Tutorial 1: Asymptotic Notation

It is not expected that you will cover all of the following in the tutorial. Each group should choose exercises that it finds helpful and discuss their solution with guidance from the tutor. In this regard it is important to understand why some approach might be incorrect or unsuitable; explore possibilities. Naturally you will derive greatest benefit if you prepare for the tutorial. Any remaining exercises should be considered as part of private study. It is essential that you do enough exercises so that using the notation becomes second nature. It is a very big error to assume that just by looking at enough sample answers everything will be alright (this is no different from learning any other skill: would you assume that to become a good guitarist all you need to do is watch enough people playing the guitar?).

1. Prove the following facts stated in Lecture Note 2.
   (a) For any constant \( a > 0 \) in \( \mathbb{R} \): \( f_1 = O(g_1) \Rightarrow af_1 = O(g_1) \).
   (b) \( f_1 = O(g_1) \) and \( f_2 = O(g_2) \) \( \Rightarrow \) \( f_1 + f_2 = O(g_1 + g_2) \). Note that the proof for this can be adapted easily to prove the corresponding fact about big-\( O \).

   These are very straightforward. Start by writing down what is given and think about what is required to prove the claim, fill in the gap.

2. 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 such that \( f(n) \geq 0 \) for all \( n \) and \( a_1, a_2 \) are non-negative constants.
   (a) Explain very briefly why we would expect that \( g(n) \geq 0 \) for all \( n \).
   (b) We might be tempted to claim that \( g = O(f) \), explain by means of a counterexample why this is not always the case (there is a very simple reason and counterexample here).

3. Let \( A \) be an algorithm and recall that the worst case runtime function \( T_A(n) \) is defined to be the maximum runtime of \( A \) over all inputs of size \( n \). Suppose we have two functions \( L(n) \) and \( U(n) \) such that
   \[ L(n) \leq T_A(n) \leq U(n) \]
   for all \( n \). State which of the following is true and which is false and justify your answer.
   (a) \( T_A = \Omega(L) \).
   (b) \( T_A = O(U) \).
   (c) For every input of size \( n \) the runtime of \( A \) is at most \( U(n) \).
   (d) For every input of size \( n \) the runtime of \( A \) is at least \( L(n) \).
   (e) There is an input of size \( n \) such that the runtime of \( A \) is at least \( L(n) \).

4. Recall that by definition \( n! = 1 \times 2 \times \cdots \times n \).
   (a) Prove that \( \lceil n/2 \rceil ^{\lceil n \rceil} \leq n! \leq n^n \).
   (b) Deduce that \( \log n! = \Theta(n \log n) \), here we assume that \( n > 0 \) so that the logarithm is defined.

5. In this part \( A \) and \( B \) are two arrays of size \( n \) with integer entries. We consider the problem of deciding if every entry in \( A \) is also an entry in \( B \) (it is possible that \( B \) has entries that do not occur in \( A \)). Consider the following pseudocode for solving this problem.

   \[
   \text{subArray}(A, B)
   \]
   1. \( \text{for } i \leftarrow 0 \text{ to } n - 1 \) do
   2. \( \quad \text{found} \leftarrow \text{FALSE} \)
   3. \( \quad \text{for } j \leftarrow 0 \text{ to } n - 1 \) do
   4. \( \quad \quad \text{if } A[j] = B[j] \text{ then } \text{found} \leftarrow \text{TRUE} \)
   5. \( \quad \text{if not found then return } \text{FALSE} \)
   6. \( \text{return } \text{TRUE} \)

   (a) Prove that the worst case runtime of the algorithm is \( O(n^2) \). (You should use the standard "laws" of \( O() \) rather than the crude method of naming constants.)
   (b) Suppose \( A \) is the array \((1, 1, \ldots, 1)\). Describe an array \( B \) which ensures that the algorithm has runtime \( O(n^2) \). Prove that the runtime is as stated.
   (c) Is it true to say that the worst case runtime of the algorithm is \( \Theta(n^2) \)? Justify your claim briefly.
   (d) Explain what changes you would make so that the algorithm has worst case runtime \( O(n \log n) \). For this should refer to any standard algorithms by name without further discussion of them but must state their worst case runtimes. Prove that the runtime is as claimed. Note that this assumes you have already met certain algorithms (we will cover similar ones later in the course); if this assumption is wrong then this part is more difficult than intended.

6. This exercise is concerned with an algorithm that is used in practice. There is a fair amount of set up needed to explain it but the ideas and questions are in fact fairly simple.

   Let \( x_1, x_2, \ldots, x_n \) be boolean variables (so they take on the values \( \text{TRUE}, \text{FALSE} \)). We will use \( \neg \) to denote the negation of a variable \( x \); note that \( \neg x = x \). A literal is either a variable or its negation. A boolean formula is said to be satisfiable if there is an assignment of truth values to the variables such that the formula evaluates to true (e.g., the formula \((x_1 \vee x_2) \land \neg x_2 \) is satisfiable as can be seen from the assignment \( x_1 = \text{TRUE}, x_2 = \text{FALSE} \)). Recall that a boolean formula is in \( \text{CNF} \) if it is of the form \( C_1 \land C_2 \land \cdots \land C_m \) where each \( C_i \) is a clause, i.e., a form of the form \( C_1 \lor C_2 \lor \cdots \lor C_m \), in which each \( C_i \) is a literal.

   We proceed to describe a well known method for deciding if a formula in \( \text{CNF} \) is satisfiable. In fact it is easier to describe the algorithm if we change notation a
little and regard formulae in CNF as sets of sets of literals: we view \( C_1 \cap C_2 \cap \cdots \cap C_m \) as \( \{ C_1, C_2, \ldots, C_m \} \) and each \( C_i \) as \( \{ y_{i1}, y_{i2}, \ldots, y_{in} \} \) (e.g., the formula \(( x_1 \lor x_2 ) \land \tau_1\) is written as \( \{ \{ x_1, x_2 \}, \{ \tau_1 \} \}\)).

We say that two clauses \( C_1, C_2 \) can be resolved if there is a literal \( x \) such that \( x \in C_1 \) and \( x \in C_2 \). In such a case the resolvent of the two clauses with respect to the literal \( x \) is the clause \(( C_1 - \{ x \} ) \cup ( C_2 - \{ \tau \} )\). Two clauses can have more than one resolvent, e.g., consider \( C_1 = \{ x_1, x_2 \} \) and \( C_2 = \{ \tau_1, \tau_2 \} \). Note that the resolvent of \( \{ x \} \) and \( \{ \tau \} \) is \( \{ \} \), the empty clause. Given a set of clauses \( S = \{ C_1, C_2, \ldots, C_m \} \) we put
\[
R(S) = S \cup \{ C | C \text{ is a resolvent of two clauses of } S \}.
\]

We extend this notation in the obvious way by setting \( R^2(S) = R(R(S)) \), \( R^3(S) = R(R^2(S)) \), etc. By definition \( R^0(S) = S \). The resolution algorithm is the following
- Form \( R^0(S), R(S), R^2(S), \ldots \) until \( R^i(S) = R^{i+1}(S) \) for some \( i \).
- If \( R^i(S) \) contains the empty clause then \( S \) is unsatisfiable else \( S \) is satisfiable.

We note that there are various sensible improvements that can be made to the algorithm that are useful in practice; however they do not change the worst case runtime. For more details see H. R. Lewis and Ch. H. Papadimitriou, *Elements of the Theory of Computation*, Prentice-Hall (1981) or E. Rich, *Artificial Intelligence*, McGraw-Hill (1983) (the first reference gives proofs of correctness; the second one places resolution in the wider context of mechanical theorem proving). The problem of satisfiability is central to many applications in Computer Science and Artificial Intelligence.

(a) Simulate the resolution algorithm to show that \( \{ \{ x_1, x_2 \}, \{ x_2 \} \} \) is satisfiable while \( \{ \{ x_1, \tau_1 \}, \{ \tau_1 \}, \{ x_2 \} \} \) is unsatisfiable. (In each case give the sets \( S, R(S), R^2(S), \ldots \).)

(b) Explain briefly why the algorithm always terminates.

(c) A set of clauses is said to be an instance of 2SAT if each clause has no more than two literals. Show that for such sets of clauses the algorithm runs in \( O(n^6) \) time (recall that \( n \) is the number of variables). In your analysis, assume that literals can be compared in constant time.

The key points to think about here are:
- When we resolve a pair of clauses (each containing no more than two literals) what sort of clause do we get?
- How many distinct clauses are there if we restrict attention to those that contain no more than 2 literals?

(d) If we allow clauses to have up to three literals the worst case runtime degenerates very badly. Explain why this is so.