Basic Algorithms (V22.0310); Fall 2005; Yap

HOMEWORK 5

Date Due: Wed Nov 23

Instructions

- Please write clearly. If you have poor handwriting, please consider printing your solutions. WE CANNOT GRADE WHAT WE CAN'T READ.
- Please start early, especially since there is programming involved.

Question 1

(20 Points)

The textbook (chap. 5, p.229) gives a bound of 15 for the merge part of divide-and-conquer algorithm for closest pair problem. Let us give an improved bound.

Let S be the subset of input points within δ from the line L (p.229 of Text). Here δ is the minimum distance between any two input points on either side of L. Suppose $s_L, s_R \in S$ achieves the minimum distance in S, i.e., $d(s_L, s_R) = \min\{d(s, s') : s, s' \in S, s \neq s'\}$. Assume that $d(s_L, s_R) < \delta$. Let B be the rectangular box defined by the horizontal lines through s_L and s_R, and by the vertical lines $x = x^* - \delta$ and $x = x^* + \delta$.

(a) Let $S' = S \cap B$. Show that $|S'| \leq 24/\pi$. HINT: Let B' be the expansion of the box B by $\delta/2$ all around. How many pairwise-disjoint balls of radius $\delta/2$ can fit inside the expanded box B'?

(b) Conclude that the number 15 can be improved to 7.

♠

Question 2

(Recurrences, 40 Points)

(a) Use Rote method to solve $T(n) = 8T(n/2) + n^2$.

(b) Use Induction method to prove that your bound in the Rote method is truly an upper bound.

(c) Recall the Master Theorem discussed in class: Let $T(n) = aT(n/b) + n^k$ where $a > 0$, $b > 1$ and k are constants. Let $w = \log_b a$. Then the theorem says:

$$T(n) = \begin{cases} \Theta(n^k) & \text{if } k > w, \\ \Theta(n^w) & \text{if } k < w, \\ \Theta(n^k \lg n) & \text{if } k = w. \end{cases}$$

Suppose you have two strategies to solve a problem: you can either divide it into 3 subproblems each of size $n/2$ or divide it into 2 subproblems each of size $n/3$. The work to combine subproblems is n in both cases. What is a better solution?

(d) Same as part (c), except that the work to combine subproblems is now n^2. Whis is a better solution?

Question 3

(Karatsuba, 50 Points)

The sizes of the primitive datatypes are fixed by the Java standard, e.g., the number type `int` is 32 bits. Many applications need large numbers beyond the sizes of the primitive number types. E.g., in cryptography and computer security applications, we routinely perform integer arithmetic on integers with up to 1024 bits or more. Java provides the class `BigInteger` to support such applications. In addition to the standard arithmetic operations of add, subtract, multiply and integer division, it support methods to perform bitwise

operations such as AND, OR, XOR, NOT, left and right shifts. There are also methods to generate random BigIntegers. We give you various sample programs so that you can easily modify it to accomplish the tasks.

- (a) Please implement Karatsuba's multiplication algorithm using the `BigInteger` class. The only rule is that you MUST NOT use the multiplication, division, reciprocal or squaring functions in the biginteger class. What you can (and SHOULD) use are its addition, subtraction and shifting functions. Your main class should be called `Karat` (so your main file should be `Karat.java`).

- (b) The second part of this question is to write a program called `Timing.java`. Suppose a program has time complexity $T(n) = Cn^e$ for some constants $C > 0$ and e. Call C and e the **constant** and **exponent** of the program. For instance, the exponent of of Karatsuba's algorithm should be $\lg 3$ or about 1.58. But the constant depends on implementation details. We want you to estimate the exponent and constant of two programs:
  (A) your implementation of Karatsuba's algorithm, and
  (B) Java's multiplication algorithm in `BigInteger`.

  Your Java program called `Timing.java` should take four optional arguments and outputs as follows:

  ```
  % Timing [noPoints] [stepSize] [startSize] [noTrials]
  > (A) exponent = 1.59, constant = 2001
  > (B) exponent = 1.29, constant = 183
  ```

  where lines (A) and (B) illustrate the kind of outputs we expect. Here is an explanation of the arguments:

  ```
  [noPoints] = number of random pairs (X,Y) of BigIntegers
                       you want to generate.  Both X and Y
                       are supposed to have the same number of bits.
  [noTrials] = for each pair (X,Y), this is the number
                       of times you want to multiply them
                       using both algorithms (A) and (B)
  [stepSize] = the increase in the bit sizes of
                       successive pairs (X,Y)
  [startSize] = the bit size of the first pair (X,Y)
  ```

  All these arguments are optional, and you must supply default values when the user does not specify them. The reason you need `[noTrials]` is because if your computer is very fast and the bit size is not that large, your time (which is in milliseconds) may not be accurate! Hence you need to execute a multiplication many times and then take their average.

  Of course, in practice almost no program has time complexity that is exactly of the form $T(n) = Cn^e$. So in running your Timing program with different parameters, you may not get consistent values for e and/or C. But you should get a consistent exponent when n gets sufficiently large. You will first need to determine the running time taken by the programs (A) and (B) on various pairs (X, Y) of BigIntegers. For this purpose, you will use BigInteger's ability to generate random numbers, and Java' function to return the current time in milliseconds:

  ```
  long startTime = System.currentTimeMillis()
  BigInteger Z = mult(X,Y);  // the code you want to time
  long endTime = System.currentTimeMillis()
  System.out.println("Time taken = " + (endTime - startTime));
  ```

  We provide examples of these timing usage.

  There are two problems with this approach: the time we measure is "user time", not the time the system actually use to run your code. So if you are in a time-sharing environment, try to do your experiments when there are not too many users on the system. Another possible source of error is

that Java may do garbage collection between startTime and endTime. You just have to live with this possibility.

FINALLY, how do you estimate the exponent and constant from the running times you have collected? The idea is to take the logarithm of the time complexity function: hence we have

```
log(T(n)) = log(C) + e. log(n)
       y  = c + e.x
```

which is a linear equation with slope e and y-intercept given by c. Our problem becomes one of finding the best straight line to "fit" the data. For this purpose, we will provide you a function

```
double[] fitLine( double[] x, double[] y)
```

where x, y are vectors of the same length and `fitLine` returns a vector of lenth 2 containing the values c, e that represents a line with the least squares error. This function is found in the program `fitLine.java`.

WHAT TO HAND IN: As usual, please submit (a) your program files, (b) README file, and (c) a Makefile. Send these as a single TAR file to the grader, with cc's to me and the T.A. Your Makefile should have these targets:
(1) The default target ("make") is to compile all your programs.
(2) Typing "make a" will test your Karatsuba program of part (a).
(3) Typing "make b" will run your timing program to show us the exponent for your Karatsuba multiplication (you should pre-select the parameters so that this will be self-evident).
(4) Typing "make c" will show us the exponent for Java's multiplication.

Remember the rule that if your programs do not compile, you get zero points.