Chapter 10

Amortized Complexity

Example

\( n \) increments of a binary counter starting with 0
Example

\( n \) increments of a binary counter starting with 0

- WCC of one increment? \( O(\log_2 n) \)
- WCC of \( n \) increments? \( O(n \cdot \log_2 n) \)
- \( O(n \cdot \log_2 n) \) is too pessimistic!
- Every second increment is cheap and compensates for the more expensive increments

WCC = worst case complexity
The problem

WCC of individual operations may lead to overestimation of WCC of sequences of operations

Amortized analysis

Idea:
Try to determine the average cost of each operation (in the worst case!)

Use cheap operations to pay for expensive ones

Method:
- Cheap operations pay extra (into a “bank account”), making them more expensive
Bank account = Potential

- The potential (“credit”) is implicitly “stored” in the data structure.
- Potential $\Phi :: \text{data-structure} \Rightarrow \text{non-neg. number}$ tells us how much credit is stored in a data structure
- Increase in potential = deposit to pay for later expensive operation
Bank account = Potential

- The potential ("credit") is implicitly "stored" in the data structure.
- Potential $\Phi :: \text{data-structure} \Rightarrow \text{non-neg. number}$ tells us how much credit is stored in a data structure.
- Increase in potential = deposit to pay for later expensive operation
- Decrease in potential = withdrawal to pay for expensive operation

Back to example: counter

Increment:
- Actual cost: 1 for each bit flip
- Bank transaction:
  - pay in 1 for final $0 \rightarrow 1$ flip
  - take out 1 for each $1 \rightarrow 0$ flip

$\Rightarrow$ increment has amortized cost $2 = 1+1$
Back to example: counter

Increment:
- Actual cost: 1 for each bit flip
- Bank transaction:
  - pay in 1 for final 0 → 1 flip
  - take out 1 for each 1 → 0 flip
implies increment has amortized cost 2 = 1+1

Data structure

Given an implementation:

Increment:
- Actual cost: 1 for each bit flip
- Bank transaction:
  - pay in 1 for final 0 → 1 flip
  - take out 1 for each 1 → 0 flip
implies increment has amortized cost 2 = 1+1

Formalization via potential:
Φ counter = the number of 1's in counter

Formalization via potential:
Φ counter = the number of 1's in counter
Data structure

Given an implementation:

- Type $\tau$
- Operation(s) $f :: \tau \Rightarrow \tau$

Data structure

Given an implementation:

- Type $\tau$
- Operation(s) $f :: \tau \Rightarrow \tau$
  (may have additional parameters)

Data structure

Given an implementation:

- Type $\tau$
- Operation(s) $f :: \tau \Rightarrow \tau$
  (may have additional parameters)
- Initial value: $\text{init} :: \tau$
  (function “empty”)

Needed for complexity analysis:

- Time/cost: $t \cdot f :: \tau \Rightarrow \text{num}$
  ($\text{num} =$ some numeric type)
Amortized and real cost

Sequence of operations: $f_1, \ldots, f_n$
Sequence of states:

$s_0 := \text{init}, s_1 := f_1 \ s_0,$

Amortized cost $:= \text{real cost} + \text{potential difference}$

$a_{i+1} := t_{f_{i+1}} \ s_i + \Phi s_{i+1} - \Phi s_i$

Amortized cost $:= \text{real cost} + \text{potential difference}$

$a_{i+1} := t_{f_{i+1}} \ s_i + \Phi s_{i+1} - \Phi s_i$

$\implies$

Sum of amortized costs $\geq$ sum of real costs
Amortized and real cost

Sequence of operations: $f_1, \ldots, f_n$
Sequence of states:

$s_0 := init$, $s_1 := f_1 s_0$, $\ldots$, $s_n := f_n s_{n-1}$

Amortized cost := real cost + potential difference

$$a_{i+1} := t_{f_{i+1}} s_i + \Phi s_{i+1} - \Phi s_i$$

⇒ Sum of amortized costs ≥ sum of real costs

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} (t_{f_i} s_{i-1} + \Phi s_i - \Phi s_{i-1})$$

Verification of amortized cost

For each operation $f$,
provide an upper bound for its amortized cost

$$a_f \in \tau \Rightarrow num$$

and prove

$$t_f s + \Phi(f s) - \Phi s \leq a_f s$$
Back to example: counter

\[ incr :: \text{bool list} \Rightarrow \text{bool list} \]

\[
\begin{align*}
\text{incr} & \left[ \right] = \left[ \text{True} \right] \\
\text{incr} \left( \text{False} \# \text{bs} \right) & = \text{True} \# \text{bs} \\
\text{incr} \left( \text{True} \# \text{bs} \right) & = \text{False} \# \text{incr} \text{bs} \\
\text{init} & = \left[ \right] \\
\Phi \text{ bs} & = \text{length (filter id bs)}
\end{align*}
\]

Lemma

\[ \text{inc_r bs} + \Phi (\text{inc_r bs}) - \Phi \text{ bs} = 2 \]
Back to example: counter

\textit{incr} :: \textit{bool list} \Rightarrow \textit{bool list}
\textit{incr} [] = [\textit{True}]
\textit{incr} (\textit{False} \# \textit{bs}) = \textit{True} \# \textit{bs}
\textit{incr} (\textit{True} \# \textit{bs}) = \textit{False} \# \textit{incr} \textit{bs}

Proof obligation summary

- \( \Phi \ s \geq 0 \)
- \( \Phi \ \textit{init} = 0 \)
- For every operation \( f :: \tau \Rightarrow ... \Rightarrow \tau \):
  \( t_{\cdot f} s \bar{x} + \Phi(f s \bar{x}) - \Phi s \leq a_{\cdot f} s \bar{x} \)

If the data structure has an invariant \textit{invar}:
assume precondition \textit{invar} s

Proof obligation summary

- \( \Phi \ s \geq 0 \)
- \( \Phi \ \textit{init} = 0 \)
- For every operation \( f :: \tau \Rightarrow ... \Rightarrow \tau \):
  \( t_{\cdot f} s \bar{x} + \Phi(f s \bar{x}) - \Phi s \leq a_{\cdot f} s \bar{x} \)

If the data structure has an invariant \textit{invar}:
assume precondition \textit{invar} s

If \( f \) takes 2 arguments of type \( \tau \):
\( t_{\cdot f} s_1 s_2 \bar{x} + \Phi(f s_1 s_2 \bar{x}) - \Phi s_1 - \Phi s_2 \leq a_{\cdot f} s_1 s_2 \bar{x} \)
Amortized analysis unsuitable for real time applications:

Real running time for individual calls may be much worse than amortized time

Amortized analysis is only correct for single threaded uses of the data structure.

Single threaded = no value is used more than once
Warning: single threaded

Amortized analysis is only correct for single threaded uses of the data structure.

Single threaded = no value is used more than once

Otherwise:

\[
\begin{align*}
\text{let } & \text{ counter } = 0; \\
\text{bad } = & \text{ increment counter } 2^n - 1 \text{ times;} \\
\_ = & \text{ incr bad;} \\
\_ = & \text{ incr bad;} \\
\_ = & \text{ incr bad;} \\
\vdots
\end{align*}
\]

Warning: observer functions

Observer function: does not modify data structure

\[\Rightarrow \text{ Potential difference } = 0\]

Warning: observer functions

Observer function: does not modify data structure

\[\Rightarrow \text{ Potential difference } = 0\]

\[\Rightarrow \text{ amortized cost } = \text{ real cost}\]
Warning: observer functions

Observer function: does not modify data structure
⇒ Potential difference = 0
⇒ amortized cost = real cost
⇒ Must analyze WCC of observer functions

This makes sense because

Observer functions do not consume their arguments!

---

### Amortized Complexity

Motivation
Formalization
Simple Classical Examples
Amortized and real cost

Sequence of operations: \( f_1, \ldots, f_n \)
Sequence of states:

\[
\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} (t_i f_i s_{i-1} + \Phi s_i - \Phi s_{i-1}) \\
\geq \sum_{i=1}^{n} t_i f_i s_{i-1}
\]

Amortized and real cost

Sequence of operations: \( f_1, \ldots, f_n \)
Sequence of states:

\[
\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} (t_i f_i s_{i-1} + \Phi s_i - \Phi s_{i-1}) \\
\geq \sum_{i=1}^{n} t_i f_i s_{i-1}
\]
A skew heap is a self-adjusting heap (priority queue)

Functions insert, merge and del_min have amortized logarithmic complexity.
A skew heap is a self-adjusting heap (priority queue)

Functions insert, merge and del_min have amortized logarithmic complexity.

Functions insert and del_min are defined via merge.

\[
\text{merge } h \langle \rangle = h \\
\text{merge } h \langle x \rangle = h
\]

Swap subtrees when descending:

\[
\text{merge } (\langle l_1, a_1, r_1 \rangle =: h_1) (\langle l_2, a_2, r_2 \rangle =: h_2) = \\
\begin{cases} 
\text{merge } h_2 r_1, a_1, l_1 & \text{if } a_1 \leq a_2 \text{ then} \\
\text{else } \langle \text{merge } h_1 r_2, a_2, l_2 \rangle & \text{else}
\end{cases}
\]
Logarithmic amortized complexity

Theorem
\[
t_{\text{merge}} t_1 t_2 + \Phi (\text{merge } t_1 t_2) - \Phi t_1 - \Phi t_2
\leq 3 \cdot \log_2 (|t_1| + |t_2|) + 1
\]

Towards the proof

Main proof

\[
t_{\text{merge}} t_1 t_2 + \Phi (\text{merge } t_1 t_2) - \Phi t_1 - \Phi t_2
\leq l_{rh} (\text{merge } t_1 t_2) + r_{lh} t_1 + r_{lh} t_2 + 1
\]
\[
\leq \log_2 |\text{merge } t_1 t_2|_1 + \log_2 |t_1|_1 + \log_2 |t_2|_1 + 1
\]
\[
= \log_2 (|t_1|_1 + |t_2|_1 - 1) + \log_2 |t_1|_1 + \log_2 |t_2|_1 + 1
\]
\[
\leq \log_2 (|t_1|_1 + |t_2|_1) + 2 \cdot \log_2 (|t_1|_1 + |t_2|_1) + 2
\]
because \[
\log_2 x + \log_2 y \leq 2 \cdot \log_2 (x + y) \quad \text{if } x, y > 0
\]