Amortized Analysis

- Not just consider one operation, but a sequence of operations on a given data structure.
- Average cost over a sequence of operations.
- With amortized analysis, we can show that the average cost of an operation is small, if we average over a sequence of operations, even though a single operation within the sequence might be expensive.
- Amortized analysis:
  - No involvement of probability
  - Average performance on a sequence of operations, even some operation is expensive.
  - Guarantee average performance of each operation among the sequence in worst case.

Three Methods of Amortized Analysis

- Aggregate analysis:
  - Total cost of *n* operations, $T(n)/n$,
- 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.
- Potential method:
  - Same as accounting method
  - But store the credit as “potential energy” of the data structure as a whole instead of associating the credit with individual objects within the data structure.

Example for amortized analysis

- Stack operations:
  - PUSH(S,x), $O(1)$
  - POP(S), $O(1)$
  - MULTIPOP(S,k), running time- min(s,k)

```
while not STACK-EMPTY(S) and $k>0$ do
        POP(S)
        $k=k-1$
```

- Let us consider a sequence of n PUSH, POP, MULTIPOP.
  - The worst case cost for MULTIPOP in the sequence is $O(n)$, since the stack size is at most n.
  - thus the cost of the sequence is $O(n^2)$. Correct, but not tight.

Aggregate Analysis

- Using aggregate analysis, we can obtain a better upper bound over a sequence of n operations.
- In fact, a sequence of *n* operations on an initially empty stack cost at most *O*(*n*). Why?

•Each object can be POPed only once (including in MULTIPOP) for each time it is PUSHed.

•#POPs is at most #PUSHs, which is at most n.

•Any sequence of n number of PUSH, POP, and MULTIPOP operations takes a total of O(n) time.Thus the average cost of an operation is $O(n)/n = O(1)$.

•Amortized cost in aggregate analysis is defined to be average cost.

Another example: Incrementing a binary counter

- Binary counter of length *k*, A[0..*k*-1] of bit array.
- INCREMENT(A)

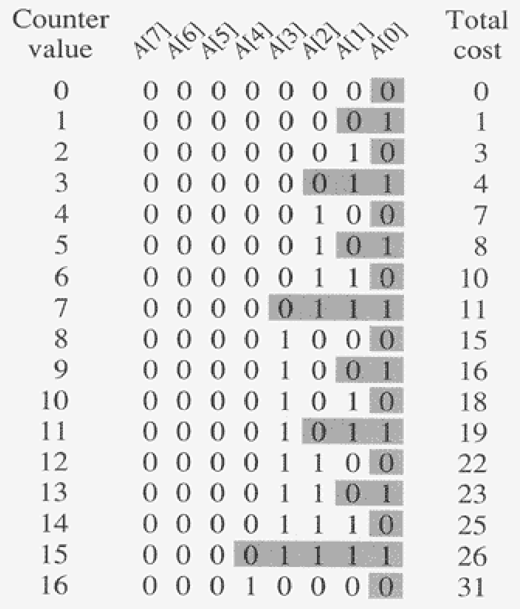
```
i=0
while i<k and A[i]==1 do
    A[i]=0 // (flip, reset)
 i=i+1
if  i<k then
    A[i]=1 // (flip, set)
```

Analysis of INCREMENT(A)

- Cursory analysis:
  - A single execution of INCREMENT takes $O(k)$ in the worst case (when A contains all 1s)
  - So a sequence of n executions takes $O(nk)$ in worst case (suppose initial counter is 0).
  - This bound is correct, but not tight.
- The tight bound is $O(n)$ for n executions.

Amortized (Aggregate) Analysis of INCREMENT(A)

Observation: The running time determined by #flips
but not all bits flip each time INCREMENT is called.



Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total 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

Figure 17.2 An 8-bit binary counter as its value goes from 0 to 16 by a sequence of 16 INCREMENT operations. Bits that flip to achieve the next value are shaded. The running cost for flipping bits is shown at the right. Notice that the total cost is never more than twice the total number of INCREMENT operations.

A[0] flips every time, total n times.
A[1] flips every other time, $\lfloor n/2 \rfloor$ times.
A[2] flips every forth time, $\lfloor n/4 \rfloor$ times.
….
for i=0,1,…,k-1, A[i] flips $\lfloor n/2^i \rfloor$ times.

Thus total #flips is $\sum_{i=0}^{k-1} \lfloor n/2^i \rfloor$
$< n\sum_{i=0}^{\infty} 1/2^i$
$=2n.$

Amortized Analysis of INCREMENT(A)

- Thus the worst case running time is $O(n)$ for a sequence of n INCREMENTs.
- So the amortized cost per operation is $O(1)$.

Amortized Analysis: Accounting Method

- Idea:
  - Assign differing charges to different operations.
  - The amount of the charge is called amortized cost.
  - Amortized cost is more or less than actual cost.
  - When amortized cost **>** actual cost, the difference is saved in specific objects as credits.
  - The credits can be used by later operations whose amortized cost **<** actual cost.
- As a comparison, in aggregate analysis, all operations have same amortized costs.

Accounting Method (cont.)

- Conditions:
  - suppose actual cost is c_i for the i^{th} operation in the sequence, and amortized cost is c_i',
  - $\sum_{i=1}^{n} c_i' >= \sum_{i=1}^{n} c_i$ should hold.
    - since we want to show the average cost per operation is small using amortized cost, we need the total amortized cost is an upper bound of total actual cost.
    - holds for all sequences of operations.
  - Total credits is $\sum_{i=1}^{n} c_i' - \sum_{i=1}^{n} c_i$, which should be nonnegative,
    - Moreover, $\sum_{i=1}^{t} c_i' - \sum_{i=1}^{t} c_i \geq 0$ for any $t>0$.

Accounting Method: Stack Operations

- Actual costs:
  - PUSH :1, POP :1, MULTIPOP: min(s,k).
- Let assign the following amortized costs:
  - PUSH:2, POP: 0, MULTIPOP: 0.
- Similar to a stack of plates in a cafeteria.
  - Suppose $1 represents a unit cost.
  - When pushing a plate, use one dollar to pay the actual cost of the push and leave one dollar on the plate as credit.
  - Whenever POPing a plate, the one dollar on the plate is used to pay the actual cost of the POP. (same for MULTIPOP).
  - By charging PUSH a little more, do not charge POP or MULTIPOP.
- The total amortized cost for n PUSH, POP, MULTIPOP is $O(n)$, thus $O(1)$ for average amortized cost for each operation.
- Conditions hold: total amortized cost $\geq$total actual cost, and amount of credits never becomes negative.

Accounting method: binary counter

- Let \$1 represent each unit of cost (i.e., the flip of one bit).
- Charge an amortized cost of \$2 to set a bit to 1.
- Whenever a bit is set, use \$1 to pay the actual cost, and store another \$1 on the bit as credit.
- When a bit is reset, the stored \$1 pays the cost.
- At any point, a 1 in the counter stores \$1, the number of 1's is never negative, so is the total credits.
- At most one bit is set in each operation, so the amortized cost of an operation is at most \$2.
- Thus, total amortized cost of *n* operations is *O*(*n*), and average is *O*(1).

The 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.

The Potential Method (cont.)

- Initial data structure D_0,
- n operations, resulting in $D_0, D_1, \ldots, D_n$ with costs $c_1, c_2, \ldots, c_n$.
- A potential function $\Phi: \{D_i\} \rightarrow$ R (real numbers)

$\forall$ $\Phi(D_i)$ is called the potential of D_i.

- Amortized cost c_i' of the ith operation is:
  - $c_i' = c_i + \Phi(D_i) - \Phi(D_{i-1})$. (actual cost + potential change)

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

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

The Potential Method (cont.)

- If $\Phi(D_n) \geq \Phi(D_0)$, then total amortized cost is an upper bound of total actual cost.
- But we do not know how many operations, so $\Phi(D_i) \geq \Phi(D_0)$ is required for any i.
- It is convenient to define $\Phi(D_0)=0$,and so $\Phi(D_i) \geq 0$, for all i.
- If the potential change is positive (i.e., $\Phi(D_i) - \Phi(D_{i-1})>0$), then c_i' is an overcharge (so store the increase as potential),
- otherwise, undercharge (discharge the potential to pay the actual cost).

Potential method: stack operation

- Potential for a stack is the number of objects in the stack.
- So $\Phi(D_0)=0$, and $\Phi(D_i) \geq 0$
- Amortized cost of stack operations:
  - PUSH:
    - Potential change: $\Phi(D_i)- \Phi(D_{i-1}) =(s+1)-s =1$.
    - Amortized cost: $c_i' = c_i + \Phi(D_i) - \Phi(D_{i-1})=1+1=2$.
  - POP:
    - Potential change: $\Phi(D_i)- \Phi(D_{i-1}) =(s-1) –s= -1$.
    - Amortized cost: $c_i' = c_i + \Phi(D_i) - \Phi(D_{i-1})=1+(-1)=0$.
  - MULTIPOP(S,k): k'=min(s,k)
    - Potential change: $\Phi(D_i)- \Phi(D_{i-1}) = –k'$.
    - Amortized cost: $c_i' = c_i + \Phi(D_i) - \Phi(D_{i-1})=k'+(-k')=0$.
- So amortized cost of each operation is $O(1)$, and total amortized cost of n operations is $O(n)$.
- Since total amortized cost is an upper bound of actual cost, the worse case cost of n operations is $O(n)$.

Potential method: binary counter

- Define the potential of the counter after the *i*th INCREMENT is $\Phi(D_i) = b_i$, the number of 1's. clearly, $\Phi(D_i) \geq 0$.
- Let us compute amortized cost of an operation
  - Suppose the *i*th operation resets t_i bits.
  - Actual cost c_i of the operation is at most $t_i + 1$.
  - If $b_i=0$, then the *i*th operation resets all *k* bits, so $b_{i-1}=t_i=k$.
  - If $b_i>0$, then $b_i=b_{i-1}-t_i+1$
  - In either case, $b_i \leq b_{i-1}-t_i+1$.
  - So potential change is $\Phi(D_i) - \Phi(D_{i-1}) \leq b_{i-1}-t_i+1-b_{i-1}=1-t_i$.
  - So amortized cost is: $c_i' = c_i + \Phi(D_i) - \Phi(D_{i-1}) \leq t_i + 1 + 1 - t_i = 2$.
- The total amortized cost of *n* operations is $O(n)$.
- Thus worst case cost is $O(n)$.