Amortised analysis
COMP4500/7500
Advanced Algorithms & Data Structures

September 17, 2018

Overview of today

- Admin/reminders
- Amortised analysis
  1. Aggregate method
  2. Accounting method
  3. Potential method
- Examples:
  1. Stack operations
  2. Incrementing a binary counter
  3. Resizing arrays

Amortised analysis

- We have so far analysed algorithms for “one-off” use
- However often we use a data structure (object) for a purpose that involves many uses of its methods.
  E.g. priority queue in Dijkstra’s shortest path algorithm.

DIJKSTRA(G, w, s)

// G is the graph, w the weight function, s the source vertex
INIT-SINGLE-SOURCE(G, s)
$S = \emptyset$ // S is the set of visited vertices
$Q = G.V$ // Q is a priority queue maintaining $G.V - S$
while $Q \neq \emptyset$
    $u =$ EXTRACT-MIN(Q)
    $S = S \cup \{u\}$
    for each vertex $v \in G.Adj[u]$
        RELAX(u, v, w)

Amortised analysis

- Consider an object x with multiple operations.
  - the "worst" is worst-case $O(n)$.
  - you design an algorithm that uses x's methods n times
  - Naive analysis: $O(n^2)$
- However, you may know that the worst case cannot happen n times in a row.. how can you prove your implementation is actually better than $O(n^2)$?
- Amortised analysis considers *sequences* of operations, typically that successively modify a data structure

Amortised analysis

Examples:

1. Java's `ArrayList` class: "The add operation runs in amortized constant time"
2. Java's `ArrayDeque` class: "Most ... operations run in amortized constant time"
3. C++ `vector` class: Operation `push_back`: "Complexity [is] Constant (amortized time, reallocation may happen)"

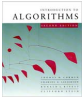
Motivating example: dynamic table

Store an initially unknown number of elements in an array.

- Double the size of the array when it runs out of space

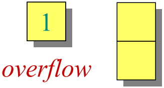
Typically inserting an element is constant time; however sometimes one must

- Allocate a new, larger array (twice the size)
- Copy all elements to the new array, including the new element



Example of a dynamic table

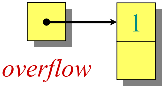
1. INSERT
2. INSERT



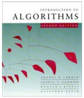
 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

Example of a dynamic table

1. INSERT
2. INSERT

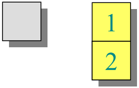


 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

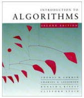


Example of a dynamic table

1. INSERT
2. INSERT

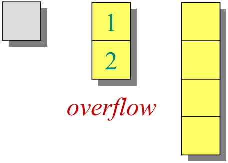


October 31, 2005 *Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson* L13.5

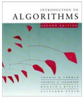


Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT

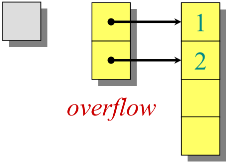


 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Example of a dynamic table

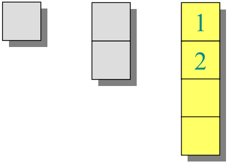
1. INSERT
2. INSERT
3. INSERT

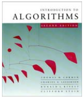


 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

Example of a dynamic table

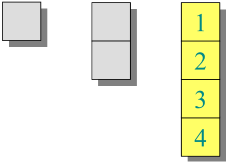
1. INSERT
2. INSERT
3. INSERT





Example of a dynamic table

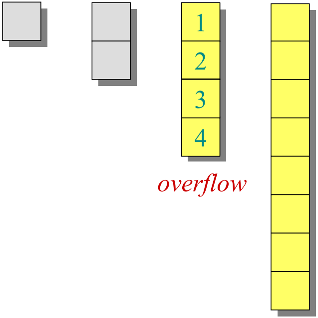
1. INSERT
2. INSERT
3. INSERT
4. INSERT



 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

Example of a dynamic table

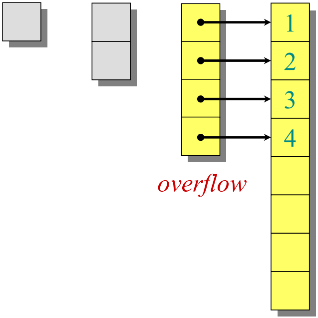
1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT



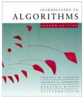
 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT

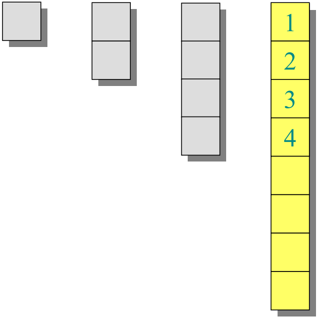


 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

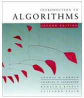


Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT

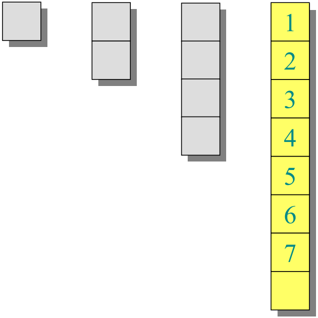


Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT
6. INSERT
7. INSERT



Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

Dynamic table

- After n operations (inserting a new element) there are n elements in the array
- Inserting an element when the capacity is full (worst case $n = 2^i$ for some i) is $\Theta(n)$.
- Thus, inserting n elements

$$\text{INSERT}(A); \text{INSERT}(B); \ldots \text{INSERT}(M); \text{INSERT}(N);$$

is $\Theta(n^2)$?

No: it is still $\Theta(n)$

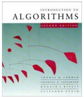
Dynamic table

We are analysing:

for $i = 1..n$
    INSERT(e_i)

for some sequence of elements $e_1, e_2, .., e_n$.

The vast majority of the insertions are constant time
How many are not, and what is their cost? This depends on i



Tighter analysis

Let $c_i =$ the cost of the ith insertion

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

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1

October 31, 2005 *Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson* L13.15

Tighter analysis

Let $c_i =$ the cost of the ith insertion

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

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	1	1	1	1	1	1	1	1	1
		1	2		4				8	

 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

Dynamic table: aggregate method

Inserting the 2nd, 3rd, 5th, 9th, ... elements
(when the array has size 1, 2, 4, 8 ...)
has an additional cost equal to the size of the array.

In general, how many resizes are there in a sequence of n INSERT operations, for $n \geq 1$?

$$\lceil \lg n \rceil$$

How much does the *jth* resize operation cost?

$$2^{j-1}$$

Dynamic table: aggregate method

Cost of n insertions is

$$
\begin{aligned}
& \textstyle\sum_{i=1}^{n} c_i \\
\leq \;& n + \textstyle\sum_{j=1}^{\lceil \lg n \rceil} 2^{j-1} \\
\leq \;& 3n \\
= \;& \Theta(n)
\end{aligned}
$$

The average cost of each dynamic-table operation is $\Theta(n)/n = \Theta(1)$.

Example: stack operations

Consider standard stack operations PUSH and POP, plus

MULTIPOP(S, k) // Assumes S is not empty and $k > 0$

```
while S is not empty and k > 0
    POP(S)
    k = k − 1
```

- MULTIPOP(S, k) can be $O(n)$ (n is the size of the stack) if $k = n$
- Hence, any sequence of n stack operations must be $O(n^2)$
- But can we prove a better bound?

Intuition:

- MULTIPOP will only iterate while the stack is not empty.
- Each element is pushed exactly once and popped exactly once, hence after m pushes there can be at most m pops

Stack operations: aggregate method

Analyse:

```
for i = 1..n
        PUSH(..)
        or
        POP(..)
        or
        MULTIPOP(..)
```

Arguing this is $O(n)$ is clumsy using the *aggregate* method.
Consider more sophisticated techniques:

- accounting method – *focus on the operations*
- potential method – *focus on the data structure*

Stack operations: accounting method

1. Calculate the *actual cost*, c_i, of each operation:
   1. PUSH: 1
   2. POP: 1
   3. MULTIPOP(S, K): k', where $k' = min(\text{SIZE}(S), k)$
2. Assign an *amortised cost*, $\hat{c}_i$, to each method.
   1. PUSH: 2
   2. POP: 0
   3. MULTIPOP(S, K): 0

For *any* sequence of stack operations, the amortised cost must be an upper bound on the actual cost

Then, one can use the amortised cost in place of the (more complicated) actual cost.
In the above case every operation has constant amortised cost, hence a sequence of n operations is $O(n)$.

Stack operations: accounting method

We must show that the total amortised cost minus the total actual cost is never negative:

$$\left(\sum_{i=1}^{n} \hat{c}_i\right) \geq \left(\sum_{i=1}^{n} c_i\right)$$

(for all sequences of all possible lengths n)

	actual cost	amortised cost
op	c_i	$\hat{c}_i$
PUSH	1	2
POP	1	0
MULTIPOP(S, k)	k'	0
	$= \text{MIN}(\#S, k)$	

Intuition: the extra credit in PUSH pays for the later (MULTI)POP.

Potential method

Focus on the data structure instead of the operations.

1. As before, determine the *actual cost*, c_i, of each operation
2. Define a *potential function*, Φ, on the data structure.
3. The *amortised cost*, $\hat{c}_i$, of an operation is the actual cost plus the *change in potential*

$$\hat{c}_i = c_i + (\Phi(D_i) - \Phi(D_{i-1}))$$

For *any* sequence of stack operations, the amortised cost must be an upper bound on the actual cost.

Since

$$\begin{aligned}\left(\sum_{i=1}^{n} \hat{c}_i\right) &= \left(\sum_{i=1}^{n} c_i + (\Phi(D_i) - \Phi(D_{i-1}))\right) \\ &= \left(\sum_{i=1}^{n} c_i\right) + (\Phi(D_n) - \Phi(D_0))\end{aligned}$$

the obligation is to show that $\Phi(D_i) \geq \Phi(D_0)$ after every operation. This is trivially true if $\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$.

Stack operations: potential method

1. Actual cost is as before $(1/1/k')$
2. $\Phi(S) = \#S$ (size of the stack)
3. Change in potential $(\Phi(D_i) - \Phi(D_{i-1}))$:
   1. PUSH: 1 (the size has increased by one)
   2. POP: -1 (the size has decreased by one)
   3. MULTIPOP: $-k'$ (the size has decreased by k')
4. Amortised cost (actual cost + change in potential):
   1. PUSH: 2 (= 1 + 1)
   2. POP: 0 (= 1 + -1)
   3. MULTIPOP: 0 (= $k' + -k'$)

Obligation: $\Phi(D_i) \geq \Phi(D_0) = 0$, which is trivial.

Therefore, all operations have constant amortised time.

For this simple example, potential and accounting method give almost exactly the same intuition

Incrementing a binary counter

```
INCREMENT(A, k)
i = 0
while i < k and A[i] == 1
        A[i] = 0
        i = i + 1
if i < k
        A[i] = 1
```

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

Incrementing a binary counter: aggregate method

Analyse:

```
for i = 1..n
    INCREMENT(A, k)
```

Intuition: the dominant cost is the number of flips (from 0 to 1 or 1 to 0)

After n INCREMENT operations, the number of times bit i is flipped is

$$\left\lfloor \frac{n}{2^i} \right\rfloor$$

The total cost is hence

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

Incrementing a binary counter: accounting method

1. The actual cost of each INCREMENT is equal to the number of low order 1s (which are flipped to 0), +1 for the flip from 0 to 1
2. Determining the amortised cost
   1. Let each flip from 0 to 1 have (amortised) cost of 2
   2. Let each flip from 1 to 0 have (amortised) cost of 0
3. The amortised cost of INCREMENT is therefore 2

We must show that this is an upper bound on the actual cost.

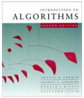
Flips from 1 to 0 can occur only after that bit has been flipped from 0 to 1. Each of the latter flips *pays in advance* for the flip back to 0.

Incrementing a binary counter: potential method

See tutorial next week

Array resizing: accounting method

1. Recall actual cost
2. Define amortised cost
3. Ensure that the amortised cost is an upper bound on the actual cost



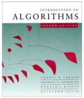
Tighter analysis

Let c_i = the cost of the ith insertion

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

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	1 1	1 2	1	1 4	1	1	1	1 8	1

 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Accounting analysis of dynamic tables

Charge an amortized cost of $\hat{c}_i = \$3$ for the i th insertion.

- \$1 pays for the immediate insertion.
- \$2 is stored for later table doubling.

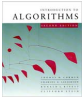
When the table doubles, \$1 pays to move a recent item, and \$1 pays to move an old item.

Example:

\$0	\$0	\$0	\$0	\$2	\$2	\$2	\$2

overflow

 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



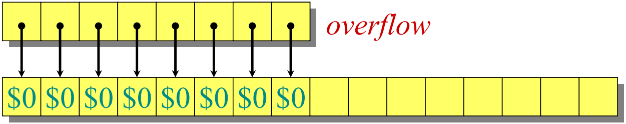
Accounting analysis of dynamic tables

Charge an amortized cost of $\hat{c}_i = \$3$ for the i th insertion.

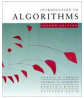
- $1 pays for the immediate insertion.
- $2 is stored for later table doubling.

When the table doubles, $1 pays to move a recent item, and $1 pays to move an old item.

Example:



October 31, 2005 *Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson* L13.22



Accounting analysis of dynamic tables

Charge an amortized cost of $\hat{c}_i = \$3$ for the i th insertion.

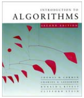
- $1 pays for the immediate insertion.
- $2 is stored for later table doubling.

When the table doubles, $1 pays to move a recent item, and $1 pays to move an old item.

Example:

$0	$0	$0	$0	$0	$0	$0	$0	$2	$2	$2					

 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Accounting analysis (continued)

Key invariant: Bank balance never drops below 0. Thus, the sum of the amortized costs provides an upper bound on the sum of the true costs.

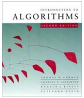
i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1
$\hat{c}_i$	2*	3	3	3	3	3	3	3	3	3
$bank_i$	1	2	2	4	2	4	6	8	2	4

*Okay, so I lied. The first operation costs only $2, not $3.

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

Array resizing: potential method

1. Recall actual cost
2. Define potential function
    - Ensure that $\Phi(D_0) \leq \Phi(D_i)$ for all $i > 0$.
3. Determine the amortised cost of each operation from Φ.



Potential analysis of table doubling

Define the potential of the table after the ith insertion by $\Phi(D_i) = 2i - 2^{\lceil \lg i \rceil}$. (Assume that $2^{\lceil \lg 0 \rceil} = 0$.)

Note:

- $\Phi(D_0) = 0$,
- $\Phi(D_i) \geq 0$ for all i.

Example:

•	•	•	•	•	•		

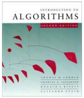
$\Phi = 2 \cdot 6 - 2^3 = 4$

(

\$0	\$0	\$0	\$0	\$2	\$2		

accounting method)

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

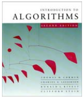


Calculation of amortized costs

The amortized cost of the i th insertion is

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Calculation of amortized costs

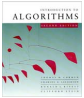
The amortized cost of the i th insertion is

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

$$= \left\{ \begin{array}{l} i \ \text{ if } i-1 \text{ is an exact power of } 2, \\ 1 \ \text{ otherwise;} \end{array} \right\}$$

$$+ \left(2i - 2^{\lceil \lg i \rceil}\right) - \left(2(i-1) - 2^{\lceil \lg (i-1) \rceil}\right)$$

 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

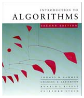


Calculation of amortized costs

The amortized cost of the i th insertion is

$$
\begin{aligned}
\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&= \left\{ \begin{array}{ll} i & \text{if } i-1 \text{ is an exact power of } 2, \\ 1 & \text{otherwise;} \end{array} \right\} \\
&\quad + \left(2i - 2^{\lceil \lg i \rceil}\right) - \left(2(i-1) - 2^{\lceil \lg (i-1) \rceil}\right) \\
&= \left\{ \begin{array}{ll} i & \text{if } i-1 \text{ is an exact power of } 2, \\ 1 & \text{otherwise;} \end{array} \right\} \\
&\quad + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil} .
\end{aligned}
$$

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

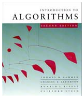


Calculation

Case 1: $i - 1$ is an exact power of 2.

$$\hat{c}_i = i + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil}$$

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Calculation

Case 1: $i-1$ is an exact power of 2.

$$\begin{aligned}\hat{c}_i &= i + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil} \\ &= i + 2 - 2(i-1) + (i-1)\end{aligned}$$

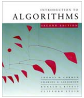
 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

Calculation

Case 1: $i - 1$ is an exact power of 2.

$$
\begin{aligned}
\hat{c}_i &= i + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil} \\
&= i + 2 - 2(i - 1) + (i - 1) \\
&= i + 2 - 2i + 2 + i - 1
\end{aligned}
$$

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

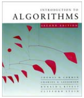


Calculation

Case 1: $i-1$ is an exact power of 2.

$$
\begin{aligned}
\hat{c}_i &= i + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil} \\
&= i + 2 - 2(i-1) + (i-1) \\
&= i + 2 - 2i + 2 + i - 1 \\
&= 3
\end{aligned}
$$

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Calculation

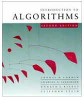
Case 1: $i - 1$ is an exact power of 2.

$$\begin{aligned}\hat{c}_i &= i + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil} \\ &= i + 2 - 2(i-1) + (i-1) \\ &= i + 2 - 2i + 2 + i - 1 \\ &= 3\end{aligned}$$

Case 2: $i - 1$ is *not* an exact power of 2.

$$\hat{c}_i = 1 + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil}$$

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Calculation

Case 1: $i - 1$ is an exact power of 2.

$$\begin{aligned}\hat{c}_i &= i + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil} \\ &= i + 2 - 2(i - 1) + (i - 1) \\ &= i + 2 - 2i + 2 + i - 1 \\ &= 3\end{aligned}$$

Case 2: $i - 1$ is *not* an exact power of 2.

$$\begin{aligned}\hat{c}_i &= 1 + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil} \\ &= 3 \qquad \text{(since } 2^{\lceil \lg i \rceil} = 2^{\lceil \lg (i-1) \rceil}\text{)}\end{aligned}$$

 Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

Calculation

Case 1: $i - 1$ is an exact power of 2.

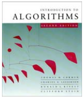
$$\begin{aligned}\hat{c}_i &= i + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil} \\ &= i + 2 - 2(i-1) + (i-1) \\ &= i + 2 - 2i + 2 + i - 1 \\ &= 3\end{aligned}$$

Case 2: $i - 1$ is *not* an exact power of 2.

$$\begin{aligned}\hat{c}_i &= 1 + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil} \\ &= 3\end{aligned}$$

Therefore, n insertions cost $\Theta(n)$ in the worst case.

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson



Calculation

Case 1: $i - 1$ is an exact power of 2.

$$
\begin{aligned}
\hat{c}_i &= i + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil} \\
&= i + 2 - 2(i - 1) + (i - 1) \\
&= i + 2 - 2i + 2 + i - 1 \\
&= 3
\end{aligned}
$$

Case 2: $i - 1$ is *not* an exact power of 2.

$$
\begin{aligned}
\hat{c}_i &= 1 + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil} \\
&= 3
\end{aligned}
$$

Therefore, n insertions cost $\Theta(n)$ in the worst case.

Exercise: Fix the bug in this analysis to show that the amortized cost of the first insertion is only 2.

Copyright © 2001-5 by Erik D. Demaine and Charles E. Leiserson

Recap

- Amortised analysis
  1. Aggregate method
  2. Accounting method
  3. Potential method
- For when you are implementing a *data structure* and a set of operations that modify/query it, rather than a specific operation (such as sorting a list).
- Find an upper bound on the complexity (cost) which still gives the desired result
- Examples:
  1. Stack operations
  2. Incrementing a binary counter
  3. Resizing arrays
- Accounting and potential method are ways of structuring proofs (operation- or data-structure-focused, respectively), which are essentially the aggregate method underneath