DM

Design & Analysis of Algorithms

Algorithms:- An algorithm is a finite set of instructions that is to be carried out in order to solve a particular task (in fact each instruction tells what action is to be performed).

→ An algorithm is a tool for solving a well ~~spec~~ specified computational problem

→ An algorithm is a sequence of unambiguous instructions for solving a problem

An algorithm is any well-defined computational procedure that takes some value or set of values as input and produces some value or set of values as output (An algorithm is thus a sequence of computational steps that transform the input into the output).

Ex:- for sorting problem

Input : A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$

output : $\langle a_1', a_2', \ldots, a_n' \rangle$ in increasing order.

Procedure → It is the i/f b/n an algorithm & the program. That mean both lang. & coding part of the design analysis of an algo. is known as program.

Program → The coding portion are

In general an instance of a problem consists of input needed to compute a sol^n to the problem.

→ Study of Algorithm provides insight into the intrinsic of the problem (power of understanding) as well as possible solution technique independent of :

- PL
- programming paradigm
- computer hardware
- or any other implementation aspects.

→ An algorithm can be specified in any language.

→ Normally algorithms are written in a pseudocode.

→ pseudocode conveys the structure of the algorithm clearly so that programmer can implement it in the language of his choice.

→ The prefix pseudo is used to give the information that the code is not meant to be compiled and executed on a computer.
→ It is easy to understand an algorithm by using pseudocode
→ The pseudocode hides the implementation details and thus one can easily focus on the computational aspects of an algorithms.
→ pseudo-code is a mixture of natural lang and high-level programming constructs that describe the main idea behind a generic implementation of algorithm.

Which algorithm is best for a given application depends on many factor
- time
- space
- no. of items to be sorted
- the extent to which the item is already somewhat sorted
- possible restrictions on the item values
- kind of storage device used (main memory / disk / tapes)

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

Algorithm → Finite Deterministic algorithm
→ Finite Non-deterministic "
→ Infinite Deterministic "

Finite Deterministic Algo Algo terminate in a finite amount of time. It always gives result that is uniquely dependent on the input.
Ex:- Finding root of a quadratic equation, square of a root.

Finite Non-deterministic Algo:- Algo terminates with in a finite amount of time but output maynot be unique.
Ex:- To generate a random number.

Infinite Deterministic Algo:- Algo which do not terminate either because a terminating condition was not satisfied for a given set of input.
Ex:- Task of monitoring the temperature in a nuclear reactor is an infinite algorithm.

Algorithm Dev → Analysis → Design → coding → Testing → maintenance

Extra: criterion 5 of algorithm is that each op^n be effective; each step must be such that it can, at least in principle, be done by a person using pencil and paper in a finite amount of time.

Ex:- performing arithmetic on integers is an example of an effective op^n, but arithmetic with real numbers is not since some values may be expressible only by infinitely long decimal expansion. Adding two such numbers would violate the effectiveness property.

→ Algorithm that are definite and effective are also called computational procedures.

→ To help us achieve the criteria of definiteness, algorithms are written in a programming language (PL).

→ Such languages are designed so that each legitimate sentence has a unique meaning.

→ A program is the expression of an algorithm in a PL.

Imp 1a The study of algorithms includes many important and active areas of research. There are four distinct areas of study & one can identify:

(1) How to devise algorithms:- creating an algorithm is an art which may never be fully automated. Various design techniques are there. By mastering these design strategies, it will become easier for you to devise new and useful algorithms.

(2) How to validate algorithms:- once an algorithm is devised, it is necessary to show that it computes the correct answer for all possible legal inputs. We refer to this process as algorithm validation.

The purpose of the validation is to assure us that this algorithm will work correctly independently of the issues concerning the PL it will eventually be written in. Once the validity of the method has been shown, a program can be written and a second phase begins. This phase is reffered to as program verification.

Scanned by CamScanner

(3) How to analyze algorithms:- As an algorithm is executed, it uses the computer's CPU to perform opn and its memory to hold the program + data

Analysis of algorithm or performance analysis refers to the task of determining how much computing time and storage an algorithm requires.

→ This is a challenging area which sometimes requires great mathematical skill

→ An important result of this study is that it allows you to make quantitative judgements about the value of one algorithm over another.

(4) How to test a program:- Testing a program consists of two phases:-

- Debugging
- profiling (or performance measurement)

Debugging is the process of executing programs on sample data sets to determine whether faulty results occur and, if so, to correct them.

However as E. Dijkstra has pointed out "debugging can only point to the presence of errors but not to their absence".

profiling or performance measurement is the process of executing a correct program on data sets and measuring the time and space it takes to compute the results.

Extra on pseudocode it is not typically concerned with issue of software engg.

→ Issues of data abstraction, modularity, error handling are ignored in order to convey the essence of algo more precisely.

Testing is the process of executing the program with an intention of finding errors, faults and failures.

Debugging is the process of fixing the bugs by the testers and then submitted to the developer

validation Evaluates the product itself. validation ensures the functionality intended behavior of the product (verification takes place before testing).

Verification evaluates the documents, plans, codes, requirements & specifications.

Basic properties:/ ALgorithm has 5 parts

① Input :- zero or more i/p data are externally supplied. That is no. of quantities are provided initially before the algo begins

② output :- There must be atleast one o/p produced.

③ Definiteness :- Each instruction must be clear and unambiguous. i.e, the processing rule specified in the algorithm must be precise and unambiguous and lead to a specific action.

Ex:- ADD 50 to p ← clearly defined 50 will be added to variable p.

ADD 50 to p or q ← does not clearly defined. so there is ambiguity because we are not sure 50 is added to p or q.

Ex:- "compute 5/0" is not clear.

④ finiteness :- Algo should terminate in finite amount of time. i.e The total time to carryout all the steps in the algorithm must be finite.

⑤ Effectiveness :- Each instruction must be sufficiently basic so that a person using paper and pencil can carry it out in a finite time.

Algorithm structure :-

⇒ Iterative → executes actions in loop
⇒ Recursive → re apply action to subproblem(s)

problem type :-

⇒ satisfying → Find any satisfactory solution.
⇒ optimization → Find best solution (v, cost metric)

Analyzing Algorithms :- Analyzing an algorithm has come to mean predicting the resources that the algorithm requires.

- Occasionally resources such as
  - memory
  - com/n bandwidth
  - Computer h/w are of primary Concern.

- But most often it is computational time that we want to measure.
- So analysis of algorithm refers to task of determining how much Computing time and storage an algorithm requires.
- In general the time taken by an algorithm grows with the size of the input, so it is traditional to describe the running time of a program as a function of the size of its input.
- The running time of an algorithm on a particular input is the number of primitive operations or steps executed.

Running time is affected by the
- h/w environment (processor, clock rate, memory disk etc)
- S/w environment (os, PL, comiler, interpreter)

Kind of Analysis :-

- worst case (usually) - upper bound on the running time for any i/p.
  - $T(n)$ = maximum time of algo on any input of size n.
    = Algo runs longest among all possible input of size n.
- Average case (sometimes)
  - $T(n)$ = expected time of algo over all i/p of size n.
- Best case
  → Algo runs the fastest among all possible inputs of size n.

NOTE Knowing the worst-case it guarantee that algorithm will not take any longer performance analysis of Algo:- To judge the performance of an algorithm there are basically two criterias:

(i) Time complexity:-(Tn) Amount of time it needs to run to completion as a function of its input size.

(ii) space complexity:-(Sn) Amount of memory it needs to run to completion as a function of its input size.

Extra [Analysis of an algo provides background info that gives us a general idea of how long an algo will take for a given problem set.]

→ The purpose of analysis is not to give a formula that will tell us exactly how many seconds or computer cycles needed.

→ Analysis of an algo is to predict the resources that the algo requires and such analysis is based on individual computational model.

How to calculate running time of an algorithm?

— By running the implementation of the algo on a computer. Alternatively we can calculate by using a technique called algo analysis.

— we can estimate by counting the no. of basic opn required by the algo to process an i/p of a certain size.

Basic opn: The time to complete basic opn does not depend on the particular values of its operands. So it takes a constant amount of time.

Ex:- add, sub, mull, div, Boolean (AND, OR, NOT), comparison, modulo & Bound operation.

Extra Execution time depends on many parameters which are independent of the particular algo; m/c clock rate, quality of code produced by compiler, whether or not the computer is multi-programmed etc.

→ Exe time of an algo is a function of the values of its i/p parameters.

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

→ When our i/p size n is going to be large enough in that case we have to take the help of asymptotic notations.

→ For large enough input the multiplicative constants and lower order terms of an exact running time are dominated by the effects of input size.

→ Notation which describe the algorithm efficiency and performance in a meaningfull way is asymptotic notation.

→ The growthrate of an algo is the rate at which running time of the algo grows as the size of the i/p grows.

$$\log n < n < n^2 < n^3 < 2^n$$
(n is very large)

Asymptotic Notations:- "Asymptotic means" - what will happen if n (i.e input size) is very large.

- The asymptotic running time of an algorithm is defined interms of function.
- Asymptotic analysis studies how the values of functions compare as their arguments converge to infinity.

~~Three~~ Five notations are used to compare orders of growth of algorithms.

(i) O → big oh → upper bound → worst case
(ii) Ω → big omega → Lower bound → best case
(iii) Θ → Theta → average case
(iv) o → small oh / little oh
(v) ω → small omega / little omega.

(1) Big Oh (O) notation: upper bound for the function f is provided by the big-oh notation.

The function $f(n) = O(g(n))$ (read as "f(n) is big-oh of g(n)") if and only if there exist a positive constant $c > 0$ and n_0 such that $f(n) \leq c * g(n)$ for all $n, n \geq n_0$

or we can write $\boxed{0 \leq f(n) \leq c*g(n) \text{ for all } n \geq n_0}$

→ So $f(n)$ always lies below $c*g(n)$ $\forall n \geq n_0$

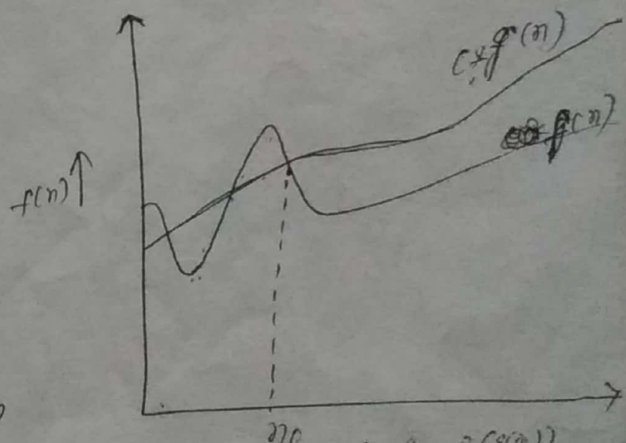
→ class of function $f(n)$ that grows no faster than $g(n)$

OR

$$\boxed{O(g(n)) = \{ f(n) : \exists \text{ positive constants } c \text{ and } n_0 \text{ such that } \forall n \geq n_0. \text{ we have } 0 \leq f(n) \leq c\,g(n) \}}$$

Intuitively: The set of all functions whose rate of growth is the same as or lower than that of $g(n)$.

So, $g(n)$ is an asymptotic upper bound for $f(n)$



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

→ n.

Ex:-

Ex:- $3n+3 = O(n)$ ← c ← g(n)
as $3n+3 \leq 4n$ $\forall n \geq 3$

Ex:- $3n+2 = O(n)$
as $3n+2 \leq 4n$, $\forall n \geq 2$

Ex:- $f(n) = 7n+5$ so $f(n) = O(n)$
as $7n+5 \leq 8n$ $\forall n \geq 5$

Ex:- $10n^2 + 4n + 2 = O(n^2)$
as $10n^2 + 4n + 2 \leq 11n^2$, $\forall n \geq 5$

Ex:- $3n+2 = O(n^2)$ as $3n+2 \leq 2n^2$, $\forall n \geq 2$

If we take any upper bound $O(n^3), O(n^4), \dots$ is satisfied.

But not $O(1)$ so $3n+2 = O(n)$
$= O(n^2)$
$= O(n^3)$
$\neq O(1)$
Imp

Ex:- $3n+2 \neq O(1)$ as $3n+2$ is not less than or equal to c for any constant c and all $n \geq n_0$.

Ex:- $10n^2+4n+2 \neq O(n) \neq O(1)$

Ex So $f(n) = 3n+1$

$$\left.\begin{array}{l} 3n+1 \leq 4n \\ 3n+1 \leq 5n \\ 3n+1 \leq n^2 \\ 3n+1 \leq n^3 \\ 3n+1 \leq 2^n \end{array}\right\} \text{Growth function}$$

$$\boxed{f(n) \leq c * g(n)}, \quad n \geq n_0$$

Incorrect bound

$7n+5 \neq O(1)$

$2n+3 \neq O(1)$

$10n^2+7 \neq O(n)$

Loose Bounds

$2n+3 = O(n^2)$

$4n^2+5n+6 = O(n^4)$

$3n^3+4n^2+n = O(n^6)$

Ex:- Let $f(n) = n+2$; then which condition is not satisfied.

(i) $O(n)$
(ii) $O(n^2)$
(iii) $O(n^3)$
(iv) $O(1)$ ✓

If $f(n)$ satisfies any lower function of $g(n)$ then it will satisfy the higher function of $g(n)$

NOTE Since O notation describes an upper bound, we use it to found the worst-case running time of an algorithm.

Ex $f(n) = 5n-2$ is $O(n)$ for $c=5$, $n_0=1$

TASK $10n^3+7n+5$ is $O(n^3)$ for $c \geq 22$, $n_0 \geq 1$

$3\log n + \log\log n$ is $O(\log n)$ for $c \geq 4$, $n_0 \geq 2$

Ex:- 2^{100} is $O(1)$ since $2^{100} \leq 2^{100} \cdot 1$

Ex:- $7/n$ is $O(\frac{1}{n})$

Ex:- $4n^3 + \Theta(n^2) = \Theta(n^3)$

NOTE We write $O(1)$ to mean computing time is constant.

Some common function

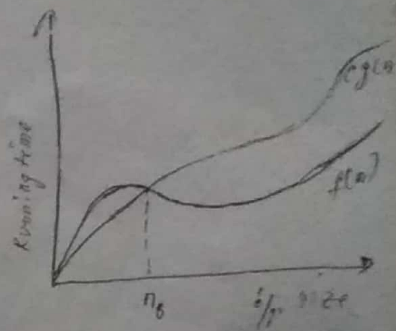
fast

1	-	constant
Log n	-	Logarithmic
n	-	Linear
n^2	-	quadratic
n^3	-	cubic
2^n	-	exponential
n!	-	factorial time
n^n	-	

slow

high time efficiency

low time efficiency



Big-O notation

NOTE :- Algo taking $O(2^n)$ time should never be Considered efficient.

$\log n < \log^2 n < \sqrt{n} < n < n\log n < n^2 < n^3 < 2^n$

$O((\log n)^k)$ - poly logarithmic time

$n \cdot \log n$ - linearithmic function

$n \log^k n$ - quasi linear time

$O(n^k)$ = polynomial time.

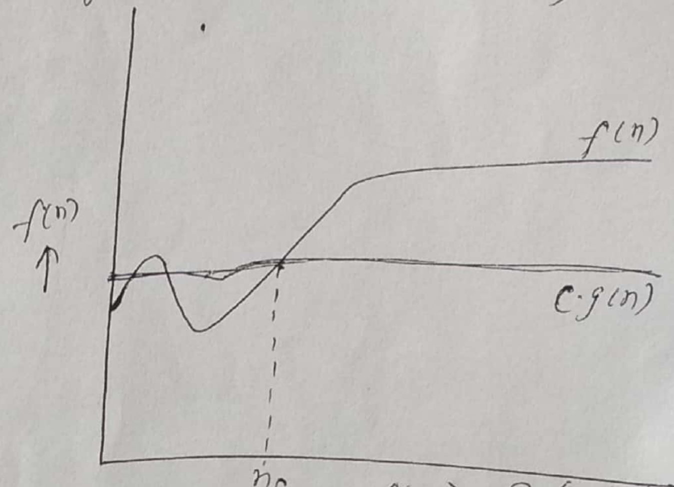
$O(1) < \log n < n < n\log n < n^2 < n^3 < 2^n$

(2) Big-omega notation — This gives asymptotic lower bound

The function $f(n) = \Omega\, g(n)$ (pronounced "big-omega of g of n" or sometimes just omega of g of n) if and only if there exist the positive c and n_0 such that $\boxed{f(n) \geq c * g(n)}$ for all n where $n \geq n_0$

$\Omega(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \leq c * (g(n)) \leq f(n)$ for all $n \geq n_0 \}$

Ex:- $3n+3 = \Omega(n)$
as $3n+3 \geq 3n$, $\forall\, n \geq 1$



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

(2) $3n+2 = \Omega(n)$
as $3n+2 \geq 3n$ $\forall\, n \geq 1$

(3) $3n+3 = \Omega(n) \neq \Omega(n^2)$
But it can be
$3n+3 = \Omega(1)$

(4) $6 * 2^n + n^2 \neq \Omega(3^n)$
$6 * 2^n + n^2 = \Omega(n^2)$
$= \Omega(n)$ or $\Omega(2^n)$ as $6 * 2^n + n^2 \geq 2^n$ for $n \geq 1$

(5) $10n^2 + 4n + 2 = \Omega(n^2)$ as $10n^2 + 4n + 2 \geq n^2$
$= \Omega(n) = \Omega(1) \neq \Omega(n^3)$

(6) $6 * 2^n + n^2 = \Omega(2^n)$ as $6 * 2^n + n^2 \geq 2^n$ for $n \geq 1$. or $\Omega(1)$

(7) $3 \log n + \log \log n$ is $\Omega(\log n)$
$3 \log n + \log \log n \geq 3 \log n$ for $n \geq 2$

(8) $f(n) = 3n+1$
$3n+1 \geq 3n$ here $f(n) = 3n+1$ and $c = 3$, $g(n) = n$
so $n_0 = 1$ so $3n+1 = \Omega(n)$

(9) prove that $3n+1 = \Omega(1)$

$f(n) \geq c * g(n)$
$f(n) = 3n+1$
$3n+1 \geq 1 \cdot 1$
$n \geq 1$
so here $c = 1$, $g(n) = 1$ so $3n+1 = \Omega(1)$ proved

③ Theta (Θ) notation:- The function $f(n) = \Theta(g(n))$ if there exist constants $c_1 > 0$ and $c_2 > 0$ and an integer constant $n_0 \geq 1$ such that $\boxed{c_1 g(n) \leq f(n) \leq c_2 g(n)}$ for every integer $n \geq n_0$

$\Theta(g(n)) = \{ f(n) : \exists$ positive constants c_1, c_2 and n_0 such that $\forall n \geq n_0, \; 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \}$

→ If $f(n) \geq c_1 * g(n)$ and $f(n) \leq c_2 * g(n)$ then $f(n) = \Theta(g(n))$

→ $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$

→ Θ-notation is tight bound.

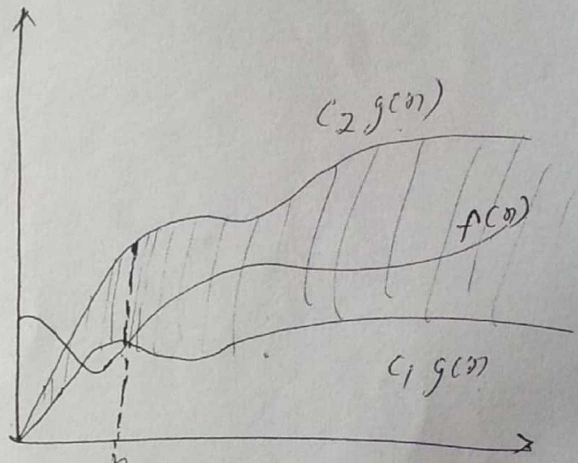
→ $T_{avg}(n) \leq T_{worst}(n)$

→ $f(n) = \Theta(g(n)) \Rightarrow f(n) = \Omega(g(n))$

$\Theta(g(n)) \subseteq \Omega(g(n))$

→ "Running time is $O(f(n))$" ⇒ worst case is $O(f(n))$

→ "Running time is $\Omega(f(n))$" ⇒ Best case is $\Omega(f(n))$



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

Ex:- $f(n) = 3n^2 + 5$

$3n^2 + 5 \leq 4n^2$

here $f(n) = 3n^2 + 5$, $c_1 = 4$, $g(n) = n^2$

so $f(n) = 3n^2 + 5 = \Theta(n^2)$ ——— ①

Again $f(n) = 3n^2 + 5$

$3n^2 + 5 \geq 3n^2$

$f(n) = 3n^2 + 5$, $c_2 = 3$, $g(n) = n^2$

So $f(n) = 3n^2 + 5 = \Omega(n^2)$ ——— ②

from (i) and (ii), we can conclude that

$$\boxed{f(n) = \Theta(n^2)}$$

Theorem-1

For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

Ex:- The function $\boxed{5n+2 \text{ is } \Theta(n)}$

As $5n+2 \geq 5n$ for all $n \geq 1$ and $5n+2 \leq 6n$ for all $n \geq 2$

So $c_1 = 5$, $c_2 = 6$ and $n_0 = 2$ for which we can verify that

$5n+2 = \Theta(n)$.

Ex:- $10n^2+4n+2 = \Theta(n^2)$ as $\boxed{5n^2 \leq 10n^2+4n+2 \leq 11n^2}$; $\forall n \geq 5$

Ex:- $3n+2 = \Theta(n)$ as $3n \leq 3n+2 \leq 4n$, $\forall n \geq 2$. Here $c_1 = 3$, $c_2 = 4$ and $n_0 = 2$.

Ex:- $f(n) = 10n^2+7$ as $10n^2 < 10n^2+7$ for all 'n' $c_1 = 10$

also $10n^2+7 \leq 11n^2$ $c_2 = 11$, $n_0 = 7$

Thus $10n^2 < 10n^2+7 \leq 11n^2$, $c_1 = 10$, $c_2 = 11$, $n \geq n_0 = 7$.

Incorrect bound

$7n+5 \neq \Theta(1)$

$7n+5 \neq \Theta(n^2)$

$2n+3 \neq \Theta(1)$

$10n^2+7 \neq \Theta(n^3)$

$10n^2+7 \neq \Theta(1)$

$10n^2+4n+2 \neq \Theta(n)$

$6*2^n+n^2 \neq \Theta(n^2)$

$6*2^n+n^2 \neq \Theta(n^{100})$

$6*2^n+n^2 \neq \Theta(1)$

Ex:- $6*2^n+n^2 = \Theta(2^n)$ as $6*2^n \leq 6*2^n+n^2 \leq 7\,2^n$, $n \geq 2$.

Scanned by CamScanner

Theorem-1 if $f(n) = a_m n^m + a_{m-1} n^{m-1} + \cdots + a_1 m + a_0$, then $f(n) = \Omega(n^m)$ (or) $f(n) = O(n^m)$ (or) $f(n) = \Theta(n^m)$

(4) **small-oh (o) notations:-** The function $f(n) = o(g(n))$ if there exist a real constant $c > 0$ and an integer constant $n_0 \geq 0$ such that $f(n) < c\,g(n)$ for every integer $n \geq n_0$.

$o(g(n)) = \{ f(n)$: for any positive constants $c > 0$, there exist a constant $n_0 > 0$ such that $\boxed{0 \leq f(n) < c\,g(n)}$ for all $n \geq n_0 \}$

Ex:- $2n = o(n^2)$ but $2n^2 \neq \overset{\leftarrow \text{small-oh}}{o}(n^2)$

→ The asymptotic upper bound provided by O-notation may or may not be asymptotically tight.
→ The bound $2n^2 = O(n^2)$ is asymptotically tight but the bound $2n = O(n^2)$ is not.
→ we use o-nation to denote an upper bound that is not asymptotically tight
→ The def^n of O-notation and o-notation are similar.
→ The main difference is that in $f(n) = O(g(n))$, the bound $0 \leq f(n) \leq c\,g(n)$ holds for some constant $c > 0$

But in $f(n) = o(g(n))$, the bound $0 \leq f(n) < g(n)$ holds for all constants $c > 0$.

→ Intuitively, in the o-notation, the function $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity; that is

$$\boxed{\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0}$$

Ex: $f(n) = 3n+2 = o(n^2)$ as $\lim_{n\to\infty} \frac{3n+2}{n^2} = 0$ zero

$\lim_{n\to\infty} \frac{3n+2}{n^2} = \lim_{n\to\infty} \frac{3}{n} = 0$

Ex: But $f(n) = 3n+2 \neq o(n)$ as $\lim_{n\to\infty} \frac{3n+2}{n} = \lim_{n\to\infty} 3+0 \neq 0$

So $\boxed{3n+2 \neq o(n) \text{ But } 3n+2 = O(n)}$

Ex:- $3n+2 = o(n\log n)$ as $\lim_{n\to\infty} \frac{3n+2}{n\log n} = \lim_{n\to\infty} \left(\frac{3}{\log n} + \frac{2}{n\log n}\right) = 0$

NOTE: ① $g(n)$ is an upper bound for $f(n)$ that is not asymptotically tight.
② For any higher term growth of big-oh notation, the little-oh is satisfied.

(5) Little omega notation (ω):- We use ω-notation to denote a lower bound that is not asymptotically tight.

$\omega(g(n)) = \{f(n) :$ for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq c\,g(n) < f(n)$ for all $n \geq n_0 \}$

→ If $f(n) \in \omega(g(n))$ if and only if $g(n) \in o(f(n))$

Ex:- $n^2/2 = \omega(n)$ but $n^2/2 \neq \omega(n^2)$

The relation $f(n) = \omega(g(n))$ implies that

$\boxed{\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty}$ or $\boxed{\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0}$

↳ That is $f(n)$ becomes arbitrarily large relative to $g(n)$ as n approaches infinity

Alternate def^n The function $f(n) = \omega(g(n))$ iff $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$

Ex: $f(n) = 3n+2 \neq \omega(n)$ as $\lim_{n\to\infty} \frac{n}{3n+2} = \frac{1}{3}$

$f(n)/g(n) = \infty$. $\frac{3n+2}{n} = 3 + \frac{2}{\infty} = 3+0 = 3$ (L Hospital rule)

Ex: But $f(n) = 3n+2 = \omega(\log n)$ as $\lim_{n\to\infty} \frac{\log n}{3n+2} = \lim_{n\to\infty} \frac{1}{3n} = 0$

Ex: But $f(n) = 3n+2 \neq \omega(n^2)$ as $\lim_{n\to\infty} \frac{n^2}{3n+2} = \lim_{n\to\infty} \frac{n}{3} + \frac{n^2}{2}$

so $3n+2 \neq \omega(n^2)$ because this n^2 is larger than $3n+2$

Ex: $3n+2 = \omega(1)$

as $\lim_{n\to\infty} \frac{1}{3n+2} = 0$ so $3n+2 = \omega(1)$

NOTE: ① for any lower growth of the omega notation (Ω) little omega relation condition is satisfied.

② for small-oh $f(n)$ is less than $g(n)$

③ for small-omega $f(n)$ is larger than $g(n)$.

Correct Bounds (little-oh)

→ $f(n) = 3n+5 = o(n^2)$ as $3n+5 = O(n^2)$

→ $f(n) = 3n^2+4n = o(n^4)$ as $3n^2+4n = O(n^4)$

→ $f(n) = 4n^3+2n+3 = o(n^4)$ as $4n^3+2n+3 = O(n^4)$.

Incorrect Bounds (little-oh)

$f(n) = 2n+3 \neq o(n)$

→ $f(n) = 27n^2+16n \neq o(n^2)$ as $\lim_{n\to\infty} \frac{27n^2+16n}{n^2} = 27 \neq 0$

→ $f(n) = 10n^2+7 \neq o(n^2)$ as $\lim_{n\to\infty} 10n^2+7/n^2 = 10 \neq 0$

→ $f(n) = 3n^3+4n \neq o(n^3)$

Ex of little-oh or small-oh

$3n+2 = o(n \log\log n)$ but $3n+2 \neq o(n)$

$6 * 2^n + n^2 = o(3^n)$

$= o(2^n \log n)$

$\neq o(2^n)$

Scanned by CamScanner

Recurrences - the fact of occurring again

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

- Recurrence relations are recursive defn of mathematical functions or sequences
- Recurrence relations arises when we analyze the running time of iterative or recursive algorithm.
- The running time of a recursive algorithm can be obtained by a recurrence.

Ex:- $\boxed{T(n) = 2 * T(n/2) + n}$ example of recurrence relation

Ex:- The complexity of many Divide and conquer algorithms is given by recurrences of the form:-

$$\boxed{T(n) = \begin{cases} T(1) & , n=1 \\ a\,T(n/b) + f(n) & , n>1 \end{cases}}$$

where 'a' is the no. of subproblems and 'b' is the size of each subproblems. i.e a and b are known constants.

We assume that $T(1)$ is known and n is power of b (i.e $n = b^k$)

Methods of solving recurrence relation :-

(1) Substitution method → Guess the form of the soln. This will work when soln is easy to guess.

(2) Iteration "

(3) Recursion tree "

(4) Master method

Consider the linear search.

Recursively look at one element then search remaining elements

1	2	3	...			n

so $\boxed{T(n) = T(n-1) + c}$

" The cost of searching n elements is the cost of ~~searching~~ looking at 1 element, plus the cost of searching n-1 elements ".

(1) Substitution method :- (guess method)

In this method we guess a bound and then use mathematical induction to prove our guess correct

So the substitution method for solving recurrences entails two steps:

① → Guess the form of the solution
② → use the mathematical induction to find the constants and show that the solution works.

This method is powerful but it is obviously can be applied only in cases when it is easy to guess the form of the answer.

- This method can be used to establish either upper or lower bounds on the recurrence.
- unfortunately there is no general way to guess the correct solutions to recurrences.
- Guessing a sol^n takes experience and occasionally creativity.

The most general method :

— Guess the form of sol^n
— verify by induction
— solve for constants

Mathematical induction :- is a method of mathematical proof typically used to establish a given stmt for all natural numbers.

✓

Ex-1 $T(n) = T(n/2) + 1$ ① Determine upper bound

Let's guess that the solution is $T(n) = O(\log n)$

So our method is to prove that $T(n) \le c \log n$ for any constant $c > 0$.

So $T(n) \le c * \log n$ —②

$T(n/2) \le c * \log(n/2)$
$\le c * \log n - c * \log 2$

Now, eq^n ① becomes,

$T(n) \le c * \log n - c * \log 2 + 1$
$\le c \log n - c + 1$
$\le c \log n$

$T(n) = O(\log n)$ // proved

Scanned by CamScanner

Recursion-Tree method:- A recursion tree is generated by tracing the execution of a recursive algorithms. It is a pictorial representation of recursion method.

A recursion tree models the costs (time) of a recursive execution of an algorithm.

→ In a recursion tree each node represents the cost of a single subproblem somewhere in the set of recursive function invocations.

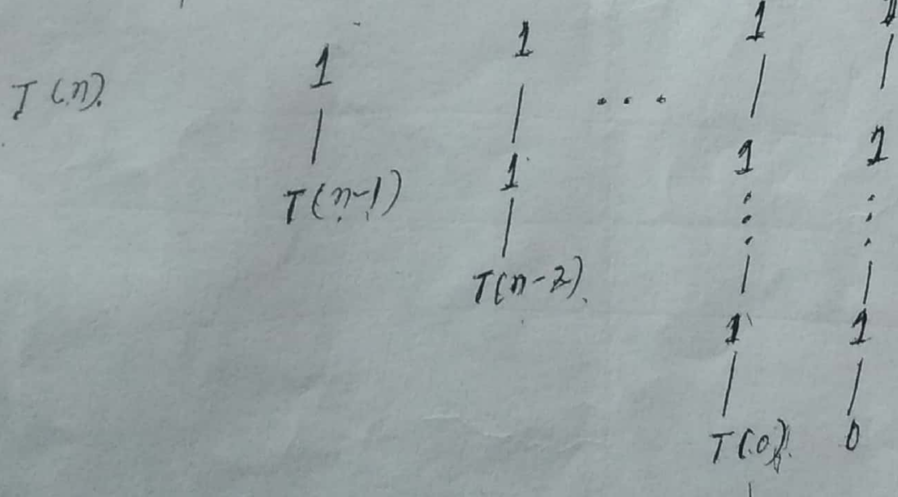
→ We sum the costs within each level of the tree to obtain a set of per-level costs and we sum all the per-level costs to determine the total cost of all levels of the recursion.

→ Recursion trees are useful when the recurrence describes the running time of a D-A-C algorithms ~~Digital to analog conversion~~ Divide and Conquer ~~algo~~ algorithm which based on multi branch recursion. This method is good for generating guasses for the substitution method.

Ex:- $T(n) = T(n-1)+1$

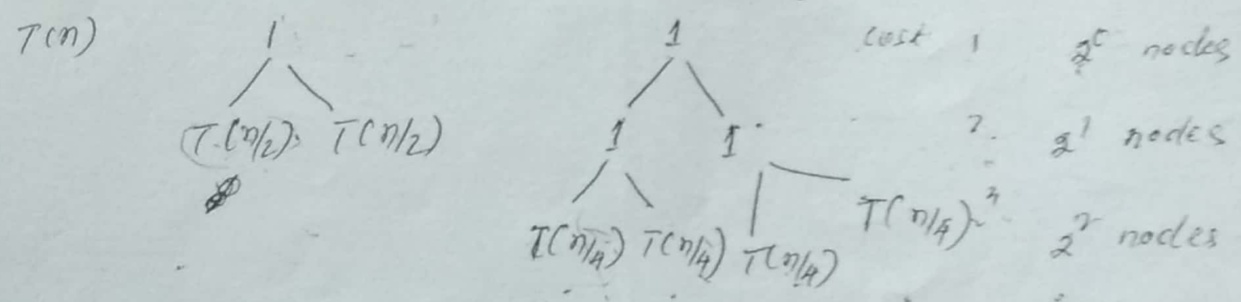
every parent replaced by constant term

At each level of tree, replace the parent of that subtree by the recurrence relation $T(n)=T(n-1)+1$ and constant term of the recurrence relation make $T(K-1)$ the child.



$T(n)$ is computed by finding the sum of elements at each level of the tree. Therefore $T(n) = \Theta(n)$

Ex:-2 $T(n) = 2T(\frac{n}{2}) + 1$ Here the no. of child = 2.



Now last $T\left(\frac{n}{2^i}\right) = 1$

so $\frac{N}{2^i} = 1 \Rightarrow N = 2^i \Rightarrow 2^i = N$ so $i = \log_2 n$ // Let $n = 2^i$

so, height $= \log_2 n$ and level $= \log_2 n + 1$

But last step contains 2^i nodes

so total $= 1 + 2 + 4 + \quad + (i+1)$

$= 1\left(\frac{2^{i+1} - 1}{2-1}\right) = 2^{i+1} - 1 = 2 \cdot 2^i - 1 = 2n - 1 \leq 3n$

so $\boxed{T(n) = O(n)}$

Master Method:- The master method applies to recurrences of the form $\boxed{T(n) = a T(n/b) + f(n)}$ where $a \geq 1$ and $b > 1$ are constants $f(n)$ is asymptotically +ve function

The above recurrence describes the running time of an algorithm that divides a problem of size n into a subproblem each of size n/b. where a and b are positive constants.

The master method depends on the following theorem

Master theorem:- Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function and let $T(n)$ be defined by the recurrence.

$$\boxed{T(n) = a T(n/b) + f(n)}$$

Then $T(n)$ can be bounded asymptotically 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 $T(n) = \Theta(n^{\log_b a} \lg n)$
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ Then and if $a f(n/b) \leq c f(n)$ for some constants $c < 1$ and all sufficiently large n, Then

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

NOTE- $\boxed{\text{It means compare } f(n) \text{ with } n^{\log_b a}}$

Three common cases:-

1. $f(n)$ grows polynomially slower than $n^{\log_b a}$ Then $T(n) = \Theta(n^{\log_b a})$
2. $f(n)$ and $n^{\log_b a}$ grows at similar rates then $T(n) = \Theta(n^{\log_b a} \lg n)$
3. $f(n)$ grows polynomially faster then $n^{\log_b a}$ and $f(n)$ satisfies the regularities condition Then $T(n) = \Theta(f(n))$

Ex:- $T(n) = 4T(n/2) + n$

Here $a = 4, b = 2$ so $n^{\log_b^a} = n^2$ But $f(n) = n$

Case-1 $f(n) = O(n^{2-\epsilon})$ for $\epsilon = 1$ // case-1 is satisfied

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

$\log_2^4 = \log_2^{2^2}$
$= 2 \log_2^2$
$= 2$
$\therefore$
$f(n) < n^{\log_b^a}$
So case-1 is satisfied

Ex:- $T(n) = 4T(n/2) + n^2$

Here $a = 4, b = 2$ so $n^{\log_b^a} = n^2$ and $f(n) = n^2$

So case-2 is satisfied

$f(n) = \Theta(n^{\log_b^a} \lg n) = \Theta(n^2 \lg n)$ //

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

Here $a = 4, b = 2$, so $n^{\log_b^a} = n^2$ but $f(n) = n^3$

So case-3 is satisfied

$f(n) = \Omega(n^{2+\epsilon})$ for $\epsilon = 1$

and $4(n/2)^3 \leq c n^3$ for $c = 1/2$

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

$f(n) = n^3 > n^2$

$a f(n/b) \leq c f(n)$.
$4 f(n/2) \leq c n^3$
$4 (n/2)^3 \leq c n^3$
$\frac{\cancel{4} \cancel{n^3}}{\cancel{8} 2} \leq c \cancel{n^3}$

Ex:- $T(n) = 9T(n/3) + n$

Here $a = 9, b = 3, f(n) = n$

$n^{\log_b^a} = n^{\log_3^9} = n^2$. Since $f(n) = n$. Case-1 is satisfied.

$f(n) = O(n^{\log_3 9 - \epsilon})$ where $\epsilon = 1$ $= O(n^{2-\epsilon})$

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

Ex:- $T(n) = 2(2n/3) + 1$

Here $a = 2, b = 3/2$ $f(n) = 1$ but $n^{\log_b^a} = n^{\log_{3/2} 1} = n^0 = 1$

So case-2 is satisfied

$f(n) = \Theta(n^{\log_b^a}) = \Theta(1)$

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

$O(n^{\log_{3/2} 1 - \epsilon})$ is < 1

Algorith classifications : Brute-force

→ This algorithm simply tries all possibilities until a satisfactory solution is found.

- Such an algorithm can be either optimizing or ~~satisfactory~~/satisficing

<u>Optimizing</u> :- Find the best solution. This may require finding all solutions, or if a value for the best solution is known, it may stop when any best sol^n is found.

- <u>Example</u>:- Finding a best path for a Travelling salesman problem; i.e cover all the cities & the total path cost is minimum

<u>Satisficing</u>:- Stop as soon as a solution is found that is good enough

- <u>Example</u>: Finding travelling salesman path which is within 10% of optimal.

<u>Improvements</u>:-

* Often Brute-force algorithms require exponential time

* Various heuristics and optimizations can be used.

Scanned by CamScanner

- **Heuristic:** A rule of thumb that helps you decide which possibilities to look at first.

Explain: We have some previous value using that previous value we guess that this is the probable path which we should explore first.

- **Optimization:** In this case, a way to eliminate certain possibilities without fully exploring them.

That means we are not going to explore all the possibilities. If we go on doing such then the complexity of algorithm will be too high. So that the optimization we may not explore all the possibilities with fullest extent. We may leave them i.e we may not execute all the possibilities we not test all the possibilities, eliminate certain possibilities without fully exploring them

Module - II

Selection Sort

Assume that ~~we~~ we have the data

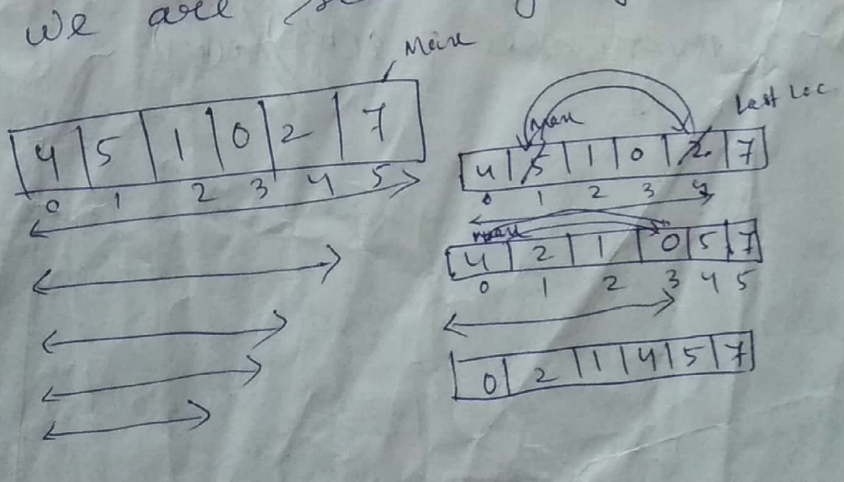
	4	5	1	0	2	7
These are the Location →	0	1	2	3	4	5

In selection sort we are going to do is in this complete data initially we have find that which is the maximum element → then put the maximum element to the last location. means we have to put in

→ That 5th location
→ Next time we ~~have~~ are going to find in this array from 0 to 4th location then find the maximum element and put that last location

Example

→ Assume we are searching by linear search



```
Selection sort (int A[ ], int n)
{
  int i, j;
  int max;
  for (i = n-1; i > 0; i--)
  {
    max = 0; index
    for (j = 1; j < i; j++)
    {
      if A[j] > max A[max];
      {
        max = j;
      }
    }

    int temp = A[max];

    A[max] = A[i]
      A[i] = temp
  }
}
```

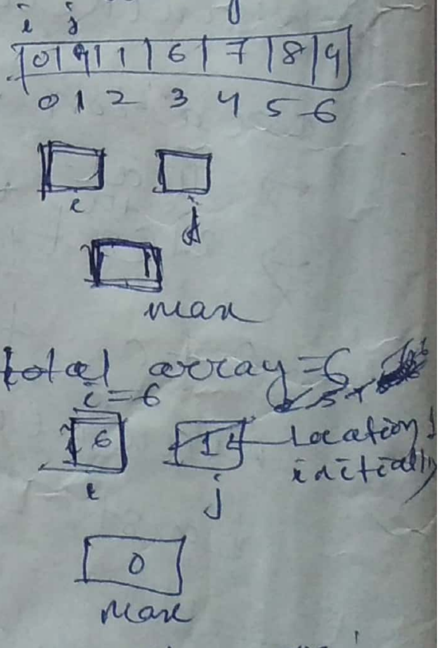
after this inner for loop we have to swap

Best case $= O(n^2)$, $\omega(n^2)$ — gnsone book

Worst case $= O(n^2)$

Avg case $= \theta(n^2)$ / $\theta(n^2)$

Assume this is the array



Here when A[j] is

GREEDY METHOD

- A greedy algorithm always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to globally optimal solution.
- → ~~But~~ greedy algorithms donot always yields optimal sol^n but for many problems they do.
- → once a decision is made it is never revoked.
- → An essential point of greedy solution is that we never have to revise our greedy decisions and this lead to fast algorithm provided.
- → The solution obtained may or maynot be optimal solution; but it is the best solution obtained.
- → Instead of finding optimal sol^n to subproblems and then making choice greedy algorithm, first make choice — the choice that looks best at the time and then solve a resulting subproblem.
- → The greedy method is also applied to optimization problems.
- → Greedy algorithm assumes that a local optimum is a part of the global optimum.

The steps for achieving greedy algorithms are:

- **feasible**: Here we check whether it satisfies all the possible problem constraints or not, so as to obtain atleast one sol^n to our problem.
- **Local optimal choice**: In this, choice should be the optimum which is selected from the current available feasible choices.
- **unalterable**: once the choice is made, at any subsequent step that choice is not altered.

Greedy (i) may or may not give optimal sol^n (ii) Top-down design (iii) In greedy we find a feasible sol^n but not in DP- (iv) In DP we make a choice at each step, but the choice ~~usually~~ depends on the sol^n of subproblems but in GM we make whatever choice seems best at the moment and then solve the subproblem that remain.

NOTE Greedy and DP are methods for solving optimization problems.

Greedy makes choice then solve subproblems but in DP solve subproblems before making the first choice

algorithm Greedy (a, n)
{
solution = ∅ (Initialized the solution)
for i = 1 to n
{
x = select(a);
if feasible (solution, x) then
solution = Union (solution, x);
}
return solution
}

→ The function select an i/p from a[ ] and removes it.
→ **Feasible** is a boolean valued function that determines whether x can be included into solution vector
→ The function Union combines x with the solution and updates the objective function.

Example of Greedy :

- Activity selection problem
- Huffman coding
- Knapsack "
- Spanning tree (e.g. MST :- prim's algo
  - kruskal ")
- Shortest path (e.g. Single source shortest path
  - Dijkstra's algo)

feasible solⁿ : Suppose we have a problem with n-inputs that require a solⁿ satisfying some constraints. Any subset that satisfy these constraints is called a feasible solution.
→ We need to find a feasible solⁿ that either maximizes or minimizes the given objective function.
→ A feasible solⁿ that does this is called an optimal solⁿ

Activity-selection problem The problem of scheduling several competing activities that require exclusive use of common resource, with the goal of selecting a maximum-size sets of mutually compatible activities.

→ Suppose we have a set $S = \{a_1, a_2, \ldots, a_n\}$ of n proposed activities that wish to use a resource, such as lecture hall, which can be used by only one activity at a time.

→ Each activity a_i has start time s_i and finish time f_i where $\boxed{0 \le s_i < f_i < \infty}$

→ Activities a_i and a_j are compatible if the interval $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap (i.e. a_i and a_j are compatible $a_i \cap a_j = \phi$. So for non interfering if $s_i \geq f_j$ or $s_j \geq f_i$).

→ The activity selection problem is to select maximum-size subset of mutually compatible activities.

NOTE
$[2, 3)$
less than or equal to 2 but less than 3.

Ex: Consider following set S

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

→ For this example, the subset $\{a_3, a_9, a_{11}\}$ consists of mutually compatible activities.

→ But it is not maximal subset subset

→ The subset $\{a_1, a_4, a_8, a_{11}\}$ is largest subsets of mutually compatible activities.

→ another is $\{a_2, a_4, a_9, a_{11}\}$.

Greedy choice property — a locally optimal choice is globally optimal

But - DP make decision based on all the decisions made in the previous stage and may reconsider the previous stage's algorithm path to soln.

But - Greedy never reconsider its choices.

Scanned by CamScanner

Greedy-activity-selector algorithm It assumes that the input activities are ordered by monotonically increasing finish time.

```
GREEDY-ACTIVITY-SELECTOR (s, f)
{
  n = length[s];
  A = {a1}
  i = 1
  for m <- 2 to n
      do if s_m >= f_i
           then A <- A U {a_m}
                i <- m
  return A.
```

→ It collects selected activities into a set A and returns this set when it is done.

→ Since the activities are considered in order of monotonically increasing finish time, f_i is always the maximum finish time of any activity in A. That is

$$\boxed{f_i = \max \{ f_k : a_k \in A \}}$$

Time complexity Above algorithm schedules a set of n-activities in $\Theta(n)$ time, assuming that the activity were already initially by their finish time.

If the activity is not sorted then it takes

$$T(n) = \Theta(n \log n) + O(n)$$

any sorting also (merge, heap, quick sort)

Recursive greedy algo — This algorithm purely greedy, top-down fashion. We give here recursive solⁿ for activity selector. It returns maximum-size set of mutually compatible activities in S_{ij}. We assume that the n-input activities are ordered by monotonically increasing finish time.

REC-ACT-SEL (S, f, i, j)

1. m ← i+1
2. while m < j and S_m < f_i // find the first activity in S_{ij}
3. do m ← m+1
4. if m < j
5. then return $\{a_m\}$ ∪ REC-ACT-SEL (S, f, m, j)
6. else return φ.

—o—

→ The while loop of line 2-3 look for the first activity in S_{ij} which can be included.

→ The loop examine $a_{i+1}, a_{i+2}, \ldots a_{j-1}$, until it finds the first activity a_m that is compatible with a_i; such an activity has $\boxed{S_m \geq f_i}$

→ loop terminates when it finds an activity.

→ Assuming the activities are sorted by finish time this algorithm takes $\boxed{\Theta(n) \text{ time}}$ because each activity is examined exactly once in the while loop test of line-2.

—o—

→ Assume a_0 activity finishes at time 0. then the initial call REC-ACT-SEL(S, f, 0, 12) is made then a_1 is selected.

→ In each recursive call, the activities that have already been selected are shaded and activity shown in white is being considered.

→ The arrow point directly up or to the right.

→ Up to the right arrow is selected.

Knapsack problem: We are given a set S of n-items and a knapsack or bag. Each item has a +ve benefit p_i and a +ve weight w_i. The maximum capacity of the knapsack is m.

If a fraction x_i, $0 \le x_i \le 1$, of object i is placed into the knapsack then a profit of $p_i x_i$ is earned.

→ The objective is to obtain a filling of the knapsack that maximizes the total profit earned.

→ Since the knapsack capacity is m, we require the total weight of all choosen object to be atmost m.

Formally the problem can be defined as

$$\text{maximize} \sum_{1 \le i \le n} p_i x_i \quad \text{—— (1)}$$

$$\text{Subject to} \quad \boxed{\sum_{1 \le i \le n} w_i x_i \le m \quad \text{—— (2)}}$$

$$\text{and} \quad 0 \le x_i \le 1, \quad 1 \le i \le n \quad \text{—— (3)}$$

for 0-1 knapsack $x_i \in \{0, 1\}$

→ The profits and weights are +ve numbers.

→ The feasible solution is anyset $(x_1, x_2, \dots x_n)$ satisfying above condition.

→ An optimal solution is a feasible soln for which eq(1) is maximized.

→ we have to pack the knapsack (or bag) in such a manner as to get the maximum total value.

The knapsack problem has two variants:

0/1 knapsack

→ In this items are individual; either we take full one item or discard it.

→ This problem solved using DP not by GP.

Fractional knapsack: we take any fraction of an item. Solvable by GP.

→ 0-1 knapsack restricts the number x_j to be zero or one.

→ Greedy strategy solve the problem by putting items into bag one-by-one.

→ This approach is greedy because once an item has put into the bag it is never removed.

Q) Consider the following instances of the knapsack problem

$n = 3$, $m = 20$ $(P_1, P_2, P_3) = (25, 24, 15)$ and

$(w_1, w_2, w_3) = (18, 15, 10)$

	x_1	x_2	x_3	$\sum w_i x_i$	$\sum P_i x_i$
1)	1	$\frac{1}{2}$	0	$18\times1 + 15\times\frac{1}{2} + 10\times0 = 25.5$ (not feasible) becoz > 20	
2)	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$	$18\times\frac{1}{2} + 15\times\frac{1}{3} + 10\times\frac{1}{4} = 16.5$ (feasible sol^n)	$12.5 + 8 + 3.5 = 24.25$
3)	1	$\frac{2}{15}$	0	$18\times1 + 15\times\frac{2}{15} + 0\times10 = 20$	$25 + 24\times\frac{2}{15} + 0\times15 = 28.2$
4)	0	$2/3$	1	$18\times0 + \frac{2}{3}\times15 + 1\times10 = 20$	$25\times0 + \frac{2}{3}\times24 + 15\times1 = 16 + 15 = 31$
5)	0	1	$\frac{1}{2}$	$0\times18 + 15\times1 + 1\times10 = 20$	31.5

→ Out of these four feasible solution, solution 4 (i.e last) yields the maximum profit.

The greedy strategy does not work for 0-1 knapsack problem.

Ex The greedy strategy does not work for 0-1 knapsack problem. Knapsack can hold 50 pounds.

Suppose there are 3 items and knapsack can hold 50 pounds.

$\omega_1 = 10$ ← pounds, $P_1 = 60$ dollars

$\omega_2 = 20$ $P_2 = 100$ "

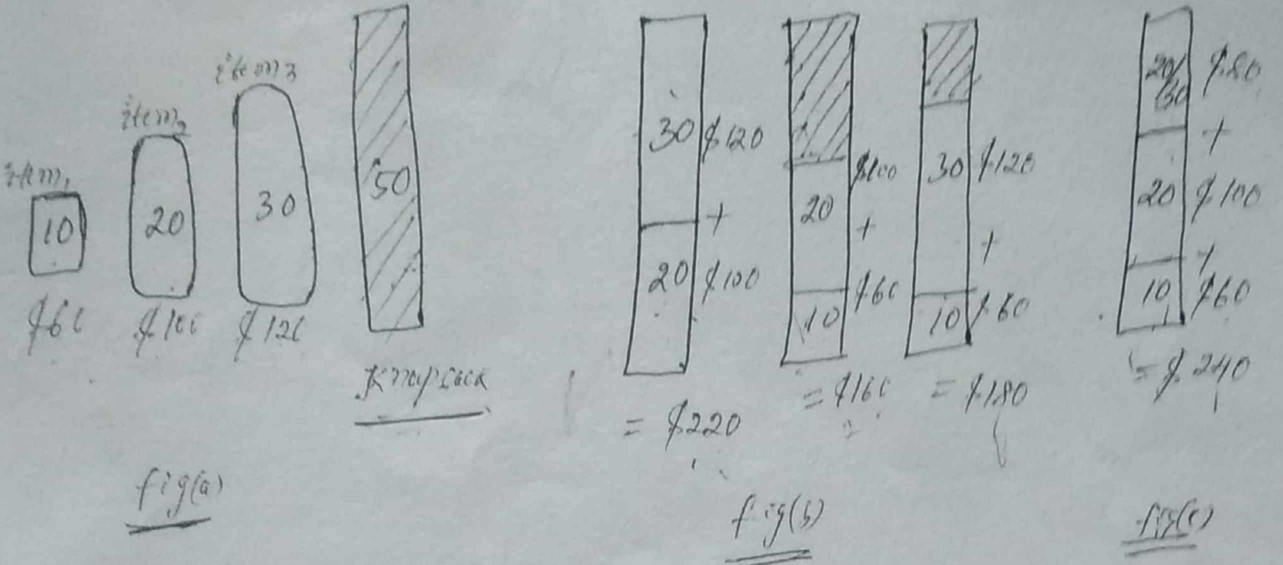
$\omega_3 = 30$ $P_3 = 120$ "

→ Thus the value per pound of item 1 is 6 dollars per pounds which is greater than item 2 (5 dollars / pounds) or item 3 (4 d/p)

→ The greedy strategy therefore would take item 1 first.

→ However optimal solution takes items 2 and 3 leaving behind 1.

→ The two possible sol^n that involve item 1 are both suboptimal.



→ The optimal Subset includes items 2 and 3.
→ any soln with item 1 is suboptimal. even though item 1 has the greatest value per pound.
→ fig(c) shows the fraction knapsack taking the items in order of greatest value per pound yields an optimal solution.

The methods used to obtain the solution is termed as greedy method because at each step we choose an object which would increase the objective function value the most. However greedy method does not yield an optimal solution.

→ To get the optimal soln ; at each step we include the object which has the maximum profit / unit capacity used.
→ This means objects are arranged by P_i/W_i order.

How do we select the next item to be put into the knapsack?

There are general possibilities:-

- Greedy by profit: At each step select from the remaining items with the highest profit. This approach tries to maximize the profit by choosing the most profitable items first.
- Greedy by weight: with one with least weight - it tries to maximize the profit by putting as many items into the knapsack as possible.
- Greedy by profit density: with the largest profit density, P_i/W_i. it tries to maximize the profit by choosing items with the largest profit per units of weight.

Example shown in next paper

Huffman codes : are widely used and very effective technique (→ bottom up manner) for compressing data, depending on the characterstics of data being compressed.

→ Huffman's greedy algorithm uses a table of the frequencies of occurrence of the characters to build up an optimal way of representing each character as a binary string.

→ used for security purpose where data is ~~transferred~~ from sender to receiver.

→ There are many ways to represent such a file of information.

Suppose we have a 100,000-character data file that we wish to store compactly. only six-characters appear and the character 'a' occurs 45,000 times. so ~~Hacked~~ OR

→ Hoffman coding is a lossless data compression also.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length code word	000	001	010	011	100	101
Variable-length Codeword	0	101	100	111	1101	1100

→ A character-coding problem.

→ A datafile of 100,000 characters contains only the character a-f

→ If each character is assigned 3-bit codeword, the file can be encoded in $\boxed{300,000 \text{ bits}}$

→ using the variable-length code shown. the file can be encoded in $\boxed{224,000 \text{ bits}}$

It is of two types :-
- Fixed length Coding
- variable length ,,

→ If we use fixed-length code we need 3 bits to represent six characters. This requires 300,000 bits.

→ A variable length code can do considerably better than a fixed length code. This coding requires

$$(45\cdot1+13\cdot3+12\cdot3+16\cdot3+9\cdot4+5\cdot4)\,1000 = 224,000 \text{ bits}$$

→ So it saves 25%.

prefix codes are widely used in applications that ~~compress~~ data including JPEG for images and MP3 for music.

Prefix codes (No codeword is a prefix of some other codeword. (prefix property)
Such codes are called prefix code (or prefix free codes).)

→ The standard soln for unique decoding is to insist that the code-word be prefix free.

→ This means that if $a, b \in \Sigma$, $a \neq b$ then code word for 'a' is not a prefix of the code-word for 'b' viceversa.

→ Encoding is always simple for any binary character code, we just concatenate the codewords representing each character of the file.

→ Suppose we code 3-character file abc as 0·101·100 = 0101100

→ prefix codes are desirable because they simplify decoding.

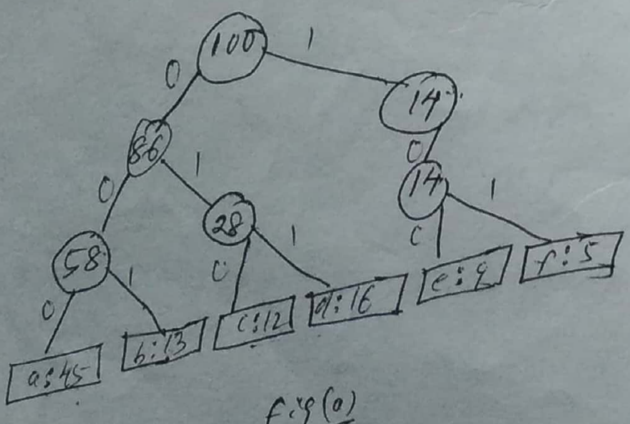
→ Since no codeword is a prefix of any other, the codeword that begin on encoded file is unambiguous.

→ we can simply identify the initial codeword, translate it back to the original character and repeat the decoding process on the remainder of the encoded file.

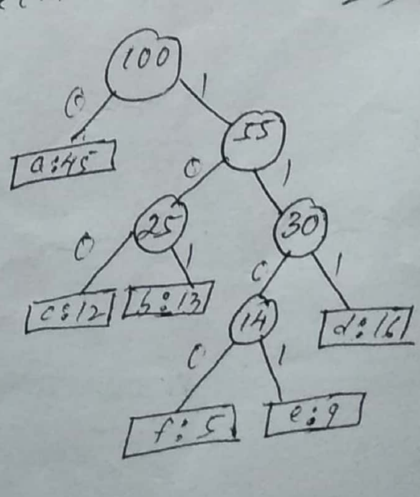
→ in our example the string 001011101 parses uniquely as 0·0·101·1101 which decodes to aabe.

The decoding process needs a convenient representation for the prefix code so that the initial codeword can be easily picked off.

→ A binary tree whose leaves are the given characters provides one such representation (0 means left child and 1 means right)



Fig(a)



Fig(b)

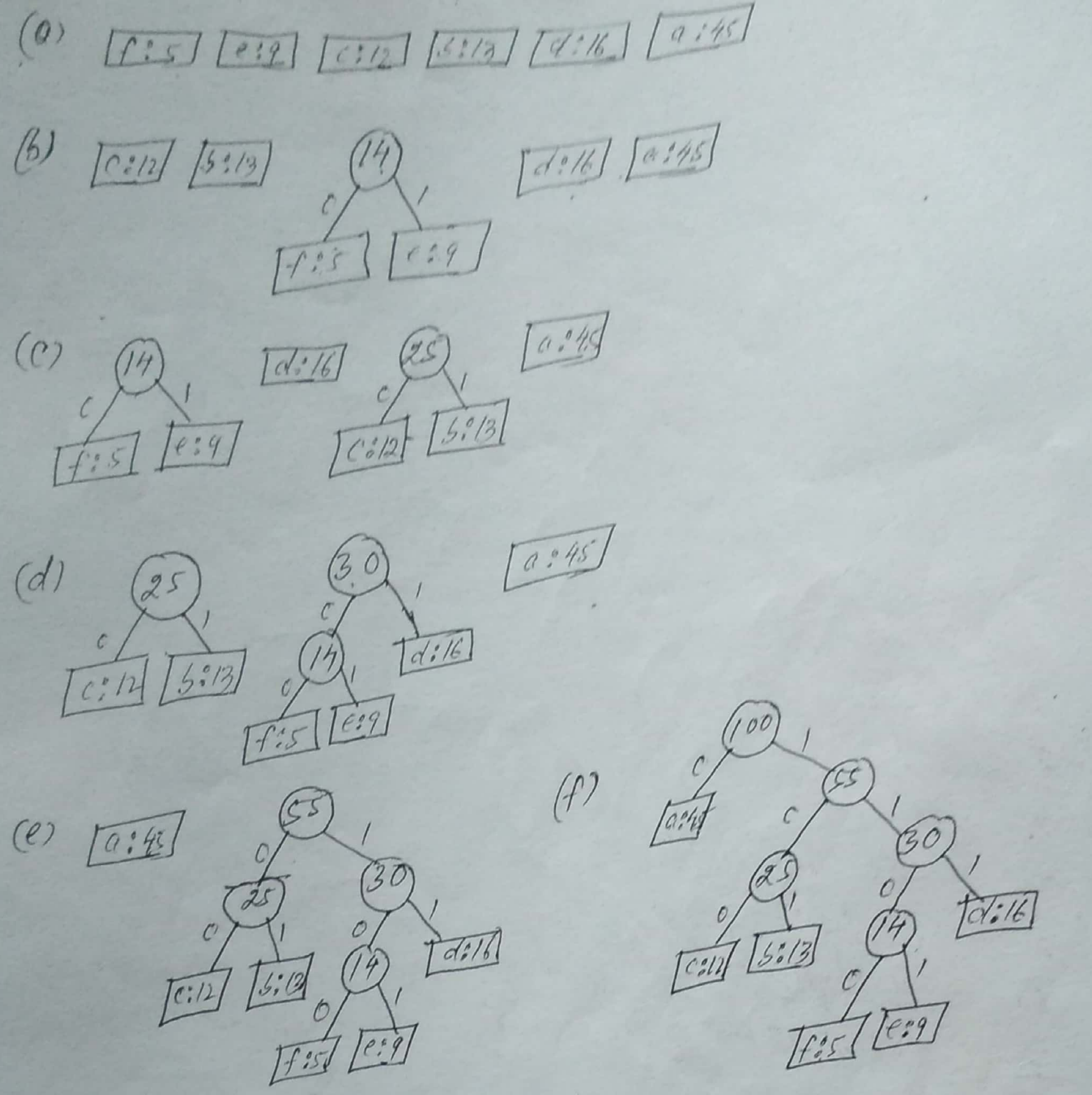
Constructing a Huffman code :- Huffman invented a greedy algorithm that constructs an optimal prefix code called a **Huffman code**.

- → C is a set of n characters and that each character $c \in C$ is an object with a defined frequency $f[c]$.
- → The algorithm ~~builds the~~ tree T corresponding to the optimal code in bottom-up manner.
- → It begins with a set of $|C|$ leaves and performs a sequence of $|C|-1$ "merging" operations to create the final tree.
- → A min-priority queue Q, keyed on f, is used to identify the two least-frequent objects to merge together.

```
HUFFMAN(C)
1. n ← |C|
2. Q ← C                              ———— O(n)
3. for i ← 1 to n-1
4.     do allocate a new node z
5.        left[z] ← x ← EXTRACT-MIN(Q)
6.        right[z] ← y ← EXTRACT-MIN(Q)
7.        f[z] ← f[x] + f[y]
8.        INSERT(Q, z)
9. return EXTRACT-MIN(Q)   // returns the root of the tree.
```

3-8 executed exactly n-1 times. and each heap opn require $O(\lg n)$ so total $O(n \lg n)$.

- → Since there are 6 letters in the alphabet, the initial queue size is n=6. and 5 merge steps are required to build tree.
- → The final tree represents the optimal prefix code.
- → The codeword for a letter is the sequence of edge labels on the path from the root to the letter.
- →



→ leaves are shown as rectangle.
→ Internal node as circle

Ex Build a binary tree using Huffman code for P: <29, 25, 20, 12, 5, 09>

Time complexity The analysis of the running time of Huffman's algo assumes that Q is implemented as a binary min-heap.

→ For a set C of n characters the initialization of Q in line 2 takes O(n) time using the BUILD-MIN-HEAP procedure.

→ The for loop in lines 3-8 is executed exactly n-1 times and since each heap operation requires time O(lg n), the loop contributes $\boxed{O(n \lg n)}$ to the running time.

Scanned by CamScanner

Dynamic programming

Dynamic programming typically applies to optimization problems in which a set of choices must be made in order to arrive at an optimal solution.

It is generally used with optimization problem where sequence of solutions are available, each solution has a value, we require to find the optimal value, each solution is called an optimal solution (one way to solve optimization problem to try all possible soln and then pick out the best).

It is similar to D And C method, which solves problems by combining the solution to subproblems. ~~But~~ D And C algorithm partition the problem into independent subproblems, solve subproblems recursively and then combine their solutions to solve the original problem. But in DP, the subproblems are not independent.

→ DP solves every subproblem once and then save its answer in a table, there by avoiding the work of recomputing the answer every time the subproblem is encountered. (Save in table at the cost of space)

DAC	DP
(1) Each part is independent	(1) Each part is dependent
(2) There is no need of table	(2) Need of table
(3) Each part soln saving is not required	(3) Required
(4) Time complexity is more	(4) Less
(5) we may or may not find optimal solution	(5) we must find the optimal solution guaranteed to find solution.
(6) subproblems of DAP are nonoverlap	(6) overlap
(7) DAC solves the sub-problem top-down	(7) DP is bottom-up technique.

NOTE (1) Dp solves the subproblems once and then save answer in a table but D and C call repeatedly the subproblems in many times so complexity is more.

(2) in DP we try to find redundancies and reduce the space for searching

→ DP is applicable when the subproblems are not independent. that is when subproblems share subsubproblems. So
→ D&C does more work than necessary (repeatedly solving the common subproblems).

The development of a DP algorithm can be broken into a sequence of four steps.

1. characterizing the structure of an optimal solution.
2. Recursively define the value of an optimal solution
3. Compute the optimal solution in a bottom up fashion.
4. Construct an optimal solution from computed information

bottom-up means:

(1) Start with the smallest subproblems.
(2) combining their soln obtain the soln to subproblems of increasing size.
(3) until arrive at the soln of the original solution.

Try to solve solve subproblem first and use their soln to build-on and arrive at soln to bigger subproblems.

EXTRA (1) The word "programming" in DP has no connection to computer program and instead come from the term "mathematical programming" - a synonym for optimization.

(2) DP is mainly used to tackle optimization problems that are solvable in polynomial time.

Matrix-chain Multiplication is example of DP.

We are given a sequence (chain) $\langle A_1, A_2, \cdots, A_n \rangle$ of n-matrices to be multiplied and we wish to compute the product $A_1 A_2 \cdots A_n$.
we can evaluate the expⁿ once we have parenthesized it to resolve all ambiguity in how the matrices are multiplied together.
→ matrix multiplication is associative and so all parenthesizations yields the same product.

Ex:- If the chain of matrices is $\langle A_1\ A_2\ A_3\ A_4 \rangle$, the product $A_1 A_2 A_3 A_4$ can be fully parenthesized in 5 distinct ways:

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

→ The way we parenthesize a chain of metrices can have a dramatic impact on the cost of evaluating the product.

Example:- Suppose the dimensions of matrices of $\langle A_1, A_2, A_3 \rangle$ are
10×100, 100×5 and 5×50

i.e $(A_1)_{10 \times 100}$ $(A_2)_{100 \times 5}$ $(A_3)_{5 \times 50}$

We can multiply the above chain of metrices according to the parenthesizations.

(i) $(A_1 A_2) A_3$ or
(ii) $A_1 (A_2 A_3)$

$(A_1 \times A_2) = (A)_{10 \times 5}$ No. of scalar multiplications $10 \times 100 \times 5 = 5000$
$(A \times A_3) = (B)_{10 \times 50}$ No " " $10 \times 5 \times 50 = 2500$

total = 7500 op need for $(A_1 A_2) A_3$

If we multiply according to (ii) then we have

$(A_2 A_3) = (C)_{100 \times 50}$ No. of scalar multiplications $100 \times 5 \times 50 = 25000$
$(A_1 \times C) = (D)_{10 \times 50}$ " " " $10 \times 100 \times 50 = 50,000$

75 000 op

→ Thus computing the product according to first parenthesization is 10 times faster

→ Note that in the matrix-chain multiplication (MCM), we are not actually multiplying matrices.

→ Our goal is only to determine an order for multiplying matrices that has the lowest cost.

MCM can be stated as follows: given a chain $(A_1, A_2, \ldots, A_n)$ of n matrices where for $i = 1, 2, \ldots, n$ 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 minimize the no of scalar multiplication

→ Any parenthesization of the product $A_i A_{i+1} \ldots A_j$ must split the product between A_k and A_{k+1} for some constant k, $i \leq k < j$.

→ The cost of the parenthesization is thus the cost of computing the matrix $A_{i \ldots k}$ + cost of computing $A_{k+1, \ldots j}$ plus(+) the cost of multiplying them together

→ Let us define the cost of an optimal solution recursively in terms of the optimal solutions to subproblems. For MCM we pick as our sub problems the problems of determining the minimum cost of a parenthesization of $A_i A_{i+1} \ldots A_j$ for $1 \leq i \leq j \leq n$.

→ Let $m[i,j]$ = minimum no of scalar multiplications needed to compute the matrix $A_{i \ldots j}$;

i.e $(A_i A_{i+1} \ldots A_j) = (A_i A_{i+1} \ldots A_k)_{P_{i-1} \times P_k} (A_{k+1} \ldots A_j)_{P_k \times P_j}$

i.e $A_{i \ldots j} = (A_{i \ldots k})_{P_{i-1} \times P_k} (A_{k+1 \ldots j})_{P_k \times P_j}$

→ If $i = j$, the problem is trivial so $m[i, i] = 0$.

$$\boxed{m[i,j] = \underbrace{m[i,k]}_{\text{cost of computing subproduct } A_{i \ldots k}} + \underbrace{m[k+1, j]}_{\text{cost of computing } A_{k+1 \ldots j}} + \underbrace{P_{i-1} P_k P_j}_{\text{cost of computing } (A_{i \ldots k})(A_{k+1 \ldots j})}}$$

→ This recursive eqn assumes that we know the value of k, which we do not. There are only $j-i$ possible values for k.

$$\boxed{k = i, i+1, \ldots, j-1}$$

→ To find the best we use one of these values for k.

$$m[3,6] = \min \begin{cases} m[3,3] + m[4,6] + P_2 P_3 P_6 \\ m[3,4] + m[5,6] + P_2 P_4 P_6 \\ m[3,5] + m[6,6] + P_2 P_5 P_6 \end{cases}$$

$$= \min \begin{cases} 0 + 3500 + 15 \times 5 \times 25 = 5375 \\ 750 + 5000 + 15 \times 10 \times 25 = 9500 \\ 2500 + 0 + 15 \times 20 + 25 = 10000 \end{cases}$$

$s[3,6] = 3$

$$m[1,5] = \min \begin{cases} m[1,1] + m[2,5] + P_0 P_1 P_5 \\ m[1,2] + m[3,5] + P_0 P_2 P_5 \\ m[1,3] + m[4,5] + P_0 P_3 P_5 \\ m[1,4] + m[5,5] + P_0 P_4 P_5 \end{cases}$$

$$= \min \begin{cases} 0 + 7125 + 30 \times 35 \times 20 = 28125 \\ 15750 + 2500 + 30 \times 15 \times 20 = 27250 \\ 7875 + 1000 + 30 \times 5 \times 20 = 11875 \\ 9375 + 0 + 30 \times 10 \times 20 = 15375 \end{cases}$$

$s[1,5] = 3$

$$m[2,6] = \min \begin{cases} m[2,2] + m[3,6] + P_1 P_2 P_6 \\ m[2,3] + m[4,6] + P_1 P_3 P_6 \\ m[2,4] + m[5,6] + P_1 P_4 P_6 \\ m[2,5] + m[6,6] + P_1 P_5 P_6 \end{cases}$$

$$= \min \begin{cases} 0 + 15375 + 35 \times 15 \times 25 = 18503 \\ 2625 + 3500 + 35 \times 5 \times 25 = 10500 \\ 4375 + 5000 + 35 \times 10 \times 25 = 18125 \\ 7125 + 0 + 35 \times 20 \times 25 = 24625 \end{cases}$$

$s[2,6] = 3$

$$m[1,6] = \min \begin{cases} m[1,1] + m[2,6] + P_0 P_1 P_6 \\ m[1,2] + m[3,6] + P_0 P_2 P_6 \\ m[1,3] + m[4,6] + P_0 P_3 P_6 \\ m[1,4] + m[5,6] + P_0 P_4 P_6 \\ m[1,5] + m[6,6] + P_0 P_5 P_6 \end{cases}$$

$$= \min \begin{cases} 0 + 10500 + 30 \times 35 \times 25 = 36750 \\ 15750 + 5375 + 30 \times 15 \times 25 = 32375 \\ 7875 + 3500 + 30 \times 5 \times 25 = 15125 \\ 9375 + 5000 + 30 \times 10 \times 25 = 21875 \\ 11875 + 0 + 30 \times 20 + 25 = 26875 \end{cases}$$

$\therefore s[1,6] = 3$

Although MATRIX-CHAIN-ORDER determines the optimal no. of scalar multiplications needed to compute a matrix-chain product, it does not directly show how to multiply the matrices. It is not difficult to construct an optimal solution from the computed information stored in the table $s[1..n, 1..n]$.

The following recursive procedure prints an optimal parenthesization of $\langle A_i, A_{i+1}, \ldots, A_j \rangle$.

PRINT-OPTIMAL-PARENS (s, i, j)

```
if i = j
   then print "A_i"
   else print "("
        PRINT-OPTIMAL-PARENS(s, i, s[i,j])
        PRINT-OPTIMAL-PARENS(s, s[i,j]+1, j)
        print ")"
```

$i=1$, $j=6$ initially

→ s, 1, [1,6] value [1 3]

→ (s, [3+1], 6)

Now $i=1$, $j=3$

Now [1,1] (A_1)

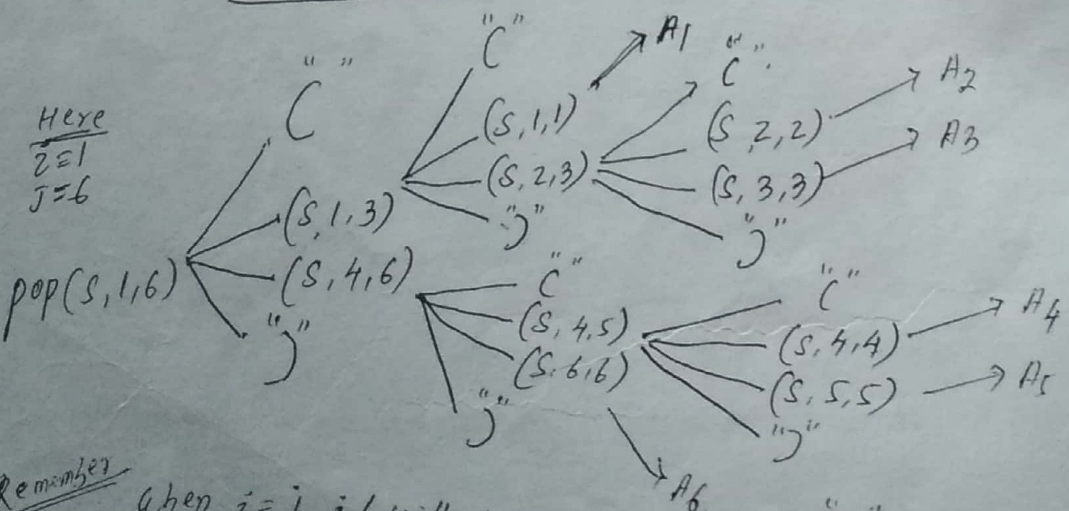
s[i,j] s[2,3]

(s = 2) (A_2)

In the above example the call PRINT-OPTIMAL-PARENS(s, prints the parenthesization

A[1,3]

$$\boxed{((A_1(A_2A_3))((A_4A_5)A_6))}$$



Remember

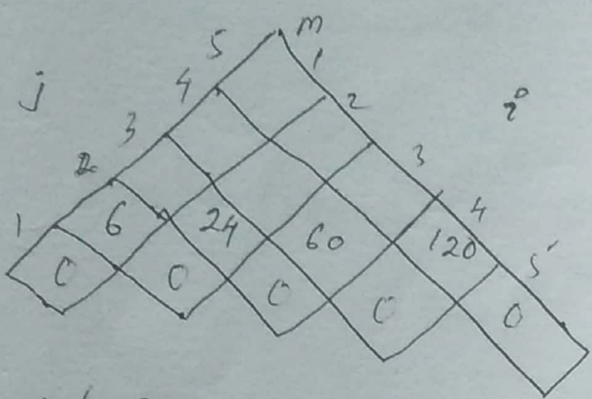
When $i=j$, it will print the matrix "A_i"

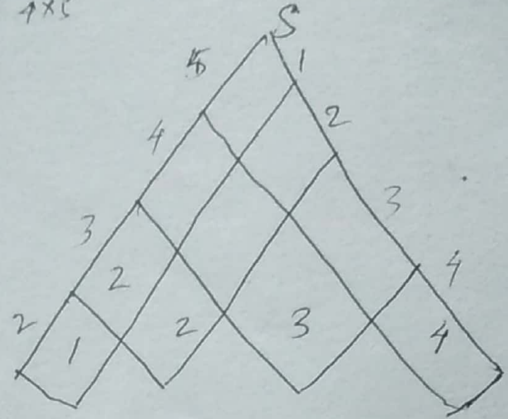
when $i \neq j$ the problem follows 4 steps

- print "("
- pop(s, i, s[i,j]
- pop(s, s[i,j]+1, j)

Ex $(A_1)_{1\times 2}$ $(A_2)_{2\times 3}$ $(A_3)_{3\times 4}$ $(A_4)_{4\times 5}$ $(A_5)_{5\times 6}$

$A_1 \rightarrow A_5$





put $l=2$, $i \rightarrow 1$ to 4 , for $i=1, j=2$ $m[1,2], m[2,3]$ $m[3,4]$ $m[4,5]$

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

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

for $k=3$; $m[3,4] = m[3,3] + m[4,4] + P_2 P_3 P_4$

for $k=4$; $m[4,5] = m[4,4] + m[5,5] + P_3 P_4 P_5$

For $l=3$ $i \rightarrow 1$ to 3 $J=3$ $m[1,3]$, $m[2,4]$, $m[3,5]$

for $k=1$ $m[1,3] = m[1,1] + m[2,3] + P_0 P_1 P_3 = 32$

for $k=2$ $m[1,3] = m[1,2] + m[3,3] + P_0 P_2 P_3 = 6+0+12 = 18$

Ex:- find the optimal parenthesis of 5, 10, 12, 2, 15, 16, 11

Ans :- $A_1 = 5\times 10$ $A_3 = 12\times 2$ $A_5 = 15\times 16$

$A_2 = 10\times 12$ $A_4 = 2\times 15$ $A_6 = 16\times 11$

P_0 P_1 P_2 P_3 P_4 P_5 P_6 are given in equation

Ex:- $A_1 = 2\times 3$ $P_0 = 2$ find (i) $m[1,4]$

$A_2 = 3\times 4$ $P_1 = 3$ (ii) $m[1,3]$

$A_3 = 4\times 5$ $P_2 = 4$ (iii) draw the parenthesis

$A_4 = 5\times 6$ $P_3 = 5$

$P_4 = 6$

$m[1,2] = 24$ $k=1$ $m[1,3] = 64$ $(k=2)$

$m[2,3] = 60$ $(k=2)$ $m[2,4] = 150$ $(k=3)$

$m[3,4] = 120$ $(k=3)$ $m[1,4] = 124$ $(k=3)$

optimal $= (((A_1 A_2) A_3) A_4)$

* Longest Common Subsequence :-

→ A subsequence of a given sequence is just the given sequence with zero or more elements left out.

→ A subsequence is any subset of the elements of a sequence that maintains the same relative order.

→ Given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, another sequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is a <u>subsequence</u> of X if there exists a strictly increasing order sequence $\langle i_1, i_2, \ldots, i_k \rangle$ of indices of X such that for all $j = 1, 2, \ldots, k$, we have

$$\boxed{x_{i_j} = z_j}$$

Ex:- $Z = \langle B, C, D, B \rangle$ is a subsequence of $X = \langle A, B, C, B, D, A, B \rangle$ with corresponding index subsequence $\langle 2, 3, 5, 7 \rangle$

→ Given two sequences X and Y, we say that a sequence Z is a common subsequence of X and Y if Z is subsequence of both X and Y.

Ex:- if $X = \langle A, B, C, B, D, A, B \rangle$

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

→ For position A of X
(i) <u>A B A</u> is a common subsequence of X Y

→ For position B of X
(ii) <u>B C A B</u> is a " " " "

→ For position C of X
(iii) <u>C A B</u> " " " "

→ For position D of X
(iv) <u>D A B</u> " " " "

→ For position A of X
(v) <u>A B</u> " " " "

→ For position B
(vi) B " " " "

→ So the longest common subsequence (LCS) is BCBA, also BDAB, ...

→ In LCS we are given two subsequences and wish to find a maximum-length common subsequence of X and Y

→ One way to solve the LCS problem is to enumerate all subsequences of X and take the longest one that is also a subsequence of Y.

Several motivation applications

(1) molecular biology

(2) file comparison

(3) Screen display

When we look for a DP sol^n to a problem, we examine the two key ingredients that an optimization problem must have in order for DP to apply.

Elements of Dynamic programming

The problem has the following properties:

I. optimal Substructure: A sol^n is optimal only if its ~~sub problems~~ sub solutions to the subproblem is optimal. (optimal sol^n to the problem contains within it optimal sol^n to subproblems)

[2] or optimal sol^n to the problem ...

(2) overlapping sub problems:

- The same subproblem is visited over and over again.
- The # of distinct subproblem is polynomial.

OR A problem is said to have overlapping sub-problems if the problem can be broken down into subproblems which are re-used several times or a recursive algo for the problem solves the same subproblem over and over rather than always generating new subproblems

OR

when a recursive algo revisits the same problems repeatedly we say that the optimization problem has overlapping subproblems

Algorithms
1. Characterizing a LCS:- A brute-force approach to solving the LCS is to enumerate all subsequences of X and check each subsequence to see if it is also a subsequence of Y, keeping track of the LS found.

The LCS problem has an optimal-substructure property

Optimal substructure of an LCS

Let $X = \langle x_1, x_2, \cdots, x_m \rangle$ and $Y = \langle y_1, y_2, \cdots, y_n \rangle$ be sequences and let $Z = \langle z_1, z_2, \cdots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and 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}

2. Recursive solution previous section implies that if $x_m = y_n$ we must find an LCS of X_{m-1} and Y_{n-1}. Appending $x_m = y_n$ to this LCS yields an LCS of X and Y.
If $x_m \neq y_n$ the we must solve two subproblems:

- finding an LCS of X_{m-1} and Y
- " " " X and Y_{n-1}

Whichever of these two LCS's is longer is an LCS of X and Y.

- Let $c[i,j]$ length of an LCS of the sequences X_i and Y_j.

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

Example: Let $X = \langle x_1, x_2, \cdots, x_n \rangle$
$Y = \langle \quad \rangle$ no elements.

So length of X $= L(X) = n$
" " Y $= L(Y) = 0$

// Brute force approach

Algorithm to build the LCS: (computing the length of LCS)

LCS-length (X, Y)

```
m ← length[X]
n ← length[Y]
for i ← 1 to m          } O(m)
    do c[i, 0] ← 0
for j ← 0 to n          } O(n)
    do c[0, j] ← 0
for i ← 1 to m          } O(m*n)
    do for j ← 1 to n
        do if x_i = y_j
            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
```

Find LCS for $X = \langle A, B, C, B, D, A, B \rangle$
$Y = \langle B, D, C, A, B, A \rangle$

It stores the $c[i, j]$ value in a table $c[0..m, 0..n]$ whose entries are computed in row-major order.
→ It also maintains the table $b[1..m, 1..n]$ to simplify construction of an optimal solution.

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

B C B A.

The running time of the procedure is $\boxed{O(mn)}$, since each table entry $O(1)$ time to compute. [or $\boxed{T(n) = O(m)+O(n)+O(m*n)}$]

Constructing an LCS

The b table returned by LCS-LENGTH can be used to quickly construct an LCS of X and Y.

→ We simply begin at $b[m, n]$ and trace through the table following the arrows

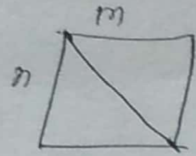
→ Whenever we encounter a "↖" in entry $b[i, j]$ it implies that $x_i = y_j$ is an element of the LCS.

→ The elements of ~~procedure~~ the LCS are encountered in reverse order by this method.

PRINT-LCS (b, X, i, j)

[length[Y]] for find the optimal value

```
if i = 0 or j = 0
    then return
if b[i, j] ← "↖"
    then PRINT-LCS (b, X, i-1, j-1)
         print x_i
else if b[i, j] = "↑"
    then PRINT-LCS (b, X, i-1, j)
else PRINT-LCS (b, X, i, j-1)
```



→ This procedure prints "B C B A".

→ This procedure takes time $\boxed{O(m+n)}$, since atleast one of i and j is decremented in each stage of the recursion.

EXERCISE Determine an LCS of $A = \{1, 0, 0, 1, 0, 1, 0, 1\}$ $m = 8$

$B = \{0, 1, 0, 1, 1, 0, 1, 1, 0\}$ $n = 9$

Minimum Spanning trees

Suppose $G=(V,E)$ is a connected undirected graph and for each edge $(u,v) \in E$, we have a weight $w(u,v)$ specifying the cost to connect u and v.

→ we then wish to find an acyclic subset $T \subseteq E$ that connects all of the vertices and whose total weight

$$\boxed{w(T) = \sum_{(u,v)\in T} w(u,v)}$$ is minimized.

→ Since T is acyclic and connects all of the vertices, it must form a tree, which we call a spanning tree, since it spans the graph G. we call the problem of determining the tree T the minimum-spanning-tree problem.

OR

→ Let a graph $G=(V,E)$ if T is a sub-graph and contains all the vertices but no cycle then T is said to be a spanning tree.

→ A connected graph is a tree if and only if it has n-vertices and n-1 edges.

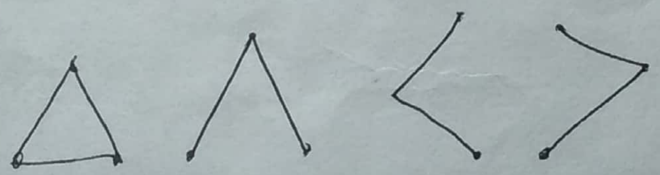
OR

Spanning tree: if G is a graph, H is the spanning tree of the graph G, if

(i) H is the subgraph of G

(ii) H is a tree (connected tree)

(iii) H contains all the vertices of G.

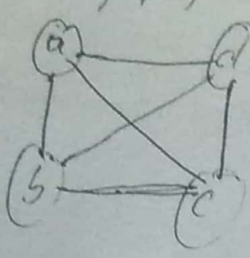


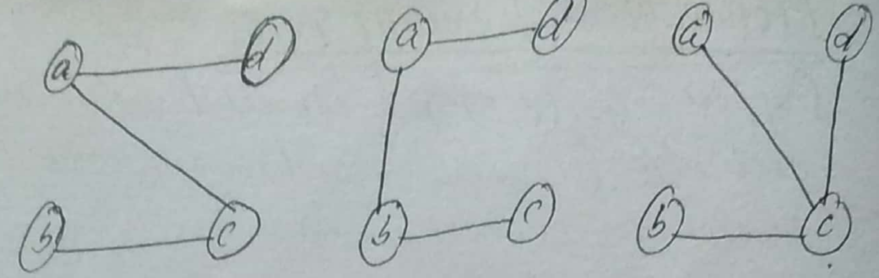
(Spanning tree of G.)

→ If the graph is a complete graph with n-vertices, then it can have n^{n-2} spanning trees

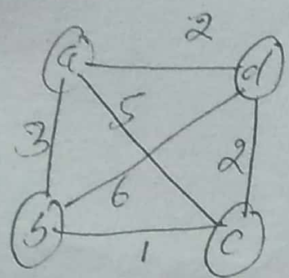
$$\boxed{\text{No. of spanning trees} = n^{n-2}}$$

A graph G

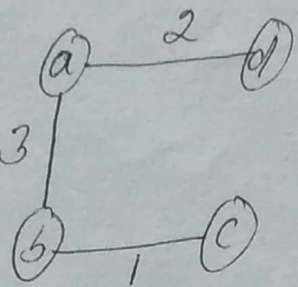


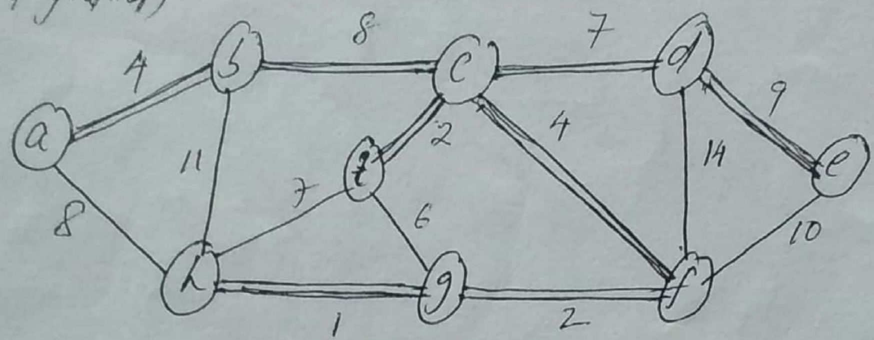


(Tree of many possible spanning tree of G)



(a weighted graph G)





→ The total weight of the tree shown is 37

→ This MST is not unique: removing the edge (b, c) and replacing it with the edge (a, h) yields another tree with weight 37.

Growing a minimum spanning tree: We have a connected, undirected graph $G = (V, E)$ with a weight function $\boxed{w: E \rightarrow R}$ and we wish to find a MST for G.

→ Two algorithm we consider in this use a greedy approach to the problem, although they differ in how they apply this approach.

→ This greedy strategy is captured by the following "generic" algorithm which grows the minimum spanning tree one edge at a time.

GENERIC-MST (G, W)

1. $A \leftarrow \phi$
2. While A does not form a spanning tree
3. do find an edge (u, v) that is safe for A
4. $A \leftarrow A \cup \{(u, v)\}$
5. return A

Cut :- A cut (S, V-S) of an undirected graph $G = (V, E)$ is a partition of V.

Cut vertex :- From the connected graph G when we remove a vertex V from the graph and G-V becomes a graph which is not connected, then the vertex V is called cut vertex.

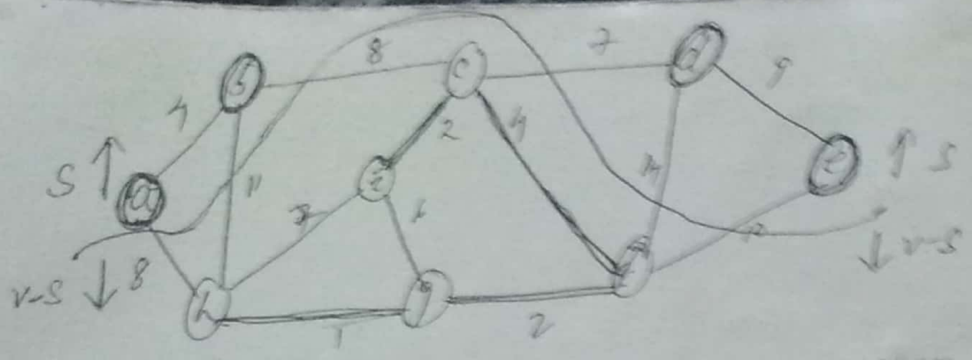
Cut set :- The cutset is a set of edges whose removal from the graph G, makes the graph disconnected then that set of edges are called cutset. (It is minimal no. of edges whose removal from G destroys all paths between these two set of vertices).

Cross edge An edge $(u, v) \in E$ crosses the cut (S, V-S) if one of its endpoints is in S and the other is in V-S.

OR cross edge are the edges which belongs to the cutset.

Light edge :- Among the cross edges, whose weight is minimum is called light edge. It is the minimum weight among the cross edge.

An edge is a light edge crossing a cut if its weight is the minimum of any edge crossing the cut.



→ The vertices in the set S are shown in black and those in V − S are shown in white.

→ The edges crossing the cut are those connecting white vertices with black vertices.

→ The edge (d, c) is the unique light edge crossing the cut.

→ A subset A of the edges is shaded, since no edge of A crosses the cut.

Single-Source Shortest paths:

In a shortest-path problem we are given a weighted, directed graph $G=(V,E)$ with weight function $w: E \to R$ mapping edges to real-valued weights.

→ The weight of path $p = \langle V_0, V_1, V_2, \dots V_k \rangle$ is the sum of the weights of its constituent edges:

$$\boxed{w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)}$$

we define the shortest-path weight from u to v by

$$\delta(u,v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{if there is a path from } u \text{ to } v. \\ \infty & \text{otherwise} \end{cases}$$

→ A shortest path from vertex u to vertex v is then defined as any path p with weight $w(p) = \delta(u,v)$

→ Edge weight used to represent time, cost, penalties, loss or any other quantity.

Variants

(1) Single-source shortest-path: from the given graph we want to find a shortest path from a given source vertex $s \in V$ to each vertex $v \in V$.

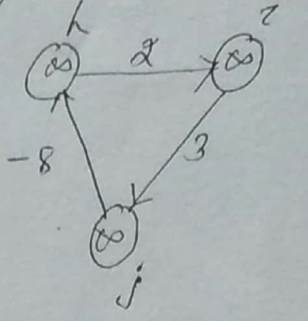
(2) Single-destination shortest-path: Find a shortest path to a given destination vertex t from each vertex v. By reversing the direction of each edge in the graph, we can reduce this problem to a single-source problem.

(3) Single-pair shortest-path problem: Find a shortest path from u to v for given vertices u and v. If we solve the single-source problem with source vertex u we solve this problem also.

(4) All-pairs shortest-path problem: Find a shortest path from u to v for every pair of vertices u and v.

Negative-weight edges In some instances of the single-source shortest path problems, there may be edges whose weights are negative.

Negative-weight cycle: cycle whose sum of weights are -ve.
Here vertices h, i and j also form a -ve weight cycle.

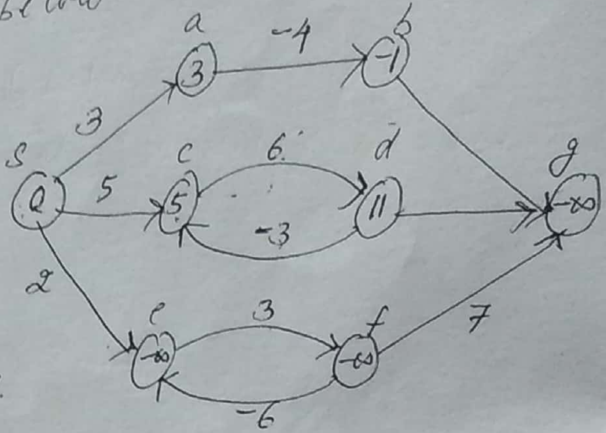


→ They are reachable from source s so their shortest path weights are ∞.

$$\boxed{\delta(s,h) = \delta(s,i) = \delta(s,j) = \infty}$$

even though they lie on -ve weight cycle.

→ Let us discuss the given figure below



→ There is only one path from s to a (the path ⟨s,a⟩),
$\delta(s,a) = w(s,a) = 3$.

→ Similarly there is only one path from s to b and so
$\delta(s,b) = w(s,a) + w(a,b) = 3 + (-4) = -1$.

→ There are infinitely many paths from s to c:
⟨s,c⟩, ⟨s,c,d,c⟩, ⟨s,c,d,c,d,c⟩ and so on.
Because the cycle ⟨c,d,c⟩ has weight $6 + (-3) = \boxed{3 > 0}$, the shortest path from s to c is ⟨s,c⟩ with $\delta(s,c) = 5$.

→ Similarly the shortest path from s to d is ⟨s,c,d⟩ with weight $\delta(s,d) = w(s,c) + w(c,d) = 11$.

→ But there are infinitely many paths from s to e:
⟨s,e⟩, ⟨s,e,f,e⟩, ⟨s,e,f,e,f,e⟩ and so on.
Since the cycle ⟨e,f,e⟩ has weight $3 + (-6) = \boxed{-3 < 0}$, however there is no shortest path from s to e. By traversing the -ve weight cycle ⟨e,f,e⟩ many times we can find path from ⟨s,e⟩

$$\boxed{\delta(s,e) = -\infty} \quad \text{Similarly} \quad \boxed{\delta(s,f) = -\infty}$$

→ vertices h, i, j also form -ve weight cycle. They are not reachable from s so $\delta(s,h) = \delta(s,i) = \delta(s,j) = \infty$.

Scanned by CamScanner

→ If there -ve weight cycle and reachable from source s then
δ(s, that vertex) = $-\infty$.

→ If there is -ve weight cycle but not reachable from s then
δ(s, that vertex) = ∞.

→ Remember :→ $\boxed{\text{Dijkstra's algorithm}}$ assume that all edge weights in the input graph are non-negative. (+ve)

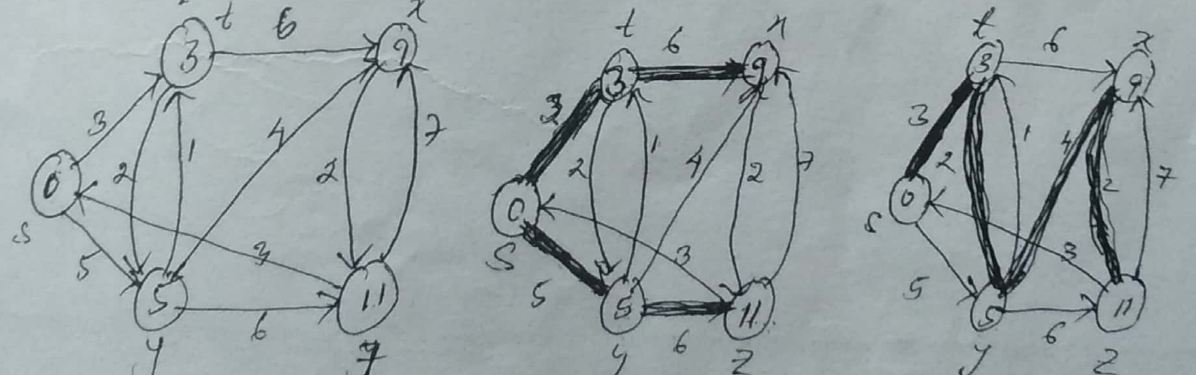
→ $\boxed{\text{Bellman – Ford algorithm}}$ allow negative-weight edges in the input graph and produce a correct answer as long as no -ve weight cycles are reachable from the source.

Shortest-path tree : rooted at s is a directed graph $G' = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E$, such that

1. V' is the set of vertices reachable from s in G.
2. G' forms a rooted tree with root s, and
3. for all $v \in V'$, the unique simple path from s to v in G' is a shortest path from s to v in G.

Shortest paths are not necessarily unique and neither are shortest-paths trees.

For example below diagram shows a weighted, directed graph and two shortest-paths trees with the same root.



(a) (b) (c)

→ fig(b) The shaded edges form a shortest-paths tree rooted at the source s.

→ In fig(c), another shortest paths tree with the same root.

Relaxation:- The process of relaxing an edge (u, v) consists of testing whether we can improve the shortest path to v found so far by going through u and if so, updating $d[v]$ and $\pi(v)$.

→ For each vertex $v \in V$, we maintain an attribute $d[v]$, which is an upper bound on the weight of a shortest path from s to v.

→ We call $d[v]$ a shortest path estimate.

We initialize the shortest-path estimates and predecessors by the following $\Theta(V)$-time procedure.

INITIALIZE-SINGLE-SOURCE (G, s)

```
for each vertex v ∈ V[G]
    do d[v] ← ∞          // distance of vertex is ∞
        π[v] ← NIL        // Parent of v is NIL i.e. each is a child.
d[s] ← 0
```

Here $d[v] = \infty$ for $v \in V - \{s\}$

—o—

→ A relaxation step may decrease the value of the shortest-path estimate $d[v]$ and update v's predecessor field $\pi(v)$.

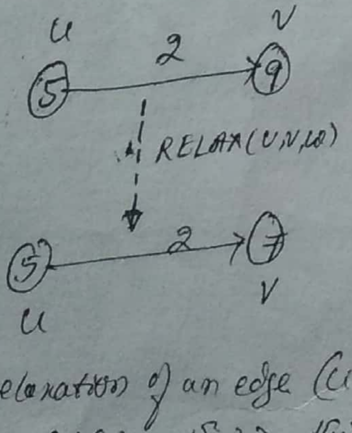
→ The following code performs a relaxation step on edge (u, v)

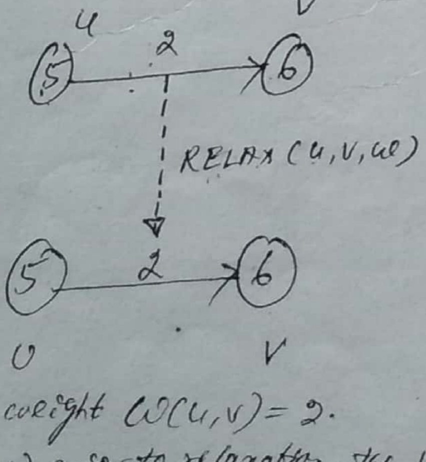
RELAX (u, v, w)

```
if d[v] > d[u] + w(u, v)
    then d[v] ← d[u] + w(u, v)
        π[v] ← u
```

NOTE (1) In dijkstra and shortest-path algo for directed acyclic graph each edge is relaxed exactly once.
(2) In Bellman-Ford algo each edge is relaxed many times.

—o—





→ Relaxation of an edge (u, v) with weight $w(u, v) = 2$.

→ In fig(a) $d[v] > d[u] + w(u, v)$ prior to relaxation, the value of $d[v]$ decreases. But in fig(b) $d[v]$ is unchanged by relaxation.

Breadth First Search (BFS) :- BFS used for Searching a graph.

→ BFS is so named because it expands the frontier (border) between discovered and undiscovered vertices uniformly across the breadth of the frontier.

→ Given a graph $G=(V,E)$ and a source vertex S, BFS systematically explores the edge of G to "discover" every vertex that is reachable from S.

→ It computes the distance (smallest no. of edges) from S to each reachable vertex.

→ It also produces a "BF tree" with root "S" that contains all reachable vertices.

→ This algorithm works on both directed and undirected graph.

To keep track of progress, BFS colors each vertex white, gray or black. Initially all vertices are white.

→ When the vertex is discovered first time, it becomes nonwhite.

→ If $(u,v) \in E$ and vertex u is black, then vertex v is either gray or black; i.e all vertices adjacent to black vertices have been discovered.

→ Gray vertices may have some adjacent white vertices; they represent the frontier (boarder) between discovered and undiscovered.

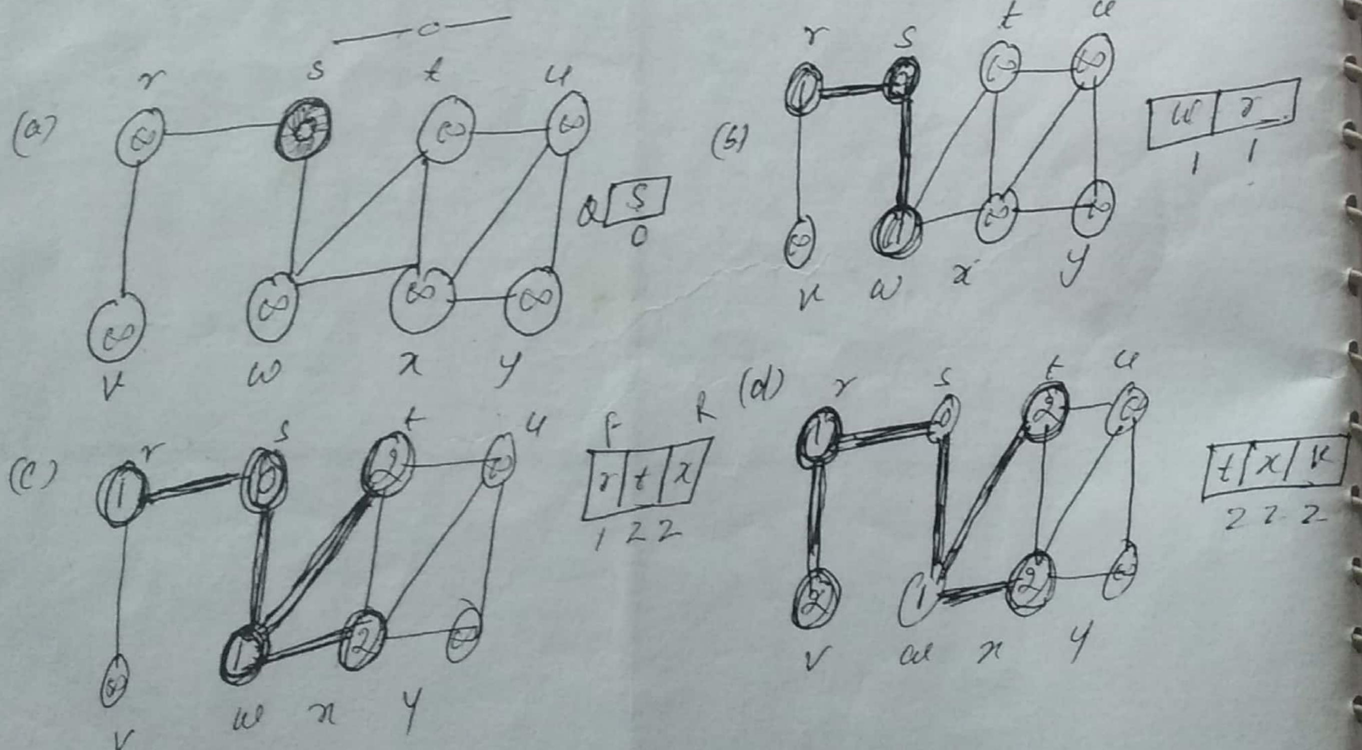
→ Since a vertex is discovered at most once, it has at most one parent.

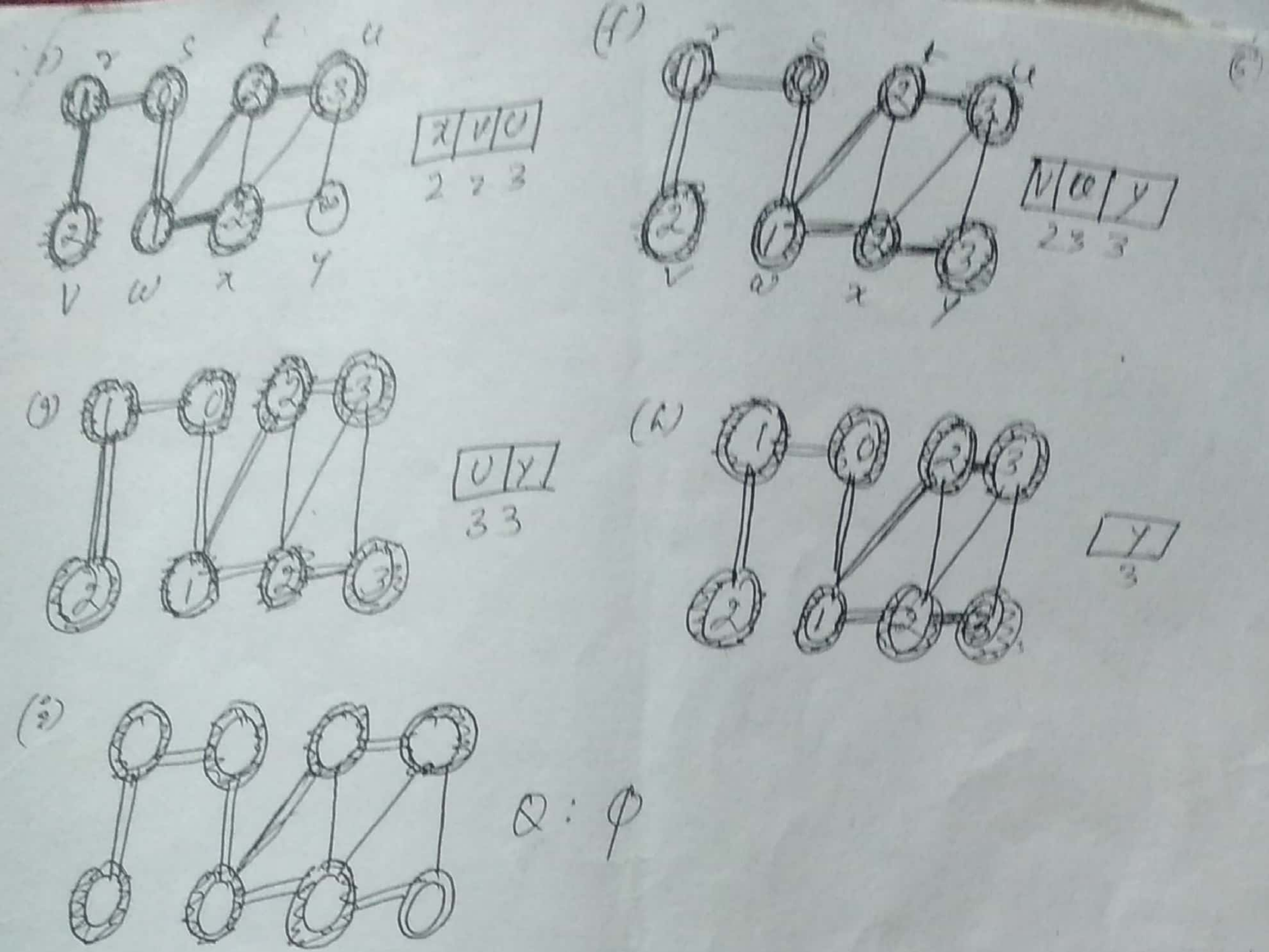
→ BFS assumes input Graph G is represented using adjacency lists.

BFS(G, s)

```
for each vertex u ∈ V(G) − {s}
    do color[u] ← white   // initially all vertices white
       d[u] ← ∞           // distance from s to u
       π[u] ← NIL         // predecessor of u is not there.
color[s] ← GRAY  ||  each step take O(1)
d[s] ← 0         ||
π[s] ← NIL       ||
Q ← φ                // O(V)
ENQUEUE(Q, s)
while Q ≠ φ
    do u ← DEQUEUE(Q)
       for each v ∈ Adj[u]
           do if color[v] = WHITE
                then color[v] ← GRAY
                     d[v] ← d[u]+1
                     π[v] ← u
                     ENQUEUE(Q, v)

       color[u] ← BLACK
```





Time complexity

→ The opⁿ of enqueuing and dequeuing take $O(1)$ time since line-13 ensures. So total time devoted to queue opⁿ is $O(V)$.

→ Because the adjacency list of each vertex is scanned only when the vertex is dequeued, each adjacency list is scanned at most once.

Since the sum of the lengths of all the adjacency list is $O(E)$, the total time spent in scanning adjacency list is $O(E)$.

→ The overhead for initialization is $O(V)$

→ Thus total time of BFS is $\boxed{O(V+E)}$

Scanned by CamScanner

OR

step 1 and 2 : $O(V)$
step 3 : $O(V)$
step 4 : $O(V)$
step 5 : $O(1)$
step 6 : $O(1)$
step 7 : $O(1)$
step 8 : $|V|$
step 9 to 11 : $O(V+E)$
step 12 : $O(V)$ or $|V|$
step 13 : $O(V)$
step 14 : $O(V)$

So time complexity $\boxed{T(n) = O(V+E)}$

Application of BFS

(1) Finding all nodes with in one connected component
(2) Finding shortest path between two nodes u and v
(3) Testing a graph for bipartites.

Scanned by CamScanner

Depth-first search (DFS):-

As its name implies it search "deeper" in the graph whenever possible.

→ Edges are explored out of the most recently discovered vertex v that still has unexplored edges leaving it.

→ when all of v's edge have been explored, the search "backtrack" to explore edge's leaving the vertex from which v was discovered. This process continues.

→ DFS timestamps each vertex. Each vertex v has two timestamps: d[v] records when v is first discovered (and grayed); f[v] " " the search finishes examining v's adjacency list (and blackens v).

→ d[u] records when it discovers vertex u

→ f[u] " " " finishes vertex u

for each vertex u,

$d[u] < f[u]$.

vertex u is WHITE before time d[u], GRAY between time d[u] and time f[u] and BLACK thereafter.

DFS (G)

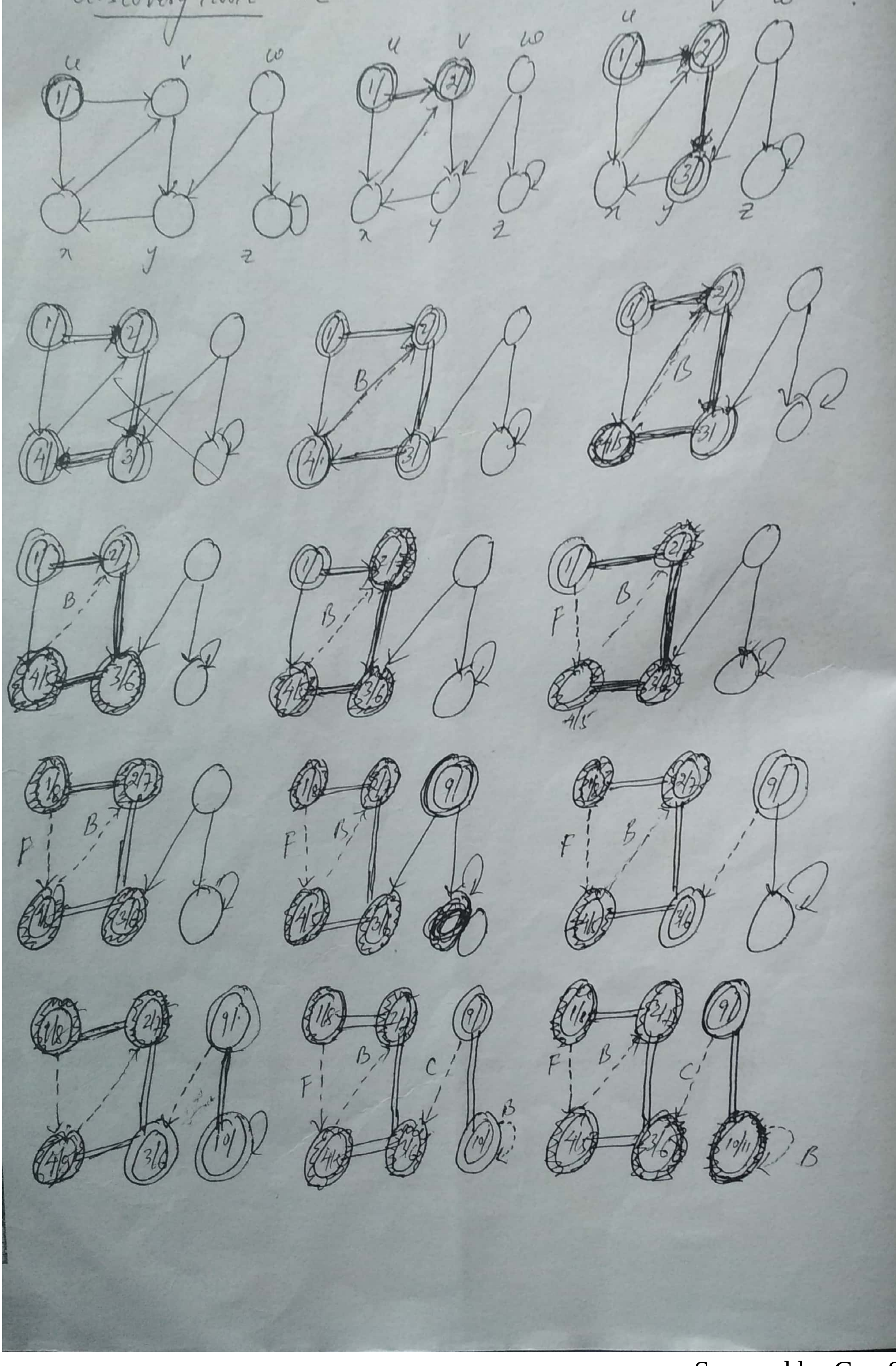
1. for each vertex $u \in V[G]$
2. do colors [u] ← WHITE
3. π [u] ← NIL
4. time ← 0
5. for each vertex $u \in V[G]$
6. do if color[u] = WHITE
7. then DFS-VISIT (u)

DFS-VISIT(u)

1. color [u] ← GRAY // white vertex u has just been discovered
2. time ← time+1
3. d[u] ← time
4. for each $v \in Adj[u]$ // Explore edge (u, v)
5. do if color[v] ← WHITE
6. then π [v] ← u
    DFS-VISIT(v)
7. color[u] ← BLACK // Blacken u; it is finished.

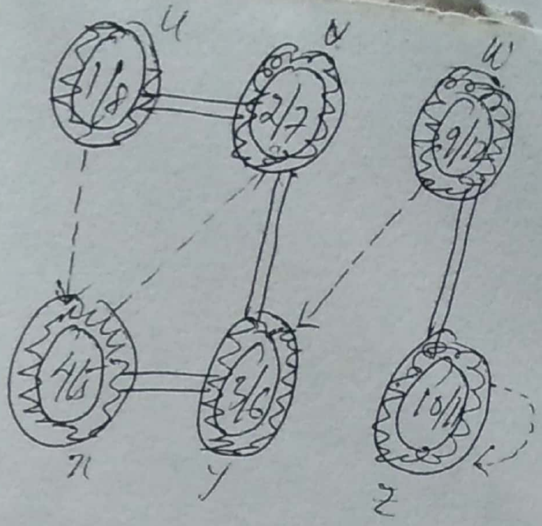
→ Every time DFS-VISIT (u) is called in line-7, vertex u becomes the root of a new tree in the DF forest.

→ When DFS returns, every vertex u has been assigned a <u>discovery time</u> d[u] and a <u>finishing time</u> f[u].



→ Nontree edges are labeled B, C or F according to whether they are back, cross or forward edges.

→ vertices are timestamped by discovery time / finishing time.



Time complexity

→ Loops on lines 1–3 and lines 5–7 of DFS take time $\Theta(V)$

→ DFS-VISIT called exactly once for each vertex $v \in V$, since DFS-VISIT is invoked only on white vertices and the first thing it does is paint the vertex gray.

→ During an execution of DFS-VISIT(v), the loop on lines 4–7 is executed $|Adj[v]|$ times since.

$$\boxed{\sum_{v \in V} |Adj[v]| = \Theta(E)}$$

The total cost of executing lines 4–7 of DFS-VISIT is $\Theta(E)$.

The running time of DFS is therefore $\underline{\Theta(V+E)}$

1 Tractable and Intractable Problems

So far, almost all of the problems that we have studied have had complexities that are *polynomial*, i.e. whose running time $T(n)$ has been $O(n^k)$ for some fixed value of k. Typically k has been small, 3 or less. We will let **P** denote the class of all problems whose solution can be computed in polynomial time, i.e. $O(n^k)$ for some fixed k, whether it is 3, 100, or something else. We consider all such problems efficiently solvable, or *tractable*. Notice that this is a very relaxed definition of tractability, but our goal in this lecture and the next few ones is to understand which problems are *intractable*, a notion that we formalize as *not being solvable in polynomial time*. Notice how *not being in* **P** is certainly a strong way of being intractable.

We will focus on a class of problems, called the *NP-complete problems*, which is a class of very diverse problems, that share the following properties: we only know how to solve those problems in time much larger than polynomial, namely *exponential time*, that is $2^{O(n^k)}$ for some k; and if we could solve one NP-complete problem in polynomial time, then there is a way to solve *every* NP-complete problem in polynomial time.

There are two reasons to study NP-complete problems. The practical one is that if you recognize that your problem is NP-complete, then you have three choices:

1. you can use a known algorithm for it, and accept that it will take a long time to solve if n is large;

2. you can settle for *approximating* the solution, e.g. finding a nearly best solution rather than the optimum; or

3. you can change your problem formulation so that it is in **P** rather than being NP-complete, for example by restricting to work only on a subset of simpler problems.

Most of this material will concentrate on recognizing NP-complete problems (of which there are a large number, and which are often only slightly different from other, familiar, problems in **P**) and on some basic techniques that allow to solve some NP-complete problems in an *approximate* way in polynomial time (whereas an exact solution seems to require exponential time).

The other reason to study NP-completeness is that one of the most famous open problem in computer science concerns it. We stated above that "we *only know* how to solve NP-complete problems in time much larger than polynomial" not that we *have proven* that NP-complete problems require exponential time. Indeed, this is the million dollar question,[1] one of the most famous open problems in computer science, the question whether "**P** = **NP**?", or whether the class of NP-complete problems have polynomial time solutions. After

[1] This is not a figure of speech. See http://www.claymath.org/prizeproblems.

decades of research, everyone believes that **P**$\neq$**NP**, i.e. that no polynomial-time solutions for these very hard problems exist. But no one has proven it. If you do, you will be very famous, and moderately wealthy.

So far we have not actually defined what NP-complete problems are. This will take some time to do carefully, but we can sketch it here. First we define the larger class of problems called **NP**: these are the problems where, if someone hands you a potential solution, then you can *check* whether it is a solution in polynomial time. For example, suppose the problem is to answer the question "Does a graph have a simple path of length $|V|$?". If someone hands you a path, i.e. a sequence of vertices, and you can *check* whether this sequence of vertices is indeed a path and that it contains all vertices in polynomial time, then the problem is in **NP**. It should be intuitive that any problem in **P** is also in **NP**, because are all familiar with the fact that checking the validity of a solution is easier than coming up with a solution. For example, it is easier to get jokes than to be a comedian, it is easier to have average taste in books than to write a best-seller, it is easier to read a textbook in a math or theory course than to come up with the proofs of all the theorems by yourself. For all this reasons (and more technical ones) people believe that **P**$\neq$**NP**, although nobody has any clue how to prove it. (But once it will be proved, it will probably not be too hard to understand the proof.)

The NP-complete problems have the interesting property that if you can solve any one of them in polynomial time, then you can solve *every* problem in **NP** in polynomial time. In other words, they are at least as hard as any other problem in **NP**; this is why they are called *complete*. Thus, if you could show that *any one* of the NP-complete problems that we will study *cannot* be solved in polynomial time, then you will have not only shown that **P**$\neq$**NP**, but also that none of the **NP**-compete problems can be solved in polynomial time.

Conversely, if you find a polynomial-time algorithm for just one NP-complete problem, you will have shown that **P=NP**.[2]

2 Decision Problems

To simplify the discussion, we will consider only problems with Yes-No answers, rather than more complicated answers. For example, consider the *Traveling Salesman Problem* (TSP) on a graph with nonnegative integer edge weights. There are two similar ways to state it:

1. Given a weighted graph, what is the minimum length cycle that visits each node exactly once? (If no such cycle exists, the minimum length is defined to be ∞.)

2. Given a weighted graph and an integer K, is there a cycle that visits each node exactly once, with weight at most K?

Question 1 above seems more general than Question 2, because if you could answer Question 1 and find the minimum length cycle, you could just compare its length to K to answer Question 2. But Question 2 has a Yes/No answer, and so will be easier for us to consider. In

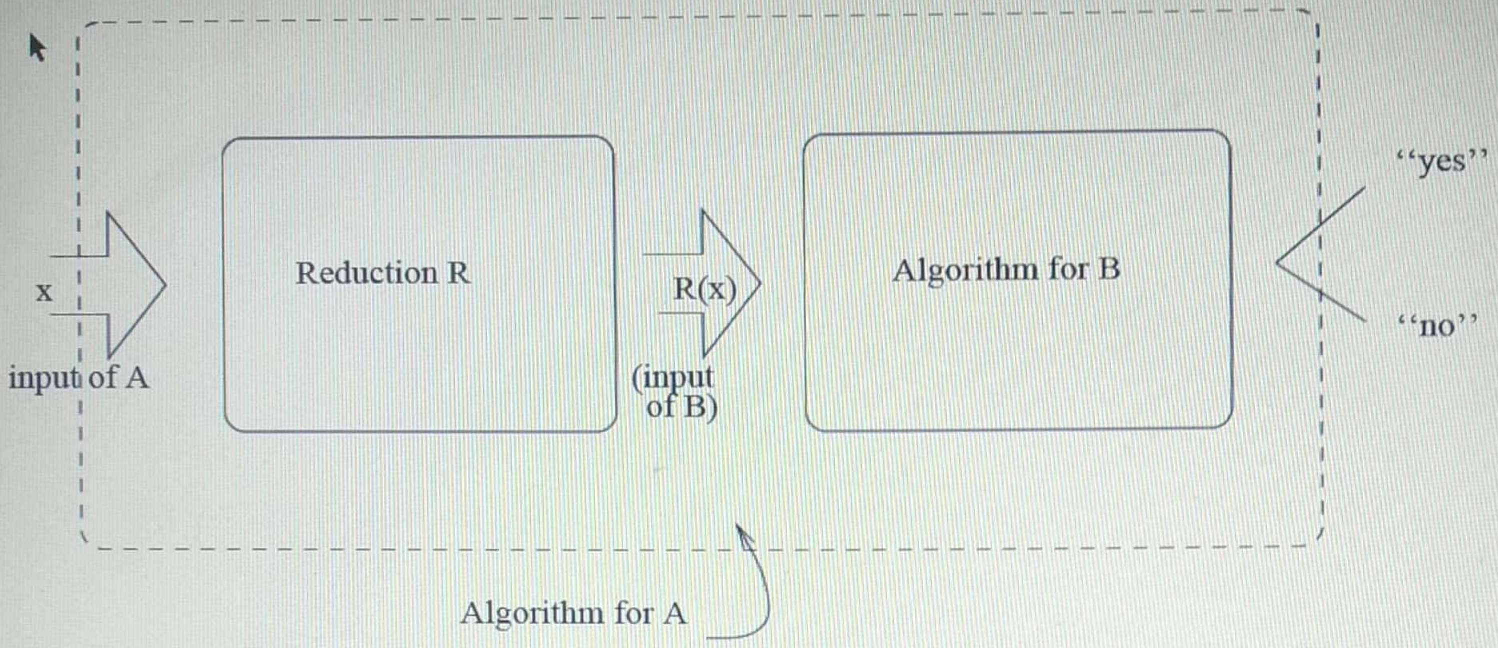


Figure 1: A reduction.

particular, if we show that Question 2 is NP-complete (it is), then that means that Question 1 is at least as hard, which will be good enough for us.[3]

Another example of a problem with a Yes-No answer is *circuit satisfiability* (which we abbreviate CSAT). Suppose we are given a Boolean circuit with n Boolean inputs $x_1, ..., x_n$ connected by AND, OR and NOT gates to one output x_{out}. Then we can ask whether there is a set of inputs (a way to assign True or False to each x_i) such that x_{out} =True. In there is a set of inputs (a way to assign True or False to each x_i) such that x_{out} = True. In particular, we will not ask what the values of x_i are that make x_{out} True.

If A is a Yes-No problem (also called a *decision* problem), then for an input x we denote by $A(x)$ the right Yes-No answer.

3 Reductions

Let A and B be two problems whose instances require Yes/No answers, such as TSP and CSAT. A *reduction* from A to B is a polynomial-time algorithm R which transforms inputs of A to equivalent inputs of B. That is, given an input x to problem A, R will produce an input $R(x)$ to problem B, such that x is a "yes" input of A if and only if $R(x)$ is a "yes" input of B. In a compact notation, if R is a reduction from A to B, then for every input x we have $A(x) = B(R(x))$.

A reduction from A to B, together with a polynomial time algorithm for B, constitute a polynomial algorithm for A (see Figure1). For any input x of A of size n, the reduction R takes time $p(n)$—a polynomial—to produce an equivalent input $R(x)$ of B. Now, this input $R(x)$ can have size at most $p(n)$—since this is the largest input R can conceivably construct in $p(n)$ time. If we now submit this input to the assumed algorithm for B, running in time $q(m)$ on inputs of size m, where q is another polynomial, then we get the right answer of x, within a total number of steps at most $p(n) + q(p(n))$—also a polynomial!

I We have seen many reductions so far, establishing that problems are easy (e.g., from matching to max-flow). In this part of the class we shall use reductions in a more sophisticated and counterintuitive context, in order to prove that certain problems are hard. If we reduce A to B, we are essentially establishing that, *give or take a polynomial, A is no harder than B*. We could write this as

$$A \leq B$$

an inequality between the complexities of the two problems. If we know B is easy, this establishes that A is easy. If we know A is hard, this establishes B is hard. It is this latter implication that we shall be using soon.

4 Definition of Some Problems

Before giving the formal definition of NP and of NP-complete problem, we define some problems that are NP-complete, to get a sense of their diversity, and of their similarity to some polynomial time solvable problems.

In fact, we will look at pairs of very similar problems, where in each pair a problem is solvable in polynomial time, and the other is presumably not.

- **minimum spanning tree:** Given a weighted graph and an integer K, is there a tree that connects all nodes of the graph whose total weight is K or less?

- **travelling salesman problem:** Given a weighted graph and an integer K, is there a cycle that visits all nodes of the graph whose total weight is K or less?

Notice that we have converted each one of these familiar problems into a decision problem, a "yes-no" question, by supplying a goal K and asking if the goal can be met. Any optimization problem can be so converted

If we can solve the optimization problem, we can certainly solve the decision version (actually, the converse is in general also true). Therefore, proving a negative complexity result about the decision problem (for example, proving that it cannot be solved in polynomial time) immediately implies the same negative result for the optimization problem.

By considering the decision versions, we can study optimization problems side-by-side with decision problems (see the next examples). This is a great convenience in the theory of complexity which we are about to develop.

- **Eulerian graph:** Given a directed graph, is there a closed path that visits each edge of the graph exactly once?

- **Hamilitonian graph:** Given a directed graph, is there a closed path that visits each *node* of the graph exactly once?

A graph is Eulerian if and only if it is strongly connected and each node has equal in-degree and out-degree; so the problem is squarely in **P** There is no known such characterization—or algorithm—for the Hamilton problem (and notice its similarity with the TSP).

- **circuit value:** Given a Boolean circuit, and its inputs, is the output T?
- **circuit SAT:** Given a Boolean circuit, is there a way to set the inputs so that the output is T? (Equivalently: If we are given *some* of its inputs, is there a way to set the remaining inputs so that the output is T.)

We know that **circuit value** is in **P**: also, the naïve algorithm for that evaluates all gates bottom-up is polynomial. How about **circuit SAT?** There is no obvious way to solve this problem, sort of trying all input combinations for the unset inputs—and this is an exponential algorithm.

General circuits connected in arbitrary ways are hard to reason about, so we will consider them in a certain standard form, called *conjunctive normal form (CNF)*: Let $x_1, ..., x_n$ be the input Boolean variables, and x_{out} be the output Boolean variable. Then a Boolean expression for x_{out} in terms of $x_1, ..., x_n$ is in CNF if it is the AND of a set of *clauses*, each of which is the OR of some subset of the set of *literals* $\{x_1, ..., x_n, \neg x_1, ..., \neg x_n\}$. (Recall that "conjunction" means "and", whence the name CNF.) For example,

$$x_{out} = (x_1 \vee \neg x_1 \vee x_2) \wedge (x_3 \vee x_2 \vee \neg x_1) \wedge (x_1 \vee x_2) \wedge (x_3)$$

is in CNF. This can be translated into a circuit straightforwardly, with one gate per logical operation. Furthermore, we say that an expression is in **2-CNF** if each clause has two distinct literals. Thus the above expression is not 2-CNF but the following one is:

$$(x_1 \vee \neg x_1) \wedge (x_3 \vee x_2) \wedge (x_1 \vee x_2)$$

3-CNF is defined similarly, but with 3 distinct literals per clause:

$$(x_1 \vee \neg x_1 \vee x_4) \wedge (x_3 \vee x_2 \vee x_1) \wedge (x_1 \vee x_2 \vee \neg x_3)$$

- **2SAT:** Given a Boolean formula in **2-CNF**, is there a satisfying truth assignment to the input variables?
- **3SAT:** Given a Boolean formula in **3-CNF** is there a satisfying truth assignment to the input variables?

2SAT can be solved by graph-theoretic techniques in polynomial time. For **3SAT**, no such techniques are available, and the best algorithms known for this problems are exponential in the worst case, and they run in time roughly $(1.4)^n$, where n is the number of variables. (Already a non-trivial improvement over 2^n, which is the time needed to check all possible assignments of values to the variables.)

- **matching:** Given a boys-girls compatibility graph, is there a complete matching?
- **3D matching:** Given a boys-girls-homes compatibility relation (that is, a set of boy-girl-home "triangles"), is there a complete matching (a set of disjoint triangles that covers all boys, all girls, and all homes)?

We know that matching can be solved by a reduction to max-flow. For **3D matching** there is a reduction too. Unfortunately, the reduction is *from 3SAT to* **3D matching**—and this is bad news for **3D matching**...

- **unary knapsack:** Given integers $a_1, \ldots, a_n$, and another integer K *in unary*, is there a subset of these integers that sum exactly to K?
- **knapsack:** Given integers $a_1, \ldots, a_n$, and another integer K *in binary*, is there a subset of these integers that sum exactly to K?

unary knapsack is in **P**—simply because the input is represented so wastefully, with about $n + K$ bits, so that a $O(n^2K)$ dynamic programming algorithm, which would be exponential *in the length of the input* if K were represented in binary, is bounded by a polynomial in the length of the input. There is no polynomial algorithm known for the real **knapsack** problem. This illustrates that you have to represent your input in a sensible way, binary instead of unary, to draw meaningful conclusions.

5 NP, NP-completeness

Intuitively, a problem is in **NP** if it can be formulated as the problem of whether there is a solution

- They are *small*. In each case the solution would never have to be longer than a polynomial in the length of the input.
- They are *easily checkable*. In each case there is a polynomial algorithm which takes as inputs the input of the problem and the alleged solution, and checks whether the solution is a valid one for this input. In the case of **3SAT**, the algorithm would just check that the truth assignment indeed satisfies all clauses. In the case of *Hamilton*

cycle whether the given closed path indeed visits every node once. And so on.

- Every "yes" input to the problem has at least one solution (possibly many), and each "no" input has none.

Not all decision problems have such certificates. Consider, for example, the problem **non-Hamiltonian graph:** Given a graph G, is it true that there is no Hamilton cycle in G? How would you prove to a suspicious person that a given large, dense, complex graph has *no* Hamilton cycle? Short of listing all cycles and pointing out that none visits all nodes once (a certificate that is certainly not succinct)?

These are examples of problems in NP:

- Given a graph G and an integer k, is there a simple path of length at least k in G?
- Given a set of integers $a_1, \ldots, a_n$, is there a subset S of them such that $\sum_{a \in S} a = \sum_{a \notin S} a$?

We now come to the formal definition.

DEFINITION 1 *A problem A is* **NP** *if there exist a polynomial p and a polynomial-time algorithm $V()$ such that x is a YES-input for problem A if and only if there exists a solution y, with $length(y) \leq p(length(x))$ such that $V(x, y)$ outputs YES.*

We also call **P** the set of decision problems that are solvable in polynomial time. Observe every problem in **P** is also in **NP**.

We say that a problem A is **NP**-hard if for every N in **NP**, N is reducible to A, and that a problem A is **NP**-complete if it is **NP**-hard *and* it is contained in **NP**. As an exercise to understand the formal definitions, you can try to prove the following simple fact, that is one of the fundamental reasons why **NP**-completeness is interesting.

LEMMA 1
If A is **NP***-complete, then A is in* **P** *if and only if* **P** =**NP**.

So now, if we are dealing with some problem A that we can prove to be **NP**-complete, there are only two possibilities:

- A has no efficient algorithm.
- All the infinitely many problems in **NP**, including factoring and all conceivable optimization problems are in **P**.

If **P** =**NP**, then, given the statement of a theorem, we can find a proof in time polynomial in the number of pages that it takes to write the proof down.

If it was so easy to find proof, why do papers in mathematics journal have theorems *and* proofs, instead of just having theorems. And why theorems that had reasonably short proofs have been open questions for centuries? Why do newspapers publish solutions for crossword puzzles? If **P** =**NP**, whatever exists can be found efficiently. It is too bizarre to be true.

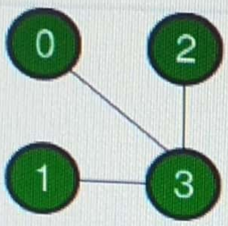
In conclusion, it is safe to assume **P** $\neq$ **NP**, or at least that the contrary will not be proved by anybody in the next decade, and it is *really* safe to assume that the contrary will not be proved by us in the next month. So, if our short-term plan involves finding an efficient algorithm for a certain problem, and the problem turns out to be **NP**-hard, then we should change the plan.

Vertex Cover Problem | Set 1 (Introduction and Approximate Algorithm)

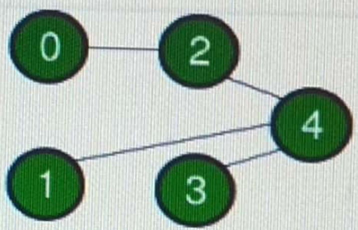
A vertex cover of an undirected graph is a subset of its vertices such that for every edge (u, v) of the graph, either 'u' or 'v' is in vertex cover. Although the name is Vertex Cover, the set covers all edges of the given graph. ***Given an undirected graph, the vertex cover problem is to find minimum size vertex cover.***

Following are some examples.

Minimum vertex cover is empty{}



Minimum vertex cover is {3}



Minimum vertex cover is {4, 2} or {4, 0}

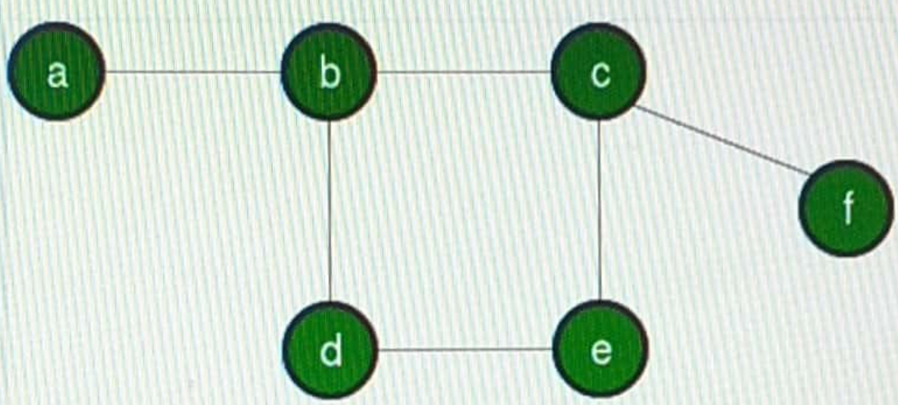
Vertex Cover Problem is a known NP Complete problem, i.e., there is no polynomial time solution for this unless P = NP. There are approximate polynomial time algorithms to solve the problem though. Following is a simple approximate algorithm adapted from CLRS book.

T&C apply W/II/3689/2018-19

Approximate Algorithm for Vertex Cover:

```
1) Initialize the result as {}
2) Consider a set of all edges in given graph.  Let the set be E.
3) Do following while E is not empty
...a) Pick an arbitrary edge (u, v) from set E and add 'u' and 'v' to result
...b) Remove all edges from E which are either incident on u or v.
4) Return result
```

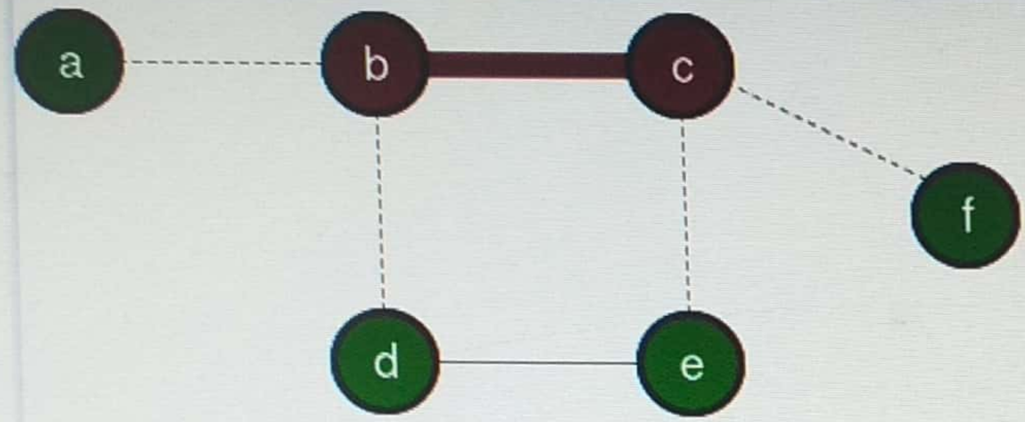
Below diagram to show execution of above approximate algorithm:



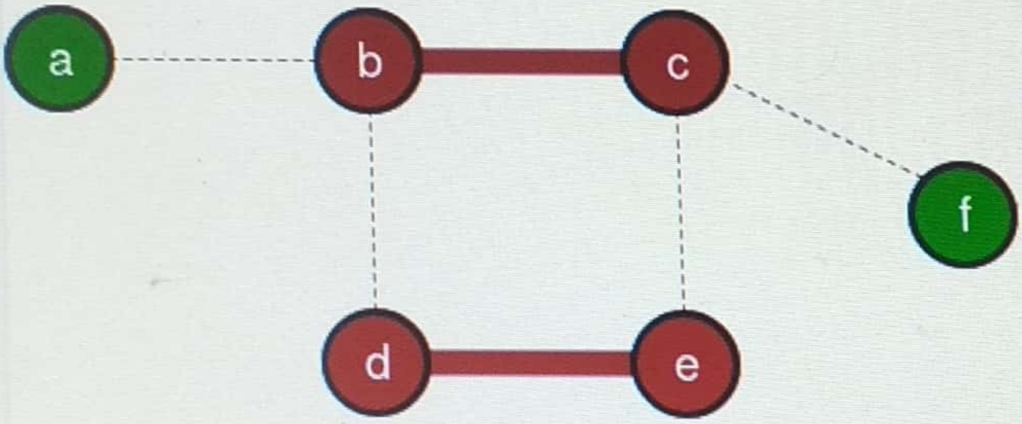
(a)

f Computer - Tu × Vertex Cover Problem | Set 1 (Int × SearchConverter ×

blem-set-1-introduction-approximate-algorithm-2/



(b)



(c)

Minimum Vertex Cover is {b, c, d} or {b, c, e}

THE TRAVELING-SALESMAN PROBLEM

Given a graph G = (V, E) non negative edge weight C(*e*), and an integer *c*, is there a Hamiltonian cycle whose total cost is at most C ?

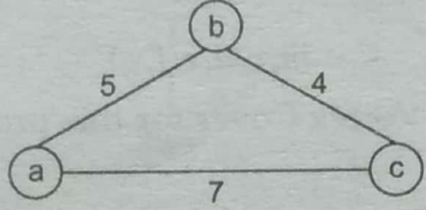
In other words we can also define TSP problem as follows : A Salesperson is required to visit a number of cities during a trip, given the distance between the cities, in what order should he travel so as to visit every city precisely once and returned home, with the minimum mileage traveled ?

Special Case of the Traveling Salesperson Problem

Here first of all we define OPT–TSP as : "Given a complete, weighted graph, find a cycle of minimum cost that visits each vertex". It is also noticeable that OPT–TSP is NP–hard.

Special Case : Edge weights satisfy the triangle inequality (which is common in many applications)

$$w(a,b) + w(b,c) \geq w(a,c)$$



Here, $w(a,b) = 5, \quad w(b,c) = 4 \quad \& \quad w(a,c) = 7$

$\Rightarrow \quad w(a,b) + w(b,c)$

$\Rightarrow \quad 5 + 4$

$\Rightarrow \quad 9 > w(a,c)$

$\Rightarrow \quad 9 > 7$

That is here the condition for triangle inequality is satisfied.

A Traveling-Salesman Problem with the Triangle Inequality

For developing an approximate algorithm for traveling salesman problem is impossible unless P = NP. We will first compute a minimum spanning tree we then crawl around each node following the edges and finally returns back to the first vertex. This is a preorder walk then edges are joined in this order to obtain approx tour walk. Cost of this tour is no more than twice that of MST Cost as long as cost function satisfies the triangle in equality.

```
TSP-APPROX (G)
Input : Weighted complete graph G satisfying triangle inequality.
Output : ATSP tour T for G.
1. M ← a minimum spanning tree for G
2. P ← an Eluer tour traversal of M. starting at some vertex s.
3. T ← Empty list
4. for each vertex v in P (in traversal order)
5.     if this is V's first appearance in P
6.         then T. insert Last (V)
7.     T. insert Last (S)
8. return T
```

Example : *Show the operation of TSP–APPROX (G) for the given set of points.*

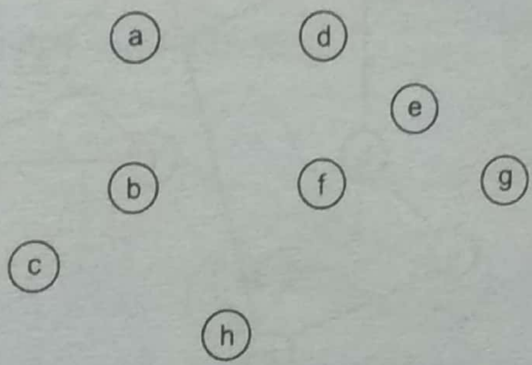


Fig 29.1 [Given set of Points]

Solution :

Step 1 : Find a minimum spanning tree for the given set of points, as :

M ← a minimum spanning tree of G.

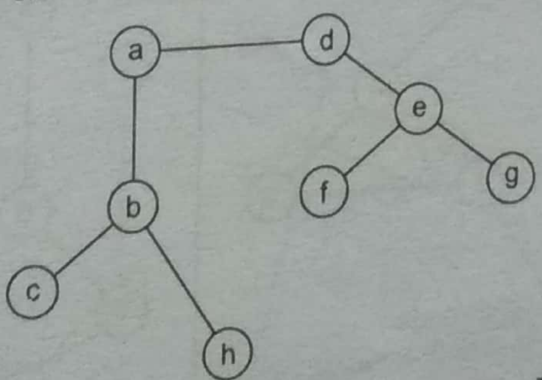


Fig 29.2 [Minimum Spanning Tree]

Step 2 : Find ordered list of vertices in "preorder walk" of M (MST) by using line 2 of TSP–APPROX (G) as :

P ← an Eluer tour traversal of M, starting at some vertex s.

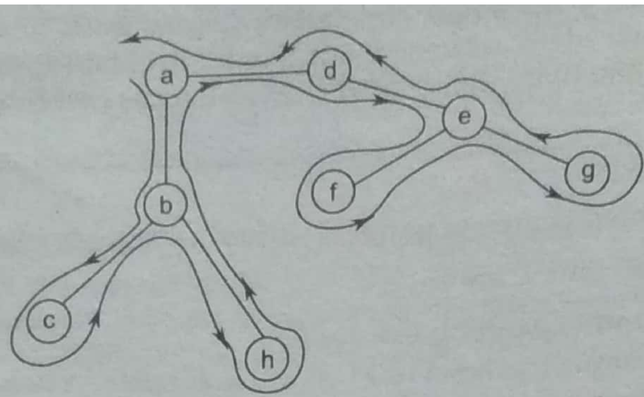
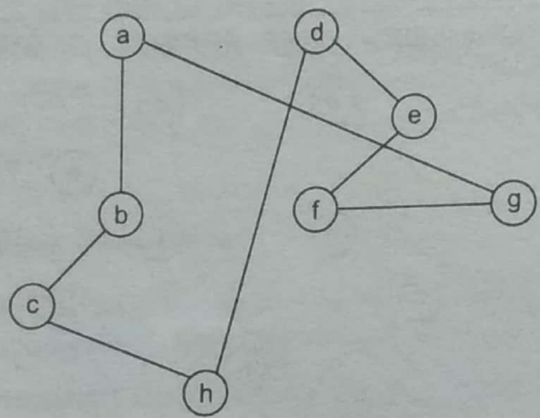


Fig 29.3 [Preorder Traversal Full walk]

Therefore preorder traversal full walk P is :

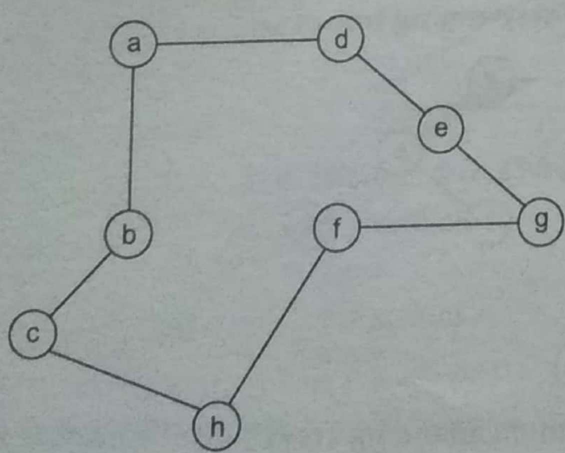
a b c b h b a d e f e g e d a

Step 3 : Find the cycle that visits the vertices in the order L.

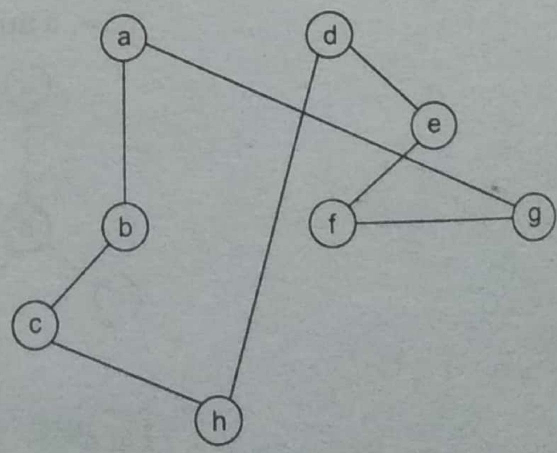


Therefore, Hamiltonian cycle is : *a b c h d e f g a*

Step 4 : Here we will show optimal tour in comparison to Hamiltonian cycle.



An optimal tour : 14.715



Hamiltonian cycle H : 19.074

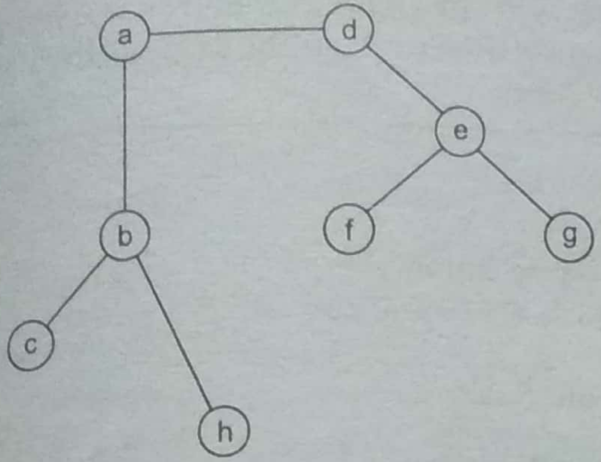
Here in both of the Cases we are assuming Euclidean distance.

Theorem 29.1 : *Approx–TSP is a 2–approximation algorithm for Δ–TSP.*

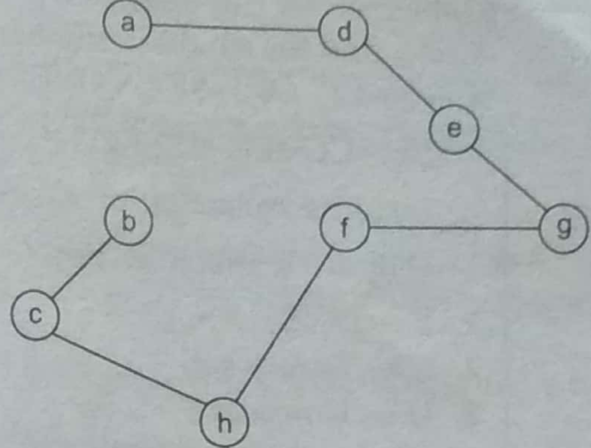
Proof : Let H* denote an optimal tour. We want to show

$$C(H) \le 2\,C(H^*)$$

(*i*) The optimal tour is a spanning tour hence $C(T) \le C(H^*)$ since we obtain spanning tree by deleting any edge from optimal tour.



[MST T]

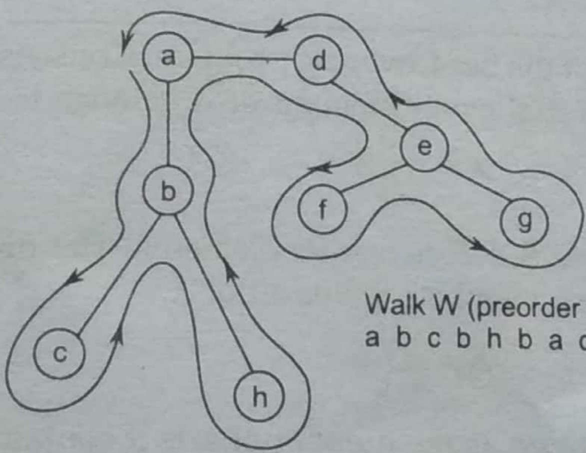


[An Optimal Tour]

(*ii*) The Euler tour $C(w)$ visits each edge of $C(T)$ twice hence

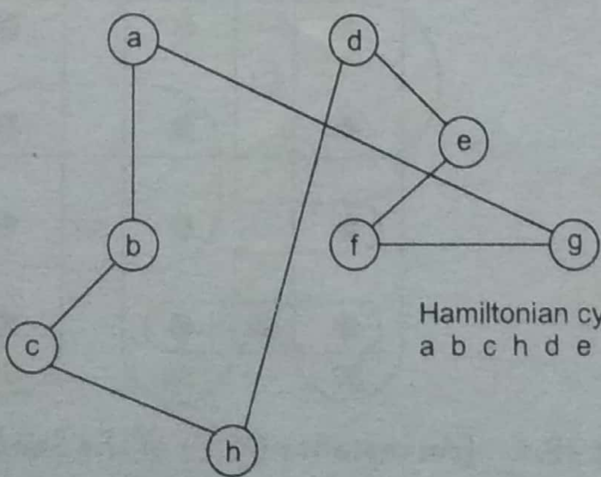
$$C(w) = 2\,C(T) \quad \Rightarrow \quad C(w) \le 2\,C(H^*)$$

Since every edge visited exactly twice.



Walk W (preorder traversal)
a b c b h b a d e f e g e d a

(*iii*) Each time we shortcut a vertex in the Euler Tour we will not increase the total length, by the triangle inequality ($w(a, b) + w(b, c) \ge w(a, c)$) and hence $C(H) \le C(w)$.



Hamiltonian cycle H
a b c h d e f g a

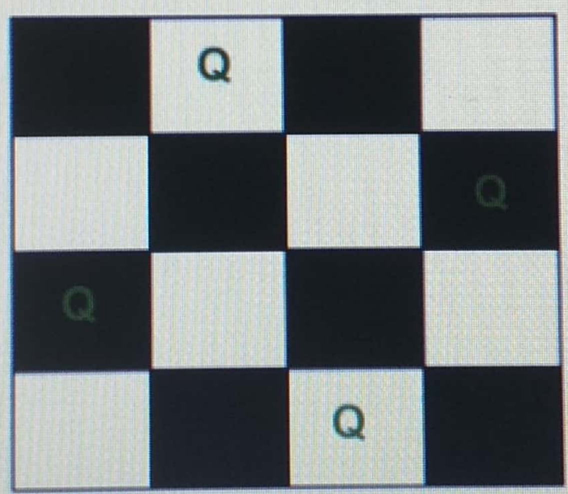
Therefore, from (*ii*) and (*iii*) points given above we can write :

$$\boxed{C(H) \le 2\,C(H^*)}$$

N Queen Problem | Backtracking-3

We have discussed Knight's tour and Rat in a Maze problems in Set 1 and Set 2 respectively. Let us discuss N Queen as another example problem that can be solved using Backtracking.

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen problem.



QUICK SORT

Quicksort is a sorting algorithm whose worst case running time is $\Theta(n^2)$ & Best case & average case is $O(n \log n)$

→ It is one of the fastest sorting algorithm.

→ Like Mergesort QUICK SORT is based on DAC paradigm.

Approaches

Divide :- The array $A[p \ldots r]$ is partitioned into two nonempty subarrays $A[p \ldots q]$ and $A[q+1 \ldots r]$ such that each element of $A[p \ldots q]$ is less than or equal to $A[q+1 \ldots r]$. The index q is computed as part of this partitioning procedure.

Conquer: The two subarrays $A[p \ldots q]$ & $A[q+1 \ldots r]$ are sorted by recursive calls to quicksort.

Combine:- Since two subarrays are sorted in place, no work is needed to combine them : The entire array $A[p \ldots r]$ is now sorted.

partition takes place in quick sort in 3 methods —

Scanned by CamScanner

(i) Find the pivot, if the pivot element is the first element then it is the worst case.

(ii) If pivot element is the middle element, then its the best case.

(iii) If the pivot element is chosen any element of the array then it is average case.

~~Race~~

Basic Idea:

1. Pick one element in the array which will be the pivot.
2. Make one pass thru the array called the partition step rearrange the entries so that:
   - the pivot is in its proper place
   - entries smaller than the pivot are to its left.
   - entries larger than the pivot are to its right.
3. Recovery call (apply) Quicksort to the part of the array

Algorithm of Quicksort

QUICKSORT (A, p, r)

//p - 1st pos' of the array
r - length of the array

1) if p < r
2) then q ← PARTITION (A, p, r)
3) QUICKSORT (A, p, q-1)
4) QUICKSORT (A, q+1, r)

PARTITION (A, p, r)

1) x ← A[p]
2) i ← p-1
3) j ← r+1
4) While TRUE
5) do repeat J ← J-1 //x - pivot
6) Until A[j] ≤ x
7) repeat i ← i+1
8) Until A[i] ≥ x
9) if i < j
10) Then exchange A[i] ↔ A[j]
11) else return j.

~~Example~~ Θ = nLogn Avg case

$O(n^2)$ = worst case

$O(n^2)$:

Appl^n - Language Library

Ex:- | 7 | 2 | 1 | 6 | 8 | 5 | 3 | 6 |

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

less then pivot left to the pivot

& greater " right to the pivot

so we have to sort left to the pivot & sort right to the pivot & two subproblem sublist sort.

partioned

| 2 | 1 | 3 |
0 1 2
pivot

| 8 | 5 | 7 | 6 |
4 5 6 7

| 2 | 1 |
pivot

Now

| 2 | 1 | 3 |

| 1 | 2 | | 3 |
0 1 1

one element
Stop recurssion

Performance of Quicksort

The running time of quick sort depends on whether partitioning is balanced or unbalanced and this in turn depends on which elements are used for partitioning.

→ If partitioning is balanced the algorithm runs as first as merge sort

→ If unbalanced it can run as slow

Best case Partitioning

Equal size of subpartitioning

$n/2 \quad n/2$

is the best case Q. Al.

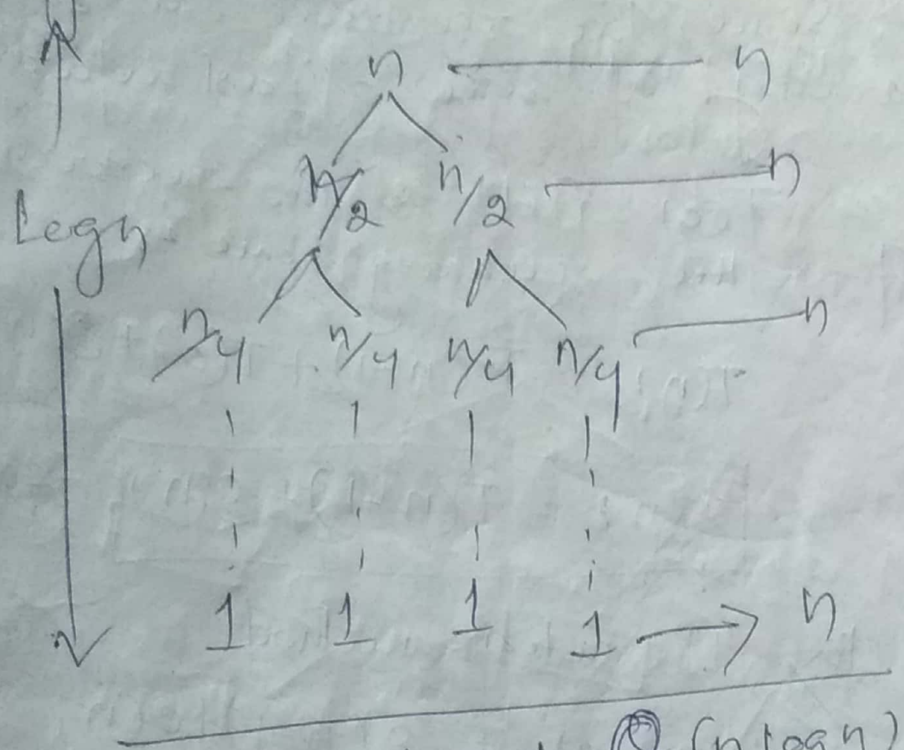
The recurrence for the running time

$$\boxed{T(n) \leq 2T(n/2) + O(n)}$$

By using master theorem case-2 it has the solr

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

Thus the equal balancing of the two sides of the partitions at every level of the recursion produces an asymptotically faster algorithm.



total = $\Theta(n \log n)$

Thus resulting running time is

$$\Theta(n \log n)$$

Worst-case partitioning

The worst-case behavior ~~for~~ QUICKSORT occurs when the partitioning routines produces one subproblem with $n-1$ elements and one with 0 elements.

→ Let us assume that this unbalanced partitioning arises in each recursive call.

→ The partitioning costs $\Theta(n)$ time