P3

- *Name & purpose*:
  → it computes n^k efficiently by reducing the number of multiplications using exponentiation by squaring

- *Inputs*:
  → **n**: integer (the base)
  → **k**: integer (the exponent)

- *Outputs*:
  → **p**: integer (the result of n^k)

- *Preconditions*:
  → **k** ≥ 0
  → both **n** and **k** are integers

- *Postconditions*:
  → returns the exact value $p = n^k$

- *High-level idea*:
  → if *k* is *odd* → multiply the result by *n* and subtract 1 from *k*
  → if *k* is *even* → square *n* and halve *k*
  → repeat until *k* becomes 0
  → it reduces the number of multiplications by halving *k* when possible

- *Pseudocode*:

```
# which is the best/ worst case of the following algorithm?
n = 2
k = 4
p = 1

while k > 0:
    if k % 2 == 1:
        p = p * n
        k = k - 1
    else:
        n = n * n
        k = k // 2
print(p)
```

- *Complexity*:
  → time: O(log k)
  → space: O(1)
  → best case: *k* is a power of 2 (fewest multiplications)
  → worst case: k is odd (extra multiplications)

- *Correctness sketch*:
  → it works because each step keeps the result equal to n^k, and when *k = 0*, *p* is the final value.

- *Edge cases*:

  → k = 0 → returns 1
  → n = 0, k = 0 → undefined, assumed 1
  → n = 0, k > 0 → returns 0
  → very large numbers may overflow