Tutorial 1: Asymptotic Notation

Example Solutions

1. Prove the following facts stated in Lecture Note 2.

   (a) For any constant $a > 0$ in $\mathbb{R}$: $f_1 = O(g_1) \implies af_1 = O(g_1)$.
   We are given that there are constants $n_0 \in \mathbb{N}$ and $c > 0$ such that

   $$0 \leq f_1(n) \leq cg_1(n)$$

   for all $n \geq n_0$. We want to find constants n_1 and $c_1 > 0$ such that

   $$0 \leq af_1(n) \leq c_1 g_1(n)$$

   for all $n \geq n_1$. Since $a > 0$ we have

   $$0 \leq af_1(n) \leq acg_1(n)$$

   so we can take $n_1 = n_0$ and $c_1 = ac$.

   (b) $f_1 = O(g_1)$ and $f_2 = O(g_2) \implies f_1 + f_2 = O(g_1 + g_2)$.
   Here we are given that there are $n_1, n_2 \in \mathbb{N}$ and $c_1, c_2 > 0$ such that

   $$0 \leq f_1(n) \leq c_1 g_1(n), \text{ for all } n \geq n_1;$$
   $$0 \leq f_2(n) \leq c_2 g_2(n), \text{ for all } n \geq n_2.$$

   So if we put $n_0 = \max(n_1, n_2)$ then the two pairs of inequalities hold at the same time and we can add them to obtain

   $$0 \leq f_1(n) + f_2(n) \leq c_1 g_1(n) + c_2 g_2(n),$$

   for all $n \geq n_0$. All that remains now is to take $c = \max(c_1, c_2)$ and observe that

   $$c_1 g_1(n) + c_2 g_2(n) \leq cg_1(n) + cg_2(n) = c(g_1(n) + g_2(n)).$$

   [In the above we have used the standard fact that if $a_1 \leq b_1$ and $a_2 \leq b_2$ then $a_1 + a_2 \leq b_1 + b_2$.]

2. Here we consider a function of the form

   $$g(n) = f(n) + \frac{a_1}{n} + \frac{a_2}{n^2},$$

   where f is some other function with $f(n) \geq 0$ for all n and a_1, a_2 are non-negative constants.

   (a) As n increases $1/n$ and $1/n^2$ both decrease. So they have their biggest value when $n = 1$. So $g(n) \leq f(n) + a_1 + a_2$ (what goes wrong if either of the constants is negative?). Well any additive constant can be replaced by 1 for the purposes of asymptotic notation. Thus $g = O(f + 1)$. (More formally, $f(n) + a_1 + a_2 = O(f(n)) + O(1) + O(1) = O(f(n) + 1)$.)

   (b) One problem with claiming that $g = O(f)$, is that f might be the zero function, i.e., $f(n) = 0$ for all n (actually all we need is that this happens for all large enough n). If it so happens that $a_1 > 0$ and $a_2 > 0$ then it is obvious that the claim is false.

   Of course there are conditions under which it is true, e.g., if $a_1 = 0$ and $a_2 = 0$. The point is that when we make an unqualified claim we are stating that it holds for all relevant objects (functions and constants in our case).

   Finally, in most applications f would be $\Omega(1)$ in which case the simplified claim is correct (but of course we would have to prove that $f = \Omega(1)$); under these conditions we can allow a_1, a_2 to be arbitrary constants.

3. (a) We use the standard assumption that each line of the algorithm takes constant time. The first loop is executed $O(n)$ times. The body of the loop executes lines 2 and 5 which take constant time as well as the loop at line 3. This loop itself is executed $O(n)$ times and its body takes constant time. Thus the body of the first loop takes time $O(1) + O(n) = O(n)$. It follows that the overall time is $O(n)O(n) = O(n^2)$.

   (b) Let $B = A$ (in fact any array that has 1 as an entry will do). The two loops are each executed n times since `found` is always set to TRUE by the inner loop so that in line 5 we do not return early. So the runtime is at least cn^2 for a constant c, i.e., $\Omega(n^2)$.

   Of course a sensible modification of the algorithm would be to exit the inner loop on lines 3 and 4 as soon as `found` is set to TRUE. If the algorithm is modified in this way then $B = \langle 2, 2, \ldots, 2, 1\rangle$ produces the worst case behaviour. Note that this observation is made here to further understanding, it is *not* required as part of the answer to the question (so bear this in mind for the exam). Can you think of other possible arrays that produce the worst case behaviour? Does 1 *have* to be at the end?

   If you have time you might like to implement the simple algorithm modified to have a counter that is initialized to 0 and is increased by 1 every time one either of the loops is executed. Get the algorithm to return this counter as the result. Don't spend a great deal of time on this, it can be done in an appropriate language within a matter of minutes.

   (c) Yes the runtime is $\Theta(n^2)$ since the runtime of the algorithm is both $O(n^2)$ and $\Omega(n^2)$.

   (d) Sort array B using any algorithm with $O(n \lg n)$ worst case runtime (e.g., mergesort, we will study this in the course if you have not yet come across it). Now replace the inner loop with binary search, so that this runs in time $O(\lg n)$. Thus the two loops together now run in time $O(n)O(\lg n) = O(n \lg n)$. Hence the overall runtime is $O(n \lg n) + O(n \lg n) = O(n \lg n)$.

4. (a) For $S = \{\{x_1, \overline{x_2}\}, \{x_2\}\}$ we have

   $$R(S) = S \cup \{\{x_1\}\}$$
   $$R^2(S) = R(S).$$

Since the empty clause is not found we conclude that the formula is satisfiable (e.g., by seting $x_1 = \text{TRUE}$ and $x_2 = \text{TRUE}$).
For $S = \{\{x_1, \overline{x_2}\}, \{\overline{x}_1\}, \{x_2\}\}$ we have

$$R(S) = S \cup \{\{\overline{x}_2\}\}, \{\{x_1\}\},$$
$$R^2(S) = R(S) \cup \{\{\}\}.$$

Since we have found the empty clause we do not need to carry on (this is an obvious optimisation of the algorithm). We conclude that the formula is not satisfiable. Check this by trying all 4 possible assignments. Quite often resolution will produce the empty clause early on, obviously this is much better than trying all 2^n possible assignments.

(b) There are finitely many clauses so the set of resolvents cannot keep growing, eventually it must repeat.

(c) The first important point to observe is that the resolvent of two clauses each with at most two literals is another such clause. So we should find out how many such clauses there are (any good upper bound would do but the situation is so simple that we can be precise). We have $2n$ distinct literals (n variables and n negated ones). Thus the number of clauses with one literal is $2n$. The number of clauses with two literals is $\binom{n}{2} = n(n-1)/2$. Of course there is only one clause with no literals. So the number of clauses under consideration is $1 + 2n + n(n-1)/2 = O(n^2)$.

At each stage we thus have $O(n^2)$ clauses and as we compare pairs there are $O(n^2) \times O(n^2) = O(n^4)$ to consider. Since comparison is assumed to be constant time this gives a bound for the time of each stage. The number of stages cannot be more than the total number of clauses (consisting of at most two literals) and is thus $O(n^2)$. So the overall runtime is bounded by $O(n^2) \times O(n^4) = O(n^6)$.

(d) The problem with having clauses that consist of three literals is that their resolvents can lead to clauses with more than three literals. These then can lead to bigger ones etc. In the worst case we obtain exponentially many clauses.

It is worth noting that nobody knows of an algorithm that is not exponential in the worst case. It is not expected that there is one but to date no proof is known. An answer to this question would be of very great significance (and win a large cash prize). The satisfiability problem is the archetypal example of an NP-complete problem. The study of these has kept many computer scientists busy and gainfully employed since the 1970's.