Amortized Cost

- **Why use amortized cost?**
  Remember, we use the number of (basic) operations as a measure of running time. However, how can such a measure be useful if same operation (e.g., each cFind) may cost differently, depending when it is applied in a sequence of operations?

- In an amortized analysis, the time required to perform a sequence of data structure operations is averaged over all the operations performed.
  - Amortized cost differs from average-case cost, because there is no probability involved.
  - Amortized cost is the average performance of each operation in the worst-case.
Three techniques for amortized cost analysis

- Aggregate method
  - Amortized cost = total actual cost / number of operations

- Accounting method
  - Amortized cost = actual cost + accounting cost
  - The sum of the accounting cost is nonnegative

- Potential method
  - Amortized cost = actual cost + increment of potential
  - Potential never below zero

A simple example

stack operations

Push(S,x)
Pop(S)
MultiPop(S,k):
  pop k top objects of S.
  If S has less than k objects,
    then pop all in S.
a sequence of n Push, Pop and MultiPop operations on initially empty stack.
Worst-case cost:
  For each operation:
    Push: O(1)
    Pop: O(1)
    MultiPop: O(n) since the stack size is at most n
There are n operations (possibly O(n) MultiPop operations), the upper bound is O(n^2).
Problem: O(n^2) upper bound is not tight.
Aggregate method

Although a single MultiPop can be expensive, any sequence of n Push, Pop, and MultiPop operations on an initially empty stack can cost at most O(n).

Proof: Pop (either directly or from inside MultiPop) can be called n times since there can be at most n objects (by n Push operations).

The amortized cost of an operation is the average:

\[ O(n)/n = O(1) \]

Accounting method

- Data structure comes with a “bank account”
- Every operation allotted a fixed $ cost (its amortized cost)
- If actual cost less than allotted amount, the difference is deposited into bank
- If actual cost more than allotted amount, withdraw from bank to pay for the operation
- Catch: always have a non-negative balance
- Benefit: we can use an operation’s amortized cost, which is a fixed number, and we know that n times the amortized cost is the upper bound of the actual cost of n operations.
Actual cost:
- Push: 1
- Pop: 1
- MultiPop: min(k,s)

Amortized cost:
- Push: 2
- Pop: 0
- MultiPop: 0

Will the bank account be balanced?
Yes. A stack of plates in a cafeteria. We start with an empty stack. Push a plate on the stack and pop a plate off the stack cost $1 each. Now, when push, we pay $2. One dollar for the actual cost, and one dollar as a credit. Since every plate on the stack has a dollar of credit on it, pop is free.

Potential method
- Prepaid work as potential that can be released to pay for future operations
- Initial data structure $D_0$, on which $n$ operations are to be performed. $D_i$ is the data structure after $i$-th operation.
- Potential $\Phi: D_i \rightarrow \Phi(D_i)$
- Amortized cost $c_i$ of the $i$-th operation is its actual cost plus the increase in potential due to the operation
  \[ c_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \]
- The total amortized cost of the $n$ operations
  \[ \sum c_i = \sum (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum c_i + \Phi(D_n) - \Phi(D_0) \]
Potential $\Phi$: number of objects on the stack.
- $D_0$ is empty stack, and $\Phi(D_0) = 0$
- $\Phi(D_i) \geq 0 = \Phi(D_0)$
- Amortized cost:
  - Push
    $$c_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$$
  - MultiPop(S, k)
    $$c_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k' - k' = 0$$
    where $k' = \min(k,s)$ is the actual number objects removed from the stack.

---

**Amortized analysis of Build-heap**

Task: to build $n$ elements into a max-heap.
Without loss of generality, let's assume $n$ is a power of 2, so the $n$ elements can be mapped into a complete binary tree. There are $n/2 - 1$ interior nodes and heapify needs to apply to each one of them.

Actual cost = $\log(h)$ where $h$ is the height of the node that heapify is applied.
Amortized cost = 2.

Use the accounting method.

Deposit $2$ to each node. Comparison of a pair of keys will cost $1$. We will show by induction that the total amount of deposit ($=2n$) is sufficient to cover the cost of $n/2 - 1$ heapify operations.

Start with the lowest level (let's call it level-1) of interior nodes. Each has two children nodes. There will be 2 comparisons, cost $2$, and $4$ will be left in this subheap of 3 nodes. This is true for each level-1 subheap. Once done with this level, move one level up. Each node has $2$, plus $2 \times 4$ from two subheaps, there are $10$, out of which $4$ will cover the cost of heapify (worst-case). So, $6$ is left for every level 2 subheap. Therefore, we do induction: each level-i subheap has $(2i + 2)$. When we heapify a level-$(i+1)$ heap, the total amount available is $2 \times (2^i + 2) + 2$, which is $2^{i+1} + 6$. The total cost for heapifying a level-$(i+1)$ heap is $2 \log(i+1) = i+1$. Therefore, we always have enough to cover the expenses, i.e., the bank account will never go bankruptcy! So we can safely say each heapify only cost $2$, in amortized sense, i.e., the actual cost exceeding $2$ (the case when it is applied to higher level subheaps) will be covered by the savings from these nodes that are never got visited by the heapify.