Algorithm Design, Efficiency Analysis, Outputs, and Answers

By: Sijie Shang and Claire

# 1. Algorithm Design

Description: Iterate through every number from range 0 to **input x** and try to find an **integer i** that makes i to the power of **integer n** equal to **integer x**.

**Pseudocode for Exhaustive Search:**

```
def exhaustive_search (n, x):
      for i in range(0, x+1):
            if (i^n == x):
                  return i
      return None
```

# 2. Efficiency Analysis

**Decrease-By-Half:**

Complexity Function:
We use the master method to get the time complexity.

Base case: T(0) = 3

T(n) = 1*T(n/2) + 4+2 = T(n/2) +6

So r =1, d=2, k=0

r=d^k

**-> O(logn)**

**Exhaustive Search:**

Complexity Function:
T(n) = O(g + c * v)

We assume x has m element
T(n) = n+1
**-> O(n)**

Proof:
T(n) = n + 1 ∈ O(n + 1) (trivial)
= O(max(n, 1)) (dominated terms)
= O(n) ✓

## Output

```
Exhaustive Search:
The positive int r is 56.
The function looped 57 times.

Decrease-by-half Search:
The positive int r is 56.
The function looped 62 times.
```

## 3. Implementation and measurement

Which algorithm did you expect to perform better? Do the results surprise you?

Based on our program, the result is 56. Which means 56^11 = 16'985'107'389'382'393'856.

When we trace the times of the loops, exhaustive search only looped 57 times. And the bisection method looped 63 times.

This result surprised us because exhaustive search should be the slow one and bisection method should be more efficient since it takes logn time.

The reason for this is because the result is small, which is only 56. It is easy to just test one by one from 1, the bisection method doesn't have much advantage in this case.

If we change the $4^{th}$ root of 1E+12, then exhaustive search needs 1001 loops and bisection search needs only 39 loops.