Lecture 4: Common Running Time and the Merge Sort

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(logn). 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.

1: $p_1, p_2 ← 1$
2: while $p_1 \leq n_1$ and $p_2 \leq n_2$ do
3: if $A_1[p_1] < A_2[p_2]$ then
4: Put $A_1[p_1]$ to the end of $A$ and $p_1 ← p_1 + 1$
5: else
6: Put $A_2[p_2]$ to the end of $A$ and $p_2 ← p_2 + 1$
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.

4. O(nlogn). 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.
Lecture 5: Divide and conquer

We now study how to analyze divide and conquer algorithms. For the merge sort, its running time can be written as \( T(n) = 2T(n/2) + \Theta(n) \), when \( n > 1 \). When \( n = 1 \), \( T(n) = \Theta(1) \). There are several methods to estimate \( T(n) \). The first is called direct substitution. This method will repetitively get rid of the sub-problems by substituting with even smaller sub-problems. Here, \( T(n) = 2T(n/2) + cn, T(n/2) = 2T(n/4) + cn/2, T(n/4) = 2T(n/8) + cn/4 \), and so on. Then, we get rid of \( T(n/2) \) term by substituting it with \( 2T(n/4) + cn/2 \), and so on. This will lead to \( T(n) = nT(1) + cn\lg n \), because we stop after \( \lg n \) substitutions and we get sub-problems of size 1. If you are unclear about this step, you should write down these recurrences and try it yourself.

A related approach is to construct the so-called recursion tree. The recursion tree is divided into levels. Nodes of a level are labeled with the work spent on this level (i.e. the divide and combine work). The conquer portion of the work is expressed its descendant nodes. For the merge sort, each node for problem size of \( n/2 \) has work \( cn/2 \). There are \( \Theta(\log n) \) levels in the tree, because each time we go down the tree by one level, the number of subproblems doubles. At the bottom, there are \( n \) subproblems (because in merge sort, the subproblems are disjoint).

A third approach of analyzing recurrences is to use the Master theorem. Refer to Section 4.3 for more details. Two things to remember: remember the small \( \epsilon \) in cases 1 and 3, and also the regularity condition for case 3.

We now switch our attention to how to design a divide and conquer algorithm. We start with a simple problem: given two (binary) integers \( x \) and \( y \), compute the product of \( x \) and \( y \). Here is a primary school algorithm for this problem.

1: \( z \leftarrow 0 \)
2: for \( i = 1 \) to \( n \) do
3: \( t \leftarrow x \) if \( y[i] = 1 \), and 0 otherwise.
4: \( \) Left-shift \( t \) by \( i-1 \) bit.
5: \( z \leftarrow z + t. \)
6: end for

Now we analyze this algorithm. We have \( n \) iterations of the loop. In each iteration, we are performing no more than \( 2n \) single bit summations for adding two integers. Note: since we left shift \( t \), when properly implemented, we will still only add no longer than \( n \) bits in the addition. Thus, the algorithm will use \( n \times 2n = 2n^2 = O(n^2) \) single bit additions.

We can apply divide and conquer to design a new algorithm. We divide \( x \) (and \( y \)) into half: the high order \( n/2 \) bits and the low order \( n/2 \) bits. That is, we write \( x = x_1 \cdot 2^{n/2} + x_0 \) and \( y = y_1 \cdot 2^{n/2} + y_0 \). Then, we recursively solve four sub-problems (each of half size as before) and then combine into the product of \( x \) and \( y \). Note combine step takes \( O(n) \) since it involves adding a few \( O(n) \) bits integers, which can be done by \( O(n) \) time. Unfortunately, the Master theorem states that the running time is \( \Theta(n^2) \). In HW2, you will find a way to improve the algorithm to make it faster.