SYLLABUS

Subject Code	Name of the Subject	L	T	P	C	
BCSPC3020	**Design and Analysis of Algorithms**	**3**	**0**	**2**	**4**	

Course Educational Objectives

CEO1	Analyze the asymptotic performance of algorithms
CEO2	Demonstrate familiarity with major algorithms
CEO3	Apply important algorithmic design paradigms and methods of analysis
CEO4	Synthesize efficient algorithms in common engineering design situations

Course Outcomes: Upon successful completion of this course, students should be able to:

CO1	***Demonstrate*** and ***analyze*** worst-case running times of algorithms using asymptotic notation
CO2	***Describe*** different algorithm design techniques and ***identify*** the appropriate one based on situation.
CO3	***Design*** the algorithms for different approach and ***evaluate*** the time complexity.
CO4	***Apply*** the approximation algorithm for time consuming problem.

CO-PO & PSO Mapping

COs	**PROGRAMMEOUTCOMES**												**PSOs**		
	1	2	3	4	5	6	7	8	9	10	11	12	1	2	3
CO1	1	1	1												
CO2	1	2	1												
CO3	2	3	3												
CO4		1	2	2											
Avg.															

SYLLABUS

Unit I **(15Hrs)**

Introduction:
Definition, Characteristics of algorithm, Growth of Functions, Asymptotic analysis, Amortized analysis, standard notations and common functions, limit theorem, Stirling's formula. Recurrences: solution of recurrences by substitution, recursion tree and Master methods, Extension Master Methods.

Unit -II **(15Hrs)**

Algorithm design techniques.

Divide-and- conquer Approach: Binary search, Quick sort, Merge sort, Heap Sort, Priority Queue, Lower bounds for sorting. Worst case analysis of Quick sort.

Dynamic programming methodology: Elements of dynamic programming, Matrix-chain multiplication, Longest common subsequence, Assembly-line scheduling.

Greedy Algorithms: Elements of Greedy strategy, Activity selection Problem, Fractional knapsack problem, Huffman codes.

Unit – III **(10Hrs)**

Graph Algorithms: Data structure for disjoint sets, Disjoint set operations, Linked list representation, path compression, Disjoint set forests. Graph Algorithms and their characteristics, Breadth first search and depth-first search, Minimum Spanning Trees, Kruskal algorithm and Prim's algorithms, single- source shortest paths (Bellman-ford Algorithm and Dijkstra's algorithms), All-pairs shortest paths (Floyd–Warshall Algorithm).

Unit – IV **(10 Hrs)**

Back tracking, Branch and Bound, Eight Queen problem, Sub Set Sum Problem. String matching algorithms, naïve string matching algorithm, Rabin-Karp algorithm, Knuth–Morris–Pratt algorithm, NP - Completeness (Polynomial time, Polynomial time verification, NP - Completeness and reducibility, NP-Complete problems (without Proofs), Approximation algorithms characteristics, Traveling Salesman Problem, vertex Cover Problem.

Teaching Methods: Chalk& Board/ PPT/Video Lectures

Text Books:

1. Introduction to Algorithms, T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, 2nd Edition, PHI Learning Pvt. Ltd.
2. Fundamentals of Algorithm, Horowitz &Sahani:, 2nd Edition, Universities Press.

Reference Books:

1. Algorithms, Design and Analysis, H. Bhasin, First Edition, Oxford University
2. Design and Analysis of Algorithm, S. Sridhar, Oxford University Press
3. Algorithms, Sanjay Dasgupta, UmeshVazirani , McGraw-Hill Education.

Design & Analysis of Algorithm

UNIT:1

Defination: It is a sequence of computational steps that transform the input into output.

- It is an efficient method that can be expressed within finite amount of time and space.
- It is a step by step solution of a specific mathematical or computer related problem.
- Independent of prog. language.

Pseudocode

- It is the statement in plain english which may be translated later into a programming language.

Imp: Time $\propto \frac{1}{\text{memory}}$ [If execution time is more then memory consumption is less]

Example:

[Swapping 2 No.'s (Using 3rd variable)]

```
int a, b, temp;
temp = a;
a = b;
b = temp;
```

// Assignment Operation 3 variable 3×2 = 6Bytes Memory

More memory but less execution time

[Swapping 2 No's without using 3rd variable]

```
int a, b;
a = a+b;
b = a-b;
a = a-b;
```

// Assignment with 4 and 2 '-' operation 2×2 = 4 bytes memory

Less Memory but High execution time

Memory day by day less price, so we have to more focous on "Time Complexity"

Algorithm Application

- Merge sort, Quick sort, Heap sort (Gmail chronological display of emails in inbox)
- Fourier Transform (Transform signal from their time domain to frequency domain & vice-versa)
- Dijkstra's Algorithm (Shortest path between two nodes)
- RSA Algorithm (Cryptography, Cyber security)
- Secure Hash algorithm (prevent phishing attack)
- Link analysis [Relation and Connection between network nodes]
  Example: google page Rank
  facebook shows news feed.
  facebook friend suggestion
  Linkedin suggestion for jobs.
  Youtube video suggestion
- Human Genome project (100,000 genes), Data analysis
- quickly access & retrive large amount of information (Search Engine Optimization)
- E-commerce (online transactions) [public key cryptography, digital signatures]

Notes Prepared by
Prof. Chandrakanta Mahanty

✓ Data Compression [Cloud computing, Database Storage]

✓ Random Number generation [lottery, Card games, Cryptography]

Books:

- Introduction to Algorithms, T.H. Cormen, 2nd Ed.
- Design & Analysis of Algorithm, S. Sridhar, Oxford University Press.

Online Source:

- Tutorialspoint.com // tutorial on DAA
- geeksforgeeks

Android App:

✓ Design and analysis of algorithm complete - Engg. wale baba.

Characteristic of an Algorithm (5)

✓ Input - An algorithm must have zero or more but must be finite no. of inputs. Zero Input algorithm : Print the ASCII code of each of the letter in the alphabet

✓ Output - at least one desirable outcome.

✓ Finiteness - An algorithm should terminate infinite number of steps and each step must finish in finite amount of time.

✓ Definiteness (No Ambiguity) - Each step of an algorithm clearly & precisely define and there should not be any ambiguity.

✓ Effectiveness - It must be possible to perform each instruction in a finite amount of time.

Example of Non-effectiveness:

$$e = \frac{1}{} + \frac{1}{(1!)} + \frac{1}{(2!)} + \frac{1}{(3)!} + \dots \quad \text{(infinite terms)}$$

Definition of Algorithm basing upon the above characteristics

∴ It is a collection of finite sequence of definite and effective instructions which terminates with the production of correct output from the given input after a finite time".

Classification of Algorithm

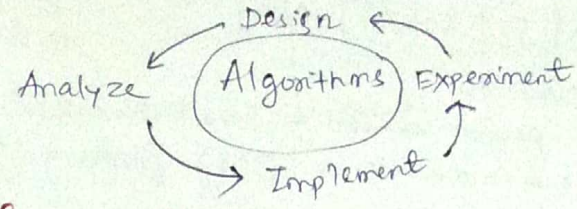
✓ Deterministic algorithm - give a particular input, will always produce the same output

✓ Non-Deterministic - even for the same input can exhibit different behaviors or different runs.

✓ Sequential/Serial - Computer execute one instruction of an algorithm at a time.

✓ Parallel - to process several instructions at once.

4 Major stages of Algorithm Analysis and Design



Life cycle of an algorithm

1. Designing the algorithm (Using different Techniques)
2. Writing the algorithm (Using Pseudocode)
3. Testing the algorithm (Using validation Techniques)
4. Analyzing the algorithm (Analysis)
   (To analyze an algorithm, there is need to determine how much Time an algorithm takes when implemented and how much space it occupies in primary memory)

How to analyze an algorithm

1. Time (How much time is taking) - Time function
2. Space (How much memory space it will consume)
3. Network consumption (How much data going to transfer)
4. Power consumption (How much power is consuming)
5. System level programming / Device drivers (How many register it's consuming)

Apriori Analysis

1. We do analysis (space and time) of an algorithm prior to running it on specific system. [Industry]
2. Algorithm
3. Independent of language
4. Hardware independent
5. Time and space function

Aposteriori Analysis

1. Do analysis of algorithm only after running it on system.
2. Program
3. Language dependent
4. Hardware dependent
5. Watch time and bytes.

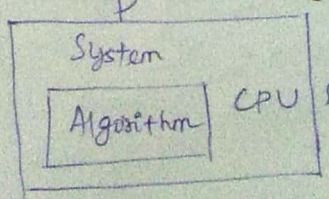
Growth of functions

- Growth functions are used to estimate the number of steps an algorithm uses as its input grows.

Algorithm Complexity

To describe running time of algorithms, we consider the size of input is very large, so we can ignore constants in describing running time.

Divide the i/p with 3 subsets
- Running time less [Best] [It is the function defined by the minimum number of steps taken on any instance of size n]
- Running time more [Worst] [Maximum no. of steps]
- Running time average [Average] [Average no. of steps]

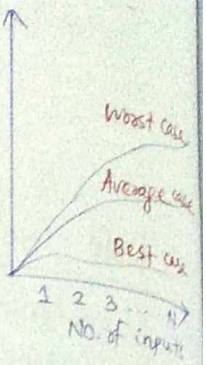


& Memory utilization

Best case ≤ Avg. case ≤ Worst case

Ex: Linear Search (N-element Array)

| 40 | 10 | 20 | 80 | 100 | 50 | 60 | 70 | A[n]



- X-element search, if X = 40
  Best case = 1 no. of comparision = O(1) [order of 1]
- if X = 60, last element
  Worst case = n no. of comparision = O(n)
- if X = 100,
  Average case = n/2 no. of comparision = O(n/2) ≅ O(n)

Asymptotic Analysis

These are the languages that allow us to analyze an algorithm running time by identifying it's behaviour as the input size of the algorithm increases.

Order of Growth: It is described by the highest degree term of the formula for running time. (Drop lower order terms. Ignore the constant coefficient in the leading term)

Ex: $f(n) = an^2 + bn + c$ [Drop lower order terms]

$f(n) = an^2$ [Ignore constant coefficient]

$f(n) = n^2$ i.e $O(n^2)$

Asymptotic notation

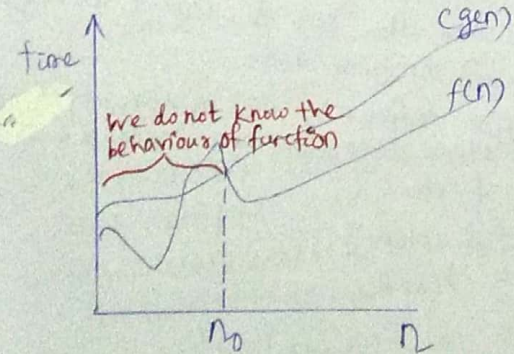
- It is a way to describe the characteristics of a function in the limit
- It describe the rate of growth of functions.
- Focus on what's important by abstracting away low-order terms and constant factors.
- It is a way to compare "sizes" of functions:

1. Big-oh $O \approx \leq$
2. Big-Omega $\Omega \approx \geq$
3. Big-Theta $(\Theta) \approx =$
4. Small-oh $o \approx <$
5. Little-omega $\omega \approx >$

1. Big-oh (O) Notation: (Asymptotic Upper Bound)

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

n_0 = minimum possible value or greater value



g(n) is an asymptotic upper bound for f(n)

Ex:1 $f(n) = 2n+3$

$2n+3 \leq 2n+3n$

$2n+3 \leq 5n$, $n=1$, $O(n)$ [Nearest one]

$5 \leq 5$

also write

$2n+3 \leq 3n$ — $O(n)$ [Nearest]

$2n+3 \leq 2n^2+3n^2$ — $O(n^2)$ } Not Nearest one

$2n+3 \leq 2^n$ — $O(2^n)$ }

$f(n) \neq O(\log n)$ (Lower Bound)

Classes of functions:

$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \dots < 2^n < 3^n < n^n$

(Constant) (log) (linear) (log) (Quadratic) (cube) (exponential)

For ex:1, Lower Bound — Upper Bound

Ex-2: IS $2n^2 = O(n^3)$

$2n^2 \le cn^3$, $c=1$, $n_0=2$

$8 \le 1 \cdot 8$ (satisfied)

so $2n^2 = O(n^3)$ ✓

Ex:3: IS $4n^3+10n^2+5n+1 = O(n^3)$

$4n^3+10n^2+5n+1 \le 4n^3+10n^3+5n^3+n^3$

$4n^3+10n^2+5n+1 \le 19n^3$ (satisfied)

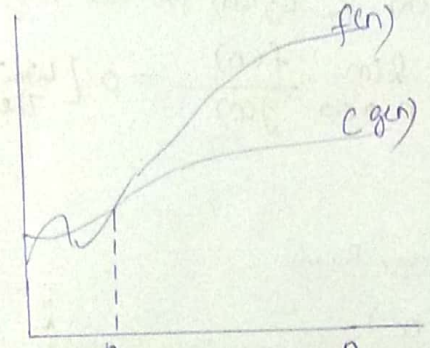
i.e $f(n) = O(n^3)$

or $f(n) \le 5n^3$.

Note: If $f(n)$ is $O(g(n))$ i.e $f(n)$ grows no faster than $g(n)$

2. Big-Omega (Ω) Notation: (Asymptotic Lower Bound)

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



g(n) is an asymptotic lower bound for f(n)

Ex:1 $f(n) = 2n+3$

$2n+3 \ge 1 \cdot n$, $n=1$

$5 \ge 1 \cdot 1$, $f(n) = \Omega(n)$ [Nearest one] ✓

$2n+3 \ge \log n$, $f(n) = \Omega(\log n)$ } Not Nearest

$2n+3 \ge \sqrt{n}$, $f(n) = \Omega(\sqrt{n})$ }

$2n+3 \ge \log\log n$, $f(n) = \Omega(\log\log n)$

$f(n) \ne \Omega(n^2)$ (Upper Bound.)

Ex 2: IS $\sqrt{n} = \Omega(\log n)$

$\sqrt{n} \ge c \log n$, $c=1$, $n_0=16$

$\sqrt{16} \ge 1 \log_2 16$

$4 \ge \log_2 4^2$

$4 \ge 4 \log_2 2$ ($\log_2 2 = 1$) (satisfied)

so $\sqrt{n} = \Omega(\log n)$

Ex 3: IS $4n^3+10n^2+5n+1 = \Omega(n^3)$

$4n^3+10n^2+5n+1 \ge 4n^3$

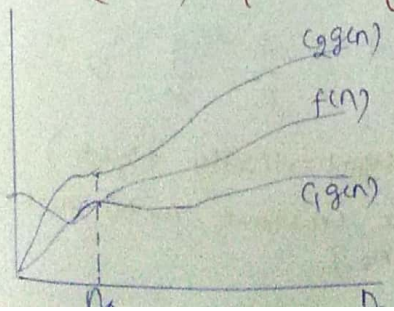
$4n^3+10n^2+5n+1 \ge 1n^3$ (satisfied)

$f(n) = \Omega(n^3)$

Note: If $f(n)$ is $\Omega(g(n))$ i.e $f(n)$ grows no slower than $g(n)$

3. Big-Theta (Θ) Notation: (Asymptotic tight Bound)

$\Theta(g(n)) = \{ f(n)$: there exist positive element constants c_1, c_2 and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0 \}$



g(n) is an asymptotically tight bound for f(n)

$f(n) = 2n+3$, Ex-1

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

$(1)n \le 2n+3 \le (5)n$

$c_1 g(n)$ $c_2 g(n)$

$f(n) = \Theta(n)$ (It shows that exactly equal to n)

X $f(n) = \Theta(n^2)$, $f(n) = \Theta(1 \cdot 3n)$

Note: we can use any notations for best case, worst case.

Ex-2: $n^2/2 - 2n = \Theta(n^2)$, $c_1 g(n) \le f(n) \le c_2 g(n)$

$\frac{1}{4} \times n^2 \le n^2/2 - 2n \le \frac{1}{2} \times n^2$

$\frac{1}{4} \times 8^2 \le 8^2/2 - 2\times 8 \le \frac{1}{2} \times 8^2$

$\frac{1}{4} \times 64 \le \frac{64}{2} - 16 \le \frac{1}{2} \times 64$

$16 \le 16 \le 32$. (satisfied)

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

, $n_0 = 8$, $c_1 = 1/4$, $c_2 = 1/2$

$f(n) = 4n^3 + 10n^2 + 5n + 1$, $g(n) = n^3$

$4 \cdot g(n) \le f(n) \le 5 \cdot g(n)$

$4 \cdot \underset{c_1\, g(n)}{n^3} \le 4n^3 + 10n^2 + 5n + 1 \le 5 \cdot \underset{c_2 \cdot g(n)}{n^3}$ (satisfied)

$4n^3 + 10n^2 + 5n + 1 = \Theta(n^3)$

Note: If $f(n)$ is $\Theta(g(n))$ i.e $f(n)$ and $g(n)$ grows at the same rate

4. Small-oh (o) Notation (Upper Bound that not asymptotically tight)
[Loose upper-bound]

$o(g(n)) = \{ f(n)$: for all constants $c > 0$, there exists a constant $n_0 > 0$, such that $0 \le f(n) < cg(n)$ for all $n \ge n_0 \}$

Another view, Probably easier to use $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ [Limit Theorem]

$n^2 = O(n^2)$ - Tight Test Upper Bound.

$n^2 = O(n^3)$ - Non Tight Test Upper Bound.

Small-oh notation will not give Tight Test Upper Bound.

$n^2 = o(n^2)$ ✗ $2n^2 \ne o(n^2)$

$n^2 = o(n^3)$ ✓ $2n = o(n^2)$ ✓

Ex:1 $f(n) = n$, $g(n) = n^2$ $\lim_{n\to\infty} \frac{n}{n^2} \Rightarrow \lim_{n\to\infty} \frac{1}{n} = \frac{1}{\infty} = 0$

$n = o(n^2)$ ✓

Ex:2 Is $\log n = o(n)$, $f(n) = \log n$, $g(n) = n$

$\lim_{n\to\infty} \frac{\log n}{n}$ ($\frac{\infty}{\infty}$ form, use L Hôpital's Rule i.e $\frac{f'(n)}{g'(n)}$ derivative)

$\lim_{n\to\infty} \frac{1/n}{1} = 1/n = 0$

$\log n = o(n)$ ✓

Note: If $f(n)$ is $o(g(n))$ i.e $f(n)$ grows slower than $g(n)$

Ex:3 : Is $4n^3 + 10n^2 + 5n + 1 = o(n^4)$

$\lim_{n\to\infty} \frac{4n^3 + 10n^2 + 5n + 1}{n^4} = \frac{n^3\left(4 + \frac{10}{n} + \frac{5}{n^2} + \frac{1}{n^3}\right)}{n^3 \cdot n} = \frac{4}{n} = \frac{4}{\infty} = 0$

5. Little-omega (ω) Notation (Lower Bound that is not asymptotically tight)
[Loose Lower bound]

$\omega(g(n)) = \{ f(n)$: for all constants $c > 0$, there exits a constant $n_0 > 0$ such that $0 \le cg(n) < f(n)$ for all $n \ge n_0 \}$

Another view, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

$n^3 = \Omega(n^3)$ – Tight Test Lower Bound

$n^3 = \Omega(n^2)$ – Non Tight Test Lower Bound.

Little-omega notation will not give tight test lower bound.

$n^3 \neq \omega(n^3)$, $n^3 = \omega(n^2)$ ✓

Ex-1: $f(n) = 3^n$, $g(n) = 2^n$ $\lim_{n\to\infty} \frac{3^n}{2^n} = \lim_{n\to\infty} \left(\frac{3}{2}\right)^n = \lim_{n\to\infty} (1.5)^\infty = \infty$

$3^n = \omega(2^n)$

Ex-2 $f(n) = n^2$, $g(n) = \log n$ $\lim_{n\to\infty} \frac{n^2}{\log n}$ ($\frac{\infty}{\infty}$ form L Hôpital's Rule find derivative)

$\lim_{n\to\infty} \frac{2n}{1/n} \Rightarrow \lim_{n\to\infty} 2n^2 \Rightarrow 2\infty^2 \Rightarrow \infty$.

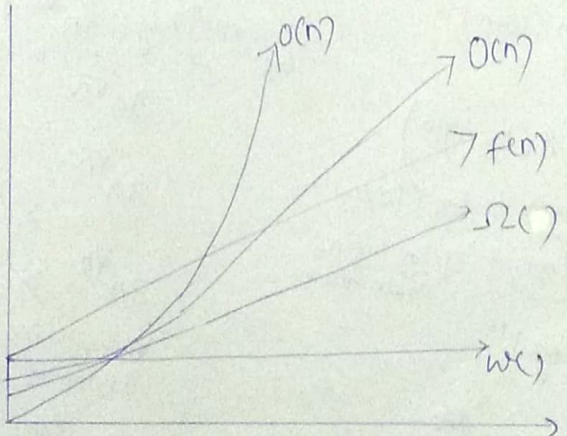
$n^2 = \omega(\log n)$

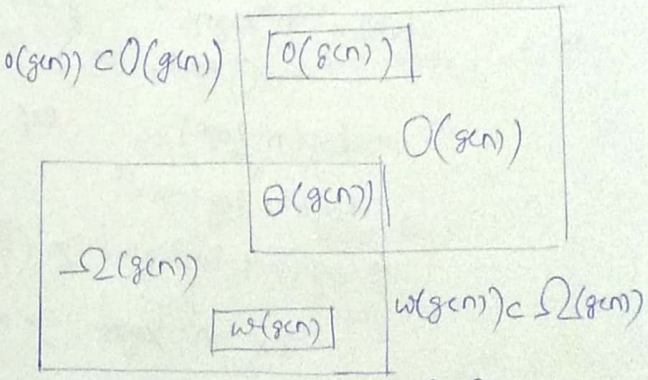
Ex-3: IS $4n^3 + 10n^2 + 5n + 1 = \omega(n^2)$?

$\lim_{n\to\infty} \frac{4n^3+10n^2+5n+1}{n^2} = \lim_{n\to\infty} \frac{n^2(4n+10+\frac{5}{n}+\frac{1}{n^2})}{n^2} = 4\infty+10 = \infty$

$4n^3 + 10n^2 + 5n + 1 = \omega(n^2)$

Note: If $f(n)$ is $\omega(g(n))$ i.e $f(n)$ grows faster than $g(n)$.





Θ is the subset of 'Ω' & 'O'

Relationship between O, Θ, Ω, o, ω

(i) $O(g) \cup \Theta(g) \subset O(g)$

(ii) $\omega(g) \cup \Theta(g) \subset \Omega(g)$

(iii) $\Theta(g) = \Omega(g) \cap O(g)$

Q) Is $2n^2 + 3n + 4 = \Theta(n^2)$?

$1n^2 \leq 2n^2+3n+4 \leq 2n^2+3n^2+4n^2$

$1n^2 \leq 2n^2+3n+4 \leq 9n^2$

$c_1 g(n) \leq f(n) \leq c_2 g(n)$

i.e $\Theta(n^2)$

Q) IS $n^2\log n + n = \Theta(n^2 \log n)$

$1 \cdot n^2\log n \leq n^2\log n + n \leq 10\, n^2 \log n$

$c_1 g(n) \leq f(n) \leq c_2 g(n)$

i.e $\Theta(n^2 \log n)$

Q) $f(n) = n! = n \times (n-1) \times (n-2) \ldots$ 3x2x1 or 1x2x3

$1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \leq n! \leq n \cdot n \cdot n \cdot n \ldots$

$1 \leq n! \leq n^n$ (Both side same function not present)

$\Omega(1)$ Lower Bound — we can not find θ or Tight Bound — $O(n^n)$ Upper Bound.

[can not fix a particular value]

Q) $f(n) = \log n!$

$\log(1 \times 1 \times 1 \times 1 \ldots) \leq \log(1 \times 2 \times 3 \ldots n) \leq \log(n \times n \times n \times n \ldots)$

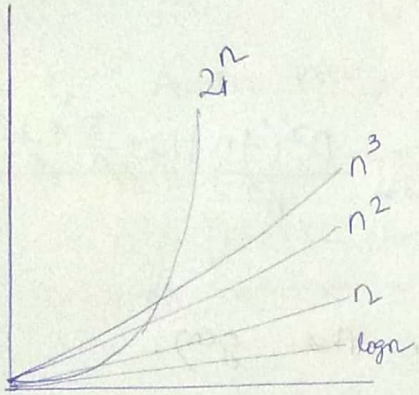
0 means constant so take it as 1

$\Omega(1)$ $\quad 1 \leq \log n! \leq \log n^n$

$\quad\quad\quad\quad\quad\quad \leq n \log n$ $O(n \log n)$

Not tight bound.

Compare class of functions:



formula: 1. $\log ab = \log a + \log b$

2. $\log \frac{a}{b} = \log a - \log b$

3. $\log a^b = b \log a$

4. $a^{\log_c b} = b^{\log_c a}$

5. $a^b = n$, then $b = \log_a n$

Ex-1: $f(n) = n^2 \log n$ $\quad g(n) = n(\log n)^{10}$

(Apply log)

$\log(n^2 \log n)$ $\quad \log\left(n(\log n)^{10}\right)$

$\log n^2 + \log \log n$ $\quad \log n + \log(\log n)^{10}$

$2 \log n + \log \log n$ $>$ $\log n + 10 \log \log n$ (small value)

i.e $n^2 \log n > n(\log)^{10}$

Ex-2: $f(n) = 3n^{\sqrt{n}}$ $\quad g(n) = 2^{\sqrt{n} \log n}$

[Also apply log] $\log(3n^{\sqrt{n}})$ $\quad \log(2^{\sqrt{n} \log n})$

(also by applying formula 3) $\quad \log_2 n^{\sqrt{n}}$

$3n^{\sqrt{n}}$ $\quad 2^{\log_2 n^{\sqrt{n}}}$

$3n^{\sqrt{n}}$ $\quad \left(n^{\sqrt{n}}\right)^{\log_2 2}$ — formula 4

$3n^{\sqrt{n}} > n^{\sqrt{n}}$

value wise greater but asymptotically equal.

Ex:3: $f(n) = n^{\log n}$ $\quad g(n) = 2^{\sqrt{n}}$

(Apply log)

$\log n^{\log n}$ $\quad \log 2^{\sqrt{n}}$

$\log n \log n$ $\quad \sqrt{n} \log_2 2^2 \to 1$

$\log^2 n$ $\quad n^{1/2}$

(Apply log again)

$2 \log \log n < \frac{1}{2} \log n$

i.e $n^{\log n} < 2^{\sqrt{n}}$

Ex:4: $f(n) = 2^{\log n}$ $\quad g(n) = n^{\sqrt{n}}$

(Apply log)

$\log n \log_2 2^1$ $\quad \sqrt{n} \log n$

$\log n < \sqrt{n} \log n$

i.e $2^{\log n} < n^{\sqrt{n}}$

Ex:5 $f(n) = 2^n$ $\quad g(n) = 2^{2n}$

(Apply log)

$n \log_2 2^1$ $\quad 2n \log_2 2^2$

i.e $n < 2n$, $2^n < 2^{2n}$

1. $(n+k)^m = O(n^m)$ $(n+3)^2 = O(n^2)$ Highest degree = 2 correct ✓

2. $2^{n+1} = O(2^n)$ $2 \cdot 2^n$ (co-efficient is ignorable), $O(2^n)$, $\Theta(2^n)$, $\Omega(2^n)$

3. $2^{2n} = O(2^n)$ $4^n > 2^2$, incorrect ✓,

4. $\sqrt{\log n} = O(\log \log n)$ (apply log) $\frac{1}{2}\log\log n > \log\log\log n$, incorrect.

5. $n^{\log n} = O(2^n)$ (apply log) $\log n \log n < n \log 2^2$, correct

Compare:

① $3^n > 2^n$ (apply log)
$n \log 3 > n \log 2$

② $2^n > n^2$
$n \log 2^1 \quad 2 \log n$
(Substitute higher no. of n)
$n \quad 2\log n$
$n = 2^{100}$
$2^{100} \quad 2\log 2^{100}$ i.e $2^{100} > 2 \times 100 \log 2^1 = 200$

③ $n^2 > n \log n$
$n \times n \quad n \log n$

④ $n > (\log n)^{100}$
$\log n \quad 100 \log \log n$
$\log 2^{128} \quad 100 \log \log 2^{128}$ $n = 2^{128}$
$128 \quad 100 \log 128^7$, $128 < 700$
we take for large values $n = 2^{1024}$
$\log 2^{1024} \quad 100 \log \log 2^{1024}$
$1024 \quad 100 \log \log 2^{1024}$
$100 \log 1024^{10} \Rightarrow 1024 > 1000$
(growing more)

⑤ $n^{\log n} > n \log n$
$\log n \log n \quad \log n + \log\log n$
$n = 2^{1024}$
$\log 2^{1024} \log 2^{1024} \quad \log 2^{1024} + \log\log 2^{1024}$
$1024 \times 1024 \quad 1024 + \log 1024^{10}$
$1024 \times 1024 > 1034$
$n = 2^{2^{20}}$
$2^{20} \cdot 2^{20} > 2^{20} + 20$

⑥ $\sqrt{\log n} > \log\log n$
$\frac{1}{2}\log\log n \quad \log\log\log n$
$\frac{1}{2}\log\log 2^{2^{10}} \quad \log\log\log 2^{2^{10}}$ $n = 2^{2^{10}}$
$\frac{1}{2}\log 2^{10} \quad \log\log 2^{10}$
$\frac{1}{2} \times 10 \quad \log 10$
3.5
$\Rightarrow 5$

⑦ $n^{\sqrt{n}} > n^{\log n}$
$\sqrt{n}\log n \quad \log n \log n$
(apply log again)
$\frac{1}{2}\log n \quad \log\log n$
$\frac{1}{2}\log 2^{128} \quad \log\log 2^{128}$ $n = 2^{128}$
$\frac{1}{2} \quad \log 128^7$
$\Rightarrow \frac{1}{2} \times 128 \Rightarrow 64$

$$f(n) = \begin{cases} n^3 & 0 < n < 10{,}000 \\ n^2 & n \geq 10{,}000 \end{cases}$$

$$g(n) = \begin{cases} n & 0 < n < 100 \\ n^3 & n \geq 100 \end{cases}$$

	0 – 99	100 – 9999	10,000 . . . ∞
f(n)	n^3	n^3	n^2
g(n)	n	n^3	n^3
			$n_0 = 10{,}000$

from 10,000 onwards g(n) becomes larger as compare to f(n)
i.e $f(n) = O(g(n))$, $n_0 = 10{,}000$

(8) $f_1 = 2^n$, $f_2 = n^{3/2}$, $f_3 = n \log n$, $f_4 = n^{\log n}$ <compare function>

Ans: $f_1 > f_4 > f_2 > f_3$

Limit Theorem

$$\lim_{n \to \infty} \frac{T(n)}{f(n)}$$

- 0: $T(n) = O(f(n))$
- $c > 0$: $T(n) = \Theta(f(n))$ - $0 < c < \infty$
  - $T(n) = O(f(n))$ - $c < \infty$
  - $T(n) = \Omega(f(n))$ - $c > 0$
- ∞: $T(n) = \omega(f(n))$

(1) Show that any real constant a & b, where $b > 0$

$$(n+a)^b = \Theta(n^b)$$

$$\lim_{n \to \infty} \frac{(n+a)^b}{n^b}, \quad \lim_{n \to \infty} \frac{n^b \left(1 + \frac{a}{n}\right)^b}{n^b} = 1, \text{ i.e } c > 0$$

i.e $(n+a)^b = \Theta(n^b)$

(2) Is $2^{n+1} = O(2^n)$

$$\lim_{n \to \infty} \frac{2^{n+1}}{2^n} \Rightarrow \lim_{n \to \infty} \frac{2^n \cdot 2}{2^n} = 2 \text{ Constant } c > 0$$

i.e $2^{n+1} = O(2^n)$

(3) Is $n! = \omega(2^n)$ $\quad n! = n \cdot (n-1)(n-2) \cdots 3 \cdot 2 \cdot 1$

$$\lim_{n \to \infty} \frac{n!}{2^n} \Rightarrow \frac{n}{2^n} \cdot \frac{n-1}{2^n} \cdot \frac{n-2}{2^n} \cdots \frac{3}{2^n} \cdot \frac{2}{2^n} \cdot \frac{1}{2^n}$$

$$\lim_{n \to \infty} \frac{n^n \left(1 - \frac{1}{n}\right)^{\to 0} \left(1 - \frac{2}{n}\right)^{\to 0} \cdots \frac{3}{n} \cdot \frac{2}{n}^{\to 0} \cdot \frac{1}{n}^{\to 0}}{2^n} = \infty$$

i.e $n! = \omega(2^n)$

(4) Is $n! = O(n^n)$

$$\lim_{n \to \infty} \frac{n!}{n^n} \Rightarrow \lim_{n \to \infty} \frac{n^n \left(1 - \frac{1}{n}\right)^{\to 0} \left(1 - \frac{2}{n}\right)^{\to 0} \cdots \frac{3}{n}^{\to 0} \cdot \frac{2}{n}^{\to 0} \cdot \frac{1}{n}^{\to 0}}{n^n}$$

$\Rightarrow \lim_{n \to \infty} (1-0)(1-0) \cdots 0 \cdot 0 \cdot 0 \Rightarrow 0$, $n! = o(n^n)$

Stirling Formula:

Stirling's approximation gives an approximate value for the factorial $n!$. Stirlings approximation gets better as the number N gets larger.

$n! = 1 \cdot 2 \cdot 3 \cdots n$

$\ln n! = \ln 1 + \ln 2 + \cdots + \ln n$ i.e $\sum_{k=1}^{n} \ln k \simeq \int_1^n \ln x \, dx$

$= [x \ln x - x]_1^n$

$= n \ln n - n + 1$

$\simeq n \ln n - n$ i.e $O(n \log n)$

Formulae: $\int uv = uv - \int v du$

or. Stirling's approximation:

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \theta\left(\frac{1}{n}\right)\right)$$

$$n! \simeq \left(\frac{n}{e}\right)^n$$

(apply logs)

$\log n! = \log \left(\frac{n}{e}\right)^n$ [$\log^b = b \log a$]

$= n \log \frac{n}{e}$

$= n \log n - n \log e$

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

Properties of Asymptotic Notations:

General Properties If $f(n)$ is $O(g(n))$ then $a * f(n)$ is also $O(g(n))$ [a: constant]

Ex: $f(n) = 2n^2 + 5$ is $O(n^2)$

$7(f(n)) = 7(2n^2+5) = 14n^2 + 35 = O(n^2)$ same degree.

$\therefore$ true for Ω and Θ also.

Reflexive If $f(n)$ is given then $f(n)$ is $O(f(n))$

ex: $f(n) = n^2$, $O(n^2)$ [same function upper bound of itself]

(Relation on self) [same function Lower bound of itself]

$\therefore$ $f(n) = O(f(n))$

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

Transitive If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$ then $f(n) = O(h(n))$

Ex: $f(n) = n$, $g(n) = n^2$, $h(n) = n^3$

$n = O(n^2)$, $n^2 = O(n^3)$, then $n = O(n^3)$

$\therefore$ true for o, Ω, Θ, O too.

Symmetric If $f(n) = \Theta(g(n))$ then $g(n)$ is $\Theta(f(n))$, true only for "Θ".

Ex: $f(n) = n^2$, $g(n) = n^2$

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

$g(n) = \Theta(n^2)$ } Symmetric to each other.

Transpose Symmetric If $f(n) = O(g(n))$ then $g(n)$ is $\Omega(f(n))$

Ex: $f(n) = n$, $g(n) = n^2$, $n = O(n^2)$ & n^2 is $\Omega(n)$

Here for n^2, n is lower bound & for n, n^2 is the upper bound.

also true for $f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$

Note: If $f(n) = O(g(n))$ & $f(n) = \Omega(g(n))$

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

[same function of both]

Note: If $f(n) = O(g(n))$, $d(n) = O(e(n))$

then $f(n) + d(n) = ?$ let $f(n) = n$, $d(n) = n^2 = O(n^2)$

$\Rightarrow O(n) + O(n^2) \Rightarrow O(n^2)$ ✓ [Leading term taken]

So we can say $O(\max(g(n), e(n)))$

Note: If $f(n) = O(g(n))$, $d(n) = O(e(n))$, let $f(n)=n$, $d(n)=n^2$

$f(n) * d(n) = ?$ $f(n) * d(n) = n * n^2 = n^3 = O(g(n) \cdot e(n))$

Recurrences: Recursion is generally expressed in terms of recurrences. When an algorithm call to itself, we can often describe its running time by recurrence equation which describes the overall running time of a problem of size n in terms of running time.

Various Techniques to solve Recurrences:

1) **Substitution Method:** we make a guess for the solution and then we use mathematical induction to prove the guess is correct or incorrect.

2) **Recurrence Tree Method:** we draw a recurrence tree and calculate the time taken by every level of tree, finally, we sum the work done at all levels.

3) **Master Theorem Method:** only for following types of recurrences or for recurrences that can be transformed to following type.

$T(n) = aT(n/b) + f(n)$, where $a \geq 1$ and $b > 1$

Recursive function (decreasing)

Ex-1

```
void Test(int n)  —T(n)
{
  if(n>0)  ——— may take 1 unit or may not (bcoz ignore constant)
  {
    printf("%d", n);  — 1 unit
    Test(n-1);  ——— T(n-1) unit
  }
}
```

From this example recurrence relation

$$T(n) = \begin{cases} 1 & n=0 \text{ (do nothing)} \\ T(n-1)+1 & n>0 \end{cases}$$

(time we don't write '0', constant time)

Substitution Method

$T(n) = T(n-1) + 1$, $T(n+1) = T(n-1-1) + 1 = \boxed{T(n-1) = T(n-2) + 1}$

(Substitute)

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

$T(n) = T(n-2) + 2$ $\quad \boxed{T(n-2) = T(n-3) + 1}$

Substitute

$T(n) = [T(n-3) + 1] + 2$

$T(n) = T(n-3) + 3$

... continue for K times

$T(n) = T(n-K) + K$, we know after $n=0$, result = 1

Assume that reached at zero. ie $n-K=0$, $n=K$

$T(n) = T(n-n) + n$

$T(n) = T(0) + n$

$T(n) = 1+n$ ie $T(n) = O(n)$

EX-2:

```
void Test (int n)          ---- T(n)
{ if (n>0)                 ---------- 1 unit
  {for(i=0; i<=n; i++)     -- n+1 unit
    {printf("%d", n);     -- n unit
    }
  Test(n-1);               ---------- T(n-1) time
} }
```

Add all the unit time.

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

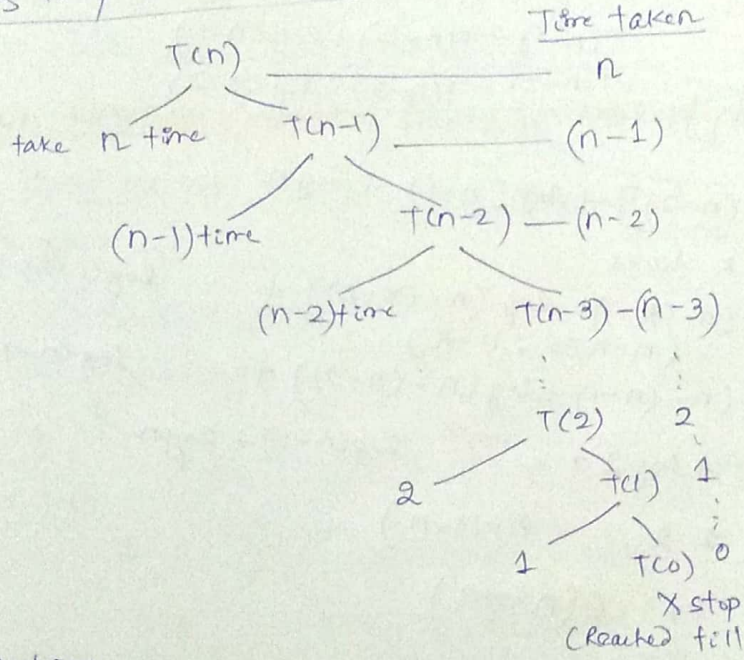
not easy to solve, convert to asymptotic notation.

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

Recurrence Relation

$$T(n) = \begin{cases} 1 & \text{any constant, } n=0 \text{ (do nothing)} \\ T(n-1)+n & , n>0 \end{cases}$$

Tracing Tree / Recursive Tree



Sum all the time

$0+1+2+\cdots+(n-3)+(n-2)+(n-1)+n$

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

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

$T(n) = O(n^2)$ (take highest degree)

Substitution

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

$T(n-1) = T(n-2) + n - 1$ (substitute)

$T(n-2) = T(n-3) + n - 2$

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

$T(n) = T(n-2) + (n-1) + n$ (substitute)

$T(n) = T(n-3) + (n-2) + (n-1) + n$

.... K times ..

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

Assume $n-K=0$, $n=K$

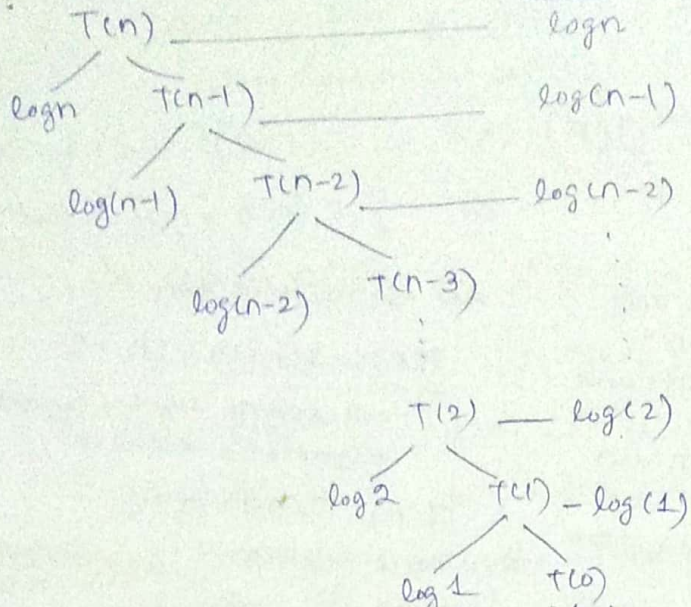
$T(n) = T(n-n) + (n-n+1) + (n-n+2) + \cdots (n-1) + n$

$T(n) = T(0) + 1 + 2 + \cdots n-1+n$

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

Example : 3 $T(n) = \begin{cases} 1 & n=0 \\ T(n-1)+\log n & n>0 \end{cases}$

Tree :



Add the time

$\log n + \log(n-1) + \log(n-2) + \dots + \log 2 + \log 1$

$\log(n*(n-1)*(n-2)\dots \times 2\times 1)$

$\log(n!)$

Stirling's Formulae

$O(n\log n)$

or $\log n!$ (no tight bound)

Upper bound of $\log n! = \log n^n$

i.e $n \log n$ ✓

Induction / Substitution

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

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

$T(n-2) = T(n-3) + \log(n-2)$

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

$T(n) = [T(n-3) + \log(n-2)] + \log[n-1] + \log n$

... k times

$T(n) = T(n-K) + \log(n-(K-1)) + \log(n-(K-2)) + \dots \log(n-1) + \log n$

$(n-K=0,\ n=K)$

$T(n) = T(n-n) + \log(n-(n-1)) + \log(n-(n-2)) + \dots \log(n-1) + \log n$

$T(n) = T(0) + \log 1 + \log 2 + \dots \log(n-1) + \log n$

$T(n) = 1 + \log(1 \cdot 2 \cdot \dots (n-1) \cdot n)$

$T(n) = 1 + \log n!$ i.e $O(n\log n)$

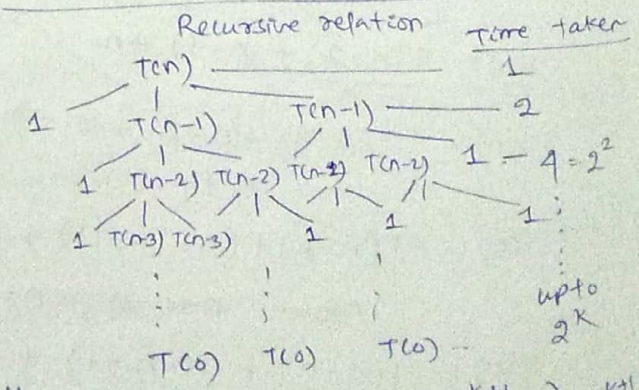
Example : 4 .

```
Algorithm Test (int n)
{ if (n>0)
  { Printf("%d", n); — 1unit
    Test(n-1); — T(n-1) unit
    Test(n-1); — T(n-1) unit
  }
}
```

$T(n) = \begin{cases} 1 & n=0 \\ 2T(n-1)+1 & n>0 \end{cases}$

Recursive relation

Time taken



add all the time :

$1 + 2 + 2^2 + 2^3 \dots 2^K$, GP series.

$a = 1, r = 2,$

i.e $a + ar + ar^2 + ar^3 \dots + ar^K = \frac{a(r^{K+1}-1)}{r-1}$ Put $a=1, r=2$ $\frac{1(2^{K+1}-1)}{2-1} = 2^{K+1}-1$

Assume $n-K=0$, $n=K$ i.e $2^{n+1}-1$ so $O(2^n)$

Substitution :

$T(n) = 2T(n-1)+1$ $\quad T(n-1) = 2T(n-2)+1$

$T(n) = 2[2T(n-2)+1]+1$ $\quad T(n-2) = 2T(n-3)+1$

$T(n) = 2^2 T(n-2) + 2 + 1$

$T(n) = 2^2[2T(n-3)+1]+2+1$

$T(n) = 2^3 T(n-3) + 2^2 + 2 + 1$

... K times.

$T(n) = 2^K T(n-K) + 2^{K-1} + 2^{K-2} + \dots + 2^2 + 2 + 1$

Assume $n-K=0$, $n=K$

$T(n) = 2^K T(n-K) + 2^K - 1$

$T(n) = 2^K T(n-n) + 2^n - 1$

$T(n) = 2^n T(0) + 2^n - 1$ ($T(0) = 1$)

$T(n) = 2^{n+1} - 1$ i.e $O(2^n)$

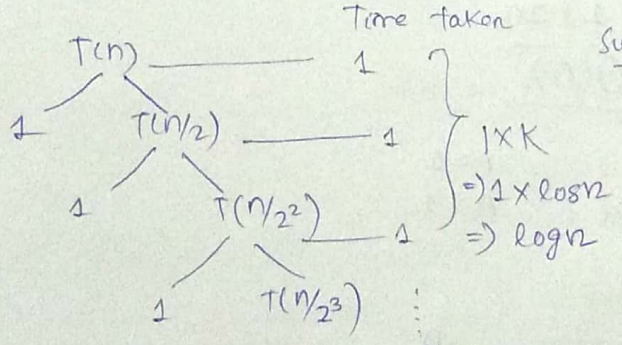
Recursive Relation (Dividing function) - Unit value of division = 1

Ex-5

```
void Test(int n)
{ if(n>1)
  { printf("%d", n);
    Test(n/2);
  }
}
```

$\rightarrow T(n) = \begin{cases} 1 & n=1 \text{ (nothing)} \\ T(n/2)+1 & n>1 \end{cases}$

Recursive Tree



Assume that $n/2^K = 1$

$n = 2^K$

$K = \log_2 n$

i.e $O(\log n)$

Substitution

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

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

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

$T(n/2^2) = T(n/2^3) + 1$

$T(n) = T(n/2^3) + 1 + 1 + 1$

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

K times

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

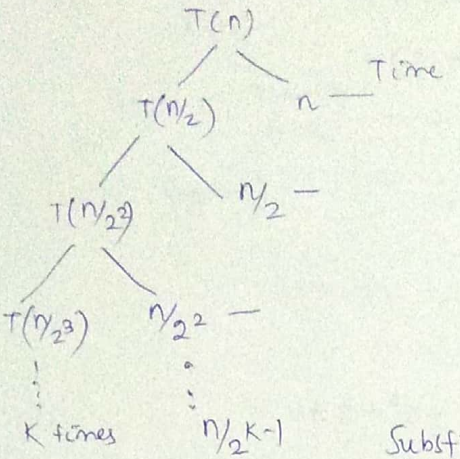
Assume $n/2^K = 1$, $n = 2^K$, $K = \log n$

$T(n) = T(1) + \log n$

$T(n) = 1 + \log n$

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

Ex-6 : $T(n) = \begin{cases} 1 & n=1 \\ T(n/2) + n & n>1 \end{cases}$



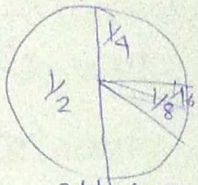
Add = $T(n) = n + n/2 + n/2^2 + \cdots \frac{n}{2^K}$ (approx)

$= n\left(1 + \frac{1}{2} + \frac{1}{2^2} + \cdots \frac{1}{2^K}\right)$ (1)

$= n(1+1)$

$T(n) = 2n$

i.e $T(n) = O(n)$



add all pieces get=1

Substitution

$T(n) = T(n/2) + n$ $\quad T(n/2) = T[n/2^2 + n/2]$

$T(n) = [T(n/2^2) + n/2)] + n$ $\quad T(n/2^2) = T(n/2^3 + n/2^2)$

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

... K times

$T(n) = T(n/2^K) + \frac{n}{2^{K-1}} + \frac{n}{2^{K-2}} + \cdots \frac{n}{2} + n$

Assume that $n/2^K = 1$, $n = 2^K$, $K = \log n$

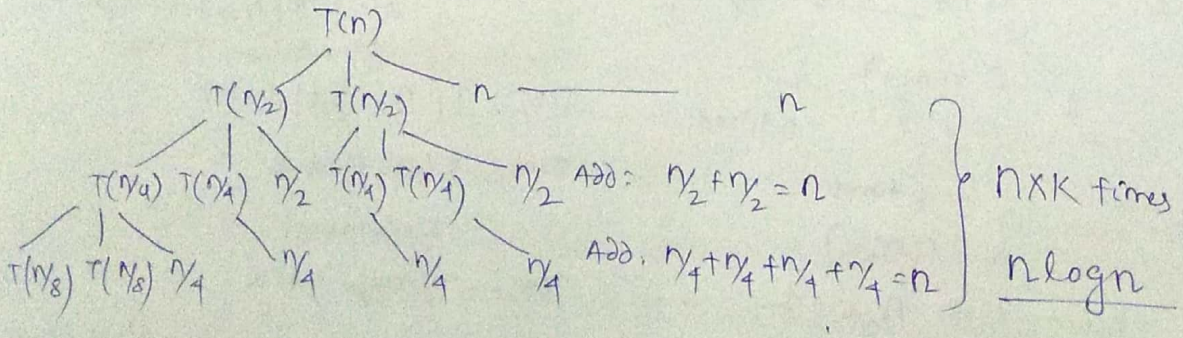
$T(n) = T(1) + n\left[\left(\frac{1}{2^{K-1}} + \frac{1}{2^{K-2}} + \cdots \frac{1}{2}\right) + 1\right]$ (1)

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

$T(n) = 1 + 2n$

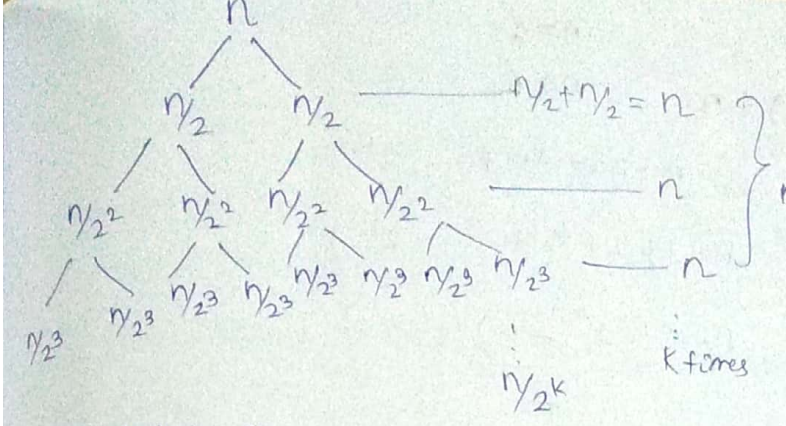
$T(n) = O(n)$

Ex-7 : $T(n) = \begin{cases} 1 & n=1 \\ 2T(n/2) + n & n>1 \end{cases}$



Assume $n/2^K = 1$, $n = 2^K$, $K = \log n$

Note: Here we can concider the 2nd term of the recurrence as a root. we do not concides the function. Avoid writing function



$n \times k$ times $\Rightarrow n \times \log n$ i.e. $O(n \log n)$

$n/2^k = 1$, $n = 2^k \Rightarrow k = \log n$

Substitute :

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

$= 2[T(n/2^2) + n/2] + n$ $\quad T(n/2^2) = 2T(n/2^3) + n/2^2$

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

$= 2^2 [2T(n/2^3) + n/2^2] + n + n$

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

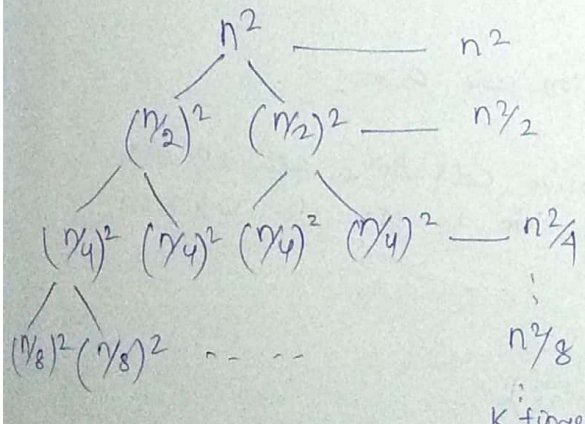
$T(n) = 2^3 T(n/2^3) + 3n$ .. k times

$T(n) = 2^k T(n/2^k) + kn$, Assume $n/2^k = 1$, $2^k = n$, $k = \log n$

$T(n) = nT(1) + n \log n$

$T(n) = n + n \log n$ i.e. $O(n \log n)$

Example - 8 : $T(n) = \begin{cases} 2T(n/2) + n^2 & n > 1 \\ 1 & n = 1 \end{cases}$



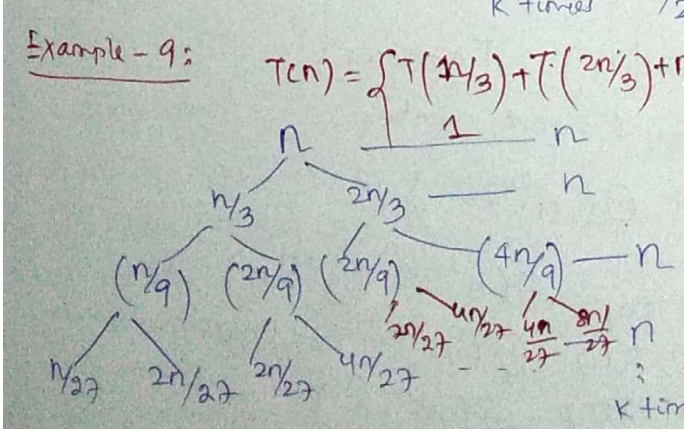
$Add = n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \frac{n^2}{8} \cdots \frac{n^2}{2^k}$

$= n^2 \left(1 + \left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} \cdots \frac{1}{2^k}\right)\right)$

$= n^2 (1 + 1)$

$T(n) = 2n^2$ i.e. $T(n) = O(n^2)$

Example - 9 : $T(n) = \begin{cases} T(n/3) + T(2n/3) + n & n > 1 \\ 1 & n = 1 \end{cases}$



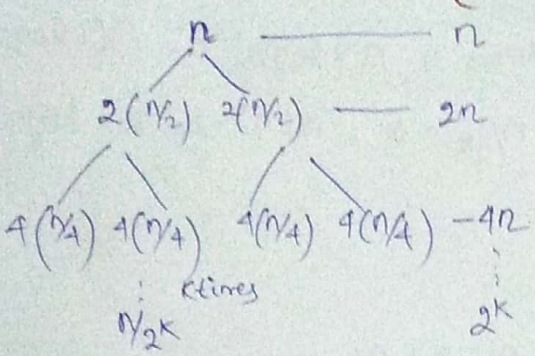
$n \times k$ times i.e. $n(\log_{3/2} n) = n \frac{\log n}{\log 3/2} \Rightarrow O(n \log n)$

$n \times \log n$, $O(n \log n)$

or. $\left(\frac{1}{3}\right)^k n = 1$, $n = (3/1)^k \Rightarrow k = \log_3 n$, Lower Bound.

i.e. $\left(\frac{2}{3}\right)^k n = 1$, $\Rightarrow n = \left(\frac{3}{2}\right)^k \Rightarrow k = \log_{3/2} n$

Example - 10 — $T(n) = \begin{cases} 1 & n=1 \\ 4T(n/2)+n & n>1 \end{cases}$



add $n+2n+4n+\cdots 2^k$

$n(1+2+4+\cdots 2^k)$

$n \cdot \frac{1(2^{k+1}-1)}{1}$

$\Rightarrow n.[2^{\log_2 n+1}-1]$

$\Rightarrow n.(n \log_2^{+1} 2 \cdot 2^1 - 1)$

$\Rightarrow 2n^2 - n$ i.e $O(n^2)$

Master Theorem for Decreasing function

$T(n) = aT(n-b) + f(n)$, $a>0$, $b>0$ and $f(n) = O(n^k)$, where $k \geq 0$

If $a=1$ — $O(n^{k+1})$ or $O(n * f(n))$

If $a>1$ — $O(n^k a^{n/b})$ or $O(n * f(n))$

If $a<1$ — $O(n^k)$ or $O(f(n))$

$T(n) = T(n-1)+1 \rightarrow O(n)$
$T(n) = T(n-1)+n \rightarrow O(n^2)$
$T(n) = T(n-1)+\log n \rightarrow O(n \log n)$
$T(n) = T(n-2)+1 \rightarrow n/2 \rightarrow O(n)$
$T(n) = T(n+100)+n \rightarrow O(n^2)$
$T(n) = 2T(n-1)+1 \rightarrow O(2^n)$
$T(n) = 2T(n-2)+1 \rightarrow O(2^{n/2})$
$T(n) = 2T(n-1)+n \rightarrow O(n2^n)$
$T(n) = 3T(n-1)+1 \rightarrow O(3^n)$

Master Theorem for Dividing function

Extension Master Methods / Advanced Master Theorem

It is used to determine running time of algorithms (divide and conquer) in terms of asymptotic notations.

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

n = size of the problem

a = number of subproblems in the recursion and $a>=1$

n/b = size of each subproblem

$f(n)$ = cost of work done outside the recursive call like dividing into subproblems and cost of combining them to get the solution.

Note: Recurrence Relation not applied —

- $T(n) = \sin n$
- $T(n) = 2T(n/2) + 2^n$ (Polynomial)

$$T(n) = aT(n/b) + \theta(n^k \log^p n) \quad \text{[Advanced]}$$

$a>=1$, $b>1$, $k>=0$, p = real number.

Then

case 1. if $a > b^k$, then $T(n) = \theta(n^{\log_b a})$

case 2. if $a = b^k$, then

a) if $p > -1$, then $T(n) = \theta(n^{\log_b a} \log^{p+1} n)$

b) if $p = -1$, then $T(n) = \theta(n^{\log_b a} \log \log n)$

c) if $p < -1$, then $T(n) = \theta(n^{\log_b a})$

case 3. if $a < b^k$, then

a) if $p >= 0$, then $T(n) = \Theta(n^k \log^p n)$

b) if $p < 0$, then $T(n) = \Theta(n^k)$

Examples: (1) $T(n) = 2^n T(n/2) + n^n$ (can't be solved, not in the form)

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

(3) $T(n) = 4T(n/2) + n^2 \Rightarrow T(n) = \Theta(n^2 \log n)$ (case 2)

(4) $T(n) = T(n/2) + 2^n \Rightarrow \Theta(2^n)$ (case 3)

(5) $T(n) = 16T(n/4) + n \Rightarrow \Theta(n^2)$ (case 1)

(6) $T(n) = 2T(n/2) + n\log n \Rightarrow \Theta(n \log^2 n)$ (case 2)

(7) $T(n) = 2T(n/2) + \frac{n}{\log n}$ => Does not apply, but by substitution/Recursion we get the answer, i.e $\Theta(n \log \log n)$,

(8) $T(n) = 2T(n/4) + n^{0.51} \Rightarrow \Theta(n^{0.51})$ (case 3)

(9) $T(n) = 0.5T(n/2) + 1/n \Rightarrow$ Does not apply ($a < 1$)

(10) $T(n) = 16T(n/4) + n! \Rightarrow \Theta(n!)$ (case 3)

(11) $T(n) = \sqrt{2}T(n/2) + \log n \Rightarrow \Theta(\sqrt{n})$ (case 1)

(12) $T(n) = 3T(n/2) + n \Rightarrow \Theta(n^{\log 3})$ (case 1)

(13) $T(n) = 3T(n/3) + \sqrt{n} \Rightarrow \Theta(n)$ (case 1)

(14) $T(n) = 4T(n/2) + cn \Rightarrow \Theta(n^2)$ (case 1)

(15) $T(n) = 3T(n/4) + n\log n \Rightarrow \Theta(n \log n)$ (case 3)

(16) $T(n) = 3T(n/3) + n/2 \Rightarrow \Theta(n \log n)$ (case 2)

(17) $T(n) = 6T(n/3) + n^2 \log n \Rightarrow \Theta(n^2 \log n)$ (case 3)

(18) $T(n) = 4T(n/2) + n/\log n \Rightarrow \Theta(n^2)$ (case 1)

(19) $T(n) = 64T(n/8) - n^2 \log n \Rightarrow$ Does not apply, f(n) not positive

(20) $T(n) = 7T(n/3) + n^2 \Rightarrow T(n) = \Theta(n^2)$ (case 3)

(21) $T(n) = 4T(n/2) + \log n \Rightarrow \Theta(n^2)$ (case 1)

(22) $T(n) = T(n/2) + n(2 - \cos n) \Rightarrow$ Does not apply.

Algorithm Design Techniques:

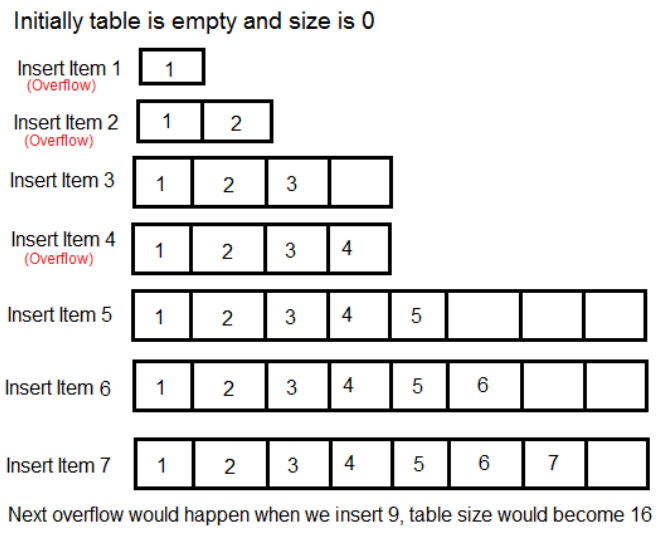
1. Divide-and-Conquer Method
2. Greedy Method
3. Dynamic programming method
4. Branch and Bound method
5. Approximation method
6. Backtracking method
7. Recursive method
8. Randomized method
9. Brutforce method
10. Incremental method.

Analysis of Algorithm | Set 5 (Amortized Analysis Introduction)

Amortized Analysis is used for algorithms where an occasional operation is very slow, but most of the other operations are faster. In Amortized Analysis, we analyze a sequence of operations and guarantee a worst case average time which is lower than the worst case time of a particular expensive operation.
The example data structures whose operations are analyzed using Amortized Analysis are Hash Tables, Disjoint Sets and Splay Trees.

Let us consider an example of a simple hash table insertions. How do we decide table size? There is a trade-off between space and time, if we make hash-table size big, search time becomes fast, but space required becomes high.

The solution to this trade-off problem is to use Dynamic Table (or Arrays). The idea is to increase size of table whenever it becomes full. Following are the steps to follow when table becomes full.
1) Allocate memory for a larger table of size, typically twice the old table.
2) Copy the contents of old table to new table.
3) Free the old table.



If the table has space available, we simply insert new item in available space.

What is the time complexity of n insertions using the above scheme?
If we use simple analysis, the worst case cost of an insertion is O(n). Therefore, worst case cost of n inserts is n * O(n) which is $O(n^2)$. This analysis gives an upper bound, but not a tight upper bound for n insertions as all insertions don't take Θ(n) time.

Item No.	1	2	3	4	5	6	7	8	9	10	.....
Table Size	1	2	4	4	8	8	8	8	16	16	.....
Cost	1	2	3	1	5	1	1	1	9	1	.....

$$\text{Amortized Cost} = \frac{(1 + 2 + 3 + 5 + 1 + 1 + 9 + 1...)}{n}$$

We can simplify the above series by breaking terms 2, 3, 5, 9.. into two as (1+1), (1+2), (1+4), (1+8)

$$\text{Amortized Cost} = \frac{[\overbrace{(1 + 1 + 1 + 1...)}^{\text{n terms}} + \overbrace{(1 + 2 + 4 + ...)}^{\lfloor \text{Log}_2(n-1) \rfloor + 1 \text{ terms}}]}{n}$$

$$<= \frac{[n + 2n]}{n}$$

$$<= 3$$

$$\text{Amortized Cost} = O(1)$$

So using Amortized Analysis, we could prove that the Dynamic Table scheme has O(1) insertion time which is a great result used in hashing. Also, the concept of dynamic table is used in vectors in C++, ArrayList in Java.

Following are few important notes.
1) Amortized cost of a sequence of operations can be seen as expenses of a salaried person. The average monthly expense of the person is less than or equal to the salary, but the person can spend more money in a particular month by buying a car or something. In other months, he or she saves money for the expensive month.

2) The above Amortized Analysis done for Dynamic Array example is called ***Aggregate Method***. There are two more powerful ways to do Amortized analysis called ***Accounting Method*** and ***Potential Method***. We will be discussing the other two methods in separate posts.

3) The amortized analysis doesn't involve probability. There is also another different notion of average case running time where algorithms use randomization to make them faster and expected running time is faster than the worst case running time. These algorithms are analyzed using Randomized Analysis. Examples of these algorithms are Randomized Quick Sort, Quick Select and Hashing. We will soon be covering Randomized analysis in a different post.

Sources:
Berkeley Lecture 35: Amortized Analysis
MIT Lecture 13: Amortized Algorithms, Table Doubling, Potential Method
http://www.cs.cornell.edu/courses/cs3110/2011sp/lectures/lec20-amortized/amortized.htm

Divide & Conquer Algorithm (Recursive strategy)

We solve a problem recursively by applying 3 steps.

1. Divide : Break the problem into several sub problems of smaller size
2. Conquer : Solve the problem recursively.
3. Combine : Combine these solutions to create a solution to the original Problem.

Control Abstraction of Divide & Conquer algorithm.

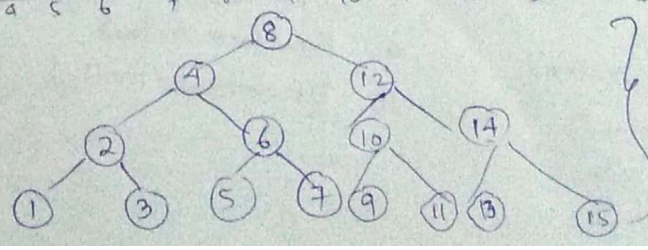
```
Algorithm D and C (P)
{
    if small(P)
        then return S(P)
    else
    {
        divide P into smaller instances P1, P2, P3 ... Pk
        Apply D and C to each sub problem.
        Return combine (D and C(P1) + D and C(P2) + ... + D and C(Pk))
    }
}
```

D & C problems : Binary Search, finding maximum & minimum, Merge sort, Quick sort, Strassen's matrix multiplication.

Binary Search : It is very efficient searching technique which works for sorted list.

```
Binary-Search (A, LB, UB, SITEM)
if (LB > UB) then
    return -1
Set MID = ⌊(LB + UB)/2⌋ <floor>
if (SITEM = A[MID]) then
    return (MID)
else if (SITEM < A[MID]) then
    return (Binary_Search(A, LB, MID-1, SITEM))
else
    return (Binary_Search(A, MID+1, UB, SITEM))
end of if
end of if
```

3	6	8	12	14	17	25	29	31	36	42	47	53	55	62
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15



Min - 1 - O(1)
max - logn - O(logn)
Avg - O(logn)

Maximum comparison depends on height of tree
logn
Approx. 15+1 = 16 elements
$\log_2 16 = 4$ comparisons

```
Algorithm RBinSearch(l,h, key)
{
  if(l==h)
  {
    if(A[l]==key)        } 1 unit
      return l;          } small problem
    else
      return 0; // key element not found.
  }
  else
  { mid = [(l+h)]/2 ;          1 unit
    if(key == A[mid])          1 unit  } large problem
      return mid;
    if(key < A[mid])           1 unit
      return RBinSearch(l, mid-1, key);   } T(n/2) times
    else                                  } calling OR
      return RBinSearch(mid+1, h, key);
  }
}
```

(Divide & Conquer approach)

✓ Best case Time complexity = O(1)
✓ Worst case Time complexity = O(logn)

Recursive Relation = $T(n) \begin{cases} 1 & n=1 \\ T(n/2)+1 & n>1 \end{cases}$

Already done (substitution & recursive Tree method)

Master Theorem: $a=1, b=2, K=0, b^K = 2^0 \Rightarrow 1, a=b^K, p=0$

$$O(n^{\log_2 1} \log^{0+1} n) \Rightarrow O(n^0 \log n) \Rightarrow \Theta(\log n)$$

Binary search takes constant (O(1)) space, meaning that the space taken by the algorithm is the same for any number of elements in the array.
O(1) in case of iterative implementation.
O(logn), in case of recursive implementation - recursion call stack space.

Quick Sort: To sort subarray A[p... r]:

p = low, A[q] = pivot element, r = high

Based on 3-step Divide & Conquer

Divide: Partition A[p... r], into two (possibly empty) subarrays A[p... q-1] and A[q+1 ... r], such that each element in the first subarray A[p... q-1] is ≤ A[q] and A[q] ≤ A[q+1 ... r] (second subarray)

Conquer: Sort the two subarrays by recursive calls to QUICKSORT

Combine: No work is needed to combine the subarrays, because they are sorted in place.

- Perform the divide step by a procedure PARTITION, which returns the index q that marks the position separating the subarrays.

we can take pivot element at last or at first or in the middle.

We recursively perform 3 steps:

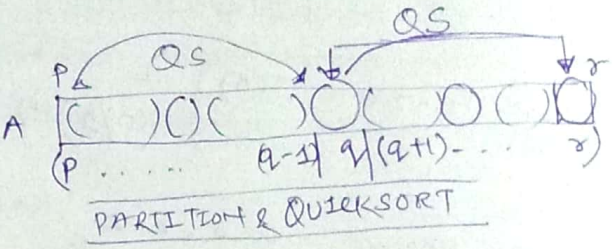
1. Bring the pivot to it's appropiate position such that left of the pivot is smaller and right is grater.
2. Quick Sort the left part.
3. Quick sort the right part.

Quick Sort Algorithm (Lumuto partition scheme)

A = array, P = low, r = high
i = index of smaller element
j = loop variable

```
PARTITION(A,P,r)
{
  x = A[r]
  i = p-1
  for (j=p to r-1)
  {
    if(A[j] ≤ x)
    {
      i=i+1;
      exchange A[i] with A[j]
    }
  }
  exchange A[i+1] with A[r]
  return i+1;
}

QUICKSORT (A,P,r)
{
  if(P<r)
  {
    q = PARTITION (A,P,r)
    QUICKSORT( A, P, q-1)
    QUICKSORT (A,q+1,r)
  }
}
```



PARTITION & QUICKSORT

Performance of Quicksort :

It depends on the particioning of the subarrays.

- If the subarrays are balanced, then QUICKSORT can run as fast as mergesort.
- If they are unbalanced, then it runs slowly.
- If no. of element is less, it performs better as compared to mergesort.

Example:

Partition — Higue

low 0	1	2	3	4	5	6 (high)
10	80	30	90	40	50	70 (pivot)

i = -1 (initially)
j = 0 (scan the element upto high - 1)

pass 1: 10 < 70 (True) Action: i++, swap(arr[i], arr[j]) i = 0, j = 0 (Both are same)

pass 2: 80 < 70 (false) No action i = 0, j = 1

pass 3: 30 < 70 (True) Action, i++ & swap i = 1, j = 2

10	30	80	90	40	50	70

pass 4: 90 < 70 (False) No action i = 1, j = 3

pass 5: $40 < 70$ (True) i++ & swap i=2, j=4

10	30	40	90	80	50	70

pass 6: $50 < 70$ (True) i++ & swap i=3, j=5

10	30	40	50	80	90	70

(Before Pass 7, j becomes 6, so we come out of the loop.) i=3, j=6

We know exchange A[i+1] with A[r] / pivot. , i=3

A[i+1] = A[4]

10	30	40	50	70	90	80

Left Right

Now 70 is brought to it's appropriate position by partition function.

We can begin Quick sorting on the left part.

10	30	40	50

left part

pivot

again apply partition function, 50 = pivot, already in correct position.

again Quicksort, 40 = pivot, similarly 30, 10 become pivot are already in correct position.

So left array is sorted.

90	80

Right part

pivot

80 & 90 swapped to bring pivot to correct position.

80	90

Now, Sorted Array is : | 10 | 30 | 40 | 50 | 70 | 80 | 90 |

Time complexity Analysis of Quick Sort

Best case — Worst case

q = Partition (A, P, r) — n — n

QUICKSORT (A, P, q-1) — T(n/2) — T(0) [0 element on one side]

QUICKSORT (A, q+1, r) — T(n/2) — T(n-1) [n-1 element in other side]

Recursive Relation — T(n) — $2T(n/2) + n$ — $T(n) = T(n-1) + n + T(0)$

Master / Substitution — $O(n \log n)$ — $T(n) = O(n^2)$

Best case : If the partitioning algorithm, give the midpoint as the pivot element then there will be a equal size partition.

Worst case : If the array is already sorted in acending order or decending order or having all equal elements.

Partition Require (n-1) unit of time i.e. n unit of times approx. because j is scan the each array element from 0 to n-1.

Ex 1: | 1 | 2 | 3 | 4 | 5 | 6 | apply Quick sort & find T(n)?

(n-1) Pivot

On this case 6 taken as pivot, but pivot element is already placed in exact/proper position. Then we take 5, 4, 3, 2, 1 as pivot & all placed in proper position.

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

Ex-2: Decending order

(6 5 4 3 2 [1])

No less element compared to 1.
then swap with pivot

so (1 (5 4 3 2 [6]) j & i same
i j j j pivot

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

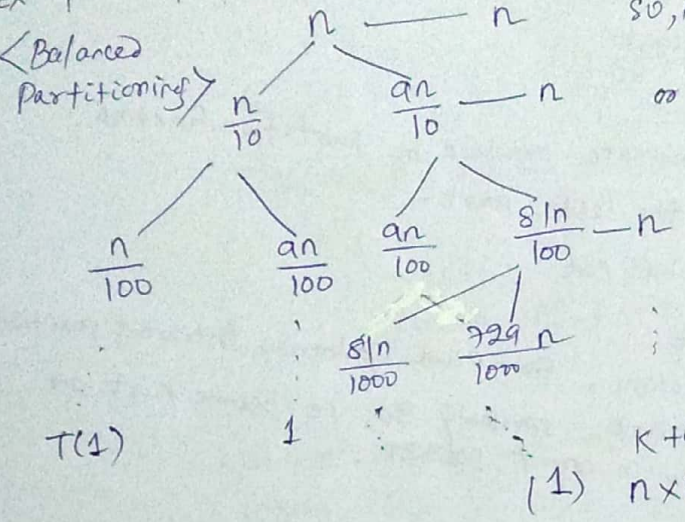
Ex-3: Aooay contains same element

((4 4 4 4 4) 4)
i j j j j pivot

j & i same $T(n) = O(n) + T(n-1) = O(n^2)$

Ex-4: partitioning algorithm, split the input in 1:9 ratio.

<Balanced Partitioning>



So, Here $n \to n/10/9 - n/(10/9)^2 \cdots 1$

or $\left(\frac{9}{10}\right)^K = 1$

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

$K = \log_{10/9} n \cong (\log n)$

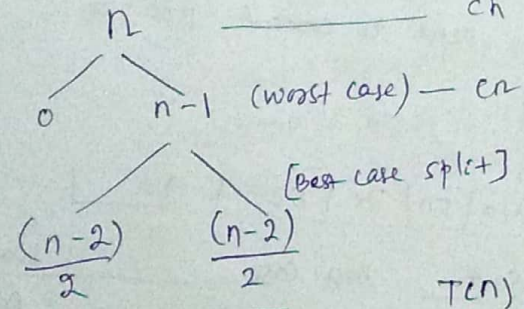
$T(n) = n \times K$ times

$= O(n \log n)$ [Best case Time]

(1) $n \times K$

gate // If the split is 1:9, 1:99, 1:999 ... - Time complexity [$n \log n$]

Ex-5: 1-level - worst case, 2nd level Best case ..., 3rd - worst, 4th - Best case, find T(n)?



so $T(n) = cn + cn + 2T\left(\frac{n-2}{2}\right)$

$2T\left(\frac{n-2}{2}\right) \simeq 2T(n/2)$

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

$T(n) = n + 2T(n/2)$ — $O(n \log n)$

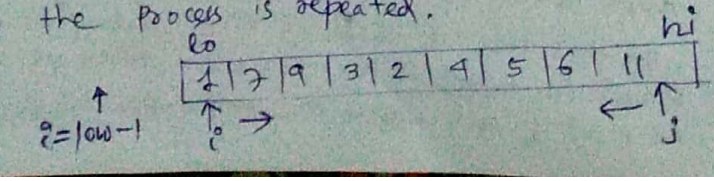
Different versions of Quick sort

There are many different versions of Quicksort that pick pivot in different ways.

1. Always pick first element as pivot
2. Always pick last element as pivot
3. Pick a random element as pivot
4. Pick median as pivot.

Hoare's partition scheme

Hoare's partition scheme works by initializing two indexes that start at two ends, the two indexes move toward each other until an inversion is (A smaller value on left side and greater value on right side) found. When an inversion is found, two values are swapped and the process is repeated.



Space Complexity:

For Best case: It depends the number of levels of height of tree
i.e $O(\log n)$, split equal halves

For worst case: not split equal halves, Here height of tree might goes as big as $O(n)$ i.e $O(n)$

Hoare's partition Algorithm (arr[], lo, hi)

Partition (A, lo, hi)

```
Pivot = A[lo]
i = lo - 1  // initialize left index
j = hi + 1  // initialize right index

do
  i = i+1
while A[i] < pivot

do
  j = j-1
while A[j] > pivot

if i >= j then
  return j

swap A[i] with A[j]
```

Note: Hoare's scheme is more efficient than Lomuto's partition scheme because it does fewer swaps. If input array is already sorted & Time complexity $O(n^2)$ Lomuto's partition scheme is easy to implement as compared to Hoare's scheme.

Randomized QuickSort

In Randomized QuickSort we use random number to pick the next pivot (or we randomly suffle the array). we use randomization to improve the performance of quicksort against those worst-case instances. It randomly picks one element in the sequence, swaps it with the first element, and then calls partition.

Randomized-Partition (A, P, r)

1. i = Random (P, r)
2. exchange A[r] with A[i]
3. return partition (A, P, r)

If the pivot is selected uniformly at random, the expected number of comparisions of randomized version of Quicksort is bounded by $O(n \log n)$

Merge Sort :

It is one of the well-known Divide-and-conquer algorithm.

Divide: Split array down the middle into two sub-sequences, each of size approx. $n/2$.

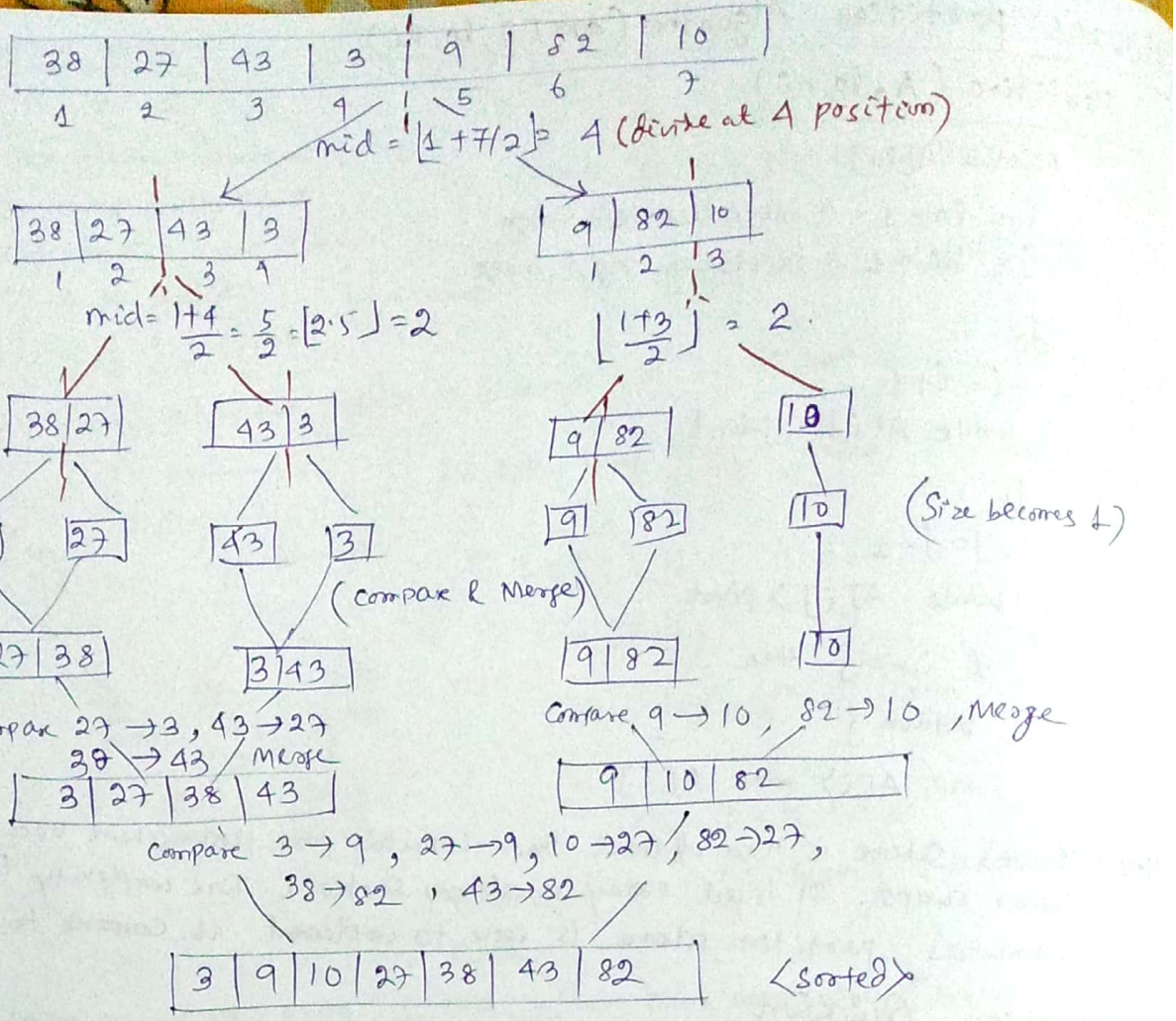
Conquer: Sort each subsequence (by calling mergesort recursively on each)

Combine: Merge the two sorted sub-sequences into a single sorted list.

- So the array is recursively divided into two halves till the size becomes 1. once the size becomes 1, the merge process comes into action and starts merging arrays back till the complete array is merged.

Heart of Merge Sort "Merging"

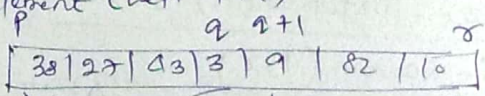
Example:



MERGE (A, P, q, r)

{
$n_1 = q - p + 1$ // Count the no. of elements present in Left Sub-Array
$n_2 = r - q$ // Count the no. of elements present in Right sub-Array
let $L[1 \ldots n_1+1]$ and $R[1 \text{ to } n_2+1]$ be new arrays

for ($i = 1$ to n_1) // copy the array element (Left-part)
  $L[i] = A[p+i-1]$



for ($j = 1$ to n_2) // copy the array element (Right part)
  $R[j] = A[q+j]$

$L[n_1+1] = \infty$ // in program take big number i.e $L[n_1+1] = 9999$
$R[n_2+1] = \infty$ // $R[n_2+1] = 9999$
$i = 1$, $j = 1$
for ($k = p$ to r)
  if ($L[i] \le R[j]$) // if left element is less, copy the left element into array & increase $i+1$
    $A[k] = L[i]$
    $i = i+1$
  else
    $A[k] = R[j]$ // if Right element is less, copy the right element into array & increase $j+1$
    $j = j+1$

Time complexity of Merge Algorithm

n unit time required for copying array element to new array.
n unit of time required for comparision.
$n + n = 2n = O(n)$ ✓

Why ∞ is taken in both the left & Right Sub-Array ?

10	20	30	∞

5	6	9	∞

Compare & merge, $10 \to 5$, $6 \to 10$, then $9 \to 10$, $\infty \to 10$

$20 \to \infty$, $30 \to \infty$

5	6	9	10	20	30

In program, we take the '∞' value as big number.

In Merging procedure Extra Array is required, so it is called as 'out of Place'

```
MERGE-SORT(A,P,r) ——T(n)
{ if P<r
    q = ⌊(P+r)/2⌋  Floor value
    MERGE_SORT(A,P,q) —T(n/2)
    MERGE_SORT(A,q+1,r)—T(n/2)
    MERGE(A,P,q,r) —— O(n) // from merge
}
```

Time

Space complexity

O(n) → extra space for extra Array

O(log n) → Recursive stack space

Total space ⇒ O(n) + O(log n)

⇒ O(n)

Merge sort is efficient if number of array element is more as compared to Quick sort.

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

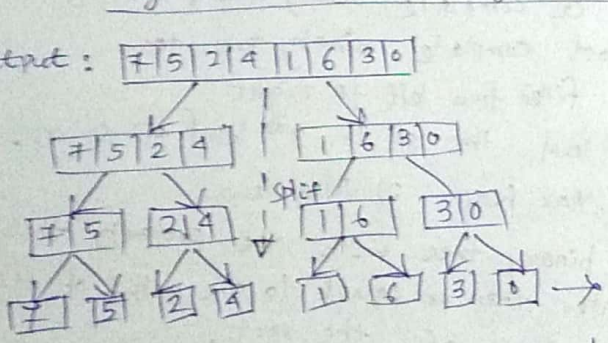
$T(n) = 2T(n/2) + n$

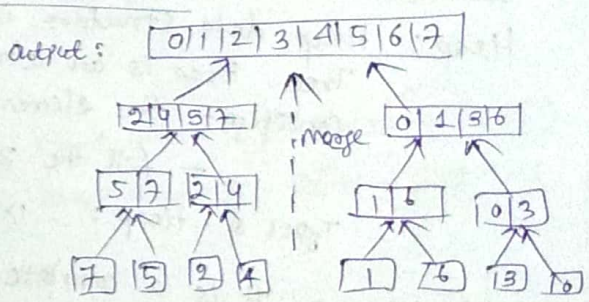
apply Master, Recursive or Substitution — $O(n \log n)$ - Time complexity

If the array is sorted, reverse sorted list the time taken is same.

So Best case, Average case and Worst case is same i.e $O(n \log n)$

High level view of algorithm Merge sort

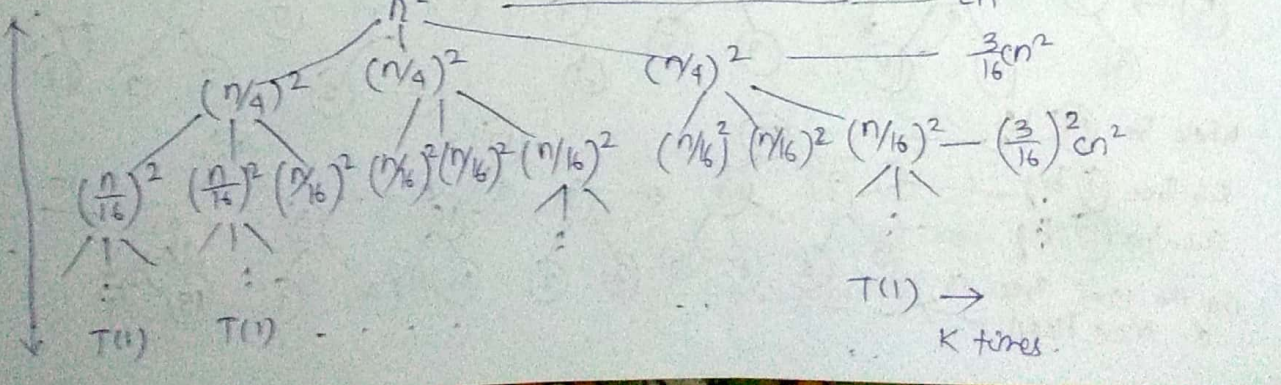




Here the "divide" phase is shown on the left side. It works top-down splitting up the list into smaller sublists.

The "conquer and combine" phases are shown on right. They work bottom up merging sorted lists together into larger sorted list.

Assignment on Recursive Relation : $T(n) = 3T(n/4) + cn^2$



Assume: $n/4^k = 1$, $k = \log_4 n$

Determine the cost at each level of the tree. each levels has 3 times more nodes than the level above.

So no. of nodes at depth k is 3^k. $(k = \log_4 n)$

$$3^k = 3^{\log_4 n} = n^{\log_4 3}$$

$n^{\log_4 3}$ no. of nodes, each contributing cost $T(1)$, for a total cost of $n^{\log_4 3} T(1)$, i.e $\Theta(n^{\log_4 3})$

$$T(n) = cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3})$$

$$= \sum_{k=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^k cn^2 + \Theta(n^{\log_4 3})$$

[for infinite decreasing Geometric Progression : $\left(\frac{1}{1-r}\right)$

$$< \sum_{k=0}^{\infty} \left(\frac{3}{16}\right)^k cn^2 + \Theta(n^{\log_4 3})$$

$$= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3})$$

$$= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3})$$

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

Heap Sort

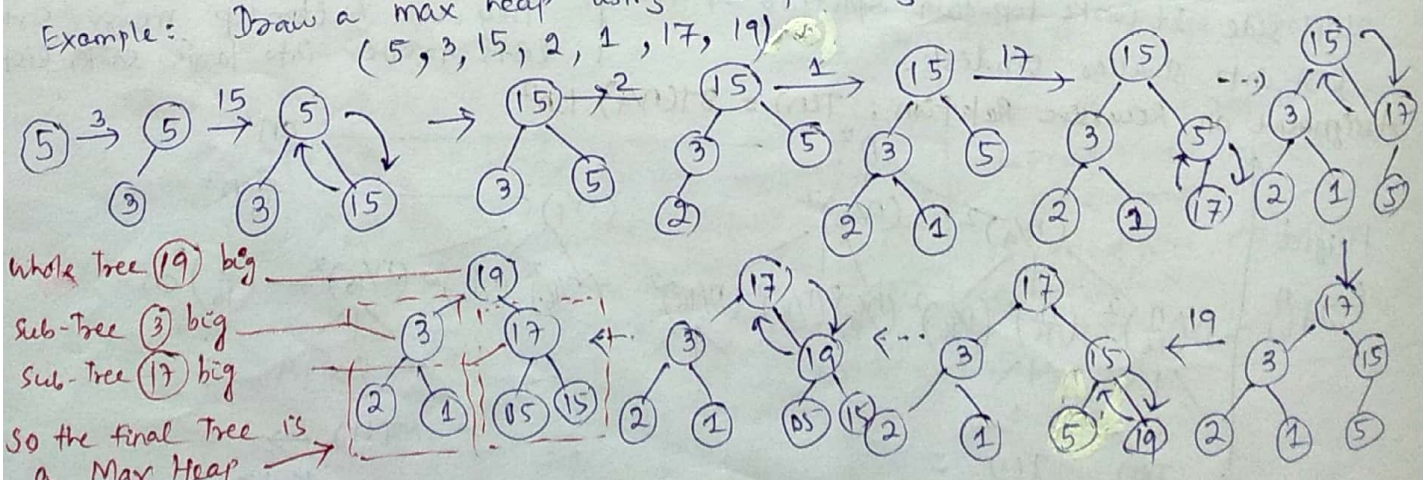
Heap: Heap data structure is a complete binary tree, Ternary Tree or n-ary Tree. Heap is an almost complete binary tree too.

Condition - the element filled from left to right
- fill the 2nd level then go to 3rd level and so on.

Types of Heap: 1) Max heap 2) Min Heap

Max Heap: It is a complete binary tree with the property that the value of each node is greater than or equal to the value of it's children. The largest element is at the root.

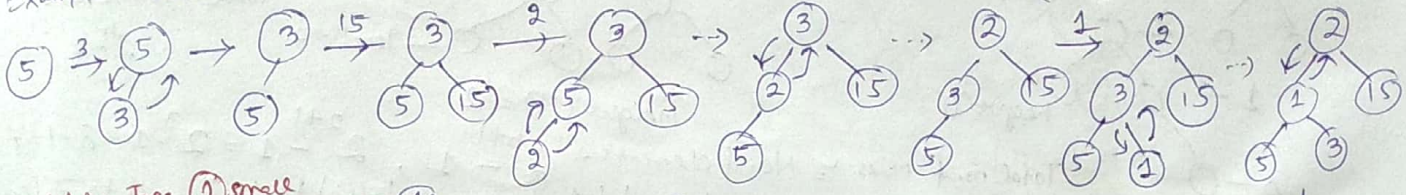
Example: Draw a max heap using the following sequence of Nos (5, 3, 15, 2, 1, 17, 19)

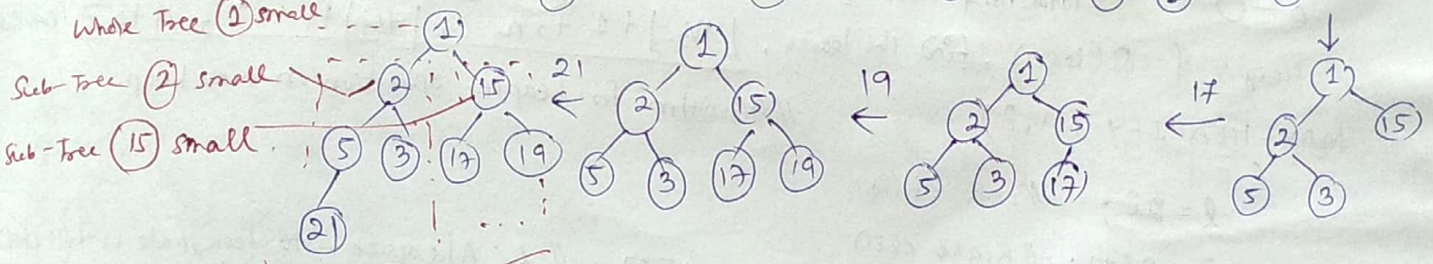


Min Heap : The min heap is a complete binary tree with a property that the value at each node is smaller than or equal to the values of it's children.

Smallest element is at root

Example: Draw a mean heap for the following sequence of no. ⟨5, 3, 15, 2, 1, 17, 19, 21⟩



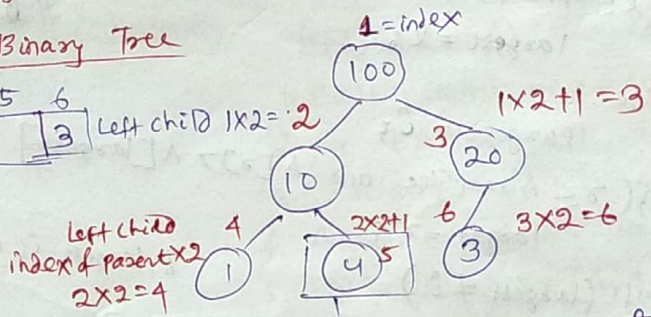


Almost complete binary tree of a Min Heap.

Interpret the Array as a Binary Tree

1	2	3	4	5	6
100	10	20	1	4	3

Left child 1×2 = 2



If we are present in the node (4) having index = 5, find parent ?

$$P = \left\lfloor \frac{\text{index}}{2} \right\rfloor_{\text{Floor}} = \left\lfloor \frac{5}{2} \right\rfloor = \lfloor 2.5 \rfloor \cong 2 \text{ (Parent is at 2 index)}$$

for node (3), index = 6, Parent $= \left\lfloor \frac{6}{2} \right\rfloor = 3$ (Parent is at 3 index)

So $P(i) = \lfloor i/2 \rfloor$

Left(i) = Left child = $2 \times i$, for node (1), parent index = 2, $2 \times 2 = 4$ ($2 \times i$)

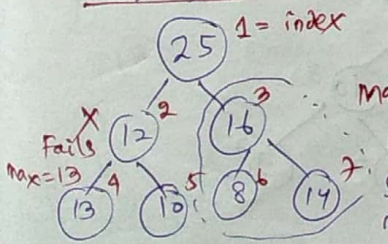
Right(i) = Right child = $2i + 1$, for node (4), parent index = 2, $2i+1 = 2 \times 2 + 1 = 5$

Left(i) = 2 = Binary 10, Left shift, $\frac{(100)}{\text{Binary}} = 4$

P(i) = ?, child = 5 = 101, $P(i) = \lfloor i/2 \rfloor$ = Right Shift, $(10)_{\text{Binary}} = 2 = \text{Parent}$

Right(i) = 2 = Binary 10, Right Shift, $\frac{100}{\text{Binary}}$ $+1 = 4 + 1 = 5$

Ex-1 25, 12, 16, 13, 10, 8, 14, represent in Binary Tree



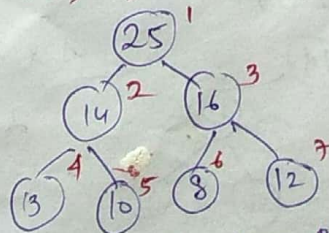
Maximum element not present in the root (sub-Tree) satisfies but 2nd level left side not satisfied. So A.heapsize = 1

A.length = 7

How many elements following Heap property

A.heapsize = 1

Ex:2: 25, 14, 16, 13, 10, 8, 12

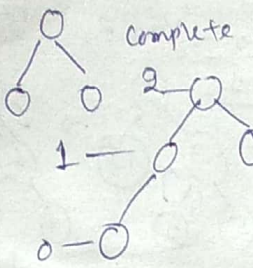


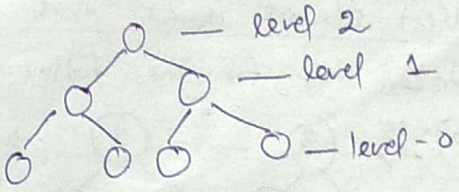
Satisfies all the heap property.

A.heapsize = 7

Array in ascending order → Min Heap., Example: 8, 10, 12, 13, 14

Array in decreasing order → Max Heap example: 14, 13, 12, 10, 8





Height (T) = 2

Complete Binary Tree
No. of leaves = 2^h
total internal nodes = $2^h - 1$

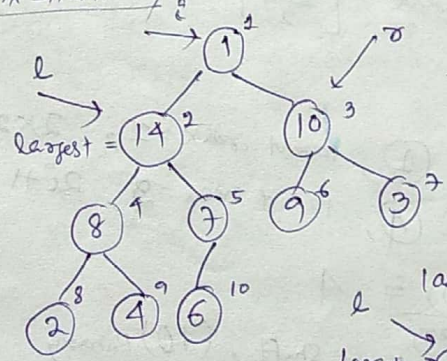
Total no. of nodes = No. of elements = $2^{h+1} - 1$, $2^{2+1} - 1 = 2^3 - 1 = 8 - 1 = 7$

Height of $O(\log n)$, find the leaves, $\lfloor n/2 \rfloor + 1$ to n i.e $\lfloor 7/2 \rfloor + 1$ to 7, 4 to 7 leaves

// Algorithm for heapify operation on max heap.

```
MAX-HEAPIFY (A, i)
{
    l = 2i;        // Left child
    r = 2i+1;      // Right child
    if (l ≤ A.heapsize and A[l] > A[i])      // l ≤ A.heapsize – to check, node is left child or not.
        largest = l;
    else
        largest = i;
    if (r ≤ A.heapSize and A[r] > A[largest])      // r ≤ A.heapsize – to check, node is right child or not
        largest = r;
    if (largest ≠ i)                          // largest is not equal to root
        exchange A[i] with A[largest]
        MAX-HEAPIFY (A, largest)
}
```

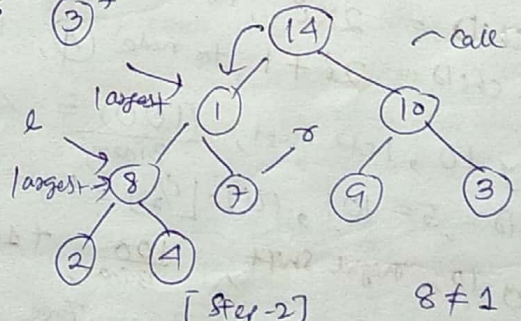
Apply MAX-HEAPIFY, [1, 14, 10, 8, 7, 9, 3, 2, 4, 6]



[Step-1]

1 & 14, 14 largest → largest & Right child, largest = 14

14 ≠ 1, exchange.



→ call MAX-HEAPIFY (A, largest)

1 & 8, 8 largest

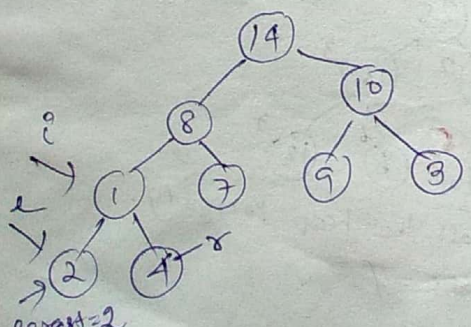
compare 8 with right child. 7

8 = largest

[Step-2]

8 ≠ 1, exchange.

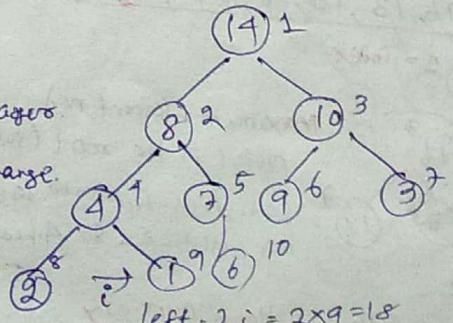
[Step-3]



1 & 2, 2 = largest

2 with 4, 4 is larger

1 ≠ 4, exchange.



left = 2i = 2×9 = 18

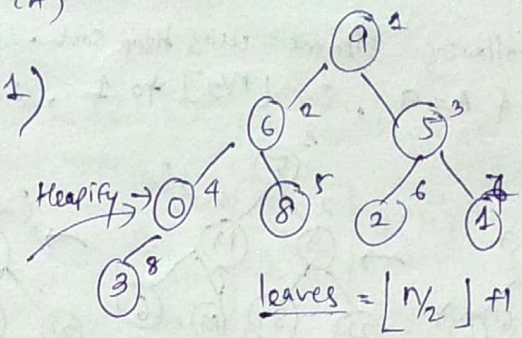
Right = 2i+1 = 2×9+1 = 19

last heapsize = 10

Halt

```
BUILD_MAX_HEAP(A)  // Algorithm to build a max heap (A)
{
  A.heap_size = A.length
  for (i = [A.length/2] down to 1)
    MAX_HEAPIFY(A, i)
}
```

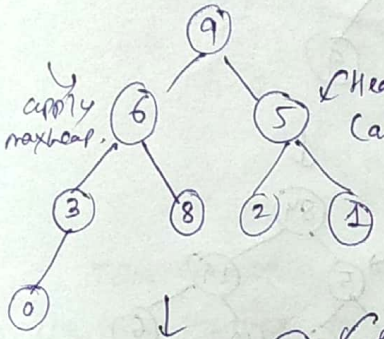
Example: 9 6 5 0 8 2 1 3

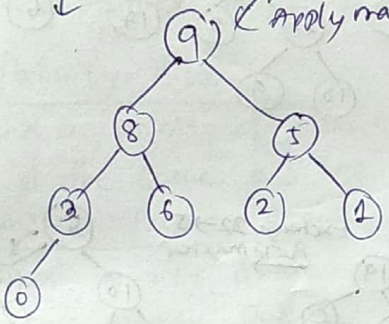


$\lfloor \frac{8}{2} \rfloor + 1$ to 8, 5 to 8.

Non leaves = 1 to $\lfloor \frac{n}{2} \rfloor$, 1 to $\lfloor \frac{8}{2} \rfloor$, 1 to 4

apply Heapify from 4 to 1 node.

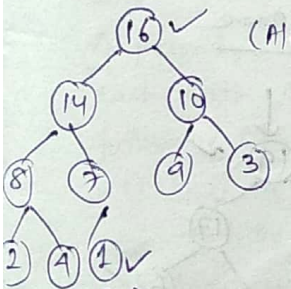


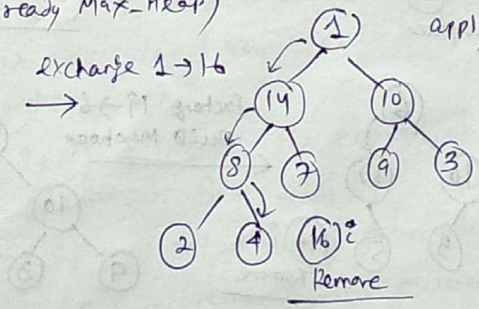


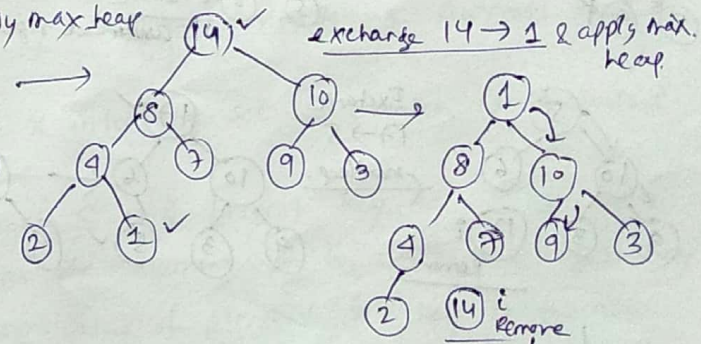
Algorithm for Heap sort using max-heap.

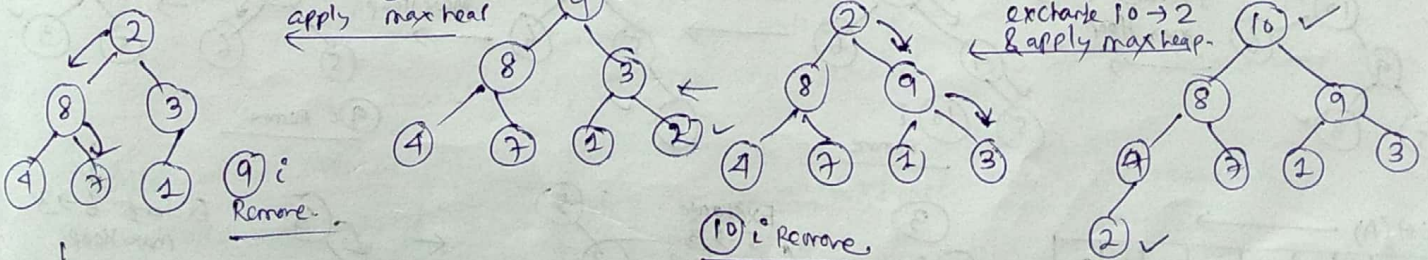
```
HEAPSORT(A)
1. BUILD_MAX_HEAP(A)
2. for i = A.length downto 2
3. exchange A[1] with A[i]
4. A.heap_size = A.heap_size - 1
5. MAX_HEAPIFY(A, 1)
```

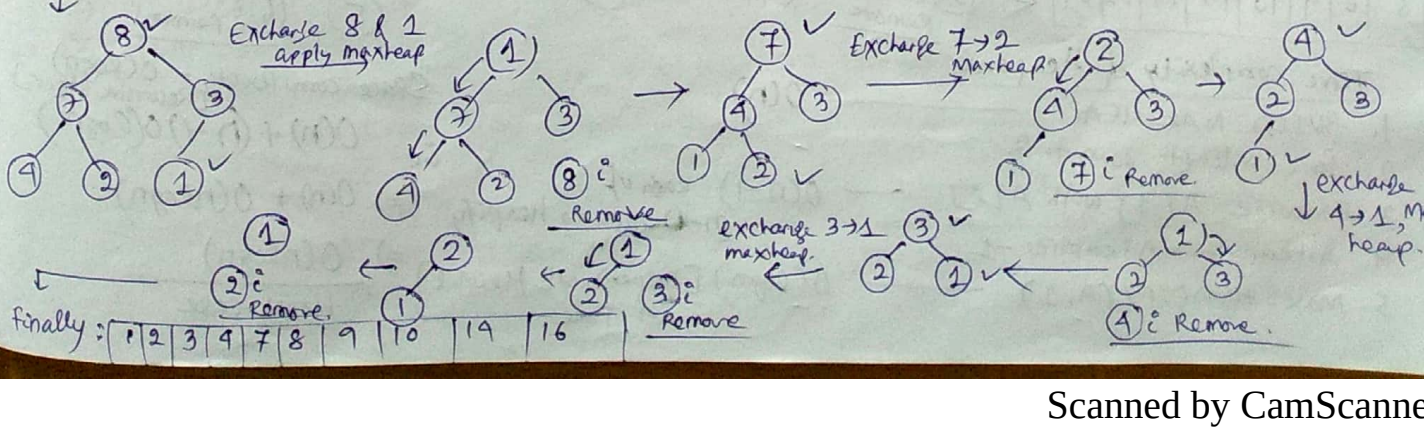
HEAPSORT(A) = {15, 14, 10, 8, 7, 9, 3, 2, 4, 1}











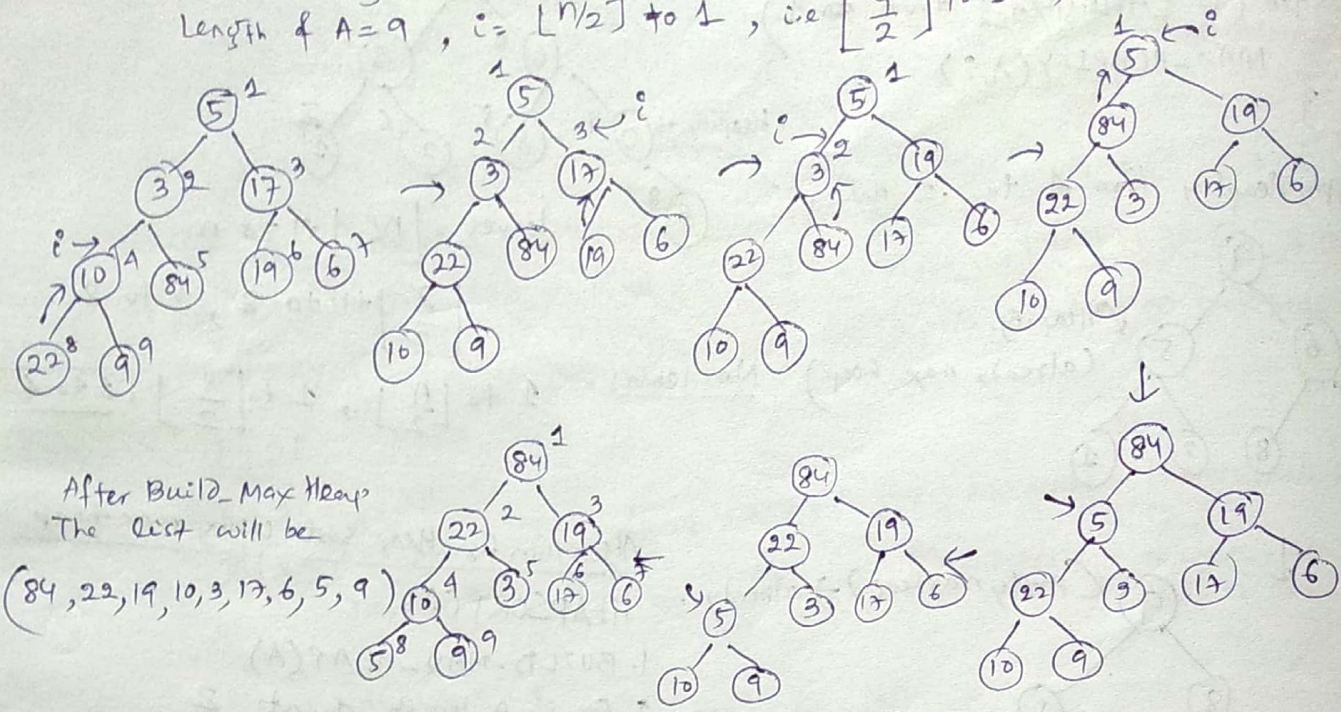
finally :

1	2	3	4	7	8	9	10	14	16

Example - 2 :

Sort the following element using Heap sort. A[5, 3, 17, 10, 84, 19, 6, 22, 9]

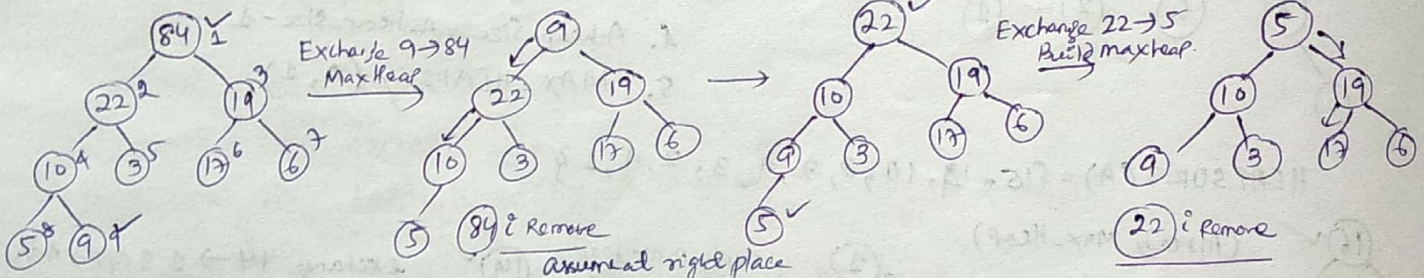
Length of A = 9, i = $\lfloor n/2 \rfloor$ to 1, i.e $\lfloor \frac{9}{2} \rfloor$ to 1, 4 to 1

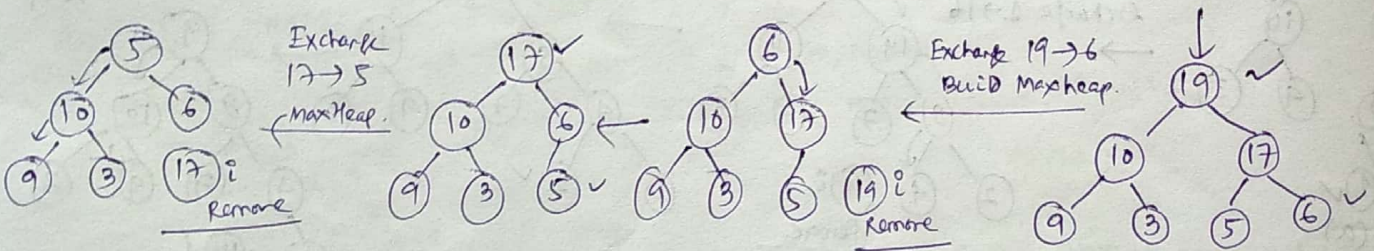


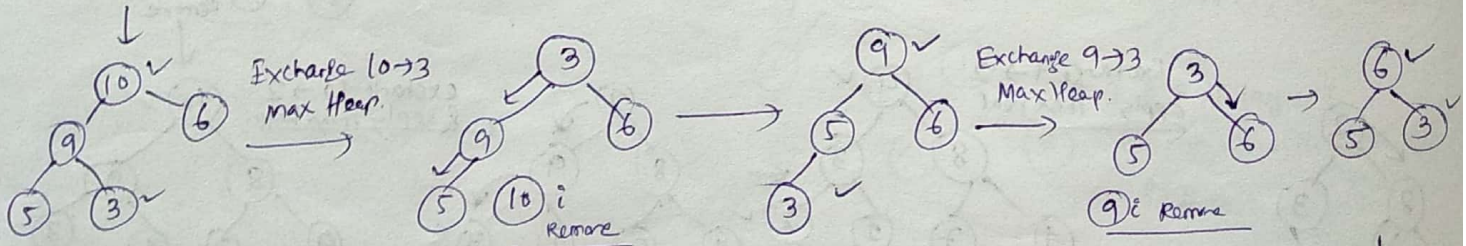
After Build_Max Heap
The list will be

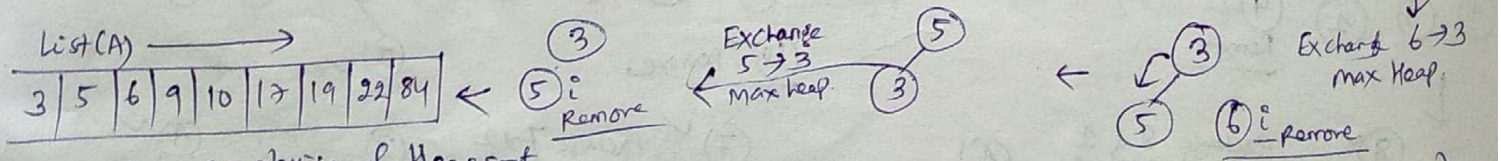
A (84, 22, 19, 10, 3, 17, 6, 5, 9)

Now HeapSort (A)









Time complexity of Heapsort

1. BUILD_MAX_HEAP ——— O(n)
2. for i = A.length down to 2
3. Exchange A[1] with A[i] ——— (n-1) each of (n-1) call to Heapify
4. A.heapsize = A.heapsize - 1
5. Max_HEAPIFY (A, 1) ——— O(logn) [Depends on Height of Tree]

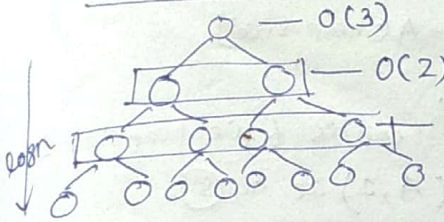
i.e $O(n) + O(n\log n)$

$\Rightarrow O(n) + O(n\log n)$

$\Rightarrow O(n\log n)$ — Worst-Case

Space complexity = O(logn) [Recursive stack]

BUILD-MAX-HEAP



no. of node at height 'h' $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ i.e $\left\lceil \frac{15}{2^{0+1}} \right\rceil = \left\lceil \frac{15}{2} \right\rceil = 8$

The work done by max heap at every level different

so we can write the time complexity as $\sum_{h=0}^{\log n} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$

i.e $\sum_{h=0}^{\log n} \left\lceil \frac{n}{2^h \cdot 2} \right\rceil (C \cdot h)$ (C constant)

$\frac{cn}{2} \sum_{h=0}^{\infty} \left(\frac{h}{2^h}\right)$ ← Implementing Integration & Differentiation Rule.

$\Rightarrow \frac{cn}{2} \sum_{h=0}^{\infty} \left(\frac{h}{2^h}\right) \Rightarrow \frac{1/2}{(1-1/2)^2} \Rightarrow 2$ i.e $\frac{cn}{2} \times 2 \Rightarrow O(n)$

Time complexity of BUILD-MAX-HEAP $\Rightarrow O(n)$

The Heap provides an efficient implementation of the priority queue.

Priority Queue

It depends on priority of elements. The element with highest priority will be moved to the front of the Queue and one with lowest priority will move to the back of the Queue.

2 Kinds of priority Queue

1) Max-priority Queue
2) Min-priority Queue

Max-priority Queue based on max-heap. Min priority Queue based on Min Heap.

A priority Queue is a data structure for maintaining a set of 'S' of elements, each with an associated value called a key. A max-priority queue supports the following operations →

1) Insert (S, x) – insert the element of x into the set S, which is equivalent to the operation $S = S \cup \{x\}$
2) Maximum (S) – returns the element of S with largest key.
3) Extract_Max (S) – removes and returns the element of S with the largest key.
4) Increase-Key (S, x, k) – increase the value of element x's key to the new value K, which is assumed to be at least as large as x's current key value.

→ we can use max-priority queues to schedule jobs on a shared computer.

Min-priority Queue

It supports the operations Insert, Minimum, Extract-Min, Decrease-key.

It is used in event-driven simulator.

(3) HEAP_EXTRACT_MAX (A)

```
if A.heap_size < 1        // heap is at least one element
    error "heap underflow"
max = A[1]
A[1] = A[A.heap_size]
A.heap_size = A.heap_size - 1
MAX_HEAPIFY (A, 1)
return max
```

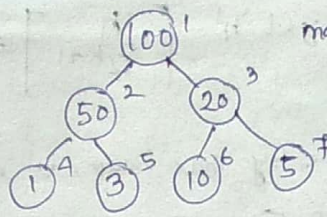
Constant time + O(logn)

Time taken for max-HEAPIFY - Height of tree = O(logn)

we call HEAPIFY at the Root (always)

Space complexity O(logn)

Example:



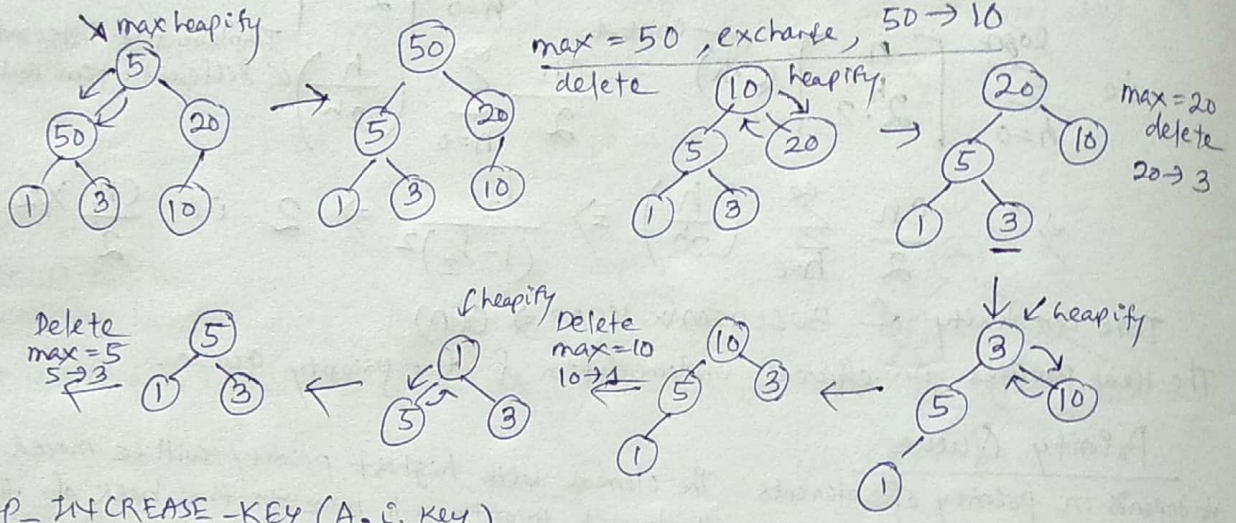
max = 100, delete

A[1] = A[7] ie A[1] = A[heap-size]

so (5) will be returned.

no. of element on the heap is 6, ie heap-size-1

then call Heapify (A, 1) ie at (5)



(A) HEAP_INCREASE-KEY (A, i, key)

if key < A[i] // increase the key not decrease

error "new key is smaller than current key"

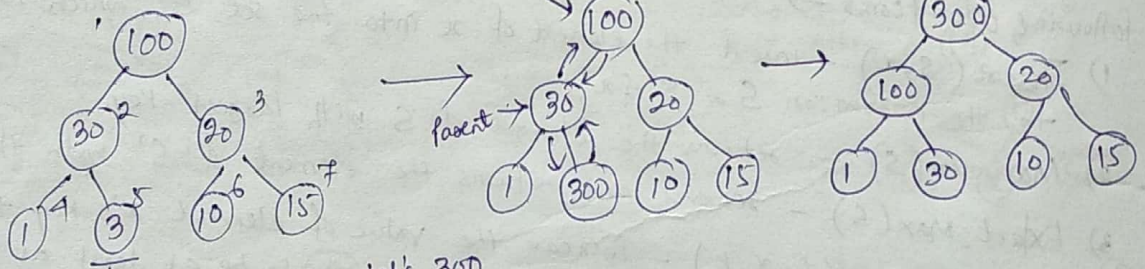
A[i] = key

while i > 1 and A[PARENT(i)] < A[i] → (if parent is less exchange) Never going beyond the root

exchange A[i] with A[PARENT(i)]

i = PARENT(i) // check with all the parent // Parent = i/2

Example:

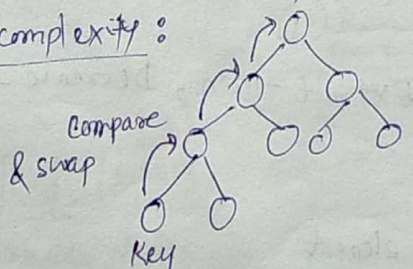


increase the value let's 300

ie (A, 5, 300)

index

Time complexity:

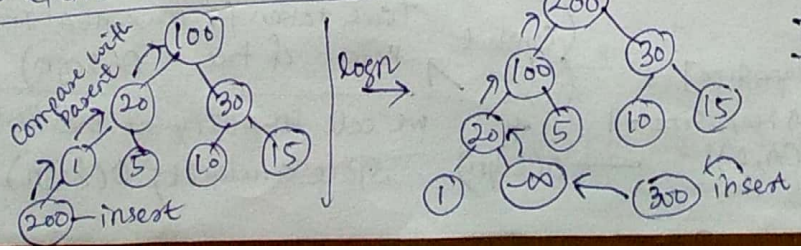


height = logn → O(logn) & Same as Decrease-Key.

Space complexity (Iterative) ie not recursive

so if is O(1), of recursive O(logn)

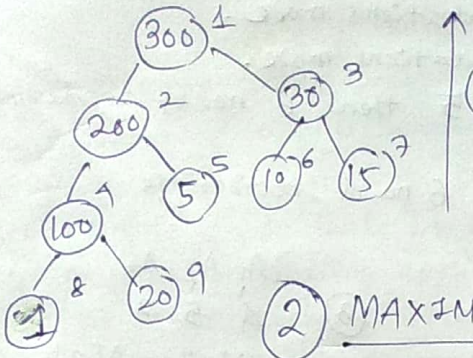
Insert element into max-heap:



Time complexity = O(logn)

−∞ = very very less element

increase −∞ to 300



(logn) time

(1) MAX-HEAP_INSERT (A, key)

A.heap-size = A.heap-size + 1
A[A.heap-size] = −∞
HEAP_INCREASE-KEY (A, A.heapsize, key)

(2) MAXIMUM (A)

return A[1] — Running time Θ(1)

Note:

	Find max	Delete max	Insert	Increase Key	Decrease Key	
Max Heap	O(1)	O(logn)	O(logn)	O(logn)	O(logn)	← call max-heapify

Find min	Search random element	Delete random element
O(n)	O(n)	O(n) + for build the heap again O(n) i.e O(n+n) = O(n)

Lower Bounds for Sorting:

A lower bound of a problem is the least time complexity required for any algorithm. Lower bound proves that we can not hope for a better algorithm.

The problem of sorting can be viewed as:

Input: A sequence of n numbers $\langle a_1, a_2, a_3 \ldots a_n \rangle$

Output: A permutation (reordering)

A sorting algorithm is comparison based if it uses comparison operators to find the order between two numbers. Comparison sorts can be viewed in terms of decision trees. A decision tree is a full binary tree that represents the comparisons between elements that are performed by a particular sorting algorithm.

The execution of the sorting algorithm corresponds to tracing a path from the root of the decision tree to a leaf.

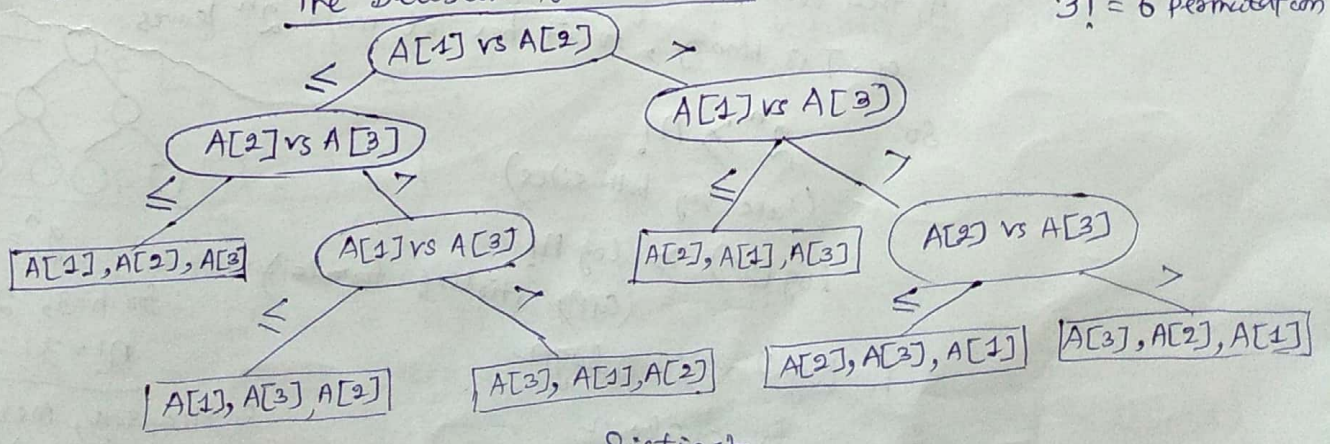
→ At each internal node, a comparison $a_i <= a_j$ is made.

→ Left subtree $a_i <= a_j$

→ Right subtree $a_i > a_j$

The Decision Tree Model (Insertion sort with n = 3)

3! = 6 permutation



Note: Here all the input elements are distinct.
For n elements n! permutations must appear.

If $n=2$, $n! = 2 \times 1 \Rightarrow$ 2 no. of permutations made.
If $n=3$, $n! = 3 \times 2 \times 1 \Rightarrow$ 6 no. of permutations made.

Example: If we take 3 elements i.e 3, 4, 5 then 6 no. of reorderings are:

① 345
② 354
③ 453
④ 435
⑤ 534
⑥ 543

} total 6 no. of reorderings

② $A_1\ A_2\ A_3$: 3 4 5
$A[1]$ vs $A[2]$: $3 \leq 4$ (yes)
$A[2]$ vs $A[3]$: $4 \leq 5$ (yes)
(Arrange in ascending)
$A[1], A[2], A[3]$

④ $A_1\ A_2\ A_3$: 4 3 5
$A[1]$ vs $A[2]$: $4 > 3$ (yes)
$A[1]$ vs $A[3]$: $4 \leq 5$ (yes)
$A[2], A[1], A[3]$

② $A_1\ A_2\ A_3$: 3 5 4
$A[1]$ vs $A[2]$: $3 \leq 5$ (yes)
$A[2]$ vs $A[3]$: $5 > 4$ (yes)
$A[1]$ vs $A[3]$: $3 \leq 4$ (yes)
$A[1], A[3], A[2]$

⑤ $A_1\ A_2\ A_3$: 5 3 4
$A[1]$ vs $A[2]$: $5 > 3$ (yes)
$A[1]$ vs $A[3]$: $5 > 4$ (yes)
$A[2]$ vs $A[3]$: $3 \leq 4$ (yes)
$A[2], A[3], A[1]$

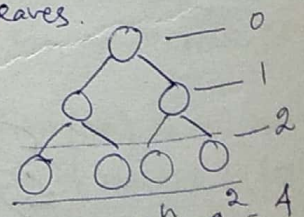
③ $A_1\ A_2\ A_3$: 4 5 3
$A[1]$ vs $A[2]$: $4 \leq 5$ (yes)
$A[2]$ vs $A[3]$: $5 > 3$ (yes)
$A[1]$ vs $A[3]$: $4 > 3$ (yes)
$A[3], A[1], A[2]$

⑥ $A_1\ A_2\ A_3$: 5 4 3
$A[1]$ vs $A[2]$: $5 > 4$ (yes)
$A[1]$ vs $A[3]$: $5 > 3$ (yes)
$A[2]$ vs $A[3]$: $4 > 3$
$A[3], A[2], A[1]$

From the above, each of the $n!$ permutations on n elements must appear as one of the leaves of the decision tree for the sorting algorithm to sort properly.

Theorem: Any decision Tree T for sorting n elements has height $\Omega(n \log n)$ [so any comparison sort algorithm requires $\Omega(n \log n)$ compares in the worst case]

Proof:
1) $n!$ permutations for n elements
2) Let x be the minimum no. of comparisions in a sorting algorithm. The maximum height of the decision tree would be x. A tree with maximum height x has at most 2^x leaves or T is binary, so it has at most 2^h leaves.



$2^h = 2^2 = 4$ leaves
for $h=3$, $2^h = 2^3 = 8$
$n! = 3! = 6$

So $2^h \geq n!$
(take log both sides)
$\log(2^h) \geq \log n!$
(apply stirling's formula)
$h \log_2 2 \geq n \log n$
$h \geq n \log n$
$h = \Omega(n \log n)$ [proved]

So Heapsort, Mergesort are having $(n \log n)$ Time complexity at worst case.

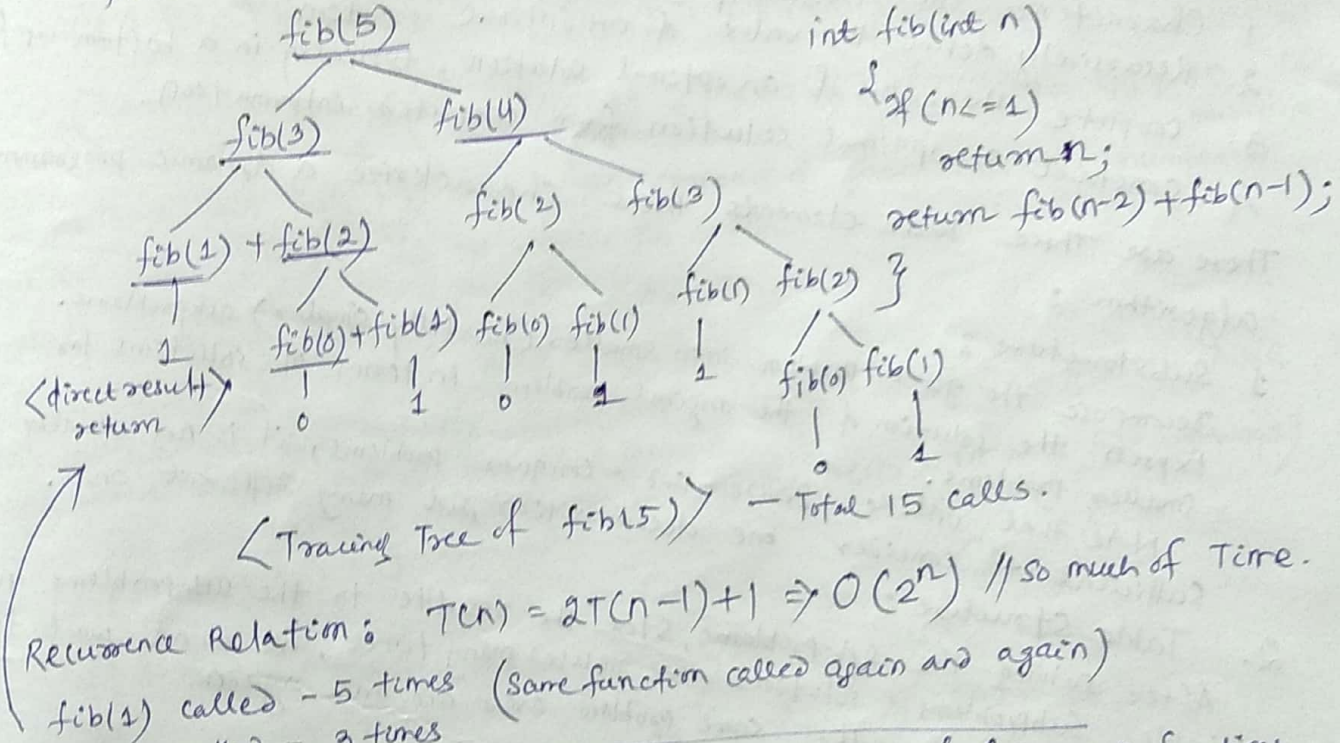
Dynamic Programming (Richard Bellman)

Dynamic Programming solves problem by combining the solutions to subproblem
Dynamic programming applies when sub problem overlap.
Dynamic Programming algorithm solves each subproblem just once and then saves it's answer in table, so that It can be used in future.
Dynamic Programming mainly applied to optimization problems.
Optimization means each solution has a value and we wish to find a solution with minimum or maximum value.
Dynamic Programming usually use/solved by Recursive method/formula.
It follows Principle of optimality (Problem can be solved by taking decisions) Every stage we take a decision.

Example: Fibonacci Series (0 1 1 2 3 5 ...)

$$fib(n) = \begin{cases} 0 & \text{if } n=0 \\ 1 & \text{if } n=1 \\ fib(n-2)+fib(n-1) & \text{if } n>1 \end{cases}$$

Recursive function



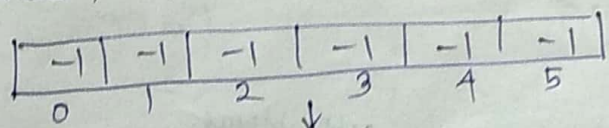

```
int fib(int n)
{ if (n<=1)
    return n;
  return fib(n-2)+fib(n-1);
}
```

<Tracing Tree of fib(5)> — Total 15 calls.

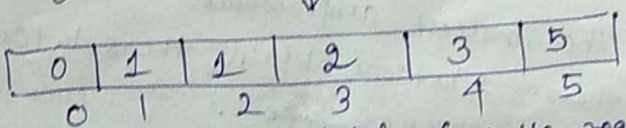
Recurrence Relation: $T(n) = 2T(n-1)+1 \Rightarrow O(2^n)$ // so much of Time.

fib(1) called - 5 times (Same function called again and again)
fib(2) called - 3 times

Reduce the function call & time by storing the results of fibonacci functions So that once fib(2) called, should not call again.



initially fill -1, nothing is known



Store it in an array.

If we apply this, total 6 calls required (underlined calls in tree)

fib(n) = n+1 calls. i.e $\simeq O(n)$

Storing the results called — memoization.

Here we see that 2^n reduce to n (Exponential to linear)

Memorization follows Top-down approach.

Same Problem with iterative method. [Tabulation Method]

fib(5), n=5

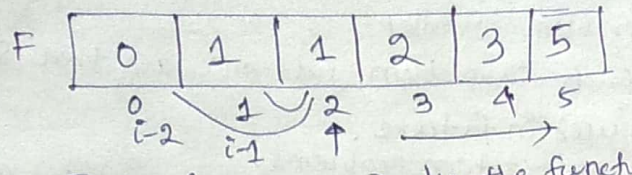


Table is generated by the functions.

Here Filling is done from smaller value onwards.

Called [Bottom up approach]

Mostly Tabulatin Method is used in Dynamic Programming

```
int fib(int n)
{ if(n<=1)
    return n;
  F[0]=0, F[1]=1;
  for(int i=2; i<=n; i++)
  {
    F[i]= F[i-2]+F[i-1];
  }
  return F[n];
}
```

Iterative Method.

When developing a dynamic-programming algorithm, we follow a sequence of 4 steps.

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

There are three basic elements that characterize a dynamic programming algorithm:

1. Substructure:
   Decompose the given problem into smaller (hopefully simpler) subproblems. Express the solution of the original problem in terms of solutions for smaller problems.
   Note that unlike divide-and-conquer problems, it is not usually sufficient to consider one decomposition, but many different ones.
2. Table Structure:
   After solving the sub-problems, store the results to the sub-problems in a table. Subproblems solutions are reused many times, we do not want to repeatedly solve the same problem over and over again.
3. Bottom-Up Computation -
   Using table (or something), combine solutions of smaller subproblems to solve larger subproblems, and eventually arrive at a solution to the complete problem.

   Bottom-up means:
   i) Start with the smallest subproblems.
   ii) Combining theirs solutions obtain the solutions to subproblems of increasing size.
   (iii) Until arrive at the solution of the original problem.

Examples:
1) Matrix-chain multiplication problem.
2) Longest common subsequence problem.
3) Assembly-line scheduling problem.

Matrix-Chain Multiplication

The main objective of matrix chain multiplication problem is to find the optimal parenthesization of a chain of matrices to be multiplied such that the number of scalar multiplications are minimized.

Rule for matrix multiplication : The no. of column of first matrix = no. of rows of 2nd matrix

$$\underset{5\times4}{[\,A\,]} \quad \underset{4\times3}{[\,B\,]} = \text{result matrix} = \underset{5\times3}{[\,C\,]} = 15 \text{ elements}$$

Each element in 'c' matrix require 4- multiplication, Total multiplication = 15X4=60

For performing chain matrix multiplication i.e $A_1 \times A_2 \times A_3 \cdots A_n$ we should use parenthesis.

Example: Concider 3 matrices of $A_{5\times3}$ $B_{3\times4}$ $C_{4\times6}$

Possible way of multiplication by using parenthesis

$((A\,B)\,C) = ((A_{5\times3}\,B_{3\times4})\,C_{4\times6}) = (5\times3\times4) + (5\times4\times6) = 60+120 = 180$ multiplication (Result 5X4)

$(A(B\,C)) = (A_{5\times3}(B_{3\times4}\,C_{4\times6})) = (5\times3\times6) + (3\times4\times6) = 90+72 = 162$ multiplication (result 3X6)

So we have seen that by changing the sequence of parenthesis the number of multiplication charges. So we require the optimal solution.

Example:

A_1	A_2	A_3	A_4
5X4	4X6	6X2	2X7

Aim: take a pair and multiply, which pair I have to select such that the total cost of multiply all of them is minimum.

Use parenthesize : $A_1A_2A_3A_4$ can be fully parenthesized in 5 distinct ways)

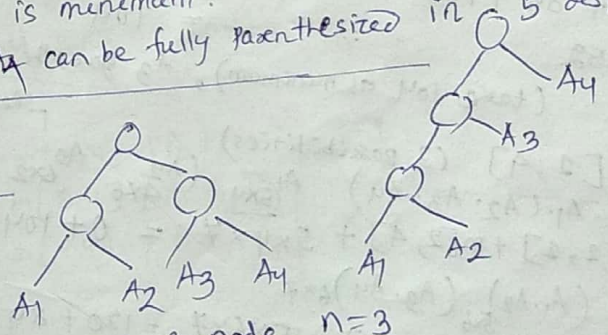
$(((A_1 \cdot A_2)\,A_3)\,A_4)$

$((A_1 \cdot A_2) \cdot (A_3 \cdot A_4))$

$(A_1(A_2(A_3\,A_4)))$

$(A_1((A_2\,A_3)\,A_4))$

$((A_1\,(A_2\,A_3))\,A_4)$



it require 3-node, n=3

Formula $= \dfrac{(2n)!}{(n+1)!\,n!} = \dfrac{(2\times3)!}{(3+1)!\,3!} = \dfrac{6!}{4!\times3!}$

$\Rightarrow \dfrac{6\times5\times4!}{4!\times3\times2\times1} = 5$ distinct ways.

The matrix chain multiplication states as follows:

given a chain $\langle A_1, A_2 \cdots A_n \rangle$ of n matrices for $(i=1 \ldots n)$ and matrix A_i has dimensions $P_{i-1} \times P_j$, fully Parenthesizing the product in a way that minimizes the no. of scalar multiplications. Let $M[i,j]$ = minimum no. of scalar multiplications needs to compute matrix $A_i \cdots A_j$ i.e $(A_i\,A_{i+1} \cdots A_j) = (A_i\,A_{i+1} \cdots A_k)(A_{k+1} \cdots A_j)$

if $i=j$, then $[i,j] = 0$, otherwise $M[i,j] = M[i,k] + M[k+1,j] + P_{i-1} \cdot P_k \cdot P_j$

where $k = i$ to $j-1$ and $i<k$

Table Method & Bottom-up Approach. P_0, P_1, P_2, P_3, P_4 – dimensions

A_1 . A_2 . A_3 . A_4

5×4 4×6 6×2 2×7

P_0 P_1 P_1 P_2 P_2 P_3 P_3 P_4

A_1 A_2 A_3 A_4

$M[1,1]=0$ $M[2,2]=0$ $M[3,3]=0$ $M[4,4]=0$

$M[1,2]$ $A_1 \cdot A_2$ 5×4 4×6 $5\times4\times6 = 120$

$M[2,3]$ $A_2 A_3$ 4×6 6×2 $4\times6\times2 = 48$

$M[3,4]$ $A_3 A_4$ 6×2 2×7 $6\times2\times7 = 84$

$M[1,3]$ (2 possibilities) — find minimum.

min { i.e $A_1 \cdot (A_2 \cdot A_3)$ 5×4 4×6 6×2 ; $(A_1 \cdot A_2) A_3$ 5×4 4×6 6×2 }

$m[1,1] + m[2,3] + 5\times4\times2$ (dimensions)

$= 0 + 48 + 40$

$= 88$

$m[1,2] + m[3,3] + 5\times6\times2$

$= 120 + 0 + 60$

$= 180$

(take minimum, 88), A_1 gives the result

m	1	2	3	4
1	0	120	88	158
2		0	48	104
3			0	84
4				0

Table.1

$m[2,4]$ (2 possibilities) — find minimum.

min { $A_2 \cdot (A_3 \cdot A_4)$ 4×6 6×2 2×7 ; $(A_2 \cdot A_3) \cdot A_4$ 4×6 6×2 2×7 }

$m[2,2] + m[3,4] + 4\times6\times7$

$0 + 84 + 168$

$= 252$

$m[2,3] + m[4,4] + 4\times2\times7$

$48 + 0 + 56$

$= 104$

(take 104 as minimum), A_3 gives the result

s	1	2	3	4
1		1	1	3
2			2	3
3				3
4				

[the one which giving the minimum]

$m[1,4]$ (3 possibilities) Result [4×7]

$A_1 \cdot (A_2 \cdot A_3 \cdot A_4)$ A_1 5×4 $(A_2$ 4×6 A_3 6×2 $A_4)$ 2×7

min { $m[1,1] + m[2,4] + 5\times4\times7 = 0 + 104 + 140 = 244$ $(A_1 \cdot A_2)_{5\times6} \cdot (A_3 \cdot A_4)_{6\times7}$

$m[1,2] + m[3,4] + 5\times6\times7 = 120 + 84 + 210 = 414$ $(A_1 \cdot A_2 \cdot A_3)_{5\times2} \cdot A_4$ 2×7

$m[1,3] + m[4,4] + 5\times2\times7 = 88 + 0 + 70 = 158$ (minimum) }

A_3 gives the result

↑ at 3 split

$A_1 \cdot A_2 \cdot A_3 \cdot A_4$

$(A_1 \cdot A_2 \cdot A_3)(A_4)$

↑ at 1, split

$((A_1) \cdot (A_2 \cdot A_3))\ A_4$

at 2, split, no need.

From the above, state the formula

$$m[i,j] = \min \{ m[i,k] + m[k+1, j] + P_{i-1} \cdot P_k \cdot P_j \}$$

Time complexity of Matrix Chain multip^n is

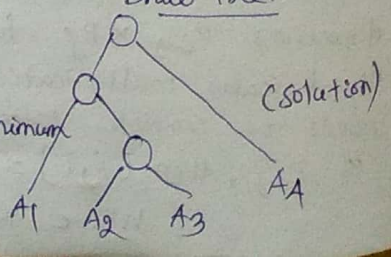
consider Table 1., 1+2+3+.. n-1, diagonal '0'

$\frac{n(n-1)}{2} \doteq n^2$

Each element we get by calculating all & find the minimum takes at most 'n' times.

$n^2 \cdot n = n^3 = O(n^3)$

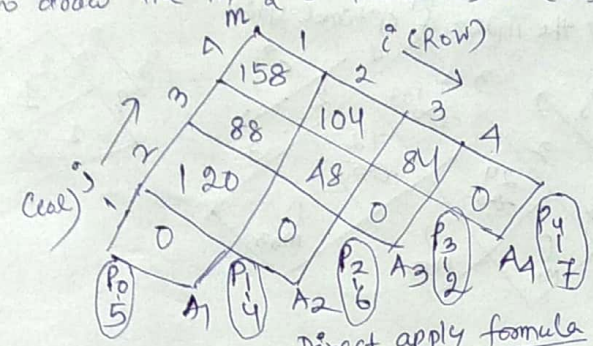
Draw Tree.

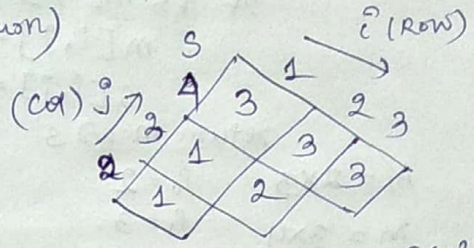


Time complexity for multiplying n matrix is n^3
& solving the parenthesis problem is also n^3
So Time complexity of matrix multiplication = $O(n^3)$

Space complexity = $O(n^2)$

we also draw the M & S table as : (Just Twin)





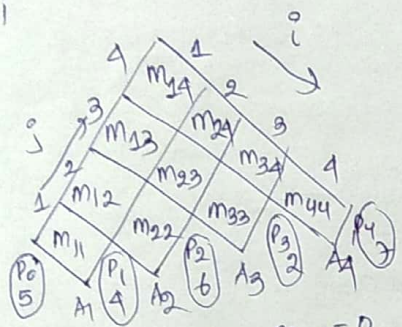
Note: Algorithm requires $O(n^2)$ space to store m and s tables.

Direct apply formula :

$m[1,2] = m[1,1] + m[2,2] + P_0 \cdot P_1 \cdot P_2 = 0+0+5\times4\times6 = 120$

$m[2,3] = m[2,2] + m[3,3] + P_1 \cdot P_2 \cdot P_3 = 0+0+4\times6\times2 = 48$

$m[3,4] = m[3,3] + m[4,4] + P_2 \cdot P_3 \cdot P_4 = 0+0+6\times2\times7 = 84$



$m_{11} = m_{22} = m_{33} = m_{44} = 0$

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

$= \min \begin{cases} 0 + 48 + 5\times4\times2 = 0+48+40 = (88) \checkmark \text{ At } A_1 \\ 120 + 0 + 5\times6\times2 = 120+0+60 = 180 \end{cases}$

$m[2,4] = \min \begin{cases} m[2,2] + m[3,4] + P_1 P_2 P_4 \\ m[2,3] + m[4,4] + P_1 P_3 P_4 \end{cases} \begin{matrix} K=2 \\ K=3 \end{matrix}$

$= \min \begin{cases} 0 + 84 + 4\times6\times7 = 252 \\ 48 + 0 + 4\times2\times7 = (104) \checkmark \end{cases}$ at A_3, minimum

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

$= \min \begin{cases} 0 + 104 + 5\times4\times7 = 244 \\ 120 + 84 + 5\times6\times7 = 414 \\ 88 + 0 + 5\times2\times7 = (158) \checkmark \end{cases}$ at A_3, minimum

Algorithm

MATRIX_CHAIN_ORDER(P) <To find m & S table>

```
n = P.length - 1
let m[1..n, 1..n] and s[1..n-1, 2..n] be new tables
for i = 1 to n
    m[i, i] = 0          // one matrix no multiplying
for l = 2 to n           // l is the chain length
    for i = 1 to n-l+1
        j = i + l - 1    // end index for chain of length l
        m[i, j] = ∞
```

for K = i to j-1

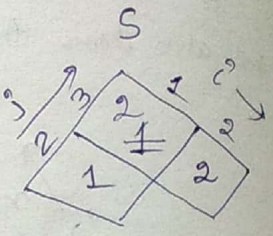
$q = m[i,K] + m[K+1, j] + P_{i-1} P_K P_j$

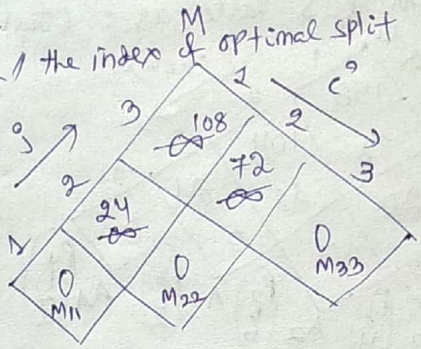
if $q < m[i,j]$

$m[i,j] = q$

$s[i,j] = K$ // the index of optimal split

return m and s





Example $A_1 = 2 \times 3$ $P_0 = 2$

$A_2 = 3 \times 4$ $P_1 = 3$

$A_3 = 4 \times 6$ $P_2 = 4$

$P_3 = 6$

① n = 3 i.e P-1 (4-1) = 3

② i = i → 3

③ m[i, i] = 0

④ for l ← 2 to 3

⑤ for i ← 1 to 2

⑥ j = 2

⑦ m[1, 2] = ∞

⑧ K → 1 to 1

⑨ $q = m[1,1] + m[2,2] + P_0 P_1 P_2$; $2 \times 3 \times 4 = 24$

⑩ $24 < \infty$

⑪ m[1, 2] = 24

⑫ s[1, 2] = 1

⑬ return 24 ⑭ return 1

for i = 2

⑥ j = 2 + 2 - 1 = 3

⑦ M[2, 3] = ∞

⑧ K ← 2 to 2

⑨ $q = M[2,2] + m[3,3] + P_1 P_2 P_3$; $3 \times 4 \times 6 = 72$

⑩ $72 < \infty$

⑪ M[2, 3] = 72

⑫ s[2, 3] = 2

⑬ Return 72, Return 2

for l = 3

⑤ i ← 1 to 1

⑥ j = 3

⑦ M[1, 3] = ∞

⑧ K ← 1 to 2

⑨ $q = M[1,1] + M[2,3] + P_0 P_1 P_3$; 72 ; $2 \times 3 \times 6 = 36 \rightarrow 108$

(10) $108 < \infty$

(11) $M[1,3] = 108$

(12) $s[1,3] = 1$

(13) return 108 , return 1

$2 \times 4 \times 6 = 48 \rightarrow 72$

$K = 2$

(9) $q = M[1,2] + M[3,3] + (P_0 P_2 P_3)$ (24, 90)

$72 < 108$

(10) $M[1,3] = 72$

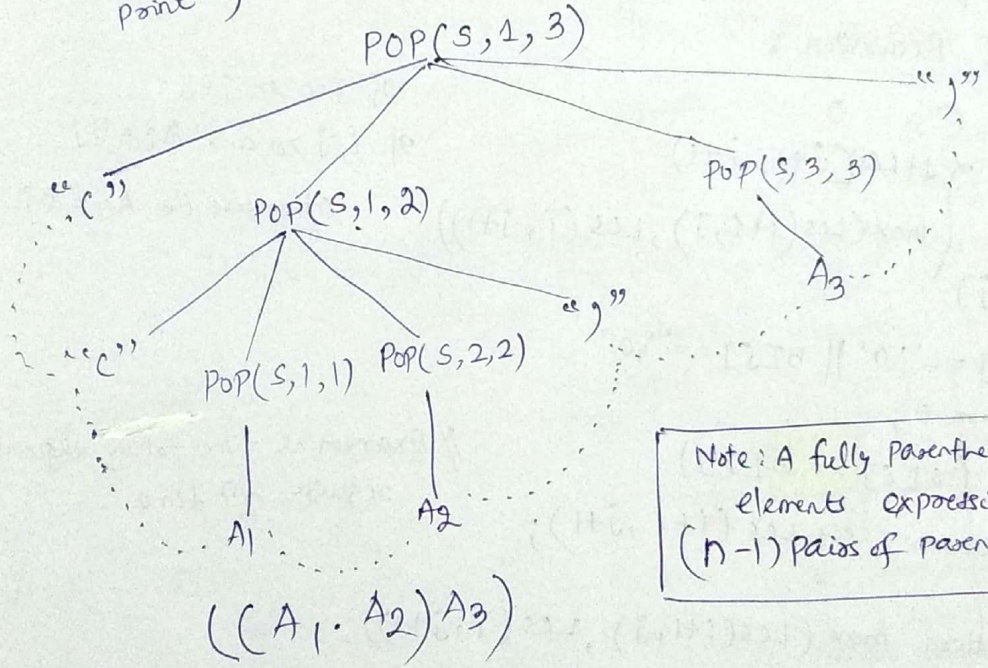
(11) $s[1,3] = 2$

(12)

(13) return 72 , return 2

PRINT_OPTIMAL_PARENS (S, i, j) [Printing the optimal parenthesis]

```
1. if i == j
       print "A"i
2.
   else print "("
3.     PRINT_OPTIMAL_PARENS (S, i, s[i,j])
4.     PRINT_OPTIMAL_PARENS (S, s[i,j]+1, j)
5.     print ")"
6.
```



$((A_1 \cdot A_2) A_3)$

Note: A fully parenthesization of an n elements expression has exactly (n-1) pairs of parenthesis

Longest Common Subsequence (LCS)

In LCS problem we are giving 2 sequences X & Y where,

$X = \langle x_1 x_2 \cdots x_m \rangle \quad Y = \langle y_1 y_2 \cdots y_n \rangle$

and we have to find a maximum length common subsequence of x & y.

i.e $Z = \{z_1, z_2 \cdots z_k\}$

Biological applications often need to compare the DNA of two or more different organisms. A strand of DNA consists of a string of molecules called bases, where possible bases are adenine, guanine, cytosine & thymine. We can express a strand of DNA as $\{A, C, G, T\}$. for example DNA of one organism is may be $S_1 = ACCGGT$, & other may be $S_2 = GTCGT$. one reason to compare

two strands of DNA is to determine how "similar" the two strands are, as some measure of how closely related the two organisms are.

"A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous".

Sequences = "a b c d e f g", "a b x d f g"

Common subsequences = "a", "b", "d", "f", "g", "ab", "df", "dfg", "abd" "abdfg"

Longest common subsequence = "abdfg"

X = a b c d e f g h i j

Y = e c d g i → egi (possible)

(skip) (skip) } intersect, not in order

X = a b d a ce

Y = a b ce

<abce>

X = a b d a ce

Y = b a b c e

skip

<abce>

X = a b c d e f g h i j

Y = e c d g i → cdgi (possible) — Longest

skip

Note: we can get multiple sequence with same length of subsequence.

LCS using Recursion:

$$LCS[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ 1+LCS[i+1,j+1) & \text{if } i,j>0 \text{ and } A_i = B_j \\ \max(LCS(i+1,j), LCS(i,j+1)) & \text{otherwise i.e } A_i \neq B_j \end{cases}$$

```
int LCS(i,j)
{
  if (A[i] == '\0' || B[j] == '\0'
      return 0;
  else if (A[i] == B[j])
      return 1+LCS(i+1, j+1);
  else
      return max(LCS(i+1, j), LCS(i, j+1));
}
```

// Exponential time taking algorithm require 2^n time

The algorithm says, if there is a match, Add 1 and move to the next alphabet for both A & B.

If there is no match, check either way more alphabet in 1st string A and more alphabet in 2nd string B.

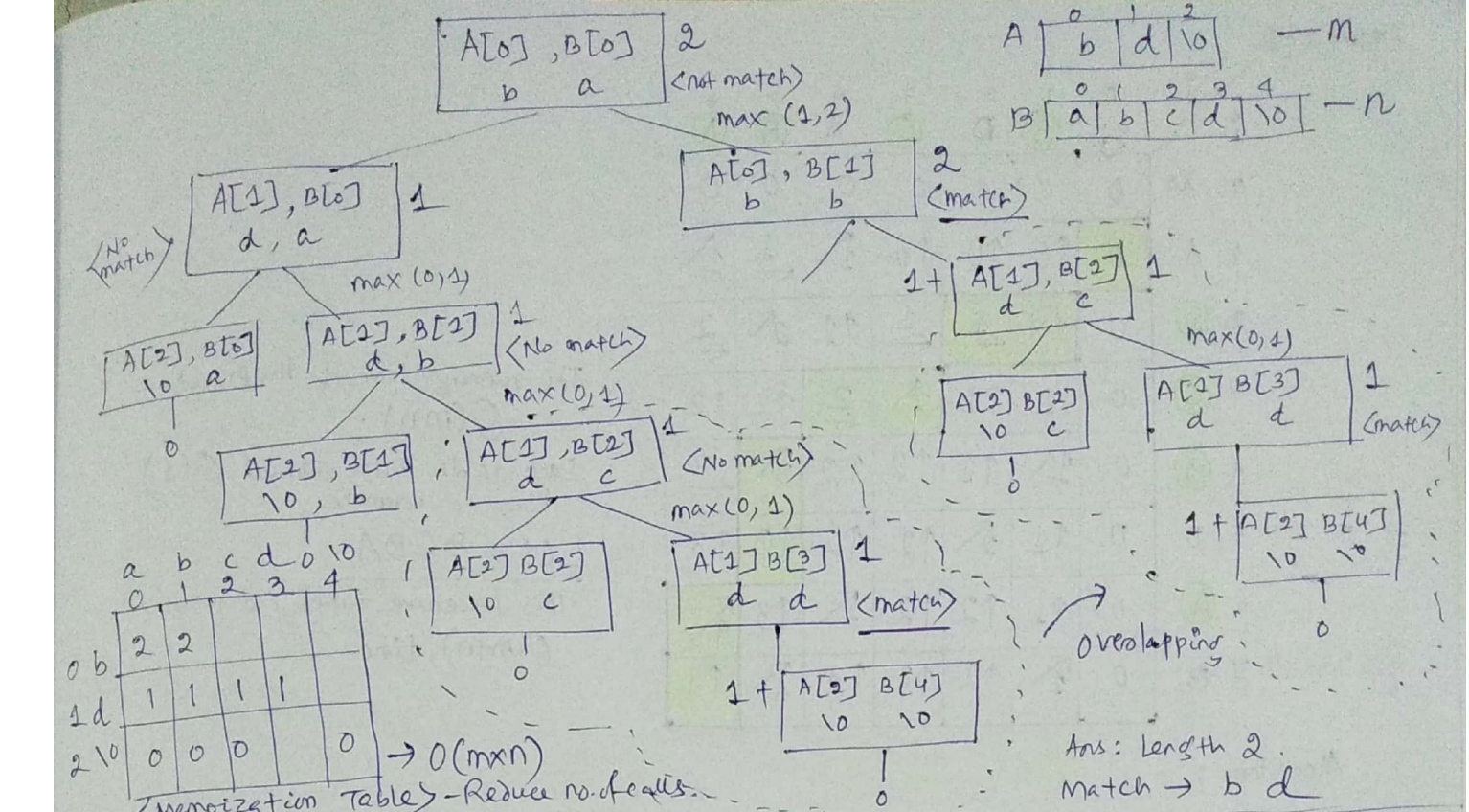
Example:

A

b	d	\0
0	1	2

B

a	b	c	d	\0
0	1	2	3	4



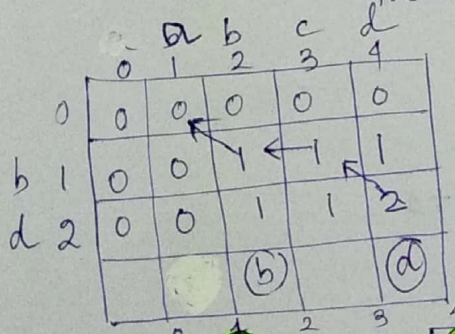
In Recursive method no. of calls is more, overlapping occurs. To improve recursion we use memoization. In case of memoization remember the answer in a table & use in future. Reduce the time complexity from 2^n to $(m \times n)$.

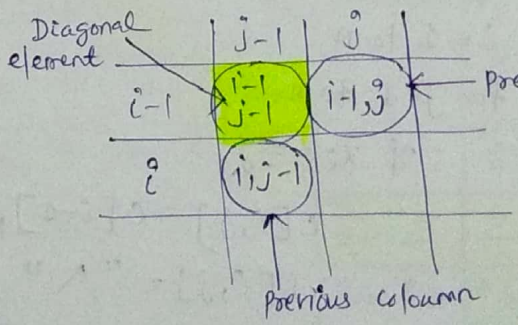
Note: Memoization (overlapping not occur)

Dynamic Programming :

$$LCS[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ LCS[i-1,j-1]+1 & i,j>0 \text{ and } A_i = B_j \\ \max(LCS[i-1,j], LCS[i,j-1]) & i,j>0 \text{ and } A_i \neq B_j \end{cases}$$

(diagonal element; previous row; previous column)





Example - 1

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

Y = A B C D H

X = A E D H F

LCS = A D H

Example-2

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	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

The running time of the procedure $\Theta(mn)$.

Each table entry takes $\Theta(1)$ time to compute

LCS = BCBA

this procedure takes i.e printing $\Theta(m+n)$ time

Algorithm :

```
LCS_LENGTH(X,Y)
1.  m = X.length
2.  n = Y.length
3.  let b[1..m, 1...n] and c[0...m, 0...n] be new tables
4.  for i = 1 to m
5.      c[i,0] = 0
6.  for j = 0 to n
7.      c[0,j] = 0
8.  for i = 1 to m
9.      for j = 1 to n
10.         if xi == yj
11.             c[i,j] = c[i-1, j-1] + 1
12.             b[i,j] = "↖"
13.         elseif c[i-1,j] ≥ c[i,j-1]
14.             c[i,j] = c[i-1,j]
15.             b[i,j] = "↑"
16.         else c[i,j] = c[i,j-1]
17.             b[i,j] = "←"
18.     return c and b
```

This above algorithm requires $\Theta(mn)$ time complexity

```
PRINT_LCS(b, X, i, j)
1. if i==0 or j==0
2.     return
3. if b[i,j] == "↖"
4.         PRINT_LCS(b, X, i-1, j-1)
5.         print Xi
6.   else if b[i,j] == "↑"
7.         PRINT_LCS(b, X, i-1, j)
8.   else  PRINT_LCS(b, X, i, j-1)
```

The above procedure takes time $O(m+n)$, since it decremented at least one of i and j in each recursive call.

Example-3

Determine the LCS of X={PRESIDENT} and Y={PROVIDENCE}

			P	R	O	V	I	D	E	N	C	E
	i \ j	0	1	2	3	4	5	6	7	8	9	10
Xi	0	0	0	0	0	0	0	0	0	0	0	0
P	1	0	1↖	1←	1←	1←	1←	1←	1←	1←	1←	1←
R	2	0	↑1	2↖	2←	2←	2←	2←	2←	2←	2←	2←
E	3	0	↑1	↑2	↑2	↑2	↑2	↑2	3↖	3←	3←	3↖
S	4	0	↑1	↑2	↑2	↑2	↑2	↑2	↑3	↑3	↑3	↑3
I	5	0	↑1	↑2	↑2	↑2	3↖	3←	↑3	↑3	↑3	↑3
D	6	0	↑1	↑2	↑2	↑2	↑3	4↖	4←	4←	4←	4←
E	7	0	↑1	↑2	↑2	↑2	↑3	↑4	5↖	5←	5←	5↖
N	8	0	↑1	↑2	↑2	↑2	↑3	↑4	↑5	6↖	6←	6←
T	9	0	↑1	↑2	↑2	↑2	↑3	↑4	↑5	↑6	↑6	↑6

Print_LCS(b, X, 9, 10) '↑'
Print_LCS(b, X, 8, 10) '←'
Print_LCS(b, X, 8, 9) '←'
Print_LCS(b, X, 8, 8) '↖'
Print_LCS(b, X, 7, 7) '↖'
Print_LCS(b, X, 6, 6) '↖'
Print_LCS(b, X, 5, 5) '↖'
Print_LCS(b, X, 4, 4) '↑'

Print_LCS(b, X, 3, 4) '↑'
Print_LCS(b, X, 2, 4) '←'
Print_LCS(b, X, 2, 3) '←'
Print_LCS(b, X, 2, 2) '↖'
Print_LCS(b, X, 1, 1) '↖'
Print_LCS(b, X, 0, 0) stop

Z = {PRIDEN}

LCS_LENGTH(9, 10) = 6

Assembly Line scheduling

A car factory has 2 assembly lines, each with n stations. A station is denoted by $S_{i,j}$ where i is either 1 or 2 and indicates the assembly line the station is on, and j indicates the number of station.

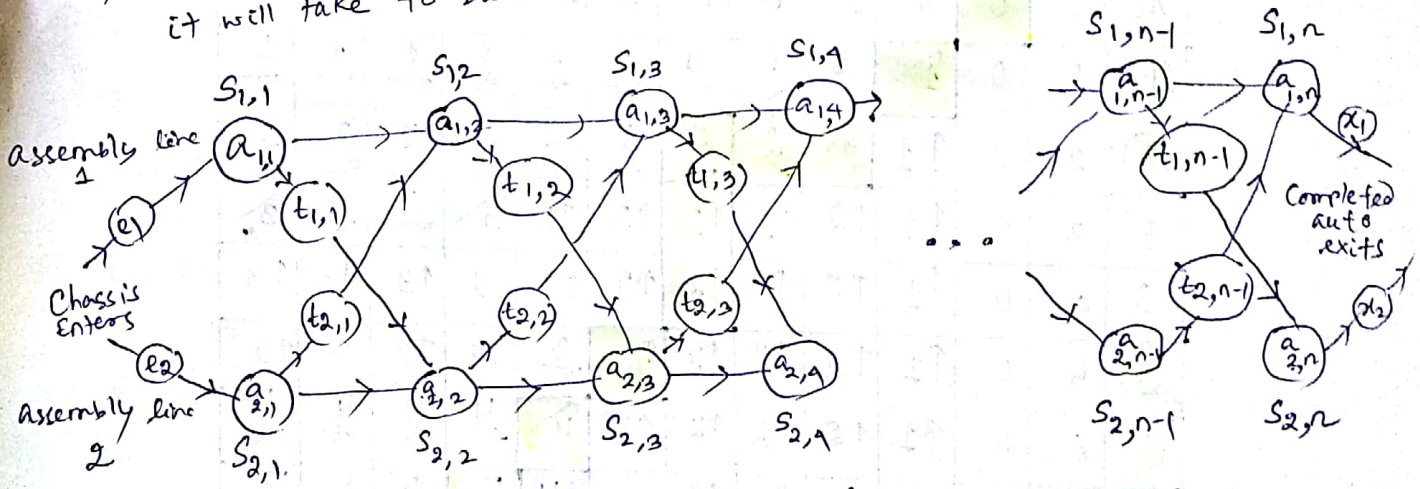
The time taken per station is denoted by a_{ij}. Each station is dedicated to some sort of work like engine fitting, body fitting, painting and so on.

A car chassis must pass through each of the n stations in order before exiting the factory. The parallel stations of the two assembly lines perform the same task. After it passes through station S_{ij}, it will continue to station $S_{i,j+1}$ unless it decide to transfer to the other line.

→ continuing on the same line incurs no extra cost, but transferring from line i at station j-1 to station j on the other line takes time t_{ij}.

→ Each assembly like take an entry time e_i and exit time x_i which may be different for the two lines.

→ So we have to design an algorithm for computing the minimum time it will take to build a car chassis.



Problem: What stations should be chosen from line 1 and which from line 2 in order to minimize the total time through factory for one car?

✓ If we apply Brute force (Enumerate all possibilities for selecting stations) technique then at each station require 2, 4, 8, 16 .... 2^n no. of comparisions. So These are 2^n possible ways to choose stations. Infeasible when n is large.

Generalization: an optimal solution to the problem "find the fastest way through $S_{1,j}$" contains within it an optimal solution to subproblems: "find the fastest way through $S_{1,j-1}$ or $S_{2,j-1}$". This is refered to as the optimal substructure property.

Defination: [Recursive solution]

f^* – the fastest time to get through the entire factory

$f_i[j]$ – the fastest time to get from the starting point through $S_{i,j}$

$f^* = \min(f_1[n] + x_1, f_2[n] + x_2)$

Base case: $j=1$, $i=1,2$

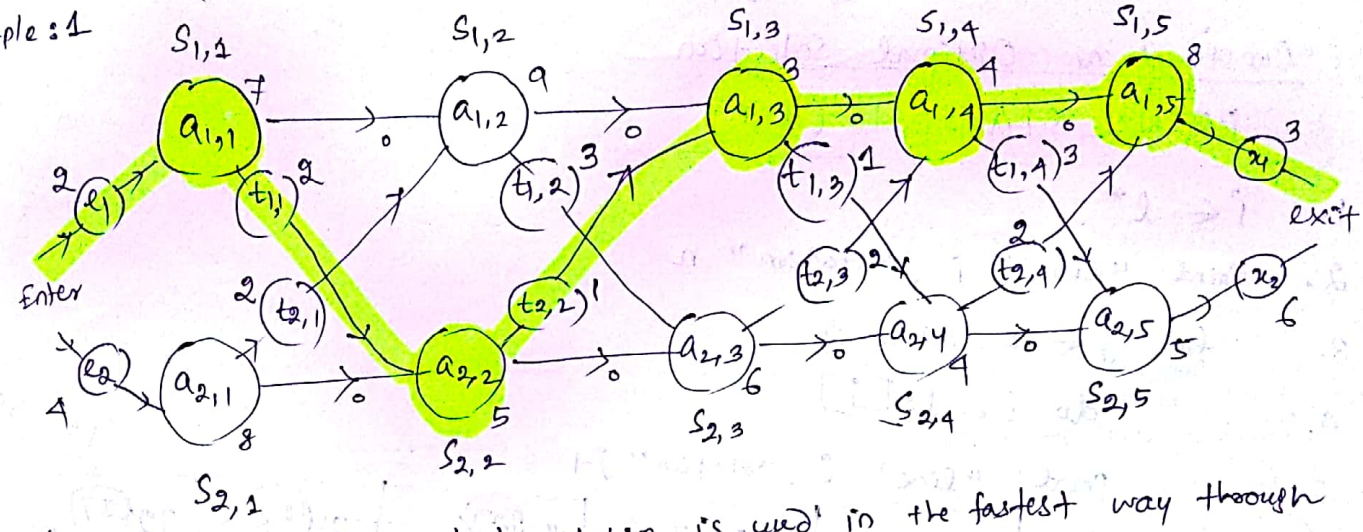
$f_1[1] = e_1 + a_{1,1}$ $\quad f_2[1] = e_2 + a_{2,1}$

General case: $j = 2, 3 \ldots n$, $i = 1, 2$

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{if } j=1 \\ \min\left(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j}\right) & \text{if } j \geq 2 \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{if } j=1 \\ \min\left(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j}\right) & \text{if } j \geq 2 \end{cases}$$

Example: 1



l^* is the line no. whose last station is used in the fastest way through the entire factory.

$l_i[j]$ helps us to track the fastest way in increasing order of j.

	1	2	3	4	5
$f_1[j]$	9	(18) 18,23	(20) 21,20	(24) 21,28	(32) 32,35
$f_2[j]$	12	(16) 17,16	(22) 22,27	(25) 26,25	(30) 30,32

+3 (exit cost) = 35

$f^* = 35$ [1]

+6 (exit cost) = 36

min

	2	3	4	5
$l_1[j]$	1	2	1	1
$l_2[j]$	1	2	1	2

Algorithm:

FASTEST_WAY (a, t, e, x, n) — Complexity $O(N)$

1. $f_1[1] \leftarrow e_1 + a_{1,1}$ } compute initial values of f_1 and f_2
2. $f_2[1] \leftarrow e_2 + a_{2,1}$
3. for $j \leftarrow 2$ to n
4. do if $f_1[j-1] + a_{1,j} \leq f_2[j-1] + t_{2,j-1} + a_{1,j}$
5. then $f_1[j] \leftarrow f_1[j-1] + a_{1,j}$
6. $l_1[j] \leftarrow 1$
7. else $f_1[j] \leftarrow f_2[j-1] + t_{2,j-1} + a_{1,j}$
8. $l_1[j] \leftarrow 2$

} Compute the values of $f_1[j]$ and $l_1[j]$

```
9.   if f2[j-1] + a2,j ≤ f1[j-1] + t1,j-1 + a2,j
10.      then f2[j] ← f2[j-1] + a2,j
11.           l2[j] ← 2
12.      else f2[j] ← f1[j-1] + t1,j-1 + a2,j
13.           l2[j] ← 1
14.  if f1[n] + x1 ≤ f2[n] + x2
15.      then f* = f1[n] + x1
16.           l* = 1
17.      else f* = f2[n] + x2
18.           l* = 2
```

Compute the values of $f_2[j]$ and $l_2[j]$

Compute the values of the fastest time through the entire factory.

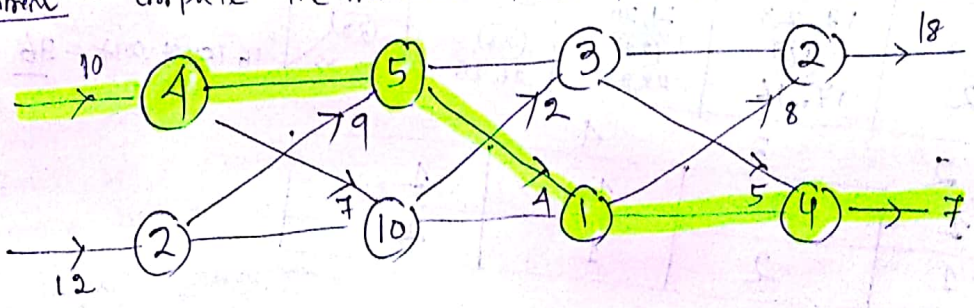
Construct an Optimal Solution

```
PRINT_STATIONS (l, n)
1. i ← l*
2. Print "line" i, "station" n
3. for j ← n down to 2
4.     do i ← li[j]
5.     Print "line" i, "station" j-1
```

$f_1[j]/l_1[j]$	9	18 [1]	20 [2]	24 [1]	32 [1]
$f_2[j]/l_2[j]$	12	16 [1]	22 [2]	25 [1]	30 [2]
	1	2	3	4	5

+3 = 35, +6 = 36, $l^* = 1$

Assignment - Compute the minimum time it will take to build a car chassis.



Dry Run.

initially $l^* = 1$, $i = 1$, Print Line 1 station 5

then $j = 5$ down to 2, $j = 5$, $i \to l_1[5]$ (=1), Print line 1 station 4

then $j = 4$, $i \to l_1[4]$ (=1), line 1 station 3

$j = 3$, $i \to l_1[3]$ (=2), line 2 station 2

$j = 2$, $i \to l_2[2]$ (=1), line 1 station 1

Note: we can compute the fastest way through the factory and the time it takes $\Theta(n)$ times.

Greedy Algorithms

It always makes the choice that seems to be the best at that moment. This means that it makes a locally-optimal choice in the hope that this choice will lead to a globally-optimal solution.

- It has only one shot to compute the optimal solution, so that it never goes back and reverses the decision.
- The greedy method is applied to solve the optimization problems that involve searching through a set of configuration to find one that minimize or maximize an objective function defined on these configurations.

Elements of Greedy strategy -

1) Determination of the optimal substructure of the problem.
2) Development of a recursive solution.
3) Prove that any stage of the recursion, one of the optimal choice is the greedy choice.
4) Show that all but one of the sub-problems included by having made the greedy choice are empty.
5) Develop a recursive algorithm that implements the greedy strategy.
6) Convert the recursive algorithm to an iterative algorithm.

Divide & Conquer	Greedy	Dynamic Programming
1. Single problem can be divided into 'n' sub problems, these n sub problems can be further divided into 'n' further sub parts.	1. It is based on local optimum solution but it can not give guarantee about the global optimum always.	1. In this approach there were exist multiple optimal solution.
2. After dividing solve the problem when further division is not possible.	2. For applying this technique the entities should be arranged in some order.	2. out of multiple solution we choose one optimal solution dynamically.
3. Examples: → Quick sort, Mergesort Heap sort Priority Queue	3. Examples: → Activity selection problem → fractional knapsack → Huffman codes. → Job scheduling with deadline	3 Examples: → Matrix Chain multiplication → Longest common subsequence → Assembly line scheduling → Travelling salesman problem.
4. Solution starts from bottom.	4. solution starts from Top.	4. Solution starts from bottom.

```
Algorithm Greedy (a, n)
{
    for i=1 to n do
    {
        x = select(a);
        if feasible(x) then
        {
            solution = solution + x;
        }
    }
}
```

n = 5

a | a_1 | a_2 | a_3 | a_4 | a_5 |
  1   2   3   4   5

Example: Purchasing Best car

- select Board
- select Top models.
- Latest & well tested models.
- features.

Company looking for a suitable candidate

- Written - GD - Technical - PI

Known method of selection called greedy.

Activity Selection Problem

It is the problem of scheduling several activities that require exclusive use of a common resource, with a goal of selecting a maximum-size set of mutually compatible activities. Here No conflict occur.

Set of activity = $\{a_1, a_2, a_3 \cdots a_n\}$

Start time = $\{s_1, s_2, s_3 \cdots s_n\}$

finish time = $\{f_1, f_2, f_3 \cdots f_n\}$

Steps: 1) Sort the activities according to the finish time i.e $f_1 < f_2 < f_3 \cdots f_n$

2) plot the graph according to the sorted sequence.

3) Apply the algorithm and find the result.

Greedy_Activity_Selector (A, S, f)

It is assumed that the i/p activities are ordered by monotonically increasing finish time. A is an array containing all activities.

1. $n \leftarrow$ length [S] or $n = S.length$
2. $SA \leftarrow \{a_1\}$
3. $j \leftarrow 1$
4. for $i \leftarrow 2$ to n
5. if $s_i \geq f_j$
6. then $SA \leftarrow SA \cup \{a_i\}$
7. $j \leftarrow i$
8. return A

Analysis

→ If the activities are already sorted. So there are one O(1) operations required per activity. Thus the total running time required for greedy-activity selector algorithm is O(n).

→ If the activities are not sorted, then we have to sort it first by using O(nlogn) time.

→ Total running time = O(nlogn) + O(n)
≅ O(nlogn)

Note: Time complexity for Greedy_activity selector algorithm (un-sorted) → O(nlogn)

Let us consider the following set S.

A $a_i \rightarrow$	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

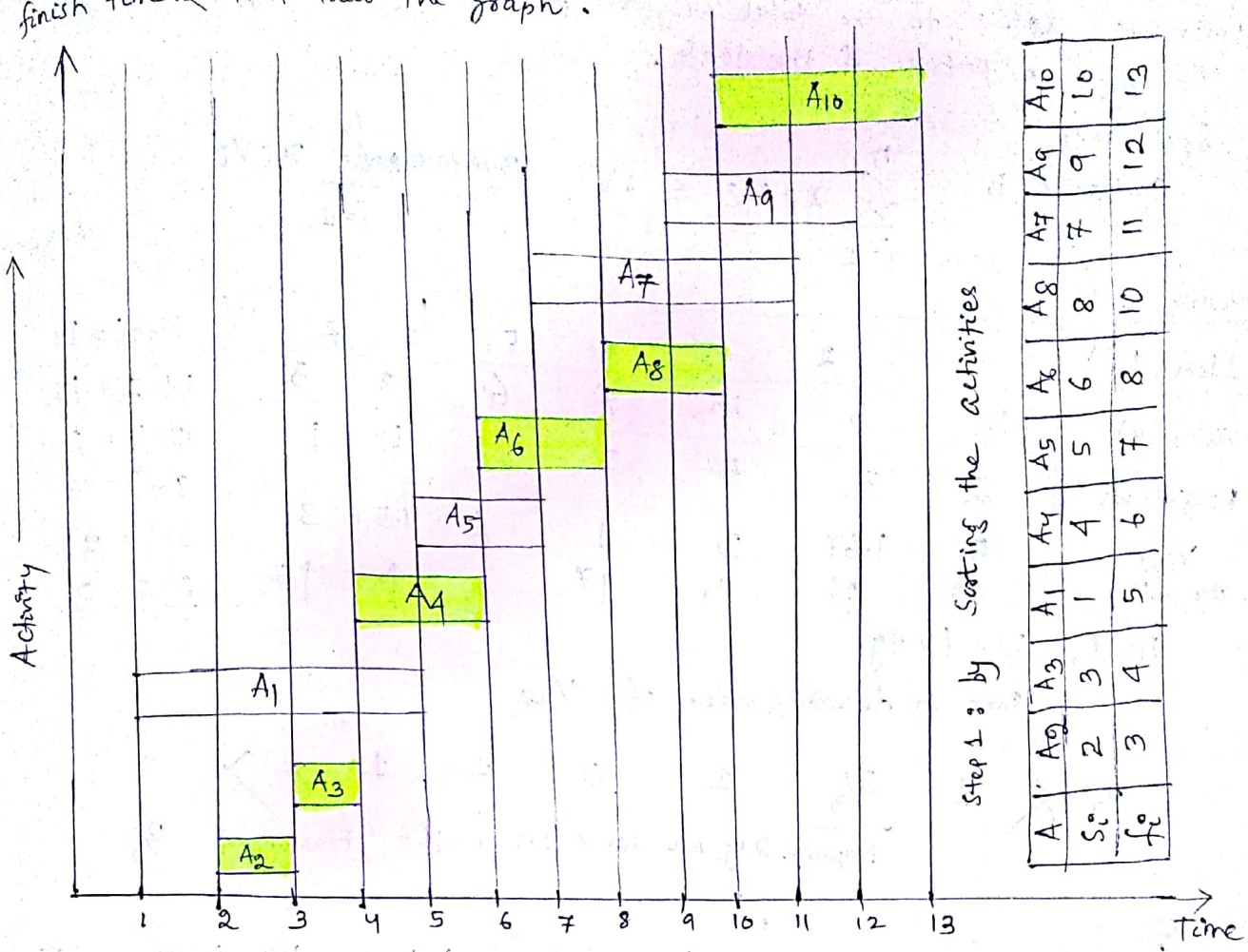
we can have several subset of mutually compatible activities like $\{a_3, a_7, a_{11}\}$, $\{a_1, a_4, a_8, a_{11}\}$, $\{a_2, a_4, a_9, a_{11}\}$ and so on. But we consider only those subsets which have maximum no. of activities and in this case it is 4.

Example: Given 10 activities along with their start and finish time as follows:

A	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}
S_i	1	2	3	4	5	6	7	8	9	10
f_i	5	3	4	6	7	8	11	10	12	13

find the optimal set of activities for the above activity-selection problem using greedy strategy.

Solution: At first we have to sort the activities in increasing order of their finish time & then draw the graph.



Step 1: by Sorting the activities

A	A2	A3	A1	A4	A5	A6	A8	A7	A9	A10
Si	2	3	1	4	5	6	8	7	9	10
fi	3	4	5	6	7	8	10	11	12	13

Final Activity Schedule: $SA = \{A_2, A_3, A_4, A_6, A_8, A_{10}\}$

Fractional - Knapsack Problem:

In Fractional Knapsack problem we can even put the fraction of any item in the Knapsack of taking the complete item is not possible i.e here items are divisible. ∴ it is solved using greedy approach.

0-1 Knapsack problem, division not possible & solved by dynamic programming.

Problem: given a Knapsack (Shoulder Bag) with a limited weight capacity. and few items each having some weight & value, which items should we place in the Knapsack so that the value or profit that we obtain by putting the items in Knapsack is maximum & the weight limit of the Knapsack also does not exceed.

Steps for solving fractional - Knapsack problem using greedy approach

1) For each item, compute its value/weight ratio
2) Arrange all the items in the decreasing order of their value/weight ratios.
3) Start putting the items in the Knapsack begining from the item with the highest ratio. put as many items as we can in the Knapsack.

n items in a store, for $i=1, 2, \dots n$, item 'i' has weight $w_i > 0$ & value $v_i > 0$, W = maximum capacity of the bag or Knapsack

only a fraction x_i of object on item i, where $0 \le x_i \le 1$. Item i contribute $x_i w_i$ to the total weight in the knapsack, and $x_i v_i$ to the value or profit of the load.

Symbolically -

Constraint is $\sum_{i=1}^{n} x_i w_i \le W$ objective max $\sum_{i=1}^{n} x_i v_i$

Example-1 :

Items (i)	1	2	3	4	5	6	7	15 - 1 = 14
Value (v)	10	5	15	7	6	18	3	14 - 2 = 12
Weight (w)	2	3	5	7	1	4	1	12 - 4 = 8
v/w	5	1.67	3	1	6	4.5	3	8 - 5 = 3
Res unit	↑2	↑6	↑1	↑7	↑1	↑3	↑5	3 - 1 = 2
								2 - 2 = 0

$n = 7$, $W = 15$ Kg.

Sort in decreasing order of v/w

$x < 1 \quad 2/3 \quad 1 \quad 0 \quad 1 \quad 1 \quad 1 >$

Require 2kg but having 3kg weight, Fraction is $2/3$

$$\sum_{i=1}^{n} x_i w_i = 1\times2 + \frac{2}{3}\times3 + 1\times5 + 0\times7 + 1\times1 + 1\times4 + 1\times1 = 15\text{kg}$$

$$\text{Profit max} \sum_{i=1}^{n} x_i v_i = 1\times10 + \frac{2}{3}\times5 + 1\times15 + 0\times7 + 1\times6 + 1\times18 + 1\times3$$

$$= 10 + 3.33 + 15 + 6 + 18 + 3 = \underline{55.33} \quad \text{(Ans)}$$

Fractional-Knapsack (v, w, W)

```
Repeat for i ← 1 to sized(v)
    do p[i] ← v[i]/w[i]
[end of for loop]
Sort in decreasing order (p)
    Set i ← 1
    Repeat while (W > 0)
        do s ← min (W, w[i])
            Solution [i] ← s
            W ← W − s
            i ← i+1
    [end of while loop].
Return (solution)
```

Maximum-value(solution, P)

```
value ← 0
Repeat for i ← 1 to size of (solution)
    do
    value ← value + solution[i] * p[i]
    [end of for loop]
return (value)
```

Here inputs = i_1, i_2, i_3 ..

weights = w_1, w_2, w_3 ..

value = v_1, v_2, v_3 ...

W = maximum capacity

Analysis : (Time complexity)

Sort the value $P=V_i/W_i$ in decreasing order will take $O(n\log n)$ [using Heap Sort]

while loop in algorithm require $O(n)$ time.

Total Time complexity = $O(n) + O(n\log n) = O(n\log n)$

Example-2

Items (I_i)	1	2	3	4	5
value (V_i)	30	20	100	90	160
weights (w_i)	5	10	20	30	40
$P=\frac{V_i}{w_i}$	6.0	2.0	5.0	3.0	4.0
	↑0	↑5	↑2	↑4	↑3

(Sort)

Dry Run according to algorithm :

initial value of $i=1$,

Capacity of knapsack $W = 60 > 0$

$S = \min(W, w[i]) \Rightarrow \min(60,5) \Rightarrow 5$

Solution[1] = 5

$W = W - 5 \Rightarrow 60 - 5 = 55$

$i = i+1 = 1+1 = 2$

Now capacity of knapsack $W = 55 > 0$

$S = \min(\because 55, 20) \Rightarrow 20$, solution[2] = 20

$W = 55 - 20 = 35$

$i = i+1 = 2+1 = 3$

Now capacity of knapsack $W = 35 > 0$

$S = \min(35, 40) \Rightarrow 35$, solution[3] = 35

$W = W - 35 = 35 - 35 \Rightarrow 0$

$i = i+1 = 3+1 = 4$

Now capacity of $W = 0$, which terminate the while loop & return solution

Now find the maximum value

initial value = 0

$i = 1$ to size of(solution)

$i = 1$ to 3

$i=1$, value = value + solution[1] * P[1] $\Rightarrow$ $0 + 5*6 \Rightarrow 30$

$i=2$, value = value + solution[2] * P[2] $\Rightarrow$ $30 + 20*5 \Rightarrow 130$

$i=3$, value = value + solution[3] * P[3] $\Rightarrow$ $130 + 35*4 \Rightarrow 130+140 = 270$

∴ Thus optimal solution of fractional-knapsack problem is 270.

Assignment :

Item	1	2	3	4	5
value	30	40	45	77	90
weight	5	10	15	22	25

n = 5

W = 60kg

max profit = 230 (Ans)

Huffman Codes : (Dr. David A. Huffman)

Huffman codes are widely used and very effective technique for compressing data. Here we consider the data to be a sequence of characters. Huffman's greedy algorithm uses a table of the frequencies of occurance of the characters to build up an optimal way of representing each character as a binary string. Used for Lossless (without loss of information) compression.

Message : B C C A B B D D A E C C B B A E D D C C — length = 20

Cost of message measured by bits.

Characters are represented in ASCII code. i.e 8 bits

A = 65 = 01000001 8 bits

Size of msg = 20 (length) × 8 bits = 160 bits

Simple message sent without encoding require 160 bits.

(1) Fixed length/size code : each character is represented by a unique binary string i.e equal no. of bits.

If in a file Characters consists of m different type character then maximum no. of bits used to represent each character is n, where $2^n \geq m$

Here 6 different characters are present, at least 3 bits required i.e $2^3 \geq 6$

Character	Count/Frequency	Code
A	3	000
B	5	001
C	6	010
D	4	011
E	2	100

Table size = 5 × 8 bits + 5 × 3 bits

= 40 bits + 15 bits

= 55 bits

Size = 20 × 3 bits = 60 bits

Encode msg + Table set transfered.

60 bits + 55 bits

⇒ 115 bits

Size is reduced.

(2) Variable length/size code : considerably better than a fixed-length code, by giving frequent characters short codewords and infrequent character as long codewords.

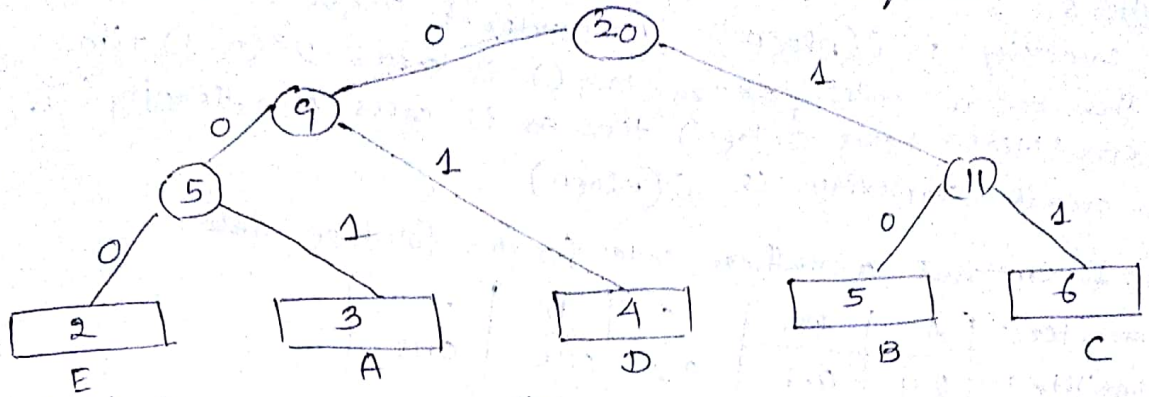
Prefix codes is the code in which there is no codeword that is a prefix (free) of other codeword. The representation of Decoding process is binary tree whose leaves are characters. we interpret the binary codeword for a character as path from the root to that character, where

→ "0" means "go to the left child"

→ "1" means "go to the right child"

It is also called as Optimal merge pattern (Arrange increase order of the count)

take two smallest one to make one node and sum the frequencies.



Char	Count	Code	Size
A	3	001	3×3 = 9 bits
B	5	10	5×2 = 10 bits
C	6	11	6×2 = 12 bits
D	4	01	4×2 = 8 bits
E	2	000	2×3 = 6 bits
			45 bits (message size)

Table Size or Tree = 5×8 bits + 12 bits ⇒ 40 bits + 12 = 52 bits

Total = message + Table/Tree = 45 bits + 52 bits = 97 bits (More reduced)

formula ⇒ distance * Frequency i.e $\sum d_i f_i$ (optimal message pattern).

Distance of E = 3, Freq = 2 similarly for all other character, we got

$3\times2 + 3\times3 + 2\times4 + 2\times5 + 2\times6 = 45$ bits

B C C D A C C B . . .
10 11 11 01 001 11 11 10 . . .
Decode : B C C D . . .

Greedy Algorithm for Huffman code :

Huffman invented a greedy algorithm that construct an optimal prefix-code called a huffman code. It begin with a set of |C| leaves and perfoms a sequence of |C|-1 merging operations to create a final node. To create the huffman code to a file we have use "min-priority Queue" Data structure.

We assume that C is a set of n characters and the each character $c \in C$ is an object with an c.freq attribute.

```
Huffman (C)
n → |C|
Q → C  // min-priority Queue
repeat for i → 1 to n-1
do  z → Allocate_node( )  // allocate a new node z
    left[z] ← x ← Extract_MIN(Q)
    right[z] ← y ← Extract_MIN(Q)
    f[z] = f[x] + f[y]
    Insert (Q, z)
End of loop
return Extract-MIN(Q)  // return the root of the tree
```

Analysis

Time complexity is $O(n \log n)$, n = number of unique characters.
If there are 'n' nodes, ExtractMin() is called $2*(n-1)$ times.
ExtractMin() takes $O(\log n)$ time as it calls Min-Heapify().
So overall complexity is $O(n \log n)$.

Example: 1 Construct a Huffman code for the following data.

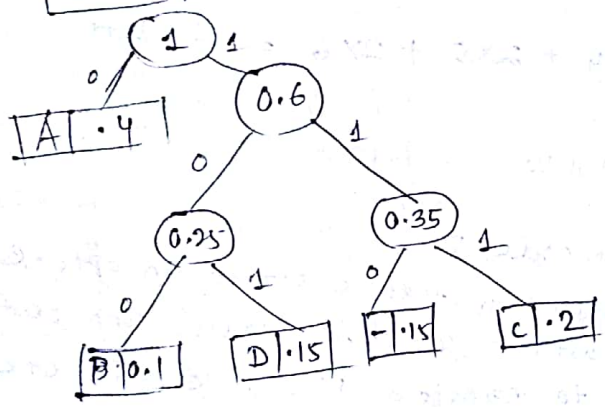
Character :	A	B	C	D	—
Probability :	0.4	0.1	0.2	0.15	0.15

Solution: initially the character & probability are inserted in the priority queue.

1. B | 0.1 ; D | 0.15 ; — | 0.15 ; C | 0.2 ; A | 0.4

2. — | .15 ; C | 0.2 ; 0.25 (B | 0.1 ; D | .15) ; A | 0.4

3. 0.25 (B | .1 ; D | .15) ; 0.35 (— | .15 ; C | 0.2) ; A | 0.4

4. 

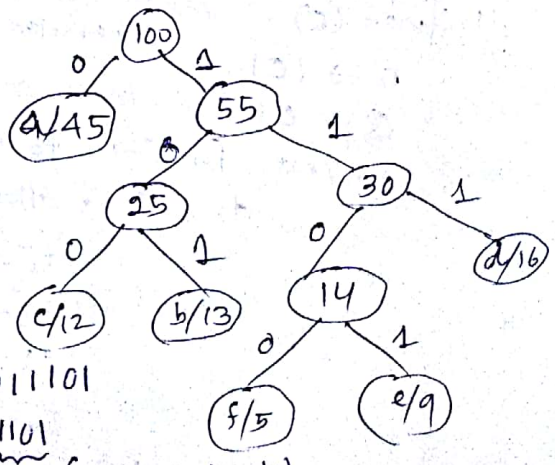
Assign code .. Left = 0, Right = 1

The huffman code for A = 0, B = 100, C = 111, D = 101, — = 110

Example: 2

Character	frequency	Variable length code
a	45	0
b	13	101
c	12	100
d	16	111
e	9	1101
f	5	1100

Ans:



String is 001011101
as 0.0.101.1101
decodes a a b e (passes uniquely)

Disjoint Set Data Structure

In computing, a **disjoint-set data structure**, also called a **union–find data structure** or **merge– find set**, is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non overlapping) subsets.

It supports the following useful operations:

• *Find*: Determine which subset a particular element is in. *Find* typically returns an item from this set that serves as its "representative"; by comparing the result of two *Find* operations, one can determine whether two elements are in the same subset.

• *Union*: Join two subsets into a single subset.

• *Make Set*, which makes a set containing only a given element (a singleton).

Applications:

• partitioning of a set

• implementing Kruskal's algorithm to find the minimum spanning tree of a graph.

• Determine the connected components of an undirected graph.

Disjoint Set Operations, Linked list Representation

• A disjoint-set is a collection S={S1, S2,…, S_k} of distinct dynamic sets.

• Each set is identified by a member of the set, called *representative*.

Disjoint set operations

– MAKE-SET(*x*): create a new set with only *x*. Assume *x* is not already in some other set.

– UNION(*x*,*y*): combine the two sets containing *x* and *y* into one new set. A new representative is selected.

– FIND-SET(*x*): return the representative of the set containing *x*.

Application: Determine the connected components of an undirected graph.

Linked list Representation

• Each set as a linked-list, with head and tail, and each node contains value, next node pointer and back-to-representative pointer.

21.2 Linked-list representation of disjoint sets

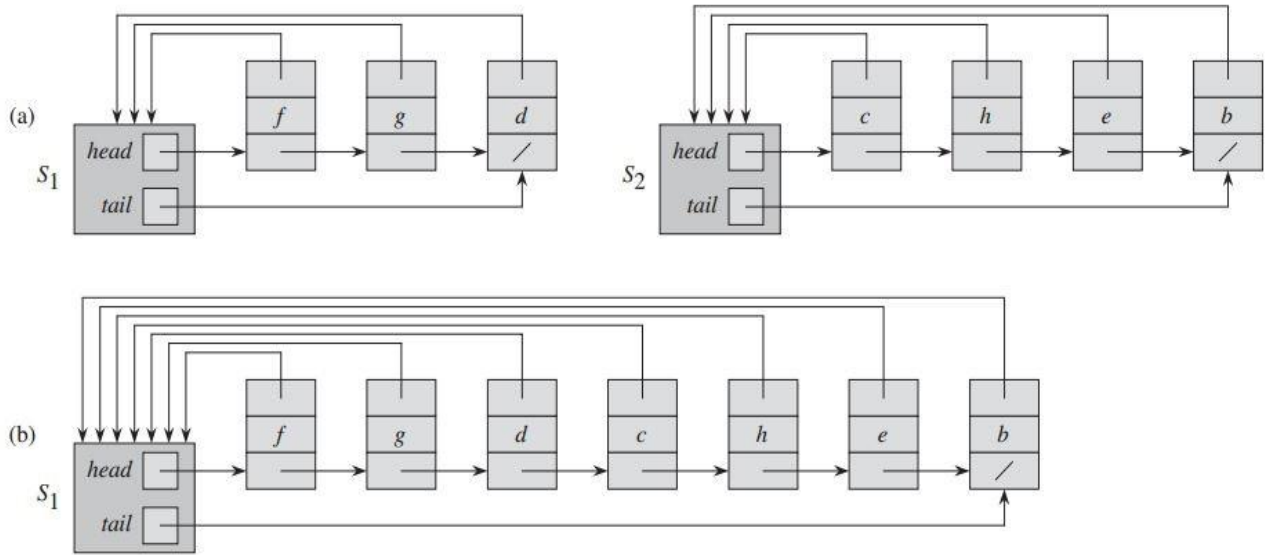
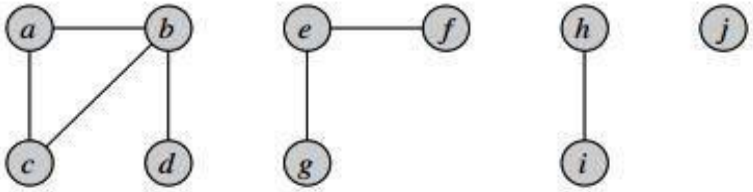


Figure 21.2 (a) Linked-list representations of two sets. Set S1 contains members d, f , and g, with representative f , and set S2 contains members b, c, e, and h, with representative c. Each object in the list contains a set member, a pointer to the next object in the list, and a pointer back to the set object. Each set object has pointers head and tail to the first and last objects, respectively. (b) The result of UNION(g, e), which appends the linked list containing e to the linked list containing g. The representative of the resulting set is f . The set object for e's list, S2, is destroyed.

The total time spent in all UNION operations is thus O(n lg n). Each MAKESET and FIND-SET operation takes O(1) time, and there are O(m) of them. The total time for the entire sequence is thus O(m + n lg n).



(a)

Edge processed	Collection of disjoint sets									
initial sets	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b,d)	{a}	{b,d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e,g)	{a}	{b,d}	{c}		{e,g}	{f}		{h}	{i}	{j}
(a,c)	{a,c}	{b,d}			{e,g}	{f}		{h}	{i}	{j}
(h,i)	{a,c}	{b,d}			{e,g}	{f}		{h,i}		{j}
(a,b)	{a,b,c,d}				{e,g}	{f}		{h,i}		{j}
(e,f)	{a,b,c,d}				{e,f,g}			{h,i}		{j}
(b,c)	{a,b,c,d}				{e,f,g}			{h,i}		{j}

(b)

Figure 21.1 **(a)** A graph with four connected components: $\{a, b, c, d\}$, $\{e, f, g\}$, $\{h, i\}$, and $\{j\}$. **(b)** The collection of disjoint sets after processing each edge.

CONNECTED-COMPONENTS(G)
1. for each vertex v∈V[G]
2. MAKE-SET(v)
3. for each edge (u,v) ∈E[G]
4. if FIND-SET(u) ≠ FIND-SET(v)
5. then UNION(u,v)

SAME-COMPONENT(*u*,*v*)
1. if FIND-SET(u)=FIND-SET(v)
2. then return TRUE
3. else return FALSE

The procedure CONNECTED-COMPONENTS initially places each vertex in its own set. Then, for each edge (u,v), it unites the sets containing u and v. By Exercise 21.1-2, after processing all the edges, two vertices are in the same connected component if and only if the corresponding objects are in the same set. Thus, CONNECTED-COMPONENTS computes sets in such a way that the procedure SAME-COMPONENT can determine whether two vertices are in the same connected component. Figure 21.1(b) illustrates how CONNECTED-COMPONENTS compute the disjoint sets.

Disjoint Forests

In a faster implementation of disjoint sets, we represent sets by rooted trees, with each node containing one member and each tree representing one set. In a disjointset forest, illustrated in Figure 21.4(a), each member points only to its parent. The root of each tree contains the representative and is its own parent. As we shall see, although the straightforward algorithms that use this representation are no faster than ones that use the linked-list representation, by introducing two heuristics—"union by rank" and "path compression"—we can achieve an asymptotically optimal disjoint-set data structure.

Union by Rank & Path Compression

• **Union by Rank:** Each node is associated with a rank, which is the upper bound on the height of the node (i.e., the height of subtree rooted at the node), then when UNION, let the root with smaller rank point to the root with larger rank.

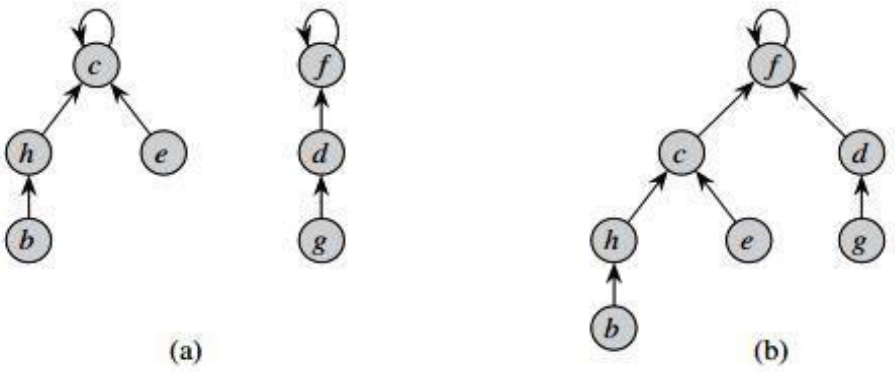
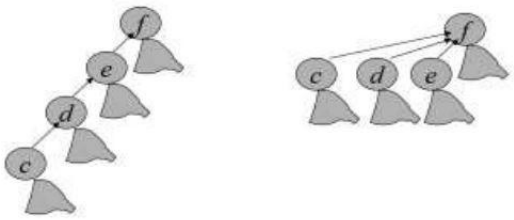


Figure 21.4 A disjoint-set forest. **(a)** Two trees representing the two sets of Figure 21.2. The tree on the left represents the set $\{b, c, e, h\}$, with c as the representative, and the tree on the right represents the set $\{d, f, g\}$, with f as the representative. **(b)** The result of UNION(e, g).

• **Path Compression:** used in FIND-SET(x) operation, make each node in the path from x to the root directly point to the root. Thus reduce the tree height.



Pseudocode for disjoint-set forests

```
MAKE-SET(x)
1  x.p = x
2  x.rank = 0

UNION(x, y)
1  LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
1  if x.rank > y.rank
2       y.p = x
3  else x.p = y
4       if x.rank == y.rank
5            y.rank = y.rank + 1
```

The FIND-SET procedure with path compression is quite simple:

```
FIND-SET(x)
1  if x ≠ x.p
2       x.p = FIND-SET(x.p)
3  return x.p
```

To implement a disjoint-set forest with the union-by-rank heuristic, we must keep track of ranks. With each node x, we maintain the integer value x.rank, which is an upper bound on the height of x. When MAKE-SET creates a singleton set, the single node in the corresponding tree has an initial rank of 0. Each FIND-SET operation leaves all ranks unchanged. The UNION operation has two cases, depending on whether the roots of the trees have equal rank. If the roots have unequal rank, we make the root with higher rank the parent of the root with lower rank, but the ranks themselves remain unchanged. If, instead, the roots have equal ranks, we arbitrarily choose one of the roots as the parent and increment its rank.

Let us put this method into pseudocode. We designate the parent of node x by x.p. The LINK procedure, a subroutine called by UNION, takes pointers to two roots as inputs. The FIND-SET procedure is a two-pass method: as it recurses, it makes one pass up the find path to find the root, and as the recursion unwinds, it makes a second pass back down the find path to update each node to point directly to the root. Each call of FIND-SET(x) returns x.p in line 3. If x is the root, then FIND-SET skips line 2 and instead returns x.p, which is x; this is the case in which the recursion bottoms out. Otherwise, line 2 executes, and the recursive call with parameter x:p returns a pointer to the root. Line 2 updates node x to point directly to the root, and line 3 returns this pointer

Graph algorithms

A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph can be defined as,

A Graph consists of a finite set of vertices(or nodes) and set of Edges which connect a pair of nodes.

Types of graphs

Undirected: An undirected graph is a graph in which all the edges are bi-directional i.e. the edges do not point in any specific direction.

Directed: A directed graph is a graph in which all the edges are uni-directional i.e. the edges point in a single direction.

Weighted: In a weighted graph, each edge is assigned a weight or cost.

Cyclic: A graph is cyclic if the graph comprises a path that starts from a vertex and ends at the same vertex. That path is called a cycle. An acyclic graph is a graph that has no cycle.

Adjacency-list representation of a graph

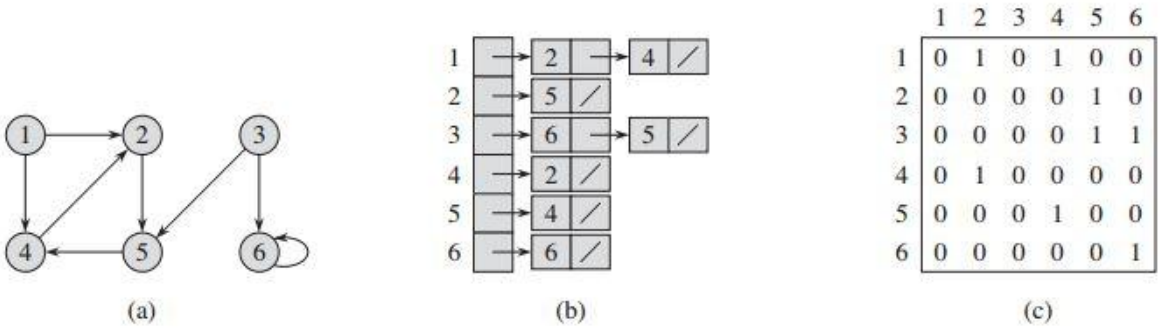


Figure 22.2 Two representations of a directed graph. **(a)** A directed graph G with 6 vertices and 8 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

Graph Algorithm - BFS and DFS

BFS	**DFS**
BFS Stands for "**Breadth First Search**". BFS starts traversal from the root node and then explore the search in the level by level manner i.e. as close as possible from the root node.	**DFS** stands for "**Depth First Search**". DFS starts the traversal from the root node and explore the search as far as possible from the root node i.e. depth wise.
Breadth First Search can be done with the help of **queue** i.e. **FIFO** implementation. This algorithm works in single stage. The visited vertices are removed from the queue and then displayed at once.	Depth First Search can be done with the help of **Stack** i.e. **LIFO** implementations. This algorithm works in two stages – in the first stage the visited vertices are pushed onto the stack and later on when there is no vertex further to visit those are popped-off.
BFS is **slower** than DFS.	DFS is more **faster** than BFS.

BFS requires **more** memory compare to DFS. **Applications of BFS** > To find Shortest path	DFS require **less** memory compare to BFS. **Applications of DFS** > useful in finding spanning trees & forest.
Example :  A, B, C, D, E, F	**Example :**  A, B, D, C, E, F

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

The procedure BFS works as follows. With the exception of the source vertex s, lines 1–4 paint every vertex white, set u.d to be infinity for each vertex u, and set the parent of every vertex to be NIL. Line 5 paints s gray, since we consider it to be discovered as the procedure begins. Line 6 initializes s.d to 0, and line 7 sets the predecessor of the source to be NIL.

Lines 8–9 initialize Q to the queue containing just the vertex s. The while loop of lines 10–18 iterates as long as there remain gray vertices, which are discovered vertices that have not yet had their adjacency lists fully examined.

Line 11 determines the gray vertex u at the head of the queue Q and removes it from Q. The for loop of lines 12–17 considers each vertex in the adjacency list of u. If is white, then it has not yet been discovered, and the procedure discovers it by executing lines 14–17. The procedure paints vertex gray, sets its distance v.d to u.d+1, records u as its parent v.π, and places it at the tail of the queue Q. Once the procedure has examined all the vertices on u's adjacency list, it blackens u in line 18. The loop invariant is maintained because whenever a vertex is painted gray (in line 14) it is also enqueued (in line 17), and whenever a vertex is dequeued (in line 11) it is also painted black (in line 18).

Analysis of BFS

The operations of enqueuing and dequeuing take O(1) time, and so the total time devoted to queue operations is O(V). Because the procedure scans the adjacency list of each vertex only when the vertex is dequeued, it scans each adjacency list at most once. The total time spent in scanning adjacency lists is O(E). The total running time of the BFS procedure is **O(V + E).**

The following procedure prints out the vertices on a shortest path from s to v, assuming that BFS has already computed a breadth-first tree:

```
PRINT-PATH(G, s, v)
1  if v == s
2       print s
3  elseif v.π == NIL
4       print "no path from" s "to" v "exists"
5  else PRINT-PATH(G, s, v.π)
6       print v
```

Example

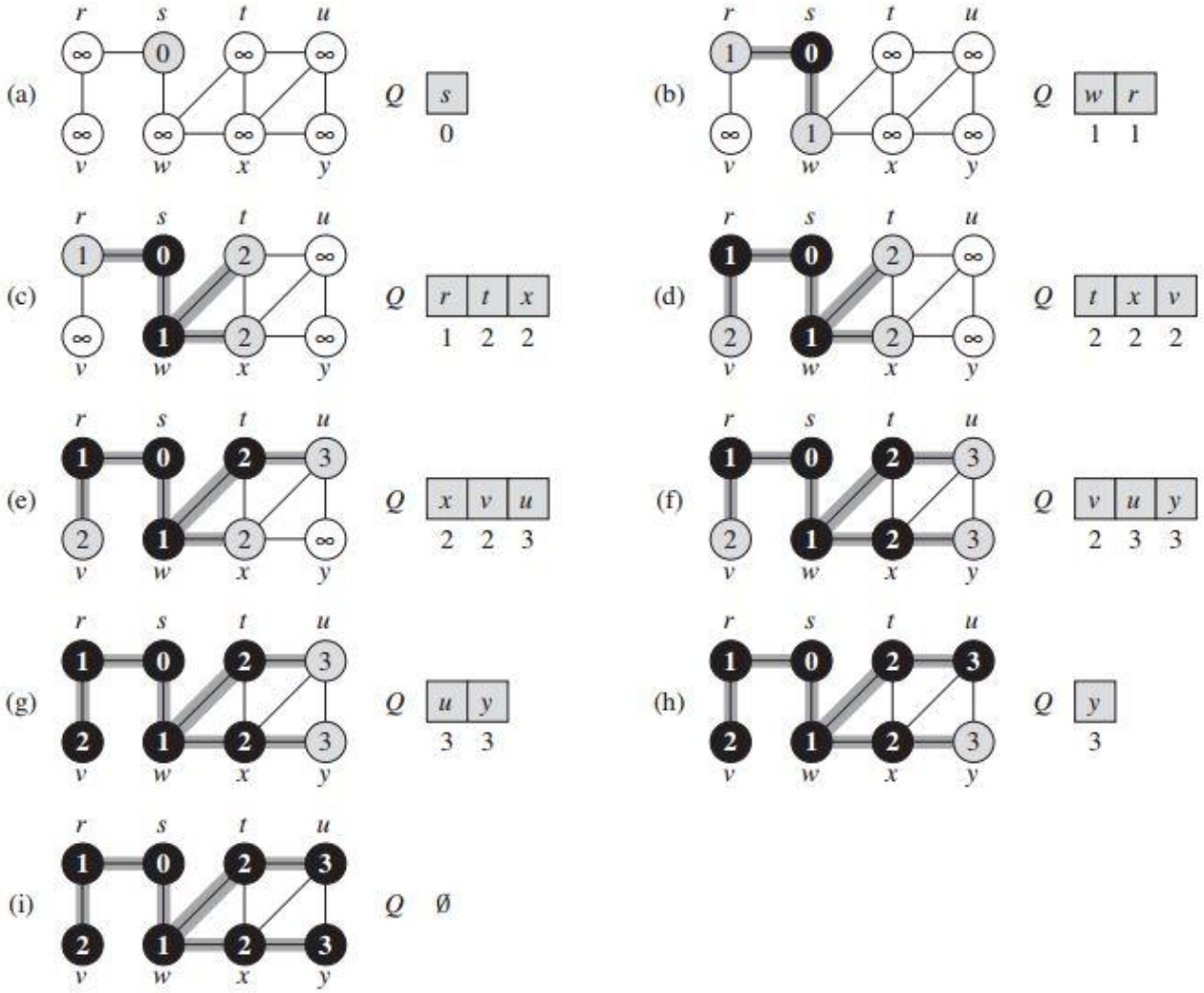


Figure: The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. The value of u.d appears within each vertex u. The queue Q is shown at the beginning of each iteration of the while loop of lines 10–18. Vertex distances appear below vertices in the queue.

Depth-first search

The strategy followed by depth-first search is, as its name implies, to search "deeper" in the graph whenever possible. Depth-first search explores edges out of the most recently discovered vertex that still has unexplored edges leaving it. Once all of v's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex. If any undiscovered vertices remain, then depth-first search selects one of them as a new source, and it repeats the search from that source. The algorithm repeats this entire process until it has discovered every vertex.

The following pseudocode is the basic depth-first-search algorithm. The input graph G may be undirected or directed. The variable *time* is a global variable that we use for timestamping.

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)

DFS-VISIT(G, u)
1  time = time + 1            // white vertex u has just been discovered
2  u.d = time
3  u.color = GRAY
4  for each v ∈ G.Adj[u]      // explore edge (u, v)
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = BLACK            // blacken u; it is finished
9  time = time + 1
10 u.f = time
```

Procedure DFS works as follows. Lines 1–3 paint all vertices white and initialize their π attributes to NIL. Line 4 resets the global time counter. Lines 5–7 check each vertex in V in turn and, when a white vertex is found, visit it using DFS-VISIT. Every time DFS-VISIT(G, u) is called in line 7, vertex u becomes the root of a new tree in the depth-first forest. When DFS returns, every vertex u has been assigned a discovery time u.d and a finishing time u.f . In each call DFS-VISIT(G, u), vertex u is initially white. Line 1 increments the global variable time, line 2 records the new value of time as the discovery time u.d, and line 3 paints u gray. Lines 4–7 examine each vertex v adjacent to u and recursively visit if it is white. As each vertex v ε Adj[u] is considered in line 4, we say that edge (u, v) is explored by the depth-first search. Finally, after every edge leaving u has been explored, lines 8–10 paint u black, increment time, and record the finishing time in u.f .

Analysis of BFS: The total running time of the BFS procedure is O(V + E).

Example:

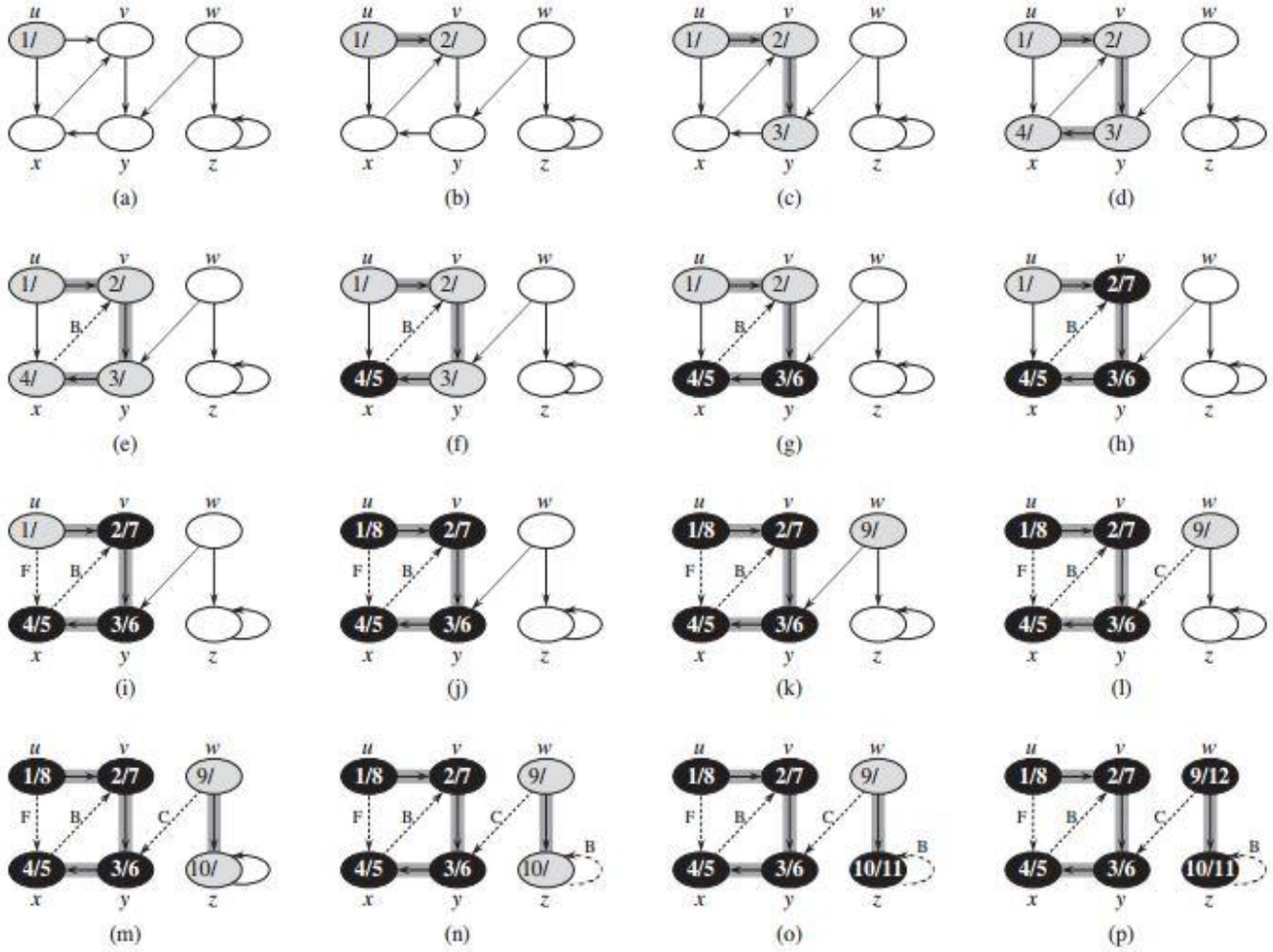


Figure: The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Timestamps within vertices indicate discovery time/finishing times.

Minimum Spanning Trees

Given an undirected and connected graph G= (V,E), a spanning tree of the graph G is a tree that spans G (that is, it includes every vertex of G) and is a subgraph of G(every edge in the tree belongs to G).

The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. There also can be many minimum spanning trees. Minimum spanning tree has direct application in the design of networks.

We can model this wiring problem with a connected, undirected graph $G = (V, E)$, where V is the set of pins, E is the set of possible interconnections between pairs of pins, and for each edge $(u, v) \in E$, we have a weight $w(u, v)$ specifying the cost (amount of wire needed) 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

$$w(T) = \sum_{(u,v)\in T} w(u, v)$$

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.

Electronic circuit designs often need to make the pins of several components electrically equivalent by wiring them together. To interconnect a set of n pins, we can use an arrangement of n-1 wires, each connecting two pins. Of all such arrangements, the one that uses the least amount of wire is usually the most desirable.

In this chapter, we shall examine two algorithms for solving the minimumspanning-tree problem: Kruskal's algorithm and Prim's algorithm.

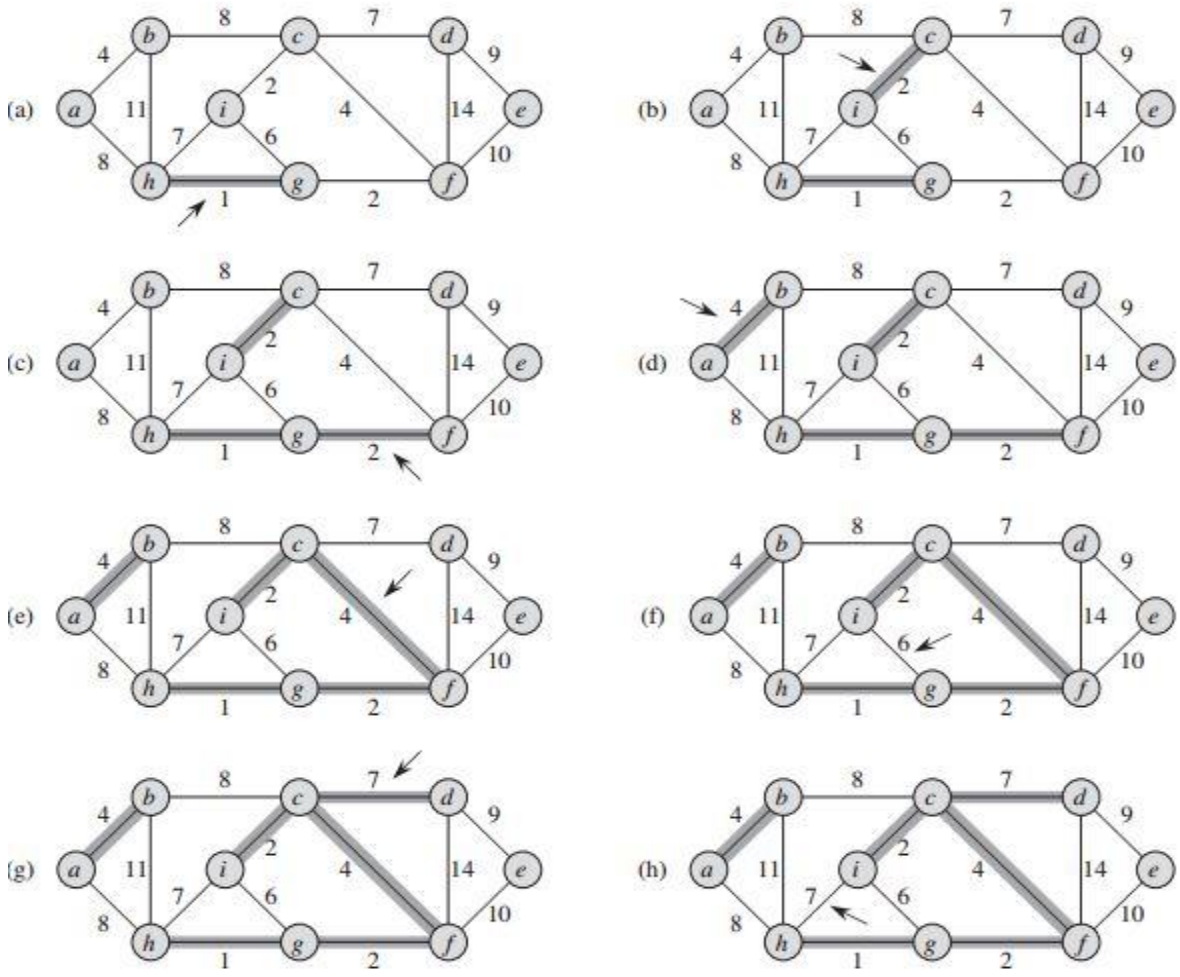
Kruskal's Algorithm

Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and adds it to the growing spanning tree.

Algorithm Steps:

- Sort the graph edges with respect to their weights.
- Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
- Only add edges which doesn't form a cycle, edges which connect only disconnected components.

Example:



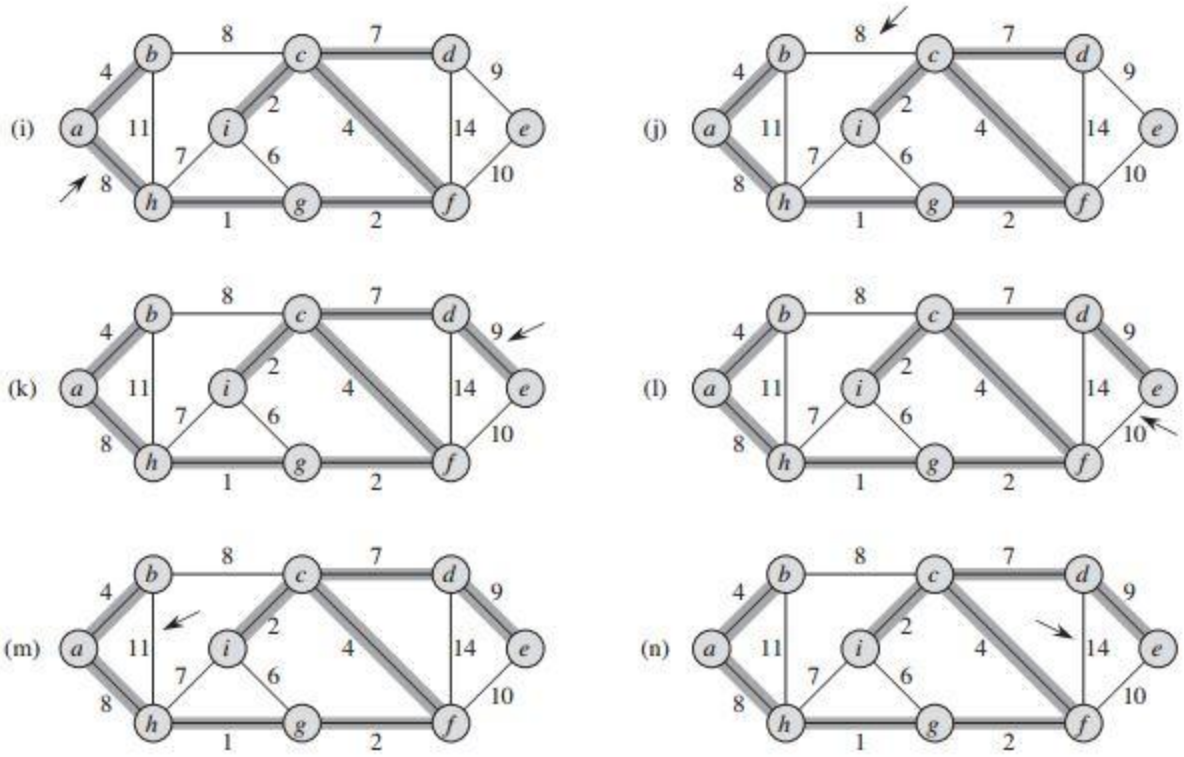


Figure: The execution of Kruskal's algorithm on the graph from Shaded edges belong to the forest A being grown. The algorithm considers each edge in sorted order by weight. An arrow points to the edge under consideration at each step of the algorithm. If the edge joins two distinct trees in the forest, it is added to the forest, thereby merging the two trees.

Kruskal's uses a disjoint-set data structure to maintain several disjoint sets of elements. Each set contains the vertices in one tree of the current forest. The operation FIND-SET(u) returns a representative element from the set that contains u. Thus, we can determine whether two vertices u and v belong to the same tree by testing whether FIND-SET(u) equals FIND-SET(v). To combine trees, Kruskal's algorithm calls the UNION procedure.

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3      MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6      if FIND-SET(u) ≠ FIND-SET(v)
7          A = A ∪ {(u, v)}
8          UNION(u, v)
9  return A
```

Lines 1–3 initialize the set A to the empty set and create |V| trees, one containing each vertex. The for loop in lines 5–8 examines edges in order of weight, from lowest to highest. The loop checks, for each edge(u, v), whether the endpoints u and v belong to the same tree. If they do, then the edge (u, v) cannot be added to the forest without creating a cycle, and the edge is discarded. Otherwise, the two vertices belong to different trees. In this case, line 7 adds the edge (u, v) to A, and line 8 merges the vertices in the two trees.

Time Complexity:

In Kruskal's algorithm, most time consuming operation is sorting because the total complexity of the Disjoint-Set operations will be O(E lg V), which is the overall Time Complexity of the algorithm.

Prim's Algorithm

Prim's Algorithm also uses Greedy approach to find the minimum spanning tree. In Prim's Algorithm we grow the spanning tree from a starting position. Unlike an edge in Kruskal's, we add vertex to the growing spanning tree in Prim's.

Algorithm Steps:

1) Maintain two disjoint sets of vertices. One is containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
2) Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues. Insert the vertices that are connected to growing spanning tree, into the Priority Queue.
3) Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.

```
MST-PRIM(G, w, r)
 1  for each u ∈ G.V
 2      u.key = ∞
 3      u.π = NIL
 4  r.key = 0
 5  Q = G.V
 6  while Q ≠ ∅
 7      u = EXTRACT-MIN(Q)
 8      for each v ∈ G.Adj[u]
 9          if v ∈ Q and w(u, v) < v.key
10              v.π = u
11              v.key = w(u, v)
```

In the pseudocode, the connected graph G and the root r of the minimum spanning tree to be grown are inputs to the algorithm. During execution of the algorithm, all vertices that are not in the tree reside in a min-priority queue Q based on a key attribute. For each vertex v, the attribute v.key is the minimum weight of any edge connecting v to a vertex in the tree; by convention, v.key = ∞ if there is no such edge. The attribute v.π names the parent of v in the tree.

Lines 1–5 set the key of each vertex to ∞ (except for the root r, whose key is set to 0 so that it will be the first vertex processed), set the parent of each vertex to NIL, and initialize the minpriority queue Q to contain all the vertices. Line 7 identifies a vertex u ε Q incident on a light edge that crosses the cut (V – Q, Q) (with the exception of the first iteration, in which u = r due to line 4). Removing u from the set Q adds it to the set V - Q of vertices in the tree, thus adding (u, u.π) to A. The for loop of lines 8–11 updates the key and π attributes of every vertex v adjacent to u but not in the tree.

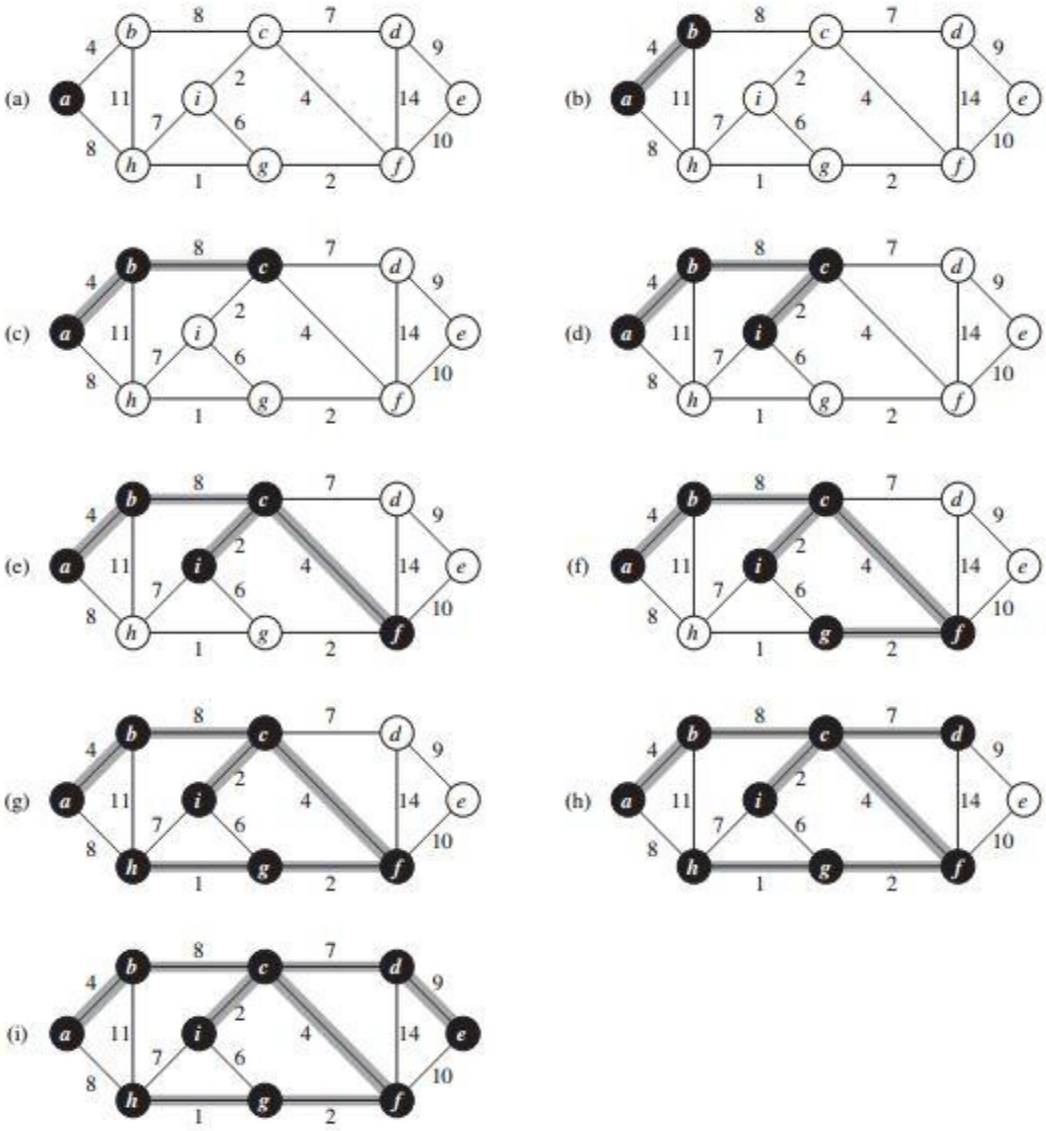


Figure: The execution of Prim's algorithm on the graph. The root vertex is a. Shaded edges are in the tree being grown, and black vertices are in the tree. At each step of the algorithm, the vertices in the tree determine a cut of the graph, and a light edge crossing the cut is added to the tree. In the second step, for example, the algorithm has a choice of adding either edge (b, c) or edge (a, h) to the tree since both are light edges crossing the cut.

Time Complexity:

A binary min-heap supports in O(lg V) time.

Thus, the total time for Prim's algorithm is O(V lg V + E lg V) = O(E lg V), which is asymptotically the same as for our implementation of Kruskal's algorithm. If we use a Fibonacci heap to implement the min-priority queue Q, the running time of Prim's algorithm improves to O(E + V lg V).

Single-Source Shortest Paths:

We define the ***shortest-path weight*** $\delta(u, v)$ from u to v by

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

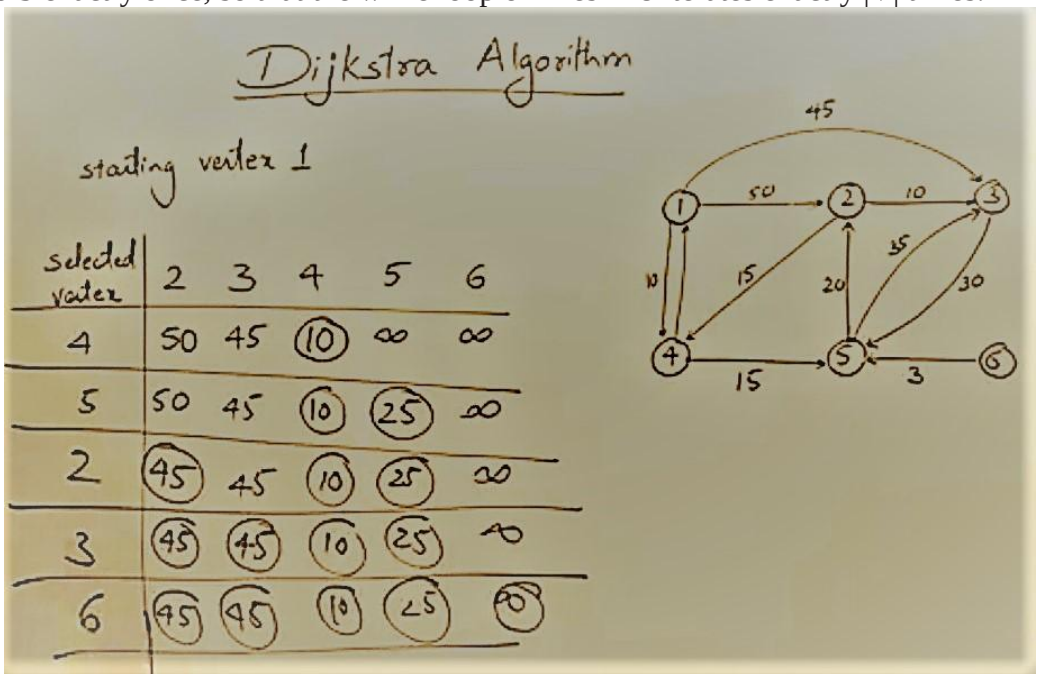
Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u. In the following implementation, we use a min-priority queue Q of vertices, keyed by their d values.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

INITIALIZE-SINGLE-SOURCE(G, s)
1. for each vertex v ∈ G.v
2. v.d = ∞
3. v.π = NIL
4. s.d = 0

RELAX(u, v, w)
1 if v.d > u.d + w(u, v)
2 v.d = u.d + w(u, v)
3 v.π = u

Dijkstra's algorithm relaxes edges as shown in Figure 24.6. Line 1 initializes the d and π values in the usual way, and line 2 initializes the set S to the empty set. The algorithm maintains the invariant that $Q = V - S$ at the start of each iteration of the **while** loop of lines 4–8. Line 3 initializes the min-priority queue Q to contain all the vertices in V; since $S = \emptyset$ at that time, the invariant is true after line 3. Each time through the **while** loop of lines 4–8, line 5 extracts a vertex u from $Q = V - S$ and line 6 adds it to set S, thereby maintaining the invariant. (The first time through this loop, $u = s$.) Vertex u, therefore, has the smallest shortest-path estimate of any vertex in $V - S$. Then, lines 7–8 relax each edge (u, v) leaving u, thus updating the estimate $v.d$ and the predecessor $v.\pi$ if we can improve the shortest path to v found so far by going through u. Observe that the algorithm never inserts vertices into Q after line 3 and that each vertex is extracted from Q and added to S exactly once, so that the while loop of lines 4–8 iterates exactly |V| times.



Example:

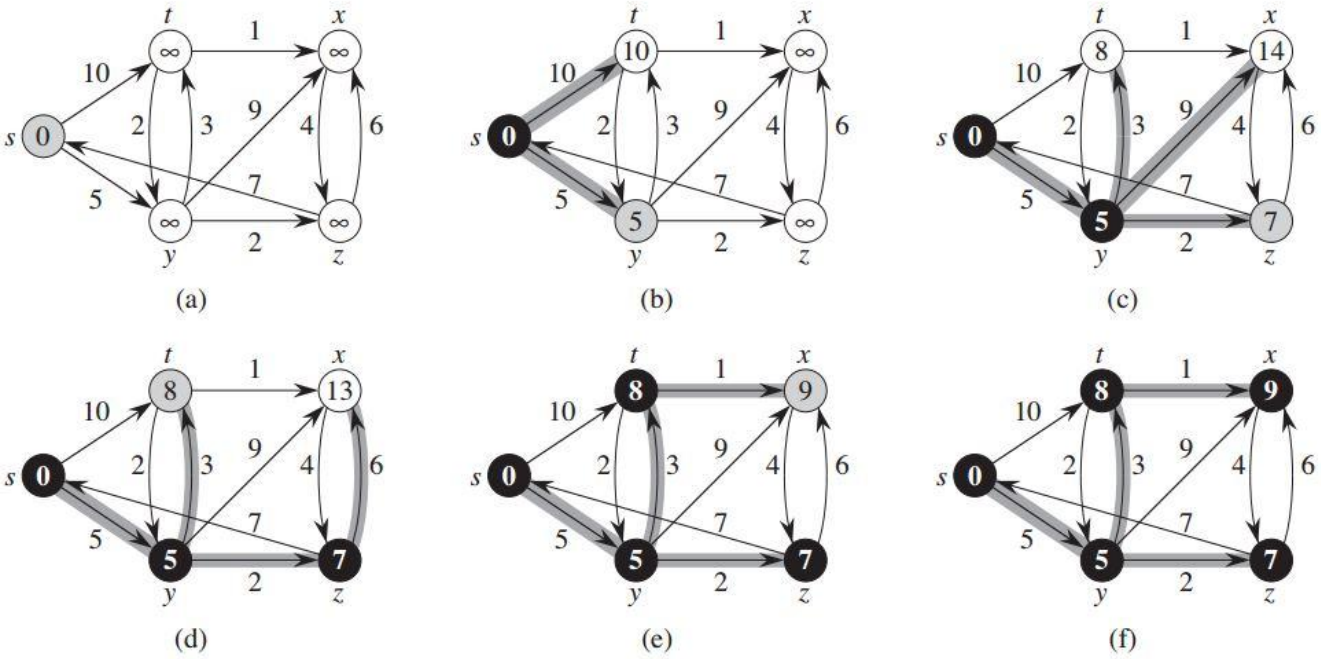


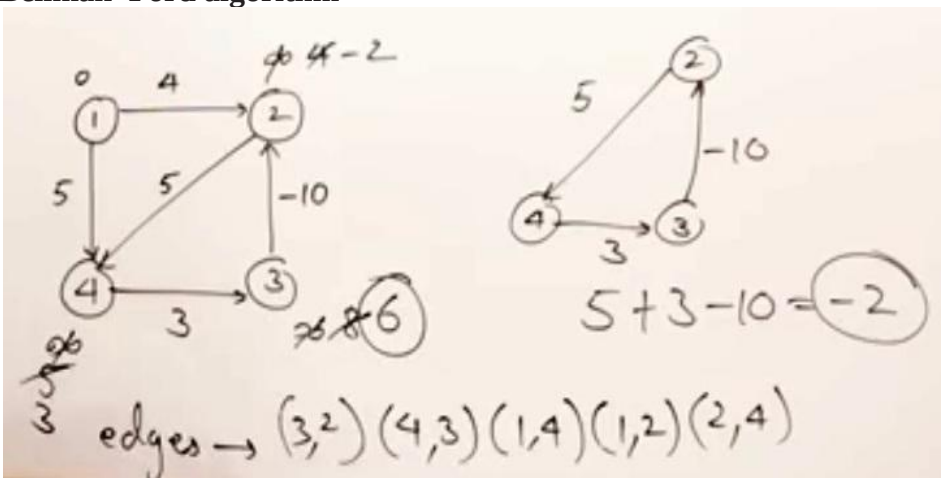
Figure The execution of Dijkstra's algorithm. The source s is the leftmost vertex. The shortest-path estimates appear within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set S, and white vertices are in the min-priority queue $Q = V - S$. **(a)** The situation just before the first iteration of the **while** loop of lines 4–8. The shaded vertex has the minimum d value and is chosen as vertex u in line 5. **(b)–(f)** The situation after each successive iteration of the **while** loop. The shaded vertex in each part is chosen as vertex u in line 5 of the next iteration. The d values and predecessors shown in part (f) are the final values.

Time Complexity:

Time Complexity of the implementation is $O(V^2)$. If the input graph is represented using adjacency list, it can be reduced to O(E log V) with the help of binary heap.

Note: Dijkstra's algorithm doesn't work for graphs with negative weight edges. For graphs with negative weight edges, Bellman–Ford algorithm can be used.

Bellman–Ford algorithm



The Bellman-Ford algorithm solves the single-source shortest-paths problem in which edge weights may be negative. It returns a boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source. If there is such a cycle, the algorithm indicates that no solution exists. If there is no such cycle, it produces the shortest paths and their weights.

24.1 The Bellman-Ford algorithm

The ***Bellman-Ford algorithm*** solves the single-source shortest-paths problem in the general case in which edge weights may be negative. Given a weighted, directed graph $G = (V, E)$ with source s and weight function $w : E \to \mathbb{R}$, the Bellman-Ford algorithm returns a boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source. If there is such a cycle, the algorithm indicates that no solution exists. If there is no such cycle, the algorithm produces the shortest paths and their weights.

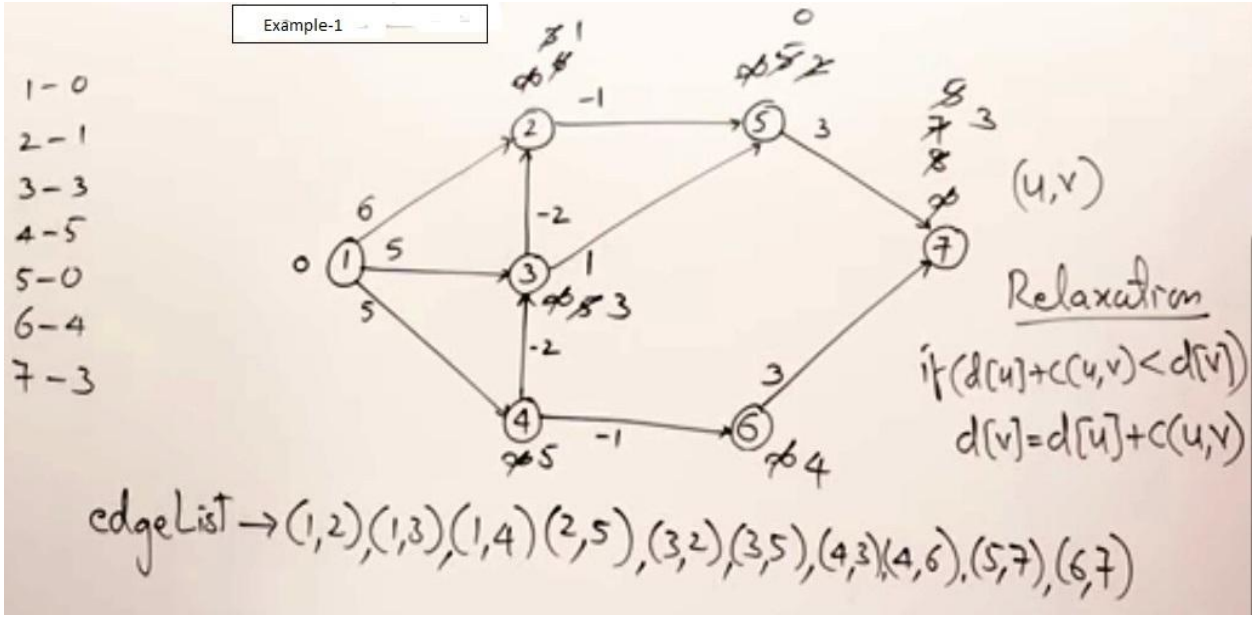
The algorithm relaxes edges, progressively decreasing an estimate $v.d$ on the weight of a shortest path from the source s to each vertex $v \in V$ until it achieves the actual shortest-path weight $\delta(s, v)$. The algorithm returns TRUE if and only if the graph contains no negative-weight cycles that are reachable from the source.

```
BELLMAN-FORD(G, w, s)                                  Time Complexity O(V.E)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3       for each edge (u, v) ∈ G.E
4            RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6       if v.d > u.d + w(u, v)
7            return FALSE
8  return TRUE
```

Figure 24.4 shows the execution of the Bellman-Ford algorithm on a graph with 5 vertices. After initializing the d and π values of all vertices in line 1, the algorithm makes $|V| - 1$ passes over the edges of the graph. Each pass is one iteration of the **for** loop of lines 2–4 and consists of relaxing each edge of the graph once. Figures 24.4(b)–(e) show the state of the algorithm after each of the four passes over the edges. After making $|V| - 1$ passes, lines 5–8 check for a negative-weight cycle and return the appropriate boolean value. (We'll see a little later why this check works.)

The Bellman-Ford algorithm runs in time $O(VE)$, since the initialization in line 1 takes $\Theta(V)$ time, each of the $|V| - 1$ passes over the edges in lines 2–4 takes $\Theta(E)$ time, and the **for** loop of lines 5–7 takes $O(E)$ time.

To prove the correctness of the Bellman-Ford algorithm, we start by showing that if there are no negative-weight cycles, the algorithm computes correct shortest-path weights for all vertices reachable from the source.



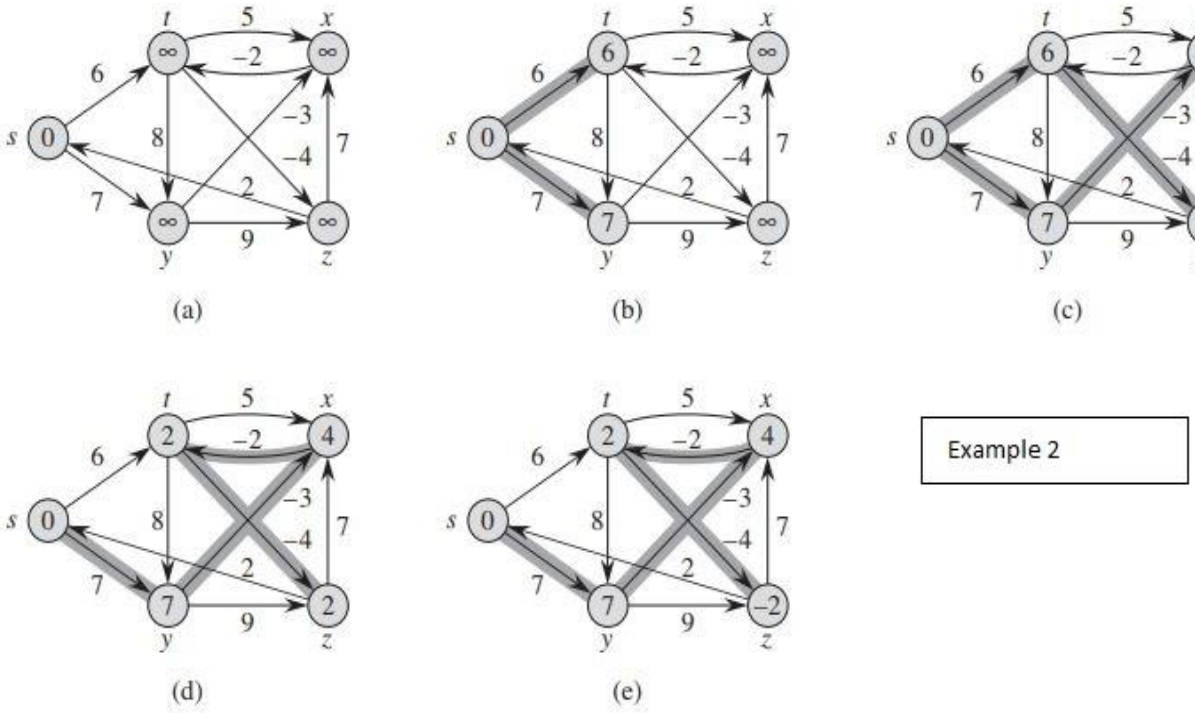


Figure The execution of the Bellman-Ford algorithm. The source is vertex s. The d values appear within the vertices, and shaded edges indicate predecessor values: if edge (u, v) is shaded, then $v.\pi = u$. In this particular example, each pass relaxes the edges in the order $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$. **(a)** The situation just before the first pass over the edges. **(b)–(e)** The situation after each successive pass over the edges. The d and π values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.

The Floyd-Warshall algorithm

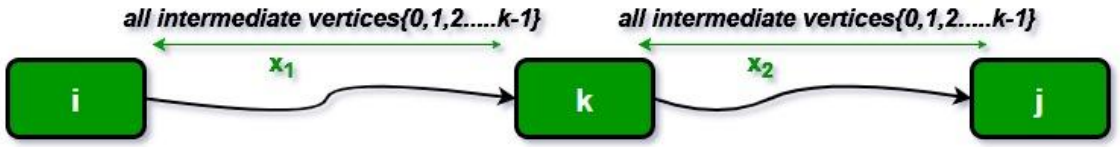
The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, ... k-1} as intermediate vertices. For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.

1) k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.

2) k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j] if dist[i][j] > dist[i][k] + dist[k][j]

The following figure shows the above optimal substructure property in the all-pairs shortest path problem.



Example 1

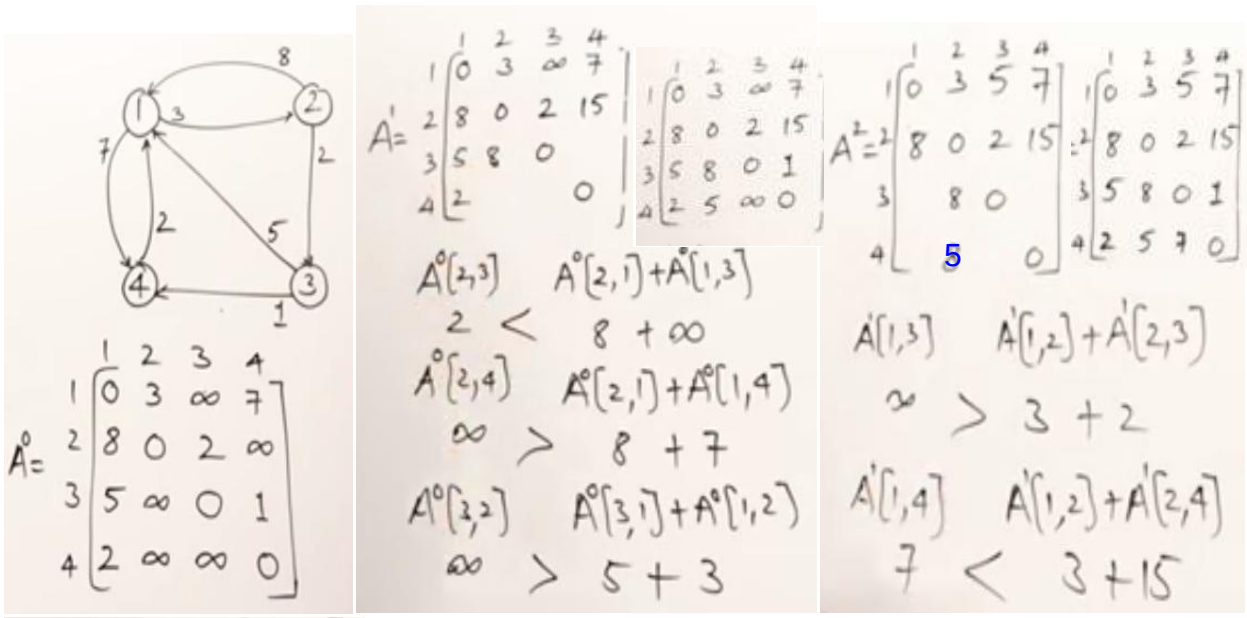
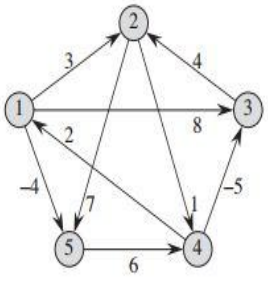


Figure 25.1 Example 2



PRINT-ALL-PAIRS-SHORTEST-PATH (π,i,j)
if i == j
print i
elseif πij == NIL
print "no path from" i "to" j "exists"
else PRINT-ALL-PAIRS-SHORTEST-PATH (π,i, πij)
print j

```
FLOYD-WARSHALL(W)
1  n = W.rows
2  D^(0) = W
3  for k = 1 to n
4      let D^(k) = (d_ij^(k)) be a new n × n matrix
5      for i = 1 to n
6          for j = 1 to n
7              d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
8  return D^(n)
```

Figure 25.4 shows the matrices $D^{(k)}$ computed by the Floyd-Warshall algorithm for the graph in Figure 25.1.

The running time of the Floyd-Warshall algorithm is determined by the triply nested **for** loops of lines 3–7. Because each execution of line 7 takes $O(1)$ time, the algorithm runs in time $\Theta(n^3)$. As in the final algorithm in Section 25.1, the code is tight, with no elaborate data structures, and so the constant hidden in the Θ-notation is small. Thus, the Floyd-Warshall algorithm is quite practical for even moderate-sized input graphs.

Constructing a shortest path

There are a variety of different methods for constructing shortest paths in the Floyd-Warshall algorithm. One way is to compute the matrix D of shortest-path weights and then construct the predecessor matrix Π from the D matrix. Exercise 25.1-6 asks you to implement this method so that it runs in $O(n^3)$ time. Given the predecessor matrix Π, the PRINT-ALL-PAIRS-SHORTEST-PATH procedure will print the vertices on a given shortest path.

Alternatively, we can compute the predecessor matrix Π while the algorithm computes the matrices $D^{(k)}$. Specifically, we compute a sequence of matrices $\Pi^{(0)}, \Pi^{(1)}, \ldots, \Pi^{(n)}$, where $\Pi = \Pi^{(n)}$ and we define $\pi_{ij}^{(k)}$ as the predecessor of vertex j on a shortest path from vertex i with all intermediate vertices in the set $\{1, 2, \ldots, k\}$.

We can give a recursive formulation of $\pi_{ij}^{(k)}$. When $k = 0$, a shortest path from i to j has no intermediate vertices at all. Thus,

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{if } i = j \text{ or } w_{ij} = \infty\,, \\ i & \text{if } i \neq j \text{ and } w_{ij} < \infty\,. \end{cases} \tag{25.6}$$

For $k \geq 1$, if we take the path $i \rightsquigarrow k \rightsquigarrow j$, where $k \neq j$, then the predecessor of j we choose is the same as the predecessor of j we chose on a shortest path from k with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. Otherwise, we

choose the same predecessor of j that we chose on a shortest path from i with all intermediate vertices in the set {1, 2,.., k-1}.

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Figure 25.4 The sequence of matrices $D^{(k)}$ and $\Pi^{(k)}$ computed by the Floyd-Warshall algorithm for the graph in Figure 25.1.

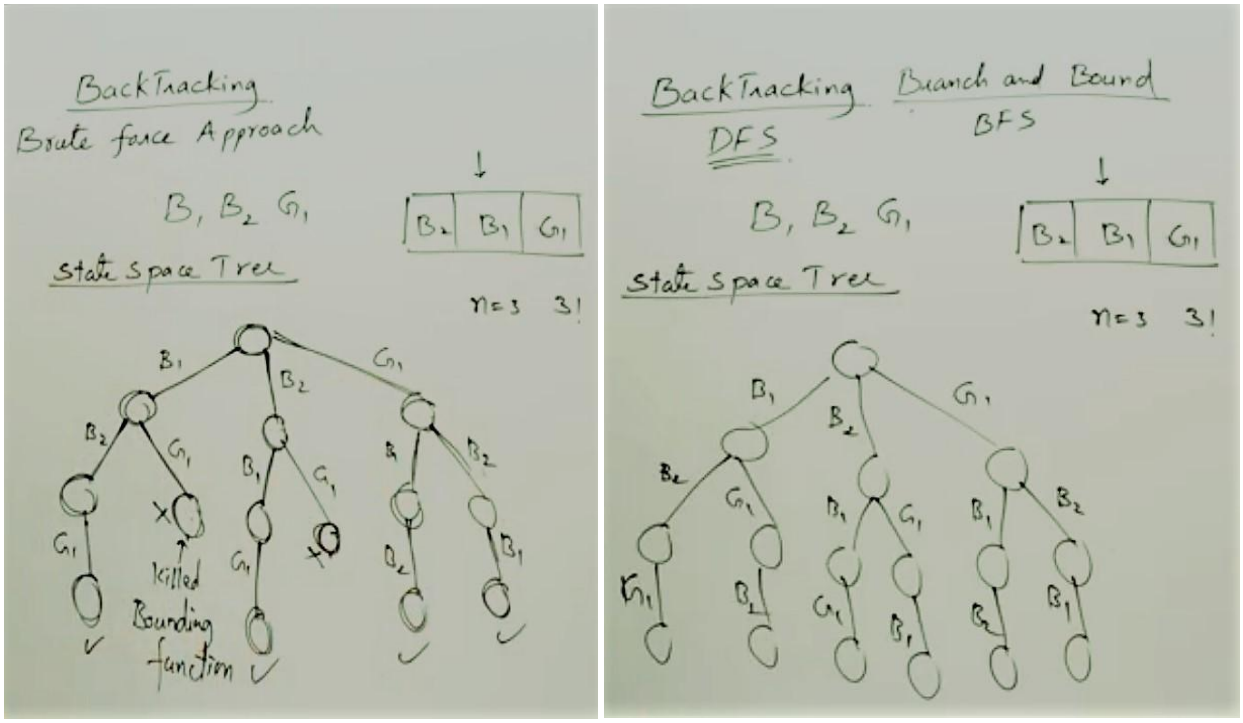
Difference between Backtracking & Branch and Bound

Backtracking:

* It is used to find all possible solution available to the problem
*It traverse tree by DFS (Depth first Search)
*It realizes that it has made a bad choice & undoes the last choice by tracking up.
*It search the state space tree until it found a solution
Some Problem Solved with Backtracking Technique
– N- Queens Problem, Sum of Subset, Sudoku Puzzle, Hamiltonian Cycle

Branch-and-Bound (BB):

*It is used to solve optimization problem
*It may traverse the tree by BFS or DFS.
*It realizes that is already has a better optional solution that the pre-solution leads to so it abandons that pre-solution.
*It completely searches the ***state space solution tree*** to get optional solution.
Some Problem Solved with Branch and Bound Technique
- Traveling Salesman Problem, Vertex Cover Problem

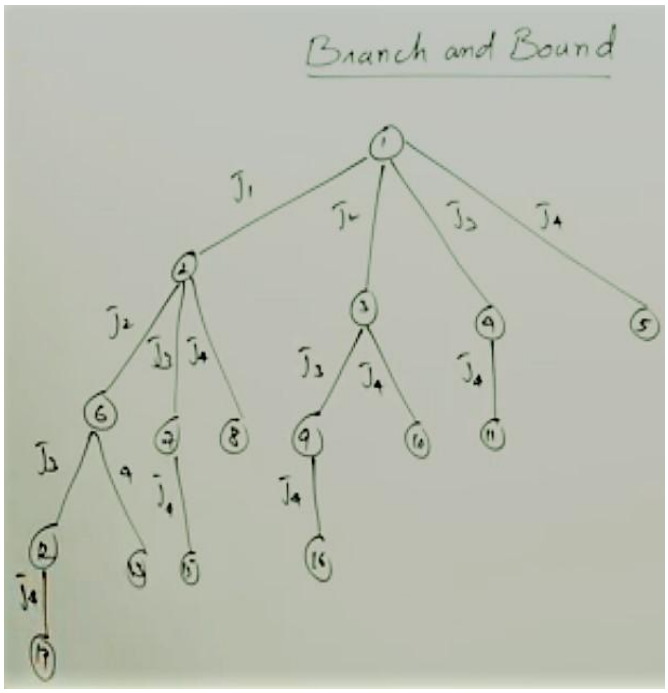


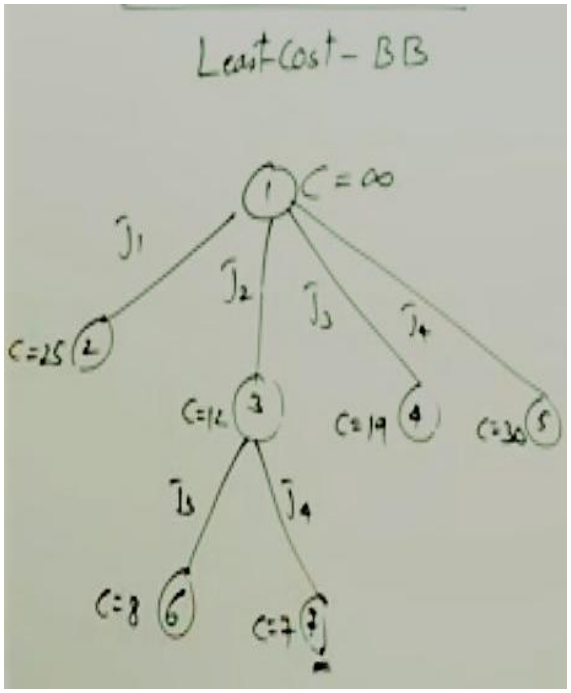
In the above figure ->***Bounding Function:*** No girl sit in the middle of two boys
Types of Branch and Bound

1. FIFO Branch and Bound Search: This leads to a breadth-first search.
For this we will use a data structure called Queue. Initially Queue is empty.

2. LIFO Branch and Bound Search: This leads to a depth-first search
For this we will use a data structure called stack. Initially stack is empty.

3. LC (lowest/least cost) Branch and Bound Search: the branch is extended by the node which adds the lowest additional costs, according to a given cost function.



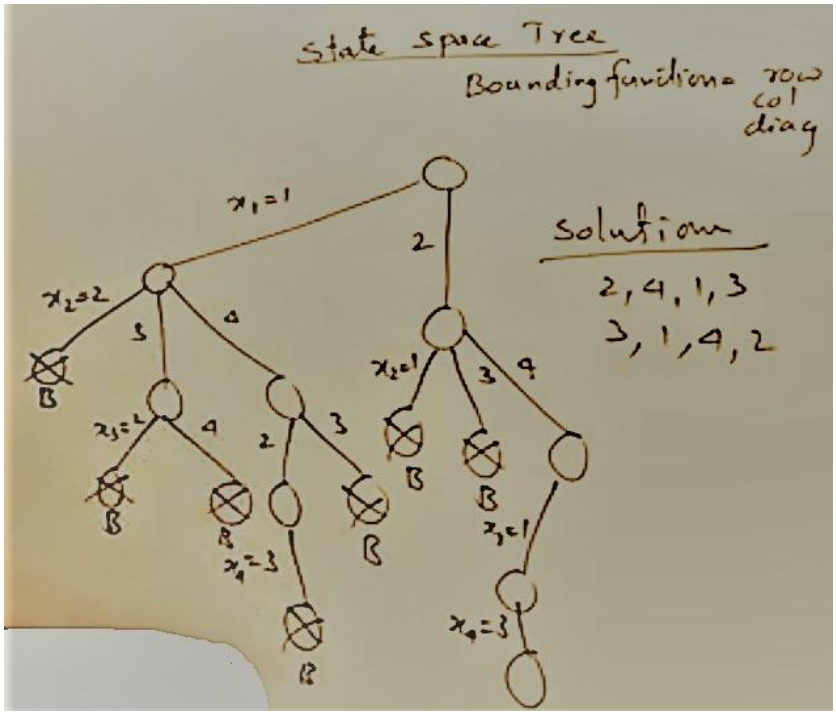


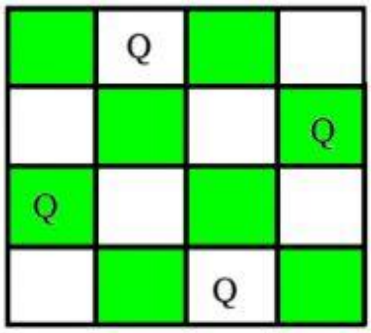
N-Queens Problem

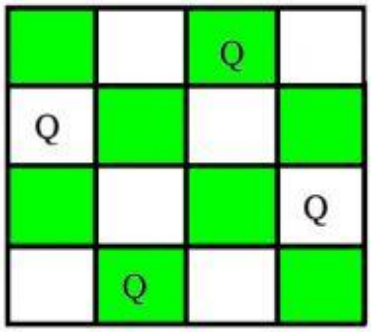
Nauck made an 8X8 Chessboard to find the first Feasible Solution. In a NxN square board N – number of queens need to be placed considering three Condition ---

• No two Queens can be placed in same row, same column, and same diagonal.

Time Complexity of N-Queen Problem is O (n!)







```
1   Algorithm Place(k, i)
2   // Returns true if a queen can be placed in kth row and
3   // ith column. Otherwise it returns false. x[ ] is a
4   // global array whose first (k − 1) values have been set.
5   // Abs(r) returns the absolute value of r.
6   {
7       for j := 1 to k − 1 do
8           if ((x[j] = i) // Two in the same column
9                or (Abs(x[j] − i) = Abs(j − k)))
10                   // or in the same diagonal
11               then return false;
12      return true;
13  }
```

Algorithm 7.4 Can a new queen be placed?

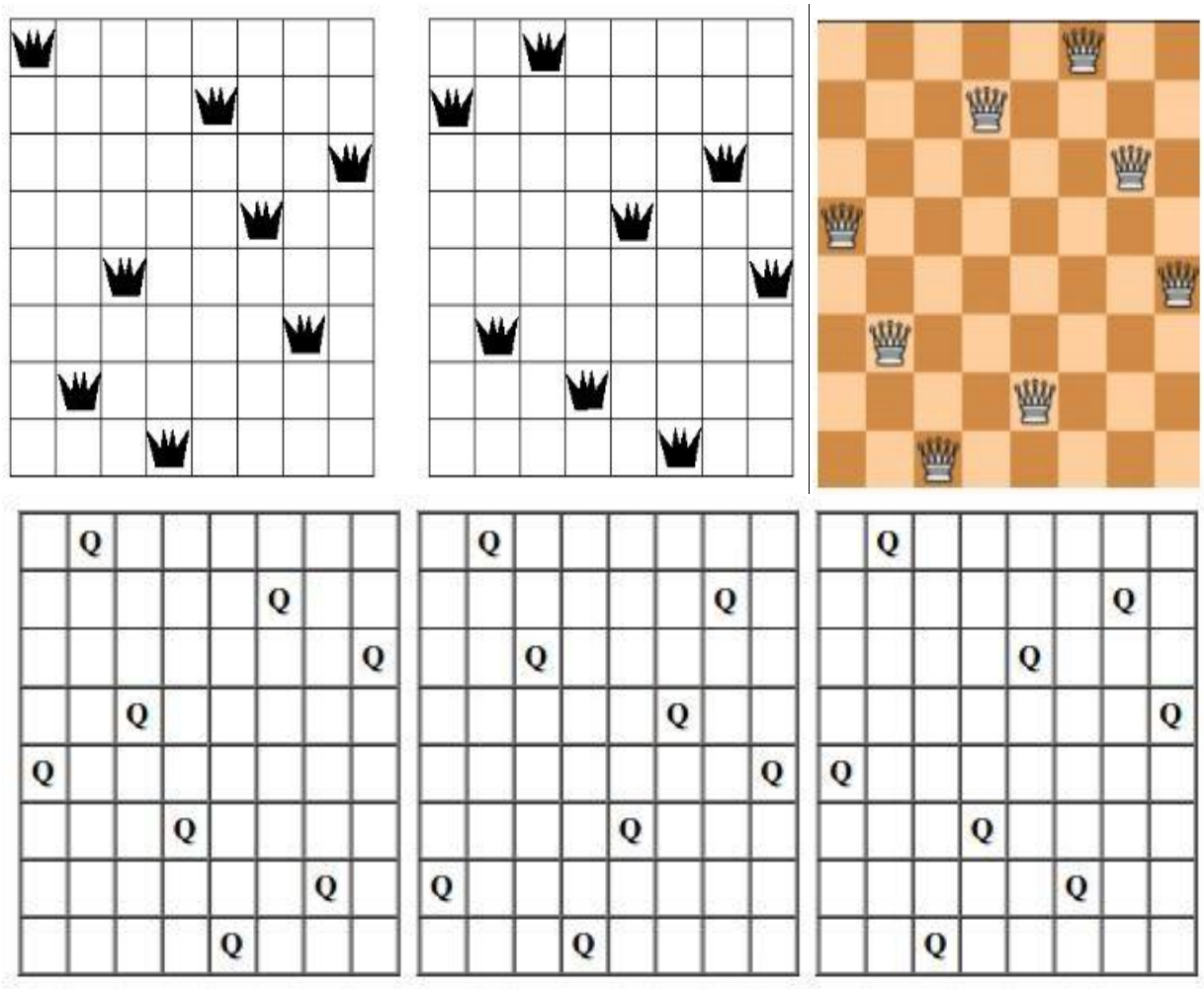
```
1   Algorithm NQueens(k, n)
2   // Using backtracking, this procedure prints all
3   // possible placements of n queens on an n × n
4   // chessboard so that they are nonattacking.
5   {
6       for i := 1 to n do
7       {
8           if Place(k, i) then
9           {
10              x[k] := i;
11              if (k = n) then write (x[1 : n]);
12              else NQueens(k + 1, n);
13          }
14      }
15  }
```

Algorithm 7.5 All solutions to the n-queens problem

Backtracking Algorithm

The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes then we backtrack and return false.

1) Start in the leftmost column
2) If all queens are placed return true
3) Try all rows in the current column. Do following for every tried row.
    a) If the queen can be placed safely in this row then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.
    b) If placing queen in [row, column] leads to a solution then return true.
    c) If placing queen doesn't lead to a solution then unmark this [row, column] (Backtrack) and go to step (a) to try other rows.
3) If all rows have been tried and nothing worked, return false to trigger backtracking.



Altogether, ***92*** solutions to the problem of ***EIGHT QUEENS.***

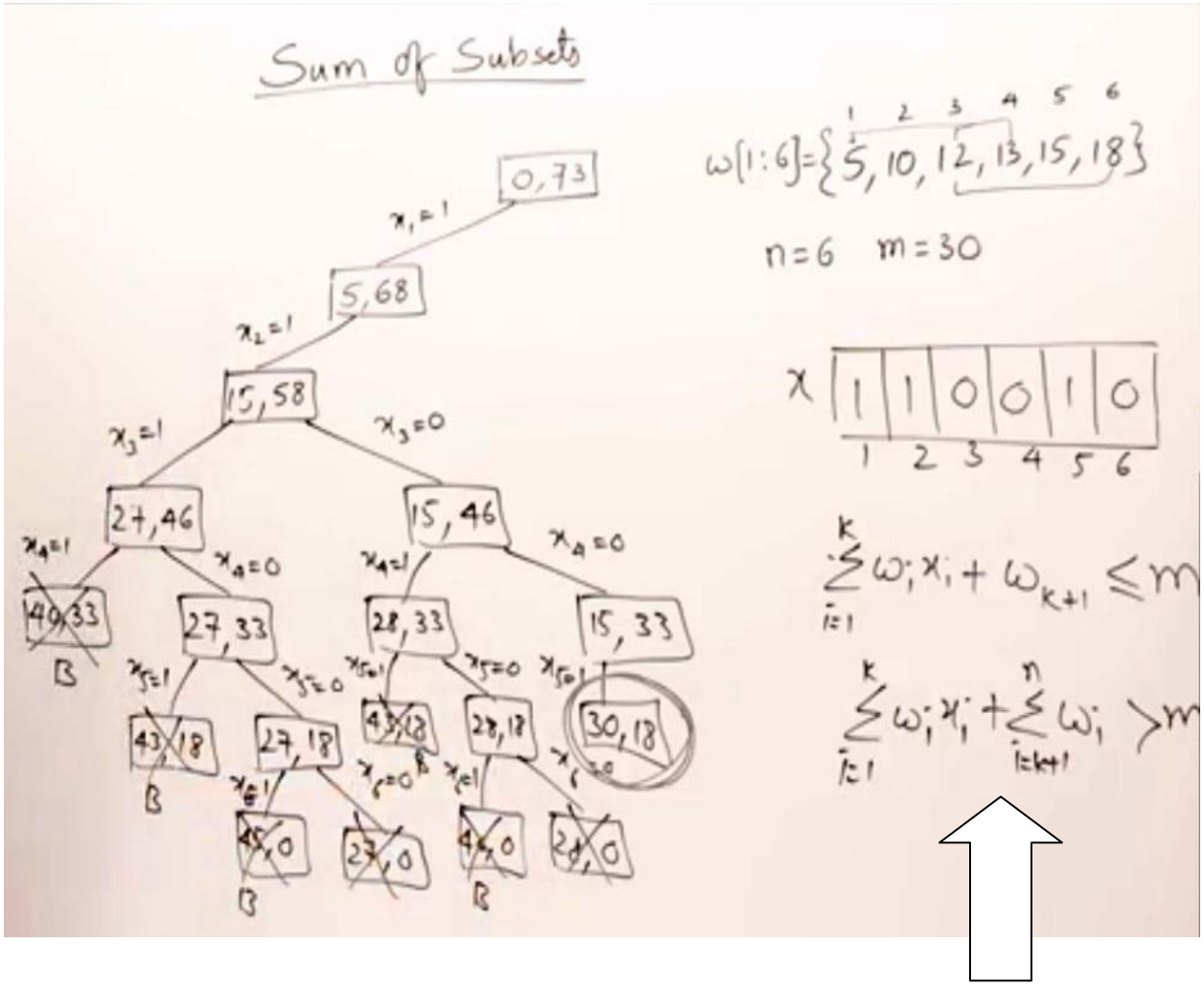
Sub Set Sum Problem/ Sum of Subset Problem

The Sum of Subset Problem is, there will be a set of distinct positive Numbers X and a given Number N. Now, we have to find all the combination of numbers from X that sum up to N. Make a Set of that Number.

Backtracking Approach:

First, organize the numbers in non decreasing order. Then generating a tree, we can find the solution by adding the weight of the nodes. Note that, here more than necessary nodes can be generated. The algorithm will search for the weighted amount of the nodes. If it goes beyond given Number N, then it stops generating nodes and move to other parent nodes for finding solution.

Example 1:

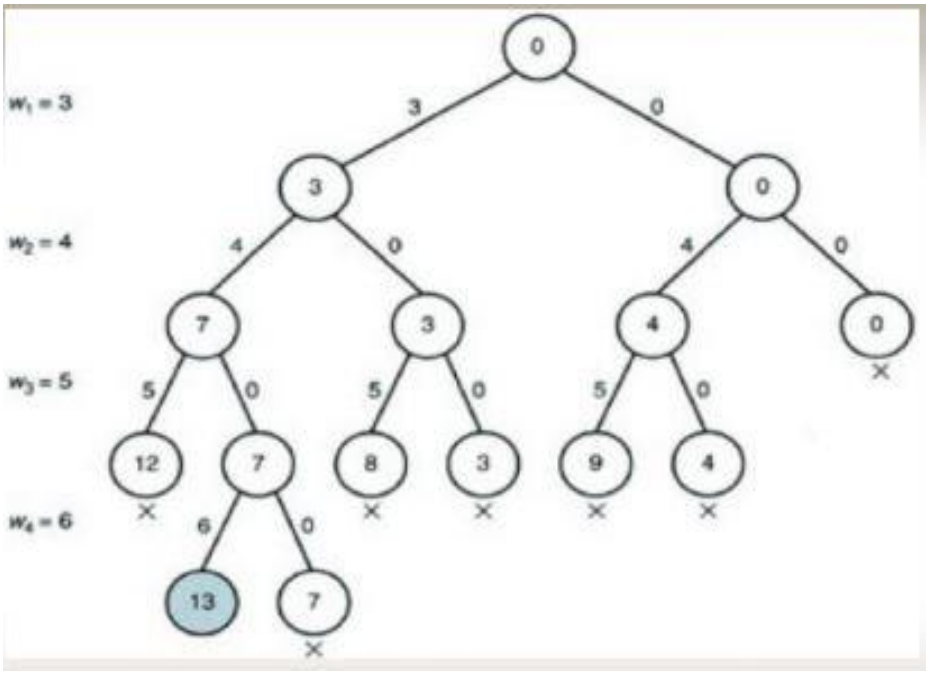


BOUNDING FUNCTION

Time Complexity of Sum of Subset Problem: O (2^n)

Example: 2

W = {4, 5, 6, 3} and M = 13




```
1   Algorithm SumOfSub(s, k, r)
2   // Find all subsets of w[1 : n] that sum to m. The values of x[j],
3   // 1 ≤ j < k, have already been determined. s = Σ_{j=1}^{k-1} w[j] * x[j]
4   // and r = Σ_{j=k}^{n} w[j]. The w[j]'s are in nondecreasing order.
5   // It is assumed that w[1] ≤ m and Σ_{i=1}^{n} w[i] ≥ m.
6   {
7       // Generate left child. Note: s + w[k] ≤ m since B_{k-1} is true.
8       x[k] := 1;
9       if (s + w[k] = m) then write (x[1 : k]); // Subset found
10          // There is no recursive call here as w[j] > 0, 1 ≤ j ≤ n.
11      else if (s + w[k] + w[k + 1] ≤ m)
12          then SumOfSub(s + w[k], k + 1, r − w[k]);
13      // Generate right child and evaluate B_k.
14      if ((s + r − w[k] ≥ m) and (s + w[k + 1] ≤ m)) then
15      {
16          x[k] := 0;
17          SumOfSub(s, k + 1, r − w[k]);
18      }
19  }
```

Recursive backtracking algorithm for Sum of Subset Problem

32 String Matching

Text-editing programs frequently need to find all occurrences of a pattern in the text. Typically, the text is a document being edited, and the pattern searched for is a particular word supplied by the user. Efficient algorithms for this problem—called "string matching"—can greatly aid the responsiveness of the text-editing program. Among their many other applications, string-matching algorithms search for particular patterns in DNA sequences. Internet search engines also use them to find Web pages relevant to queries.

We formalize the string-matching problem as follows. We assume that the text is an array $T[1..n]$ of length n and that the pattern is an array $P[1..m]$ of length $m \leq n$. We further assume that the elements of P and T are characters drawn from a finite alphabet Σ. For example, we may have $\Sigma = \{0, 1\}$ or $\Sigma = \{a, b, \ldots, z\}$. The character arrays P and T are often called ***strings*** of characters.

Referring to Figure 32.1, we say that pattern P ***occurs with shift s*** in text T (or, equivalently, that pattern P ***occurs beginning at position s + 1*** in text T) if $0 \leq s \leq n - m$ and $T[s + 1..s + m] = P[1..m]$ (that is, if $T[s + j] = P[j]$, for $1 \leq j \leq m$). If P occurs with shift s in T, then we call s a ***valid shift***; otherwise, we call s an ***invalid shift***. The ***string-matching problem*** is the problem of finding all valid shifts with which a given pattern P occurs in a given text T.

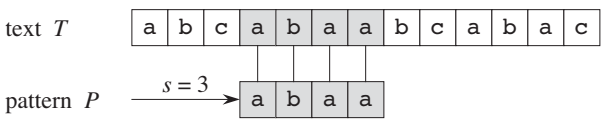


Figure 32.1 An example of the string-matching problem, where we want to find all occurrences of the pattern $P = \texttt{abaa}$ in the text $T = \texttt{abcabaabcabac}$. The pattern occurs only once in the text, at shift $s = 3$, which we call a valid shift. A vertical line connects each character of the pattern to its matching character in the text, and all matched characters are shaded.

32.1 The naive string-matching algorithm

The naive algorithm finds all valid shifts using a loop that checks the condition $P[1\,..\,m] = T[s+1\,..\,s+m]$ for each of the $n-m+1$ possible values of s.

```
NAIVE-STRING-MATCHER(T, P)
1  n = T.length
2  m = P.length
3  for s = 0 to n − m
4      if P[1..m] == T[s + 1..s + m]
5          print "Pattern occurs with shift" s
```

Figure 32.4 portrays the naive string-matching procedure as sliding a "template" containing the pattern over the text, noting for which shifts all of the characters on the template equal the corresponding characters in the text. The **for** loop of lines 3–5 considers each possible shift explicitly. The test in line 4 determines whether the current shift is valid; this test implicitly loops to check corresponding character positions until all positions match successfully or a mismatch is found. Line 5 prints out each valid shift s.

Procedure NAIVE-STRING-MATCHER takes time $O((n-m+1)m)$, and this bound is tight in the worst case. For example, consider the text string $\mathtt{a}^n$ (a string of n a's) and the pattern $\mathtt{a}^m$. For each of the $n-m+1$ possible values of the shift s, the implicit loop on line 4 to compare corresponding characters must execute m times to validate the shift. The worst-case running time is thus $\Theta((n-m+1)m)$, which is $\Theta(n^2)$ if $m = \lfloor n/2 \rfloor$. Because it requires no preprocessing, NAIVE-STRING-MATCHER's running time equals its matching time.

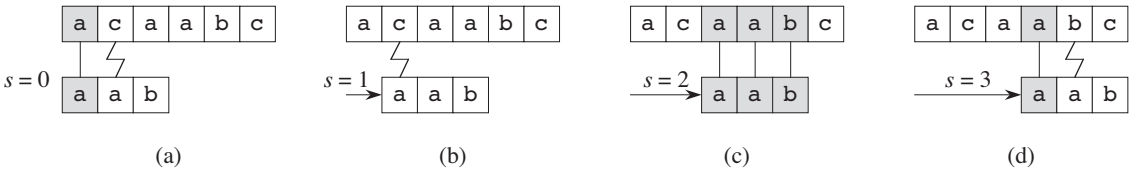


Figure 32.4 The operation of the naive string matcher for the pattern $P = \mathtt{aab}$ and the text $T = \mathtt{acaabc}$. We can imagine the pattern P as a template that we slide next to the text. **(a)–(d)** The four successive alignments tried by the naive string matcher. In each part, vertical lines connect corresponding regions found to match (shown shaded), and a jagged line connects the first mismatched character found, if any. The algorithm finds one occurrence of the pattern, at shift $s = 2$, shown in part (c).

KMP (Knuth Morris Pratt) Pattern Searching

The Naive pattern searching algorithm doesn't work well in cases where we see many matching characters followed by a mismatching character. Following are some examples.

```
txt[] = "AAAAAAAAAAAAAAAAAB"
pat[] = "AAAAB"

txt[] = "ABABABCABABABCABABABC"
pat[] =  "ABABAC" (not a worst case, but a bad case for Naive)
```

The KMP matching algorithm uses degenerating property (pattern having same sub-patterns appearing more than once in the pattern) of the pattern and improves the worst case complexity to O(n). The basic idea behind KMP's algorithm is: whenever we detect a mismatch (after some matches), we already know some of the characters in the text of the next window. We take advantage of this information to avoid matching the characters that we know will anyway match. Let us consider below example to understand this.

```
Matching Overview
txt = "AAAAABAAABA"
pat = "AAAA"

We compare first window of txt with pat
txt = "AAAAABAAABA"
pat = "AAAA"  [Initial position]
We find a match. This is same as Naive String Matching.

In the next step, we compare next window of txt with pat.
txt = "AAAAABAAABA"
pat =  "AAAA" [Pattern shifted one position]
This is where KMP does optimization over Naive. In this
second window, we only compare fourth A of pattern
with fourth character of current window of text to decide
whether current window matches or not. Since we know
first three characters will anyway match, we skipped
matching first three characters.

Need of Preprocessing?
An important question arises from the above explanation,
how to know how many characters to be skipped. To know this,
we pre-process pattern and prepare an integer array
lps[] that tells us the count of characters to be skipped.
```

Preprocessing Overview:

- KMP algorithm preprocesses pat[] and constructs an auxiliary **lps[]** of size m (same as size of pattern) which is used to skip characters while matching.
- **name lps indicates longest proper prefix which is also suffix.**. A proper prefix is prefix with whole string **not** allowed. For example, prefixes of "ABC" are "", "A", "AB" and "ABC". Proper prefixes are "", "A" and "AB". Suffixes of the string are "", "C", "BC" and "ABC".
- We search for lps in sub-patterns. More clearly we focus on sub-strings of patterns that are either prefix and suffix.
- For each sub-pattern pat[0..i] where i = 0 to m-1, lps[i] stores length of the maximum matching proper prefix which is also a suffix of the sub-pattern pat[0..i].

```
lps[i] = the longest proper prefix of pat[0..i]
           which is also a suffix of pat[0..i].
```

Note : lps[i] could also be defined as longest prefix which is also proper suffix. We need to use properly at one place to make sure that the whole substring is not considered.

```
Examples of lps[] construction:
For the pattern “AAAA”,
lps[] is [0, 1, 2, 3]

For the pattern “ABCDE”,
lps[] is [0, 0, 0, 0, 0]

For the pattern “AABAACAABAA”,
lps[] is [0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5]

For the pattern “AAACAAAAAC”,
lps[] is [0, 1, 2, 0, 1, 2, 3, 3, 3, 4]

For the pattern “AAABAAA”,
lps[] is [0, 1, 2, 0, 1, 2, 3]
```

Searching Algorithm:

Unlike Naive algorithm, where we slide the pattern by one and compare all characters at each shift, we use a value from lps[] to decide the next characters to be matched. The idea is to not match a character that we know will anyway match.

How to use lps[] to decide next positions (or to know a number of characters to be skipped)?

- We start comparison of pat[j] with j = 0 with characters of current window of text.
- We keep matching characters txt[i] and pat[j] and keep incrementing i and j while pat[j] and txt[i] keep **matching**.
- When we see a **mismatch**
  - We know that characters pat[0..j-1] match with txt[i-j+1…i-1] (Note that j starts with 0 and increment it only when there is a match).
  - We also know (from above definition) that lps[j-1] is count of characters of pat[0…j-1] that are both proper prefix and suffix.
  - From above two points, we can conclude that we do not need to match these lps[j-1] characters with txt[i-j…i-1] because we know that these characters will anyway match. Let us consider above example to understand this.

```
txt[] = "AAAAABAAABA"
pat[] = "AAAA"
lps[] = {0, 1, 2, 3}

i = 0, j = 0
txt[] = "AAAAABAAABA"
pat[] = "AAAA"
txt[i] and pat[j] match, do i++, j++

i = 1, j = 1
txt[] = "AAAAABAAABA"
pat[] = "AAAA"
txt[i] and pat[j] match, do i++, j++
```

```
i = 2, j = 2
txt[] = "AAAAABAAABA"
pat[] = "AAAA"
pat[i] and pat[j] match, do i++, j++

i = 3, j = 3
txt[] = "AAAAABAAABA"
pat[] = "AAAA"
txt[i] and pat[j] match, do i++, j++

i = 4, j = 4
Since j == M, print pattern found and reset j,
j = lps[j-1] = lps[3] = 3

Here unlike Naive algorithm, we do not match first three
characters of this window. Value of lps[j-1] (in above
step) gave us index of next character to match.
i = 4, j = 3
txt[] = "AAAAABAAABA"
pat[] =  "AAAA"
txt[i] and pat[j] match, do i++, j++

i = 5, j = 4
Since j == M, print pattern found and reset j,
j = lps[j-1] = lps[3] = 3

Again unlike Naive algorithm, we do not match first three
characters of this window. Value of lps[j-1] (in above
step) gave us index of next character to match.
i = 5, j = 3
txt[] = "AAAAABAAABA"
pat[] =   "AAAA"
txt[i] and pat[j] do NOT match and j > 0, change only j
j = lps[j-1] = lps[2] = 2

i = 5, j = 2
txt[] = "AAAAABAAABA"
pat[] =    "AAAA"
txt[i] and pat[j] do NOT match and j > 0, change only j
j = lps[j-1] = lps[1] = 1

i = 5, j = 1
txt[] = "AAAAABAAABA"
pat[] =     "AAAA"
txt[i] and pat[j] do NOT match and j > 0, change only j
j = lps[j-1] = lps[0] = 0

i = 5, j = 0
txt[] = "AAAAABAAABA"
pat[] =      "AAAA"
txt[i] and pat[j] do NOT match and j is 0, we do i++.

i = 6, j = 0
txt[] = "AAAAABAAABA"
pat[] =       "AAAA"
txt[i] and pat[j] match, do i++ and j++

i = 7, j = 1
txt[] = "AAAAABAAABA"
pat[] =       "AAAA"
```

txt[i] and pat[j] match, do i++ and j++
We continue this way...

Knuth–Morris–Pratt algorithm { Worst case Time complexity to O(n)}

q	1	2	3	4	5	6	7
P[q]	a	b	c	d	a	b	d
π[q]	0	0	0	0	1	2	0

COMPUTE-PREFIX-FUNCTION(P)

Input: ***pattern P of length m***

output: table π[1,…,m]

```
1   m = P.length
2   let π[1,, m] be new array
3   π[1] = 0
4   k = 0
5   For q = 2 to m      // Scan the Pattern P[2,,m] from left to right
6      while k > 0 and P[k +1] ≠ P[q]
7            k = π[k]     // The prefix P_k is not the suffix of P_q
8      if P[k +1] ==  P[q]
9           k = k + 1   // The longest prefix P_k is  also a proper suffix of P_q      k = 2
10     π[q] = k
11  Return π
```

KMP-MATCHER(T,P)

Input: ***pattern P of length m and text T of length n***

Output: ***liist of all numbers s, such that P occurs with shift s in T***

```
1.  n = T.length
2.  m = P.length
3.  π = COMPUTE-PREFIX-FUNCTION(P)
4.  q = 0                  // number of characters matched
5.  for i = 1 to n     // scan the text T[1,,n] from left to right
6.     while q > 0 and P[q +1] ≠ T[ i ]  // when mismatch occurred
7.        q = π[ q]
8.     If  P[q +1] == T[ i ]   //next character matches
9.        q = q +1
10.   If q == m                  // is all of Pattern P[1,,m] matched?
11.       Print "Pattern occurs with shift" i - m
12.       q = π[q]                 // look for the next match
```

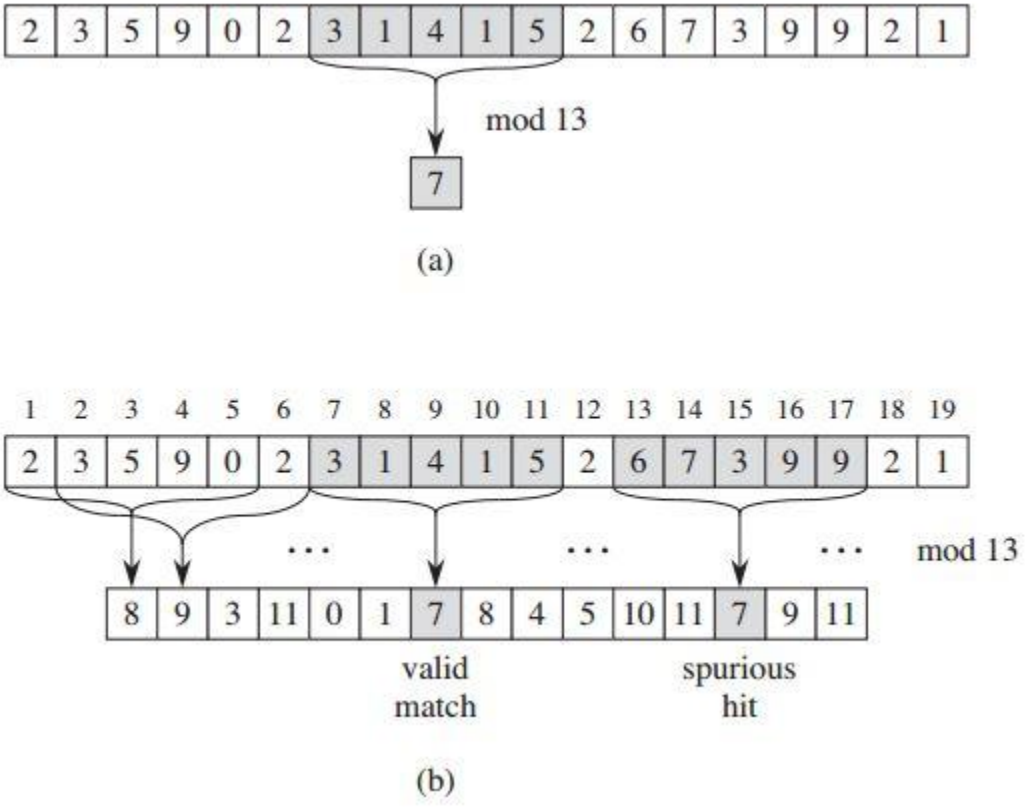
The Rabin-Karp algorithm

Rabin-Karp is another pattern searching algorithm to find the pattern in a more efficient way. It also checks the pattern by moving window one by one, but without checking all characters for all cases, it finds the hash value. When the hash value is matched, then only it tries to check each character. This procedure makes the algorithm more efficient.

The Rabin-Karp algorithm makes use of hash functions and the rolling hash technique. A hash function is essentially a function that maps one thing to a value. In particular, hashing can map data of arbitrary size to a value of fixed size.

Hashing: Hashing is a way to associate values using a hash function to map an input to an output. The purpose of a hash is to take a large piece of data and be able to be represented by a smaller form.

Example 1:



The Rabin-Karp algorithm. Each character is a decimal digit, and we compute values modulo 13. (a) A text string. A window of length 5 is shown shaded. The numerical value of the shaded number, computed modulo 13, yields the value 7. (b) The same text string with values computed modulo 13 for each possible position of a length-5 window. Assuming the pattern P = 31415, we look for windows whose value modulo 13 is 7, since 31415 ≅7 (mod 13). The algorithm finds two such windows, shown shaded in the figure. The first, beginning at text position 7, is indeed an occurrence of the pattern, while the second, beginning at text position 13, is a spurious hit.

Time complexity:

The time complexity is O(m+n)[Non Spurious hit], but for the worst case, it is O(mn) [Spurious hit]

Example 2:
Text=31415926535 Pattern=26
Hash function= 26 mod 13=0
Window of length 2 taken
31 mod 13=5 Slide Window
14 mod 13=1
41 mod 13=2
15 mod 13=2
59 mod 13=7
92 mod 13=1
26 mod 13=0[Non Spurious hit][valid match]
65 mod 13=0[Spurious hit][not valid match]
53 mod 13=1
35 mod 13=9

Example 3:

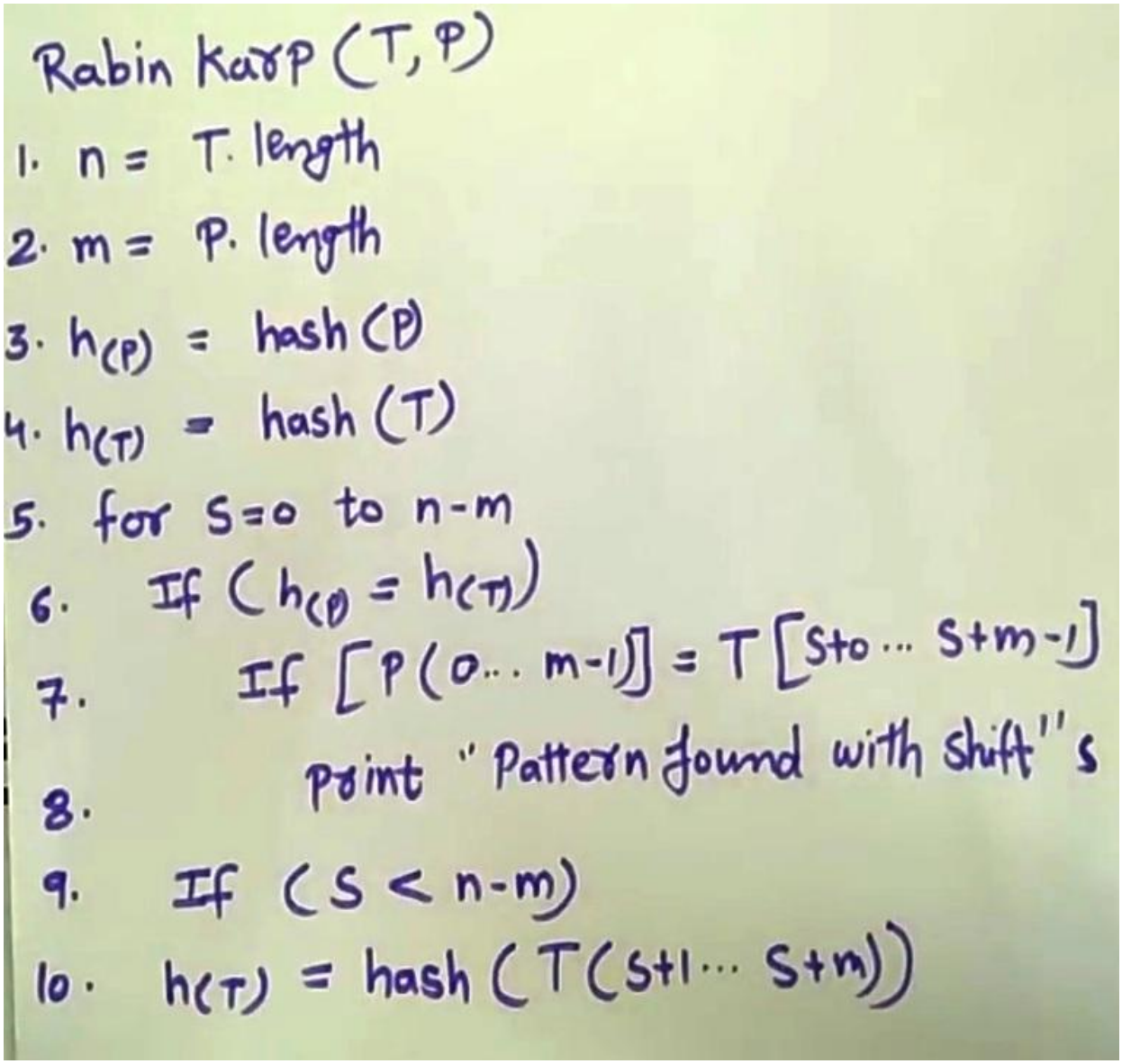
Text= cadba {Take ASCII Value}

Text= 99971009897

Pattern= db

Pattern= 10098

Hash Function=10098 mod 13=10



NP-Completeness | Set 1 (Introduction)

We have been writing about efficient algorithms to solve complex problems, like shortest path, Euler graph, minimum spanning tree, etc. Those were all success stories of algorithm designers. In this post, failure stories of computer science are discussed.

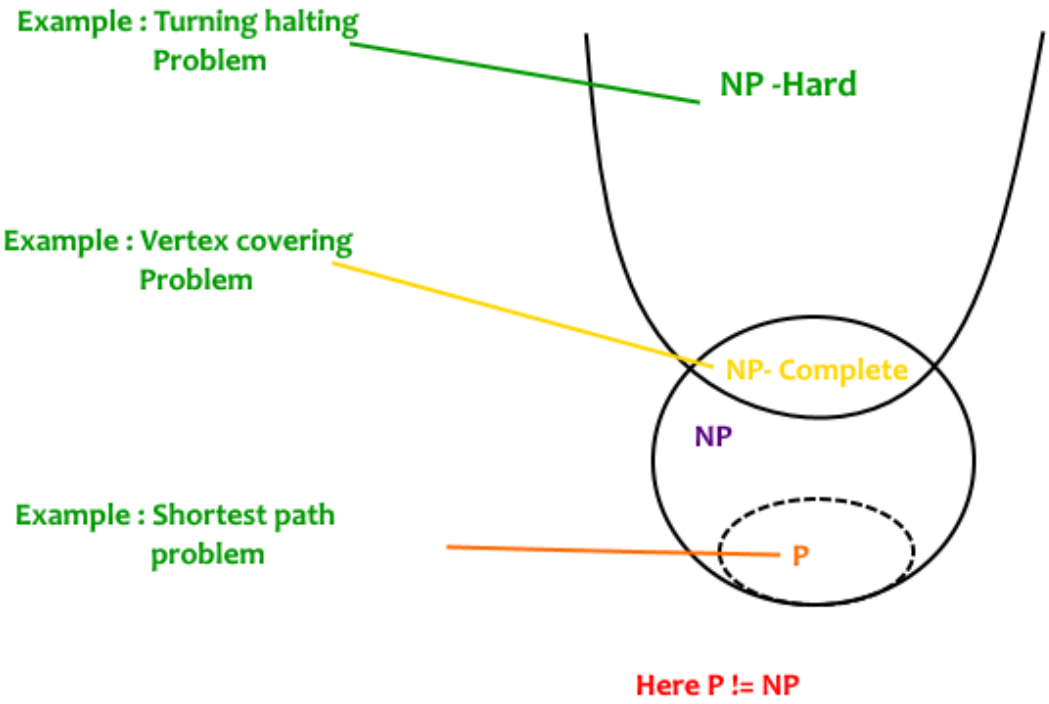
Can all computational problems be solved by a computer? There are computational problems that can not be solved by algorithms even with unlimited time. For example Turing Halting problem (Given a program and an input, whether the program will eventually halt when run with that input, or will run forever). Alan Turing proved that general algorithm to solve the halting problem for all possible program-input pairs cannot exist. A key part of the proof is, Turing machine was used as a mathematical definition of a computer and program (Source Halting Problem).
Status of NP Complete problems is another failure story, NP complete problems are problems whose status is unknown. No polynomial time algorithm has yet been discovered for any NP complete problem, nor has anybody yet been able to prove that no polynomial-time algorithm exist for any of them. The interesting part is, if any one of the NP complete problems can be solved in polynomial time, then all of them can be solved.

What are NP, P, NP-complete and NP-Hard problems?
P is set of problems that can be solved by a deterministic Turing machine in **P**olynomial time.

NP is set of decision problems that can be solved by a **N**on-deterministic Turing Machine in **P**olynomial time. P is subset of NP (any problem that can be solved by deterministic machine in polynomial time can also be solved by non-deterministic machine in polynomial time).
Informally, NP is set of decision problems which can be solved by a polynomial time via a "Lucky Algorithm", a magical algorithm that always makes a right guess among the given set of choices (Source Ref 1).

NP-complete problems are the hardest problems in NP set. A decision problem L is NP-complete if:
1) L is in NP (Any given solution for NP-complete problems can be verified quickly, but there is no efficient known solution).
2) Every problem in NP is reducible to L in polynomial time (Reduction is defined below).

A problem is NP-Hard if it follows property 2 mentioned above, doesn't need to follow property 1. Therefore, NP-Complete set is also a subset of NP-Hard set.



Decision vs Optimization Problems

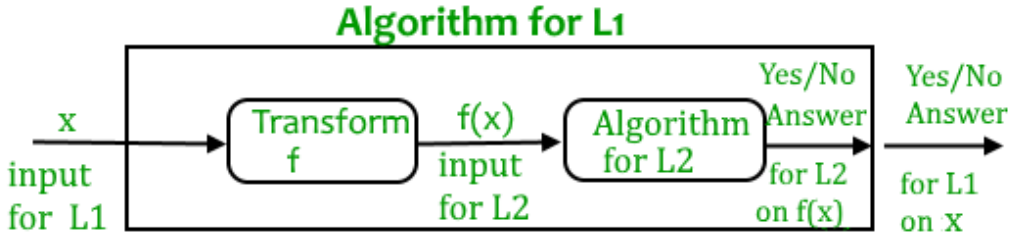
NP-completeness applies to the realm of decision problems. It was set up this way because it's easier to compare the difficulty of decision problems than that of optimization problems. In reality, though, being able to solve a decision problem in polynomial time will often permit us to solve the corresponding optimization problem in polynomial time (using a polynomial number of calls to the decision problem). So, discussing the difficulty of decision problems is often really equivalent to discussing the difficulty of optimization problems. (Source Ref 2).

For example, consider the vertex cover problem (Given a graph, find out the minimum sized vertex set that covers all edges). It is an optimization problem. Corresponding decision problem is, given undirected graph G and k, is there a vertex cover of size k?

What is Reduction?

Let L_1 and L_2 be two decision problems. Suppose algorithm A_2 solves L_2. That is, if y is an input for L_2 then algorithm A_2 will answer Yes or No depending upon whether y belongs to L_2 or not.

The idea is to find a transformation from L_1 to L_2 so that the algorithm A_2 can be part of an algorithm A_1 to solve L_1.



Learning reduction in general is very important. For example, if we have library functions to solve certain problem and if we can reduce a new problem to one of the solved problems, we save a lot of time. Consider the example of a problem where we have to find minimum product path in a given directed graph where product of path is multiplication of weights of edges along the path. If we have code for Dijkstra's algorithm to find shortest path, we can take log of all weights and use Dijkstra's algorithm to find the minimum product path rather than writing a fresh code for this new problem.

How to prove that a given problem is NP complete?

From the definition of NP-complete, it appears impossible to prove that a problem L is NP-Complete. By definition, it requires us to that show every problem in NP is polynomial time reducible to L. Fortunately, there is an alternate way to prove it. The idea is to take a known NP-Complete problem and reduce it to L. If polynomial time reduction is possible, we can prove that L is NP-Complete by transitivity of reduction (If a NP-Complete problem is reducible to L in polynomial time, then all problems are reducible to L in polynomial time).

What was the first problem proved as NP-Complete?

There must be some first NP-Complete problem proved by definition of NP-Complete problems. SAT (Boolean satisfiability problem) is the first NP-Complete problem proved by Cook (See CLRS book for proof).

It is always useful to know about NP-Completeness even for engineers. Suppose you are asked to write an efficient algorithm to solve an extremely important problem for your company. After a lot of thinking, you can only come up exponential time approach which is impractical. If you don't know about NP-Completeness, you can only say that I could not come with an efficient algorithm. If you know about NP-Completeness and prove that the problem as NP-complete, you can proudly say that the polynomial time solution is unlikely to exist. If there is a polynomial time solution possible, then that solution solves a big problem of computer science many scientists have been trying for years.

We will soon be discussing more NP-Complete problems and their proof for NP-Completeness.

References:

MIT Video Lecture on Computational Complexity
Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E.

P, NP-Complete, NP, and NP-Hard (in Short)

NP problems have their own significance in programming, but the discussion becomes quite hot when we deal with differences between NP, P, NP-Complete and NP-hard.

P- Polynomial time solving: Problems which can be solved in polynomial time, which take time like O(n), O(n^2), O(n^3). E.g.: finding maximum element in an array or to check whether a string is palindrome or not. So there are many problems which can be solved in polynomial time.

NP- Non deterministic Polynomial time solving: Problem which can't be solved in polynomial time like TSP (travelling salesman problem) or an easy example of this is subset sum problem: given a set of numbers, does there exist a subset whose sum is zero? But NP problems are checkable in polynomial time means that given a solution of a problem, we can check that whether the solution is correct or not in polynomial time.

Now we will discuss about NP-Complete and NP-hard.

Take two problems A and B both are NP problems.

Reducibility- If we can convert one instance of a problem A into problem B (NP problem) then it means that A is reducible to B.

NP-hard-- Now suppose we found that A is reducible to B, then it means that B is at least as hard as A.

NP-Complete -- The group of problems which are both in NP and NP-hard are known as NP-Complete problem.

Now suppose we have a NP-Complete problem R and it is reducible to Q then Q is at least as hard as R and since R is an NP-hard problem. Therefore Q will also be at least NP-hard, it may be NP-complete also.

Approximation Algorithm

Given an optimization problem P, an algorithm A is said to be an approximation algorithm for P, if for any given instance I, it returns an approximate solution that is a feasible solution.

Approximation algorithms

1. Guaranteed to run in polynomial time.
2. Guaranteed to get a solution which is close to the optimal solution.
3. Obstacle: need to prove a solution's value is close to optimum value, without even knowing what the optimum value is!

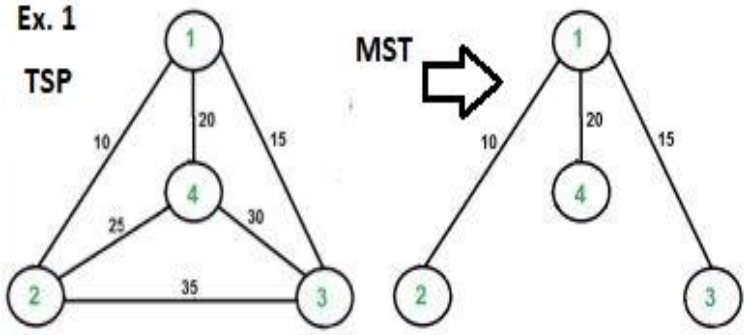
Types of approximation

P -An optimization problem, A -An approximation algorithm, I -An instance of P,

$A^*(I)$ -Optimal value for the instance I, A(I) -Value for the instance I generated by A

1. Absolute approximation: A is an absolute approximation algorithm if there exists a constant k such that, for every instance I of P, $A^*(I) - A(I) \leq k$. Example: Planar graph coloring.

2. Relative approximation: A is an relative approximation algorithm if there exists a constant k such that, for every instance I of P, $\max\{ A^*(I)/A(I) , A(I)/A^*(I) \} \leq k$. Example: Vertex cover.



The full walk of above tree would be 1-2-1-4-1-3-1. Apply Hamiltonian cycle so that the output is 1-2-4-3-1
Cost=10+25+30+15=80

In this case, the approximate algorithm produces the optimal tour, but it may not produce optimal tour in all cases.

A full walk is lists all vertices when they are first visited in preorder, it also list vertices when they are returned after a subtree is visited in preorder. The full walk of above tree would be 1-2-1-4-1-3-1.
Following are some important facts that prove the 2-approximateness.
1) The cost of best possible Travelling Salesman tour is never less than the cost of MST.
(The definition of MST says, it is a minimum cost tree that connects all vertices).
2) The total cost of full walk is at most twice the cost of MST (Every edge of MST is visited at-most twice)
3) The output of the above algorithm is less than the cost of full walk. In above algorithm, we print preorder walk as output. In preorder walk, two or more edges of full walk are replaced with a single edge. For example, 2-1 and 1-4 are replaced by 1 edge 2-4. So if the graph follows triangle inequality, then this is always true.
From the above three statements, we can conclude that the cost of output produced by the approximate algorithm is never more than twice the cost of best possible solution.

A Hamiltonian cycle, also called a Hamiltonian circuit, Hamilton cycle, or Hamilton circuit, is a graph cycle (i.e., closed loop) through a graph that visits each node exactly once. Determining whether such paths and cycles exist in graphs is the Hamiltonian path problem, which is NP-complete.

Traveling-salesman problem

Imagine we are a salesman, and we need to visit n cities. We want to start a *tour* at a city and visit every city *exactly one time*, and finish the tour at the city from where we start. There is a non-negative cost $c(i, j)$ to travel from city i to city j. The goal is to find a tour (which is a Hamiltonian cycle) of minimum cost. We assume every two cities are connected. Such problem is called *Traveling-salesman problem* (TSP).

We can model the cities as a complete graph of n vertices, where each vertex represents a city.

It can be shown that TSP is NPC.

If we assume the cost function c satisfies the *triangle inequality*, then we can use the following approximate algorithm.

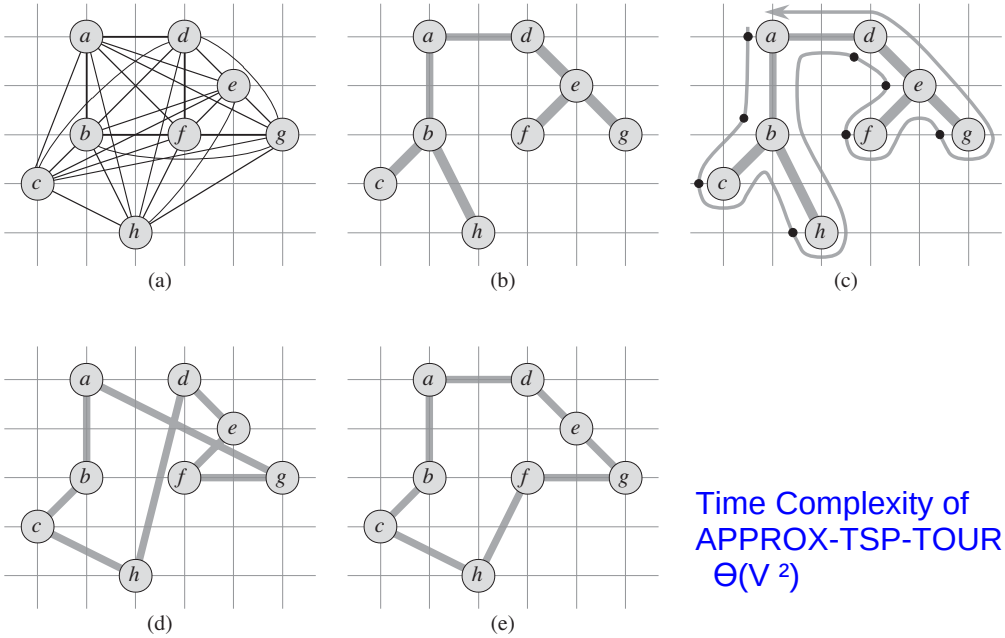
Triangle inequality. Let u, v, w be any three vertices, we have

$$c(u, w) \le c(u, v) + c(v, w)$$

One important observation to develop an approximate solution is if we remove an edge from H^*, the tour becomes a *spanning tree.*

```
Approx-TSP (G = (V, E)) {
1. compute a MST T of G ;
2. select any vertex r be the root of
   the tree ;
3. let L be the list of vertices
   visited in a preorder tree walk
   of T ;
4. return the hamiltonian cycle H that
   visits the vertices in the order L ;
   }
```

Example:2



Time Complexity of APPROX-TSP-TOUR
Θ(V ²)

Figure 35.2 The operation of APPROX-TSP-TOUR. **(a)** A complete undirected graph. Vertices lie on intersections of integer grid lines. For example, f is one unit to the right and two units up from h. The cost function between two points is the ordinary euclidean distance. **(b)** A minimum spanning tree T of the complete graph, as computed by MST-PRIM. Vertex a is the root vertex. Only edges in the minimum spanning tree are shown. The vertices happen to be labeled in such a way that they are added to the main tree by MST-PRIM in alphabetical order. **(c)** A walk of T, starting at a. A full walk of the tree visits the vertices in the order $a, b, c, b, h, b, a, d, e, f, e, g, e, d, a$. A preorder walk of T lists a vertex just when it is first encountered, as indicated by the dot next to each vertex, yielding the ordering a, b, c, h, d, e, f, g. **(d)** A tour obtained by visiting the vertices in the order given by the preorder walk, which is the tour H returned by APPROX-TSP-TOUR. Its total cost is approximately 19.074. **(e)** An optimal tour H^* for the original complete graph. Its total cost is approximately 14.715.

Recall from Section 12.1 that a preorder tree walk recursively visits every vertex in the tree, listing a vertex when it is first encountered, before visiting any of its children.

Figure 35.2 illustrates the operation of APPROX-TSP-TOUR. Part (a) of the figure shows a complete undirected graph, and part (b) shows the minimum spanning tree T grown from root vertex a by MST-PRIM. Part (c) shows how a preorder walk of T visits the vertices, and part (d) displays the corresponding tour, which is the tour returned by APPROX-TSP-TOUR. Part (e) displays an optimal tour, which is about 23% shorter.

Vertex Cover Problem

vertex cover of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if (u, v) is an edge of G, then either $u \in V'$ or $v \in V'$ (or both). The size of a vertex cover is the number of vertices in it.

The ***vertex-cover problem*** is to find a vertex cover of minimum size in a given undirected graph. We call such a vertex cover an ***optimal vertex cover***. This problem is the optimization version of an NP-complete decision problem.

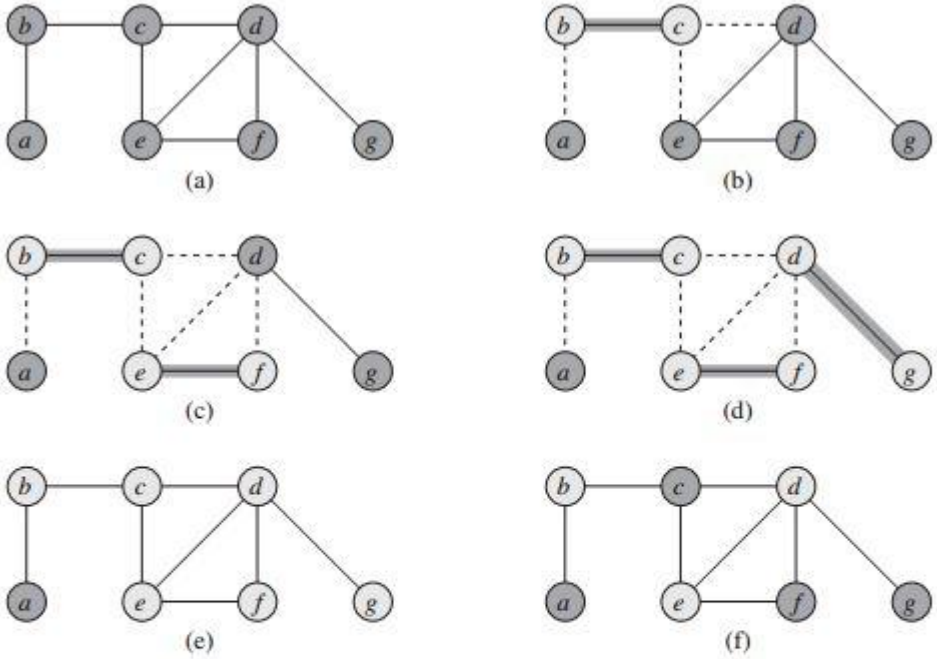


Figure 35.1 The operation of APPROX-VERTEX-COVER. **(a)** The input graph G, which has 7 vertices and 8 edges. **(b)** The edge (b, c), shown heavy, is the first edge chosen by APPROX-VERTEX-COVER. Vertices b and c, shown lightly shaded, are added to the set C containing the vertex cover being created. Edges (a, b), (c, e), and (c, d), shown dashed, are removed since they are now covered by some vertex in C. **(c)** Edge (e, f) is chosen; vertices e and f are added to C. **(d)** Edge (d, g) is chosen; vertices d and g are added to C. **(e)** The set C, which is the vertex cover produced by APPROX-VERTEX-COVER, contains the six vertices b, c, d, e, f, g. **(f)** The optimal vertex cover for this problem contains only three vertices: b, d, and e.

```
APPROX-VERTEX-COVER(G)
1  C = ∅
2  E' = G.E
3  while E' ≠ ∅
4      let (u, v) be an arbitrary edge of E'
5      C = C ∪ {u, v}
6      remove from E' every edge incident on either u or v
7  return C
```

endpoints u and v to C, and deletes all edges in E' that are covered by either u or v. Finally, line 7 returns the vertex cover C. The running time of this algorithm is $O(V + E)$, using adjacency lists to represent E'.