

CSE 101 Calibration Homework

Spring, 2003

Background, (Order and Recurrence Relations, simple algorithms), MergeSort

100 points total, DOES NOT COUNT TOWARDS GRADE

Answer Key

Analyzing Loops Assume $\text{proc}(I)$ is an algorithm that takes $\Theta(I)$ time and does not change I . What is the order of the running time of the following two algorithms?

Alg1(n)

1. begin;
2. $I \leftarrow 1$;
3. While $I \leq n$ do:
4. begin;
5. $\text{proc}(I)$
6. $I++$
7. end;
8. end;

The time for the inside loop is bounded by cI for some $c > 0$, since $\text{proc}(I)$ takes $\Theta(I)$ time. Since I ranges from 1 to n , this is at most cn time for each of n loops, for a total time of $O(n^2)$. Also, $\text{proc}(I)$ takes at least time $c'I$, so the total time is at least $\sum_{I=1}^n c'I = c'(n)(n+1)/2 \in \Omega(n^2)$, so the total time is $\Theta(n^2)$, being both $O(n^2)$ and $\Omega(n^2)$.

Alg2(n)

1. begin;
2. $I \leftarrow 1$;
3. While $I \leq n$ do:
4. begin;
5. $\text{proc}(I)$
6. $I \leftarrow 2 * I$
7. end;
8. end;

You might think $n \log n$ is the time's order, since there are $\log N$ iterations of at most time $O(n)$. But we can do better through enhanced precision: On the t 'th loop, I has value $I_t = 2^{t-1}$, and we halt when $I_t = 2^{t-1} > n$,

i.e, when $t > \log_2 n + 1$. The inside proc takes at most time $cI_t = c2^{t-1}$, so the total time is at most $\sum_{t=1}^{t=\log n+1} c2^{t-1} = c \sum_{k=0}^{k=\log t} 2^k = c(2^{\log n+1} - 1) < 2cn \in O(n)$. During the last loop, $I_t > n/2$, so the last proc is $\Omega(n/2) = \Omega(n)$, so the time is $\Theta(n)$.

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

Yes, For any $n \geq 1$, $4^{\lceil \log n \rceil} \leq 4^{\log n+1} = (2^2)^{\log n+1} = 2^{2\log n+2} = 4 \cdot 2^{2\log n} = 4 \cdot 2^{\log n^2} = 4n^2$. 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. $n! = n \cdot (n-1) \cdot \dots \cdot 2 \cdot 1 < n \cdot n \cdot n \dots \cdot n$ (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, $n/2 \in o(n \log n)$, 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 $n + (n-1) + (n-2) + \dots + 1 \in O(n)$? 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, $n + (n-1) + \dots + 1 = n(n-1)/2 \in \Theta(n^2)$, so it is NOT in $O(n)$.

Triangles(20 points) Let G be an undirected graph with nodes v_1, \dots, 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 determining if a graph G , given by its adjacency matrix representation, has a triangle. 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 $m \ll n^2$.

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. $Found \leftarrow False$.
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 $Found \leftarrow True$
7. Return $Found$;

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 $m \in O(n^2)$, this is always $O(n^3)$ as above, but it is faster if m is small.

Binary Conversion(10 points): Consider the following algorithm to convert a decimal number to binary. More precisely, the input is a decimal representation of a number, given as an array of digits, $D[n - 1], \dots, D[0]$, representing $X = \sum_{I=0}^{I=n-1} D[I]10^I$. The output should be an array of bits $B[n' - 1] \dots B[0]$ so that $X = \sum_{I=0}^{I=n'-1} B[I]2^I$.

The following algorithm uses a “long division by two” algorithm *LDIV* that takes linear time ($O(n)$) to compute the decimal representation of $X \text{div} 2$, given X in decimal..

The binary conversion algorithm is: Convert(D[0..n-1]: array of digits): array of bits

1. Initialize $B[0, ..4n - 1]$ array of bits.
2. FOR $I = 0$ TO $4n - 1$ do:
3. begin;
4. $B[I] \leftarrow D[0] \text{mod} 2$;
5. $D \leftarrow LDiv2[D]$;
6. end;

7. Return B (possibly removing initial 0's, if you want).

Prove that this algorithm is correct, and give a time analysis in terms of the number n of digits.

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. Also, why is $LDiv2$ $O(n)$? We saw in class that dividing an n digit number by an m digit number can be done in time $O(m(n - m))$, here $m = 1$ is a constant, so this is $O(n)$.

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.

Why is the algorithm correct? Let x_t be the integer represented by the array D in decimal, before the iteration when $I = t$; and z_t be that represented by the array B in binary. We want x_0 , the input, to equal z_{4n} , the output. Originally, B is initialized to 0, so $z_0 = 0$. The idea of the algorithm is that z_t will be the least significant t bits of x_0 in binary, and x_t will be converted and placed in the $t + 1 \dots 4n$ 'th bit places. Thus, the value x_0 should always be in binary, x_t (converted) followed by z_t , so x_0 should be equal to $x_t(2^t) + z_t$. We'll prove $x_0 = x_t(2^t) + z_t$ as a loop invariant by induction on t .

First, the base case is $t = 0$, when $z_0 = 0$. Thus $x_0 = x_0(1) + 0 = x_0(2^0) + z_0$, so the loop invariant holds as a precondition before the loop starts. Assume the loop invariant holds before iteration $I = t$, $x_0 = x_t(2^t) + z_t$. If x_t is odd, $x_{t+1} = x_t div 2 = (x_t - 1)/2$. Then z_{t+1} in binary is a 1 in the t 'th position, followed by z_t in binary, so $z_{t+1} = 2^t + z_t$. Then $x_{t+1}(2^{t+1}) + z_{t+1} = (x_t - 1)/2(2^{t+1}) + 2^t + z_t = x_t(2^t) - 2^t + 2^t + z_t = x_t(2^t) + z_t = x_0$. If x_t is even, $x_{t+1} = x_t/2$ and z_{t+1} in binary is z_t preceded by a 0, which doesn't change its value. Therefore $x_{t+1}2^{t+1} + z_{t+1} = x_t/2(2^{t+1}) + z_t = x_t(2^t) + z_t = x_0$. So in either case, the invariant still holds.

Thus, at the end of the loop, $x_0 = x_{4n}2^{4n} + z_{4n}$. Now, since $x_0 < 10^n < 16^n = 2^{4n}$, and since x_{4n} and z_{4n} are non-negative integers, this means $x_{4n} = 0$. Thus, at the end of the loop $z_{4n} = x_0$, so the output represents in binary what the input represents in decimal. Thus, we've proved the algorithm is correct.

Summing triples (20 points) Let $A[1, \dots, n]$ 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. $A[1..n] \leftarrow \text{MergeSort}[A]$
3. $Found \leftarrow \text{False}, I \leftarrow 1$
4. While $Found = \text{False}$ and $I \leq n$ do:
5. begin;
6. $J \leftarrow I$;
7. While $J \leq n$ and $Found = \text{False}$ do:
8. begin;
9. IF $\text{BinSearch}[A, A[I] + A[J]]$ THEN $Found \leftarrow \text{True}$;
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 $O(n^2)$.

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?

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 with slope 2, $\log T = \log(cn^2 \log n) = 2 \log n + \log \log n + \log c$.

Thus, when you plot the second you should get approximately a line with twice the slope of the first distribution. The moral is: Not all kinds of inputs give the worst-case complexity; and, average-case complexity is sensitive to the exact distribution of inputs, and so is not a terribly robust measure of complexity.