

## 6.046 Recitation 1 Handout

February 8, 2008

### Describing Algorithms for Homework

When stating what your algorithm is, you should always explain it in English to show your understanding of how it works. Pseudocode can be added and referred to to make it easier to describe. In the following example, we are trying to find a number  $x$  in a sorted array  $A$  between the positions  $a$  and  $b$  in the array. English explanation for binary search:

To find the number  $x$  between the positions of  $a$  and  $b$  in the sorted array  $A$ , we find the middle element  $mid$  located halfway between  $A[a]$  and  $A[b]$  and compare it to  $x$ . If they are equal, we have found  $x$ . If not, we recurse and look in the appropriate half of the array we're looking at. If  $x > mid$ , then we look in the array from indices  $mid + 1$  to  $b$ . Otherwise, we look in the array from indices  $a$  to  $mid - 1$ .

Example pseudocode to accompany English explanation:

```
BinarySearch(A, a, b, x)
1  if a > b then
2    return false
3  else
4    mid <- floor((a+b)/2)
5    if x = A[mid] then
6      return true
7    if x < A[mid] then
8      return BinarySearch(A, a, mid - 1, x)
9    else
10   return BinarySearch(A, mid + 1, b, x)
```

Note: Syntax isn't important, but if you make a crucial error in your pseudocode that isn't corrected or addressed in your English explanation, you may get docked points.

### Algorithm Correctness

Given the problem, you can write down the:

- Precondition: what may be assumed at the beginning
- Postcondition: what should be true at the end

Thus you want to show Precondition + Algorithm  $\rightarrow$  Postcondition.

Here we prove the correctness of fairly simple algorithms to the point of overkill. We do not expect this level of detail of simple algorithms such as these. The point however is that you should be able to use the same ideas when you need to analyze more complex algorithms that are not nearly so obviously correct. Another note about homework is that you can always reference existing algorithms detailed in CLRS or lecture. No point in reinventing the wheel that we've already given you.

### Recursive Algorithms

Binary search example

Precondition: all inputs valid (e.g.  $a$  and  $b$  in array, array has numbers in it, etc.), array is sorted

Postcondition: found  $x$  if it was in the array,  $A$  unchanged

Proof technique: Use induction to show that algorithm works on smallest base case (size 0 array, i.e. when  $b < a$ ) then for inductive case (for  $b \geq a$ )

### Iterative Algorithms

Summing an array  $A$  from indices  $a$  to  $b$

```
1 i <- a, sum <- 0
2 while i != b+1 do
3   sum <- sum + A[i]
4   i <- i + 1
```

Precondition: valid input ( $a$ ,  $b$ , and  $A$  valid),  $a \leq b + 1$  (otherwise  $i = b + 1$  will never happen and the loop won't terminate)

Postcondition:  $sum = \sum_{j=a}^b A[j]$ ,  $A$  unchanged

Proof technique: Use induction to prove a *loop invariant*, or what should be true at the beginning of each iteration of the loop. Here it's  $sum = \sum_{j=a}^{i-1} A[j]$  for the loop in lines 2-4. Then prove the loop terminates and show that when it does, the postcondition has been achieved.

### Recurrences

For convenience, here's the Master Method. Note that the second case is not exactly the same as in CLRS but is a more general version. The additional  $\lg n$  factor comes from the height of the recursion tree when both  $n^{\log_b a}$  and  $f(n)$  sort of match. You can play with the recursion tree method to get an intuition as to why it happens.

Let  $a \geq 1$  and  $b > 1$  be constants, let  $f(n)$  be a function, and let  $T(n)$  be defined on the nonnegative integers by the recurrence  $T(n) = aT(n/b) + f(n)$  where  $n/b$  means either  $\lfloor n/b \rfloor$  or  $\lceil n/b \rceil$ . Then

1. If  $f(n) = O(n^{\log_b a - \epsilon})$  for some constant  $\epsilon > 0$ , then  $T(n) = \Theta(n^{\log_b a})$ .
2. If  $f(n) = \Theta(n^{\log_b a} \lg^k n)$  for some constant  $k \geq 0$ , then  $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$ .
3. If  $f(n) = \Omega(n^{\log_b a + \epsilon})$  for some constant  $\epsilon > 0$  and  $af(n/b) \leq cf(n)$  for some constant  $c < 1$  and all sufficiently large  $n$ , then  $T(n) = \Theta(f(n))$ .

### Practice

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

$$\begin{aligned} f(n) &= n^3 \\ &= n^{\log_2 4 + 1} \\ &= \Omega(n^{\log_2 4 + \epsilon}) \text{ where } \epsilon = 1 > 0 \end{aligned}$$

$\Rightarrow$  candidate for case 3 of the Master Method.

Check the second necessary constraint for case 3:

$4(n/2)^3 \leq cn^3$  is true for constants  $c \in [1/2, 1)$ , so case 3 it is. Thus  $T(n) = \Theta(n^3)$

2.  $T(n) = 4T(n/2) + n^2 \lg n$

$$\begin{aligned} f(n) &= n^2 \lg n \\ &= n^{\log_2 4} \lg n \\ &= \Theta(n^{\log_b a} \lg^k n) \text{ where } k = 1 \geq 0 \end{aligned}$$

$\Rightarrow$  case 2 of the Master Method. Thus  $T(n) = \Theta(n^2 \lg^2 n)$ .

3.  $T(n) = 5T(n/2) + n^2 \lg n$

$$\begin{aligned} f(n) &= n^2 \lg n \\ &= n^{\log_2 5 - (\log_2 5 - 2)} \lg n \\ &= O(n^{\log_b a - \epsilon}) \text{ where any } \epsilon \in (0, \log_2 5 - 2) \text{ works} \end{aligned}$$

(Note that  $\lg n = o(n^\epsilon)$  for any  $\epsilon > 0$ .)

$\Rightarrow$  case 1 of the Master Method. Thus  $T(n) = \Theta(n^{\log_2 5})$ .

4.  $T(n) = 8T((n - \sqrt{n})/4) + n^2$

We can't apply the Master Method directly. But we can ignore the lower order terms and note that  $T(n) \leq 8T(n/4) + n^2$ . For less clear cases, you can use the substitution method afterwards to check to make sure your final answer works. But let's apply the Master Method:

$$\begin{aligned} f(n) &= n^2 \\ &= n^{\log_4 8 + (2 - \log_4 8)} \\ &= \Omega(n^{\log_b a + \epsilon}) \text{ where any } \epsilon \in (0, 2 - \log_4 8] \text{ works} \end{aligned}$$

$\Rightarrow$  candidate for case 3 of the Master Method.

Check the second necessary constraint for case 3:

$8(n/4)^2 \leq cn^2$  is true for constants  $c \in [1/2, 1)$ , so case 3 it is. Thus  $T(n) = \Theta(n^2)$

5.  $T(n) = 2T(\sqrt{n}) + \lg \lg n$

We can't apply the Master Method directly, but we can do a change of variables. Let's pick  $m = \lg n$ . We thus have

$$T(2^m) = 2T(2^{m/2}) + \lg m$$

Let's define a new recurrence  $S(m) = T(2^m)$ . We thus have

$$S(m) = 2S(m/2) + \lg m$$

Notice that the last term has not changed because the last substitution only affects how we represent  $T(n)$ ; no variables are actually changing. Now we can apply the Master Method to  $S(m)$ .

$$\begin{aligned} f(m) &= \lg m \\ &= m^{\log_2 2 - \log_2 2} \lg m \\ &= O(m^{\log_b a - \epsilon}) \text{ for any } \epsilon \in (0, \log_2 2) \end{aligned}$$

$\Rightarrow$  case 1 of the Master Method. Thus  $S(m) = \Theta(m)$  and  $T(2^m) = \Theta(m)$ . Substituting back  $n$ , we get  $T(n) = \Theta(\lg n)$ .

## Sloppiness Theorem

Merge sort has the recurrence  $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n)$ . When else can we ignore floors and ceilings? In the following,  $g(n)$  is the solution to the recurrence  $T(n) = aT(n/b) + f(n)$  when you ignore floors and ceilings (i.e. you've assumed you have input sizes of perfect powers of  $b$ ). Here is the Sloppiness Theorem:

If the following are true:

1.  $T(n)$ ,  $g(n)$  both monotonically increasing
2.  $T(b^i) \leq g(b^i)$  for all integers  $i > 0$
3.  $g(n) = O(g(n/b))$  (i.e.  $g(n)$  grows polynomially, not exponentially or such)

Then  $T(n) = O(g(n))$ .

Proof:

We want to show  $T(n) = O(g(n))$ , i.e.  $0 \leq T(n) \leq cg(n)$  for some constant  $c$  and large enough  $n$ . We have the following:

$$\begin{aligned} T(n) &= T(b^{\lceil \log_b n \rceil}) \\ &\leq T(b^{\lceil \log_b n \rceil}) \quad \text{by (1)} \\ &\leq g(b^{\lceil \log_b n \rceil}) \quad \text{by (2)} \\ &\leq cg(b^{\lceil \log_b n/b \rceil}) \quad \text{by (3)} \\ &= cg(b^{\lceil \log_b n \rceil - 1}) \\ &\leq cg(b^{\lceil \log_b n \rceil}) \quad \text{by (1)} \\ &= cg(n) \quad \checkmark \end{aligned}$$

Merge sort example

We have  $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n)$  and  $g(n) = O(n \lg n)$ .

1.  $T(n)$ ,  $g(n)$  both monotonically increase  $\checkmark$
2.  $T(b^i) \leq O(b^i \lg b^i) \checkmark$
3.  $n \lg n = O(\frac{n}{b} \lg \frac{n}{b}) \checkmark$

Thus  $T(n) = O(n \lg n)$ .