Signals

Design And Analysis of Algorithm

Page No.
Date

Running Time ← RT = n^2.
Speed = 10^{10}
Steps = $2n^2$
$n = 10^7$

A

Time Taken = $\frac{\text{No. of steps}}{\text{Speed}}$

(A) $= \frac{2n^2}{(10)^{10}}$

$= \frac{2(10^7)^2}{(10)^{10}}$

$= 20,000s$

B

RT = $n\log n$
Speed = 10^7
Steps = $50n\log n$

Time Taken = $\frac{50n\log n}{10^7}$

(B) $= \frac{50}{10^7} \frac{10^7 \log n}{10^7}$

∴ B is efficient bcz of its growth rate is higher.

Computational Tractability

Analyzing algorithm ⇒ Means to know how their resource requirements – the amt of time & space they use – will scale with increasing i/p size.

(*) Computational Tractability. (Easily done)

[Means that we want algorithms that run quickly, But should be efficient to use for other resources aswell].

the amt of space used by an algorithm is an issue & we will use diff methods to resolve.

Also means that a specific problem Statement at hand can be solved with an algorithm.

(*) Initial attempts at defining Efficiency.

{An algorithm is efficient if, when implemented, it runs quickly on real input instances}

The concrete def^n of efficiency is

① Platform - independent

② Instance - dependent

(Ans) worst case running time and Brute-force search.

① worst case running time : the most / maximum time to complete a task on the basis of the given input

② Average case is where it looks for more about the input provided rather than the search analysis.

⑤ Best case it is rarely found where the analysis and input are perfectly present.

what is running time bound?
The running time bound refers to a function for that running time.

Running time analysis - It estimates and anticipate the increase in running time (or execution time) of an algorithm as its input size (usually denoted n) increases.

$O(f(n))$ — upper bound
$O(\Omega)$ — lower bound
$O(\Theta)$ — highest bound
} $f(N)$

∴ The time complexity is been defined by $O(n)$.

$f(N)$ is a function and N is the input size.

Models of Computation.

① Turing Machine — linear

② Word RAM — this computes the basic arithmetic operations.

③ Brute-force search.

An algorithm is efficient if it achieves qualitatively better worst-case performance.

① Even in the worst-case running time if it solves a problem faster, than it just keeps trying every possible solution.

② Brute force search checks all the possible solution one by one

③ A better algorithm finds the answer faster with smart techniques.

Polynomial time as a Definition of Efficiency.

mathematical series predicating a func^n defining func^n w.r.t to time

Page No.
Date 11 / 03 / 25

Asymptotic order of g(N) → a simple func^n to compare to the count or the growth

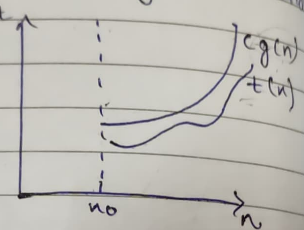
order of magnitude

To analyze the behaviour of an algorithm in worst case behaviour scenarios

It calculated on func^n (f(n)) on the time & input size (N)

Mathematic way to represent the time complexity.

Big-Oh asymptotic notation



$t_n \in O\,g(n)$

g(n) upper bound for t(n) if beyond some point

Asymptotic Notation (mostly followed)

1. O → Big-Oh
2. Ω → Big-Omega
3. Θ → Big-Theta
4. o → Small-Oh
5. ω → small-omega

(1–3: commonly used)

N → input size

f(n) → growth of input size (N) w.r.t time (T)

T(n) → Time complexity, any algorithm worst case running time w.r.t (N) input size.

C(N) → basic operation count of (any algorithm)

Asymptotic Order of Growth. Thus,

- Tight bound Proof.

$f(n) <= c\,g(n)$ for all $n >= n_0$ ⇒ $f(n) = O(g(n))$ — upper bound

$f(n) >= \frac{1}{2} c\,g(n)$ for all $n >= n_0$ ⇒ $f(n) = \Omega(g(n))$ — lower bound

let f & g be two fun:- such that $\lim_{n \to \infty} \frac{f(n)}{g(n)}$ exists & is equal to some no $c > 0$, then $f(n) = \Theta(g(n))$ — tight bound

If two fun:- $f(n)$ & $g(n)$ grow at the same rate, as n becomes very large then we say that $f(n) = \Theta(g(n))$ — meaning that $f(n)$ and $g(n)$ converges with a +ve constant c.

By the def:- of any limits in calculus for any small +ve number there exists a large no such that for all

b/w $\frac{1}{2} c$ and $2c$ — lower bound, upper bound

If we have 3 fun^n
h(n) g(n) F(n)
The we say that the fun^n
if is asmptotic by a f(n) upper bounder
fun^n g(n)
And g(n) assumes upper bounded
by H(n). Then

a) If f(n) = O(g(n))
b) If g(n) = O(h(n))
then, we conclude that
[f(n) = O(h(n))]

f(n) ≤ c g(n) —— (1)
& g(n) ≤ c dh(n) —— (2)

(1) The exists a^r constants
n0 and c^* such that for
all n ≥ n

(2) There exists

Such that for all
n > n0

If f(n) = O(h(n)) — a
If g(n) = O(h(n)) — b
then,
f(n)+g(n) = O(h(n))

Consider Eq^n a fn = O(h(n))
meaning there exist
cons C1 & n0 for all
n ≥ n0 can rewrite as
f(n) ≤ C1 h(n) —— (1)

Sully given Eq^n b g(n) = O(h(n))

g(n) ≤ C2 h(n) —— (2)

Summing the two fun^n

f(n) + g(n) ≤ C1(h(n)) + C2(h(n))
" ≤ (C1 + C2)(h(n))
C
≤ C(h(n))

f1(n) = 3n^2
f2(n) = 5n^2
f3(n) = 2n^2

f1(n) + f2(n) + f3(n)
3n^2 + 5n^2 + 2n^2
10n^2
Bounded by the same order of growth

4th property
If 2 fun^n are
also taking no -ve
value ST g(n) = O(f(n))
then f(n) + g(n) = O(f(n))

Page No.
Date

4th property
Suppose that f & g are 2 fu=n Polynominals:
such that $g(n) = o(f(n))$,
then the sum $f(n)+g(n)$ is asymptotically tight-bounded by $f(n)$.

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

the fastest growing fu=n will be dominated by sum of fu=n

① The dominant term (largest exponent, n^d) dictates the growth rate
② Lower-degree terms like n^{d-1}, n^{d-2} etc, growth much slower than n^d, & become insignific in Big O notation
③ Since Big O ignores constants factors, we only consider the high-d terms growth
④ The sum of multiple $O(n^d)$ is still an $O(n^d)$

Asymptotic order of growth — asymptotic bounds for some common fu=n

- Properties
① Polynomials - 2nd
② Logarithms - 3rd slowest
③ Exponentials - higher rate of growth (1st)

variables that carry variable with power
polynomial fun=n → $f(x) = 3x^2-5$
$g(x) = -7x^3+2x$
$h(x) = 3x^4+7x^3-12x^2$
$f(n) = a_0 + a_1 n + \dots + a_d n^d$

Logarithmic fun=n → $f(x) = b^x$ formula

Logarithmic
Even the smallest polynomial fu=n grows faster than a logarithmic fu=n as 'n' becomes large

Page No.
Date

① $\log_b(n)$
$b = 2$, $n = 1000$
$\log_2 1000 = 9.965$

$\log_2 (0.01) = -6.643$

② $\log_b n$
$n = 2^n$ → $\log_2(2^{0.001}) = 0.01$

$\log_b n$ → $\log_2 2^{1000} = 1000$

Consider a logarithmic fun=n $\log_2 n$ & $n^{0.01}$ - polyn f=n highest
Compute & compare these 2 fun=n for increasing values of n
list ur obs and PT the log=n slower order of growth compared to polynomial.

$\log_2 10 = 3.3219$

$\log_2 1.0232 = 0.0330$
$n^{0.01} = 1.0232$

$\log_2(100) = 6.643$

Expentiational:
Every exponential fun=n out grows the not matter how larger the polynomial exponent 'D' is for larger values of 'n'

Exponential Growth fu=n > Polynomial Growth fun=n > Logarithmic growth fun=n.
or exponential running time > P^w running time > log=n running time.

Survey of Common running times

1) $O(1)$	Constant	Accessing an array element	Each women has a list of ranking of men
2) $O(\log n)$	Logarithmic	Binary Search	
3) $O(n)$	Linear	Finding max	A set of n pairs in an array $[m, i, w, j]$ with each pair is paired exactly with women
4) $O(n \log n)$	Near Linear	Merge sort, Quick sort,	
5) $O(n^2)$	Quadratic	Bubble sort, Floyd Warshall	Gale – Shapley algo :-
6) $O(n^3)$	Cubic	Matrix multi = n^n	Stable matching
7) $O(2^n)$	Exponential	Recursive Fibonacci	A set of n pairs (m_i, w_j) where each pair is paired exactly with one women
8) $O(n!)$	Factorial	Travelling Salesman Problem	$M = \{(m_1, w_2)(m_2, w_3)(m_3, w_1)\}$

Stable Matching Problem

Consider two sets $X = \{m_1, m_2, \ldots m_n\}$ then $Y = \{w_1, w_2, \ldots w_n\}$ if →

Each man has a ranking list of women.

Matching is stable if

① There is no blocking pairing

② An instance of a problem in 2 ways

② 2 sets of preferences are given

③ Ranking matrix

By a ranking matrix is where each man is ordered set as per their preference and same for women

Rows = men

Column = women

Explain an instance & Numbers

	women A	B	C
men Bob	(2, 3)	(1, 2)	(3, 3)
Jim	(3, 1)	(1, 3)	(2, 1)
Tom	(3, 2)	(2, 1)	(1, 2)

men preference, women preference

Men's preference list

	1st	2nd	3rd
Bob	A	B	C
Jim	C	B	A
Tom	B	A	C

Blocking pair : A pair (m, w) is a blocking pair if → ① they are not matched in the current pairing n ② Man m prefers w over than given its current more

Step 0: At the beginning all the men & women are unpaired

Step 1: Proposal → b): Response

Preproposal → select any 3 men

response → ① If the women w is free ② Accept offer accepting she is Engaged ② Not free & Engaged If w is not free ① She compares m with her current m, If she prefers free m with current m → Else she rejects.

Unit - II

Page No.
Date 26 | 3 | 25

Divide & Conquer Tekique factor by which the problem size is reduced
→ It mainly focuses on reducing the running time in the each recursion to a lower polynomial
It is better than brute force search

$O(n^d)$ = Cost of dividing at each level

* Recurrence relation
Ways to solve
① Recursion tree
② Substitution tree
③ Masters theorem

Case 1: Divide dominates
If d is greater than $\log_b a$ then $T(n) = O(n^d)$

Case 2: Equal both
If d is Equal to $\log_b a$ then
$T(n) = O(n^d \log n)$

Case 3: Conquer dominates
If d is less than $\log_b a$
$T(n) = O(n^{\log_b a})$

Masters theorem for recurrence relation:
It provides a way to solve RR that arise in divide & conquer relation then

$T(n) = a\,T(n/b) + O(n^d)$

a = no of sub problems into which the original problem is divided
b = Reduction factor

Recurrence relation.
Analyzing the running time of a divide & conquer algorithm involves solving a recurrence relation

Base case: It is the time base case: It is a condition where the algorithm will stop its execution and gives us end result.
-ning condition

Solve the following recurrence relations using masters thm
① $T(n) = 4T(n/2) + n$
② $T(n) = 8T(n/3) + n^2$

Quick sort algorithm
① Pivot Selection
② Partitioning
③ Recursion.

Apply merge sort algorithm to the following elements
{5, 2, 4, 6, 1, 3, 2, 6}
{20, 10, 5, 15, 25, 30, 50, 35}

quick sort: ① Pivot - Element around which the array is divided
② Index pointer - Keeps tracks of the boundary where elements smaller than the pivot should be placed
③ Walker pointer - Scans the array to identify elements smaller than or Equal to the pivot.

Unit - 11

Graphs :

* A graph is a way of encoding pair wise relationship among set of objects.

* A graph will have a collection 'V' of nodes or vertices and a collection of E of edges

* Collection of edges "joins" two nodes of the graph.

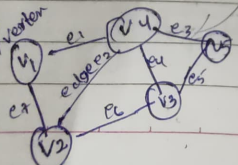
* An edge of a graph ($e \in E$) is represented as a two element subset of V: $e = \{u, v\}$ for some $u, v \in V$ where u & v are the ends of e.

* Symmetric relationships in graph indicated by an undirected graph where the edges in the graph indicates a symmetric relationship b/w their ends.

Considering 2 vertices two v b/w

e_3 = E node/vertex

v_1 = V



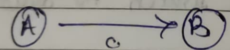
A symmetric relationships in graphs Edges indicating a one-way relationship and is represented by a Directed graph G that consists of a set of nodes V and set of directed edges E.

Rules of (u,v)

① The position of u & v are interchangable.

② u is called as the tail of the edge & v is called the head of the edge

③ Also it means, edge e leaves node u & enters node v.

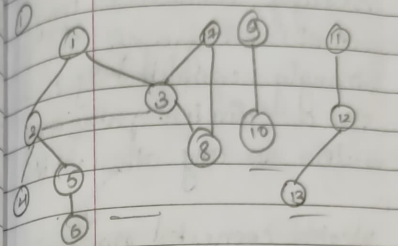


Paths & Connectivity

Fund=ml operation in a graph - Traversing a sequence of nodes connected by edges

Path (P) in a graph is an undirected graph G = (V, E) is a sequence P of nodes $v_1, v_2, v_3 \ldots v_{k-1}, v_k$

Cycle - is a path in which the sequence of path [v_1 to v_k] in which the it sequence back from where it starts



It is complex undirected graph.

For it to be

① Multiple paths, ② Cycle

③ Disconnected components

will be present

start node → ① → Queue[1]

visited{1} bcz one node should be visited only once.

• visit 1 → neighbours 2, 3 → Queue [2, 3] visited {1, 2, 3}

• visit 2 → not ① bcz already visited → 1, 3, 4, 5 → Queue [3, 4, 5] visited {1, 2, 3, 4, 5}

• visit 3 → 1, 2, 7, 8 → Queue [4, 7, 8] visited {1, 2, 3, 4, 5, 7, 8}

• visit 4 → 2 → Already visited Queue [2] visited {1, 2, 3, 4, 5, 7, 8}

• visit 5 → 2, 6 → Queue {2, 6} visited {1, 2, 3, 4, 5, 6, 7, 8}

• visit 7 → 3, 8 → Queue {3, 8} visited {1, 2, 3, 4, 5, 6, 7, 8}

• visit 8 → Queue { } visited {1, 2, 3, 4, 5, 7, 8, 6}

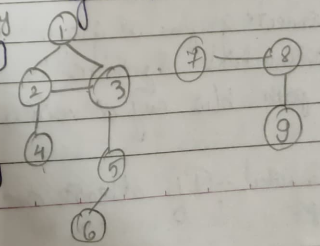
• visited 9 → 10 → Queue {10} visited {9}

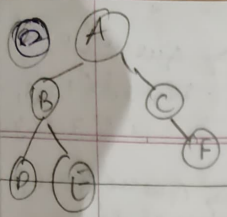
• visit 10 → 9 → Queue {9} visited {9, 10}

visit 11 → 12 → Queue {12} visited {11}

{11, 12, 13}

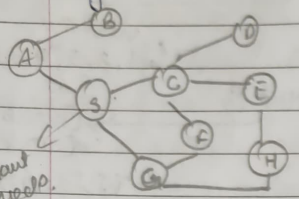
② Implement a BFS and back traversal of BFS.





(5) Implement the BFS

(1) Print out BFS order
(2) Construct the BFS spanning tree



DFS :- Stack (LIFO)
DFS is a go deep into the path until you can't go any further, then backtrack and take the alternate path until found.

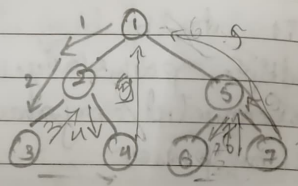
* Connected graph. A graph is connected if there is a path b/w any 2 vertices

DFS connected → If atleast one pair of vertices and one path b/w them.

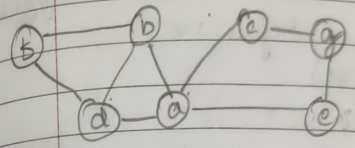
Strongly connected graph
A directed graph is strongly connected if there is a directed path any vertex to any other vertex

Weakly connected graph.-

DFS



(1) Consider the graph given in the fig=2
(1) Starting at vertex a and resolving the ties by vertex alphabetical order, traverse the graph by DFS rules
(1) Construct a DFS spanning tree
(2) Show the traces of traversal stack for DFS and specify the order in which vertices were pushed onto & poped of the stack
(3) Identify back edges if any
(4) "All DFS forest will have same number of tree edges & back edges"



adjacency
a : b, c, d, e
b : a, d, f
c : a, g
d : a, b, f
e : a, g
f : b, d
g : c, e

→ Start from vertex a.
visited { }
Stack [ ]
DFS trees edges = [ ]
DFS Order of traversal = [ ]

→ Iteration 1
* Stack : [a]
* visited { a }
DFS Order [a]
→ Neighbours of a. [in alphabetical order]
: b, c, d, e.
→ push to Stack : e, d, c, b

→ Iteration 2.
Stack : [e d c b]
pop = b [visited]
visited { a b }
neighbours of b : [d, f]
push [f, e, d, c]

→ Iteration 3.
Stack : [f, e, d, c]
pop = c [visited]
visited { a, b, c }
neighbours of c [g]
push [g, f, e, d]

Iteration 4
Stack : [g, f, e, d]
pop = d [visited]
visited { a, b, c, d }
neighbours of d [f]

Dfs Order [a, b]

Finally Dfs spanning tree edges
(a, b) (b, d) (d, f) (a, c)
(c, g) (g, e)

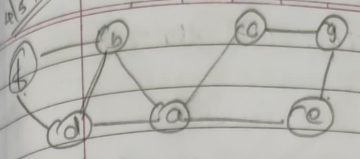
Stack Trace (push order & pop order)
Push order : a → b → c → d → e
→ f → d → f → g → e
Pop order : a → b → d → f → f (skip)
→ c → g → e → e (skip) → d (skip)

Back edges connect to already visited ancestors but one condition but Immidi-ate pair of parents should not be visited
Identified back edges from the traversal
* (d, a) → Back (a is already visited & a is not its parent)
* (b, f) → Back edge (f is visited via d)
* (e, a) → Back edge

Directed acyclic graph:
A linear ordering of vertices such that for every directed edge (u → v) 'u' appears before 'v'.

For a topological sort algorithm
① Directed graph represented as an adjacency list
② Starting node
③ A visited set to keep track of visited

16/5/24



① Start a
* neigh = [b, c, d, e] → pick b
② From b
* neigh = [a, d, f] → pick d
③ From d
* neigh = [a, b, f] → pick f
④ From f
* neigh = [b, d] → Backtrack.
⑤ Backtrack to d → no more unvisited neighbours
⑥ Backtrack to b → next neigh is f → Backtrack to a
⑦ From a,
* N = [c]
N = [a, g] → pick g
⑧ From g
* N = [a, e] → pick e
⑨ From e
* N = [a, g] → Backtrack g, c then a.

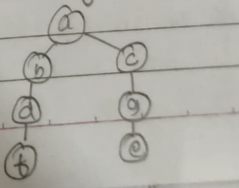
a → b → d → f → c → g → e

DFS spanning tree
a → b b → d e → g
a → c b → f a → e
a → d c → g

* Dfs spanning tree concept
Include only the edges we use to reach a node for the first time
Ignore the edges used during backtracking or cycles

Start at a.
visit b → edge a → b
from b
visit d → edge b → d
from d
visit f → edge d → f
Backtrack to a
visit c → edge a → c
from c.
visit g → edge c → g
from g
visit e → edge g → e

DFS Spanning tree edges
a → b c → g
a → c d → f
b → d g → e



Stack Tracing

Push order (visit)

a → b → d → f → c → g → e

Pop order:

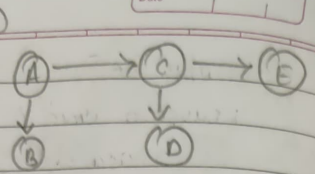
f → d → b → c → g → c → a

Back edge

In DFS a back edge is a edge that goes from a node to its ancestor in a DFS tree. It forms a cycle in a graph.

How to find back edge

1. DO DFS
2. Keep track of DFS tree
3. For the every edge in the original graph check
   1. If it's not a tree edge
   2. It connects the current node to the an ancestor to an already visited



There are two ways to do the topological sorting

1. Kahn's algorithms (BFS + in-degree counting)
2. In-degree is counting the incoming nodes

Nodes	In-degree
A	0
B	1 [from A]
C	1 [from A]
D	1 [from C]
E	1 [from C]

3. Add all the nodes with in-degree value zero to a queue.

Queue = [A]

While queue is not empty

1. remove the node from queue
2. Add it to the result
3. Reduce the in-degree of the neighs

Queue = [A]

Pop A → results [A]

B → In-degree becomes 0 → add it to queue

C → In-degree = 0 → add to queue

Q = [B, C]

Pop B → result [A, B]

No outgoing edges from B

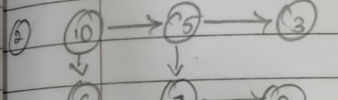
Queue [C]

pop C → result [A B C]

D → In-degree = 0 → add to Q

E → " " = 0 → "

Q = [D, E]



Performing DFS

Algorithm.

1. Perform DFS from each unvisited node
2. After visiting all decendance of a node, add the node to a stack

3. At the end reverse the Stack to get the topological order

Visit A

→ Mark A as visited

→ Explore A's B, C neighbors

visit B (from A)

→ Mark B as visited

→ Explore B's neighbours D (No neighbours) stack

visit C from

→ Mark C as visited

→ Explore C's neighbors D, E

visit D from C

→ Mark D as visited

→ D has no neighbours add D to the stack

Stack = [B, D]

visit E from C

→ Mark E as visited

→ E has no neigh

→ add it to Stack

Stack = [B, D, E]

Kanns algo = source removal

Back to C.

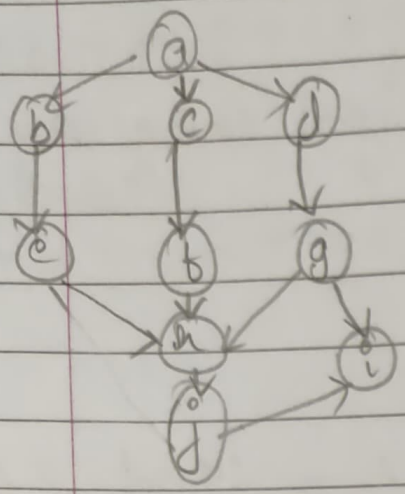
→ all weights visited

Heap Property

① Shape property

② parental property

Heap order property



Unit - 3

A binary tree is a

Heap data structure has a shape that each node has such two children.

Each node in the tree stores a key

A key is a value to determine the order of the heap

Each node in a tree has holds exactly only three