PAGE NO.:
DATE: / /

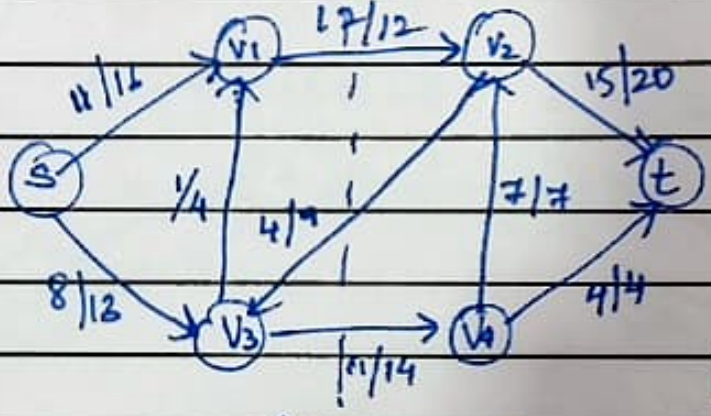
Proof to Ford fulkason Algorithm.

cuts of flow network (S, T) Cut

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

A ofaph is divided into 2 Subsets S, & T.

Ex:



|f| = 19

S = { s, v1, v3 }
T = { t, v2, v4 }

→ If f is the flow then the net flow across the cut (S, T) is denoted by f(S, T) and it can be calculated as

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

$$= 12 + 11 - 4 = 19$$

Capacity of the cut (S, T) is denoted as C(S, T)

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

Consider those capacity with Edges u → v = 12 + 14 = 26 //

1|12|21 - Aishwarya, Durgesh, Anjali, Ashwini, Harsh.

PAGE NO.:
DATE: / /

→ Max-flow min-cut theorem - proof to ford fulkson algorithm

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.

Proof through rule of transitivity
$1 \Rightarrow 2 \Rightarrow 3 \Rightarrow 1$

$1 \Rightarrow 2$

⇒ Suppose for the sake of Contradiction that f is the max flow in G but G_f has an augmenting path p. Then we can augment the bottle neck Capacity. $c_f(p) > 0$ unit flow along p & gets a better value flow which contradicts that f is the max flow.

Let See how $2 \Rightarrow 3$ i.e. $|f| = c(S,T)$

Suppose G_f has no augmenting path i.e G_f Contains no path from s to t. Define $S = \{v \in V$: there Exists a path from s to v in $G_f\}$.

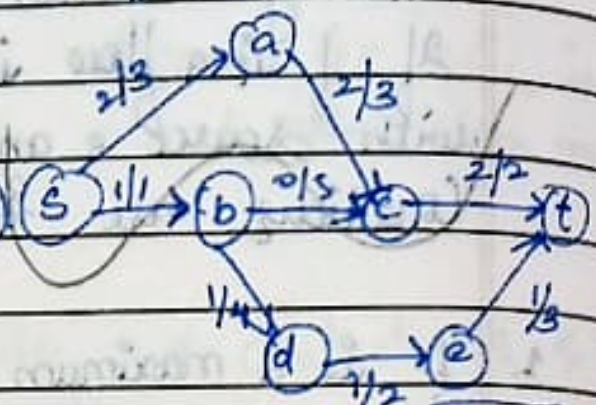
PAGE NO.:
DATE: / /

Suppose.

$S = \{s, a, c\}$ - Vertices reachable from s

$\therefore T = V - S = \{t, b, d, e\}$

Now we will have a cut (S,T)



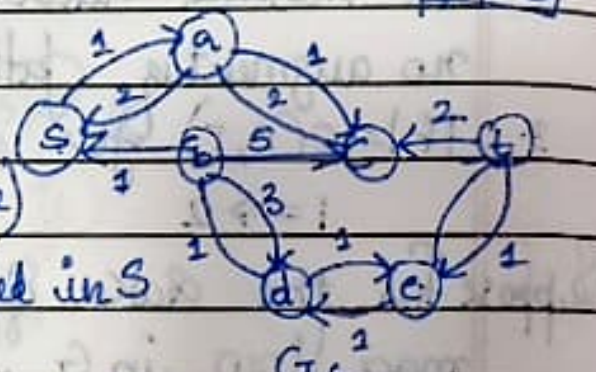
→ There will be Edges denoted by (u,v) where $u \in S$ & $v \in T$

$f(u,v) = c(u,v)$

$s \to b$, $f=c$: $1=1$; $c \to t$: $2/2$, $f=c$

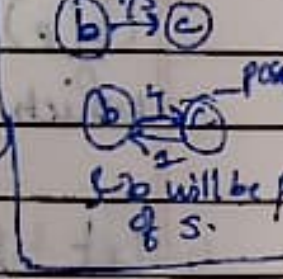
u v u v (if this is not true)

Otherwise v will be placed in S



→ There will be some edges denoted by (u,v) where $u \in T$, & $v \in S$ such that $f(u,v) = 0$.

Otherwise, we can have backward Edge. $G_f(u,v) > 0$ and v should be in S.



(∵ |f| = Summation of Capacities running from S → T).

$|f| = 3$ & $S \to t \to 3$, Equal.

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

$$= \sum_{u \in S} \sum_{v \in T} f(u,v) - 0$$

Coz. no path from t to s

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

$f(S,T) = c(S,T)$ //. $|f| = c(S,T)$ //

PAGE NO.:
DATE: / /

3 ⇒ 1

The value of any flow f in a flow network G is bounded from above by the capacity of any cut of G.

i.e $|f| \leq c(S,T)$ for all cuts (S,T)

Let (S,T) be a cut in G and f be any flow $|f| = f(S,T)$

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

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

$\Rightarrow$ the condition $|f| \leq c(S,T)$ thus implies that f is the max flow.

$\therefore |f| = c(S,T)$

1956

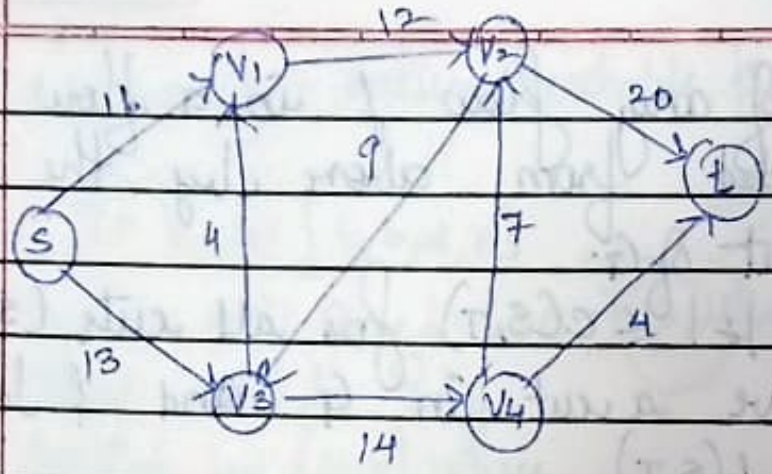
⟶ Ford - Fulkerson Algorithm

- objective - to find max flow in a flow n/w.

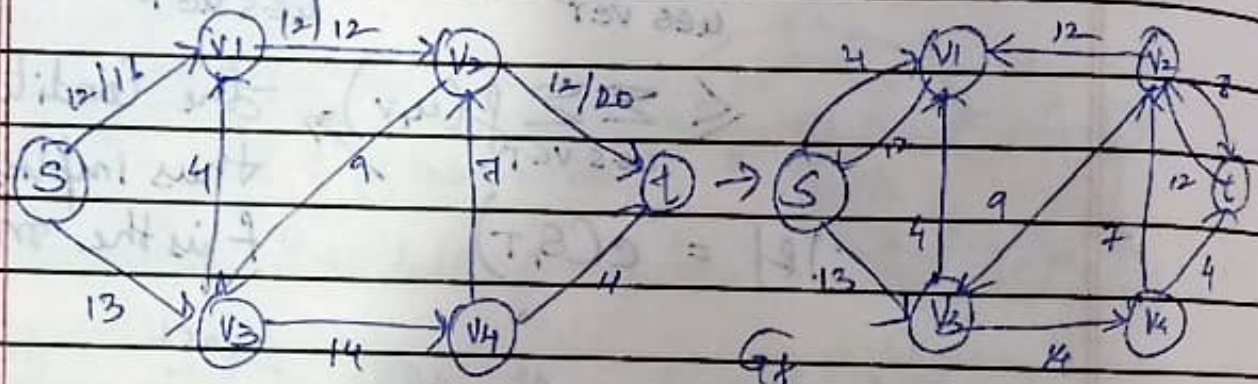
Ford - Fulkerson - Method (G, s, t).

1. Initialize flow f to 0
2. while there exist an augmenting path p in the residual network G_f.
   Augment flow f along P
3. return f — maxflow.

PAGE NO.:
DATE: / /

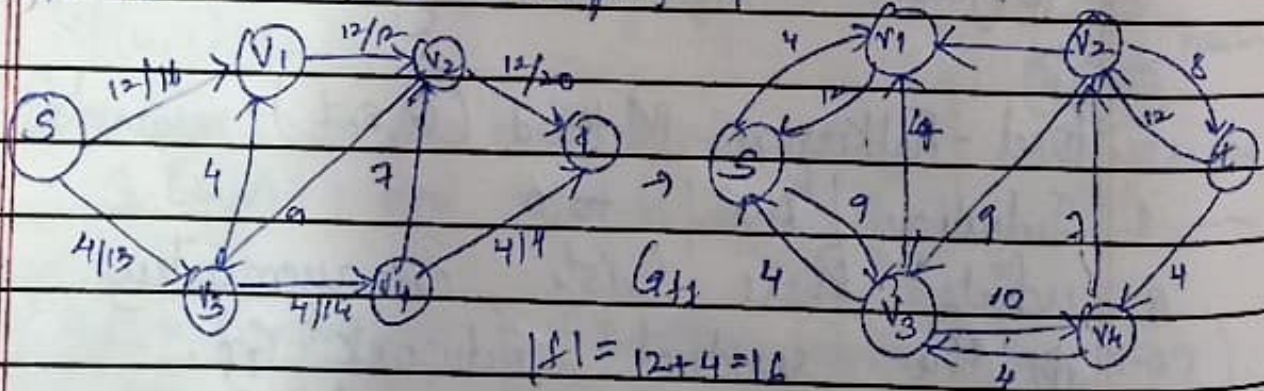


(i) $P_1 = S \to V_1 - V_2 - t = 12$ - Bottleneck Capacity-min=12.



(ii) $P_2 = S - V_3 - V_4 - t$ = min capacity = 4

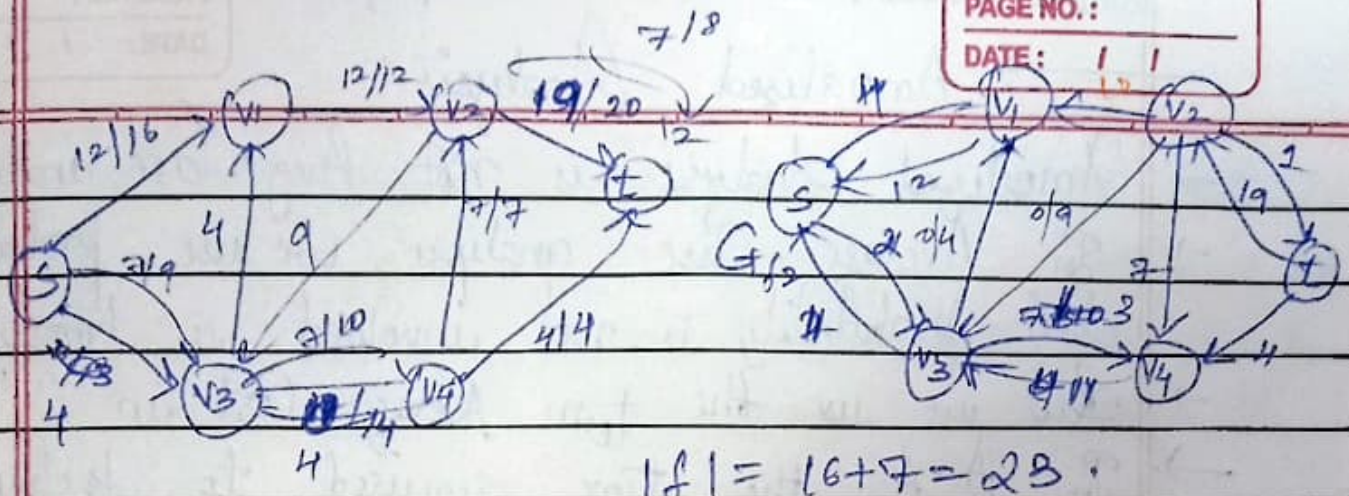
|f| = 12



|f| = 12+4 = 16

(iii) P_3

$S \to V_3 - V_4 - V_2 - t = 7$



Ford-Fulkerson (G, s, t)

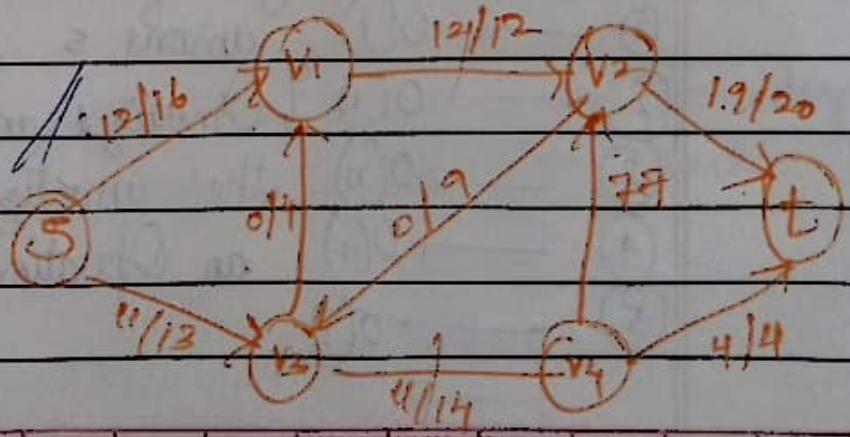
1. for each edge $(u,v) \in G.E$ — $O(E)$
2. $(u,v).f \leftarrow 0$
3. while there exists a path p from s to t in a residual network G_f — D.F.S / B.F.S — or $O(E+V) \rightarrow O(E)$.
4. $C_f(p) = \min\{C_f(u,v) : (u,v) \text{ is in } p\}$
5. for each edge (u,v) in p
6. if $(u,v) \in E$ — increase flow on all edge gradually
7. then $(u,v).f = (u,v).f + C_f(p)$
8. else $(v,u).f = (v,u).f - C_f(p)$

Calculate the max flow f. Worst case increase flow by 1 unit.

$|f| = O(E \cdot f)$

$|H| = 23$

$C(S,T) = 12+11 = 23$



$12+19+7+4 = 0$

$= 24+7 = 31$

→ We average the time required to perform a sequence of data structure operations over all the operations performed.

PAGE NO.:
DATE: / /

⟶ Amortized Analysis.

→ Amortized Analysis is not Avg Case analysis
→ In Average Case analysis, we use probability but probability is not involved in Amortized analysis
→ But we use the term Average Cost in "
→ In A.A, the time required to perform a sequence of data structure operation is average over all the operations performed
Avg. Cost →
Its not the average Case Efficiency.

Normal analysis of a particular Operation/Operations

Worst Case ① —— O(n) n is the input size
Best " ② —— O(1)
on the particular ③ —— O(n) 1 operation W.C = O(n)
data structure ④ —— O(n) n operations, it should take.
⑤ —— O(1) i.e. n × O(n) = n²

[Avg. Case = n²/n = n]

Amortised Analysis of a particular Operation/Operations

① —— O(1) among 5 only 1 Operation takes O(1).
② —— O(1) In Amortized analysis, we need to prove that
③ —— O(n) the amortized cost (avg. cost) of performing
④ —— O(1) an Operation is O(1). not O(n).
⑤ —— O(1).

→ We average the time required to perform a sequence of data structure operations over all the operations performed.

PAGE NO.:
DATE: / /

⟶ Amortized Analysis.

→ Amortized Analysis is not Avg Case analysis
→ In Average Case analysis, we use probability but probability is not involved in Amortized analysis
→ But we use the term Average Cost in "
→ In A.A, the time required to perform a sequence of data structure operation is average over all the operations performed
Avg Cost →
this not the average Case Efficiency.

Normal analysis of a particular Operation/Operations

Worst Case ① —— O(n) n is the input size
Best " ② —— O(1)
on the particular ③ —— O(n) 1 operation w.c = O(n)
data structure ④ —— O(n) n operations, it should take.
⑤ —— O(1) this n × O(n) = n²

Avg. Case = $\frac{n^2}{n} = n$

Amortised Analysis of a particular Operation/Operations

① —— O(1) among 5 only 1 Operation takes O(1).
② —— O(1) In Amortized analysis, we need to prove that
③ —— O(n) the amortized cost (avg. cost) of performing
④ —— O(1) an Operation is O(1). not O(n).
⑤ —— O(1)

PAGE NO.:
DATE: / /

→ Amortized Analysis can be used to show that the average cost of an Operation is small, if one average is over a sequence of Operations, even though a single Operation within the sequence might be expensive.

1. Aggregate Analysis
2. Accounting Method
3. Potential (Method) Function.

→ Buffay.

* Aggregate Analysis :- All Operat' have same amortized cost.

In Aggregate Analysis, we prove that all Operations have same amortized cost.

Ex :→ STACK

MULTIPO(S,K)
1. while not stack-empty(S) & K>0
2. POP(S)
3. K=K-1

Perform n Operations of push, pop and multipop.

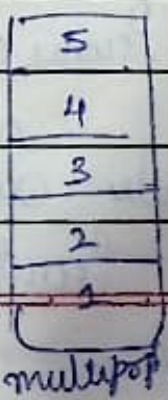
Amortized Cost of n " " " " " " " .

push - O(1) (Constant time)

pop - O(1) "

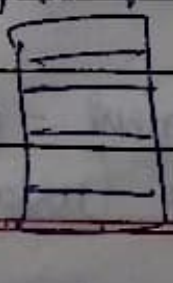
multipop depends on how many elements you want to pop from stack.

multipop(4) — no of elements to be popped



multipop (3)

multipop(3) only (2) elements popped



PAGE NO.:
DATE: / /

multipop(k) → pop k Elements if Element > k no problem
k > no. of Elements in stack.

multipop(k) = min (no. of Elements in stack, k)

Assume that. Assume that there are n Operations
of multipop.

n operations
→ push
→ multipop
→ multipop
→ pop
→ pop
→ multipop
→ push
→ multipop
→ push

Cost of multipop = how many Elements are popped
almost n

Cost of n multipop = O(n).

Cost of remaining (n-1) push & pop Operations are (n) = O(n).

Total Cost = O(n) + O(n) = O(n)

Amortized Cost = O(n)/n = O(1).

Ex 2 bit Binary Counter

2 bit Counter
0 0
1 0 1
2 1 0
3 1 1

Operations:

i) increment - increment on a Counter adds a bit 1 to the Current Value of the Counter. i.e 001101 after Increment becomes 001110

ii) decrement - decrement on a Counter Subtracts a bit 1 to the Current Value of the Counter.

PAGE NO.:
DATE: / /

i.e 001110 after decremant becomes 001101.

iii) reset - makes the Counter bits to 0s.

Worst Case Scenario of increment Operation → flipping of bit Everytime. (all elements flipping) → K bit Counter → O(K).
n Such Increment Operation → O(nk).

* Amortized Cost = O(1). (Proove).
K-bit.

0 - k-1
2 - no.
$n = \sum_{i=0}^{k-1} A[i] \cdot 2^i$
A[i] = 0 for 0, ... k-1
Increment(A)
1. i = 0
2. while i < A.length and A[i] = 1
3. A[i] = 0
4. i = i+1
5. if i < A.length
6. A[i] = 1

Counter Value	A(7)	(6)	(5)	(4)	(3)	(2)	(1)	A(0)
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	1	1
4	0	0	0	0	0	1	0	0
5	0	0	0	0	0	1	0	1
6	0	0	0	0	0	1	1	1
7	0	0	0	0	1	0	0	0
8	0	0	0	0	1	0	0	1
9	0	0	0	0	1	0	1	0
10	0	0	0	0	1	0	1	1
11	0	0	0	0	1	1	0	0
12	0	0	0	0	1	1	0	1
13	0	0	0	0	1	1	1	0
14	0	0	0	0	1	1	1	1
15	0	0	0	1	0	0	0	0
16								

A(0) flips Every time
A(1) flips $\frac{n}{2}$ times every other time.
A(2) flips every 4th time $\frac{n}{4}$ times.
In general a bit A(i) flips every 2^i times i.e $\frac{n}{2^i}$ times.
for $i = 0, 1, 2, \ldots \lfloor \log n \rfloor$ bit A(i) flips $\frac{n}{2^i}$ times.
for $i > \lfloor \log n \rfloor$ bit A(i) never flips.
$\log_2 16 = 4$ n=16.
i > 4 - 0, 1, 2, 3
4, 5, 6, 7 - 1 - 15 bits never flip.
i > 5
0, 1, 2, 3, 4

PAGE NO.:
DATE: / /

From for 'n' increment Operations the actual time is

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

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

$$< 2n = O(n)$$

Cost of 1 Operation is $\frac{O(n)}{n} = O(1)$.

→ Accounting Method — charge for particular Operation.
In accounting method, we assign different charges to different operations. The amount we charge is called as amortized cost. — ala Carte,

→ We need to put ourself as an "accountant" for each Operation we need to charge something if we charge extra that can be stored for further use.
Ex: Data given for a month, charged daily 1GB, remaing extra gets stored used for later usage. (airtel airtel)
C_i = is the actual cost. Ex: broadband, mobile
$\hat{C}_i$ = is the amortized cost

note $\sum_{i=1}^{n} \hat{C}_i \geq \sum_{i=1}^{n} C_i$ — we charge extra for storage & usage later.

(never get -ve) → Credit = $\sum_{i=1}^{n} \hat{C}_i - \sum_{i=1}^{n} C_i$

prepaid

unused amount stored ≥ 0 (should always be greater than or equal to zero)

Can be used for later operations

Later → Credit pay for Operations that are Charged less than they actually Cost.

PAGE NO.:
DATE: / /

Binary Ex:

Binary Ex	Counter Value	A(7) 6 5 4 3 2 1 A(0)	Total Cost	Charge Amt
1$	0	0 0 0 0 0 0 0 0	0_{+1}	2$ 1$
1$	1	0 0 0 0 0 0 0 1	1_{+2}	-(0-1) 2$ 2$
2$	2	0 0 0 0 0 0 1 0	3_{+1}	-(2 flips + 1 = 3$) 2$
1$	3	0 0 0 0 0 0 1 1	4_{+3}	2$ 3$
	4	0 0 0 0 0 1 0 0	7_{+1}	2$
	5	0 0 0 0 0 1 0 1	8_{+2}	2$
	6	0 0 0 0 0 1 1 0	10_{+1}	2$
	7	0 0 0 0 0 1 1 1	11_{+4}	2$
	8	0 0 0 0 1 0 0 0	15_{+1}	2$
	9	0 0 0 0 1 0 0 1	16_{+2}	2$
	10	0 0 0 0 1 0 1 0	18_{+1}	2$
	11	0 0 0 0 1 0 1 1	19_{+3}	2$
	12	0 0 0 0 1 1 0 0	22_{+1}	2$
	13	0 0 0 0 1 1 0 1	23_{+2}	2$
	14	0 0 0 0 1 1 1 0	25_{+1}	2$
	15	0 0 0 0 1 1 1 1	26+5	2$
	16	0 0 0 1 0 0 0 0	31	2$ (Same cost for each operation)

→ for each increment Operation we will assign pay 2$
1$ for flip 0 to 1 & remaining 1$ is stored.

	Avg	A	R					
0→1					0-1	2	1	1
1→0	2	1	1	← use 1-0.	1-0, 0-1	2	1+1=2	1
0→1	2.	2	1		0-1, 1-1.	2	1	2, 1+
1→0	2	1	1		1-0, 1-0, 0-1	2	1+1+1 = 3	1
1→1	2	2	1					

PAGE NO.:
DATE: / /

Ex. For Stack Example. we charge
Operations - i) Push - 2$ } Some Operations may be free
ii) Pop } $ " " " " charged
iii) Multipop.

Push - 2$ → 1$ for actual push, +1$ is stored.
Pop → 0$ (used from storage).
multipop → 0$.

		A	U	
— push	2	1	1	Cost of push is 2$ = O(1)
— push	2	1	2	
— pop	0	1	1	Cost of pop is 0$ = O(1)
— push	2	1	3	
— multipop	0	0		Cost of multipop is 0$ = O(1).

PAGE NO.:
DATE: / /

Amortized Analysis. - we average the time required to perform a sequence of data-structure operations over all the operations performed.

* Aggregate - We determine an upper bound $T(n)$ on the total cost of a sequence of n operations. The average cost per operation is then $T(n)/n$. Thus we take the average cost as the amortized cost of each operation, so that all operations have the same amortized cost.

* Accounting - We determine an amortized cost of each operation. When there is more than one type of operation, each type of operation may have a different amortized cost. The accounting method overcharges some operations early in the sequence, storing the overcharge as "prepaid credit" on specific objects in the data structure. Later in the sequence, the credit pays for the operations that are charged less than they actually cost.

* Potential - Same as the accounting method but it maintains the credit as the "potential energy" of the data structure as a whole instead of associating the credit with individual objects within data structure.

$x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$, $x = 0$, $A[i] = 0$ for $i = 0 \dots k-1$

PAGE NO.:
DATE:

Binary Counter

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

$$\hookrightarrow \sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \quad A \cdot 6$$

Increment (A)

1. $i = 0$
2. while $i < A.length$ and $A[i] == 1$
3. $A[i] = 0$ } 1 to 0
4. $i = i + 1$
5. if $i < A.length$
6. $A[i] = 1$ } 0 to 1

$i = 0$

$\rightarrow$ 3 2 1 0
0 0 0 0
1

if $0 < 3$
$A[0] = 1$

$i = 0$; $i < 3$ and $A[i] == 1$? ✗

0 0 0 0
0 0 0 1
1 0

$0 < 3$, and $0 == 1$. ✓

$A[0] = 0$
$i = 1 + 1 = 1$ | $1 < 3$ and $A[i] == 1$ ✗
$\hookrightarrow 1 < 3 \rightarrow 1 == 1$

02/02/24 → Abdullah, Anjali