CS 3343 · ANALYSIS & DESIGN OF ALGORITHMS · FALL 2026

Correct, Then Fast.

A mini-textbook for the exam.
Count the work. Name the bound.
Prove the loop did what you claim.

Cases and counting
Big-O, Θ, Ω, and the strict cousins
Hoare triples, forward and backward
Loop invariants that survive the exit
Insertion sort and selection sort

Prepared for James Guerrero · ywp789

Study packet · guided steps, practice, and a full answer key

The number you plan around is the one you can guarantee.

HOW TO USE THIS

Read it in this order

This packet is built from the CS 3343 notes and homework already in your analysis folder: the quick notes on cases, counting, asymptotic notation, predicates, Hoare triples, and loop invariants; the insertion-sort verification notes; and Homework 3 on selection sort. The pop-culture explainers are in here too, shortened into stories that carry a definition rather than replace one.

Keep this

Correctness comes before speed. A fast program that returns the wrong array is not an algorithm you can defend. The title is the order of work on every question: say what the code guarantees, then say how the cost grows.

The homework questions keep the same skeleton, and that skeleton is the one worth drilling:

Move	What a complete answer contains
1	Trace the code on a given input. Watch ties, and watch the last failed test.
2	Say whether best and worst case differ, and name the input that causes each.
3	Count cost times executions. Sum. Then drop constants for Θ.
4	State an invariant tied to the loop counter.
5	Initialization, maintenance, termination. Exit value exact, not "the loop ended."

Guided boxes walk a method. Walkthrough boxes hold the writeup — cover them with a sheet of paper, try the step, then look. The practice exam near the end has no walkthrough on the same page. The answer key is after it. The last page is the morning-of crib sheet.

Contents

1 What you are actually judging

An **algorithm** is a precise, finite list of steps that turns an input into an output and is guaranteed to stop. Design is inventing the steps. Analysis is judging them: does it do the right thing on every legal input, and how does the work grow as the input grows?

We do not time it with a stopwatch. A stopwatch measures a laptop, a language, and a mood. A **step count** measures the algorithm. Same code, same input shape, same count on any machine.

Field story

Picture the Travis Scott onsale from the notes. The website is the algorithm: it does not change at 10:00am. The crowd is the input. Presale, first in line, is the friendliest input. General sale, position 480,000, sold out when you arrive, is the meanest. A typical fan is somewhere in the middle. You still publish the worst case, because that is the only experience you can promise.

Case	Which input	What it is for
Best	Fewest steps	A brag, not a guarantee.
Worst	Most steps	The bound you plan around.
Average	Typical over many inputs	Useful, and usually harder to derive.

1.1 Linear search, told as a shelf

A librarian walks a shelf of N books left to right and stops at the first match.

Guided

Step 1. Invent a best-case shelf. Where is the title?
Step 2. Invent two different worst-case shelves. One of them still contains the book.
Step 3. Why can those two worst cases cost the same number of comparisons even though the outcomes differ?

Walkthrough — cover this, then check

Best case: the title is in slot 0. One comparison, then stop. That is $\Theta(1)$ for that input.
Worst case, reason one: the title is in the last slot. You compare N times and find it.
Worst case, reason two: the title is not on the shelf at all. You also compare N times, then report failure.
Both are $\Theta(N)$. "Found it at the end" and "it was never there" are different outputs and the same amount of work. Average case, if the book is equally likely to sit anywhere, lands near $N/2$ comparisons — still $\Theta(N)$, with a smaller constant. The guarantee you tell a user is the worst case.

Keep this

Best, worst, and average are properties of an *input family* for a fixed algorithm. They are not three different algorithms. Insertion sort is one piece of code that behaves like a sprinter on sorted data and like a crawl on reversed data.

2 Counting before you abbreviate

The recipe has three lines, and the third one is where people get sloppy.

1. Give every simple statement a constant cost c_i.
2. Count how many times that statement runs, as a function of N.
3. Multiply, add, and only then throw away constants and lower-order terms.

House rules from the notes, worth memorizing because they decide exam points:

- An assignment or a simple test costs a constant. Sequential statements **add**.
- An `if/else` is charged the more expensive branch when you want a single formula that covers every input.
- A loop's cost is the sum of the work of its iterations, not "about N."
- A `while` test runs **one more time** than the body. The last check is the one that fails.

Field story

Groundhog Day, but for a loop counter. Phil wakes up, checks the radio, and only then lives the day. On the morning he finally leaves town, he still checks the radio once — and that check is the one that says the loop is over. If your trace table has five bodies, it needs six tests. Forgetting that row is the off-by-one that quietly ruins a sum.

2.1 The sums that keep showing up

Sum	Closed form
$\sum_{i=1}^{n} 1 = n$	the loop really does run n times
$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$	$1 + 2 + \cdots + n$
$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$	when the inner work itself grows with i
$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$	doubling each round
$\sum_{i=0}^{n-1} c\, r^i = c\,\frac{r^n - 1}{r - 1}$	geometric, $r \neq 1$

2.2 A loop you should be able to count cold

```
s = 0
k = 1
while (k ≤N)
    s = s + k
    k = k + 1
```

Statement	Cost	Times
`s = 0`	c_1	1
`k = 1`	c_2	1
`while (k <= N)`	c_3	$N + 1$
`s = s + k`	c_4	N
`k = k + 1`	c_5	N

$$T(N) = (c_1 + c_2) + c_3(N + 1) + (c_4 + c_5)N = (c_3 + c_4 + c_5)\,N + (c_1 + c_2 + c_3).$$

The leading term is linear, so $T(N) = \Theta(N)$. Best, worst, and average are the same: nothing in the bounds depends on the values, only on N.

2.3 When the inner loop is a triangle

```
c = 0
for i = 0 to N-1
    for j = 0 to i-1
        c = c + 1
```

For a fixed i, the inner body runs i times. The $i = 0$ pass runs 0 times.

$$\sum_{i=0}^{N-1} i = \sum_{i=1}^{N-1} i = \frac{(N-1)N}{2}.$$

So c ends at $N(N-1)/2$, and the whole fragment is $\Theta(N^2)$. This triangle is the same sum selection sort pays, and the same sum insertion sort pays on reversed input.

Guided

Step 1. In the double loop above, how many times does the *inner test* run? (Body count is not test count.)
Step 2. Does the data in some array change either answer? If you replaced `c = c + 1` with `if (A[j] < A[i]) c++`, would the Θ class change under the "charge the expensive branch" rule?

Walkthrough — cover this, then check

For each i, the inner test runs $i+1$ times (the bodies, plus the failing test). Summed, that is $\sum_{i=0}^{N-1}(i+1) = \sum_{k=1}^{N} k = N(N+1)/2$, still $\Theta(N^2)$.
The `if` does not change the iteration count. Under the house rule you charge the pricey branch every time, so the class stays $\Theta(N^2)$. A `break` or a `return` would be different: that can make the count depend on the data, which is exactly why insertion sort has a best case and this loop does not.

3 Five symbols, one race

Four racers share a track that keeps getting longer. The length is N.

Racer	Class	What doubling the track feels like
Flash	$\Theta(\log N)$	One extra stride.
Jogger	$\Theta(N)$	The run doubles.
Weaving	$\Theta(N \log N)$	A little more than double.
Crawler	$\Theta(N^2)$	The run quadruples.

On a short track the crawler can look fine. The notation is a statement about large N, written "as $N \to \infty$." Insertion sort's best case is the jogger. Its worst case is the crawler. Same algorithm, two racers, chosen by the input.

3.1 The definitions, said with constants

We write $f(N) = O(g(N))$ even though $O(g)$ is a *set*. The equals sign means "is a member of."

Keep this

$f(N) = O(g(N))$ means: there exist constants $c > 0$ and $N_0 > 0$ such that

$$0 \le f(N) \le c\,g(N) \quad \text{for all } N \ge N_0.$$

Upper bound. Grows no faster than g. The bound may be loose: $2N = O(N^2)$ is true and sloppy.
$f(N) = \Omega(g(N))$ flips the inequality: $f(N) \ge c\,g(N)$ for large N. Lower bound. Grows no slower. Also allowed to be loose.
$f(N) = \Theta(g(N))$ means both. Same growth rate. This is the one to report when you actually counted.

The strict cousins refuse a tie.

Symbol	Limit test	Plain English
$f = o(g)$	$\lim f/g = 0$	strictly slower
$f = O(g)$	f/g stays bounded	no faster, tie allowed
$f = \Theta(g)$	f/g tends to a positive constant	tied
$f = \Omega(g)$	f/g stays at least a positive constant	no slower, tie allowed
$f = \omega(g)$	$\lim f/g = \infty$	strictly faster

Little-o, formally: for *every* $c > 0$, not merely some c, there is an N_0 past which $f(N) < c\,g(N)$. That is why $2N^2 \neq o(N^2)$. The ratio tends to 2, not to 0. They are Θ of each other.

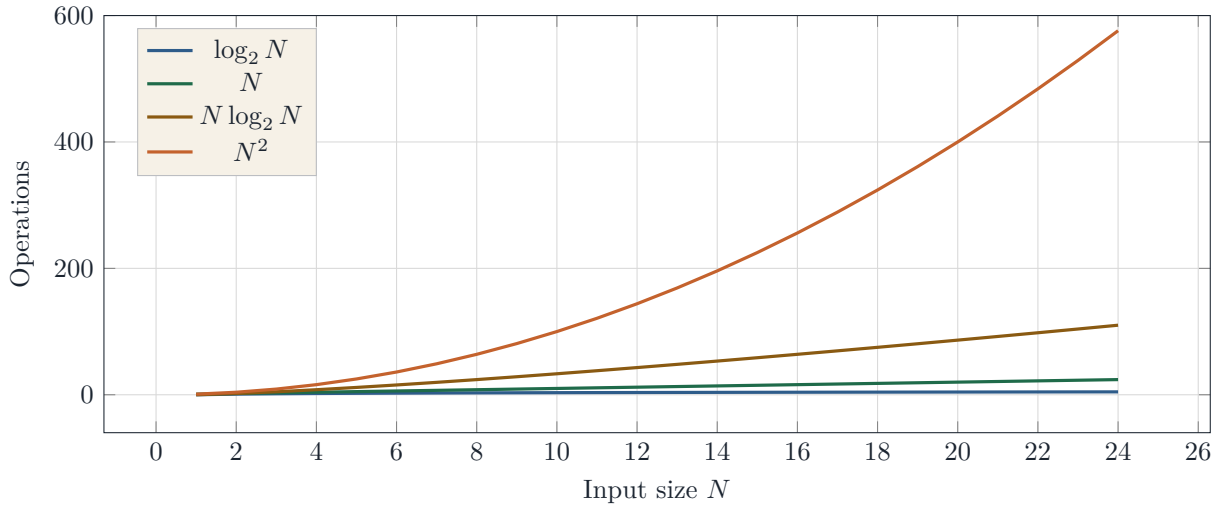
Guided

Step 1. Name the leading term of $T(N) = 7N^3 + 100N^2 + 4$, and the Θ class.
Step 2. Check a small N before you trust the leading term. At $N = 10$, which of $7N^3$ and $100N^2$ is larger? At what N does the cubic term take the lead?
Step 3. Is $3N \log N = o(N^2)$? Is $2N^2 = o(N^2)$?

Walkthrough — cover this, then check

Leading term $7N^3$, so $T(N) = \Theta(N^3)$. It is also $O(N^3)$ and $\Omega(N^3)$. It is not $o(N^3)$, because the ratio tends to 7, not 0.
At $N = 10$, $7N^3 = 7000$ and $100N^2 = 10000$. The "lower-order" term is still winning. Set $7N^3 = 100N^2$ and cancel N^2 (for $N > 0$): $7N = 100$, so $N \approx 14.3$. From $N = 15$ upward the cubic term is larger, and after that the gap only grows. Θ describes the far end of the track, not the first lap. That is also why a $\Theta(N^2)$ algorithm can beat a $\Theta(N \log N)$ algorithm on a tiny input: constants and lower terms are allowed to matter while N is small. For $N = 10^6$ you take the slower-growing one without a second thought.
$3N \log N = o(N^2)$ because $3 \log N / N \to 0$. And $2N^2 \neq o(N^2)$, as above.

3.2 How fast the families pull apart



The quadratic is clipped in spirit by the eye: at $N = 24$ it is already 576, while $N \log_2 N$ is still under 120. The log curve is the one lying on the floor. That picture is the whole reason the notation exists.

N	$\log_2 N$	N	$N \log_2 N$	N^2	2^N
10	3.3	10	33	100	1 024
20	4.3	20	86	400	$\approx 10^6$
40	5.3	40	213	1 600	$\approx 10^{12}$
100	6.6	100	664	10 000	forget it

Change-of-base is a constant, and constants die inside Θ, so $\log_2 N$, $\ln N$, and $\log_{10} N$ are the same class. What base you write does not change an answer of $\Theta(\log N)$.

If the class is...	Replacing N by $2N$ multiplies the work by...
$\Theta(1)$	1
$\Theta(\log N)$	adds about one, it does not really "multiply"
$\Theta(N)$	2
$\Theta(N \log N)$	a bit more than 2, and the extra shrinks as N grows
$\Theta(N^2)$	4
$\Theta(N^3)$	8
$\Theta(2^N)$	the old amount, squared

4 Predicates and Hoare triples

A **predicate** is a true-or-false claim about the program's state. In this course it wears square brackets: $[x > 0]$. Code wears braces: `{x = x + 1;}`. A **Hoare triple** is the whole sandwich:

$$[\text{precondition}] \quad \{\text{statements}\} \quad [\text{postcondition}].$$

Read it as a promise: if the precondition is true, and the statements then run to completion, the postcondition is true.

Field story

Thanos can snap only because the precondition is ridiculous: six stones, one gauntlet. The statement is `snap()`. The postcondition is that half of life is gone. Iron Man later executes a statement with the same shape and a different precondition — different stones, his own hand — and the postcondition is a different army turning to dust. The line of code never tells you the result by itself. The precondition does. That is why a triple with a missing left-hand side is not an argument.

4.1 The subscript-zero snapshot

x_0 is not a variable the program updates. It is a name for whatever x was at a chosen instant, frozen, like a photograph. After `x = x + 1` you want $[x = x_0 + 1]$, not the limp $[x > x_0]$. The second one is true and too weak: it also allows $x = x_0 + 40$.

```
[x = x0] { y = 2*x; } [x = x0 AND y = 2*x0]
```

The postcondition of one line is the precondition of the next. You slide forward.

Guided

Fill both holes before you look.

$$[a = a_0]\ \{\texttt{a = a + 1;}\}\ [\ ?\ ]\ \{\texttt{a = a * 2;}\}\ [\ ?\ ]$$

Walkthrough — cover this, then check

After the increment, $[a = a_0 + 1]$. The old a_0 does not change when a does; that is the point of the snapshot.
After the multiply, $a = 2(a_0 + 1) = 2a_0 + 2$. Writing $[a = 2a]$ is nonsense: a cannot appear on both sides as if it had two values. If you need the intermediate, give it a name, or substitute.

4.2 A swap, traced like a card trick

You want a and b exchanged. The temporary is part of the story.

$$[a = a_0 \wedge b = b_0]\ \{\texttt{t = a;}\}\ [t = a_0 \wedge a = a_0 \wedge b = b_0]$$
$$\{\texttt{a = b;}\}\ [t = a_0 \wedge a = b_0 \wedge b = b_0]$$
$$\{\texttt{b = t;}\}\ [t = a_0 \wedge a = b_0 \wedge b = a_0].$$

That last line is the strongest thing this code guarantees from that start. Dropping $t = a_0$ is allowed if nobody asked about t, but it is a weaker postcondition. Stronger means more restrictive: fewer states still qualify.

4.3 Branches join with “or”

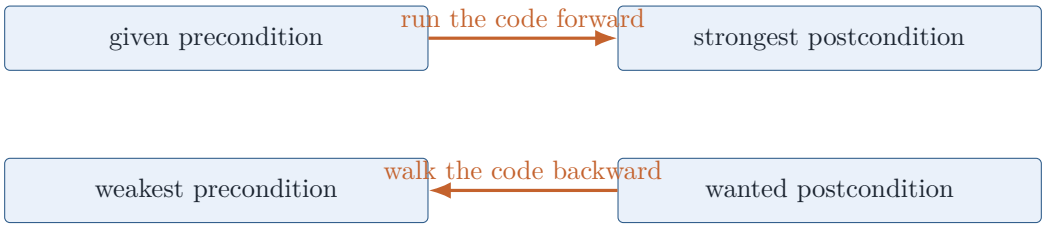
Exactly one branch runs. You prove each branch with its own condition folded in, then combine with $\vee$, never $\wedge$.

```
if (x > 7)
    y = x;
else
    y = 0;
```

Starting from no information other than the branch test, the two possible landings are $[x > 7 \wedge y = x]$ and $[x \leq 7 \wedge y = 0]$. The postcondition is their disjunction. Using $\wedge$ would claim both branches ran, which is a different program.

5 Forward is strongest, backward is weakest

If $P \Rightarrow Q$, then P is **stronger**. It rules out more worlds. $[x = 3]$ is stronger than $[x > 0]$. Every state that satisfies the first satisfies the second, and not the other way around.



- The question gives you a start and a statement, and asks what you know afterward. That is **strongest postcondition**. Forward.
- The question gives you a statement and a goal, and asks what must have been true beforehand. That is **weakest precondition**. Backward. “Weakest” means the most generous start that still forces the goal — no extra restrictions you do not need.

5.1 One assignment, backward

Find the weakest precondition of $\{\texttt{x = x + 1;}\}$ for the goal $[x \leq 10]$.

The new x is the old x plus one. You need old $+ 1 \leq 10$, so the weakest precondition is $[x \leq 9]$. Asking for $[x \leq 8]$ also works, but it is stronger than necessary: it rejects $x = 9$, which would have been fine.

5.2 A branch, backward

Goal: $[q > 6]$ after

```
if (p < 10) q = p + 1;
else q = p - 10;
```

Guided

Step 1. On the true branch, combine $p < 10$ with whatever $p + 1 > 6$ demands.
Step 2. On the false branch, the test you know is $p \geq 10$. Combine it with $p - 10 > 6$.
Step 3. Join with $\vee$. Then sanity-check one integer just inside the gap, if there is a gap.

Walkthrough — cover this, then check

True branch: $p < 10$ and $p > 5$, so $5 < p < 10$.
False branch: $p \geq 10$ and $p > 16$. The second condition is stricter, so this branch contributes $p > 16$.
Weakest precondition: $(5 < p < 10) \vee (p > 16)$.
Check the boundary you might have flipped. If $p = 5$, the true branch sets $q = 6$, and $6 > 6$ is false, so $p = 5$ is correctly excluded. If $p = 16$, the false branch sets $q = 6$, also excluded. If $p = 10$, you take the false branch and $q = 0$. The gap is real.

Trap

A "proof" claims that from $[x = x_0]$, after `x = x + 1`, the strongest postcondition is $[x \geq x_0]$. The inequality is true and it is not the strongest. Strongest here is $[x = x_0 + 1]$. Anything you can loosen — $\geq$, $>$, "x changed" — is a weaker statement wearing a confident voice.

6 Loop invariants

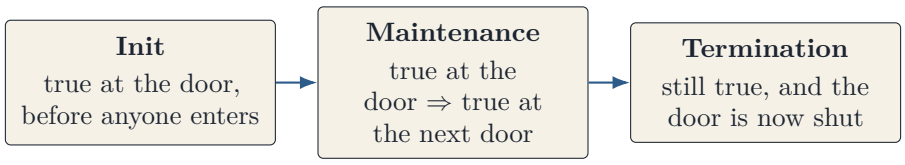
A **loop invariant** is a predicate that is true every time control is at the top of the loop: before the first iteration, between iterations, and when the loop condition has just failed. It is a comment you prove. It does not execute.

Field story

Phil Connors wakes up on the same morning and the same Sonny and Cher song. The day resets. What he is allowed to carry is the invariant: a fact that is still true at the top, no matter which copy of the day this is. Spider-Man's multiverse version of the same idea: the details change per universe, and "bitten by a spider" does not. Initialization is the first universe. Maintenance is one portal jump that preserves the fact. Termination is what you can conclude once the jumping stops.

Three steps, and they are not the same kind of argument:

1. **Initialization.** True before iteration one. Often an empty range, a one-element range, or a variable that is still 0.
2. **Maintenance.** Assume it at the start of an arbitrary iteration. Show it at the start of the next one. This is induction. You must use the assumption, not the thing you are still proving.
3. **Termination.** The invariant is still true, and the loop test is now false. Name the counter's exact value. Deduce the goal. This is not induction anymore.



6.1 How to invent one

Ask: at the top of the loop, with the counter sitting on j, what useful fact is already true about the work finished so far? Finished almost always means the indices strictly before j.

Loop's job	Shape that usually works
Sum or product	s equals the sum (product) of $A[0..j-1]$
Count	`count` equals how many of $A[0..j-1]$ passed the test
Max or min	m equals the max (min) of $A[0..j-1]$
Search	`found` is true exactly when the needle is in the prefix
Insertion sort	the prefix is the original prefix, sorted
Selection sort	the prefix is the smallest so far, sorted, and $\leq$ the suffix

Then plug in the first value of the counter. If the claim becomes awkward or false, the range is off by one. Fix the range before you write a heroic maintenance paragraph.

Field story

Vacuous truth is the initialization gift. "Every dragon in this closet is house-trained" is true when the closet is empty, because there is no dragon available to be a counterexample. An empty prefix is sorted. Every element of an empty prefix is $\leq$ every element of the rest of the array. Selection sort's invariant at $k = 0$ is that sentence. It feels like a cheat the first time and it is the correct base case.

6.2 Model proof: the product that becomes $N!$

```
p = 1
i = 1
while (i ≤N)
    p = p * i
    i = i + 1
```

Invariant. At the top of the loop, $p = (i-1)!$ (the product of the integers from 1 through $i-1$, and the empty product is 1).

Initialization. $i = 1$ and $p = 1$. The product through 0 is empty, so $p = 1 = 0!$. Holds.

Maintenance. Assume that at the start of an iteration, $p = (i-1)!$, and that the body runs, so $i \leq N$. The assignment `p = p * i` makes $p = (i-1)! \cdot i = i!$. Then `i = i + 1`. Let i' be the new value, so $i' = i + 1$ and the current p equals $(i'-1)!$. That is the invariant at the next test.

Termination. i starts at 1 and increases by exactly 1, so the first time $i \leq N$ fails we have $i = N + 1$ (for $N \geq 1$). The invariant says $p = (N + 1 - 1)! = N!$. If $N = 0$, the test $1 \leq 0$ fails immediately and $p = 1 = 0!$, which matches.

Guided

Same skeleton, different job. The loop should leave m equal to the maximum of $A[0..N-1]$, assuming $N \geq 1$.

```
m = A[0]
j = 1
while (j < N)
    if (A[j] > m) m = A[j]
    j = j + 1
```

Step 1. Write the invariant in terms of j.
Step 2. Check $j = 1$.

Step 3. In maintenance, what do you know about m *before* the `if`, and what is true about m *after* it, regarding $A[0..j]$?
Step 4. What is j at exit?

Walkthrough — cover this, then check

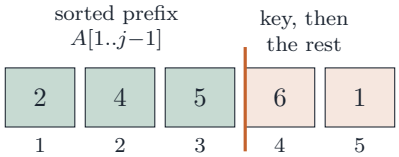
Invariant: $m = \max(A[0..j-1])$.
Initialization: $j = 1$, so the range is $A[0..0]$, and $m = A[0]$. A one-element range is its own maximum.
Maintenance: assume $m = \max(A[0..j-1])$. The `if` sets m to $A[j]$ when $A[j]$ is larger, and leaves m alone otherwise. Either way, m becomes $\max(A[0..j])$. Then j increases by 1, and the invariant is restored for the new j.
Termination: j increases by 1 from 1, so exit is exactly $j = N$. Then $m = \max(A[0..N-1])$.
The `if` does not change how many times the loop runs. This scan is $\Theta(N)$ on every input of length N. There is no best case hiding in the data.

7 Insertion sort: the hand of cards

You sort a hand the way a card table does. The cards already in your hand are sorted. You pick up the next card and slide it left until it sits between a smaller-or-equal neighbor and a larger one.

The course writes it 1-indexed, the CLRS way:

```
INSERTION-SORT(A, n)
for j = 2 to n
    key = A[j]
    i = j - 1
    while i > 0 and A[i] > key
        A[i+1] = A[i]
        i = i - 1
    A[i+1] = key
```



The picture is the moment j points at the next card. Nothing to the right of `key` has been rearranged. That sentence is the difference between this sort and selection sort.

7.1 The outer invariant

At the start of each iteration of the `for` loop, $A[1..j-1]$ holds the same elements that were originally in $A[1..j-1]$, now in sorted order.

Initialization. $j = 2$. The claim is about $A[1..1]$, one element. A single element is sorted, and it is the original element. Done.

Maintenance. Assume the prefix $A[1..j-1]$ is the original prefix, sorted. `key` saves $A[j]$. The inner loop shifts strictly larger elements one slot right and stops at the first element that is $\leq$ `key`, or at the left wall. The hole it leaves is the unique place `key` belongs. After `A[i+1] = key`, the prefix $A[1..j]$ is sorted and contains the original elements of $A[1..j]$. The `for` loop then advances j, which restores the invariant.

Termination. A `for j = 2 to n` exits with $j = n + 1$. The invariant says $A[1..n]$ is the original array, sorted. That is the specification of sorting.

The inner loop has its own invariant if a question asks: at the top of the `while`, the elements of $A[i+1..j]$ that have already been shifted are the old occupants, moved one right, and each of them is $>$ `key`. You rarely need this unless the question points at the inner loop by name.

7.2 Best case, worst case, and the number t_j

Let t_j be the number of times the inner `while` *test* is evaluated during outer iteration j, for $j = 2, \ldots, n$.

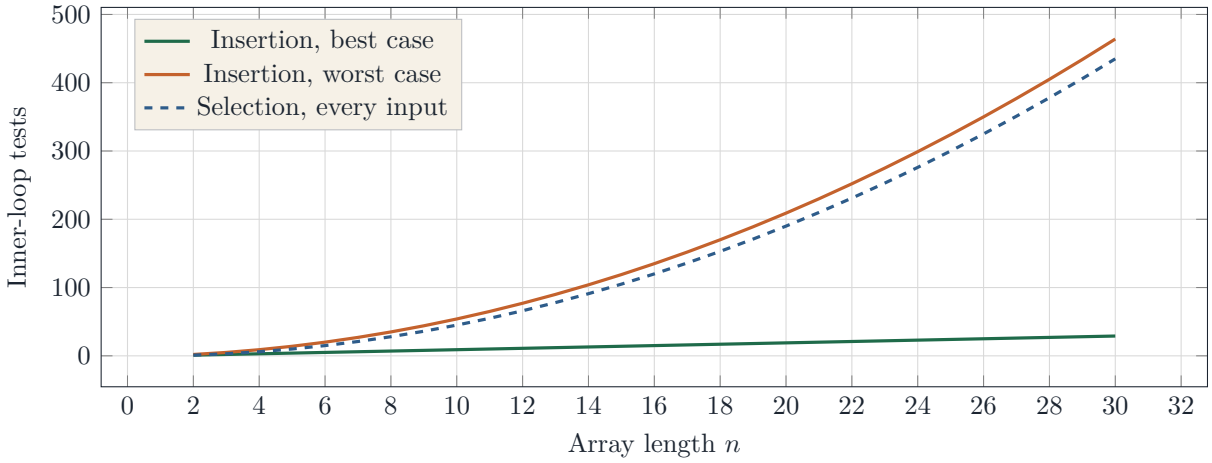
Input	Each t_j	Total inner tests
Already sorted	$t_j = 1$	$\sum_{j=2}^{n} 1 = n - 1 = \Theta(n)$
Reverse sorted	$t_j = j$	$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 = \Theta(n^2)$

Already sorted: i starts at $j - 1 \geq 1$, the test runs once, $A[i] >$ `key` is false, and the body never runs. One test per outer step, $n - 1$ of those, plus the other outer lines which are also linear. Best case $\Theta(n)$.

Reverse sorted: every new key is smaller than the entire sorted prefix, so the loop shifts all $j - 1$ elements and then fails when i hits 0. Tests: $j - 1$ successes plus one failure, so $t_j = j$.

$$\sum_{j=2}^{n} j = \Big(\sum_{j=1}^{n} j\Big) - 1 = \frac{n(n+1)}{2} - 1.$$

Drop the constants: $\Theta(n^2)$.



The dashed curve is selection sort, waiting in the next section. It never gets to ride the green line. Insertion's worst case sits just above selection, because $t_j = j$ includes that extra failed test; the comparison count people sometimes quote for insertion's shifts is the slightly smaller $n(n-1)/2$. Either way the class is $\Theta(n^2)$. Say which quantity you counted.

7.3 A short trace, so the subscript means something

Array $A[1..5] = [5,\ 2,\ 4,\ 6,\ 1]$.

j	`key`	array after the insertion	t_j
2	2	[2, 5, 4, 6, 1]	2
3	4	[2, 4, 5, 6, 1]	2
4	6	[2, 4, 5, 6, 1]	1
5	1	[1, 2, 4, 5, 6]	5

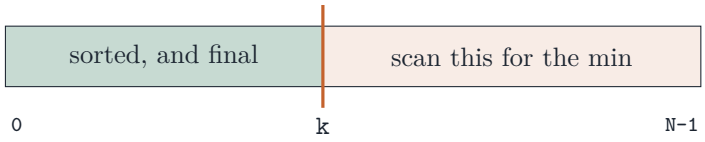
Sum of t_j is $2+2+1+5=10$. Best-case sum would be 4. Worst-case sum would be $2+3+4+5=14$. This input is between them, pulled toward the worst case by the 1 that has to tunnel all the way to the front. "Between" is an acceptable answer when you show the two formulas you compared against.

8 Selection sort: the wall that only moves right

Homework 3's code is 0-indexed. Do not "fix" it into the insertion-sort indexing in the middle of a proof.

```
void selectionSort(int A[], int N) {
  int k = 0;
  while (k < N-1) {
    int mi = k;
    for (int j = k+1; j < N; j++) {
      if (A[j] ≤A[mi]) mi = j;
    }
    int t = A[k];
    A[k] = A[mi];
    A[mi] = t;
    k++;
  }
}
```

Each pass scans the unsorted suffix, finds the minimum, and swaps it into the cell at k. Then the wall moves one cell right. The prefix is not merely sorted. It is *finished*: those values will never move again.



8.1 Why `mi` is an index

The swap needs a location. If you stored only the minimum value, you could write it into $A[k]$ and you would not know which cell should receive the old $A[k]$. That value would be duplicated or lost, and you would scan again to find the hole. The index gives you both: the value is A[`mi`], and the slot is `mi`. The swap is $O(1)$.

The test is `<=`, not `<`. On a tie, `mi` moves to the later index. If a trace includes two equal minima, the rightmost one wins. Using `<` in your head is the classic wrong row.

8.2 There is no best case

The inner loop's bounds are $j = k+1, \ldots, N-1$. That is $N-1-k$ trips, decided by N and k, never by the values in A. There is no `break`. An already-sorted array and a reversed array both pay

$$\sum_{k=0}^{N-2} (N-1-k) = (N-1) + (N-2) + \cdots + 1 = \frac{N(N-1)}{2}$$

comparisons of the form `A[j] <= A[mi]`. The only data-dependent statement is the assignment `mi = j`, and it cannot change the trip count. Best, worst, and average are the same algorithm wearing the same stopwatch: $\Theta(N^2)$.

Selection sort cannot "notice" that the suffix is sorted. Confirming that $A[k]$ is the minimum means looking at every remaining element. Insertion sort gets to stop early. That single difference is the green line versus the dashed line on the chart in the previous section.

8.3 The cost line, once, cleanly

Piece	Times	Why
`k = 0`	1	before the loop
outer test `k < N-1`	N	$N-1$ successes, one failure
`mi = k` and the swap and `k++`	$N-1$	once per successful outer pass
inner body / comparison	$N(N-1)/2$	the triangle
inner test `j < N`	$N(N+1)/2-1$	one extra test per inner loop

Inner tests, summed: for each k from 0 to $N-2$ the test runs $(N-1-k)+1 = N-k$ times.

$$\sum_{k=0}^{N-2}(N-k) = N + (N-1) + \cdots + 2 = \frac{N(N+1)}{2} - 1.$$

Whatever constants you hang on these rows, the N^2 term arrives from the triangle and $T(N) = \Theta(N^2)$. If the question only wants comparisons, box $N(N-1)/2$ and stop. Say so.

8.4 The invariant has three clauses

At the start of each outer iteration, when `k < N-1` is about to be tested:

(i) $A[0..k-1]$ is sorted non-decreasing;

(ii) every element of $A[0..k-1]$ is $\leq$ every element of $A[k..N-1]$;

(iii) $A[0..N-1]$ is a permutation of the original array.

In one sentence: the first k slots hold the k smallest values, in order.

Clause (i) alone is the insertion-sort invariant, and it is the wrong invariant here. Selection sort pulls values from anywhere in the suffix. The prefix is not "the original prefix, rearranged." Clause (ii) is what makes those prefix values final. A proof that only says "the left side is sorted" cannot, at the end, force $A[N-1]$ to be the largest.

Initialization. $k = 0$. The prefix $A[0..-1]$ is empty. Empty is sorted. Every element of an empty set is $\leq$ the suffix, vacuously. Nothing has been swapped, so (iii) holds.

Maintenance. Assume all three, and assume $k < N-1$ so the body runs. The inner loop's job, which you may take as given if the question says so, is that $A[\texttt{mi}]$ is a minimum of the suffix $A[k..N-1]$. Call that value m. The swap:

- (iii) survives because a swap rearranges;
- the old prefix was not touched, since both indices in the swap are $\geq k$. If $k = 0$ the new prefix is the single value m, which is sorted. If $k > 0$, clause (ii) says $A[k-1] \leq m$, so $A[0..k]$ is sorted;
- m is $\leq$ everything that remains in the new suffix, and the old prefix was already $\leq$ all of those values, so clause (ii) holds at the new wall.

Then `k++` makes those claims line up with the invariant again.

Termination. k starts at 0 and rises by 1, so the test $k < N-1$ fails at exactly $k = N-1$ (for $N \geq 1$). Substitute:

- $A[0..N-2]$ is sorted;
- every element of that prefix is $\leq A[N-1]$;
- the array is still a permutation of the input.

So $A[0] \leq A[1] \leq \cdots \leq A[N-1]$, with the original elements. There is no pass for $k = N-1$. Once $N-1$ smallest values sit in place, the last cell is the maximum for free, which is clause (ii) paying rent. For $N = 0$ the test $0 < -1$ fails immediately and the empty array is sorted.

8.5 One trace with a tie

$A = [4,\ 9,\ 2,\ 2,\ 7]$, so $N = 5$. The two 2s are the trap.

k	`mi`	array after the swap	what the scan did
–	–	[4, 9, 2, 2, 7]	start
0	3	[2, 9, 2, 4, 7]	4 loses to the later 2
1	2	[2, 2, 9, 4, 7]	9 loses to index 2
2	3	[2, 2, 4, 9, 7]	$4 \le 9$
3	4	[2, 2, 4, 7, 9]	$7 \le 9$

There is no row of work for $k = 4$. The loop stops when $k < 4$ fails. At $k = 0$, `mi` moves $0 \to 2$ (first 2) $\to 3$ (second 2, because `<=`). If you wrote `mi`$= 2$, you simulated a strict $<$.

8.6 Side by side, so they stop trading invariants

	Insertion	**Selection**
Indexing in the notes	$1..n$, outer $j = 2..n$	$0..N-1$, outer $k = 0..N-2$
Prefix means	original prefix, sorted	the smallest k values, sorted
Suffix	not yet examined	everything left is $\ge$ the prefix
Inner loop	stops early when the hole is found	always scans the whole suffix
Best case	sorted input, $\Theta(n)$	none
Worst case	reversed, $\Theta(n^2)$	every input, $\Theta(N^2)$
Exit value	$j = n + 1$	$k = N - 1$

Trap

If a question uses 0-based insertion sort, do not panic and do not quote $j = n + 1$ from memory. Re-derive the exit from the test that is actually written. The invariant's *idea* survives a change of indexing. The exit number does not.

9 How a proof should sound

The notes are explicit: these proofs are structured arguments, not a ritual in symbolic logic. A reader should be able to check each sentence against the code. Four sentences of the right shape beat a page of symbols with a hole in the middle.

Sounds finished, isn't	**Write this instead**
"It's obviously true."	"By the inductive hypothesis, . . . "
"Then it just works."	"After this line, s becomes $s + A[k]$, so . . . "
"The loop ends eventually."	"Exit is exactly when $k = N$, because k rises by 1."
"So we're done."	"Invariant plus that exit value is the postcondition."

A maintenance paragraph you can imitate:

Keep this

Assume that at the start of iteration j, [invariant, restated for this j]. The body [one sentence about what happens to the tracked variables]. Afterward, [the new equation]. Because j becomes $j + 1$, that equation is the invariant for the next test.

9.1 The mistakes that actually cost points

Trap	The fix, in one line
While-body count used as the test count	Tests = bodies +1.
Invariant says $j-1$ but the code finished j	Plug in the first counter value before you fall in love with the sentence.
"$k \geq N$" written as "$k = N$"	Only legal if you also say k increases by exactly 1.
$[j \geq 0]$ after `j-`	That lost the "exactly one." Use $j = j_0 - 1$.
Forward question answered backward	Start state given $\Rightarrow$ strongest post. Goal given $\Rightarrow$ weakest pre.
Branch results joined with $\wedge$	One branch ran. Join with $\vee$.
Branch test thrown away	$[x > 7]\{\texttt{y=x;}\}$ yields $[x > 7 \wedge y = x]$, not $[y = x]$ alone.
Constants dropped in the middle of a sum	Drop them after the closed form exists.
"The loop runs N times"	Nested loops, early exits, and t_j do not owe you that.
Only one case reported	If they can differ, give best and worst. If they cannot, say why.
Insertion invariant on selection sort	Add "prefix $\leq$ suffix," or the last cell never gets proved.

Guided

Rewrite this maintenance step so a classmate can check it. The invariant is "at the top, s is the sum of 1 through $k-1$." The body is `s = s + k; k = k + 1`.
"The loop adds k to s and then k goes up, so it still works."

Walkthrough — cover this, then check

Assume that at the start of an iteration, s equals the sum of the integers from 1 through $k-1$, and that the body runs. After `s = s + k`, s equals the sum from 1 through k. Then k becomes $k+1$. Relative to the new counter, s is the sum from 1 through (new k) -1, which is the invariant at the next test.

10 Practice exam

Eight questions, in the shape of the homework. Write on paper or in the gaps. The answer key starts on the next section; don't scroll there on the first pass. None of these is a pasted homework problem, but every method is one you have already used.

P1. Cases — **a shelf, then a promise**

Linear search walks $A[0..N-1]$ from the left and returns the index of the first occurrence of `target`, or -1 if it is absent.

a) Describe a best-case input. How many comparisons, as a function of N?

b) Describe two worst-case inputs with different return values. Comparisons?

c) A classmate says the average case is "about $N/2$, so the running time is $\Theta(1)$ for practical purposes." What did they confuse?

P2. Count this, then classify it

```
total = 0
i = 0
while (i < N)
    j = 0
    while (j < N)
        total = total + 1
        j = j + 1
    i = i + 1
```

How many times does `total = total + 1` run? How many times does the inner test run? Give $T(N)$ up to Θ. Is there a separate best case?

P3. True, false, or sloppy

Mark each statement true or false. One sentence each, limit or leading term.

a) $4N^2 + 9N = \Theta(N^2)$

b) $N = O(N^2)$, and this bound is tight

c) $\log N = o(N)$

d) $N^2 = \omega(N \log N)$

e) $2N^2 = o(N^2)$

f) $3N \log N = O(N^2)$

P4. Strongest postcondition

Chain forward, one statement at a time.

$$[x = x_0 \wedge y = y_0] \quad \{\texttt{t = x; x = y; y = t;}\}$$

What is the strongest postcondition? Which piece may you drop if the question never mentions t again, and what kind of weakening is that?

P5. Weakest precondition

Find the weakest precondition on a so that $[z \geq 0]$ holds after:

```
if (a > 0) z = a - 3;
else z = -a;
```

Join the branches correctly. Name one integer that fails, so you know the gap is real.

P6. A full invariant proof

Prove the postcondition $[\, s =$ the sum of $A[0..N-1] \,]$, for $N \geq 0$.

```
s = 0
k = 0
while (k < N)
    s = s + A[k]
    k = k + 1
```

State the invariant. Initialization, maintenance, termination, including the exact exit value of k. What is the Θ class, and does it depend on the array contents?

P7. Insertion sort, small and specific

$A[1..3] = [3,\ 1,\ 2]$.

a) Show the array at the start and after each outer iteration. Give t_j for each j.

b) Compare $\sum t_j$ with the best-case sum and the worst-case sum for $n = 3$. Where does this input sit?

c) State the outer invariant at the moment $j = 3$, *before* that iteration's body.

P8. Selection sort, the tie, and the exit

$A = [8,\ 3,\ 3,\ 1]$, $N = 4$, using the course's `<=` scan.

a) Trace k, `mi`, and the array after each swap.

b) How many `A[j] <= A[mi]` comparisons is that, and what is the general formula?

c) Why is "there is a best case when the array is sorted" false for this code?

d) State the three invariant clauses and the value of k at exit. One sentence: why is clause (ii) not optional?

11 Answer key

P1

(a) `target` sits in $A[0]$. One comparison. $\Theta(1)$ for that input only.

(b) `target` equals $A[N-1]$, return value $N-1$; or `target` is absent, return value -1. Both take N comparisons, $\Theta(N)$.

(c) They confused a smaller constant with a slower-growing class. $N/2$ is still linear. Θ does not become $\Theta(1)$ because a constant factor is pleasant. $\Theta(1)$ would mean the cost stops growing with N.

P2

The body `total = total + 1` runs N times for each of N values of i, so N^2 times. `total` finishes at N^2.

For each of the N outer passes the inner test runs $N+1$ times, so $N(N+1)$ inner tests. The outer test runs $N+1$ times.

$T(N) = \Theta(N^2)$. There is no separate best case: both loop bounds are N, and nothing returns early.

P3

(a) True. Leading term $4N^2$.

(b) $N = O(N^2)$ is true and the bound is not tight. Tight would be $\Theta(N)$. The ratio $N/N^2 = 1/N \to 0$, so in fact $N = o(N^2)$.

(c) True. $\log N/N \to 0$.

(d) True. $N^2/(N \log N) = N/\log N \to \infty$, so N^2 is strictly faster-growing. (Careful with the English: ω means the function on the left grows strictly faster, which is the worse running time. The symbol is about growth, not about which algorithm you prefer.)

(e) False. The ratio tends to 2, not 0. They are $\Theta(N^2)$ of each other.

(f) True, and loose. $3N \log N$ is also $o(N^2)$. An O answer does not claim tightness.

P4

After `t = x`: $[t = x_0 \wedge x = x_0 \wedge y = y_0]$.

After `x = y`: $[t = x_0 \wedge x = y_0 \wedge y = y_0]$.

After `y = t`: $[t = x_0 \wedge x = y_0 \wedge y = x_0]$.

That is the strongest postcondition. If t is never mentioned again you may keep only $[x = y_0 \wedge y = x_0]$. Dropping a conjunct is a weakening: more states satisfy the shorter predicate.

P5

True branch, $a > 0$: need $a - 3 \geq 0$, so $a \geq 3$. Combined with the branch, $a \geq 3$.

False branch, $a \leq 0$: $z = -a$. For every such a, $-a \geq 0$ already. The whole false branch is acceptable.

Weakest precondition: $(a \geq 3) \vee (a \leq 0)$. Equivalently, a is not in $\{1, 2\}$ if we are talking about integers; as a real inequality, a is outside the open interval $(0, 3)$.

A failing integer: $a = 1$ takes the true branch and sets $z = -2$. Also $a = 2$. The boundaries work: $a = 3$ gives $z = 0$, and $a = 0$ gives $z = 0$.

P6

Invariant. At the top of the `while`, s equals the sum of $A[0..k-1]$ (empty sum $= 0$).

Initialization. $k = 0$, $s = 0$. The range $A[0..-1]$ is empty. Holds.

Maintenance. Assume the invariant, and assume the body runs, so $k < N$. After `s = s + A[k]`, s is the sum of $A[0..k]$. Then k increases by 1, and s is the sum of $A[0..k'-1]$ for the new counter.

Termination. k starts at 0 and increases by exactly 1, so the test $k < N$ fails at $k = N$. (If $N = 0$ it fails immediately, same value.) The invariant yields the sum of $A[0..N-1]$.

The body runs N times regardless of the values stored in A. The class is $\Theta(N)$ for every input of length N.

P7

Start: [3, 1, 2].

$j = 2$, key $= 1$. $A[1] = 3 > 1$, shift, i falls to 0, place key. Array [1, 3, 2]. Tests: one success and one failure on $i > 0$, so $t_2 = 2$.

$j = 3$, key $= 2$. $A[2] = 3 > 2$, shift; then $i = 1$ and $A[1] = 1 \not> 2$, stop. Place key at index 2. Array [1, 2, 3]. Tests: $t_3 = 2$.

Sum $= 4$. For $n = 3$ the best-case sum is $n - 1 = 2$ and the worst-case sum is $2 + 3 = 5$. This input sits strictly between them, one test away from the worst case.

At the start of $j = 3$, before the body: $A[1..2]$ is the original prefix [3, 1], sorted, hence [1, 3]. The 2 is still sitting in $A[3]$, untouched.

P8

(a) Start [8, 3, 3, 1].

k	`mi`	array after the swap
0	3	[1, 3, 3, 8]
1	2	[1, 3, 3, 8]
2	2	[1, 3, 3, 8]

At $k = 0$ the scan moves off 8 to the first 3, then to the second 3 because $3 \leq 3$, then to 1. At $k = 1$ the two remaining 3s tie and the later index wins, so `mi`$= 2$; swapping index 1 with index 2 exchanges equal values. At $k = 2$, 8 is larger than 3, so `mi` stays 2 and the swap is a self-swap. No body for $k = 3$.

(b) Comparisons: $3 + 2 + 1 = 6$. In general $N(N-1)/2$. Here $4 \cdot 3/2 = 6$.

(c) The inner bounds do not look at the data and there is no early exit. Sorted input still scans every remaining cell. The $\Theta(N^2)$ count is the count for every input.

(d) At each outer test: (i) $A[0..k-1]$ is sorted; (ii) every prefix element is $\leq$ every suffix element; (iii) A is a permutation of the original. Exit at $k = N - 1 = 3$. Clause (ii) is what makes the prefix final, so when $k = N - 1$ the last cell is forced to be the maximum. "The prefix is sorted" alone does not relate $A[N-1]$ to the cells before it.

12 Morning-of crib sheet

Order of work. What is true before. What the code does. What is true after. Then, and only then, the growth rate.

Cases. Best = friendliest input. Worst = the guarantee. Average = typical, often the same Θ with a different constant. If the loop bounds ignore the data, say "no separate best case" and mean it.

Counting. Cost times times, then sum. While-tests = bodies +1. `if/else` charged at the pricey branch unless you are explicitly doing cases. Nested cost is a sum, usually $N(N-1)/2$ when the inner length shrinks by one each pass.

Sums.

$$\sum_{i=1}^{n} i = n(n+1)/2$$
$$\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6$$
$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

Symbols. O no faster, Ω no slower, Θ tied. o strictly slower (limit 0, every $c > 0$). ω strictly faster (limit ∞). Loose O can be true. Drop constants only at the end. log bases do not matter inside Θ.

Doubling. $N \to 2N$ multiplies $\Theta(N)$ by 2, $\Theta(N^2)$ by 4, $\Theta(N^3)$ by 8. $\Theta(2^N)$ squares.

Notation. $[\,P\,]$ is a predicate. $\{$ `code` $\}$ is code. x_0 is a frozen value, not a program variable. Stronger $\Rightarrow$ more restrictive.

Direction. Forward from a given start: strongest postcondition. Backward from a goal: weakest precondition. Branches: fold the test into each side, join with $\vee$.

Invariant, three beats. Init (often empty or one cell). Maintenance (assume, then one pass, then the counter moves). Termination (exact exit value + invariant $\Rightarrow$ goal).

Insertion, 1-indexed. Invariant: $A[1..j-1]$ is the original prefix, sorted. Best: $t_j = 1$, sum $n-1$, $\Theta(n)$. Worst: $t_j = j$, sum $n(n+1)/2-1$, $\Theta(n^2)$. Exit: $j = n+1$.

Selection, 0-indexed. Invariant: first k cells are the k smallest, sorted, and $\leq$ the suffix; array is a permutation. Comparisons always $N(N-1)/2$. Class always $\Theta(N^2)$. Ties use `<=`, so the later index wins. Exit: $k = N-1$. No last pass, because the final cell is already forced.

Before you hand it in. First counter makes the invariant obvious? Maintenance says "assume"? Exit value is a number, not a vibe? Both cases, or a sentence explaining why there is only one? Branch results joined with $\vee$?

Correct, then fast. The prefix you can describe is the proof.
James Guerrero · CS 3343 · Fall 2026