Finger Exercises 15

Searching & Approximation

≈ 60 min + stretch

Comp 102 — Programming Fundamentals

What this is. A workout for what we just covered in class. Most of the numbers and programs here are new, so you have to *work them out*, not remember them. For a search, ask: *which guess comes next, and when does the loop stop?* For a float: *is this exact, or only close?*

How to do it. On **paper**, **no computer**, in one sitting. If an item takes you more than a minute, skip it and come back. Parts A–M are for *everyone*; Part N is a **stretch**. The Console Quests are for tonight, *at* a computer. The answer key is on the last pages — do not peek until you are done.

Predict first, then check. Write down what you think *before* you look anything up or type anything in — even for the Console Quests. A guess that turns out wrong teaches you far more than an answer you only watched the computer produce.

Part A • Do You Remember?

RECALL ≈ 4 min

A1. Guess-and-check works on a problem when you know the set of all possible _____, and you can _____ whether one of them is correct. You keep guessing until you find the answer, or until you have _____.

A2. In a sequential (systematic) search, `next_guess = prev_guess +` _____.
In a bisection search, `next_guess = (`_____ `+` _____`) / 2`.

A3. A Boolean used as a signal that something happened is called a Boolean _____. It is set to _____ *before* the search, and changed to _____ when the answer is found.

A4. Bisection search needs two things to be true. The search space must have an _____, and after each guess you must be told whether the guess was too _____ or too _____.

A5. Two floats should not be compared with `==`. Instead we test whether they are within a small amount of each other, called _____. A smaller one makes the answer more _____, but a sequential search then needs a smaller _____, and so many more guesses.

A6. Circle **True** or **False**:

(a) `0.1 + 0.1 + 0.1 == 0.3` is `True` in Python. True / False

(b) 0.1 written in binary goes on forever, so it is never stored exactly. True / False

(c) A sequential search over floats can usually land *exactly* on $\sqrt{5}$. True / False

(d) In a bisection search the search range stays the same size from guess to guess. True / False

(e) With two nested loops, the inner loop runs all the way through once for *each* value of the outer loop. True / False

(f) If the search finds nothing and the program has no flag, it may print nothing at all. True / False

(g) Bisection search can find a PIN when the only answer is "correct" or "wrong". True / False

(h) Every float sum is wrong in the last digit. True / False

Part B • What Does It Print?

TRACE ≈ 5 min

Write every line of output.

B1.

```
1 x = 9
2 guess = 0
3 while guess**2 < x:
4     guess = guess + 1
5 print(guess)
```

Output: _____

```
1 x = 10
2 guess = 0
3 while guess**2 < x:
4     guess = guess + 1
5 print(guess)
```

Output: _____

B2.

```
1 x = 0
2 for i in range(10):
3     x += 0.1
4 print(x == 1)
```

Output: _____

```
1 y = 0
2 for i in range(10):
3     y += 0.125
4 print(y == 1.25)
```

Output: _____

B3.

```
1 print(0.1 + 0.2)
2 print(0.1 + 0.2 == 0.3)
3 print(0.3 * 3)
4 print(0.5 + 0.25 == 0.75)
```

Output:

_____

_____

_____

_____

B4. A systematic search that stops with `break`.

```
1 guess = 0
2 while guess < 10:
3     if guess * guess > 20:
4         break
5     guess += 1
6 print(guess)
```

Output: _____

Times line 3 asked its question: _____

B5. The same flag, without and with `break`.

```
1 found = False
2 for g in range(1, 6):
3     if g * 3 == 12:
4         found = True
5 print(found, g)
```

Output: _____

```
1 found = False
2 for g in range(1, 6):
3     if g * 3 == 12:
4         found = True
5         break
6 print(found, g)
```

Output: _____

Part C • One Program, Five Inputs

TRACE ≈ 5 min

```
1 x = int(input("Enter an integer: "))
2 guess = 0
3 while guess**2 < x:
4     guess = guess + 1
5 if guess**2 == x:
6     print("Square root of", x, "is", guess)
7 else:
8     print(x, "is not a perfect square")
```

This is the slides' program. Run it in your head for each input. Write the value of `guess` when the loop ends, how many times line 4 ran, and what is printed.

Input	guess at the end	Line 4 ran	Printed
16	_____	_____	_____
15	_____	_____	_____
1	_____	_____	_____
0	_____	_____	_____
-4	_____	_____	_____

Part D • Every Pair, in Order

TRACE ≈ 5 min

D1. Write every line this prints, in order.

```
1 for a in range(3):
2     for b in range(2):
3         print(a, b)
```

Output:

_____ _____ _____

_____ _____ _____

Lines printed: _____

The 4th line: _____

D2. How many times does the innermost line run? Do not list them; count.

	Loops	Times
1	`for ali in range(18):` / `for fatima in range(18):` / `check(ali, fatima)`	_____
2	`for ali in range(11):` / `for zain in range(11):` / `for fatima in range(11):` / `check(...)`	_____
3	`for a in range(4):` / `for b in range(a):` / `print(a, b)`	_____
4	`for a in range(5):` / `for b in range(5):` / `if a + b == 4:` / `print(a, b)`	_____

D3. The ticket problem from the slides: Fatima sells 3 fewer than Ali, and together they sell 17.

```
1 checks = 0
2 for ali in range(18):
3     for fatima in range(18):
4         checks += 1
5         if ali + fatima == 17 and ali - fatima == 3:
6             print(ali, fatima, checks)
7 print(checks)
```

Line 6 prints:

_____

Line 7 prints: _____

A `break` is added after line 6, inside the `if`. It leaves only the *inner* loop. Now line 7 prints _____.

Part E • Word Problem to Code

TRANSLATE ≈ 5 min

E1. Each sentence is one test in a guess-and-check, like `two_less = (zain == ali-2)` in the slides. Write it as a Boolean expression. Row 1 is done for you.

Sentence	Test
Together, Ali and Fatima sell 17 tickets.	`ali + fatima == 17`
Fatima sells 3 fewer tickets than Ali.	_____
Fatima sells twice as many as Ali.	_____
Sara sold 5 more than Hina.	_____
Ali sold at least 4 tickets.	_____
The number Ali sold is even.	_____
The three of them sold 10 in all.	_____

E2. Hamza and Sana have 20 marbles together, and Hamza has 4 more than Sana. Fill in the blanks so that the program finds them.

```
for hamza in range(_____):
    for sana in range(_____):
        if _____________ and _____________:
            print(hamza, sana)
```

It prints: _____

Why is the range not `range(20)`?

_____

E3. Not every word problem has an answer in whole numbers.

```
found = False
for a in range(13):
    for b in range(13):
        for c in range(13):
            if a+b+c == 12 and b == a+1 and c == 2*b:
                found = True
                print(a, b, c)
if not found:
    print("No solution found")
```

It prints:

_____

Hint: write `b` and `c` in terms of `a`, and add the three up.

Part F • Exact, or Only Close?

CLASSIFY ≈ 4 min

Tick what each line prints.

Line	True	False
`print(0.1 + 0.2 == 0.3)`	☐	☐
`print(0.5 + 0.25 == 0.75)`	☐	☐
`print(0.125 * 8 == 1.0)`	☐	☐
`print(0.1 * 3 == 0.3)`	☐	☐
`print(0.3 * 3 == 0.9)`	☐	☐
`print(1/4 + 1/4 == 0.5)`	☐	☐
`print(0.1 + 0.7 == 0.8)`	☐	☐
`print(1.5 * 1.5 == 2.25)`	☐	☐
`print(abs((0.1 + 0.2) - 0.3) < 0.0001)`	☐	☐

Part G • Close Enough? TRACE ≈ 3 min

The test from the slides is `(guess**2 >= n-epsilon) and (guess**2 <= n+epsilon)`. Here `n = 5`. Tick the guesses that pass, first with `epsilon = 0.1` and then with `epsilon = 0.01`.

guess	guess**2 (about)	passes, epsilon = 0.1	passes, epsilon = 0.01
2.0	4.0	☐	☐
2.2	4.84	☐	☐
2.23	4.9729	☐	☐
2.236	4.9997	☐	☐
2.24	5.0176	☐	☐
2.26	5.1076	☐	☐

With `epsilon = 0.1`, `guess**2` must lie between _____ and _____.

Part H • Sequential Search on the Number Line DRAW ≈ 6 min

The slides' approximation program, with a counter added:

```
1  found = False
2  guess = 0
3  tries = 0
4  while guess**2 < n:
5      if (guess**2 >= n-epsilon) and (guess**2 <= n+epsilon):
6          found = True
7          break
8      guess += increment
9      tries += 1
10 if found:
11     print(f"Square root of {n} is {guess}, in {tries} tries")
12 else:
13     print(f"Couldn't find the square root of {n}")
```

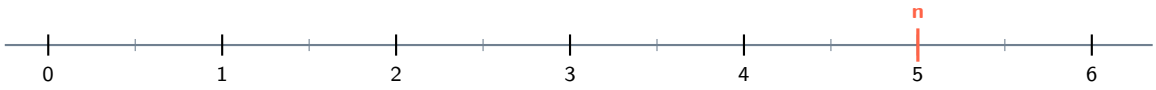
For each run, **shade** the window on the number line (where `guess**2` has to land), draw an **arrow** down to the value of `guess**2` for every guess that line 5 checks, and write what the program prints.

H1. `n = 5`, `increment = 0.5`, `epsilon = 1.0`.



Prints: _____

H2. `n = 5`, `increment = 0.25`, `epsilon = 0.5`.



Prints: _____

The next guess, `2.25`, has `2.25**2 = 5.0625`, which is inside the window. Why was it never checked?

_____

H3. `n = 4`, `increment = 0.5`, `epsilon = 0.1`. $\sqrt{4}$ is exactly `2.0`, and `2.0` is one of the guesses. What is printed?

_____

H4. To find $\sqrt{5} \approx 2.236$, a sequential search from `0` with `increment = 0.01` makes about _____ guesses. With `increment = 0.0000001` it makes about _____.

Part I • Trace a Bisection Search

TRACE ≈ 6 min

```
guess = (low + high)//2
while high - low > 1:
    if guess == n:
        break
    elif guess < n:
        low = guess
    else:
        high = guess
    guess = (low + high)//2
```

I1. Ali's number is from 1 to 99, so, as in the slides, the search starts one step outside that range, at `low = 0` and `high = 100`. Ali picks `n = 81`. Fill in the table after each wrong guess.

	low	high	guess
at the start	0	100	50
after guess 1	_____	_____	_____
after guess 2	_____	_____	_____
after guess 3	_____	_____	_____

Guesses used in total: _____

A sequential search from 0 would have used: _____

I2. Same code, `low = 0`, `high = 100`. Write the guesses in order.

`n = 37`: _____

`n = 70`: _____

I3. The fast square root from the slides finds $\sqrt{3}$. Fill in the first rows.

```
n = 3
low = 0
high = n
guess = (low + high)/2
while high-low > epsilon:
    if guess**2 < n:
        low = guess
    else:
        high = guess
    guess = (low + high)/2
```

low	high	guess	guess**2 < n?
0	3	1.5	2.25: yes
_____	_____	_____	_____
_____	_____	_____	_____
_____	_____	_____	_____

Part J • How Many Guesses? TRACE ≈ 3 min

Each guess of a bisection search throws away half of what is left.

J1. A **sorted** list holds 8 items. After one guess, _____ items are left; after two, _____; after three, _____.

J2. Fill in the table. "About" is fine.

Numbers to search	Sequential, at worst	Bisection, about
16	_____	_____
1024	_____	_____
1000	_____	_____
1 000 000 (a million)	_____	_____
10 000 000 000	_____	_____

J3. The range to search is made twice as big. A bisection search then needs _____ more guesses; a sequential search needs _____ as many.

Part K • Can Bisection Search Be Used? CLASSIFY ≈ 4 min

K1. Tick one column for each situation.

Situation	Yes	No
(a) Ali picks an integer from 1 to 100 and says "larger" or "smaller".	☐	☐
(b) Ali picks a 3-digit PIN and says only "correct" or "wrong".	☐	☐
(c) Finding $\sqrt{5}$ to within `epsilon`.	☐	☐
(d) A number from 0 to 10 with 4 decimal digits; you are told "too high" or "too low".	☐	☐
(e) Finding a word in a printed dictionary.	☐	☐
(f) Finding a name on an attendance sheet written in the order students came in.	☐	☐
(g) Finding `72` in the sorted list `[12, 30, 45, 72, 88, 91]`.	☐	☐
(h) Guessing a friend's birth month; they only nod when you are right.	☐	☐

K2. Sequential search, bisection search, or both? Tick one column for each.

Statement	Sequential	Bisection	Both
(a) The search range stays fixed.	☐	☐	☐
(b) The search range shrinks after every guess.	☐	☐	☐
(c) It can use a flag to report "not found".	☐	☐	☐
(d) It works on a list that is not sorted.	☐	☐	☐
(e) It needs a "too high / too low" answer.	☐	☐	☐
(f) It may look at every candidate.	☐	☐	☐
(g) It is a guess-and-check method.	☐	☐	☐

Part L • Circle the Bug, Then Fix It FIX ≈ 8 min

Each snippet has exactly one bug. Say what actually happens (the wrong output, the name of the error, or "it never ends"), then write the fix.

L1. Ali really did sell 10 tickets, but this always prints `False`.

```
1 for ali in range(18):
2     found = False
3     for fatima in range(18):
4         if ali+fatima == 17 and ali-fatima == 3:
5             found = True
6 print(found)
```

Circle the misplaced line. Where does it belong?

_____

L2. Two off-by-one bugs.

(a) The slides' `for` version. Input: `1`.

```
1 x = int(input())
2 found = False
3 for guess in range(x):
4     if guess**2 == x:
5         root = guess
6         found = True
7 if found:
8     print("Root:", root)
9 else:
10    print("Not a perfect square")
```

It prints: _____

Fix: _____

(b) Ali's number is between 1 and 100. Input: `100`.

```
1 n = int(input())
2 found = False
3 guess = 0
4 while guess < 100:
5     if guess == n:
6         found = True
7         break
8     guess += 1
9 if found:
10    print("Found", guess)
11 else:
12    print("Not found")
```

It prints: _____

Fix: _____

L3. Input: `150`.

```
1 n = int(input())
2 for guess in range(100):
3     if guess == n:
4         found = True
5         break
6 if found:
7     print("You guessed:", guess)
8 else:
9     print("Not between 0 and 99")
```

Error (and on which line): _____

Fix:

_____

L4. We wanted $\sqrt{5}$.

```
1 n = 5
2 found = False
3 increment = 0.001
4 guess = 0
5 while guess**2 < n:
6     if guess**2 == n:
7         found = True
8         break
9     guess += increment
10 print(found)
```

It prints: _____

Rewrite line 6:

_____

L5. Two broken bisection searches.

(a) `n = 81`, `low = 0`, `high = 100`

```
found = False
guess = (low + high)//2
while high - low > 1:
    if guess == n:
        found = True
        break
    elif guess < n:
        high = guess
    else:
        low = guess
    guess = (low + high)//2
print(found)
```

It prints: _____

Fix: _____

(b) $\sqrt{5}$ with `epsilon = 0.01`

```
n = 5
low = 0
high = n
guess = (low + high)//2
while high-low > epsilon:
    if guess**2 < n:
        low = guess
    else:
        high = guess
    guess = (low + high)//2
print(guess)
```

What happens: _____

Fix: _____

Part M • One Sentence Each

EXPLAIN ≈ 3 min

M1. Sequential search needed 22 million tries to find $\sqrt{5}$; bisection needed 49. Why?

_____

M2. Why must an approximation stop when the guess is "close enough" instead of exactly right?

_____

M3. The slides' first ticket program printed nothing when there was no solution. What did the Boolean flag add?

_____

M4. For a number from 1 to 10, the slides start the search at `low = 0` and `high = 11`, one step *outside* the range. Why not at 1 and 10?

_____

M5. A PIN is a number, and numbers are in order. Why can bisection still not find a PIN when the only answer is "correct" or "wrong"?

_____

Part N • Write the Code

STRETCH ≈ 10 min, optional

Write each program in full. Run it on paper before you call it done.

N1. The slides' You Try: rewrite the perfect-square program with a `for` loop and a flag. Input `25` prints `Square root of 25 is 5`; input `26` prints `26 is not a perfect square`. Make sure `1` and `0` work too.

N2. A farm has chickens and cows: 30 heads and 74 legs in all. Write a guess-and-check with two nested loops that prints how many of each, or `No solution found`.

N3. The slides' last You Try: a bisection search for the cube root of `cube = 27`, stopping when `guess**3` is within `epsilon = 0.01` of `cube`. Start from `low = 0` and `high = cube`.

Console Quests

tonight · 5 min · at a computer

When unsure, just try it in the console. These are deliberately **not** answered anywhere on this sheet — the console is the answer key. Predict first, then type.

1. Type `0.1 + 0.2`, then `10 * 0.1`, then the sum of ten `0.1`s with a loop. One multiplication and ten additions give different answers. Which one is exactly `1.0`, and why might that be?
2. Run the program of Part H with `n = 5`, `increment = 0.001` and `epsilon = 0.01`, and note `tries`. Now make the increment ten times smaller, and then ten times smaller again. Predict `tries` each time before you press Run.
3. Add a counter to the fast square root (I3) and find $\sqrt{5}$ with `epsilon = 0.00000000000001`. Do you get the slides' 49 tries? Then try $\sqrt{5000}$: how many more tries, and why so few?

Stuck on more than two items? Re-read the Lecture 15 slides "Guess and Check", "Square Root with while Loop", "Generating Pairs", "Floats can't be trusted!", "Approximations" and "Sequential Search vs Bisection Search", and the two bisection walk-throughs, before the next class — then redo Parts D, H and I cold. For a search, say aloud *what is the next guess, and when does the loop stop?* For a float, say *close enough, never equal.*

Answer Key

Check your work *after* attempting everything. If an answer surprises you, that item is worth re-reading in the slides. Console Quests have no key — the console is the key.

A1. all possible **solutions**; you can **check** whether one is correct; until you have **tried every possible solution**. Then you stop, found or not.

A2. `increment`; `(low + high) / 2`. Sequential crawls by a fixed step; bisection jumps to the middle of what is left.

A3. a **flag**; set to `False` before the search; changed to `True` when the answer is found. After the loop, `if found:` decides which message to print.

A4. an **order**; too **low** or too **high**. Without that feedback there is no half to throw away.

A5. `epsilon`; more **accurate**; a smaller **increment**. Halve the step and the search takes twice as many guesses.

A6. (a) **False** — the sum is `0.30000000000000004`. (b) **True**. (c) **False** — $\sqrt{5}$ never ends either; we stop when the guess is close enough. (d) **False** — it is halved every time; that is where the speed comes from. (e) **True** — for each of Ali's guesses, all of Fatima's. (f) **True** — the slides' first ticket program did exactly that. (g) **False** — "wrong" does not say which way to move. (h) **False** — `0.125` and `0.25` add up exactly; only some decimals, like `0.1`, cannot be stored.

B1. Left `3`; right `4`. On the right $3^2 = 9$ is still less than 10, so the loop takes one more step and *overshoots* the square root. That is why the slides' program then asks `if guess**2 == x`: 10 is not a perfect square.

B2. Left `False`; right `True`. `0.1` cannot be stored exactly, so the ten small errors add up to `0.9999999999999999`; `0.125` is stored exactly, so that sum lands on `1.25`. You cannot tell which is which by looking, so never use `==` on floats.

B3. `0.30000000000000004`, `False`, `0.8999999999999999`, `True`. What gets printed is what is in memory, and for `0.1`, `0.2` and `0.3` that is not quite the decimal you typed. Halves and quarters are stored exactly.

B4. `5`, after **6** questions (for `guess` 0 to 5). $4 \times 4 = 16$ is not more than 20, so the loop goes on; $5 \times 5 = 25$ is, so `break` leaves the loop *before* line 5 can add one more.

B5. Left `True 5`; right `True 4`. Both find `4`. Without `break` the loop goes on to the end of the range, and the flag stays `True` because nothing sets it back. With `break`, `g` is left at the answer, and the last guess is never tried.

C. `16`: 4, 4 times, `Square root of 16 is 4.`
`15`: 4, 4 times, `15 is not a perfect square` ($4^2 = 16$ overshoots 15).
`1`: 1, once, `Square root of 1 is 1.`
`0`: 0, **never**, `Square root of 0 is 0` — `0 < 0` is `False` at once, and $0^2 = 0$.
`-4`: 0, **never**, `-4 is not a perfect square`. The loop never starts, and the `if` gives the right answer anyway: no integer squared is negative.

D1. `0 0`, `0 1`, `1 0`, `1 1`, `2 0`, `2 1`: **6** lines (3×2), and the 4th is `1 1`. `a` is held fixed while `b` runs through all its values, the way the slides kept Ali fixed and tried every guess for Fatima.

D2. 1: **324** (18×18). 2: **1331** ($11 \times 11 \times 11$): a third person multiplies the work by 11 again. 3: **6** — the inner loop runs 0, 1, 2 and 3 times. 4: **5** — the loops run 25 times, but only `(0, 4)`, `(1, 3)`, `(2, 2)`, `(3, 1)` and `(4, 0)` pass the test.

D3. Line 6: `10 7 188`. Ali's guesses 0 to 9 cost $10 \times 18 = 180$ checks, and Fatima's 0 to 7 eight more. Line 7: `324` — with no `break` every pair is still tried after the answer is found. With the `break`: **314**. The inner loop for `ali = 10` stops after 8 checks, but the outer loop goes on through Ali's guesses 11 to 17, another $7 \times 18 = 126$.

E1. `fatima == ali - 3`; `fatima == 2 * ali`; `sara == hina + 5`; `ali >= 4`; `ali % 2 == 0`; `ali + zain + fatima == 10`. "Fewer" and "more" are easy to turn round: test one case. If Ali sells 10, Fatima sells 7, and `7 == 10 - 3` is `True`.

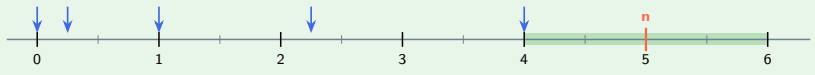
E2. `range(21)` both times; `hamza + sana == 20` and `hamza == sana + 4`. It prints `12 8`. `range(20)` stops at 19, and one of them could hold all 20 marbles: the set of all possible solutions is 0 to 20. (The slides used `range(18)` for 17 tickets for the same reason.)

E3. `No solution found`. With `b = a + 1` and `c = 2b = 2a + 2`, the total is $4a + 3$. That is 12 only for $a = 2.25$, and no integer guess is ever 2.25. This is why the slides went on from integer search to

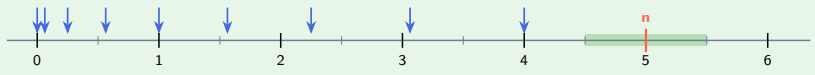
approximation.

F. **False, True, True, False, False, True, False, True, True.**
The ones built only from halves, quarters and eighths (`0.5`, `0.25`, `0.125`, `1.5`) are stored exactly, so their sums and products are exact. The ones with `0.1`, `0.2`, `0.3` or `0.7` carry a small error: `0.3 * 3` is `0.8999999999999999` and `0.1 + 0.7` is `0.7999999999999999`. The last row is the safe way to compare: the two numbers are within a small amount of each other, so it is `True`.

G. The window is `4.9` to `5.1` for `epsilon = 0.1`, and `4.99` to `5.01` for `0.01`.
`epsilon = 0.1`: `2.23`, `2.236` and `2.24` pass. `epsilon = 0.01`: only `2.236`.
The test is on `guess**2`, not on `guess`: we cannot check against $\sqrt{5}$ itself, because that is the number we do not know. A smaller `epsilon` lets fewer guesses through, so the answer is more accurate.

H1.


The window is 4 to 6. The guesses `0`, `0.5`, `1.0`, `1.5`, `2.0` put their squares at `0`, `0.25`, `1.0`, `2.25`, `4.0`, and `4.0` is inside. Prints `Square root of 5 is 2.0, in 4 tries`. With a wide window, a poor answer is "close enough".

H2.


The window is 4.5 to 5.5. Nine guesses, `0` to `2.0`, all land below it. Prints `Couldn't find the square root of 5`. When `guess` becomes `2.25`, the `while` test on line 4, `5.0625 < 5`, is already `False`, so the loop ends before line 5 can look at it. This loop only ever checks guesses whose square is *below* `n`: the slides' "Just missed!".

H3. `Couldn't find the square root of 4`. For the same reason as H2: when `guess` reaches `2.0`, `4.0 < 4` is `False` and the loop ends without checking it. The guesses checked are `0`, `0.5`, `1.0` and `1.5`, with squares `0`, `0.25`, `1.0` and `2.25`, none of them within `0.1` of 4.

H4. About **224** ($2.236 \div 0.01$), and about **22 million** ($2.236 \div 0.0000001$): the slides' 4.5 seconds. The number of guesses is the distance to the answer divided by the step.

I1. After guess 1: 50, 100, 75 (50 was too low). After guess 2: 75, 100, 87 (75 too low). After guess 3: 75, 87, 81 (87 too high, so `high` comes down). The 4th guess is 81: **4** guesses. Sequential search from 0 would have used **82** (0, 1, . . . , 81).

I2. `n = 37`: `50, 25, 37` (3 guesses). `n = 70`: `50, 75, 62, 68, 71, 69, 70` (7 guesses). Each new guess is the middle of what is left; `//` drops the half, so the middle of 50 and 75 is 62.

I3. Row 2: `1.5, 3, 2.25`; `5.0625`: no. Row 3: `1.5, 2.25, 1.875`; `3.515625`: no. Row 4: `1.5, 1.875, 1.6875`; `2.84765625`: yes. A "yes" moves `low` up to the guess, a "no" moves `high` down to it. The guesses close in on 1.732 . . . from both sides, and the loop stops when `high` and `low` are less than `epsilon` apart.

J1. 4, 2, 1. Three halvings take 8 down to 1, because $2^3 = 8$.

J2. Sequential: **16, 1024, 1000, a million, 10 billion** — every number, when the answer is the last one. Bisection: **4, 10, 10, 20, 33**. $2^{10} = 1024$ is about a thousand, so every factor of a thousand adds about 10 guesses. The last row is the slides' 33 steps.

J3. **one** more guess (the first guess cuts it back to the old size); **twice** as many.

K1. **Yes, No, Yes, Yes, Yes, No, Yes, No.** (b) and (h) are in order, but "wrong" never says *which way* to move, so no half can be thrown away. (f) gives no order: the names are not sorted. (d) works, and 0 to 10 in steps of 0.0001 is only 100 001 numbers, so it takes about 17 guesses. In (e) the order is alphabetical.

K2. (a) Sequential. (b) Bisection. (c) Both. (d) Sequential. (e) Bisection. (f) Sequential. (g) Both — each one guesses and checks; they differ only in how they choose the next guess.

L1. `found = False` is *inside* the outer loop, so it is reset on every round of `ali` and wipes out the `True` set at `ali = 10`. Move it **above** the loop, where a flag belongs.

L2. (a) `Not a perfect square`. `range(1)` holds only `0`, so the guess `1` is never tried (and for `0` the loop never runs at all). Fix: `for guess in range(x + 1):`.
(b) `Not found`. The loop stops at 99. Fix: `while guess <= 100:`. Both bugs leave the *last* possible solution out of the search.

L3. `NameError` on line 6. `found` is only ever made when a guess matches; for 150 none does, so line 6 asks about a name that does not exist. Fix: add `found = False` before the loop. A flag needs a value for the "nothing happened" case.

L4. `False`. No float squared is exactly 5, so `==` never holds, and the loop runs on until `guess**2`

passes 5. Fix: set `epsilon = 0.01` and test `if (guess**2 >= n-epsilon) and (guess**2 <= n+epsilon):`. Close enough, not exact.

L5. (a) `False`. A guess that is too *low* moves `high` down, so the search throws away the half that holds the answer: the guesses go 50, 25, 12, 6, 3, 1, and then `high` is 1 and `low` is 0, so the loop stops. Fix: swap lines 8 and 10 (`low = guess` when `guess < n`).
(b) **It never ends.** `//` keeps every guess a whole number. After 2 (too low) and 3 (too high), `low` is 2 and `high` is 3, and the guess is `(2 + 3)//2 = 2` again and again; `high - low` stays 1. Fix: use `/` on lines 4 and 10.

M1. Sequential search steps forward by the same tiny increment over a fixed range, while bisection throws away half of what is left at every guess.

M2. Because most square roots never end and most decimals cannot be stored exactly, so `guess**2 == n` may never be true; we accept any answer within `epsilon`.

M3. A record, kept after the loop, of whether the answer was found, so that the program can print "No solution found" instead of staying silent.

M4. So that the answer is always strictly between `low` and `high`: every guess then lands inside, and the range shrinks every time. Starting at `high = 10`, Ali's 10 would leave `low` stuck at 9 and the guess at 9 forever.

M5. Because "wrong" does not say whether the PIN is higher or lower, so you cannot tell which half to throw away; you are back to trying the PINs one at a time.

N1. `x = int(input())` / `found = False` / `for guess in range(x + 1):` / `if guess**2 == x:` / `root = guess` / `found = True` / then, after the loop, `if found:` print the root, `else:` print that it is not a perfect square. `range(x + 1)` is the fix of L2 (a).

N2. `found = False` / `for chickens in range(31):` / `for cows in range(31):` / `if chickens + cows == 30 and 2*chickens + 4*cows == 74:` / `print(chickens, cows)` / `found = True`; then `if not found: print('No solution found')`. It prints `23 7`.

N3. `guess = (low + high)/2` / `while abs(guess**3 - cube) >= epsilon:` / `if guess**3 < cube: low = guess` / `else: high = guess` / `guess = (low + high)/2`; then `print(guess)`. It stops at about `3.0001` after 14 updates. Only the test changes from the square root: `**3` for `**2`.