CS 140
Boerkoel & Chen

Midterm

- Will receive it at the end of this review section!
- You will have 2 hours to take it
  - watch your time!
  - if you get stuck on a problem, move on and come back
- Must take it and turn it in by 2:45 PM on Tuesday (next lecture)
- You may use:
  - ONLY a double-sided page of notes
- Do NOT discuss it with anyone until after Wednesday at 4pm

What to Expect

- Lighter-weight versions of homework style questions
- All topics are fair game
- Important topics
  - Course objectives
  - Topics repeated in lecture, homeworks, etc.

Midterm

- General
  - what is an algorithm?
  - algorithm properties
  - pseudocode
  - proving correctness
    - loop invariants
  - run time analysis

Midterm

- Asymptotic notation
  - proving bounds
  - ranking/ordering of functions
- Amortized analysis
- Recurrences
  - solving recurrences
    - substitution method
    - recursion-tree
    - master method

Midterm

- Sorting
  - insertion sort
  - merge sort
    - merge function
  - quick sort
    - partition function
  - heap sort

Midterm

- Divide and conquer
  - divide up the data (often in half)
  - recurse
  - possibly do some work to combine the answer
- Calculating order statistics/medians
- Basic data structures
  - set operations
  - array
  - linked lists
  - stacks
  - queues

Midterm

Abstract Data Types (ADT)

- Heaps
  - Binary heaps
  - Binomial heaps
- Search trees
  - BSTs
  - Red-black
- Disjoint sets (very briefly)
  - Linked list
  - Forests

Midterm

- Other things to know:
  - Run-times
  - When to use an algorithm
  - Proof techniques
    - Look again at proofs by induction
      - Make sure to follow the explicit form we covered in class
    - Proof by contradiction
    - Read WritingProofs.pdf (Piazza)

Proofs

- Prove by induction: 1+x^n >= 1+nx for all nonnegative integers n and all x >= −1.

-- Base case

-- Inductive case

--- Inductive hypothesis

--- Proof of inductive step (s)

Proofs

- Prove by contradiction:

For all integers n, if n+2 is odd, then n is odd.

Big O: Upper bound

O(g(n)) is the set of functions:

$$O(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

Provides an upper bound on runtime for algs!

Omega: Lower bound

Ω(g(n)) is the set of functions:

$$\Omega(g(n)) = \left\{ f(n): \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

Provides a general lower bound for difficulty of a problem!

Theta: Upper and lower bound

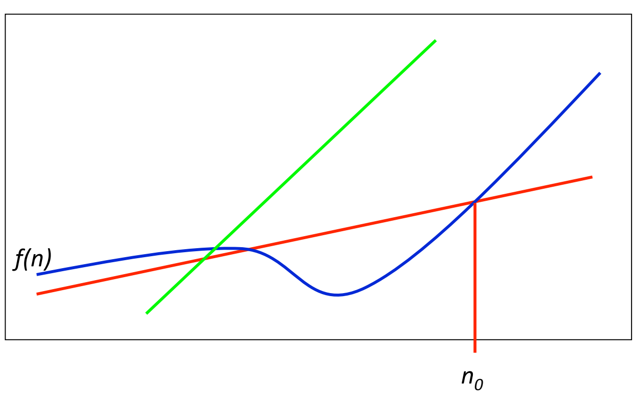
Θ(g(n)) is the set of functions:

$$\Theta(g(n)) = \left\{ f(n): \begin{array}{l} \text{there exists positive constants } c_1, c_2 \text{ and } n_0 \text{ such that} \\ 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

We can bound the function *f(n)* above **and** below by some constant factor of *g(n)* (though different constants)

Note: A function is theta bounded **iff** it is big O bounded and Omega bounded

Visually: lower bound



worst-case vs. best-case vs. average-case

worst-case: what is the worst the running time of the algorithm can be?

best-case: what is the best the running time of the algorithm can be?

average-case: given random data, what is the running time of the algorithm?

Don't confuse this with O, Ω and Θ. The cases above are *situations*, asymptotic notation is about bounding particular situations

Proving bounds: find constants that satisfy inequalities

Show that $5n^2 - 15n + 100$ is $\Theta(n^2)$

Step 1: Prove $O(n^2)$ – Find constants c and n_0 such that
$5n^2 - 15n + 100 \le cn^2$ for all $n > n_0$

$$cn^2 \ge 5n^2 - 15n + 100$$
$$c \ge 5 - 15/n + 100/n^2$$

Let $n_0 = 1$ and $c = 5 + 100 = 105$.
$100/n^2$ only get smaller as n increases and we ignore $-15/n$ since it only varies between -15 and 0

Proving bounds

Step 2: Prove $\Omega(n^2)$ – Find constants c and n_0 such that
$5n^2 - 15n + 100 \ge cn^2$ for all $n > n_0$

$$cn^2 \le 5n^2 - 15n + 100$$
$$c \le 5 - 15/n + 100/n^2$$

Let $n_0 = 4$ and $c = 5 - 15/4 = 1.25$ (or anything less than 1.25). $15/n$ is always decreasing and we ignore $100/n^2$ since it is always between 0 and 100.

Disproving bounds

Is $5n^2$ $O(n)$?

$$O(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

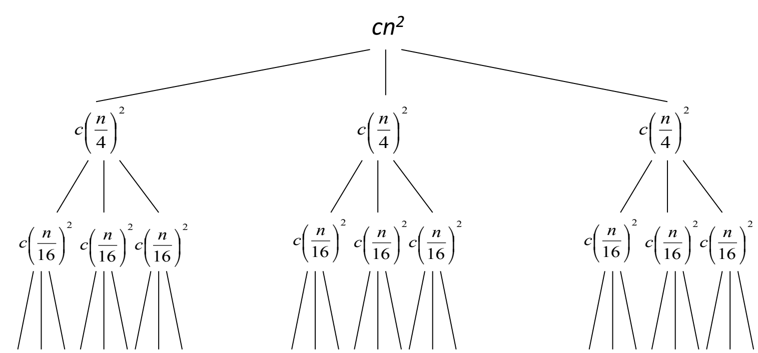
Assume it's true.

That means there exists some c and n_0 such that

$$5n^2 \le cn \text{ for } n > n_0$$
$$5n \le c$$ contradiction!

$$T(n) = 3T(n/4) + n^2$$

cost



cn^2

$3/16cn^2$

$(3/16)^2cn^2$

What is the cost at each level? $\left(\frac{3}{16}\right)^d cn^2$

What is the depth of the tree?

At each level, the size of the data is divided by 4

$$\frac{n}{4^d} = 1$$

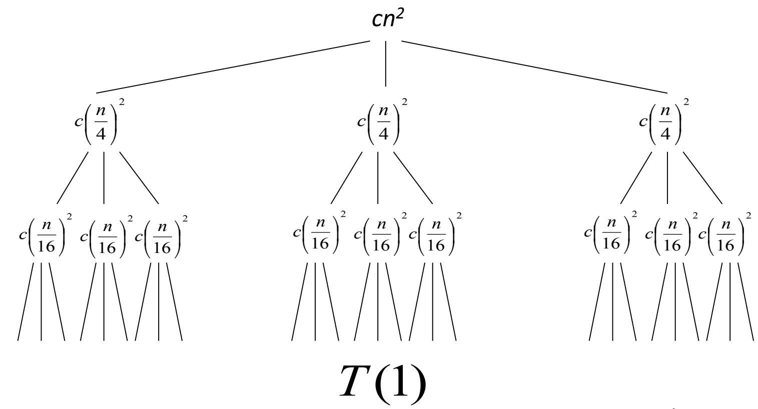
$$\log\left(\frac{n}{4^d}\right) = 0$$

$$\log n - \log 4^d = 0$$

$$d \log 4 = \log n$$

$$d = \log_4 n$$

$$T(n) = 3T(n/4) + n^2$$



How many leaves are there? $3^d = 3^{\log_4 n}$

Total cost

$$T(n) = \frac{16}{13}cn^2 + \Theta(3^{\log_4 n})$$

$$3^{\log_4 n} = 4^{\log_4 3^{\log_4 n}}$$

$$= 4^{\log_4 n \log_4 3}$$

$$= 4^{\log_4 n^{\log_4 3}}$$

$$= n^{\log_4 3}$$

$$T(n) = \frac{16}{13}cn^2 + \Theta(n^{\log_4 3})$$

$$T(n) = O(n^2)$$

Amortized analysis

What does “amortize” mean?

am·or·tized | **am·or·tiz·ing**

Definition of AMORTIZE

1 : to pay off (as a mortgage) gradually usually by periodic payments of principal and interest or by payments to a sinking fund

2 : to gradually reduce or write off the cost or value of (as an asset) <*amortize* goodwill> <*amortize* machinery>

— **am·or·tiz·able** *adjective*

Amortized analysis

There are many situations where the worst case running time is bad

However, if we average the operations over *n* operations, the average time is more reasonable

This is called *amortized* analysis

- This is different than average-case running time, which requires probabilistic reasoning about input
- The worst case running time doesn't change

From 4b: Dictionaries

We want to support fast lookup and insertion (i.e. faster than linear)

Arrays can easily made to be fast for one or the other

- fast search: keep list sorted
  - O(n) insert
  - O(log n) search
- fast insert: extensible array
  - O(1) insert (amortized)
  - O(n) search

From 4b: Dictionaries

Idea: store data in a collection of arrays

- array *i* has size 2^i
- an array is either full or empty (never partially full)
- each array is stored in sorted order
- no relationship between arrays

From 4b: Dictionaries

Which arrays are full and empty are based on the number of elements

- specifically, binary representation of the number of elements
- 4 items = 100 = A2-full, A1-empty, A_0-empty
- 11 items = 1011 = A_3-full, A_2-empty, A_1-full, A_0-full

A_0: [5]
A_1: [4, 8]
A_2: empty
A_3: [2, 6, 9, 12, 13, 16, 20, 25]

Lookup: binary search through each array

- Worse case runtime?

From 4b: Dictionaries

A_0: [5]
A_1: [4, 8]
A_2: empty
A_3: [2, 6, 9, 12, 13, 16, 20, 25]

Lookup: binary search through each array

Worse case: all arrays are full

- number of arrays = number of digits = log n
- binary search cost for each array = O(log n)
- O(log n log n)

From 4b: Dictionaries

Insert(A, item)

- starting at i = 0
- current = [item]
- as long as the level *i* is full
  - merge current with A_i using *merge* procedure
  - store to current
  - A_i = empty
  - i++
- A_i = current

Insert running time

Worse case

- merge at each level
- 2 + 4 + 8 + … + n/2 + n = O(n)

There are many insertions that won't fall into this worse case

What is the amortized worse case for insertion?

insert: amortized analysis

- Consider inserting *n* numbers

	times	cost
– how many times will A_0 be empty?	n/2	O(1)
– how many times will we need to merge with A_0?	n/2	2
– how many times will we need to merge with A_1?	n/4	4
– how many times will we need to merge with A_2?	n/8	8
– …		
– how many times will we need to merge with $A_{\log n}$?	1	n

total cost: log n levels * O(n) each level

O(n log n) cost for *n* inserts

O(log n) amortized cost!

Accounting Method

- Idea:
  - Some operations are charged an *amortized cost* that is more than actual cost
  - Store difference to specific item in data structure as **credit**
  - Use credit to pay for when actual cost > amortized cost
- Key points:
  - In accounting method, different operations have different costs
  - Credit must never go negative
    - Otherwise amortized cost is not an upper bound on actual cost
    - Amortized cost would tell us nothing

Accounting Method Overview

Let c_i = actual cost of ith operation ,

$\hat{c}_i$ = amortized cost of ith operation .

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

$$\text{Total credit stored} = \sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \underbrace{\geq}_{\text{had better be}} 0 .$$

Accounting Method: Multipop Stack

Let c_i = actual cost of ith operation ,

$\hat{c}_i$ = amortized cost of ith operation .

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

$$\text{Total credit stored} = \sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \underbrace{\geq}_{\text{had better be}} 0 .$$

Stack

operation	actual cost	amortized cost
PUSH	1	?
POP	1	?
MULTIPOP	$\min(k, s)$	?

- Total amortized cost:
- Upper bound on actual cost!

Accounting Method: Binary Counter

Let c_i = actual cost of ith operation ,

$\hat{c}_i$ = amortized cost of ith operation .

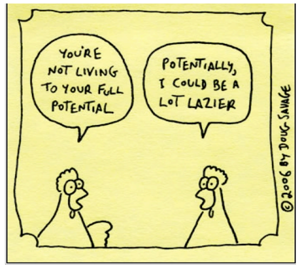
Then require $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$ for *all* sequences of n operations.

$$\text{Total credit stored} = \sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \underbrace{\geq}_{\text{had better be}} 0 .$$

	Actual cost	Amortized cost
$A[k] \rightarrow 1$	1	?
$A[k] \rightarrow 0$	1	?

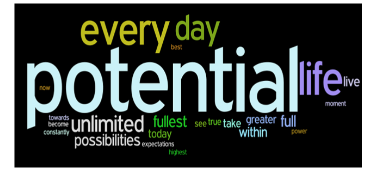
- Total amortized cost:
- Upper bound on actual cost!

Potential Method



- Similar to accounting method, but we think of credit as **potential** stored with the entire data structure
- Key Ideas:
  - Accounting method stores *credit* with *specific objects*
  - Potential method stores *potential* in the *data structure* as a whole
  - Can release potential to pay for future operations
  - Most flexible of the amortized analysis

The Potential Function



Potential function $\Phi : D_i \to \mathbb{R}$

$\Phi(D_i)$ is the *potential* associated with data structure D_i.

$$\begin{aligned} \hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= c_i + \underbrace{\Delta\Phi(D_i)}_{\text{increase in potential due to } i\text{th operation}} . \end{aligned}$$

$$\begin{aligned} D_i &= \text{data structure after } i\text{th operation} , \\ D_0 &= \text{initial data structure} , \\ c_i &= \text{actual cost of } i\text{th operation} , \\ \hat{c}_i &= \text{amortized cost of } i\text{th operation} . \end{aligned}$$

The (amortized) costs of having potential

Potential function $\Phi : D_i \to \mathbb{R}$

$\Phi(D_i)$ is the *potential* associated with data structure D_i.

$$\begin{aligned} \hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= c_i + \underbrace{\Delta\Phi(D_i)}_{\text{increase in potential due to } i\text{th operation}} . \end{aligned}$$

$$\begin{aligned} \text{Total amortized cost} &= \sum_{i=1}^{n} \hat{c}_i \\ &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \end{aligned}$$

(telescoping sum: every term other than D_0 and D_n is added once and subtracted once)

$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) .$$

The (amortized) costs of having potential

Potential function $\Phi : D_i \to \mathbb{R}$

$\Phi(D_i)$ is the *potential* associated with data structure D_i.

$$\begin{aligned} \hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= c_i + \underbrace{\Delta\Phi(D_i)}_{\text{increase in potential due to } i\text{th operation}} . \end{aligned}$$

- If we require:

$$\Phi(D_i) \geq \Phi(D_0) \forall i$$

then amortized cost is always an upper bound on actual cost

- In practice:

$$\Phi(D_0) = 0; \Phi(D_i) \geq 0 \forall i$$

Potential Method: Stack

$$\Phi = \text{\# of objects in stack}$$

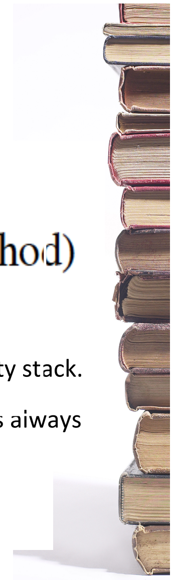
$$(= \text{\# of \$1 bills in accounting method})$$

$$\Phi(D_0) = 0\,?$$

Yes! D_0 represents an empty stack.

$$\Phi(D_i) \geq 0 \,\forall i\,?$$

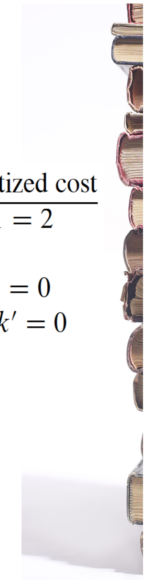
Yes! *# of* objects on stack is always



Potential Method: Stack

operation	actual cost	$\Delta\Phi$	amortized cost
PUSH	1	$(s+1) - s = 1$ where s = # of objects initially	$1 + 1 = 2$
POP	1	$(s-1) - s = -1$	$1 - 1 = 0$
MULTIPOP	$k' = \min(k, s)$	$(s - k') - s = -k'$	$k' - k' = 0$

Therefore, amortized cost of a sequence of n operations $= O(n)$.



Potential Method: Binary Counting

$$\Phi = b_i = \text{\# of 1's after } i\text{th INCREMENT}$$

- Suppose i^{th} operation resets t_i bits to 0
- $c_i \leq t_i + 1$ (resets t_i bits, sets ≤ *1 bit to 1)*
- $b_i \leq b_{i-1} - t_i + 1$

$$\begin{aligned}\Delta\Phi(D_i) &\leq (b_{i-1} - t_i + 1) - b_{i-1} \\ &= 1 - t_i\,.\end{aligned}$$

$$\begin{aligned}\hat{c}_i &= c_i + \Delta\Phi(D_i) \\ &\leq (t_i + 1) + (1 - t_i) \\ &= 2\,.\end{aligned}$$

If counter starts at 0, $\Phi(D_0) = 0$.

Therefore, amortized cost of n operations $= O(n)$.

Algorithms Review!
Computer Science 140
Spring 2013
Boerkoel & Chen

Problem 1: Sounds Like a Duck!
A *quack* is an abstract data type that is part queue and part stack. Specifically, a quack supports the following three operations:

- PUSH(x) takes a value x and puts it at the "top" of the stack;
- POP() removes and returns the element at the "top" of the stack.; and
- DEQUEUE() removes and returns the item at the "bottom" of the stack (the element that has been in the stack the longest).

Your goal is to implement a quack using only stacks. However, in order to conserve memory, *each item can only appear on one of the three stacks at any given time!*

a. Describe an algorithm that implements a quack using three stacks such that *any* sequence of n operations takes a total of $O(n)$ time. (You don't need to do the amortized analysis in this part, just describe the algorithm.)

b. Prove that your algorithm takes $O(n)$ time for any n sequence of n operations using an accounting argument.

c. Prove that your algorithm takes $O(n)$ time for any n sequence of n operations using a potential method argument.

Problem 2: Max Priori-d Heap

A max priori-d heap supports the typical operations of a priority queue ADT:

- INSERT(S, x): inserts the element x into the set S, which is equivalent to the operation $S = S \cup \{x\}$.
- MAXIMUM(S) returns the element of S with the largest key.
- EXTRACT-MAX(S) removes and returns the element of S with the largest key.
- INCREASE-KEY(S, x, k) increases the value of element xs key to the new value k, which is assumed to be at least as large as xs current key value.

A d-ary heap is like a binary heap, but (with one possible exception) non-leaf nodes have d children instead of 2 children.

a. How would you represent a d-ary heap in an array?

b. What is the height of a d-ary heap of n elements in terms of n and d?

c. Give an efficient implementation of EXTRACT-MAX in a d-ary max-heap. Analyze its running time in terms of d and n.

d. Give an efficient implementation of INSERT in a d-ary max-heap. Analyze its running time in terms of d and n.

e. Give an efficient implementation of INCREASE-KEY, which flags an error if $k < A[i]$, but otherwise sets $A[i] = k$ and then updates the d-ary maxheap structure appropriately. Analyze its running time in terms of d and n.

Problem 3: All things being equal, I'd prefer my sort Quick
ur in class analysis of the expected running time of randomized quicksort assumes that all element values are distinct. In this problem, we examine what happens when they are not.

a. Suppose that all element values are equal. What would be randomized quicksorts running time in this case?

b. The PARTITION procedure returns an index q such that each element of $A[p..q-1]$ is less than or equal to $A[q]$ and each element of $A[q+1..r]$ is greater than $A[q]$. Modify the PARTITION procedure to produce a procedure PARTITION'A, p, r, which permutes the elements of $A[p..r]$ and returns two indices q and t, where $p \leq q \leq t \leq r$ such that

- all elements of $A[q..t]$ are equal,
- each element of $A[p..q-1]$ is less than $A[q]$, and
- each element of $A[t+1..r]$ is greater than $A[q]$.

Like PARTITION, your PARTITION' should take $\Theta(r-p)$.

c. Modify the QUICKSORT procedure to QUICKSORT', which calls RANDOMIZED-PARTITION' and recurses only on partitions of elements not known to be equal to each other.

d. Using QUICKSORT', how would you adjust the analysis in Section 7.4.2 to avoid the assumption that all elements are distinct?