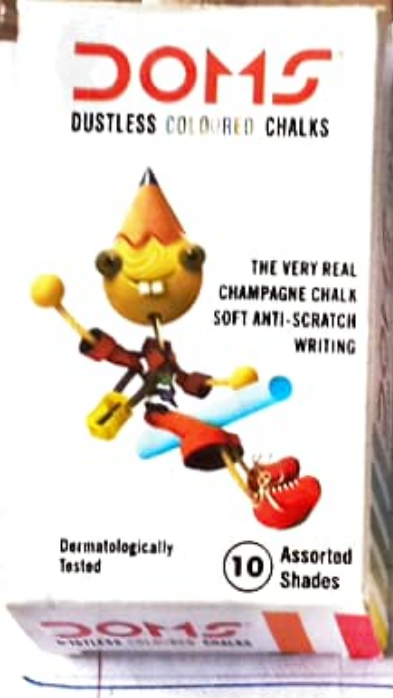


Johnson's Algorithm Uses Technique Reweighting.

The new set of Edge Weights $\hat{w}$ must satisfy two imp. properties.

1. for all pairs of vertices $u, v \in V$, a path p is a shortest path from u to v using weight function w if and only if p is also a shortest path from u to v using weight function $\hat{w}$.
2. for all Edges (u,v), the new weight $\hat{w}(u,v)$ is non-negative.

As we shall see in a moment, we can preprocess G to determine the new weight function $\hat{w}$ in $O(VE)$ time.

Preserving Shortest paths by reweighting.

Use δ to denote the shortest path weights derived from weight w and $\hat{\delta}$ to denote shortest-path weights derived from weight $\hat{w}$.

→ **Lemma** → Reweighting does not change shortest paths.

for a weighted directed Graph $G=(V,E)$ $w: E \to R$, let $h: V \to R$ be any mapping Vertices to real nos. for each Edge $(u,v) \in E$, define $\hat{w}(u,v) = w(u,v) + h(u) - h(v)$.

Let $P = \langle v_0, v_1, \ldots, v_k \rangle$ be path from Vertex v_0 to v_k. The P = shortest P from v_0 to v_k.

So $w(p) = \delta(v_0, v_k)$ iff $\hat{w}(p) = \hat{\delta}(v_0, v_k)$

p - Shortest path from v_0 - v_k.

Proof → $\hat{w}(p) = w(p) + h(v_0) - h(v_k)$

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

$= \sum_{i=1}^{k} (w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i))$ — Take out of brackets.

$= \sum_{i=1}^{k} w(v_{i-1}, v_i) + h(v_0) - h(v_k)$

$= w(p) + h(v_0) - h(v_k)$ //

$w(p) = \delta(v_0, v_k)$ iff $\hat{w}(p) = \hat{\delta}(v_0, v_k)$ — $h(v_0)$ & $h(v_k)$ do not depend on the path.

Corollary: Consider cycle $c = \langle v_0, v_1, \ldots, v_k \rangle$ where $v_0 = v_k$.

$\hat{w}(c) = w(c) + h(v_0) - h(v_k)$

$= w(c)$,

Thus -ve weight cycle using w also -ve using $\hat{w}$.

Johnson (G, w) G prime

1. 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 $v \in G.V$. } $O(V)$
2. if BELLMAN-FORD $(G', w, s) == FALSE$ } $O(VE)$
3. print "the input graph contains a -ve weighted cycle."
4. else for each vertex $v \in G'.V$ } $O(V)$
5. set $h(v)$ to the value of $\delta(s,v)$ computed by B.F. Algorithm
6. for each edge $(u,v) \in G'.E$ } $O(E)$ Reweighting
7. $\hat{w}(u,v) = w(u,v) + h(u) - h(v)$
8. let $D = (d_{uv})$ be a new $n \times n$ matrix $O(VxV) = O(V^2)$
9. for each vertex $u \in G.V$
10. run DIJKSTRA $(G, \hat{w}, u)$ to compute $\hat{\delta}(u,v)$ for all $v \in G.V$. } $O(VE \log V)$
    $O(E \log V)$ - for each vertex
11. for each vertex $v \in G.V$
12. $d_{uv} = \hat{\delta}(u,v) + h(v) - h(u)$
13. return D

— matrix with shortest paths is returned.

Analysis

3 stages in Johnson algm.

Stage 1: Computing G which takes $O(1)$ time.

Stage 2: Running Bellman ford algm on G' which takes $O(VE)$ time

Stage 3: Dijkstras algm time Complexity is $O(V+E \log V)$ which can be expressed as $O(E \log V)$. Now we need to run Dijkstras on all vertices, hence it is $V * O(E \log V)$ which will be $O(VE \log V)$ //:

Change of representation - Concept.

most efficient Approach on Sparse Graphs.

Improve performance heap. Fibonacci.

→ Solve the following recurrence relation using recursion tree method.

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

Solution -

Step :-1:

Draw a recursion tree based on the given recurrence relation.

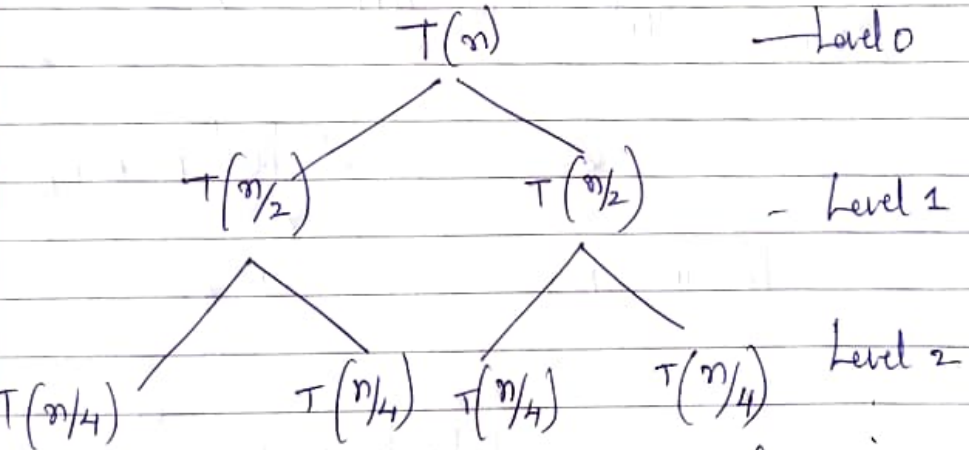
The given recurrence relation shows:

→ A problem of size n will be divided into 2 subproblems of size $n/2$.

→ Then, each sub-problem of size $n/2$ will get divided into 2 sub-problems of size $n/4$ and so on.

→ At the bottom most layer, the size of sub problems will be reduce to 1.

This is illustrated through following recursion tree -

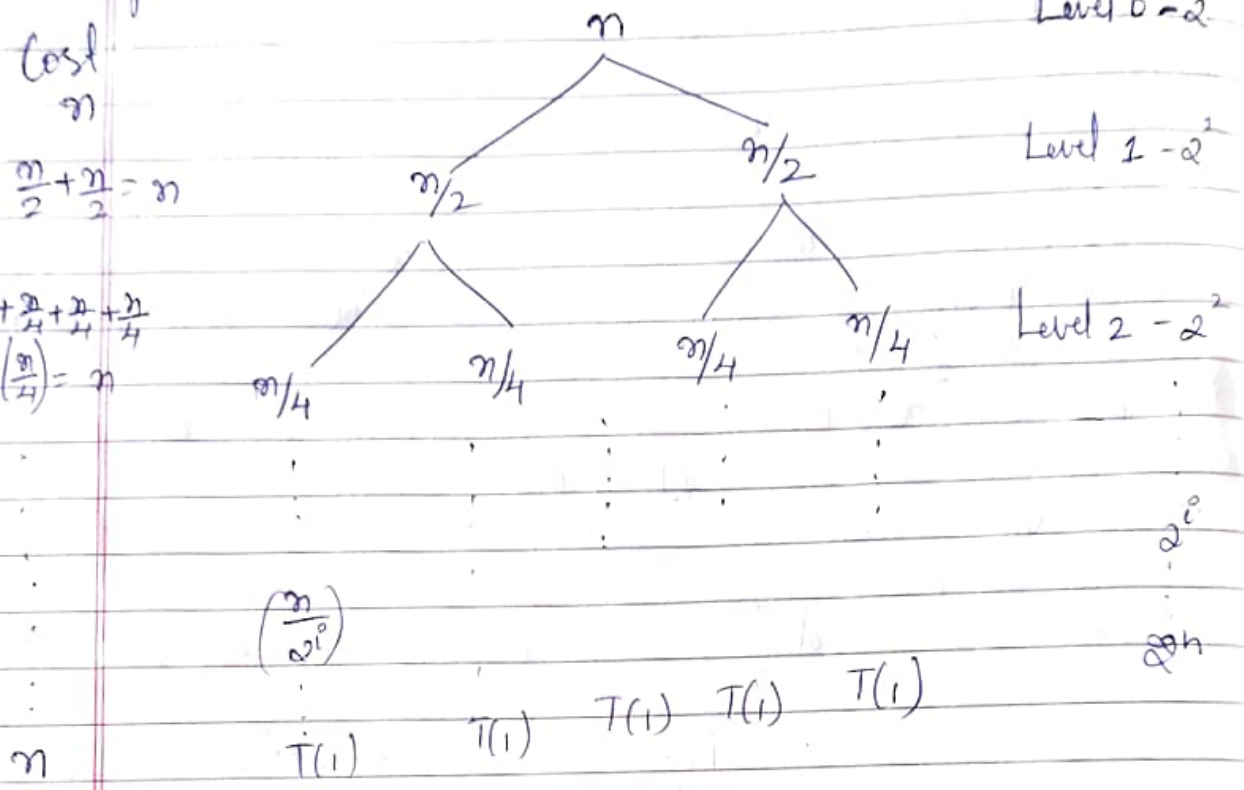


The given recurrence relation shows:

→ The cost of dividing a problem of size n into its 2 sub-problems and then combining its solution is n.

→ The cost of dividing a problem of size $n/2$ into its 2 sub-problems and then combining its solution is $n/2$ and so on.

Step 2: This is illustrated through following recursion tree where each node represents the cost of the corresponding sub-problem.



Step 3 → Size of sub-problem at level $i = n/2^i$

At level h (last level, height), size of sub-problem becomes 1. Then,

$$n/2^h = 1$$
$$2^h = n \quad \text{Take log on both sides}$$
$$\log n = \log 2^h$$
$$\log n = h \log 2$$
$$\boxed{h = \log n} \text{ //}$$

∴ Total number of levels in the recursion tree = $\boxed{\log n + 1}$ //

Step 4 → Cost of last level = $a^h \times T(1)$

$= a^{\log_2 n} \times T(1)$

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

$= n \longrightarrow \Theta(n)$. ... (1)

Step 5 → Add cost of all the levels of the recursion tree and simplify the expression obtained in terms of asymptotic notation -

$T(n) = \{ n + n + n + \cdots \} + \Theta(n)$ from (1)

For $\log_2 n$ levels (h)

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

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

By formula we get $\sum_{i=l}^{u} 1 = u - l + 1$.

$\sum_{i=0}^{h-1} 1 = (h-1) - 0 + 1$

$= h - 1 - 0 + 1$

$= h$

$= \log_2 n$

$\times \log_2 n$

$= n(\log_2 n) + \Theta(n)$

$= n \log_2 n + \Theta(n)$

↑ higher order term or upper bound.

$\therefore T(n) = \Theta(n \log_2 n)$ //

classmate
Date
Page

Potential function method.

Initially define potential function, which Captures some structural parameters.
(ex. no. of elements present in data structure).
(size of hash table while performing insertion in dynamic table)
(In Binary Counter - no. of 1's present after incrementing)
(In Stack - Every time we push, no. of elements get inserted whereas no. of elements get popped (decreased) for pop operation).

→ Let D represents some data structure, & the Operation i transforms the $D_{i-1} \rightarrow D_i$ existing particular data structure.

→ Let ϕ be potential function.
D_0 – initial data structure
$\phi(D_0) = 0$
$\phi(D_i) \geq 0$ for all i.

→ Amortized Cost $\hat{C}_i$ can be defined as.

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

$\Delta\phi_i > 0$; the Operation i stores some Energy to performing next few Operations.

IIIrd $\Delta\phi_i < 0$: the Operation i uses the stored (from previous Operation) data to deliver the Operation. (for single Operation)

Amortized Cost on performing 'n' Operations can be denoted as:-

$$\sum_{i=1}^{n} \hat{C_i} = \sum_{i=1}^{n} \left(C_i + \phi(D_i) - \phi(D_{i-1}) \right)$$

(potential Difference)

$$\sum_{i=1}^{n} \hat{C_i} = \sum_{i=1}^{n} C_i + \phi(D_n) - \phi(D_0)$$

≥ 0 ; $\phi(D_0) = 0$

$$\boxed{\sum_{i=1}^{n} \hat{C_i} \geq \sum_{i=1}^{n} C_i} \quad //$$ — in General.

→ Stack - Push, Pop, Multipop.

Structural parameter be no. of Elements

potential function.

Amortized cost: i) $\hat{C}_{push} = C_{push} + \phi_i - \phi_{i-1}$ during $i-1$ operation let x be the no. of elements

$= 1 + 1 + x - x$ i^{th} operation need elements $1+x$.

$= 2$ //

ii) $\hat{C}_{pop} = C_{pop} + \phi_i - \phi_{i-1}$ $i-1^{th}$ Operation let $(x+1)$ be the no of elements

$= 1 + x - (x+1)$ i^{th} Operation - no of elements will be x.

$= 1 + x - x - 1$

$= 0$.

→ Multipop Operation

iii) $\hat{C}_{multipop} = C_{multipop} + \phi_i - \phi_{i-1}$

$= k + n - (k+n)$ — $i-1^{th}$ Operation no. of Element be $(k+n)$

$= k + n - k - n$ — i^{th} operation no. of Element n.

$= 0$ //

Ex: Binary Counter (after i^{th} operation) Structural parameter: no. of 1's on the counter

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	0
7	0	0	0	0	0	1	1	1
8	0	0	0	0	1	0	0	0
9	0	0	0	0	1	0	0	1
10	0	0	0	0	1	0	1	0
→ 11	0	0	0	0	1	0	1	1
12	0	0	0	0	1	1	0	0
13	0	0	0	0	1	1	1	0

$x = 3$ // $x = 2$ //

Let x denote the total no. of ones before the i^{th} Operation and t denote the no. of ones after the last Zero.

$x = 3 \quad t = 2$

After i^{th} operation there will be $(x - t + 1)$ ones. ϕ_i

$3 - 2 + 1 = 2$

$\hat{C}_i = C_i + \phi_i - \phi_{i-1}$

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

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

$= 2$ //.

Note - During i^{th} iteration all ones after the last Zero is set to Zero & last Zero is set to one.

Potential method is same as accounting method. Something called prepaid is used later.

→ Different from accounting method.

→ The prepaid work not as credit, but as potential Energy, or potential ϕ.

→ The potential is associated with the data structure as a whole rather than with specific data within the data structure.

The amortized cost $\hat{C}_i$ of the i^{th} operation w.r.t potential function ϕ is defined by

$$\hat{C}_i = C_i + \phi(D_i) - \phi(D_{i-1})$$ — Data structure (stage) denoted by

i.e (actual cost + potential change).

Where $\hat{C}_i$ is amortized cost of the i^{th} operation
C_i is Actual " " " "
D_i is Data Structure
A potential function $\phi : \{D_i\} \to R$ (real nos)
$\phi(D_i)$ is called the potential of D_i.

Increment(A)

```
1  i ← 0
2  while (i < length(A) and A(i) = 1)
3        do A(i) ← 0
4           i ← i+1
5  if i < length(A)
6  then A(i) ← 1.
```

Counter Value	A[2]	A[1]	A[0]	Actual Cost	$\phi(D_i)$	$\phi(D_{i-1})$	Potential Difference	Amortized Cost
0	0	0	0	0	0	0	0+0	0
1	0	0	1	1	1	0	1-0	1+1=2
2	0	1	0	2	1	1	0	2+0=2
3	0	1	1	1	2	1	1	1+1=2
4	1	0	0					

lsb - least significant bit

Amortize Analysis: Potential of the Counter after i^{th} increment() Operation to be the number of 1's in the Counter after i^{th} op.

Therefore $\phi(D_i) = b_i$, then no. of 1's clearly, $\phi(D_i) \geq 0$.

potential i^{th} → no. of 1^{st} in Counter =

initial Counter State = 0 ∴ $D_0 = 0$, $\phi(D_i) \geq 0$.

Counter Value	2	1	0	Actual Cost	$\phi(D_i)$	$\phi(D_{i-1})$	Potential Diff	Amortized Cost
0	0	0	0	0	0	0	0	0 (0+0)
1	0	0	1	1	1	0	1	2 (1+1)
2	0	1	0	2	1	1	0	2 (2+0)
3	0	1	1	1	2	1	1	2 (1+1)
4	1	0	0	3	1	2	-1	2 (3+(-1))
5	1	0	1	1	2	1	1	2 (1+1)
6	1	1	0	2	2	2	0	2 (2+0)
7	1	1	1	1	3	2	1	2 (1+1)

Total Amortized cost 14

total amortized Cost of 7 Operations = 14

" " " " " " = 2 * 7

" " " of n " = 2*n

∴, asymptotically,

the total amortized Cost of n Operations is O(n).

∴ Thus Worst Case Cost is O(n).

Stack Operations

→ Amortized cost of each Operation is therefore its actual cost plus increase in potential due to the Operation.

The total Amortized cost of the Operation is:

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} [c_i + \phi(D_i) - \phi(D_{i-1})]$$

$$= \sum_{i=1}^{n} c_i + \phi(D_n) - \phi(D_0) \quad //..$$

Stack Operations

Goal: find Worst Case time, $T(n)$ for n Operations.

→ Push (x, s) has Complexity $O(1)$

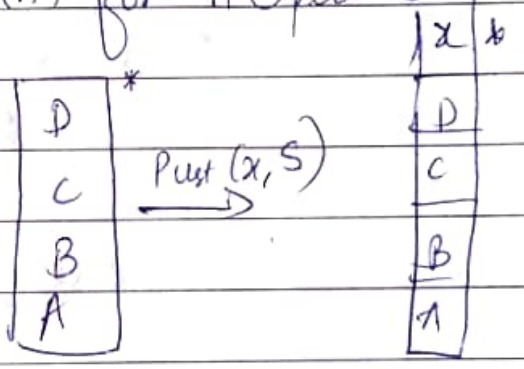
Potential = no. of objects (items) in Stack.

So $\phi(D_0) = 0$, and $\phi(D_i) \geq 0$

Potential Change: $\phi(D_i) - \phi(D_{i-1})$ (current stack, previous stack)

$= (S+1) - S = 1$.

Amortized cost: $\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1}) = 1 + 1 = 2$ // (constant) $\hat{c}_i = O(1)$.



→ Pop(s) returns popped object, Complexity is $O(1)$.

Potential Change:

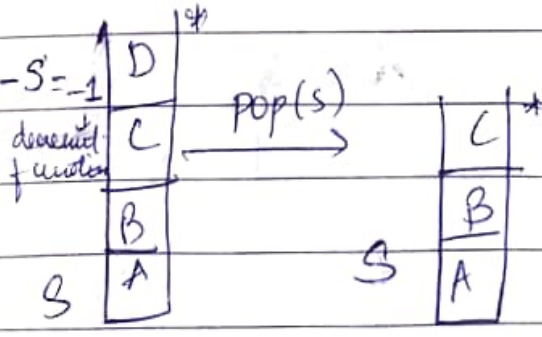
$\phi(D_i) - \phi(D_{i-1}) = (S-1) - S = -1$ (decreased function)

Amortized Cost:

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

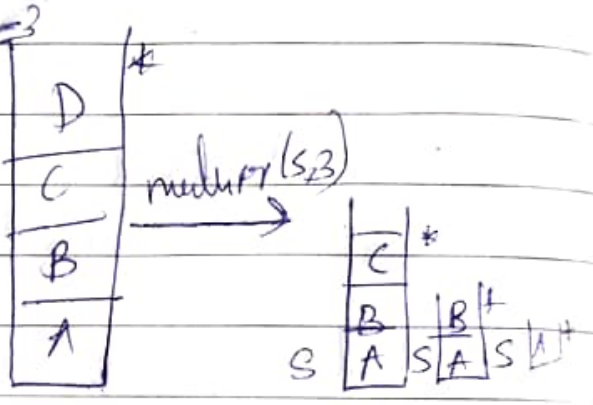
$= 1 + (-1) = 0$

$\hat{c}_i = O(1)$.



Multipop (S, k)
while not Stack-Empty (S) and k != 0
  do Pop(S)
    K ← K - 1

Complexity is min (S, k) where S is the Stack Size.
O(k).



Potential Func:
$\phi(D_i) - \phi(D_{i-1}) =$
$(s-k) - s = -k$

Amortized Cost:
$\hat{c_i} = c_i + \phi(D_i) - \phi(D_{i-1})$
$= k + (-k) = 0$
$= k + (-k) = 0$.
$\hat{c_i} = O(1)$.

Each operation $\rightarrow O(1)$
n Operations $\rightarrow O(n)$.
Worst case of n Operation is $O(n)$
$T(n) = n$.