Amortized Analysis

ICCS313 Algorithm Analysis

Chaya Hiruncharoenvate

What is Amortization?

(Accounting)

Gradually write off the initial cost of an asset over a period

(CS)

Not just consider one operation, but a sequence of operations

Amortized Analysis

- Not just consider one operation, but a sequence of operations on a given data structure.
- Average cost over a sequence of operations.

Amortized Analysis

- Probabilistic analysis:
  - ➢Average case running time: average over all possible inputs for one algorithm (operation).
  - ➢If using probability, called expected running time.
- 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 in operations/*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 (not covered in this course):
  - Same as accounting method
  - But store the credit as “potential energy” and as a whole.

Aggregate Analysis

Example 1: Stack Operation

What's the running time for the following operations?

- PUSH(S, x) insert x at the top of stack S
  - $O(1)$
- POP(S) pops the top of stack S and returns the popped object
  - $O(1)$
- $O(1)$ ~ cost 1
- A sequence of n PUSH and POP operations
  - $T(n) = n * 1 = O(n)$

Example 1: Stack Operation

What's the running time for the following operations?

- MULTIPOP(S, k) remove k top objects from stack S

MULTIPOP(S, k)

1. **while** not STACK-EMPTY(S) and $k > 0$
2. POP(S)
3. $k = k - 1$

➢ $T(n) = min(|S|, k) = O(n)$

Example 1: Stack Operation

What's the running time for the following operations?

- A sequence of n PUSH, POP, and MULTIPOP operations
  - $T(n) = n * \max(1, 1, n) = n * n = O(n^2)$

Example 1: Stack Operation

- In fact, a sequence of n operations on an initially empty stack cost at most $O(n)$.
- Why?
  - ➢Each object can be POP only once (including MULTIPOP) for each PUSH
  - ➢# of POP ≤ # of PUSH ≤ n
  - ➢Total cost of operations = $O(n)$
- Average cost = $O(n)/n = O(1)$
- Amortized cost in aggregate analysis is defined to be average cost.

Example 2: Increasing a binary counter

- Binary counter of length k, $A[0 \ldots k\text{-}1]$
- How do you increase a binary counter?

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

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

Example 2: Increasing a binary counter

- Binary counter of length k, $A[0 \ldots k\text{-}1]$

INCREMENT(A)

1. $i = 0$

2. while $i < k$ and $A[i]$=1

3. $\quad A[i] = 0$ (flip, reset)

4. $\quad i = i + 1$

5. if $i < k$

6. $\quad A[i] = 1$ (flip, set)

Analysis of INCREMENT

Cursory analysis

- A single execution of INCREMENT takes O(k) in the worst case (when A is all 1s)
- → A sequence of n executions = O(nk) in the worst case
- The bound is correct, but not tight
- The tight bound is O(n) for n executions
    - ➢ Why?

Amortized (Aggregate) Analysis of INCREMENT

Observation: The running time is determined by #flips,

but not all bits flop 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

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

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

Accounting Method

Amortized Analysis: Accounting Method

- Assign differing charges to different operations
- The amount of charge is called **amortized cost**
- When **amortized cost > actual cost**, the difference is saved as credits
- When **amortized cost < actual cost**, the credits are used to offset the cost

- As a comparison, in aggregate analysis, all operations have the same amortized costs.

Conditions of Accounting Method

- Suppose, for the i^{th} operation, the actual cost is c_i and the amortized cost is c_i'
- $\sum_{i=1}^{n} c_i' \geq \sum_{i=1}^{n} c_i$ must hold
  - Since we want to show the average cost per operation is small using *amortized cost*, we need the total *amortized cost* to be an upper bound of the total *actual cost*
  - Holds for all sequences of operations
- Total credits is $\sum_{i=1}^{n} c_i' - \sum_{i=1}^{n} c_i$ must be nonnegative
  - Furthermore, $\sum_{i=1}^{t} c_i' - \sum_{i=1}^{t} c_i \geq 0$ for any $t > 0$

Example 1: Stack Operations

- Actual costs:
  - PUSH: 1, POP:1, MULTIPOP: min(s, k)
- Assign the following amortized costs
  - PUSH: 2, POP: 0, MULTIPOP: 0
- Visualization
  - When pushing an item, you use \$1 to pay the actual cost of pushing, and leave \$1 on the item as credit
  - When popping an item, the \$1 on the item is used to pay the actual cost of POP (same for MULTIPOP)

Example 1: Stack Operations

- The total amortized cost for n PUSH, POP, MULTIPOP is $O(n)$ → $O(1)$ for average amortized cost for each operation
- The conditions hold
  - ➢total amortized cost ≥ total actual cost
  - ➢Credits never become negative

Example 2: 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 $O(1)$.

Example: Dynamic Table

Dynamic Table

- We want to store our data in a table (maybe a hash table), but we don’t know how large in advance
- Table operations: insertion, deletion
- Table may expand with insertion
- Table may contract with deletion

Goals:

- $O(1)$ amortized cost
- Unused space always ≤ constant fraction of allocated space

Dynamic Table

- **Load factor** $\alpha = num/size$
  - ➢ *num* = # items stored, *size* = allocated size
  - ➢ size = 0 → num = 0 → $\alpha = 1$
- Never allow $\alpha > 1$
- Keep α > a constant fraction (Goal 2)

Dynamic Table: Expansion with Insertion

- Consider only insertion
- When the table becomes full, double its *size* and reinsert all existing items
- Guarantee that $\alpha \geq 1/2$
- Each time we insert an item, it's an ***elementary insertion***

```
TABLE-INSERT(T, x)
if T.size == 0
    allocate T.table with 1 slot
    T.size = 1
if T.num == T.size
    allocate new-table with 2 · T.size slots
    insert all items in T.table into new-table      T.num elements insertion
    free T.table
    T.table = new-table
    T.size = 2 · T.size
insert x into T.table      1 element insertion
T.num = T.num + 1
```

Initially, $num[T] = size[T] = 0$

TABLE-INSERT: Aggregate Analysis

- **Running Time**: Charge 1 per ***elementary insertion***. Count only elementary insertions
  - ➢Since all other costs together are constant per call
- c_i = actual cost of i^{th} operation
  - ➢If not full, $c_i = 1$
  - ➢If full → have i-1 items in the table at the start of the i^{th} operation → have to copy all i-1 existing items and insert the i^{th} item → $c_i = i$
- *Cursory analysis*: n operations, each operation worst case = $O(n)$ → $O(n^2)$ time for n operations

TABLE-INSERT: Aggregate Analysis

- However, we don't always expand:

$$c_i = \begin{cases} i & \text{if } i-1 \text{ is the exact power of } 2 \\ 1 & \text{otherwise} \end{cases}$$

- Total cost = $\sum_{i=1}^{n} c_i \leq n + \sum_{i=0}^{\log n} 2^i \leq n + 2n = 3n$
- Therefore, by aggregate analysis, amortized cost per operation = $3n/n$ = 3

TABLE-INSERT: Accounting Method

- Charge \$3 per insertion of *x*
  - \$1 pays for *x*'s insertion.
  - \$1 pays for *x* to be moved in the future.
  - \$1 pays for some other item to be moved.
- Suppose we've just expanded, *size* = *m* before next expansion, *size* = 2*m* after next expansion.
- Assume that the expansion used up all the credit, so that there's no credit stored after the expansion.
- Will expand again after another *m* insertions.
- Each insertion will put \$1 on one of the *m* items that were in the table just after expansion and will put \$1 on the item inserted.
- Have \$2*m* of credit by next expansion, when there are 2m items to move. Just enough to pay for the expansion, with no credit left over!

Expansion and Contracting

- When α drops too low, contract the table.
  - Allocate a new smaller one
  - Copy all items
- Still want
  - α bounded from below by a constant
  - Amortized cost per operation = $O(1)$
- Measure cost in terms of elementary insertions and deletions

Obvious Strategy

- Double size when inserting into a full table (when $\alpha = 1$, so that after insertion α would be < 1).
- Halve size when deletion would make table less than half full (when $\alpha = 1/2$, so that after deletion α would become $>= 1/2$).
- Then always have $1/2 \leq \alpha \leq 1$.

Obvious Strategy (cont.)

- Suppose we fill table.
  - Then insert → double
  - 2 deletes → halve
  - 2 inserts → double
  - 2 deletes → halve
  - …
  - Cost of each expansion or contraction is $\Theta(n)$, so total n operation will be $\Theta(n^2)$.
- Problem is that: Not performing enough operations after expansion or contraction to pay for the next one.

Simple Solution

- Double as before: when inserting with $\alpha = 1 \rightarrow$ after doubling, $\alpha = 1/2$.
- Halve size when deleting with $\alpha = 1/4 \rightarrow$ after halving, $\alpha = 1/2$.
- Thus, immediately after either expansion or contraction, have $\alpha = 1/2$.
- Always have $1/4 \le \alpha \le 1$.

Intuition: Simple Solution

- Want to make sure that we perform enough operations between consecutive expansions/contractions to pay for the change in table size.
- Need to delete half the items before contraction.
- Need to double number of items before expansion.
- Either way, number of operations between expansions/contractions is at least a constant fraction of number of items copied.

Aggregate Analysis/Accounting Method

- Left as homework in Assignment 5