

C241 Homework 10: Time Complexity

Solutions

1) Find the asymptotic complexity (the closest fitting Big-O: $O(1)$, $O(\log_2(n))$, $O(n \log_2(n))$, $O(n^2)$, etc...) for the following functions:

a) $f(n) = 3n + 7$

This is $O(n)$, since $3n$ is the dominating term

b) $f(n) = 3 + \sin(1/n)$

This is $O(1)$ (or constant), since $\sin(1/n)$ will never be more than 1 and thus the total function will always have a value between 3 and 4.

c) $f(n) = 3n^3 - 5n^2 + 25n - 165$

This is $O(n^3)$, since n^3 is the dominating term

d) $f(n) = 5n^2 + 3n \log_2(n)$

$\log_2(n)$ grows more slowly than n , so $5n^2$ is the dominating term here. This is $O(n^2)$.

e) $f(n) = n^2 + (n - 1)^3$

When you expand the $(n - 1)^3$ term, you get a n^3 term, which dominates the n^2 . This is $O(n^3)$

f) $f(n) = \frac{n(n+1)(n+2)}{(n+3)}$

When you multiply out the top of the fraction, the fastest growing term is n^3 . However, after you divide out by the $(n + 3)$ on the bottom of the fraction, the largest term in the function will be some multiple of n^2 . This is $O(n^2)$.

g) $f(n) = 2 + 4 + \dots + 2n$

This is equal to $2(1 + 2 + 3 + \dots + n) = 2 \sum_{i=0}^n i = 2 \frac{n(n+1)}{2} = n(n+1) = n^2 + n$. So this is $O(n^2)$. Remember this, it'll come in handy later!

h) $f(n) = 56n^3 + 2^n$

2^n grows much, much faster than n^3 , (in fact, y^n for any value of y will always grow much faster than any n^x , regardless of what x is). So this is $O(2^n)$. We say this function takes "exponential time".

2) Let $f(n) = n + 100$ and $g(n) = n^2$.

a) Use the definition of Big-O to prove that $f(n) \in O(g(n))$

The definition of Big-O states that $f(n) \in O(g(n))$ if and only if $\exists m \in \mathbb{N}$ and $c \in \mathbb{R}^+$ such that $f(n) \leq c \times g(n), \forall n \geq m$. So if we want to show that $n + 100 \in O(n^2)$, we need to show that there is some integer m and some positive real number c such that $n + 100 \leq c(n^2)$ for all $n \geq m$. I think that $c = 101$ and $m = 1$ will work. Let's prove it:

$$n + 100 \stackrel{?}{\leq} 101(n^2)$$

$$n + 100 \stackrel{?}{\leq} n^2 + 100n^2$$

We can prove this by showing that $n \leq n^2$ and $100 \leq 100n^2, \forall n \geq 1$.

$$n \stackrel{?}{\leq} n^2$$

$1 \leq n$ This is clearly true for all $n \geq 1$.

$$100 \stackrel{?}{\leq} 100n^2$$

$1 \leq n^2$ This is also clearly true for all $n \geq 1$.

Since we've found a valid choice for m and c , we satisfied the definition of Big-O, and we know that $f(n) \in O(g(n))$.

b) Use the definition of Big-O, and a proof by contradiction, to prove that $g(n) \notin O(f(n))$

We're trying to disprove this, so let's do it by contradiction. We'll start out by assuming that it's true that $n^2 \in O(n + 100)$, and then we'll show how if that *was* true something else that's clearly impossible would also have to be true. This will mean that our original assumption $n^2 \in O(n + 100)$ must be wrong, and in fact $n^2 \notin O(n + 100)$.

So, if it were true that $n^2 \in O(n + 100)$, then there must be an c, N such that for all $n > N, n^2 \leq c(n + 100)$.

We can write this as: $n^2 \leq cn + c100$ for all $n > N$

which is equivalent to: $n^2 - cn \leq c100$ for all $n > N$

which we can write as: $n(n - c) \leq c100$ for all $n > N$.

But, if we're claiming this works for *all* $n > N$, it will be true for some values of n that are greater than $c \times 100$. (since we're claiming it works for *all* numbers greater than some finite N , it will work for arbitrarily large values of n , including an infinite number of values that are bigger than $c \times 100$).

But look, if we plug $n = 100c$ into our inequality, then we get:

$100c(100c - c) = 100c(99c) \leq 100c$, which is blatantly false. And if our inequality isn't even true for $n = 100c$, it surely won't be true for the infinite number of values of n that are *greater* than $100c$.

Thus we've found our contradiction, and our original assumption must have been wrong.

Therefore $n^2 \notin O(n + 100)$

3) Let $f(n) = n$ and $g(n) = n + (1/n)$.

a) Use the definition of Big-O to prove that $f(n) \in O(g(n))$

b) Use the definition of Big-O to prove that $g(n) \in O(f(n))$

These proofs will use the same definition we used above.

Proof a: $n \in O(n + (1/n))$

Best not to think about this one too hard. $c = 1, m = 1$ will work:

$$n \stackrel{?}{\leq} n + (1/n)$$

$0 \leq (1/n)$ This is clearly true for all $n \geq 1$.

Proof b: $n + 1/n \in O(n)$

We'd like to have as many terms on the right as we have on the left, so let's choose $c = 2, m = 1$:

$$n + 1/n \stackrel{?}{\leq} 2(n)$$

$$n + 1/n \stackrel{?}{\leq} n + n$$

$$1/n \stackrel{?}{\leq} n$$

$1 \leq n^2$ This is clearly true for any $n \geq 1$.

4) Use Numerical Induction on k and the definition of Big-O to prove that for all $k \in \mathbb{N}$: $A_0n^0 + A_1n^1 + A_2n^2 \dots + A_{k-1}n^{k-1} + A_kn^k \in O(n^k)$. Here A_0, A_1, \dots, A_n represent any arbitrary coefficients chosen from \mathbb{N} .

Base Case: $A_0n^0 + A_1n^1 \in O(n^1)$

We can simplify this slightly: $A_0 \times 1 + A_1n \in O(n)$

$A_0 + A_1n \in O(n)$

Now, using the definition of Big-O, let's pick $C = A_0 + A_1 + 1$ and $N = 1$. Then the question becomes:

$A_0 + A_1n \leq (A_0 + A_1 + 1)n, \forall n > 1$.

$A_0 + A_1n \leq A_0n + (A_1 + 1)n, \forall n > 1$

$0 \leq A_0(n - 1) + n$

This is clearly true for all $n > 1$.

Induction Hypothesis: Assume $A_0n^0 + A_1n^1 + A_2n^2 \dots + A_{k-1}n^{k-1} + A_kn^k \in O(n^k)$. In other words, assume that there is a C_n and N such that: $A_0n^0 + A_1n^1 + \dots + A_kn^k \leq C_n n^k$, for all $n > N$.

Induction Step: Show that $A_0n^0 + A_1n^1 + \dots + A_kn^k + A_{k+1}n^{k+1} \in O(n^{k+1})$.

Well, we know that $A_0n^0 + A_1n^1 + \dots + A_kn^k \leq C_n n^k$ for all $n > N$,

So we also know that: $(A_0n^0 + A_1n^1 + \dots + A_kn^k) + A_{k+1}n^{k+1} \leq C_n n^k + A_{k+1}n^{k+1}$.

And, for any $n > 1$, it's clear that: $C_n n^k + A_{k+1}n^{k+1} \leq C_n n^{k+1} + A_{k+1}n^{k+1} = (C_n + A_{k+1})n^{k+1}$

So, if we pick $C_{n+1} = (C_n + A_{k+1})$ we have that:

$A_0n^0 + A_1n^1 + \dots + A_kn^k + A_{k+1}n^{k+1} \leq C_{n+1}n^{k+1}$ for all $n > N$.

So by the definition of Big-O, $A_0n^0 + A_1n^1 + \dots + A_kn^k + A_{k+1}n^{k+1} \in O(n^{k+1})$

5) Find the time complexity of the following pieces of code:

a)
for(i : 1 to n)
 x = x + 1

This loop executes n times, so it's time complexity will be some constant (the approximate amount of time it takes to do a single execution of the loop) times n . In other words, it'll be $O(n)$.

b)
for(i : 1 to n)
 for (j:1 to m)
 x = x + 1

Each time the outer loop executes, the inner loop executes m times, thus the total work of the loop will be repeated $n \times m$ times: $O(nm)$

c) (note that the index k is used in both loops)
for (k : 1 to n)
 for (j: 1 to k)
 x = x + 1

This one is kinda neat. On the first iteration of the outer loop $k = 1$, and the inner loop will execute only once. On the second iteration of the outer loop $k = 2$, and the inner loop will execute twice. This pattern continues until $k = n$ and the outer loop stops running. So, in total the inner loop will execute: $1+2+3+\dots+n$ times. This should look familiar! (if it doesn't yet, trust me it will before you're done with your degree). $1+2+3+\dots+n = n(n+1)/2 = (1/2)(n^2+n)$. This loop will run in time $O(n^2)$.

d)
for (i : 1 to n)

```
y = y + 1
for (h: 1 to p)
  z = z + 3
for (k: 1 to q)
  x = x + 1
```

Every time the outer loop here executes, the first inner loop will execute p times and the second inner loop will execute q times. Since we don't know what p and q are, we can't say which one will dominate. So, we can write this time complexity two ways: $O(n(p + q))$ or $O(n * \max\{p, q\})$

```
e)
for (k : 1 to n)
  for (j: 1 to  $n^2$ )
    x = x + 1
```

Each time the outer loop executes, the inner loop executes n^2 times. This will have time complexity $O(n^3)$.

Proofs:

6) Easy Proof: Prove that following problem can be solved in linear time, by describing in simple pseudo-code how you would solve it and showing that your algorithm takes time $O(n)$: Given an list of n numbers, what's the smallest number in the list?

Min-finder:

```
min = A[1]
for (i : 1 to n)
    if (A[i] < min)
        { min = A[i] }
```

Since this code will check every number in the list A once, (and there are n of those numbers), it will take $O(n)$ time to run. Thus, this problem can be solved in time $O(n)$.

7) Medium/Hard Proof: Describe in simple pseudo-code an algorithm to solve the following problem. Show that you can solve it in $O(n^2)$ time (do better than that for extra-credit): Given an array of n positive integers, return an array with the numbers in sorted order from least to greatest.

The simplest way to solve this is to make use of the Min-finding program we just wrote. First we find the smallest number in the input list, and we remove it from the input list and put it at the front of our sorted list. Then we do the same thing again, to find the second smallest number in the input list and put it at the second spot in our sorted list (in other words, append it to the end of our sorted list). This will end up with us using the Min-finding program n times (first on a list of size n , then on a list of size $n - 1$, then on a list of size $n - 2$, and so on.) And since the Min-finder takes time proportional to the length of the list that we feed it, our total sorting algorithm will take time: $n + (n - 1) + (n - 2) + \dots + 3 + 2 + 1 = \frac{n(n-1)}{2} \in O(n^2)$.

```
for (i : 1 to n)
  x = Min-Finder(A)
  A.Remove(x)
  S.Append(x)
return S
```

If we wanted to get $O(n \lg n)$ time, we could use mergesort. Check wikipedia, or your C211 notes, for details on this.

8) (extra credit, possibly fortune and fame) Very Hard Proof: The 'Satisfaction' problem is the classical example of an NP-complete problem: This means that no one has found an algorithm solve it which takes time less than $O(2^n)$, but we also haven't yet proven that that's impossible. Still, there are tricks which can speed the solution up for most inputs, and some of those aren't too hard to figure out. Take a shot at solving this problem yourself, and I'll give you some extra credit.

The Satisfaction Problem: Given a logic statement with n variables, in a form similar to: $(v_1 \vee v_2 \vee v_3) \wedge (\neg v_2) \wedge (v_2 \vee v_4)$ (A series of 'or' clauses, connected by 'ands', with the 'nots' only applied directly to the variables, not to clauses. This is called 'CNF' form): Is the statement a contradiction, or is it possible for the statement to evaluate to True (be 'satisfied')?