NORTH AMERICA PROGRAMMING CAMP ONLINE

NAPC-O

2026 COURSE BOOKLET

Seven institutions. Fourteen assignments.
One shared training program.

Edition 1.2 • September 7, 2026
Prepared for print and digital distribution

NAPC-O 2026 Course Booklet

Edition: 1.2, September 7, 2026.
Sources reviewed: September 7, 2026. See the source register and chapter-level provenance notes.
Coverage: 7 offerings; 14 assignments; 163 problem placements.

Corrections, accessibility requests, and permissions: ICPC Foundation contact page; info@icpc.foundation.

Typeset with LaTeX (pdfLaTeX). Font: Latin Modern family.

Contents

The Training Programs **iv**
0.1 North America Programming Camp . . . iv
0.2 NAPC — Online . . . iv

Weekly Learning Model **v**

2026 Schedule **vi**

Trainer Directory **vii**

How to Use This Booklet **viii**
0.3 Access and participation logistics . . . viii
0.4 Spoilers and instructor material . . . viii
0.5 Problem-planning rubric . . . ix

About the Lecture Guides **x**

Independent Study Protocol **xi**

I Course Catalog 1

1 UCF: Dynamic Programming **2**
1.1 Lecture guide . . . 2
1.2 Technical notes . . . 3
1.3 Advanced workshop . . . 4
1.4 Detailed case study . . . 6
1.5 Study framework . . . 6
1.6 Worked example . . . 7
1.7 Implementation template . . . 8
1.8 Practice lab . . . 9
1.9 DP Problem Set . . . 9
1.10 DP Timed Contest . . . 10

2 Rose-Hulman: Data Structures **15**
2.1 Lecture guide . . . 15
2.2 Technical notes . . . 15
2.3 Advanced workshop . . . 17
2.4 Detailed case study . . . 19
2.5 Study framework . . . 19
2.6 Worked example . . . 20
2.7 Implementation template . . . 21
2.8 Practice lab . . . 22
2.9 Data Structures Problem Set . . . 22
2.10 Data Structures Timed Contest . . . 23

3 UT Austin: Geometry **28**
3.1 Lecture guide . . . 28
3.2 Technical notes . . . 29

3.3 Advanced workshop . . . 30
3.4 Detailed case study . . . 32
3.5 Study framework . . . 32
3.6 Worked example . . . 33
3.7 Implementation template . . . 34
3.8 Practice lab . . . 35
3.9 Geometry Problem Set . . . 35
3.10 Geometry Timed Contest . . . 36

4 McGill: Combinatorics & Number Theory 41
4.1 Lecture guide . . . 41
4.2 Technical notes . . . 42
4.3 Advanced workshop . . . 43
4.4 Detailed case study . . . 45
4.5 Study framework . . . 45
4.6 Worked example . . . 46
4.7 Implementation template . . . 47
4.8 Practice lab . . . 48
4.9 Problem Set . . . 48
4.10 Timed Contest . . . 49

5 Columbia: Gaussian Elimination & FFT 54
5.1 Lecture guide . . . 54
5.2 Technical notes . . . 55
5.3 Advanced workshop . . . 56
5.4 Detailed case study . . . 58
5.5 Study framework . . . 58
5.6 Worked example . . . 59
5.7 Implementation template . . . 60
5.8 Practice lab . . . 61
5.9 Problem Set . . . 61
5.10 Timed Contest . . . 62

6 UIUC: Strings 67
6.1 Lecture guide . . . 67
6.2 Technical notes . . . 67
6.3 Advanced workshop . . . 68
6.4 Detailed case study . . . 71
6.5 Study framework . . . 71
6.6 Worked example . . . 72
6.7 Implementation template . . . 73
6.8 Practice lab . . . 74
6.9 Strings Problem Set . . . 74
6.10 Strings Timed Contest . . . 75

7 MIT: Graphs 80
7.1 Lecture guide . . . 80
7.2 Technical notes . . . 80
7.3 Advanced workshop . . . 82
7.4 Detailed case study . . . 84
7.5 Study framework . . . 84
7.6 Worked example . . . 85
7.7 Implementation template . . . 86
7.8 Practice lab . . . 87
7.9 Problem Set . . . 87
7.10 Timed Contest . . . 88

II Common Contest Reference 93

8 Stress Testing and Differential Checks 94

8.1 Four-component harness . . . 94
8.2 Failure-preserving loop . . . 94
8.3 Harness audit . . . 94

9 Numeric Safety and Boundary Discipline **95**
9.1 Pre-submission numeric audit . . . 95

10 Team Contest Workflow **96**
10.1 Opening scan . . . 96
10.2 Ownership protocol . . . 96
10.3 Submission checklist . . . 96
10.4 Scoreboard discipline . . . 96

11 Post-Contest Review and Curriculum Loop **97**
11.1 Review record . . . 97
11.2 Failure taxonomy . . . 97
11.3 Three-pass learning cycle . . . 97
11.4 Curriculum maintenance . . . 97

III Instructor and Publication Reference **98**

12 Short-Exercise Answer Notes **99**
12.1 UCF: Dynamic Programming . . . 99
12.2 Rose-Hulman: Data Structures . . . 99
12.3 UT Austin: Geometry . . . 99
12.4 McGill: Combinatorics and Number Theory . . . 100
12.5 Columbia: Gaussian Elimination and FFT . . . 100
12.6 UIUC: Strings . . . 100
12.7 MIT: Graphs . . . 100
12.8 Exit-ticket rubric . . . 100

13 Algorithm and Problem Map **101**
13.1 Problem-selection sequence . . . 101

14 Glossary and Acronyms **102**

15 Print Link Directory **103**

Teaching Network and Sources **104**

Publication Notes **105**

The Training Programs

ICPC NORTH AMERICA TRAINING

0.1 North America Programming Camp

The in-person North America Programming Camp (NAPC) is exclusive to NAC-qualified teams and concentrates on championship-level algorithms, data structures, combinatorics, geometry, and team-strategy discussions informed by training records.

Program lead: Nick Wu.

Training team: Jingbang Chen; Zac Friggstad; Yanru Guan; Andrew He; Gennady Korotkevich; Kevin Sun; Nick Wu.

0.2 NAPC — Online

NAPC-O is the accessible online companion: university-led weekly instruction, Kattis practice, timed competition, and recorded discussion. The program is designed as a sustainable training ecosystem that can expand from NAC-bound teams to broader World Finals, regional, and coach-development preparation.

Program lead: Grace Lim.

Organizing committee: Etienne Vouga; Fredrik Niemelä; Grace Lim.

Official NAPC and NAPC-O program page. The source page was last updated January 24, 2026.

Weekly Learning Model

NAPC-O LEARNING MODEL

Beginning January 26, 2026, each week was designed to include one or two recorded lectures, a Kattis problem set, a timed Kattis contest, and one or two recorded interactive discussions. Each lecture is paired with a Markdown note, a one-to-two-hour recorded video, and an optional slide deck.

Activity	Published timing
Lecture 0	Monday or Tuesday
Lecture 1	Wednesday or Thursday
Problem set	Monday 12 a.m. PT release; Friday 9 p.m. PT due
Timed contest	Saturday, 10 a.m.–3 p.m. PT
Discussion	Sunday, 10 a.m. PT

Timing note. The source labels this cadence tentative; the **official program page** remains authoritative for changes.

2026 Schedule

SEVEN WEEKS, SEVEN UNIVERSITY HOSTS

The published schedule connects each host with a focused training topic. Dates are historical; readers should verify any rerun or replacement schedule on the official program page.

Week	Host	Trainer / co-trainer	Topic
Jan. 26–Feb. 1	UCF	Arup Ratan Guha; Sachin Sivakumar; Tyler Marks	Dynamic Programming
Feb. 2–8	Rose-Hulman	Rachel Krohn	Data Structures
Feb. 9–15	UT Austin	Etienne Vouga	Geometry
Feb. 16–22	McGill	David Becerra; Ari Blondal	Combinatorics; Number Theory
Feb. 23–Mar. 1	Columbia	Christian Yongwhan Lim; Josh Alman; Grace Lim	Gaussian Elimination; Fast Fourier Transform
Mar. 2–8	UIUC	Mattox Beckman	Strings
Mar. 9–15	MIT	Jaehyun Koo	Graphs

Names after the lead trainer are listed as co-trainers on the official schedule. See the **official final schedule** for updates.

The course catalog that follows uses this same chronological order. Schedule and trainer information was rechecked on September 7, 2026.

Trainer Directory

EXPERIENCE BEHIND THE CURRICULUM

These concise profiles are paraphrased from the official ICPC North America trainer biographies, reviewed on September 7, 2026. Formal and shortened names are normalized to the forms used by the official profile and schedule pages.

Arup Ratan Guha — UCF

Senior Instructor and longtime UCF ICPC coach; recognized for coaching more than 15 World Finals appearances.

Sachin Sivakumar — UCF

Former USACO and ICPC competitor, twice ninth at NAC and a World Finals participant; now a problem setter and coach.

Tyler Marks — UCF

Former NAC and World Finals competitor, contest problem author, and current UCF programming-team coach.

Rachel Krohn, PhD — Rose-Hulman Institute of Technology

Assistant Professor of Computer Science and Software Engineering; teaches algorithms and data structures and contributes as an ICPC coach, judge, and problem writer.

Etienne Vouga — UT Austin

Associate Professor; South Central Regional Contest Director, NAC Chief Judge, and ICPC Curriculum Committee co-chair.

David Becerra — McGill

Faculty Lecturer teaching programming, algorithms, and competitive programming while coaching McGill ICPC teams.

Ari Blondal — McGill

PhD student in mathematics and computer science with a theoretical-computing and software-development background.

Christian Yongwhan Lim — Columbia

ICPC Foundation Director of Internships, Columbia adjunct and head coach, contest leader, and curriculum organizer.

Josh Alman — Columbia

Associate Professor specializing in algorithms and complexity theory, Columbia ICPC co-coach, and former ICPC World Finals competitor.

Grace Lim — Columbia

Columbia ICPC co-coach and ICPC North America Greater New York regional leader.

Mattox Beckman — UIUC

Teaching Associate Professor, advanced competitive-programming educator, and UIUC ICPC coach.

Jaehyun Koo — MIT

Theoretical CS PhD candidate, elite competitive programmer, MIT coach, and longtime olympiad trainer and problem setter.

Read the complete NAPC-O trainer profiles.

How to Use This Booklet

READER GUIDE

1. **Choose an offering.** Select the university or topic track that matches your training objective.
2. **Start with the problem set.** Review techniques, implementation details, and common edge cases without a clock.
3. **Simulate the contest.** Use the paired timed assignment under contest conditions. Confirm current access and timing on Kattis.
4. **Open the authoritative page.** Every assignment and problem title in this PDF is clickable.
5. **Check licensing before reuse.** Open the problem's Metadata tab for its author, source event, limits, downloads, and displayed license.

Availability note. The assignments were marked ended when this edition was prepared, while offering end dates were displayed as open-ended. Kattis remains authoritative for current availability.

0.3 Access and participation logistics

Participants need a Kattis account and access to the NAPC-O course offering before opening restricted assignment pages. Some lecture folders also require an authorized ICPC Foundation Google account. If a resource asks for access, use the ICPC Foundation contact page; do not request that another participant forward credentials or restricted files.

Published meeting times use Pacific Time. Because daylight-saving rules and local offsets vary, convert the dated event in a timezone-aware calendar before attending. Historical dates in this edition do not create a current registration or meeting invitation. Recording links, passcodes, and participant correspondence are intentionally excluded.

0.4 Spoilers and instructor material

Problem titles, planning metadata, and technique categories are participant-safe. Worked examples and implementation references may reveal reusable ideas, so complete the opening scan and an honest attempt before reading them. Coaches should distribute answer notes only after the corresponding practice window. The timed-contest lists contain links and titles but no copied statements or official solutions.

0.5 Problem-planning rubric

For each problem, mark one learner-relative tier: **Foundation** when the principal technique is already known, **Core** when one modeling step remains, and **Stretch** when the problem combines unfamiliar ideas. Record a target time before starting and revise the tier after review. This keeps the booklet useful when Kattis ratings or assignment membership change.

Problem / slug	F–C–S	Technique hypothesis	Target	Outcome

Review loop. After the attempt, record the first incorrect assumption, the smallest failing test, and the final complexity. If the actual technique differs from the hypothesis, update the catalog annotation rather than erasing the original guess; that mismatch is useful evidence about recognition skills.

About the Lecture Guides

FROM TEACHING MATERIALS TO A PORTABLE STUDY PLAN

The lecture guides paraphrase the supplied slide decks and notes; they do not reproduce slides, speaker notes, recordings, or private correspondence. Each guide is followed by an original technical chapter; an advanced workshop with proof, implementation, testing, discussion, and stretch material; a detailed case study with a complete trace and benchmark plan; prerequisites and learning outcomes; a worked example; pseudocode; a complexity reference; common mistakes; short exercises; a guided Kattis progression; a two-page implementation reference; and a coach lesson plan. A shared contest-reference appendix closes the manual. The complete teaching assets remain in Google Drive and are linked from the corresponding host chapter. Standalone versions of all seven C++17 excerpts are compiled and executed by the project quality checks.

For UCF, Rose-Hulman, UT Austin, and Columbia, the guides are derived directly from course-public lecture files in the shared NAPC-O materials folder. Where no deck was available for UIUC, MIT, or McGill, the booklet labels the section as a *published-course roadmap* and limits it to the official weekly topic and the visible assignment sequence.

Authorized delivery metadata, including recording-asset notices, was reviewed only to confirm the existence and delivery of NAPC-O lecture assets. The recording portal itself was not used as a content source. No private message text, recipient data, recording link, passcode, or unpublished personal information is reproduced here.

Independent Study Protocol

A REPEATABLE CYCLE FOR EVERY HOST TOPIC

Use each university chapter as a three-session learning cycle. The goal is not merely to read an algorithm, but to produce an invariant, a tested implementation, and a record of what changed your approach.

Session	Primary material	Evidence of completion
Understand	Lecture guide; core theory; prerequisites	Restate the model and complexity in your own words; complete the worked example without looking ahead.
Justify	Advanced workshop; proof blueprint	Explain why the model is complete and why each transition or operation preserves the invariant.
Implement	Language notes; pseudocode; complexity table	Produce a language-specific template and a minimal test for every listed hazard.
Challenge	Testing matrix; discussion prompts	Construct adversarial cases, compare alternatives, and identify the constraint that decides the method.
Apply	Exercises; Kattis progression; stretch directions	Solve in order, record failed approaches, then attempt one extension or randomized cross-check.
Review	Submission workflow; complete Drive resources	Compare with the lecture deck or recording, then revise the template and add one stress test.

Suggested pacing. Spend 45–60 minutes on theory and the worked example, 45–90 minutes implementing and testing the template, and a separate contest-style block on the guided problems. A problem that remains unsolved is still productive when its attempted states, counterexamples, and complexity barriers are written down.

Team use. Have one teammate explain the invariant, another challenge edge cases, and a third audit complexity and numeric bounds. Rotate roles between topics so every participant practices explanation, implementation review, and adversarial testing.

Part I

Course Catalog

Chapter 1

UCF: Dynamic Programming

ARUP RATAN GUHA

Each offering pairs a practice problem set with a timed contest. The two assignment pages below link directly to Kattis.

1.1 Lecture guide

Lecture-note outline

Source basis: UCF course-public dynamic-programming notes and the NAPC-O Kattis sequence.

Verification: Files and assignment membership rechecked September 7, 2026.

The UCF notes emphasize that advanced dynamic programming is often won by changing the state or reducing transition cost.

1.1.1 State design and faster transitions

- Analyze running time as the number of reachable states multiplied by the cost of each transition.
- When memory or time is excessive, reformulate the state before attempting low-level optimization.
- Use range-minimum-query structure and other preprocessing when many states ask for closely related transition minima.
- Case studies include Contest Construction, Roller Coaster, Family Visits, Meeting Free Fridays, and Branch Assignment.

1.1.2 Dynamic programming on trees

- Use DFS to define subtree states and combine independent child contributions on an acyclic structure.
- Apply subtree convolution when a parent must distribute a budget or selected-count among several child subtrees.
- Reroot a DP to obtain an answer for every possible root; prefix/suffix combinations support rerooting even when the combine operation is not invertible.
- Use small-to-large merging to control total work, and count fixed-length paths by assigning each path to its lowest common ancestor.

Materials reviewed: NAPCO-DP1-AllNotes.pdf (33 pages); NAPC_O_Tree_D_P_s.pdf (33 pages); Lec-DP1-NAPC-O.docx

1.2 Technical notes

1.2.1 A disciplined DP model

Define a state by the minimum information needed to make every future decision. Then write the recurrence, base cases, evaluation order, and answer extraction before coding. A useful first estimate is

$$\text{time} = \#\text{reachable states} \times \text{average transition cost}, \qquad \text{memory} = \#\text{stored states}.$$

If either estimate is too large, look for symmetry, dominance, a different processing order, or an aggregate that replaces many individual transitions. Rolling arrays reduce memory only when a layer depends on a bounded number of previous layers.

Suppose

$$dp[i] = c_i + \min_{j \in L(i)} (dp[j] + w_j).$$

If $L(i)$ is a sliding interval, a deque may maintain its minimum in amortized $O(1)$. Arbitrary interval minima may call for a segment tree or sparse-table-like preprocessing. If the cost separates into a form such as $m_j x_i + b_j$, convex-hull or Li Chao optimization may be appropriate. Always prove the required monotonicity before using a specialized optimization.

1.2.2 Tree DP and subtree convolution

Root the tree and let each state summarize a processed subtree. For a selection problem, $dp_v[k]$ might store the best result from the subtree of v using exactly k selections. Combining child u is a knapsack convolution:

$$next[a+b] = \text{best}\big(next[a+b],\ dp_v[a] + dp_u[b] + \text{joinCost}(a,b)\big).$$

Allocate arrays only to the current subtree size and merge smaller children first when possible. Although one merge is quadratic in its two array lengths, charging pairs of selected vertices can yield a tighter total bound than multiplying n by the worst local bound.

1.2.3 Rerooting

Many all-roots problems have a downward pass followed by an upward pass. For sum of distances, if $sz[u]$ is the child-subtree size and the tree has n vertices, moving the root across edge vu gives

$$ans[u] = ans[v] + n - 2sz[u].$$

For a general associative combine operation, compute prefix and suffix aggregates of the child contributions. The contribution passed from v to child u_i combines the parent-side value with every child except u_i in constant time after preprocessing. No inverse operation is required.

1.2.4 Small-to-large merging

When each subtree maintains a map, set, or frequency table, preserve the largest child's container and insert every smaller container into it. Each element moves only when its destination size at least doubles, so it moves $O(\log n)$ times. With logarithmic map insertion this typically gives $O(n \log^2 n)$; hash tables or problem-specific arrays may reduce the factor.

1.2.5 DP review checklist

- State meaning includes whether a vertex, edge, or boundary condition is selected.
- Base cases represent an empty prefix or leaf consistently.
- Impossible states use a sentinel that cannot overflow when combined.
- Every transition is counted once; subtree combinations use a fresh buffer.
- Complexity is justified over all merges, not only for one state.

1.3 Advanced workshop

1.3.1 Recognizing when dynamic programming is the right model

A dynamic program is justified when many candidate solutions share the same unfinished subproblem. The decisive modeling step is to identify exactly what the future needs to know about the past. If two partial solutions have identical future-relevant information, they belong to the same state; everything else should be discarded. This equivalence-class viewpoint is a practical test for whether a state is sufficient and whether it contains unnecessary dimensions. On trees, the parent-child boundary usually supplies that information. A subtree can report a compact summary to its parent because no edge connects it to the rest of the tree except through the parent. When the summary also contains a selected-count, distance, or budget, combining children becomes a knapsack-style convolution. Before coding, estimate the sum of all merge sizes; multiplying a worst-case bound by every vertex often exaggerates or conceals the true cost.

1.3.2 Correctness-proof blueprint

1. Define the state as an optimum over a precise set of feasible partial solutions, including every boundary condition.
2. For each transition, prove both directions: every constructed candidate is feasible, and every optimal solution is represented by some transition.
3. Choose an induction order that matches evaluation: prefix length for sequences, postorder for subtrees, or increasing mask size for subsets.

1.3.3 Implementation notes across languages

- In C++17, use a named constant well below `LLONG_MAX` for impossible states and guard before addition.
- In Java, allocate primitive arrays when possible; nested object tables can make an otherwise valid DP exceed memory limits.
- In Python, prefer dictionaries only for genuinely sparse states and avoid copying a full map at every tree edge.
- For reconstruction, store the chosen transition separately from the numeric optimum so tie-breaking remains explicit.

1.3.4 Adversarial testing matrix

Case	Construction	What it exposes
Minimal structure	One item or one vertex	Base cases and initialization
Forced choice	Only one feasible transition	State semantics
Symmetric optima	Several equal best answers	Tie handling and reconstruction
Path and star trees	Opposite tree shapes	Recursion depth and merge cost

1.3.5 Discussion prompts

1. Which pieces of history can be removed from the state without changing any legal continuation?
2. Can a transition minimum be maintained incrementally or answered with a range-query structure?
3. What family of inputs makes the nominal state-times-transition bound tight?

1.3.6 Stretch directions

- Reroot the same subtree summary to compute an answer for every root.
- Replace a dense child convolution with small-to-large merging and compare aggregate work.
- Add solution reconstruction and define a deterministic rule for equal optima.

1.4 Detailed case study

1.4.1 Tree selection with an adjacency restriction

Scenario. Given a weighted tree, select a set of vertices of maximum total weight such that no selected vertices share an edge. Use the five-vertex tree with edges 1–2, 1–3, 3–4, 3–5 and weights $[4, 5, 3, 6, 2]$. Root the tree at vertex 1.

1.4.2 Step-by-step trace

1. For every vertex v, define $dp[v][1]$ as the best value in its subtree when v is selected and $dp[v][0]$ when it is not.
2. Leaves give $(dp[2][0], dp[2][1]) = (0, 5)$, $(dp[4][0], dp[4][1]) = (0, 6)$, and $(dp[5][0], dp[5][1]) = (0, 2)$.
3. At vertex 3, selecting it forbids both children: $dp[3][1] = 3 + 0 + 0 = 3$. Skipping it permits each child to choose independently: $dp[3][0] = 6 + 2 = 8$.
4. At vertex 1, $dp[1][1] = 4 + dp[2][0] + dp[3][0] = 12$. Also $dp[1][0] = \max(0, 5) + \max(8, 3) = 13$.
5. The optimum is 13, reconstructed as vertices 2, 4, and 5. The reconstruction follows whichever child state supplied each maximum.

1.4.3 Correctness argument

Induct on subtree height. If v is selected, every child must be excluded, so the first transition enumerates all feasible solutions of that class. If v is excluded, the subtrees are independent and each child may use its better state. These two classes partition all feasible solutions in the subtree of v.

1.4.4 Benchmark plan

Family	Scale or construction	Purpose
Path	$n = 2 \cdot 10^5$	Recursion depth and iterative postorder
Star	One center, many leaves	High-degree aggregation
Random tree	Compare for $n \leq 20$	Brute-force correctness oracle

1.4.5 Coach-check questions

1. How would the state change if selected vertices had to be at distance at least three?
2. Which additional data must be stored to count the number of optimal selections?
3. Why does the same two-state transition fail on a general graph with cycles?

1.5 Study framework

1.5.1 Prerequisites

- Recursion or iterative DFS on rooted trees.
- One-dimensional and two-dimensional dynamic programming.
- Range-query structures and amortized complexity.

1.5.2 Learning outcomes

- Define a state whose information is sufficient and minimal.
- Recognize when state reduction or transition acceleration is the real task.
- Combine subtree states and reroot answers without double counting.

1.6 Worked example

1.6.1 Maximum independent set on a tree

For each vertex v, let $dp[v][1]$ be the maximum number of selected vertices in the subtree of v when v is selected, and $dp[v][0]$ the value when it is not. A selected vertex forbids every child, while an unselected vertex permits either child state:

$$dp[v][1] = 1 + \sum_{u \text{ child of } v} dp[u][0], \qquad dp[v][0] = \sum_{u \text{ child of } v} \max(dp[u][0], dp[u][1]).$$

Leaves have values $(0, 1)$. A postorder traversal guarantees that every child is known before its parent. The final answer is $\max(dp[root][0], dp[root][1])$.

This example illustrates state sufficiency. The future only needs to know whether the boundary vertex v is selected; the exact configuration deeper in the subtree is irrelevant once its optimum has been summarized. If the task additionally required exactly k selected vertices, the state would gain a count dimension and child combination would become knapsack convolution.

To reconstruct a solution, remember which child choice attained each maximum, then traverse from the chosen root state. Reconstruction data is often omitted during first implementation; deciding whether the problem asks only for a value prevents unnecessary memory.

1.7 Implementation template

1.7.1 Pseudocode

```
function dfs(v, parent):
    dp[v][0] = 0
    dp[v][1] = 1
    for u in adj[v]:
        if u == parent: continue
        dfs(u, v)
        dp[v][1] += dp[u][0]
        dp[v][0] += max(dp[u][0], dp[u][1])

answer = max(dp[root][0], dp[root][1])

# Generic child convolution for an exact-count state
next = impossible
for a in current counts:
    for b in child counts:
        next[a+b] = best(next[a+b], cur[a] + child[b])
cur = next
```

1.7.2 Complexity reference

Method	Bound	Best fit
Fixed states per tree edge	$O(n)$	Independent set and include/exclude DPs.
Tree knapsack	Often $O(nk^2)$ or tighter	Exact-count subtree selections.
Rerooting with prefix/suffix	$O(n)$	Constant-size associative contributions.
Small-to-large maps	$O(n \log^2 n)$ typical	Subtree frequency or set summaries.

1.7.3 Common mistakes

- Defining a state in prose that does not match its initialization.
- Updating a knapsack array in place and reusing one child multiple times.
- Using an impossible-state sentinel that overflows when a cost is added.
- Claiming a per-merge bound as the total complexity.

1.8 Practice lab

1.8.1 Short exercises

1. Extend the independent-set recurrence to vertex weights.
2. Write states for minimum vertex cover on a tree and compare the transitions.
3. Derive the reroot formula for the sum of distances when the root crosses one edge.

1.8.2 Guided problem progression

1. **Airport Coffee**. Practice explicit state meaning and transition order.
2. **Range Editing**. Look for transition aggregation rather than nested scanning.
3. **Zuma**. Model an interval or compressed-state DP carefully.
4. **Everything Is A Nail**. Classify the state boundary before optimizing.

1.8.3 Submission workflow

1. Write the invariant, state meaning, or mathematical precondition before coding.
2. Build minimal examples for every boundary case identified in the chapter.
3. Compare against a brute-force solver on small random instances whenever practical.
4. For a small tree, compare every DP state with exhaustive subset enumeration and verify reconstruction independently.
5. After acceptance, record the decisive idea, complexity, and one implementation hazard.

Complete lecture resources
Open the UCF materials folder in Google Drive. The folder is the authoritative location for complete decks, notes, and future revisions. Access may require an authorized ICPC Foundation account.

1.9 DP Problem Set

12 problems • Instructor(s): Arup Ratan Guha

Assignment: Open on Kattis

Planning key: mark each problem Foundation, Core, or Stretch after an opening scan; record a personal target time before starting. Kattis difficulty and availability can change, so this booklet does not freeze a platform rating.

1. Advertising ICPC — kattis: advertisingicpc

tier: F / C / S • target: ____ min • technique: _____

2. Airport Coffee — kattis: airportcoffee

tier: F / C / S • target: ____ min • technique: _____

3. Ascending Photo — kattis: ascendingphoto

tier: F / C / S • target: ____ min • technique: _____

4. Cut It Out! — kattis: cutitout
tier: F / C / S • target: ___ min • technique: _____

5. Explosion Exploit — kattis: explosion
tier: F / C / S • target: ___ min • technique: _____

6. Heaps of Fun — kattis: heapsoffun
tier: F / C / S • target: ___ min • technique: _____

7. Leaderboard Effect — kattis: leaderboardeffect
tier: F / C / S • target: ___ min • technique: _____

8. Matrix Fraud — kattis: matrixfraud
tier: F / C / S • target: ___ min • technique: _____

9. Matryoshka — kattis: matryoshka
tier: F / C / S • target: ___ min • technique: _____

10. On-Call Team — kattis: oncallteam
tier: F / C / S • target: ___ min • technique: _____

11. Range Editing — kattis: rangeediting
tier: F / C / S • target: ___ min • technique: _____

12. Zuma — kattis: zuma
tier: F / C / S • target: ___ min • technique: _____

Tip. Open Metadata on a problem page to verify author, source, limits, downloads, and license before redistributing content.

1.10 DP Timed Contest

12 problems • Instructor(s): Arup Ratan Guha

Assignment: Open on Kattis

Planning key: mark each problem Foundation, Core, or Stretch after an opening scan; record a personal target time before starting. Kattis difficulty and availability can change, so this booklet does not freeze a platform rating.

1. ABC String — kattis: abcstring
tier: F / C / S • target: ___ min • technique: _____

2. Apocalypse Attack! — kattis: apocalypse
tier: F / C / S • target: ___ min • technique: _____

3. Black and White — kattis: blackandwhite
tier: F / C / S • target: ___ min • technique: _____

4. Surely You Congest — kattis: congest
tier: F / C / S • target: ___ min • technique: _____

5. Cram — kattis: cram
tier: F / C / S • target: ___ min • technique: _____

6. Everything Is A Nail — kattis: everythingisanail
tier: F / C / S • target: ___ min • technique: _____

7. Fair Division — kattis: fairdivision2

tier: F / C / S • target: ___ min • technique: _____

8. GCD Harmony — kattis: gcdharmony

tier: F / C / S • target: ___ min • technique: _____

9. Islands from the Sky — kattis: islandsfromthesky

tier: F / C / S • target: ___ min • technique: _____

10. Magic Bean — kattis: magicbeancube

tier: F / C / S • target: ___ min • technique: _____

11. Riddle of the Sphinx — kattis: riddleofthesphinx

tier: F / C / S • target: ___ min • technique: _____

12. Tomb Raider — kattis: tombraider2

tier: F / C / S • target: ___ min • technique: _____

Tip. Open Metadata on a problem page to verify author, source, limits, downloads, and license before redistributing content.

1.11 Implementation reference

1.11.1 C++17 tested reference excerpt

A reusable postorder tree-DP skeleton for maximum-weight independent set. Replace the two scalar states with vectors when the problem adds a selected-count or budget.

```
using ll = long long;
vector<vector<int>> adj;
vector<array<ll,2>> dp;

void dfs(int v, int parent) {
    dp[v][0] = 0;
    dp[v][1] = weight[v];
    for (int u : adj[v]) if (u != parent) {
        dfs(u, v);
        dp[v][0] += max(dp[u][0], dp[u][1]);
        dp[v][1] += dp[u][0];
    }
}

dfs(root, -1);
ll answer = max(dp[root][0], dp[root][1]);
```

1.11.2 Template contract

- $dp[v][b]$ summarizes exactly the rooted subtree of v under boundary choice b.
- Every undirected edge is processed once from parent to child.
- Impossible states, if added, must be checked before arithmetic.

1.11.3 Porting and diagnostics

Language notes

- Java: use `long[][] dp` and iterative postorder when recursion can exceed the stack.
- Python: build a parent array iteratively, then process vertices in reversed traversal order.
- C++: reserve adjacency storage when degrees are known and use `long long` for accumulated weights.

Diagnostic probes

1. One vertex of negative, zero, and positive weight.
2. A path, star, and balanced tree with the same number of vertices.
3. Random trees up to 20 vertices checked against all subsets.
4. A deepest-possible tree to test traversal-stack safety.

Performance envelope

Measure	Bound	Interpretation
Time	$O(n)$	One constant-size merge per edge.
Memory	$O(n)$	Adjacency, parent stack, and two states.
Numeric	$O(nW)$	Use 64 bits when weights reach W.

Submission audit

- Match every array dimension and numeric type to the largest legal instance.
- Run the diagnostic probes before optimization, then preserve them as regression tests.
- State the invariant and the complexity in the solution notes before submission.

1.12 Coach lesson plan

1.12.1 Ninety-minute session objective

Derive a correct tree-DP state and defend its merge complexity.

1.12.2 Agenda

Time	Activity	Evidence
0–10 min	Retrieval warm-up	Learners restate the chapter invariant without notes.
10–25 min	Model critique	Teams compare two candidate models and reject one with a counterexample.
25–45 min	Guided trace	The class completes the detailed case study and predicts each intermediate state.
45–70 min	Implementation lab	Pairs adapt the reference skeleton and run at least three diagnostic probes.
70–90 min	Debrief and transfer	Learners identify the constraint that selects the method on a new problem.

1.12.3 Misconceptions to surface

- State meaning differs from initialization.
- A child is accidentally reused during convolution.
- Recursion depth is ignored.

1.12.4 Instructor checkpoints

1. Ask for a counterexample to an incomplete state, predicate, or graph model.
2. Require one learner to justify correctness while another audits numeric and asymptotic bounds.
3. Do not reveal the final transition until teams can name the invariant it must preserve.

1.12.5 Exit ticket

1. In at most 25 words, state what each DP state remembers and why that information is sufficient.
2. Name a tree or ordering that breaks a greedy or incomplete-state solution.
3. Identify the first state dimension or bound that makes the current transition too slow.

Chapter 2

Rose-Hulman: Data Structures

RACHEL KROHN

Each offering pairs a practice problem set with a timed contest. The two assignment pages below link directly to Kattis.

2.1 Lecture guide

Lecture-deck outline

Source basis: Rose-Hulman course-public Data Structures deck (62 slides) and the NAPC-O Kattis sequence.

Verification: Deck identity and assignment membership rechecked September 7, 2026.

This lecture connects standard-library containers to the specialized structures that recur in contest solutions.

2.1.1 Containers, range queries, and connectivity

- Select arrays, dynamic containers, queues, stacks, priority queues, sets, maps, multisets, bitmasks, and custom comparators by the operations a problem requires.
- Begin static range-sum work with prefix sums; move to segment trees for logarithmic queries and updates, adding lazy propagation for range updates.
- Use Fenwick trees for compact logarithmic prefix aggregates and point updates, driven by least-significant-bit index arithmetic.
- Represent disjoint sets as forests and combine union by rank with path compression for near-constant amortized connectivity operations.
- Treat integer overflow and floating-point precision as data-structure design constraints, not afterthoughts.

Materials reviewed: Data Structures.pptx (62 slides)

2.2 Technical notes

2.2.1 Choose by operations, not by habit

An array gives constant-time indexing and excellent locality. A stack or queue expresses a restricted processing order. A heap supports insertion and extreme-element removal in $O(\log n)$ but not efficient arbitrary deletion.

Ordered sets and maps support predecessor, successor, and sorted iteration in $O(\log n)$; hash tables offer expected constant-time lookup without order. A Boolean array or bitset can beat either when the key universe is small.

2.2.2 Prefix sums and Fenwick trees

For a static array, $P[i] = \sum_{0 \le j < i} a[j]$ answers a range sum by $P[r] - P[l]$. A Fenwick tree makes both point updates and prefix queries logarithmic. In one-based indexing, the node at i summarizes the interval ending at i with length

$$\text{lowbit}(i) = i \,\&\, (-i).$$

A query repeatedly subtracts lowbit; an update repeatedly adds it. Fenwick trees work whenever the maintained aggregate supports the needed subtraction or inverse for a range query. Two Fenwick trees can support range additions and range-sum queries through the identity for a piecewise-linear prefix sum.

2.2.3 Segment trees and lazy propagation

A segment tree stores an aggregate for each interval in a balanced binary partition. Point update and range query visit $O(\log n)$ nodes. The combine operation should be associative and have an identity, making min, max, sum, gcd, and small custom records natural choices.

For a range update, attach a lazy tag to a node whose entire interval is covered. Applying a tag updates the node aggregate and composes the pending tag; pushing sends the tag to children before descending. Composition order matters for noncommutative updates such as assignment followed by addition. Define three functions explicitly: combine child aggregates, apply a tag to an aggregate of known length, and compose old and new tags.

2.2.4 Disjoint-set union

DSU represents each component by a rooted forest. Path compression flattens the find path; union by size attaches the smaller tree below the larger, while union by rank attaches the lower-rank root below the higher-rank root. A sequence of operations runs in $O(m\alpha(n))$, effectively linear for contest constraints.

Kruskal's minimum-spanning-tree algorithm sorts edges by weight and adds an edge exactly when its endpoints are currently in different components. DSU also supports offline connectivity, component-size tracking, and "next unprocessed position" tricks. It does not support arbitrary edge deletion or path aggregates without additional machinery.

2.2.5 Structure selection checklist

- Static range aggregate: prefix sums or sparse table, depending on whether an inverse exists.
- Point updates plus prefix aggregate: Fenwick tree.
- General interval queries or range updates: segment tree, possibly lazy.
- Dynamic component merging: DSU.
- Sliding minimum or maximum: monotone deque.
- Coordinate values are large but sparse: compress coordinates before array-based structures.

2.3 Advanced workshop

2.3.1 Deriving a data structure from an operation contract

A data-structure problem should begin with a contract, not a remembered template. List every update and query, whether operations arrive online, and whether the aggregate is invertible. Prefix sums win when there are no updates. Fenwick trees exploit invertible prefix aggregation. Segment trees handle a more general associative combine operation, while lazy propagation is needed when a range update must be summarized without visiting every leaf. The algebra of the operation explains the implementation. Associativity permits a tree to combine intervals in any parenthesization. An identity value represents an empty interval. Inverses allow one prefix aggregate to be removed from another, but minimum and maximum have no useful inverse. Disjoint-set union solves a different contract: components only merge, so it cannot support arbitrary deletions without an offline transformation or a more powerful dynamic-connectivity method.

2.3.2 Correctness-proof blueprint

1. State the invariant for every stored node or index, including the exact interval it represents.
2. Prove that an update preserves the invariant locally, then use the tree structure to extend the argument to all ancestors.
3. For a query, show that returned pieces are disjoint, cover the requested range, and combine to the required aggregate.

2.3.3 Implementation notes across languages

- Use half-open intervals consistently in iterative segment trees; mixing them with inclusive recursion is a common off-by-one source.
- For Fenwick trees, document whether external indices are zero-based while the internal array is one-based.
- In disjoint-set union, keep component metadata only at representatives and merge it in the same branch that links roots.
- Use 64-bit storage for sums even when each individual value fits in 32 bits.

2.3.4 Adversarial testing matrix

Case	Construction	What it exposes
Single position	Repeated update and query	Index conversion
Whole range	Query $[0, n)$	Root aggregate
Overlapping updates	Nested and crossing intervals	Lazy composition order
Redundant union	Join an existing component	Cycle safety and metadata

2.3.5 Discussion prompts

1. Which algebraic properties does the requested aggregate have: associativity, identity, inverse, or idempotence?
2. Does the problem require online answers, or can sorting and offline processing remove a difficult operation?
3. What is the simplest structure meeting the contract within the constraints?

2.3.6 Stretch directions

- Extend a range-sum tree to maintain both sum and maximum under range addition.
- Implement an order-statistics query by descending a frequency Fenwick or segment tree.
- Solve a deletion problem offline by reversing time and using disjoint-set union.

2.4 Detailed case study

2.4.1 Tracing lazy range addition and range sum

Scenario. Maintain the array $[2, 1, 3, 0, 4, 5, 1, 2]$ under range addition and range-sum queries. Use half-open intervals. Apply $+3$ to $[2, 7)$, query $[1, 6)$, then apply -2 to $[0, 4)$ and query the whole array.

2.4.2 Step-by-step trace

1. The initial root sum is 18. The first update covers canonical nodes rather than five individual leaves; each covered node increases by 3 times its interval length and stores a pending tag.
2. After the first update the conceptual array is $[2, 1, 6, 3, 7, 8, 4, 2]$, whose sum is 33. A query descends only where its requested range partially intersects a node.
3. The sum on $[1, 6)$ is $1 + 6 + 3 + 7 + 8 = 25$. Before descending through a tagged node, push its addition to both children so their aggregates become current.
4. Applying -2 to $[0, 4)$ reduces the root sum by $2 \cdot 4 = 8$, producing 25. Lazy tags compose by addition, so no ordering ambiguity occurs.
5. The final whole-range query returns the root aggregate immediately; it does not need to push every pending tag to the leaves.

2.4.3 Correctness argument

Each node stores the exact aggregate of its interval after all updates that fully cover it, even if the effect has not reached its children. A push preserves the parent aggregate while distributing the pending operation. Every query partitions its range into disjoint stored intervals, so summing their exact aggregates returns the requested value.

2.4.4 Benchmark plan

Family	Scale or construction	Purpose
Single index	Alternating update/query	Boundary conversion
Full range	$2 \cdot 10^5$ operations	Lazy root handling
Random differential	$n \leq 100$	Compare with a plain array

2.4.5 Coach-check questions

1. How would tag composition change for range assignment followed by range addition?
2. Which node fields are needed to maintain both sum and maximum?
3. When would two Fenwick trees be simpler than this segment tree?

2.5 Study framework

2.5.1 Prerequisites

- Arrays, recursion, binary representations, and logarithms.

- Associative operations and identity elements.
- Basic graph connectivity and sorting.

2.5.2 Learning outcomes

- Choose a structure from the required operations and update pattern.
- Implement Fenwick trees, segment trees, and DSU from invariants.
- Recognize when an offline ordering simplifies a dynamic problem.

2.6 Worked example

2.6.1 One array, three levels of range support

Suppose $a = [3, 1, 4, 1, 5]$. A static prefix array is $P = [0, 3, 4, 8, 9, 14]$, so the half-open range sum $a[1..4)$ is $P[4] - P[1] = 6$. This is optimal when no updates occur.

If point values change, a Fenwick tree partitions each prefix into binary-sized suffix blocks. Updating position i touches $i, i + lowbit(i), \ldots$; querying a prefix walks in the opposite direction. The representation is compact and usually the best choice for invertible prefix aggregates.

If updates and queries are arbitrary intervals, a segment tree stores the sum of each node interval. Query $[l, r)$ by accepting nodes fully inside the range, rejecting disjoint nodes, and splitting partial overlap. For adding x to a whole interval, a lazy tag stores the unpushed update. Applying it to a sum node of length len increases the stored sum by $x \cdot len$.

The lesson is not that one structure is strongest. The prefix array has the smallest constant and simplest proof; the Fenwick tree trades a little generality for compactness; the segment tree exposes the most flexible interval algebra.

2.7 Implementation template

2.7.1 Pseudocode

```
# Fenwick tree, one-based indexing
function add(i, delta):
    while i <= n:
        bit[i] += delta
        i += i & -i

function prefix(i):
    result = 0
    while i > 0:
        result += bit[i]
        i -= i & -i
    return result

function range_sum(l, r):       # inclusive
    return prefix(r) - prefix(l-1)

# DSU core
find(x):
    if parent[x] != x: parent[x] = find(parent[x])
    return parent[x]
```

2.7.2 Complexity reference

Method	Bound	Best fit
Prefix array	$O(1)$ query, $O(n)$ update	Static data.
Fenwick tree	$O(\log n)$	Point updates and prefix/range aggregates.
Segment tree	$O(\log n)$	General interval queries and updates.
DSU	$O(\alpha(n))$ amortized	Component merges and connectivity.

2.7.3 Common mistakes

- Mixing zero-based external indices with a one-based Fenwick implementation.
- Forgetting interval length when applying a lazy addition to a sum node.
- Composing lazy tags in the wrong chronological order.
- Using DSU for deletions or path queries that it cannot represent.

2.8 Practice lab

2.8.1 Short exercises

1. List the Fenwick indices visited by update(6) and prefix(13).
2. Define the lazy-tag composition for range assignment followed by range addition.
3. Prove that union by size alone keeps tree height logarithmic.

2.8.2 Guided problem progression

1. **Fenwick Tree**. Build confidence with bit-index arithmetic by implementing a basic Fenwick tree.
2. **Union-Find**. Implement a basic DSU structure with path compression and union by size.
3. **Almost Union-Find**. Adapt DSU to support element movement, set size, and set sum.
4. **Movie Collection**. Combine ordering with a dynamic prefix structure.

2.8.3 Submission workflow

1. Write the invariant, state meaning, or mathematical precondition before coding.
2. Build minimal examples for every boundary case identified in the chapter.
3. Compare against a brute-force solver on small random instances whenever practical.
4. Compare every range update and query with a plain array, including empty, singleton, and full-range intervals.
5. After acceptance, record the decisive idea, complexity, and one implementation hazard.

Complete lecture resources
Open the Rose-Hulman materials folder in Google Drive. The folder is the authoritative location for complete decks, notes, and future revisions. Access may require an authorized ICPC Foundation account.

2.9 Data Structures Problem Set

13 problems • Instructor(s): Rachel Krohn

Assignment: Open on Kattis

Planning key: mark each problem Foundation, Core, or Stretch after an opening scan; record a personal target time before starting. Kattis difficulty and availability can change, so this booklet does not freeze a platform rating.

1. Administrative Difficulties — kattis: administrativeproblems

tier: F / C / S • target: ____ min • technique: _____

2. Almost Union-Find — kattis: almostunionfind

tier: F / C / S • target: ____ min • technique: _____

3. Binary search tree — kattis: bst

tier: F / C / S • target: ____ min • technique: _____

4. Fenwick Tree — kattis: fenwick
tier: F / C / S • target: ___ min • technique: _____

5. Getting Through — kattis: gettingthrough
tier: F / C / S • target: ___ min • technique: _____

6. Juggler — kattis: juggler
tier: F / C / S • target: ___ min • technique: _____

7. Mårten's Theorem — kattis: more10
tier: F / C / S • target: ___ min • technique: _____

8. Movie Collection — kattis: moviecollection
tier: F / C / S • target: ___ min • technique: _____

9. Pyro Tubes — kattis: pyro
tier: F / C / S • target: ___ min • technique: _____

10. RATS — kattis: rats
tier: F / C / S • target: ___ min • technique: _____

11. Semi-prime H-numbers — kattis: hnumbers
tier: F / C / S • target: ___ min • technique: _____

12. Union-Find — kattis: unionfind
tier: F / C / S • target: ___ min • technique: _____

13. Worst Weather Ever — kattis: worstweather
tier: F / C / S • target: ___ min • technique: _____

Tip. Open Metadata on a problem page to verify author, source, limits, downloads, and license before redistributing content.

2.10 Data Structures Timed Contest

13 problems • Instructor(s): Rachel Krohn

Assignment: Open on Kattis

Planning key: mark each problem Foundation, Core, or Stretch after an opening scan; record a personal target time before starting. Kattis difficulty and availability can change, so this booklet does not freeze a platform rating.

1. Association for Control Over Minds — kattis: control
tier: F / C / S • target: ___ min • technique: _____

2. Association of Cats and Magical Lights — kattis: magicallights
tier: F / C / S • target: ___ min • technique: _____

3. BASIC Interpreter — kattis: basicinterpreter
tier: F / C / S • target: ___ min • technique: _____

4. Chess Tournament — kattis: chesstournament
tier: F / C / S • target: ___ min • technique: _____

5. Clock Construction — kattis: clockconstruction
tier: F / C / S • target: ___ min • technique: _____

6. Cookie Selection — kattis: cookieselection

tier: F / C / S • target: ___ min • technique: _____

7. Dictionary Attack — kattis: dictionaryattack

tier: F / C / S • target: ___ min • technique: _____

8. Eko — kattis: eko

tier: F / C / S • target: ___ min • technique: _____

9. Folded Map — kattis: foldedmap

tier: F / C / S • target: ___ min • technique: _____

10. Jewel Heist — kattis: jewelheist

tier: F / C / S • target: ___ min • technique: _____

11. Number Sets (Hard) — kattis: numbersetshard

tier: F / C / S • target: ___ min • technique: _____

12. Orphan Backups — kattis: orphanbackups

tier: F / C / S • target: ___ min • technique: _____

13. Program — kattis: program

tier: F / C / S • target: ___ min • technique: _____

Tip. Open Metadata on a problem page to verify author, source, limits, downloads, and license before redistributing content.

2.11 Implementation reference

2.11.1 C++17 tested reference excerpt

A recursive lazy segment-tree core for range addition and range sums. The separate apply, push, and pull operations make the invariant auditable.

```
struct Node { long long sum=0, lazy=0; };
vector<Node> st(4*n);

void apply(int p, int l, int r, long long x) {
    st[p].sum += x * (r-l);
    st[p].lazy += x;
}
void push(int p, int l, int r) {
    if (!st[p].lazy || r-l == 1) return;
    int m=(l+r)/2;
    apply(2*p,l,m,st[p].lazy);
    apply(2*p+1,m,r,st[p].lazy);
    st[p].lazy=0;
}
void add(int p,int l,int r,int ql,int qr,long long x){
    if(qr<=l || r<=ql) return;
    if(ql<=l && r<=qr){ apply(p,l,r,x); return; }
    push(p,l,r); int m=(l+r)/2;
    add(2*p,l,m,ql,qr,x); add(2*p+1,m,r,ql,qr,x);
    st[p].sum=st[2*p].sum+st[2*p+1].sum;
}
```

2.11.2 Template contract

- Intervals are half-open everywhere: node p represents $[l, r)$.
- `sum` already includes the node lazy tag; children may not yet include it.
- Every descent pushes first and every returning update pulls once.

2.11.3 Porting and diagnostics

Language notes

- Java: parallel primitive arrays for sum and lazy avoid millions of small node objects.
- Python: iterative trees are often safer for tight limits; PyPy recursion still has overhead.
- C++: keep interval length in the apply function to prevent duplicated arithmetic.

Diagnostic probes

1. Empty, one-element, prefix, suffix, and full-range updates.
2. Positive and negative tags composed on the same node.
3. Random operations compared with a direct array.
4. Values near numeric limits with maximum interval length.

Performance envelope

Measure	Bound	Interpretation
Update	$O(\log n)$	Canonical interval decomposition.
Query	$O(\log n)$	For one contiguous range.
Memory	$O(n)$	At most four tree nodes per element.

Submission audit

- Match every array dimension and numeric type to the largest legal instance.
- Run the diagnostic probes before optimization, then preserve them as regression tests.
- State the invariant and the complexity in the solution notes before submission.

2.12 Coach lesson plan

2.12.1 Ninety-minute session objective

Choose a structure from an operation contract and state its invariant.

2.12.2 Agenda

Time	Activity	Evidence
0–10 min	Retrieval warm-up	Learners restate the chapter invariant without notes.
10–25 min	Model critique	Teams compare two candidate models and reject one with a counterexample.
25–45 min	Guided trace	The class completes the detailed case study and predicts each intermediate state.
45–70 min	Implementation lab	Pairs adapt the reference skeleton and run at least three diagnostic probes.
70–90 min	Debrief and transfer	Learners identify the constraint that selects the method on a new problem.

2.12.3 Misconceptions to surface

- Half-open and closed intervals are mixed.
- Lazy tags are pushed but not composed.
- A more complicated structure is chosen than necessary.

2.12.4 Instructor checkpoints

1. Ask for a counterexample to an incomplete state, predicate, or graph model.
2. Require one learner to justify correctness while another audits numeric and asymptotic bounds.
3. Do not reveal the final transition until teams can name the invariant it must preserve.

2.12.5 Exit ticket

1. In at most 25 words, derive the data structure from its required updates and queries.
2. Name an operation sequence that exposes a missing push, pull, or index conversion.
3. Identify the first operation or bound that requires a different structure.

Chapter 3

UT Austin: Geometry

ETIENNE VOUGA

Each offering pairs a practice problem set with a timed contest. The two assignment pages below link directly to Kattis.

3.1 Lecture guide

Lecture-deck outline

Source basis: UT Austin course-public geometry decks and the NAPC-O Kattis sequence.

Verification: Deck identities and assignment membership rechecked September 7, 2026.

The geometry lectures use difficult contest problems to motivate a compact, robust geometry toolkit.

3.1.1 Robust planar geometry

- Model polygon and segment questions with orientation tests, cross products, and segment–segment intersection predicates.
- Handle degeneracies deliberately: vertex hits, collinear or coincident edges, valid endpoints, and segments that merely touch the boundary.
- For line–polygon feasibility, sort all edge intersections, split the line into intervals, and classify interval midpoints with point-in-polygon tests.
- Treat floating-point behavior as part of the algorithm: know the exact-integer range of IEEE double precision and separate exact predicates from approximate construction.

3.1.2 Circles and angular sweeps

- Transform circle-placement constraints by inflating obstacles and walls by the candidate radius.
- Binary-search the answer radius, then convert circle intersections into forbidden angular intervals around each obstacle.
- Sweep interval endpoints to detect an uncovered tangent placement; use atan2 and a doubling trick to manage wraparound at zero radians.
- Derive circle–circle intersections either algebraically or with the law of cosines, checking nonintersection, containment, and tangency first.

Materials reviewed: Geometry1.pptx (49 slides); Geometry3.pptx (30 slides)

3.2 Technical notes

3.2.1 Predicates before constructions

Let a, b, c be points in the plane. The signed orientation value

$$\text{orient}(a, b, c) = (b_x - a_x)(c_y - a_y) - (b_y - a_y)(c_x - a_x)$$

is positive for a counterclockwise turn, negative for a clockwise turn, and zero for collinearity. This one predicate supports convex hulls, segment intersection, polygon orientation, and many point-in-polygon implementations.

For closed segments ab and cd, first test whether their bounding boxes overlap. In the general noncollinear case, they intersect when c and d lie on opposite sides of the line through ab and a and b lie on opposite sides of the line through cd. The collinear case must separately test one-dimensional interval overlap. Decide in advance whether touching at an endpoint counts.

Key idea. Keep decisions exact whenever possible. If the input coordinates are integers and products fit in a wider integer type, compute orientation exactly. Use floating point for constructed coordinates, distances, and angles, not for a sign test that can remain integral.

3.2.2 Line–polygon feasibility

To test the portions of an infinite line that lie inside a polygon, intersect the line with every polygon edge and sort the resulting line parameters. Consecutive parameters define open intervals whose classification is constant unless the line overlaps an edge. Test a midpoint from each interval with a point-in-polygon routine. This avoids trying to encode every vertex-touch case in a single intersection-count formula.

A ray-crossing point-in-polygon test toggles parity each time a horizontal ray crosses an edge. Use a half-open rule for edge endpoints so a polygon vertex is counted once rather than twice. If boundary membership matters, run an explicit point-on-segment test first.

3.2.3 Circle placement as monotone feasibility

Suppose a new circle of radius r must avoid existing circles with radii R_i. Inflate each existing obstacle to radius $R_i + r$ and offset each wall inward by r. A position is feasible exactly when its center avoids all inflated obstacles. If feasibility is monotone in r, binary search converts an optimization problem into repeated decision problems.

For a candidate center constrained to a circle around obstacle i, every other obstacle cuts out a forbidden angular interval. If two centers are separated by distance d, the boundary angle can be obtained from

$$\cos\theta = \frac{d^2 + (R_i + r)^2 - (R_j + r)^2}{2d(R_i + r)}.$$

Use atan2 for the center direction, clamp the cosine argument to $[-1, 1]$, and split any interval that crosses 0. Duplicating endpoints with an added 2π is often simpler than special-casing wraparound throughout the sweep.

3.2.4 Geometry implementation checklist

- State whether boundaries are open or closed and whether tangency is legal.
- Compare squared distances when no actual distance is required.
- Scale the tolerance to the magnitude of the values being compared; a universal epsilon is rarely universal.

- Normalize angles only at API boundaries. Internally, permit an interval representation that is easy to sweep.
- Test collinearity, duplicate points, zero-length segments, containment, tangency, and nearly parallel lines.

3.3 Advanced workshop

3.3.1 Separating exact decisions from approximate constructions

Robust computational geometry treats predicates and constructions differently. A predicate answers a discrete question such as clockwise versus counterclockwise or intersecting versus disjoint. Whenever coordinates are integral and bounds permit, compute that decision exactly with cross products. A construction produces a coordinate, angle, or distance and often requires floating point. Keeping exact decisions upstream prevents a tiny rounding error from changing the combinatorial structure of the solution. Many geometry optimizations become manageable after exposing monotonicity. For a candidate radius, inflate forbidden objects and ask only whether a placement exists. If feasibility can change only from true to false as the radius grows, binary search isolates the optimum. The inner feasibility test must still handle tangency, coincident endpoints, angular wraparound, and the distinction between an open and closed forbidden interval.

3.3.2 Correctness-proof blueprint

1. Express every orientation or side-of-line claim as the sign of a cross product and account explicitly for zero.
2. When binary searching, prove monotonicity and define whether equality belongs to the feasible or infeasible side.
3. For an angular sweep, prove that every forbidden placement corresponds to an interval and that the event order maintains the active count correctly.

3.3.3 Implementation notes across languages

- Promote integer coordinates before multiplication; a 32-bit product can overflow before assignment to a 64-bit result.
- Use one angle normalization convention, such as $[0, 2\pi)$, and split wrapped intervals rather than improvising comparisons.
- Compare squared distances when only ordering matters and delay square roots until coordinates are actually required.
- Choose an epsilon relative to problem scale; never use it to blur an exact integer predicate.

3.3.4 Adversarial testing matrix

Case	Construction	What it exposes
Collinear points	Overlapping and disjoint segments	Zero cross products
Endpoint touch	One shared endpoint	Open versus closed boundaries
Near tangency	Distance close to radius sum	Numeric stability
Angle wrap	Interval crossing zero	Sweep normalization

3.3.5 Discussion prompts

1. Which decisions can remain exact even if the final answer is floating point?

2. What degeneracy changes the combinatorial event order?
3. Can the optimization be rewritten as a monotone decision problem?

3.3.6 Stretch directions

- Replace floating-point orientation with an exact wide-integer implementation and establish safe coordinate limits.
- Build a reusable segment-intersection classifier that distinguishes proper crossing, touching, overlap, and disjointness.
- Add randomized comparisons between an angular sweep and a slow sampled feasibility checker.

3.4 Detailed case study

3.4.1 Classifying segment intersection without floating point

Scenario. Classify the intersection of segment AB from $(0, 0)$ to $(6, 4)$ and segment CD from $(1, 4)$ to $(5, 0)$. Then modify D to $(7, 4)$ and explain why the classification changes.

3.4.2 Step-by-step trace

1. Compute $orient(A, B, C) = \text{cross}(B - A, C - A) = 20 > 0$ and $orient(A, B, D) = -20 < 0$ for the first pair.
2. Compute $orient(C, D, A) = -20 < 0$ and $orient(C, D, B) = 20 > 0$. Each segment has the other segment endpoints on opposite sides of its supporting line.
3. Because all four orientations are nonzero and both sign products are negative, the segments have a proper interior crossing. No coordinate construction is needed.
4. With $D = (7, 4)$, both C and D lie above AB, so the first sign product is nonnegative and a proper crossing is impossible.
5. If any orientation were zero, test whether the corresponding point lies in the closed bounding box. That separate branch distinguishes touching, overlap, and disjoint collinearity.

3.4.3 Correctness argument

For noncollinear endpoints, a segment crosses the supporting line of the other exactly when its endpoints have opposite orientation signs. Requiring this condition in both directions is necessary and sufficient for a proper crossing. Zero orientations are excluded from that proof and handled by the on-segment predicate.

3.4.4 Benchmark plan

Family	Scale or construction	Purpose
Grid exhaustive	Coordinates in $[-3, 3]$	Symmetry and degeneracy
Large coordinates	Near input bounds	Integer overflow
Near-collinear float	Perturb by 10^{-12}	Need for exact predicates

3.4.5 Coach-check questions

1. What maximum coordinate magnitude is safe when cross products use signed 64-bit integers?
2. How should the classifier report an overlapping subsegment rather than a single point?
3. Which parts of the algorithm would change for open segments that exclude endpoints?

3.5 Study framework

3.5.1 Prerequisites

- Vectors, dot products, cross products, and parametric lines.
- Binary search on a monotone predicate and careful asymptotic analysis.
- Comfort with integer overflow, floating-point error, and boundary cases.

3.5.2 Learning outcomes

- Implement exact orientation and robust segment intersection.
- Reduce polygon and circle-placement questions to ordered one-dimensional events.
- Choose intentionally between integer, rational, and floating-point computation.

3.6 Worked example

3.6.1 From a geometric picture to a reliable predicate

Consider two closed segments ab and cd. A drawing may suggest that they cross, but the implementation must also classify endpoint contact and collinear overlap. Compute

$$o_1 = \text{orient}(a, b, c), \quad o_2 = \text{orient}(a, b, d), \quad o_3 = \text{orient}(c, d, a), \quad o_4 = \text{orient}(c, d, b).$$

If o_1 and o_2 have opposite signs and o_3 and o_4 have opposite signs, the segments cross properly. If an orientation is zero, test whether that point lies inside the other segment's bounding box. Thus the policy for closed segments is explicit: proper crossing, endpoint contact, and collinear overlap all count.

Now suppose the input coordinates are bounded by 10^9. Each coordinate difference can approach $2 \cdot 10^9$, and a product can approach $4 \cdot 10^{18}$. Subtracting two such products can exceed signed 64-bit range. The picture has not changed, but the correct implementation type has: use a wider integer type for the predicate or prove a tighter bound from the actual constraints.

For a maximum-radius placement problem, define $feasible(r)$ by inflating every obstacle by r. If a radius works, every smaller radius works, so the feasible set is an interval. Binary search is valid only after this monotonicity argument. Within $feasible$, interval endpoints are angles; the geometric problem has become a coverage sweep on $[0, 2\pi)$.

3.7 Implementation template

3.7.1 Pseudocode

```
function orient(a, b, c):
    return cross(b - a, c - a) using a wide integer type

function on_segment(a, b, p):
    return orient(a,b,p) == 0 and
           min(a.x,b.x) <= p.x <= max(a.x,b.x) and
           min(a.y,b.y) <= p.y <= max(a.y,b.y)

function intersects(a, b, c, d):
    o1 = orient(a,b,c); o2 = orient(a,b,d)
    o3 = orient(c,d,a); o4 = orient(c,d,b)
    if opposite(o1,o2) and opposite(o3,o4): return true
    return on_segment(a,b,c) or on_segment(a,b,d) or
           on_segment(c,d,a) or on_segment(c,d,b)

lo = known_feasible; hi = known_infeasible
repeat enough iterations:
    mid = (lo + hi) / 2
    if feasible(mid): lo = mid
    else: hi = mid
```

3.7.2 Complexity reference

Method	Bound	Best fit
Orientation / segment test	$O(1)$	Exact local predicates.
Convex hull	$O(n \log n)$	Sort once, then maintain turns.
Angular interval sweep	$O(n \log n)$	Sort interval endpoints.
Binary search + sweep	$O(n \log n \log(1/\varepsilon))$	Continuous optimization with tolerance ε.

3.7.3 Common mistakes

- Using an epsilon in an integer orientation test.
- Taking a square root when squared distances are sufficient.
- Sorting angles with a comparator that is not transitive near the branch cut.
- Binary-searching before proving which side of the answer is feasible.

3.8 Practice lab

3.8.1 Short exercises

1. Give coordinates for every segment-intersection case: proper crossing, endpoint contact, collinear overlap, and disjoint collinear segments.
2. Prove that obstacle inflation makes circle-placement feasibility monotone in the new radius.
3. Design a half-open convention that counts a polygon vertex once in a ray-crossing test.

3.8.2 Guided problem progression

1. **Criss-Cross**. Start with orientation, intersection policy, and degenerate cases.
2. **Floating Points**. Audit numeric range and precision decisions.
3. **Drawing Circles**. Apply obstacle inflation, angular intervals, and monotone search.
4. **Largest Triangle**. Finish with a larger geometric optimization problem.

3.8.3 Submission workflow

1. Write the invariant, state meaning, or mathematical precondition before coding.
2. Build minimal examples for every boundary case identified in the chapter.
3. Compare against a brute-force solver on small random instances whenever practical.
4. Test every predicate on endpoint contact, collinear overlap, reversed orientation, and coordinates near numeric limits.
5. After acceptance, record the decisive idea, complexity, and one implementation hazard.

Complete lecture resources
Open the UT-Austin materials folder in Google Drive. The folder is the authoritative location for complete decks, notes, and future revisions. Access may require an authorized ICPC Foundation account.

3.9 Geometry Problem Set

12 problems • Instructor(s): Etienne Vouga

Assignment: Open on Kattis

Planning key: mark each problem Foundation, Core, or Stretch after an opening scan; record a personal target time before starting. Kattis difficulty and availability can change, so this booklet does not freeze a platform rating.

1. Around the Track — kattis: aroundthetrack

tier: F / C / S • target: ____ min • technique: _____

2. Criss-Cross — kattis: crisscross

tier: F / C / S • target: ____ min • technique: _____

3. Cut It Out! kattis: cutitout

tier: F / C / S • target: ___ min • technique: _____

4. Drawing Circles kattis: drawingcircles1

tier: F / C / S • target: ___ min • technique: _____

5. Floating Points kattis: floatingpoints

tier: F / C / S • target: ___ min • technique: _____

6. Great Fireball kattis: greatfireball

tier: F / C / S • target: ___ min • technique: _____

7. Kiwi Trees kattis: kiwitrees

tier: F / C / S • target: ___ min • technique: _____

8. Missile Defense kattis: missiledefense

tier: F / C / S • target: ___ min • technique: _____

9. Planetary Grid kattis: planetarygrid

tier: F / C / S • target: ___ min • technique: _____

10. How many squares? kattis: squares

tier: F / C / S • target: ___ min • technique: _____

11. 3D Printer kattis: threedprinter

tier: F / C / S • target: ___ min • technique: _____

12. Unreal Estate kattis: unrealestate

tier: F / C / S • target: ___ min • technique: _____

Tip. Open Metadata on a problem page to verify author, source, limits, downloads, and license before redistributing content.

3.10 Geometry Timed Contest

12 problems • Instructor(s): Etienne Vouga

Assignment: Open on Kattis

Planning key: mark each problem Foundation, Core, or Stretch after an opening scan; record a personal target time before starting. Kattis difficulty and availability can change, so this booklet does not freeze a platform rating.

1. Arable Area kattis: arable

tier: F / C / S • target: ___ min • technique: _____

2. Cavli kattis: cavli

tier: F / C / S • target: ___ min • technique: _____

3. Firing the Phaser kattis: firingphaser

tier: F / C / S • target: ___ min • technique: _____

4. Freestyle Masonry kattis: freestylemasonry

tier: F / C / S • target: ___ min • technique: _____

5. Hole in One kattis: holeinone

tier: F / C / S • target: ___ min • technique: _____

6. Kingdom of Kittens — kattis: kingdomofkittens

tier: F / C / S • target: ___ min • technique: _____

7. Largest Triangle — kattis: largesttriangle

tier: F / C / S • target: ___ min • technique: _____

8. Mountainous Landscape — kattis: mountainouslandscape

tier: F / C / S • target: ___ min • technique: _____

9. Sculpture — kattis: sculpture

tier: F / C / S • target: ___ min • technique: _____

10. Square Bounce — kattis: squarebounce

tier: F / C / S • target: ___ min • technique: _____

11. Snakes — kattis: snakes

tier: F / C / S • target: ___ min • technique: _____

12. Structural Integrity — kattis: structuralintegrity

tier: F / C / S • target: ___ min • technique: _____

Tip. Open Metadata on a problem page to verify author, source, limits, downloads, and license before redistributing content.

3.11 Implementation reference

3.11.1 C++17 tested reference excerpt

Exact integer predicates for orientation, on-segment testing, and complete closed-segment intersection classification.

```
using i128 = __int128_t;
struct P { long long x, y; };

i128 cross(P a, P b, P c) {
    return (i128)(b.x-a.x)*(c.y-a.y)
         - (i128)(b.y-a.y)*(c.x-a.x);
}
int sgn(i128 x) { return (x>0)-(x<0); }
bool box(P a, P b, P p) {
    return min(a.x,b.x)<=p.x && p.x<=max(a.x,b.x)
        && min(a.y,b.y)<=p.y && p.y<=max(a.y,b.y);
}
bool intersects(P a,P b,P c,P d){
    i128 x1=cross(a,b,c), x2=cross(a,b,d);
    i128 x3=cross(c,d,a), x4=cross(c,d,b);
    if(x1==0 && box(a,b,c)) return true;
    if(x2==0 && box(a,b,d)) return true;
    if(x3==0 && box(c,d,a)) return true;
    if(x4==0 && box(c,d,b)) return true;
    return sgn(x1)*sgn(x2)<0 && sgn(x3)*sgn(x4)<0;
}
```

3.11.2 Template contract

- Inputs are integral and products fit signed 128-bit arithmetic.
- Intersection includes endpoint touching and collinear overlap.
- No epsilon is used in a discrete integer predicate.

3.11.3 Porting and diagnostics

Language notes

- Java: use `BigInteger` if 64-bit cross products can overflow; otherwise prove the input bound.
- Python: integers are unbounded, so the direct formula is exact but object overhead is higher.
- C++: promote before subtraction and multiplication, not after an overflow has occurred.

Diagnostic probes

1. Proper crossing, endpoint touch, T-junction, overlap, and disjoint collinearity.
2. Zero-length segments treated as points.
3. All permutations of each segment endpoint order.
4. Coordinates at positive and negative input limits.

Performance envelope

Measure	Bound	Interpretation
Predicate	$O(1)$	Four cross products and box tests.
Batch	$O(q)$	For q independent segment pairs.
Numeric	$O(C^2)$	Cross magnitude for coordinate bound C.

Submission audit

- Match every array dimension and numeric type to the largest legal instance.
- Run the diagnostic probes before optimization, then preserve them as regression tests.
- State the invariant and the complexity in the solution notes before submission.

3.12 Coach lesson plan

3.12.1 Ninety-minute session objective

Separate exact predicates from approximate geometric constructions.

3.12.2 Agenda

Time	Activity	Evidence
0–10 min	Retrieval warm-up	Learners restate the chapter invariant without notes.
10–25 min	Model critique	Teams compare two candidate models and reject one with a counterexample.
25–45 min	Guided trace	The class completes the detailed case study and predicts each intermediate state.
45–70 min	Implementation lab	Pairs adapt the reference skeleton and run at least three diagnostic probes.
70–90 min	Debrief and transfer	Learners identify the constraint that selects the method on a new problem.

3.12.3 Misconceptions to surface

- An epsilon replaces an exact zero test.
- Integer products overflow before promotion.
- Tangency and overlap are treated as ordinary crossing.

3.12.4 Instructor checkpoints

1. Ask for a counterexample to an incomplete state, predicate, or graph model.
2. Require one learner to justify correctness while another audits numeric and asymptotic bounds.
3. Do not reveal the final transition until teams can name the invariant it must preserve.

3.12.5 Exit ticket

1. In at most 25 words, distinguish the exact predicate from any approximate construction.
2. Give coordinates for a degeneracy that a picture-based solution is likely to miss.
3. Identify the coordinate bound at which the chosen numeric type becomes unsafe.

Chapter 4

McGill: Combinatorics & Number Theory

ARI BLONDAL; DAVID BECERRA

Each offering pairs a practice problem set with a timed contest. The two assignment pages below link directly to Kattis.

4.1 Lecture guide

Published-course roadmap

Source basis: Official weekly topic, trainer listing, and visible NAPC-O Kattis assignment sequence; no lecture deck was available.

Verification: Public course metadata rechecked September 7, 2026; this roadmap is editorial synthesis, not a lecture transcript.

This roadmap follows the published Combinatorics and Number Theory topic and the associated Kattis problem sequence.

4.1.1 Counting and arithmetic toolkit

- Identify whether a problem needs direct counting, inclusion–exclusion, recurrence relations, or a dynamic program over combinatorial states.
- Use gcd, extended Euclid, modular inverses, fast exponentiation, and the Chinese remainder theorem only under their required preconditions.
- Precompute factorials and inverse factorials for repeated binomial-coefficient queries under a suitable modulus.
- Factor constraints drive method choice: trial division, sieving, prime-exponent accounting, and divisor enumeration cover different input ranges.

Materials reviewed: No lecture deck was present in the shared materials folder or relevant attachment search at review time.

4.2 Technical notes

4.2.1 Euclid and modular inverses

Euclid's algorithm uses $\gcd(a, b) = \gcd(b, a \bmod b)$. Extended Euclid also finds x, y with $ax + by = g$, where $g = \gcd(a, b)$. The congruence $ax \equiv 1 \pmod{m}$ has an inverse exactly when $\gcd(a, m) = 1$; the coefficient x from extended Euclid is one such inverse after normalization.

Binary exponentiation computes $a^e \bmod m$ in $O(\log e)$. If m is prime and $a \not\equiv 0$, Fermat gives a^{m-2} as an inverse. Do not use this shortcut for a composite modulus without a valid totient-based argument.

Two congruences $x \equiv a \pmod{m}$ and $x \equiv b \pmod{n}$ are compatible when $a \equiv b \pmod{\gcd(m, n)}$. The Chinese remainder theorem then combines them modulo $\operatorname{lcm}(m, n)$. This generalized condition matters when moduli are not coprime.

4.2.2 Primes, factors, and divisors

The sieve of Eratosthenes lists primes through N in $O(N \log \log N)$. A smallest-prime-factor table permits fast factorization of many values not exceeding N. For one moderate value, trial division need only test through its current square root; after removing a prime factor repeatedly, any remaining value greater than one is prime.

If $n = \prod p_i^{e_i}$, then the number of divisors is $\prod(e_i + 1)$ and their sum is $\prod(1 + p_i + \cdots + p_i^{e_i})$. Generate divisors recursively from the prime powers rather than scanning all integers up to n.

4.2.3 Binomial coefficients and counting

For prime modulus p and $n < p$, precompute factorials and inverse factorials:

$$\binom{n}{k} \equiv fact[n]\, invfact[k]\, invfact[n-k] \pmod{p}.$$

For composite moduli or $n \geq p$, the denominator may not be invertible; use Pascal DP, prime-exponent accounting, Lucas-type methods, or CRT as the constraints require.

Inclusion–exclusion counts objects avoiding forbidden properties:

$$\left| U \setminus \bigcup_i A_i \right| = \sum_S (-1)^{|S|} \left| \bigcap_{i \in S} A_i \right|.$$

The formula is simple, but the task is to evaluate intersections efficiently, often by a bitmask, lcm, or compressed state. When subsets are too numerous, look for symmetry, a transform, or a recurrence.

4.2.4 Counting checklist

- Decide whether order matters, repetition is allowed, and objects are distinguishable.
- Normalize every modular subtraction; promote products before reducing.
- Verify inverse preconditions and whether the modulus is prime.
- Separate counting all constructions from counting equivalence classes under symmetry.
- Test $k = 0$, empty products, impossible constraints, and the all-selected case.

4.3 Advanced workshop

4.3.1 Turning arithmetic conditions into structural information

Number-theory solutions become clearer when every computation is tied to a theorem and its preconditions. The extended Euclidean algorithm does more than compute a gcd: its coefficients describe every integer linear combination of two moduli. A modular inverse exists precisely when the gcd is one. A generalized Chinese remainder merge exists precisely when the residue difference is divisible by the gcd. Writing these tests beside the code prevents formulas from being used outside their domain. Prime factorization converts multiplicative questions into independent choices of exponents. The divisor count, divisor sum, Euler phi function, and many combinatorial counts follow by multiplying local contributions. Under a modulus, division is multiplication by an inverse only when that inverse exists. This distinction matters for factorial formulas, composite moduli, and intermediate values divisible by the modulus.

4.3.2 Correctness-proof blueprint

1. Derive an arithmetic formula from prime exponents or Bezout coefficients before translating it into code.
2. For congruence merging, prove compatibility, construct one solution, and show that all solutions differ by the least common multiple.
3. For inclusion-exclusion, identify the universe and prove that an object satisfying t properties contributes exactly once.

4.3.3 Implementation notes across languages

- Normalize residues after every operation so language-specific negative remainders do not escape into later formulas.
- Divide by the gcd before multiplying moduli to reduce overflow risk in a generalized CRT merge.
- Use modular multiplication techniques or wider integer types when the product can exceed the native range.
- Precompute smallest prime factors when many values from a bounded range must be factored.

4.3.4 Adversarial testing matrix

Case	Construction	What it exposes
Noncoprime compatible	$x = 2 \pmod 6$, $x = 8 \pmod 9$	Generalized CRT
Noncoprime impossible	$x = 2 \pmod 6$, $x = 4 \pmod 9$	Compatibility check
Prime power	$n = p^k$	Exponent formulas
Large near-square	Two close prime factors	Factorization limits

4.3.5 Discussion prompts

1. What theorem licenses each division or inverse in the proposed formula?
2. Can a multiplicative function be computed independently on each prime power?
3. Which intermediate products can overflow before the final modular reduction?

4.3.6 Stretch directions

- Extend pairwise CRT merging to a list of congruences and report the first incompatible pair.
- Compute binomial coefficients modulo a prime when n is much larger than the modulus.
- Compare trial division, a smallest-prime-factor sieve, and segmented methods on different query distributions.

4.4 Detailed case study

4.4.1 A complete generalized CRT merge

Scenario. Merge $x \equiv 2 \pmod 8$ and $x \equiv 6 \pmod{12}$. The moduli are not coprime, so a direct coprime-only CRT formula is invalid.

4.4.2 Step-by-step trace

1. Compute $g = \gcd(8, 12) = 4$. Compatibility requires $6 - 2 = 4$ to be divisible by g, which it is.
2. Divide the reduced equation $8t \equiv 4 \pmod{12}$ by 4 to get $2t \equiv 1 \pmod 3$. The inverse of 2 modulo 3 is 2, so $t \equiv 2 \pmod 3$.
3. Substitute into $x = 2 + 8t$: choosing $t = 2$ gives $x = 18$. Check directly that $18 \bmod 8 = 2$ and $18 \bmod 12 = 6$.
4. The combined modulus is $\operatorname{lcm}(8, 12) = 24$, so every solution is $x \equiv 18 \pmod{24}$.
5. If the second residue were 4, the difference 2 would not be divisible by 4 and the system would have no solution.

4.4.3 Correctness argument

Writing $x = a + mt$ turns the merge into a linear congruence $mt \equiv b - a \pmod n$. Such a congruence is solvable exactly when $\gcd(m, n)$ divides the right side. After reduction, the coefficient is invertible, giving one residue class for t and therefore one class for x modulo the least common multiple.

4.4.4 Benchmark plan

Family	Scale or construction	Purpose
Compatible chain	Many shared factors	Repeated modulus reduction
Early conflict	Conflict at merge two	Failure reporting
Large moduli	Near 64-bit limits	Overflow before reduction

4.4.5 Coach-check questions

1. Why is the new modulus the least common multiple rather than the product?
2. How can multiplication be made safe when the least common multiple exceeds 64 bits?
3. What should a library routine return when the system is compatible but the normalized solution is negative?

4.5 Study framework

4.5.1 Prerequisites

- Integer division, remainders, induction, and prime factorization.
- Fast exponentiation and basic modular arithmetic.
- Counting principles, subsets, and binomial coefficients.

4.5.2 Learning outcomes

- Check invertibility and compatibility before manipulating congruences.
- Choose among sieving, factorization, CRT, and combinatorial DP from constraints.
- Turn inclusion–exclusion into an efficiently evaluable intersection formula.

4.6 Worked example

4.6.1 Merging two noncoprime congruences

Solve

$$x \equiv 2 \pmod 6, \qquad x \equiv 5 \pmod 9.$$

Write $x = 2 + 6t$. Substitution gives $6t \equiv 3 \pmod 9$. Since $g = \gcd(6, 9) = 3$ divides the right-hand side, a solution exists. Divide the entire congruence by g to obtain $2t \equiv 1 \pmod 3$. The inverse of 2 modulo 3 is 2, hence $t \equiv 2 \pmod 3$. Therefore

$$x \equiv 2 + 6 \cdot 2 \equiv 14 \pmod{18}.$$

The new modulus is $\operatorname{lcm}(6, 9) = 18$. Checking gives $14 \bmod 6 = 2$ and $14 \bmod 9 = 5$.

The compatibility test is essential. For $x \equiv 2 \pmod 6$ and $x \equiv 4 \pmod 9$, the residue difference is 2, not divisible by 3, so no solution exists. A CRT routine that assumes coprime moduli would silently produce nonsense.

4.7 Implementation template

4.7.1 Pseudocode

```
function extended_gcd(a, b):
    if b == 0: return (a, 1, 0)
    (g, x1, y1) = extended_gcd(b, a mod b)
    return (g, y1, x1 - floor(a/b) * y1)

function merge(a mod m, b mod n):
    (g, x, _) = extended_gcd(m, n)
    if (b-a) mod g != 0: return NO_SOLUTION
    step = ((b-a)/g * x) mod (n/g)
    lcm = m/g * n
    answer = (a + m*step) mod lcm
    return (answer, lcm)
```

4.7.2 Complexity reference

Method	Bound	Best fit
gcd / inverse	$O(\log m)$	One pair of integers.
Sieve	$O(N \log \log N)$	All primes or factors through N.
Divisor generation	$O(\tau(n))$	After factorization.
Subset inclusion–exclusion	$O(2^k)$	Small number of properties.

4.7.3 Common mistakes

- Using Fermat inversion without a prime modulus and a nonzero residue.
- Multiplying moduli before dividing by their gcd, causing avoidable overflow.
- Forgetting the empty subset term in inclusion–exclusion.
- Assuming a remaining trial-division value is composite after all primes through its square root were tested.

4.8 Practice lab

4.8.1 Short exercises

1. Merge $x \equiv 3 \pmod 8$ with $x \equiv 7 \pmod{12}$ or prove inconsistency.
2. Derive the divisor-count and divisor-sum formulas from prime exponents.
3. Explain why factorial inverses fail when the prime modulus divides a denominator.

4.8.2 Guided problem progression

1. **Coprime Integers**. Warm up with gcd structure and counting.
2. **Divisors**. Move between factor exponents and divisor functions.
3. **Chinese Remainder Theorem (non-relatively prime moduli)**. Practice compatibility and generalized merging.
4. **Discrete Logging**. Finish with a more advanced modular-search technique.

4.8.3 Submission workflow

1. Write the invariant, state meaning, or mathematical precondition before coding.
2. Build minimal examples for every boundary case identified in the chapter.
3. Compare against a brute-force solver on small random instances whenever practical.
4. Cross-check modular routines against brute force for small composite moduli, including compatible and incompatible congruences.
5. After acceptance, record the decisive idea, complexity, and one implementation hazard.

Complete lecture resources
Open the McGill materials folder in Google Drive. No lecture deck was present at the review date; use this folder to check for later additions. Access may require an authorized ICPC Foundation account.

4.9 Problem Set

13 problems • Instructor(s): Ari Blondal; David Becerra

Assignment: Open on Kattis

Planning key: mark each problem Foundation, Core, or Stretch after an opening scan; record a personal target time before starting. Kattis difficulty and availability can change, so this booklet does not freeze a platform rating.

1. Coprime Integers — kattis: coprimeintegers

tier: F / C / S • target: ____ min • technique: _____

2. Hopscotch — kattis: hopscotch

tier: F / C / S • target: ____ min • technique: _____

3. Divisors — kattis: divisors

tier: F / C / S • target: ____ min • technique: _____

4. Chinese Remainder Theorem (non-relatively prime moduli) — kattis: generalchineseremainder
   tier: F / C / S • target: ___ min • technique: _____
5. So You Like Your Food Hot? — kattis: soyoulikeyourfoodhot
   tier: F / C / S • target: ___ min • technique: _____
6. GCD Sum — kattis: gcdsum
   tier: F / C / S • target: ___ min • technique: _____
7. Discrete Logging — kattis: discretelogging
   tier: F / C / S • target: ___ min • technique: _____
8. Counting Trees — kattis: countingtrees
   tier: F / C / S • target: ___ min • technique: _____
9. Drowning Combinatorist — kattis: drowningcombinatorist
   tier: F / C / S • target: ___ min • technique: _____
10. Inverse Totient — kattis: inversetotient
    tier: F / C / S • target: ___ min • technique: _____
11. Towers of Powers 2: Power Harder — kattis: towers
    tier: F / C / S • target: ___ min • technique: _____
12. Fun with Fibonacci — kattis: funwithfibonacci
    tier: F / C / S • target: ___ min • technique: _____
13. Grid Guardian — kattis: gridguardian
    tier: F / C / S • target: ___ min • technique: _____

Tip. Open Metadata on a problem page to verify author, source, limits, downloads, and license before redistributing content.

4.10 Timed Contest

13 problems • Instructor(s): Ari Blondal; David Becerra

Assignment: Open on Kattis

Planning key: mark each problem Foundation, Core, or Stretch after an opening scan; record a personal target time before starting. Kattis difficulty and availability can change, so this booklet does not freeze a platform rating.

1. Jug Hard — kattis: jughard
   tier: F / C / S • target: ___ min • technique: _____
2. Grazed Grains — kattis: grazedgrains
   tier: F / C / S • target: ___ min • technique: _____
3. Another Coin Weighing Puzzle — kattis: anothercoinweighingpuzzle
   tier: F / C / S • target: ___ min • technique: _____
4. Number Trick — kattis: trick
   tier: F / C / S • target: ___ min • technique: _____
5. Number Magic — kattis: numbermagic
   tier: F / C / S • target: ___ min • technique: _____

6. Witchwood — kattis: witchwood

tier: F / C / S • target: ___ min • technique: _____

7. Necklace — kattis: necklace

tier: F / C / S • target: ___ min • technique: _____

8. Sky's the Limit — kattis: skysthelimit

tier: F / C / S • target: ___ min • technique: _____

9. Diagonal Cut — kattis: diagonalcut

tier: F / C / S • target: ___ min • technique: _____

10. Primal Partitions — kattis: primal

tier: F / C / S • target: ___ min • technique: _____

11. Random Digital Exponentiation — kattis: randomdigitalexponentiation

tier: F / C / S • target: ___ min • technique: _____

12. The Calculus of Ada — kattis: ada

tier: F / C / S • target: ___ min • technique: _____

13. Code Permutations — kattis: codepermutations

tier: F / C / S • target: ___ min • technique: _____

Tip. Open Metadata on a problem page to verify author, source, limits, downloads, and license before redistributing content.

4.11 Implementation reference

4.11.1 C++17 tested reference excerpt

A generalized Chinese remainder merge that accepts shared factors, rejects incompatible residues, and normalizes the result.

```
using i128 = __int128_t;
long long egcd(long long a,long long b,long long& x,long long& y){
    if(!b){ x=1; y=0; return a; }
    long long x1,y1,g=egcd(b,a%b,x1,y1);
    x=y1; y=x1-(a/b)*y1; return g;
}

optional<pair<long long,long long>> merge_crt(
    long long a,long long m,long long b,long long n){
    long long x,y,g=egcd(m,n,x,y);
    long long diff=b-a;
    if(diff%g) return nullopt;
    long long ng=n/g;
    i128 step=(i128)(diff/g)*x;
    step=(step%ng+ng)%ng;
    i128 lcm=(i128)(m/g)*n;
    i128 ans=((i128)a+(i128)m*step)%lcm;
    if(ans<0) ans+=lcm;
    return pair{(long long)ans,(long long)lcm};
}
```

4.11.2 Template contract

- Each input modulus is positive and each residue is normalized or safely representable.
- The return value is either no solution or (a, m) meaning $x \equiv a \pmod{m}$.
- The combined least common multiple must fit the advertised output type.

4.11.3 Porting and diagnostics

Language notes

- Java: use `BigInteger` for products and its gcd and modular-inverse operations.
- Python: arbitrary precision simplifies overflow, but normalize the result explicitly.
- C++: keep intermediate products in `__int128` and range-check before narrowing.

Diagnostic probes

1. Coprime, noncoprime-compatible, identical-modulus, and incompatible pairs.
2. Negative input residues and zero normalized residues.
3. A chain whose modulus grows near the output limit.
4. Random small systems checked by exhaustive search through one least common multiple.

Performance envelope

Measure	Bound	Interpretation
One merge	$O(\log \min(m, n))$	Extended Euclidean algorithm.
k **congruences**	$O(k \log M)$	Sequential merging.
Output modulus	lcm	May grow beyond native integers.

Submission audit

- Match every array dimension and numeric type to the largest legal instance.
- Run the diagnostic probes before optimization, then preserve them as regression tests.
- State the invariant and the complexity in the solution notes before submission.

4.12 Coach lesson plan

4.12.1 Ninety-minute session objective

Apply number-theory formulas only after checking their preconditions.

4.12.2 Agenda

Time	Activity	Evidence
0–10 min	Retrieval warm-up	Learners restate the chapter invariant without notes.
10–25 min	Model critique	Teams compare two candidate models and reject one with a counterexample.
25–45 min	Guided trace	The class completes the detailed case study and predicts each intermediate state.
45–70 min	Implementation lab	Pairs adapt the reference skeleton and run at least three diagnostic probes.
70–90 min	Debrief and transfer	Learners identify the constraint that selects the method on a new problem.

4.12.3 Misconceptions to surface

- An inverse is assumed to exist.
- CRT moduli are multiplied despite a shared gcd.
- Negative residues are not normalized.

4.12.4 Instructor checkpoints

1. Ask for a counterexample to an incomplete state, predicate, or graph model.
2. Require one learner to justify correctness while another audits numeric and asymptotic bounds.
3. Do not reveal the final transition until teams can name the invariant it must preserve.

4.12.5 Exit ticket

1. In at most 25 words, state the theorem and precondition used by the arithmetic reduction.
2. Name a composite-modulus case that breaks a prime-only formula.
3. Identify the bound that changes factorization, precomputation, or multiplication strategy.

Chapter 5

Columbia: Gaussian Elimination & FFT

CHRISTIAN YONGWHAN LIM; JOSH ALMAN; GRACE LIM

Each offering pairs a practice problem set with a timed contest. The two assignment pages below link directly to Kattis.

5.1 Lecture guide

Lecture-deck outline

Source basis: Columbia course-public Gaussian-elimination, generating-functions, FFT, and NTT decks plus the NAPC-O Kattis sequence.

Verification: Deck identities and assignment membership rechecked September 7, 2026.

The Columbia materials link linear algebra, generating functions, and fast polynomial multiplication into a contest-ready algebra toolkit.

5.1.1 Elimination and generating functions

- Implement Gauss–Jordan elimination and classify systems with no solution, a unique solution, or infinitely many solutions.
- Adapt elimination to modular arithmetic; over the field with two elements, pack rows into bitsets for wide XOR operations.
- Compute determinant and rank through elimination, and recognize LU-style factorization as a related computational viewpoint.
- Encode sequences with ordinary or exponential generating functions; use shifts, derivatives, products, powers, and prefix sums as algebraic transformations.
- Work through geometric-series, Fibonacci, Catalan, permutation-cycle, and set-partition examples.

5.1.2 FFT and NTT

- Interpret the discrete Fourier transform as evaluating a polynomial at roots of unity, with the inverse recovering its coefficients.
- Split even and odd coefficients recursively to obtain an O(n log n) transform, then multiply polynomials by transform–multiply–invert.

- Use an iterative implementation when predictable memory access and lower recursion overhead matter.
- For exact modular convolution, use a number theoretic transform with a suitable prime modulus, primitive root, and modular inverse normalization.

Materials reviewed: Gaussian Elimination (97 slides); Fast Fourier Transform (22 slides)

5.2 Technical notes

5.2.1 Gauss–Jordan elimination

Process columns from left to right while maintaining the next pivot row. Choose a nonzero pivot at or below that row, swap it into place, normalize it when division is valid, and eliminate the column from every other row. The number of pivots is the rank. A row with all-zero coefficients and a nonzero right-hand side proves inconsistency; otherwise, nonpivot variables are free.

Over floating point, select the largest-magnitude available pivot to reduce numerical error and compare against a scale-aware tolerance. Over a prime field, replace division by multiplication with a modular inverse. Over $\mathbb{F}_2$, normalization disappears and elimination is row XOR; packing coefficients into machine words or bitsets gives a large constant-factor improvement.

For a square matrix, elimination computes the determinant as the signed product of pivots before row normalization. Every row swap changes the sign. If a pivot cannot be found, the determinant is zero. The usual dense complexity is $O(n^3)$ time and $O(n^2)$ memory.

5.2.2 Generating functions as algebra on sequences

The ordinary generating function of (a_n) is $A(x) = \sum_{n\geq 0} a_n x^n$. Multiplication performs convolution:

$$[x^n]A(x)B(x) = \sum_{k=0}^{n} a_k b_{n-k}.$$

Multiplication by x^t shifts indices; differentiation multiplies coefficient a_n by n and shifts it down; division by $1 - x$ forms prefix sums. The geometric identity $(1 - x)^{-1} = 1 + x + x^2 + \cdots$ is therefore both an algebraic formula and an algorithmic pattern.

For labeled combinatorial objects, exponential generating functions $\sum a_n x^n/n!$ naturally account for choosing label sets. Distinguish OGF and EGF conventions before manipulating a formula: their products encode different combinatorial operations.

5.2.3 FFT convolution

Choose a transform length N that is a power of two and at least the required result length. Pad both coefficient arrays, compute their DFTs, multiply pointwise, and apply the inverse DFT. The recurrence splits even and odd coefficients:

$$A(\omega_N^k) = A_{even}(\omega_{N/2}^k) + \omega_N^k A_{odd}(\omega_{N/2}^k),$$

with the second half obtained by subtraction. This yields $O(N \log N)$ time.

Complex FFT uses rounding, so integer convolution should round only after the inverse transform and may need coefficient splitting when values are large. A number theoretic transform instead works modulo a prime $p = c2^k + 1$ with a primitive 2^kth root. The inverse uses the inverse root and multiplies every coefficient by $N^{-1} \bmod p$.

5.2.4 Algebra implementation checklist

- Record the field or ring: which nonzero elements are invertible?
- In elimination, track pivot columns separately from row indices.
- In FFT/NTT, pad to the result length $n + m - 1$, not merely the larger input length.
- Normalize the inverse transform exactly once and trim padded coefficients.
- Bound coefficient growth before selecting floating point, one NTT modulus, or several moduli plus CRT.

5.3 Advanced workshop

5.3.1 Viewing algebraic algorithms as representation changes

Elimination and Fourier transforms are both changes of representation. Row operations preserve the solution set while exposing pivots, rank, and inconsistency. A Fourier transform preserves a polynomial while replacing coefficients with evaluations at carefully selected points. In the evaluation representation, convolution becomes pointwise multiplication; in row-echelon form, linear constraints become directly interpretable. The representation is useful because it makes the required operation cheaper or more visible. The coefficient domain determines every low-level rule. Over floating point, pivot selection and tolerances control stability. Over a prime field, division uses modular inverses. Over the field with two elements, rows become bitsets and elimination becomes XOR. For convolution, complex FFT needs rounding analysis, while NTT provides exact modular arithmetic only when the modulus contains a sufficiently large power-of-two root of unity.

5.3.2 Correctness-proof blueprint

1. For elimination, show each row operation preserves exactly the same solution set and interpret pivot and inconsistent rows.
2. For FFT, derive the even-odd split from evaluating at roots whose squares form the smaller transform domain.
3. For convolution, prove that the transform length is at least the full coefficient-result length before cyclic wraparound occurs.

5.3.3 Implementation notes across languages

- Keep pivot columns and row swaps explicitly; determinant sign and solution reconstruction depend on them.
- Under floating point, choose the largest available pivot magnitude in the column rather than the first nonzero-looking value.
- For iterative FFT or NTT, test bit-reversal ordering separately from butterfly arithmetic.
- Normalize the inverse transform exactly once and round only after returning to the coefficient representation.

5.3.4 Adversarial testing matrix

Case	Construction	What it exposes
Dependent rows	Duplicate equation	Rank and free variables
Inconsistent row	$0 = 1$	No-solution detection
Impulse polynomial	One nonzero coefficient	Transform indexing
Maximum degree	Result just above a power of two	Padding and wraparound

5.3.5 Discussion prompts

1. Which representation makes the expensive operation local or pointwise?
2. What properties must the coefficient domain provide for each division?
3. How will the implementation distinguish numerical noise from genuine rank?

5.3.6 Stretch directions

- Implement elimination over both doubles and a prime field behind the same conceptual interface.
- Use bitset elimination to solve a parity system and compare it with scalar XOR rows.
- Validate NTT convolution against quadratic multiplication on random small polynomials.

5.4 Detailed case study

5.4.1 Elimination that reports structure as well as values

Scenario. Solve and classify the system $x + y + z = 4$, $2x - y + z = 1$, and $3x + 2z = 5$. Track pivot columns so the routine can distinguish a unique solution from free variables.

5.4.2 Step-by-step trace

1. Use the first row as the pivot in column x. Replace row 2 by $R_2 - 2R_1$ and row 3 by $R_3 - 3R_1$, obtaining rows $[0, -3, -1 \mid -7]$ and $[0, -3, -1 \mid -7]$.
2. Choose row 2 as the pivot in column y. Subtract it from row 3, producing the all-zero row $[0, 0, 0 \mid 0]$.
3. There are two pivots for three variables and no inconsistent row, so the system has infinitely many solutions with one free variable.
4. Let $z = t$. The second row gives $-3y - t = -7$, hence $y = (7 - t)/3$. The first row gives $x = 4 - y - t = (5 - 2t)/3$.
5. Changing the final right-hand side from 5 to 6 would produce $[0, 0, 0 \mid 1]$ after elimination, proving inconsistency.

5.4.3 Correctness argument

Swapping rows, multiplying a row by a nonzero scalar, and adding a multiple of one row to another are reversible operations, so they preserve the solution set. Pivot columns identify dependent variables. A zero coefficient row with nonzero right-hand side is impossible; otherwise each nonpivot variable may be chosen freely.

5.4.4 Benchmark plan

Family	Scale or construction	Purpose
Rank deficient	Construct duplicate rows	Free-variable handling
Nearly singular	Small floating pivots	Pivot strategy and tolerance
Finite field	Random prime-modulus systems	Modular inverses

5.4.5 Coach-check questions

1. How should the routine return a basis for the null space?
2. Why is partial pivoting relevant over doubles but not over an exact prime field?
3. How can the same pivot information be used to compute a determinant?

5.5 Study framework

5.5.1 Prerequisites

- Matrix arithmetic, modular inverses, and polynomial coefficients.
- Complex numbers or finite fields and divide-and-conquer recurrences.
- Basic counting recurrences and convolution.

5.5.2 Learning outcomes

- Use elimination to find rank, solutions, and determinants over a chosen field.
- Translate sequence operations into generating-function algebra.
- Select FFT, NTT, or direct multiplication from size and exactness requirements.

5.6 Worked example

5.6.1 Elimination with classification, not just arithmetic

Solve

$$x + y = 3, \qquad 2x - y = 0.$$

The augmented matrix is

$$\left[\begin{array}{cc|c} 1 & 1 & 3 \\ 2 & -1 & 0 \end{array}\right].$$

Replace the second row by $R_2 - 2R_1$ to obtain $[0, -3 \mid -6]$, so $y = 2$ and then $x = 1$. There is a pivot in each variable column, hence the solution is unique.

If the second equation were $2x + 2y = 7$, elimination would produce $[0, 0 \mid 1]$, proving inconsistency. If it were $2x + 2y = 6$, the second row would vanish, leaving one pivot and one free variable. The arithmetic routine must therefore return structural information: pivot columns, rank, and whether an inconsistent row exists.

Over a prime modulus, divide a pivot row by multiplying by the modular inverse. Over $\mathbb{F}_2$, the only nonzero pivot is one and elimination is XOR. Over floating point, select a large pivot and use a tolerance tied to scale.

5.7 Implementation template

5.7.1 Pseudocode

```
row = 0
pivot_columns = []
for col = 0 .. variables-1:
    pivot = choose_nonzero_row(row..n-1, col)
    if none: continue
    swap(matrix[row], matrix[pivot])
    normalize matrix[row] by its pivot
    for r = 0 .. n-1:
        if r != row:
            matrix[r] -= matrix[r][col] * matrix[row]
    pivot_columns.push(col)
    row++

if an all-zero coefficient row has nonzero RHS: no solution
else if |pivot_columns| == variables: unique solution
else: infinitely many solutions
```

5.7.2 Complexity reference

Method	Bound	Best fit
Dense elimination	$O(n^3)$	Rank, systems, determinant.
Bitset elimination over $\mathbb{F}_2$	$O(n^3/w)$	Pack w columns per word.
Direct convolution	$O(nm)$	Small coefficient arrays.
FFT / NTT	$O(N \log N)$	Large polynomial products.

5.7.3 Common mistakes

- Returning only transformed rows without recording pivot columns.
- Dividing modulo a composite modulus when the pivot is not invertible.
- Padding convolution inputs to the larger input length rather than the result length.
- Forgetting inverse-transform normalization or rounding too early.

5.8 Practice lab

5.8.1 Short exercises

1. Classify a system with rank two and four variables, assuming it is consistent.
2. Show how multiplying an OGF by $1/(1-x)$ forms prefix sums.
3. Choose an FFT padding length for polynomials of lengths 700 and 900.

5.8.2 Guided problem progression

1. **Generating Numbers**. Translate a recurrence or counting rule into algebra.
2. **Fun with Fibonacci**. Connect linear recurrences with algebraic representations.
3. **The Big Painting**. Look for transform-based matching or convolution structure.
4. **Easy Query**. Practice selecting the right algebraic preprocessing.

5.8.3 Submission workflow

1. Write the invariant, state meaning, or mathematical precondition before coding.
2. Build minimal examples for every boundary case identified in the chapter.
3. Compare against a brute-force solver on small random instances whenever practical.
4. Check rank classification on unique, inconsistent, and underdetermined systems, then compare small convolutions with quadratic multiplication.
5. After acceptance, record the decisive idea, complexity, and one implementation hazard.

Complete lecture resources
Open the Columbia materials folder in Google Drive. The folder is the authoritative location for complete decks, notes, and future revisions. Access may require an authorized ICPC Foundation account.

5.9 Problem Set

10 problems • Instructor(s): Christian Yongwhan Lim; Josh Alman; Grace Lim

Assignment: Open on Kattis

Planning key: mark each problem Foundation, Core, or Stretch after an opening scan; record a personal target time before starting. Kattis difficulty and availability can change, so this booklet does not freeze a platform rating.

1. A Totient Quotient — kattis: atotientquotient

tier: F / C / S • target: ____ min • technique: _____

2. Most Scenic Cycle — kattis: mostsceniccycle

tier: F / C / S • target: ____ min • technique: _____

3. Solar Farm — kattis: solarfarm

tier: F / C / S • target: ____ min • technique: _____

4. Polygon Partition — kattis: polygonpartition
   tier: F / C / S • target: ____ min • technique: _____
5. Humans vs AI — kattis: humansvsai
   tier: F / C / S • target: ____ min • technique: _____
6. Shadow Line — kattis: shadowline
   tier: F / C / S • target: ____ min • technique: _____
7. Generating Numbers — kattis: generatingnumbers
   tier: F / C / S • target: ____ min • technique: _____
8. Fun with Fibonacci — kattis: funwithfibonacci
   tier: F / C / S • target: ____ min • technique: _____
9. Easy Query — kattis: easyquery
   tier: F / C / S • target: ____ min • technique: _____
10. The Big Painting — kattis: bigpainting
    tier: F / C / S • target: ____ min • technique: _____

Tip. Open Metadata on a problem page to verify author, source, limits, downloads, and license before redistributing content.

5.10 Timed Contest

10 problems • Instructor(s): Christian Yongwhan Lim; Josh Alman; Grace Lim

Assignment: Open on Kattis

Planning key: mark each problem Foundation, Core, or Stretch after an opening scan; record a personal target time before starting. Kattis difficulty and availability can change, so this booklet does not freeze a platform rating.

1. Geometry Rush — kattis: geometryrush
   tier: F / C / S • target: ____ min • technique: _____
2. Cookie Cutter — kattis: cookiecutter2
   tier: F / C / S • target: ____ min • technique: _____
3. Triangular Logs — kattis: triangularlogs
   tier: F / C / S • target: ____ min • technique: _____
4. Space Alignment — kattis: spacealignment
   tier: F / C / S • target: ____ min • technique: _____
5. Repetitive String Invention — kattis: repetitivestringinvention
   tier: F / C / S • target: ____ min • technique: _____
6. Power of Divisors — kattis: powerofdivisors
   tier: F / C / S • target: ____ min • technique: _____
7. Circle of Friends — kattis: circleoffriends
   tier: F / C / S • target: ____ min • technique: _____
8. Train Line — kattis: trainline
   tier: F / C / S • target: ____ min • technique: _____

9. Who Watches the Watchmen? kattis: whowatchesthewatchmen

tier: F / C / S • target: ___ min • technique: _____

10. Special Cycle kattis: specialcycle

tier: F / C / S • target: ___ min • technique: _____

Tip. Open Metadata on a problem page to verify author, source, limits, downloads, and license before redistributing content.

5.11 Implementation reference

5.11.1 C++17 tested reference excerpt

Gauss-Jordan elimination over doubles with partial pivoting. The routine returns rank and leaves enough structure to classify and reconstruct solutions.

```
const double EPS=1e-10;
int gauss(vector<vector<double>>& a,int vars){
    int n=a.size(), row=0;
    pivot_col.clear();
    for(int col=0; col<vars && row<n; ++col){
        int p=row;
        for(int i=row;i<n;++i)
            if(abs(a[i][col])>abs(a[p][col])) p=i;
        if(abs(a[p][col])<EPS) continue;
        swap(a[p],a[row]);
        double z=a[row][col];
        for(int j=col;j<=vars;++j) a[row][j]/=z;
        for(int i=0;i<n;++i) if(i!=row){
            double f=a[i][col];
            for(int j=col;j<=vars;++j) a[i][j]-=f*a[row][j];
        }
        pivot_col.push_back(col); ++row;
    }
    return row;
}
```

5.11.2 Template contract

- Column `vars` is the right-hand side; earlier columns are variables.
- Pivot columns are recorded in increasing order and rank equals their count.
- Inconsistency is checked afterward on every zero-coefficient row.

5.11.3 Porting and diagnostics

Language notes

- Java: store rows as primitive `double[]` arrays and use `Math.abs`.
- Python: pure nested loops are slow for large dense systems; constraints may require optimized arrays.
- For prime fields, replace tolerance and division with nonzero tests and modular inverses.

Diagnostic probes

1. Unique, underdetermined, overdetermined-consistent, and inconsistent systems.
2. Rows requiring a swap because the current pivot is zero.
3. Nearly dependent floating rows at several numeric scales.
4. Known-rank random matrices constructed from low-rank factors.

Performance envelope

Measure	Bound	Interpretation
Square system	$O(n^3)$	Dense elimination.
Memory	$O(n^2)$	Augmented matrix.
Bitset $\mathbb{F}_2$	$O(n^3/w)$	Pack w columns per word.

Submission audit

- Match every array dimension and numeric type to the largest legal instance.
- Run the diagnostic probes before optimization, then preserve them as regression tests.
- State the invariant and the complexity in the solution notes before submission.

5.12 Coach lesson plan

5.12.1 Ninety-minute session objective

Interpret pivot structure and choose the correct coefficient domain.

5.12.2 Agenda

Time	Activity	Evidence
0–10 min	Retrieval warm-up	Learners restate the chapter invariant without notes.
10–25 min	Model critique	Teams compare two candidate models and reject one with a counterexample.
25–45 min	Guided trace	The class completes the detailed case study and predicts each intermediate state.
45–70 min	Implementation lab	Pairs adapt the reference skeleton and run at least three diagnostic probes.
70–90 min	Debrief and transfer	Learners identify the constraint that selects the method on a new problem.

5.12.3 Misconceptions to surface

- Rank is returned without pivot columns.
- Floating zero is tested exactly.
- FFT padding uses input length instead of result length.

5.12.4 Instructor checkpoints

1. Ask for a counterexample to an incomplete state, predicate, or graph model.
2. Require one learner to justify correctness while another audits numeric and asymptotic bounds.
3. Do not reveal the final transition until teams can name the invariant it must preserve.

5.12.5 Exit ticket

1. In at most 25 words, name the representation change that makes the target operation easier.
2. Give one singular system and one transform-size case that expose a nearly correct implementation.
3. Identify when numerical error or quadratic multiplication requires another field or transform.

Chapter 6

UIUC: Strings

MATTOX BECKMAN

Each offering pairs a practice problem set with a timed contest. The two assignment pages below link directly to Kattis.

6.1 Lecture guide

Published-course roadmap

Source basis: Official weekly topic, trainer listing, and visible NAPC-O Kattis assignment sequence; no lecture deck was available.

Verification: Public course metadata rechecked September 7, 2026; this roadmap is editorial synthesis, not a lecture transcript.

This roadmap is inferred from the published Strings topic and its Kattis assignment titles, not from an unpublished lecture deck.

6.1.1 Strings practice progression

- Begin with direct matching, periodicity, and careful parsing before moving to batched pattern matching.
- Use rolling hashes for substring comparison while accounting for collisions and index normalization.
- Study suffix ordering and repeated-substring structure for problems that require global information about every suffix.
- Connect transform-based string representations, including the Burrows–Wheeler transform, to reversible ordering and matching ideas.

Materials reviewed: No lecture deck was present in the shared materials folder at review time.

6.2 Technical notes

6.2.1 Linear-time pattern matching

For a pattern p, the prefix function $\pi[i]$ is the length of the longest proper prefix of $p[0..i]$ that is also a suffix. When a comparison fails after j matched characters, replace j with $\pi[j-1]$ rather than restarting. Each fallback strictly decreases j, so construction and matching are both linear.

The Z-function gives the length of the longest prefix match beginning at each position. Concatenate p, a separator not in the alphabet, and the text; every Z-value at least $|p|$ marks an occurrence. KMP and Z solve similar tasks, but one representation may align more naturally with a border, period, or prefix-comparison argument.

Key idea. A period q of a string of length n is often visible through a border of length $n - q$. Check divisibility when the problem asks for a complete repetition rather than merely a suffix-prefix overlap.

6.2.2 Rolling hashes

Represent a string as a polynomial and precompute prefix hashes:

$$H[i+1] = (H[i]B + s_i) \bmod M.$$

Then the hash of $s[l..r)$ is $H[r] - H[l]B^{r-l}$ modulo M. Normalize negative residues. Hashing gives fast substring comparisons and supports binary-searching a longest common prefix, but collisions are possible. Use two independent moduli or a 64-bit strategy when a probabilistic answer is acceptable; use suffix structures when the solution must be deterministic.

6.2.3 Suffix order and repeated substrings

A suffix array stores all suffix starting positions in lexicographic order. The doubling algorithm sorts by pairs of equivalence classes for substrings of length 2^k, giving $O(n \log^2 n)$ with comparison sorting or $O(n \log n)$ with counting/radix sorting of class pairs. Kasai's algorithm computes the adjacent longest-common-prefix array in $O(n)$ by reusing all but one character of the previous comparison.

Repeated-substring questions frequently reduce to the LCP array: the maximum LCP finds the longest repeated substring, while a window minimum over LCP values measures the common prefix shared by a block of suffixes. Multi-string problems can concatenate inputs with distinct separators and track which source owns each suffix.

6.2.4 Burrows–Wheeler viewpoint

The Burrows–Wheeler transform orders cyclic rotations and records the character preceding each sorted rotation. Its last-to-first mapping follows from stable occurrence ranks: the kth occurrence of a character in the last column maps to the kth occurrence in the first column. This makes the transform reversible and underlies backward search in FM-indexes.

6.2.5 Selection checklist

One pattern KMP or Z.

Many patterns Trie or Aho–Corasick.

Substring equality Rolling hash or suffix-array/LCP preprocessing.

All suffixes Suffix array; suffix automaton or suffix tree when their state transitions fit the task.

Periods/borders Prefix function, Z-function, or gcd reasoning on candidate periods.

6.3 Advanced workshop

6.3.1 Choosing a string representation for the query workload

String algorithms trade preprocessing for the ability to answer repeated structural questions. Prefix-function and Z-algorithm preprocessing captures borders and matches against one reference pattern. Rolling hashes

support fast substring comparisons but introduce a collision model. A suffix array orders every suffix, turning repeated-substring and lexicographic questions into range questions over longest-common-prefix information. The correct representation depends on whether the workload is one pattern, many patterns, arbitrary substring equality, or global suffix structure. Index discipline matters as much as asymptotic complexity. Decide whether intervals are closed or half-open, whether a match position refers to the combined pattern-text string or the original text, and how sentinel characters compare with the input alphabet. For hashes, normalize exponents or use prefix formulas that compare substrings in a common orientation. For suffix structures, separate construction from the query layer and verify the rank-to-position mapping.

6.3.2 Correctness-proof blueprint

1. For a failure function, maintain the invariant that the stored length is the longest border of the processed prefix.
2. For rolling hashes, derive substring extraction algebraically from the prefix definition and state the collision assumption.
3. For suffix ordering, prove that each doubling round sorts by the first 2^k characters using ranks from the previous round.

6.3.3 Implementation notes across languages

- Use a sentinel outside the input alphabet when concatenating pattern and text, and assert that it never appears in either.
- With hashes, use two independent moduli or a wide randomized hash when adversarial collision risk matters.
- Compress rank pairs to consecutive integers in suffix-array doubling so counting sort remains linear per round.
- Keep byte versus Unicode assumptions explicit; contest inputs are often ASCII, but language string indexing may not be.

6.3.4 Adversarial testing matrix

Case	Construction	What it exposes
All equal	`aaaaaa`	Borders and repeated matches
No repetition	Distinct characters	Base behavior
Pattern longer	Short text	Bounds and empty result
Overlapping match	`ababa` / `aba`	Fallback correctness

6.3.5 Discussion prompts

1. Is the problem asking about one pattern, arbitrary substrings, or every suffix?
2. Can a deterministic structure meet the constraints without accepting collision risk?
3. Which index space does every returned position belong to?

6.3.6 Stretch directions

- Build longest-common-extension queries from a suffix array, rank array, LCP array, and range-minimum structure.

- Compare prefix-function, Z-algorithm, and rolling-hash solutions to the same matching workload.
- Reconstruct a string from its Burrows-Wheeler transform using stable occurrence ranks.

6.4 Detailed case study

6.4.1 Tracing KMP through overlapping matches

Scenario. Find every occurrence of pattern `aba` in text `ababa`. The occurrences overlap, so resetting the matched length to zero after a match would miss the second answer.

6.4.2 Step-by-step trace

1. The prefix function for `aba` is $[0, 0, 1]$: the full pattern has border `a` of length 1.
2. Read text positions 0, 1, and 2. The matched length grows from 0 to 3, producing a match starting at $2 - 3 + 1 = 0$.
3. After reporting, fall back to the border length $\pi[2] = 1$ rather than zero. The retained `a` may begin an overlapping occurrence.
4. Reading positions 3 and 4 extends the retained match through `b` and `a`, producing the second start at $4 - 3 + 1 = 2$.
5. The final answer is $[0, 2]$. Every character causes only a bounded number of total fallback steps, so the scan is linear.

6.4.3 Correctness argument

Before processing each text character, the matched length is the longest pattern prefix equal to a suffix of the processed text. A mismatch follows border links until extension is possible or the length becomes zero. The invariant therefore remains true, and reaching the full pattern length reports exactly one occurrence ending at the current position.

6.4.4 Benchmark plan

Family	Scale or construction	Purpose
All equal	10^6 copies of `a`	Fallback and overlap
Alternating	`ababab...`	Long borders
Random differential	Short strings	Compare with naive matching

6.4.5 Coach-check questions

1. Why does the total number of fallback steps remain linear?
2. How can the same prefix table detect the shortest period of a string?
3. When would Aho–Corasick be preferable to repeating KMP for many patterns?

6.5 Study framework

6.5.1 Prerequisites

- Arrays, zero-based indexing, and amortized-analysis intuition.
- Polynomial evaluation modulo an integer and basic collision awareness.
- Lexicographic order and stable sorting.

6.5.2 Learning outcomes

- Move between borders, periods, prefix-function values, and Z-values.
- Select hashing or suffix structures according to determinism and query volume.
- Use suffix order and LCP information for repeated-substring problems.

6.6 Worked example

6.6.1 Tracing KMP instead of restarting

Take the pattern $p =$ `ababaca`. The prefix-function values are

$$[0, 0, 1, 2, 3, 0, 1].$$

At position four, the prefix `aba` is also a suffix, so $\pi[4] = 3$. At position five the next character is `c`; the candidate border of length three expects `b`, so fall back to $\pi[2] = 1$. That candidate also fails, so fall back to zero. No text character is revisited.

During matching, let j be the number of matched pattern characters. On a mismatch, set $j = \pi[j - 1]$ until the next pattern character matches or $j = 0$. On a full match, report its ending position and continue from $\pi[|p| - 1]$ to allow overlaps. The linear bound follows because j increases at most once per processed character and every fallback decreases it.

The same border data exposes periods. If a string of length n has final border length b, then $q = n - b$ is a candidate period. It represents a complete repetition only when $n \bmod q = 0$. For partial periodicity, divisibility may not be required; the statement determines the convention.

6.7 Implementation template

6.7.1 Pseudocode

```
function prefix_function(p):
    pi = array(|p|, 0)
    for i = 1 .. |p|-1:
        j = pi[i-1]
        while j > 0 and p[i] != p[j]: j = pi[j-1]
        if p[i] == p[j]: j++
        pi[i] = j
    return pi

function find_all(text, p):
    pi = prefix_function(p); j = 0
    for i = 0 .. |text|-1:
        while j > 0 and text[i] != p[j]: j = pi[j-1]
        if text[i] == p[j]: j++
        if j == |p|:
            report i-|p|+1
            j = pi[j-1]
```

6.7.2 Complexity reference

Method	Bound	Best fit
KMP / Z	$O(n+m)$	One pattern or border structure.
Aho–Corasick	$O(L+z)$	Many patterns; total length L, matches z.
Suffix array + LCP	$O(n \log n)$	Global suffix order and repeated substrings.
Rolling hash query	$O(1)$ after $O(n)$	Fast probabilistic substring equality.

6.7.3 Common mistakes

- Forgetting to continue from a border after reporting an overlapping match.
- Using a separator that may appear in the pattern or text.
- Comparing raw substring hashes without length normalization.
- Assuming one modular hash is deterministic proof of equality.

6.8 Practice lab

6.8.1 Short exercises

1. Compute the prefix function and all borders of `abcababcab`.
2. Explain why the KMP fallback loop is linear over the whole text, not per character.
3. Show two suffix-array neighbors whose LCP identifies the longest repeat in `banana`.

6.8.2 Guided problem progression

1. **String Matching**. Implement and validate a linear matcher.
2. **Power Strings**. Translate borders into complete periods.
3. **Suffix Sorting**. Build and query global suffix order.
4. **Repeated Substrings**. Use adjacent suffixes and LCP values.
5. **Burrows-Wheeler**. Connect sorted rotations with stable occurrence ranks.

6.8.3 Submission workflow

1. Write the invariant, state meaning, or mathematical precondition before coding.
2. Build minimal examples for every boundary case identified in the chapter.
3. Compare against a brute-force solver on small random instances whenever practical.
4. Compare matches, borders, and repeats with naive routines on empty, periodic, overlapping, and highly repetitive strings.
5. After acceptance, record the decisive idea, complexity, and one implementation hazard.

Complete lecture resources
Open the UIUC materials folder in Google Drive. No lecture deck was present at the review date; use this folder to check for later additions. Access may require an authorized ICPC Foundation account.

6.9 Strings Problem Set

12 problems • Instructor(s): Mattox Beckman

Assignment: Open on Kattis

Planning key: mark each problem Foundation, Core, or Stretch after an opening scan; record a personal target time before starting. Kattis difficulty and availability can change, so this booklet does not freeze a platform rating.

1. String Matching — kattis: stringmatching

tier: F / C / S • target: ____ min • technique: _____

2. Power Strings — kattis: powerstrings

tier: F / C / S • target: ____ min • technique: _____

3. String Hashing — kattis: hashing
tier: F / C / S • target: ___ min • technique: _____

4. Evil Straw Warts Live — kattis: evilstraw
tier: F / C / S • target: ___ min • technique: _____

5. String Factoring — kattis: stringfactoring
tier: F / C / S • target: ___ min • technique: _____

6. Suffix Sorting — kattis: suffixsorting
tier: F / C / S • target: ___ min • technique: _____

7. String Multimatching — kattis: stringmultimatching
tier: F / C / S • target: ___ min • technique: _____

8. Buzzwords — kattis: buzzwords
tier: F / C / S • target: ___ min • technique: _____

9. Repeated Substrings — kattis: substrings
tier: F / C / S • target: ___ min • technique: _____

10. Burrows-Wheeler — kattis: burrowswheeler
tier: F / C / S • target: ___ min • technique: _____

11. Join Strings — kattis: joinstrings
tier: F / C / S • target: ___ min • technique: _____

12. Animal Classification — kattis: animal
tier: F / C / S • target: ___ min • technique: _____

Tip. Open Metadata on a problem page to verify author, source, limits, downloads, and license before redistributing content.

6.10 Strings Timed Contest

11 problems • Instructor(s): Mattox Beckman

Assignment: Open on Kattis

Planning key: mark each problem Foundation, Core, or Stretch after an opening scan; record a personal target time before starting. Kattis difficulty and availability can change, so this booklet does not freeze a platform rating.

1. Suffix Array Re-construction — kattis: suffixarrayreconstruction
tier: F / C / S • target: ___ min • technique: _____

2. Dvaput — kattis: dvaput
tier: F / C / S • target: ___ min • technique: _____

3. Chasing Subs — kattis: chasingsubs
tier: F / C / S • target: ___ min • technique: _____

4. Magical String — kattis: magicalstring
tier: F / C / S • target: ___ min • technique: _____

5. Spiderman's Workout — kattis: spiderman
tier: F / C / S • target: ___ min • technique: _____

6. Word Clouds Revisited — kattis: wordclouds

tier: F / C / S • target: ___ min • technique: _____

7. Phone List — kattis: phonelist

tier: F / C / S • target: ___ min • technique: _____

8. Radio Transmission — kattis: radiotransmission

tier: F / C / S • target: ___ min • technique: _____

9. Clock Pictures — kattis: clockpictures

tier: F / C / S • target: ___ min • technique: _____

10. Palindrome Substring — kattis: palindromesubstring

tier: F / C / S • target: ___ min • technique: _____

11. Making Palindromes — kattis: makingpalindromes

tier: F / C / S • target: ___ min • technique: _____

Tip. Open Metadata on a problem page to verify author, source, limits, downloads, and license before redistributing content.

6.11 Implementation reference

6.11.1 C++17 tested reference excerpt

A prefix-function and KMP matcher that reports overlapping occurrences and never revisits a text character.

```
vector<int> prefix_function(const string& s){
    vector<int> pi(s.size());
    for(int i=1;i<(int)s.size();++i){
        int j=pi[i-1];
        while(j>0 && s[i]!=s[j]) j=pi[j-1];
        if(s[i]==s[j]) ++j;
        pi[i]=j;
    }
    return pi;
}
vector<int> match(const string& p,const string& t){
    if(p.empty()) return {};
    vector<int> pi=prefix_function(p), ans;
    int j=0;
    for(int i=0;i<(int)t.size();++i){
        while(j>0 && t[i]!=p[j]) j=pi[j-1];
        if(t[i]==p[j]) ++j;
        if(j==(int)p.size()){
            ans.push_back(i-j+1);
            j=pi[j-1];
        }
    }
    return ans;
}
```

6.11.2 Template contract

- The pattern is nonempty; define empty-pattern behavior separately if required.
- Returned positions are zero-based indices in the original text.
- Falling back after a match preserves overlapping occurrences.

6.11.3 Porting and diagnostics

Language notes

- Java: use `charAt` under an ASCII-input assumption and an `int[]` prefix array.
- Python: direct loops are linear but may need buffered input and local-variable bindings for speed.
- C++: signed indices avoid unsigned underflow in expressions such as $i - j + 1$.

Diagnostic probes

1. Pattern longer than text, exact equality, no match, and match at each boundary.
2. All-equal strings and strongly overlapping periodic strings.
3. Random short strings compared with direct substring checks.
4. Million-character worst cases to verify linear behavior.

Performance envelope

Measure	Bound	Interpretation
Preprocess	$O(\|p\|)$	Prefix function.
Scan	$O(\|t\|)$	Amortized fallback.
Memory	$O(\|p\| + z)$	z reported matches.

Submission audit

- Match every array dimension and numeric type to the largest legal instance.
- Run the diagnostic probes before optimization, then preserve them as regression tests.
- State the invariant and the complexity in the solution notes before submission.

6.12 Coach lesson plan

6.12.1 Ninety-minute session objective

Select a string representation for the query workload and keep indices consistent.

6.12.2 Agenda

Time	Activity	Evidence
0–10 min	Retrieval warm-up	Learners restate the chapter invariant without notes.
10–25 min	Model critique	Teams compare two candidate models and reject one with a counterexample.
25–45 min	Guided trace	The class completes the detailed case study and predicts each intermediate state.
45–70 min	Implementation lab	Pairs adapt the reference skeleton and run at least three diagnostic probes.
70–90 min	Debrief and transfer	Learners identify the constraint that selects the method on a new problem.

6.12.3 Misconceptions to surface

- A match resets to zero and loses overlaps.
- Combined-string positions are reported as text positions.
- Hash collisions are treated as impossible.

6.12.4 Instructor checkpoints

1. Ask for a counterexample to an incomplete state, predicate, or graph model.
2. Require one learner to justify correctness while another audits numeric and asymptotic bounds.
3. Do not reveal the final transition until teams can name the invariant it must preserve.

6.12.5 Exit ticket

1. In at most 25 words, state which string representation answers the required query workload.
2. Name a periodic or overlapping string that breaks a restart-based or off-by-one implementation.
3. Identify when collision risk, alphabet size, or total length changes the method.

Chapter 7

MIT: Graphs

JAEHYUN KOO

Each offering pairs a practice problem set with a timed contest. The two assignment pages below link directly to Kattis.

7.1 Lecture guide

Published-course roadmap

Source basis: Official weekly topic, trainer listing, and visible NAPC-O Kattis assignment sequence; no lecture deck was available.

Verification: Public course metadata rechecked September 7, 2026; this roadmap is editorial synthesis, not a lecture transcript.

This roadmap is organized from the published Graphs topic and the accompanying Kattis practice sequence.

7.1.1 Graph modeling and algorithm selection

- Translate the story into vertices, edges, direction, weights, capacities, or states before choosing an algorithm.
- Separate reachability and component questions from shortest-path, spanning-tree, flow, matching, and ordering questions.
- Exploit structural promises such as acyclicity, bipartiteness, tree shape, or small state dimensions.
- Validate complexity against both vertex and edge bounds, and test disconnected graphs, parallel edges, self-loops, and unreachable targets.

Materials reviewed: No lecture deck was present in the shared materials folder or relevant attachment search at review time.

7.2 Technical notes

7.2.1 Model first

Vertices should represent the smallest states between which one legal action moves. Edges encode those actions; direction expresses irreversibility, weights express additive cost, and capacities express shared throughput.

Sometimes the correct graph is implicit: generate neighbors during the search rather than storing every edge. A state-expanded graph can add a parity bit, resource amount, last color, or automaton state when future legality depends on history.

7.2.2 Shortest-path decision table

Unweighted	BFS, $O(V + E)$.
Weights 0 or 1	0–1 BFS with a deque, $O(V + E)$.
Nonnegative	Dijkstra with a binary heap, $O((V + E) \log V)$.
Negative edges	Bellman–Ford, or a problem-specific reweighting; also detect reachable negative cycles.
DAG	Topological relaxation, $O(V + E)$, even with negative weights.
All pairs	Floyd–Warshall in $O(V^3)$ for small dense graphs; repeated single-source methods otherwise.

In Dijkstra's algorithm, ignore a heap entry whose distance no longer equals the current best value. Marking a vertex permanently visited when it is inserted, rather than when its minimum key is removed, is a common bug.

7.2.3 Components and ordering

DFS or BFS finds undirected connected components. Directed reachability is different: strongly connected components partition a directed graph into maximal mutually reachable sets. Contracting each SCC produces a DAG, after which topological DP becomes available. Kahn's algorithm topologically sorts by repeatedly removing indegree-zero vertices; processing fewer than V vertices proves a directed cycle exists.

For minimum spanning trees, Kruskal sorts edges and uses DSU, while Prim grows a tree from a frontier. Both rely on cut optimality and assume the objective is to connect an undirected weighted graph. A shortest-path tree solves a different problem.

7.2.4 Flows and matchings

A residual graph records unused forward capacity and cancellable reverse flow. Augment only along residual edges with positive capacity. Dinic's algorithm alternates a BFS level graph with DFS blocking flows; it is a practical default for many contest networks. Bipartite matching is a unit-capacity flow instance, though Hopcroft–Karp gives a specialized $O(E\sqrt{V})$ bound.

Modeling details dominate flow problems: split a vertex into in/out copies to impose vertex capacity, add a super-source or super-sink for multiple terminals, and use lower-bound transformations only after satisfying flow conservation.

7.2.5 Graph review checklist

- Confirm direction, indexing, and whether parallel edges or self-loops are allowed.
- Initialize unreachable distances safely and avoid overflow in $d[u] + w$.
- Distinguish a tree, a spanning tree, a shortest-path tree, and a DAG.
- Test disconnected inputs and reconstruct paths with parent edges only after reachability is known.

7.3 Advanced workshop

7.3.1 Modeling first, selecting the graph algorithm second

Graph problems are often difficult because the vertices and edges are hidden inside the story. A vertex may represent a location, a task, a configuration, or a pair consisting of a location and a resource state. An edge may represent physical travel, precedence, compatibility, or one legal transition. Before selecting an algorithm, write the graph model and state what a path, cut, matching, or component means in the original problem. Algorithm choice then follows from structure. Breadth-first search solves unweighted shortest paths; Dijkstra requires nonnegative weights; topological dynamic programming uses acyclicity; minimum spanning trees optimize connectivity rather than a route; flow models conserved capacity; bipartite matching models pairwise assignments. State expansion can remove a global-looking constraint by making the relevant resource or parity part of each vertex, but the expanded size must still fit the limits.

7.3.2 Correctness-proof blueprint

1. Prove a correspondence between feasible story solutions and the graph objects being optimized.
2. State the invariant of the chosen algorithm, such as finalized shortest distances or a maintained valid residual flow.
3. Translate optimality back to the original problem, including why no omitted edge or state can improve the answer.

7.3.3 Implementation notes across languages

- Use adjacency lists unless the graph is dense or the algorithm explicitly needs a matrix.
- In Dijkstra, ignore stale priority-queue entries and use a distance type large enough for the longest possible path.
- For iterative traversals, mark a vertex at insertion time when duplicate work would otherwise grow rapidly.
- In residual graphs, store reverse-edge indices so every augmentation updates both directions consistently.

7.3.4 Adversarial testing matrix

Case	Construction	What it exposes
Disconnected graph	Target unreachable	Sentinel and reporting
Parallel edges	Different weights or capacities	Input modeling
Zero-weight cycle	Nonnegative cyclic graph	Queue discipline
Multiple optima	Equal paths or matchings	Correctness without uniqueness

7.3.5 Discussion prompts

1. What precisely is a vertex, and what real-world action creates an edge?
2. Which structural promise rules out a more general and expensive algorithm?
3. Does adding state dimensions make a hard constraint local at an acceptable size?

7.3.6 Stretch directions

- Convert a shortest-path problem with one discounted edge into a layered graph.
- Prove the cut interpretation of a small assignment or selection problem modeled as flow.
- Generate random small graphs and compare an optimized solver with exhaustive enumeration.

7.4 Detailed case study

7.4.1 A layered shortest path with one discount

Scenario. Find a shortest route from s to t in a nonnegative weighted directed graph when at most one edge may be traversed at half cost. The discount must become part of the state rather than a special case outside the graph.

7.4.2 Step-by-step trace

1. Create two copies of every vertex: $(v, 0)$ means the discount is unused and $(v, 1)$ means it has been used.
2. For each original edge $u \to v$ of weight w, add $(u, 0) \to (v, 0)$ and $(u, 1) \to (v, 1)$ with weight w.
3. Add the cross-layer edge $(u, 0) \to (v, 1)$ with weight $w/2$. There is no edge back from layer 1 to layer 0, so the discount cannot be reused.
4. Run Dijkstra from $(s, 0)$. The answer is $\min(dist[(t, 0)], dist[(t, 1)])$ because using the discount is optional.
5. The expanded graph has $2V$ vertices and $3E$ edges, preserving $O((V + E) \log V)$ asymptotic complexity.

7.4.3 Correctness argument

Every legal original route maps to exactly one layered path: remain in layer 0 until the discounted edge, cross once, then remain in layer 1. Conversely, every layered path crosses at most once and therefore represents a legal route. Corresponding path costs are equal, so the shortest layered path gives the optimum.

7.4.4 Benchmark plan

Family	Scale or construction	Purpose
Discount unused	Already optimal zero edge	Optional transition
Parallel edges	Different weights	Edge preservation
Random small graph	$V \le 8$	Compare with edge enumeration

7.4.5 Coach-check questions

1. How many layers are required if up to k discounts may be used?
2. What changes if the discounted weight depends on how many discounts remain?
3. Why does the method require nonnegative expanded-edge weights for Dijkstra?

7.5 Study framework

7.5.1 Prerequisites

- BFS, DFS, queues, priority queues, and adjacency lists.
- Proof by invariant and basic greedy reasoning.
- Trees, directed graphs, and weighted paths.

7.5.2 Learning outcomes

- Encode the right graph state before selecting an algorithm.
- Match edge-weight structure to BFS, 0–1 BFS, Dijkstra, or DAG relaxation.
- Use components, contraction, spanning trees, and flows as modeling tools.

7.6 Worked example

7.6.1 Why 0–1 BFS uses both ends of a deque

Imagine a grid in which following the arrow printed in a cell costs zero and choosing any other outgoing direction costs one. The graph has one vertex per cell and edge weights only zero or one. Ordinary BFS is invalid because edges have unequal costs; Dijkstra works, but a heap is unnecessary.

Maintain a deque. When relaxing a zero-cost edge, push the improved vertex to the front; for a one-cost edge, push it to the back. At every step the deque contains vertices in nondecreasing tentative distance: a new distance is either the current distance or exactly one larger. Removing a stale entry or checking the stored distance prevents repeated work from changing correctness.

This is also a modeling example. The cost belongs to the action, not to the destination cell. If the rule depended on the previous direction, the vertex state would need to include that direction. Expanding the state is correct when it makes the future independent of the earlier path.

7.7 Implementation template

7.7.1 Pseudocode

```
dist = array(V, INF)
dist[source] = 0
deque.push_front(source)

while deque not empty:
    v = deque.pop_front()
    for edge (v, u, w):          # w is 0 or 1
        if dist[v] + w < dist[u]:
            dist[u] = dist[v] + w
            parent[u] = v
            if w == 0: deque.push_front(u)
            else:      deque.push_back(u)

# For Dijkstra, replace the deque with a min-heap and
# ignore a popped pair when its key differs from dist[v].
```

7.7.2 Complexity reference

Method	Bound	Best fit
BFS / DFS	$O(V+E)$	Unweighted reachability and components.
0–1 BFS	$O(V+E)$	Binary edge weights.
Dijkstra	$O((V+E)\log V)$	General nonnegative weights.
Dinic flow	$O(V^2E)$ general	Often faster on contest networks.

7.7.3 Common mistakes

- Marking a Dijkstra vertex final when inserted instead of when removed at minimum distance.
- Forgetting that a shortest-path tree is not a minimum spanning tree.
- Storing an enormous implicit graph instead of generating neighbors.
- Omitting reverse residual edges in a flow implementation.

7.8 Practice lab

7.8.1 Short exercises

1. Construct a three-vertex counterexample showing ordinary BFS fails with weights zero and one.
2. Add a parity bit to a shortest-path state and describe the expanded graph.
3. Explain why contracting strongly connected components always produces a DAG.

7.8.2 Guided problem progression

1. **Currents**. Classify edge costs and practice shortest-path modeling.
2. **Dark Ride**. Separate graph construction from the optimization algorithm.
3. **Wind Turbines**. Look for connectivity and spanning structure.
4. **Maximum Color Clique**. Finish with a more demanding structural graph problem.

7.8.3 Submission workflow

1. Write the invariant, state meaning, or mathematical precondition before coding.
2. Build minimal examples for every boundary case identified in the chapter.
3. Compare against a brute-force solver on small random instances whenever practical.
4. Compare shortest paths with a small all-pairs oracle and include disconnected graphs, parallel edges, and stale heap entries.
5. After acceptance, record the decisive idea, complexity, and one implementation hazard.

Complete lecture resources
Open the MIT materials folder in Google Drive. No lecture deck was present at the review date; use this folder to check for later additions. Access may require an authorized ICPC Foundation account.

7.9 Problem Set

9 problems • Instructor(s): Jaehyun Koo

Assignment: Open on Kattis

Planning key: mark each problem Foundation, Core, or Stretch after an opening scan; record a personal target time before starting. Kattis difficulty and availability can change, so this booklet does not freeze a platform rating.

1. A String Problem — kattis: stringproblem

tier: F / C / S • target: ____ min • technique: _____

2. Currents — kattis: currents

tier: F / C / S • target: ____ min • technique: _____

3. Gift Boxes — kattis: giftboxes

tier: F / C / S • target: ____ min • technique: _____

4. Dark Ride — kattis: darkride
tier: F / C / S • target: ___ min • technique: _____

5. IMO — kattis: imo
tier: F / C / S • target: ___ min • technique: _____

6. Wind Turbines — kattis: windturbines
tier: F / C / S • target: ___ min • technique: _____

7. Dodgeball Diplomacy — kattis: diplomacy2
tier: F / C / S • target: ___ min • technique: _____

8. Garbage Collection — kattis: garbage
tier: F / C / S • target: ___ min • technique: _____

9. Xoracle — kattis: xoracle
tier: F / C / S • target: ___ min • technique: _____

Tip. Open Metadata on a problem page to verify author, source, limits, downloads, and license before redistributing content.

7.10 Timed Contest

11 problems • Instructor(s): Jaehyun Koo

Assignment: Open on Kattis

Planning key: mark each problem Foundation, Core, or Stretch after an opening scan; record a personal target time before starting. Kattis difficulty and availability can change, so this booklet does not freeze a platform rating.

1. Pieces of Parentheses — kattis: piecesofparentheses
tier: F / C / S • target: ___ min • technique: _____

2. Stars in a Can — kattis: starsinacan
tier: F / C / S • target: ___ min • technique: _____

3. Stretching Streamers — kattis: stretchingstreamers
tier: F / C / S • target: ___ min • technique: _____

4. Heaps from Trees — kattis: heapstrees
tier: F / C / S • target: ___ min • technique: _____

5. Blazing New Trails — kattis: blazingnewtrails
tier: F / C / S • target: ___ min • technique: _____

6. Incremental Double Free Strings — kattis: idf
tier: F / C / S • target: ___ min • technique: _____

7. Apple Market — kattis: applemarket
tier: F / C / S • target: ___ min • technique: _____

8. Maximum Color Clique — kattis: maxcolorclique
tier: F / C / S • target: ___ min • technique: _____

9. Ski Resort — kattis: skiresort
tier: F / C / S • target: ___ min • technique: _____

10. Yin and Yang Stones — kattis: yinyangstones

tier: F / C / S • target: ___ min • technique: _____

11. Unbalanced Parentheses — kattis: unbalancedparentheses

tier: F / C / S • target: ___ min • technique: _____

Tip. Open Metadata on a problem page to verify author, source, limits, downloads, and license before redistributing content.

7.11 Implementation reference

7.11.1 C++17 tested reference excerpt

Dijkstra on a two-layer graph for a route with at most one half-price edge. The state records whether the discount has been consumed.

```
using ll=long long;
const ll INF=(1LL<<62);
vector<array<ll,2>> dist(n,{INF,INF});
using State=tuple<ll,int,int>; // distance, vertex, used
priority_queue<State,vector<State>,greater<State>> pq;
dist[s][0]=0; pq.push({0,s,0});
while(!pq.empty()){
    auto [d,v,used]=pq.top(); pq.pop();
    if(d!=dist[v][used]) continue;
    for(auto [u,w]:adj[v]){
        if(d+w<dist[u][used]){
            dist[u][used]=d+w;
            pq.push({d+w,u,used});
        }
        if(!used && d+w/2<dist[u][1]){
            dist[u][1]=d+w/2;
            pq.push({d+w/2,u,1});
        }
    }
}
ll answer=min(dist[t][0],dist[t][1]);
```

7.11.2 Template contract

- All expanded-edge weights are nonnegative and integer division matches the problem rule.
- State 0 means unused and state 1 means already used; transitions never go backward.
- Stale heap entries are discarded before relaxation.

7.11.3 Porting and diagnostics

Language notes

- Java: use a small immutable heap record or packed array and compare distances with `long`.
- Python: tuples work directly in `heapq`; localize adjacency and distance references for speed.
- C++: guard additions if weights and paths can approach the infinity sentinel.

Diagnostic probes

1. The best path uses no discount, uses it first, uses it last, or has equal alternatives.
2. Unreachable targets, self-loops, and parallel edges.
3. Random small graphs compared by enumerating the discounted edge.
4. Large sparse graphs that maximize heap churn and stale entries.

Performance envelope

Measure	Bound	Interpretation
Vertices	$2V$	One state bit.
Edges	$3E$	Two normal and one discount transition.
Time	$O((V+E)\log V)$	Binary-heap Dijkstra.

Submission audit

- Match every array dimension and numeric type to the largest legal instance.
- Run the diagnostic probes before optimization, then preserve them as regression tests.
- State the invariant and the complexity in the solution notes before submission.

7.12 Coach lesson plan

7.12.1 Ninety-minute session objective

Translate a story into graph state, edges, and the least-general valid algorithm.

7.12.2 Agenda

Time	Activity	Evidence
0–10 min	Retrieval warm-up	Learners restate the chapter invariant without notes.
10–25 min	Model critique	Teams compare two candidate models and reject one with a counterexample.
25–45 min	Guided trace	The class completes the detailed case study and predicts each intermediate state.
45–70 min	Implementation lab	Pairs adapt the reference skeleton and run at least three diagnostic probes.
70–90 min	Debrief and transfer	Learners identify the constraint that selects the method on a new problem.

7.12.3 Misconceptions to surface

- Vertices represent the wrong real-world object.
- Dijkstra is used with negative expanded weights.
- State expansion is correct but too large.

7.12.4 Instructor checkpoints

1. Ask for a counterexample to an incomplete state, predicate, or graph model.
2. Require one learner to justify correctness while another audits numeric and asymptotic bounds.
3. Do not reveal the final transition until teams can name the invariant it must preserve.

7.12.5 Exit ticket

1. In at most 25 words, define vertices and edges so every legal action has one graph counterpart.
2. Name a graph that breaks the chosen traversal or an early-visited optimization.
3. Identify the first weight, capacity, or state-space change that requires a different algorithm.

Part II

Common Contest Reference

Chapter 8

Stress Testing and Differential Checks

TURN PLAUSIBLE CODE INTO EVIDENCE

A stress test compares the intended solver with a simpler oracle on many small generated instances. It is most valuable when the optimized solution has complicated states, numeric edge cases, or several interacting data structures.

8.1 Four-component harness

1. **Generator.** Produce valid small instances with a deterministic seed. Bias some cases toward empty, maximal, repeated, symmetric, and nearly degenerate structures.
2. **Oracle.** Use exhaustive search, direct simulation, or a slower textbook algorithm whose correctness is easy to inspect.
3. **Comparator.** Normalize outputs before comparing. For floating point, compare according to the problem's accepted tolerance or validate the defining constraints directly.
4. **Reducer.** When a mismatch appears, delete vertices, operations, characters, or constraints while preserving the failure. A small counterexample teaches more than a large random seed.

8.2 Failure-preserving loop

```
for seed = 0, 1, 2, ...:
    instance = generate(seed)
    expected = brute_force(instance)
    received = optimized(instance)
    if normalize(expected) != normalize(received):
        print(seed, shrink(instance))
        stop
```

8.3 Harness audit

Test the oracle on hand-computed examples; ensure the generator reaches every structural family; print the seed and full input; and run sanitizers or runtime assertions during local testing. Never submit the generator, oracle, debug output, or expensive invariant checks with the contest solution.

Chapter 9

Numeric Safety and Boundary Discipline

BUDGET EVERY VALUE BEFORE CODING

Numeric correctness begins with a bound for every intermediate expression, not only the final answer.

Risk	Typical trigger	Required response
Integer overflow	Products, path sums, cross products	Promote before arithmetic; prove a bound; use wider or arbitrary-precision integers.
Sentinel overflow	Adding to infinity	Guard impossible states before addition and keep the sentinel below the type maximum.
Floating comparison	Tangency, rank, binary search	Scale the tolerance, separate predicates from constructions, and define equality policy.
Modular division	Inverses and factorial ratios	Check gcd and modulus assumptions before applying an inverse.
Index failure	Empty ranges and $n-1$	Adopt one interval convention and assert conversion boundaries.

9.1 Pre-submission numeric audit

1. Write the largest possible count, sum, product, distance, and modulus on paper.
2. Check the type of each operand at the moment arithmetic occurs; a later cast cannot repair earlier overflow.
3. Test zero, one, maximum size, maximum magnitude, negative values when legal, and values immediately around every threshold.
4. For binary search, record the invariant for both endpoints and prove termination under the chosen integer or floating update rule.

Key idea. A numeric assumption belongs in the correctness proof. If the proof needs exact sign, do not delegate that decision to a rounded construction.

Chapter 10

Team Contest Workflow

CONVERT SHARED TIME INTO ACCEPTED SOLUTIONS

The team should optimize information flow as deliberately as algorithmic complexity.

10.1 Opening scan

During the first 10–15 minutes, every teammate reads different problems and records three items: likely technique, implementation risk, and confidence. Select an early problem with a short proof and low debugging risk, not merely the shortest statement.

10.2 Ownership protocol

For each active problem, name a primary implementer and a reviewer. Before coding, the implementer states the model, invariant, complexity, and two edge cases. The reviewer challenges one assumption and confirms the input-output contract. A teammate may switch problems, but ownership must be transferred explicitly with current findings and failed approaches.

10.3 Submission checklist

1. Re-read limits, indexing, required precision, and output formatting.
2. Run samples plus a minimal, maximal, degenerate, and hand-computed case.
3. Remove debug output; preserve assertions only when they are safe under full constraints.
4. Estimate the worst-case operation count and memory footprint one final time.
5. After a wrong answer, classify the evidence before changing code: model, proof, implementation, numeric, or interface.

10.4 Scoreboard discipline

Use scoreboard information as weak evidence, never as a substitute for analysis. A widely solved problem may still mismatch the team's strengths. Near the end, freeze new high-risk implementations, prioritize review and testing, and assign one teammate to audit every pending submission's boundary cases.

Chapter 11

Post-Contest Review and Curriculum Loop

MAKE EVERY CONTEST IMPROVE THE NEXT ONE

A contest is complete only after its decisions and failures have been converted into reusable knowledge.

11.1 Review record

For each attempted problem, store the decisive model, the invariant or theorem, final complexity, first wrong assumption, smallest failing test, and one implementation hazard. Link to the authoritative problem page rather than copying a statement whose license may differ.

11.2 Failure taxonomy

Model	The graph, state, geometry object, or algebraic representation was wrong.
Proof	The transition omitted cases or used a theorem outside its preconditions.
Code	The model was correct but indexing, mutation, or reconstruction violated it.
Numeric	Overflow, rounding, sentinel arithmetic, or modular division changed the result.
Process	Time allocation, communication, review, or test selection failed.

11.3 Three-pass learning cycle

Within 24 hours, reconstruct the intended solution without copying code. Within one week, implement it from a blank file and pass the original tests. Within one month, solve a transfer problem that uses the same invariant in a different surface form.

11.4 Curriculum maintenance

Promote repeated failures into team drills and template tests. Retire a template when no teammate can explain its invariant. Keep references versioned, cite their sources, and check Kattis metadata before redistributing any problem statement, illustration, or data file.

Part III

Instructor and Publication Reference

Chapter 12

Short-Exercise Answer Notes

CONCISE CHECKS, NOT FULL SOLUTION EDITORIALS

These notes give the expected invariant or result for each three-question practice lab. Coaches should accept equivalent arguments and should ask learners to supply missing implementation details and boundary tests.

12.1 UCF: Dynamic Programming

1. For vertex weight w_v, use $dp[v][1] = w_v + \sum_u dp[u][0]$ and $dp[v][0] = \sum_u \max(dp[u][0], dp[u][1])$ over children u.
2. For minimum vertex cover, $vc[v][1] = 1 + \sum_u \min(vc[u][0], vc[u][1])$ and $vc[v][0] = \sum_u vc[u][1]$.
3. If u is a child of v, then $answer[u] = answer[v] + n - 2\,size[u]$: the vertices in u's subtree become one step closer and all others one step farther.

12.2 Rose-Hulman: Data Structures

1. With one-based Fenwick indexing, `update(6)` visits $6, 8, 16, \ldots$ while indices remain within the array; `prefix(13)` visits $13, 12, 8$.
2. Represent a pending tag as optional assignment plus addition. A new assignment replaces both old components; a new addition increments the addition component. Push assignment before addition.
3. Under union by size, a node's depth increases only when its component joins one at least as large, so its component size at least doubles. Its depth is therefore at most $\lfloor \log_2 n \rfloor$.

12.3 UT Austin: Geometry

1. Examples: $(0,0)$–$(2,2)$ with $(0,2)$–$(2,0)$ crosses; $(0,0)$–$(1,0)$ with $(1,0)$–$(2,1)$ touches; $(0,0)$–$(3,0)$ with $(1,0)$–$(4,0)$ overlaps; $(0,0)$–$(1,0)$ with $(2,0)$–$(3,0)$ is disjoint and collinear.
2. Increasing the candidate radius inflates every forbidden region. Any placement feasible at radius r_2 is feasible at every $r_1 \le r_2$; therefore feasibility is monotone nonincreasing in radius.
3. Count an edge only when its endpoints lie on opposite sides of the horizontal ray using $(a_y > p_y) \neq (b_y > p_y)$, then test the crossing coordinate. The strict comparison makes the vertical interval half open and counts a vertex once.

12.4 McGill: Combinatorics and Number Theory

1. The difference $7 - 3 = 4$ is divisible by $\gcd(8, 12) = 4$, so the system is compatible; the normalized merged result is $x \equiv 19 \pmod{24}$.
2. If $n = \prod_i p_i^{e_i}$, then $\tau(n) = \prod_i (e_i + 1)$ and $\sigma(n) = \prod_i (1 + p_i + \cdots + p_i^{e_i}) = \prod_i (p_i^{e_i+1} - 1)/(p_i - 1)$.
3. When the prime modulus divides a denominator, that denominator is zero in the field and has no multiplicative inverse; the factorial/inverse-factorial formula is therefore outside its preconditions.

12.5 Columbia: Gaussian Elimination and FFT

1. A consistent system of rank two in four variables has two free variables and infinitely many solutions.
2. Multiplication by $1/(1 - x) = 1 + x + x^2 + \cdots$ convolves the coefficient sequence with all ones, so coefficient n becomes the prefix sum through n.
3. The product has at most $700 + 900 - 1 = 1599$ coefficients; the usual radix-two transform length is therefore 2048.

12.6 UIUC: Strings

1. For `abcababcab`, the prefix function is $[0, 0, 0, 1, 2, 1, 2, 3, 4, 5]$; the nonempty proper border lengths are 5 and 2.
2. Each successful comparison advances the text, and each fallback strictly decreases the matched-prefix length. Across the scan, total advances and fallbacks are linear.
3. In the suffix array of `banana`, adjacent suffixes `ana` and `anana` have LCP 3, revealing the longest repeated substring `ana`.

12.7 MIT: Graphs

1. Let $s \to a$ cost 1, $s \to b$ cost 0, and $b \to a$ cost 0. Minimum edge count prefers the direct edge, while minimum cost prefers the two-edge path; ordinary BFS therefore solves the wrong objective.
2. Replace each vertex v by $(v, 0)$ and $(v, 1)$. An edge toggles the parity bit when parity records path length; source and accepted target layers encode the required parity.
3. A directed cycle among two or more contracted components would make every component on that cycle mutually reachable, contradicting their maximality as distinct strongly connected components.

12.8 Exit-ticket rubric

A complete exit ticket must define the model precisely, provide a concrete failing input rather than a category name, and identify a numeric or asymptotic threshold that genuinely changes the algorithm. Award partial credit when the idea is correct but the boundary or invariant is not yet explicit.

Chapter 13

Algorithm and Problem Map

FIND THE RIGHT CHAPTER QUICKLY

Chapter	Core tools	Start with
UCF	State design, tree DP, rerooting, small-to-large	Airport Coffee; Range Editing
Rose-Hulman	Fenwick tree, segment tree, lazy propagation, DSU	Fenwick Tree; Union-Find
UT Austin	Orientation, segment intersection, angular sweep	Criss-Cross; Drawing Circles
McGill	GCD, modular inverse, CRT, divisor formulas	Coprime Integers; Divisors
Columbia	Elimination, rank, generating functions, FFT/NTT	Fun with Fibonacci; The Big Painting
UIUC	KMP, hashing, suffix order, BWT	String Matching; Power Strings
MIT	Graph modeling, shortest paths, SCC, flows	Currents; Dark Ride

13.1 Problem-selection sequence

Use Foundation entries to validate the chapter template, Core entries to practice modeling, and Stretch entries to test transfer. The planning fields beside every catalog item are learner-relative; they are intentionally not presented as official Kattis ratings.

Chapter 14

Glossary and Acronyms

BFS	Breadth-first search; shortest paths only when every edge has equal cost.
BWT	Burrows–Wheeler transform; a reversible ordering of cyclic rotations used in string indexing.
DAG	Directed acyclic graph.
DP	Dynamic programming; optimization or counting over reusable subproblems with explicit state meaning.
DSU	Disjoint-set union, also called union–find.
FFT	Fast Fourier transform over complex roots of unity.
Invariant	A statement that remains true throughout an algorithm and supports its correctness proof.
KMP	Knuth–Morris–Pratt linear-time pattern matching using prefix information.
NTT	Number-theoretic transform over a finite field.
Relaxation	An attempt to improve the known value of a state through one edge or transition.
SCC	Strongly connected component of a directed graph.
Stable oracle	A simpler implementation used to check an optimized solver on small instances.

Chapter 15

Print Link Directory

SHORT DESTINATIONS AND ASSIGNMENT CODES

For print readers, begin at the short stable roots below, then use the host and assignment code from the table. The digital PDF links directly to every destination.

Program	https://nac.icpc.global/napc/
Trainers	https://nac.icpc.global/napc-trainers/
Kattis course	https://napc-o.kattis.com/courses/2026-pre-nac
Corrections	https://foundation.icpc.global/contact/

Host	Practice code	Timed code
UCF	xjdn8u	dc43ky
Rose-Hulman	wbppmh	vfobk7
UT Austin	chjxjo	i2r7fu
McGill	my4b4c	yhwp2m
Columbia	vyxswb	roznwm
UIUC	ouxxcc	n7u9hw
MIT	sw8egk	zac4dv

Teaching Network and Sources

The public course pages identify the following instructors and teaching staff.

UCF	Arup Ratan Guha
Rose-Hulman	Rachel Krohn
UT Austin	Etienne Vouga
McGill	Ari Blondal; David Becerra
Columbia	Christian Yongwhan Lim; Josh Alman; Grace Lim
UIUC	Mattox Beckman
MIT	Jaehyun Koo

Course source

2026 NAPC-O — 2026-pre-nac

Official ICPC program sources

Training Programs: NAPC and NAPC-O
NAPC-O and NAPC Trainer Profiles

Internal course-public materials reviewed

NAPC-O shared Google Drive folder: geometry decks from UT Austin; dynamic-programming notes from UCF; a data-structures deck from Rose-Hulman; and Gaussian-elimination, generating-functions, FFT, and NTT decks from Columbia. Authorized ICPC Foundation Gmail metadata, including Zoom asset notices, was used only for delivery verification and is not republished; the Zoom recording portal was not used as a content source.

No lecture deck was available at review time for McGill, UIUC, or MIT. Those three lecture guides are explicitly labeled published-course roadmaps and are based only on public program metadata and visible assignment sequences.

Sources were reviewed on September 7, 2026. Schedules, assignments, biographies, availability, and links may change after publication.

Technical reference basis

The technical chapters are original editorial synthesis checked against standard algorithmic definitions and the cited course-public materials. They do not claim to reproduce an instructor's exact lecture. Problem-specific statements, constraints, ratings, and licenses remain authoritative only on Kattis.

Publication Notes

Revision history

Version	Date	Changes
1.0	Aug. 14, 2026	Initial 100-page course catalog snapshot.
1.1	Aug. 23, 2026	Source provenance, access and spoiler guidance, learner planning fields, chapter-specific diagnostics, exercise answers, glossary, algorithm map, print links, contact route, accessibility improvements, and automated quality checks.
1.2	Sep. 7, 2026	Final trainer review; Rachel Krohn profile and guided progression updates; technical wording corrections; refreshed link, rights, and accessibility audit.

Accessibility and alternate formats

This edition uses a logical heading hierarchy, PDF bookmarks, selectable Unicode-mapped text, descriptive link text, and text labels in addition to color. Tables repeat their meaning in headings and surrounding prose. The legacy pdfLaTeX output is not claimed to be a fully tagged PDF; readers who require a tagged or reflowable version should request an alternate format through the ICPC Foundation contact page.

Quality gates

The project build checks the expected 14 assignments and 163 problem placements, scans the final log for unresolved references and duplicate destinations, and compiles and executes the standalone C++17 reference tests when a compiler is available. Network link validation is a separate opt-in check because Kattis and Google Drive may redirect to authenticated pages.

Corrections and rights

Report broken links, factual corrections, accessibility requests, or permissions questions through foundation.icpc.global/contact or info@icpc.foundation. Include the edition number, page, and section title. Do not send passwords, recording passcodes, or private participant information.

This document was typeset using LaTeX and compiled with pdfLaTeX.
Font: Latin Modern family.

NAPC-O 2026 Course Booklet
ICPC Foundation

For the latest course catalog, visit:
https://napc-o.kattis.com/courses