0 Problems vs. Algorithms (Jan. 19)

We are so used to thinking about algorithms, the way to do things, that we often forget that they are solutions to problems and that the two are not the same. A problem is a question, a request, a demand for some knowledge. An algorithm is the method to find that knowledge. A problem can be ill-posed, have contradictory assumptions, even be inherently unsolvable, or with luck and rearrangement, be solvable. An algorithm is just a list of instructions attempting to attack what ever problem is set before it. As we design and analyze algorithms, we will want to know the often implied but unstated formulation of the problem it is attempting to solve, and we will want to know whether some attempt, an algorithm, is successful or not according to that formulation.

0.1 What we want to know about problems and algorithms

There are a handful of things that are what we commonly ask about an algorithm and how it relates to the problem that it attempts to solve.

- correctness - When an algorithm returns an answer, we want to *know* that the answer is correct. This gives us the late night programming session joke 'The program compiles. It *must* work!'. Correctness really does capture the essence of the relation between a problem and an algorithm for it. For problem statements that are vague or have hidden assumptions, it is often up to us to state explicitly what those assumptions are in order to make the algorithm correct (by fiat) but also fit general expectations.

- termination - This seems a little deficient in importance because we usually design our programs to end with an answer. However it is not always so obvious. It is easy to tell whether a bounded 'for' loop terminates, but not so a 'while' loop with a guard that has changing variables in it. From a program verification perspective though, one really considers this question as not only of equal importance as correctness but also the flip side of the same coin. We will sometimes see this because the proof of correctness may also have a proof of termination embedded in it.

- resource usage - This usually refers to time or number of particular operations like swaps or multiplications that are a good indication of time. Space is another resource that is measured: number of local variables, registers, size of arrays or data structures, stack depth (each recursion takes up stack space). This is measured with the ubiquitous Big-Oh or order notation which in essence ignores constants that will presumably be overcome easily by technological advance.

  Since resource usage depends on the distribution of the input, we often consider worst case and average case, the former being much easier to analyze. Once we have an algorithm with a given worst case, we have an upper bound for that of the general problem (since we have this algorithm in hand, we won't want to use a worse one). This doesn't address the question of knowing what the optimal case is, which is a property of the problem. If we then find an algorithm with the resource usage of the best possible for the problem, then we really can't do substantially better, we have an optimal algorithm.

- other properties - things like data or process flow structure, floating point accuracy, error stability, robustness, software engineering properties (reusability, structured, etc.), intermediate action (what does the partial solution look like before the algorithm is complete?), beauty (well, not very quantifiable, but it is something that is noticed).

0.2 How do we come to know these facts?

Mathematical proof. The problem statement should be made explicit, in a mathematical, quantified, logical language. It need not be so formally sterile that it is unreasonably unreadable, though that is always a matter of taste. All sorts of mathematical knowledge and proof methods come into play. Combinatorics and algebra are the most common since most algorithms act on discrete structures or symbols. Analysis, geometry, probability, you name it, we'll use it. and logic overall. A very common proof method is induction because of the common design methods of divide and conquer or recursion. We won't be formal as much as rigorous. Formality has its place, but mostly in automatic program verification, where algorithms are designed to prove correctness of other algorithms. Here, rigor will be sought, expressed though it may be in looser but still mathematically sound terms.

0.3 An example: searching

So far I have been sufficiently vague and philosophical as to sound like nothing. Hopefully, we'll hit some of the above points in the following example.

Problem: given a finite set S from a universe U, and an element x from U, is $x \in S$? Assume that all one can do is test for equality.

This problem has many related problems, with different assumptions and different questions. Sometimes the set S or U may be infinite, there may be additional structure on either S or U. What this structure means is that there are some relations that can be used to find out more information on those sets, it does not mean that there is a data structure on it. We might want to construct a data structure in an algorithm to take advantage of that structure, but the problem itself has only relations involved.

Since we have absolutely no knowledge about the set S other than that it is finite, it seems at first sight that there is really nothing we can do to attack this problem. We can make a few real world assumptions that fit with the above statement. We can assume that since the set is finite, we have a representative of each element storable (and stored somehow) in our computer. Again since it is finite, we can mathematically label each element distinctly with a number from 1 to the size of the set. This almost immediately presents to us a likely candidate for a simple data structure to organize these otherwise unorganized elements: an array. Since all we can do is test for equality, let us take the natural brute force method of comparing element x with every one in the list.

```
BruteSearch(S[1..n], x):
    for k ← 1 to n
        if x equals S[k]
            return true
    return false
```

To show that this really is correct we have to prove that if $x \in S$ if and only if the program returns true on inputs x,S [1]. Loosely, (i.e. without being too formal) we can justify that if $x \in S$ at

[1]The statement of this objective is logically equivalent to checking that when $x \in S$, the return value is true and

some point it will return true, since it compares x with every element in S, that is no elements of S are skipped over in the sequence of comparisons that the loop does. If $x \notin S$, then the loop will exit (because n is a constant), and so will reach the final return of false. Termination is apparent since in either case the algorithm returns something.

As to time, the worst case is if $x \notin S$ which takes n steps of the loop to reach. Average case, since each possibility of $x = S_k$ and $x \notin S$ can be assumed to be equally likely, $\sum^{n+1} \frac{k}{n+1} = (n+2)/2$. With different assumptions, different analysis of the average case. What is optimal for the problem? Since knowledge of one item in the set, does not give us any information with respect to the other items an optimal solution *must* consider at least every element (it would be silly though in this instance to check more!) So an optimal algorithm would be $\Omega(n)$, so the above algorithm is optimal (under those given assumptions). Since all we use is a single loop iterator, the space complexity of the algorithm (which does not include the input size) is $\Theta(1)$. All measures are in terms of the input size, the number of elements in the array, n.

0.4 An extension: searching with more knowledge

Suppose now that we have a tiny bit of extra knowledge, that the set U, the universe from which S and x come, has a total (linear) ordering relation on it, that is every element can be compared with every other, just like '$\leq$'. Now what? We could be content, knowing that we can ignore this extra information and use our ready-made algorithm above. Let us pursue instead an algorithm that uses this new information. The set S, as a subset of U can be put into a sequence that respects the ordering relation (because S is finite). This is a roundabout way of saying we can *sort* the elements first. What does this buy us? It buys us extra knowledge that will give us a faster algorithm. Suppose we compare x with the middle (i.e. median) element in that sorted sequence of S. Instead of getting just 'unequals', we can know if x is greater or less than $S[n/2]$. Knowing that comparison for a single element, by transitivity of the ordering relation, we now a whole lot more, that x is greater or less than one half of the set S. This is just an explicit justification of what we easily recognize this algorithm to be: gold old binary search.

```
BinarySearch(S[1..n], x):
    sort S
    i = 1
    j = n
    while (i does not equal j)
        k = ⌊(i + j)/2⌋
        if (x equals S[k])
            return true;
        else
            if x < S[k]
                j = k
            else
                i = k
        return false
```

when $x \notin S$ the return value is false; these logical manipulations can be convenient for proofs but may sometimes obscure the intent.

Now this is not quite so trivial as the example above. Correctness? $\mathtt{i}$ and $\mathtt{j}$ seem to converge, but maybe there's a chance they might cross over ($\mathtt{j}$ become less than $\mathtt{i}$). Of course this won't happen, but it is not trivial to show that. Termination? like for correctness, depends on the guard of the loop. Run time? $\mathtt{i} - \mathtt{j}$ divides by 2 each iteration giving log time. And so on.

Is this the final answer? Not at all. we may have optimal answer for the problem, but more consideration may come into play. Suppose we want to add and delete items from S? Suppose we only have a probability that a particular item is in S. Suppose that we don't care if the algorithm makes a few mistakes sometimes, as long as it's correct most of the time. These and more are the kinds of questions we'll be asking and providing methods for solving for the rest of the semester.