

# CS 361 Data Structures and Algorithms I

## Test #1: Solutions

March 3, 2000

**Problem 1.** Ten functions are given below; for each, give a  $\Theta(\ )$  expression that is as simplified as possible, then rank the 10 functions in increasing order of growth rate. Justify your answers briefly.

$$\begin{array}{ll} n^2 \cdot 3^{\log_5 n} + n^2 - 6n\sqrt{n} & (\log n)^{\log n} \\ n^2 / \log n + 2n\sqrt{n} & 2n^3 / (5^{\log_2 n}) + n\sqrt{n} \\ \sqrt{n}(\log n)^4 + (\log n)^5 & 2^{0.00001n} \\ 6n^2(\log \log 2n) / \log n & 2^n \\ 2\sqrt{\log_2 n} & (\sqrt{n})^3 \log n + 10(n / \log n)^2 \end{array}$$

Let us simplify what we can (mostly constants raised to log functions and multiplicative constants on the “ground floor”); then, when we have a sum of terms, we need to identify the dominant one.

- $n^2 \cdot 3^{\log_5 n} + n^2 - 6n\sqrt{n} = n^2 \cdot n^{\log_5 3} + n^2 - 6n\sqrt{n} = n^{2+\log_5 3} + n^2 - 6n^{1.5} = \Theta(n^{2+\log_5 3})$ , because that is the largest of the three powers of  $n$ .
- $n^2 / \log n + 2n\sqrt{n} = \Theta(n^2 / \log n)$ , because the ratio of the first term to the second is  $\sqrt{n} / (2 \log n)$ , which grows arbitrarily large.
- $\sqrt{n}(\log n)^4 + (\log n)^5 = \Theta(\sqrt{n}(\log n)^4)$ , because the square root term dominates all the logs—the ratio of the first term to the second is  $\sqrt{n} / \log n$ , which grows arbitrarily large.
- $6n^2(\log \log 2n) / \log n = \Theta(n^2(\log \log n) / \log n)$
- $2\sqrt{\log_2 n} = \Theta(2\sqrt{\log_2 n})$
- $(\log n)^{\log n} = \Theta((\log n)^{\log n})$
- $2n^3 / (5^{\log_2 n}) + n\sqrt{n} = 2n^3 / n^{\log_2 5} + n\sqrt{n} = 2n^{3-\log_2 5} + n^{1.5} = \Theta(n^{1.5})$ , since  $\log_2 5 > 2$ .
- $2^{0.00001n} = \Theta(2^{0.00001n})$
- $2^n = \Theta(2^n)$
- $(\sqrt{n})^3 \log n + 10(n / \log n)^2 = n^{1.5} \log n + 10(n / \log n)^2 = \Theta((n / \log n)^2)$ , because the polynomial terms dominate the log terms—the ratio of the first term to the second is  $(\log n)^3 / n^{0.5}$ , which grows arbitrarily small.

Now we can order them

- 1:  $\Theta(2\sqrt{\log_2 n})$ ; grows slower than  $n^{0.5}$ , as can be seen by taking the log of each:  $\log(n^{0.5}) = 0.5 \log n$ , whereas  $\log 2\sqrt{\log_2 n} = \sqrt{\log n}$ .
- 2:  $\Theta(\sqrt{n}(\log n)^4)$ ; grows pretty much like  $n^{0.5}$ .
- 3:  $\Theta(n^{1.5})$ ; grows slower than  $n^2$ .
- 4:  $\Theta((n / \log n)^2)$ ; this one and the next two are all dominated by  $n^2$ ; this one is also divided by  $(\log n)^2$ , whereas the next is only divided by  $\log n$ .
- 5:  $\Theta(n^2 / \log n)$ ; see above; the next one is the same as this, but multiplied by  $\log \log n$ .
- 6:  $\Theta(n^2(\log \log n) / \log n)$ ; see above; still grows somewhat less fast than  $n^2$ .
- 7:  $\Theta(n^{2+\log_5 3})$ , the only polynomial here growing faster than  $n^2$ .
- 8:  $\Theta((\log n)^{\log n})$  grows faster than any polynomial—we can always find an  $n > 2^k$  such that it is larger than  $k^{\log n} = n^{\log_2 k}$  and thus make the power of  $n$ ,  $\log_2 k$ , arbitrarily large. Another way to see this is to substitute  $k$  for  $n$  in the above, obtaining  $\Theta(k^{\log \log k})$ —which clearly shows the ever-increasing degree.
- 9:  $\Theta(2^{0.00001n})$  is second fastest (clearly slower than  $2^n$ ).

- 10:  $\Theta(2^n)$  is fastest.

**Problem 2.** Solve the following recurrences in  $\Theta(\ )$  terms; show your steps.

- $f(n) = 2f(n/2) - f(n/4) + \Theta(n^2)$
- $f(n) = 3f(n/2) + \Theta(n^2)$
- $f(n) = 5f(n-1) - 6f(n-2) + \Theta(2^n)$
- $f(n) = 2f(n/2) - f(n/4) + \Theta(n^2)$ : first note that this recurrence is only defined for  $n$  a power of 2 (not a power of 4, since, if  $n$  is a power of 4,  $n/2$  will not be). Thus we write  $n = 2^k$ , substitute, and let  $g(k) = f(2^k)$  to obtain  $g(k) = 2g(k-1) - g(k-2) + \Theta(4^k)$ , where the driving term has been simplified using  $(2^k)^2 = 2^{2k} = (2^2)^k = 4^k$  in order to show the correct radix. The homogenous equation is  $r^2 - 2r + 1 = 0$ , with double root of 1; thus the homogeneous solution is of the form  $ak + b$ , or  $\Theta(k)$ . The driving term is a polynomial of degree 0 (a constant) times the exponential  $4^k$ ; since 4 is not a characteristic root, the inhomogeneous solution is of the same form as the driving term and is thus  $\Theta(4^k)$ . The inhomogeneous solution thus dominates the homogeneous one, so that we can write  $g(k) = \Theta(4^k)$ , or  $f(n) = \Theta(n^2)$ .
- $f(n) = 3f(n/2) + \Theta(n^2)$ : first note that this recurrence is only defined for  $n$  a power of 2. Thus we write  $n = 2^k$ , substitute, and let  $g(k) = f(2^k)$  to obtain  $g(k) = 3g(k-1) + \Theta(4^k)$ . The characteristic equation is  $r - 3 = 0$ , with single root of 3, so that the homogeneous solution is of the form  $a3^k$  or  $\Theta(3^k)$ . The driving term is a polynomial of degree 0 (a constant) times the exponential  $4^k$ ; since 4 is not a characteristic root, the inhomogeneous solution is of the same form as the driving term and is thus  $\Theta(4^k)$ . The inhomogeneous solution thus dominates the homogeneous one, so that we can write  $g(k) = \Theta(4^k)$ , or  $f(n) = \Theta(n^2)$ .
- $f(n) = 5f(n-1) - 6f(n-2) + \Theta(2^n)$ : the characteristic equation is  $r^2 - 5r + 6 = 0$ , with single roots 2 and 3. Thus the homogeneous solution is of the form  $a2^n + b3^n$  or  $\Theta(3^n)$ . The driving term is a polynomial of degree 0 (a constant) times the exponential  $2^n$ ; since 2 is a characteristic root, the inhomogeneous solution is of the same form as the driving term, but with the polynomial degree increased by 1, and is thus of the form  $(cn+d)2^n$  or  $\Theta(n2^n)$ . The homogeneous solution thus dominates the inhomogeneous one, so that we can write  $f(n) = \Theta(3^n)$ .

**Problem 3.** Consider the following program (in some type of pseudocode). Analyze its running time (when called with `left=1` and `right=n`, as shown in the fragment) as a function of the variable  $n$ ; show your work in setting up the analysis and in solving any resulting equations or recurrences.

```

subroutine whatever(left, right, array)
  if left = right then return
  mid <- (left + right) / 2
  call whatever(left, mid, array)
  call whatever(mid, right, array)
  i <- 1
  while (i*i) <= (right - left) do
    for j=left to i do array[j]++
    i++

whatever(1,n,array) /* the array has n elements */

```

The overall recurrence is clearly  $f(n) = 2f(n/2) + g(n)$ , where  $g(n)$  is the time taken by the `while` loop. That loop runs  $\sqrt{m}$  times, where  $m$  is the size of the subarray—initially just  $n$ . Inside the loop, the `for` loop only rarely runs: most of the time `left` is larger than `i`, which can only grow to a maximum of  $\sqrt{n}$ .

We can bound  $g(n)$  as follows. For an easy upper bound, assume that we always have `left = 1`; then the `for` loop runs  $i$  times, so we get a total running time for the `while` loop proportional to  $\sum_{i=1}^{\sqrt{n}} i$ , which

is basically  $n/2$  (because the sum of  $i$ s from 1 to  $k$  is  $k(k+1)/2$ ) and thus  $O(n)$ . Using  $g(n) = O(n)$  in the overall recurrence, we get  $f(n) = O(n \log n)$ .

A trivial lower bound is to assume that the `for` loop never runs, with the result that the `while` loop takes order  $\sqrt{n}$  time, yielding  $g(n) = \Theta(\sqrt{n})$ ; solving the recurrence with this value of  $g(n)$  takes some care. As usual write  $n = 2^k$  and thus obtain  $h(k) = 2h(k-1) + \Omega(\sqrt{2^k})$ ; we need to rewrite the driving term as a simple exponential; we have  $\sqrt{2^k} = (2^k)^{\frac{1}{2}} = 2^{\frac{k}{2}} = (2^{\frac{1}{2}})^k = \sqrt{2}^k$ . Since  $\sqrt{2}$  is not a characteristic root (there is only one characteristic root, 2), the overall solution is dominated by the homogeneous solution and thus we have  $h(k) = \Omega(2^k)$  and hence  $f(n) = \Omega(n)$ .

Which of the two bounds is loose? Intuitively, the upper bound—it assumed that the `for` loop always ran, whereas it clearly almost never runs. We can verify this simply: suppose we are  $k$  levels deep in the recursion; then the array size at this level is  $n/2^k$  and the `while` loop runs  $\sqrt{n/2^k}$  times—that is, the largest value of `i` is  $\sqrt{n/2^k}$ . But the value of `left` at that level is  $1 + an/2^k$ , where  $a = 0, 1, \dots, n-1$ . When is `left` small enough to allow the `for` loop to run? We need  $1 + an/2^k < \sqrt{n/2^k}$ , which can only occur for  $a = 0$ —in other words, only the leftmost recursive calls, all with `left` = 1, ever have a running `for` loop! The contribution of the `while` loops on these calls (each of these loops take time linear in the size of the array it processes, as we have seen) is  $n + \frac{n}{2} + \frac{n}{4} + \dots + \frac{n/2^k}{2} \dots + 1 < 2n$ . This gets added to the running time when no `for` loop ever runs, which is the lower bound we derived above. But adding order  $2n$  to our linear time leaves a linear running time. Thus we can write  $f(n) = \Theta(n)$ .

**Problem 4.** Consider the following program (in some type of pseudocode).

```
for i=1 to n do
  for j=i to n do
    print(j)
```

where `print(j)` writes the integer  $j$  (in base 10) on a line by itself, with no sign, no leading or trailing spaces, and no leading zeros. How many lines are printed? How many digits? Show your work.

The number of lines printed is the number of times the body of the inner loop is executed. We can easily verify that the number  $i$  gets printed  $i$  times, so that the total number of values printed is  $\sum_{i=1}^n i = n(n+1)/2$ .

The number of digits printed is simply the previous sum, where the inside term ( $i$ ) needs to be multiplied by the number of digits it takes to print  $i$  in base 10, which is  $\lfloor \log_{10} i \rfloor + 1$  (for  $i \geq 1$ , naturally). Thus the number of digits printed is

$$\sum_{i=1}^n i(\lfloor \log_{10} i \rfloor + 1) = \frac{n(n+1)}{2} + \sum_{i=1}^n i \lfloor \log_{10} i \rfloor$$

The sum cannot be reduced to nice closed form, but can certainly derive a  $\Theta$  for it: we bound it from above by letting each term of the sum be  $i \log_{10} n$ , getting an upper bound of  $O(\frac{1}{2}n(n+1) \log_{10} n) = O(n^2 \log n)$  and bound it from below by using only the second half of the sum and replacing each term by  $i \log_{10}(n/2)$ , obtaining a matching lower bound of  $\Omega(n^2 \log n)$ , so that the number of digits printed is  $\Theta(n^2 \log n)$ . We can derive much more precise upper and lower bounds using integration: replace the discrete sum by a continuous integral and then replace  $\lfloor \log_{10} i \rfloor$  by  $(\log_{10} i) - 1$  for a lower bound and by  $\log_{10} i$  for an upper bound. This type of trick (replace a discrete sum by a continuous integral) is quite useful to derive very precise bounds, but not particularly useful in our context, where we are usually satisfied with  $\Theta$  estimates.