Analyzing Loops Assume proc(I) is an algorithm that takes Θ(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 ← 1;
3. While I ≤ n do:
   4. begin;
   5. proc(I)
   6. I++
   7. end;
8. end;

The time for the inside loop is bounded by cI for some c > 0, since proc(I) takes Θ(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, proc(I) takes at least time c'I, so the total time is at least ∑_{I=1}^{n} c'I = c'(n)(n+1)/2 ∈ \Omega(n^2), so the total time is Θ(n^2), being both O(n^2) and \Omega(n^2).

Alg2(n)
1. begin;
2. I ← 1;
3. While I ≤ n do:
   4. begin;
   5. proc(I)
   6. I ← 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 i’th loop, I has value \(I_i = 2^{i-1}\), and we halt when \(I_i = 2^{i-1} > n\),
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 $\sum_{t = \log_2 n +1}^{\infty} c2^{t-1} = c \sum_{k=0}^{n} 2k = c(2^2 \log_2 n + 1) = 2^4 n = 42$, i.e, when $t > \log n$. 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 $4^{\log n} \in O(n^2)$? Why or why not? (When unspecified, logs are base 2).

Yes, For any $n \geq 1$, $4^{\log n} \leq 4^{\log n+1} = (2^2)^{\log n+1} = 2^2 \log n + 2 = 42^2 \log n = 42 \log n^2 = 4n^2$. 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. $n! = n \cdot \ldots \cdot 2 \cdot 1 < n \cdot n \cdot n \ldots n$ (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 $n + (n - 1) + (n - 2) + \ldots + 1 \in O(n)$? 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, $n + (n - 1) + \ldots + 1 = n(n-1)/2 \in \Theta(n^2)$, so it is NOT in $O(n)$.

**Summing triples (20 points)** Let $A[1, \ldots n]$ 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 \(\leftarrow\) 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;


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 approximatively 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.