Lecture 17

Numerical algorithms: worked examples.

Fibonacci sum: Recall that the Fibonacci sequence is $F_0, F_1, \ldots = 1, 1, 3, 5, 8, 13, 21, \ldots$ (other than the first two numbers, every following number is the sum of the previous two). Develop an algorithm to calculate the sum $F_0 + F_1 + \ldots + F_n$.

Example solution: Let $S_n = F_0 + F_1 + \ldots + F_n$. Note

$$\begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} F_i \\ F_{i-1} \\ S_{i-1} \end{bmatrix} = \begin{bmatrix} F_i + F_{i-1} \\ F_i \\ S_{i-1} + F_i \end{bmatrix}.$$

If we let A be the matrix on the lefthand side above, then

$$A^n \cdot \begin{bmatrix} F_1 \\ F_0 \\ S_0 \end{bmatrix} = \begin{bmatrix} F_{n+1} \\ F_n \\ S_n \end{bmatrix}.$$

This, we can compute S_n in $O(\log_2 n)$ arithmetic operations via fast powering. One can also prove that $S_n = F_{n+2} - 1$, so another option is to just reuse the code for computing Fibonacci numbers and subtracting one.

Identifying perfect powers: We say a positive integer n is a **perfect power** if we can write $n = a^b$ for two integers $a, b > 1$. For example, $n = 9$ is a perfect power since $9 = 3^2$. Also $125 = 5^3$, $64 = 2^6$, and $256 = 4^4$ are all perfect powers. 15 on the other hand is not a perfect power.

We would now like to solve the following **problem:** given an iteger n, decide whether it is a perfect power (`True` or `False`).

What are some algorithms to solve this problem?

Solution 1: One approach is to loop over all possibilities for a, then try all possible b.

```
def perfpower(n):
    if n < 2:
        return False
    a = 2
    while a*a <= n:
        x = a*a
        while x < n:
            x *= a
        if x == n:
            return True
        else:
            a += 1
    return False
```

What is the runtime of the above method? Since we only check a with $a^2 \leq n$, that means we only check at most $\sqrt{n}$ values of a. For each such a, we can only power a with an exponent of at most $\log_a n \leq \log_2 n$ before $x < n$ stops holding. Each multiplication of x by a takes $O(\log^2 n)$ time since each of a, x is at most n and thus has at most $\log_2 n$ digits. Thus the total runtime is $O(\sqrt{n} \log^3 n)$. Note it is not too difficult to come up with an input which makes this algorithm take $\Omega(\sqrt{n})$ time: just give as input any n which is not a prime power. Alternatively, give an n which is the square of a prime. For example, try running the above code with $n = 965211250482432409$ (which is 982451653^2).

Solution 2: A better approach is to loop over all possibilities for b, then figure out what a should be using binary search. For a given guess of a, we can compute a^b quickly using fast exponentiation.

```
def fastExp(a, b):
    if b == 0:
        return 1
    x = fastExp(a, b/2)
    x *= x
    if b % 2 == 1:
        x = x * a
    return x

def perfpower(n):
    if n < 2:
        return False
    twopow = 1
    log = 0
    while twopow <= n:
        log += 1
        twopow *= 2
    for b in xrange(log):
        lo = 2
        hi = n
        while lo < hi:
            mid = (lo+hi)/2
            x = fastExp(mid, b)
            if x > n:
                hi = mid
            elif x == n:
                return True
            else:
                lo = mid+1
    return False
```

Unfortunately the above code is slow for very large n which are not a perfect power. For example, try

$$n = 2582249878086908589655919172003011874329705792829223512830659356540647622016841194629645353280137831435903171972747493377$$

which is $2^{400}+1$. The issue is that `fastExp` is slow since very large numbers are being exponentiated, so the intermediate numbers being calculated are quite big.

Let's make the runtime estimation a little bit more precise. We iterate over $\lfloor \log_2 n \rfloor$ values of b. For each value of b, we binary search over n values, taking $O(\log n)$ time. In each iteration of binary search, we perform a fast exponentiation mid^b. During the fast exponentiation we multiply $O(\log n)$-digit numbers, taking $O(\log^2 n)$ time, and we do this $O(\log b) = O(\log \log n)$ times. Thus the call to `fastExp(mid, b)` takes time $O(\log^2 n \log \log n)$. Thus the overall runtime is $O(\log^4 n \log \log n)$. This is not bad, but we can make things slightly faster ...

Solution 3: This solution is very similar to Solution 2 above, but with one optimization. The main observation is that if we attempt to compute mid^b in `fastExp` but realize that the output will be larger than n before finishing the computation, we can abort early (since the binary search only cares whether the result is larger than n, equal to n, or smaller than n). See the code below; `fastExp` returning -1 means that it has aborted because the result is too big (bigger than n).

```
# returns -1 if a^b > limit
def fastExp(a, b, limit):
    if b == 0:
        if 1 > limit:
            return -1
        return 1
    x = fastExp(a, b/2, limit)
    if x == -1 or x > limit:
        return -1
    x *= x
    if x > limit:
        return -1
    if b % 2 == 1:
        x = x * a
        if x > limit:
            return -1
    return x

def perfpower(n):
    if n < 2:
        return False
    twopow = 1
    log = 0
    while twopow <= n:
        log += 1
        twopow *= 2
    for b in xrange(log):
        lo = 2
        hi = n
        while lo < hi:
```

```
        mid = (lo+hi)/2
        x = fastExp(mid, b, n)
        if x == -1:
            hi = mid
        elif x == n:
            return True
        else:
            lo = mid+1
return False
```