Recurrence

Recurrences

Recurrences

- An algorithm contains a recursive call to itself, its running time can be described by a recurrence.
- Many algorithms (divide and conquer) are recursive in nature.
- When we analyze them, we get a recurrence relation for time complexity.
- We get running time as a function of n (input size) and we get the running time on inputs of smaller sizes.

Recurrences

- A recurrence is a recursive description of a function, or a description of a function in terms of itself.
- A recurrence relation recursively defines a sequence where the next term is a function of the previous terms.

Recurrences…

- Three methods to solve recurrences
  - substitution method
    - Forward Substitution
    - Backward Substitution
  - recursion-tree method
    - converts the recurrence into a tree whose nodes present the costs incurred at various levels of the recursion
  - master method
    - provides bounds for recurrences of the form
    - $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, Where $a \geq 1, b > 1$ and $f(n)$ given function

Substitution Method – Example 1

- Void Test(int n)

```
{
        if(n>0)
      {
                for(i=0;i<n;i++)
                {
                        printf("%d",i);
                }
        Test(n-1);
        }
}
```

Substitution Method – Example 1

- Void Test(int n)……………………………………………………T(n)

```
{
        if(n>0)……………………………………………………………1
    {
                for(i=0;i<n;i++)…………………………………(n+1)
                {
                        printf("%d",i);……………………….n
                }
        Test(n-1);………………………………………………T(n-1)
        }
}
```

T(n)=T(n-1)+2n+2

- Recurrence Relation:

$$T(n)=\begin{cases} 1 & n=0 \\ T(n-1)+n & n>0 \end{cases}$$

Substitution Method – Example 1

$$T(n) = \underline{T(n-1)} + n \quad \text{--- (1)}$$

- Replacing n by $n-1$ and $n-2$, we can write following equations.

$$\underline{T(n-1)} = \underline{T(n-2)} + n - 1 \quad \text{--- (2)}$$

$$\underline{T(n-2)} = T(n-3) + n - 2 \quad \text{--- (3)}$$

- Substituting equation 3 in 2 and equation 2 in 1 we have now,

$$T(n) = T(n-3) + n - 2 + n - 1 + n \quad \text{--- (4)}$$

Substitution Method – Example 1

$$T(n) = T(n-3) + n - 2 + n - 1 + n \quad \text{---} \; 4$$

- From above, we can write the general form as,

$$T(n) = T(n-k) + (n-k+1) + (n-k+2) + \ \dots + n$$

- Suppose, if we take $k \ = \ n$ then,

$$T(n) = T(n-n) + (n-n+1) + (n-n+2) + \ \dots + n$$

$$T(n) = 0 + 1 + 2 + \ \dots + n$$

$$\boxed{T(n) = \frac{n(n+1)}{2} = O(n^2)}$$

Substitution Method – Example 2

$$t(n) = \begin{cases} c1 \; if \; n = 0 \\ c2 + t(n-1) \quad o/w \end{cases}$$

- Rewrite the equation, $t(n) = c2 + \underline{t(n-1)}$
- Now, replace **n** by **n – 1** and **n − 2**

$$t(n-1) = c2 + \underline{t(n-2)}$$
$$\underline{t(n-2)} = c2 + t(n-3)$$

$$\therefore \underline{t(n-1)} = c2 + c2 + t(n-3)$$

- Substitute the values of **n – 1** and **n − 2**

$$t(n) = c2 + c2 + c2 + t(n-3)$$

- In general,

$$t(n) = kc2 + t(n-k)$$

- Suppose if we take $k = n$ then,

$$t(n) = nc2 + t(n-n) = nc2 + t(0)$$
$$\boxed{t(n) = nc2 + c1 = \boldsymbol{O(n)}}$$

Example 3

- Void Test(int n)

```
{
        if(n>0)
      {
          for(i=1;i<n;i=i*2)
                {
                        printf("%d",i);
                }
        Test(n-1);
       }
}
```

Example 3

```
void Test(int n)…………………………………………………….T(n)
{
        if(n>0) ………………………………………………………..1
     {
        for(i=1;i<n;i=i*2)
               {
                       printf("%d",i);…………………(log n)
               }
        Test(n-1);…………………………………………….T(n-1)
     }
}
```

Recurrence Relation:

$$T(n)=\begin{cases} 1 & n=0 \\ T(n-1)+\log n & n>0 \end{cases}$$

Recurrence Relation for Dividing Function

Example 4:

```
Algorithm Test(int n)
{
        if(n>1)
        {
                printf("%d",n);
                Test(n/2);
        }
}
```

Recurrence Relation for Dividing Function

```
Algorithm Test(int n)  ..................................T(n)
{
        if(n>1)  ..........................................1
        {
                printf("%d",n); .............................1
                Test(n/2);...................................T(n/2)
        }
}
```

Recurrence Relation for Dividing Function

- Recurrence Relation:

$$T(n) = \begin{cases} 1 & n = 1 \\ T(n/2) + 1 & n > 1 \end{cases}$$

Recurrence Relation for Dividing Function

- **Example 5:**
- Recurrence Relation:

$$T(n) = \begin{cases} 1 & n = 1 \\ T(n/2) + n & n > 1 \end{cases}$$

Example 6:

Find the recurrence relation for following:

```
void test(int n)
{
        if(n>0)
        {
                printf("%d",n);
                test(n-1);
        }
}
```

- **Example 7:**

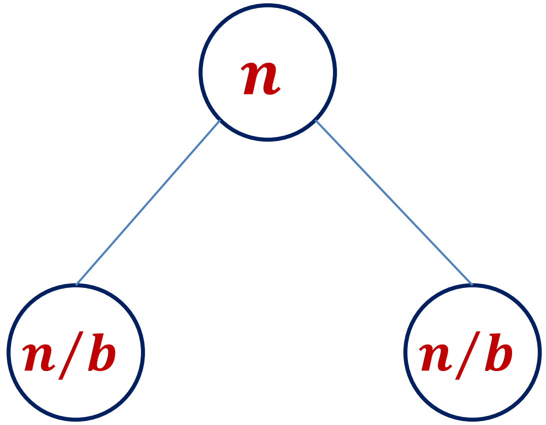
```
void test(int n)
{
        if(n>1)
        {
                for(i=0;i<n;i++)
                {
                        stmts;
                }
                test(n/2);
                test(n/2);
        }
}
```

Recursion tree method

Recursion tree method

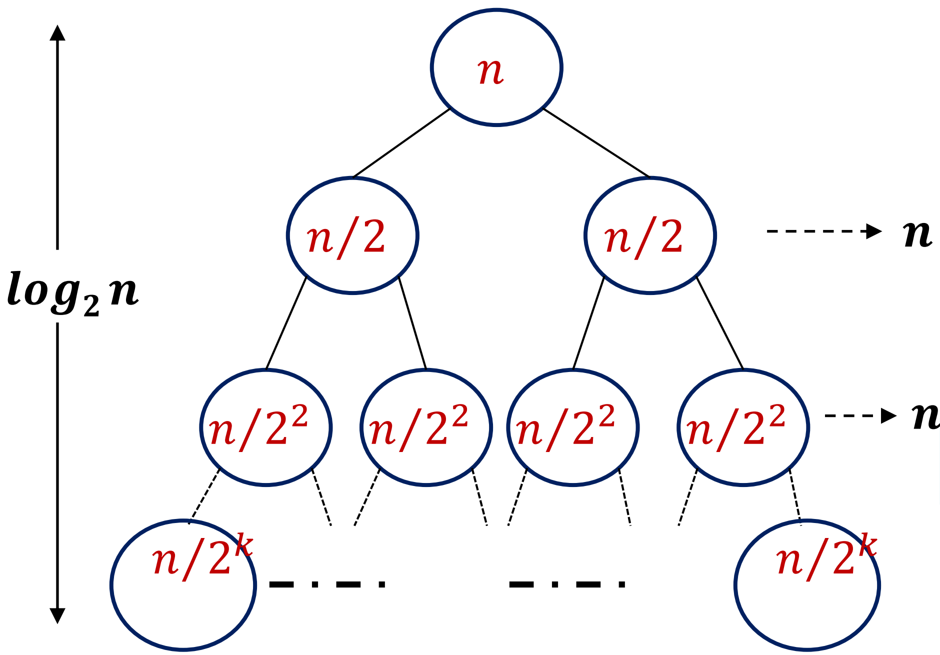
- Each node represents **the cost of a single sub problem** somewhere in the set of recursive function invocations.
- Sum the **costs within each level** of the tree to obtain a set of per-level costs.
- Sum all the **per-level costs** to determine the total cost of all levels of the recursion.
- Recursion tree useful when the recurrence describes the running time of a divide-and-conquer algorithm.
- **When we have more than one recursive term in R.H.S. in recurrence relation then we can solve that relation using recursion tree method.**
- $T(n)=\mathbf{T(n/2)+T(n/3)}+n$

- E.g., $T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$
- $F(n)$ is the cost of **splitting or combining** the sub problems.



Recurrence Tree Method

The recursion tree for this recurrence is



Example 1: $T(n) = 2T(n/2) + n$

- $n/2^k=1$
- k=logn

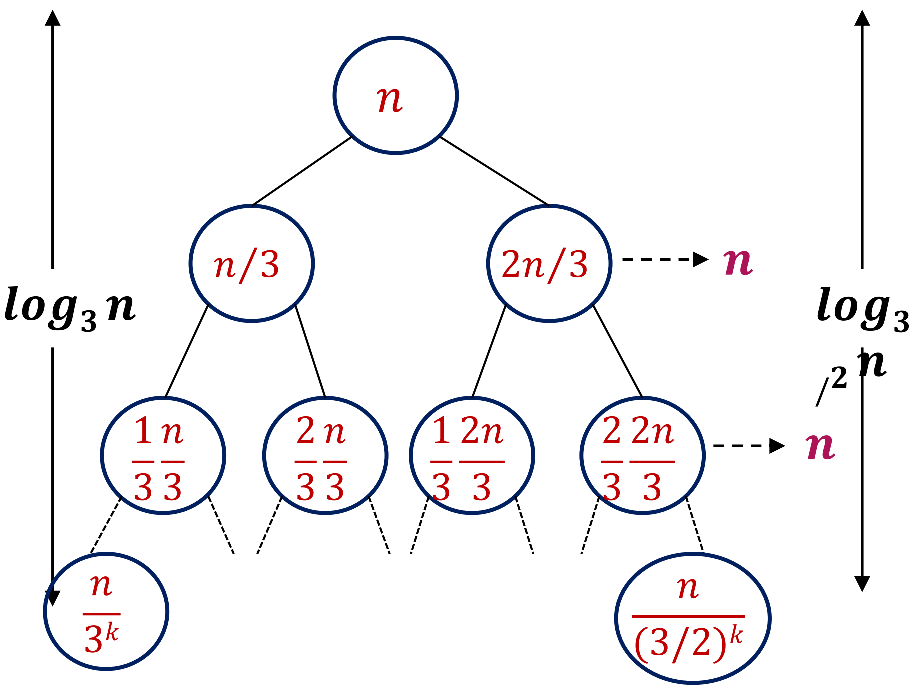
- For Running Time:
- Level 0-n
- Level 1-n
- ….
- Level k-n

Total Time=(n*k)

T(n)=(n log n)

Recurrence Tree Method

The recursion tree for this recurrence is



Example 2: $T(n) = \underline{T(n/3)} + \underline{T(2n/3)} + n$

Height of Left Subtree

$n/3^k=1$

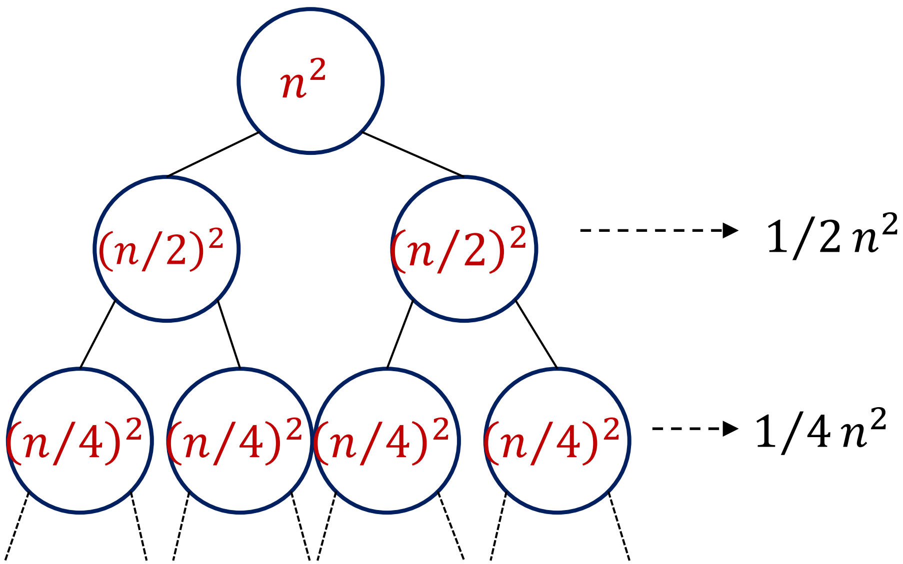
- $K=\log_3 n$

- Height of Right Subtree:
- $n/(3/2)^k = 1$
- $K=\log_{(3/2)} n$

- Computing Time=n*k
- $\boxed{T(n)= n \log_{(3/2)} n}$

Recurrence Tree Method

The recursion tree for this recurrence is



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

$$\boxed{T(n) = O\left(n^2\right)}$$

The Master Theorem

Master Theorem

- Suppose you have a recurrence of the form

$$T(n) = aT(n/b) + f(n)$$

Time to divide & recombine

Number of sub-problems

Time required to solve a sub-problem

- Where $a \geq 1$ and $b > 1$ are constants and $f(n) = \boldsymbol{\theta}(\, n^k \log^p n)$

- **This recurrence would arise in the analysis of a recursive algorithm.**

- When input size $\boldsymbol{n}$ is large, the problem is divided up into $\boldsymbol{a}$ sub-problems each of size $\boldsymbol{n/b}$. Sub-problems are solved recursively and results are recombined.

- The work to split the problem into sub-problems and recombine the results is $\boldsymbol{f(n)}$.

The Master Theorem

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

Where $\boldsymbol{a} \geq \mathbf{1}$ and $\boldsymbol{b} > \mathbf{1}$ are constants and $\boldsymbol{f}(\boldsymbol{n})$ **=$\boldsymbol{\theta}$(n^k $\log^p n$)**

- Case 1: if $\log_b a > k$ then $T(n) = \boldsymbol{\theta}(\mathbf{n}^{\log_a b})$

- Case 2: if $\log_b a = k$ then
  - If $p > -1$ then $T(n) = \theta(n^k \log^{p+1} n)$
  - If $p = -1$, then $T(n) = \theta(n^k \log\log n)$
  - If $p < -1$, then $T(n) = \theta(n^k)$

- Case 3: if $\log_b a > k$ then
  - If $p < 0$, then $T(n) = O(n^k)$
  - If $p >= 0$, then $T(n) = \theta(n^k \log^p n)$

The Master Theorem: Example #1

The Master Theorem: Example #1

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

The Master Theorem

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

- Case 1: if $\log_a b > k$ then $T(n) = \boldsymbol{\theta}(\mathbf{n}^{\log_a b})$

The Master Theorem: Example #2

- $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{logn}$
- Case 2: if log $_a$b = k then
  - If p > -1 then T(n) = θ ($n^k log^{p+1} n$)
  - If p = -1, then T(n) = θ (n^k loglogn)
  - If p < -1, then T(n) = θ (n^k)

The Master Theorem: Example #3

- $T(n) = 4T\left(\frac{n}{2}\right) + n^3$
- Case 3: if log $_a$b > k then
  - If p < 0, then T(n) = O (n^k)
  - If p >= 0, then T(n) = θ ($n^k \log^p n$)

The Master Theorem Examples

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

Changing Variable

Changing Variable

- Consider the following recurrence
- $T(n) = 2T\sqrt{n} + \log n$

Changing Variable

- Consider the following recurrence
- $T(n) = 2T(\sqrt{n}) + \log n$
  - Take, $m = \log n$

Changing Variable

- Consider the following recurrence
- $T(n) = 2T(\lfloor\sqrt{n}\rfloor) + \log n$
  - Take, $m = \log n$
  - n=2^m
  - $T(2^m) = 2T\left(2^{\frac{m}{2}}\right) + m$

Changing Variable

- Consider the following recurrence
- $T(n) = 2T\left(\lfloor\sqrt{n}\rfloor\right) + \log n$
  - Take, $m = \log n$
  - $T(2^m) = 2T\left(2^{\frac{m}{2}}\right) + m$
  - Still we can not apply master theorem.
  - $S(m) = T(2^m)$ to produce the new recurrence

Changing Variable

- Consider the following recurrence
- $T(n) = 2T\left(\lfloor\sqrt{n}\rfloor\right) + \log n$
  - Take, $m = \log n$
  - $T(2^m) = 2T\left(2^{\frac{m}{2}}\right) + m$

$$S(m) = 2S\left(\frac{m}{2}\right) + m$$

Changing Variable

- Consider the following recurrence
- $T(n) = 2T(\lfloor\sqrt{n}\rfloor) + \log n$
  - Take, $m = \log n$
  - $T(2^m) = 2T\left(2^{\frac{m}{2}}\right) + m$
  - Now, we can rename $S(m) = T(2^m)$ to produce the new recurrence

$$S(m) = 2S\left(\frac{m}{2}\right) + m$$

Solution for $S(m) =?$

Changing Variable

- Consider the following recurrence
- $T(n) = 2T(\lfloor\sqrt{n}\rfloor) + \log n$
  - Take, $m = \log n$
  - $T(2^m) = 2T\left(2^{\frac{m}{2}}\right) + m$
  - Now, we can rename $S(m) = T(2^m)$ to produce the new recurrence

$$S(m) = 2S\left(\frac{m}{2}\right) + m$$

Solution for $S(m) = O(m \log m)$

Changing Variable

- Consider the following recurrence
- $T(n) = 2T\left(\left\lfloor\sqrt{n}\right\rfloor\right) + \log n$
  - Take, $m = \log n$
  - $T(2^m) = 2T\left(2^{\frac{m}{2}}\right) + m$
  - Now, we can rename $S(m) = T(2^m)$ to produce the new recurrence

$S(m) = 2S\left(\frac{m}{2}\right) + m$

Solution for $S(m) = O(m \log m)$

Changing back from $S(m)$ to $T(n)$ then $T(n) =$?

Changing Variable

- Consider the following recurrence
- $T(n) = 2T\left(\left\lfloor\sqrt{n}\right\rfloor\right) + \log n$
  - Take, $m = \log n$
  - $T(2^m) = 2T\left(2^{\frac{m}{2}}\right) + m$
  - Now, we can rename $S(m) = T(2^m)$ to produce the new recurrence

$S(m) = 2S\left(\frac{m}{2}\right) + m$

Solution for $S(m) = O(m \log m)$

Changing back from $S(m)$ to $T(n)$

$T(n) = T(2^m) = S(m) = O(m \log m) = O(\log n \log \log n)$

AMORTIZED ANALYSIS

Introduction

- Amortized analysis considers not just one operation, but a sequence of operations on a given data structure or a database.

- Amortized Analysis is used for algorithms where an occasional operation is very slow, but most of the other operations are faster.

- The time required to perform a sequence of data structure operations is averaged over all operations performed.

- In Amortized Analysis, 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.

- So, Amortized analysis can be used to show that the average cost of an operation is small even though a single operation might be expensive.

Amortized Analysis Techniques

- There are three most common techniques of amortized analysis,
  1. The aggregate method
     - A sequence of n operation takes worst case time $T(n)$
     - Amortized cost per operation is $T(n)/n$
  2. The accounting method
     - Assign each type of operation an (different) amortized cost
     - Overcharge some operations
     - Store the overcharge as credit on specific objects
     - Then use the credit for compensation for some later operations
  3. The potential method
     - Same as accounting method
     - But store the credit as "potential energy" and as a whole.

Amortized Analysis - Example

Counter value	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]	Increment cost	Total cost
0	0	0	0	0	0	0	0	0	———	0
1	0	0	0	0	0	0	0	$\boxed{1}$	1	1
2	0	0	0	0	0	0	$\boxed{1}$	$\boxed{0}$	2	3
3	0	0	0	0	0	0	1	$\boxed{1}$	1	4
4	0	0	0	0	0	$\boxed{1}$	$\boxed{0}$	$\boxed{0}$	3	7
5	0	0	0	0	0	1	0	$\boxed{1}$	1	8
6	0	0	0	0	0	1	$\boxed{1}$	$\boxed{0}$	2	10
7	0	0	0	0	0	1	1	$\boxed{1}$	1	11
8	0	0	0	0	$\boxed{1}$	$\boxed{0}$	$\boxed{0}$	$\boxed{0}$	4	15
9	0	0	0	0	1	0	0	$\boxed{1}$	1	16
10	0	0	0	0	1	0	$\boxed{1}$	$\boxed{0}$	2	18
11	0	0	0	0	1	0	1	$\boxed{1}$	1	19

+

Incrementing a Binary Counter

- Implementing a k-bit binary counter that counts upward from $0\ to\ n$.
- Use array $A[0 \ldots k-1]$ of bits as the counter where,

$$\boldsymbol{length[A] = k}$$

- $\mathrm{A}[0]$ is the least significant bit.
- $\mathrm{A}[k-1]$ is the most significant bit.

Amortized Analysis - Example

Counter value	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]	Increment cost	Total cost
0	0	0	0	0	0	0	0	0	——	0
1	0	0	0	0	0	0	0	$\boxed{1}$	1	1
2	0	0	0	0	0	0	$\boxed{1}$	$\boxed{0}$	2	3
3	0	0	0	0	0	0	1	$\boxed{1}$	1	4
4	0	0	0	0	0	$\boxed{1}$	$\boxed{0}$	$\boxed{0}$	3	7
5	0	0	0	0	0	1	0	$\boxed{1}$	1	8
6	0	0	0	0	0	1	$\boxed{1}$	$\boxed{0}$	2	10
7	0	0	0	0	0	1	1	$\boxed{1}$	1	11
8	0	0	0	0	$\boxed{1}$	$\boxed{0}$	$\boxed{0}$	$\boxed{0}$	4	15
9	0	0	0	0	1	0	0	$\boxed{1}$	1	16
10	0	0	0	0	1	0	$\boxed{1}$	$\boxed{0}$	2	18
11	0	0	0	0	1	0	1	$\boxed{1}$	1	19

Aggregate Method

- The running time of an increment operation is proportional to the **number of bits** flipped.
- However, **all bits are not flipped** at each INCREMENT.
- $A[0]$ flips ***at each*** increment operation;
- $A[1]$ flips ***at alternate*** increment operations;
- $A[2]$ flips ***only once for 4 successive*** increment operations;
- In general, bit $A[i]$ **flips** $\lfloor n/2^i \rfloor$ ***times*** in a sequence of n INCREMENTs.

Aggregate Method

- For $\boldsymbol{k} = \mathbf{4}$ (no. of bits) and $\boldsymbol{n} = \mathbf{8}$ (counter value) total number of flips of bit can be given as,

$$A = \frac{8}{2^0} + \frac{8}{2^1} + \frac{8}{2^2} + \frac{8}{2^3}$$

- total bit flipping operations can be

$$A = 8 + 4 + 2 + 1 = \mathbf{15}$$

$$\boldsymbol{A} = \sum_{\boldsymbol{i}=\mathbf{0}}^{\boldsymbol{k}-\mathbf{1}} \frac{\boldsymbol{n}}{\mathbf{2}^{\boldsymbol{i}}}$$

n	Counter value	Number of bit flips
0	0 0 0 0	0
1	0 0 0 1	1
2	0 0 1 0	2
3	0 0 1 1	1
4	0 1 0 0	3
5	0 1 0 1	1
6	0 1 1 0	2
7	0 1 1 1	1
8	1 0 0 0	4
Total Flips = 15		

Aggregate Method

- Therefore, the total number of flips in the sequence is,

$$\sum_{i=0}^{K\text{-}1} \lfloor n/2^i \rfloor < n \sum_{i=0}^{\infty} 1/2^i = 2n$$

- Total time $\boldsymbol{T(n) = O(n)}$
- The amortized cost of each operation is $\mathbf{O(n)/\,n = O(1)}$

Accounting Method

- If we charge an amortized cost of ₹2 to set a bit to **1**.

- If we charge an amortized cost of ₹0 to set a bit to 0.

- When a bit is set we use ₹1 to pay for the actual setting of the bit and we place the other ₹1 on the bit **as a credit**.

Amortized Analysis - Example

Counter value	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]	Amortized cost	Actual cost	Credit
0	0	0	0	0	0	0	0	0	———	0	0
1	0	0	0	0	0	0	0	$\boxed{1}$	2	1	1
2	0	0	0	0	0	0	$\boxed{1}$	$\boxed{0}$	2	1	1
3	0	0	0	0	0	0	1	$\boxed{1}$	2	2	2
4	0	0	0	0	0	$\boxed{1}$	$\boxed{0}$	$\boxed{0}$	2	1	1
5	0	0	0	0	0	1	0	$\boxed{1}$	2	2	2
6	0	0	0	0	0	1	$\boxed{1}$	$\boxed{0}$	2	2	2
7	0	0	0	0	0	1	1	$\boxed{1}$	2	3	3

Total Amortized Cost 14

Amortized Analysis - Example

Counter value	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]	Amortized cost	Actual cost	Credit
0	0	0	0	0	0	0	0	0	———	0	0
1	0	0	0	0	0	0	0	$\boxed{1}$	2	1	1
2	0	0	0	0	0	0	$\boxed{1}$	$\boxed{0}$	2	1	1
3	0	0	0	0	0	0	1	$\boxed{1}$	2	2	2
4	0	0	0	0	0	$\boxed{1}$	$\boxed{0}$	$\boxed{0}$	2	1	1
5	0	0	0	0	0	1	0	$\boxed{1}$	2	2	2
6	0	0	0	0	0	1	$\boxed{1}$	$\boxed{0}$	2	2	2
7	0	0	0	0	0	1	1	$\boxed{1}$	2	3	3

Total Amortized Cost 14

Accounting Method

- Total Amortized cost= (0+2+2+2+2+2+2+2) for 7 increments
- Total Amortized cost= 14 for 7 increments
- Total Amortized cost=2*7 for 7 increments
- Total Amortized cost= 2 * n for n increments
- T(n)=2n
- The total amortized cost is $O(n)$.

Potential Method

- Same as accounting method: something prepaid is used later.
- Different from accounting method.
  - The prepaid work not as credit, but as "potential energy", or "potential".
  - The potential is associated with the data structure as a whole rather than with specific objects within the data structure.

Potential Method

- The Amortized cost c_i' of the i th operation with respect to potential function Φ is defined by :

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

  ie (actual cost + potential change)

  where

  - c_i' is Amortized cost of the i th operation .
  - c_i is Actual cost of the i th operation .
  - D_i is Data structure.
  - A potential function Φ: $\{D_i\} \rightarrow$ R (real numbers)
  - $\Phi(D_i)$ is called the potential of D_i.

- Potential of the counter after i^{th} Increment() operation to be b_i the number of 1's in the counter after i^{th} operation .

- Therefore $\Phi(D_i) = b_i$, the number of 1's. clearly, $\Phi(D_i) \geq 0$.

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
1	0	0	1	1	1	0	1	2
2	0	1	0	2	1	1	0	2
3	0	1	1	1	2	1	1	2

- Total Amortized cost for 7 operations =14
- Total Amortized cost for 7 operations = 2 * 7
- Total Amortized cost of n operations =2 * n
- The total amortized cost is $\boldsymbol{O}(\boldsymbol{n})$

Mathematical Proof Techniques

- Deduction, or direct proof
- Proof by Contradiction
- Proof by Mathematical Induction.

Probabilistic Analysis