Algorithm Intro: Repeated Squaring, Binary Search

Repeated Squaring

In the problem of modular exponentiation, you are given natural numbers a, b and m and you are required to output $a^b \mod m$. For the purpose of this problem the efficiency of an algorithm is measured by the total number of modular multiplications it uses. That is each operation of the form $x * y \mod m$ costs exactly 1, and all other operations cost exactly 0. What is the most efficient algorithm that you can design for this problem?

Here is a more formal and concise statement of the problem.

Problem name: modular exponentiation.

Problem input: $a, b, m \in \mathbb{N}$.

Problem output: $a^b \mod m$.

A simple (brute-force) way of solving this problem is by having a running total t initialized to 1, and multiplying t by a b times. This uses b multiplications. Can you design a more asymptotically efficient algorithm? In this section, we describe an algorithm that solves modular exponentiation with $O(\log b)$ multiplication. This provides exponential speedup over the brute-force solution.

Algorithm 1 The REPEATED SQUARING algorithm.

```
procedure RepeatedSquaring(a, b, m)
    t ← 1
    x ← a mod m
    p ← b
    while p > 0 do
        if p is odd then
            t ← t * x mod m
            p ← p − 1
        else
            x ← x * x mod m
            p ← p/2
    return t mod m
```

We analyze the above algorithm. The analysis consists of two parts: correctness and running time/efficiency. For this problem we analyze the running time/efficiency with respect to the measure introduced at the beginning of this section - number of multiplications.

To prove correctness of an iterative algorithm with loops we need to mathematically describe the precise nature of "progress" that the algorithm makes in each iteration of the loop. This is captured by the concept of a *loop invariant* - a statement that, if true prior to a given iteration of the loop, remains true prior to the next iteration of the loop. For our algorithm we use the following loop invariant:

$$t * x^p \mod m = a^b \mod m.$$

You can formally prove that this is a loop invariant by induction. Informally, it is easy to see - in a given iteration, if p is odd, then one factor of x gets absorbed into t and p decreases by 1; if p is even then x gets replaced by x^2 but p gets replaced by $p/2$ and $x^p = x^{2*(p/2)}$.

To finish the proof of correctness we need three things: base case, termination condition, and an argument that the algorithm actually terminates. Base case is whether the loop invariant holds prior to the first iteration of the loop. Indeed, it does. Initially, we have $t = 1, x = a, p = b$ so $t * x^p \mod m = 1 * a^b \mod m = a^b \mod m$. Termination condition means the implication of the loop invariant after the algorithm exists from the loop. The loop finishes when $p = 0$. In this case, the loop invariant means $t * x^p \mod m = a^b \mod m$, but we also have $x^0 = 1$ for all x. Therefore the termination condition simplifies to $t \mod m = a^b \mod m$, which the algorithm returns in the last line. So far we have proved that if the algorithm returns, it returns the right answer. Lastly, let's convince ourselves that the algorithm terminates. We start with $p = b > 0$. Then in each iteration p is decreased by at least 1. Thus, eventually p will reach 0 and the algorithm would break out of the loop. Therefore the algorithm runs in finite number of steps.

The next step is to analyze the efficiency with respect to our measure of interest - number of multiplications. Note that the algorithm performs one multiplication per iteration of the loop. Thus, it suffices to count the number of iterations of the loop. Observe that if p is odd in one iteration of the loop, it will be even in the next iteration of the loop. Moreover, when p is even it is halved in the corresponding iteration. From this, it follows that p is halved every two iterations of the loop. Let p_i be the value of p prior to the ith iteration. Note $p_1 = b$. Then we have $p_{1+2} \leq p_1/2 = b/2$. Similarly, we get $p_{1+4} \leq b/2^2$, and $p_{1+2*3} \leq b/2^3$, etc. By induction we have $p_{1+2k} \leq b/2^k$. As soon as p_i drops below 1 we terminate. Thus, we are interested in k such that $p_{1+2k} < 1$. This is guaranteed as soon as $b/2^k < 1$. This inequality is satisfied for $k = \log b + 1$. Thus, the total number of iterations before algorithm exists from the loop is $\Theta(\log b)$.

Important note:

- Why did we use the number of multiplications as proxy for efficiency? First of all, it makes the measure mathematically easy to work with and clean - and this is often very important. Secondly, we think of multiplication as being really expensive compared to other operations. All the other steps - index manipulations, moving variables around in memory, initializing variables - are performed alongside multiplications, but they tend to not cost as much. In reality, they won't have 0 time cost, but the asymptotic running time would be dominated by the term "(number of multiplications)*(time cost of a multiplication)". Lastly, analyzing the number of multiplications allows us to quickly argue about running times of similar algorithms in other scenarios, e.g., taking powers of matrices with entries reduced modulo m. In this case, we would quickly conclude that the running time is of the order $\log b$ * (time to multiply two matrices).

Binary Search

Problem name: binary search.

Problem input: sorted array A of n real numbers, i.e., $A[1] \leq A[2] \leq \cdots \leq A[n]$.
A real number t.

Problem output: NO if t is not one of the elements of A.
index i such that $A[i] = t$ otherwise.

We would like to design an algorithm that uses as few comparisons as possible. That is, for this problem the measure of efficiency is the number of comparisons between real numbers. We present the binary search algorithm.

Algorithm 2 The BINARY SEARCH algorithm.

```
procedure BinarySearch(A[1...n], t)
    ℓ ← 1
    u ← n
    if A is empty then return NO
    while u > ℓ do
        m ← ⌊(ℓ+u)/2⌋
        if t ≤ A[m] then
            u ← m
        else
            ℓ ← m + 1
    if A[ℓ] = t then return ℓ
    else return NO
```

Next steps are to prove correctness and analyze efficiency. To prove correctness use the following loop invariant:

$$\text{If } t \text{ is in } A \text{ then } A[\ell] \leq t \leq A[u].$$

To analyze the efficiency note that the algorithm performs one comparison of real numbers per one iteration of the loop. Thus, we need to count the number of iterations. Note that the algorithm keeps halving the interval $\ell..u$ in each iteration until $u = v$. At that point if t is in A then $A[\ell] = A[u] = t$ - the algorithm checks for this condition explicitly at the end. Thus, the algorithm uses $O(\log n)$ comparisons to solve the problem.

Careful proofs of the above statements are left to the reader. Careful proofs are extremely important! As a way of motivating you to write out the details of the proof, I present an incorrect version of binary search. Your task is to find why the algorithm below fails (give example of actual input, on which it fails). This subtle mistake is not just of theoretical value - it has appeared in practice many times.

Algorithm 3 The INCORRECT (!!!) BINARY SEARCH algorithm.

```
procedure IncorrectBinarySearch(A[1...n], t)
    ℓ ← 1
    u ← n
    if A is empty then return NO
    while u > ℓ do
        m ← ⌊(ℓ+u)/2⌋
        if t ≥ A[m] then
            ℓ ← m
        else
            u ← m − 1
    if A[ℓ] = t then return ℓ
    elsereturn NO
```