Lecture 3: Analysis of Algorithms and Asymptotic Notations

Once we design an algorithm, the first thing is to make sure its correctness. We will justify correctness of algorithms throughout the class. Another fundamental task on algorithms is analysis of algorithm (in particular, running time analysis). One way of performing analysis of algorithms is keeping track of the exact number of times each statement will execute. This can be done as we will show in the analysis of the insertion sort algorithm. But as you can see, it takes quite bit of efforts for such a simple algorithm. This will motivate the application of using asymptotic analysis later. Now, we will analyze the running time of the insertion sort algorithm, which is well known (check the posted slides for more details if you are unfamiliar with this algorithm). Note: usually running time of an algorithm depends on many factors (e.g. the kind of input). In the insertion sort, for example, the running time depends on the number of comparisons made when the new item is inserted. In the best case, each time only a single comparison is made. In the worst case, it needs to be:

\[ T(n) = C_1 n^2 + C_2 n + C_3 \]

where \( C_1, C_2, C_3 \) are constants.

In algorithm analysis, these constants are not essential: constants depend on many factors (e.g. the type of hardware), which makes comparing different algorithm difficult. We thus use the order of growth \( n \) to be:

\[ n \]

ignore the constant, which is less significant than the input size (usually denoted as \( n \)). In insertion sort, we focus on the leading terms (and ignoring lower order term). We also ignore the constant, which is less significant than the \( n^2 \) term.

We now come to the first major concept: the big-O notation. Formally, for two functions \( f(n) \) and \( g(n) \), we say \( f(n) = O(g(n)) \) if \( f(n) \leq C g(n) \) for some constant \( C \) when \( n \) is large enough (i.e. \( n \geq n_0 \) for some fixed \( n_0 \)). For insertion sort, the run time \( T(n) = a_2 x^2 + a_1 x + a_0 \). We claim \( T(n) = O(n^2) \). This is because we can pick \( C = a_2 + a_1 + a_0 \). The main motivation is, the highest order term, \( n^2 \), dominates the run time when \( n \) is large. Thus, we want to focus on this term.

Asymptotic notations are for big pictures. Big-O is for upper bounds and \( \Omega \) is for lower bounds. You need to know how to prove say \( 2n^2 + 4n + 3 = O(n^2) \). To do this, you need to find a constant \( C \), s.t. \( 2n^2 + 4n + 3 \leq C n^2 \). A natural choice for \( C = 2 + 4 + 3 = 9 \) because \( 4n \leq 4n^2 \) and \( 3 \leq 3n^2 \). Now review the concepts of \( O, \omega, \Omega, \theta \) by reading the Chapter 3 of the textbook.

For the example of insertion sort, we previously showed its worst running time \( T(n) = C_1 n^2 + C_2 n + C_3 \). Thus, \( T(n) = O(n^2) \). Note this is only the worst case running time. When the list is already sorted, insertion sort only takes \( \Theta(n) \) time. One should note that insertion sort can not run faster than \( \Theta(n) \) since it needs to process each element in the list at least once. The worst-case running time is the most used scheme for algorithm analysis. We say an algorithm is efficient if its worst-case running time \( T(n) = O(n^d) \) for some constant \( d \). This is called a polynomial time algorithm. Also, we will use worst-case running time in measuring the efficiency of algorithms. It is not perfect but usually works well in practice.

Moreover, big-O notations are often used in expressions like: \( T(n) = 2n^2 + 5n + 1 = 2n^2 + O(n) \). This means \( T(n) = 2n^2 + f(n) \) for some \( f(n) = O(n) \). Now, suppose we have a loop, where the number of iterations of the loop is \( O(n) \), and each iteration takes time \( O(n) \). Then we know the total time is \( O(n \times n) = O(n^2) \).

Lecture 4: Asymptotic Notations and Common Running Time

As shown in class, we can prove that if \( f_1(n) = O(g_1(n)) \) and \( f_2(n) = O(g_2(n)) \), we have: \( f_1(n) \times f_2(n) = O(g_1(n) \times g_2(n)) \). Review the lecture notes if you forget about it.

Finally, four common sense rules are useful when dealing with asymptotic notations.

1. Omit constants, e.g. write \( 14n^2 = O(n^2) \).
2. \( n^a \) dominates (grows faster) than \( n^b \) when \( a > b \) (i.e. \( n^b = O(n^a) \)).
3. Any exponential dominates polynomial, e.g. \( n^5 = O(2^n) \).
4. Any polynomial dominates logarithm, e.g. \( \log^3 n = O(n) \).
Often you apply these common sense rules to simplify asymptotic notations. For example, \( T(n) = 2n^2 + 5n + 3 = O(2n^2) = O(n^2) \), by applying the second rule first and then the first rule. I can show that \( \log_a(n) = \Theta(\log_b(n)) \). That is, logarithms wrt different bases are essentially the same asymptotically. I also show how to prove \( n^k = O(e^n) \). This can be shown by the fact: \( e^n = 1 + n + n^2/2! + \ldots + n^k/k! + \ldots \). Then we know \( n^k \leq k!e^n \) (note \( k! \) is a constant for any constant \( k \)).

With all these, we now can simplify the running time analysis using asymptotic notations. This way, we don’t need to give the exact counting on how many times a statement runs. Instead, we can simply use asymptotic notations, which are usually much simpler to obtain. In the class, I use the insertion sort as the example showing how one can analyze the running time using asymptotic notations. Make sure you understand how this is done.

We now list the common running times you will see. This also helps you to get more familiar with asymptotic notations and simple algorithm running time.

1. \( O(1) \). The fastest running time I can think of for any practical algorithm is constant time algorithm with \( O(1) \) time. Example: pushing an element into a stack.

2. \( O(\log n) \). Example: binary search. Everyone should know this. If not, you should make sure you understand it: it is widely used.

3. \( O(n) \). Often called linear time. Example: find the maximum value from an array. Note you can not do much better than this: you need to look at each array element at least once to ensure you find the largest one. Another example is the problem of merging two sorted list: given sorted lists \( A_1 \) and \( A_2 \) with \( n_1 \) and \( n_2 \) elements each, generate \( A \) that is also sorted and combines the elements in \( A_1 \) and \( A_2 \). Here is the algorithm.

\[
\begin{align*}
1: & \quad p_1, p_2 \leftarrow 1 \\
2: & \quad \textbf{while } p_1 \leq n_1 \text{ and } p_2 \leq n_2 \text{ do} \\
3: & \quad \quad \textbf{if } A_1[p_1] < A_2[p_2] \text{ then} \\
4: & \quad \quad \quad \text{Put } A_1[p_1] \text{ to the end of } A \text{ and } p_1 \leftarrow p_1 + 1 \\
5: & \quad \quad \textbf{else} \\
6: & \quad \quad \quad \text{Put } A_2[p_2] \text{ to the end of } A \text{ and } p_2 \leftarrow p_2 + 1 \\
7: & \quad \textbf{end if} \\
8: & \quad \textbf{end while} \\
9: & \quad \text{if there is still element in } A_1 \text{ or } A_2 \text{ not yet added to } A, \text{ add them to } A \text{ in the order they appear in the original list.}
\end{align*}
\]

This algorithm runs in \( O(n_1 + n_2) \) time, because each element will be added to \( A \) exactly one time, and each insertion takes \( O(1) \) time. Note: although this is a simple algorithm, it can be useful (we will use it later in the merge sort).

4. \( O(\log n) \). One of the most common running time, e.g. in sorting.

5. \( O(n^2) \). Often occurs when examining all pairs of states. Example: brute-force algorithm for finding two closest points among \( n \) points on a 2D plane.

6. \( O(n^3) \). Example: matrix multiplication algorithm in the homework.


Note that we also discussed in class that there are \( O(n^k) \) ways of selecting \( k \) items from a set of \( n \) items. This is useful in enumerating all pairs of points for the closest point problem, and also useful for some other problems.