Module - 6

Travelling Salesman problem →

The travelling salesman problem can be solved using naive method and dynamic programming. Both of the solutions are infeasible.
There is no polynomial time solution available for this problem, as the problem is a known NP-Hard problem.
There are approximate algorithms to solve the problem. The approximate algorithms work only if the problem instance satisfies the triangle-inequality.

Triangle - inequality →

The minimum distance path to reach a vertex j from i is always a direct path from i to j, rather than through some other vertex K (or other vertices)
ie. dist (i,j) is always less than or equal to dist (i,k) + dist (k,j).

When the cost function satisfies the triangle inequality, then an approximate algorithm can be designed for travelling salesman problem, that returns a tour whose cost is never more than twice the cost of an optimal tour. For this, we use the minimum spanning tree (MST)

* MST based algorithm →

1) Let 1 be the starting and ending point for the salesman.
2) Construct MST with 1 as the root, using prim's algorithm.
3) List the vertices visited in preorder traversal of the constructed MST and add 1 at the end.

Consider the following example:-

Given graph:-

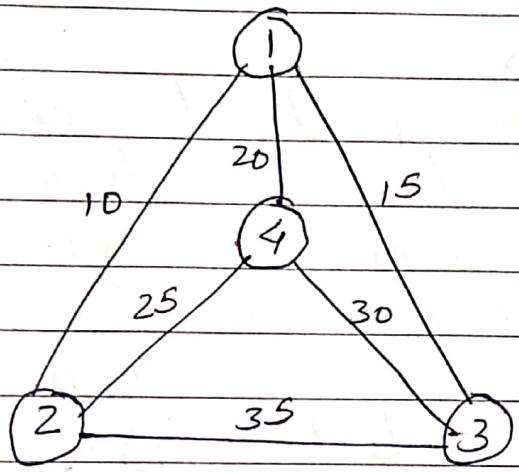


Fig ①

MST of the graph:-

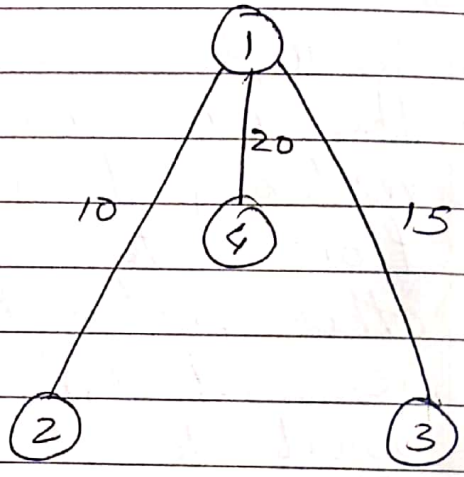


Fig ②

→ Fig ① shows the given graph
Fig ② shows the MST constructed with 1 as the root.

→ The preorder traversal of MST is 1-2-4-3. Adding 1 at the end gives 1-2-4-3-1, which is the output of

→ This algorithm is approximate algorithm. The cost of the output produced by the above algorithm is never more than twice the cost of the best possible output.

proof :→

→ Let us define a full traversal. A full traversal is a list of all vertices when they are first visited in preorder manner. It also contains the vertices when they are returned after a subtree is visited in preorder manner.

→ The full traversal of the above tree is 1-2-1-4-1-3-1.

→ Following are the important facts that proves the approximateness :-

1) The cost of best possible travelling salesman tour is never less than the cost of the MST, since MST is a minimum cost tree that connects all the vertices.

2) The total cost of full traversal is at most twice the cost of MST, since every edge of MST is visited at most in full traversal.

For example, ~~2→~~ 2-1 and 1-4 are replaced by one edge 2-4.

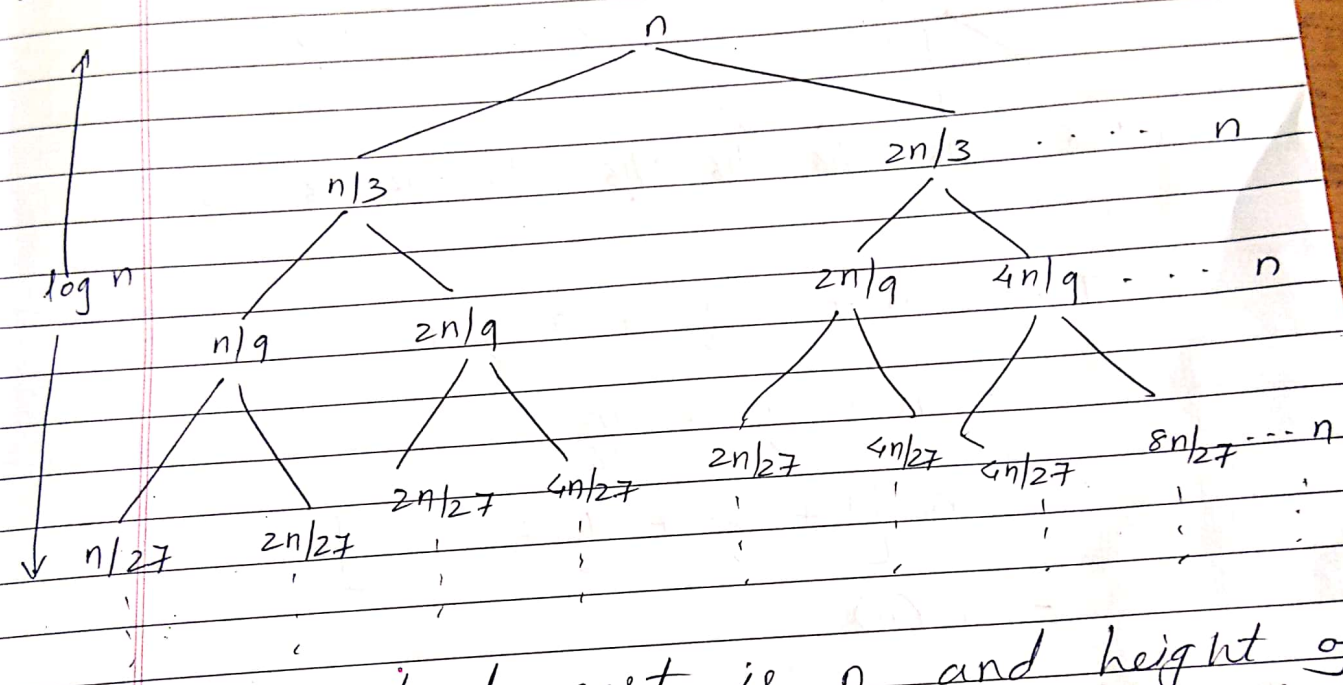
→ So if the graph follows the triangle inequality then this is always true.

→ From the above three statements, we can conclude that the cost of the output produced by the approximate algorithm is never more than twice the cost of the best possible solution.

~~Basics~~ Basics for P NP NP-complete

* Recursion Tree Examples →

May-19
1) $T(n) = T(n/3) + T(2n/3) + n$



Every level cost is n and height of the tree is log n.

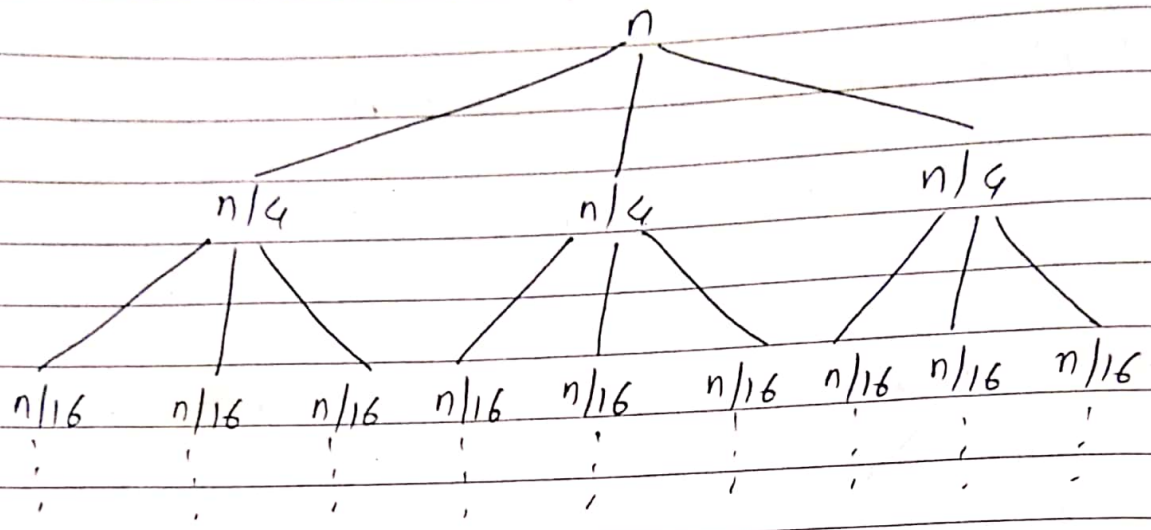
Hence,

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

Smita Gurao

Date
Page 86

2) $T(n) = 3T(n/4) + n$



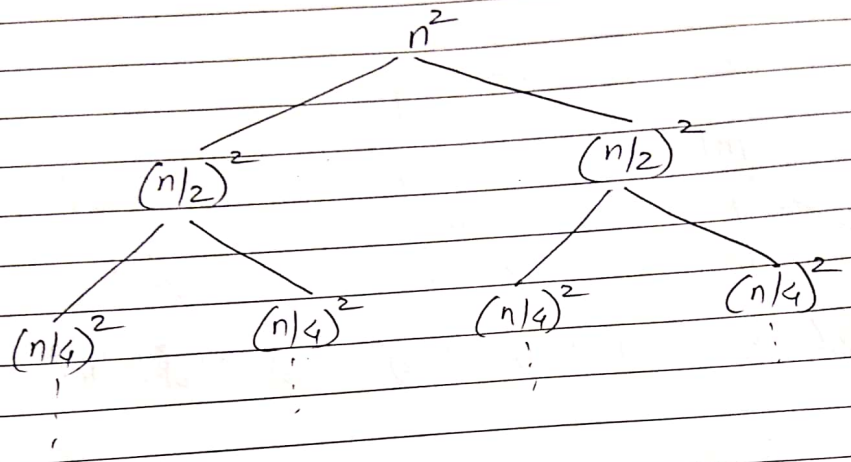
$$T(n) = n + 3 \cdot \frac{n}{4} + 9 \cdot \frac{n}{16} + \cdots$$

$$= n + \frac{3n}{4} + \frac{9n}{16} + \cdots$$

$$= n \left[1 + \frac{3}{4} + \frac{9}{16} + \cdots \right]$$

$$= O(n)$$

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



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

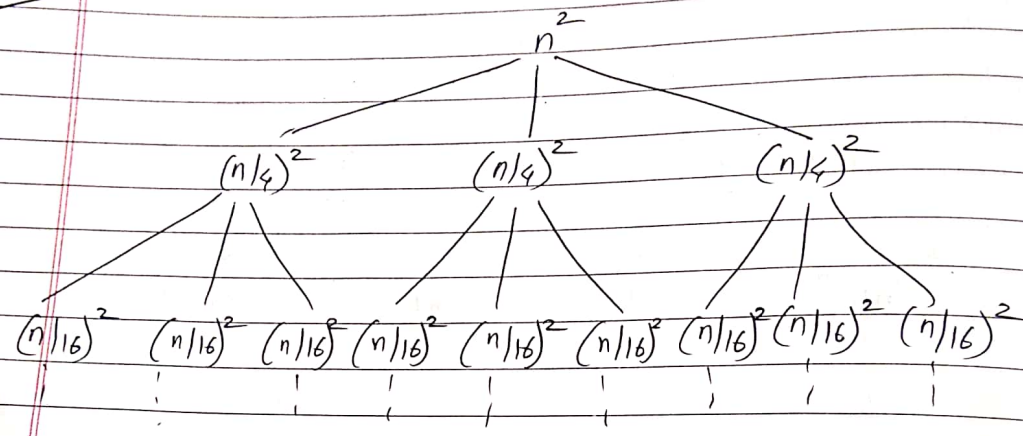
$$= n^2 + \frac{2n^2}{4} + \frac{4n^2}{16} + \cdots$$

$$= n^2 \left[1 + \frac{2}{4} + \frac{4}{16} + \cdots\right]$$

$$= n^2 \left[1 + \frac{1}{2} + \frac{1}{4} + \cdots\right]$$

$$= O(n^2)$$

4) Dec-18 $T(n) = 3T(n/4) + cn^2$



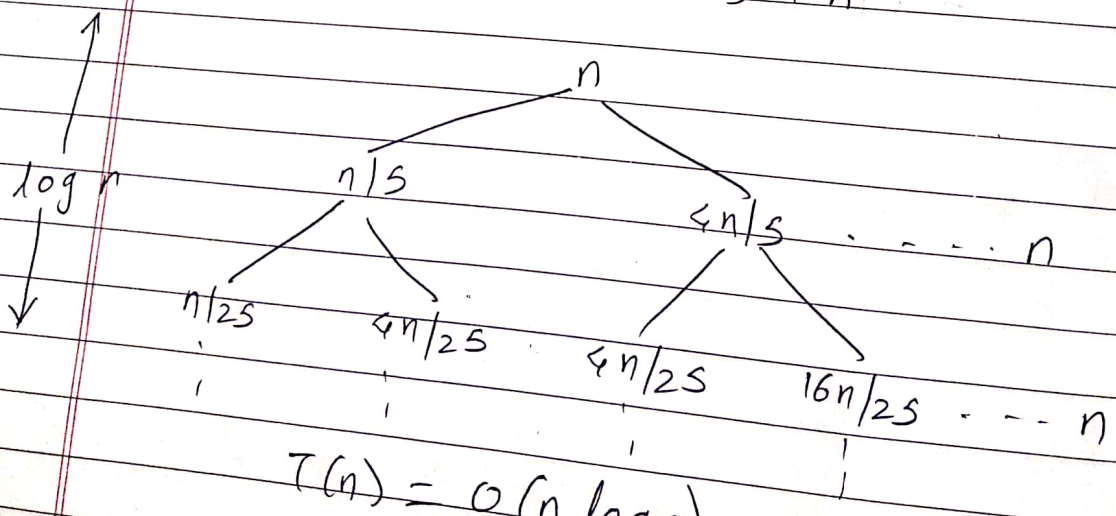
$$T(n) = n^2 + 3\cdot\left(\frac{n}{4}\right)^2 + 9\cdot\left(\frac{n}{16}\right)^2 + \cdots$$

$$= n^2 + \frac{3n^2}{4^2} + \frac{9n^2}{16^2} + \cdots$$

$$= n^2\left[1 + \frac{3}{4^2} + \frac{9}{16^2} + \cdots\right]$$

$$= O(n^2)$$

5) $T(n) = T(n/5) + T(4n/5) + n$



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

x Amortized Analysis →

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

→ Amortized analysis is used for algorithms where a single operation is very slow, but most of the other operations are faster.

→ We analyze a sequence of operations and guarantee a worst case average time which is lower than the worst case time of a particular expensive operation.

→ The amortized cost of an operation is equal to the average of the total cost of n operations. If T(n) is the total time required to perform sequence of n operations, then the amortized cost for each operation is T(n)/n.

→ Amortized analysis differs from average-case analysis. Amortized analysis does not involve probability.

eg) Hash Tables, disjoint sets and splay tree are the data structures whose operations are analysed using amortized analysis.

→ Amortized analysis is not just a tool for analysis, its a way of thinking about the design, since designing and analysis are closely related.

Intuition →
Amortized analysis is important because one bad operation shouldn't ruin a data structure if the operation is relatively uncommon.

eg. Making of a cake has 2 essential steps:-
1) Mix batter (fast)
2) Bake in oven (slow & you can only fit one cake at a time)

→ Mixing the batter takes relatively lesser time as compared to baking.
→ When we have to decide whether the cake making process is slow, medium or fast, we choose medium because we take average of the two operations (slow and fast), to get the medium result.
→ Now, if the number of cakes to be made is 100, then there are 2 operations options for how to bake 100 cakes.
1) You can mix the batter for a single cake, bake it, and repeat.
2) You can mix the batter for all 100 cakes, then bake all of them, one after the

There are 3 main types of amortized analysis:-

1) Aggregate Analysis
2) Accounting Analysis
3) potential Analysis.

The main difference in all this methods is the way in which the cost is assigned. Amortized analysis determines the average time without the use of probability.

1) Aggregate Method →
The following are the characteristics of aggregate method:-

a) It computes the worst case time $T(n)$ for a sequence of n operations.
b) The amortized cost is $T(n)/n$ per operation.
c) It gives the average performance of each operation in the worst case.
d) This method is less precise than the other methods, since all the operations are assigned the same cost.

Applications:- 1) Stack operations
2) Binary counter.

Smita Attarde

1) Stack operations →
push and pop are the operations performed on the stack.
push (s, x) - pushes element x onto the top of the stack.
pop (s) - pops and returns top of the stack.

→ Each of these operations runs in O(1) time
→ Thus, the total cost of a sequence of n push and pop operations is therefore O(n).

→ Now, consider the stack operation
Multipop (s, k)
which removes the k top elements from the stack s, or pops the entire stack if it contains less than k elements

```
Multipop (s, k)
    while stk_not_empty and k ≠ 0
        pop (s)
        k = k - 1
```

→ The worst case for Multipop is O(n). If this function is called n times, then the complexity would be $O(n^2)$.
→ This complexity is unfair, because each element can be popped only each time it is pushed.

→ In a sequence of n mixed push and pop operations, the maximum times multipop can be called is n/2.
→ Since, cost of sequence of n push and pop operations is O(n), the amortized cost of an operation is the average. That is,

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

→ Here, no probability is involved.

2) Binary Counter →
→ Consider the problem of implementing a k-bit binary counter.
→ There is an array A, which holds k-bits, so A.length = k is the counter.
→ A binary number x that is stored in the counter has its lowest-order bit in A[0] and its highest-order bit in A[k-1], so that,

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

The pseudocode for the increment function is

```
Increment
    i = 0
    while i < A.length and A[i] = 1
        A[i] = 0
        i = i + 1
```

→ A single execution of Increment takes $O(k)$ in worst case when the array A contains all 1's.
→ Thus, a sequence of n Increment operations on an initially zero counter takes $O(nk)$ in worst case.
→ But this is not always the case.
→ We can tighten this bound by showing that not all the bits flip each time.

count	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0
1	0	0	0	0	1
2	0	0	0	1	0
3	0	0	0	1	1
4	0	0	1	0	0
5	0	0	1	0	1
6	0	0	1	1	0
7	0	0	1	1	1
8	0	1	0	0	0

→ Bit A[0] flips each time (n times)
→ Bit A[1] flips every other time (n/2 times)
→ Bit A[2] flips every $n/4^{th}$ time ($n/4$ times)

⋮

In general, for $i = 0, 1, \ldots$
Bit A[i] flips every $n/2^{i}$ th time

The total number of flips in a sequence is thus,

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

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

$$= n.$$

→ If $T(n)$ is the worst case time, then
$T(n) = O(n)$

→ Thus, amortized cost of each call is -
$\frac{O(n)}{n} = O(1)$.

2) Accounting Method →
→ The ~~accor~~ accounting method is named because it borrows ideas and terms from accounting.
→ Here, each operation is assigned a charge, called the amortized cost. Some operations can be charged more or less than they actually cost.
→ If an operation's amortized cost exceeds its actual cost, we assign the difference, called as credit, to specific objects in the data structure.
→ Credit can be used later to pay for other operations whose amortized cost is less than their actual cost.
→ Credit can never be negative, in any sequence of operations.
→ The amortized cost of an operation is split between an operation's actual cost and credit that is either deposited or used up.
→ Each operation can have a different amortized cost, unlike aggregate analysis.
→ Choosing the amortized cost for each operation is important, but the costs must always be the same for a given operation, no matter what is the sequence of operations.

Applications:-

1) Stack operations →

→ Using aggregate analysis, the cost of each operation is:-

push : 1
pop : 1
Multipop : min (stack.size, k)

→ Multipop's cost will either be k, if k is less than the number of elements in the stack, or it will be the size of the stack.

→ Assigning amortized cost to those functions, we get:-

push : 2
pop : 0
Multipop : 0

→ The amortized cost for multipop is constant, while the actual cost is variable.

→ Consider, the stack as an actual stack of plates.

→ When a plate is pushed onto the stack, we pay 1$ for the actual cost of the operation and we are left with the remaining 1$ as credit. This is because we have taken the amortized cost for push as 2$

→ We will place this rupee on top of the plate which we have just placed.

→ So, at any point in time, every plate in the stack has 1$ of credit on it.

Smita Attarde

→ This 1$ placed on the top of the plate will act as the money needed to pop the plate. Thus, every plate has exactly 1$ on top of it which can be used to pop.

→ Multipop uses pop as a function. Calling multipop costs no money, but the pop function within ~~mul~~ multipop will use this 1$ on top of each plate to remove it. Since there is always $1 on top of every plate in the stack, credit is never negative.

→ performing pop or multipop doesn't make any sense until something has been pushed to the stack. So, the worst case cost of n operations is O(n).

2) Incrementing a Binary Counter →

→ ~~By~~ Accounting method shows that O(1) is the amortized cost per increment operation, and the total work has time O(n).

→ In a binary counter, the total cost (work) is never more than the total number of increment operations.

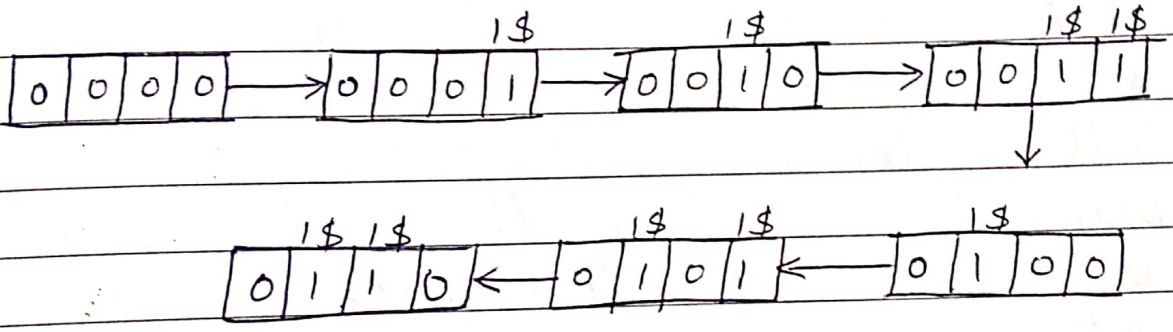
→ A charge of 2$ is assigned for each flipping of bit from 0 to 1, out of which 1$ pays for the 0→1 flip operation and the other 1$ is deposited to pay for the 1→0 flip.

→ Therefore, a sequence of n increment operations needs -

$$T(n) = 2n.$$

→ Thus, each increment operation has an amortized cost of $T(n)/n = 2n/n = O(1)$.

eg) For 4-bit counter, following are the credit invariants :-



→ charge 2$ for every flip of 0→1
[1$ pays for actual operation and 1$ remains as credit]
→ Every 1 bit has deposit 1$ to pay for flip of 1→0 later.

3) Potential Method →

→ The potential method is similar to the accounting method. However, instead of thinking about the analysis in terms of cost and credit, the potential method thinks of work already done as potential energy that can pay for later operations. This is similar to how rolling a rock up the hill creates potential energy that can then bring it back down the hill with no effort.

→ Unlike the accounting method, however, potential energy is associated with the data structure as a whole, not with the individual operations.

→ ~~The potential method works as follows:-~~

→ It starts with an initial data structure D_0.

→ Then n operations are performed, turning the initial data structure into $D_1, D_2 \ldots D_n$.

→ C_i is the cost associated with the i^{th} operation.

→ D_i is the data structure resulting from the i^{th} operation.

→ Φ is the potential function

The potential method works as follows:-
Let:-

D_0 → initial data structure
n → operations performed which turns the initial data structure into $D_1, D_2 \cdots D_n$
c_i → cost associated with the i^{th} operation.
D_i → Data structure resulting from the i^{th} operation.
Φ → potential function which maps the data structure D to a number. Thus, $\Phi(D)$ is the potential associated with that data structure.

→ The amortized cost a_i of the i^{th} operation is defined by -

$$a_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \rightarrow (1)$$

→ The amortized cost of each operation is therefore its actual cost plus the change in potential due to the operation.
→ From equation (1), the total amortized cost of n operations will be -

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} \left[c_i + \Phi(D_i) - \Phi(D_{i-1})\right]$$

Since this is a telescopic sum, we get -

$$\sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

→ To prove that the total amortized cost of n operations is an upper bound on the actual total cost, it is required that,

- $\Phi(D_i) \geq \Phi(D_0)$
- we define $\Phi(D_0) = 0$ and then show that $\Phi(D_i) \geq 0$

→ For any sequence of operations, the i^{th} operation will have a potential difference of $\Phi(D_i) - \Phi(D_{i-1})$

→ If this value is positive, then the amortized cost a_i is an overcharge for this operation and the potential energy of the data structure will increase.

→ If it is negative, it is an undercharge, and the potential energy of the data structure will decrease.

Application

Stack operations →

→ We define the potential function Φ to be the number of objects in the stack.

→ For an empty stack, $\Phi(D_0) = 0$.

→ Since the number of objects in the stack is never negative, the stack D_i that results after the i^{th} operation has non-negative potential, and thus

$$\Phi(D_i) \geq 0$$
$$= \Phi(D_0)$$

→ If the i^{th} operation on a stack containing s objects is a PUSH operation, then the potential difference is -

$$\Phi(D_i) - \Phi(D_{i-1}) = (s+1) - s$$
$$= 1$$

From equation ①, the amortized cost of PUSH operation is-

$$a_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$
$$= 1 + 1$$
$$= 2$$

→ Suppose that the i^{th} operation on the stack is Multipop (S, K), which causes $k' = \min(K, s)$ objects to be popped off the stack. The actual cost of the operation is k' and the potential difference is-

$$\Phi(D_i) - \Phi(D_{i-1}) = -k' - s = -k'$$

Thus, the amortized cost of the Multipop operation is-

$$a_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$
$$= k' - k'$$
$$= 0$$

→ Similarly, amortized cost of an ordinary pop operation is 0.

→ The amortized cost of each of the three operations is O(1), and thus the total amortized cost of a sequence of n operations is O(n).

Smita Attarde

classmate
Date
Page 104

→ Since $\Phi(D_i) \geq \Phi(D_0)$, the total amortized cost of n operations is an upper bound on the total actual cost. The worst case cost of n operations is therefore O(n).