CS 125	**Course Notes 1**	Fall 2016

Welcome to CS 125, a course on *algorithms* and *computational complexity*. First, what do these terms means?

An **algorithm** is a recipe or a well-defined procedure for performing a calculation, or in general, for transforming some input into a desired output.

In this course we will ask a number of basic questions about algorithms:

- Does the algorithm halt?
- Is it correct?

  That is, does the algorithm's output always satisfy the input to output specification that we desire?
- Is it efficient?

  Efficiency could be measured in more than one way. For example, what is the running time of the algorithm? What is its memory consumption?

Meanwhile, **computational complexity theory** focuses on classification of problems according to the computational resources they require (time, memory, randomness, parallelism, etc.) in various computational models. Computational complexity theory asks questions such as

- Is the class of problems that can be solved time-efficiently with a deterministic algorithm exactly the same as the class that can be solved time-efficiently with a randomized algorithm?
- For a given class of problems, is there a "complete" problem for the class such that solving that one problem efficiently implies solving all problems in the class efficiently?
- Can every problem with a time-efficient algorithmic solution also be solved with extremely little additional memory (beyond the memory required to store the problem input)?

1.1 Algorithms: arithmetic

Some algorithms very familiar to us all are those those for adding and multiplying integers. We all know the grade school algorithm for addition from kindergarten: write the two numbers on top of each other, then add digits right

$$
\begin{array}{rrrrrr}
 & & & 1 & 7 & 8 \\
 & & \times & 2 & 1 & 3 \\
\hline
 & & & 5 & 3 & 4 \\
 & & 1 & 7 & 8 & \\
+ & 3 & 5 & 6 & & \\
\hline
 & 3 & 7 & 9 & 1 & 4
\end{array}
$$

Figure 1.1: Grade school multiplication.

to left while keeping track of carries. If the two numbers being added each have n digits, the total number of steps (for some reasonable definition of a "step") is n. Thus, addition of two n-digit numbers can be performed in n steps. Can we take fewer steps? Well, no, because the answer itself can be n digits long and thus simply writing it down, no matter what method you used to obtain it, would take n steps.

How about integer multiplication? Here we will describe several different algorithms to multiply two n-digit integers x and y.

Algorithm 1. We can multiply $x \cdot y$ using repeated addition. That is, add x to itself y times. Note all intermediate sums throughout this algorithm are between x and $x \cdot y$, and thus these intermediate sums can be represented between n and $2n$ digits. Thus one step of adding x to our running sum takes at most $2n$ steps. Thus the total number of steps is at most $2n \cdot y$. If we want an answer purely in terms of n, then an n-digit number y cannot be bigger than $10^n - 1$ (having n 9's). Thus $2n \cdot y \leq 2n \cdot (10^n - 1) \approx 2n \cdot 10^n$. In fact usually in this course we will ignore leading constant factors and lower order terms (more on that in section!), so this bound is at most roughly $n \cdot 10^n$, or as we will frequently say from now on, $O(n \cdot 10^n)$. This bound is quite terrible; it means multiplying 10-digit numbers already takes about 100 billion steps!

Algorithm 2. A second algorithm for integer multiplication is the one we learned in grade school, shown in Figure 1.1. That is, we run through the digits of y, right to left, and multiply them by x. We then sum up the results after multiplying them by the appropriate powers of 10 to get the final result. What's the running time? Multiplying one digit of y by the n-digit number x takes n steps. We have to do this for each of the n digits of y, thus totaling n^2 steps. Then we have to add up these n results at the end, taking n^2 steps. Thus the total number of steps is $O(n^2)$. As depicted in Figure 1.1 we also use $O(n^2)$ memory to store the intermediate results before adding them. Note that we can reduce the memory to $O(n)$ without affecting the running time by adding in the intermediate results to a running

sum as soon as we calculate them.

Now, it is important at this point to pause and observe the difference between two items: (1) the problem we are trying to solve, and (2) the algorithm we are using to solve a problem. The problem we are trying to solve is integer multiplication, and as we see above, there are multiple algorithms which solve this problem. Prior to taking this class, it may have been tempting to equate integer multiplication, the *problem*, with the grade school multiplication procedure, the *algorithm*. However, they are not the same! And in fact, as algorithmists, it is our duty to understand whether the grade school multiplication algorithm is in fact the most efficient algorithm to solve this problem. In fact, as we shall soon see, it isn't!

Algorithm 3. Let's assume that n is a power of 2. If this is not the case, we can pad each of x, y on the left with enough 0's so that it does become the case (and doing so would not increase n by more than a factor of 2, and recall we will ignore constant factors in stating our final running times anyway). Now, imagine splitting each number x and y into two parts: $x = 10^{n/2}a + b, y = 10^{n/2}c + d$. Then

$$xy = 10^n ac + 10^{n/2}(ad + bc) + bd.$$

The additions and the multiplications by powers of 10 (which are just shifts!) can all be done in linear time. We have therefore reduced our multiplication problem into four smaller multiplications problems. Thus if we let $T(n)$ be a function which tells us the running time to multiply two n-digit numbers. Then $T(n)$ satisfies the equation

$$T(n) = 4T(n/2) + Cn$$

for some constant C. Also when $n = 1$, $T(1) = 1$ (we imagine we have hardcoded the 10×10 multiplication table for all digits in our program). This equation where we express $T(n)$ as a function of T evaluated on small numbers is is what's called a *recurrence relation*; we'll see more about this next lecture. Thus what we are saying is that the time to multiply two n-digit numbers equals that to multiply $n/2$ digit numbers, 4 times (in each of our recursive subproblems), plus an additional Cn time to combine these problems into the final solution (using additions and shifts). If we draw a recursion tree, we find that at the root we have to do all the work of the 4 subtrees, plus Cn work at the root itself to combine results from these recursive calls. At the next level of the tree, we have 4 nodes. Each one, when receivng the results from its subtrees, does $Cn/2$ work to combine those results. Since there are 4 nodes at this level, the total work at this level is $4 \cdot Cn/2 = 2Cn$. In general, the total work in the kth level (with the root being level 0) is $2^k \cdot Cn$. The height of the tree is $h = \log_2 n$, and thus the total work is

$$Cn + 2Cn + 2^2Cn + \ldots + 2^hCn = Cn(2^{h+1} - 1) = Cn(2n - 1) = O(n^2).$$

Thus, unfortunately we've done nothing more than give yet another $O(n^2)$ algorithm more complicated than Algorithm 2.

Algorithm 4. Algorithm 3, although it didn't give us an improvement, can be modified to give an improvement. The following algorithm is called *Karatsuba's algorithm* and was discovered by the Russian mathematician Anatolii Alexeevitch Karatsuba in 1960. The basic idea is the same as Algorithm 3, but with one clever trick. The key thing to notice here is that four multiplications is too many. Can we somehow reduce it to three? It may not look like it is possible, but it is using a simple trick. The trick is that *we do not need to compute ad and bc separately; we only need their sum* $ad+bc$. Now note that

$$(a+b)(c+d) = (ad+bc)+(ac+bd).$$

So if we calculate ac, bd, and $(a+b)(c+d)$, we can compute $ad+bc$ by the subtracting the first two terms from the third! Of course, we have to do a bit more addition, but since the bottleneck to speeding up this multiplication algorithm is the number of smaller multiplications required, that does not matter. The recurrence for $T(n)$ is now

$$T(n) = 3T(n/2)+Cn.$$

Then, drawing the recursion tree again, there are only 3^k nodes at level k instead of 4^k, and each one requires doing $Cn/2^k$ work. Thus the total work of the algorithm is, again for $h = \log_2 n$,

$$Cn+\frac{3}{2}Cn+\left(\frac{3}{2}\right)^2 Cn+\ldots+\left(\frac{3}{2}\right)^h Cn = Cn\cdot\frac{\left(\frac{3}{2}\right)^{h+1}-1}{\frac{3}{2}-1},$$

where we used the general fact that $1+p+p^2+\ldots+p^m = (p^{m+1}-1)/(p-1)$ for $p \neq 1$. Now, using the general fact that we can change bases of logarithms, i.e. $\log_a m = \log_b m/\log_b a$, we can see that $(3/2)^{\log_2 n} = (3/2)^{\log_{3/2} n/\log_{3/2} 2} = n^{1/\log_{3/2} 2}$. Then changing bases again, $\log_{3/2} 2 = \log_2 2/\log_2(3/2) = 1/(\log_2 3-1)$. Putting everything together, our final running time is then $O(n^{\log_2 3})$, which is $O(n^{1.585})$, much better than the grade school algorithm! Now of course you can ask: is this the end? Is $O(n^{\log_2 3})$ the most efficient number of steps for multiplication? In fact, it is not. The Schönhage-Strassen algorithm, discovered in 1971, takes a much smaller $O(n\log_2 n\log_2\log_2 n)$ steps. The best known algorithm to date, discovered in 2007 by Martin Fürer, takes $O(n\log_2 n 2^{C\log^* n})$ for some constant $C \leq 8$. Here, $\log^* n$ is the number of base-2 logarithms one must take of n to get down to a result which is at most 1. The point is, it is a *very* slow growing function. If one took $\log^*$ of the number of particles in the universe, the result would be at most 5!

Also, Karatsuba's algorithm is not just a source of fun for theorists, but actually is used in practice! For example, it is used for the implementation of integer multiplication in Python. If you want to check it out for yourself, here's what I did on my GNU/Linux machine:

```
wget https://www.python.org/ftp/python/3.5.2/Python-3.5.2.tar.xz
tar xvfJ Python-3.5.2.tar.xz
emacs Python-3.5.2/Objects/longobject.c
```

Now look through that file for mentions of Karatsuba! Most of the action happens in the function `k_mul`.

Now that we've seen just how cool algorithms can get. With the invention of computers in this century, the field of algorithms has seen explosive growth. There are a number of major successes in this field:

- Parsing algorithms - these form the basis of the field of programming languages
- Fast Fourier transform - the field of digital signal processing relies heavily on this algorithm.
- Linear programming - this algorithm is extensively used in resource scheduling.
- Sorting algorithms - until recently, sorting used up the bulk of computer cycles.
- String matching algorithms - these are extensively used in computational biology.
- Number theoretic algorithms - these algorithms make it possible to implement cryptosystems such as the RSA public key cryptosystem.
- Compression algorithms - these algorithms allow us to transmit data more efficiently over, for example, phone lines.
- Geometric algorithms - displaying images quickly on a screen often makes use of sophisticated algorithmic techniques.

In designing an algorithm, it is often easier and more productive to think of a computer in abstract terms. Of course, we must carefully choose at what level of abstraction to think. For example, we could think of computer operations in terms of a high level computer language such as C or Java, or in terms of an assembly language. We could dip further down, and think of the computer at the level AND and NOT gates.

For most algorithm design we undertake in this course, it is generally convenient to work at a fairly high level. We will usually abstract away even the details of the high level programming language, and write our algorithms in "pseudo-code", without worrying about implementation details. (Unless, of course, we are dealing with a programming assignment!) Sometimes we have to be careful that we do not abstract away essential features of the problem.

Also, of course when we want to understand the efficiency with which we can solve a particular computational problem, the model of computation we use matters in this understanding! To illustrate, let us turn to the problem of sorting.

1.2 Sorting

In the sorting problem, we are given a list (or array) of n objects, and we would like to sort them (to be concrete, let's say we want to sort them in increasing order). Objects can be, say, numbers, but they can also be strings (lexicographic comparison) or baseball teams if using the (unrealistic) assumption that team skill levels are totally ordered and are perfect predictors of who wins a match.

1.2.1 Comparison model

We will first study the so-called *comparison model*. In this model, given two objects we can compare them to see which is smaller (or whether they are equal), but we cannot perform other computation on an object such as multiplying it with a number, or doing a bitwise XOR (what does doing a bitwise XOR on a baseball team even mean!).

For concreteness, henceforth we will assume that we are sorting numbers.

1.2.1.1 BubbleSort

As I imagine many of you know, there are many algorithms that can sort n numbers using a quadratic number of operations, i.e. $O(n^2)$. Here's one such algorithm.

The Bubblesort algorithm is very simple. (You can read more about it on Wikipedia, including various pseudocode.) One way of viewing it is to compare the first and second elements, swapping the order if necessary. Then compare the second and third, then the third and the fourth, and so on, until you get to the end. Now repeat this process n times.

Why does this work? It's easy to check that the first complete pass moves the largest element to the end, and inductively, after k passes, the k largest elements will be at the end. (In fact, you can optimize based on this; you know the k elements are at the end after k passes, so you only have to compare the first $n-k$ elements on the $(k+1)$st pass.) There are at most n^2 comparisons, and therefore at most $O(n^2)$ operations, including swaps, assuming every

operation can be done in unit time.

Example:

7	5	9	3	1
7	**5**	9	3	1
5	**7**	**9**	3	1
5	7	**9**	**3**	1
5	7	3	**9**	**1**
5	7	3	1	9
5	**7**	3	1	9
5	**7**	**3**	1	9
5	3	**7**	**1**	9
5	3	1	7	9
5	**3**	1	7	9
3	**5**	**1**	7	9
3	1	5	7	9
3	**1**	5	7	9
1	3	5	7	9

1.2.1.2 MergeSort

So we can sort in $O(n^2)$ operations. Can we do better? Yes. There are a number of algorithms that can sort in just $O(n\log n)$ operations. (Note that asymptotically $n\log_2 n$ is much much smaller than n^2; the ratio $n\log_2 n/n^2$ goes to 0.) One of simplest is mergesort:

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

Break the list into two equal-sized (or as equal as possible) sublists, sort each sublist recursively, and merge the two sorted lists (by going through them together in order). A merge function is given below (using basic list operators push and pop).

```
function merge (s,t)
    list s,t
    if s = [ ] then return t
      else if t = [ ] then return s
      else if s(1) ≤ t(1) then u:= pop(s)
                          else u:= pop(t)
      return push(u, merge(s,t))
end merge
```

Let $T(n)$ be the number of comparisons mergesort performs on lists of length n. Then $T(n)$ satisfies the recurrence relation $T(n) \leq 2T(n/2) + n - 1$. (If n is odd, then really $T(n) \leq T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n - 1$; not a significant difference.) 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 the general theory on solutions of recurrence relations, we find that $T(n) = O(n \log n)$.

Exercise: Prove directly that the recurrence $T(n) \leq 2T(n/2) + n$ with $T(2) = 1$ yields $T(n) \leq n \log_2 n$ for n a power of 2 by expanding the recurrence directly.

1.2.1.3 Lower Bounds

So far we've been considering what we *can do* – how fast we can sort. Now we want to think about something more challenging – are there limits to how fast we can sort? This is more challenging because when we ask how fast, we just need to consider and analyze one algorithm. When we ask about limits, we seek a lower bound for *all possible algorithms*, or at least all possible algorithms in a certain class of algorithms. Trying to prove a statement about all possible algorithms can be, as you might imagine, difficult.

Let us try to frame the problem in order to obtain a lower bound. The BubbleSort algorithm and the MergeSort algorithm are both based on comparing elements to put them in the right place. How many comparisons are needed to sort n elements correctly every time?

Formally, let us consider algorithms that start with an array $A = (a_i)$ and *only* do two kinds of operations; they compare elements ("is $a_i > a_j$"), and they move elements.

Our goal will be to prove the following theorem:

Theorem 1.1 *Any deterministic comparison-based sorting algorithm must perform $\Omega(n \log n)$ comparisons to sort n elements in the worst case. More specifically, for any such algorithm, there exists some input with n elements that requires at least $\log_2(n!)$ comparisons.*

Proof: We'll describe two ways of thinking about this proof (they're equivalent, but one might be easier to think about than another for you). For convenience, let us assume our input is always just a permutation of the numbers from 1 to n. (Notice that since our algorithm just does comparisons, the "size" of the numbers in the input doesn't really matter; as long as we have distinct numbers, this argument holds.)

Let us define the "state" of our algorithm as a list of all the comparisons that have been done along with their outcomes. We do not have to worry about the current order of the elements for our "state", since we are only considering comparisons in our lower bound; the algorithm could always move any of the elements at any time "for free" based on the state given by the comparisons, so we can without loss of generality save the moves to the end.

The algorithm cannot terminate with the same state for two different permutations, because if it did, it would necessarily give the wrong answer on one of them. But if the algorithm does at most k comparisons, there are at most 2^k different possible states the deterministic algorithm could end up in, because there are 2 outcomes for each comparison. And there are $n!$ different permutations. Hence if k is the largest number of comparisons done for any possible input we must have

$$2^k \geq n!.$$

This gives the theorem.

Here's an alternative view, which might seem easier if you like the game 20 Questions. At the beginning, there are $n!$ possible input permutations. Each time the algorithm does a comparison, it determines information that makes some of those inputs impossible. Let X_j be the set of initial permutations consistent with the information that the algorithm has determined after j comparisons. For the algorithm to work correctly, the algorithm must ask enough questions so that $|X_j| = 1$. Initially, $|X_0| = n!$.

Each comparison splits X_j into two subsets, call them Y_j and and Z_j, depending on the outcome of the comparison. Now suppose an adversary just tries to make the algorithm take as long as possible, by returning the answer to the comparison that corresponds to the biggest set of Y_j and Z_j. In that case,

$$|X_{j+1}| = \max |Y_j|, |Z_j| \geq |X_j|/2.$$

It follows that there's some permutation the adversary can use to ensure that at least $k = \lceil \log_2(n!) \rceil$ comparisons are used by the algorithm before it can get to a point where $|X_k| = 1$. ■

Usually, this result is interpreted as meaning that we have tight bounds for sorting. There are algorithms that take $O(n \log n)$ comparisons, and that many comparisons are required. But this is a limited point of view. The result has shown that algorithms based on comparisons in the worst case can require $\Omega(n \log n)$ comparisons. So if we want to do better, we will have to *change the problem* in some way. One way to change the problem is to assume something about the input so that it is not worst case, and we will give some examples of that. But another way to change the problem is to consider operations other than comparisons. That is, we change and enrich our model of computation.

1.2.2 Counting Sort

Suppose that we restrict our input so that we are sorting n numbers that are integer values in the range from 1 to k. Then we can sort the numbers in $O(n+k)$ space and $O(n+k)$ time. Notice that if k is "smaller than" $n \log n$ – in notation we'll discuss, if k is $o(n \log n)$, then this is a sorting algorithm that breaks the $\Omega(n \log n)$ barrier. Naturally, this means counting sort is not a comparison based sorting algorithm.

Instead, we build an array *count* keeping track of what numbers are in our input. (More generally, if we can have multiple copies of the same number, we build a histogram of the frequencies of the numbers in our input. See

the Wikipedia page for more details – here we'll assume the elements are distinct.) If our input is $a_1, a_2, \ldots, a_n$, we keep track of counts; for each a_i, we increase its count:

$$\text{count}[a_i] += 1.$$

Now we walk through the count array, updating it via what is called a prefix sum operation, so that the count[j] contains the number of inputs that are less than or equal to j.

Finally, we walk back through the inputs; for each a_i, it's sorted position is now count[a_i]. The two passes through the input take $O(n)$ operations; the pass through the count array to compute the prefix sum takes $O(k)$ operations.

Here we are making assumptions about the input, and expanding our model of computation. Instead of just comparisons, we're accessing a non-trivially sized array. In particular, we are assuming that we can maninpulate values with $\lceil \log_2 k \rceil$ bits in one unit of time, and use these values to do things like access arrays. If k for example is n, this means we are assuming we can work with number of size $\log_2 n$ bits.

1.3 Interlude on Models of Computation, Complexity, and Computability

As the discussion above suggests, in order to precisely specify an algorithm, we need to precisely specify a *model of computation* — what kind of operations are allowed, and on what kind of data? Having a precise model is particularly important if we want to be able to understand the limitations of algorithms — what is impossible for algorithms to achieve. An example is the above proof that any comparison-based algorithm for sorting requires $\Omega(n \log n)$ comparisons. But the fact that we can beat this lower bound by working in a more general, but still very reasonable, model of computation, suggests that the choice of model might be very important.

Can there be a robust theory of computing, that is not sensitive to arbitrary details of our model of computation? Remarkably, the answer turns out to be yes. In a few weeks, we will see that there is a simple model of computation, known as the Turing machine, that can compute the same functions as any reasonable model of computation. It was formulated in the 1930's by Alan Turing, motivated by questions in the foundations of mathematics (related to Godel's Incompleteness Theorem). It inspired the development of actual physical computers, and its relevance has not diminished despite all the changes in computing technology. Using the Turing machine model, we will prove that there are some well-defined computational problems that cannot be solved by any algorithm whatsoever. For example:

- Given a computer program (in your favorite language), can it be made to go into an infinite loop?
- Given a multivariate polynomial equation $p(x_1, \ldots, x_n) = 0$, does there exist an integer solution?

Now, as we all know when we purchase a new computer, the choice of the computational model (= computing hardware) does make a difference, at least to how *fast* we can compute. But, it turns out that it does not make a very big difference. In particular, while it may make the difference between $O(n \log n)$ and $O(n)$ time (as we saw above), it will not make a difference between polynomial and exponential time. This leads to a robust theory of *computational complexity*, which we will also study. Here many questions are open (notably, the famous P vs. NP question, widely considered one of the most important open problems in computer science and in mathematics) — there are many problems conjectured to require exponential time, but for which we still have no proof. But the beautiful theory of NP-completeness allows us to show that a vast collection of important problems all have the same complexity: if one has a polynomial-time algorithm, all do, and if one requires exponential time, all do.

1.4 Bucket Sort

For bucket sort, we again make assumptions about the input – this time, we assume the inputs are randomly selected according to some distribution. Because of this "randomness" in the input, we consider the "expected time" to sort instead of worst case time. Here we assume that the n input numbers are uniform over a collection of numbers in the range $(0, M]$, and show that the expected time to sort is $O(n)$. (More general results are possible.)

We put the numbers into n buckets of size M/n, with the numbers 1 to M/n going in the first bucket, the next M/n numbers going in the second bucket, and so on. We then sort by using BubbleSort on each bucket, and then concatenating the results from each bucket in order. If X_i is the number of elements that land in the ith bucket, the expected number of comparison operations to sort T is given by:

$$\begin{aligned} \mathbb{E}\,T &\leq \mathbb{E}\sum_{i=1}^{n} X_i^2 \\ &= \sum_{i=1}^{n} \mathbb{E}\,X_i^2 \\ &= n\,\mathbb{E}\,X_1^2. \end{aligned}$$

Now, let us define indicator random variables $Y_1, \ldots, Y_n$ where Y_j is 1 if item j landed in bucket 1, else $Y_j = 0$. Then $X_1 = \sum_{j=1}^{n} Y_j$, so

$$\begin{aligned} \mathbb{E}\,X_1^2 &= \mathbb{E}\left(\sum_{j=1}^{n} Y_j\right)^2 \\ &= \sum_{j=1}^{n} \mathbb{E}\,Y_j + \sum_{j \neq j'} \mathbb{E}\,Y_j Y_{j'} \\ &= n \cdot \frac{1}{n} + n(n-1) \cdot \frac{1}{n^2} < 2 \end{aligned}$$

1.5 Conclusion

Sorting remains an active area for research; the best sorting algorithm might still not be known, in part because the right model for machine operations is open to interpretation. One often hears of an $\Omega(n \log n)$ lower bound for sorting. That lower bound is correct in the most general setting for data where all one can do with two elements is compare them, but taking advantage of even simple aspects of the data, such as that they are numbers within some fixed range (e.g. counting sort), can lead to sorting algorithms that break the lower bound.