CS 124	Lecture 3 Notes	Spring 2022

In order to discuss algorithms effectively, we need to start with a basic set of tools. Here, we explain these tools and provide a few examples. Rather than spend time honing our use of these tools, we will learn how to use them by applying them in our studies of actual algorithms.

O Notation

When measuring, for example, the number of steps an algorithm takes in the worst case, our result will generally be some function $T(n)$ of the input size, n. One might imagine that this function may have some complex form, such as $T(n) = 4n^2 - 3n\log n + n^{2/3} + \log^3 n - 4$. In very rare cases, one might wish to have such an exact form for the running time, but in general, we are more interested in the rate of growth of $T(n)$ rather than its exact form.

The O notation was developed with this in mind. With the O notation, only the fastest growing term is important, and constant factors may be ignored. More formally:

Definition 3.1 *We say for non-negative functions $f(n)$ and $g(n)$ that $f(n)$ is $O(g(n))$ if there exist positive constants c and N such that for all $n \geq N$,*

$$f(n) \leq cg(n).$$

Let us try some examples. We claim that $2n^3 + 4n^2$ is $O(n^3)$. It suffices to show that $2n^3 + 4n^2 \leq 6n^3$ for $n \geq 1$, by definition. But this is clearly true as $4n^3 \geq 4n^2$ for $n \geq 1$. (**Exercise:** show that $2n^3 + 4n^2$ is $O(n^4)$.)

We claim $10\log_2 n$ is $O(\ln n)$. This follows from the fact that $10\log_2 n \leq (10\log_2 \mathrm{e})\ln n$.

If $T(n)$ is as above, then $T(n)$ is $O(n^2)$. This is a bit harder to prove, because of all the extraneous terms. It is, however, easy to see; $4n^2$ is clearly the fastest growing term, and we can remove the constant with O notation. Note, though, that $T(n)$ is $O(n^3)$ as well! The O notation is not tight, but more like a $\leq$ comparison.

Similarly, there is notation for $\geq$ and $=$ comparisons.

Definition 3.2 *We say for non-negative functions $f(n)$ and $g(n)$ that $f(n)$ is is $\Omega(g(n))$ if there exist positive constants c and N such that for all $n \geq N$,*

$$f(n) \geq cg(n).$$

We say that $f(n)$ is $\Theta(g(n))$ if both $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

The O notation has several useful properties that are easy to prove.

Lemma 3.3 *If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$ then $f_1(n) + f_2(n)$ is $O(g_1(n) + g_2(n))$.*

Proof: There exist positive constants c_1, c_2, N_1, and N_2 such that $f_1(n) \leq c_1 g_1(n)$ for $n \geq N_1$ and $f_2(n) \leq c_2 g_2(n)$ for $n \geq N_2$. Hence $f_1(n) + f_2(n) \leq \max\{c_1, c_2\}(g_1(n) + g_2(n))$ for $n \geq \max\{N_1, N_2\}$. ■

Exercise: Prove similar lemmata for $f_1(n) f_2(n)$. Prove the lemmata when O is replaced by Ω or Θ.

Finally, there is a bit for notation corresponding to $<<$, when one function is (in some sense) much less than another.

Definition 3.4 *We say for non-negative functions $f(n)$ and $g(n)$ that $f(n)$ is is $o(g(n))$ if*

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

Also, $f(n)$ is $\omega(g(n))$ if $g(n)$ is $o(f(n))$.

We emphasize that the O notation is a tool to help us analyze algorithms. It does not always accurately tell us how fast an algorithm will run in practice. For example, constant factors make a huge difference in practice (imagine increasing your bank account by a factor of 10), and they are ignored in the O notation. Like any other tool, the O notation is only useful if used properly and wisely. Use it as a guide, not as the last word, to judging an algorithm.

Recurrence Relations

A recurrence relation defines a function using an expression that includes the function itself. For example, the Fibonacci numbers are defined by:

$$F(n) = F(n-1) + F(n-2), \; F(1) = F(2) = 1.$$

This function is well-defined, since we can compute a unique value of $F(n)$ for every positive integer n.

Note that recurrence relations are similar in spirit to the idea of induction. The relations defines a function value $F(n)$ in terms of the function values at smaller arguments (in this case, $n-1$ and $n-2$), effectively reducing the problem of computing $F(n)$ to that of computing F at smaller values. Base cases (the values of $F(1)$ and $F(2)$) need to be provided.

Finding exact solutions for recurrence relations is not an extremely difficult process; however, we will not focus on solution methods for them here. Often a natural thing to do is to try to guess a solution, and then prove it by induction. Alternatively, one can use a symbolic computation program (such as Maple or Mathematica); these programs can often generate solutions.

We will occasionally use recurrence relations to describe the running times of algorithms. For our purposes, we often do not need to have an exact solution for the running time, but merely an idea of its asymptotic rate of growth. For example, the relation

$$T(n) = 2T(n/2) + 2n,\ T(1) = 1$$

has the exact solution (for n a power of 2) of $T(n) = 2n\log_2 n + n$. (**Exercise:** Prove this by induction.) But for our purposes, it is generally enough to know that the solution is $\Theta(n\log n)$.

The following theorem is extremely useful for such recurrence relations:

Theorem 3.5 *The solution to the recurrence relation $T(n) = aT(n/b) + cn^k$, where $a \geq 1$ and $b \geq 2$ are integers and c and k are positive constants satisfies:*

$$T(n) \text{ is } \begin{cases} O\left(n^{\log_b a}\right) & \text{if } a > b^k \\ O\left(n^k \log n\right) & \text{if } a = b^k \\ O\left(n^k\right) & \text{if } a < b^k. \end{cases}$$

Data Structures

We shall regard integers, real numbers, and bits, as well as more complicated objects such as lists and sets, as primitive data structures. Recall that a list is just an ordered sequence of arbitrary elements.

$$\text{List } q := [x_0, x_1, \ldots, x_{n-1}].$$

x_0 is called the head of the list.

x_{n-1} is called the tail of the list.

$n = |q|$ is the size of the list.

We denote by $\circ$ the concatenation operation. Thus $q \circ r$ is the list that results from concatenating the list q with the list r.

The operations on lists that are especially important for our purposes are:

head(q)	return(x_0)
push(q,x)	$q := [x] \circ q$
pop(q)	$q := [x_1,\ldots,x_{n-1}]$, return($x_0$)
inject(q,x)	$q := q \circ [x]$
eject(q)	$q := [x_0,x_1,\ldots,x_{n-2}]$, return($x_{n-1}$)
size(q)	return(n)

The head, pop, and eject operations are not defined for empty lists. Appropriate return values (either an error, or an empty symbol) can be designed depending on the implementation.

A *stack* is a list that supports operations head, push, pop.

A *queue* is a list that supports operations head, inject and pop.

A *deque* supports all these operations.

Note that we can implement lists either by arrays or using pointers as the usual linked lists. Arrays are often faster in practice, but they are often more complicated to program (especially if there is no implicit limit on the number of items). In either case, each of the above operations can be implemented in a constant number of steps.

Application: Insertion Sort

For the rest of the lecture, we will review sorting algorithms. The input is a list of n numbers, and the output is a list of the given numbers sorted in increasing order.

Algorithm 1 InsertionSort

```
Input: A
for $i = 0$ to $n-2$ do
  $j = i-1$
  while $j \geq 0$ and $A[j+1] < A[j]$ do
    swap $A[j]$ and $A[j+1]$
    j = j - 1
  end while
end for
```

To prove the correctness of the algorithm, we show by induction on i the claim that at the start of each iteration of the loop, the prefix of the list in places 0 through i are sorted.

As a base case, when $i = 0$, there is only one element, and every one-element list is sorted.

For general i, the (inner) while loop keeps the value of $A[j+1]$ unchanged, because it decreases j by 1 whenever it swaps $A[j+1]$ with $A[j]$. Also, the while loop keeps the list of the first i elements other than $A[j+1]$ unchanged. Also, at the start of the while loop, $A[j+1]$ is greater than any of the first i elements that come after it in the list, because there are no such elements, and this invariant is maintained by the swaps. Finally, the while loop halts only when $A[j]$ is not greater than any elements before it, that is, $A[j+1]$ is in sorted order in the first i elements. This completes the induction.

When $i = n-1$, the statement we've proven by induction is that the whole list is in sorted order.

How many additions, comparisons, variable assignments, and swaps does InsertionSort take? This depends on how many times the inner loop runs, which may be as few as n times (if the input list is in sorted order) or as many as $\binom{n}{2}$ times (if the input list is sorted in reverse). If we let that number of times be s, then there are

1. between $2s+1$ and $2s+n$ additions, depending on whether we recompute $n-2$ each time or cache it,
2. $n+2s$ comparisons
3. $n+s$ assignments to variables
4. $n+2s$ additions
5. s swaps.

Our convention in analyzing the run time of algorithms makes this easier:

1. We consider only the *worst case* running time of an algorithm on inputs of length n. In InsertionSort, we assume $s = \binom{n}{2}$.
2. We measure the running time only up to constant factors. Comparisons, additions, swaps, and variable assignments may take different amounts of time, and may even take different relative amounts of time on different machines or in different languages, but each of them takes a constant amount of time.

So, we say that the running time of InsertionSort is just $O(n^2)$.

Application: Mergesort

For the rest of the lecture, we will review the procedure mergesort. The input is a list of n numbers, and the output is a list of the given numbers sorted in increasing order. The main data structure used by the algorithm will be a queue. We will assume that each queue operation takes 1 step, and that each comparison (is $x > y$?) takes 1 step. We will show that mergesort takes $O(n \log n)$ steps to sort a sequence of n numbers.

The procedure mergesort relies on a function merge which takes as input two *sorted* (in increasing order) lists of numbers and outputs a single sorted list containing all the given numbers (with repetition).

```
function merge $(s,t)$
    list $s,t$
    if $s = [\,]$ then return $t$
       else if $t = [\,]$ then return $s$
       else if $s(0) \leq t(0)$ then $u$:= pop$(s)$
                                   else $u$:= pop$(t)$
       return inject($u$, merge$(s,t)$)
end merge
```

```
function mergesort $(s)$
    list $s$, $q$
    $q = [\,]$
    for $x \in s$
        inject($q$, $[x]$)
    rof
    while size$(q) \geq 2$
        $u$ := pop$(q)$
        $v$ := pop$(q)$
        inject($q$, merge$(u,v)$)
    end
    if $q = [\,]$ return $[\,]$
        else return $q(0)$
end mergesort
```

The correctness of the function merge follows from the following fact: the smallest number in the input is either $s(1)$ or $t(1)$, and must be the first number in the output list. The rest of the output list is just the list obtained by merging s and t *after* deleting that smallest number.

The number of steps for each invocation of function merge is $O(1)$ steps. Since each recursive invocation of merge removes an element from either s or t, it follows that function merge halts in $O(|s|+|t|)$ steps.

Question: Can you design an iterative (rather than recursive) version of merge? How much time does is take? Which version would be faster in practice– the recursive or the iterative?

The iterative algorithm mergesort uses q as a queue of lists. (Note that it is perfectly acceptable to have lists of

$$Q : [\,[7,9],[1,4],[6,16],[2,10] * [3,11,12,14],[5,8,13,15]\,]$$
$$Q : [\,[6,16],[2,10] * [3,11,12,14],[5,8,13,15],[1,4,7,9]\,]$$

Figure 3.1: One step of the mergesort algorithm.

lists!) It repeatedly merges together the two lists at the front of the queue, and puts the resulting list at the tail of the queue.

The correctness of the algorithm follows easily from the fact that we start with sorted lists (of length 1 each), and merge them in pairs to get longer and longer sorted lists, until only one list remains. To analyze the running time of this algorithm, let us place a special marker $*$ initially at the end of the q. Whenever the marker $*$ reaches the front of q, and is either the first or the second element of q, we move it back to the end of q. Thus the presence of the marker $*$ makes no difference to the actual execution of the algorithm. Its only purpose is to partition the execution of the algorithm into phases: where a phase is the time between two successive visits of the marker $*$ to the end of the q. Then we claim that the total time per phase is $O(n)$. This is because each phase just consists of pairwise merges of disjoint lists in the queue. Each such merge takes time proportional to the sum of the lengths of the lists, and the sum of the lengths of all the lists in q is n. On the other hand, the number of lists is halved in each phase, and therefore the number of phases is at most $\log n$. Therefore the total running time of mergesort is $O(n \log n)$.

An alternative analysis of mergesort depends on a recursive, rather than iterative, description. Suppose we have an operation that takes a list and splits it into two equal-size parts. (We will assume our list size is a power of 2, so that all sublists we ever obtain have even size or are of length 1.) Then a recursive version of mergesort would do the following:

```
function mergesort ($s$)
    list $s$, $s_1$, $s_2$
    if size($s$) $= 1$ then return($s$)
    split($s, s_1, s_2$)
    $s_1 =$ mergesort($s_1$)
    $s_2 =$ mergesort($s_2$)
    return(merge($s_1, s_2$))
end mergesort
```

Here split splits the list s into two parts of equal length s_1 and s_2. The correctness follows easily from induction.

Let $T(n)$ be the number of comparisons mergesort performs on lists of length n. Then $T(n)$ satisfies the recurrence relation $T(n) \leq 2T(n/2) + n - 1$. This follows from the fact that to sort lists of length n we sort two sublists of length $n/2$ and then merge them using (at most) $n-1$ comparisons. Using our general theorem on solutions of recurrence relations, we find that $T(n) = O(n \log n)$.

Question: The iterative version of mergesort uses a queue. Implicitly, the recursive version is using a stack. Explain the implicit stack in the recursive version of mergesort.

Question: Solve the recurrence relation $T(n) = 2T(n/2) + n - 1$ exactly to obtain an upper bound on the number of comparisons performed by the recursive mergesort variation.