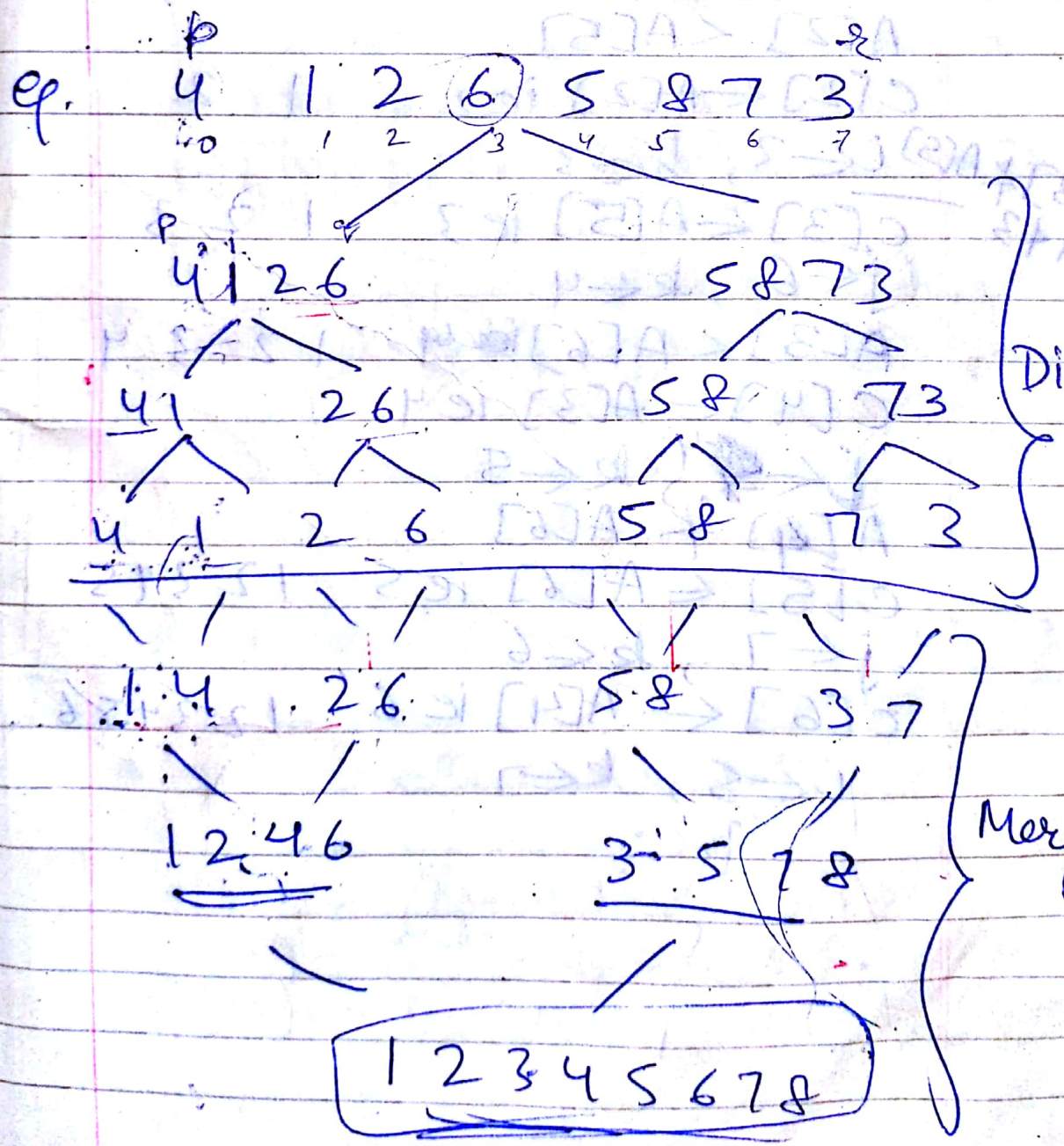
Scanned by CamScanner

▷ copy $C[p \ldots r]$ to $A[p \ldots r]$

```
l ← p
while l ≤ r
do A[l] ← C[l]
   l ← l+1
```

Merge procedure takes $T(n) = \Theta(n)$ where n is the no. of elements in array A ie $n = r - p + 1$

eg.



Merge procedure

1 2 4 6 | 3 5 7 8

$p=1$
$q=4$
$r=8$
$i \leftarrow 1$
$j \leftarrow 5$
$k \leftarrow 1$

$A[1] < A[5]$ — C
$C[1] \leftarrow A[1]$ ie 1 — 1
$i \leftarrow 2$, $k \leftarrow 2$
$A[2] < A[5]$
$C[2] \leftarrow A[2]$ ie 2 — 1 2
$i \leftarrow 3$, $k \leftarrow 3$

$A[3] \not< A[5]$
4 < 3
$C[3] \leftarrow A[5]$ ie 3 — 1 2 3
$j \leftarrow 6$, $k \leftarrow 4$
$A[3] < A[6]$ ie 4 — 1 2 3 4
$C[4] \leftarrow A[3]$ ie 4
$i \leftarrow 4$, $k \leftarrow 5$
$A[4] \not< A[6]$
$C[5] \leftarrow A[6]$ ie 5 — 1 2 3 4 5
$j \leftarrow 7$, $k \leftarrow 6$
$C[6] \leftarrow A[4]$ ie 6 — 1 2 3 4 5 6
$i \leftarrow 5$, $k \leftarrow 7$

Scanned by CamScanner

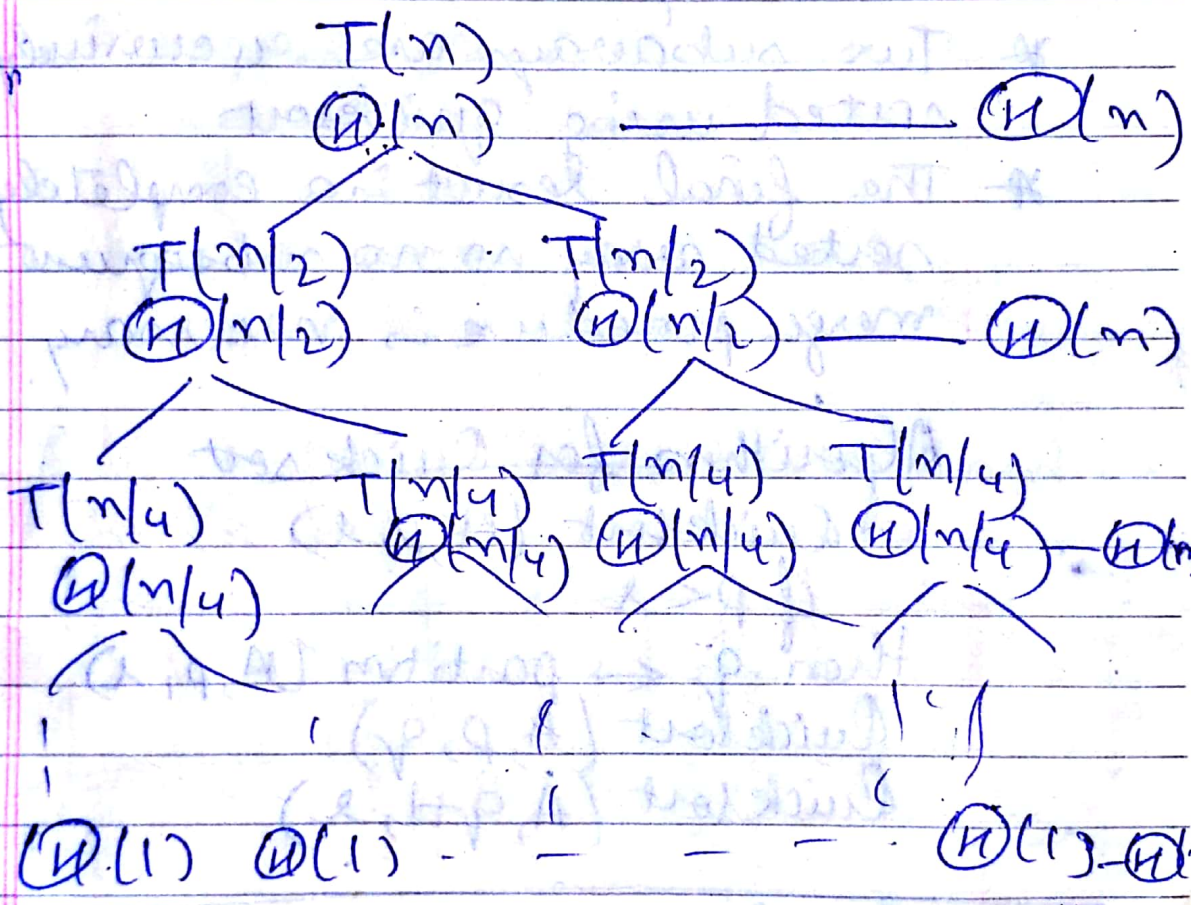
Running time for merge sort

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

$$T(1) = \Theta(1)$$

$$T(n) = 2T(n/2) + \Theta(n)$$

$$T(1) = \Theta(1)$$



No. of levels $= \lg n + 1$

$$T(n) = (\lg n + 1)\,\Theta(n)$$

$$T(n) = \Theta(n \lg n) + \Theta(n)$$

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

Scanned by CamScanner

Quick sort

* We use divide and conquer technique
* Array is partitioned into 2 subarrays such that elements in left subarray are less than or equal to elements in right subarray.
* Two subarrays are recursively sorted using quicksort.
* The final result is a completely sorted array so no subsequent merge procedure is necessary.

Algorithm for Quick sort

```
Quicksort (A, p, r)
    if p < r
  then q ← partition (A, p, r)
    Quicksort (A, p, q)
    Quicksort (A, q+1, r)
```

Initial call to quicksort is
Quicksort (A, 1, n)

A[p - - r]

A[p - - q] A[q+1, - - r]

partition procedure returns it

Algorithm for partition procedure

```
Partition (A, p, r)
  x ← A[p]
  i ← p-1
  j ← r+1
  while true
  do repeat j ← j-1
      until A[j] ≤ x
  do repeat i ← i+1
      until A[i] ≥ x
      if i<j
  then exchange A[i] ↔ A[j]
    else return j
```

Running time for partition procedure is $\Theta(n)$

eg,

pivot x

1	2	3	4	5	6	7	8
5	3	2	6	4	1	3	7

i=1 j=7

≤ (equal to also)

compare with pivot only

3	3	2	6	4	1	5	7

i=4 j=6

1	2	3	4	5	6	7	8
3	3	2	1	4	6	5	7

j=5 i=6

Partition as i<j

Scanned by CamScanner

As $j = 5$

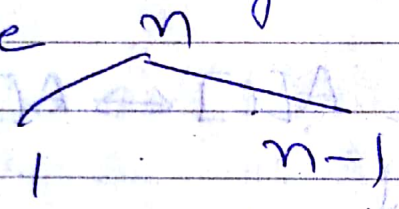
3 3 2 1 4 6 5 7

Running time for quick sort

Worst-case

Array will be partitioned into 2 subarrays, one with $(n-1)$ elements and other with a single element.

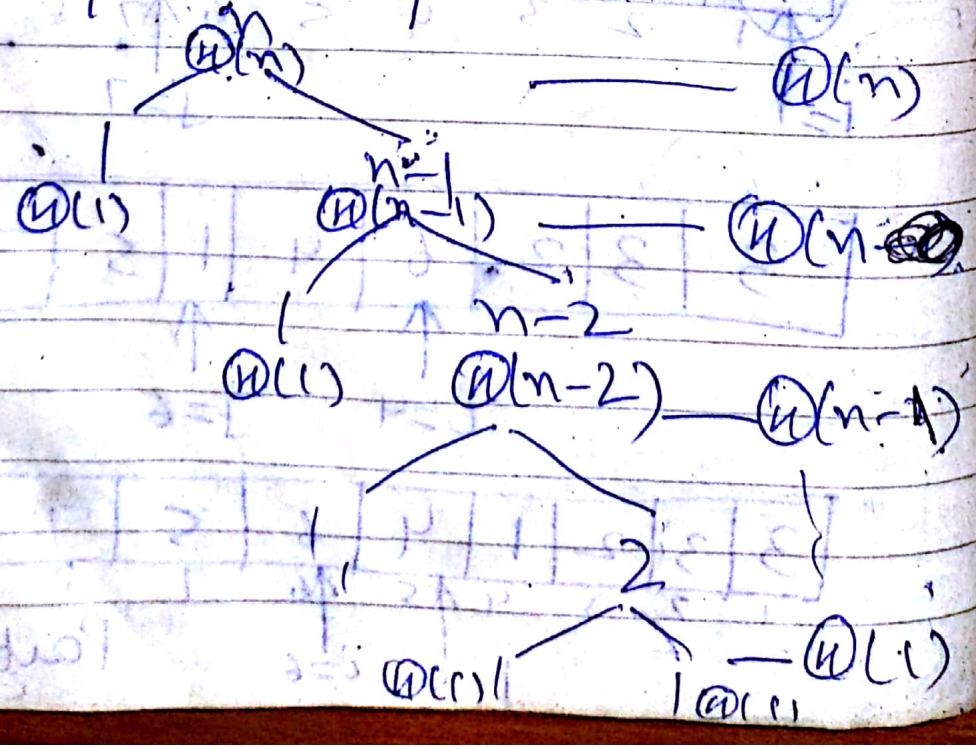
ie



$$T(n) = T(n-1) + T(1) + \Theta(n)$$

(Partition procedure)

Note: This occurs when array is already sorted.



$$T(n) = T(n-1) + T(1) + \Theta(n)$$

$$T(n) = \Theta(n) + \Theta(n-1) + \Theta(n-2) + \cdots \Theta(1)$$

$$T(n) = \Theta\left(\sum_{i=1}^{n} i\right)$$

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

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

eg.

1	2	3	4	5	6	7	8

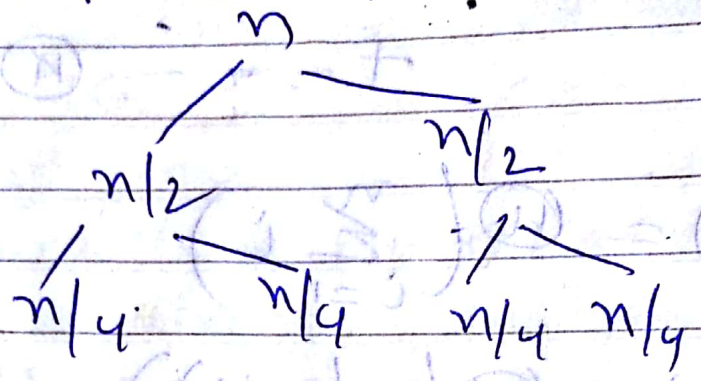
$i \geq 1$

$i = 1$

1 | 2 3 4 5 6 7 8

Best case:-

If partition procedure partitions array in 2 equal subarrays



$$T(n) = 2T(n/2) + \Theta(n)$$

for equal partitions

for partition procedure

same as merge sort

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

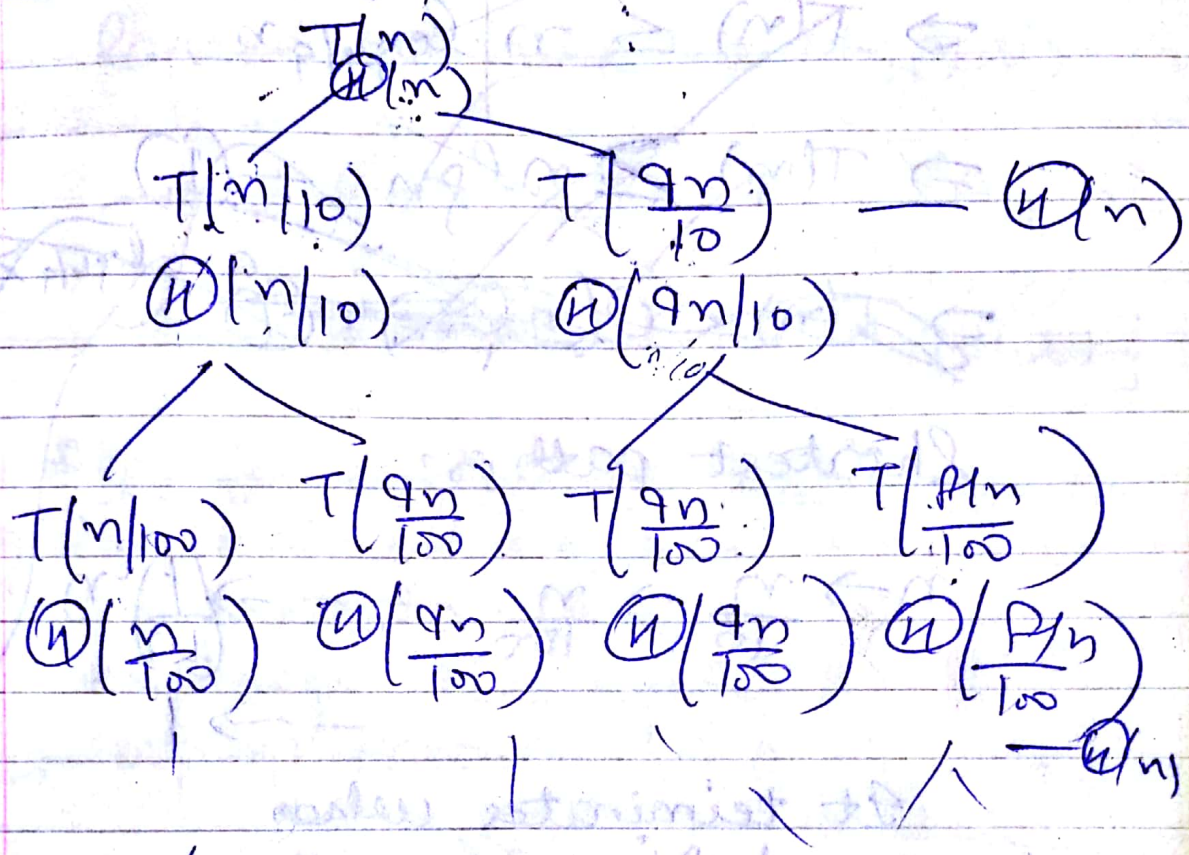
Average case: It is closer to best case. ie

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

eg. if partition procedure produces 9 to 1 proportional split.

Scanned by CamScanner

$$T(n) = T(n/10) + T(9n/10) + \Theta(n)$$



Longest path is $n \to \frac{9n}{10} \to \frac{81n}{100} \cdots \to \left(\frac{9}{10}\right)^i n \to 1$

It terminates when $\left(\frac{9}{10}\right)^i n = 1$

$$\Rightarrow n = \left(\frac{10}{9}\right)^i$$

$$\Rightarrow i = \log_{10/9} n$$

$$= \frac{\log_2 n}{\log_2 10/9} = \frac{\lg n}{\lg \frac{10}{9}}$$

$$\cancel{T(n) \le (\log_{10/9} n)n}$$

$$\cancel{\Rightarrow T(n) \le n \log_{10/9} n}$$

$$\cancel{\Rightarrow T(n) \ge n \lg n} \quad \cancel{(1)}$$

$$\cancel{\Rightarrow T(n) = \Theta(n \lg n)} \quad \cancel{(1)}$$

as $\log_{10/9} > \log_2$

Shortest path is:-

$$n \to \frac{n}{10} \to \frac{n}{100} \cdots \to \left(\frac{1}{10}\right)^i n$$

$$\to 1$$

It teiminates when

$$\left(\frac{1}{10}\right)^i n = 1$$

$$\Rightarrow n = 10^i$$

$$\Rightarrow i = \log_{10} n = \frac{\log_2 n}{\log_2 10} = \frac{\lg n}{k_2}$$

$$\Rightarrow T(n) \ge n \log_{10} n$$

$$\cancel{\Rightarrow T(n) \ge n \lg n} \Rightarrow T(n) \le n \lg n \quad (2)$$

$$\cancel{\Rightarrow T(n) = \Omega(n \lg n)} \quad \cancel{(2)}$$

as $\log_{10} <$ $\log_2$

~~From (1) & (2)~~

$$\cancel{\Rightarrow T(n) = \Theta(n \lg n)}$$

$$\Rightarrow T(n) \ge n\left(\frac{\lg n}{k_2} + 1\right)$$

$$\Rightarrow T(n) \ge \frac{1}{k_2} n \lg n + n$$

$$\Rightarrow T(n) = \Omega(n \lg n) \quad (2)$$

Scanned by CamScanner

$\frac{\log_{10/9} n}{\log_2 n}$

Randomized Quick Sort

Algo for Randomized Quick Sort

```
Randomized QuickSort (A, p, r)
  if p < r {
    q ← Randomized Partition (A, p, r)
    Randomized QuickSort (A, p, q)
    Randomized QuickSort (A, q+1, r) }
```

Algo for Randomized Partition

```
Randomized Partition (A, p, r)
  i ← Random (p, r)
  exchange A[p] ↔ A[i]
  return Partition (A, p, r)
```

$$T(n) \le n\left(\frac{\lg n}{k_1} + 1\right)$$

$$\le \frac{1}{k_1} n \lg n + n$$

$$T(n) = O(n \lg n) \quad \text{--- (1)}$$

$\lg_2$

From (1) + (2)

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

Scanned by CamScanner

Matrix Multiplication

We are given 2 $n \times n$ matrices A and B, we have to multiply A + B and store result in matrix C.

```
Matrix_Multiply (A, B, n)
  for i ← 1 to n
  do for j ← 1 to n
     do Cij ← 0
  for k ← 1 to n
     do Cij ← Cij + aik . bkj
  return C
```

Running time :-

$$T(n) = \Theta(n^3)$$

(As there are 3 nested for loops running till 'n')

$$\underset{A}{\begin{bmatrix} a & b \\ c & d \end{bmatrix}} \underset{B}{\begin{bmatrix} e & f \\ g & h \end{bmatrix}} = \underset{C}{\begin{bmatrix} i & j \\ k & l \end{bmatrix}}$$

$i = ae + bg$

$j = af + bh$

$k = ce + dg$

$l = cf + dh$

Scanned by CamScanner

No. of multiplications $= n^3 \; (2^3 = 8)$
No. of additions $= (n^2) \; (2^2 = 4)$

Using divide and conquer technique
If we use divide and conquer technique, we divide each of A, B and C matrices into four $n/2 \times n/2$ matrices and rewriting the equation $C = AB$ as follows:-

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & g \\ f & h \end{pmatrix}$$

where:

$r = ae + bf$
$s = ag + bh$
$t = ce + df$
$u = cg + dh$

$T(n/2)$

$n = 2 = n^1$

Each of four equations specifies 2 multiplications of $n/2 \times n/2$ matrices and addition of their $n/2 \times n/2$ products.
In this case, we derive the following recurrence:-

$$T(n) = 8T(n/2) + \Theta(n^2)$$

8 multiplications of $n/2 \times n/2$ matrices

for four additions

$n^{\log_b a}$
$\Rightarrow n^{\log_2 8}$
$\Rightarrow n^3$ (1st case)

matrices & $\Theta(n^2)$ scalar

$$T(n) = \Theta(n^3)$$

Strassen's Matrix Multiplication

This algo requires only 7 ~~scalar~~ recursive multiplications (to multiply two 2×2 matrices. We divide each of A, B and C matrices into four $n/2 \times n/2$ matrices as:

We rewrite $C = AB$ as:

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & g \\ f & h \end{pmatrix}$$

It yields the recurrence:

$$T(n) = 7T(n/2) + \Theta(n^2)$$

$$n^{\log_b a} = n^{\log_2 7}$$
$$= n^{2.81} \quad (1^{st} \text{ case})$$

$$\cancel{T(n) = \Theta(n^{\lg 7})}$$

$$\cancel{T(n) = \Theta(n^{2.81})}$$

$$T(n) = \Theta(n^{\lg 7})$$
$$T(n) = O(n^{2.81})$$

addition and subtraction.

$n = 2^{x}$

Strassen's method has 4 steps:

1. Divide the input matrices A and B into $n/2 \times n/2$ submatrices.
2. Compute 14 $n/2 \times n/2$ matrices $A_1, B_1, A_2, B_2 \ldots A_7, B_7$.
3. Recursively compute the seven matrix products $P_i = A_i B_i$ for $i = 1, 2 \ldots 7$
4. Compute the desired submatrices r, s, t, u of result matrix C by adding and/or subtracting various combinations of the P_i matrices.

<u>Determining the submatrix products</u>

It is not clear exactly how Strassen discovered the submatrix products that are key to making this algo work.

Let us given that each matrix product P_i can be written in the form:-

$$P_i = A_i B_i$$
$$= (\alpha_{i1} a + \alpha_{i2} b + \alpha_{i3} c + \alpha_{i4} d) * (\beta_{i1} e + \beta_{i2} f + \beta_{i3} g + \beta_{i4} h)$$

where coefficients α_{ij}, β_{ij} are all drawn from set $\{-1, 0, 1\}$.

Scanned by CamScanner

We shall use 4×4 matrices to represent linear combination of products of submatrices, where each product combines one submatrix of A with one submatrix of B.

eg. $r = ae + bf$

$$= \begin{array}{c|cccc} & e & f & g & h \\ \hline a & + & \cdot & \cdot & \cdot \\ b & \cdot & + & \cdot & \cdot \\ c & \cdot & \cdot & \cdot & \cdot \\ d & \cdot & \cdot & \cdot & \cdot \end{array}$$

"+" ⇒ +1

"·" ⇒ 0

"–" ⇒ –1

$s = ag + bh$

$$= \begin{pmatrix} \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

$t = ce + df$

$$= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \end{pmatrix}$$

Scanned by CamScanner

$u = cg + dh$

$$= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}$$

$P_1 = A_1 B_1$

$= a(g-h)$

$= ag - ah$

$$= \begin{pmatrix} \cdot & \cdot & + & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

$P_2 = A_2 B_2$

$= (a+b) \cdot h$

$= (ah + bh)$

$$= \begin{pmatrix} \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

$\Rightarrow S = P_1 + P_2$

Scanned by CamScanner

$$P_3 = A_3 B_3$$
$$= (c+d)\cdot e$$
$$= ce + de$$
$$= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \end{pmatrix}$$

and

$$P_4 = A_4 B_4$$
$$= d\cdot(f-e)$$
$$= df - de$$
$$= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & + & \cdot & \cdot \end{pmatrix}$$

$$\Rightarrow \quad t = P_3 + P_4$$

$$P_5 = A_5 B_5$$
$$= (a+d)(e+h)$$
$$= ae + ah + de + dh$$
$$= \begin{pmatrix} + & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & + \end{pmatrix}$$

Scanned by CamScanner

$$P_6 = A_6 B_6$$
$$= (b-d).(f+h)$$
$$= bf + bh - df - dh$$

$$= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & - & \cdot & - \end{pmatrix}$$

$$P_5 + P_4 - P_2 = ae + dh + df - bh$$

$$\left(= ae + \cancel{ah} + \cancel{de} + dh + df - \cancel{de} - \cancel{ah} - bh \right)$$

$$= \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & + \end{pmatrix}$$

$$r = P_5 + P_4 - P_2 + P_6$$
$$= ae + \cancel{dh} + \cancel{df} - \cancel{bh} + bf + \cancel{bh} - \cancel{df} - \cancel{dh}$$

$$= ae + bf$$

$$\begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

Scanned by CamScanner

$$P_7 = A_7 B_7$$
$$= (a-c)\cdot(e+g)$$
$$= ae + ag - ce - cg$$
$$= \begin{pmatrix} + & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & - & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

$$P_5 + P_1 - P_3 = ae + ag - ce + dh$$
$$= \begin{pmatrix} + & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}$$

$$u = P_5 + P_1 - P_3 - P_7$$
$$= \cancel{ae} + \cancel{ag} - \cancel{ce} + dh - \cancel{ae} - \cancel{ag} + \cancel{ce} + cg$$
$$= cg + dh$$
$$= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}$$

Scanned by CamScanner

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & g \\ f & h \end{pmatrix}$$

So, we have :-

$r = P_5 + P_4 - P_2 + P_6$

$s = P_1 + P_2$

$t = P_3 + P_4$

$u = P_5 + P_1 - P_3 - P_7$

where,

$P_1 = a(g-h)$

$P_2 = (a+b).h$

$P_3 = (c+d).e$

$P_4 = d(f-e)$

$P_5 = (a+d).(e+h)$

$P_6 = (b-d).(f+h)$

$P_7 = (a-c).(e+g)$

$\boxed{7}$

Strassen's algo is often not the method of choice for matrix multiplication for 4 reasons:-

1. Constant factor hidden in running time of Strassen's algo is larger than the constant factor in naive $\Theta(n^3)$ method.
2. When matrices are sparse, methods tailored for sparse matrices are faster.
3. Strassen's algo is not quite as numerically stable as the naive method.

Scanned by CamScanner

(4) Submatrices formed at the levels of recursion consume space.

Q: $\begin{pmatrix} 1 & 3 \\ 5 & 7 \end{pmatrix} \begin{pmatrix} 8 & 4 \\ 6 & 2 \end{pmatrix}$

Scanned by CamScanner

Medians and Order Statistics

The ith order statistic of a set of n distinct elements is the ith smallest element.

eg. minimum of a set of elements is the first order statistic $(i=1)$ and the maximum is the nth order statistic $(i=n)$.

Median: is the halfway point of the set.

If n is odd, median is unique at $i = \left(\frac{n+1}{2}\right)$

If n is even, there are 2 medians at $i = n/2$ and $i = n/2 + 1$

Scanned by CamScanner

Minimum

Algo to find min^n in array

```
Minimum (A, n)
min ← A[1]
for i ← 2 to n
    do if min > A[i]
        then min ← A[i]
return min
```

No. of comparisons = $n-1$

Algo to find max^m

```
Maximum (A, n)
max ← A[1]
for i ← 2 to n
    do if A[i] > max
    then max ← A[i]
```

No. of comparison = $n-1$

Scanned by CamScanner

Simultaneous minimum + maximum

We can find minimum and maximum independently using $n-1$ comparisons each, so total $\Rightarrow 2n-2$ $(2(n-1))$ comparisons

```
MaxMin (A, n)
min ← A[1]
max ← A[1]
~~for i ← 2 to length[A]~~
for i ← 2 to n
do if min > A[i]
   then min ← A[i]
   if A[i] > max
   then max ← A[i]
```

No of comparisons $= 2(n-1)$
$= 2n-2$

Scanned by CamScanner

We compare pairs of element from the input first with each other and then compare the smaller to current minimum and the larger to current maximum

```
MaxMinPair (A, n)
    max ← A[1]
    min ← A[1]
    for i ← 1 to n/2
        do if A[2i] > A[2i-1]
        then mint ← A[2i-1]
             maxt ← A[2i]
        else mint ← A[2i]
             maxt ← A[2i-1]
        if maxt > max
            then max ← maxt
        if min > mint
            then min ← mint
```

Total comparison $= \frac{3n}{2}$

(It divides n elements into n/2 and 3 comparisons
(1) Pairs are compared with each other
(2) Smaller to current min
(3) Larger to " max)

C A.262 = 7 (101) n
B 1 (1x1) i

65

Selection in expected linear time

It is used to find i^{th} smallest element from a set of n distinct elements

The selection problem can be specified as follows:-

Input: A set A of n (distinct) numbers and a number i, with $1 \le i \le n$.

Output:- The element $x \in A$ that is larger than exactly $i-1$ other elements of A.

One solution is to sort array using merge sort and simply index the ith element in output array. It takes

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

Second solution

→ RANDOMIZED-SELECT uses procedure Randomized-Partition. It is like Quicksort, but quicksort recursively processes both sides of partition, but Randomized-Select only works on one side of the partition

Randomized-Select(A, p, r, i)

1. if p = r
2. then return A[p]
3. q ← Randomized-Partition(A, p, r)
4. k ← q − p + 1
5. if i ≤ k
6. then return Randomized-Select(A, p, q, i)
7. else return Randomized-Select(A, q+1, r, i − k)

Worst case running time $= \Theta(n^2)$ (To find min, we could be unlucky and partition around largest remaining element)

Working

(1) A[p ... r]

A[p ... q] A[q+1 ... r]

such that each element of A[p ... q] is less than each element of A[q+1 ... r]

(2) k → no. of elements in left subarray A[p ... q]

(3) Determine in which 2 subarrays A[p ... q] and A[q+1 ... r], the ith smallest element lies.

Scanned by CamScanner

(4) If $i \le k$ → low side of partition then select unip (6)

(5) If $i > k$ → high side of partition, we know k values are smaller than i^{th} smallest element of $A[p \ldots r]$ ie desired element is $(i-k)^{th}$ smallest element of $A[q+1 \ldots r]$ found in line 7.

eg. 3 2 5 4 $\boxed{1}$ (Pivot)

Exchange

4 2 5 3 1

$i=1$ $j=5$

1 2 5 3 4

$i=3$ $j=4$

1 2 3 | 5 4

$j=3$ $i=4$

1 2 3 5 4 eg $i^{th} = 2$

$k = 3-1+1 = 3$

$4 \nless 3$

~~(A, 1, 3, 2)~~

A (4, 5, 1)

5 | 4

$i=1$ $j=2$

4 5

$j=1$ $i=2$

4 5

$k=1$

$j^{th} = 1$

RS (A, 1, 1, 1)

Ans: A[1] = 4

(larger than 3 other elements)

1, 2, 3

Avg. Case

Randomized-Partition produces a partition where probability of occurence (for each element to be chosen as pivot) $= \frac{1}{n}$ (for n elements)

If we assume that in Randomized-Select, the ith element is ~~always~~ present in larger side of partition, we get:

k $\quad n-k \quad \frac{1}{n} \quad 1 \quad n-1$

69

$$T(n) \le \frac{1}{n}\left(T(\max(1, n-1)) + \sum_{k=1}^{n-1} T(\max(k, n-k))\right) + O(n)$$

for partition procedure

$$T(n) \le \frac{1}{n}\left(T(n-1) + \sum_{k=1}^{\lceil n/2 \rceil - 1} T(n-k) + \sum_{k=\lceil n/2 \rceil}^{n-1} T(k)\right) + O(n)$$

as $\max(1, n-1) = n-1$

and for $\max(k, n-k)$

for $k \ge n-k$

$2k \ge n$

$\boxed{k \ge \frac{n}{2}}$

for $k < n-k$

$2k < n$

$\boxed{k < \frac{n}{2}}$

ie $\max(k, n-k) = \begin{cases} k & \text{if } k > \lceil n/2 \rceil \\ n-k & \text{if } k < \lceil n/2 \rceil \end{cases}$

Scanned by CamScanner

eg. $f(n) = 2n^2 + 3n + 1$
$\leq 2n^2 + 3n^2 + n^2$
$\leq 6n^2$

$$T(k) \leq \frac{1}{n}\left(T(n-1) + \sum_{k=1}^{\lceil n/2 \rceil - 1} T(n-k) + \sum_{k=\lceil n/2 \rceil}^{n-1} T(k)\right) + O(n)$$

as if $f(n) = \sum_{k=1}^{\lceil n/2 \rceil - 1} T(n-k) + \sum_{k=\lceil n/2 \rceil}^{n-1} T(k)$

$$\leq 2 \sum_{k=\lceil n/2 \rceil}^{n-1} T(k)$$

$$T(k) \leq \frac{1}{n}\left(T(n-1) + 2\sum_{k=\lceil n/2 \rceil}^{n-1} T(k)\right) + O(n)$$

$$= \frac{2}{n} \sum_{k=\lceil n/2 \rceil}^{n-1} T(k) + O(n)$$

We solve by substitution
we assume
$T(n) \leq cn$
for some constant c.

Scanned by CamScanner

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} - \frac{(n-1)n}{2}$$

$$T(n) \le \frac{2}{n} \sum_{k=\lceil n/2 \rceil}^{n-1} ck + \Theta(n)$$

$$\le \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k\right) + \Theta(n)$$

$$= \frac{2c}{n}\left(\frac{1}{2}(n-1)n - \frac{1}{2}\left(\left\lceil \frac{n}{2} \right\rceil - 1\right)\left\lceil \frac{n}{2} \right\rceil\right) + \Theta(n)$$

$$\le c(n-1) - \frac{c}{n}\left(\frac{n}{2} - 1\right)\left(\frac{n}{2}\right) + \Theta(n)$$

$$\le cn - c - \frac{cn}{4} + \frac{c}{2} + \Theta(n)$$

$$= c\left(\frac{3n}{4} - \frac{1}{2}\right) + \Theta(n)$$

$$\le cn$$

$$T(n) = O(n)$$

$$\sum_{k=1}^{n-1} = \sum_{k=1}^{\lfloor n/2 \rfloor - 1} + \sum_{k=\lceil n/2 \rceil}^{n-1}$$

Selection in worst-case linear time

To ~~get~~ design a Select algorithm which takes $O(n)$ time in worst-case. The idea behind this algo is to guarantee a good split when array is partitioned. Select algo uses the Partition algo from Quicksort.

Select algo has 1 all steps:

1. Divide the n elements of the input array into $\lceil n/5 \rceil$ groups of 5 elements each and atmost one group made up of remaining $n \bmod 5$ elements.
2. Find median of each of $\lceil n/5 \rceil$ groups by insertion sorting and taking middle element (If the group has even no. of elements, take larger of 2 medians)
3. Use Select recursively to find median x of the $\lceil n/5 \rceil$ medians found in step 2.

Scanned by CamScanner

(4) Partition the input array around the median of medians x using a modified version of PARTITION. Let k be no. of elements on low side of partition, so that $n-k$ be no. of elements on high side.

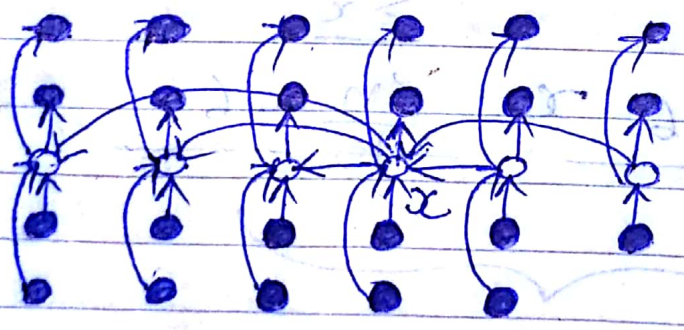
(5) Use Select recursively to find the i'th smallest element on low side if $i \le k$ or $(i-k)^{th}$ smallest element on high side if $i > k$.

---

eg. if $n = 29$

$$\frac{n}{5} = \left\lceil \frac{29}{5} \right\rceil = \lceil 5.8 \rceil = 6$$

$29 \bmod 5 = 4$ last gp has element



n elements are represented by small circles & each gp occupies a column. The medians of gps are whitened and median of medians x is labeled.

Scanned by CamScanner

3 on of every gp. to right of x are $> x$
& 3 " " gp to left " x are $< x$

Arrows are drawn from larger to smaller elements. Atleast half of $\lceil n/5 \rceil$ gps contribute 3 elements that are greater than x, except for one group that has fewer than 5 elements (if 5 does not divide n exactly) and one group containing x itself.

ie no of elements greater than x is atleast

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil - 2\right)$$

$$\geq \frac{3n}{10} - 6$$

$$\underbrace{\frac{7n}{10} + 6 \quad x \quad \frac{3n}{10} - 6}_{\text{Total} = n}$$

$> x$

$$n - \frac{3n}{10} + 6$$

$$= \frac{10n - 3n}{10} + 6$$

$$= \frac{7n}{10} + 6$$

Scanned by CamScanner

So, in worst case, Select is called recursively on atmost $\frac{7n}{10}+6$ elements in step 5.

Step 1, 2, and 4 take $O(n)$ time
Step 3 takes $T(\lceil n/5 \rceil)$ time

Step 5 takes $T\left(\frac{7n}{10}+6\right)$ time

$$T(n) \leq T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\frac{7n}{10}+6\right) + O(n)$$

We prove it by substitution
Assume that $T(n) \leq cn$

$$T(n) \leq c\left(\left\lceil \frac{n}{5} \right\rceil\right) + c\left(\frac{7n}{10}+6\right) + O(n)$$

$$\leq \frac{cn}{5} + c + \frac{c7n}{10} + 6c + O(n)$$

$$\leq \frac{9cn}{10} + 7c + O(n)$$

$$\leq cn$$

$$T(n) = O(n)$$

Scanned by CamScanner

Dynamic Programming

- It is like divide and conquer method, it solves problems by combining solutions to subproblems.

Diff

(1) But divide and conquer partitions problem into subproblems, solve the subproblems recursively and then combine their solutions to solve original problem.

- But D.P. is applicable when subproblems are not independent ie when subproblems share subsubproblems.

(2) A divide + conquer algo does more work than necessary repeatedly solving the common subsubproblem.

- But D.P. algo solves every subsubproblem just once + then saves its answer in a table, thereby avoiding work of recomputing the answer every time the subsubproblem is encountered.

Scanned by CamScanner

3) Divide and conquer is top-down approach.
But D.P is bottom up technique.
We start with smallest subproblems and combining their solutions, until finally we arrive at solution of original problem.

- D.P is typically applied to optimization problems, where there are many possible solution and each solution has a value and we wish to find an optimal solution (with min or max value).

For D.P technique to be applicable, a problem must have:

1) Optimal substructure: optimal solution to the problem contains within it optimal solution to subproblems.

, in MCM, the optimal parenthesization of $A_i A_{i+1} \ldots A_j$ that splits product b/w A_k and A_{k+1} contains within it optimal solution to problem of parenthesizing $A_i A_{i+1} \ldots A_k$ and $A_{k+1} A_{k+2} \ldots A_j$

(2) Overlapping subproblem:
when a recursive algo revisits the same problem over + over again, we say that the problem has overlapping subproblem.

So, solves each subproblem once + stores soln in a table which can be looked up when needed.

Eg, in MCM, it looks up the solution to subproblems in lower rows when solving subproblems in higher rows.

Eg. m[3,4] is referenced when m[2,4], m[1,4] -- are computed

(X)

Proof of Master Theorem

The 1st part of proof of master theorem analyzes the master recurrence:-

$$T(n) = aT(n/b) + f(n)$$

under assignment that n is an exact power of $b > 1$.

Lemma 1

Let $a \geq 1$ and $b > 1$ be constants + let $f(n)$ be a non negative function defined on exact powers of b. Define $T(n)$ on exact powers of b by the recurrence:

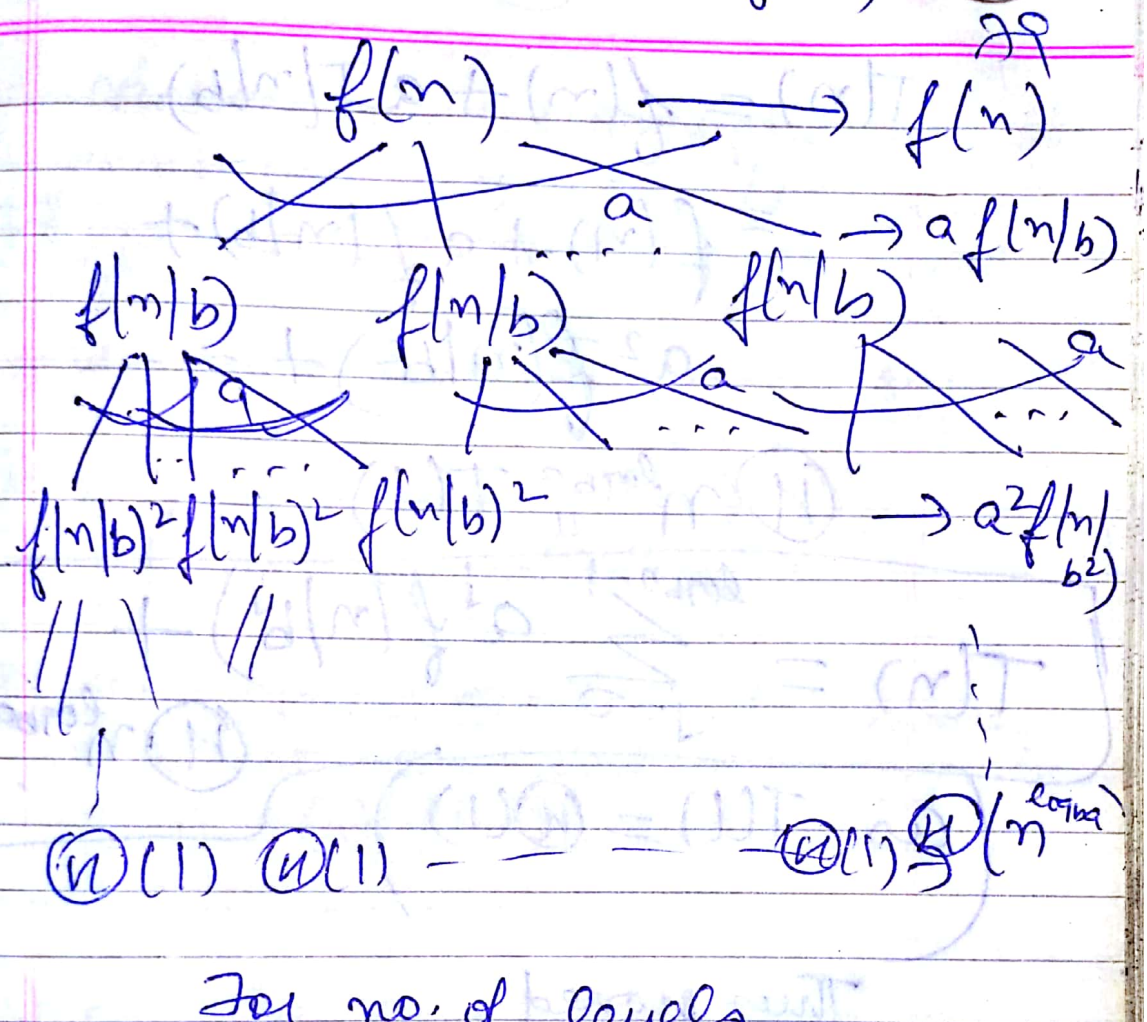
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ aT(n/b) + f(n) & \text{if } n = b^i \end{cases}$$

where i is the integer, then,

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

Scanned by CamScanner

$$T(n) = aT(n/b) + f(n)$$



For no. of levels

$$\Rightarrow T\left(\frac{n}{b^i}\right) = T(1) = 1$$

ie $\frac{n}{b^i} = 1$

$$n = b^i$$

$$i = \log_b n$$

$$(0, 1 - - - \log_b n)$$

$$T(n) = f(n) + aT(n/b)$$

$$= f(n) + af(n/b) + a^2 f(n/b^2) + \cdots$$

$$\Theta\, n^{\log_b a}\, T(1)$$

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

(as $T(1) = \Theta(1)$)

Thus proved

Lemma 2

If $a \geq 1$ and $b > 1$ be constant and let $f(n)$ be a non-ve function defined on exact powers of b. A function $g(n)$ defined over exact powers of ~~by~~ b by:

$$g(n) = \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) \quad \text{— (2)}$$

Scanned by CamScanner

can be bounded asymptotically for exact powers of b as follows:-

(1) If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $g(n) = \Theta(n^{\log_b a})$

(2) If $f(n) = \Theta(n^{\log_b a})$, then $g(n) = \Theta(n^{\log_b a} \lg n)$

(3) If $a f(n/b) \leq c f(n)$ for some constant $c < 1$ and all $n \geq b$, then $g(n) = \Theta(f(n))$

<u>Proof</u>: For case 1 we have

$$f(n) = O(n^{\log_b a - \epsilon})$$

$$f(n/b^j) = O((n/b^j)^{\log_b a - \epsilon}) \quad \text{--- (1)}$$

Substituting (1) in (2), we get

$$g(n) = O\left(\underbrace{\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a - \epsilon}}\right) \quad \text{--- (3)}$$

$$\Rightarrow n^{\log_b a - \epsilon} \sum_{j=0}^{\log_b n - 1} \left(\frac{a b^{\epsilon}}{b^{\log_b a}}\right)^j$$

$$\cup$$

$$a^{\log_b a} = a$$

Scanned by CamScanner

$$= n^{\log_b a - \epsilon} \sum_{j=0}^{\log_b n - 1} (b^{\epsilon})^j$$

$$= n^{\log_b a - \epsilon} \left(\frac{b^{\epsilon \log_b n} - 1}{b^{\epsilon} - 1} \right)$$

$$= n^{\log_b a - \epsilon} \left(\frac{n^{\epsilon} - 1}{b^{\epsilon} - 1} \right)$$

$$= n^{\log_b a - \epsilon} O(n^{\epsilon})$$

$$= O(n^{\log_b a}) \quad \text{--- (4)}$$

Subⁿ (4) in (3), yields

$$g(n) = O(n^{\log_b a})$$

case 1 is proved

Case 2

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

$$f(n/b^j) = \Theta((n/b^j)^{\log_b a})$$

subⁿ in (2), we get

$$\Rightarrow g(n) = \Theta\left(\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a} \right)$$

Scanned by CamScanner

$$\Rightarrow \sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a}$$

$$\Rightarrow n^{\log_b a} \sum_{j=0}^{\log_b n-1} \left(\frac{a}{b^{\log_b a}}\right)^j$$

$$\Rightarrow n^{\log_b a} \sum_{j=0}^{\log_b n-1} 1$$

$$\Rightarrow n^{\log_b a} \log_b n$$

$$g(n) = \Theta\left(n^{\log_b a} \cdot \log_b n\right)$$

$$= \Theta\left(n^{\log_b a} \cdot \lg n\right)$$

Case 2 is proved

Case-3

$$g(n) = \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) \quad \text{--- (5)}$$

$$g(n) = \Omega(f(n)) \quad \text{--- (7)} \quad \text{(for exact powers of } b\text{)}$$

(If all terms of $g(n)$ are non-ve)

(X)

$$a f(n/b) \le c f(n)$$

$c < 1$ and all $n \ge b$

we have $a^j (f(n/b^j)) \le c^j f(n)$

from (5)

$$g(n) \le \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)$$

$$\le \sum_{j=0}^{\log_b n - 1} c^j f(n)$$

$$\le f(n) \sum_{j=0}^{\infty} c^j$$

$$= f(n) \left(\frac{1}{1-c} \right) \qquad \boxed{\frac{a}{1-r}}$$

$$g(n) = O(f(n)) \quad - (6)$$

we have

$$f(n) = \Theta(f(n))$$

Scanned by CamScanner

Lemma 3

Let $a \geq 1$ and $b > 1$ be constants and let $f(n)$ be a non-ve function defined on exact powers of b. Define $T(n)$ on exact powers of b by the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ aT(n/b) + f(n) & \text{if } n = b^i \end{cases}$$

where i is a +ve integer. Then $T(n)$ can be bounded asymptotically for exact powers of b as follows:-

(1) If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$

(2) If $f(n) = \Theta(n^{\log_b a})$, then

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

(3) If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$ and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all large n, then $T(n) = \Theta(f(n))$

Proof: We use bounds in lemma 2 to evaluate summations from lemma 1

(X)

Case 1

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

Case 2

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

Case 3

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

Scanned by CamScanner

Matrix - chain multiplication

MCM problem can be stated as:-
Given a chain $(A_1, A_2 \ldots A_n)$ of n matrices where $i = 1, 2, 3 \ldots n$ ~~where~~ Matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1, A_2 \ldots A_n$ in a way that minimizes the number of scalar multiplications.

Total number of parenthesizations $P(n)$ for a sequence of n matrices is given by:-

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)\, P(n-k) & \text{if } n \geq 2 \end{cases}$$

eg. for 4 matrices (we can have 5 ways)

$$P(4) = \sum_{k=1}^{3} P(k)\, P(4-k)$$

$$= P(1)\, P(3) + P(2)\, P(2) + P(3)\, P(1)$$

$$= 2 + 1 + 2 = 5$$

$$P(3) = \sum_{k=1}^{2} P(k)\, P(3-k)$$

$$= P(1)\, P(2) + P(2)\, P(1)$$

$$= 2$$

Scanned by CamScanner

$$P(2) = \sum_{k=1}^{1} P(k)\, P(2-k)$$

$$= P(1)\, P(1)$$

$$= 1$$

Let there be 4 matrices A_1, A_2, A_3, A_4, to compute it, it can be done in 5 ways:-

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. $(A_1 ((A_2 A_3) A_4))$

Note: If $A_1 - x \times y$

$A_2 - y \times z$

$A_1 A_2 - x \times z$

Total no. of scalar multiplication $= xyz$

eg. We have to compute matrix chain product $A_1 A_2 A_3$

Scanned by CamScanner

order of A_1 is 10×100
" " A_2 " 100×5
" " A_3 " 5×50

It can be computed in 2 ways:-

(i) $(A_1(A_2 A_3))$

To compute $A_2 A_3$, S.M $= 100 \times 5 \times 50$
$= 25,000$

" " $A_1(A_2 A_3)$, S.M $= 10 \times 100 \times 50$
100×50 $= 50,000$

Total SM $= 75,000$

(ii) $((A_1 A_2) A_3)$

To compute $(A_1 A_2)$, S.M $= 10 \times 100 \times 5$
$= 5000$

To compute $((A_1 A_2) A_3)$
$= 10 \times 5 \times 50$
$= 2500$

Total SM $= 7500$

Using Dynamic Programming to solve MCM problem:-

(1) We have to find optimal parenthesization of $A_1, A_2, \ldots A_n$. An optimal parenthesization of $A_1, A_2 \ldots A_n$ splits product between A_k and A_{k+1} for some k between $1 \le k < n$ ie first of all compute optimal parenthesization of $A_1, A_2 \ldots A_k$, then compute optimal parenthesization of $A_{k+1} \ldots A_n$ and finally multiply $A_{1 \ldots k}$ and $A_{k+1 \ldots n}$ to produce the final product $A_{1 \ldots n}$.

(2) Recursively define value of an optimal solution. Let $m[i,j]$ be the minimum no. of S.M. needed to compute $A_{i \ldots j}$. i.e. $m[2,5]$ is min no. of S.M. needed to compute $A_{2 \ldots 5}$ ie $A_2 A_3 A_4 A_5$.

If $i = j$, then chain consists of just one matrix $A_{i \ldots i} = A_i$ $\therefore$

$m[i,i] = 0$ for $i = 1, 2, 3 \ldots n$

0			
	0		
		0	
			0

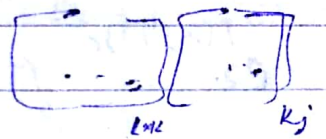
Matrix m

i will be always less than or equal to j.
When $i < j$, let the optimal parenthesization splits product $A_i A_{i+1} \ldots A_j$ between A_k and A_{k+1} where $i \le k < j$

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$$

Because k can be $i, i+1 \ldots j-1$

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min \begin{cases} m[i,k] + \\ m[k+1,j] + \\ p_{i-1} p_k p_j \end{cases} & \text{if } i < j \\ i \le k < j \end{cases}$$



$m[i,j]$ gives the minimum no. of scalar multiplications required to find $A_i A_{i+1} \ldots A_j$

To keep track of how to construct an optimal solution we use array $s[1 \ldots n][1 \ldots n]$
We store value k at $s[i,j]$ at which we can split product $A_i A_{i+1} \ldots A_j$.

We compute optimal solution in bottom up fashion i.e first we compute $m[i,j]$ for $i=j$.
Then we compute $m[i,i+1]$ for $i=1,2,3 \ldots n-1$.

eg. We have 6 matrices $A_1 \ldots A_6$ given as follows:

A_1 30×35
A_2 35×15
A_3 15×5
A_4 5×10
A_5 10×20
A_6 20×25

15×5×15
P2

30	35	15	5	10	20	25
P_0	P_1	P_2	P_3	P_4	P_5	P_6

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

$k=1$

$\Rightarrow 0 + 0 + 30 \times 35 \times 15$
$\Rightarrow 15750$

$m[2,3] = m[2,2] + m[3,3] + P_1 P_2 P_3$

$k=2$

$= 0 + 0 + 35 \times 15 \times 5$
$= 2625$

Scanned by CamScanner

$m[3,4] = m[3,3] + m[4,4] + p_2 p_3 p_4$

$k=3$ $\quad = 0 + 0 + 15 \times 5 \times 10 = 750$

$k=4$ $\quad m[4,5] = m[4,4] + m[5,5] + p_3 p_4 p_5$

$= 0 + 0 + 5 \times 10 \times 20$

$= 1000$

$k=5$

$m[5,6] = m[5,5] + m[6,6] + p_4 p_5 p_6$

$= 0 + 0 + 10 \times 20 \times 25 = 5000$

$m[1,3] = \min \{ m[1,1] + m[2,3] + p_0 p_1 p_3,\ m[1,2] + m[3,3] + p_0 p_2 p_3 \}$

$1 \le k < 3$

$k=1$

$k=2$ $\quad = \min \{ 0 + 2625 + 30 \times 35 \times 5,\ 15750 + 0 + 30 \times 15 \times 5 \}$

$= \min \{ 7875, 18000 \}$

$= 7875$

$m[2,4] = \min \{ m[2,2] + m[3,4] + p_1 p_2 p_4,\ m[2,3] + m[4,4] + p_1 p_3 p_4 \}$

$\Rightarrow \min \{ 0 + 750 + 35 \times 15 \times 10,\ 2625 + 0 + 35 \times 5 \times 10 \}$

$\Rightarrow \min \{ 6000, 4375 \} = 4375$

Scanned by CamScanner

$m[3,5] = \min\{ m[3,3] + m[4,5] + p_2 p_3 p_5,\ m[3,4] + m[3,5] + p_2 p_4 p_5 \}$

$= \min\{ 0 + 1000 + 15 \times 5 \times 20,\ 750 + 0 + 15 \times 10 \times 20 \}$

$= \min\{ 2500, 3750 \}$

$= 2500$

$m[4,6] = \min\{ m[4,4] + m[5,6] + p_3 p_4 p_6,\ m[4,5] + m[6,6] + p_3 p_5 p_6 \}$

$\Rightarrow \min\{ 0 + 5000 + 5 \times 10 \times 25,\ 1000 + 0 + 5 \times 20 \times 25 \}$

$\Rightarrow \min\{ 6250, 3500 \}$

$\Rightarrow 3500$

$m[1,4] = \min\{ m[1,1] + m[2,4] + p_0 p_1 p_4,\ m[1,2] + m[3,4] + p_0 p_2 p_4,\ m[1,3] + m[4,4] + p_0 p_3 p_4 \}$

1, 2, 3

$\Rightarrow \min \left\{ \begin{array}{l} 0+4375+30\times35\times10,\ 15750+ \\ 750+30\times15\times10,\ 7875+0+ \\ 30\times5\times10 \end{array} \right\}$

$\Rightarrow \min \{14875, 21000, 9375\}$

$\Rightarrow 9375$ $\quad k=3$

$m[2,5] = \min \{ m[2,2]+m[3,5]+ p_1 p_2 p_5,\ m[2,3]+m[4,5]+p_1 p_3 p_5,\ m[2,4]+m[5,5]+p_1 p_4 p_5 \}$

$= 7125$

$k=3$

Minimum no. of S.M. reqd. to compute $A_{1..6} = 15125$ & optimal parenthesization is:-

$(A_1 A_2 A_3)(A_4 A_5 A_6)$

$A_{16} \rightarrow k=3$

$\downarrow$

$A_{13} \rightarrow k=1$

$((A_1)(A_2 A_3))(A_4 A_5 A_6)$

$A_{46} \rightarrow k=5$

$\boxed{((A_1)(A_2 A_3))((A_4 A_5)(A_6))}$

Scanned by CamScanner

m[0, 0+1] m[6, n] =

	1	2	3	4	5	6
1	0	15750	7875	9375	11875	15125
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

Array m

	1	2	3	4	5	6
1		1	1	3	3	3
2			2	3	3	3
3				3	3	3
4					4	5
5						5
6						

Array s ↓ we store values of k

```
Print_optimal_paren (s, i, j)
  if i = j
  then print "A"i
  else print "("
Print_optimal_paren (s, i, s[i, j])
Print_optimal_paren (s, s[i, j]+1, j)
  print ")"
```

Scanned by CamScanner

Algo

Matrix-chain-order(p, n)
for i ← 1 to n
do m[i, i] ← 0
for l ← 2 to n
do for i ← 1 to n-l+1
do j ← i+l-1
m[i, j] ← ∞
for k ← i to j-1
do q ← m[i, k] + m[k+1, j]
+ $p_{i-1} p_k p_j$
if q < m[i, j]
then m[i, j] ← q
s[i, j] ← k
return m + s

l → chain length
1st iteration ; m[i, i+1]
2nd " ; m[i, i+2]

Running time = $O(n^3)$

j)

Eg. $(S, 1, 6)$

Print $(\quad (S, 1, 3) + (S, 4, 6)$

$(S, 1, 3)$

Print $((A_1 \quad (S, 1, 1) + (S, 2, 3)$

$(S, 1, 1)$

Print $((A_1$

$(S, 2, 3)$

Print $((A_1(\quad (S, 2, 2) + (S, 3, 3)$

$(S, 2, 2)$

Print $((A_1(A_2$

$(S, 3, 3)$

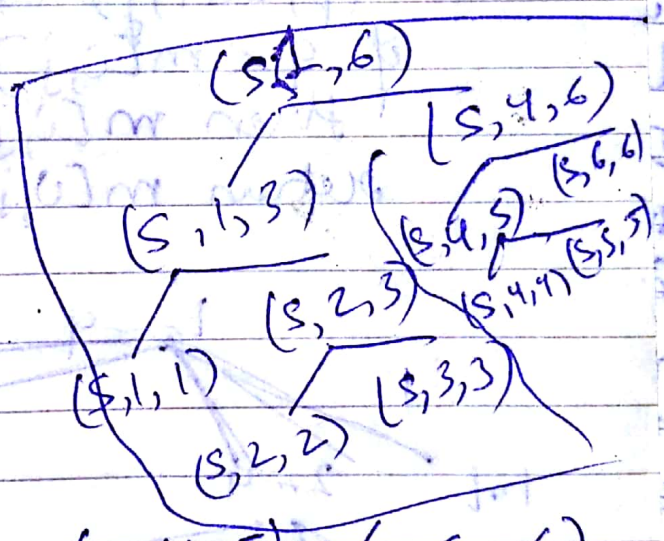
Print $((A_1(A_2A_3)) \quad (S, 1, 1)$

$(S, 4, 6)$. $\quad (S, 4, 5) + (S, 6, 6)$

$((A_1(A_2A_3))C$

$(S, 4, 5)$

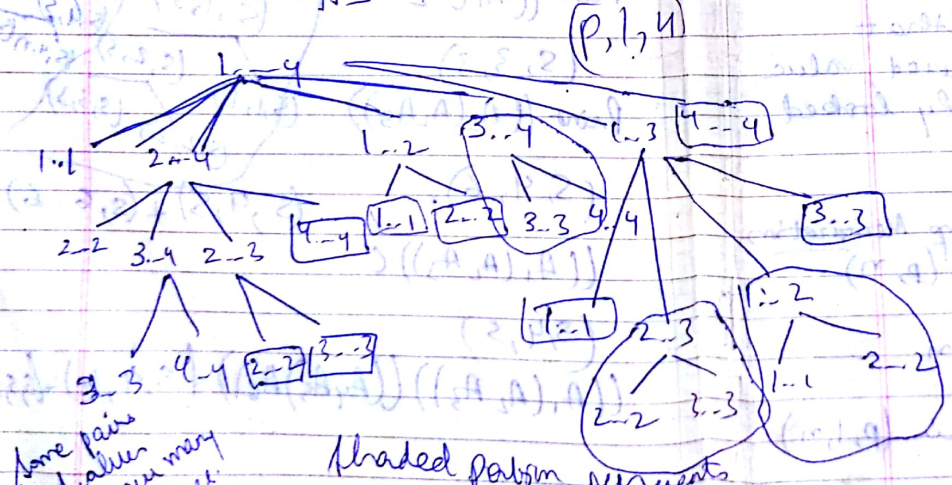
$((A_1(A_2A_3))((A_4A_5)A_6)) \quad (S, 4, 4) + (S, 5, 5)$



Scanned by CamScanner

If every entry has to be recomputed rather than looked up, it would increase running time dramatically. To see this, consider follⁿ recursive procedure:-

```
Recursive - Matrix Chain (p,i,j)
if i=j
then return 0
m[i,j] ← ∞
for k ← i to j-1
  do q ← R-M-C (p, i, k) + R-M-C
     (p, k+1, j) + p_{i-1} p_k p_j
  if q < m[i,j]
  then m[i,j] ← q
return m[i,j]
```



some pairs of values occur many times.

Shaded portion represents values that are computed again.

But in memoization, shaded portion entries are looked up that are recomputed.

It is recursive approach, so top-down.

Memoization

* It is variation of dynamic programming
* It is a top-down technique but problem must have optimal substructure and overlapping subproblems property.
* It solves every subproblem just once & saves its answer in a table to avoid recomputing answer every time, the subproblem is encountered.
* Each table entry initially contains special value to indicate that the entry has yet to be filled in. When subproblem is first encountered during execution of recursive algorithm, its solution is computed and then stored in table.
* Each subsequent time that subproblem is encountered, value stored in table is simply looked up and returned.

Solving MCM problem using Memoization

```
Memoized-Matrix Chain (p, n)
    for i ← 1 to n
        do for j ← i to n
            do m[i, j] ← ∞
    return look-up-chain (p, 1, n)
```

we have not found entry, ∞ is value is ∞

Algo for look-up-chain

```
look-up-chain (p, i, j)
    if m[i, j] < ∞
  then return m[i, j]
    if i = j
  then m[i, j] ← 0
  else for k ← i to j-1
  do q ← look-up-chain(p, i, k) +
look-up-chain (p, k+1, j) +
p(i-1) pk pj
    if q < m[i, j]
  then m[i, j] ← q
  return m[i, j]
```

Scanned by CamScanner

Optimal binary search tree

Suppose we are designing a program to translate text from English to French. One way to perform these lookup operations is to build a binary search tree. We want frequently occuring words in the text to be placed nearer the root and also words which have no French translation might not appear in BST at all. We need an OBST.

Given a sequence $K = (k_1, k_2 \ldots k_n)$ of n distinct keys in sorted order (ie $k_1 < k_2 \ldots < k_n$) and we wish to build a BST from these. Each key k_i has a prob. p_i. Some searchs may ~~not~~ be for values not in K, so we have $(n+1)$ dummy keys $d_0, d_1, \ldots d_n$.

d_0 represents all values less than k_1

d_n " " " " " k_n

and for $i = 1, 2 \ldots n-1$,

d_i represents all values between k_i and k_{i+1}.

For each d_i, we have prob. q_i.

k_i is an internal node and each dummy key d_i is a leaf.

T, BST

Every search is either successful (finding some key k_i) or unsuccessful (finding some dummy key d_i), so we have

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1 \quad \text{—— (1)}$$

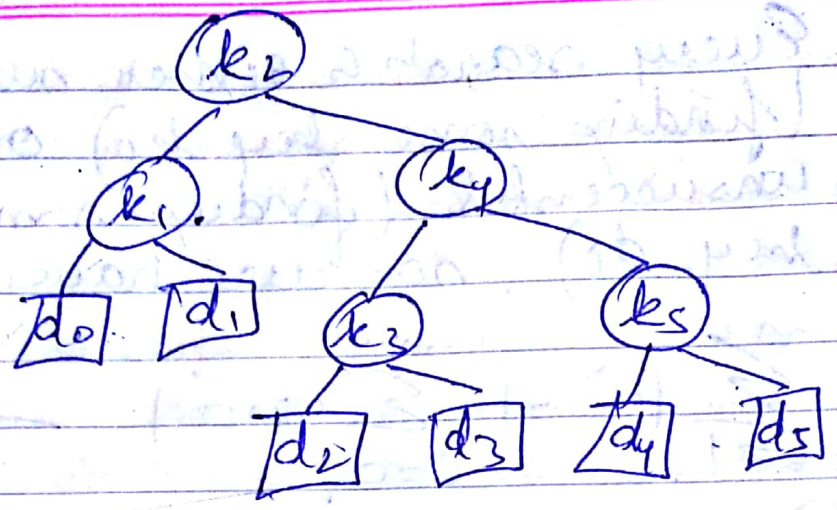
Let us assume that actual cost of search is the no. of nodes examined ie depth of node found by search in T, plus 1. Then the expected cost of search in T is:-

$$E[\text{search cost in } T] = \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (\text{depth}_T(d_i) + 1) \cdot q_i$$

From (1), we have

$$\Rightarrow 1 + \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=0}^{n} \text{depth}_T(d_i) \cdot q_i$$

depth_T: node's depth in tree T.



i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

(depth) × prob =

node	depth	prob	Contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40

$\boxed{2.80}$

This BST has Expected cost of 2.80, we want to find BST with lowest cost. Such a tree is OBST.

Structure of an optimal binary search tree

Consider any subtree of BST, which has keys $k_i \ldots k_j$ for $1 \le i \le j \le n$, and has dummy keys $d_{i-1}, \ldots d_j$

Given keys $k_i \ldots k_j$, one of these keys k_r $(i \le r \le j)$ will be root.

(root) k_r

$k_i \ldots k_{r-1}$ (left subtree) $\quad$ $k_{r+1} \ldots k_j$ (rt. subtree)

Recursive Solution

Let us define $e[i,j]$ as expected cost of searching an OBST containing keys $k_i \ldots k_j$

When ~~i=j~~ $j = i-1$, there are no actual keys, only dummy key d_{i-1}, then:

$$e[i, i-1] = q_{i-1}$$

$j = i-1$ $\quad$ (prob. of d_{i-1} $= q_{i-1}$)

When ~~goal~~ $i \le j$, we need to select a root k_r from among $k_i - - k_j$ and make OBST with keys $k_i - - k_{r-1}$ as lt-subtree + OBST with keys $k_{r+1} - - k_j$ as rt-subtree.

For subtree with keys $k_i - - k_j$, let us denote sum of prob. as

$$w(i,j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$$

If k_r is root, then;

$$e[i,j] = p_r + (e[i,r-1] + w(i,r-1)) + (e[r+1,j] + w(r+1,j))$$

$$w(i,j) = w(i,r-1) + p_r + w(r+1,j)$$

we have

$$e[i,j] = e[i,r-1] + e[r+1,j] + w(i,j)$$

Scanned by CamScanner

$$e[i,j] = \begin{cases} q_{i-1} & \text{if } j = i-1 \\ \min\limits_{i \le r \le j} \{ e[i,r-1] + e[r+1,j] + w(i,j) \} & \text{if } i < j \end{cases}$$

To record root, we use matrix root[i,j]

We store e[i,j] values in a table e[1..n+1, 0..n]

$n+1 \rightarrow d_n$

$0 \rightarrow d_0$

w[1..n+1, 0..n]

For w,

$w[i, i-1] = q_{i-1}$ for $1 \le i \le n$

For $i \le j$, we compute

$w[i,j] = w[i,j-1] + p_j + q_j$

$w[i,j] = w[i,j-1] + p_j + q_j$

$0, i$

```
OPTIMAL-BST (p, q, n)
  for i ← 1 to n+1
    do e[i, i-1] ← q_{i-1}
       w[i, i-1] ← q_{i-1}
  for l ← 1 to n
    do for i ← 1 to n-l+1
         do j ← i+l-1
            e[i, j] ← ∞
            w[i, j] ← w[i, j-1] + p_j + q_j
            for r ← i to j
              do t ← e[i, r-1] + e[r+1, j]
                     + w[i, j]
                 if t < e[i, j]
                   then e[i, j] ← t
                        root[i, j] ← r
  return e and root
```

Running time = $O(n^3)$

1st iteration $e[i, i]$ and $w[i, i]$ $l=1$

2nd " $e[i, i+1]$ and $w[i, i+1]$ $l=2$

Q:

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Scanned by CamScanner

e_j

i \ j	0	1	2	3	4	5
1	0.05	0.45	0.90	1.25	1.75	2.75
2		0.10	0.40	0.70	1.20	2.00
3			0.05	0.25	0.60	1.30	
4				0.05	0.30	0.90
5					0.05	0.50
6						0.10

$w[1,1] = w[1,0] + p_1 + q_1$

$= 0.05 + 0.15 + 0.10$

$= 0.30$

$e[1,1] = e[1,0] + e[2,1] + w[1,1]$

$1 \le r \le 1$

$r = 1 \quad = 0.05 + 0.10 + 0.30$

$= 0.45$

$w[1,2] = w[1,1] + p_2 + q_2$

$= 0.30 + 0.10 + 0.05$

$= 0.45$

$e[1,2] = \begin{cases} e[1,0] + e[2,2] + w[1,2] \\ e[1,1] + e[3,2] + w[1,2] \end{cases}$

$r = 1, 2$

$$\Rightarrow \left\{ \begin{matrix} 0.05 + 0.40 + 0.45, \\ 0.45 + 0.05 + 0.45 \end{matrix} \right\}$$

$$\Rightarrow \{0.90, 0.95\}$$

$\Rightarrow$ 0.90 for $\lambda = 1$

w

i \ j	0	1	2	3	4	5
1	0.05	0.30	0.45	0.55	0.70	1.00
2		0.10	0.25	0.35	0.50	0.80
3			0.05	0.15	0.30	0.60
4				0.05	0.20	0.50
5					0.05	0.35
6						0.10

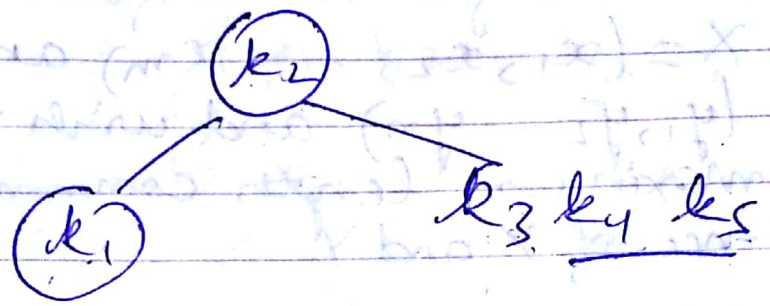
Root

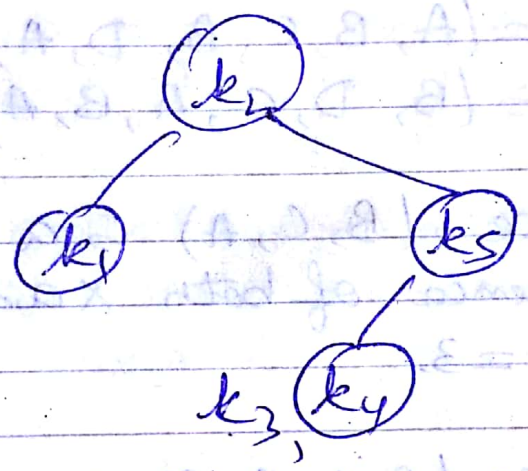
	1	2	3	4	5
1	1	1	2	2	2
2		2	2	2	4
3			3	4	5
4				4	5
5					5

Scanned by CamScanner

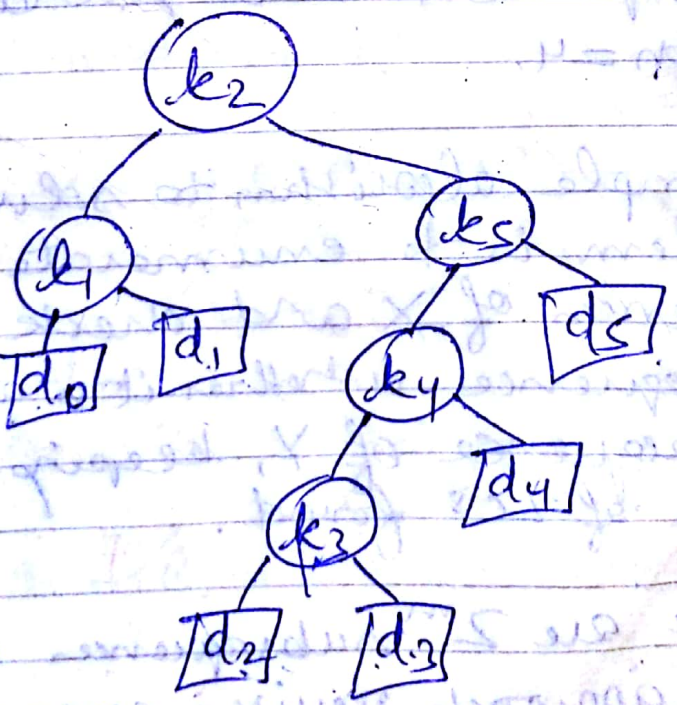
To construct tree

$k_1 \text{ — — — } k_5$ $\quad 1,5 \rightarrow 2$





$4,5 \rightarrow 5$



$3,4 \rightarrow 4$

Dynamic Programming

Longest common subsequence problem

We are given 2 sequences $X = (x_1, x_2, \ldots x_m)$ and $Y = (y_1, y_2 \ldots y_n)$ and wish to find a maximum length common subsequence of X and Y.

eg. $X = (A, B, C, B, D, A, B)$
$Y = (B, D, C, A, B, A)$

Sequence (B, C, A) is a common subsequence of both X and Y and length $= 3$.

Sequence (B, C, A, B) is a common subsequence to both X and Y and length $= 4$.

* A simple algorithm to solve LCS problem is to enumerate all sequences of X and check each subsequence whether it is also a subsequence of Y, keeping track of LCS found.

There are 2^m subsequences of X, so this approach requires exponential time.

Scanned by CamScanner

* LCS problem has an optimal substructure property.

Let $X = (x_1, x_2 ---, x_m)$
$Y = (y_1, y_2 ---, y_n)$

Let $Z = (z_1, z_2 --- z_k)$ be any LCS of X and Y.

(1) If $x_m = y_n$ then $z_k = x_m = y_n$ + z_{k-1} is an LCS of X_{m-1} + Y_{n-1}.

(2) If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.

(3) If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

eg. (1) $X = (A, B, C, B, A, C, D)$
$Y = (E, C, F, A, D)$
$Z = (C, A, D)$

Now $x_6 = D = y_5 = z_3$
$\therefore Z_2 = (C, A)$ is an LCS of (A, C, B, A, C) and (E, C, F, A)

Scanned by CamScanner

(2) Let $X = (A, C, B, A, C, D, E)$
$Y = (E, C, F, A, D)$
LCS $Z = (C, A, D)$
$x_7 = E \neq y_5$
but $z_3 \neq x_7$
$\therefore$ Z is LCS of (A, C, B, A, C, D)
and (E, C, F, A, D)
$c[i-1, j]$

(3) Let $X = (A, C, B, A, C, D)$
$Y = (E, C, A, D, F)$
$Z = (C, A, D)$
$x_6 \neq y_5$ and $z_3 \neq y_5$
$\therefore$ Z is an LCS of (A, C, B, A, C, D)
and (E, C, A, D)

Solving LCS problem using dynamic programming

We are finding LCS of $X = (x_1, x_2 \ldots x_m)$ and $Y = (y_1, y_2 \ldots y_n)$

If $x_m = y_n$, we must find LCS of X_{m-1} and Y_{n-1}. Appending $x_m = y_n$ to this LCS yields LCS of X and Y.

Scanned by CamScanner

Let $c[i,j]$ be length of an LCS of sequences $X_i = (x_1, x_2, \ldots x_i)$ and $Y_j = (y_1, y_2, \ldots y_j)$. If last elements of X and Y are same, then :-

$c[i,j] = c[i-1, j-1] + 1$

if $i=0$ or $j=0$, then $c[i,j]=0$

If $i,j>0$ and $x_i \neq y_j$, we must solve 2 subproblems:-

1. Finding an LCS of X_{m-1} and Y.
2. Finding an LCS of X and Y_{n-1}.

whichever of these 2 LCS is larger is an LCS of X and Y.

$\therefore c[i,j] = \max\{c[i,j-1], c[i-1,j]\}$

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

We also maintain the table $b[1..m, 1..n]$ to simplify the construction of an optimal solution.

<u>Algo.</u>

no. of elements in sequence X

```
LCS-length (X, Y, m, n)
```

no. of elements in sequence Y.

```
for i ← 0 to m
    do c[i, 0] ← 0
for j ← 0 to n
    do c[0, j] ← 0
for i ← 1 to m
do for j ← 1 to n
    do if xi = yj
then c[i, j] ← c[i-1, j-1] + 1
     b[i, j] ← " ↖ "
else if c[i-1, j] ≥ c[i, j-1]
  then c[i, j] ← c[i-1, j]
       b[i, j] ← " ↑ "
else c[i, j] ← c[i, j-1]
     b[i, j] ← " ← "
return c and b
```

Running time $= O(mn)$

Scanned by CamScanner

eg. Let $X = (D, A, C, B, E)$

$Y = (A, F, C, B)$

		0	1 (A)	2 (F)	3 (C)	4 (B)
	0	0	0	0	0	0
D	1	0	0↑	0↑	0↑	0↑
A	2	0	1↖	1←	1←	1←
C	3	0	1↑	1↑	2↖	2←
B	4	0	1↑	1↑	2↑	3↖
E	5	0	1↑	1↑	2↑	3↑

Length of LCS = 3

LCS = A C B

$c[1,1] = \max\{c[1,0], c[0,1]\}$
$= \max(0,0) = 0$

$c[1,2] = \max\{c[1,1], c[0,2]\}$
$= \max(0,0) = 0$

$c[2,1] = c[1,0] + 1$
$= 0 + 1 = 1$

Scanned by CamScanner

For table

* Initially when $i=0$ and $j=0$, then $c[i,j]=0$ i.e 1st row + 1st column becomes 0.

* If $x_i \neq y_j$, then check top + left element + write max^m element (and put ↑ or ←)
  By default: "↑"

* If $x_i = y_j$, (eg. $x_2 = y_1$, ie A = A
  $c[2,1] = c[1,0]+1 = 0+1 = 1$)
  ie if $x_i = y_j$, then 1 is added to diagonal element + put "↖"

* For length of LCS and to find LCS. Length of LCS is 3 (4)
  ie last entry.

* To find LCS, walk from last entry in direction of arrows + write only diagonal arrow entries in reverse order.

Scanned by CamScanner

Algo for constructing LCS:

```
Print_LCS (b, X, i, j)
  if i=0 or j=0
then return
  if b[i,j] = "↖"
then print_LCS (b, X, i-1, j-1)
Print xi
else if b[i,j] = "↑"
then Print_LCS (b, X, i-1, j)
else Print_LCS (b, X, i, j-1)
```

Running time = $O(m+n)$

Initial call is Print_LCS (b, X, m, n)

Find LCS of X = ABCBDAB
Y = BDCABA

		0	B 1	D 2	C 3	A 4	B 5	A 6
	0	0	0	0	0	0	0	0
A	1	0	0↑	0↑	0↑	1↖	1←	1↖
B	2	0	1↖	1←	1←	1↑	2↖	2←
C	3	0	1↑	1↑	2↖	2←	2↑	2↑
B	4	0	1↖	1↑	2↑	2↑	3↖	3←
D	5	0	1↑	2↖	2↑	2↑	3↑	3↑
A	6	0	1↑	2↑	2↑	3↖	3↑	4↖
B	7	0	1↖	2↑	2↑	3↑	4↖	4↑

i ↓ j →

Length = 4 LCS = BCBA

Scanned by CamScanner

Greedy algorithms

* A greedy algo always makes the choice that looks best at the moment.

* These makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

* The greedy algos do not always yield optimal solution, but for many problems they do.

Activity selection problem

* Suppose we have a set $S = \{1, 2, 3 \ldots n\}$ of n proposed activities that wish to use a resource such as a lecture hall, which can be used by only one activity at a time.

* Each activity i has a start time s_i and a finish time f_i where $s_i \leq f_i$.

* If selected, activity i takes place during half open time interval $[s_i, f_i)$

Scanned by CamScanner

* Activities i and j are compatible if the intervals $[s_i, f_i)$ and $[s_j, f_j]$ do not overlap ie if
$s_i \geq f_j$ or $s_j \geq f_i$

* Activity selection problem is to select a maximum size set of mutually compatible activities.

We assume that input activities to this problem are in sorted order of increasing finishing time :- ie
$f_1 \leq f_2 \leq f_3 \ldots \leq f_n$

```
Greedy-Activity-Selector (s, f)
n ← length [S]
A ← {1}
j ← 1
for i ← 2 to n
    do if si ≥ fj
       then A ← A ∪ {i}
            j ← i
return A
```

Set A collects selected activities
j ← specifies most recent addition to A
Initially j is 1, ie A contains 1st activity.

Scanned by CamScanner

Condition is that its start time s_i is not earlier than finish time f_j of activity most recently added to A.

Running time if not sorted by finish time = $O(n \lg n)$

Running time if sorted by finish time = $\Theta(n)$

Proving greedy algo correct

Greedy algo do not always produce optimal solution. However, Greedy-activity-selector always finds an optimal solution to an instance of activity selection problem.

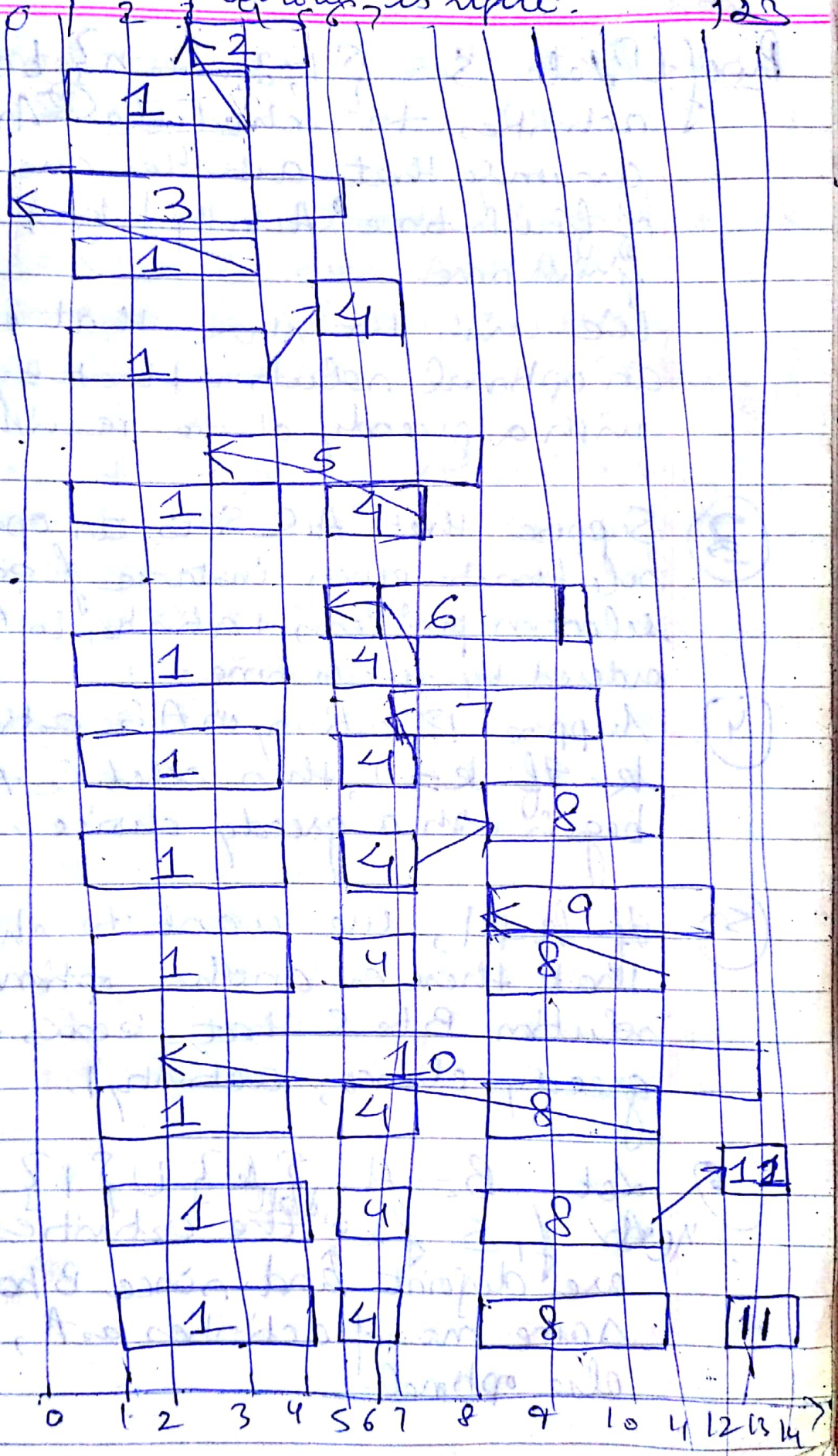
Theorem :-

Algorithm Greedy-Activity-Selector produces solution of maximum size for activity selection problem.

Scanned by CamScanner

Each row corresponds to an iteration of for loop. Activity is rejected if s_i occurs before finishing time f_j (arrows left).
Activity Accepted if arrows is right.

i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14



Proof: (1) Let $S = \{1, 2, \ldots n\}$ be set of activities to schedule (2) As we are assuming that activities are in order of finish time. Activity 1 has earlier finish time.

We wish to show that there is an optimal solution that begins with a greedy choice ie with activity 1.

(3) Suppose that $A \subseteq S$ is an optimal solution to given instance of activity selection problem + activities in A are ordered by finish time.

(4) Suppose 1st activity of A is activity k. If $k = 1$, then schedule A begins with a greedy choice.

(5) If $k \neq 1$, we want to show that there is another optimal solution B to S that begins with greedy choice, activity 1.

(6) Let $B = A - \{k\} \cup \{1\}$

$f_1 \leq f_k$ and all the activities in B are disjoint and since B has the same no. of activities as A, it is also optimal.

Scanned by CamScanner

Thus B is an optimal solution for S that contains greedy choice of activity 1.

If A is an optimal solution to original problem S, then $A' = A - \{1\}$ is an optimal solution to activity selection problem $S' = \{i \in S : s_i \geq f_1\}$

i.e, when greedy choice of activity 1 is made, the problem reduces to finding an optimal solution for activity selection problem over those activities in S that are compatible with activity 1.

---

eg. If $k \neq 1$

eg. $A = \{2, 8, 14\}$

Let $B = \{1, 8, 14\}$

$f_1 \leq f_2$

Scanned by CamScanner

For greedy strategy to be applicable problem must have:-

Elements of greedy strategy
(Two ingredients that are exhibited by most problems that lend themselves to greedy strategy are:-)

(i) Greedy choice property:- ie a globally optimal solution can be arrived at by making a locally optimal (greedy) choice.

Diff: b/w DP + greedy

(a) But in dynamic programming, we make a choice at each step + choice depends on solution to subproblems. And in greedy we make choice that looks best at the moment (it does not depend on solution to subproblems)

(b) DP is bottom up technique
Greedy is top-up strategy where one greedy choice is made after another, iteratively reducing each given problem to smaller one.

Scanned by CamScanner

common to D.P + greedy.

problem has optimal substructure

(2) Optimal substructure: if optimal sol^n to problem contains within it optimal solutions to subproblems.

eg. in activity selection problem if optimal solution A begins with activity 1, then set of activities $A' = A - \{1\}$ is an optimal solution to activity selection problem

$$S' = \{i \in S : s_i \geq f_1\}.$$

Greedy vs D.P.

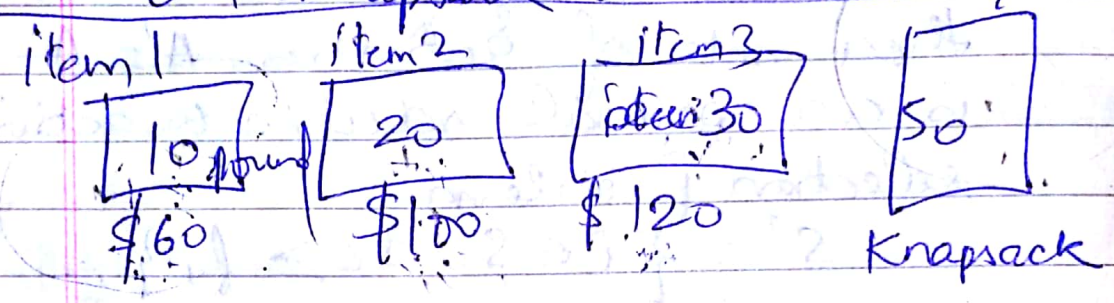
0-1 Knapsack problem:- A thief robbing a store finds n items, i^{th} item is worth v_i dollars + weighs w_i pounds where v_i and w_i are integers. He can carry atmost W pounds. Thief can take either an item or not (not fractional amount).

In fractional knapsack problem, setup is same, but thief can take fractions of items, rather than having to make binary choice for each item.

Scanned by CamScanner

Both knapsack problems exhibit optimal-substructure property. But fractional knapsack problem is solvable by greedy strategy whereas 0-1 knapsack is not.

0-1 knapsack is solvable by D.P.



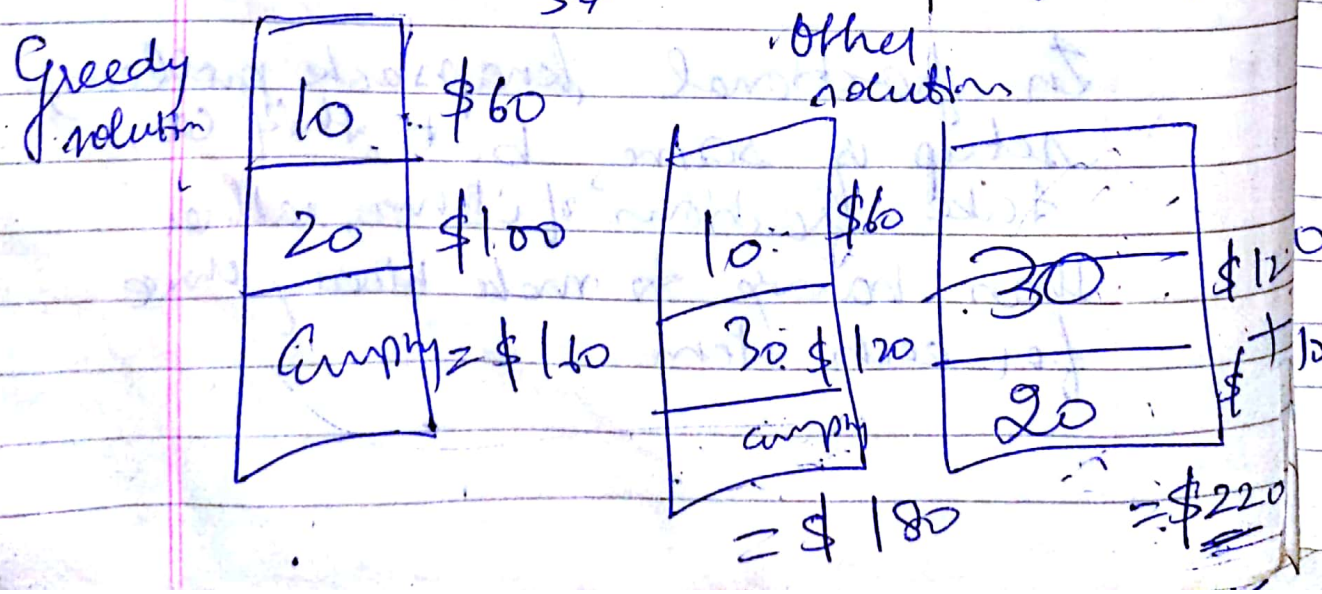
0-1 Knapsack problem

We calculate v_i / w_i per item

Item 1 = $\frac{60}{10}$ = 6 dollar/pound

Item 2 = $\frac{100}{20}$ = 5 dollar/pound

Item 3 = $\frac{120}{30}$ = 4 dollar/pound



Greedy takes item 1 and 2, but optimal takes 2 and 3 leaving 1, so 0-1 knapsack cannot be solved using greedy.

Fractional Knapsack

10	\$60
20	\$100
20/30	\$80

$(1, 1, 2/3)$

$10 + 20 + \frac{2}{3} \times 30$

$= 30 + 20$

$= 50$

$\frac{2}{3} \times \$240 = 80$

$= \$240$

0

100

Task scheduling problem

- It is problem of optimally scheduling unit time tasks on a single processor, where each task has a deadline + a penalty that must be paid if deadline is missed.

- A unit time task is a job such as program to be run on a computer, that requires exactly one unit of time to complete.

- Given a finite set S of unit time tasks a schedule for S is a permutation specifying order in which these tasks are to be performed. (1st task begins at time 0 + finishes at time 1, 2nd task at time 1 + " " " 2 + so on)

- The problem has foll^n inputs:-
  1. A set $S = \{1, 2, \ldots n\}$ of n unit time tasks
  2. A set of n integer deadlines $d_1, d_2, \ldots d_n$ s.t. $1 \le d_i \le n$ + task i is supposed to finish by time d_i
  3. A set of n nonnegative weights or penalties $w_1, w_2, \ldots w_n$ s.t. w_i is incurred if task i is not finished by time d_i + no penalty is incurred if a task finishes by its deadline

- We need to find a schedule for S that minimizes total penalty incurred for missed deadlines.

- A task is late in schedule if it finishes after its deadline, otherwise task is early.

- A schedule is put into early first form in which early tasks precede late tasks.

- An arbitrary schedule can always be put into canonical form in which early tasks precede late tasks and early tasks are scheduled in order of nondecreasing deadlines.

- A set A of tasks is independent if there exists a schedule for these tasks such that no tasks are late.

Scanned by CamScanner

Task

eg.	1	2	3	4	5	6	7
deadline d_i	4	2	4	3	1	4	6
penalty w_i	70	60	50	40	30	20	10

(1) Arrange tasks in ↓ing order of penalties and start filling deadlines

a_4	a_2	a_3	a_1		a_7	

0 1 2 3 4 5 6 7

Total penalty $w_5 + w_6$
$\Rightarrow 30 + 20 = 50$

(2) Sort tasks in order of deadlines in ↑ing order.

	a_4	a_2	a_3	a_1	a_7
d_i	3	2	4	4	6

↳ sorting

	a_2	a_4	a_3	a_1	a_7
d_i	2	3	4	4	6

Final optimal schedule is
$(a_2, a_4, a_1, a_3, a_7, a_5, a_6)$
or
$(2, 4, 1, 3, 7, 5, 6)$
Total Penalty $w_5 + w_6 = 50$

Scanned by CamScanner

Graph algorithms

Representations of graphs

There are 2 standard ways to represent a graph $G=(V,E)$:-

1. Collection of adjacency lists
2. Adjacency matrix

- Adjacency lists is preferred when graph is sparse ie when $|E|$ is much less than $|V|^2$.

- Adjacency matrix is preferred when graph is dense ie when $|E|$ is close to $|V|^2$.

- Adjacency list representation of a graph $G=(V,E)$ consists of array Adj of $|V|$ lists, one for each vertex in V. For each $u \in V$, the adjacency list Adj[u] contains all vertices v such that there is an edge $(u,v) \in E$. The vertices in each list are stored in arbitrary order.

- If G is directed graph, sum of lengths of adjacency lists is $|E|$ as edge of form (u,v) is represented by having v appear in Adj[u].

Scanned by CamScanner

- If G is undirected graph, sum of lengths of all adjacency list is $2|E|$, as (u,v) is represented by having u appear in v's adjacency list & vice versa.

- The amount of memory required is $O(V+E)$

- Adjacency lists can be used to represent weighted graphs where each edge has a weight. The weight $w(u,v)$ of edge (u,v) is stored with vertex v in u's adjacency list.

- Disadv. of adjacency list is that there is no quicker way to determine if a given edge (u,v) is present in graph, than to search for v in adjacency list $Adj[u]$.

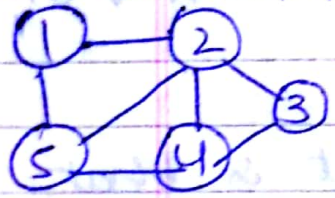
- In the Adjacency-matrix representation of graph $G=(V,E)$, we assume vertices are numbered $1, 2 \ldots |V|$ in arbitrary manner. The adjacency matrix rep. of graph G consists of $|V| \times |V|$ matrix $A = (a_{ij})$ such that

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise.} \end{cases}$$

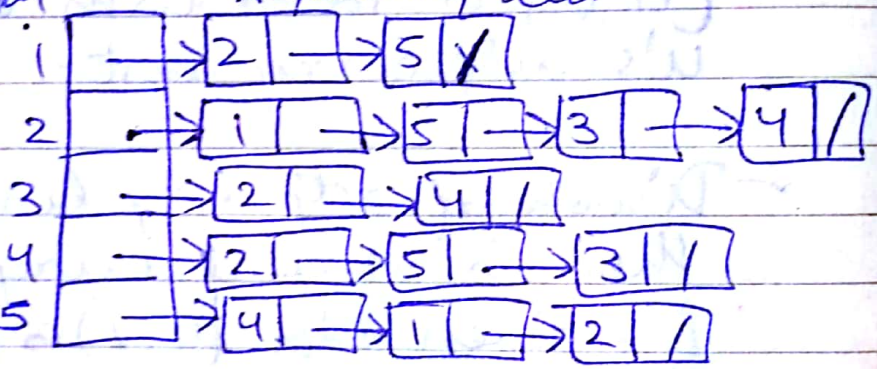
- It requires $\Theta(V^2)$ memory.

- In undirected graph, (u,v) + (v,u) are same edge, the adjacency matrix A of undirected graph is its own transpose ie $A = A^T$.
- It can also be used for weighted graph. The weight $w(u,v)$ of edge $(u,v) \in E$ is stored in row u and column v of matrix, If an edge does not exist, a NIL or 0 or ∞ value can be stored.

Undirected graph



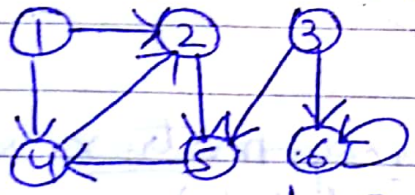
Adjacency list

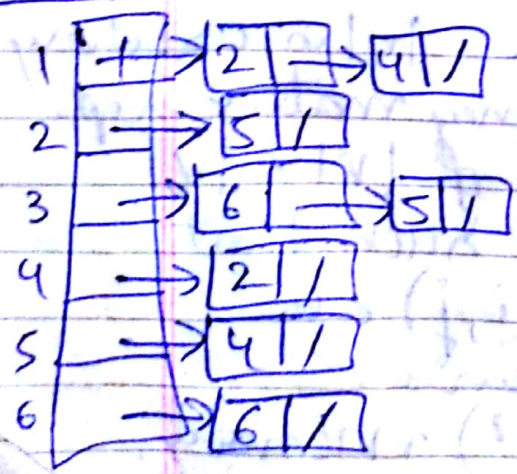


	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Adj matrix

Directed graph





	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Scanned by CamScanner

0 1 1 2 3 5 - -

132

$f(0) = 0$
$f(1) = 1$

$f[i] = f[i-1] + f[i-2]$

$$f[i] = \begin{cases} 0 & \text{if } i = 0 \\ 1 & \text{if } i = 1 \\ f[i-1] + f[i-2] & \text{for } i \geq 2 \end{cases}$$

$\frac{\log \alpha}{\log n}$ $\log_n \alpha$

$(1-\alpha)^i = \frac{1}{n}$

$i = \log_{(1-\alpha)} \frac{1}{n}$

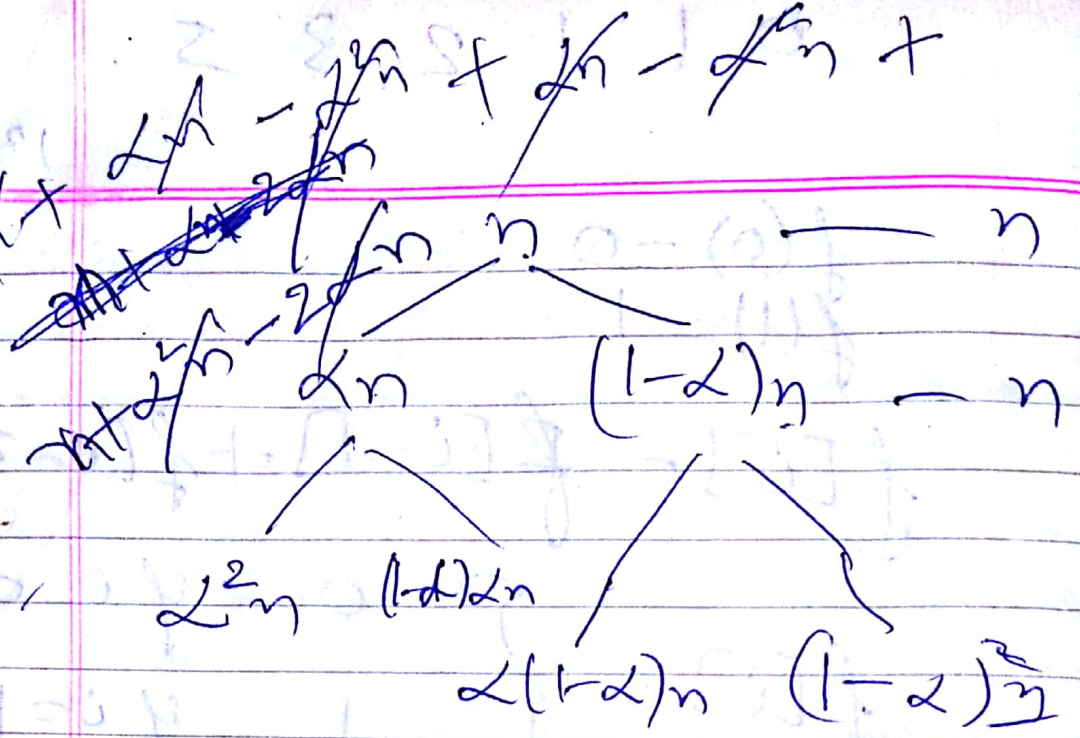
$= \frac{\log \frac{1}{n}}{\log (1-\alpha)}$

$= \frac{-\log n}{-\log \alpha} = \frac{+\log n}{+\log \alpha} = \boxed{\log_\alpha n}$

Scanned by CamScanner

$\alpha^2 n + \alpha n - \alpha^2 n + \alpha n - \alpha^2 n +$

$\alpha n + \alpha^2 n - 2\alpha n$



$\alpha n + n - \alpha n$

$n \to \alpha n \to \alpha^2 n$

α^n

$(\alpha)^i n = 1 \Rightarrow \alpha^i = n$

$n - \alpha n$

$i = \log_\alpha n \quad (1-\alpha)^i n = 1$

$0.3n$

$0.7n$

$n \log_\alpha n \quad (1-\alpha)^i = n$

$n^{\log(1-\alpha)}$

$\log_\alpha n$

$\frac{\log n}{\log \alpha} - \frac{\log n}{\log \alpha}$

$\log_{\frac{1}{n}} n$

$$T(n) = T(\underline{\alpha n}) + T((1-\alpha)n] + n$$

$n =$ $\quad n = \alpha n$

$$= T(\alpha^2 n) + T(1-\alpha)\alpha n + \alpha n$$

$$+ T(1-\alpha)n] + n$$

$$\Rightarrow T(\alpha^2 n) + T(1-\alpha)n\,(1+\alpha)$$

$n = \alpha^2 n \quad + n(1+\alpha)$

$$\Rightarrow T(\alpha^3 n) + T((1-\alpha)\alpha^2 n) + \alpha^2 n + T(1-\alpha)n(1+\alpha) + n(1+\alpha)$$

$$\Rightarrow T(\alpha^3 n) + T(1-\alpha)n\,(1+\alpha+\alpha^2) + \cdots n(1+\alpha+\alpha^2)$$

$$\alpha^i = \frac{1}{n} \quad \Rightarrow -\log_\alpha n$$

$$\cancel{\log}\ i = \log_\alpha \frac{1}{n} \quad = \frac{\log n}{\log \alpha^{-1}}$$

$$= \underline{\log_\alpha 1} - \log_\alpha n \quad = \log_{\frac{1}{\alpha}} n$$

Scanned by CamScanner

$$T(n) = 3T(n/3 + 5) + n/2$$

$$n/3 + 5 = m$$

$$n = 3(m-5)$$

$$T(n) = 3T(m) + \frac{3(m-5)}{2}$$

$$T(n) = T(\alpha n) + T(n - \alpha n) + n$$

$$x^{\alpha n} + x^{n - \alpha n} + n$$

$$T(n) = \cancel{T(\alpha n)} + \cancel{T(n)} - \cancel{T(\alpha n)} + n$$

$$= T(n) + n$$

Scanned by CamScanner

P_2 ab — b → 0 (741)

a → 1

P_3 aba — ba

2

1 3,4,5

P_4 abab — b 3,4

ab → 2 4

bab 3

2

1 5 — b 4,5

P_5 ababb — bb 5

4 — abb 4

baibb

3

P_6

Left → value dec

Rt → value greater

(5, 1, 6)

$n! = n(n-1)(n-2) \cdots 3 \cdot 2 \cdot 1$

$n! = 1 \cdot 2 \cdot 3 \cdots n$

$1 \cdot 2 \cdot 3 \cdots n \leq \underbrace{n \cdot n \cdots n}_{(n \text{ times})}$

$\leq n^n$

$< n^n$ for $(n \geq 2)$

$n! = O(n^n)$

$n! \quad 1 \cdot 2 \cdot 3 \cdots n \geq \underbrace{2 \cdot 2 \cdots 2}_{(n \text{ times})}$

$n! \geq 2^n$

$n! > 2^n$ (for $n \geq$)

$5 \times 4 \times 3 \times 2$
24
$\times 5$
120

(P)

$2 \cdot \frac{1}{2} \cdot 1 \cdot \frac{3}{2} \cdot 2 \cdots \frac{n}{2} \qquad \left(\frac{n}{2}\right)^n$

120
$\times 6$
720

$$T(n) - T(\alpha n) - T(1-\alpha)n] - n = 0$$

$$\cancel{x^n} \; x^{\alpha n} - x^{1-\alpha n} - n = 0$$

$$T(n) = T(\alpha n) + T(1-\alpha)n + n$$

$$= T(\alpha^2 n) + T(1-\alpha)(\alpha n) + \alpha n$$

$$\alpha n = m$$

$$n = m/\alpha$$

$$T(m) + T\left(\frac{m}{\alpha} - \cancel{\alpha \times \frac{m}{\alpha}}\right) + \frac{m}{\alpha}$$

$$T(m) + T\left(\frac{m - m\alpha}{\alpha}\right) + \frac{m}{\alpha}$$

Scanned by CamScanner

Subtract and Conquer Master Theorem

March 6, 2014 Leave a comment

Let's talk about Subtract and Conquer Master Theorem. Suppose we have a recurrence relation of the form

$T(n) = aT(n-b) + f(n)$ when $n>1$

$= c$ when $n<=1$

Here $f(n) = O(n^d)$

Therefore, $T(n) = aT(n-b) + O(n^d)$ when $n>1$

Now to find order of complexity for this kind of recurrence relation we can use subtract and conquer Master theorem. It states as follows:

$T(n) = O(n^d)$ if $a<1$

$= O(n^{d+1})$ if $a=1$

$= O(n^d a^{n/b})$ if $a>1$

This is very helpful for finding order of complexity for recurrence relation of the form $T(n) = aT(n-b)+f(n)$ Generally we come across Master Theorem but that is not helpful for this kind of recurrence.

Let's try some example:

Q: $T(n) = 3T(n-1)+1$ when $a>0$

$=1$ otherwise

A: Here $a=3, b=1$ and $d=0$ hence $T(n) = O(n^0 3^{n/1})$ $(a>1)$ ie $T(n)=O(3^n)$.

Q: $T(n) = 5T(n-3)+O(n^2)$ when $n>0$

$=1$ otherwise

A: Here $a=5, b=3, d=2$ hence $T(n)=O(n^2 5^{n/3})$.

Scanned by CamScanner

Remark: This theorem is written to reveal a similarity to the Master theorem. The first case is there for the sake of similarity. It doesn't occur in algorithm analysis, since if a is the number of recursive calls, $a < 1$ implies no recursive calls and no need for the theorem. The second case arises often. Insertion sort and `modexp()` are two examples at hand. The third case is not so common, but applies for instance to the iconic Towers of Hanoi problem. We give it a quick sketch.

function HanoiPuzzle(n)
Input: n = height of stack of disks to be moved from one peg to another.
Rules of puzzle: There are three pegs and a stack of n disks on one of the pegs. In the stack each disk has smaller radius than the one below it. The goal of the game is to move the stack to another peg. Each move consists in moving a disk from one peg to another. A disk may never be put on top of a smaller disk.
Output: puzzle solved - stack is moved.
if $n = 0$, return
HanoiPuzzle($n-1$) *[Move n-1 disks to another peg following rules of the game.]*
Move one disk *[Move the largest disk to the open peg (a legal move).]*
HanoiPuzzle($n-1$) *[Move n-1 disks so as to end up on top of the largest disk.]*

For T(n) being the number of moves to solver HanoiPuzzle(n), the recurrence is

$$T(n) = 2T(n-1) + 1.$$

In this case $a = 2, b = 1, d = 0$, and the theorem tells us we have 2^n disk moves necessary to solve the Towers of Hanoi puzzle.

Scanned by CamScanner

Computing Binomial Coefficients

Definition of Binomial Coefficients

- The binomial coefficient, denoted $C(n,k)$ or $\binom{n}{k}$ is the number of subsets of k elements from an n-element set.
- The binomial formula is:

  $(a+b)^n = C(n,0)\,a^n + \cdots + C(n,k)\,a^{n-k}b^k + \cdots + C(n,n)\,b^n$

- One property of binomial coefficients is:

  $C(n,k) = C(n-1,k-1) + C(n-1,k)$
  $C(n,0) = C(n,n) = 1$

- A dynamic programming approach stores the values of $C(n,k)$ as they are computed.

Table for Computing Binomials

FIGURE 8.1 [illegible] programming algorithm

Binomial Algorithm

```
algorithm Binomial(n, k)
  // Computes C(n, k) using dynamic programming
  // Input: Integers n ≥ k ≥ 0
  // Output: The value of C(n, k)
  for i ← 0 to n do
    for j ← 0 to min(i, k) do
      if j = 0 or j = i then
        C[i, j] ← 1
      else C[i, j] ← C[i − 1, j − 1] + C[i − 1, j]
  return C[n, k]
```


Comments on Binomial Algorithm

- The algorithm computes each value $C(i,j)$ once where $j \le i$, $j \le k$, and $i \le n$.
- upper triangle rectangle

  $\underbrace{1+2+\cdots+k}_{k \text{ terms}} + \underbrace{(k+1)+\cdots+(k+1)}_{n-k+1 \text{ terms}}$ ⇒ total n

  $0 \le j \le i < k$ $\quad 0 \le j \le k \le i \le n$ $\quad (n-k)+1$ (starts with 0)

- $\frac{k(k+1)}{2} + (k+1)(n-k+1) = (k+1)(n+1-k/2) \in \Theta(nk)$
- More-efficient algorithms are suggested in the exercises.

Scanned by CamScanner

$c[5, 2]$

		k		
		0	1	2
	0	1		
	1	1	1	
	2	1	2	1
n	3	1	3	3
	4	1	4	6
	5	1	5	10

$$c[2,1] = c[1,0] + c[1,1]$$
$$= 1 + 1$$

$$c[3,1] = c[2,0] + c[2,1]$$
$$= 1 + 2$$
$$= 3$$

$$c[3,2] = c[2,1] + c[2,2]$$
$$= 2 + 1$$
$$= 3$$

$$c[4,2] = c[3,1] + c[3,2]$$
$$= 3 + 3$$
$$= 6$$

$$c[5,2] = c[4,1] + c[4,2]$$
$$= 4 + 6$$
$$= 10$$

Scanned by CamScanner

CISC320 Algorithms — Recurrence Relations
Master Theorem and Muster Theorem

Big-O upper bounds on functions defined by a recurrence may be determined from a big-O bounds on their parts. Here is a key theorem, particularly useful when estimating the costs of divide and conquer algorithms.

Master Theorem (for divide and conquer recurrences):
Let T(n) be a function defined on positive n, and having the property

$$T(n) \leq \begin{cases} c, & \text{if } n \leq 1. \\ aT(n/b) + f(n), & n > 1. \end{cases}$$

for some constants $c, a > 0, b > 1, d \geq 0$, and function $f(n)$. If $f(n)$ is in $O(n^d)$, then

$$T(n) \text{ is in } \begin{cases} O(n^d), & \text{if } d > \log_b(a), \\ O(n^d \log(n)), & \text{if } d = \log_b(a), \\ O(n^{\log_b(a)}), & \text{if } d < \log_b(a). \end{cases}$$

We will discuss many applications of the Master theorem. First for the sake of comparison, here is a theorem with similar structure, useful for the analysis of some algorithms. Jokingly, call it the

Muster Theorem (for subtract and conquer recurrences):
Let $T(n)$ be a function defined on positive n, and having the property

$$T(n) \leq \begin{cases} c, & \text{if } n \leq 1. \\ aT(n-b) + f(n), & n > 1. \end{cases}$$

for some constants $c, a > 0, b > 0, d \geq 0$, and function $f(n)$. If $f(n)$ is in $O(n^d)$, then

$$T(n) \text{ is in } \begin{cases} O(n^d), & \text{if } a < 1. \\ O(n^{d+1}), & \text{if } a = 1, \\ O(n^d a^{n/b}), & \text{if } a > 1. \end{cases}$$

□

Proof: Expanding the recursion by one step, we have

$$\begin{aligned} T(n) &\leq a(\ T(n-2b) + f(n-b)\ ) + f(n) \\ &= a^2 T(n-2b) + af(n-b) + f(n). \end{aligned}$$

Taking another step, we get

$$T(n) \leq a^3 T(n-3b) + a^2 f(n-2b) + af(n-b) + f(n).$$

etc. This leads to $T(n) \leq \sum_{i=0}^{n/b} a^i f(n-ib)$ plus a constant. Since $f(n-ib)$ is in $O(n-ib)$ which is in $O(n)$, we have that $T(n)$ is in $O(n^d \sum_{i=0}^{n/b} a^i)$. The job is finished by this property of the sum:

$$\sum_{i=0}^{n/b} a^i \text{ is in } \begin{cases} O(1), & \text{if } a < 1. \\ O(n), & \text{if } a = 1. \\ O(a^{n/b}), & \text{if } a > 1. \end{cases}$$

Scanned by CamScanner

Floyd-Warshall algorithm

- It uses dynamic programming to solve all-pairs shortest path problem on a directed graph $G=(V,E)$. We assume that there are no negative weight cycles.

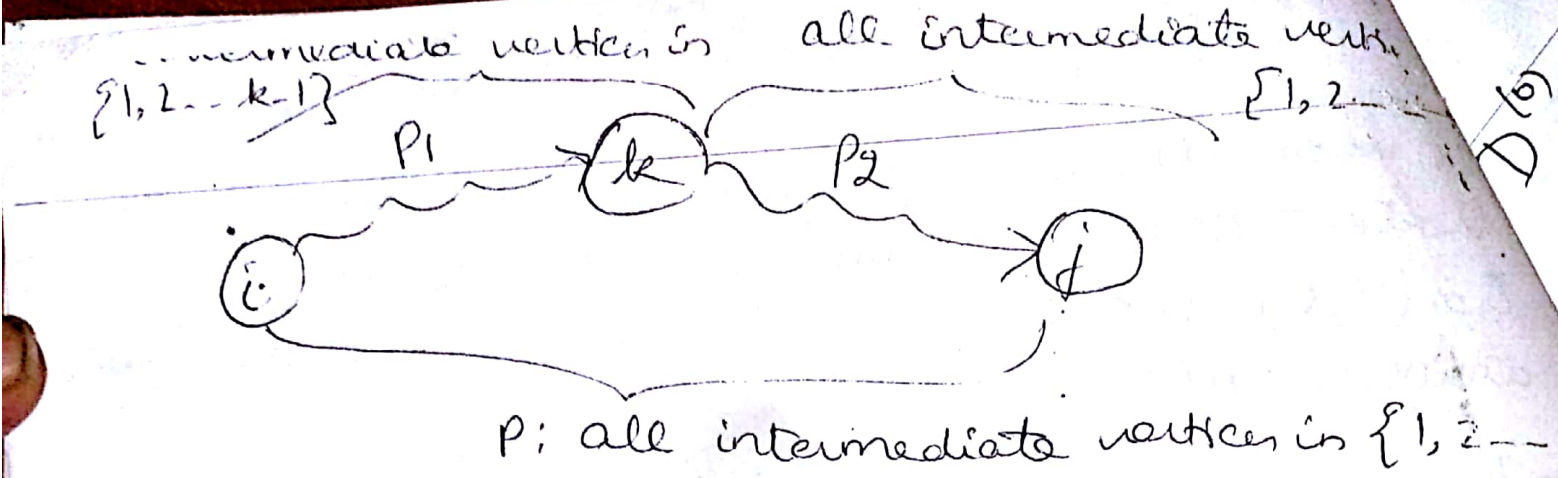
- This algo considers the intermediate vertices of a ~~shortest~~ simple path $p = (v_1, v_2 \ldots v_l)$ where intermediate vertex is any vertex other than v_1 or v_l.

- Let vertices of G be $V = \{1, 2 \ldots n\}$ and consider a subset $\{1, 2 \ldots k\}$ of vertices for some k.
- For any pair of vertices $i, j \in V$, consider all paths from i to j whose intermediate vertices are drawn from $\{1, 2 \ldots k\}$ and let p be a minimum weight path amongst them.

- This algo exploits a relationship between paths p and shortest paths from i to j with all intermediate vertices in the set $\{1, 2 \ldots k-1\}$. This relationship depends on whether or not k is an intermediate vertex of path p.

* If k is not an intermediate vertex of path p, then all intermediate vertices of path p are in set $\{1, 2 \ldots k-1\}$
* If k is an intermediate vertex of path p, then we break p down into $i \xrightarrow{p_1} k \xrightarrow{p_2} j$.
  - p_1 is shortest path from i to k with all intermediate vertices in the set $\{1, 2, \ldots k-1\}$
  - p_2 is shortest path from vertex k to vertex j with all intermediate vertices in set $\{1, 2 \ldots k\}$

Scanned by CamScanner



- Let $d_{ij}^{(k)}$ be weight of shortest path from vertex i to j with all intermediate vertices in set $\{1, 2 \ldots k\}$

When $k=0$, it has atmost one edge,

$\therefore \quad d_{ij}^{(0)} = w_{ij}$

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k=0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 \end{cases}$$

Floyd-Warshall (W)

1) $n \leftarrow rows[W]$
2) $D^{(0)} \leftarrow W$
3) for $k \leftarrow 1$ to n
4) do for $i \leftarrow 1$ to n
5) do for $j \leftarrow 1$ to n
6) $d_{ij}^{(k)} \leftarrow \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
7) return $D^{(n)}$

Running time $= \Theta(n^3)$

Scanned by CamScanner

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \pi^{(0)} = \begin{pmatrix} NIL & 1 & 1 & N & \\ NIL & NIL & NIL & 2 & \\ NIL & 3 & NIL & NIL & \\ 4 & NIL & 4 & NIL & N \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \pi^{(1)} = \begin{pmatrix} NIL & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & NIL & NIL \\ 4 & 1 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \pi^{(2)} = \begin{pmatrix} NIL & 1 & 1 & 2 & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 2 & 2 \\ 4 & 1 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \pi^{(3)} = \begin{pmatrix} NIL & 1 & 1 & 2 & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 2 & 2 \\ 4 & 3 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \pi^{(4)} = \begin{pmatrix} NIL & 1 & 4 & 2 & 1 \\ 4 & NIL & 4 & 2 & 1 \\ 4 & 3 & NIL & 2 & 1 \\ 4 & 3 & 4 & NIL & 1 \\ 4 & 3 & 4 & 5 & NIL \end{pmatrix}$$

Scanned by CamScanner

...ructing a shortest path

- can compute the predecessor matrix π.

$\pi_{ij}^{(k)}$ is defined as predecessor of vertex j on a shortest path from vertex i with intermediate vertices in set $\{1, 2 \ldots k\}$

- when $k = 0$:-

$$\pi_{ij}^{(0)} = \begin{cases} NIL & \text{if } i = j \text{ or } w_{ij} = \infty \\ i & \text{if } i \neq j \text{ and } w_{ij} < \infty \end{cases}$$

- when $k \geq 1$,

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

Scanned by CamScanner

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} NIL & 3 & 4 & 5 & 1 \\ 4 & NIL & 4 & 2 & 1 \\ 4 & 3 & NIL & 2 & 1 \\ 4 & 3 & 4 & NIL & 1 \\ 4 & 3 & 4 & 5 & NIL \end{pmatrix}$$

Scanned by CamScanner

4.6.5 0/1 Knapsack Problem

We have N items $i_1, i_2, i_3, \ldots, i_N$ and each item has the corresponding weight and profit value $w_1, w_2, w_3, \ldots, w_N$ and $p_1, p_2, p_3, \ldots, p_N$, respectively. We have the knapsack with the capacity of M, that is, we can carry maximum weight of W in the knapsack. The objective of 0/1 knapsack problem is to find the most valuable subset of the items that fit into the knapsack, so we have to fill the knapsack with the items in a way that maximizes the profit value of the included object, while respecting the capacity constraints. The 0/1 knapsack problem is different from the fractional knapsack problem in a way that we cannot take a fraction of an item into 0/1 knapsack problem, that is, the objects cannot be broken into smaller pieces. Therefore, we may decide either to take an object or leave it. We either have to take the whole item (1) or none (0) of it that is why it is called 0/1 knapsack problem. Now the 0/1 knapsack problem is formally described as follows:

> "The 0/1 knapsack problem is to find the subsets of items from the set of N items such that $\sum_{1 \le i \le n} p_i x_i$ is maximum subjected to knapsack capacity constraints. $\sum_{1 \le i \le n} w_i x_i <= W$ where p_i and w_i are profit and weight values, respectively, corresponding to the ith item. W is the capacity of the knapsack and x_i is either 0 or 1 corresponding to the inclusion of the item or exclusion of the item. So if $x_i = 1$, it means that ith item is included in the subset, and if $x_i = 0$, it means that ith item is not included in the subset."

The knapsack problem can also be described in an equation form. We have a set S of N items with weight "w" and profit value "p" are associated with each item and we can carry the maximum weight of "W" in our knapsack. The objective of 0/1 knapsack problem is to maximize the total value of items in the knapsack, that is,

$$p_1x_1 + p_2x_2 + \ldots + p_nx_n$$

subject to the knapsack capacity constraints W, that is,

$$w_1x_1 + w_2x_2 + \ldots + x_nw_n \le W$$

1. **Characterize optimal substructure (optimal substructure of 0/1 knapsack problem):** We know that dynamic programming deals with the problem whose optimal solution can be obtained as the result of a sequence of decisions. The sequence of decisions may maximize/minimize the objective function. 0/1 knapsack problem can also be solved using dynamic programming technique because its optimal solution can be obtained in a sequence of decisions, that is, the sequence of selection of items. The decision for 0/1 knapsack can be taken as follows:
   - The x_i values may be taken in the $x_1, x_2, x_3, \ldots, x_N$ order. In this case, the initial state of the 0/1 knapsack problem may be defined as Knapsack (1, N, W) because items 1 to N are available with their corresponding weights and profits and the available knapsack capacity is W. After taking the first decision at the initial state there are two possibilities:
     (a) The next state becomes Knapsack (2, N, W) when the item selected in the initial state is not taken into the knapsack, that is, decision is to set $x_1 = 0$. So items $x_2, \ldots, x_N$ are available now for inclusion and the remaining capacity of the knapsack remains the same, that is, M since item x_1 is not included.
     (b) The next state becomes Knapsack (2, N, $W-w_1$) when the item selected in the initial state is taken into the knapsack, that is, decision is to set $x_1 = 1$. So items $x_2, \ldots, x_N$ are available now for inclusion and the remaining capacity of the knapsack is $M - w_1$ since item x_1 is included.

Scanned by CamScanner

- The x_i values may be taken in the $x_N, x_{n-1}, x_{n-2}, \ldots, x_1$ order. In this case, the initial state of the 0/1 knapsack problem may be defined as Knapsack (k, N, W) where $K = N$ because items N through 1 are available with their corresponding weights and profits and the available knapsack capacity is W. After taking the first decision at the initial state and there are following two possibilities:
  (a) The next state becomes Knapsack $(K-1, N, W)$ when the item selected in the initial state is not taken into the knapsack, that is, decision is to set $x_N = 0$. So items $x_{N-1}, \ldots, x_1$ are available now for inclusion and the remaining capacity of the knapsack remains M, since item x_N is not included.
  (b) The next state becomes Knapsack $(K-1, N, W-w_N)$ when the $(N-1)$th item (since $K = N$) selected in the initial state is taken into the knapsack, that is, decision is to set $x_N = 1$. So items $x_{N-1}, \ldots, x_1$ are available now for inclusion and the remaining capacity of the knapsack remains $M - w_N$, since item x_N is included.
- Any other order: The two possibilities for this case may also defined as explained in earlier two cases.

We may obtain the optimal solution to the 0/1 knapsack problem if each decision produces the optimal solution. So if first decision is taken about the item x_1 then the remaining decisions about the variables $x_2, x_3, \ldots, x_N$ must also be optimal in order to get the final solution of the 0/1 knapsack optimal. Let us consider the knapsack with maximum capacity W and a set S consisting of N items with each ith item having weight of w_i and profit of p_i. After taking the decision, the original problem is divided into two subproblems according to the inclusion/exclusion of the selected item. Let us say at some state, the subproblem consists of the set of items $\{1, \ldots, i\}$ where $i \le N$, and the weight of the knapsack is w such that $0 \le w \le W$. Let $B[i][w]$ be the value of optimal solution to the knapsack $(1, i, w)$, that is, the benefit achieved for the subproblem having items $\{1 \ldots i\}$ and total weight w or in other words the optimal solution for the first i items, involving up to w weights. Our goal is to find the optimal solution for the original problem Knapsack $(1, N, W)$, that is, $B[N][W]$.

2. **Use recursive solution:** To design a dynamic programming algorithm for 0/1 knapsack problem, we first need to define the value of an optimal solution by expressing it in terms of optimal solutions for smaller problems. This is done by using the recurrence relation that expresses a solution to an instance in terms of solutions to its smaller instances of the knapsack problem. Let $B[k][w]$ denote the value obtained by filling a knapsack of size "w" with items taken from the first "k" items $\{k_1, k_2, \ldots, k_i\}$ in an optimal way. The range of k is from 0 to N and the range of w is from 0 to W and we obtain the value of $B[N, W]$. The recursive formula for $B[k, W]$ is given here as follows:

$$B[k][w] = \begin{cases} 0, & \text{if } k = 0 \text{ or } w = 0 \\ B[k-1][w], & \text{if } w < w_k \text{ and } k \ne 0, w \ne 0 \\ \max\{p_k + B[k-1][w-w_k], B[k-1][w]\}, & \text{if } w >= w_k \text{ and } k \ne 0, w \ne 0 \end{cases}$$

This recursive formula is defined corresponding to the following three different possibilities:

- When $k = 0$ means that there is no item to be selected for any capacity of knapsack; therefore, $B[k][w] = 0$, that is, $B[0][w] = 0$; so this case corresponds to the situation when there are no items to choose from.
- When $w = 0$ means that the capacity of the knapsack is zero so it cannot hold any item; therefore, $B[k][w] = 0$, that is, $B[k][0] = 0$; so this case corresponds to the situation when there is no weight allowed.
- When $k \ne 0$ and $w \ne 0$, that is, there are some items to select from and knapsack has the capacity to accommodate the selected items, then there are the following two possibilities:
  (a) **First case:** $w_k > w$. Item "k" cannot be part of the solution, since if it was, the total weight would be $w_k > w$, which is unacceptable

$$B[k-1][w] \quad \text{if } k \ne 0 \; w \ne 0 \text{ and } w < w_k$$

Scanned by CamScanner

So this case corresponds to the maximum value obtained by filling a knapsack of size w with items taken from $\{i_1, i_2, \ldots, i_{k-1}\}$ only in an optimal way.

(b) **Second case:** $w_k \leq w$, then the item "k" can be in the solution, and we choose the case with greater value.

$$\text{Max}\{B[k-1][w], B[k-1][w-w_k]+p_k\} \quad \text{if } k \neq 0 \; w \neq 0 \text{ and } w \geq w_k$$

This case corresponds to the maximum value obtained by filling a knapsack of size $w - w_k$ with items taken from $\{i_1, i_2, \ldots, i_{k-1}\}$ in an optimal way plus the value of item i_k, that is, p_k.

These cases are shown in Fig. 5.

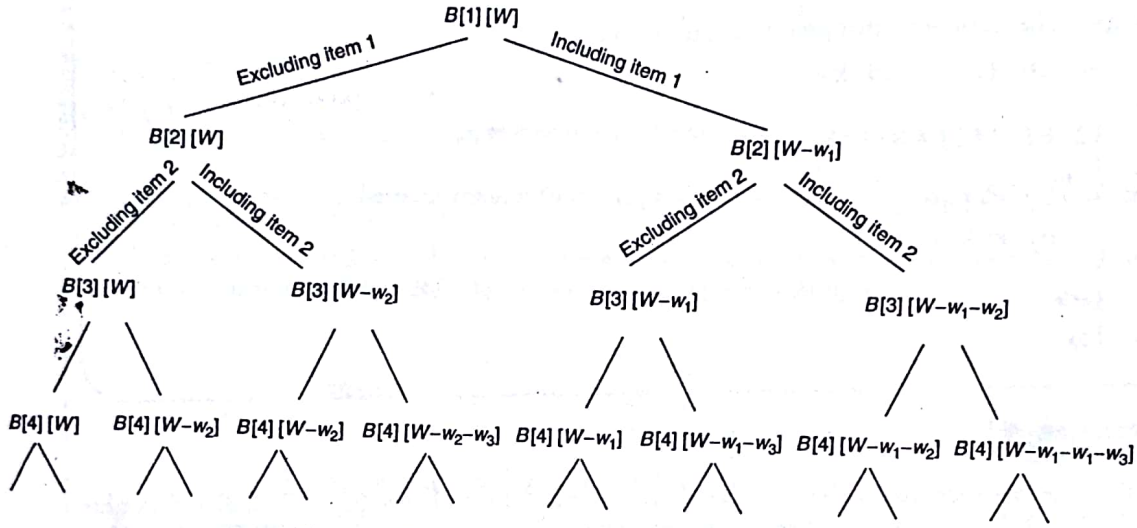


Figure 5 Tree structure for 0/1 knapsack problem.

3. **Compute the optimal value:** To compute the value of an optimal solution, a table of B[(number of item + 1)][(capacity of knapsack + 1)] is constructed using the recursive formula described in the previous step. Therefore, we use the table $B[0 \ldots N][0 \ldots W]$ for $1 \leq i \leq N$ and $0 \leq w \leq W$ and the value of $B[i][j]$ is evaluated in a bottom-up fashion. The entry $B[k][w]$ indicates the optimal (total) profit after including the items from the subsets of items $\{i_1, i_2, i_3, \ldots, i_k\}$ with the total weights at the most w. Algorithm 9 shows filling the entries of the table.
Now we can see that the filling an entry in a row requires two entries from the previous row:

Algorithm 9 `Knapsack (W[1 ... N], P[1 ... N], B[0 ... N][0 ... W])`

Input: W: It is an array that contains the weights of all items.
P: It is an array that contains the values of all items.
B: It is an optimal cost matrix that is used to store the results from the dynamic programming algorithm.
Output: The last value of array `B[N][W]` contains the optimal solution of the knapsack problem for the given capacity W.

Scanned by CamScanner

The table is filled in row major order. The table does not tell which subsets give the optimal solution that is given in next step.

4. **Construct the optimal solution:** We know that the optimal value is placed at $B[N][W]$, so we start from there and check whether $B[k]$ and $B[i-1][k]$ are equal or not. In case they are not equal, then we mark that ith item is to be included into the knapsack.

Algorithm 10 KNAPSACKPATH (B i k)

Input: The optimal cost matrix B obtained from the previous step and the initial value of $i = N$ and $k = W$.

Output: The set of items that produces optimal cost.

```
1. While (i>0 and k>0)
2. {
3. If B[i][j]≠B[i-1][j]
4. {
      a. Print i
      b. k=k-wi
5. }
6. i=i-1
7. }
```

Problem 8

Consider the knapsack instance with 5 objects and a capacity $M = 11$, $P = (5, 4, 7, 2, 3)$ and weights $W = (4, 3, 6, 2, 2,)$. Solve it using dynamic programming. **(RGPV, Dec 2014)**

Solution:

Step 1: After the initialization step of the first row and the first column, the matrix B will be completed as follows:

$B[i][0] = 0$ for $1 \le i \le N$

$B[0][w] = 0$ for $1 \le w \le W$

B[i][w]	Weights 0	1	2	3	4	5	6	7	8	9	10	11	Allowed items
ITEM 0	0	0	0	0	0	0	0	0	0	0	0	0	{ }
1	0												
2	0												
3	0												
4	0												
5	0												

Scanned by CamScanner

Step 2: Start filling the entries for row 1. When $i = 1$ and $w = 1$

$$p_1 = 5,\ w_1 = 4,\ w = 1,\ w - w_1 = 1 - 4 = -3$$

Negative value of $w - w_i$ says that $w_i > w$, so ith item cannot be the part of knapsack. So just copy the value of the same column from the previous row as given in algorithm 10.

$$B[i][w] = B[i-1][w], \text{ that is, } B[1][1] = B[0][1] = 0$$

When $i = 1$ and $w = 2$

$$p_i = 5,\ w_i = 4,\ w = 2,\ w - w_i = 2 - 4 = -2$$

So just copy the value of the same column from the previous row, that is, $B[i][w] = B[i-1][w]$, that is, $B[1][2] = B[0][2] = 0$.

When $i = 1$ and $w = 3$, $w_1 = 4$, $w - w_1 = 3 - 4 = -1$, so copy the value of the same column from the previous row, that is, $B[i][w] = B[i-1][w]$, which means $B[1][3] = B[0][3] = 0$.

Weights →

B[I][W]	0	1	2	3	4	5	6	7	8	9	10	11	Allowed items
ITEM 0	0	0	0	0	0	0	0	0	0	0	0	0	{ }
1	0	0	0	0									{1}
2	0												
3	0												
4	:0												
5	0												

When $i = 1$ and $w = 4$, $p_1 = 5$, $w_1 = 4$, $w - w_1 = 4 - 4 = 0$, the value is calculated as

$B[i][w] = \text{Max}\{p_i + B[i-1][w - w_i], B[i-1][w]\}$

$B[1][4] = \text{Max}\{p_1 + B[1-1][w - w_1], B[1-1][w]\}$

$B[1][4] = \text{Max}\{5 + B[0][0], B[0][4]\}$

$B[1][4] = \text{Max}\{5 + 0, 0\}$

$B[1][4] = 5$

Scanned by CamScanner

This is shown in the following diagram.

B[I] [W] (Weights →)	0	1	2	3	4	5	6	7	8	9	10	11
ITEM 0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0								
2	0											
3	0											
4	:0	:	:	:	:	:	:					
5	0											

B[I] [W] Items (Weights →)	0	1	2	3	4	5	6	7	8	9	10	11	Allowed items
ITEM 0	0	0	0	0	0	0	0	0	0	0	0	0	{ }
1	0	0	0	0	5								{1}
2	0												
3	0												
4	:0	:	:	:	:	:	:						
5	0												

When $i = 1$ and $w = 5$, $p_1 = 5$, $w_1 = 4$, $w - w_1 = 5 - 4 = 1$, the value is calculated as

$B[i][w] = \text{Max}\{p_i + B[i-1][w-w_i], B[i-1][w]\}$

$B[1][5] = \text{Max}\{p_1 + B[1-1][5-w_1], B[1-1][5]\}$

$B[1][5] = \text{Max}\{5 + B[0][5-4], B[0][5]\}$

$B[1][5] = \text{Max}\{5 + B[0][1], B[0][5]\}$

$B[1][5] = \text{Max}\{5 + 0, 0\}$

$B[1][5] = 5$

Similarly, we can find the values for $B[1][6] = B[1][7] = B[1][8] = B[1][9] = B[1][10] = B[1][11] = 5$ as shown in the following diagram:

B[I] [W] (Weights →)	0	1	2	3	4	5	6	7	8	9	10	11	Allowed Items
ITEM 0	0	0	0	0	0	0	0	0	0	0	0	0	{ }
1	0	0	0	0	5	5	5	5	5	5	5	5	{1}
2	0												
3	0												
4	:0												
5	0												

Scanned by CamScanner

Step 3: When $i = 2$ and $w = 1$, we have $p_2 = 4$, $w_2 = 3$, $w - w_2 = 1 - 3 = -2$. Negative value of $w - w_i$ says that $w_i > w$, so ith item cannot be the part of knapsack. So just copy the value of the same column from the previous row as given in the algorithm.

$$B[i][w] = B[i-1][w], \text{ that is, } B[2][1] = B[1][1] = 0$$

Similarly, $B[2][2] = B[1][1] = 0$.

B[I][W] (ITEM)	Weights 0	1	2	3	4	5	6	7	8	9	10	11	Allowed Items
0	0	0	0	0	0	0	0	0	0	0	0	0	{ }
1	0	0	0	0	5	5	5	5	5	5	5	5	{1}
2	0	0	0										{1, 2}
3	0												
4	:0												
5	0												

When $i = 2$ and $w = 3$, we have $p_2 = 4$, $w_2 = 3$, $w - w_2 = 3 - 3 = 0$. So the values are calculated as

$B[i][w] = \text{Max}\{p_i + B[i-1][w - w_i], B[i-1][w]\}$
$B[2][3] = \text{Max}\{p_2 + B[2-1][3 - w_2], B[2-1][3]\}$
$B[2][3] = \text{Max}\{4 + B[1][3-3], B[1][3]\}$
$B[2][3] = \text{Max}\{4 + B[1][0], B[1][3]\}$
$B[2][3] = \text{Max}\{4 + 0, 0\}$
$B[2][3] = 4$

When $i = 2$ and $w = 4$, we have $p_2 = 4$, $w_2 = 3$, $w - w_2 = 4 - 3 = 1$. So the values are calculated as

$B[i][w] = \text{Max}\{p_i + B[i-1][w - w_i], B[i-1][w]\}$
$B[2][4] = \text{Max}\{p_2 + B[2-1][4 - w_2], B[2-1][4]\}$
$B[2][4] = \text{Max}\{4 + B[1][4-3], B[1][4]\}$
$B[2][4] = \text{Max}\{4 + B[1][1], B[1][4]\}$
$B[2][4] = \text{Max}\{4 + 0, 5\}$
$B[2][4] = 5$

When $i = 2$ and $w = 5$, we have $p_2 = 4$, $w_2 = 3$, $w - w_2 = 5 - 3 = 2$. So the values are calculated as

$B[i][w] = \text{Max}\{p_i + B[i-1][w - w_i], B[i-1][w]\}$
$B[2][5] = \text{Max}\{4 + B[1][w - w_1], B[1][5]\}$
$B[2][5] = \text{Max}\{4 + B[1][5-3], B[1][5]\}$
$B[2][5] = \text{Max}\{4 + B[1][2], B[1][5]\}$
$B[2][5] = \text{Max}\{4 + 0, 5\} = 5$

Scanned by CamScanner

Similarly other values for row 2 are calculated as follows

$A[2][6] = \text{Max}[6 \times A[1][6 - 5],\ A[1][6]] = \text{Max}[6 \times A[1][1],\ A[1][6]] = \text{Max}[6 \times 0,\ 5] = 5$

$A[2][7] = \text{Max}[6 \times A[1][7 - 5],\ A[1][7]] = \text{Max}[6 \times A[1][2],\ A[1][7]] = \text{Max}[6 \times 5,\ 5] = 5$

$A[2][8] = \text{Max}[6 \times A[1][8 - 5],\ A[1][8]] = \text{Max}[6 \times A[1][3],\ A[1][8]] = \text{Max}[6 \times 5,\ 5] = 5$

$A[2][9] = 5,\ A[2][10] = 5,\ A[2][11] = 5$

	Weights w												Allowed items
$A[i][w]$	0	1	2	3	4	5	6	7	8	9	10	11	
$i = 0$	0	0	0	0	0	0	0	0	0	0	0	0	{}
$i = 1$	0	0	0	0	5	5	5	5	5	5	5	5	{1}
$i = 2$	0	0	0	0	5	5	5	5	5	5	5	5	{1, 2}
$i = 3$	0												
$i = 4$	0												
$i = 5$	0												

Step 6: When $i = 3$, $w_i = 6$ and $p_i = 7$. Since $w_1 > 1$, $w_2 > 2$, $w_3 > 3$, $w_4 > 4$, $w_5 > 5$, so $w - w_i$, $w + \delta_i$, $w - w_1$, $w - w_2$, $w - w_3$ will result in values less than 0. So

$A[3][1] = A[2][1] = 0$

$A[3][2] = A[2][2] = 0$

$A[3][3] = A[2][3] = 0$

$A[3][4] = A[2][4] = 5$

$A[3][5] = A[2][5] = 5$

P = 5 4 7 2 3

W = 4 3 6 2 2

7 − 9

	Weights w												Allowed items
$A[i][w]$	0	1	2	3	4	5	6	7	8	9	10	11	
$i = 0$	0	0	0	0	0	0	0	0	0	0	0	0	{}
$i = 1$	0	0	0	0	0	0	0	0	0	0	0	0	{1}
$i = 2$	0	0	0	0	0	0	0	0	0	0	0	0	{1, 2}
$i = 3$	0	0	0	0	0	0	7	0	0	11	12	12	{1, 2, 3}
$i = 4$	0	0	2	4	5	6	7	9	9	11	12	13	{
$i = 5$	0	0	3	4	5	7	8	9	10	12	14	14	1, 2,

Scanned by CamScanner

Similarly other values for row 2 are calculated as follows:

$B[2][6] = \text{Max}\{4 + B[1][6-3], B[1][6]\} = \text{Max}\{4 + B[1][3], B[1][6]\} = \text{Max}\{4 + 0, 5\} = 5$

$B[2][7] = \text{Max}\{4 + B[1][7-3], B[1][7]\} = \text{Max}\{4 + B[1][4], B[1][7]\} = \text{Max}\{4 + 5, 5\} = 9$

$B[2][8] = \text{Max}\{4 + B[1][8-3], B[1][8]\} = \text{Max}\{4 + B[1][5], B[1][8]\} = \text{Max}\{4 + 5, 5\} = 9$

$B[2][9] = 9$; $B[2][10] = 9$; $B[2][11] = 9$

B[i][w] (ITEM)	Weights 0	1	2	3	4	5	6	7	8	9	10	11	Allowed items
0	0	0	0	0	0	0	0	0	0	0	0	0	{ }
1	0	0	0	0	5	5	5	5	5	5	5	5	{1}
2	0	0	0	4	5	5	5	9	9	9	9	9	{1, 2}
3	0												
4	:0												
5	0												

Step 4: When $i = 3$, $w_3 = 6$ and $p_3 = 7$. Since $w_3 > 1$, $w_3 > 2$, $w_3 > 3$, $w_3 > 4$, $w_3 > 5$, so $w - w_1$, $w - w_2$, $w - w_3$, $w - w_4$, $w - w_5$ will result in values less than 0. So

$B[3][1] = B[2][1] = 0$

$B[3][2] = B[2][2] = 0$

$B[3][3] = B[2][3] = 4$

$B[3][4] = B[2][4] = 5$

$B[3][5] = B[2][5] = 9$

P= 5 4 7 2 3

W= 4 3 6 2 2

7−9

10 → 12 5

B[i][w] (ITEM)	Weights 0	1	2	3	4	5	6	7	8	9	10	11	Allowed items
0	0	0	0	0	0	0	0	0	0	0	0	0	{ }
1	0	0	0	0	5	5	5	5	5	5	5	5	{1}
2	0	0	0	4	5	5	5	9	9	9	9	9	{1, 2}
3	0	0	0	4	5	5	7	9	9	11	12	12	{1, 2, 3}
4	:0	0	2	4	5	6	7	9	9	11	12	13	{
5	0	0	3	4	5	7	8	9	10	12	12	14	{1, 2,

Scanned by CamScanner

When $i = 3$, $w_3 = 6$, $p_3 = 7$ and $w = 6$, then $w - w_3 = 6 - 6 = 0$. So

$B[3][6] = \text{Max}\{7 + B[i-1][w - w_i], B[i-1][w]\}$

$B[3][6] = \text{Max}\{7 + B[2][0], B[2][6]\} = 7$

When $i = 3$, $w_3 = 6$, $p_3 = 7$ and $w = 7$, then $w - w_3 = 7 - 6 = 1$. So

$B[3][7] = \text{Max}\{7 + B[2][1], B[2][7]\} = 9$

When $i = 3$, $w_3 = 6$, $p_3 = 7$ and $w = 8$, then $w - w_3 = 8 - 6 = 2$. So

$B[3][8] = \text{Max}\{7 + B[2][2], B[2][8]\} = 9$

When $i = 3$, $w_3 = 6$, $p_3 = 7$ and $w = 9$, then $w - w_3 = 9 - 6 = 3$. So

$B[3][9] = \text{Max}\{7 + B[2][3], B[2][9]\} = 11$

When $i = 3$, $w_3 = 6$, $p_3 = 7$ and $w = 10$, then $w - w_3 = 10 - 6 = 4$. So

$B[3][10] = \text{Max}\{7 + B[2][4], B[2][10]\} = 12$

When $i = 3$, $w_3 = 6$, $p_3 = 7$ and $w = 11$, then $w - w_3 = 11 - 6 = 5$. So

$B[3][11] = \text{Max}\{7 + B[2][5], B[2][11]\} = 12$

Step 5: When $i = 4$, $w_4 = 2$ and $p_4 = 2$. Since $w_4 > 1$ so $w - w_4 = 1 - 2 = -1$. So

$B[4][1] = B[3][1] = 0$

When $i = 4$, $w_4 = 2$, $p_4 = 2$ and $w = 2$, then $w - w_4 = 2 - 2 = 0$. So

$B[4][2] = \text{Max}\{2 + B[i-1][w - w_i], B[i-1][w]\} = \text{Max}\{2 + B[3][0], B[3][2]\} = 2$

When $i = 4$, $w_4 = 2$, $p_4 = 2$ and $w = 3$, then $w - w_4 = 3 - 2 = 1$. So

$B[4][3] = \text{Max}\{2 + B[i-1][w - w_i], B[i-1][w]\} = \text{Max}\{2 + B[3][1], B[3][3]\} = 4$

When $i = 4$, $w_4 = 2$, $p_4 = 2$ and $w = 4$, then $w - w_4 = 4 - 2 = 2$. So

$B[4][4] = \text{Max}\{2 + B[i-1][w - w_i], B[i-1][w]\} = \text{Max}\{2 + B[3][2], B[3][4]\} = 5$

When $i = 4$, $w_4 = 2$, $p_4 = 2$ and $w = 5$, then $w - w_4 = 5 - 2 = 3$. So

$B[4][5] = \text{Max}\{2 + B[i-1][w - w_i], B[i-1][w]\} = \text{Max}\{2 + B[3][3], B[3][5]\} = 6$

When $i = 4$, $w_4 = 2$, $p_4 = 2$ and $w = 6$, then $w - w_4 = 6 - 2 = 4$. So

$B[4][6] = \text{Max}\{2 + B[i-1][w - w_i], B[i-1][w]\} = \text{Max}\{2 + B[3][4], B[3][6]\} = 7$

When $i = 4$, $w_4 = 2$, $p_4 = 2$ and $w = 7$, then $w - w_4 = 7 - 2 = 5$. So

$B[4][7] = \text{Max}\{2 + B[i-1][w - w_i], B[i-1][w]\} = \text{Max}\{2 + B[3][5], B[3][7]\} = 9$

When $i = 4$, $w_4 = 2$, $p_4 = 2$ and $w = 8$, then $w - w_4 = 8 - 2 = 6$. So

$B[4][8] = \text{Max}\{2 + B[i-1][w - w_i], B[i-1][w]\} = \text{Max}\{2 + B[3][6], B[3][8]\} = 9$

When $i = 4$, $w_4 = 2$, $p_4 = 2$ and $w = 9$, then $w - w_4 = 9 - 2 = 7$. So

$B[4][9] = \text{Max}\{2 + B[i-1][w - w_i], B[i-1][w]\} = \text{Max}\{2 + B[3][7], B[3][9]\} = 11$

When $i = 4$, $w_4 = 2$, $p_4 = 2$ and $w = 10$, then $w - w_4 = 10 - 2 = 8$. So

$B[4][10] = \text{Max}\{2 + B[i-1][w - w_i], B[i-1][w]\} = \text{Max}\{2 + B[3][8], B[3][10]\} = 12$

When $i = 4$, $w_4 = 2$, $p_4 = 2$ and $w = 11$, then $w - w_4 = 11 - 2 = 9$. So

$B[4][11] = \text{Max}\{2 + B[i-1][w - w_i], B[i-1][w]\} = \text{Max}\{2 + B[3][9], B[3][11]\} = 13$

Scanned by CamScanner

B[I] [W]	0	1	2	3	4	5	6	7	8	9	10	11	Allowed items
ITEM 0	0	0	0	0	0	0	0	0	0	0	0	0	{ }
1	0	0	0	0	5	5	5	5	5	5	5	5	{1}
2	0	0	0	4	5	5	5	9	9	9	9	9	{1, 2}
3	0	0	0	4	5	5	7	9	9	11	12	12	{1, 2, 3}
4	:0	0	2	4	5	6	7	9	9	11	12	13	{1, 2, 3, 4}
5	0												

Step 6: When $i = 5$, $w_5 = 2$ and $p_5 = 3$. Since $w_5 > 1$, so $w - w_1 = 1-2 = -1$. Therefore

$B[5][1] = B[4][1] = 0$

When $i = 5$, $w_5 = 2$, $p_5 = 3$ and $w = 2$, then $w - w_5 = 2 - 2 = 0$. So

$B[5][2] = \text{Max}\{3 + B[i-1][w-w_i], B[i-1][w]\} = \text{Max}\{3 + B[4][0], B[4][2]\} = 3$

When $i = 5$, $w_5 = 2$, $p_5 = 3$ and $w = 3$, then $w - w_5 = 3 - 2 = 1$. So

$B[5][3] = \text{Max}\{3 + B[i-1][w-w_i], B[i-1][w]\} = \text{Max}\{3 + B[4][1], B[4][3]\} = 4$

When $i = 5$, $w_5 = 2$, $p_5 = 3$ and $w = 4$, then $w - w_5 = 4 - 2 = 2$. So

$B[5][4] = \text{Max}\{3 + B[i-1][w-w_i], B[i-1][w]\} = \text{Max}\{3 + B[4][2], B[4][4]\} = 5$

When $i = 5$, $w_5 = 2$, $p_5 = 3$ and $w = 5$, then $w - w_4 = 5 - 2 = 3$. So

$B[5][5] = \text{Max}\{3 + B[i-1][w-w_i], B[i-1][w]\} = \text{Max}\{3 + B[4][3], B[4][5]\} = 7$

When $i = 5$, $w_5 = 2$, $p_5 = 3$ and $w = 6$, then $w - w_4 = 6 - 2 = 4$. So

$B[5][6] = \text{Max}\{3 + B[i-1][w-w_i], B[i-1][w]\} = \text{Max}\{3 + B[4][4], B[4][6]\} = 8$

When $i = 5$, $w_5 = 2$, $p_5 = 3$ and $w = 7$, then $w - w_4 = 7 - 2 = 5$. So

$B[5][7] = \text{Max}\{3 + B[i-1][w-w_i], B[i-1][w]\} = \text{Max}\{3 + B[4][5], B[4][7]\} = 9$

When $i = 5$, $w_5 = 2$, $p_5 = 3$ and $w = 4$, then $w - w_4 = 8 - 2 = 2$. So

$B[5][8] = \text{Max}\{3 + B[i-1][w-w_i], B[i-1][w]\} = \text{Max}\{3 + B[4][6], B[4][8]\} = 10$

When $i = 5$, $w_5 = 2$, $p_5 = 3$ and $w = 9$, then $w - w_4 = 9 - 2 = 7$. So

$B[5][9] = \text{Max}\{3 + B[i-1][w-w_i], B[i-1][w]\} = \text{Max}\{3 + B[4][7], B[4][9]\} = 12$

When $i = 5$, $w_5 = 2$, $p_5 = 3$ and $w = 10$, then $w - w_4 = 10 - 2 = 8$. So

$B[5][10] = \text{Max}\{3 + B[i-1][w-w_i], B[i-1][w]\} = \text{Max}\{3 + B[4][8], B[4][10]\} = 12$

When $i = 5$, $w_5 = 2$, $p_5 = 3$ and $w = 11$, then $w - w_4 = 11 - 2 = 9$. So

$B[5][11] = \text{Max}\{3 + B[i-1][w-w_i], B[i-1][w]\} = \text{Max}\{3 + B[4][9], B[4][11]\} = 14$

Scanned by CamScanner

B[i] [w]	Weights 0	1	2	3	4	5	6	7	8	9	10	11	Allowed Items
ITEM 0	0	0	0	0	0	0	0	0	0	0	0	0	{}
1	0	0	0	0	5	5	5	5	5	5	5	5	{1}
2	0	0	0	4	5	5	5	9	9	9	9	9	{1, 2}
3	0	0	0	4	5	5	7	9	9	11	12	12	{1, 2, 3}
4	:0	0	2	4	5	6	7	9	9	11	12	13	{1, 2, 3, 4}
5	0	0	3	4	5	7	8	9	10	12	12	14	{1, 2, 3, 4, 5}

The optimal profit of the given knapsack problem is 14.

Now we find the items of the optimal cost using Algorithms `KNAPSACKPATH (B i k)` with initial value of $i = N$ and $k = W$.

- We know that if $B[i][k] \neq B[i-1][k]$, that is, $B[i][k]$ and $B[i-1][k]$ are different then this means that the ith item is selected. So print i and set $k = k - w_i$ and $i = i - 1$. If $B[i][k] = B[i-1][k]$ then set $i = i - 1$ only.
- So now following the bottom-up approach and starting from $i = 5$ and $k = 11$ we have

$$B[5][11] \neq B[4][11] \text{ so } \textbf{print 5}$$

Set $k = k - w_i = k - w_5 = 11 - 2 = 9$ and set $i = -i - 1 = 4$.

$$B[4][9] = B[3][9] \text{ so set } i = -i - 1 = 3$$

$$B[3][9] \neq B[2][9] \text{ so } \textbf{print 3}$$

and set $K = 9 - 6 = 3$, $i = i - 1 = 2$

$$B[2][3] \neq B[1][3] \text{ so } \textbf{print 2}$$

and $K = 3 - 3 = 0$ and $i = 2 - 1 = 1$. Since the value of $k = 0$ (either k or i is zero) so stop. Now read the printed items in reverse order, that is, Print 2, Print 3, Print 5. So the optimal solution of the given knapsack problem contains item number 2, 3 and 5, that is, $S = \{2, 3, 5\}$ and the corresponding total weights $w = \{3 + 6 + 2\} \leq 11(W)$ and the total profit is $\{4 + 7 + 3\} = 14$ as shown by the bottom right most entry in the table.

The solution may also be obtained using the arrow concept as used in LCS. Let us obtain the solution for the above problem using arrow concept. Here each diagonal arrow corresponds to adding one item into the knapsack.

Scanned by CamScanner

- $B[5][11] = \text{Max}\{3 + B[i-1][w-w_i], B[i-1][w]\} = \text{Max}\{3 + B[4][9], B[4][11]\} = 14$
  The value of $B[5][11]$ is 14 when we have added the value of $B[4][9]$ so diagonal arrow (left-up) is from $B[5][11]$ to $B[4][9]$. Print the value of the row of the matrix square from where the diagonal array starts; here it is 5 as shown in the following diagram.

Weights →

B[i][w] (ITEM)	0	1	2	3	4	5	6	7	8	9	10	11	Optimal structure
0	0	0	0	0	0	0	0	0	0	0	0	0	
1	0	0	0	0	5	5	5	5	5	5	5	5	
2	0	0	0	4	5	5	5	9	9	9	9	9	Print 2
3	0	0	0	4	5	5	7	9	9	11	12	12	Print 3
4	:0	0	2	4	5	6	7	9	9	11	12	13	
5	0	0	3	4	5	7	8	9	10	12	12	14	Print 5

- $B[4][9] = \text{Max}\{2 + B[i-1][w-w_i], B[i-1][w]\} = \text{Max}\{2 + B[3][7], B[3][9]\} = 11$
  The value of $B[4][9]$ is 11 when we have taken the value of $B[3][9]$, so upward arrow from $B[4][9]$ to $B[3][9]$. No diagonal array so need not to take any value for printing.
- $B[3][9] = \text{Max}\{7 + B[2][3], B[2][9]\} = 11$
  The value of $B[3][9]$ is 11 when we have added the value of $B[2][3]$ so diagonal arrow (left-up) from $B[3][9]$ to $B[2][9]$. Print the value of the row of the matrix square from where the diagonal array starts; here it is 3.
- $B[2][3] = \text{Max}\{4 + B[1][0], B[1][3]\} = 4$
  The value of $B[2][3]$ is 4 when we have added the value of $B[1][0]$ so diagonal arrow (left-up) from $B[2][3]$ to $B[1][0]$. Print the value of the row of the matrix square from where the diagonal array starts; here it is 2.
- $B[1][0]$ means the knapsack has no more capacity to accommodate the items so stop. If one of the element of $B[i][w]$ is zero (either there is no more item to select or no more space in the knapsack) then stop.
- So the solution contains 5, 3, 2 in bottom-up fashion. Write these values in top-down fashion as 2, 3, 5.

4.6.6 Travelling Salesperson Problem

Travelling salesperson problem (TSP) is to obtain the shortest tour from a given set of cities while visiting each city exactly once and returning to the city from where we started. TSP is represented as a weighted graph $G(V, E)$ where V is the set of vertices (cities) and E is the set of edges (path between the cities). Every edge is labeled with the weight that denotes the distance between the connecting cities. The graph is

Scanned by CamScanner

The set of items to take can be deduced from the table, starting at $c[n, W]$ and tracing backwards where the optimal values came from. If $c[i, w] = c[i-1, w]$ item i is not part of the solution, and we are continue tracing with $c[i-1, w]$. Otherwise item i is part of the solution, and we continue tracing with $c[i-1, w-W]$.

Analysis

This dynamic-0-1-kanpsack algorithm takes $\Theta(nW)$ times, broken up as follows: $\Theta(nW)$ times to fill the c-table, which has $(n+1).(W+1)$ entries, each requiring $\Theta(1)$ time to compute. $O(n)$ time to trace the solution, because the tracing process starts in row n of the table and moves up 1 row at each step.

Scanned by CamScanner

$n = 3$

$W = 10$

$(P_1, P_2, P_3) = (8, 5, 5)$

$(w_1, w_2, w_3) = (6, 5, 5)$

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	8	8	8	8	8
2	0	0	0	0	0	5	8	8	8	8	8
3	0	0	0	0	0	5	8	8	8	8	10

Total profit = 10

$\boxed{\text{Item no. } (2, 3)}$

B[5, 11]

Scanned by CamScanner

Data structure for disjoint set

Disjoint-set operations

5 34

A disjoint-set data structure maintains a collection $S = \{S_1, S_2 \ldots S_k\}$ of disjoint dynamic set. Each set is identified by a representative, which is some member of the set.

Let x be an object in a set, then consider the following operations:-

1. Make-Set(x):- creates a new set whose only member (and thus representative) is pointed to by x.

2. Union(x, y): unites dynamic sets that contain x and y, say S_x and S_y, into a new set that is union of these 2 sets.

3. Find-Set(x): returns a pointer to the representative of the set containing x.

An application of disjoint-set data structures

One of many applications of disjoint-set data structures arises in determining the connected components of an undirected graph.

Scanned by CamScanner

The procedure connected-components uses disjoint set operations to compute the connected components of a graph G with set of vertices V[G] and set of edges E[G]. It initially places each vertex v in its own set. Then for each edge (u,v), it unites sets containing u and v.

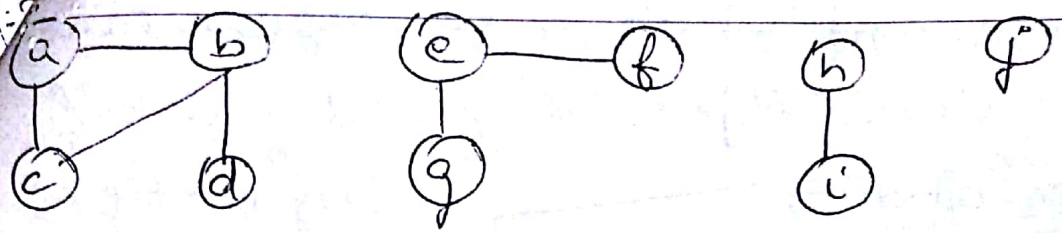
Then the procedure Same-Components indicates whether two vertices are in the same connected component.

```
Connected-Components [G)
for each vertex v ∈ V[G]
        do Make-Set(v)
for each edge (u,v) ∈ E[G]
  do if Find-Set(u) ≠ Find-Set(v)
     then Union (u,v)

Same-Component (u,v)
if Find-Set(u) = Find-Set(v)
   then return True
   else return False
```

Scanned by CamScanner

Consider a graph with four connected components.



The collection of disjoint sets after each edge is processed using Connected-Components procedure.

Edge Processed	Collection of disjoint sets									
Initial sets	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b,d)	{a}	{b,d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e,g)	{a}	{b,d}	{c}		{e,g}	{f}		{h}	{i}	{j}
(a,c)	{a,c}	{b,d}			{e,g}	{f}		{h}	{i}	{j}
(h,i)	{a,c}	{b,d}			{e,g}	{f}		{h,i}		{j}
(a,b)	{a,b,c,d}				{e,g}	{f}		{h,i}		{j}
(e,f)	{a,b,c,d}				{e,f,g}			{h,i}		{j}
(b,c)	{a,b,c,d}				{e,f,g}			{h,i}		{j}

Linked List representation of disjoint sets

A simple way to implement a disjoint-set data structure is to represent each set by a linked list. The first object in list serves as set's representative. Each object in linked list contains:

1) A set member
2) A pointer to object containing next set member
3) A pointer back to the representative.

Scanned by CamScanner

Linked list representation of 2 sets. One contains ~~objects b, c, e and h with c as representative~~ and other contains objects d, f and g with f as representation:- (Fig 1 + Fig 2)

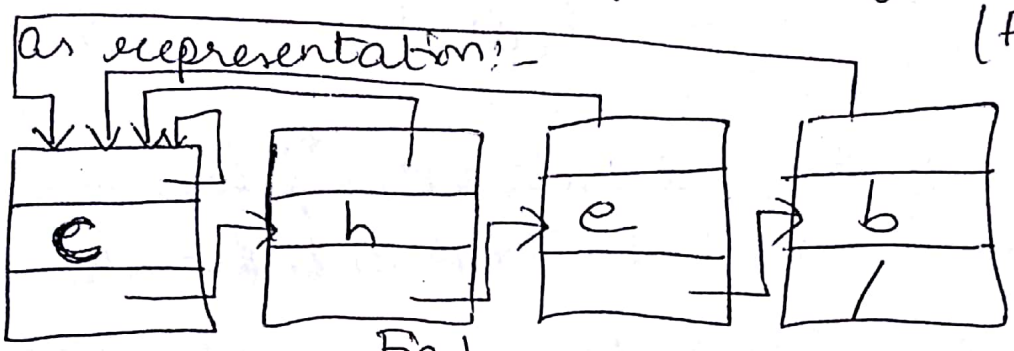


Fig 1

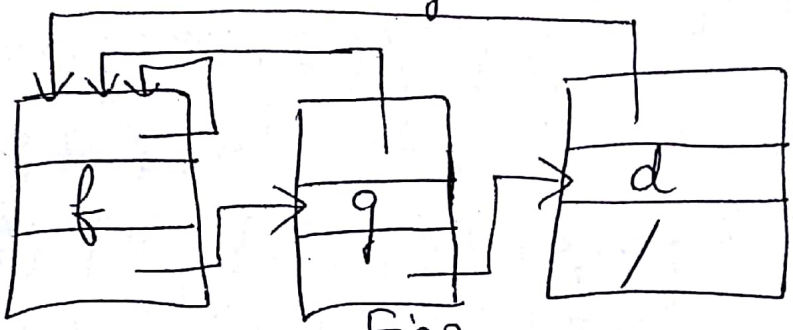


Fig 2

With the linked list representation, we have following operations:-

- Make-set (x):- we create a new linked list whose only object is x. It requires O(1) time.

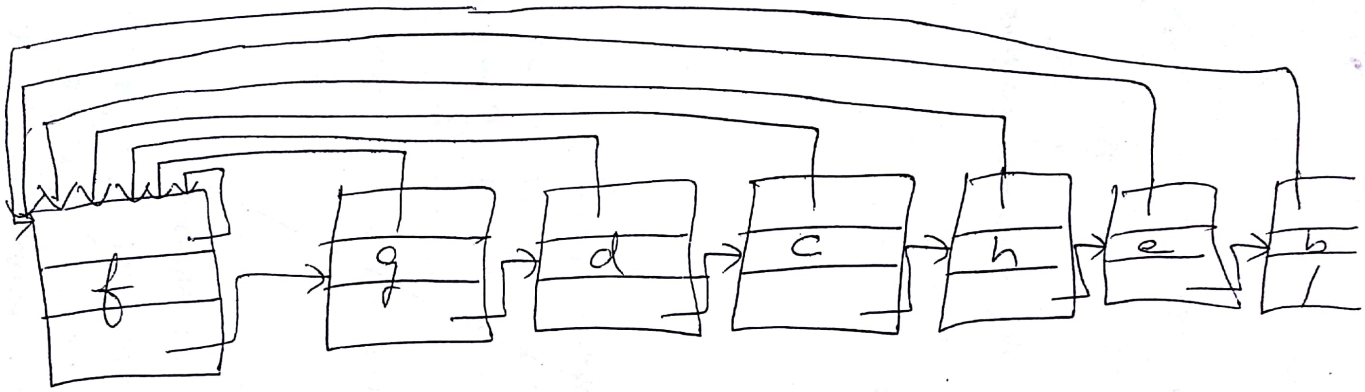
- Find-set (x):- we just return pointer from x back to the representative. It also requires O(1) time.

- Union (x, y): it is done by appending x's list onto end of y's list. For m operations, each operation requires on average $\Theta(m)$ time. (Fig 3)

A weighted union heuristic :- As Union requires $\Theta(m)$ time per call because we may be appending a longer list onto a shorter list. i.e we must update the pointer to the representative for each member of longer list.

Scanned by CamScanner

(3)



Representative of resulting set is f.

Fig 3.
Result of Union (e, g)

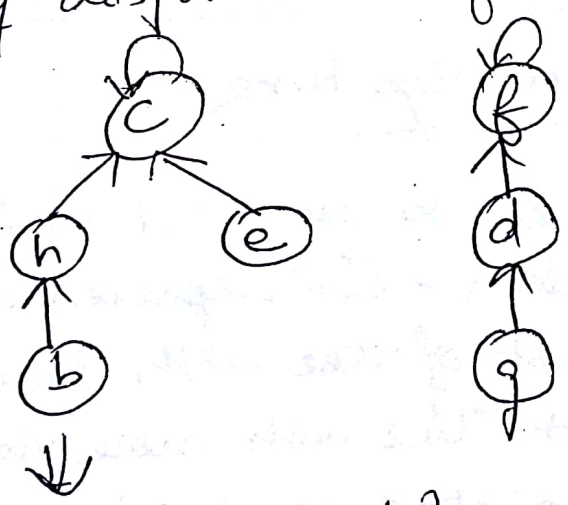
Scanned by CamScanner

In this weighted union heuristic, each representative also includes length of the list and we always append smaller list to the longer. With this weighted union heuristic, a single union operation can still take $\Omega(m)$ time if both sets have $\Omega(m)$ members.

Disjoint-set forests

In disjoint-set forests, we represents sets by rooted trees with each node containing one member. Here, each member points only to its parent. The root of each tree contains the representative and is its own parent.

eg. of disjoint-set forest



⇒ For set $\{d, f, g\}$
f :- representative

For set $\{b, c, e, h\}$
c :- representative

Fig 11

Scanned by CamScanner

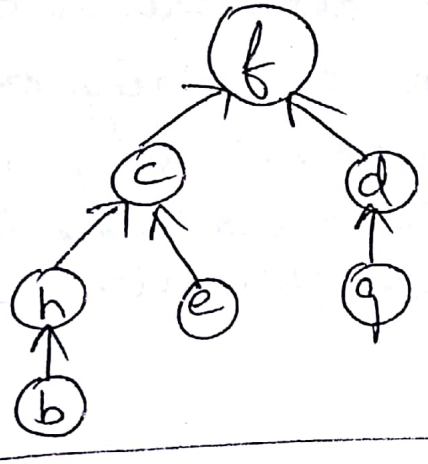
We perform disjoint set operations as follows:

(1) Make-set operation:- Creates a tree with just one node.

(2) Find-set operation:- We chase parent pointers until we find root of tree. The nodes visited on this path toward root constitute the find path.

(3) Union operation: Cause the root of one tree to point to root of the other.

eg.



⇐ Result of Union (e, g) for disjoint-set forest in Fig 11

Heuristics to improve the running time

(1) Union by rank: It is similar to weighted-union-heuristic we used with linked-list representation. The idea is to make the root of tree with fewer nodes point to the root of the tree with more nodes. Here, for each node, we maintain a rank which is an upper bound on height of node.

In Union by rank, root with smaller rank is made to point to the root with larger rank during a Union operation.

Scanned by CamScanner

Find-Set (x)

1. If $x \neq p[x]$
2. then $p[x] \leftarrow$ Find-set $(p[x])$
3. return $p[x]$

Scanned by CamScanner

Pseudocode for disjoint-set forests

For union by rank heuristic, we keep track of ranks with node x, we maintain rank$[x]$ which is upper bound on height of x.

In
- Make-set, the intial rank of single node is corresponding tree is 0.
- In Find-set, it leaves all ranks unchanged.

To apply Union to two trees, we make root of higher rank the parent of root of lower rank. In case of tie, we arbitarily choose one of roots as parent and increment its rank.

- Let us designate parent of node x by $p[x]$.

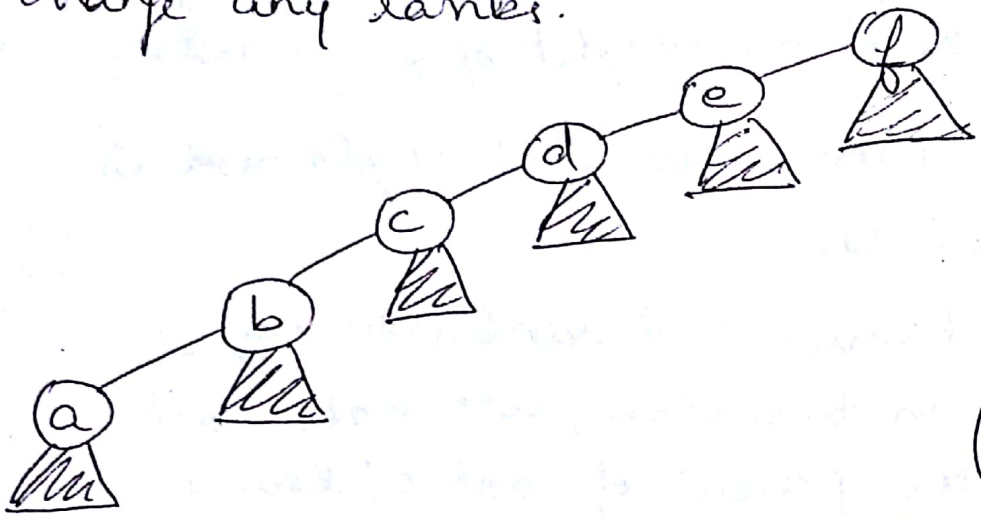
```
Make-set(x)
p[x] ← x
rank[x] ← 0
```

```
Union(x, y)
Link(Find-Set(x), Find-Set(y))
```

```
Link(x, y)
  If rank[x] > rank[y]
     then p[y] ← x
     else p[x] ← y
  if rank[x] = rank[y]
  then rank[y] ← rank[y] + 1
```

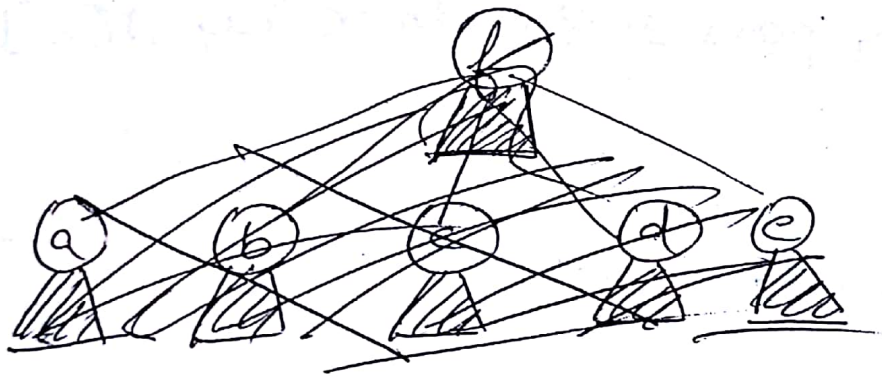
Scanned by CamScanner

2. Path-compression: It is used during the find-set operations to make each node on the find path point directly to root. It does not change any ranks.

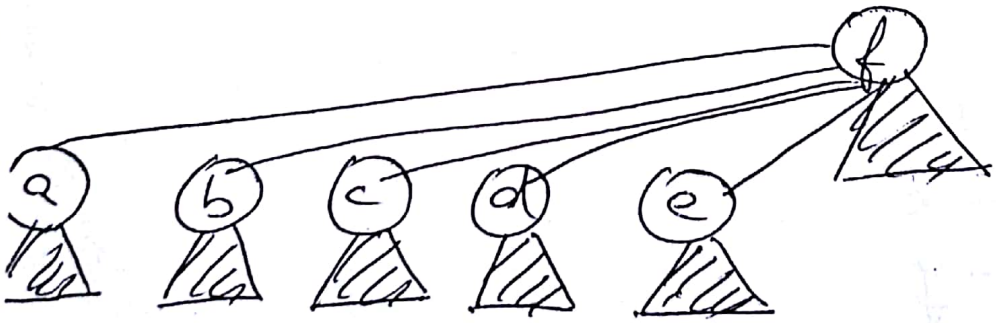


Triangles represent subtrees (arrows + self loops are omitted)

Tree representing a set prior to executing Find-set(a)



Set after executing Find-set(a)



Scanned by CamScanner