Finger Exercises 19

Sorting

≈ 70 min + stretch

Comp 102 — Programming Fundamentals

What this is. A workout for Lecture 19. You will sort cards by hand, run selection sort and the sieve in your head, fix broken copies, count orders and answer patterns, and sort by counting.

How to do it. On **paper**, **no computer**, in one sitting. Stuck for more than a minute? Skip the item and come back. Parts A–P are for *everyone*; Part Q is a **stretch**. The Console Quests are for tonight, *at* a computer. The key is on the last pages: do not peek until you are done.

Predict first, then check. Write your answer *before* you type anything. A wrong guess teaches you more than an answer you only watched the computer give.

Your tool. The bottom of every page has a strip of 2^k (answer patterns of k comparisons) and $n!$ (orders of n items). Use it in Part L.

Part A • Do You Remember?

RECALL ≈ 4 min

A1. Each pass of selection sort finds the _____ item of the rest and swaps it to the _____ of the rest.

A2. For a list of n items, selection sort makes _____ passes and _____ comparisons in all. Its family is _____.

A3. k comparisons can give _____ answer patterns. n different items can be in _____ orders. A plan can *always* sort them only if _____ ≥ _____.

A4. Any sort that *compares* needs about _____ comparisons. Python's `sorted()` is in the family _____.

A5. `sieve(n)` returns a list of length _____. Place `k` of it holds `True` when _____.

A6. Counting sort uses each mark as a _____ in a list. It never _____ two marks.

A7. Circle **T** (true) or **F** (false):

	(T / F)		*(T / F)*
(a) `sorted(L)` changes `L`.	T / F	(e) The sieve divides by smaller primes.	T / F
(b) `L.sort()` returns the sorted list.	T / F	(f) 3 bags need 3 weighings in the worst case.	T / F
(c) Selection sort is faster on a sorted list.	T / F	(g) A faster computer beats the speed limit.	T / F
(d) Equal values sit side by side when sorted.	T / F	(h) Counting sort can sort names.	T / F

Part B • Sorted Lists Make Jobs Easy

TRACE ≈ 4 min

```
1 def has_repeat(L):
2     for i in range(len(L) - 1):
3         if L[i] == L[i + 1]:
4             return True
5     return False
```

`has_repeat` is meant for a **sorted** list: it checks only neighbours.

```
S = sorted([21, 8, 44, 3, 8, 15, 30])
```

Write the value. If the line stops with an error, write the error's name.

Expression	Value or error
`S`	_____
`S[0], S[-1]`	_____
`S[len(S) // 2]`	_____
`S[len(S)]`	_____
`sorted([4, 1, 3, 2])[2]`	_____
`has_repeat(S)`	_____
`has_repeat([21, 8, 44, 3, 8])`	_____
`has_repeat(sorted([21, 8, 44, 3, 8]))`	_____
`has_repeat([]), has_repeat([7])`	_____
`has_repeat([5, 5])`	_____

B1. How many times does line 3 run? `has_repeat(S)`: ____ `has_repeat([1, 2, 3, 4, 5])`: ____

k	0	1	2	3	4	5	6	7	8	9	10
2^k	1	2	4	8	16	32	64	128	256	512	1024

n	1	2	3	4	5	6	7	8
$n!$	1	2	6	24	120	720	5040	40320

Part C • Swap in One Line

TRACE ≈ 3 min

What does each program print? If it stops with an error, write the error's name.

(a)

```
1 x, y = 5, 9
2 x, y = y, x
3 print(x, y)
```

prints _____

(b)

```
1 x, y = 5, 9
2 x = y
3 y = x
4 print(x, y)
```

prints _____

(c)

```
1 L = [5, 2, 8, 1]
2 L[0], L[3] = L[3], L[0]
3 print(L)
```

prints _____

(d)

```
1 L = [5, 2, 8]
2 L[1], L[1] = L[1], L[1]
3 print(L)
```

prints _____

(e)

```
1 a, b, c = 1, 2, 3
2 a, b, c = b, c, a
3 print(a, b, c)
```

prints _____

(f)

```
1 L = [5, 2, 8]
2 L[0], L[3] = L[3], L[0]
3 print(L)
```

prints _____

Part D • Selection Sort by Hand

TRACE ≈ 7 min

Write `L` after each pass of `selection_sort`, one number in each box. Write = beside a row if nothing moved in that pass. Under each list, write how many comparisons the whole sort made.

D1. `[7, 2, 9, 4, 1]`

start	7	2	9	4	1
1					
2					
3					
4					

comparisons: _____

D2. `[5, 4, 3, 2, 1]`

start	5	4	3	2	1
1					
2					
3					
4					

comparisons: _____

D3. `[1, 2, 3, 4]`

start	1	2	3	4
1				
2				
3				

comparisons: _____

D4. `[3, 1, 3, 1]`

start	3	1	3	1
1				
2				
3				

comparisons: _____

D5. How many passes for `selection_sort([42])`? _____ For `selection_sort([])`? _____ Does either crash? _____

Part E • Inside One Pass

TRACE ≈ 4 min

```
1 def selection_sort(L):
2     n = len(L)
3     for i in range(n - 1):
4         smallest = i
5         for j in range(i + 1, n):
6             if L[j] < L[smallest]:
7                 smallest = j
8         L[i], L[smallest] = L[smallest], L[i]
```

E1. `L = [4, 6, 2, 7, 2]`. Trace the **first** pass (`i = 0`), one row for each `j`.

j	L[j]	L[smallest] before	line 6 True?	smallest after
1	___	___	_____	___
2	___	___	_____	___
3	___	___	_____	___
4	___	___	_____	___

After line 8, `L` is _____.

E2. Line 6 now reads `if L[j] <= L[smallest]:`. In the same first pass, `smallest` ends at _____ and `L` becomes _____. Is the final list still sorted? _____

E3. In a list of 5 items, during the pass with `i = 2`, `j` takes the values _____. In a list of 6 items, `i` takes the values _____, and in the last pass `j` takes _____.

k	0	1	2	3	4	5	6	7	8	9	10
2^k	1	2	4	8	16	32	64	128	256	512	1024

n	1	2	3	4	5	6	7	8
$n!$	1	2	6	24	120	720	5040	40320

Part F • Counting the Work

COUNT ≈ 5 min

F1. Fill in the table for `selection_sort`.

n (items)	1	2	3	6	10	100
passes	____	____	____	____	____	____
comparisons	____	____	____	____	____	____

F2. A computer makes 10 million comparisons a second. It sorts 100,000 numbers. Use $\log_2 100{,}000 \approx 17$.

selection sort: _____ comparisons, about _____ seconds

`sorted()`: _____ comparisons, about _____ seconds

F3. Name the family of each job: $O(n)$, $O(n \log n)$ or $O(n^2)$.

(a) `any_repeat(L)` (every pair) _____ (b) `has_repeat(sorted(L))` _____

(c) `has_repeat(L)`, `L` already sorted _____ (d) `my_sort(L)` (Part H) _____

(e) `counting_sort(marks, 10)` _____ (f) `sorted(L)[0]` vs `min(L)` _____

Part G • Bug Hunt

FIX ≈ 8 min

Each version below has **one** change to `selection_sort` in Part E; every other line is the same. Write what it does, then the fix.

G1. Line 4 now reads `smallest = 0`

On `[3, 1, 2]` it gives _____. On `[2, 1]` it gives _____. Fix: _____

G2. Line 5 now reads `for j in range(i + 1, n - 1):`

On `[5, 2, 8, 1, 4]` it gives _____. Which item is never looked at? _____

G3. Line 8 is indented one level more, so the swap happens inside the `j` loop, after every comparison. Tick the lists it still sorts **correctly**:

☐ `[2, 1]` ☐ `[4, 3, 2, 1]` ☐ `[5, 2, 8, 1, 4]`

☐ `[9, 1, 5, 3]` ☐ `[3, 1, 2]`

G4. Line 8 is split into two lines: `L[i] = L[smallest]` then `L[smallest] = L[i]`

On `[5, 2, 8, 1, 4]` it gives _____.

G5. Line 6 now reads `if L[j] > L[smallest]:`

On `[5, 2, 8, 1, 4]` it gives _____. Is it a bug? _____

G6. Tick each change after which the function **still sorts correctly**, on every list:

☐ line 5: `for j in range(i, n):` ☐ line 3: `for i in range(n):`

☐ line 6: `if L[j] <= L[smallest]:` ☐ line 4: `smallest = n - 1`

G7. The function is right, but the caller wrote:

`marks = selection_sort(marks)` then `print(marks)`

It prints _____. Why? _____

k	0	1	2	3	4	5	6	7	8	9	10
2^k	1	2	4	8	16	32	64	128	256	512	1024

n	1	2	3	4	5	6	7	8
$n!$	1	2	6	24	120	720	5040	40320

Part H • Sort with `min()`

TRACE ≈ 4 min

```
1 def my_sort(L):
2     rest = L[:]
3     answer = []
4     while len(rest) > 0:
5         smallest = min(rest)
6         rest.remove(smallest)
7         answer.append(smallest)
8     return answer
```

`remove(x)` takes out the **first** `x` only.

H1. Trace `my_sort([4, 1, 3, 1])`, one row for each pass of the `while` loop.

`smallest`	`rest` after line 6	`answer` after line 7
____	_____	_____
____	_____	_____
____	_____	_____
____	_____	_____

H2. Write what the last line prints after each change.

(a) Line 2 is `rest = L`. Then `A = [4, 1, 3, 1]`, `B = my_sort(A)`, `print(B, A)` _____

(b) Lines 6–7 become `answer.append(rest.remove(smallest))`. `print(my_sort([4, 1, 3, 1]))` _____

(c) Line 5 uses `max`, not `min`. `print(my_sort([4, 1, 3, 1]))` _____

(d) No change: `print(my_sort([]))` _____

Part I • Cross Out by Hand

TRACE ≈ 6 min

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60

Sieve the grid, the way the lecture did. Cross out 1. Then for `p = 2, 3, 5, 7`: if `p` is not crossed, circle it and cross out $2p$, $3p$, $4p$, … up to 60. At the end, circle every number that is left.

I1. For each `p`, the first number it crosses that was **not** crossed before:

`p = 2`: ____ `p = 3`: ____

`p = 5`: ____ `p = 7`: ____

The pattern: _____

I2. `p = 11` would cross 22, 33, 44, 55. Are any of them new? _____ Why?

_____

I3. How many primes up to 60? ____

The largest? ____

I4. How many times is each number crossed out? 30: ____ 42: ____ 49: ____ 59: ____

k	0	1	2	3	4	5	6	7	8	9	10
2^k	1	2	4	8	16	32	64	128	256	512	1024

n	1	2	3	4	5	6	7	8
$n!$	1	2	6	24	120	720	5040	40320

Part J • Inside `sieve`

TRACE ≈ 6 min

```
1 def sieve(n):
2     is_prime = [True] * (n + 1)
3     is_prime[0] = False
4     is_prime[1] = False
5     for p in range(2, n + 1):
6         if is_prime[p]:
7             for m in range(2 * p, n + 1, p):
8                 is_prime[m] = False
9     return is_prime
```

J1. `flags = sieve(12)`. Write the value or the error.

`flags[11]`, `flags[12]`	_____
`flags[13]`	_____
`len(flags)`	_____
`flags.count(True)`	_____
`list(range(2 * 3, 13, 3))`	_____
`list(range(2 * 7, 13, 7))`	_____
`sieve(2)`	_____
`sieve(1)`	_____
`sieve(0)`	_____

J2. Each row changes **one** line of `sieve`. Write the primes that `sieve(10)` now reports (or the error), and whether the function is still right.

	primes reported	*still right?*
(a) line 7: `for m in range(p, n + 1, p):`	_____	_____
(b) line 2: `is_prime = [True] * n`	_____	_____
(c) line 4 deleted	_____	_____
(d) line 6 deleted (every `p` crosses)	_____	_____
(e) line 5: `for p in range(2, n):`	_____	_____

J3. Count the crossings (line 8) in `sieve(20)`:

`p = 2:` ____ `p = 3:` ____ `p = 5:` ____ `p = 7:` ____ `p = 11:` ____ total: _____

Part K • Pretty Average Primes

TRACE ≈ 4 min

```
1 flags = sieve(2000000)
2 T = int(input())
3 for t in range(T):
4     N = int(input())
5     for A in range(2, N + 1):
6         B = 2 * N - A
7         if flags[A] and flags[B]:
8             print(A, B)
9             break
```

K1. For each `N`, write the pairs `(A, B)` the loop tries, and what it prints.

`N`	`(A, B)` tried	prints
4	_____	_____
10	_____	_____
18	_____	_____
2	_____	_____
3	_____	_____

K2. For `N = 7`, `7 7` is a right answer, but the program prints `3 11`. Why? _____

Without line 9, how many lines would `N = 10` print? _____

K3. Why `sieve(2000000)` and not `sieve(1000000)`, when `N` is at most 1,000,000?

_____

k	0	1	2	3	4	5	6	7	8	9	10
2^k	1	2	4	8	16	32	64	128	256	512	1024

n	1	2	3	4	5	6	7	8
$n!$	1	2	6	24	120	720	5040	40320

Part L • Orders and Answer Patterns

COUNT ≈ 6 min

L1. Three bags A, B, C, all different weights. A new plan: **1st** weigh A and B (is A lighter?), **2nd** weigh A and C (is A lighter?). In each box, write every order (lightest first) that gives those answers. Ring the boxes where the plan cannot decide.

yes yes	yes no	no yes	no no

L2. Fill in the table. "Fewest" is the smallest k with $2^k \geq n!$. Use the strip at the bottom of the page.

n (items)	1	2	3	4	5	6
orders $n!$	____	____	____	____	____	____
fewest comparisons	____	____	____	____	____	____
selection sort	____	____	____	____	____	____

L3. Judge each claim: **impossible**, or **the rule allows it**. Show the two numbers you compared.

(a) "My sort always orders 8 items with 15 comparisons." _____

(b) "... with 16 comparisons." ($2^{16} = 65{,}536$) _____

(c) "My sort always orders 10 items with 21 comparisons." _____

($2^{21} = 2{,}097{,}152$ and $10! = 3{,}628{,}800$)

(d) "If the 3 bags are already in order, 2 weighings can show it." _____

Part M • Counting Sort

TRACE ≈ 7 min

```
1 def counting_sort(marks, top):
2     counts = [0] * (top + 1)
3     for m in marks:
4         counts[m] += 1
5     result = []
6     for value in range(top + 1):
7         for k in range(counts[value]):
8             result.append(value)
9     return result
```

M1. `counting_sort([2, 0, 2, 5, 1, 2], 5)`. Fill in `counts` after each mark.

place	0	1	2	3	4	5
start	0	0	0	0	0	0
read 2						
read 0						
read 2						
read 5						
read 1						
read 2						

It returns _____

M2. What does each call return? Write the list or the error.

`counting_sort([], 4)` _____ `counting_sort([3, 3, 3], 3)` _____

`counting_sort([5, 1], 9)` _____ `counting_sort([4], 3)` _____

`counting_sort([0, 0], 0)` _____ `counting_sort([1.5], 3)` _____

`counting_sort(['b', 'a'], 3)` _____ `counting_sort([2, -1], 3)` _____

M3. Two broken copies. Write what each returns.

(a) Line 6 is `for value in range(top):` on `([3, 10, 7, 10], 10)` _____

(b) Line 8 is `result.append(counts[value])` on `([3, 1, 3], 3)` _____

k	0	1	2	3	4	5	6	7	8	9	10
2^k	1	2	4	8	16	32	64	128	256	512	1024

n	1	2	3	4	5	6	7	8
$n!$	1	2	6	24	120	720	5040	40320

M4. Tick the jobs where counting sort is a good fit. For each one you do not tick, write the reason in two or three words.

☐ 500 quiz marks out of 10 _____ ☐ the names of 500 students _____

☐ prices such as 1299.50 _____ ☐ ages of 240 million people _____

☐ 13-digit CNIC numbers _____ ☐ temperatures from −10 to 50 _____

Part N • `sorted()`, `.sort()` and `key=` TRACE ≈ 6 min

Two rules: **capitals come before small letters**, and strings are compared letter by letter from the left (`'1'` comes before `'9'`). When two items have the **same key**, `sorted` keeps them in their old order — also with `reverse=True`.

```
R = [('Ali', 72), ('Sana', 91), ('Omar', 65)]
```

N1. Write the value or the error.

`sorted([5, 2, 8, 1], reverse=True)[0]` _____

`sorted([5, 2, 8, 1])[1:3]` _____

`sorted([2.5, -1, 0])` and `sorted([])` _____

`sorted([10, 9, 100])` and `sorted(['10', '9', '100'])` _____

`sorted([3, 'a'])` _____

`sorted(['b', 'A', 'a', 'B'])` _____

`sorted(['Hub', 'Swat', 'Lahore', 'Dir'], key=len)` _____

`sorted(['bb', 'a', 'cc'], key=len, reverse=True)` _____

`sorted([25, 31, 12, 40], key=lambda x: x % 10)` _____

`sorted(['bilal', 'Ali', 'zara'], key=lambda w: w.lower())` _____

`sorted(R, key=lambda r: r[0])` _____

`sorted(R, key=lambda r: -r[1])` _____

N2. What does each program print? Each one starts with `L = [3, 1, 2]`.

(a) `M = sorted(L); print(L, M)` _____ (b) `L = L.sort(); print(L)` _____

(c) `M = L; M.sort(); print(L)` _____ (d) `print(L.sort()[0])` _____

Part O • Finish the Function FILL IN ≈ 4 min

Fill in every blank. You may call the lecture's functions.

O1.

```
def is_sorted(L):
    for i in range(________):
        if L[i] ____ L[i + 1]:
            return ______
    return ______
```

`True` when every item is ≤ the next one. What should `is_sorted([])`, `is_sorted([5])` and `is_sorted([2, 2])` give? _____

O2. The middle mark of a list of odd length, and the three best marks:

```
def median(marks):
    return sorted(marks)[____________]

def top_three(marks):
    return sorted(marks, ____________)[______]
```

What does `top_three([7, 9])` give? _____

k	0	1	2	3	4	5	6	7	8	9	10
2^k	1	2	4	8	16	32	64	128	256	512	1024

n	1	2	3	4	5	6	7	8
$n!$	1	2	6	24	120	720	5040	40320

O3. All the primes up to `n`, with `sieve`:

```
def primes_to(n):
    flags = sieve(n)
    return [k for k in range(________) if __________]
```

Part P • One Sentence Each

EXPLAIN ≈ 3 min

P1. Why does `has_repeat` only work on a sorted list?

P2. Why can no plan always sort 3 bags with 2 weighings?

P3. Counting sort is $O(n)$, faster than $n \log_2 n$. Why does it not break the speed limit?

Part Q • Write the Code

STRETCH ≈ 10 min, optional

Q1. Write `twin_primes(n)`: a list of every pair `(p, p + 2)` where both are prime and `p + 2` $\leq$ `n`. Call `sieve` only once. `twin_primes(20)` is `[(3, 5), (5, 7), (11, 13), (17, 19)]`.

Q2. Write `sort_temps(temps)` for whole temperatures from −10 to 50, by counting. Hint: a temperature `t` uses place `t + 10`.
`sort_temps([3, -2, 3, -10, 50])` is `[-10, -2, 3, 3, 50]`.

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. Start from `lec19.py`.

1. Run `race(10000)`, then `race(20000)`. Did `selection_sort`'s time go up about 4 times? And `sorted`'s?
2. Type `for n in [1000, 2000, 4000]: print(n, crossings(n))`. Does the count roughly double each time?
3. Type `sorted('pear')`. What did the string become? Then try `sorted([('Omar', 65), ('Ali', 72)])` with no `key=`: what does Python compare first?

Stuck on more than two items? Re-read the Lecture 19 slides "Selection Sort in Python", "The Sieve in Python", "6 Orders, 4 Patterns", "Counting Sort in Python" and "Sort by Something Else: `key=`" before the next class — then redo Parts D, G and M cold.

k	0	1	2	3	4	5	6	7	8	9	10
2^k	1	2	4	8	16	32	64	128	256	512	1024

n	1	2	3	4	5	6	7	8
$n!$	1	2	6	24	120	720	5040	40320

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. **smallest**; **front**. After pass p, the first p places are done (the green cards).

A2. $\mathbf{n-1}$; $\mathbf{n(n-1)/2}$; $\mathbf{O(n^2)}$ (the handshakes of Lecture 18).

A3. $\mathbf{2^k}$; $\mathbf{n!}$; $2^k \geq n!$: each order needs its own pattern.

A4. $\mathbf{n \log_2 n}$; $\mathbf{O(n \log n)}$.

A5. $\mathbf{n+1}$ (places 0 to n); **k is prime**.

A6. **place** (an index); **compares**.

A7. (a) **False** — it makes a *new* list. (b) **False** — it sorts `L` in place and returns `None`. (c) **False** — it still makes all $n(n-1)/2$ comparisons; it never notices (see D3). (d) **True** — that is why `has_repeat` checks only neighbours. (e) **False** — it never divides: it counts up in steps of p. (f) **True** — 6 orders, but 2 weighings give only 4 answer patterns; 3 give 8. (g) **False** — the limit counts *comparisons*, not seconds. (h) **False** — `counts['Ali']` is a `TypeError`: a name cannot be a place.

B. `[3, 8, 8, 15, 21, 30, 44]`; `3` (the smallest); `44` (the largest); `15` (the middle, place 3 of 7); **IndexError** (places stop at `len(S) - 1`); `3`; `True` (the two 8s are neighbours); `False` — **wrong**! The list is not sorted, so the two 8s are not side by side; `True` once it is sorted; `False` and `False`: `range(-1)` and `range(0)` are empty, so the loop never runs and there is no error; `True`.

B1. **2**: 3 vs 8, then 8 vs 8 returns `True`. **4**: with no repeat it checks all $n-1$ neighbours. That is $O(n)$, after the list is sorted.

C. (a) `9 5`. (b) `9 9`: line 3 copies the *new* `x`; the 5 is lost. That is why Lecture 2 needed `temp`. (c) `[1, 2, 8, 5]`. (d) `[5, 2, 8]`: a place swapped with itself does not change. Selection sort does this when the smallest is already in front ("stays"). (e) `2 3 1`: the right side is made first, as the tuple `(2, 3, 1)`. (f) **IndexError**: there is no `L[3]` in a list of 3.

D. D1: `[1, 2, 9, 4, 7]`; `[1, 2, 9, 4, 7]` = (2 stays); `[1, 2, 4, 9, 7]`; `[1, 2, 4, 7, 9]`. **10** comparisons. D2: `[1, 4, 3, 2, 5]`; `[1, 2, 3, 4, 5]`; = ; = . Sorted after only 2 passes, but it does all 4 anyway: **10**. D3: = , = , = . Still **6** $(3+2+1)$: it never notices the list is sorted. D4: `[1, 3, 3, 1]` (the *first* 1 is picked, because `<` is false for an equal value); `[1, 1, 3, 3]`; = . **6**. D5: **0** and **0**: `range(0)` and `range(-1)` are empty. **No** crash.

E1. j = 1: 6, 4, False, 0. j = 2: 2, 4, True, 2. j = 3: 7, 2, False, 2. j = 4: 2, 2, **False**, 2 (an equal value is not smaller). After the swap: `[2, 6, 4, 7, 2]`.

E2. **4** (the *last* 2); `[2, 6, 2, 7, 4]`. **Yes**: `[2, 2, 4, 6, 7]`. Picking a different 2 does not matter for numbers.

E3. **3, 4**. **0, 1, 2, 3, 4** (that is `range(5)`); only **5**: one comparison.

F1. Passes: **0, 1, 2, 5, 9, 99** $(n-1)$. Comparisons: **0, 1, 3, 15, 45, 4950** $(n(n-1)/2)$.

F2. 100,000 × 99,999/2 ≈ **5 billion**, so about **500 s** (over 8 minutes). 100,000 × 17 = **1.7 million**, so about **0.17 s**. Same computer: the method made the difference.

F3. (a) $O(n^2)$. (b) $O(n \log n)$: the sort costs more than the $n-1$ checks. (c) $O(n)$. (d) $O(n^2)$: `min` hides a loop, $4+3+2+1$ for 5 items. (e) $O(n)$ (`top` is fixed). (f) $O(n \log n)$ vs $O(n)$: to find only the smallest, do not sort.

G1. `[3, 1, 2]` — the list comes back unsorted: pass 2 swaps the 1 back out of the front. `[1, 2]`: right, by luck (only one pass). Fix: `smallest = i`. Test with 3 or more items.

G2. `[1, 2, 5, 8, 4]`. The **last** item: `j` stops at `n - 2`. Fix: `range(i + 1, n)`.

G3. Right: `[2, 1]`, `[4, 3, 2, 1]`, `[9, 1, 5, 3]`. Wrong: `[5, 2, 8, 1, 4]` gives `[4, 1, 2, 5, 8]` and `[3, 1, 2]` gives `[2, 1, 3]`. Lucky inputs hide this bug, so test more than one list. Fix: line 8 back under `for j`, at the level of `smallest = i`.

G4. `[1, 1, 1, 1, 4]`: the first line overwrites `L[i]`, so the second copies it straight back (C, part b). Values are lost. Fix: the one-line swap.

G5. `[8, 5, 4, 2, 1]`: largest first. A bug only if you wanted smallest first; it is the same as `reverse=True`. (The name `smallest` is now wrong, though.)

G6. The first three still sort. `range(i, n)` adds one useless comparison of `L[i]` with itself (14 instead of

k	0	1	2	3	4	5	6	7	8	9	10
2^k	1	2	4	8	16	32	64	128	256	512	1024

n	1	2	3	4	5	6	7	8
$n!$	1	2	6	24	120	720	5040	40320

10 for 5 items); `range(n)` adds a last pass with no `j` at all; `<=` picks the last of equal values (E2). `smallest = n - 1` is a bug: `L[i]` itself is never a candidate, so `[1, 2, 3]` becomes `[2, 3, 1]`. Not every change is a bug — but each one must be checked.

G7. `None`: `selection_sort` changes the list and returns nothing, like `L.sort()`. The name `marks` now points to `None`. Fix: just `selection_sort(marks)`.

H1. 1, `[4, 3, 1]`, `[1]`. 1, `[4, 3]`, `[1, 1]`. 3, `[4]`, `[1, 1, 3]`. 4, `[]`, `[1, 1, 3, 4]`.

H2. (a) `[1, 1, 3, 4] []`: `rest` is another name for `A` (Lecture 12), so emptying `rest` empties the caller's list. That is why line 2 copies. (b) `[None, None, None, None]`: `remove` returns `None`, like `sort`. (c) `[4, 3, 1, 1]`: largest first. (d) `[]`: the `while` never runs.

I1. **4, 9, 25, 49**: always $p \times p$. A smaller multiple such as 3×5 already lost to 3.

I2. **No**: each has a smaller factor (2, 3, 2, 5). The first new one would be $11 \times 11 = 121$, past 60.

I3. **17** (2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59). The largest is **59**.

I4. 30: **3** (by 2, 3, 5). 42: **3** (by 2, 3, 7). 49: **1** (by 7). 59: **0** — a prime is never crossed, so it costs no work at all.

J1. `True`, `False`; **IndexError** (places 0 to 12 only); `13`; `5` (2, 3, 5, 7, 11); `[6, 9, 12]`; `[]` (14 is past 12, so `p = 7` crosses nothing); `[False, False, True]`; `[False, False]` (the `for` loop is `range(2, 2)`: empty); **IndexError** at line 4: a list of length 1 has no place 1.

J2. (a) **none**: every `p` crosses out *itself* first. (b) **IndexError**: `m` reaches 10, but the places stop at 9. (c) `1, 2, 3, 5, 7`: 1 is reported as prime. (d) `2, 3, 5, 7` — still right, only slower: `p = 4` crosses 8 again, and so on (27 crossings instead of 18 for `sieve(20)`). (e) `2, 3, 5, 7` — still right: `p = n` would start at $2n$, past the end.

J3. **9** (4 to 20), **5** (6 to 18), **3** (10, 15, 20), **1** (14), **0**: total **18**, well under $3n = 60$. Each row is about n/p.

K1. 4: (2, 6), (3, 5) → `3 5`. 10: (2, 18), (3, 17) → `3 17`. 18: (2, 34), (3, 33), (4, 32), (5, 31) → `5 31`. 2: (2, 2) → `2 2` ($A = B$ is allowed). 3: (2, 4), (3, 3) → `3 3`.

K2. It **stops at the first pair** (`break`), and `A = 3` comes before `A = 7`. Without `break`: **2** lines, `3 17` and `7 13`.

K3. `B = 2 * N - A` can be almost 2,000,000, and `flags[B]` needs that place. With `sieve(1000000)` a big `N` gives **IndexError**.

L1. yes yes: **ABC** or **ACB** (ring). yes no: **CAB** only. no yes: **BAC** only. no no: **BCA** or **CBA** (ring). This plan also fails 2 times out of 4: **B** and **C** are never weighed against each other. Any plan must fail: 6 orders, 4 boxes.

L2. Orders: **1, 2, 6, 24, 120, 720**. Fewest: **0, 1, 3, 5, 7, 10** ($2^{10} = 1024 \geq 720$, but $2^9 = 512$ is not). Selection sort: **0, 1, 3, 6, 10, 15**. One item needs no comparison at all.

L3. (a) **Impossible**: $2^{15} = 32{,}768 < 8! = 40{,}320$. (b) **The rule allows it**: $65{,}536 \geq 40{,}320$. (c) **Impossible**: $2{,}097{,}152 < 3{,}628{,}800$; 22 is the fewest. (d) **Allowed**: yes, yes ends it. That is a *lucky* case; the limit is about *always* (the worst case).

M1. `[0, 0, 1, 0, 0, 0]`, `[1, 0, 1, 0, 0, 0]`, `[1, 0, 2, 0, 0, 0]`, `[1, 0, 2, 0, 0, 1]`, `[1, 1, 2, 0, 0, 1]`, `[1, 1, 3, 0, 0, 1]`. It returns `[0, 1, 2, 2, 2, 5]`. Places 3 and 4 stay 0, so nothing is appended for them.

M2. `[]`; `[3, 3, 3]`; `[1, 5]` (a `top` bigger than needed is fine, only slower); **IndexError** (4 is past `top`: test the top value!); `[0, 0]`; **TypeError** (a place must be a whole number); **TypeError**; `[2, 3]` — **no error, but wrong**: `counts[-1]` is the *last* counter, place 3, so -1 comes back as 3.

M3. (a) `[3, 7]`: the two 10s vanish, because the top value's counter is never read. (b) `[1, 2, 2]`: it appends how *many*, not the value.

M4. Good: **quiz marks**; **ages** (0 to 120: a huge n with a small `top` is the best case). Not good: names (not a place: `TypeError`); prices (not whole numbers); CNIC (10,000,000,000,000 counters); temperatures (negative places go wrong, as in M2 — unless you first add 10 to each, see Q2).

N1. `8`; `[2, 5]`; `[-1, 0, 2.5]` and `[]`; `[9, 10, 100]` but `['10', '100', '9']` (strings: `1` before `9`, like a dictionary); **TypeError** (a number and a string cannot be compared); `['A', 'B', 'a', 'b']`; `['Hub', 'Dir', 'Swat', 'Lahore']` (Hub and Dir tie at 3 and keep their order); `['bb', 'cc', 'a']`; `[40, 31, 12, 25]` (by the last digit: 0, 1, 2, 5); `['Ali', 'bilal', 'zara']` (capitals no longer jump ahead); `[('Ali', 72), ('Omar', 65), ('Sana', 91)]` (by name); `[('Sana', 91), ('Ali', 72), ('Omar', 65)]`: the merit list, the same as `key=lambda r: r[1], reverse=True`.

k	0	1	2	3	4	5	6	7	8	9	10
2^k	1	2	4	8	16	32	64	128	256	512	1024

n	1	2	3	4	5	6	7	8
$n!$	1	2	6	24	120	720	5040	40320

N2. (a) `[3, 1, 2] [1, 2, 3]`: `sorted` leaves `L` alone. (b) `None`: the sorted list is lost. (c) `[1, 2, 3]`: `M` is another name for the same list. (d) **TypeError**: `None` has no place 0.

O1. `len(L) - 1`; `>`; `False`; `True`. All three give `True`. With `>=` instead of `>`, `[2, 2]` would wrongly be "not sorted".

O2. `len(marks) // 2`. `reverse=True` and `:3`. `[9, 7]`: a slice past the end does not crash, it just gives what there is.

O3. `n + 1` (or `len(flags)`); `flags[k]`. `primes_to(12)` is `[2, 3, 5, 7, 11]`.

P1. It checks only neighbours, and only in a sorted list are equal values sure to be neighbours.

P2. There are 6 orders but only $2 \times 2 = 4$ answer patterns, so two orders must share a pattern and the plan cannot tell them apart.

P3. The limit is for sorts that *compare* (2 answers per question). Counting sort never compares: `counts[m] += 1` asks "what is `m`?", which has many answers.

Q1. `flags = sieve(n)`, then a loop `for p in range(2, n - 1):` that appends `(p, p + 2)` when `flags[p] and flags[p + 2]`. Each check is one look: no dividing.

Q2. Like `counting_sort` with 61 counters: `counts = [0] * 61`, then `counts[t + 10] += 1`, and when reading place `value`, append `value - 10`. Shift in, shift back out.

k	0	1	2	3	4	5	6	7	8	9	10
2^k	1	2	4	8	16	32	64	128	256	512	1024

n	1	2	3	4	5	6	7	8
$n!$	1	2	6	24	120	720	5040	40320