

Lecture 3: Asymptotic Notations

Asymptotic notations are for big pictures. Big-O is for upper bounds and Ω 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. \( 2n^2 + 4n + 3 \leq Cn^2 \). A natural choice for \( C = 2 + 4 + 3 = 9 \) because \( 4n \leq 4n^2 \) and \( 3 \leq 3n^2 \). Now review the concepts of \( O, o, \Omega, \omega, \Theta \) 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 \( O(n \times n) = O(n^2) \). 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 \( 14n^2 \) as \( O(n^2) \).
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.

Lecture 4: Common Running Time and the Merge Sort

We first wrap up asymptotic notations. First, if \( f(n) = O(g(n)) \) roughly means \( f(n) \) is less or equal than \( g(n) \) asymptotically. Similarly, \( \Omega \) roughly means \( \geq \), and \( \Theta \) roughly means \( = \). The little-\( o \) and little-\( \omega \) roughly means strict less than and greater than. All these are in the asymptotic sense. 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 \( k \)).

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{verbatim}
1: p_1, p_2 \leftarrow 1
2: \textbf{while } p_1 \leq n_1 \text{ and } p_2 \leq n_2 \textbf{ do}
   \textbf{end while}
\end{verbatim}

3: if $A_1[p_1] < A_2[p_2]$ then
4: 
5: 
6: else 
7: end if
8: end while
9: if there is still element in $A_1$ or $A_2$ not yet added to $A$, add them to $A$ in the order they appear in the original list.

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. $O(n \log n)$. 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.

Now, we start the divide and conquer. We study the merge sort algorithm in class. The key is to divide the array into two equal sized sub-arrays, and then recursively sort the two sub-arrays. Once we have two sub-arrays sorted, we can apply the previous Merge algorithm to create the entire sorted array. Refer to the textbook for more details. Here are some key points about divide and conquer. You should decide how to divide a problem into smaller sub-problems. Then, you will solve these smaller sub-problems recursively. Finally, you combine these subproblems to get the answer for the original problem. Now, if you are unfamiliar with divide and conquer, try to complete the example of merge sort I showed in class: how the problem is divided and results from subproblems are combined. I claim this algorithm runs in $O(n \log n)$ time for an array with $n$ elements. I will show it in the next lecture.