Austin Bennett
Final Question #2:

1.
Determining the number of bits needed to represent a number N is much like determining the number of digits
$\rightarrow \lceil \log_{10}(N) \rceil$ where $\lceil \; \rceil$ is rounding up.
To determine the number of bits we would take $\lceil \log_2(N) \rceil$.
The Big-O would then be $O(\log_2(N))$.

2. The worst case of multiplying two integers between 1 and N would be $N \cdot N$ as they have the highest number of bits.
The process of binary multiplication consists of multiplying each bit of one number by all of the bits of the other number + some additional constant time for carrying bits.
So multiplying $N \cdot N$ would require each bit (total of $\log_2(N)$) by all of the bits of N ($\log_2(N)$). Thus $O(\log_2(N)^2)$

3. Fast exponentation for N^K would be like
$N^1 = N$
$N^2 = (N)^2$
$N^4 = (N^2)^2$
$N^8 = (N^4)^2$
where we are simply repeatedly squaring powers of 2 until we have exponent values adding up to K.

Ex. $N^{11} = N^{8+2+1} = (N^4)^2(N^2)(N)$
Thus for N^k we are computing at most N^{2^d} where $d = \lfloor \log_2 k \rfloor$
$\lfloor \log_2(11) \rfloor = 3 \to N^{2^3} = N^8$ and by computing up to N^8 (the closest power of 2 to k) we have then all powers of N necessary for the multiplication to N^k.

At this point we have:
$$\sum_{d=1}^{\lfloor \log_2 k \rfloor} O(2^{2d}(\log_2 N)^2)$$

Which is the total work to compute all necessary powers of N up to $\lfloor \log_2 k \rfloor$.
The only remaining work being the multiplications of our select powers of N adding to k.
Thus we have approximately
$$\sum_{d=1}^{\lfloor \log_2 k \rfloor} O(2^{2d}(\log_2 k) \cdot (\log_2 N)^2).$$

4. Our worst case would be $a^2 > N$ such that N does not have an integer square root, implying we must search the entire list. Hence we must also compute the square of every integer in our list.
We previously proved the big-O of computing $N \cdot N \equiv (\log_2 N)^2$
$\therefore O(N(\log_2 N)^2)$
- $N \cdot N$ multiplications, N times = loose upper bound.

Austin Bennett
Final Question #2:

5.
The complexity of an ordinary binary search is $O(\log N)$.
Our binary search is unique in the sense that we are really searching for $\sqrt{N}$ but we do not know an algorithm for finding square roots so we must compute a^2 and compare to N.
Thus, in addition to computing $\log(N)$ ~~te~~ shrink our list size we must also compute $a^2 \Rightarrow$ worst case N^2.
$\therefore O((\log_2 N)^2 \cdot (\log_2 N)) = O((\log_2 N)^3)$

6. Again the worst case is when $a^k > N$ such that ~~k~~ to $\sqrt[k]{N}$, we do ~~not~~ have $a^k = N$, so we must compute a^k, $\sqrt[k]{N}$ times.
Assuming the worst possible case we are computing N^k, N times.
N^k can be expressed as $N \cdot N$, K times or $K(\log_2(N)^2)$
$\therefore O(\sqrt[k]{N} K(\log_2(N)^2))$

7. Ordinary binary search: $O(\log N)$
For each divide and conquer we are instead computing $N^k \Rightarrow O(K(\log_2(N)^2)$
$O(\log_2 N) O(K \log_2(N)^2) = O(K(\log_2(N)^3))$

Note: Theoretically our best choice of largest K would be $\lfloor \log_K N \rfloor$ but K is originally unknown and thus we must use $\lfloor \log_2 N \rfloor$ as to ensure we do not miss possible roots.

Perfect

8.

The smallest value of K that may need to be considered is 1 because if $K=0$ $N^{1/0} \Rightarrow$ undefined

The largest value of K necessary is $\lfloor \log_2 N \rfloor$ since we are attempting to find perfect Kth root numbers. Taking the Kth root $> \log_2 N$ is futile as N is strictly smaller than $2^{\lceil \log_2 N \rceil}$.

$\therefore O(\log_2 N)$

9.

```
boolean kth Root (int N){
    loop i=1 to N {
        loop j=1 to ⌊log2 N⌋ {
            if ((i^(1/j)) % 1 == 0)
                ~~print "i is a perfect jth power"~~
                ~~return true;~~
        }
    }
}
~~return false;~~

def binary_search (list x, int N):
    if |list| = 0 : return -1
    let midpoint be the middle value in the list
    if kthRoot (midpoint) == 1 return position
        of midpoint.
    if midpoint < N : return binary_search (upper half
        of the list, N)
    if midpoint > N : return binary_search (
        lower half of list, N)
```

Austin Bennett
Final Question #2

9. continued..

ordinary binary search + computing all kth powers for each N = total work

$O(\log_2 N)$ - binary search

$\sum_{d=1}^{\lfloor \log_2 k \rfloor}$ $O(2^{2d}(\log_2 N)^2)$ - computing N^k, doing this multiple times, for k many times.

$\Rightarrow \sum_{d=1}^{\lfloor \log_2 k \rfloor} O(2^{2d} k (\log_2 N)^2)$

We compute the above, N many times for 1 to N to find all perfect kth roots.

total work $= \sum_{d=1}^{\lfloor \log_2 k \rfloor} O(2^{2d} N k (\log_2 N)^3)$