Collaborators : None

Sources : Lecture Notes; https://www.geeksforgeeks.org/check-if-a-given-number-is-a-perfect-square-using-binary-search/

Q4 Perfect Powers

(a) Give pseudocode for an algorithm that takes as input a positive integer A and determines whether or not A is a perfect square. If it is, your algorithm should also determine the number B such that $B^2 = A$. If A is n binary digits long, your algorithm should take $O(nM(n))$ steps, where $M(n)$ is the number of steps required to multiply two numbers of at most n binary digits. Thus, your algorithm should take $O(n^3)$ with the usual grade school multiplication algorithm; or, it would be $O(n^2 \log n \log \log n)$ steps with the sophisticated Schönhage–Strassen multiplication algorithm. (Hint: binary search.)

Ans: To determine whether a number A is a perfect square, we can use **binary search** on the range from 1 to A. We search for an integer B such that $B^2 = A$. The binary search ensures that the algorithm runs efficiently in $O(n^3)$ with grade-school multiplication, or faster with advanced algorithms like Schönhage–Strassen.

Pseudocode:

```
isPerfectSquare(A):    // Returns (Boolean(Square or not),B)
  1. If A == 0 or A == 1:
       return (True, A)
  2. Set low = 1, high = A
  3. While low ≤ high:
       a. Set mid = (low + high) // 2
       b. Set square = mid * mid
       c. If square == A:
            return (True, mid)
       d. If square < A:
            Set low = mid + 1
       e. Else:
            Set high = mid − 1
```

4. Return (False, None)

Explanation:

- We initialize two pointers, $[low, high]$, to the range $[1, A]$. This is the search space to find square-root.
- Each step involves checking the square of the middle value (mid) and comparing it to A.
- If $mid^2 = A$, then A is a perfect square and we return $B = mid$.
- If $mid^2 < A$, we know the square root must be greater, so we update $low = mid + 1$ as we don't consider the search space below mid as their square roots will obviously be below A.
- If $mid^2 > A$, the square root must be smaller, so we update $high = mid - 1$ as we don't consider the search space above mid as their square roots will obviously be above A.

Time Complexity:

- **Binary Search**: The binary search runs in $O(log\ A)$ steps, which is $O(n)$ since A has n bits.

$$O(log\ A) \approx O(log\ 2^n) \approx O(n)$$

- **Multiplication**: Each multiplication of n-bit numbers (to compute mid^2) takes $O(n^2)$ steps with the grade-school multiplication algorithm or $O(n\ log\ n\ log\ log\ n)$ with the Schönhage–Strassen algorithm.
- **Total**: The overall complexity with binary search and multiplication is $O(n^3)$ using grade-school multiplication or $O(n^2\ log\ n\ log\ log\ n)$ with Schönhage–Strassen.

(b) Give pseudocode for an algorithm that takes as input a positive integer A and determines whether or not A is a perfect power (i.e., a perfect square, cube, fourth power, etc.). If it is, your algorithm should also determine numbers B and $C > 1$ such that $B^C = A$. When A is an n-bit number, justify that your algorithm takes at most $O(n^d)$ steps for some constant d (such as $d = 5$).

Ans: To determine whether A is a perfect power (i.e., if there are integers B and $C > 1$ such that $A = B^C$), we can iterate over possible values of C and use binary search to find B. The main idea is to try different values of C and check whether A is a perfect C-th power.

Pseudocode:

isPerfectPower(A):   // Returns (Boolean(Power or not),B,C)

1. If A == 1, return (True, 1, any C > 1)

2. For C = 2 to ⌊log2(A)⌋:

a. Set low = 1, high = A

b. While low ≤ high:

i. Set mid = (low + high) // 2

ii. Set power = mid^C

iii. If power == A, return (True, mid, C)

iv. If power < A, set low = mid + 1

v. Else, set high = mid − 1

3. Return (False, None, None)

Explanation:

- The outer loop iterates over possible values of C from 2 up to $\log_2 A$. (As $C = \log_B A \leq \log_2 A\ as\ B \geq 2$)
- For each C, we use binary search to find a base B such that $B^C = A$.
- In the inner loop, we perform binary search on B in the range from $1\ to\ A$, checking whether $B^C = A$.
- If such a B is found, the algorithm returns B and C.

Time Complexity:

- **Outer loop**: There are $O(logA) \approx O(n)$ possible values of C, where n is the number of bits in A.
- **Binary search for $\boldsymbol{B}$**: For each C, we perform binary search on B, which takes $O(logA) \approx O(n)$ iterations, each involving an exponentiation.
- **Exponentiation**: The computation of B^C can be done using fast exponentiation, which takes $O(logC) \approx O(log\ log_2 A) \approx O(log\ n)$ steps with binary exponentiation and $O(n^2)$ for each multiplication. So, worst-case exponentiation costs $O(n^2 log\ n)$ time complexity. This results in a time complexity $of\ O(log\ A\ n^2 log\ n) \approx O(n^3 \log n)$ per iteration.
- **Total**: The total time complexity is $O(log\ A\ n^3\ log\ n) \approx O(n^4 \log n)$ with grade-school multiplication, or $O(n^3\ (\log n)(\log n)(\log\log n))$ with more advanced multiplication algorithms.
- Thus, as $O(n^4 \log n) < O(n^5),$ for $d = 5$, the algorithm doesn't take more than $O(n^5)$ steps to check if A is a perfect power.