CS 3110

Lecture 16: Amortized Analysis

Prof. Clarkson

Spring 2015

Today's music: "Money, Money, Money" by ABBA

Review

Current topic: Reasoning about performance

- Efficiency
- Big Oh
- Recurrences

Today:

- Alternative notions of efficiency
- Amortized analysis
  - Efficiency of data abstractions, not just individual functions

Question #1

How much of PS3 have you finished?

A. None
B. About 25%
C. About 50%
D. About 75%
E. I'm done!!!

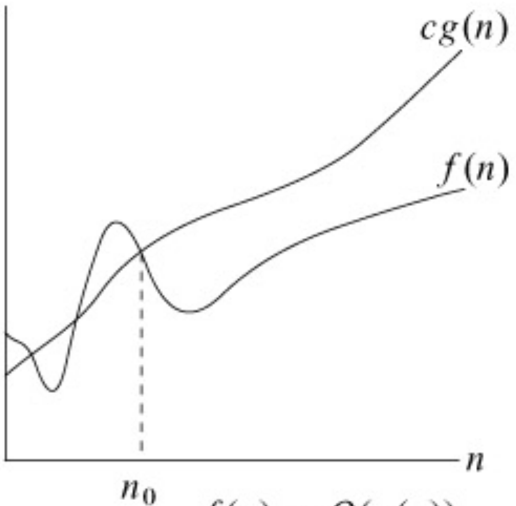
Review: What is "efficiency"?

Final attempt: An algorithm is efficient if its worst-case running time on input of size N is O(N^d) for some constant d.

Asymptotic bounds

Big Oh:

- *asymptotic upper bound*
- O(g) = {f | exists c>0, n0>0, forall n >= n0, f(n) <= c * g(n)}
- intuitions: f <= g, f is at least as efficient as g



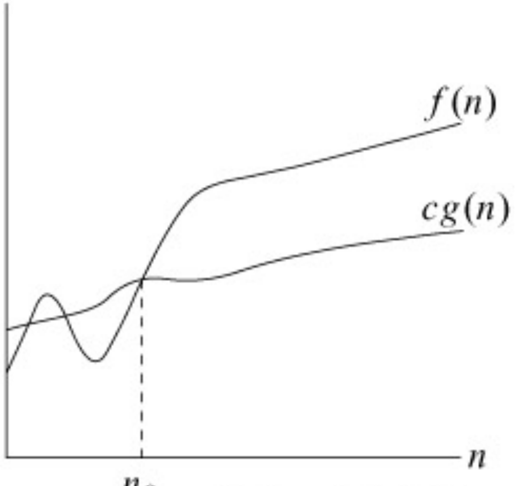
$f(n) = O(g(n))$

(b)

Asymptotic bounds

Big Omega

- *asymptotic lower bound*
- Ω(g) = {f | exists c>0, n0>0, forall n >= n0, f(n) >= c * g(n)}
- intuitions: f >= g, f is at most as efficient as g



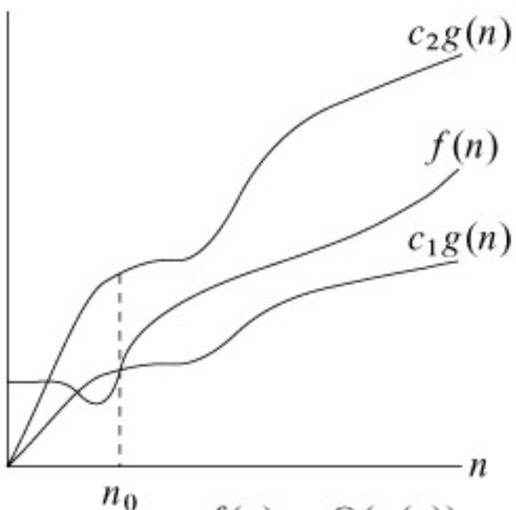
$f(n) = \Omega(g(n))$

(c)

Asymptotic bounds

Big Theta

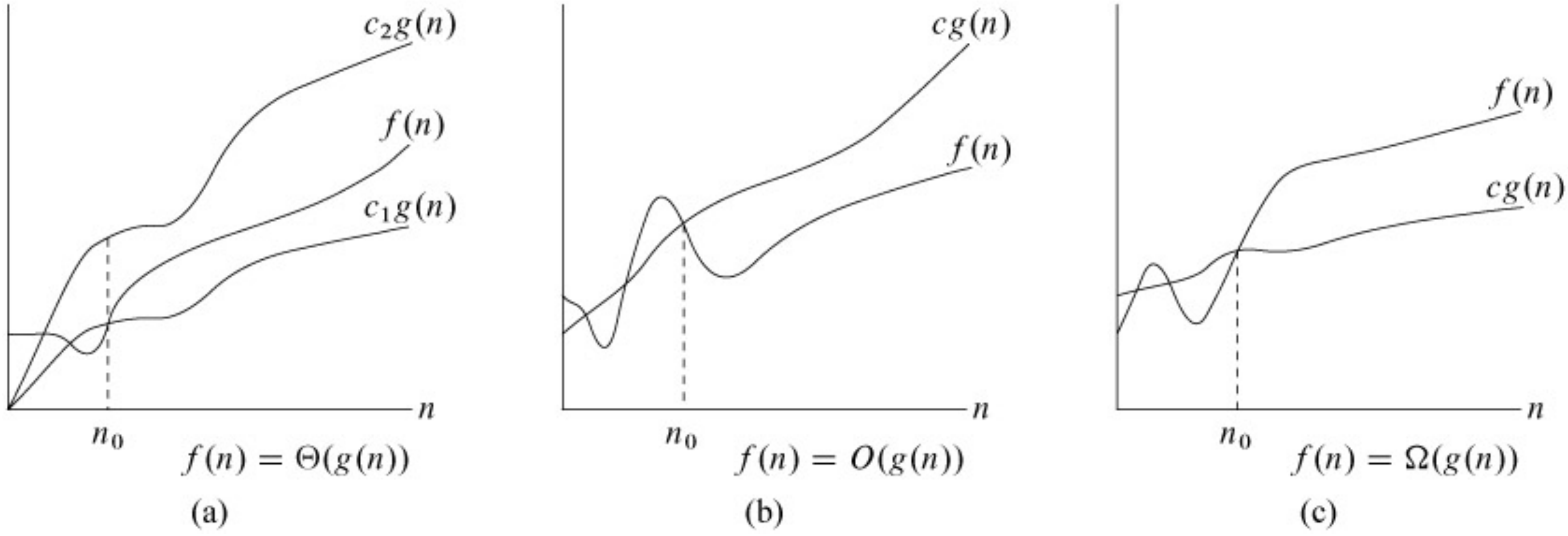
- *asymptotic tight bound*
- Θ(g) = O(g) ∩ Ω(g)
- Θ(g) = {f | exists c1>0, c2>0, n0>0, forall n >= n0,
  c1 * g(n) <= f(n) <= c2 * g(n)}
- intuitions: f = g, f is just as efficient as g
- beware: some authors write O(g) when they really mean Θ(g)



$f(n) = \Theta(g(n))$

(a)

Asymptotic bounds



[Cormen et al. *Introduction to Algorithms*, 3rd ed, 2009]

Alternative notions of efficiency

- Expected-case running time
  - Instead of worst case
  - Useful for randomized algorithms
  - Maybe less useful for deterministic algorithms
    - Unless you really do know something about probability distribution of inputs
    - All inputs are probably not equally likely
- Space
  - How much memory is used? Cache space? Disk space?
- Other resources
  - Power, network bandwidth, ...
- Efficiency of an entire data abstraction...

Stacks with multipop

```
module type STACK = sig
  type 'a t
  exception Empty

  val empty : 'a t
  val is_empty : 'a t -> bool
  val push : 'a -> 'a t  -> 'a t
  val peek : 'a t -> 'a
  val pop : 'a t -> 'a t
  val multipop : int -> 'a t  -> 'a t
end
```

Stacks with multipop

```
module Stack : STACK = struct
  type 'a t = 'a list
  exception Empty

  let empty = []
  let is_empty s = s = []
  let push x s = x :: s
  ...
```

Stacks with multipop

```
module Stack : STACK = struct
  type 'a t = 'a list
  exception Empty

  let empty = []                (* O(1) *)
  let is_empty s = s = [] (* O(1) *)
  let push x s = x :: s   (* O(1) *)
  ...
```

Stacks with multipop

```
module Stack : STACK = struct
  ...
  let peek = function
   | []    -> raise Empty
   | x::xs -> x
  let pop = function
   | []    -> raise Empty
   | x::xs -> xs
  ...
```

Stacks with multipop

```
module Stack : STACK = struct
  ...
  let peek = function          (* O(1) *)
   | []    -> raise Empty
   | x::xs -> x
  let pop = function           (* O(1) *)
   | []    -> raise Empty
   | x::xs -> xs
  ...
```

Stacks with multipop

```
module Stack : STACK = struct
  ...
  let multipop k s =
   let rec repeat m f x =
     if m=0 then x
         else repeat (m-1) f (f x)
   in repeat k pop s
end
```

Stacks with multipop

```
module Stack : STACK = struct
  ...
  let multipop k s =
   let rec repeat m f x =
     if m=0 then x
         else repeat (m-1) f (f x)
   in repeat k pop s
  (* O(min(k, |s|))
   * which is O(n) where n = |s|*)
end
```

Question #2

- Start with an initially empty stack
- Do a sequence of STACK operations
- Suppose maximum length stack ever reaches is *n*
- Suppose (coincidentally) that the sequence of operations is of length *n*
- **What is worst-case running time of entire sequence?**

A. O(1)
B. O(n)
C. O(n log n)
D. O(n^2)
E. O(2^n)

Question #2

- Start with an initially empty stack
- Do a sequence of STACK operations
- Suppose maximum length stack ever reaches is *n*
- Suppose (coincidentally) that the sequence of operations is of length *n*
- **What is worst-case running time of entire sequence?**

A. O(1)
B. O(n)
C. O(n log n)
D. O(n^2) *possible answer*
E. O(2^n)

Why?

- n operations
- each is O(n)
- n*O(n) = O(n^2)

...that's correct but pessimistic

Improved analysis of efficiency

- Consider the average cost of each operation in the sequence, still in the worst case
  - average = arithmetic mean = T(n)/n
    - where T(n) is total worst-case cost of n operations
  - average <> expected value of random variable

Improved analysis of efficiency

- **Fact:** each value pushed onto stack can be popped off at most once
  - In a sequence of *n* operations, can't be more than *n* calls to **`push`**
  - So can't be more than *n* calls to **`pop`**, including calls **`multipop`** makes to **`pop`**
  - Each of those calls to **`push`** and **`pop`** is O(1)
- So worst-case running time of entire sequence is T(n) = n * O(1) = O(n)
- And average worst-case running time of each operation in sequence is T(n)/n = O(n)/n = O(1)

A monetary analysis

- Real cost:
  - `push`: $1
  - `pop`: $1
  - `multipop`: $min(k, |s|)
- Let's engage in some "creative accounting"
- Billed cost:
  - `push`: $2
  - `pop`: $0
  - `multipop`: $0
- **Fact:** we can use billed cost to pay the real cost of any sequence of operations

A monetary analysis

Operation	Stack after op	Real cost	Billed cost
`push`	`[x]`	1	2
`push`	`[y;x]`	1	2
`pop`	`[x]`	1	0
`push`	`[z;x]`	1	2
`push`	`[a;z;x]`	1	2
`multipop 2`	`[x]`	2	0
`push`	`[b;x]`	1	2
`multipop 3`	`Empty`	2	0
TOTAL		10	10

A monetary analysis

- Cost of **`push`**:
  - $2 billed
  - use $1 of that to pay the real cost
  - save an extra $1 in that element's "bank account"
- Cost of **`pop`**:
  - $0 billed
  - use the saved $1 in that element's account to pay the real cost
- Cost of **`multipop`**:
  - (see **`pop`**)
- So cost of any operation is O(1)
  - Because 2 and 0 are both O(1)
- These costs are called *amortized costs*

A monetary analysis

- Amortized cost of **`push`**:
  - $2 billed
  - use $1 of that to pay the real cost
  - save an extra $1 in that element's "bank account"
- Amortized cost of **`pop`**:
  - $0 billed
  - use the saved $1 in that element's account to pay the real cost
- Amortized cost of **`multipop`**:
  - (see **`pop`**)
- So amortized cost of any operation is O(1)
  - Because 2 and 0 are both O(1)
- These costs are called *amortized costs*

Amortized analysis of efficiency

- *Amortize:* put aside money at intervals for gradual payment of debt [Webster's 1964]
  - *L. "mort-" as in "death"*
- Pay extra money for some operations as a *credit*
- Use that credit to pay higher cost of some later operations
- a.k.a. *banker's method* and *accounting method*
- Invented by Sleator and Tarjan (1985)

Robert Tarjan

b. 1948

Turing Award Winner (1986) with Prof. John Hopcroft

For fundamental achievements in the design and analysis of algorithms and data structures.

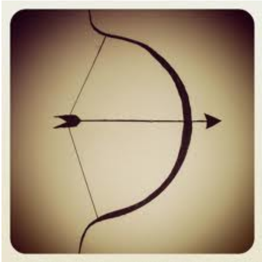
Cornell CS faculty 1972-1973

Another kind of amortized analysis

- Banker's method required tracking credit from sequence of operations
- Alternative idea:
  - determine amount of credit available just from state of data structure, not from its history
  - i.e., "let's ignore history"
- Leads to *physicist's method* a.k.a. *potential method*

Physicist's method

- Potential energy: stored energy of position possessed by an object
  - drawn bow
  - stretched spring
  - child on playground at height of swing



- Suppose we have function U(d) giving us the "potential energy" stored in a data structure
- We'll use that stored energy to pay for expensive operations

Physicist's method

- Suppose operation changes data structure from d0 to d1
- Define amortized cost of operation to be
  = realcost(op) + U(d1) – U(d0)
- Amortized cost of sequence of two operations
  = realcost(op1) + U(d1) – U(d0)
  + realcost(op2) + U(d2) – U(d1)
  = realcost(op1) + realcost(op2) + U(d2) – U(d0)
- Amortized cost of sequence of *n* operations
  = $[\sum_{i=1..n} (\text{realcost}(op\_i))] + U(dn) - U(d0)$
- *Telescoping sum:* intermediate potentials cancel out; we can ignore them in analysis

A physical analysis

Potential of stack is length of list: U(s) = length(s)

Operation	Stack after op	Real cost	U(s)
`---`	`[]`	---	0
`push`	`[x]`	1	1
`push`	`[y;x]`	1	2
`pop`	`[x]`	1	1
`push`	`[z;x]`	1	2
`push`	`[a;z;x]`	1	3
`multipop 2`	`[x]`	2	1
`push`	`[b;x]`	1	2
`multipop 3`	`Empty`	2	0
TOTAL		**10**	**---**

A physical analysis

- Amortized cost of **`push`**:
  - real cost is 1
  - change in potential is 1
    - because U(**`x::s`**) – U(**`s`**) = 1
  - so amortized cost is 2 = O(1)

A physical analysis

- Amortized cost of **`pop`**:
  - real cost is 1
  - change in potential is −1
    - because U(**`s`**) – U(**`x::s`**) = −1
  - so amortized cost is 0 = O(1)

A physical analysis

- Amortized cost of **`multipop`**:
  - real cost is min(k, |s|)
  - change in potential is also -min(k, |s|)
  - so amortized cost is 0 = O(1)
- So amortized cost of any operation is O(1)

Recall from Lec14: Hash tables

- If load factor gets too high, make the array bigger, thus reducing load factor
  - OCaml **`Hashtbl`** and **`java.util.HashMap`**: if load factor > 2.0 then double array size, bringing load factor back to around 1.0
  - Rehash elements into new buckets
  - Efficiency:
    - **`insert`**: O(1)
    - **`find`** & **`remove`**: O(2), which is O(1)
    - rehashing: arguably still constant time; will return to this later in course
- If load factor gets too small (hence memory is being wasted), could shrink the array, thus increasing load factor
  - Neither OCaml nor Java do this

Hash tables: physicist's method

- Simplifying assumptions:
  - no **`remove`** operation
  - ignore cost of all operations until load factor reaches 1 for the first time
- Potential: $U(h) = 4(n - m)$
  - where n is number of elements in h
  - and m is number of buckets in h
  - Causes potential to increase as load factor (=n/m) grows
  - When load factor is 1, it holds that m=n, so $U(h) = 0$
    - no extra credit stored up immediately after resize
  - When load factor is 2, it holds that m=n/2, so $U(h) = 2n$
    - enough extra credit stored up to pay to rehash and insert each element just when we need to resize

Hash tables: physicist's method

- Amortized cost of **`insert`** (including resize)
  - Let n be # elements and m be # buckets before insert
  - If no resize is triggered:
    - Cost of 1 each to hash and insert element
    - Change in potential = $4(n+1-m) - 4(n - m) = 4n + 4 - 4m - 4n + 4m = 4$
    - Amortized cost = $1 + 1 + 4 = 6 = O(1)$

Hash tables: physicist's method

- Amortized cost of `insert` (including resize)
  - Let n be # elements and m be # buckets before insert
  - If resize is triggered:
    - Then n+1 = 2m
    - Cost of 2(n+1) to hash and insert n+1 elements
    - Change in potential = 4(n+1 – 2m) – 4(n – m) = 4n + 4 – 8m – 4n + 4m = 4 – 4m = 4 – 2(2m) = 4 – 2(n+1) = 4 – 2n – 2
    - Amortized cost = 2(n + 1) + 4 – 2n – 2 = 2n + 2 + 4 – 2n – 2 = 4 = O(1)
- Whether resize occurs or not, amortized cost of O(1)

Hash tables: physicist's method

- Suppose we did have **`remove`** operation
  - Cost of remove itself is 1 to hash
  - Plus expected worst-case time of at most 2 to delete element from bucket
    - because load factor is at most 2
  - Potential: $U(h) = \max(4(n - m), 0)$
    - No "negative potential" or "negative credit": always pay for expensive operations in advance, otherwise might end a sequence without ever paying off debt
  - Analysis of insert proceeds as before

- Conclusion: resizing hash tables have *amortized expected worst-case running time* that is constant!