

CSE 101 Homework 2
Due Tuesday, May 13, 2 PM
Divide-and-Conquer
100 points total = 10 %

Is there a triangle? 20 pts In the calibration homework, we gave an algorithm to list all the triangles in an undirected graph in $O(n^2 + nm)$ time, which is $\Theta(n^3)$ when the graph is dense. Show how, given a graph G in matrix representation, it is possible to use Strassen's matrix multiplication algorithm (see MM, Chapter 2.5) to determine if the graph has a triangle in time $o(n^3)$. (You can use the algorithm without description or proof, but you must explain connection to the existence of a triangle in the graph).

First, I claim that there is a path of length 3 from x to itself in the graph if and only if the graph has a triangle containing x . Suppose x, y, z is a triangle; then $x \leftarrow y \leftarrow z \leftarrow x$ is such a path. Conversely, suppose $x \leftarrow a \leftarrow b \leftarrow x$ is such a path. Then x must be connected to a , a must be connected to b , and B to x , so they must all be different, and form a triangle.

Second, I claim that the (u, v) th co-ordinate in M_G^k is positive if and only if there is a path of length k from u to v in G . Note that no negative values occur in M_G , so there will only be non-negative values in M_G^k . Then $M^k[u, v] = \sum_w M^{k-1}[u, w] * M[w, v]$ is non-zero if and only if there is some w so that $M^{k-1}[u, w] > 0$ and $M[w, v] > 0$. By induction on k , this happens if and only if there is a path of length $k - 1$ from u to w and an edge from w to v , i.e., if and only if there is a path of length k from u to v whose penultimate step is w .

Together, there is a triangle in G if and only if there is a non-zero co-ordinate $M_G^3[x, x]$.

Therefore we can use the following algorithm:

Step 1: Multiply M by itself twice, using Strassen's matrix multiplication algorithm from Section 2.5. This takes $O(n^{\log 7}) = O(n^{2.8\dots})$ time. Step 2: For each i , test wheter the i, i 'th co-ordinate in the resulting product matrix is greater than 0. If so, halt and output, "Has a Triangle". This step is $O(n)$. Step 4: If not, output "No Triangles".

The run-time is dominated by Step 2, so the total time is $O(n^{\log 7})$.

Binary Conversion: 20 pts . Last homework, we found an $O(n^2)$ algorithm for converting a decimal integer to binary, where the basic operations involved single digits. Present and analyze a divide-and-conquer algorithm that does better. You will probably need to use the faster integer multiplication algorithm, *prod2*, discussed in pages 75-78 of NN.

The following recursive algorithm uses the divide and conquer method to convert an n bit binary integer $x_{n-1} \dots x_0$ into decimal. The reverse can be done the same way. It uses the $O(n^{\log_2 3})$ time divide-and-conquer multiplication algorithm *Multiply2* from class and the text; and the grade school linear time ($O(n)$) *Add* algorithm as sub-routines. We assume *Add* and *Multiply* are defined to take decimal integers as input and output. Note that 2^n , in binary, is a 1 followed by n 0's, so is easy to construct as a binary integer in linear time. Let *ConstructPower2*, given n , construct 2^n in binary in time $O(n)$.

ConvertToDecimal($x_{n-1} \dots x_0$: Binary integer represented as an array of bits): decimal integer;

1. IF $n = 1$ return x_0 .
2. $y \leftarrow x_{n-1} \dots x_{n/2}$
3. $z \leftarrow x_{n/2-1} \dots x_0$
4. $w \leftarrow \text{ConstructPower2}(n/2)$ (in binary)
5. $a \leftarrow \text{ConvertToDecimal}(y)$
6. $b \leftarrow \text{ConvertToDecimal}(z)$
7. $c \leftarrow \text{ConvertToDecimal}(w)$
8. $d \leftarrow \text{Multiply2}(a, c)$
9. $e \leftarrow \text{Add}(d, b)$
10. Return e

For the time analysis, we make three recursive calls, on w, y and z . Now, w, y, z are all $n/2$ bit binary integers. The time to construct them is $O(n)$. The results are decimal versions, and so have fewer digits (by about a log 10 factor). Thus a, b, c, d are at most $O(n)$ digits each, so the time for the two Adds is $O(n)$ and the Multiply is $O(n^{\log 3})$. SO the total time out of the recursion is $O(n^{\log 3})$.

This gives $T(n) = 3T(n/2) + O(n^{\log 3})$ as the recurrence. This meets the format of Theorem B.5 with $A = 3, B = 2, K = \log 3$. Then since $3 = 2^{\log 3}$, we are in the steady-state case, so $T(n) \in O(n^{\log 3} \log n)$.

One thing we could do better is observe that for $n = 2^k$, all the powers of 2 we use in the divide and conquer are actually for 2^{2^i} , $0 \leq i \leq \log n$. So we could pre-compute all of these using $\text{Exponent}[i] = \text{Multiply}[\text{Exponent}(i-1), \text{Exponent}(i-1)]$ and then replace the recursive call to get c by setting c to be the precomputed element. This will remove the $\log n$ factor from the order, which isn't much. However, if we improve the Multiply algorithm, we also get the improvement in this algorithm, unlike the one we did first. So the limit might be using Fast-Fourier Multiplication in the modified algorithm sketched above.

Median of two sorted lists: 20 pts. The *median* of a set of numbers is an element of that set so that half the elements (round down) are less than that number and half are at least as large as that number. Present and analyze a divide-and-conquer algorithm that, given two sorted arrays of distinct integers, $A[1..n], B[1..n]$, returns the median (n 'th largest) of the set of elements that appear in either list.

We want to find the median of the union of two sorted lists, $A[1..n]$ and $B[1..n]$, each of length n , i.e., the n 'th largest element. The algorithm goes as follows: We'll maintain three values, L_1, L_2, K and maintain the invariant that the element we are looking for is the median (K 'th largest) of the sets $A[L_1..L_1 + K - 1]$ and $B[L_2..L_2 + K - 1]$ (the important feature is that the two sub-arrays we are examining are the same size, K). We first compare the medians of the sub-arrays, $A[L_1 + \lceil K/2 \rceil - 1]$ and $B[L_2 + \lceil K/2 \rceil - 1]$. If $A[L_1 + \lceil K/2 \rceil - 1] \geq B[L_2 + \lceil K/2 \rceil - 1]$, we know $A[M] \geq B[M']$ for every $M \geq L_1 + \lceil K/2 \rceil - 1$ and every $M' \leq L_2 + \lceil K/2 \rceil - 1$. Thus, for each of the $\lfloor K/2 \rfloor$ values of M with $L_1 + K - 1 \geq M \geq L_1 + \lceil K/2 \rceil$, we know that it is larger than at least $\lceil K/2 \rceil$ elements in the B list, and at least $M - L_1 + 1 \geq \lceil K/2 \rceil + 1$ elements in the A list. Therefore, it is greater than $K + 1$ elements total, and is therefore greater than the median. Similarly, for all $\lfloor K/2 \rfloor$ values of M' in the range $L_2 \leq M' \leq L_2 + \lfloor K/2 \rfloor - 1$, we know that M' is smaller than $\lceil K/2 \rceil$ elements in the B list and $\lceil K/2 \rceil + 1$ elements in the A list. So each such M' is smaller than the median. Thus, if we delete $A[M]$ and $B[M']$ for M and M' in the range above, we have deleted $\lfloor K/2 \rfloor$ elements smaller than the median and the same number of elements larger than the median. Thus, the median will be the median of the remaining elements. To do this deletion, we reset $K \leftarrow \lceil K/2 \rceil$ and $L_2 \leftarrow L_2 + \lfloor K/2 \rfloor$.

The other case is handled symmetrically. When $K = 1$, we return the larger element between $A[L_1]$ and $B[L_2]$.

The algorithm takes constant time before and after making a recursive call to two arrays each of half the size. Therefore, the time is given by $T(n) = T(n/2) + O(1)$. Applying the Main recurrence theorem with $a = 1, b = 2, k = 0$, since $a = 1 = 2^0 = b^k$, we are in the steady state case, and $T(n) \in \Theta(n^k \log n) = \Theta(\log n)$.

All consecutive sums Give an efficient algorithm for the following problem: You are given an array $A[1..n]$ of real numbers, and want to compute the upper half of an $n \times n$ matrix $Sums[1..n, 1..n]$ where for each $1 \leq I \leq J \leq n$, $Sums[I][J] = \sum_{\{I \leq K \leq J\}} A[K]$. (Try to beat the obvious $O(n^3)$ algorithm.)

Say we split the array into $A[1..n/2]$ and $A[n/2 + 1..n]$. The set of all consecutive sums between I and J fall into three groups: Ones with $1 \leq I \leq J \leq n/2$, ones with $n/2 + 1 \leq I \leq J \leq n$ and those with $1 \leq I \leq$

$n/2 < J \leq n$. Those in the first group can be computed by a recursive call to the first half of the array, those in the second, by a recursive call to the second half of the array.

For the third category, we can use the fact that $Sum[I][J] = \sum_{I \leq K \leq J} A[K] = \sum_{I \leq K \leq n/2} A[K] + \sum_{n/2+1 \leq K \leq J} A[K] = Sum[I][n/2] + Sum[n/2+1][J]$. This leads to the following recursive algorithm:

AllConsSums(A[1..n])

1. Initialize an $n \times n$ array $Sums[1..n][1..n]$
2. If $n = 1$ THEN $Sums[1][1] \leftarrow A[1]$; return Sums;
3. $m \leftarrow \lfloor n/2 \rfloor$
4. $Sums[1..m][1..m] \leftarrow AllConsSums(A[1..m])$.
5. $Sums[m+1..n][m+1..n] \leftarrow AllConsSums(A[m+1..n])$.
6. FOR $I = 1$ to m do:
7. FOR $J = m+1$ to n do:
8. $Sums[I][J] \leftarrow Sums[I][m] + Sums[m+1][J]$.
9. Return Sums.

Since the two recursive calls are to arrays of size roughly $n/2$ and the two nested for loops go through $n/2$ iterations each, we have $T(n) = 2T(n/2) + \Theta(n^2)$. We can apply the Main Recursion Theorem with $a = 2, b = 2, k = 2$, to see that, since $2 < 2^2$, we are in the top-heavy case, and $T(n) \in \Theta(n^2)$.

Implementation: 20 pts Implement the grade-school and clever divide-and-conquer multiplication algorithms, where the input is coded as an array of digits. Plot both performances on a log-log scale, for random integers of length n for n different powers of 2. Then combine them to use a threshold T as follows: IF $n < T$ use Gradeschool Multiply ELSE use the Divide-and-Conquer recurrence (but recursive calls are to the thresholded algorithm). What is the value of T that gives the best performance on random n digit numbers? Is it bigger or smaller than the cross-over point where divide-and-conquer beats gradeschool multiplication? Show the data to support your conclusions.

You should see that the optimal threshold, which can depend on architectural details, is much less than the crossover point without a threshold. Since the d&c algorithm is bottom-heavy, putting in even a small threshold dramatically improves its running time, which moves the crossover point ever lower.