Analysis of algorithms

The theoretical study of computer program performance and resource usage

↳ how to make things fast

What's more important than performance?
- scalability, security, modularity, correctness, user friendly etc.

Why study algorithms and performance?
→ often performance measures the line b/w the feasible and non-feasible
→ Algorithms are the cutting edge of technologies
→ Algorithms are not depended on language

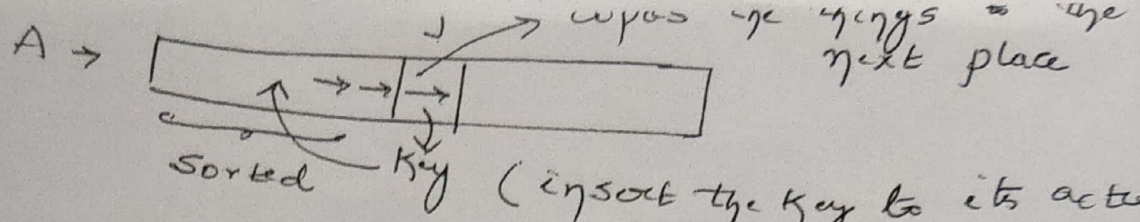
Problem : Sorting

Input : Sequence $\langle a_1, a_2, a_3 \ldots a_n \rangle$ of numbers
Output : permutation $\langle a_1', a_2', \ldots a_n' \rangle$ such that
$a_1' \leq a_2' \leq a_3' \ldots \leq a_n'$
(monotonically increasing)

Insertion sort

```
Insertion-sort(A, n) // sorts A[1..n]
    for j ← 2 to n
        do key ← A[j]
           i ← j-1
           while i>0 and A[i] > key
               do A[i+1] ← A[i]
                  i ← i-1
           A[i+1] ← key
```

Scanned by CamScanner



(insert the key to its actual position so its called insertion sort)

Ex : 8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

2 3 4 8 9 6

2 3 4 6 8 9

Running time

→ Depends on input (eg. already sorted)

→ Depends on input size (6 elm Vs 6×10^9)

↓ how to handle

— Parameterize in input size

→ We want upper bounds.

— guarantee to user

Different kinds of analysis

Worst case (usually)

$T(n)$ = max time on any input of size n

Average case (sometimes)

$T(n)$ = expected time over all input of size n

(proportability of assuming the input)

(Need assumption of stat. distribution)

Best Case (bogus)

Slow algorithm ...

Scanned by CamScanner

Is insertion sort fast?

- Moderately so, for small n
- Not at all for large n

Mergesort

$T(n)$	Mergesort $A[1..n]$
$\Theta(1)$	1. If $n=1$, done
$2T(n/2)$	2. Recursively sort $A[1..\lceil n/2 \rceil]$ and $A[\lceil n/2 \rceil + 1 \; n]$
$\Theta(n)$	3. Merge 2 sorted lists

Key subroutine : Merge

20	~~12~~
13	~~11~~
~~7~~	~~9~~
~~2~~	~~1~~

output array

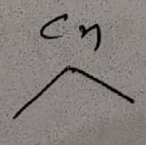
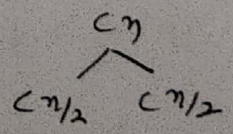
1 2 7 9 11 12 13 20

Time = $\Theta(n)$ on n total elements
linear time

Recurrence for merge sort

$$T(n) = \begin{cases} \Theta(1) & \text{if } n=1 \text{ (usually omit)} \\ 2T(n/2) + \Theta(n) & \text{if } n>1 \end{cases}$$

Recursive tree = $T(n) = 2T(n/2) + cn$, constant $c>0$

$T(n) =$  $=$  $= \ldots$

Scanned by CamScanner

What is insertion sort's worst case time?

→ Depends on computer
- on a super computer, desktop etc.
- relative speed (on same machine)
- absolute speed (on diff. machines)

'BIG IDEA' — Asymptotic analysis.

Basic things of Asymptotic analysis

(1) Ignore machine dependent constants

(2) look at growth of $T(n)$ as $n \to \infty$ (instead of actual time)

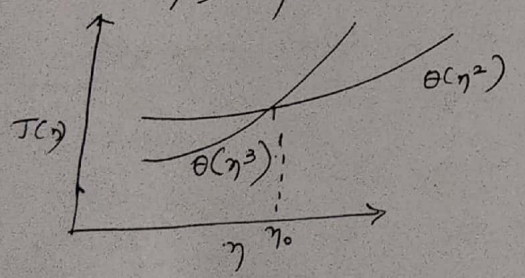
Asymptotic notation

Θ-notation (theta): From formula drop lower order terms, ignore leading const

Eg:- $3n^3 + 90n^2 - 5n + 6046 = \Theta(n^3)$

As $n \to \infty$, $\Theta(n^2)$ algorithm always beat a $\Theta(n^3)$ algo.

even $\Theta(n^2)$ is executed on a slower comput & $\Theta(n^3)$ on a faster computer

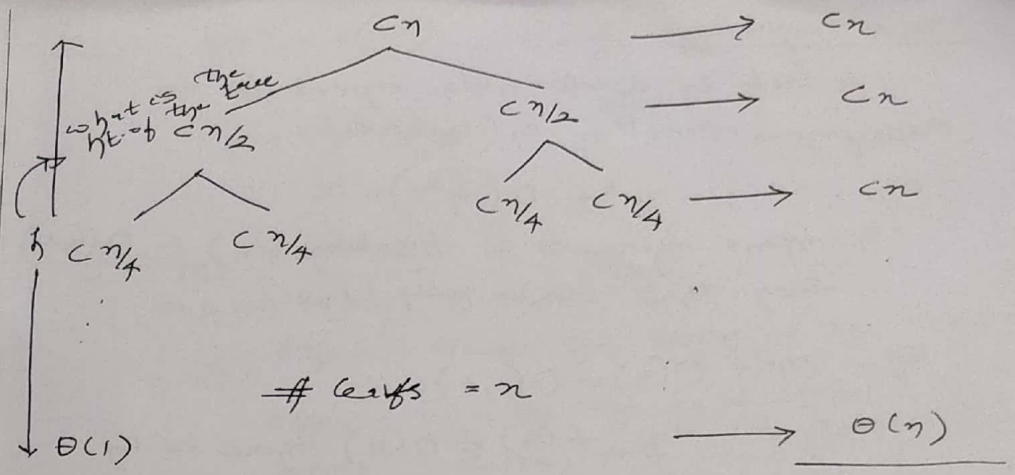


Insertion sort analysis

Worst case: input reverse sorted

$$T(n) = \sum_{j=2}^{n} \Theta(j) = \Theta(n^2)$$ (arithmetic series)

↳ asymptotically j times.



leaves $= n$

$h = \log n$

$\Theta(n)$

Total $= (cn)\log n + \Theta(n)$

$= \Theta(n \log n)$

↓ is asymptotically fast compared to $\Theta(n^2)$

∴ Mergesort is faster algo compared to insertion sort

O notation (big oh)

$f(n) = O(g(n))$ means there are consts $c > 0$, $n_0 > 0$ such that

$0 \le f(n) \le c \cdot g(n)$ for all $n \ge n_0$

Eg:- $2n^2 = O(n^3)$

Set definition

$O(g(n)) = \{ f(n) :$ there are consts $c > 0$, $n_0 > 0$ such that $0 \le f(n) \le c \cdot g(n)$ for all $n \ge n_0 \}$

means $2n^2 \in O(n^3)$ ∵ the set of n^3

Macro Convention

A set in a formula represents an anonymous function in that set.

Eg:- $f(n) = n^3 + O(n^2)$

this means there is a function $h(n) \in O(n^2)$ such that $f(n) = n^3 + h(n)$

Eg:- $n^2 + O(n) = O(n^2)$

means for any $f(n) \in O(n)$ there is an $h(n) \in O(n^2)$ such that $n^2 + f(n) = h(n)$

Ω notation (big omega)

$\Omega(g(n)) = \{ f(n) :$ there exist consts $c > 0$, $n_0 > 0$ such that

$$0 \le g(n) \le f(n)$$

for all $n \geq n_0 \}$

Eg:- $\sqrt{n} = \Omega(\lg n)$

Analogies

O	Ω	Θ
$\le$	$\geq$	$=$

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

o (little oh) and ω (little omega) notations

$<$ $\qquad$ $>$

must hold for all constants $c > 0$

Eg:- $2n^2 = o(n^3)$ $\qquad n_0 = \frac{2}{c}$

$$\frac{1}{2} n^2 = \Theta(n^2)$$
$$\neq o(n^2)$$

Solving recurrences

(1) Substitution method

step 1 : Guess the form of the solution

step 2 : Verify by induction

step 3 : Solve for consts.

Eg:-

$$T(n) = 4T(n/2) + n$$
$$[T(1) = \Theta(1)]$$

$\Theta(n^2)$

→ Guess $T(n) = O(n^3)$

→ Assume $T(k) \le c \cdot k^3$ for $k < n$

$$T(n) = 4(T(n/2)) + n$$
$$\le 4 \cdot c\left(\frac{n}{2}\right)^3 + n$$
$$= \frac{1}{2} cn^3 + n$$
$$= \underbrace{cn^3}_{\text{desired}} - \underbrace{\left(\frac{1}{2}cn^3 - n\right)}_{\text{residual}}$$
$$\le cn^3 \text{ if } \frac{1}{2}cn^3 - n \geq 0$$

if $c \geq 1$ its true

Base : $T(1) = \Theta(1) \le c$ is true if c is choosen sufficiently large.

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

Assume $T(k) = c \cdot k^2$ for $k \le n$

$$T(n) = 4 \cdot T\left(\frac{n}{2}\right) + n$$
$$\le 4c \cdot \left(\frac{n}{2}\right)^2 + n$$
$$= cn^2 + n$$

Scanned by CamScanner

$= cn^2 - (-n)$ $\quad = O(n^2)$ (but useless)

↳ is true for $n = 0$

$\nleq cn^2$

– Assume $T(k) \leq c_1 k^2 - c_2 k$ for $k < n$

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

$$= 4c_1\left(\frac{n}{2}\right)^2 - c_2\left(\frac{n}{2}\right) + n$$

$$= c_1 n^2 + (1 - 2c_2)n$$

$$= c_1 n^2 - c_2 n - \underbrace{(-1 + c_2)}_{\text{want} \geq 0} n$$

$c_2 - 1 \geq 0$

if $c_2 \geq 1$

$$\leq c_1 n^2 - c_2 n \quad \text{if } c_2 \geq 1$$

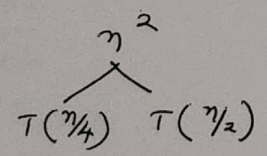
$T(1) \leq c_1 - c_2$ if c_1 is sufficient large w.r.t c_2.

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

<u>Recursion-tree method</u>

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

Expand $T(n) =$



Scanned by CamScanner

Master method

Applies to recurrences of the form

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

where $a \geq 1$, $b > 1$

$f(n)$ is asymptotically positive

$f(n)$ is greater than zero

$f(n) > 0 \;\forall\, n \geq n_0$

compare $f(n)$ with $n^{\log_b a}$

Case 1: $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ $f(n)$ is smaller

for some $\epsilon > 0$

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

Case 2: $f(n) = \Theta\left(n^{\log_b a} \lg^k n\right)$ $f(n)$ is pretty much equal

for some $k \geq 0$

$$\Longrightarrow T(n) = \Theta\left(n^{\log_b a} \lg^{k+1} n\right)$$

Case 3: $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ $f(n)$ is bigger

for some $\epsilon > 0$

and $a f\left(\frac{n}{b}\right) \leq (1-\epsilon') \cdot f(n)$ for some $\epsilon' > 0$

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

(1) $T(n) = 4\,T\left(\frac{n}{2}\right) + n$

$a = 4$ $b = 2$ $f(n) = n$

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

$f(n)$ is smaller

$\therefore$ case 1 $T(n) = \Theta(n^2)$

(2) $T(n) = 4\,T\left(\frac{n}{2}\right) + n^2$ $k = 0$

$\therefore$ case 2 $T(n) = \Theta(n^2 \log n)$

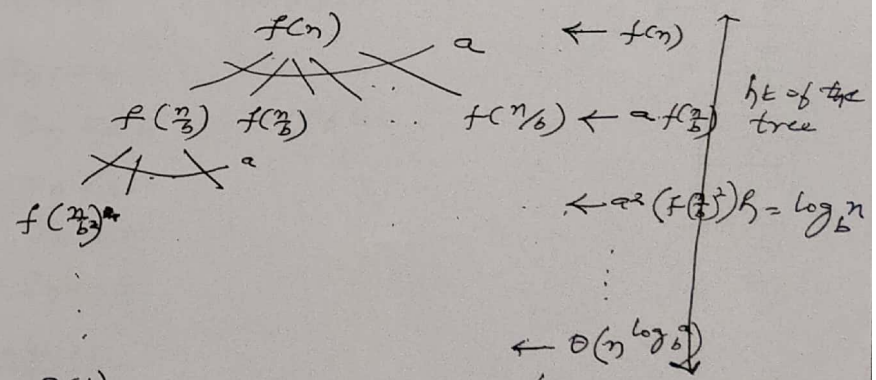
(3) $T(n) = 4\,T\left(\frac{n}{2}\right) + n^3$

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

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

Can't apply master method

Proof sketch / intuition for master theorem



$\Theta(1)$

$$\#\text{ of leaves} = a^h = a^{\log_b n} = n^{\log_b a}$$

Scanned by CamScanner

Case 3 costs decrease geometrically as we go down

$\Rightarrow$ dominated by $f(n)$

Case 1 costs increase geometrically $\downarrow$

$\Rightarrow$ dominated by $\Theta(n^{\log_b a})$

Case 2 each level is roughly same

$\Rightarrow$ cost $= f(n) \cdot h$

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

Amortized Analysis

How large should a hash table be..?

- As large as possible (time)
- space as small as possible (space)
  - ↳ because search should take less time
- $\Theta(n)$ for n items

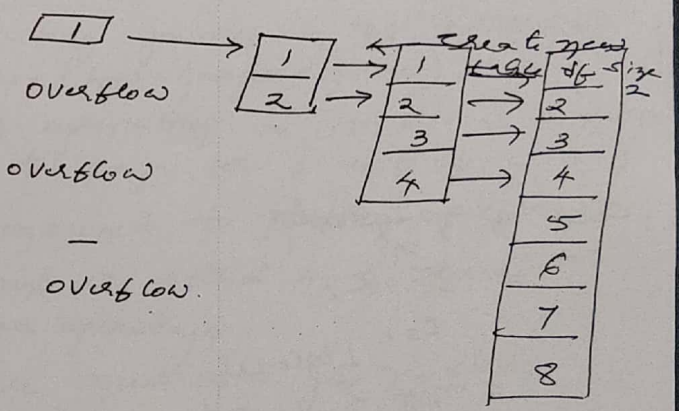
Problem what if we don't know n in advance

Solution Dynamic tables

Idea: whenever the table gets to overflow we need to grow the table

1. Allocate (malloc or new) a larger table
2. Move the items from the old table to new table
3. Free old table

1. Insert
2. Insert — overflow
3. Insert — overflow
4. Insert —
5. Insert — overflow
6. Insert
7. Insert
8. Insert



Analysis

Sequence of n insertion operation

worst case cost of 1 insert $= \Theta(n)$

$\therefore$ worst case cost of n insert $= n \cdot \Theta(n) = \Theta(n^2)$

wrong

n insert take $O(n)$ time

Scanned by CamScanner

Let c_i = cost of ith insert

$$= \begin{cases} i & (i-1) \text{ is exact power of } 2 \\ 1 & ; \text{otherwise} \end{cases}$$

i	1	2	3	4	5	6	7	8	9	10
size i	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1

$$c_i \begin{cases} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ & & & & 4 & \cdot & \cdot & \cdot & 8 & \\ & 1 & 2 & & & & & & & \end{cases}$$

Cost of n inserts

$$= \sum_{i=1}^{n} c_i$$

$$= n + \sum_{j=0}^{\lfloor \lg(n-1) \rfloor} 2^j \quad \text{geometric series}$$

$$= n + 2n$$

$$\leq 3n$$

$$= \Theta(n)$$

Thus the average cost of insert is $= \frac{\Theta(n)}{n} = \Theta(1)$

Amortized analysis

We analyze a sequence of operations to show that average cost per operation is small, even though 1 operation may be expensive.

→ here average is there but no probability

— No probability – average perf. in worst case

Types of amortized arguments

- Aggregate analysis (just we saw)
- accounting } more precise than
- potential } "allocate specific amortized cost" to each operation.

Accounting method

↳ Putting yourself in the position of an accounter. (how charges amount)

→ charge i^{th} operation a fictitious amortized cost c_i ($\$1$ pays for 1 unit of work)

→ Fee is consumed to perform operation

→ Unused amount is stored in the "bank" for use by later operations.

→ Bank balance must not go negative

Must have $\underbrace{\sum_{i=1}^{n} c_i}_{\text{true cost}} \leq \underbrace{\sum_{i=1}^{n} \hat{c}_i}_{\text{amortized cost}} \quad \forall n$

Dynamic table

- charge $\hat{c}_i = \$3$ for i^{th} insert
  $\$1$ pay for immediate insert
  $\$2$ stored for ~~doubling~~ doubling the table
- When the table doubles, [illegible] to move [illegible] elems & 1$ for

0	0	0	0	2	2	2	2

0	0	0	0	0	0	0	0	2	2	2	2				

by charging 3\$, we can easily handle table doubling

Invariant Bank balance ≥ 0

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

$3n$

i	1	2	3	4	5	6	7	8	9	10
size$_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1
c_i	1	1	1	1	1	1	1	1	1	1
		1	2		4				8	
$\hat{c}_i$	2	3	3	3	3	3	3	3	3	3
bank balance	1	2	2	4	2	4	6	8	2	4

special case
choosing \$2 is enough to be charged for 1st element.

\$2 for each operation is not sufficient (balance -ve)
\$4 is too much

Potential method

"Bank account" viewed as potential energy of dynamic set.

Framework

- start with some data structure D_0
- operation i transforms $D_{i-1} \Rightarrow D_i$

Cost of operation i is c_i

Define the potential function

$$\phi : \{D_i\} \to \mathbb{R} \ni$$

such that $\phi(D_0) = 0$ and $\phi(D_i) \geq 0 \ \forall i$

We define amortized cost $\hat{c}_i$ w.r.t ϕ as

$$\hat{c}_i = c_i + \underbrace{\phi(D_i) - \phi(D_{i-1})}_{\text{change in potential or potential difference } \Delta\phi_i}$$

If $\Delta\phi_i \geq 0$ then $\hat{c}_i > c_i$, i.e operation i stores work in data structure for later

If $\Delta\phi_i < 0$ then $\hat{c}_i < c_i$, data structure delivers up stored work to help pay for operation i

Total amortized cost of n ops is

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \phi(D_i) - \underbrace{\phi(D_{i-1})}_{\text{terms telescope}})$$

$$= \sum_{i=1}^{n} c_i + \underbrace{\phi(D_n)}_{\geq 0} - \underbrace{\phi(D_0)}_{0}$$

$$\therefore \geq \sum_{i=1}^{n} c_i$$

$\therefore$ Amortized cost is upper bound on the true cost.

Scanned by CamScanner

Table-doubling

Define $\phi(D_i) = 2i - 2^{\lceil \lg i \rceil}$

Assume $2^{\lceil \lg 0 \rceil} = 0 \quad (2^{-\infty} = 0)$

Note $\phi(D_0) = 0$

$\phi(D_i) \geq 0 \quad \forall i \quad [2^{\lg i + 1} = i - 2]$

Eg :-

0	0	0	0	0	0			

0 0 0 0 2 2

$\phi = 2 \cdot 6 - 2^3$
$= 12 - 8 = 4$

↓

in Accounting method $\phi = 2 \cdot 8 - 2^3$
$= 16 - 8 = 8$

Amort. cost of ith insert

$$\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1})$$

$$= \begin{cases} i & \text{if } (i-1) \text{ is exact power of } 2 \\ 1 & \text{; otherwise} \end{cases}$$

$$+ \left(2i - 2^{\lceil \lg i \rceil}\right) - \left(2(i-1) - 2^{\lceil \lg (i-1) \rceil}\right)$$

$$= \begin{cases} i, & \text{if} \ldots \\ 1 \end{cases} + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil}$$

Case 1 : $(i-1)$ is exact power of 2

$$\hat{c}_i = i + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil}$$

$$= i + 2 - 2(i-1) + i - 1 \quad (i-1)$$

$$= i + 2 - 2i + 2 + i - 1$$

$$= 3$$

Case 2 $(i-1)$ is not exact power of 2

$$\hat{c}_i = 1 + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil}$$

both are same

$$= 1 + 2$$

$$= 3$$

$\therefore$ n insert costs $\Theta(n)$ in worst case

Conclusions

→ Amort. cost provides a clear abstraction for data structure performance.

→ Any method can be used, but each has situation where it is arguably the simplest or most precise.

→ Different potential functions or accounting cost may yield different bounds.

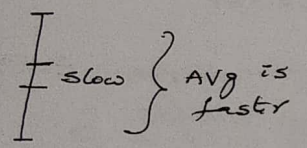
Scanned by CamScanner

Amortized Analysis

In an amortized analysis, the time required to perform a sequence of data structure operation is average of over all the operations performed.

→ Using amortized analysis, we can calculate the time & space complexity.

Application: Amortized analysis can be used to show that the average cost of an operation is small, if one averages over a sequence of operations, even though a single operation within the sequence might be expensive.



Amortized analysis Vs Average case analysis

AA differs from ACA in that probability is not involved, an amort. analysis guarantees the average performance of each operation in the worst case.

Different techniques used in amortized Analysis are

(1) Aggregate Analysis
(2) Accounting method
(3) Potential method.

Aggregate Analysis

In Aggregate analysis for all n, a sequence of n operations takes worst case time $T(n)$ in total. In the worst case, the average cost or amort. cost per operation is $T(n)/n$.

Note that this amortized cost applies to each operation, even when there are several types of operations in the sequence. (This is not true for other two methods).

Example

Stack operation

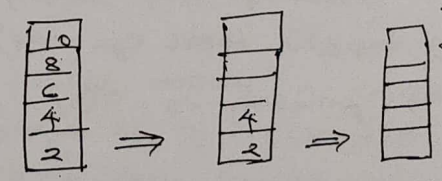
(1) push(x) — $O(1)$
(2) pop() — $O(1)$

The total cost of a sequence of n push and pop operations is n or $O(n)$

(3) multipop(k)

```
{ while (stack not empty && k ≠ 0)
      pop()
      k = k-1;
}
```

→ removes the k top objects of stack
→ pop the entire stack if it contains fewer than k objects



multipop(3) multipop(3)

i.e The total cost of multipop is $\min(n, k)$ and the running time is linear.

Scanned by CamScanner

Suppose a sequence of n push, pop, multipop operations are there.

→ The worst case cost of multipop operation in the sequence is $O(n)$ [$\min(n,k)$ if $n \le k$ then $\min(n,k) = n$]

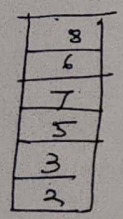
& hence a sequence of n operations cost $O(n^2)$. Since we have $O(n)$ multipop operations costing $O(n)$ each

1 multipop — $O(n)$
n multipop — $O(n^2)$

Any sequence of n push, pop and multipop operations on an initially empty stack can cost at most $O(n)$.

Each object can be popped atmost once for each it is pushed. The no. of times that pop can be called on a nonempty stack, including calls within multipop is at most the no. of push operations which is atmost n. Therefore for any value of n, any sequence of n push, pop and multipop operation takes a total of $O(n)$ time.

The avg. cost of an operation is $O(n)/n = O(1)$



stack

6 data push — 6 $\quad$ n data
pop $\Rightarrow 6$ $\quad$ n data
multipop $\Rightarrow$ 6

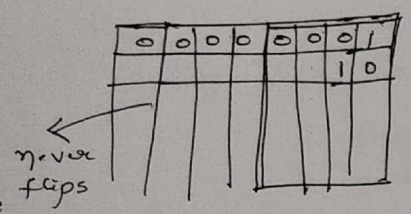
$$\frac{O(n)}{n} = O(1)$$

Scanned by CamScanner

i.e $A[i]$ flips every 2^i times $i = \lfloor \frac{n}{2^i} \rfloor$ times

for $i = 0, 1, \ldots \lfloor \log n \rfloor$, bit $A[i]$ flips $\lfloor \frac{n}{2^i} \rfloor$ times in a sequence of ~~each~~ n increment operation

For $i > \lfloor \lg n \rfloor$, bit $A[i]$ never flips at all



The actual time $i = 0, 1, 2, \ldots \lfloor \log n \rfloor$ the total flips is $\frac{n}{2^i}$

$$\sum_{i=0}^{\lfloor \log n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < \sum_{i=0}^{\infty} \frac{n}{2^i}$$

$$< n \sum_{i=0}^{\infty} \frac{1}{2^i}$$

$$\leq 2n$$

$$= O(n)$$

For n operations, avg cost is $= \frac{O(n)}{n} = O(1)$

Accounting method

In this method, we assign different charges to different operations, with some operation charges more or less than actually cost. The amount we charge an operation is called "amort. cost"

$$\text{credit} = \underset{\text{amort-cost}}{\text{An operation}} - \underset{\text{cost}}{\text{Actual}}$$

Credit can be used later for operation where amort. cost is less than their actual cost.

→ In Aggregate analysis, all operations have the same amort. cost

→ In accounting method, all operations do not have the same amort. cost.

c_i - is the actual cost

$\hat{c}_i$ - is the amortised cost

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

$$\text{Total credit} = \sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i$$

≥ 0 (should always be greater than equal to zero)

Stack operation

(1) push(x) — $O(1)$

(2) pop() — $O(1)$

(3) multipop(k) ⇒ min(k, n)

Scanned by CamScanner

String Matching Algorithms

Naive-string matching

Eg1

	1	2	3	4	5	6	7	8
string :	a	b	c	d	e	f	g	h
pattern :	d	e	f					
		d	e	f				
			d	e	f			
				d	e	f		

pattern is found at index 4

Eg2

	i	i	i	i	i							
	1	2	3	4	5	6	7	8	9	10	11	12
string :	a	b	c	d	a	b	c	a	b	c	d	f
pattern :	a	b	c	d	f							
	1	2	3	4	5							
	j	j	j	j	j (mismatch)							
	a	b	c	d	f							

(this is the problem in naive string matching) → shift size limited to 1

** characters in the string are compared again & again

Worst case

	i 1	i 2	i 3	i 4	i 5	i 6	i 7	i 8
str :	a	a	a	a	a	a	a	b
pattern :	a	a	a	b				
	j 1	j 2	j 3	j 4				

→ naive algorithm do read pattern, it just shifts by 1 on mismatch

Time = $O(mn)$

"observation"

→ Based on doing some preprocessing to the pattern, we can have some pattern matching algo, one such algo is "KMP".

Knuth-Morris-Pratt (KMP) algorithm

→ objective is to find pattern in the string

string :- a b c d e f g h

pattern : d e f

pattern : a b c d a b c
1 2 3 4 5 6 7

prefix : a, ab, abc, abcd (taking from left)

suffix : c, bc, abc, dabc & more (taking from right)

Idea in KMP is to find the prefix same as suffix
i.e abc

→ i.e to check the beginning part of the pattern repeating again.
i.e we use a notation called π → longest prefix table

π or LPS

P_1 : a b c d a b e a b f
0 0 0 0 1 2 0 1 2 0 ← π table

P_2 : a b c d e a b f a b c
0 0 0 0 0 1 2 0 1 2 3

P_3 : a a b c a d a a b c
0 1 0 0 1 0 1 2 3 0

P_4 : a a a a b a a a d
0 1 2 3 0 1 2 0 0

Scanned by CamScanner

String: a b a b c a b c d
1 2 3 4 5 6 7 8 9 10 11 12 13 14

pattern:

j 0	j 1	j 2	j 3	j 4	j 5
a	b	a	b	d	
0	0	1	2	0	

(Note we are starting from 0)

compare str(i) & pat(j+1)

Now (j+1) and (i) is not matching, bring j to 2 (i.e the value under j.

	j 1	j 2	3	4	5
	a	b	a	b	d
	0	0	1	2	0

Now p(j+1) and s(i) are not matching bring j to 0

j 0	j 1	j 2	3	4	5
	a	b	a	b	d
	0	0	1	2	0

Now also p(j+1) and s(i) are mismatch, but we can't move j left, coz

∴ move i to right side i.e position 6

(Note that i is not backtracked)

& compare p(j+1) & s(i)

Now p(j+1) is a & s(i) is c mismatch

∴ move j to 0

j 0	j 1	j 2	j 3	j 4	5
	a	b	a	b	d
	0	0	1	2	0

Now compare p(j+1) & s(i)
a & c mismatch
move i

Compare p(j+1) & s(i)

Now p(i+1) is d & s(i) is a (mismatch)

∴ move j to 2

	1	j 2	j 3	j 4	j 5
0	a	b	a	b	d
	0	0	1	2	0

compare p(j+1) & s(i)

Time complexity ⟹ string − n
pattern − m

$O(m+n)$

m: to construct table (π)

n: No. of moves w.r.t i

```
Naive-String-Matcher(T, P)
n = T.length
m = P.length
for s = 0 to n-m
    if P[1..m] == T[s+1 ... s+m]
        print "pattern occurs with shift" s
```

```
COMPUTE-PREFIX-FUNCTION(P)
m = P.length
let π[1..m] be a new array
π[1] = 0
k = 0
for q = 2 to m
    while k > 0 and P[k+1] ≠ P[q]
        k = π[k]
    if P[k+1] == P[q]
        k = k+1
    π[q] = k
```

Scanned by CamScanner

String: a b a b c a b c a b d

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

pattern:

j 0	1	2	3	4	5
a	b	a	b	d	
0	0	1	2	0	

(Note we are starting from 0)

compare str(i) & pat(j+1)

Now (j+1) and (i) is not matching, bring j to 2 (i.e the value under j.

	1	j 2	3	4	5
	a	b	a	b	d
	0	0	1	2	0

Now p(j+1) and s(i) are not matching bring j to 0

j 0	1	2	3	4	5
	a	b	a	b	d
1	0	0	1	2	0

Now also p(j+1) and s(i) are mismatch, but we can't move j left, wise

∴ move i to right side i.e position 6

(Note that i is not backtracked)

& compare p(j+1) & s(i)

Now p(j+1) is a & s(i) is c mismatch

∴ move j to 0

j 0	1	2	3	4	5
	a	b	a	b	d
	0	0	1	2	0

Now compare p(j+1) & s(i)

a & c mismatch

move i

Compare p(j+1) & s(i)

Now p(i+1) is d & s(i) is a (mismatch)

∴ move j to 2

	1	2	3	4	5
0	a	b	a	b	d
	0	0	1	2	0

Compare p(j+1) & s(i)

Time complexity ⟹ String − n
Pattern − m

$O(m + n)$

m: to construct table (π)

n: No. of moves w.r.t i

```
Naive-String-Matcher(T, P)
n = T.length
m = P.length
for s = 0 to n-m
    if P[1..m] == T[s+1 ... s+m]
        print "pattern occurs with shift" s
```

```
COMPUTE-PREFIX-FUNCTION(P)
m = P.length
let π[1..m] be a new array
π[1] = 0
k = 0
for q = 2 to m
    while k > 0 and P[k+1] ≠ P[q]
        k = π[k]
    if P[k+1] == P[q]
        k = k+1
    π[q] = k
```

Scanned by CamScanner

```
KMP-MATCHER(T, P)
n = T.length
m = P.length
π = COMPUTE-PREFIX-FUNCTION(P)
q = 0
for i = 1 to n
    while q > 0 and P[q+1] ≠ T[i]
        q = π[q]
    if P[q+1] == T[i]
        q = q+1
    if q == m
        print "pattern occurs with shift" i-m
        q = π[q]        // look for the next match
```

Rabin-Karp algorithm

	1	2	3	4	5	6
Text :	a	a	a	a	a	b

	1	2	3
pattern :	a	a	b

Rather than checking for individual character wise, try to convert the entire pattern into a numeric value

Eg: a — 97 (ascii value) (or) a-1 e-5
b — 98 to make it simple b-2 f = 6
c ⋮ c-3 g-7
d d-4 & so on

Now pattern = a a b
$\underbrace{1+1+2}_{\text{hash function } h(p)} = 4$ hash code

Now

Text: a a a a a b

$1+1+1=3$

$1+1+1=3$

$1+1+1=3$

$1+1+2=4$

(Now a match)
check the characters

Sliding to the next value is called "Rolling hash function"

Eg3

Text : a b c d a b c e

$1+2+3=6$

next
$6-1=5+4$

9

& proceed further

Pattern
b c e
$2+3+5=10$

Time — Avg case = $O(n-m+1)$

Scanned by CamScanner

Eg 3

Text : c c a c c a a e d b a

7

7

7

pattern = d b a

$2+4+1=7$

1st hash value same but pattern doesn't match

2nd —"— —"— —"—

3rd —"— —"— —"—

Note that every time, the hash code of the pattern matches, but not the actual characters (collision)

so this is the drawback

↓

"Spurious hits"

↓

waste hits

Time taken = $\Theta(mn)$, but we expect time to be $\Theta(n-m+1)$ this can happen only if we avoid spurious hits.

→ All the above problem of spurious hits are bcoz of the simple hash function, so to avoid spurious hits we need a strong hash function

pattern = d + b + a

$2\times10^2 + 4\times10^1 + 1\times10$

why 10^2 bcoz pattern length is 3

or in general we can have pattern length as m

we have $P(1)\times10^{m-1} + P(2)\times10^{m-2} + P(3)\times10^{m-3} \ldots$

↳ base value if i consider characters a to j (10 th char)

↓

can be different. 26 for alphabets

Text : c c a c c a a e d b a

$3\times10^2+3\times10+1$

331

(i) (no match) 313

not a spurious hit

133

331

311

115

154

542

pattern : d b a

$2\times10^2 + 4\times10 + 10 = 421$

(ii) $[(3\times10^2 + 3\times10 + 1\times10^0) - 3\times10^2]\times10 + 3\times10^0$

↓ 331

now cac $= 331 - 3\times10^2$

$= 331 - 300 = 31$

$= 31\times10 = 310+3 = 313$

not a match

(iii) acc

↳ $313 - 3\times10^2$

$= 13\times10 = 130 + 3\times10^0 = 133$

not a match

(iv) cc a

$= 133 - 1\times10^2$

$= 33\times10 = 330 + 1\times10^0 = 331$

not a match

(v) c a a

$= 331 - 3\times10^2$

$= 31\times10 = 310+1 = 311$

not a match

(vi) a a e

$= 311 - 3\times10^2$

$= 311 - 300 = 11\times10 = 110 + 5\times10^0 = 115$

not a match

(vii) a e d

$= 115 - 1\times10^2 = 115 - 100 = 15\times10 = 150 + 4\times10^0$

$= 154$

not a match

(viii) e d b

$= 154 - 1\times10^2 = 54\times10 = 540 + 2\times10^0 = 542$

not a match

↳ a $542 - 5\times10^2$ . . . match.

Scanned by CamScanner

If we don't have a spurious hit, then the time taken will be $\Theta(n-m+1)$

Note :- while calculating hash value (Code) such as

(i) d a b e

$4\times10^3 + 1\times10^2 + 2\times10^1 + 5\times10^0 =$ 4000
100
20
5
4125

this value is within the range of int

(ii) if pattern is i a c j h a a c

$9\times10^7 + 1\times10^6 + 3\times10^5 + 10\times10 + \ldots$ so on

this value is not within the range

Therefore to avoid this use <u>mod</u> function

as $(4\times10^3 + 1\times10^2 + 2\times10^1 + 5\times10^0) \bmod 2^{31}$ & 2^{32}

for 4 bytes int

d – is base value

```
Rabin-Karp-Matcher(T, P, d, q)
n = T.length
m = P.length
h = d^(m-1) mod q
p = 0
t_0 = 0
for i = 1 to m          // preprocessing
    p = (dp + P[i]) mod q
    t_0 = (dt_0 + T[i]) mod q
for s = 0 to n-m        // matching
    if p == t_s
        if P[1..m] == T[s+1 ... s+m]
            print "pattern occurs with shift" s
    if s < n-m
        t_{s+1} = (d(t_s - T[s+1]h)
                  + T[s+m+1]) mod q
```

Running time in worst case $= \Theta(n-m+1)m$

If $P = a^m$ & $T = a^n$ → then the verification time $\Theta((n-m+1)m)$ since each of $(n-m+1)$ shifts is valid.

In many applications, only few ~~applications~~ shifts & each shift takes $O(1)$ time

∴ then the expected time is $O(n+m)$ plus the time required to process spurious hit. $O(\frac{n}{q})$

Exercise

Working modulo $q = 13$, how many spurious hits does the Rabin-Karp matcher encounter in the text

$T = 3141592653589793$ when looking for the pattern $p = 26$

$T = 3\ 1\ 4\ 1\ 5\ 9\ 2\ 6\ 5\ 3\ 5\ 8\ 9\ 7\ 9\ 3$

5, 1, 2

pattern $= 26$

$= 2 \times 10 + 6 = 26 \bmod 13 = 0$ match ←

31 = 5
14 = 1
41 = 2
15 = 2
59 = 7
92 = 1
26 = 0
65 = 0
53 = 1
35 = 9
58 = 6
89 = 11
97 = 6
79 = 1
93 = 2

Text : 89573210268323554 4031

pattern : 683255 $\quad q = 11$

→ no. of spurious hit

Text : 3141592653589793

pattern : 26 $\quad q = 11$

String matching with finite automata

$\Sigma = \{e, f\}$

text $T[1..n]$: e e f f e f e f f f

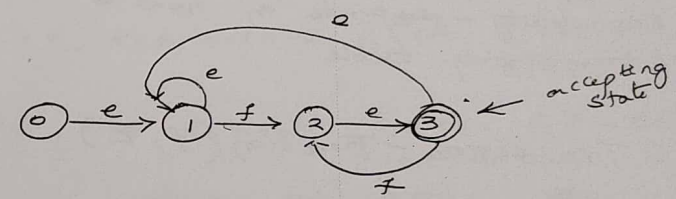
pattern $P[1..m]$: e f e

shift, s = 4 →

finite automation, 5 tuple

$(Q, q_0, A, \Sigma, \delta)$

- Q → finite set of states
- $q_0 \in Q$ → start state
- A → set of accepting states, $A \subseteq Q$
- Σ → input symbol
- δ → (mapping transfer function) transfer function, $Q \times \Sigma \to Q$



(state machine)

δ → transfer function

	e	f
0	1	0
1	1	2
2	3	0
3	1	2

transition table

state

$\delta(q, a) = \sigma(P_q a)$

→ any input character

$P_q a$ = the longest prefix of P that is a suffix of $P_q a$

$\delta(1, e) = 1$, since $P_1 e = ee$ and $\sigma(P_1 e) = 1$

$\delta(2, e) = 3$, since $P_2 e = efe$ & $\sigma(P_2 e) = 3$

Scanned by CamScanner

FA preprocessing algorithm for

→ construct a string matching given pattern.

FA - MATCHER (T, δ, P)

1. n = T.length
2. q = 0
3. for i = 1 to n
4. q = δ(q, T[i])
5. if q == m
   pattern occurs with shift (i-m)
6.

O(n)
linear time

Complexity - depends on how to build the transition table

Transition - Function (P, Σ)

1. m = P.length
2. for q = 0 to m
3. for each character x ∈ Σ
4. k = min(m+1, q+2)
5. repeat
6. k = k-1
7. until $P_k \sqsupset P_q x$
8. δ(q, x) = k
9. return δ

until suff P_k is found

complexity = $O(m^3 \Sigma)$

Tracing the Transition function

pattern = efe = 3 characters

q = 0 x = e, k = 2, k = 2-1 = 1, P_1 = e $P_0 e$ = e
δ(0, e) = 1

x = f, k = 2, k = 2-1 = 1, P_1 = e $P_0 f$ = f
e is not the suffix of f
∴ k = 1-1 = 0 δ(0, f) = 0

q = 1 x = e, k = 3, k = 3-1 = 2 P_2 = ef, $P_1 e$ = ee
ef is not the suffix of ee
∴ k = 2-1 = 1, P_1 = e, δ(1, e) = 1

x = f, k = 3, k = 3-1 = 2 P_2 = ef, $P_1 f$ = ef
δ(1, f) = 2

q = 2 x = e, k = 4, k = 4-1 = 3 P_3 = efe, $P_2 e$ = efe
δ(2, e) = 3

x = f, k = 4, k = 4-1 = 3 P_3 = efe, $P_2 f$ = eff
not a suffix
∴ k = 3-1 = 2 P_2 = ef
∴ δ(2, f) = 0

q = 3
x = e k = 4, k = 4-1 = 3 P_3 = efe $P_3 e$ = efee
which is not the suffix
k = 3-1 = 2 P_2 = e

Scanned by CamScanner

② Text: aabaabadaba

pattern: bada

Transition table

δ

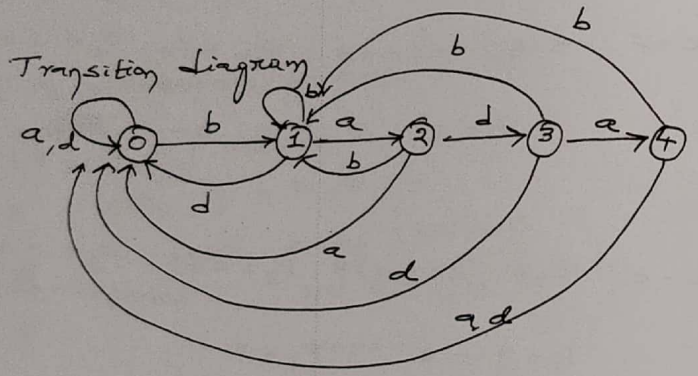
	a	b	d
0	0	1	0
1	2	1	0
2	0	1	3
3	4	1	0
4	0	1	0

baa

badb

badad

Transition diagram



a a b a a b a d a b a

0 1 2 0 1 2 3 4 1 0

i	1	2	3	4	5	6	7	8	9	10	11
T[i]	a	a	b	a	a	b	a	d	a	b	a
φ(Ti)	0	0	0	1	2	0	1	2	3	4	1

pattern

③ Text: ababacaba

pattern: ababaca

Transition table (δ)

	a	b	c	
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

ababaca

δ(aba, b) = 4

δ(aba, a) = 1

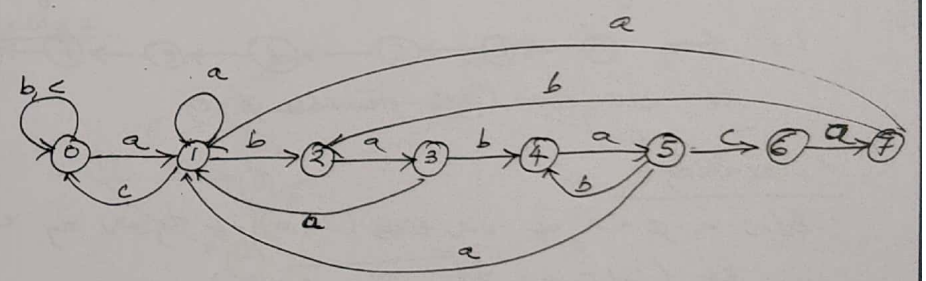
δ(aba, c) = 0

ababaca

δ(ababa, c) = 6

δ(ababa, a) = 1

δ(ababa, b) = 4



i	1	2	3	4	5	6	7	8	9	10	11
T[i]	a	b	a	b	a	b	a	c	a	b	a
φ(Ti)	0	1	2	3	4	5	4	5	6	7	2	3

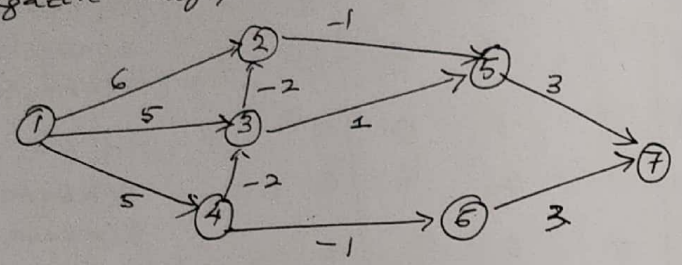
pattern

Bellman - Ford Algorithm

— Single source shortest path algorithm

Objective is to find out the shortest path to all other vertices from a given source vertex

→ it is different from Dijkstra's Algorithm w.r.t to negative weights.



$|V| = n = 7$

shud ~~vertex~~ be relaxed $(n-1)$ times.

Idea is if there are 7 nodes, max. no. of edges 6

i.e from a → b → c → d → e → f → g

it will cover all possible path

Relaxation

b/w a pair of vertices (u, v) if there are any edge

if $(d[u] + c(u,v) < d[v])$

$d[v] = d[u] + c[u,v]$

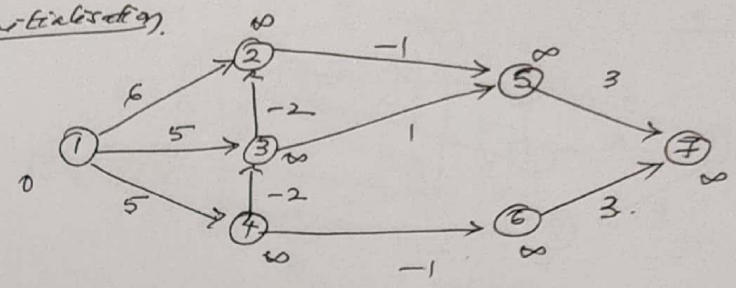
First prepare the edge list.

(1,2) (1,3) (1,4) (2,5) (3,2) (3,5) (4,3) (4,6)
(5,7) (6,7)

Mark source as 0

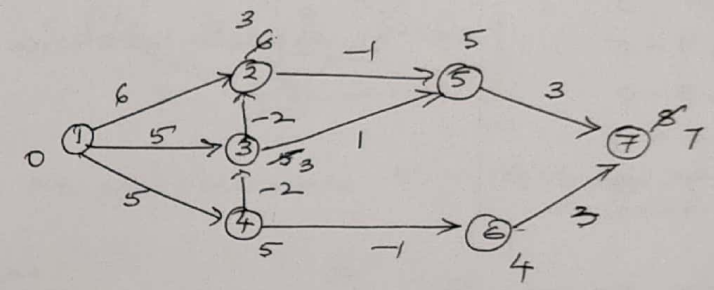
all other vertex with ∞

Initialization

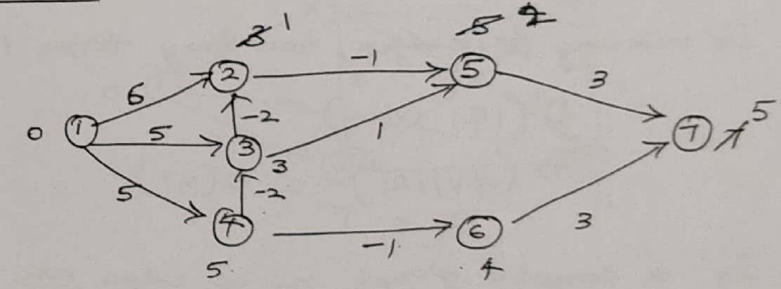


w.r.t to the above example we shud relax 6 times as there are 7 vertices.

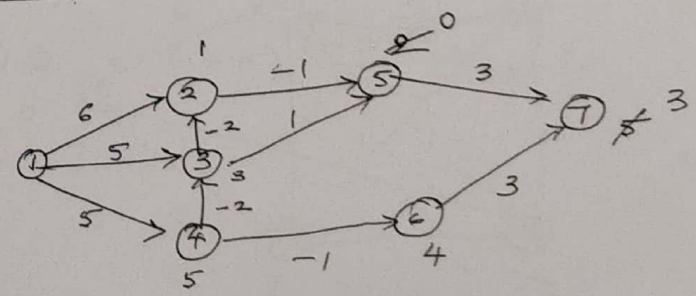
Relax 1



Relax 2

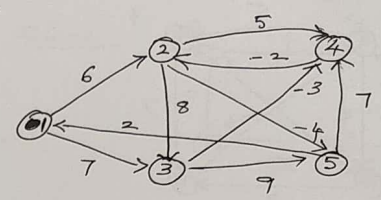


Relax 3



Scanned by CamScanner

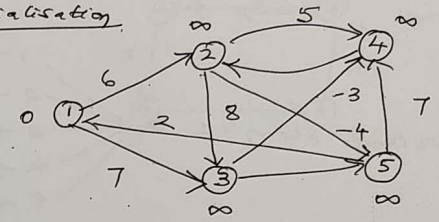
Eg:



(1, 2)
(1, 3)
(2, 3)
(2, 4)
(2, 5)
(3, 4)
(3, 5)
(4, 2)
(5, 1)
(5, 4)

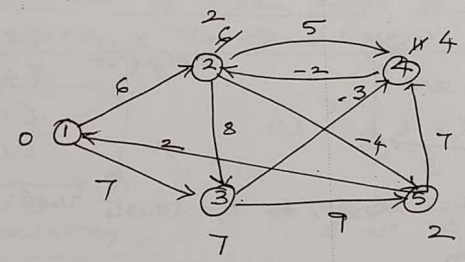
Let 1 be the source vertex

Initialisation

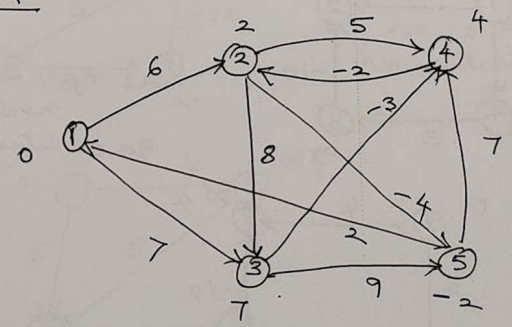


No. of vertex are 5 ; therefore no. of relaxation = 5-1 = 4

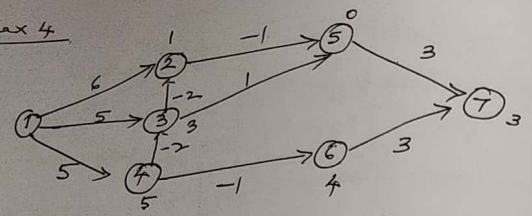
Relax 1



Relax 2



Relax 4



∴ No changes ∴ Relax 5 & Relax 6 remains unch

1 — 0
2 — 1
3 — 3
4 — 5
5 — 0
6 — 4
7 — 3

is the shortest distance

Time complexity

its relaxing |E| edges, how many times |V|-

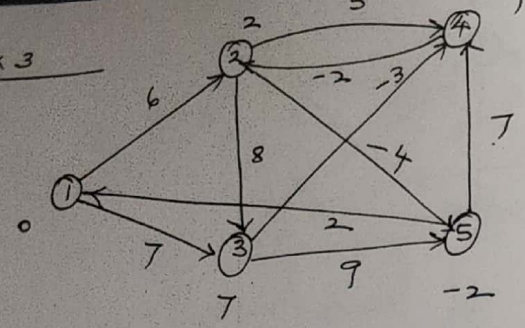
$$O(|E| \cdot |V|-1)$$

$$O(\underset{n}{|V|}\underset{n}{|E|}) = O(n^2)$$

In a complete graph no. of edges are n(
edge b/w every pair

$$\therefore \text{ie } \frac{n(n-1)}{2} \cdot (n-1) = O(n^3)$$

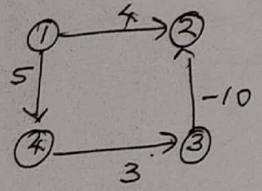
Relax 3



No change, ∴ stop here

∴ shortest distance from 1 to
- 1 → 0
- 2 → 2
- 3 → 7
- 4 → 4
- 5 → -2

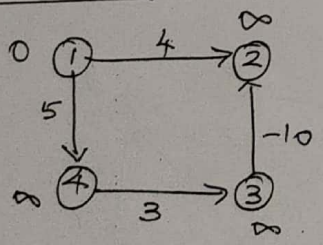
Eg 3



edgelist
(3, 2)
(4, 3)
(1, 4)
(1, 2)

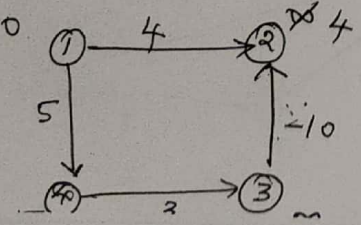
Let 1 be the source vertex

Initialisation

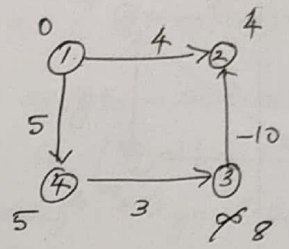


as there are 4 nodes
no. of relax = 4-1

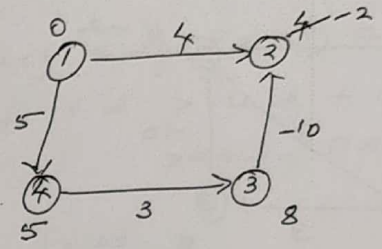
Relax 1



Relax 2



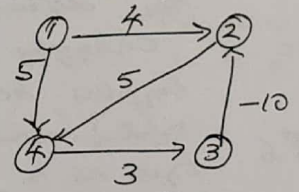
Relax 3



even if you do any more relaxation, the values doesn't change

∴ shortest distance are
1 − 1 = 0
1 − 2 = −2
1 − 3 = 8
1 − 4 = 5

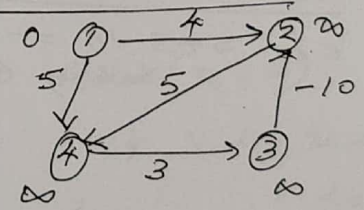
Let's add an extra edge to Eg:- 3 to understand the draw back



Edge list
(3 2) (4 3) (1 4) (1 2) (2

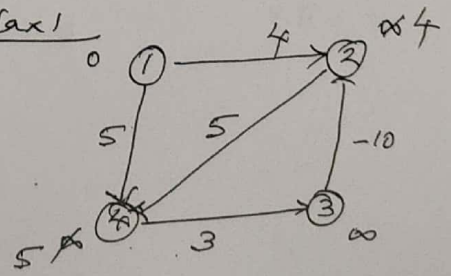
source vertex with 0 & all other vertices to ∞

Initialisation

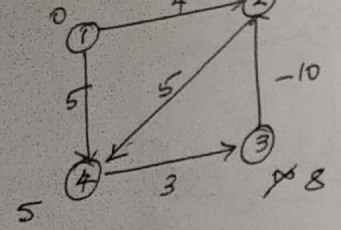


No. of vertex = 4
No. of relax = 4-1 = 3

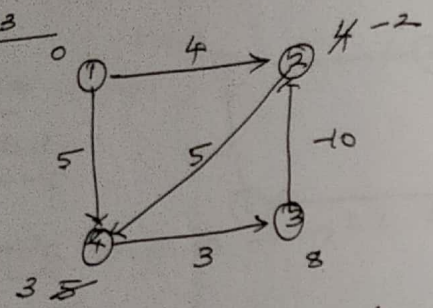
Relax 1



Relax 2

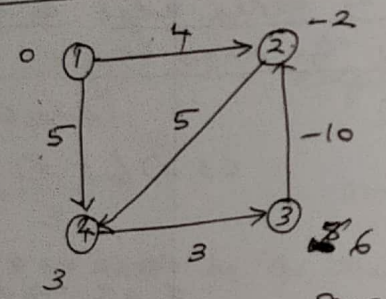


Relax 3



do an additional relax & check

Relax - 4



the distances changing, so implies, bellman fold fails & there is a negative cycle

i.e $5+3-10=-2$ between 3

BELLMAN-FORD (G, w, s) $\quad G=(V,E)$

```
Initialize-single-source (G, s)
for i = 1 to |G.V| - 1
    for each edge (u, v) ∈ G.E
        Relax(u, v, w)
for each edge (u, v) ∈ G.E
    if v.d > u.d + w(u, v)
        return FALSE
return True
```

Initialize-single-source (G, s)

```
for each vertex v ∈ G.V
    v.d = ∞
    v.π = NIL    // not required (which holds the predecessor)
s.d = 0
```

d & π are always

Relax(u, v, w)

```
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```

Scanned by CamScanner

Single-source shortest paths in acyclic graphs

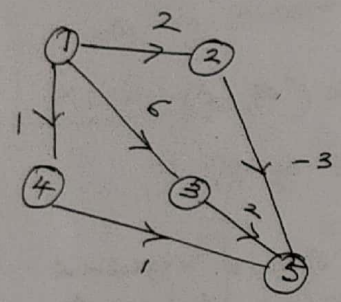
Note

→ distance from start to start is 0

→ distance from start vertex to any other vertex is ∞. (+ve)

→ ***Iterate thru all vertices in topological order

→ Examine outgoing edges & opp. vertices & perform relaxation operation whenever possible



Topological ordering

1 2 3 4 5

	1	2	3	4	5
	0	∞	∞	∞	∞
1	0	2	6	1	∞
2	0	2	6	1	-
3	0	2	6	1	-
4	0	2	6	1	
5	0	2	6	1	

DAG-SHORTEST-PATHS (G, w, s)

1. Topologically sort the vertices of G
2. Initialize-single-source (G, s)
3. for each vertex u, taken in topological order
4. for each vertex v ∈ G.Adj[u]
5. Relax (u, v, w)

Time →

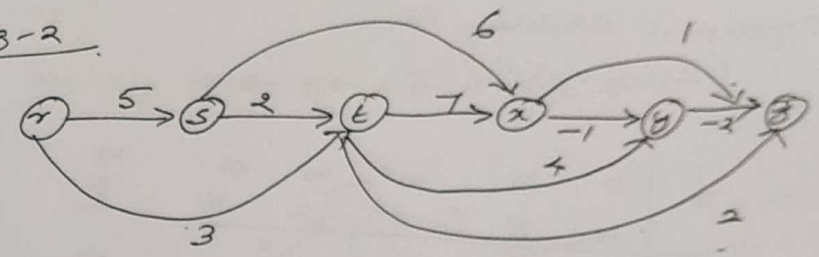
step1 – $\Theta(V+E)$

step2 – $\Theta(V)$

step3–step5 – $\Theta(1)$, (Aggregate analysis)

∴ Total time = $\Theta(V+E)$

Ex 8-2

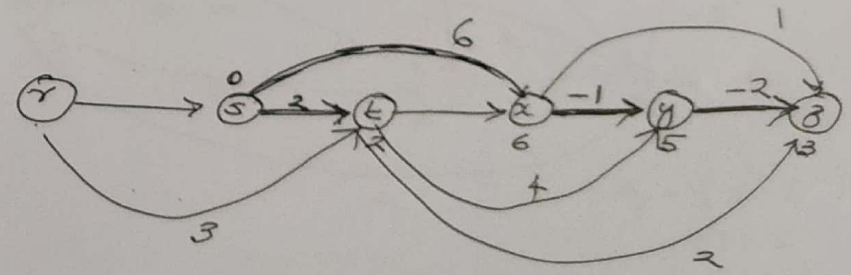


Topological ordering

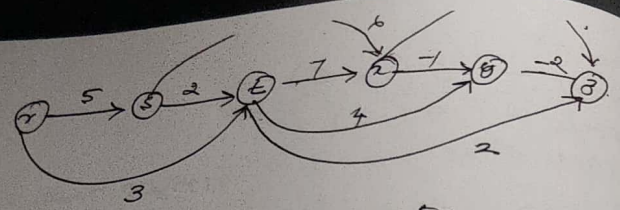
r s t x y z

s be source vertex

	r	s	t	x	y	z
	∞	0	∞	∞	∞	∞
r	∞	0	3	∞	∞	∞
s	∞	0	2	6	∞	∞
t	∞	0	2	6	6	4
x	∞	0	2	6	5	4
y	∞	0	2	6	5	3
z	∞	0	2	6	5	3



Scanned by CamScanner

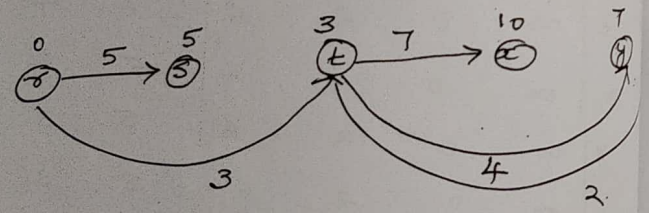


Consider r as source vertex

Topological Ordering is

$r \to s \to t \to x \to y \to z$

	0	∞	∞	∞	∞	∞
	r	s	t	x	y	z
r	0	5	3	∞	∞	∞
s	0	5	3	11	∞	∞
t	0	5	3	10	7	5
x	0	5	3	10	7	5
y	0	5	3	10	7	5
z	0	5	3	10	7	5



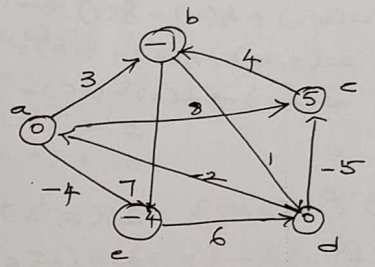
Johnson's algorithm for sparse graphs.

↳ All pair shortest path algorithm

→ Works with the help of both Dijkstra's & Bellman ford.

→ Johnson algorithm uses the technique "reweighting"

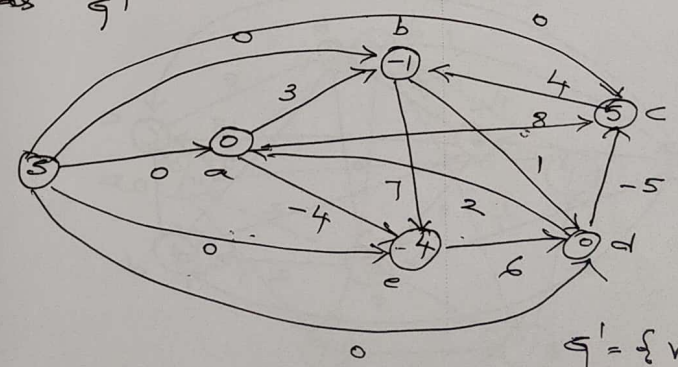
→ running time is $O(V^2 \lg V + VE)$



$G = \{V\ E\}$
$|V| = 5$
$|E| = 9$

→ The above graph G is the result of bellman-ford (single-source shortest path) with source vertex as a

→ To the above graph, add an extra node s, and have an edge connecting to all the vertices with a cost of 0 & call the graph as G'



$G' = \{V'\ E'\}$
$|V'| = 5$
$|E'| = 14$

Now calculate shortest distance from s to all other vertices

$h(a) = \delta(s, a) = 0$

$h(b) = \delta(s, b) = s \to d \to c \to b = 0 - 5 + 4 = -1$

$h(c) = \delta(s, c) = s \to d \to c = 0 - 5 = -5$

$h(d) = \delta(s, d) = s \to d = 0$

$h(e) = \delta(s, e) = s \to a \to e = 0 - 4 = -4$

$w'(u,v) = w(u,v) + h(u) - h(v)$ ← Calculate for all edges

$w'(a,b) = w(a,b) + h(a) - h(b) = 3 + 0 - (-1)$

$w'(a,c) = w(a,c) + h(a) - h(c) = 8 + 0 - (-5)$

$w'(a,e) = 0$

$w'(b,d) = 0$

$w'(b,e) = 10$

$w'(c,b) = 0$

$w'(d,c) = 0$

$w'(d,a) = 2$

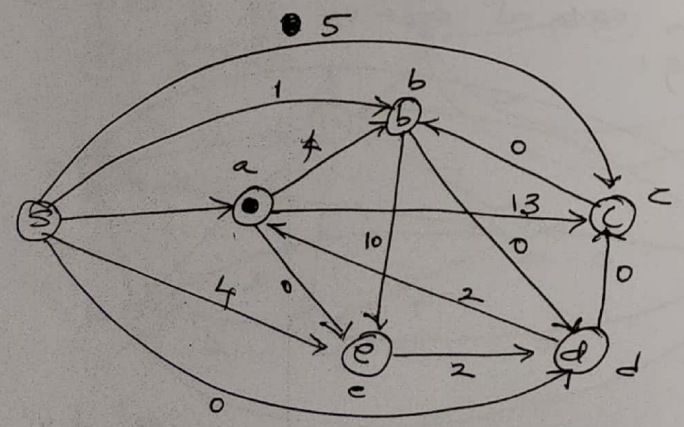
$w'(s,a) = 0$

$w'(s,b) = 1$

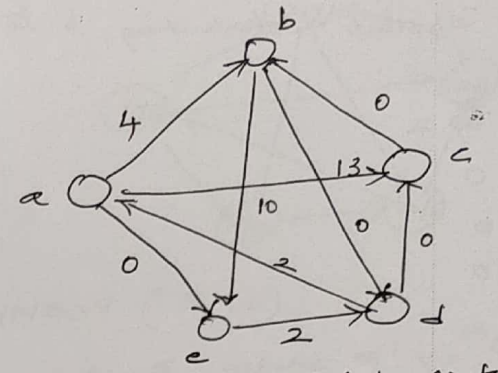
$w'(s,c) = 5$

$w'(s,d) = 0$

$w'(s,c) = 4$



Now delete vertex s



(i) Now calculate (shortest distance from a to all other v)

$\delta'(a,a) = 0$

$\delta'(a,b) = 2$

$\delta'(a,c) = 2$

$\delta'(a,d) = 2$

$\delta'(a,e) = 0$

Now find δ i.e. $\delta(u,v) = \delta'(u,v) - h(u) + h(v)$

$\delta(a,a) = \delta'(a,a) - h(a) + h(a)$

$= 0 - 0 + 0 = 0$

$\delta(a,b) = \delta'(a,b) - h(a) + h(b)$

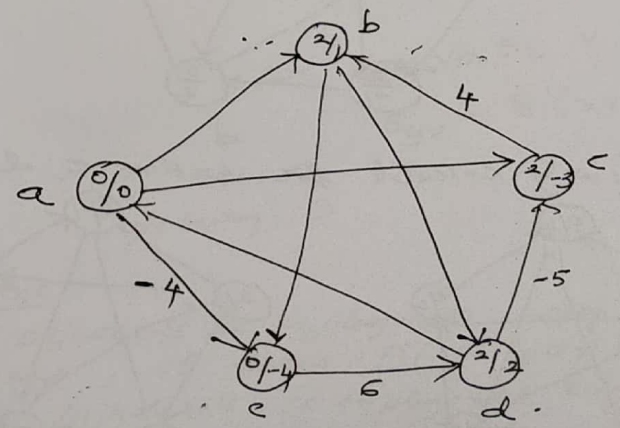
$= 2 - 0 - 1 = 1$

$\delta(a,c) = -3$

$\delta(a,d) = 2$

$\delta(a,e) = 0 - 0 - 4 = -4$

$\delta'(a,a)$
$\delta(a,a)$



(ii) Now calculate δ' for other vertices

$\delta'(b,a) = 2$

$\delta'(b,b) = 0$

$\delta'(b,c) = 0$

$\delta'(b,d) = 0$

$\delta'(b,e) = 2$

find δ i.e $\delta(u,v) = \delta'(u,v) - h(u) + h(v)$

$\delta(b,a) = \delta'(b,a) - h(b) + h(a)$

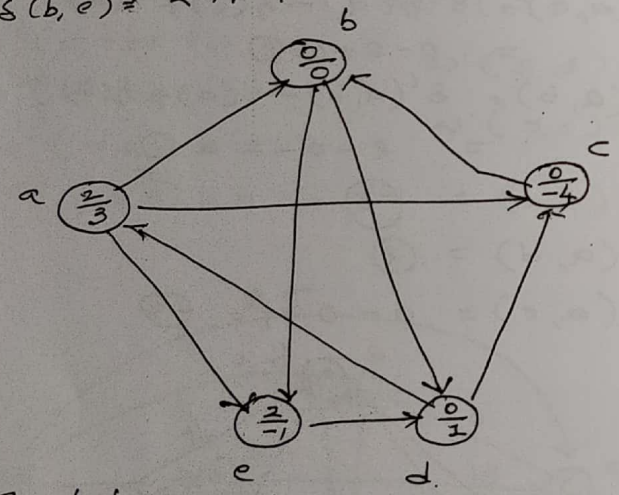
$= 2 + 1 + 0 = 3$

$\delta(b,b) = 0$

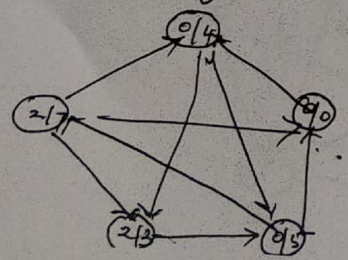
$\delta(b,c) = 0 + 1 - 5 = -4$

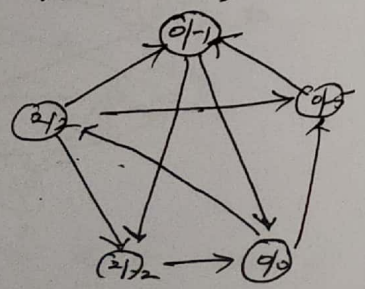
$\delta(b,d) = 0 + 1 + 0 = 1$

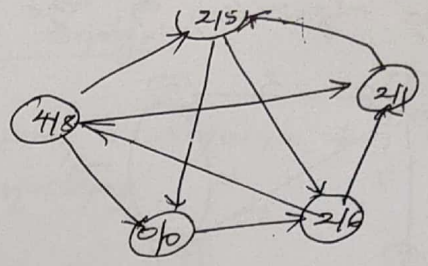
$\delta(b,e) = 2 + 1 - 4 = -1$



Similarly calculate for vertices c, d & e







JOHNSON (G, w)

compute G' where $G'.V = G.V \cup \{s\}$
$G'.E = G.E \cup \{(s,v) : v \in G.V\}$ and
$w(s,v) = 0$ for all $v \in G.V$

if Bellman-Ford $(G', w, s) == FALSE$
print "input graph contains a negative wt. cycle"

else for each vertex $v \in G'.V$
set $h(v)$ to the value of $\delta(s,v)$ computed by bellman-ford algo

for each edge $(u,v) \in G'.E$
$w'(u,v) = w(u,v) + h(u) - h(v)$

Let $D = (d_{uv})$ be a new $n \times n$ matrix

for each vertex $u \in G.V$
run Dijkstra's (G, w', u) to compute $\delta'(u,v)$ for all $v \in G.V$

for each vertex $v \in G.V$
$d_{uv} = \delta'(u,v) + h(v) - h(u)$

return D

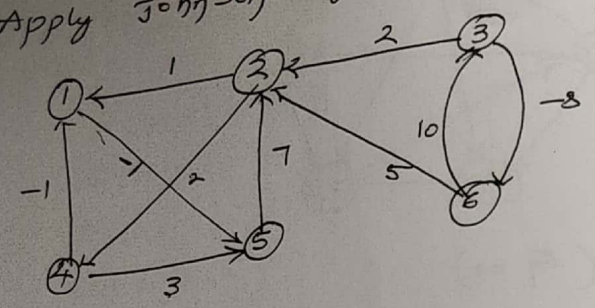
Run time

(i) if Dijkstra's is implemented using fibonacci heap, it takes $O(V^2 \lg V + VE)$

(ii) if Dijkstra's is implemented using binary heap it takes $O(VE \log V)$

Scanned by CamScanner

Apply Johnson



Kmp String matching

1. Text: b a c b a b a b a b a c a

pattern: a b a b a c a

π table

j	1	2	3	4	5	6	7
	a	b	a	b	a	c	a
	0	0	1	2	3	0	1

2. Text: 0 0 1 0 0 1 0 0 2 0 0 0 1 0 0 2 0 1 2 2 0 0

pattern: 0 0 1 0 0 2 0 1

1	2	3	4	5	6	7	8
0	0	1	0	0	2	0	1
0	1	0	1	2	0	1	0

Amortized Analysis

1. Stack : With a sequence of n operations of push pop and multipop.

Aggregate Analysis

In a sequence of n operations, let l be the no. of multi-pop operations and the rest are push & pop operations.

Clearly l multi-pop operations perform at most n pops & therefore the cost is $O(n)$

The cost of $(n-l)$ push & pop operations is $O(n)$

∴ The total cost $= O(n) + O(n)$

Amortized cost $= \frac{O(n)}{n} = O(1)$

Accounting method :-

Let assign 2 credit(dollar) for an push operation i.e 1 credit will be used for pushing the element which is the actual cost & 1 more credit is stored at x (element pushed)

For pop operation, we don't charge anything i.e pop operation is free (but the actual cost is 1)

Similarly multipop is a free operation

Cost of push is 2 = $O(1)$

Cost of pop is 0 = $O(1)$

Cost of multipop is 0 = $O(1)$

Scanned by CamScanner

Potential function method

— need to define a potential function capturing some structural parameter.

— A natural structural parameter of interest for stack is the no. of elements. so lets take no. of elements as potential function.

1. push

$$\hat{c}_{push} = c_{push} + \phi_i - \phi_{i-1}$$

$$= 1 + \cancel{x} + 1 - \cancel{x}$$

$$= 2$$

where x is no. of elements after (i-1)

2. pop

$$\hat{c}_{pop} = c_{pop} + \phi_i - \phi_{i-1}$$

$$= 1 + x - (1 + x)$$

$$= 1 + x - 1 - x$$

$$= 0.$$

where x is the no. of elements at

3. Multipop

$$\hat{c}_{multipop} = c_{multipop} + \phi_i - \phi_{i-1}$$

$$= k + n - (k + n)$$

$$= k + n - k - n$$

$$= 0$$

where n

∴ The amortized cost is $O(1)$

Binary counter of k bits initially set to all 0's.

→ increment operation — increment on a counter adds a bit 1 to the current value of the counter

001101

→ After increment it becomes 001110

→ decrement operation — decrement on a counter subtracts a bit 1 to the current value of the counter

001101

→ After decrement it becomes 001100.

→ reset operation — makes the counter bits to 0's

Amortized Analysis on a sequence of n increment operations.

Normally, for each increment in worst case it may take $O(k)$ time.

for n such operations, it will take $O(nk)$

objective here is to show that amortized time as $O(1)$

Aggregate analysis

A[0] flips each time increment is called

A[1] flips every other time i.e $\lfloor \frac{n}{2} \rfloor$ times

A[2] flips every fourth time i.e $\lfloor \frac{n}{4} \rfloor$ time

in general

A[i] flips every 2^i times i.e $\left\lfloor \frac{n}{2^i} \right\rfloor$ times.

for $i = 0, 1, 2 \ldots \lfloor \log n \rfloor$, bit A[i] flips $\left\lfloor \frac{n}{2^i} \right\rfloor$ ti

for $i > \lfloor \log n \rfloor$, bit A[i] never flips at all.

Scanned by CamScanner

∴ The actual time $i = 0, 1, 2 \ldots \log n$

$$\sum_{i=0}^{\lfloor \log n \rfloor} \frac{n}{2^i} < \sum_{i=0}^{\infty} \frac{n}{2^i}$$

$$< n \sum_{i=0}^{\infty} \frac{1}{2^i}$$

$$\leq 2n$$

$$= O(n)$$

∴ For n operations, avg cost is $\frac{O(n)}{n} = O(1)$

Accounting method

→ We assign 2 credits with each bit when it is flipped from 0 to 1

1 credit for the actual flip & the other 1 credit will be stored at bit th

→ when a bit is flipped from 1 to 0 in subsequ increments, it is done for free. (the credit stor earlier will be used)

		charge	time
flipping	bit 0 to 1	2	$O(1)$
flipping	bit 1 to 0	0	$O(1)$

∴ Amortized cost of increment is $O(1)$

Potential function

→ Structural parameter of interest is the num of 1's in the counter. During ith iteration ones after the last zero is set to zero & the last zero is set to 1. (take an example)

counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

→ Let x denotes the total no. of 1's before the ith operation & t denotes the no. of one's after the last zero. $x = 4$, $t = 4$

→ At the end of ith operation there will be $x - t + 1$ ones, t are changed to 0 & last zero is changed to 1. [Actual cost for increment is $1 + t$]

→ ∴ Amortized cost is

$$\hat{c}_i = c_i + \phi_i - \phi_{i-1}$$

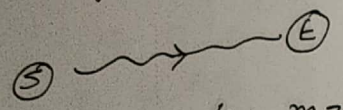
$$= 1 + t + (x - t + 1) - x$$

$$= O(1)$$

Scanned by CamScanner

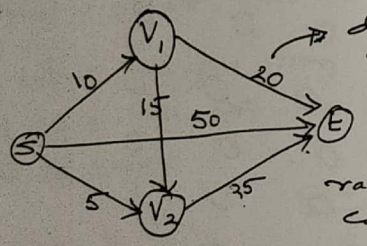
Maximum flow :-

Imagine a material coursing through a system from a source (s) where the material is produced to a sink or terminal (t) where it is consumed.



The source produces the material at some steady rate & the sink consumes the material at the same rate.

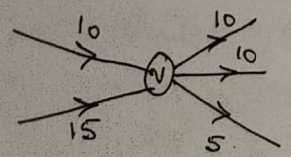
→ The "flow" of the material at any point in the system is intuitively the rate at which the material flows. Flow networks can model many problems including liquids flowing thru pipes, parts through assembly lines, current thru electrical networks or information thru communication networks.



→ directed edge are conduct for the material

→ Each conduct has stated capacity (rate at which the material can flow.

→ In other words, the rate at which material enters a vertex must equal the rate at which it leaves the vertex. We call this property as "flow conservation".



In max-flow problem, we wish to find the greatest rate at which we can ship material from the source to the sink without violating any capacity constraints.

↳ flow shud never be greater than the capacity.

Flow network (Formal definition)

→ A flow network $G = (V, E)$ is a directed graph in which each edge $(u,v) \in E$ has a nonnegative capacity $c(u,v) \geq 0$. We further require that if E contains an edge (u,v) then there is no edge (v,u) in the reverse direction.

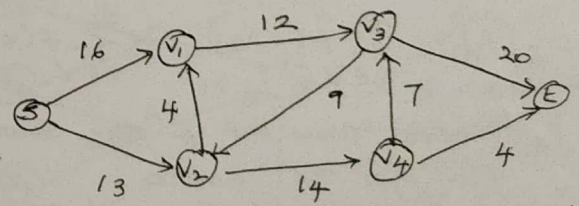
→ If $(u,v) \notin E$, then for convenience we define $c(u,v) = 0$ and we disallow self-loops.

→ 2 special vertex s – source & t – sink or terminal

→ We further assume that each vertex lies on some path from source to sink. i.e for each vertex $v \in V$, the flow n/w contains the path $s \rightsquigarrow v \rightsquigarrow t$

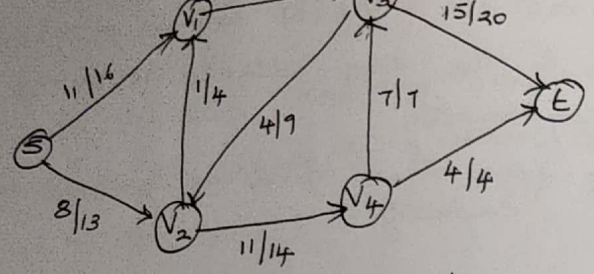
The graph is therefore connected & since each vertex other source (s) has atleast one entering (incoming edge) $|E| \geq |V| - 1$

→ The figure below shows an example of a flow n/w



→ s is the source, t is the sink & each edge has a capacity.

Scanned by CamScanner



A flow f in G with value $|f| = 19$. Each edge (u,v) is labeled by $f(u,v) / c(u,v)$.

Flow

Let $G = (V, E)$ be a flow network with a capacity function c. Let s be the source and t be the sink.

A flow in G is a real-valued function $f : V \times V \to \mathbb{R}$ that satisfies the following 2 properties :-

(1) Capacity constraint → for all $u, v \in V$, we require
$$0 \leq f(u,v) \leq c(u,v)$$

(2) Flow conservation → for all $u \in V - \{s, t\}$ we require
$$\sum_{u \in V} f(v,u) = \sum_{v \in V} f(u,v)$$

when $(u,v) \notin E$, there can be no flow from u to v & $f(u,v) = 0$.

The non-negative quantity $f(u,v)$ is termed flow from vertex u to v. The value of $|f|$ is defined as

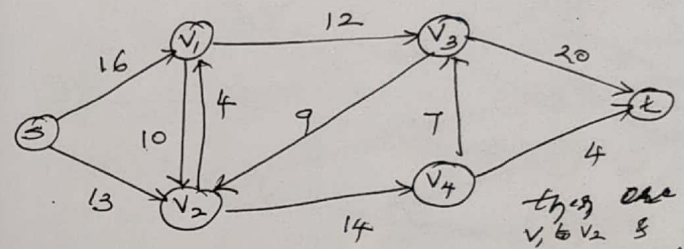
$$|f| = \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s)$$

ie total flow out of the source minus the flow into the source.

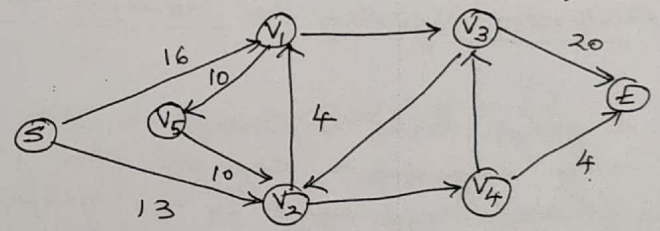
→ typically a flow network will not have any edges into the source & the flow into the source, given by the summation $\sum_{v \in V} f(v,s)$ will be 0.

In the max-flow problem, we are given a flow network G with source s and sink t, & we wish to find a flow of maximum value.

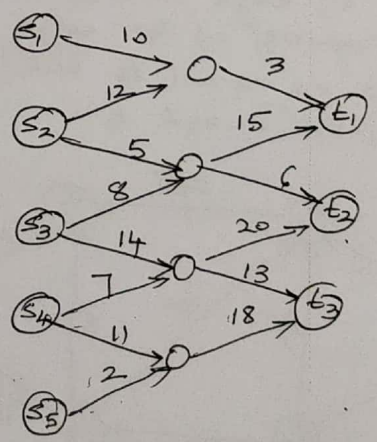
Modelling problems with antiparallel edges.



there are edges from V_1 to V_2 & V_2 to V_1 therefore the n/w should be transformed to



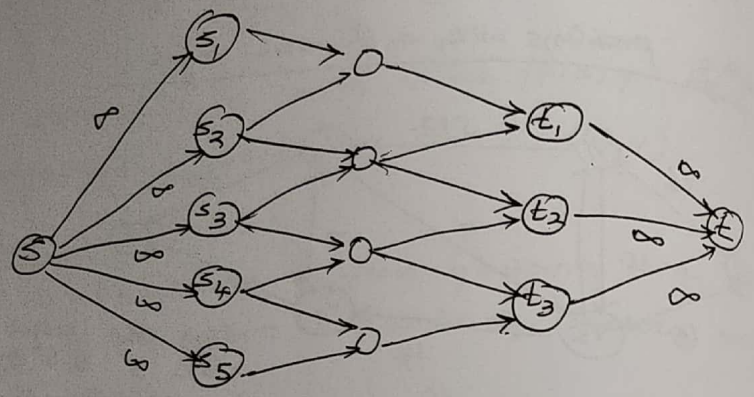
Network with multiple source & sinks.



here we need to add a supersource s & add directed edges from s to s_i with capacity $c(s, s_i) = \infty$ for $i = 1, 2, \ldots, m$ (where m is no. of sources)

Similarly we ... t_i to ... capacity directed edges from ... for $i = 1, 2, \ldots n$

ie $c(t_i, t) = \infty$ (n is no. of sinks)

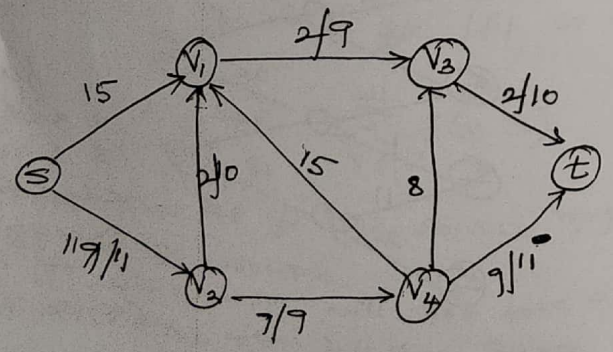


Ford-Fulkerson Algorithm for Maximum flow problem.

Given a graph which represents a flow net... where every edge has a capacity. Also given ... vertices source s and sink t in the graph. Fin... out the maximum possible flow from s to t with following constraints.

a) Flow on an edge doesn't exceed the given capacity of the edge.

b) In-flow is equal to out-flow for every vertex s and t.



Terminologies

1. Residual graph – Its a graph which indicates additional possible flow. If there is such path from source to sink then there is a possibility to add flow.

2. Residual capacity – It's original capacity of the edge minus flow.

3. Minimal cut – Also known as the bottle neck capacity, which decides max. possible flow from s to t thru an augmented path

4. Augmenting path – Augmenting path can be done in 2 ways
   1) Non-full forward edges
   2) Non-empty backward edges

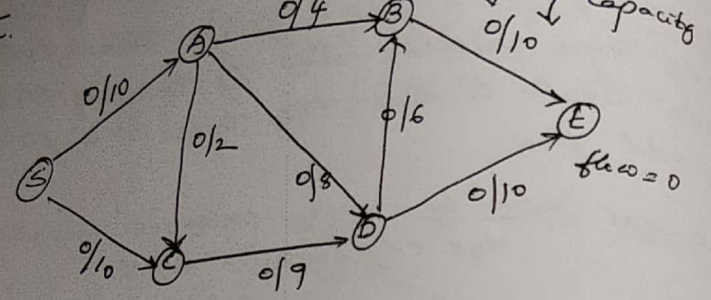
Algorithm

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

1. Initialize flow f to 0
2. while there exists an augmenting path P in the residual network G_f
3. Augment flow f along P
4. return f

Q:-

①



Augmenting path

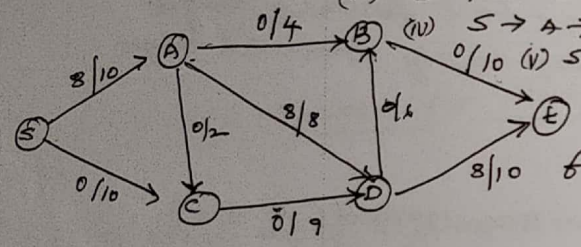
(i) $S \to A \to D \to t$ 8

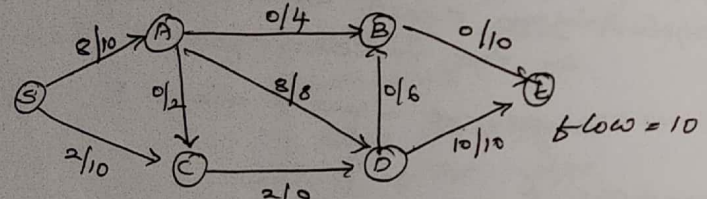
(ii) $S \to C \to D \to t$ 2

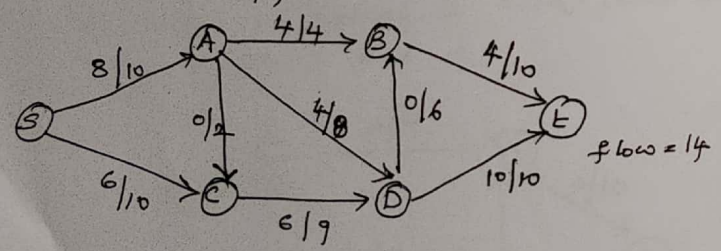
(iii) $S \to C \to D \to A \to B \to t$ 4

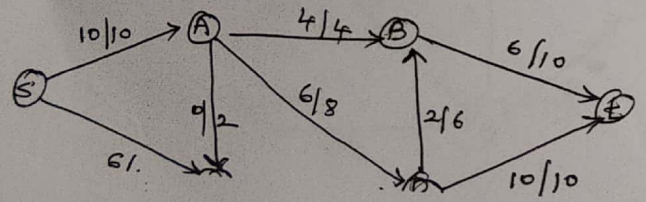
(iv) $S \to A \to D \to B \to t$ 2

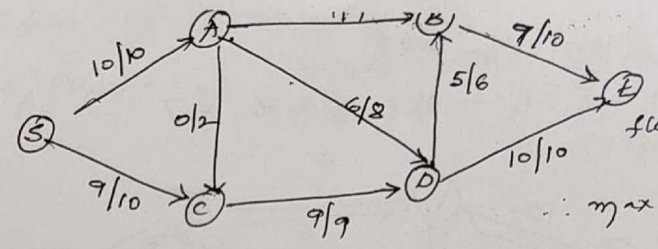
(v) $S \to C \to D \to B \to t$ 3





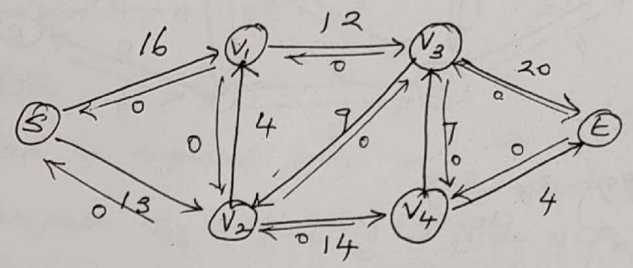






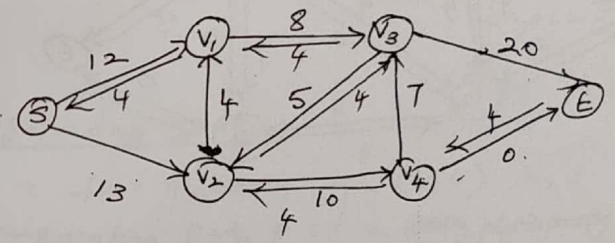
∴ max flow = 19

②



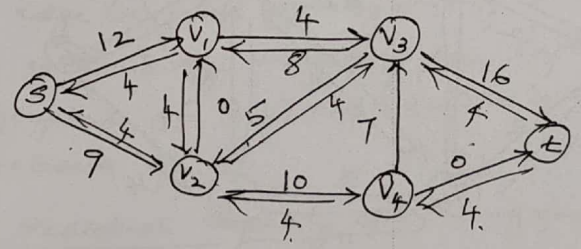
(i) Augmenting path

$S \to V_1 \to V_3 \to V_2 \to V_4 \to t = 4$



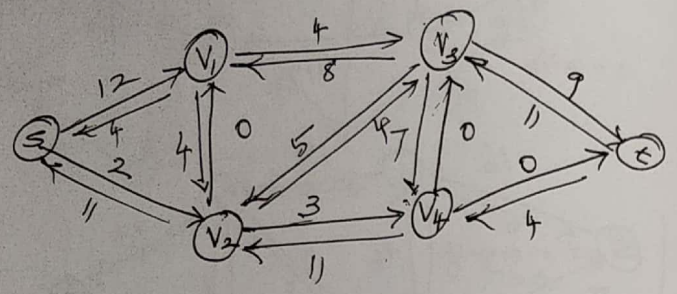
(ii) Augmenting path

$S - V_2 - V_1 - V_3 - t = 4$



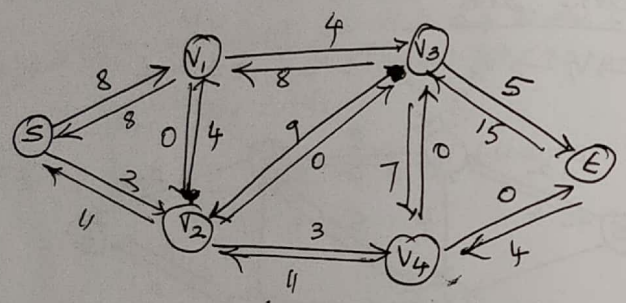
(ii) Augmenting path

$s - v_2 - v_4 - v_3 - t = 7$



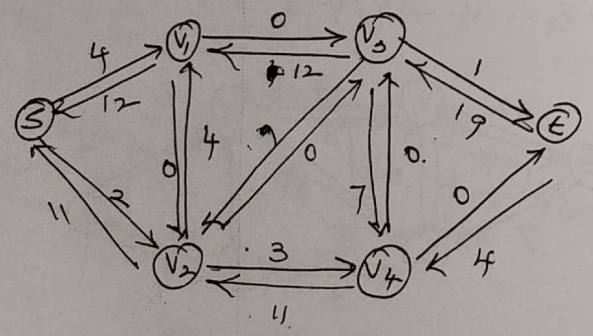
(iii) Augmenting path

$s - v_1 - v_2 - v_3 - t = 4$



(v) Augmenting path

$s - v_1 - v_3 - t = 4$



Max flow = 23

We define the residual [illegible]

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E \\ f(v,u) & \text{if } (v,u) \in E \\ 0 & \text{otherwise} \end{cases}$$

Complexity

Steps

(1) Find an augment path $\longrightarrow O(|E|)$

(2) Compute the bottleneck capacity

(3) Augment each edge & total flow

$\longrightarrow O(|f|)$

$\therefore$ Total time complexity $= O(E.F)$

Augmenting path

→ Given a flow network $G = (V, E)$ and a flow f, an augmenting path p is a simple path from s to t in the residual network G_f. By the definition of the residual network, we may increase the flow on a edge (u,v) of an augmenting path by up to $c_f(u,v)$ without ~~the~~ violating the capacity constraint on whichever of (u,v) and (v,u) is in the original flow network G.

→ The residual capacity of p (augmenting path is given by

$$c_f(p) = \min\left(c_f(u,v) : (u,v) \text{ is on } p\right)$$

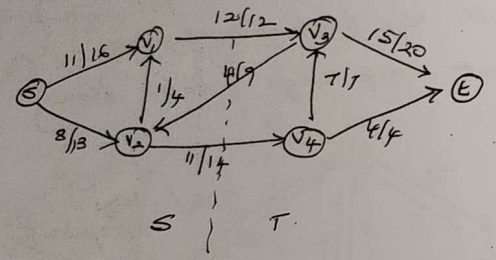
Scanned by CamScanner

Cuts of flow network

A cut (S,T) of flow network $G = (V, E)$ is a partition of V into S and $T = V - S$ such that $s \in S$ and $t \in T$.

→ If f is a flow, then the net flow $f(S,T)$ across the cut (S,T) is defined to be

$$f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u)$$



$S = \{s, v_1, v_2\}$

$T = \{v_3, v_4, t\}$

∴ The network flow across the cut is

$$f(v_1, v_3) + f(v_2, v_4) - f(v_3, v_2) = 12 + 11 - 4 = 19$$

→ The capacity of the cut (S,T) is

$$c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v)$$

$$= 12 + 14$$

$$= 26$$

Max-flow min-cut theorem

If f is a flow in a flow network $G = (V,E)$ with source s and sink t, then the following conditions are equivalent

1. f is a maximum flow in G.
2. The residual network G_f contains no augmenting path
3. $|f| = c(S,T)$ for some cut (S,T) of G.

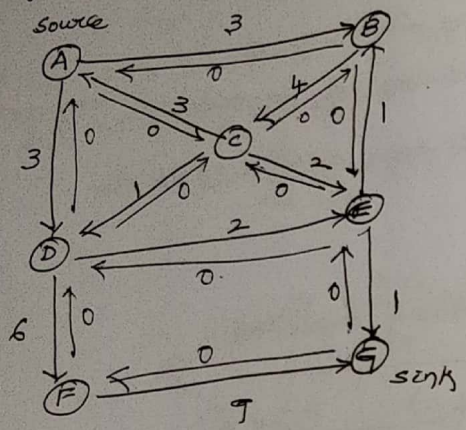
The basic Ford-Fulkerson algorithm

```
FORD-FULKERSON(G, s, t)
for each edge (u,v) ∈ G.E
    (u,v).f = 0
while there exists a path p from s to t in the residual network G_f
    c_f(p) = min{ c_f(u,v) : (u,v) is in p }
    for each edge (u,v) in p
        if (u,v) ∈ E
            (u,v).f = (u,v).f + c_f(p)
        else (v,u).f = (v,u).f - c_f(p)
```

Scanned by CamScanner

Edmonds-Karp algorithm

→ Uses BFS traversal for find an augmenting path from s to t.



Residual graph

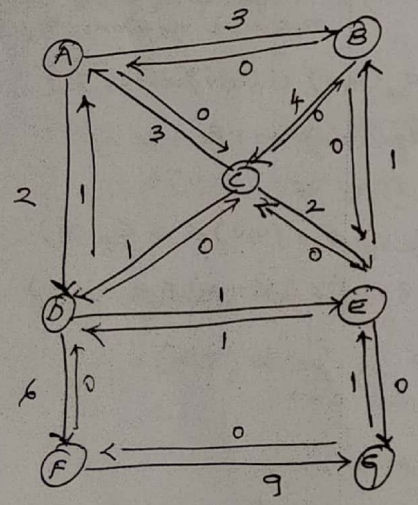
queue	parent map	Visited set
~~A~~ ~~B~~ ~~D~~ ~~E~~ ~~E~~ F	B → A D → A C → B E → D F → D G → E	A B C E

current = ~~A~~ ~~B~~ ~~D~~ ~~E~~ ~~E~~

flow = 1

maxflow = 1 +

A →(3) D →(2) E →(1) G = 1



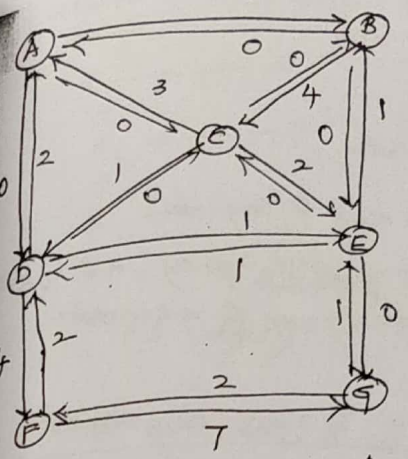
queue	parent map	Visited
~~A~~ ~~B~~ ~~D~~ ~~E~~ ~~E~~ ~~F~~ ~~G~~	B → A D → A C → B E → D F → D G → F	A B D C E F

current = ~~A~~ ~~B~~ ~~D~~ ~~E~~ ~~E~~ ~~F~~

flow = 2

maxflow = 1 + 2

A →(2) D →(6) F →(9) G = 2



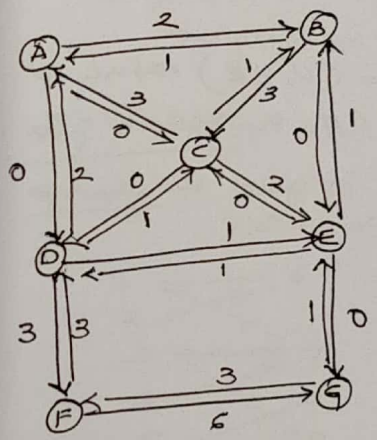
queue	parent map	Visited set
~~A~~ ~~B~~ ~~C~~ ~~D~~ ~~E~~ ~~F~~ ~~G~~	B → A C → B D → C E → C F → D G → F	A B C D E F

current = ~~A~~ ~~B~~ ~~C~~ ~~D~~ ~~E~~ F

flow = 1

maxflow = 1 + 2 + 1

A →(3) B →(4) C →(1) D →(4) F →(7) G = 1



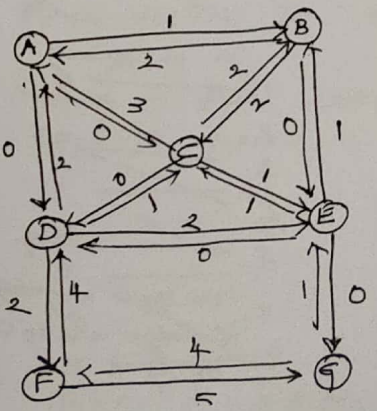
queue	parent map	Visited set
~~A~~ ~~B~~ ~~C~~ ~~D~~ ~~E~~ ~~F~~	B → A C → B E → C D → E F → D G → F	A B D E C F

current = ~~A~~ ~~B~~ ~~C~~ ~~D~~ ~~E~~ ~~F~~

flow = 1

maxflow = 1 + 2 + 1 + 1

A →(2) B →(3) C →(2) E →(1) D →(3) F →(6) G = 1



queue	parent map	Visited set
~~A~~ ~~B~~ ~~C~~ ~~E~~ ~~E~~ ~~D~~ dead end	B → A C → B E → C	A B C E

current = ~~A~~ ~~B~~ ~~C~~

flow =

maxflow = 1 + 2 + 1 + 1 = 4

no augmenting path

Bipartite Maximum matching

Eg:- J jobs C candidates

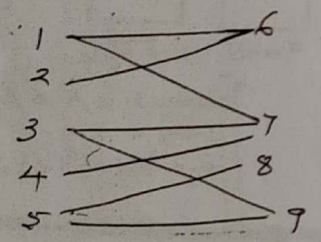
L_i : List of jobs candidate i can do

Constraint: Each candidate must be given at most 1 job. Each job must be assigned to at most 1 candidate

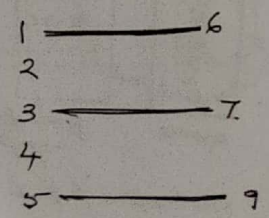
Goal :- Assign candidates to job such that maximum no. of jobs are filled.

Definition

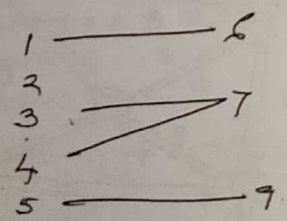
Input: Bipartite graph $G = (U, V, E)$ where U and V are vertex sets with $|U| = n_1$ and $|V| = n_2$ & E the set of edges.



Matching :- $M \subseteq E$ is said to be a matching if atmost one edge from M is incident on any vertex (in U or in V)

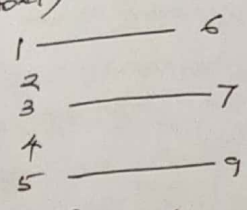


is a matching



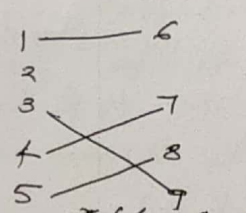
is not a matching 2 edges are incident on 7

output : Matching of maximum size (goal)



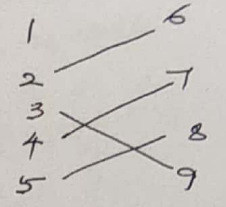
(is not max matching)

* (3 edges)



* (4 edges) is maximum matching

Note max matching are not unique



Relate to Jobs & candidates.

Candidates – 1, 2, 3, 4, 5

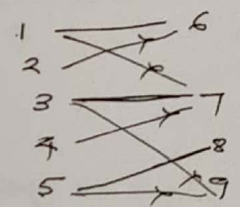
jobs – 6, 7, 8, 9

Edge(u, v) = candidate u can do job v

Matching – is just a job assignment.

An idea:-

Greedy → Keep on adding edges into M till no more edges can be added.



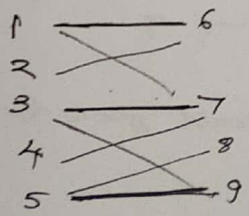
→ Have we reached the maximum matching

→ Answer is no.

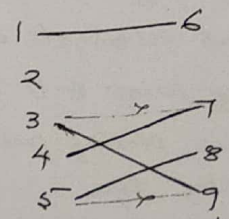
So we need more than "Greedy technique", called "Kho-Kho idea"

Free vertex for M : not an endpoint of any edge in M.

Match a free vertex. disturbs matching. Remove conflict. Repeat



Vertex 3, 1, 2 are free



Added 3 edges

Removed 3 edges

Path traced

4 – 7 – 3 – 9 – 5 – 8

alternate

Augmenting path for a matching M.

sequence P of vertices $v_1, v_2 \ldots v_k$ such that

→ $v_1 \in U$ and $v_k \in V$ are free in M.

$(v_1, v_2)\ (v_3, v_4) \ldots (v_{k-1}, v_k) \in E - M$

$(v_2, v_3)\ (v_4, v_5) \ldots (v_{k-2}, v_{k-1}) \in M$

Now, bigger matching = $M \oplus P$

↳ symmetric difference

$Q \oplus R$ = set of elements in Q, or in R but not in

∴ Augmenting path play an important role while finding the maximum matching

Edmond's algorithm :-

M = Empty matching
while there is an augmenting path P for M
$M = M \oplus P$

Output M

Berge's theorem

A matching M in a bipartite graph is maximum if only if there does not exist an augmenting path for M.

How to find augmenting path

An augmenting path starts & ends at a free vertex, & has alternate edges from M.

∴ Starting point ∈ U

If we knew starting point, we could grow the path from there.

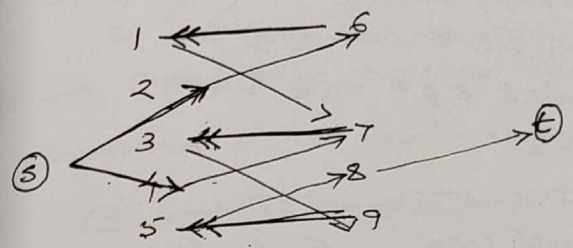
Key idea :-

Grow paths from all free vertices in U

paths grow forward using edges in E − M, and backward using edges in M.

If we reach a free node in V through any path, we are done.

BFS on auxillary graph for M



Starting vertices for path 2, 4, take a extra vertex S

Auxillary graph for G, M

$G' = (V', E')$ where $V' = \text{Union}(U, V, \{s, t\})$

$E' = E_s, E_f, E_b, E_t$ where

E_s = arcs from s to every free vertex in U

$E_f = \{(u,v) \mid u \in U, v \in V, (u,v) \in E\}$

$E_b = \{(v,u) \mid u \in U, v \in V, (u,v) \in M\}$

E_t = arcs from every free vertex in V to vertex t

Claim :- G has an augmenting path for M iff G' has a directed path from s to t

Edmonds's Algorithm

M = Empty matching
while there is an augmenting path P for M
$M = M \oplus P$

output M

Augmenting Path (G, M)

{ G' = Auxiliary graph for G, M.
P = path from s to t (use BFS)
if P = null, delete s & t and return
else return false
}

$m = |V| + |V|$, $n = |E|$ $\quad$ G - be list representation

Time (constructing G') $= O(m+n) =$ Time (of G)

Time (augmenting path) $= O(m+n)$
$=$ Time (computing $M \oplus P$)

No. of augmentations $\leq$ Matching size $\leq n/2$

Total time $= O(n(m+n)) = O(mn)$

Advanced Data structures

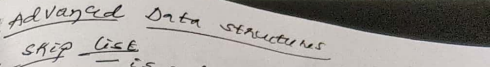
1. **Skip list** — is a randomized data structure
↓
efficiency is expressed in terms of probability rather than the expected ops.

→ William Pugh (1989), invented skip list
→ easy to implement than AVL, Red black etc.
→ maintains a dynamic set of n elements in $O(\log n)$ time per operation
→ search takes expected $\log n$ time. (with high probability.)

Considered a link list (unsorted)

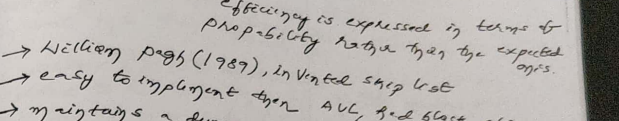
— search takes $O(n)$ time

If list is sorted



— search takes $O(n)$ time

Let's assume a sorted linked list structured in a specific way



Search(66) = 14 — 42 → 42 → 66
L_1 walk L_0

Search(x) (2 linked list)
→ walk right in top linked list (L_1) until going right would go too far.
→ walk down to the bottom list (L_0)
→ walk right in L_0 until the element is found (or not)

Scanned by CamScanner

Analysis.

$$\text{Search cost} \approx |L_1| + \frac{|L_0|}{|L_1|}$$

$|L_1|$ → cost of traversal in top list

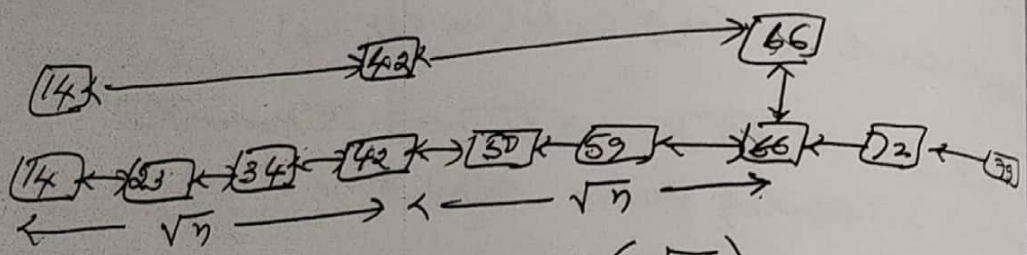
$\frac{|L_0|}{|L_1|}$ → cost of trav in bottom

minimize this: when terms are equal

$$|L_1|^2 = |L_0| = n$$

$$|L_1| = \sqrt{n}$$

→ how many elements should be in the top list.



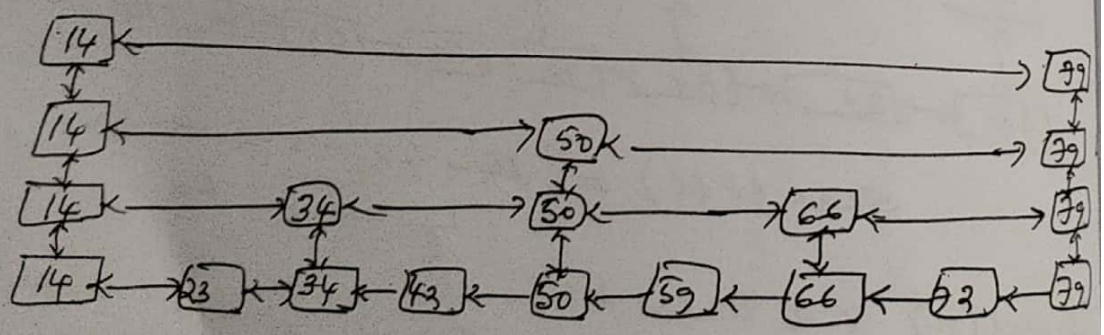
Search cost = $O(\sqrt{n})$

2 sorted list = $2\sqrt{n}$

3 sorted list = $3\sqrt[3]{n}$

k sorted list = $k\sqrt[k]{n}$

lg n sorted list = $\lg n \sqrt[\lg n]{n} = 2 \lg n$



Search(72) → pop down from L_3 to L_2, in L_2 m
from 14 to 50 pop down to L_1, in L_1
from 50 to 66, pop down to L_0, m
from 66 to 72

Red-Black Tree

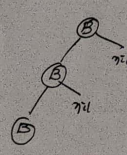
→ is a self-balancing binary search tree. as similar to AVL trees but in Red black tree no. of rotations done are less, hence the time complexity is same as of AVL tree.

→ Examples of Red-black are Treeing-up and Sorting in Java.

→ Every node in red-black tree should either be red or black.

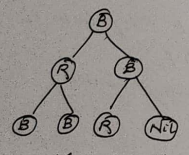
Rules for red-black tree

1. Root is always black.
2. No red-red parent child relationship
3. No. of black nodes from root to node with less than 2 children is same



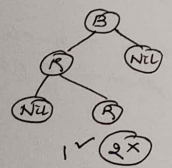
1✓ 2✓ 3✗

Not a red black tree



1✓ 2✓ 3✓

is a red black tree



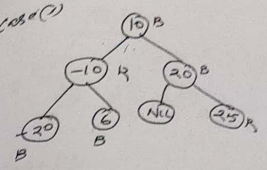
1✓ 2✗

Not a red black tree

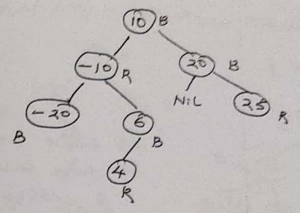
Insertion

1. If empty tree, create black root node
2. Insert new leaf node as red
   a) if its parent is black then done
   b) if its parent is red
      - if black or absent sibling rotate, recolor
      - if sibling then recolor & check

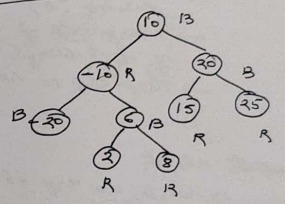
case (1)



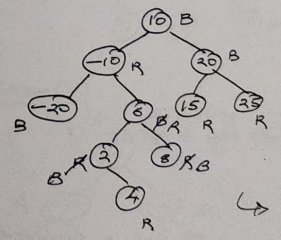
→ is a red black tree
Let's try to insert 4



→ is a red black tree (done)



Lets try to insert 4



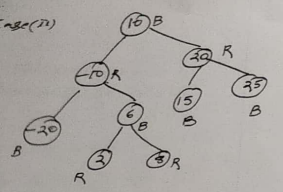
→ on inserting 4 Rule 2 is violated

→ check the color of parent's sibling, if the color are same recolor & check again

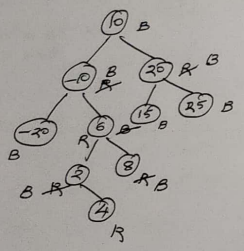
↳ hence - done needs to be continued.

Scanned by CamScanner

Case (ii)



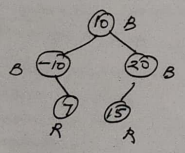
Lets insert 4



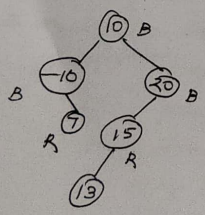
→ Rule 2 violated
→ change color of parent & sibling (2 & 8)
→ change color of its parent (6)
→ change color of parent & its sibling (-10, 20)
→ its parent is root, so

Case (iii)

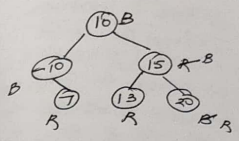
if sibling ~~is black~~ absent
↳ here we either do any of the four rotations L, R, LR, RL



Lets try to insert 13

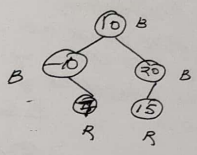


→ Rule 2 violated
→ sibling of 15 is absent (do rotation)
↓
& change

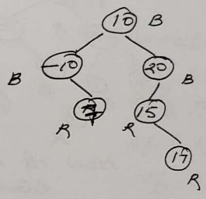


→ is a red black tree now.

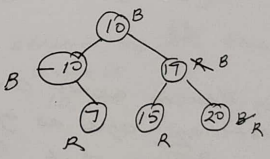
Case (iv)
rotation – left right rotation



Lets try to insert 17

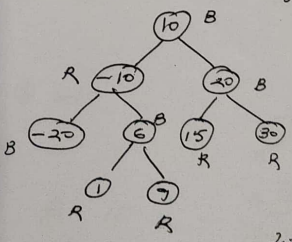


→ Rule 2 violated
→ sibling of 15 is absent (do rotation)
↓
LR
& change color



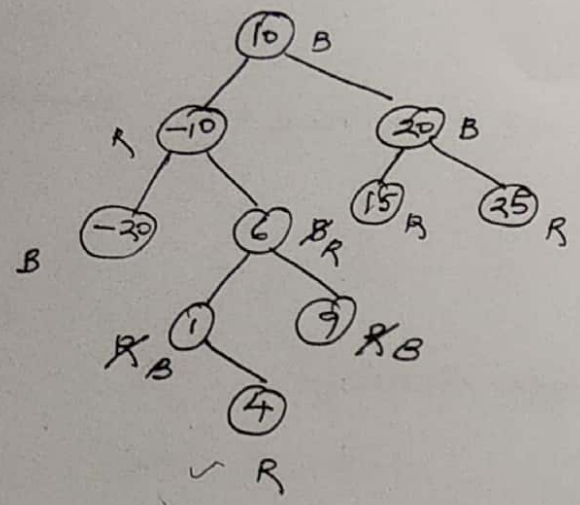
Case (v)
if sibling is black

LL → R
RL → LR
RR → L
LR → RL



Lets insert 4

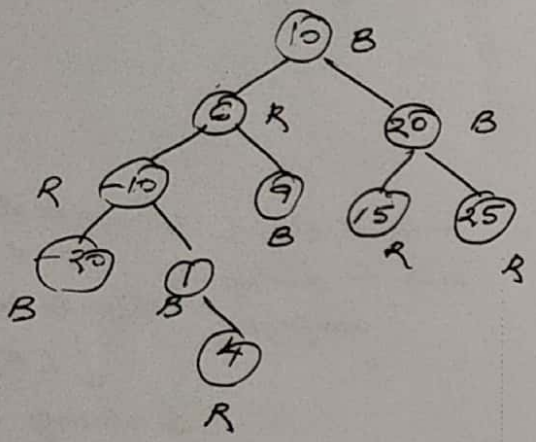
Scanned by CamScanner



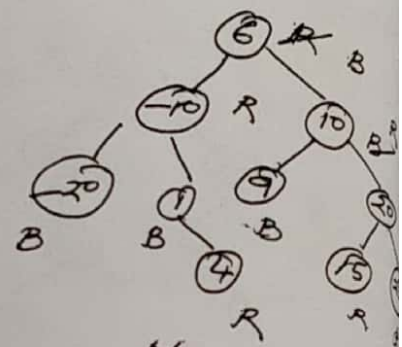
→ Rule 2 violated
→ Change the color of 1 & 9
→ Change the color of
→ Now check the color of -10 & 20, they not same
∴ we need to do rotation

conflict occurs at -10 & -10 is left child of 10
6 is right child of -10
RL → LR

↓ Left



→ right

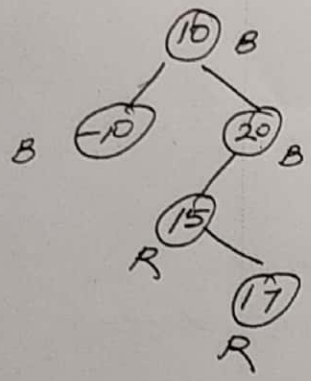


Now its a red black tree

Note that
- In Red black tree in worst case we do only 2 rotations, where as in an AVL tree we do more than that.

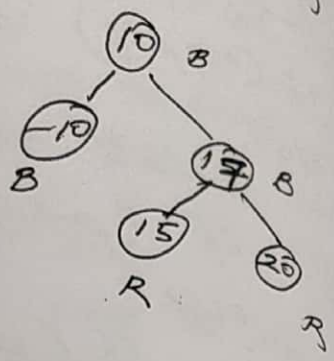
Scanned by CamScanner

17

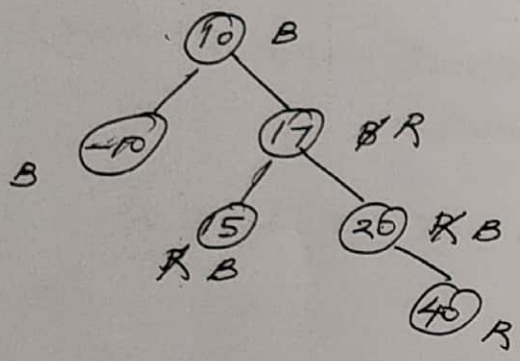


$\Longrightarrow$ LR

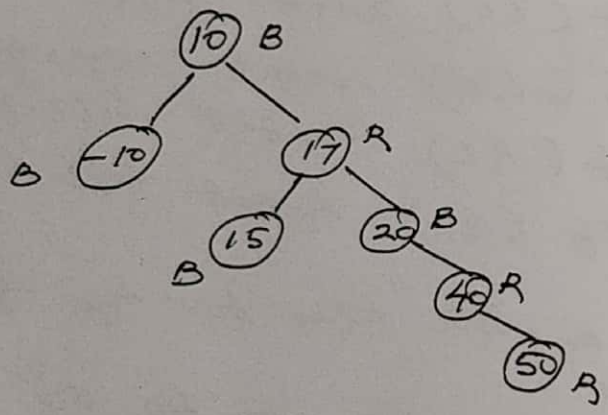
RL – log



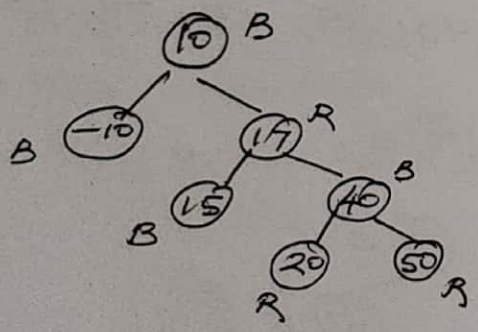
(6) 40



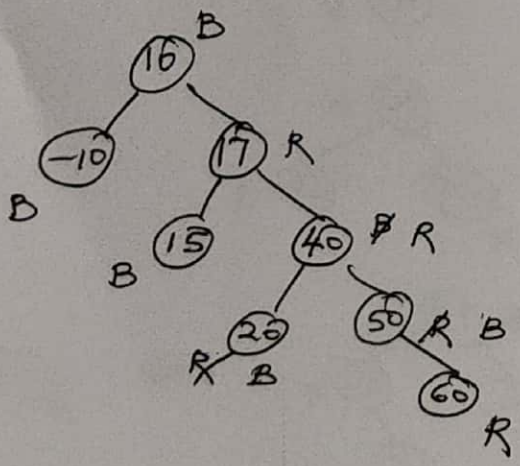
(7) 50



$\Longrightarrow$ RR → do left rotation



(8) 60



40 is right child
17 is right child
RR → do left rotation

Scanned by CamScanner

Binomial Heap & Fibonacci Heap

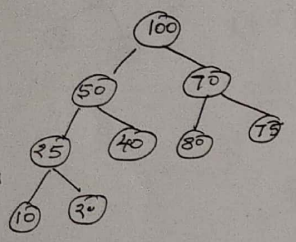
Binary Heap

– is a binary tree, which has 2 important properties i) structural property – it should be a almost complete binary tree

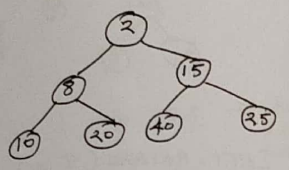
ii) parent dominance property – parent should be either equal to or more dominant compared to its children.

There are 2 types of heap

(i) Max-heap



(ii) Min-heap



Application of binary-heap

1. Heap sort
2. priority based job scheduling
3. Graph Algorithms – Shortest path (Dijkstra's algo)
4. Minimum spanning tree

operations on heap

	Time
1. Make-Heap(H)	$O(n)$
2. Insert Key x on to a heap(H, x)	$O(\log n)$
3. Maximum(H) or Minimum(H) (Find max / Find min)	$O(1)$
4. ...	$O(\log n)$

5. IncreaseKey(H, x, k) or DecreaseKey(H, x, k) $O(\log n)$

→ Difference b/w Maximum(H) & Extract-Max(H) is that in Maximum we just return the key of the root whereas in Extract-Max(H) we delete the root & heapify the tree once again.

Binomial heap & Fibonacci heap are also termed as mergeable heaps.

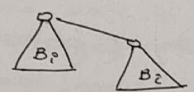
Note:- Merging 2 binary heap is more time consuming & difficult

Binomial tree of order i is defined inductively as

$B_0 \Rightarrow$ 0 (single node

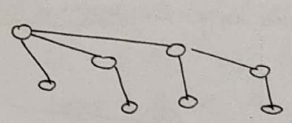
$B_1 \rightarrow$

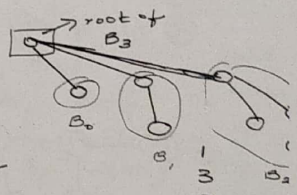
& B_{i+1} —



$B_2 \rightarrow$

$B_3 \rightarrow$





properties of binomial tree

1. # nodes in $B_i \Rightarrow 2^i$
2. # nodes in level i in B_k (root at level 0) $\Rightarrow \binom{k}{i} = {}^kC_i = \frac{k}{k-i}$

1
3
3
1

3. The subtrees rooted at the children of the root of B_k are $B_0, B_1, \ldots B_{k-1}$

* 4. For any n, there is a unique binomial queue with a total of n nodes

For Eg:- $n = 5$ (101), $Q = (B_0, B_2)$

$n = 25$ (11001), $Q = (B_0, B_3, B_4)$

Scanned by CamScanner

Binomial queue

→ is a collection of binomial trees such that for any i, there is at most one B_i in the collection.

↓ means we can't have more than one B_i in the queue

$Q = \{B_0, B_7, B_{11}\}$

No. of nodes $= 2^0 + 2^7 + 2^{11}$

Eg:- we can't have $2\ B_0$ or $2\ B_7$ or $2\ B_{11}$ in Q.

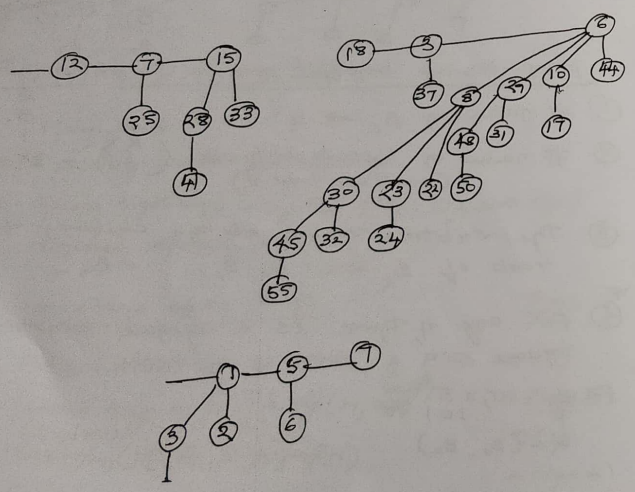
Binomial heap (Min or Max Binomial heap)

— Structural is a binomial queue with values on the nodes satisfying 'maxheap' or 'minheap' properties.

(ordered collection of binomial tree)

Note:- ① # trees in a binomial heap with n nodes $\leq \log n$

② ht. of any tree in a binomial heap on n nodes $\leq \log n$



① Findmin → $O(1)$ with hard or $O(\log n)$

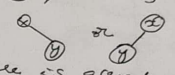
② insert (x, H) → create a B_0 with x

$O(\log n)$ in worst case

$O(1)$ amortised cost

↳ ||| (as to binary counter (increment operation))

$\{Q, \textcircled{1}, \ldots\}$



i.e. if there is already a B_0, merge to B_1

if there is already a B_1, merge to B_2

③ decreasekey (x, y, H) → Binomial heap

↳ swap values in the tree containing x as done in binary heap

$O(\log n)$ worst case

④ merge (H_1, H_2)

$H_1 = \{B_0, B_3, B_7, B_{12}\}$ — n_1 nodes

$H_2 = \{B_0, B_1, B_2, B_4, B_5\}$ - n_2 nodes

$H = H_1 \cup H_2$ and fix the violations of more than one B_i by linking them

worst case ⇒ $\log n_1 + \log n_2$

⑤ Extract-min (H)

Let B_i be the tree whose root has the min of the binary heap H

delete the root, from property (3), its children form a binomial heap $H_1 = \{B_0, B_1, \ldots B_{i-1}\}$

& merge (H_1, H) ~ $O(\log n)$

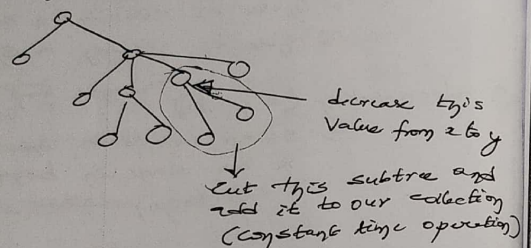
Note that → Decreasekey (H, x, y) operation takes $(\log n)$ time which is similar to the Decreasekey operation time in binary heap. The main objective

of fibonacci heap is the decrease the time (reduce) of decreasekey operation.

→ original motivation to reduce Dijkstra's time to $O(E + V\log V)$ from $O(E\log V)$

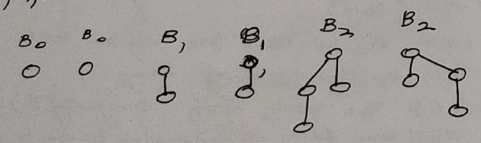
Fibonacci Heap [Fredman & Tarjan, 1986]

— no proper defination here is to reduce the time of decreasekey operation.



Make sure that each node loose atmost one child. If that happen cut the subtree rooted at its parent and add it to the collection

→ Fibonacci heap is an unordered collection of binomial tree



Note:- B_k is composed of $B_{k-1}, B_{k-2} \ldots B_0$ in unordered manner.

potential method (Amortized cost) $\phi(\cdot)$

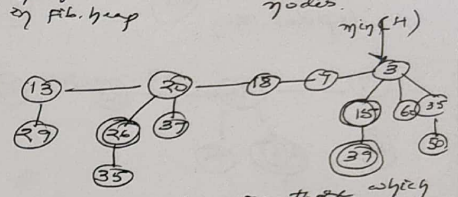
$$\phi(H) = t(H) + 2(m(H))$$

$t(H)$ → # no of trees in Fib. heap

$m(H)$ → # no of marked nodes.

Operation

(1) Insert (H, x)



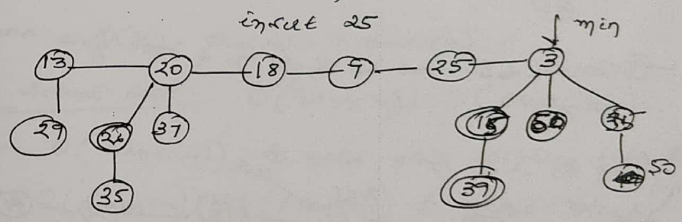
marked nodes are those which has lost its child (26, 15, 39)

$$\phi(H) = t(H) + 2(m(H))$$
$$= 5 + 2(3) = 11$$

add 25 to H

Creat a new singleton tree
Add to root list; update min pointer (if necessary)

insert 25



Actual Cost $O(1)$
Change in potential $+1$
Amortized cost $O(1)$

Notation

n = # of nodes in heap $n = 14$

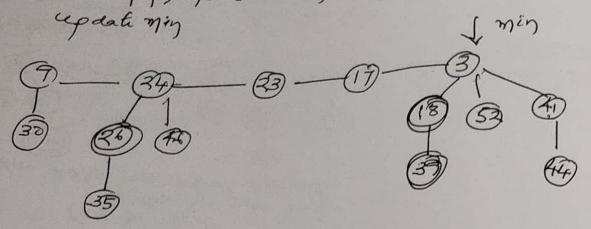
$rank(x)$ = number of children of node x

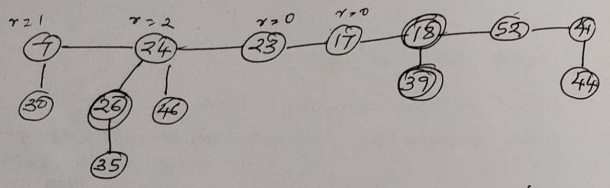
$rank(H)$ = max rank of any node in heap H $rank = 3$

$trees(H)$ = # of trees in H (5) $marks(H)$ = # of marked

Delete min (H)

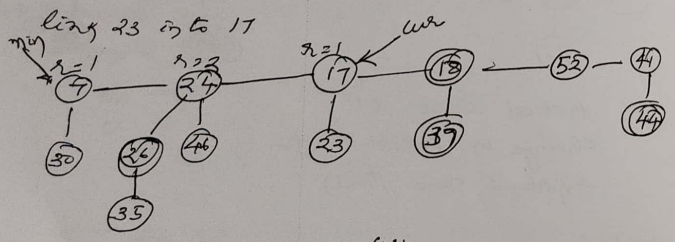
— ① Delete min; meld its children into root list; update min



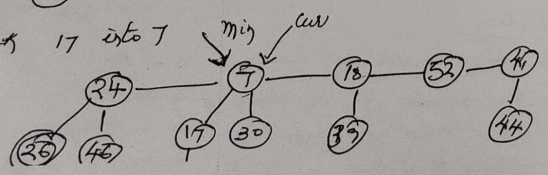


— ② Consolidate trees so that no 2 roots have same rank

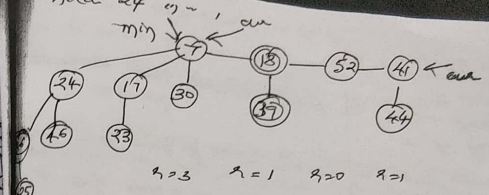
link 23 into 17



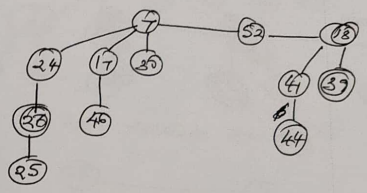
link 17 into 7



Add 24 in ~, cur



link 41 into 18



$\phi(H) = trees(H) + 2 \cdot marks(H)$

Actual cost $O(rank(H)) + O(trees(H))$

— $O(rank(H))$ to meld min's children into root list

— $O(rank(H)) + O(trees(H))$ to update min

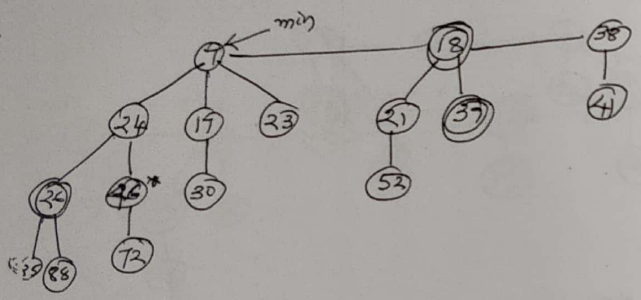
— $O(rank(H)) + O(trees(H))$ to consolidate trees.

Amortized cost $O(rank(H))$

Scanned by CamScanner

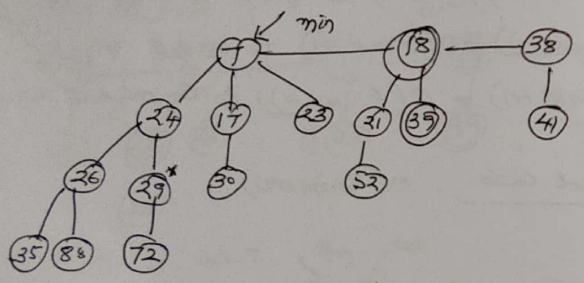
Decrease Key (x, y, H)

- if heap order is not violated, just decrease the key of x.
- otherwise, cut tree rooted at X & meld into root list
- To keep trees flat: as soon as a node has its second child cut, cut it off & meld to its root list (& unmark it)



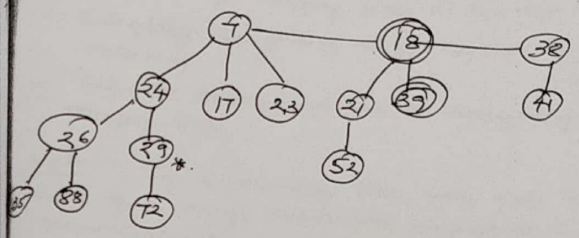
Case 1: heap order not violated

- Decrease key of X

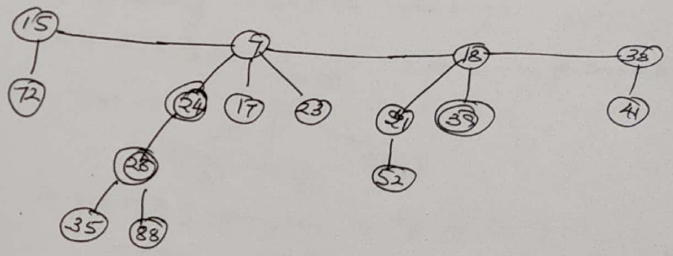


Case 2 [heap order violated]

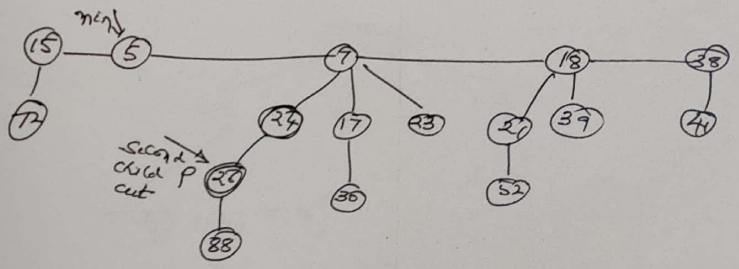
- Decrease key of X
- Cut tree rooted at X, meld into root list & unmark
- if parent p of x is unmarked (mark it) otherwise cut p meld into root list & unmark



now decrease 39 to 15



change 35 to 5



Scanned by CamScanner

Solving Recurrence

Three methods
(1) Recursion Tree method
(2) Master theorem method.
(3) substitution method.

① Recursion Tree method.

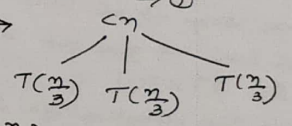
- open up the recursion diagrammatically.
- Look at the running time at the top two levels.
- Identify the pattern as you continue to open up the recursion
- When it bottoms out, do a summation to get total running time.

Note :- In a recursion tree, each node represents the cost of a single subproblem somewhere in the set of recursive function invocations.

Solve the following recurrence equations.

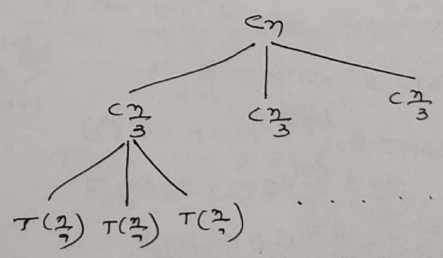
(1) $T(n) = 3T(\frac{n}{3}) + cn$ ↳ ① ; c is a +ve constant

$T(n) \rightarrow$



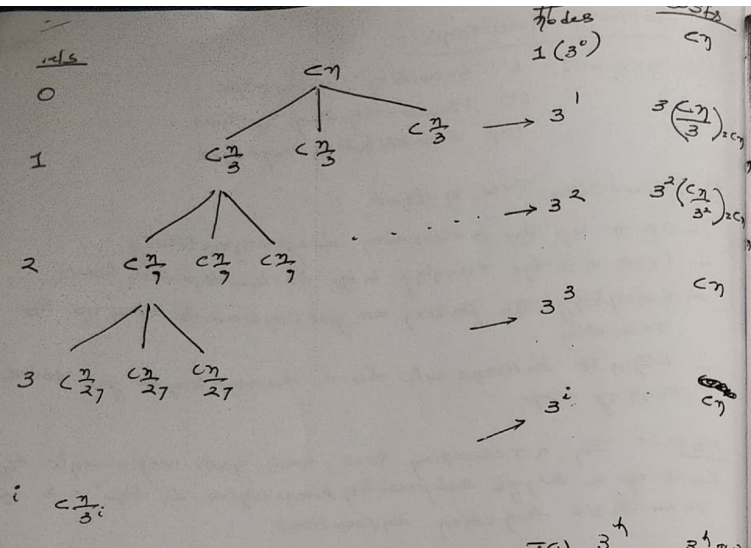
To get $T(\frac{n}{3})$ ⇒ replace n by $\frac{n}{3}$ in eq. ①

$T(\frac{n}{3}) = 3.T(\frac{n}{9}) + c.\frac{n}{3}$



$T(\frac{n}{9}) = 3.T(\frac{n}{27}) + c(\frac{n}{9})$

Scanned by CamScanner



→ Let h be the height of the

Level sums are same at all levels

At the last level we are not expanding $T(1)$, so cost will be $3^h \times T(1)$

→ If we observe the diagram the size of the problem is reducing by 3 times. Until it reaches 1.

→ At last level the size of the tree will be

$$\frac{n}{3^h} = 1 \implies 3^h = n \implies h = \log_3 n$$

$$T(n) = 3^h \times T(1) + \sum_{i=0}^{h-1} cn \qquad \sum_{i=0}^{h-1} cn = cn(h)$$

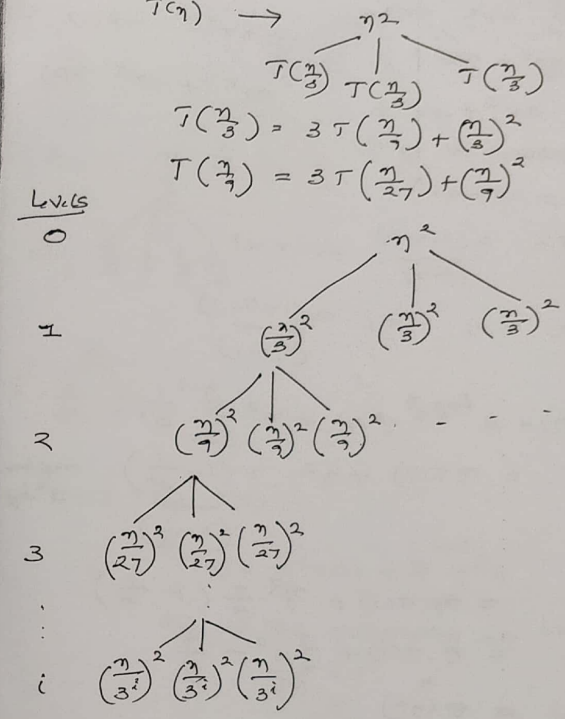
$$= 3^{\log_3 n} T(1) + cn \cdot (h)$$

$$= 3^{\log_3 n} T(1) + cn \log_3 n$$

$$= \underbrace{n\,T(1)}_{\text{lower order term}} + \underbrace{cn \log_3 n}_{\text{dominant term}}$$

$$= \Theta(n \log_3 n)$$

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



$\frac{n}{3^h} = 1$ → To get the height of the tree in terms of n

$$3^h = n$$

$$h = \log_3 n$$

$T(n)$ = sum of all level sums (replace h by $\log_3 n$)

$$= 3^h T(1) + \sum_{i=0}^{h-1} \frac{n^2}{3^i}$$

$$= 3^{\log_3 n} T(1) + n^2 \sum_{i=0}^{h-1} \frac{1}{3^i}$$

↳ geometric series

Scanned by CamScanner

Geometric series

$a, ar, ar^2, ar^3 \ldots$

$S = a + ar^2 + ar^3 + \ldots + ar^{k-1} \rightarrow (1)$

$rS = ar + ar^2 + ar^3 + \ldots ar^{k-1} + ar^k \rightarrow (2)$

Subtract (2) by (1)

$(1-r)S = a - ar^k$

$S = \frac{a(1-r^k)}{(1-r)}$ for $r < 1$

for $r > 1$ $S = \frac{a(r^k-1)}{(r-1)}$

$T(n) = 3^{\log_3 n} T(1) + n^2 \sum_{i=0}^{h-1} \frac{1}{3^i}$

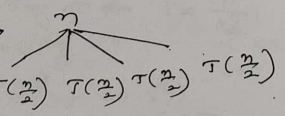
$= n T(1) + n^2 \cdot \frac{1\left(1-\frac{1}{3^h}\right)}{\left(1-\frac{1}{3}\right)}$ replace 3^h by n

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

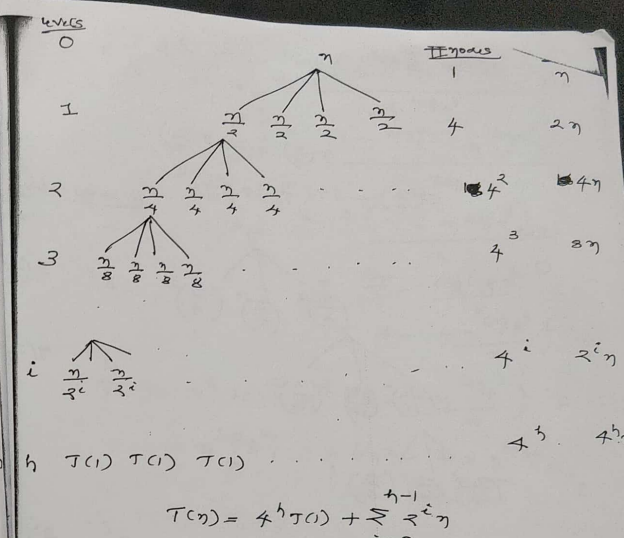
$= n \cdot T(1) + \frac{3}{2}n^2 - \frac{3n}{2}$

$= \Theta(n^2)$

(5) $T(n) = 4T\left(\frac{n}{2}\right) + n$

$T(n) \longrightarrow$ 

$T\left(\frac{n}{2}\right) = 4 \cdot T\left(\frac{n}{4}\right) + \frac{n}{2}$



$T(n) = 4^h T(1) + \sum_{i=0}^{h-1} 2^i n$

size of the subproblem at level h is

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

$2^h = n \quad \therefore h = \log_2 n$

$T(n) = 4^{\log_2 n} T(1) + n \sum_{i=0}^{h-1} 2^i$

$4^{\log_2 2} = n^{\log_2 4} = n^2$

$\therefore T(n) = n^2 \cdot T(1) + n(1 + 2 + 2^2 + \ldots 2^{h-1})$

G.P (increasing series)

$\frac{1(2^h-1)}{2-1} = 2^h - 1$

$= 2^{\log_2 n} - 1$

$= n - 1$

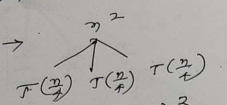
$\therefore T(n) = n^2 T(1) + n(n-1)$

$= \Theta(n^2)$

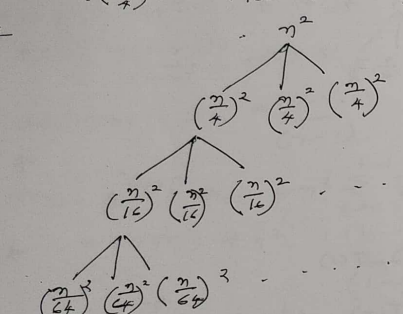
Scanned by CamScanner

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

$T(n) \rightarrow$



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



# nodes	Level size
1	1
3^1	$3\left(\frac{n}{4}\right)^2$
3^2	$9\left(\frac{n}{16}\right)^2$
3^i	$3^i\left(\frac{n}{4^i}\right)^2$
3^h	$3^h \cdot T(1)$

i $\left(\frac{n}{4^i}\right)^2 \left(\frac{n}{4^i}\right)^2$ . . . . . .

h $T(1)$ $T(1)$ $T(1)$ . . . . . .

$$T(n) = 3^h \cdot T(1) + \sum_{i=0}^{h-1} 3^i$$

$$= 3^h \cdot T(1) + n^2 \sum_{i=0}^{h-1} \left(\frac{3}{4}\right)^i$$

$\frac{n}{4^h} = 1$

$4^h = n$

$h = \log_4 n$

$$\sum_{i=0}^{h-1} \left(\frac{3}{4}\right)^i = \frac{1\left(1 - \frac{3^h}{4^h}\right)}{1 - \frac{3}{4}} = 4\left(\frac{4^h - 3^h}{4^h}\right)$$

$$= 4\left(\frac{4^h - 3^h}{4^h}\right) = \frac{4\left(4^{\log_4 n} - 3^{\log_4 n}\right)}{4^{\log_4 n}}$$

$$= \frac{4\left(n^{\log_4 4} - n^{\log_4 3}\right)}{n^{\log_4 4}}$$

$$= \frac{4\left(n - n^{\log_4 3}\right)}{n}$$

$$= \frac{4n}{n} - \frac{4n^{\log_4 3}}{n}$$

$$= 4 - \frac{4n^{\log_4 3}}{n}$$

$$T(n) = 3^h \cdot T(1) + n^2\left(4 - \frac{4n^{\log_4 3}}{n}\right)$$

$$= 3^{\log_3 n} T(1) + 4n^2 - 4 \cdot n^{\log_4 3}$$

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

Scanned by CamScanner

Master Theorem

$T(n) = a.T(\frac{n}{b}) + f(n)$

where we interpret n/b to mean either $\lfloor \frac{n}{b} \rfloor$ or $\lceil \frac{n}{b} \rceil$

then $T(n)$ has the following asymptotic bounds

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

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

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

① $T(n) = 9T(\frac{n}{3}) + n$

$a = 9 \quad b = 3 \quad f(n) = n$

$n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$

since $f(n) = O(n^{\log_3 9 - \epsilon})$ where $\epsilon = 1$, we can apply case 1 of the master theorem & conclude that the solution is

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

② $T(n) = T(\frac{2n}{3}) + 1$

$a = 1 \quad b = \frac{3}{2} \quad f(n) = 1$

$n^{\log_b a} = n^{\log_{\frac{3}{2}} 1} = n^0 = 1$

case 2 applies since $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$

& the solution is $T(n) = \Theta(1 . \log n)$

③ $T(n) = 3T(\frac{n}{4}) + n \log n$

$a = 3 \quad b = 4 \quad f(n) = n \log n$

$n^{\log_b a} = n^{\log_4 3} = \Theta(n^{0.793})$

Since $f(n) = \Omega(n^{\log_4 3 + \epsilon})$ where $\epsilon \approx 0.2$

Case 3 applies if we can show that the regularity condition for $f(n)$

we have $a f(\frac{n}{b}) = 3.f(\frac{n}{4} . \log \frac{n}{4}) \leq (\frac{3}{4}) n . \log n$
$= c f(n)$

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

$\therefore$ By case 3, the solution is $T(n) = \Theta(n \log n)$

Scanned by CamScanner