Amortized Analysis

```
for (i=0; i<n; i++) P;
or
…P1…P2….Pi………Pn…
```

- Operation P is called n times.
- Each call to P is not independent: its execution time depends on the previous calls.
- The "average" cost to P considers the average over successive calls.
  - Compared to the "average-case" analysis which considers the average over all instances based on their distribution

Example: memory allocation

- Memory allocation in garbage collection-based programming languages such as java, .net
  - An allocation is fast when there is space available
  - It triggers garbage collection when no enough space left to satisfy the request

Example: binary counter

```
void IncrementCounter(Bit A[], int k) // k = length(A)
{
  int i = 0;
  while (i<k && A[i] == 1) {
    A[i] = 0;
    i++;
  }
  if (i < k)
       A[i] = 1;
}
```

- Each call increases the counter by 1 until the counter rolls back to 0

Example: binary counter

```
void IncrementCounter(Bit A[], int k) // k = length(A)
{
  int i = 0;
  while (i<k && A[i] == 1) {
    A[i] = 0;
    i++;
  }
  if (i < k)
       A[i] = 1;
}
```

We count #flips for total cost

- Worst case the loop executes k times and takes time $\Theta(k)$.
- It takes time O(nk) for n calls

Experimental results

k (#bits)	Total iterations	Average iterations
1	2	1.0
2	6	1.5
4	30	1.875
8	510	1.9922
16	131070	2.0000

Three analyzing methods

- Aggregate analysis
- The accounting method
- The potential method

An aggregate analysis: binary counter

- For n consecutive operations
  - A[0] flips each time incrementCounter() is called
  - A[1] flips $\left\lfloor \frac{n}{2} \right\rfloor$ times
  - …
  - A[i] flips $\left\lfloor \frac{n}{2^i} \right\rfloor$ times
  - …
- Total flips is

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

- Average flips per operation ≈ 2

Stack operations

- Three operations
  - PUSH(S, x): pushes object x onto stack S. Cost O(1).
  - POP(S): pops the top of stack and return the popped object. Cost O(1).
  - MULTIPOP(S, k): pops the top k objects or the entire stack if it contains less than k objects. Cost O(min(s, k)) where s is the size of the stack.

```
MULTIPOP(S, k)
{
  while (!empty(S) and k > 0) {
      POP(S);
      k--;
  }
}
```

Amortized analysis for Stack

- What is the worst case cost of n stack operations on an initially empty stack
  - One MULTIPOP takes O(n)
  - n operations take $O(n^2)$
  - Average is O(n) ← This is not tight!
- An aggregate analysis
  - Observation:
    - the total numbers of PUSH(), P <= n
    - The total number of POP() called including those in MULTIPOP <= P <=n
  - The total cost is O(n)
  - And the average is O(1)

The accounting method

- Set up a virtual bank with the initial balance 0
- "Guess" an upper bound of the amortized cost of each call/operation. Let it be τ.
  - Deposit the amortized cost, τ, dollars for each operation
    - τ can be associated with a specific object
  - Draw the actual cost, c, dollars of each operation from the bank
    - If $c < \tau$, you get extra credits that can be associated with specific objects
    - If $c > \tau$, you spend your savings
- Show that the bank is never overdrawn

Accounting for binary counter

- Assume amortized cost: 2
  - Allocate 2 dollars for each call
  - Associate the 2 dollars with the bit set
- Actual cost:
  - Spend one dollar when a bit is flipped (set or reset)
- Analysis
  - Each bit "1" gets 1 dollar credit associated with it
  - Pay the flipping cost of each bit using the credit
  - Balance = the number of 1's which is never negative

Accounting for Stack

- Assume amortized cost:
  - PUSH: 2
  - POP: 0
  - MULTIPOP: 0
- Actual cost:
  - PUSH(S, x): 1
  - POP(S): 1
  - MULTIPOP(S, k): min(s, k)
- Show the bank is never overdrawn
  - When an object is pushed, it gets 1 dollar credit (deposit 2, pay 1)
  - When an object is popped, pay using its credit.
  - The bank balance = # objects in the stack

Potential functions

- A potential function describes the state of "cleanliness" before a process/operation executes.
  - A large value of the state means "dirtier": it denotes the amortized cost of the following processes
  - Let $\Phi(D_0)$ be the value of the initial state and $\Phi(D_i)$ be that of the state after the i[th] call, the amortized time taken by the i-th call is $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
  - Let T_n denote the total time required for the n calls, and $\hat{T}_n$ be the total amortized time, we have
    $$\hat{T}_n = T_n + \Phi(D_n) - \Phi(D_0)$$
  - $\hat{T}_n$ can be an upper bound for T_n when $\Phi(D_n) \geq \Phi(D_0)$

Potential method for Binary Counter

- We use the number of ones as potential function
- Then the amortized cost of adding one to the counter is
  - The counter value is even. The least significant bit (A[0]) is set and it adds one more 1. The amortized cost is 1+1=2.
  - All bits of the counter is 1. The loop executes k times and all k 1s change to 0s. The amortized cost is m+(0-m) = 0.
  - In other cases, assume the loop executes i times. It flips each of the rightmost i bits from 1 to 0, and set (i+1)-th bit from 0 to 1. The 1s decreases by i-1. The amortized cost if (i+1)-(i-1) = 2.

Potential Method for Stack

- Let $\Phi(D_i)$ =# objects in the stack
- Actual cost:
  - PUSH(S, x): 1
  - POP(S): 1
  - MULTIPOP(S, k): *k'*= min(s, k)
- Amortized cost:
  - PUSH: $1 + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$
  - POP: $1 + \Phi(D_i) - \Phi(D_{i-1}) = 1 + (-1) = 0$
  - MULTIPOP: $k' + \Phi(D_i) - \Phi(D_{i-1}) = k' + (-k') = 0$