Finger Exercises 18

Counting Steps, Big-O

≈ 60 min + stretch

Comp 102 — Programming Fundamentals

What this is. A workout for what we just covered in class. Most of the code here is new, so you have to *work it out*, not remember it. For every piece of code, ask: *when the data doubles, what happens to the steps?*

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.

The counting rules from class. One simple line is **one step**, every time it runs. A loop line runs once for each pass. The `def` line is not a step. A line with a **hidden loop** (such as `if x in L:`) costs n steps each time it runs. Everywhere, n is the size of the data: `len(L)`, or the number `n` passed in.

Part A • Do You Remember?

RECALL ≈ 4 min

A1. One simple line is _____ step, every time it _____.

A2. The two rules of Big-O: 1. keep the _____ part. 2. drop the _____.

A3. Write the five families in order, from the one that grows *slowest* to the one that grows *fastest*:

_____ < _____ < _____ < _____ < _____

A4. $\log_2 n$ is the number of times you can _____ n before you reach _____.

A5. Circle **True** or **False**:

(a) A stopwatch gives the same time for the same code on every computer. True / False

(b) $O(n)$ code is faster than $O(n^2)$ code for *every* size of data, even 5 items. True / False

(c) Big-O plans for the worst case. True / False

(d) `x in d`, for a dictionary `d`, checks the keys one by one. True / False

(e) Bisection search works on any list, sorted or not. True / False

(f) When the data doubles, $O(\log n)$ code takes twice as many steps. True / False

(g) On a 1,000 times faster computer, an $O(n)$ program handles 1,000 times more data in the same time. True / False

(h) The line `for x in L:` hides a search of all of `L`. True / False

Part B • Count the Runs

TRACE ≈ 8 min

Beside each line, write how many times it runs. Then add them up, put in the number, and give the Big-O.

B1. `count_big` counts the items bigger than 10. Count the **worst case**: every item is bigger than 10.

```
1 def count_big(L):          # runs
2     c = 0                  # -------
3     for x in L:            # -------
4         if x > 10:         # -------
5             c = c + 1      # -------
6     return c               # -------
```

Steps = _____

$n = 1{,}000$: _____ steps

Best case (no item is bigger than 10): _____ steps

Big-O: _____

B2. `stars` prints a small block of stars.

```
1 def stars(n):                # runs
2     for i in range(n):       # _______
3         for j in range(5):   # _______
4             print('*')       # _______
```

Steps = _____

For $n = 10$: _____ steps

Big-O: _____

B3. `pairs` has a loop inside a loop, and then one more loop.

```
1 def pairs(n):                    # runs
2     count = 0                    # _______
3     for i in range(n):           # _______
4         for j in range(n):       # _______
5             count = count + 1    # _______
6     for k in range(n):           # _______
7         count = count + 1        # _______
8     return count                 # _______
```

Steps = _____

For $n = 10$: _____ steps

For $n = 20$: _____ steps

How many times more? _____

Big-O: _____

Part C • Keep the Biggest Part

TRANSLATE ≈ 4 min

Write the Big-O of each step count.

Steps	Big-O	Steps	Big-O
$4n + 9$	_____	$n^2/2$	_____
$3n^2 + 50n + 7$	_____	$2^n + n^2$	_____
1000	_____	$10 \log_2 n + 3$	_____
$n + n + n$	_____	$6 + n/2$	_____
$n(n + 1)$	_____	$5 \log_2 n + n$	_____

C1. Ali's code takes $500n$ steps. Sana's code takes n^2 steps.

(a) For 100 items: Ali _____ steps, Sana _____ steps. Whose is faster? _____

(b) For 1,000 items: Ali _____ steps, Sana _____ steps. Whose is faster? _____

(c) From what size on does Sana's code lose for good? $n =$ _____

Part D • The Data Doubles

TRANSLATE ≈ 5 min

Fill in what happens to the number of **steps**.

Family	data ×2 ⇒ steps . . .	data ×4 ⇒ steps . . .
$O(1)$	_____	_____
$O(\log n)$	_____	_____
$O(n)$	_____	_____
$O(n^2)$	_____	_____

D1. Each program below was timed once. Predict the new time.

(a) An $O(n)$ program takes 2 s on 1 million items. On 2 million: _____ On 10 million: _____

(b) An $O(n^2)$ program takes 3 s on 1,000 items. On 2,000: _____ On 10,000: _____

(c) A bisection search needs 20 looks on 1 million sorted items. On 2 million: _____ On 4 million: _____

(d) An $O(1)$ lookup takes 0.001 s with 1,000 keys. With 1 million keys: _____

D2. Exponential is different. A program that tries every on/off pattern of n switches takes 1 s for 30 switches. For 31 switches: _____ For 40 switches: _____

Part E • Name the Family

CLASSIFY ≈ 7 min

Write the Big-O of each function. `L` is a list of n items, and `d` is a dictionary.

(a)

```
def ends(L):
    return L[0] + L[-1]
```

Big-O: _____

(b)

```
def hello(n):
    for i in range(100):
        print('hello')
```

Big-O: _____

(c)

```
def two_passes(L):
    s = 0
    for x in L:
        s = s + x
    for x in L:
        s = s - x
    return s
```

Big-O: _____

(d)

```
def top_count(L):
    c = 0
    for x in L:
        if x == max(L):
            c = c + 1
    return c
```

Big-O: _____

(e)

```
def halve(n):
    count = 0
    while n > 1:
        n = n // 2
        count = count + 1
    return count
```

Big-O: _____

(f)

```
def countdown(n):
    while n > 0:
        print(n)
        n = n - 2
```

Big-O: _____

(g)

```
def grid(n):
    for i in range(n):
        for j in range(n):
            print(i, j)
```

Big-O: _____

(h)

```
def all_patterns(n):
    for p in range(2 ** n):
        print(p)
```

Big-O: _____

(i)

```
def in_both(L, d):
    c = 0
    for x in L:
        if x in d:
            c = c + 1
    return c
```

Big-O: _____

(j)

```
def backups(L):
    saved = []
    for x in L:
        saved.append(L.copy())
    return saved
```

Big-O: _____

Part F • One Step or n Steps? CLASSIFY ≈ 4 min

`L` is a list of n items and `d` is a dictionary with n keys. Each line runs **once**. Circle **1** if it costs one step, or ***n*** if it has a hidden loop.

Line			Line			Line		
`len(L)`	1	n	`L.index(7)`	1	n	`d['ali']`	1	n
`L[-1]`	1	n	`sum(L)`	1	n	`d['ali'] = 5`	1	n
`L.append(7)`	1	n	`L.copy()`	1	n	`'ali' in d`	1	n
`7 in L`	1	n	`L.remove(7)`	1	n	`L[3] = 0`	1	n
`max(L)`	1	n	`min(L)`	1	n	`len(d)`	1	n

F1. The word `in` does two different jobs: `for x in L:` and `if x in L:`. What does `in` do in each? Which one is a hidden loop?

`for x in L:` _____

`if x in L:` _____

F2. `A` and `B` are lists of n names each.

```
def count_in(A, B):
    c = 0
    for x in A:
        if x in B:
            c = c + 1
    return c
```

Line 4 runs _____ times.

Each run costs _____ steps.

So line 4 alone costs _____ steps.

Big-O of `count_in`: _____

Part G • Best Case, Worst Case

TRACE ≈ 4 min

```
1 def find(L, x):
2     for item in L:
3         if item == x:
4             return True
5     return False
```

Count the steps of each call, with the rules from class. `L = [5, 2, 8, 6]`.

Call	Steps
`find(L, 5)`	_____
`find(L, 8)`	_____
`find(L, 6)`	_____
`find(L, 7)`	_____

G1. For a list of n items, the best case takes _____ steps and the worst case takes _____ steps. Which one does Big-O use, and so what is the Big-O of `find`? _____

G2. `L` has 1,000 items and `x` is not in it. How many steps? _____

Part H • Halving: Bisection Search

TRACE ≈ 5 min

```
1 def find_sorted(L, x):
2     low = 0
3     high = len(L) - 1
4     while low <= high:
5         mid = (low + high) // 2
6         if L[mid] == x:
7             return True
8         elif L[mid] < x:
9             low = mid + 1
10        else:
11            high = mid - 1
12    return False
```

`S = [2, 5, 9, 14, 20, 27, 31]`
Each look is one pass of the loop. Fill in one row per look.

`find_sorted(S, 27)`

low	high	mid	S[mid]

`find_sorted(S, 4)`

low	high	mid	S[mid]

H1. Use the `halve` function of Part E, part (e). `halve(8)` _____ `halve(10)` _____ `halve(1)` _____ `halve(1024)` _____

H2. About how many looks does bisection need, at most? Hint: $2^{10} = 1{,}024 \approx 1{,}000$.
1,024 items: _____ 2,048 items: _____ 1 million: _____ 1 billion: _____

H3. The list is now **not** sorted: `U = [9, 2, 31, 5]`.
What does `find_sorted(U, 9)` return? _____
Why? _____

Part I • How Long Will It Take?

TRANSLATE ≈ 4 min

A computer does **10 million** (10^7) steps a second. Write the time in the unit that makes sense (seconds, minutes, hours, years). Helpful: 1 hour = 3,600 s; 1 day ≈ 86,400 s; $2^{10} \approx 1{,}000$.

Family	Size	Steps	Time
$O(\log n)$	$n = 1$ billion	about 30	_____
$O(n)$	$n = 1$ million	_____	_____
$O(n^2)$	$n = 100{,}000$	_____	_____
$O(n^2)$	$n = 1$ million	_____	_____
$O(2^n)$	$n = 40$	_____	_____

Part J • Count the Patterns

TRANSLATE ≈ 3 min

J1. A bike lock has 3 dials, each 0 to 9. How many codes? _____ With 4 dials? _____ One more dial multiplies the codes by _____.

J2. A row of 5 light switches, each on or off. How many patterns? _____ With 6 switches? _____

J3. A password is 4 small letters, a to z. How many passwords? _____ A 5th letter multiplies this by _____.

J4. A program tries every 7-digit PIN, 10 million a second. How long? _____ Every 9-digit PIN? _____

Part K • Build an Index by Hand

TRACE ≈ 3 min

```
1 def build_index(pages):
2     index = {}
3     for num, text in pages.items():
4         for word in text.split():
5             if word in index:
6                 index[word].append(num)
7             else:
8                 index[word] = [num]
9     return index
10
11 pages = {1: 'cat sat', 2: 'cat ran',
12          3: 'dog sat down'}
13 index = build_index(pages)
```

This is the slide's `build_index`, on three new pages. Write the whole dictionary `index` it returns, in the order the keys were added.

_____

_____

K1. `index['sat']` _____ `index['dog']` _____ `'cat' in index` _____ `index['bird']` _____

K2. Building the index reads every word of every page once. After that, what is the Big-O of one search such as `index['sat']`? _____

Part L • Make It Fast with a Dictionary

FILL IN ≈ 6 min

Each function is almost done. Fill in every blank.

L1. Is any value in `L` twice? `has_repeat([4, 1, 7, 1])` returns `True`. The `mystery` function from class did this in $O(n^2)$; this one is $O(n)$.

```
def has_repeat(L):
    seen = ________
    for x in L:
        if _________________:
            return True
        _________________ = True
    return False
```

L2. Part F's `count_in`, made fast: put `B` in a dictionary first.

```
def count_in_fast(A, B):
    in_B = {}
    for x in _________:
        in_B[x] = True
    c = 0
    for x in A:
        if _________________:
            c = c + 1
    return c
```

Big-O: _____

L3. `unique([3, 1, 3, 2, 1])` returns `[3, 1, 2]`. It works, but one line is slow.

```
def unique(L):
    result = []
    for x in L:
        if x not in result:
            result.append(x)
    return result
```

Slow line: _____ Big-O: _____

```
def unique_fast(L):
    seen = {}
    result = []
    for x in L:
        if _______________:
            seen[x] = True
            result.append(x)
    return result
```

Part M • One Sentence Each

EXPLAIN ≈ 3 min

M1. Why do we count steps instead of timing the code with a stopwatch?

M2. $2n$ and $100n$ are both $O(n)$. Why is it fine to drop the number in front?

M3. Why does a 1,000 times faster computer barely help an $O(2^n)$ program?

Part N • Write the Code

STRETCH ≈ 10 min, optional

Write each function in full, then give its Big-O.

N1. `pair_sum(L, t)` returns `True` if two items at *different* positions of `L` add up to `t`. `pair_sum([3, 8, 1, 6], 9)` is `True` (3 + 6, or 8 + 1); `pair_sum([3, 8], 6)` is `False` (3 + 3 uses one item twice). Use a loop inside a loop.

N2. Now write `pair_sum_fast(L, t)` in $O(n)$. Hint: walk through `L` once, and keep a dictionary of the items you have already seen. For each `x`, which number would you need to have seen already?

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. Copy `pairs` from Part B, and add `print(count)` before the `return`. Call it with 100, 200 and 400. Does the count go up about 4 times each time?
2. Type `L = list(range(10_000_000))`, then `-1 in L`. Now type `L[-1]`. Which one makes you wait? Why?
3. Copy `halve` from Part E. Predict `halve(1_000_000)` and `halve(2 ** 30)`, then check. Was the first one 20?

Stuck on more than two items? Re-read the Lecture 18 slides "What Is One Step?", "Counting the Steps of `total(L)`", "Keep the Biggest Part", "Big-O: The Name of the Family", "Hidden Loops" and "Five Families, One Picture" before the next class — then redo Parts B, E and F cold. For every piece of code, ask: *when the data doubles, what happens to the steps?*

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. **one**; **runs**. A line inside a loop runs once for each pass, so it counts once for each pass.

A2. 1. keep the **biggest** (fastest-growing) part. 2. drop the **number in front**.

A3. $O(1) < O(\log n) < O(n) < O(n^2) < O(2^n)$: constant, logarithmic, linear, quadratic, exponential.

A4. **halve**; **1**. So $\log_2 16 = 4$: $16 \to 8 \to 4 \to 2 \to 1$.

A5. (a) **False** — the time changes with the computer and with what else it is running. The *steps* stay the same. (b) **False** — the number in front can win for small data: at $n = 5$, $100n = 500$ steps but $n^2 = 25$. Big-O is about *big* data. (c) **True**. (d) **False** — a dictionary jumps straight to the key's shelf: one step. (e) **False** — only a *sorted* list. On an unsorted list it throws away the wrong half (see H3). (f) **False** — it takes just **one** more step (one more halving). (g) **True** — for $O(n)$, 1,000 times the speed buys 1,000 times the data. For $O(n^2)$ it buys only about 32 times, for $O(2^n)$ about 10 more items. (h) **False** — here `in` only hands out the next item: one step a pass. It is `if x in L:` that searches.

B1. Runs: 1, n, n, n, 1. Steps $= 3n + 2$; for 1,000 items, **3,002**. Best case: line 5 never runs, so $2n + 2 =$ **2,002**. Both are $O(n)$: the family does not change, only the number in front.

B2. Runs: n, $5n$, $5n$. Steps $= 11n$; for $n = 10$, **110**. Big-O: $\boldsymbol{O(n)}$, not $O(n^2)$: the inner loop always runs 5 times, whatever n is. A loop inside a loop is $O(n^2)$ only when *both* loops grow with n.

B3. Runs: 1, n, n^2, n^2, n, n, 1. Steps $= 2n^2 + 3n + 2$. For $n = 10$: $200 + 30 + 2 =$ **232**. For $n = 20$: $800 + 60 + 2 =$ **862**, about **4** times more (3.7 exactly; the small parts pull it under 4, and less so as n grows). Big-O: $\boldsymbol{O(n^2)}$: the loop after the nested loops adds only n.

C. Left column: $O(n)$; $O(n^2)$; $O(1)$ (a fixed number, however big, is constant); $O(n)$ (it is $3n$); $O(n^2)$ (it is $n^2 + n$). Right column: $O(n^2)$ (dividing by 2 is a number in front); $O(2^n)$ (exponential beats quadratic); $O(\log n)$; $O(n)$; $O(n)$ (n grows faster than $\log_2 n$).

C1. (a) Ali 50,000, Sana 10,000: **Sana's** is faster. (b) Ali 500,000, Sana 1,000,000: **Ali's** is faster. (c) They meet at $n = 500$ ($500 \times 500 = 250{,}000$ both). After that n^2 wins (loses!) forever. The number in front only moves the meeting point.

D. $O(1)$: the same; the same. $O(\log n)$: +1; +2 (two more halvings). $O(n)$: ×2; ×4. $O(n^2)$: ×4; ×16 (4^2).

D1. (a) **4 s**; **20 s** (10 times the data, 10 times the time). (b) **12 s** (×4); **300 s**, 5 minutes (10 times the data, $10^2 = 100$ times the time). (c) **21**; **22** looks. (d) **0.001 s** — constant does not care about the size.

D2. **2 s** — one more switch doubles the patterns. **1,024 s**, about 17 minutes: 10 more switches is ×2 ten times, $2^{10} = 1{,}024$. Adding *one* item multiplies the steps.

E. (a) $O(1)$ — two index lookups, whatever the size. (b) $O(1)$ — 100 passes, but 100 does not grow with n. (c) $O(n)$ — two loops one *after* the other: $2n$, not n^2. (d) $O(n^2)$ — `max(L)` is a hidden loop, run once on every pass. (e) $O(\log n)$ — `n` is halved each pass. (f) $O(n)$ — `n` goes down by 2 each pass: about $n/2$ passes. Taking away is linear; halving is logarithmic. (g) $O(n^2)$. (h) $O(2^n)$. (i) $O(n)$ — `x in d` is one step for a dictionary. (j) $O(n^2)$ — `L.copy()` copies all n items, and it runs n times.

F. One step: `len(L)`, `L[-1]`, `L.append(7)`, `d['ali']`, `d['ali'] = 5`, `'ali' in d`, `L[3] = 0`, `len(d)`. n steps: `7 in L`, `max(L)`, `L.index(7)`, `sum(L)`, `L.copy()`, `L.remove(7)`, `min(L)`. Python keeps the length written down and can jump to any position or key; the others must look at every item.

F1. In `for x in L:` the `in` only hands out the next item, one step per pass. In `if x in L:` it asks a question, "is `x` somewhere in `L`?", and to answer it Python checks the items one by one: that is the **hidden loop**, n steps each time.

F2. n times; n steps each; $n \times n = n^2$ steps; $\boldsymbol{O(n^2)}$. Only one loop is written, but line 4 hides the second one. (Part L makes it fast.)

G. `find(L, 5)`: **3** (line 2 once, line 3 once, `return True`). `find(L, 8)`: **7** (lines 2 and 3 three times each, then return). `find(L, 6)`: **9**. `find(L, 7)`: **9** (lines 2 and 3 four times each, then `return False`).

G1. Best: **3** steps (the first item matches). Worst: $\boldsymbol{2n + 1}$ (not there, or last). Big-O plans for the **worst** case: $O(n)$.

G2. $2 \times 1{,}000 + 1 =$ **2,001**.

H. `find_sorted(S, 27)`: (0, 6, 3, 14) then (4, 6, 5, 27): found, `True` after **2** looks. `find_sorted(S, 4)`: (0, 6, 3, 14), (0, 2, 1, 5), (0, 0, 0, 2); then `low` becomes 1, which is bigger than `high` = 0, so the loop stops: `False` after **3** looks.

H1. **3** ($8 \to 4 \to 2 \to 1$); **3** ($10 \to 5 \to 2 \to 1$, since `//` drops the half); **0** (the loop never runs); **10** ($2^{10} = 1{,}024$).

H2. About **10**; **11** (double the data, one more look); **20** ($1{,}000 \times 1{,}000$, so $10 + 10$); **30**.

H3. `False`, though 9 is in `U`. First look: `mid` = 1, `U[1]` = 2 < 9, so it throws away the left half, which holds the 9. Bisection is only right on a **sorted** list.

I. $O(\log n)$: 30 steps, about 3 millionths of a second: **instant**. $O(n)$: 10^6 steps, **0.1 s**. $O(n^2)$, n = 100,000: 10^{10} steps, 1,000 s, **about 17 minutes**. $O(n^2)$, n = 1 million: 10^{12} steps, 100,000 s, **about 28 hours**. $O(2^n)$, $n = 40$: $2^{40} \approx 10^{12}$ steps, so also **about a day** (30 hours exactly) — and $n = 50$ would take about 3.5 *years*.

J1. **1,000** (10^3); **10,000**; × **10**.

J2. **32** (2^5); **64**. One more switch, twice the patterns.

J3. 26^4 = **456,976**; × **26**. Each extra letter makes a guessing program 26 times slower: here exponential growth is on your side.

J4. 10^7 PINs: **1 s**. 10^9 PINs: **100 s**, under 2 minutes. Two more digits, 100 times longer.

K. `{'cat': [1, 2], 'sat': [1, 3], 'ran': [2], 'dog': [3], 'down': [3]}`. Each list holds the pages a word is on.

K1. `[1, 3]`; `[3]`; `True`; **KeyError** (no page has the word).

K2. ***O*(1)** — one dictionary lookup. Pay once to build, then every search is one step.

L1. `{}`; `x in seen`; `seen[x]`. Every value is remembered in the dictionary the first time it is seen, so asking about it later is one step.

L2. `B`; `x in in_B`. Big-O: ***O*(*n*)** — one pass over `B` to build the dictionary, one pass over `A` with one-step checks: $n + n$.

L3. Line **4**: `x not in result` searches the list `result`, which grows up to n items, and it runs n times: $O(n^2)$. The blank is `x not in seen`: a one-step check, so `unique_fast` is $O(n)$. The list keeps the order; the dictionary only answers "seen it?".

M1. The time changes with the computer and with what else it is running; the number of steps depends only on the code and the data, so it is the same everywhere.

M2. For big data the *power* of n decides how fast the steps grow; the number in front only moves the point where a better family takes over (and a faster computer changes it anyway).

M3. Each extra item doubles the steps, so 1,000 times the speed pays for only about 10 more items ($2^{10} \approx 1{,}000$). A better family beats a faster computer.

N1. `for i in range(len(L)):` / `for j in range(i + 1, len(L)):` / `if L[i] + L[j] == t:` / `return True` / `return False` (after both loops). Every pair once, like `mystery`: $O(n^2)$.

N2. `seen = {}` / `for x in L:` / `if t - x in seen:` / `return True` / `seen[x] = True` / `return False`. The partner of `x` is `t - x`; asking the dictionary is one step, so the whole function is $O(n)$. Checking *before* adding `x` stops 3 + 3 from using one item twice.