IE170: Algorithms in Systems Engineering: Lecture 10

Jeff Linderoth

Department of Industrial and Systems Engineering
Lehigh University

February 5, 2007

What We've Learned

1. Summation Formulae, Induction and Bounding
2. How to compare functions: $o, \omega, O, \Omega, \Theta$
3. How to count the running time of algorithms
4. How to solve recurrences that occur when we do (3)
5. Data Structures:
   - Hash
   - Binary Search Trees
   - Heap

The World's First Algorithm

Euclid's Algorithm(m, n)

1. Divide m by n and let r be the remainder.
2. If $r = 0$, then $\gcd(m, n) = n$.
3. Otherwise, $\gcd(m, n) = \gcd(n, r)$

Summation Formulae

Arithmetic Series

$$1 + 2 + \cdots + n = \sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

Sum Of Squares

$$\sum_{k=0}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$$

- Often, such formulae can be proved via *mathematical induction*

Geometric Series

$$\sum_{k=0}^{n} x^k = \frac{1-x^{n+1}}{1-x}$$

If $|x| < 1$, then the series converges to

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}.$$

Harmonic Series

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{k} = \sum_{k=1}^{n} \frac{1}{k} \approx \ln(n)$$

Bounding Sums by Integrals

- When f is a (monotonically) increasing function, then we can approximate the sum $\sum_{k=m}^{n} f(k)$ by the integrals:

$$\int_{m-1}^{n} f(x)dx \le \sum_{k=m}^{n} f(k) \le \int_{m}^{n+1} f(x)dx.$$

and a decreasing function can be approximated by

$$\int_{m}^{n+1} f(x)dx \le \sum_{k=m}^{n} f(k) \le \int_{m-1}^{n} f(x)dx$$

O, Ω, Θ definitions

$$\Theta(g) = \{f : \exists\ c_1, c_2, n_0 > 0 \text{ such that } c_1 g(n) \le f(n) \le c_2 g(n)\ \forall n \ge n_0\}$$

$\Omega(g) = \{f \mid \exists$ constants $c, n_0 > 0$ s.t. $0 \le cg(n) \le f(n)\ \forall n \ge n_0\}$

$O(g) = \{f \mid \exists$ constants $c, n_0 > 0$ s.t. $f(n) \le cg(n)\ \forall n \ge n_0\}$

o, ω Notation

$$f \in o(g) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

$$f \in \omega(g) \Leftrightarrow g \in o(f) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

$$f \in \Theta(g) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = c$$

- $f \in o(g) \Rightarrow f \in O(g) \setminus \Theta(g)$.
- $f \in \omega(g) \Rightarrow f \in O(g) \setminus \Theta(g)$.
- $f \in \Theta(g) \Leftrightarrow g \in \Theta(f)$

Remember This!

The Upshot!

- $f \in O(g)$ is like "$f \leq g$,"
- $f \in \Omega(g)$ is like "$f \geq g$,"
- $f \in o(g)$ is like "$f < g$,"
- $f \in \omega(g)$ is like "$f > g$," and
- $f \in \Theta(g)$ is like "$f = g$."

Functions

- Polynomials f of degree k are in $\Theta(n^k)$.
- Exponential functions always grow faster than polynomials
- Polylogarithmic functions always grow more slowly than polynomials.

Count 'em Up

- You should be able to look at a short code module, and write down how many times each line is done.
- Like the InsertionSort, MergeSort, and Towers of Hanoi examples in class.
- If the algorithm is recursive, you should be able to look at the recurrence and compute its running time

More Algorithm Stuff

- What is the difference between in-place and out-of-place?
- How do I prove correctness of an algorithm? **Loop invariant**
  - **Base Case:** It is true prior to the first iteration of the loop
  - **Maintenance:** If it is true before a loop iteration, it is true after the loop iteration
  - **Termination:** Hopefully, the invariant will have a useful property when the loop terminates. In this case, it would "prove" that the array is sorted.

Analyzing Recurrences

Deep Thoughts

To understand recursion, we must first understand recursion

- General methods for analyzing recurrences
  - Substitution
  - Master Theorem
- When we analyze a recurrence, we may not get or need an exact answer, only an asymptotic one

The Master Theorem

- Most recurrences that we will be interested in are of the form

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ aT(n/b) + f(n) & n > 1 \end{cases}$$

- The Master Theorem tells us how to analyze recurrences of this form.
- If $f \in O(n^{\log_b a - \epsilon})$, for some constant $\epsilon > 0$, then $T \in \Theta(n^{\log_b a})$.
- If $f \in \Theta(n^{\log_b a})$, then $T \in \Theta(n^{\log_b a} \lg n)$.
- If $f \in \Omega(n^{\log_b a + \epsilon})$, for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and $n > n_0$, then $T \in \Theta(f)$.

Sorting Algorithms

Simple Sorting Algorithms:

- Merge Sort:
  - Divide the list into smaller pieces. Sort the small pieces. Then merge together sorted lists.
- Insertion Sort:
  - Insert item j into $A[0 \ldots j-1]$
- Selection Sort
  - Find j^{th} smallest element and put it in $A[j]$
- Bubble sort:
  - Make n passes through the list. If adjacent elements are out of position, swap them.

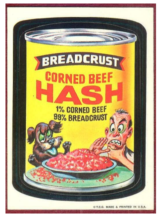
Java Collections

You need to know a little about the Java Collections

What is a `Set`, `List`, `Map`, `SortedSet`. What are the different implementations of each?

- A Set is a Collection that cannot contain duplicate elements.
- Set implementations: HashSet, Treeset, LinkedHashSet
- A List can contain suplicate elements
- A Map is a set of (unique) keys, each key being paired with a value.

More on Hash



- In a hash table the number of keys stored is small relative to the number of possible keys
- A hash table is an array. Given a key k, we don't use k as the index into the array – rather, we have a hash function h, and we use $h(k)$ as an index into the array.
- Given a "universe" of keys K.
  - Think of K as all the words in a dictionary, for example
- $h : K \rightarrow \{0, 1, \ldots m-1\}$, so that $h(k)$ gets mapped to an integer between 0 and $m-1$ for every $k \in K$
- We say that k hashes to $h(k)$

More on Data Structures

- A `LinkedHashSet` is a `HashSet` that also keeps track of the order in which elements were inserted.
- (Think of laying a linked list on top of the Hash Table)
- A `TreeSet` stores its elements in a alertred-black tree.
- A red-black trees, is a balanced binary search tree
- Hash table is "good" at INSERT(), SEARCH(), DELETE(). But what if you also want to support (efficiently) MINIMUM(), MAXIMUM()

Storing Binary Trees

Array

- The root is stored in position 0.
- The children of the node in position i are stored in positions $2i+1$ and $2i+2$.
- This determines a unique storage location for every node in the tree and makes it easy to find a node's parent and children.
- Using an array, the basic operations can be performed very efficiently.

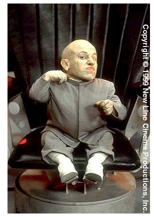
Binary Search Tree

- A binary search tree is a data structrue that is conceptualized as a binary tree, but has one additional property:

Binary Search Tree Property

If y is in the left subtree of x, then $k(y) \leq k(x)$

Short Is Beautiful



- SEARCH() takes $O(h)$
- MINIMUM(), MAXIMUM() also take $O(h)$
- Slightly less obvious is that INSERT(), DELETE() also take $O(h)$
- Thus we would like to keep out binary search trees "short" (h is small).

Sorted

- We saw in the lab that the Java Tree Set allowed you to iterate through the list in sorted order. How long does it take to do this?

```
INORDER-TREE-WALK(x)
if x ≠ NIL
   then INORDER-TREE-WALK(ℓ(x))
        print k(x)
        INORDER-TREE-WALK(r(x))
```

- What is running time of this algorithm?

Operations

SUCCESSOR(x)

- How would I know "next biggest" element?
- If right subtree is not empty: MINIMUM($r(x)$)
- If right subtree is empty: Walk up tree until you make the first "right" move

INSERT(x)

- Just walk down the tree and put it in. It will go "at the bottom"

DELETE()

- If 0 or 1 child, deletion is fairly easy
- If 2 children, deletion is made easier by the following fact:

Binary Search Tree Property

- If a node has 2 children, then
  - its successor will not have a left child
  - its predecessor will not have a right child

Heaps

- Heaps are a bit like binary search trees, however, they enforce a different property

Heap Property: Children are Horrible!

- In a max-heap, the key of the parent node is always at least as big as its children:

$$k(p(x)) \geq k(x) \quad \forall x \neq root$$

Heapify

HEAPIFY(x)

1. Find largest of $k(x)$, $k(\ell(x))$, $k(r(x))$
2. If $k(x)$ is largest, you are done
3. Swap x with largest node, and call HEAPIFY() on the new subtree

- $\Rightarrow$ HEAPIFY a node in $O(\lg n)$
- Alternatively, HEAPIFY node of height h is $O(h)$
- Building a heap out of an array of size n takes $O(n)$

Operations on a Heap

- The node with the highest key is always the root.
- To delete a record
  - Exchange its record with that of a leaf.
  - Delete the leaf.
  - Call `heapify()`.
- To add a record
  - Create a new leaf.
  - Exchange the new record with that of the parent node if it has a higher key.
  - This is like insertion sort – just move it up the path...
  - Continue to do this until all nodes have the heap property.
  - Note that we can change the key of a node in a similar fashion.

Time for Heap Operations

CREATE	$O(n)$
MAXIMUM	$\Theta(1)$
HEAPIFY	$O(\lg n)$, or $O(h)$
EXTRACT-MAX	$O(\lg n)$
HEAP-INCREASE-KEY	$O(\lg n)$
INSERT	$O(\lg n)$

Heap Sort

- Suppose the list of items to be sorted are in an array of size n
- The heap sort algorithm is as follows.
  1. Put the array in heap order as described above.
  2. In the i^{th} iteration, exchange the item in position 0 with the item in position $n-i$ and call `heapify()`.
- What is the running time? $\Theta(n \lg n)$

Misery Loves Company

I'm in a baaaaaaaaaaaaaaaaaad mood.

- Quiz on Wednesday
- I will be out of town Tuesday and Wednesday: I am going to drive to Indianapolis and punch Peyton Manning in the nose
- It is closed book, closed notes.
- I will give you a piece of paper with some useful formulae