# COMP 251

## Algorithms & Data Structures (Winter 2021)

## Extras – Amortized Analysis

School of Computer Science
McGill University

Slides of Langer (2014) & Cormen et al., 2009 & Comp251-Fall McGill

# Amortized Analysis – Introduction 1

- Make good use of class time (minimum requirements).
  - I assume you are working for 40 hours a week and you are taking five courses: 8 work hours per week per course * 13 weeks = 104 hours total, which breaks down to:
    - 39 hours of scheduled lecture time (3 hours per week)
    - 39 hours of review/exercises, including studying for midterm exams (3 hours per week)
    - 26 hours for 3 assignments ('amortized' 2 hours per week)

When I say that you spend an average of 2 hours per week on assignments, I mean that this is the amount of work you do per week, averaged or "amortized" over the semester.

- There is nothing random going on here.

# Amortized Analysis – Introduction 2

When you buy a house for 500K, you do not (usually) pay the whole amount up front and then live in it for free for n = 25 years. Rather, you pay say 100K and the bank pays 400K, and you make regular (equal) mortgage payments to the bank for n = 25 years. The amount is determined by an amortization table.

- There is nothing random going on here.

Wikipedia:

Amortization is the process of accounting for an amount over a period.

Wikipedia: In computer science, **amortized analysis** is a method of analyzing the execution cost of algorithms over a sequence of operations.

# Amortized Analysis

- Analyze a sequence of operations on a data structure.
- We will talk about average performance of each operation in the worst case (i.e. not averaging over a distribution of inputs. No probability!)
- **Goal:** Show that although some individual operations may be expensive, on average the cost per operation is small.
- 3 methods:
  1. aggregate analysis
  2. accounting method
  3. potential method (See textbook for more details)

# Amortized Analysis – Aggregate Analysis

- **Goal:** Show that although some individual operations may be expensive, on average the cost per operation is small.

- 3 methods:
  1. aggregate analysis.
     - A sequence of n operations takes worst-case time $T(n)$ in total. In the worst case, the average cost, or amortized cost, per operation is therefore $T(n)/n$.
     - Note that this amortized cost applies to each operation, even when there are several types of operations in the sequence.
     - Although some operations might be expensive, many others might be cheap.
     - 26 hours during the term (13 weeks) . You spend an average of 2 hours per week on assignments. $T(n) = 26$, $T(n)/n = 2$.

# Aggregate Analysis – Example 1

**Stack operations**

- PUSH($S$, $x$): $O(1)$ each $\Rightarrow$ $O(n)$ for any sequence of $n$ operations.
- POP($S$): $O(1)$ each $\Rightarrow$ $O(n)$ for any sequence of $n$ operations.
- MULTIPOP($S$,$k$):

```
while S≠Ø and k>0 do
    POP(S)
    k⟵k−1
```

Running time of MULTIPOP?

# Aggregate Analysis – Example 1

Running time of MULTIPOP:

- Let each PUSH/POP cost 1.
- # of iterations of **while** loop is $\min(s, k)$, where $s$ = # of objects on stack. Therefore, total cost = $\min(s, k)$.

Sequence of $n$ PUSH, POP, MULTIPOP operations:

- Worst-case cost of MULTIPOP is $O(n)$.
- Have $n$ operations.
- Therefore, worst-case cost of sequence is $O(n^2)$.

**But:**

- Each object can be popped only once per time that it is pushed.
- Have $\leq n$ PUSHes $\Rightarrow \leq n$ POPs, including those in MULTIPOP.
- Therefore, total cost = $O(n)$.
- Average over the $n$ operations $\Rightarrow O(1)$ per operation on average.

# Aggregate Analysis – Example 2

- $k$-bit binary counter $A[0\,..\,k-1]$ of bits, where $A[0]$ is the least significant bit and $A[k-1]$ is the most significant bit.
- Counts upward from 0.
- Value of counter is: $\sum_{i=0}^{k-1} A[i] \cdot 2^i$
- Initially, counter value is 0, so $A[0\,..\,k-1] = 0$.
- To increment, add 1 (mod $2^k$ ):

```
Increment(A,k):
i←0
while i<k and A[i]=1 do
    A[i]←0
    i←i+1
if i < k then
    A[i] ← 1
```

# Aggregate Analysis – Example 2

Let
k=3

| Counter Value | A<br>2 1 0 | cost |
|---|---|---|
| 0 | $0\ 0\ \underline{0}$ | 0 |
| 1 | $0\ \underline{0\ 1}$ | 1 |
| | $0\ 1\ \underline{0}$ | 3 |
| 3 | $\underline{0\ 1\ 1}$ | 4 |
| 4 | $1\ 0\ \underline{0}$ | 7 |
| 5 | $1\ \underline{0\ 1}$ | 8 |
| 6 | $1\ 1\ \underline{0}$ | 10 |
| 7 | $\underline{1\ 1\ 1}$ | 11 |
| 0 | $0\ 0\ 0$ | 14 |

We underline the bits we will flip at the next `increment`

Cost of INCREMENT = $\Theta$(# of bits flipped)
***Analysis:*** Each call could flip *k* bits, so *n* INCREMENTs takes *O*(*nk*) time.

# Aggregate Analysis – Example 2

| Bit | Flips how often | Time in n INCREMENTs |
|---|---|---|
| 0 | Every time | n |
| 1 | ½ of the time | floor($n$/2) |
| 2 | ¼ of the time | floor($n$/4) |
| … | … | |
| i | $1/2^i$ of the time | floor($n/2^i$) |
| … | … | |
| i≥k | Never | 0 |

Thus, total # flips = $\sum_{i=0}^{k-1} \lfloor n/2^i \rfloor < n \cdot \sum_{i=0}^{\infty} 1/2^i = n\left(\frac{1}{1-1/2}\right) = 2 \cdot n$

Therefore, $n$ INCREMENTs costs $O(n)$.
Average cost per operation = $O(1)$.

# Amortized Analysis – Accounting Method

- **Goal:** Show that although some individual operations may be expensive, on average the cost per operation is small.

- 3 methods:

  2. Accounting Method.
     - Assign different charges to different operations.
       - Some are charged more than actual cost.
       - Some are charged less.
     - ***Amortized cost*** = amount we charge.
       - When amortized cost is higher than the actual cost, store the difference *on specific objects* in the data structure as ***credit (we need to guarantee that the credit never goes negative!!!!!)***.
       - Use credit later to pay for operations whose actual cost is higher than the amortized cost.
     - 26 hours during the term (13 weeks) . You worked on the assignments only on the week of the due date. Use the credit (sleep - energy) stored in the other weeks to pay (work ~8hours) the week of the due date

# Amortized Analysis – Accounting Method

- **Goal:** Show that although some individual operations may be expensive, on average the cost per operation is small.

- 3 methods:
  2. Accounting Method.
     - Differs from aggregate analysis:
       - In the aggregate analysis, all operations have same cost.
       - In accounting method, different operations can have different costs.

# Amortized Analysis – Accounting Method

Let $c_i$ = cost of actual i$^{th}$ operation.

$\hat{c}_i$ = amortized cost of i$^{th}$ operation.

Then require $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$ for all sequences of n operations.

Total credit stored = $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \geq 0$

# Accounting Method – Example 1

| Operation | Actual cost | Amortized cost |
|---|---|---|
| PUSH | 1 | 2 |
| POP | 1 | 0 |
| MULTIPOP | min(k,s) | 0 |

***Intuition:*** When pushing an object, pay \$2.

- \$1 pays for the PUSH.
- \$1 is prepayment for it being popped by either POP or MULTIPOP.
- Since each object has \$1, which is credit, the credit can never go negative.
- Total amortized cost (= $O(n)$) is an upper bound on total actual cost.

# Accounting Method – Example 2

Charge \$2 to set a bit to 1.

- \$1 pays for setting a bit to 1.
- \$1 is prepayment for flipping it back to 0.
- Have \$1 of credit for every 1 in the counter.
- Therefore, credit $\geq 0$.

Amortized cost of INCREMENT:

- Cost of resetting bits to 0 is paid by credit.
- At most 1 bit is set to 1.
- Therefore, amortized cost $\leq$ \$2.
- For $n$ operations, amortized cost = $O(n)$.

# Amortized Analysis – Example 3 – Dynamic Tables

**Scenario**

- Have a table (maybe a hash table).
- Don't know in advance how many objects will be stored in it.
- When it fills, must reallocate with a larger size, copying all objects into the new, larger table.
- When it gets sufficiently small, *might* want to reallocate with a smaller size.

**Goals**

*1.* $O(1)$ amortized time per operation.

2. Unused space always ≤ constant fraction of allocated space.

**Load factor** $\alpha$ = (# items stored) / (allocated size)

Never allow $\alpha > 1$; Keep $\alpha >$ a constant fraction ⇒ Goal 2.

# Dynamic Tables - Expansion

Consider only insertion.

- When the table becomes full, double its size and reinsert all existing items.
- Guarantees that $\alpha \geq \frac{1}{2}$.
- Each time we insert an item into the table, it is an ***elementary insertion***.

```
TABLE-INSERT(T,x)
if size[T ]=0
      then allocate table[T] with 1 slot
            size[T]←1
if num[T]=size[T] then
      allocate new-table with 2 · size[T] slots
      insert all items in table[T] into new-table
      free table[T]
      table[T]←new-table
      size[T]←2·size[T]
insert x into table[T]
num[T]←num[T] + 1
```

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

# Dynamic Tables – Aggregate Analysis

- Cost of 1 per elementary insertion.
- Count only elementary insertions (other costs = constant).

$c_i$ = actual cost of $i^{\text{th}}$ operation

- If not full, $c_i = 1$.
- If full, have $i-1$ items in the table at the start of the $i^{\text{th}}$ operation. Have to copy all $i - 1$ existing items, then insert $i^{\text{th}}$ item $\Rightarrow c_i = i$.

Naïve: $n$ operations $\Rightarrow c_i = O(n) \Rightarrow O(n^2)$ time for $n$ operations

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

Total cost = $\sum_{i=1}^{n} c_i \le n + \sum_{j=0}^{\lfloor \log n \rfloor} 2^j = n + \dfrac{2^{\lfloor \log n \rfloor + 1} - 1}{2-1} < n + 2n = 3n$

Amortized cost per operation = 3.

# Dynamic Tables – 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.

# Dynamic Tables – 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.

Prove the credit never goes negative:

- *size*=$m$ before and *size*=$2m$ after expansion.
- Assume that the expansion used up all the credit, thus that there is no credit available after the expansion.
- We 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 \$$2m$ of credit by next expansion, when there are $2m$ items to move. Just enough to pay for the expansion…

# Outline

- Extras.
  - Amortized Analysis.
  - Randomized algorithms.
  - Probabilistic Analysis.
  - Review Final Exam.