Notes on runtime and elements of sorting

One can argue that the single key notion in all of computer science is that of massive repetition. That is, computers (when properly programmed) are great at doing things over and over, and that is where their enormous utility comes from.

This means that lines of code can get executed over and over, instead of just once. (Imagine what it would be like if we had to program a separate line for each execution of an instruction -- to do 1000 steps would take 1000 lines of code!)

On the other hand, if execution of (such a massively repetitious) program required unacceptably large amounts of time, or space, etc, then the utility would not be so enormous after all. Thus we are usually interested in the "efficiency" of a program, as well as that it solve the problem we are interested in. There are several concerns hidden in the notion of efficiency, among them being space (how much memory is needed), time (how long does it take to run to completion), and coding-time (how hard is it to design/write/debug the program).

Our main concern here will be that of time, usually called RUNTIME. However, this does not usually mean the literal clock time measured as the program runs; this is for many reasons. For one thing, that depends on lots of factors in addition to the code itself, eg: the compiler, the operating system, the hardware, not to mention the data. The latter is an especially interesting issue; we normally expect the runtime to increase if the size of the dataset increases, although this is not necessarily the case. However, this suggests that we think of runtime as a function T_p (n), where n is a measure of the amount of data and p is the program. This notation also suggests that we are abstracting away from any physically exact time measure since the other factors above (compiler, OS, hardware) are not specified. But then we need to decide on what "time" measure we are using.

We will aim, as a first approximation, at something like the total number of line executions that occur until completion; this depends on the data and on the program, and nothing else. For many purposes this is too complicated, however, and we can often find a useful (second) approximation (to this first approximation). For instance, for very many sorting algorithms -- of which we will see two below, and more later -- counting the number of executions of data comparisons [Boolean statements such as (x < y)] gives an excellent measure of the overall time performance. Here "excellent" means this: even though the actual count we come up with this way may not tell us about the actual clock time that will pass, and even though it may not even tell us the total number of line executions, we do learn about how time performance depends on data size.

There is an implicit understanding here that, although different instructions may differ in how long they take to execute, these differences should not be very large. Thus a simple assignment (x <-- y) statement may take longer than one that has evaluations embedded in it (x <-- (w + r)/z); but as long as we are confident that no instructions differ in clock time by more than a constant factor (say, 10) then our second approximations are still useful.

Thus if we find that, for one program, p, that solves problem P, $T_p (n) = n^2$, whereas for program q, $T_q (n) = 30n$. In this case, we'd conclude that q is the "better" program since it responds to more data with a proportionate increase in runtime, whereas p magnifies a data increase quadratically, eg:

$T_p (10) = 100$ and $T_q(10) = 300$

but

$T_p (100) = 10000$ and $T_q(100) = 3000$

and worse:

$T_p (10^{10}) = 10^{20}$ and $T_q(10^{10}) = 30*10^{10}$

Thus the runtime for q grows reasonably with data size -- it is reasonable that if we have enough memory to accommodate ten times as much data, then we can pay for ten times as much speed (or are willing to wait ten times as long). Similarly for 10^{10}. But no one can afford to wait 10^{20} --even if the units are nanoseconds -- since that is 10^{11} seconds, about 3000 years! Note that 10^{10} nanoseconds is, by contrast, only 10 seconds.

The lesson here is, in part, that a multiplicative factor, even a seemingly large one such as 300, is relatively unimportant compared to a (seemingly small) power (such as 2). In other words, a linear runtime function T(n) is generally much preferred over a quadratic one, especially if we envision using larger and larger data sets.

Now back to massive repetitions:

There are at least two standard ways to program massive repetition:

(i) iterations, ie looping, as in For-loops, While-loops, etc

(ii) recursion

Each of these programming methodologies leads to a characteristic mathematical formulation for runtime. For iterations, the formulation involves summations; and for recursion it involves recurrence formulas.

Example 1: Insertion Sort

The best-case runtime (counting only data comparisons) is clearly n-1, since each new card will be compared only to the one before it. There are n-1 cards that are "new" after the first one, so we get a sum of 1, n-1 times. This is good, in that it is a linear function, and so only grows proportionally to the size n of the data set. This occurs, however, when the data is already sorted, so there is no point in running a sorting algorithm!

The worst-case runtime occurs when the data is reverse-sorted, so each new card has to be compared to all the cards before it. That is, the second card is compared to the first (1 comparison), then the 3rd to the first two, etc: 1+2+3+...+(n-1) = (n-1)n/2. This is quadratic in n and thus grows with the square of the amount of data. If we increase the data by a factor of 100, the runtime increases by a factor of 10,000.

But perhaps this poor quadratic performance is rare (eg only when the data is reverse sorted); perhaps most of the time the performance is better than quadratic. For this we need an "average-case" runtime analysis.

An intuitive but imprecise argument is this: When we are about to place card i in the correct position among the previous i-1 cards, it has equal chance of belonging in any of the i possilbe positions from 1 to i, so in average it ought to go in the middle position i/2 (assuming i is even). This then gives (roughly) (i-1)/2 comparisons, since we need to compare it to (roughly) half the first i-1 cards. Summing for all i starting with 2, we get 1/2 + 2/2 + 3/2 + ... + (n-1)/2, ie, 1/2 [1 + 2 + ... + (n-1)] = (n-1)n/4. Is this is more or less correct reasoning, then the average case runtime is about 1/2 that is the worst-case, ie still quadratic and thus not very good.

Now we do it again more precisely, using our probabilistic machinery.

Let C be the random variable giving the number of comparisons when Insertion Sort is run on an input string of n particular items. We assume that the input string of n items is provided by some "fair" process (an experiment) that gives equal probability to all n! permutations of the string. This means that when item i is being processed to see where it belongs relative to the previous i-1 items, there is an equal chance that its proper place is in each of the i possible positions (in position 1, position 2, ... , position i. So the probability that it belongs in any position j (where 1 <= j <= i) is 1/i. Note that this is simply another way of saying that the probability distribution is uniform for positions, for a given card i.

The outcomes (the sample space elements, or elementary events) are the n! possible repositionings into an ordered list (smallest to largest -- and as we noted above, this has a uniform probability distribution). For each such outcome, C gives the number of comparisons that were made to get that outcome (by Insertion Sort).

What we want to calculate is the expected value E(C) -- this will be the average number of comparisons. But it is hard to calculate this directly. Instead we use a trick.

Note that C = C1 + ... + Cn, where each Ci is the comparison count for the ith item as it is being processed to see which position j it belongs in. Recall that a new card (card i) might end up all the way in front (ie in position 1, and this will take i-1 comparisons), if it is the smallest value so far, all the way to the back (position i, and this will take only 1 comparison), or anywhere in between. And E(C) = E(C1) + ... + E(Cn). So we need to find E(Ci) = Sum xPr(Ci=x) for each i, and this will be easier to do, that working with E(C) directly.

We know that x can take on any integer value between 1 and i-1, so E(Ci) = Sum_k=1..(i-1) kPr(Ci=k). The case of k=i-1 is special, since it corresponds to item j belonging in either position 1 or position 2 (each of these positions requires comparison of card i with all i-1 previous cards). The other values of k each correspond to a unique position. So we write
E(Ci) = (i-1) Pr(C1=i-1) + Sum_k=1..(i-2) kPr(Ci=k).

But Pr(Ci=i-1) is just 2/i (1/i for each of the two positions). And Pr(Ci=k) = 1/i for all the other values of k. We end up then with
E(Ci) = (i-1)2/i + [1/i + 2/i + 3/i + ... + (i-2)/i], which is just
(i-1)2/i + [1 + 2 + 3 + ... + (i-2)]/i = (i-1)/i + [1 + 2 + ... + (i-1)]/i
= (i-1)/i + (1-i)i/(2i) = (i-1)/i + (i-1)/2.

Finally, E(C) = Sum E(Ci) for i=1...n :

E(C) = Sum_i=1..n [(i-1)/i + (i-1)/2] = [Sum_i=1..n (i-1)/i] + (n-1)n/4.

What does this extra [Sum (i-1)/i] contribute, that we did not see in our initial rough argument above? This can be rewritten as Sum_i=1..n [1 - 1/n] = n - Sum_i=1..n 1/n; the latter expression is a famous sum, called the harmonic series: 1/1 + 1/2 + 1/3 + 1/4 + ... + 1/n . It can be approximated by integrals, and is found to be between lg(n) and lg(n) + 1. This is a very low-order term compared to n^2. Thus our final result is that the average-case runtime for Insertion Sort is indeed quadratic in its highest-order term: (n^2)/4 + 3n/4 - O(lg n) where the big-O means something that behaves like lg n -- we'll study this concept shortly.

We have two debts to pay here: we need to prove the above approximation fact about the harmonic series, and we need to get a clearer understanding of the "behavior" of functions, eg their higher-order terms. Both of these will be covered soon.

Example 2: Merge Sort

Here the best-case, worst-case, and average-case are all the same, since merge sort performs exactly the same number of comparisons no matter what the data items are, ie it depends only on the size n of the data set. We get the recurrence equation -- in the case where n is an exact power of 2:

T(n) = 2T(n/2) + n - 1

We need to find a method for solving recurrence formulas.