CS/AI — Mathematics for AI

Lecture 2: Mathematical Induction, Loop Invariants, and Recursive Algorithms

Lecture Scribe — based on the lecture transcript and worked examples

Lecture Roadmap

The lecture opens with a **recap of proof techniques** (direct proof, contraposition, contradiction) → introduces **mathematical induction** as a generic, reusable "for-loop" mechanism → applies induction to a real-world **round-robin tournament ordering** argument → connects induction to **recursion** through activation records on the call stack → proves the correctness of **factorial**, **exponentiation**, and **multiplication by repeated addition** using induction → introduces **loop invariants** as the matching correctness tool for iterative code → and finally applies both tools together to **binary search**, **segregation**, **quick sort**, and **merge sort**.

1 Introduction

> *If a claim has to hold for every natural number, how can a finite proof possibly cover infinitely many cases?*

The lecture introduces **mathematical induction** as the answer to this question, and shows how it underlies the correctness of recursive algorithms. Before doing so, the lecture briefly recaps the proof techniques from Lecture 1, since induction is best understood sitting alongside direct proof, contraposition, and contradiction as another way of establishing that a statement is true.

Once induction is introduced formally, it is applied first to a real-life example — ordering the players of a round-robin tournament — and then to a sequence of recursive algorithms: factorial, exponentiation, multiplication, binary search, segregation, quick sort, and merge sort. Alongside recursive correctness, the lecture also introduces **loop invariants**, the matching tool used to argue that an *iterative* piece of code (a loop) computes the correct result.

2 Recap: Implication and Proof Techniques

> *What exactly is being assumed, and what is being contradicted, in each of the standard proof strategies?*

2.1 Implication

Recall the implication $P \to Q$, together with its truth table.

P	Q	$P \to Q$
T	T	T
T	F	F
F	T	T
F	F	T

The table shows a single important pattern: if P is false, the implication is true no matter what Q is; if P is true, the implication is only true when Q is also true. Written out, this is exactly the statement

$$\boxed{(P \to Q) \equiv (\neg P \vee Q).}$$

A second equivalence, also visible directly from the truth table, is the **contrapositive**:

$$\boxed{(P \to Q) \;\equiv\; (\neg Q \to \neg P).}$$

2.2 Direct Proof

In a direct proof of $P \to Q$, the argument does not work with P alone. Alongside the given premise P, the entire background of mathematics is available: a large collection of statements $s_1, s_2, s_3, \ldots$ taken to be true without re-proving them (for example, closure properties of the integers). A direct proof starts from P together with this background and proceeds, step by step, to Q.

2.3 Proof by Contraposition

In contraposition, none of the background statements $s_1, s_2, \ldots$ are touched. Only P and Q are involved: the proof assumes $\neg Q$ and shows that it disproves the given premise, i.e. that it implies $\neg P$. By the equivalence above, showing $\neg Q \to \neg P$ is exactly the same as showing $P \to Q$.

2.4 Proof by Contradiction: A Generalization

Why contradiction is strictly more powerful

Proof by contradiction is normally more powerful than contraposition, because of what it is allowed to contradict. The proof again assumes $\neg Q$, but this time alongside *everything* known to be true — not only the given premise P, but the whole background $s_1, s_2, s_3, \ldots$ that a direct proof would rely on:

$$\boxed{(s_1 \wedge s_2 \wedge \cdots \wedge s_n \wedge P) \wedge \neg Q \;\longrightarrow\; \text{a contradiction.}}$$

A contradiction can now be reached by disproving *any one* of these statements, not necessarily P itself. This is precisely why contradiction generalizes contraposition: a contraposition proof is the special case in which the falsehood reached is $\neg P$; a contradiction proof is free to instead falsify any s_i, written $\neg s_i$, and still conclude that $\neg Q$ must have been wrong.

3 Mathematical Induction

Is there a way to prove a statement for every natural number without ever checking infinitely many cases one at a time?

3.1 Motivation

Suppose the goal is to prove a statement $P(n)$ for every value of n. One approach is to check $n = 0, 1, 2, 3, \ldots$ one at a time, but this can never finish, since there is no way to directly check infinitely many values. Mathematical induction is a tool that proves $P(n)$ for all n without ever checking infinitely many cases individually.

Definition: Mathematical Induction

Mathematical induction is a proof technique that establishes $P(n)$ for every n starting from some starting position, by proving exactly two things: that the statement holds at the starting position, and that whenever it holds at some value k, it also holds at $k + 1$.

3.2 Formal Structure

The two things that must be proven are:

1. **Base Case:** $P(n_0)$ is true, for some starting position n_0. This starting position need not be 0; it is simply wherever the argument begins, and it must be genuinely *proven*, not assumed.
2. **Inductive Step:** $P(k) \to P(k+1)$, for an arbitrary $k \geq n_0$. Here $P(k)$ is **assumed** true — the **inductive hypothesis** — and using it, $P(k+1)$ is proven.

The inductive step is not merely a fact about one particular k: it must be proven for *every* $k \geq n_0$, and the conclusion drawn only covers values from n_0 onward. Making this quantification explicit, the two facts above give:

$$\boxed{P(n_0) \ \wedge \ \big(\forall k \geq n_0,\ P(k) \to P(k+1)\big) \implies \big(\forall n \geq n_0,\ P(n) \text{ is true}\big).}$$

3.3 Why This Is a Complete Proof: Induction as a Generic "For Loop"

A natural question is why proving only these two things establishes $P(n)$ for *every* n. The key is that the inductive step is not a one-time fact about a single number — it is a *generic mechanism* that carries any k to $k+1$. Since the base case is already proven, this mechanism can be reapplied indefinitely:

$$n_0 \to n_0 + 1 \to n_0 + 2 \to n_0 + 3 \to \cdots$$

For example, to prove $P(100)$ starting from a proven base case at 0: apply the mechanism once to go from 0 to 1, again from 1 to 2, again from 2 to 3, and so on, all the way up to 100.

The programmer's reading

This is exactly the behavior of a `for` loop: the mechanism itself is built only once (the inductive step), and then it is run repeatedly. Induction is a mathematical way of expanding a solution outward from a base case, one step at a time — a generic algorithm that, given a solution for k, constructs the solution for $k+1$.

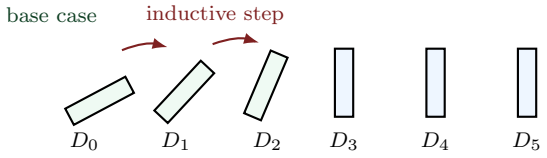
3.4 The Domino Analogy

The same idea is captured by a row of dominoes $D_0, D_1, D_2, \ldots$, spaced so that whenever D_k falls, D_{k+1} is guaranteed to fall too:

$$\boxed{D_k \text{ falls} \implies D_{k+1} \text{ falls.}}$$

This spacing — the guarantee that one falling domino knocks over the next — is exactly the inductive step, $P(k) \to P(k+1)$, proven once for a generic k. The base case is the act of physically knocking over the *first* domino, D_0:

$$\boxed{P(0) \implies \text{the first domino falls} \implies \text{all dominoes will fall.}}$$



So proving a statement by induction always has exactly two responsibilities: prove the base case (knock over the first domino), and prove the generic inductive step (show that the spacing between any two consecutive dominoes is correct).

4 Worked Example: Ordering a Round-Robin Tournament

Before connecting induction to recursive algorithms, it is worth proving a real-life claim by induction, precisely so that induction does not feel like "just a programming trick."

4.1 Problem Statement

Suppose n players have played a round-robin tournament (every player has played every other player exactly once, and no match ends in a draw).

Tournament Ordering Claim

For every $n \geq 2$, the n players can always be arranged in some order $i_1, i_2, \ldots, i_n$ such that each player beat the next player in the order:

$$i_1 \text{ beat } i_2, \qquad i_2 \text{ beat } i_3, \qquad \ldots, \qquad i_{n-1} \text{ beat } i_n.$$

Note carefully what is *not* claimed: nothing is said about who beats whom between non-consecutive players in the ordering. The only guarantee used is that every pair of players has played, so a winner is known for every pair. (The claim is also trivially true for $n = 1$, since there is only one player and no consecutive pair to check; the theorem is stated for $n \geq 2$ because that is precisely the range covered by the base case and inductive step below.)

4.2 Base Case

Take $n = 2$, with players P and S. Since there are no draws, exactly one of two outcomes occurred:

- If S beat P, order them as (S, P).
- If P beat S, order them as (P, S).

In both cases a valid ordering exists, so the base case is proven for $n = 2$.

4.3 Inductive Hypothesis

Assume that for some $k \geq 2$ players, a valid ordering already exists:

$$i_1,\ i_2,\ i_3,\ \ldots,\ i_{k-1},\ i_k \qquad \text{with} \qquad i_1 \text{ beat } i_2,\ i_2 \text{ beat } i_3,\ \ldots,\ i_{k-1} \text{ beat } i_k.$$

This ordering is *assumed*, not reconstructed — exactly as the inductive hypothesis requires. How it was produced is neither known nor needed.

4.4 Inductive Step

A new player, p_m, joins the tournament. Since p_m has played everyone in the existing ordering, p_m's results against i_1 and against i_k are both known. There are three cases.

1. **p_m beat i_1.** Place p_m first: $p_m,\ i_1,\ i_2,\ \ldots,\ i_k$. This is valid, since p_m beat i_1 and the rest of the chain is unchanged.
2. **i_k beat p_m.** Place p_m last: $i_1,\ i_2,\ \ldots,\ i_k,\ p_m$. This is valid, since i_k beat p_m and the rest of the chain is unchanged.
3. **i_1 beat p_m, and p_m beat i_k.** Here p_m belongs somewhere in the middle. Compare p_m against i_2: if p_m beat i_2, insert p_m between i_1 and i_2, since i_1 beat p_m and p_m beat i_2. Otherwise i_2 beat p_m, so compare p_m against i_3 next, repeating the same test. This scan must stop at or before i_k, because p_m is already known to beat i_k; the first index j at which p_m beats i_j (having lost to i_{j-1}) is exactly where p_m is inserted:

$$i_1,\ \ldots,\ i_{j-1},\ p_m,\ i_j,\ \ldots,\ i_k.$$

4.5 Conclusion

In every case, the new ordering of $k + 1$ players is valid. Together with the base case for $n = 2$, this proves by induction that a valid ordering exists for every $n \geq 2$.

Connection to recursion

The inductive hypothesis here plays exactly the same role that a recursive call plays in code: the ordering of the first k players is not re-derived, it is simply trusted as already correct, and the argument focuses only on the one new step of extending it. This is precisely the connection developed in the next section.

5 Recursion and Mathematical Induction

5.1 Recursion is Induction in Programming

Recursive algorithms and mathematical induction have closely matching structures.

Recursion and induction describe closely related processes, viewed from opposite ends. **Recursion** says: to solve a problem, call yourself on a smaller instance, and go down to that smaller instance first, solving it before returning. **Induction** is the matching argument that goes the other way: it is a mathematical way of *expanding* a solution, starting from a base case and building the solution for $k+1$ out of the solution for k. The two are not literally identical — recursion is a computational technique, induction is a proof technique — but for recursively defined algorithms, a recursive call corresponds naturally to the inductive hypothesis, and the base case of the recursion corresponds to the base case of the induction.

During an inductive correctness proof, a recursive call is not re-derived from scratch, step by step. Under the inductive hypothesis, the recursive call is assumed to correctly solve the smaller instance, and the proof shows only that the current call correctly uses that result to solve the larger instance — exactly the way $P(k)$ is assumed while proving $P(k+1)$. This is why, from this point on, a recursive call such as `factorial(n - 1)` is treated as already correct under the inductive hypothesis, and the only remaining work is to show how the answer for n is built from it. (A dry run of the actual execution is still a useful separate exercise for understanding stack behavior, which is the subject of the next subsection.)

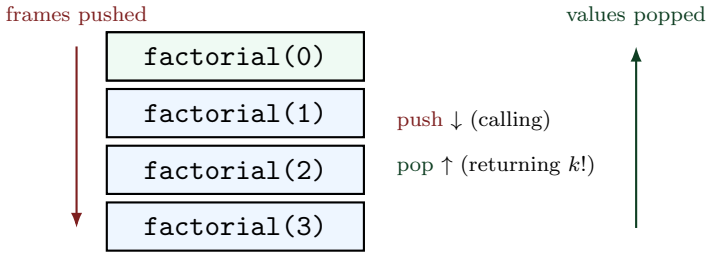
5.2 Stack Trace: Activation Records

The "assume the smaller call is already solved" idea has a concrete counterpart in how a program actually executes. Every recursive call creates a new **activation record** (a stack frame) holding that call's local variables and the point it must resume at. As `factorial(n)` calls `factorial(n - 1)`, a new frame is pushed, and the calling frame is suspended, waiting for the smaller call to return:

Recursive call $\Rightarrow$ push a new activation record; suspend, awaiting the smaller answer.

This continues until the base case is reached, at which point the stack has reached its maximum depth and no further calls are pushed. Once the base case returns, each suspended frame resumes execution, uses the returned value to compute its own result, and returns that result to its caller, removing its own activation record in the process:

Call returns $\Rightarrow$ pop the activation record, returning the result to the caller.



This push-then-pop pattern is the runtime picture of the inductive step: pushing frames down to the base case mirrors unfolding the chain $P(n) \to P(n-1) \to \cdots \to P(0)$, and popping them back up, each one trusting the value it was handed, mirrors applying $P(k) \Rightarrow P(k+1)$ repeatedly on the way back to $P(n)$.

6 Factorial

$$n! = \begin{cases} (n-1)! \times n, & n > 0, \\ 1, & n = 0. \end{cases}$$

6.1 C++ Implementation

```
int factorial(int n)
{
    if (n == 0)
    {
        return 1;
    }
    return n * factorial(n - 1);
}
```

6.2 Correctness Through Induction

Let $P(n)$: `factorial`$(n) = n!$.

- **Base Case:** `factorial`$(0) = 1 = 0!$, so $P(0)$ holds.
- **Inductive Hypothesis:** assume `factorial`$(k-1) = (k-1)!$ for some $k > 0$; this is exactly the value the recursive call is trusted to have already produced.
- **Inductive Step:**

$$\texttt{factorial}(k) = k \cdot \texttt{factorial}(k-1) = k \cdot (k-1)! = k!.$$

Hence, by mathematical induction,

$$\boxed{\texttt{factorial}(n) = n!}$$

for all $n \geq 0$.

Domain and overflow limits

This holds for $n \in \mathbb{Z}_{\geq 0}$; the recursion is not well-defined for negative n, since `n` only ever decreases and the base case `n == 0` would never be reached. The implementation uses `int`, so the equality holds exactly only while $n!$ remains representable in that type; for larger n, `long long` (or an arbitrary-precision type) would be needed.

7 Exponentiation

$$\texttt{power}(a, b) = a^b = \begin{cases} a \times \texttt{power}(a, b-1), & b > 0, \\ 1, & b = 0. \end{cases}$$

7.1 C++ Implementation

```
int power(int a, int b)
{
    if (b == 0)
    {
        return 1;
    }
    return a * power(a, b - 1);
}
```

7.2 Correctness Through Induction

Let $P(b)$: $\texttt{power}(a, b) = a^b$.

- **Base Case:** $\texttt{power}(a, 0) = 1 = a^0$, so $P(0)$ holds.
- **Inductive Hypothesis:** assume $\texttt{power}(a, k-1) = a^{k-1}$ for some $k > 0$.
- **Inductive Step:**

$$\texttt{power}(a, k) = a \cdot \texttt{power}(a, k-1) = a \cdot a^{k-1} = a^k.$$

Hence, by mathematical induction, $\boxed{\texttt{power}(a, b) = a^b}$ for all $b \geq 0$.

Domain and overflow limits

This holds for $a \in \mathbb{Z}$ and $b \in \mathbb{Z}_{\geq 0}$; `power(a, -1)` would never reach the base case, since `b` only ever decreases and there is no base case for negative `b`. As with factorial, the equality holds exactly only while a^b is representable in `int`.

A question worth returning to

This recursive definition performs exactly b multiplications to compute a^b. It is worth keeping in mind, for a later lecture, whether a^b can be computed using noticeably fewer than b multiplications — the answer turns out to be yes, and it becomes an important example once complexity analysis is developed further.

8 Multiplication by Repeated Addition

$$\texttt{mult}(a, b) = \begin{cases} a + \texttt{mult}(a, b-1), & b > 0, \\ 0, & b = 0, \end{cases} \qquad b \geq 0.$$

8.1 C++ Implementation

```
int mult(int a, int b)
{
    if (b == 0)
    {
        return 0;
    }
    return a + mult(a, b - 1);
}
```

Ratio-of-work example

If 9×9 has already been computed, then 9×10 is obtained with a single addition: $9 \times 10 = (9 \times 9) + 9$.

8.2 Correctness Through Induction

Let $P(b)$: $\texttt{mult}(a, b) = a \times b$.

- **Base Case:** $\texttt{mult}(a, 0) = 0 = a \times 0$, so $P(0)$ holds.
- **Inductive Hypothesis:** assume $\texttt{mult}(a, k-1) = a \times (k-1)$ for some $k > 0$.
- **Inductive Step:**

$$\texttt{mult}(a, k) = a + \texttt{mult}(a, k-1) = a + a(k-1) = ak.$$

Hence, by mathematical induction, $\boxed{\texttt{mult}(a, b) = a \times b}$ for $a \in \mathbb{Z}$ and $b \in \mathbb{Z}_{\geq 0}$.

Restricted domain

This algorithm computes multiplication by repeated addition, restricted to nonnegative b: although true integer multiplication $a \times b$ is also defined for negative b, `mult(a, -1)` would never reach the base case, since `b` only ever decreases. As before, the `int` implementation is exact only while $a \times b$ stays representable in that type.

9 Loop Invariants

Recursive algorithms are analyzed with induction. A loop, which has no recursive calls at all, needs the matching tool for *iterative* correctness: the **loop invariant**.

Definition: Loop Invariant

A loop invariant is a property of a program's variables that remains true at the beginning of every iteration of a loop, including before the first iteration. The invariant alone does not establish that the loop terminates: proving an iterative algorithm correct requires three separate ingredients — initialization, maintenance, and termination — and at termination, the invariant together with the loop's exit condition must imply the correct final result.

1. **Initialization:** the property holds before the loop starts.
2. **Maintenance:** the property is preserved by every iteration of the loop.
3. **Termination:** the loop actually terminates, and once it does, the maintained property (together with the reason the loop stopped) gives exactly the answer wanted.

9.1 A Simple Example: Summing an Array

```
int sum = 0;
for (int i = 0; i < size; i++)
{
    sum = sum + arr[i];
}
```

Invariant: before iteration i, `sum` holds the sum of all entries of `arr` with index less than i:

$$\texttt{sum} = \sum_{j<i} \texttt{arr}[j].$$

- **Initialization:** before the loop starts, $i = 0$ and $\texttt{sum} = 0$, exactly the (empty) sum of no elements.
- **Maintenance:** if the invariant holds with $\texttt{sum} = \sum_{j<i} \texttt{arr}[j]$ before an iteration, adding $\texttt{arr}[i]$ and incrementing i gives $\texttt{sum} = \sum_{j<i+1} \texttt{arr}[j]$ before the next iteration, so the invariant is preserved.
- **Termination:** the counter i increases by one each iteration and is bounded by $\texttt{size}$, so the loop terminates; it ends when $i = \texttt{size}$, so the invariant gives

$$\boxed{\texttt{sum} = \sum_{j<\texttt{size}} \texttt{arr}[j]},$$

the sum of the entire array.

9.2 Loop Invariants vs Mathematical Induction

The matching shape of the two arguments

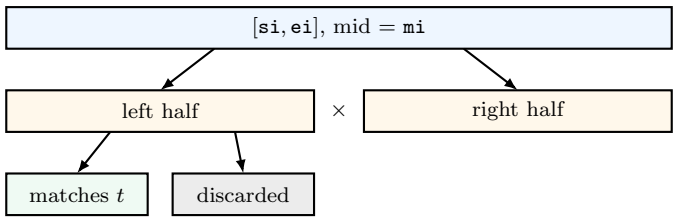
A loop invariant is proven with a similar shape of argument as induction: initialization plays the role of the base case, and maintenance plays the role of the inductive step. The one addition is termination, which has no separate counterpart in the basic induction principle (induction proves a statement for every n, whereas a loop invariant is specifically read off at the moment the loop stops, so termination must be argued separately). This tool is reused repeatedly for the loops inside segregation, quick sort, and merge sort.

10 Binary Search

Binary search operates on a sorted array with indices from $\texttt{si}$ (start index) to $\texttt{ei}$ (end index), searching for a target value t.

10.1 Idea

Check the middle element. If it equals t, return its index. If it is greater than t, recurse on the left half; if it is smaller, recurse on the right half.



10.2 The Midpoint Overflow Bug

A silent overflow

A natural way to compute the middle index is $\texttt{mi} = (\texttt{si} + \texttt{ei})/2$. This can overflow when $\texttt{si}$ and $\texttt{ei}$ are both large: the addition is performed before the division, so the intermediate sum can exceed the range of a 32-bit integer. When that happens, the addition overflows, the sign bit is set, and $\texttt{mi}$ silently becomes a *negative* index. This overflow issue is real and well known — a widely cited instance is a bug once found in Java's `Arrays.binarySearch` implementation — though the exact history varies across languages and libraries, so no specific C-library attribution is made here.

The fix is to compute the distance from `si` first, and only then shift by `si`, so the intermediate value never approaches the overflow range:

$$\boxed{\texttt{mi} = \texttt{si} + \frac{\texttt{ei} - \texttt{si}}{2}.}$$

10.3 C++ Implementation

```
int bin_search(int arr[], int si, int ei, int t)
{
    if (si > ei)
    {
        return -1;
    }

    int mi = si + (ei - si) / 2;

    if (arr[mi] == t)
    {
        return mi;
    }
    else if (arr[mi] > t)
    {
        return bin_search(arr, si, mi - 1, t);
    }
    else
    {
        return bin_search(arr, mi + 1, ei, t);
    }
}
```

The base case `si > ei` represents an empty search interval, and correctly returns -1. When the middle element is not the target, the recursive call excludes the middle element, so the new interval is strictly smaller than the previous one; the recursion therefore always reaches the empty interval, which is exactly what makes it well-defined.

10.4 Correctness Through Induction

Let $n = \texttt{ei} - \texttt{si} + 1$ denote the number of elements in the interval $[\texttt{si}, \texttt{ei}]$, and let $P(n)$ be the statement that `bin_search` correctly searches an interval containing n elements. The base case is $n \leq 0$ (equivalently $\texttt{si} > \texttt{ei}$), an empty interval, which correctly returns -1. The inductive step assumes the recursive call correctly searches the strictly smaller half-interval it is given (either $[\texttt{si}, \texttt{mi} - 1]$ or $[\texttt{mi} + 1, \texttt{ei}]$, each containing strictly fewer than n elements), and the comparison against $\texttt{arr}[\texttt{mi}]$ guarantees the discarded half could not have contained t. Since every recursive call strictly decreases n, and the recursion always terminates once $n \leq 0$, the recursion is well-defined and correct for every $n \geq 0$.

10.5 Complexity

Each call halves the interval and does $O(1)$ work outside the recursive call:

$$T(n) = T\left(\frac{n}{2}\right) + O(1) \quad \Longrightarrow \quad \boxed{T(n) = O(\log n)}.$$

11 Segregation

11.1 Problem Statement

Given an array `arr` and a value t, rearrange the array in place so every value less than t appears before every value greater than or equal to t. Relative order within each group need not be preserved.

Value boundary vs. element boundary

Segregation partitions the array around a *value* t — it does not, by itself, guarantee that any particular *element* of the array ends up at the boundary between the two regions. That distinction matters when segregation is reused inside quick sort in the next section.

11.2 Two Weaker Approaches

Extra space

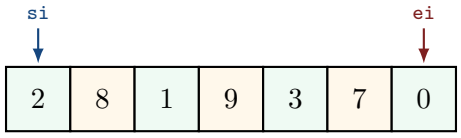
Allocate two new arrays; scan once, placing each element into the "small" array or the "big" array as appropriate. This works, but uses $O(n)$ extra space, and is disallowed when the algorithm must work **in place**.

Repeated adjacent swaps

Repeatedly scan the array, and whenever `arr`$[i] \geq t$ and `arr`$[i+1] < t$, swap them. A single pass is not enough — a value that needs to move several positions only moves one position per pass — so the scan must be repeated until a full pass produces no swaps. This works, but costs $O(n^2)$ time in the worst case, too expensive.

11.3 The Two-Pointer In-Place Algorithm

Keep two indices, `si` starting at the left end and `ei` starting at the right end. Advance `si` rightward past values already $< t$; move `ei` leftward past values already $\geq t$. When both stop, `arr`[`si`] is a value that belongs on the right and `arr`[`ei`] is a value that belongs on the left, so swap them. Repeat until the two indices meet or cross. The two pointers only move inward and never move back outward, so each pointer advances across at most n positions overall; although a position may be examined again after a swap moves a new value into it, the total number of comparisons and pointer movements performed by the two scans is $O(n)$, with at most $n/2$ swaps.



swap `arr`[`si`] ($\geq t$) with `arr`[`ei`] ($< t$), then move both pointers inward

11.4 C++ Implementation and Boundary Conditions

Boundary conditions are genuinely subtle

Without a bound on *each* inner scan, either scan can run past the end of the array if, for example, every remaining value already belongs on one side.

```
int segregate(int arr[], int si, int ei, int t)
{
    while (si < ei)
    {
        while (si < ei && arr[si] < t)
        {
            si++;
```

```
        }
        while (ei > si && arr[ei] >= t)
        {
            ei--;
        }
        if (si < ei)
        {
            swap(arr[si], arr[ei]);
        }
    }
    return si;
}
```

The condition `si < ei` guards all three places it could go wrong: the outer loop, the inner scan from the left (so it cannot run past `ei`), and the inner scan from the right (so it cannot run past `si`). The same guard is placed once more immediately before the swap, since after both inner scans stop it is still possible that `si` and `ei` have already crossed; swapping in that case would corrupt an already-correct pair. `segregate` returns `si`, the boundary index at which the array splits into the two regions. Called on a full array, `segregate(arr, 0, size - 1, t)` partitions the whole array; called on a sub-range `[si, ei]`, it partitions only that sub-range.

11.5 Loop Invariant

$$\boxed{\forall i < \texttt{si},\ \texttt{arr}[i] < t \qquad \text{and} \qquad \forall i > \texttt{ei},\ \texttt{arr}[i] \geq t.}$$

- **Initialization:** initially `si` and `ei` are the two ends of the range, so both parts hold vacuously.
- **Maintenance:** each swap exchanges a misplaced pair so that the element now before `si` satisfies $< t$ and the element now behind `ei` satisfies $\geq t$, after which both indices move inward.
- **Termination:** `si` and `ei` move strictly toward each other on every step in which they change, and the range $[\texttt{si}, \texttt{ei}]$ is finite, so the outer loop terminates; it ends when $\texttt{si} \geq \texttt{ei}$, at which point the invariant guarantees the array is correctly split into a $< t$ region followed by a $\geq t$ region, with `si` as the boundary.

11.6 A Remark on Three-Way Partitioning

An unjustified shortcut

It is tempting to handle values *equal* to t by running `segregate` once with a $\geq$ boundary and then running it again on the resulting right-hand region with an $=$ boundary, assuming this second pass processes only about half the array. This assumption is not justified in general: the values equal to t could be a small fraction of the whole array, or concentrated anywhere within it, so no fixed fraction such as $n/2$ can be assumed for the second pass. The correct three-way partition (into $< t$, $= t$, and $> t$ regions) requires more careful bookkeeping than simply chaining two calls to `segregate` and assuming a balanced split.

12 Quick Sort

Quick sort reuses the two-pointer idea from segregation, but needs a stronger guarantee than segregation alone provides.

12.1 Partition: Placing a Pivot in Its Final Position

A real correctness bug: conflating segregation with partition

Segregation partitions an array around a *value*: it guarantees everything before the returned boundary is $< t$ and everything from the boundary onward is $\geq t$, but does not guarantee any particular element — in particular, the pivot value itself — ends up sitting exactly at that boundary. Quick sort's partition step needs something stronger: it must place the pivot *element* into its final sorted position, so the pivot can be safely excluded from both recursive calls. For example, segregating $[2, 3, 1]$ around the value 2 can produce $[1, 3, 2]$ with boundary index 1; sorting [1] and [2] separately then leaves the middle element 3 untouched, so the array is never actually sorted. This is a genuinely different operation from segregation, and conflating them is a real correctness bug.

A standard way to achieve the stronger guarantee is Lomuto partitioning. A random index is chosen to pick the pivot value, that element is moved to the end of the range, and a single scan then moves every smaller element to the front while tracking a boundary `pi`; finally the pivot is swapped into its correct position at `pi`.

12.2 C++ Implementation

```
#include <cstdlib>
#include <algorithm>

int partition(int arr[], int si, int ei)
{
    int ri = si + rand() % (ei - si + 1);
    swap(arr[ri], arr[ei]);

    int pivot = arr[ei];
    int pi = si;

    for (int i = si; i < ei; i++)
    {
        if (arr[i] < pivot)
        {
            swap(arr[i], arr[pi]);
            pi++;
        }
    }

    swap(arr[pi], arr[ei]);
    return pi;
}
```

The expression `rand() % (ei - si + 1)` produces a value in $[0,\ \texttt{ei} - \texttt{si}]$; adding `si` shifts this into $[\texttt{si}, \texttt{ei}]$, a uniformly random index within the current sub-array. After the pivot value is moved to `arr[ei]`, the scan maintains the invariant that everything before `pi` is $<$ pivot; the final swap places the pivot at index `pi`, now its correct sorted position, since every element before `pi` is smaller and every element between `pi` and `ei` (by construction of the scan) is $\geq$ pivot.

12.3 Quick Sort

```
void quick_sort(int arr[], int si, int ei)
{
    if (si >= ei)
    {
        return;
```

```
    }

    int pi = partition(arr, si, ei);

    quick_sort(arr, si, pi - 1);
    quick_sort(arr, pi + 1, ei);
}
```

Because `arr[pi]` is now known to already be in its final sorted position, it is correct to exclude it from both recursive calls.

12.4 Why Randomize the Pivot?

Robustness against adversarial input

Choosing the pivot randomly, rather than always from a fixed position such as the first or last element, makes the running time far less dependent on adversarial or already-sorted input patterns, and gives $O(n \log n)$ *expected* running time. Randomization does not, however, guarantee a balanced partition on every single run: an unlucky sequence of random choices can still produce a highly unbalanced partition, so the $O(n^2)$ worst case remains possible in principle, just increasingly unlikely as the input grows. The precise reason this matters for the expected running time is best appreciated once complexity analysis of quick sort is covered in detail; for now it is enough to note that this is why `rand()` appears in `partition` rather than a fixed index.

12.5 Correctness Through Induction

The partition step directly guarantees three properties: $\forall i <$ `pi`, `arr`$[i] <$ `pivot`; `arr`[`pi`] $=$ `pivot`; and $\forall i >$ `pi`, `arr`$[i] \geq$ `pivot`. The correctness of `quick_sort` as a whole is a claim about recursive structure, proven by induction on the size of the sub-array: assuming (inductive hypothesis) that `quick_sort` correctly sorts every strictly smaller sub-array, the two recursive calls correctly sort [`si`, `pi` -1] and [`pi` $+1$, `ei`]; the pivot at `pi` is already in its final position; and since every element in the first range is smaller than the pivot and every element in the second range is at least as large, the whole range [`si`, `ei`] ends up sorted.

Where each tool does its job

The scan inside `partition` justifies a single partitioning pass that places one element correctly, while **induction** justifies that repeating this over smaller and smaller sub-arrays sorts the whole array.

12.6 A Note on Duplicate Values

Even with a correct pivot-placing partition, ordinary two-way quick sort (splitting only into $<$ `pivot` and $\geq$ `pivot`) can perform poorly when many elements are equal to the pivot, since all of them land on one side and can repeatedly produce unbalanced partitions. A three-way partition, splitting the array into $<$ `pivot`, $=$ `pivot`, and $>$ `pivot` regions, avoids this degradation. This lecture does not implement three-way quick sort, but the distinction is worth keeping in mind.

12.7 Complexity

When the random pivot happens to split the range into two roughly equal halves:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) \quad \Longrightarrow \quad \boxed{T(n) = O(n \log n)}.$$

In the worst case, when the pivot repeatedly splits off only a single element:

$$T(n) = T(n-1) + O(n) \quad \Longrightarrow \quad \boxed{T(n) = O(n^2)}.$$

Because the pivot is chosen randomly, what actually matters in practice is the *expected* running time, averaged over the random choices made during a single run:

$$\boxed{\mathbb{E}[T(n)] = O(n \log n)}.$$

Case	Time Complexity
Best / balanced partitions	$O(n \log n)$
Expected, with randomized pivot	$O(n \log n)$
Worst case	$O(n^2)$

Randomization reduces the likelihood of consistently unbalanced partitions; it does not eliminate the $O(n^2)$ worst case.

13 Merge and Merge Sort

13.1 Merging Two Sorted Arrays

Given two already-sorted collections `a` and `b`, produce a single sorted collection `c` containing all their elements. Both inputs are taken by `const` reference, so no copies of `a` or `b` are made before merging begins; the only newly allocated memory is the output vector `c`.

```
vector<int> merge(const vector<int>& a, const vector<int>& b)
{
    vector<int> c(a.size() + b.size());
    int ai = 0;
    int bi = 0;

    for (int ci = 0; ci < c.size(); ci++)
    {
        if (ai == a.size())
        {
            for (int j = ci; j < c.size(); j++)
            {
                c[j] = b[bi];
                bi++;
            }
            break;
        }
        if (bi == b.size())
        {
            for (int j = ci; j < c.size(); j++)
            {
                c[j] = a[ai];
                ai++;
            }
            break;
        }
        if (b[bi] < a[ai])
        {
            c[ci] = b[bi];
            bi++;
        }
        else
```

```
        {
            c[ci] = a[ai];
            ai++;
        }
    }
    return c;
}
```

Signed/unsigned comparison

`vector::size()` returns the unsigned type `size_t`, while `ai`, `bi`, and `ci` are declared as `int`; comparisons such as `ci < c.size()` therefore mix signed and unsigned types, which compiles but can trigger compiler warnings. `int` is retained here for simplicity in the lecture presentation, but production code would typically use `size_t` (or an explicit cast) for indices compared directly against container sizes.

On every iteration, exactly one value is written into `c`, so the loop writes `a.size()` + `b.size()` values in total, giving $O(n)$ time. If `a` is exhausted first, the remaining values of `b` are simply appended (and symmetrically if `b` is exhausted first); ties are broken arbitrarily, since either value is a valid next element.

13.2 Loop Invariant for `merge()`

Before iteration `ci` $= i$, `c`$[0..i-1]$ contains the i smallest elements of `a` $\cup$ `b`, in sorted order.

- **Initialization:** before the loop starts, `ci` $= 0$, so `c`$[0..-1]$ is empty, and the invariant holds vacuously.
- **Maintenance:** since `a` and `b` are each individually sorted, the smaller of `a`[`ai`] and `b`[`bi`] is always the next-smallest value not yet placed into `c`; writing it into `c`[`ci`] and advancing the matching index extends the invariant from i to $i+1$.
- **Remainder copying preserves the invariant.** When one input array is exhausted, every remaining element of the other array is necessarily greater than or equal to every element already written into `c`, because both input arrays were individually sorted and every element placed so far came from a comparison against the (now exhausted) other array. Copying the remaining suffix directly into `c`, in order, therefore preserves the invariant — exactly what the two remainder-copying loops above accomplish.
- **Termination:** the index `ci` strictly increases each iteration and is bounded by `c.size()`, so the loop terminates; it ends once `ci` reaches `c.size()` (with either array possibly exhausted along the way, handled by the remainder-copy step), at which point the invariant guarantees all of `c` is filled in sorted order.

13.3 Merge Sort

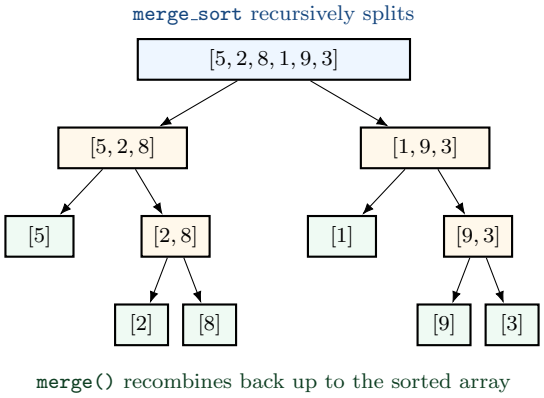
```
void merge_sort(vector<int>& a)
{
    if (a.size() <= 1)
    {
        return;
    }

    vector<int> left(a.begin(), a.begin() + a.size() / 2);
    vector<int> right(a.begin() + a.size() / 2, a.end());

    merge_sort(left);
    merge_sort(right);
```

```
    a = merge(left, right);
}
```

The split uses the standard C++ convention that a range `[begin, end)` excludes its end iterator, so `left` and `right` do not overlap and together cover all of `a`.



13.4 Correctness: A System Argument

Let $P(n)$ be the statement that `merge_sort` correctly sorts every array of size n.

- **Base Case:** for $n \leq 1$, the array is already sorted, so $P(n)$ holds trivially.
- **Inductive Hypothesis:** assume `merge_sort` correctly sorts every array of size smaller than k.
- **Inductive Step:** for an array of size k, `merge_sort` splits it into two smaller halves. By the inductive hypothesis, both recursive calls sort their halves correctly. The loop invariant of `merge()` then guarantees combining these two sorted halves produces a fully sorted array of size k.

Hence, by induction, $\boxed{\texttt{merge\_sort} \text{ correctly sorts arrays of all sizes } n \geq 0.}$

A general correctness pattern

This pattern is useful in general, not just for merge sort: if a small piece of code (here, `merge`) is proven correct on its own terms (here, by a loop invariant), and the way it is *connected* to the rest of the algorithm is proven correct (here, by induction on the recursive structure), then the entire system built from these pieces is correct. This is the same reasoning used, at a larger scale, to argue that bigger software systems work correctly: correctness of the parts, plus correctness of how the parts are wired together.

13.5 Recurrence and Complexity

At every level, the array is split into two subproblems, and the merge step costs $O(n)$:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) \quad \Longrightarrow \quad \boxed{T(n) = O(n \log n)}.$$

Not in-place

This implementation is not in-place: each call allocates `left`, `right`, and (inside `merge`) a new output vector `c`, so the standard analysis also gives an **auxiliary space** complexity of

$$\boxed{O(n)}$$

for the version shown here.

14 Key Takeaways from Lecture 2

1. Mathematical induction proves $P(n)$ for all $n \geq n_0$ by proving a base case $P(n_0)$ and a single, reusable inductive step $\forall k \geq n_0,\ P(k) \Rightarrow P(k+1)$; this single step can be reapplied indefinitely, exactly like the body of a `for` loop, which is why one proof covers infinitely many cases.
2. Proof by contradiction generalizes contraposition: contraposition may only falsify P, while contradiction may falsify any background statement s_i or P itself.
3. The round-robin tournament ordering claim shows induction applied outside of code: a new player is always insertable into an existing valid ordering, by one of three cases (beats the front, loses to the back, or slots in via a linear scan).
4. Recursion and induction have closely matching structures: recursion descends to a smaller instance and solves it first; induction expands a solution outward from a base case. Under the inductive hypothesis, a recursive call's result is trusted rather than re-derived — the same trust placed in $P(k)$ while proving $P(k+1)$. This trust has a concrete runtime picture: pushing activation records down to the base case, then popping them back up while each frame reuses the value it was handed.
5. Factorial, exponentiation, and multiplication by repeated addition are each proven correct by the same three-part pattern: base case, inductive hypothesis (trust the recursive call), inductive step; each is restricted to nonnegative arguments and is exact only while results stay within the chosen integer type.
6. Loop invariants give the matching tool for iterative code, with three separate ingredients: initialization (the property holds before the loop starts), maintenance (it is preserved by every iteration), and termination (the loop actually stops, at which point the invariant implies the correct final result).
7. Binary search's midpoint should be computed as $\texttt{si} + (\texttt{ei} - \texttt{si})/2$ to avoid integer overflow; its recursive correctness and $O(\log n)$ complexity both follow from strictly shrinking the search interval.
8. Segregation's two-pointer algorithm partitions in place around a *value* in $O(n)$ time, but its boundary conditions must guard every inner loop and the swap itself; it does not, by itself, place any specific element at the boundary.
9. Quick sort's partition must place the pivot in its *final* position (not merely segregate around its value) for the recursive exclusion of `pi` to be valid; its randomized pivot gives $O(n \log n)$ expected time but does not eliminate the $O(n^2)$ worst case, and ordinary two-way partitioning degrades on many duplicate values.
10. Merge sort's standard implementation runs in $O(n \log n)$ time with $O(n)$ auxiliary space, since it is not in-place; its correctness combines a loop invariant for `merge()` with induction on the recursive splitting structure — the general pattern of proving parts correct and proving their wiring correct.

One sentence to remember

Induction proves a property across an unbounded chain of cases by building one reusable step from k to $k+1$; recursion and loop invariants are that very same reusable step, written once as a trusted smaller call and once as a trusted previous iteration.

Practice Direction from the Lecture

The next lecture is expected to develop complexity analysis in more depth — formal recurrence solving for divide-and-conquer algorithms such as quick sort and merge sort — and to revisit two threads left open here: whether a^b can be computed in noticeably fewer than b multiplications, and how to implement a genuine three-way partition for arrays with many duplicate values.