Bob's Notes for COS 226 Fall 2013
2: Induction, Rate of Growth Estimation, Amortized Efficiency

Robert E. Tarjan
9/15/13

Mathematical Induction

To extent we do proofs in this course, they will often use the technique of mathematical induction, which I'll review briefly here. I encourage those of you not facile with the use of induction to read the relevant section of a textbook on discrete mathematics or material on the web and to do enough proofs by induction that you understand how the technique works and how to apply it. The idea is very simple. We want to prove the some proposition P holds for all natural numbers (the counting numbers 1, 2, 3,… An example is $P(n)$ = "the sum of the first n integers is $n(n + 1)/2$." Since n can be one of an infinite number of possibilities, we cannot prove that P is true for all n by verifying the equality for any finite set of possibilities, no matter how large. Instead, we use the principle of induction, which states that $P(n)$ is true for all n provided that two things are true: (1) $P(1)$ is true; and (2) If $P(k)$ is true for some $k \geq 1$, then $P(k + 1)$ is true. To prove $P(n)$ by induction, we need to prove (1) and (2). The principle of induction is a mathematical axiom; that is, it is a part of the fundamental framework of mathematics.

Let's prove by induction that the sum of the first n integers is $n(n + 1)/2$. The base case ($n = 1$) states that $1 = 1(1 + 1)/2$, which is true by calculation. What about the inductive step? We need to prove that the sum of the first $k + 1$ integers is $(k + 1)(k + 2)/2$, given that the sum of the first k integers is $k(k + 1)/2$. We argue that the sum of the first $k + 1$ integers is the sum of the first k integers plus $k + 1$, which is $k(k + 1)/2$ plus $k + 1$ by the induction hypothesis ($P(k)$ is true), which is $k(k + 1)/2 + (k + 1) = (k/2)(k + 1) + (k + 1) = (k/2 + 1)(k + 1) = (k + 2)(k + 1)/2 = (k + 1)(k + 2)/2$. Here I have used fundamental properties of arithmetic, in particular commutativity of multiplication ($xy = yx$) and distributivity ($xz + yz = (x + y)z$); normally one omits justifying such elementary calculations.

There are several important variants of induction. One can start at 0 instead of at 1, if one is trying to prove a statement about all non-negative integers. Then the base case is "$P(0)$ is true," and the inductive step is "if $P(k)$ is true for some $k \geq 0$, then $P(k + 1)$ is true." An important variant is strong induction, or course-of-values induction: to prove $P(n)$ for all n, prove "for all k, if $P(j)$ is true for all $j < k$, then $P(k)$ is true." That is, one can assume as the induction hypothesis that the proposition is true for all smaller values of k, not just the next-smaller value. This form of induction combines the base case and the inductive step, since for the smallest possible value of k the inductive hypothesis is trivially true. In applying strong induction it may still be necessary to prove the base case and the inductive step separately. Yet another version turns strong induction into a proof by contradiction: Assume $P(n)$ is false for some n. Then there must be a smallest such n, say k. Prove that there is a $j < k$ such that $P(j)$ is false. This is a contradiction, which means that $P(n)$ must be true for all n.

Perhaps the most important use of induction in computer science is in proving the correctness of computer programs. In this application, the induction is over the number of "steps" taken by the program, where "step" must be defined appropriately: it might be a single operation, or it might be an entire iteration of a loop. One states the correctness of the program as one or more *invariants*, which are propositions claimed to hold at certain points in the computation. One then proves that each invariant holds by induction on the number of times the program execution passes through the corresponding points. Formal verification is a vast and important field, but the fundamental principle underlying it is mathematical induction.

Again I encourage you to become familiar with induction and to do enough proofs by induction to become at ease with the technique.

Rate of Growth Estimation

Here are a couple of rules of thumb to estimate the growth of a function, such as the running time of a program as a function of the input size. Assume the input sizes grow by powers of 2, say 1000, 2000, 4000,… Consider the ratios of successive function values: $f(2N)/\mathrm{f}(N)$. If these ratios are about constant, or tend to a constant as N grows, it is reasonable to suppose that the function is approximately of the form $f(N) = aN^b$: if this is true, $f(2N)/\mathrm{f}(N) = a(2N)^b/(aN^b) = 2^b$, independent of N. Furthermore $\lg(f(2N)/\mathrm{f(N)})$ is an estimate of b, and knowing b one can estimate a by choosing any pair $N, f(N)$ and solving for $a = f(N)/N^b$.

If the unkown function is approximately of the form $aN\lg N$, or more generally of the form $aN^b(\lg N)^c$, the method of the previous paragraph will over-estimate b unless N is huge. But one can extend the idea if one assumes that b is an integer, which it often is. Estimate b by the largest integer such that $f(2N)/f(N) \leq 2^b$. Then compute $g(N) = f(N)/N^b$. If f has the desired form, and your estimate of b is correct, $g(N) = a(\lg N)^c$, and you can apply the method of the previous paragraph to estimate c, and then a. In the special case that $f(N)$ is approximately $aN\lg N$, dividing $f(N)$ by N will give approximately $a\lg N$, which increases by the addition of a for each doubling of N.

Amortized Efficiency

I included a few words about amortized efficiency in my 2010 notes. Here are a few more. The idea of *amortized analysis* applies to a situation in which we are doing not just a single operation but a long sequence of operations, such as a sequence of pushes and pops on a stack, and we are interested in the total cost of all the operations rather than the cost of each operation in isolation. We assign an a*mortized cost* to each operation such that, for any sequence of operations, the sum of the actual costs is at most the sum of their amortized costs. We can use this idea to smooth out variations in the costs of individual operations, in effect charging cheap operations extra to pay for expensive operations. Of course, for this idea to work, we must be able to show that, indeed, the sum of the actual costs is always at most the sum of the amortized costs. How can we do this? One way is to apply the *banker's method* of amortized analysis. We imagine that our computer is coin-operated. We assign a certain number of credits to each operation. Each credit pays for a fixed amount of computation. We spend the credits to do the operation. There are three possibilities. The first is that the number of credits is exactly enough to pay for the operation. The second is that we finish the operation before we run out of credits. In this case we can use the extra credits to pay for later operations, or to pay for earlier operations not yet paid for. The third is that we run out of credits before finishing the operation, in which case we pay for the remaining part of the operation using previously saved credits or by borrowing, which must later be paid off. Our goal is to show that at the end of any sequence of operations all borrowing has been paid off. Then we can say that the number of credits allocated per operation is an upper bound on the amortized cost per operation. To keep track of saved credits we use a *credit invariant* that relates the current number of saved or borrowed credits to the current state of the data structure.

Let's apply this idea to the simulation of a queue by two stacks discussed in precept. We use two stacks, B (bottom) which contains items most recently added to the queue, and T (top), which contains items least recently added to the queue (soonest to be removed). To add an item to the queue, we merely push it on B. To remove an item from the queue, we pop an item from T unless T is empty, in which case we repeatedly pop an item from B and push it on T until B is empty, and then pop one item from T. Suppose one credit can pay for one stack operation (a push or pop), and we allocate 3 credits per add-to-queue operation and 1 credit per remove-from-queue operation. Our credit invariant is that the number of saved credits equals twice the number of items on B. Adding an item to B consumes one credit and increases the number of items on B by 2, accounting for the 3 credits allocated to the add-to-queue. If T is non-empty, popping an item from T consumes the credit allocated to the delete-from-queue operation and does not affect the credit invariant: the size of B does not change. If T is empty on the other hand, moving the items on B one-by-one to T consumes two credits per item on B, exactly equal to the number of saved credits. When B is empty, there are no saved credits, but the credit invariant still holds. Popping an item from T now consumes the credit allocated to the delete-from-queue operation and does not affect the credit invariant. Since the credit invariant holds, the number of saved credits is always non-zero, which implies that the total number of stack operations is at most 3 times the number of add-to-queue operations plus the number of delete-from queue operations. That is, the amortized number of stack operations is at most 3 per add-to-queue plus 1 per delete-from-queue, assuming we start with an empty queue.

Note: In precept I proposed allocating 4 credits per add-to-queue operation and none per delete-from-queue operation. This also works, but it gives a slightly weaker bound than the one in the previous paragraph, since the number of add-to-queue operations is no less than the number of delete-from-queue operations. In general there are many ways to allocate credits. The goal is to find one that allocates as few as possible and that smooths out the cost variations among operations.

We can turn the banker's method around and start with the credit invariant, not with the allocation of credits to the operations. This is the *physicist's method* of amortized analysis. For each state of the data structure, we define a *potential*, which we think of as measuring the maximum excess cost of future operations. We then define the *amortized cost* of an operation to be its actual cost plus the net increase in potential produced by the changes the operation makes to the data structure. If the initial potential is zero and the final potential is non-negative, the sum of the amortized costs of operations in a sequence is an upper bound on the sum of the actual costs. To apply this idea to the queue simulation, we define the potential of a pair of stacks B and T to be twice the number of items on B. Then the amortized cost of an add-to-queue-operation is 3 and the amortized cost of a delete-from-queue operation is 1. (Exercise: verify this.)