CSE 101 Calibration Homework
Fall, 2002
Background, (Order and Recurrence Relations, simple algorithms), MergeSort
100 points total, DOES NOT COUNT TOWARDS GRADE

Order Is $4^{\lceil \log n \rceil} \in O(n^2)$? Why or why not? (When unspecified, logs are base 2).

Yes, For any $n \geq 1$, $4^{\lceil \log n \rceil} \leq 4^{\log n+1} = (2^2)^{\log n+1} = 2^{2\log n+2} = 42^{2\log n} = 42^{\log n^2} = 4n^2$. So using $n_0 = 1, c = 4$ in the definition of order, the function is in $O(n^2)$.

Is $\log(n!) \in \Omega(n \log n)$? Why or why not?

Yes. $n! = n \cdot (n-1) \cdot ... 2 \cdot 1 < n \cdot n \cdot n ... \cdot n$ (n times). If we look at the first $n/2$ factors, they are each at least $n/2$, and all the others are at least 1. Thus $n! \geq (n/2)^{n/2}$, and $\log n! \geq \log(n/2)^{n/2} = n/2 \log(n/2) = n/2(\log n - 1) = n/2 \log n - n/2$. By the 6th property of order on page 36, $n/2 \in o(nlogn)$, so subtracting it does not change the order. Thus, ipso chango, we have: $\log n! \geq n/2 \log n - n/2 \in \Omega(n \log n)$, and so $\log n! \in \Omega(n \log n)$.

Is $4^n \in O(2^n)$? Why or why not?

No. To get a contradiction, assume it were in $O(2^n)$. Then there would exist $n_0, c > 0$ so that for each $n \geq n_0$, $4^n \leq c2^n$. Then we would also have $4^n/2^n \leq c$, but $4^n/2^n = (4/2)^n = 2^n$ is greater than a constant c for $n \geq \log c$. From this contradiction, $4^n \notin O(2^n)$.

Is $n + (n-1) + (n-2) + ... + 1 \in O(n)$? Why or why not?

Some students are tempted to say, "A sum's order is its largest term, and the largest term here is n, so the sum is $O(n)$." However, that only applies to sums of CONSTANTLY many terms, not a number of terms that grows with n. As can be seen here, $n+(n-1)+...+1 = n(n-1)/2 \in Theta(n^2)$, so it is NOT in $O(n)$.

Page 496-501 of text; 10 pts.each 1e, 6

Triangles(20 points) Let G be an undirected graph with nodes $v_1, ..v_n$. The *adjacency matrix* representation for G is the $n \times n$ matrix M given by: $M_{i,j} = 1$ if there is an edge from v_i to v_j, and $M_{i,j} = 0$. A *triangle* is a set $\{v_i, v_j, v_k\}$ of 3 distinct vertices so that there is an edge from v_i to v_j, another from v_j to v_k and a third from v_k to v_i. Give and analyze an algorithm for counting the number of triangles in a graph G, given by its adjacency matrix representation. Analyze your algorithm's worst-case performance first in terms of just the number of nodes n of the graph, then in terms of n and the number of edges m of the graph. Your algorithm should be faster when $m << n^2$.

The obvious method is to look at all triples of nodes and check whether they are all connected. This would be $O(n^3)$ to check. But we can do better for sparse graphs, $m \in o(n^2)$. The high level strategy is, instead of checking for pairs of nodes, check for each edge and each node not on the edge, whether the two endpoints and the additional node form a triangle. In the matrix representation, this would work as in the following pseudo-code.

Triangles[M:[1..n][1..n]: matrix of Booleans]: list of triples of nodes (integers in the range 1,..n)

1. Initialize List as an empty list of triples.
2. FOR $I = 1$ TO $I = n - 2$ do:
3. FOR $J = I + 1$ TO $J = n - 1$ do:
4. IF $M[I, J] = 1$ THEN do:
5. FOR $K = J + 1$ TO $K = n$ do:
6. IF $M[I, K]$ and $M[J, K]$ then append (I, J, K) to List;
7. Return List;

While there are three nested loops, each taking $O(n)$ time, we can do better than $O(n^3)$ in our time analysis. The loop in lines 5 and 6 is only performed if $M[I, J]$, i.e., at most once per edge $\{I, J\}$. Thus, it is really an $O(n)$ loop executed m times, so the total time spent on lines 5 and 6 is $O(mn)$. The time spent on the rest of the algorithm is just $O(n^2)$ from the two nested loops, with the $O(1)$ time to check whether $M[I][J]$ in line 4. Thus the total time is $O(mn + n^2)$. Since you are told $m > n$, this is $O(mn)$.

Binary Conversion(15 points): Say that your input is a decimal representation of a number, given as an array of digits, $X[n-1], ...X[0]$, representing $X = \sum_{I=0}^{I=n-1} X[I]10^I$. Describe an algorithm to find the binary representation of X that runs in time $O(n^2)$.

The main strategy we'll use is: We can tell whether a decimal number is even or odd by looking at the least significant digit. We can then record this as the least significant bit. By themselves the other bits in binary represent $xdiv2$, so we divide by 2 and repeat.

To use this strategy, we'll need to see how long it takes to divide numbers. Fortunately we only need to divide by 2, not a general division algorithm. In fact, if we use the long division algorithm from grade school, we see that we only need one digit carry to divide by a single digit number like 2. Each step then becomes divide an at most 2 digit number by a one digit number, subtract the product of two one digit numbers from a two digit number, and use the resulting carry as the first digit for the next

operation, bringing down one digit of the input string. So the total work of long division by 2 is $O(n)$, where n is the number of digits in the number we are halving. We can think of the long division algorithm as given to us as a procedure $LDiv2$ that takes an input X as its aarray of n single digits, and returns $Xdiv2$ expressed in decimal as its array of at most n single digits. Then our binary conversion algorithm is: Convert(X[0..n-1]: array of digits): array of bits

1. Initialize $BX[0,..4n]$ array of bits.
2. $I \leftarrow 0$ {a pointer to which bit we are computing}
3. Until $I > 4n$ do:
4. begin;
5. BX[I] $\leftarrow X[0]div2$;
6. $X \leftarrow LDiv2[X]$;
7. I++;
8. end;
9. Return BX (possibly removing initial 0's, if you want).

A few things need explaining. First, why did we initialize $4n$ bits in BX? This is because $10 < 16 = 2^4$, so an n digit number X is at most $10^n < 2^{4n}$, so the length of X in binary is at most 4 times its length in decimal. Thus, the main loop executes $O(n)$ times, and each time it calls the $O(n)$ time operation $LDiv2$. Since $X[0]$ is just a single digit, we can take $X[0]div2$ in $O(1)$ time, and so the rest of the loop is $O(1)$. Thus, the inside of the loop is $O(n)$, and we repeat it 4n times, so the total time is $O(n^2)$. $4n$ is an overestimate on the number of bits required, so we could then go back and decrease the dimension of the array until we see a bit with value 1. This would be an additional $O(n)$ time, which would not change the $O(n^2)$ total complexity.

Summing triples (20 points) Let $A[1,...n]$ be an array of positive integers. A *summing triple* in A is 3 distinct indices $1 \leq i, j, k \leq n$ so that $A[i] + A[j] = A[k]$. Give and analyze an algorithm that, given A, determines whether there is any summing triple in A. Try to be better than $O(n^3)$.

If we try all triples, we would take $O(n^3)$. We would compute, for each $A[I]$ and $A[J]$, $V = A[I] + A[J]$ and check for each $A[K]$ whether the sum equals $A[K]$. But a simple observation will save considerable time: it is easier to check whether an element is in a sorted array than an unsorted array. In fact, if we are going to repeatedly do such checks, say n^2 times as above, it is worthwhile to sort the array before starting. The pseudo-code, using sorting and binary search sub-routines from the textbook, would be as follows: MergeSort is a sorting algorithm, and BinSearch[A,v] checks whether value v is in sorted array A.

1. SumCheck[A[1...n]: array of integers]: Boolean;
2. $A[1.n] \leftarrow MergeSort[A]$
3. $Found \leftarrow False, I \leftarrow 1$
4. While $Found = False$ and $I \leq n$ do:
5. begin;
6. $J \leftarrow I$;
7. While $J \leq n$ and $Found = False$ do:
8. begin;
9. IF $BinSearch[A, A[I] + A[J]]$ THEN $Found \leftarrow True$;
10. J++
11. end;
12. I++;
13. end;
14. Return Found;

The above algorithm uses binary search for each I, J until it finds a pair whose sum is in A; it exits the loop once one is found.

The total time is $O(n^2 \log n)$, because the first sorting is $O(n \log n)$, then there are two nested loops of $O(n)$ length each, with an $O(\log n)$ time binary search procedure inside the two, giving a total time of $O(n^2 \log n + n \log n) = O(n^2 \log n)$.

It is not too hard to improve this to $O(n^2)$. Any group who gave a correct $O(n^2)$ (with correct argument and time analysis) solution or an individual who hands in a correct $O(n^2)$ solution (with correctness proof and time analysis) by Tuesday gets one extra-credit point added to their total grade.

Implementation (20 points) Implement the algorithm you gave for the summing triples problem above. Try it on random arrays where each element $A[i]$ is chosen in the range $1...n$, for $n = 2^6, 2^8, 2^10, 2^12, 2^14, 2^16, 2^18$ and 2^20. Plot its performance on a $log_2 n$ vs. log_2 of the time scale. Then try the same experiment on random arrays where each element is chosen in the range $1, ..n^2$. Do you see a difference? If so, can you explain it?

See the course webpage for an implementation.

The first distribution produces arrays where it is very likely that there is a sum whose first element is one of the smallest few elements of the array. The main loops in the above algorithm should almost always terminate before I reaches 2. So the MergeSort will actually take most of the time. When you plot a log-log curve, you should get almost a line with slope very close to 1, $\log T = \log(cn \log n) = \log n + \log \log n + \log c$.

The second distribution produces arrays where it is less likely to be any sum pair, at least not involving small values of I. This distribution should give a log-log curve closer to a line withslope 2, $\log T = \log(cn^2 \log n) = 2\log n + \log\log n + \log c$. Actually, I made a mistake: to get very small probabilities of sum pairs, I should have said, pick values in the range $[1, ..n^3]$; in the above examples, Joe's implementation found sums in about as much time as to do the MergeSort, so the slope was still 1.