Lecture Notes: Math Preliminaries

Chris Brown

September 12, 1998

Generally, the analysis of algorithms does not call for terribly difficult mathematics. Further, there are many good books devoted entirely to algorithm analysis (classic works that every computer scientist should own are [1; 2] – that is where I am cribbing these notes from!). Your text has useful references as well. The UR CS department has the course CSC282, which is devoted to interesting algorithms and their analysis.

This material expands on the text (Weiss), Sections 1.1.1 through 1.2.4, which should of course be thoroughly understood. This extra stuff should help you understand what is in Weiss and may go a bit beyond as well. I should appreciate any constructive feedback on how useful or readable these papers are, as well as on typos, errors, etc.

1 Writing Mathematics

It would be a good idea to prepare all your homework using a text processing system. If you are majoring in CS, learning LaTeX would be a good investment. However, the medium is not as important as the message for homeworks and in-class exams as long as the work is legible.

Mathematics is written like English. Just look at any textbook to see what to do. There are English complete sentences, with the mathematics set off (often centered, often equations are numbered, etc.) and fully punctuated. Write like that. As you may know, departments are being asked to take writing and communication in general quite seriously in our curricula, and this course is included. Get yourself a good style manual. Read Strunk and White. Read Fowler (the original is better than "The New"). If you don't know who I'm talking about, now is a good time to further your education with a little research. Use spell checkers and use your dictionary. Spelling competence is on a the decline on a nationwide basis, but we can be an oasis of quality.

2 A Partial Analysis

To illustrate the need for certain sorts of mathematics, consider the following algorithm, which is to find the maximum `max_A` of an array `A` of `n` numbers and the array index of the maximum, `max_j`. If there are ties we want the maximum `max_j`. Here's some pseudo-code...

```
max_j = n-1; max_A = A[n-1];          // initialize current maxima
 for (k = n-2; k>=0; k--)             // walk through entire array
    if (A[k] > max_A)                 // check if new maximum
      {max_j = k; max_A = A[k];       // New Maximum  Found
```

Question: why would we want to go through the array backwards like this?

Our basic question at this point is "how long does this algorithm take for an array of size n?". For the purposes of this section only, let us assume that we measure time in terms of the number of statements executed, where statements are the numbered lines of the above code. (In fact this measure is, in this case, proportional to the usual time units of elementary operations, since there are a constant number of elementary operations per statement). We should like to answer such questions as the minimum running time of the code (usually not of interest in algorithm analysis since mostly we are pessimists and want to see how bad things can get), the maximum running time (the most common measure), and the expected (average) running time (clearly important, but usually almost impossible to calculate and heavily dependent on assumptions about the input.)

So, how often is statement 1 executed? Once. Statements 2 and 3 are done n-1 times, and statement 4 is done a variable number of times (call it T), depending on the content and arrangement of the numbers in the array. Of course, Rule 4 of Weiss Section 2.4.2. is the typical pessimistic worst-case running time assumption: Just assume Statement 4 is executed each time that the `if` statement is, or once each time through the loop. By this worst-case analysis the behavior of this algorithm is O(n). That means that the worst-case running time is proportional to the length of the array. This is simple but pretty boring. It turns out that trying to answer the question of average running time in this simple-seeming problem leads to the sorts of techniques presented below. Further, we of course might want to know both the mean and the variance of the running time, or even the full probability distribution of the running time. As it turns out all these quantities can be computed for some assumptions about the input. I am not going to analyze even this simple problem (if you want to see it done right, *cf.* [1], Section 1.2.10.), but I'll point out a few things and draw a graph of T for one case.

Clearly if A[n] is the maximum value, T is zero. Likewise in the worst case T can be $n-1$ (under what conditions is this?). But T can take on all values in between (make sure you understand how), and so we can wonder what an "average" T would be. Well, this is going to depend on how the values of the array A are chosen. If they are always chosen to be equal, that makes things simple, since then T will be uniformly zero. If each element is chosen independently and at random from a set of m integers, things get surprisingly hairy (*cf.* [1], exercises 7-10 in section 1.2.10.) I had thought this case might be easy since "everything would average out", but so much for my intuition.

An interesting case is when the A[k] are all distinct, and when each of the permutations of the n elements is equally likely. Thus you can check that when n is 3, there are 6 cases and T is 0 for two of them, 1 for three of them, and 2 for one of them. Let us denote by $p(n,k)$ the probability that T has the value k for an array of size n. We can write down a recursive way to calculate p. If $n = 1$, we are guaranteed that we need zero swaps, and never any more than zero, so we can say $p(1,k) = 1$ for $k = 0$ and $p(1,k) = 0$ for all larger k. Also $p(n,k) = 0$ if $k < 0$. Now assume that we know $p(n-1,k)$, how would we use that to calculate $p(n,k)$ for an array one larger in size? There will be a new biggest element, n, and the issue is how its placement affects the number of swaps. If you think about it, you'll see that the only issue is what happens at A[0]: if this element is n, we have to execute Statement 4. Otherwise, the maximum value n has been found before and we do not execute Statement 4. Thus $1/n$ of the time that $T = k$ we reach the value k by incrementing at the last step, and $(n-1)/n$ of the time we reach the value k before getting to the last step. The probabilities of these two paths are determined by $p(n-1,k-1)$ and $p(n-1,k)$, respectively:

$$p(n,k) = \frac{1}{n}p(n-1,k-1) + \frac{n-1}{n}p(n-1,k)$$

This equation is easy to implement if we want to see what the probability distribution function looks like: I wrote some code using a Pascal's triangle-like construction whose main interesting statement looks just like the formula above. I picked $n = 12$ to duplicate Knuth's graph of the same function. The average k is 2.11, and the probability distribution is shown in Fig. 1. The smallness of this average may be surprising.

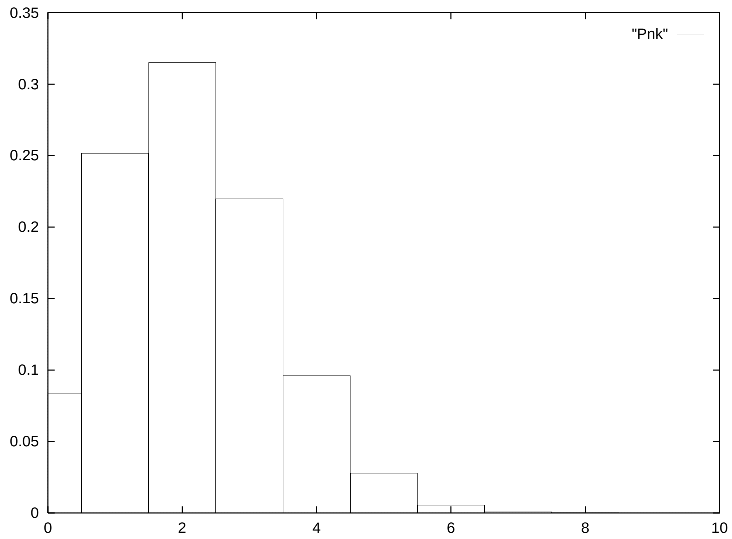

Figure 1: Probability distribution for the number of executions in step 4 with n = 12.

After the above computational argument, it bears repeating that our $p(n, k)$ is equal to (the number of permutations of n objects for which $T = k$) $/n!$. These combinatoric quantities are indicative of the necessity to deploy techniques such as those we cover below. It is fair to say, however, three things.

1. We shall mostly be worried about worst-case complexity and shall not often need all the techniques presented below.

2. On the other hand, even what I show you below is inadequate to solve explicitly the question of the average-case behavior of our algorithm, the mean of $p(n, k)$. Knuth uses a generating function approach, which is a powerful tool that is often used in algorithm analysis (see the handout on Recurrences and Generating Functions).

3. Thus this selection represents a middle ground between the approaches of Weiss and Knuth (and is non-professional and informal in its presentation, of course).

3 Numbers, Powers, Logarithms

The integers are the whole numbers $(\ldots -2, -1, 0, 1, 2, 3 \ldots)$. Rationals are ratios of two integers. Reals have decimal (or binary, or ...) expansions in terms of negative powers of the base: e.g. in decimal, $x = n + 0.d_1 d_2 d_3 \ldots d_k$ with n an integer, d_i a digit between 0 and 9, no infinite series of 9's, and $n + \frac{d_1}{10} + \frac{d_2}{100} + \ldots \frac{d_k}{10^k} \leq x < n + \frac{d_1}{10} + \frac{d_2}{100} + \ldots \frac{d_k}{10^k} + \frac{1}{10^k}$, for all positive integers k.

If b is a positive real,

$$b^0 = 1,\ b^n = b^{n-1} b \text{ if } n > 0,\ b^n = b^{n+1}/b \text{ if } n < 0.$$

You can prove the exponent laws (Weiss 1.2.1 the first and third equations) by induction (try it when we get to inductive proofs).

If u is real and m a positive integer, there is a unique positive real v that is u's "mth root", such that $v^m = u$. We can then define b^r for rational numbers $r = p/q$, or $b^{p/q}$, as the qth root of b^p. If $b = 1$, $b^x = 1^x$, and $1^x = 1$. If $b < 1$, $b^x = (1/b)^x$. We can define b^x for any real x by limiting it between powers of rational expansions of x, but shall not pursue that here.

The inverse of the exponentiation problem is to find a number x such that a given y is equal to some base b raised to the x power. The answer is that x is the logarithm to the base b of y, or $x = \log_b y$. By definition,

$$x = b^{\log_b x} = \log_b(b^x). \tag{1}$$

Various notation exists for logarithms, including the use of ln for logarithms to the base e and lg for $\log_2$. You recall that $e = 2.718281828459045...$, and the "natural" logarithms to this base have many pleasant properties.

In the analysis of algorithms we usually regard the base of the logarithm as irrelevant. To peek ahead a little, it turns out that we are going to be interested in how fast functions grow, and we shall count as the same growth rate any two rates that are linearly related. Thus if some function grows at the rate of $f(n)$ as a function of its input n, and another one grows at the rate of $af(n) + b$, where a and b are constants, then although the second one may be growing faster, it is growing faster in a way that technology can soon cope with: if $a = 10$ say, we just need a processor ten times as fast to make the second function execute as fast as the first. Thus for us in the algorithm analysis business, the difference is not interesting – and as it happens logarithms to different bases are in fact related by a multiplicative constant. This is Theorem 1.1 of Weiss.

Question: how would you go about computing the logarithm of x from scratch?

4 Sums and Products

I'm assuming you know your Σ sum notation. In its most familiar form it has an index running from some starting point (written below the summation sign) to some ending point (written above), as in

$$\sum k = 1\infty.$$

A generalization of the "equals" relation is to sum over all indexes such that any relation R holds, such as

$$\sum k > 5, k < 35, kodd$$

to sum all the odd numbers between 5 and 35 inclusive.

Recall then that when summing over an index i such that some relation $R(i)$ holds, the distributive law operates:

$$\left(\sum_{R(i)} a_i\right)\left(\sum_{S(j)} b_j\right) = \sum_{R(i)}\left(\sum_{S(j)} a_i b_j\right).$$

Of course you can change the variable i to any other single variable or a permutation of the range of summation $p(i)$ without affecting the value of the sum.

The simple operation of interchanging the order of summation can be surprisingly useful and should be kept in mind. (But remember that it is not always valid to exchange order in infinite series sums).

Sometimes summing is simplified if the domains are manipulated... for instance a sum over integers can be broken up into two parts, one over the even integers and one over the odd integers.

$$\sum_{R(i)} a_i + \sum_{S(i)} a_i = \sum_{[R(i) \text{ or } S(i)]} a_i + \sum_{[R(i) \text{ and } S(i)]} a_i.$$

Note: good examples in [1], pp. 30-32., and note how the derivations in Weiss Section 1.2.3 actually use these principles.

5 Modular Arithmetic

Recall the notation $\lfloor x \rfloor$ for the floor, or greatest integer less than or equal to x, and $\lceil x \rceil$ for the ceiling, or least integer greater than or equal to x.

We can then define

$$x \bmod y = x - y\lfloor x/y \rfloor, \quad \text{if} \quad y \neq 0; \quad x \bmod 0 = x.$$

You can convince yourself that $x - (x \bmod y)$ is an integral multiple of y, and that $0 < (x \bmod y) < y$ if $y > 0$. So $x \bmod y$ can be thought of as the remainder when x is divided by y. The mod function is useful for dealing with periodic functions, as in $\sin(x) = \sin(x \bmod 2\pi)$.

In number theory, mod is used in a slightly different way and is associated with the word "modulo" and the concept of congruence, as in Weiss.

Now, $x \equiv y(\text{modulo} \quad z)$ is also written $x \equiv y \pmod z$, and means that $x \bmod z = y \bmod z$, which in turn yields the usual definition, namely that $x - y$ is an integral multiple of z, or again that for some integer k, $y = kz + x$.

In number theoretic contexts, all numbers are assumed to be integers. Two integers are relatively prime if they have no common factor (e.g. 12 and 35). Some identities for congruences are:

(Weiss) If $a \equiv b$ and $x \equiv y$ then $a \pm x \equiv b \pm y$ and $ax \equiv by$ (modulo m).

If $ax \equiv by$ and $a \equiv b$, and if a is relatively prime to m, then $x \equiv y$ (modulo m).

$a \equiv b$ (modulo m) if and only if $an \equiv bn$ (modulo mn) , when $n \neq 0$.

If r is relatively prime to s, then $a \equiv b$ (modulo rs) if and only if $a \equiv b$ (modulo r) and $a \equiv b$ (modulo s).

So we have rules for adding, subtracting and multiplying and thus powers in modular arithmetic just as for regular arithmetic. The second identity lets us do division sometimes, and the third and fourth give rules for changing the modulus.

6 Permutations and Factorials

It is important to know how to count things, and in general combinatorics is an important field in mathematics. There are many good books on combinatorics, as well as a UR course. Here, we shall work our way toward the binomial theorem and binomial coefficients.

If we have n distinct objects in a row, how many ways can they be ordered? Clearly you can choose the first one in n ways, and the 2nd in $n - 1$ ways, so the total number of permutations of n things is $n(n-1)...(2)(1) = n!$, or n factorial.

For fun, write a little program to produce all the permutations of n integers.

Another useful result (convince yourself that it is true) is that the number of distinguishable permutations that can be formed from a collection of n objects in which the first object appears k_1 times, the second appears k_2 times, etc. up to the tth object occuring k_t times, is

$$\frac{n!}{k_1!k_2!...k_t!}.$$

The factorial function is notorious because it appears in "combinatorial" problems that involve permutations and combinations, and because it grows fast: 10! is about 3.5 million, and 1000! has more than 2500 decimal digits. Factorial can be approximated by Stirling's elegant formula:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n.$$

Factorials can also be defined for rational and real values of n. The generalization is called the gamma function.

How many different ways can you choose k objects from a collection of n different objects? That is, how many combinations of n objects taken k at a time are there? Clearly there are $n(n-1)...(n-k+1)$ ways to choose the k objects, and each combination of k things appears $k!$ times in this set of choices. Denoting the combinations of n things taken k at a time by the binomial coefficient $\left(\begin{array}{c} n \\ k \end{array}\right)$, we have

$$\left(\begin{array}{c} n \\ k \end{array}\right) = \frac{n(n-1)...(n-k+1)}{k(k-1)...(1)}.$$

You should remember Pascal's triangle, which shows you the binomial coefficients as well as a recursive way of computing them. With n running down the rows, the 0th, 1st, 2nd ... columns are $\left(\begin{array}{c} n \\ 0 \end{array}\right)$, $\left(\begin{array}{c} n \\ 1 \end{array}\right)$, $\left(\begin{array}{c} n \\ 2 \end{array}\right)$, ...

n	n 0	n 1	n 2	n 3	n 4	n 5	n 6
0	1	0	0	0	0	0	0
1	1	1	0	0	0	0	0
2	1	2	1	0	0	0	0
3	1	3	3	1	0	0	0
4	1	4	6	4	1	0	0
5	1	5	10	10	5	1	0
...							

The following identities or properties are useful for manipulating binomial coefficients.

They can be represented by factorials: for $n \geq k$ integers ≥ 0,

$$\left(\begin{array}{c} n \\ k \end{array}\right) = \frac{n!}{k!(n-k)!}.$$

For integer $n \geq 0$, integer k

$$\left(\begin{array}{c} n \\ k \end{array}\right) = \left(\begin{array}{c} n \\ n-k \end{array}\right)$$

For integer $k \neq 0$

$$\left(\begin{array}{c} n \\ k \end{array}\right) = \frac{n}{k}\left(\begin{array}{c} n-1 \\ k-1 \end{array}\right).$$

For integer $k \neq n$,

$$\left(\begin{array}{c} n \\ k \end{array}\right) = \frac{n}{n-k}\left(\begin{array}{c} n-1 \\ k \end{array}\right).$$

For integer k,

$$\left(\begin{array}{c} n \\ k \end{array}\right) = \left(\begin{array}{c} n-1 \\ k \end{array}\right) + \left(\begin{array}{c} n-1 \\ k-1 \end{array}\right)$$

(this is the basis of Pascal's triangle.)

Applying this last repeatedly, we can get for integer $n >= 0$,

$$\sum_{0 \le k \le n} \binom{r+k}{k} = \binom{r}{0} + \binom{r+1}{1} + \ldots + \binom{r+n}{n} = \binom{r+n+1}{n}. \tag{2}$$

Also, for integer $m >= 0, n >= 0$,

$$\sum_{0 \le k \le n} \binom{k}{m} = \binom{0}{m} + \binom{1}{m} + \ldots + \binom{n}{m} = \binom{n+1}{m+1}.$$

The binomial theorem is a very useful tool (for one thing it justifies the name of the binomial coefficients): for integer $r \ge 0$,

$$(x+y)^r = \sum_k \binom{r}{k} x^k y^{r-k}.$$

Here it appears we are summing over all integers k, but when $k < 0$ or $k > r$ the terms are zero. If r is not an integer, the sum becomes an integral and we have the binomial theorem of the calculus, which is true as stated for all r as long as $\mid x/y \mid < 1$.

The binomial theorem gives us an answer to that thorny question, what does 0^0 equal? (What is it?) Another useful simple case is the classic

$$(1+x)^r = \sum_k \binom{r}{k} x^k$$

The formulae work with negative upper indices: for integer k,

$$\binom{-r}{k} = (-1)^k \binom{r+k-1}{k}.$$

Products of binomial coefficients can often be rewritten using the factorial expansions: e.g. for integer m and k

$$\binom{r}{m}\binom{m}{k} = \binom{r}{k}\binom{r-k}{m-k}$$

Further identities can be proven and Knuth gives several, of which he says the following is the most important:

$$\sum_k \binom{r}{k}\binom{s}{n-k} = \binom{r+s}{n}.$$

References

[1] Donald E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*. Addison-Wesley, 1975.

[2] Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching*. Addison-Wesley, 1975.