Lecture 14: 5/19/2009

Announcements: Ps8 out. Ps7, solutions out tomorrow

From last time: Binary Search (assumes the input array A is sorted):

Binary-Search(X,A, low, high):Looks for X in A[low..high]
Start
1.**If**(low>high) **Return**("not found"); // (range of A is empty)
2. Middle ←⌊(low+high)/2⌋;
3. If(A[Middle] = X) **Return** (Middle)
4. If (A[Middle] < X) low ← Middle+1 // X not in A[1..Middle]
5. If (A[Middle] >X) high ← Middle-1 // X not in A[Middle..high]
6. Binary-Search(X,A,low,high);
End;

To start call Binary-Search(X,A,1,n).

Analysis of Run time:

We can assume that each iteration of the program (not including the recursive call in the last line), takes constant time (that is, it does not depend on n). Thus to get the big Theta run time, we only need to count the number of calls.

Let T(n) be the maximum number of calls for an array of size *n*, not including the final failure call (if not found).

T(1) = 1
T(n) = T(\lceil n/2\rceil) + 1

Usually ok to ignore ceilings, floors:

T(n) = T(n/2) + 1 // 2^0
= T(n/4) + 2 = T(n/2^2) + 2 //2^1
= T(n/8) + 3 = T(n/2^3) + 3 //
= T(n/2^k) + (k-1)
...
= T(n/n) + (lg n -1)
= 1 + (lg n -1)
= lg n

Correctness:
Binary Search is a subtle program and easy to get wrong (many experienced programmers have implemented it with bugs). To prove the above code is correct we use computational induction. It is easy to see that if the program returns "Middle" in line 3 then X is in the array A in position Middle. Trickier is to be sure that if "not found" is returned in line 1, then X is not in the array. To show this we argue that at all times we have the following property **P**:
either X is somewhere in A[low..high], or X is not in A.
Clearly this is true for our initial call where low=1 and high =n (assuming A[1..n] is the entire array). For each subsequent call, we either update low in line 4, where X > A[Middle], so we claim X is not in A[1..Middle], so when we set low to Middle+1 we maintain **P**. Similarly for the update in line 5 which also maintains **P** by lowering high when we know X is not in A[Middle..n].
This is in essence an induction proof on the number of calls to Binary-Search: if true at the start of call i, then true for the $i+1^{st}$ call (and for the basis, true initially).

EXAMPLE 2: Towers of Hanoi.

We did this.

$$T_n = 2\,T_{n-1} + 1 \quad \text{for } n \geq 1$$
$$0 \qquad \text{for } n=0$$

Now show that you prove the answer by induction on n (as in the homework)
Claim: The solution to the recurrence above is

$T_n = 2^n - 1$
$T_0 = 0$ OK; ($T_1 = 1$ OK)[($T_2 = 3$ OK)

Suppose true for n = k.
$T_k = 2^k - 1$.
Want to show true for n = k+1:

T_{k+1} = 2T_k - 1
= 2(2^k-1) + 1
=2^{k+1} - 2 + 1
=2^{k+1} + 1 QED

EXAMPLE 3: Recursion Trees: $T(n) = 3\ T(n/2) + n^2$

$T(n) = 1 \qquad n \leq 1$

Tree with root n and 3 under each node of half the size. Work at level i sums all nodes sizes squared so:

Level 0: n^2

Level 1: 3 *(n/2)^2 = n^2 (3/2^2)

Level 2: 3^2 * (n/4)^2 = n^2 * (3/2^2)^2

Level i: n^2 * (3/2^2)^i

The total is $n^2 (1 + ¾ + (3/4)^2 + (3/4)^3 \ldots) = \Theta(n^2)$ (Since ¾ < 1 it goes down as)

Master Theorem

If $T(n)= aT(n/b) + \Theta (n^d)$ for a,b>0, d≥0, then

$T(n) = \Theta(n^{\log_b a})$ if $b^d < a$

$T(n) = \Theta (n^d)$ if $b^d > a$

$T(n) = \Theta (n^d \log n)$ if $b^d = a$

Idea: as in the example above, we get a tree with $\log_b n$ levels, and the total work at each level i is $\Theta (n^d) * r^i$ where $r = a/ b^d$

Thus, if $b^d < a$, $r > 1$ and the work is increasing as the level increases. The work is dominated by the number of leaves (final levels work), which is $\Theta(n^{\log_b a})$.

If $b^d < a$, $r < 1$ and the work is decreasing as the level increases. The work is dominated by the top, $\Theta (n^d)$

If $b^d = a$, $r = 1$ and the work is the same at each level. The number of levels is $\log_b n$, so the total is $\Theta (n^d \log_b n) = \Theta (n^d \log n)$ (since b is a constant).

eg:

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

Idea: consider a tree of recursive calls. Root (and each node except at the bottom), has a further calls. Number of calls at level i is a^i and problem size at level i is (n/ b^i). Thus the value associated with each problem of that size is $\Theta(m^d)$ where $m=(n/ b^i)$., thus the value is $\Theta((n/ b^i).^d) = \Theta(n^d/ b^{id.})$
Since there are a^i at level i , the total for level i is: $a^i \Theta(n^d/ b^{id.}) = (a/ b^d)^i \Theta(n^d)$

Thus if $b^d < a$, $(a/ b^d)^i$ increases as i grows, so dominated by bottom terms
If $b^d > a$, then $(a/ b^d)^i$ decreases as i grows, so dominated by top term
if $b^d = a$, then $(a/ b^d)^i$ is always 1, so $\Theta(n^d)$ at each level, and log n levels.

Counting (Chapter 5)
---

n! = number of ways to order n different items
2^n = number of subsets of n items

P(n,r) = n!/(n-r)! = n*(n-1)*...*(n-r+1)
= the number of ways to fill r bins, one item per bin, using items drawn from 1,...,n. No replacement; an item, once used, is gone.

C(n,r) = n! / r!(n-r)! = P(n,r)/r!
= number of r-element subsets from a set of n different items. No replacement; an item, once used, is gone. Also *n choose r* the number of ways to choose r things from a set of size n.

product rule = if event A can occur in *a* ways and, independent of this, event B can occur in *b* ways then the number of combinations of ways for A and B to occur is *ab*.
(Really just a statement that $|A \times B| = |A|\,|B|$ for finite A, B.)