# BECOMING NATIONAL CHAMPION

## FIRST EDITION

IRFAN SUBAKTI


Profitina

profitina.com

# TABLE OF CONTENTS

SESSION 1 · STRATEGY SESSION

# The Champion's Playbook: Contest-Day Strategy, Tips & Tricks

Medals are not decided during the contest — they are decided by how you train and by your execution discipline on the day. This session gives you a champion's standard operating procedure: from the first minute of reading to the final submit, everything measured, everything deliberate.

**Session goals — after these 2 hours you can:**

(1) Run the first-10-minutes protocol: map every problem before writing a single line of code

(2) Allocate the 5 OSN hours with strict cut-loss rules

(3) Hunt subtasks: turn a 100-point problem into a climbable 15-30-55 ladder

(4) Debug by protocol, not panic: small tests, edge tests, brute-force cross-checks

(5) Read a two-page statement and extract its formal model in 5 minutes

## 1.1 Before the Contest: Ammunition Is Prepared, Not Improvised

A national champion does not write a segment tree from scratch in the contest hall — they pour it out of muscle memory trained hundreds of times. Prepare and memorize a personal template kit: fast I/O, DSU, BFS/DFS, Dijkstra, segment tree, BIT, modular combinatorics. Practice rewriting each in under 4 minutes without references.

- Sleep well the last two nights; your brain at contest hour 4 is determined by sleep, not coffee.
- Eat a complex-carb breakfast; bring water. Two percent dehydration measurably lowers concentration.
- Know the contest machine's editor and compiler: `-O2 -std=c++17` flags, stack limits, how the local judge works.
- Arrive early; the same small ritual before every contest lowers anxiety.

**Champion's trick: the error notebook**

After every practice, write ONE sentence: which bug wasted the most time today. Before the contest, reread that list — you are reading your personal trap map.

## 1.2 Minutes 0-10: The Problem-Mapping Protocol

Do not touch the keyboard. Read ALL the problems first. For each, write on paper: (1) its formal model in one sentence, (2) the bounds on N and their complexity implication, (3) a guessed algorithm class, (4) the subtask scores. Order your plan by highest expected points per minute (Table 1.1).

| Bound on N | Safe complexity | Typical algorithm class |
|---|---|---|
| N ≤ 20 | O(2^N · N) | bitmask, meet in the middle, backtracking |
| N ≤ 500 | $O(N^3)$ | table DP, Floyd-Warshall |
| N ≤ 5.000 | $O(N^2)$ | quadratic DP, smart brute force |
| N ≤ 200.000 | O(N log N) | sorting, segment tree/BIT, D&C, binary search on answer |
| N ≤ $10^7$ | O(N) | sieves, prefixes, two pointers |
| N ≤ $10^{18}$ | O(log N) | matrix power, fast doubling, digit DP |

*Table 1.1 — Reading the bounds = reading the problemsetter's intent.*

**Fatal beginner mistake #1**

Falling in love with problem one and burning 2 hours there. Champion's rule: 45 minutes without measurable progress means LEAVE — grab the subtasks, move on. Return only after the other problems are secured.

## 1.3 Subtask Hunting: The Mathematics of Medals

OSN is scored per subtask, IOI style. The national champion is almost never the one who solves everything perfectly — it is the one who never leaves cheap points on the table. 100 points from four problems × 25-40 subtask points beats 100 from one perfect solve plus three zeros.

- The 'small N' subtask is almost always a brute force you can write in 10 minutes. Write it. Submit it. Bank it.
- That same brute force later becomes the verifier of your full solution — two benefits, one piece of code.
- Keep submitted solutions safe; never overwrite AC code with experiments.
- Final 15 minutes: stop writing anything new; make sure every half-done solution is submitted for its subtasks.

## 1.4 The Debug Protocol: Cold, Fast, Systematic

- Reproduce with the SMALLEST failing case — shrink the input until the bug is naked.
- Automatic edge tests: N=1, all equal, all negative, empty, maximum. This kills 60% of WAs.
- Cross-check: brute force + random generator + comparison script. Thirty lines that save medals.
- Overflow: any product of two ints that can pass $2\times10^9$ → `long long`. Write this on your paper.
- Off-by-one: 1-based or 0-based — pick ONE lifetime convention and never switch mid-contest.

```cpp
// Kerangka stress-test — hafalkan polanya / Stress-test skeleton — memorize the pattern
// g++ -O2 sol.cpp -o sol && g++ -O2 brute.cpp -o brute && g++ -O2 gen.cpp -o gen
for (int i = 0; i < 1000; i++) {
  system("./gen > in.txt");
  system("./sol   < in.txt > out1.txt");
  system("./brute < in.txt > out2.txt");
  if (system("diff -q out1.txt out2.txt")) { puts("BEDA! / DIFF!"); break; }
}
```

**The psychological trick when panicking**

Breathe, then write down what is CERTAINLY correct (tested) versus what is NOT yet. Panic comes from mixing the two. Champions separate them in ink.

## 1.5 HARD Case Study: HOTLINE (SPOJ #13)

**Problem card**

SPOJ HOTLINE | Classical | Hard | Natural-Language Statement Parsing

Source: https://www.spoj.com/problems/HOTLINE/ — solve and submit directly on SPOJ.

### Problem statement (concise, faithful to the original SPOJ text)

A dialogue consists of simple English sentences: statements (ending '.'), questions (ending '?'), with subjects like i, you, everybody, nobody, or names. The program must remember all positive and negative statements, detect contradictions, then answer 'do/does ...?' and 'who ...?' questions exactly following the grammar and answer-format rules of the statement. One stray character = Wrong Answer: the purest reading test there is.

- Multiple dialogues; each ends with a blank line — output 'Dialogue #k:' then each question followed by its answer.
- Verb rules: is/are/am, third-person -s forms, don't/doesn't — all must be normalized.

## Phases 1-2 — Understand and Model

Build statement maps: 'subject|verb|object' keys into two dictionaries (positive and negative). Every new sentence is normalized (verbs to base form, don't/doesn't marking negatives), stored without duplicates, and collision-checked; every question is answered purely from the statements seen so far.

## Phase 3 — Design the Algorithm

1. Split each line by its ending mark (., ?, !) into words; normalize verbs to base form and detect negation (don't/doesn't, subject 'nobody').
2. Store new facts into the positive/negative maps; the instant a colliding pair appears — including via 'everybody'/'nobody' — flag the whole dialogue contradictory ('I am abroad.').
3. 'Do/does X ...?' questions: look up the specific key, then the sheltering 'everybody'/'nobody' keys; answer yes/no/maybe with correct i ↔ you swapping.
4. 'Who ...?' questions: collect all matching positive subjects in appearance order, handling 'everybody' and the 'A, B and C' list format.

## Phase 4 — Complexity Analysis

At most 100 statements per dialogue; each question scans maps of size ≤ 100 — linear total work, far under the limit. All of this problem's difficulty lives in rule fidelity, not speed.

## Phase 5 — Implementation (Public Edition — ~50% revealed, ~50% your task)

This is the Public Edition of this solution. Roughly half of the code is shown below; the rest — the part that actually solves the problem — is your task, guided by the hints beneath the code and the walkthrough in Phases 1-4 above.

```
#include <cstdio>
#include <cstring>
#include <string>
#include <vector>
#include <map>
#include <set>
using namespace std;
static char buf[120];
bool isIorYou(const string& s){return s=="I"||s=="you";}
string verbS(const string& v,const string& s){return isIorYou(s)?v:v+"s";}
string dontS(const string& s){return isIorYou(s)?"don't":"doesn't";}
string swap_iy(const string& s){if(s=="I")return "you";if(s=="you")return
"I";return s;}
string baseVerb(const string& v){
  if(!v.empty()&&v.back()=='s')return v.substr(0,v.size()-1);
  return v;
}
struct Stmt{string subj;bool neg;string verb,obj;};
void processDialogue(){
  vector<Stmt> stmts;
  map<string,bool> posMap,negMap;
  bool contradiction=false;
  vector<pair<string,string>> out;
  while(fgets(buf,sizeof(buf),stdin)){
    int len=strlen(buf);
    while(len>0&&(buf[len-1]=='\n'||buf[len-1]=='\r'))buf[--len]=0;
```

```
    if(!len)continue;
    char end=0;
    vector<string> words;string word;
    for(int i=0;i<len;i++){
      char c=buf[i];
      if(c=='.'||c=='?'||c=='!'){if(!word.empty())
{words.push_back(word);word.clear();}end=c;break;}
      else if(c==' '){if(!word.empty()){words.push_back(word);word.clear();}}
      else word+=c;
    }
    if(end=='!'){
      for(auto&[q,a]:out)printf("%s\n%s\n\n",q.c_str(),a.c_str());
      printf("%s\n\n",buf);return;
    }
    if(end=='.'&&words.size()>=2){
      string subj=words[0];bool neg=false;string verb,obj;
      if(words[1]=="don't"||words[1]=="doesn't"){
        neg=true;if(words.size()<3)continue;
        verb=words[2];
        for(int i=3;i<(int)words.size();i++){if(!obj.empty())obj+="
";obj+=words[i];}
      }else{
        string vs=words[1];
        if(subj=="everybody"||subj=="nobody")verb=baseVerb(vs);
        else if(isIorYou(subj))verb=vs;
        else verb=baseVerb(vs);
        for(int i=2;i<(int)words.size();i++){if(!obj.empty())obj+="
";obj+=words[i];}
        if(subj=="nobody")neg=true;
      }
      string key=subj+"|"+verb+"|"+obj;
/* ================================================================
   >>> PUBLIC EDITION — roughly half of this solution is hidden for you to
complete. <<<
Rebuild it using the hints below the code and the reasoning in Phases 1-4 above.
   ================================================================ */
    }
  }
}
int main(){
  int T;scanf("%d\n",&T);
  for(int d=1;d<=T;d++){printf("Dialogue #%d:\n",d);processDialogue();}
}
```

### Hints for the missing half — enough for a champion, no more

The missing part does two things: (1) records new statements into posMap/negMap without duplicates, (2) checks contradictions — a positive statement colliding with its negative, or with 'nobody ...'; symmetrically for negatives against 'everybody ...'.

Map keys look like 'subject|verb|object'. For everybody/nobody, match the '|verb|object' suffix across all keys.

For do/does questions: answer 'yes, ...' when a positive fact exists (direct or via everybody), 'no, ...' when a negative exists (direct or via nobody), otherwise 'maybe.'. Also now hidden: the entire question-answering branch — the "who" answer builder (collecting subjects, joining them with "and"/commas, matching singular/plural verb forms), the "what" answer builder (listing every action a subject is known to (not) do), and the "maybe." fallback. Rebuild these from the Phase 3 description above: match by verb+object key, prefer everybody/nobody blanket facts, and always append a trailing period.

**Verification log — this code was re-checked before printing**

The full code was recompiled with g++ -O2 -std=c++17 (0 errors) and smoke-tested. Archive status: ACCEPTED on SPOJ; this problem's official samples are not fully published, so the final verification is a fresh SPOJ submit — make that part of your drill.

## 1.6 Session 1 Drills

1. Write your personal template kit (fast I/O + 6 core structures) from scratch, timed. Target: under 25 minutes total.
2. Take last year's 8-problem OSN set: do ONLY the minutes-0-10 protocol (no coding), then compare your map with the official editorial.
3. Finish HOTLINE below: a problem won not by deep algorithms but by careful reading — exactly the skill the first line of OSN tests.

## 1.R Session Summary and Winning Points

- Map first, code later: 10 minutes of mapping saves 60 minutes of wandering.
- Points per minute is the compass; cheap subtasks get banked first.
- The 45-minute rule and the debug protocol guard you against the two medal killers: stubbornness and panic.
- Every weapon (templates, stress tests, index conventions) is prepared BEFORE the day.

## References

[1] Panduan resmi OSN bidang Informatika, Puspresnas/BPTI.

[2] IOI Syllabus (International Olympiad in Informatics), ioinformatics.org.

[3] Cormen, Leiserson, Rivest, Stein — Introduction to Algorithms, 4th ed., MIT Press, 2022.

[4] Halim, S., Halim, F. — Competitive Programming 4, 2020.

[5] SPOJ — Sphere Online Judge, spoj.com (problem HOTLINE).

## [6] SESSION 2 · LECTURE SESSION (2 HOURS)

# The Segment Tree: A Range-Answering Machine

Nearly every OSN carries at least one problem screaming 'range': the sum on [l,r], the max on [l,r], or — as in our case study — the best subarray inside [l,r]. The segment tree is the answer, and mastering it to the point of designing your own nodes separates silver from gold.

**Session goals — after these 2 hours you can:**

(1) Build a segment tree in O(N) and answer queries in O(log N)

(2) Design multi-field nodes and an associative merge operation

(3) Derive GSS1's 4-field node (sum, pre, suf, ans) from scratch

(4) Recognize when a segment tree loses to prefix sums / sparse tables (and vice versa)

## 2.1 The Core Idea: Range Answers from Half-Range Answers

A segment tree halves [1..N] recursively, forming a binary tree of depth log N. Every node stores the ANSWER for its range. There is one key: a node's answer must be computable ONLY from its two children's answers — an associative merge. If you can write combine(left, right), you can answer any range query in O(log N).

- Build: O(N) — every node computed once from its children.
- Query [l,r]: O(log N) — the range splits into ≤ 2 log N canonical nodes.
- Memory: a `tree[4N]` array — memorize that factor 4, the most common RE cause.
- Point update: O(log N) — recompute nodes along one root-to-leaf path.

**Why not just prefix sums?**

Prefix sums answer static range SUMS in O(1) — unbeatable. Segment trees win when there are UPDATEs, or when the operation is not simple addition (max, gcd, GSS1's composite nodes).

## 2.2 Designing the Node: The Real Art

HARD problems never ask for 'range sum'. They ask something strange — and your job is to find the node that makes the strangeness mergeable. Recipe: (1) write down what is asked; (2) try merging two neighbor answers — what information is MISSING?; (3) add that to the node; (4) repeat until closure. For maximum subarray: left ans and right ans are not enough because the answer may CROSS the boundary — you need the left's best suffix and the right's best prefix; and for pre/suf to merge you need sum. Four fields: perfect closure.

```
// Node GSS1 — hasil dari resep penutupan / GSS1 node — the closure recipe's
```

```
result
struct Node { long long sum, pre, suf, ans; };
// sum: jumlah seluruh rentang          / total of the range
// pre: subarray terbaik menempel kiri  / best subarray glued to the left edge
// suf: subarray terbaik menempel kanan / best subarray glued to the right edge
// ans: subarray terbaik bebas          / best subarray anywhere
```

**Champion's mindset**

At OSN, 'is this a segment tree?' is the wrong question. The champion's question: 'what information makes this range answer MERGEABLE?' — the data structure follows.

## 2.3 HARD Case Study: GSS1 (SPOJ #1043)

**Problem card**

SPOJ GSS1 | Data Structures | Hard | Max-Subarray Segment Tree

Source: https://www.spoj.com/problems/GSS1/ — solve and submit directly on SPOJ.

### Problem statement (concise, faithful to the original SPOJ text)

Given a sequence A[1..N] (N ≤ 50,000, |Ai| ≤ 15,007) and M queries (x, y). For each query print the maximum of A[i] + A[i+1] + ... + A[j] with x ≤ i ≤ j ≤ y — the largest-sum (non-empty) subarray lying entirely inside [x, y].

- Line 1: N. Line 2: N integers. Line 3: M, then M lines of x y pairs.
- All elements can be negative — the empty subarray is not allowed.

### Official sample input/output

```
Input:
3
-1 2 3
1
1 2
Output:
2
```

### Phases 1-2 — Understand and Model

Each segment-tree node stores four values for its range: the total (sum), best prefix (pre), best suffix (suf), and best subarray (ans). The four are sufficient — and necessary — for two adjacent ranges to merge without losing boundary-crossing answers.

### Phase 3 — Design the Algorithm

1. Build the tree from make_node(a[l]) leaves; internal nodes = combine(left, right).

2. combine: sums add; pre = max(left pre, left sum + right pre); suf symmetric; ans = max(left ans, right ans, left suf + right pre).
3. Queries return whole NODES (not numbers), so range fragments merge with the same combine.
4. All fields are long long and may go negative — no 'max with 0' anywhere.

## Phase 4 — Complexity Analysis

O(N) build, O(log N) per query: for N = 50,000 and M queries, well under a second total. Memory 4N nodes × 32 bytes ≈ 6 MB — safe.

## Phase 5 — Implementation (Public Edition — ~50% revealed, ~50% your task)

This is the Public Edition of this solution. Roughly half of the code is shown below; the rest — the part that actually solves the problem — is your task, guided by the hints beneath the code and the walkthrough in Phases 1-4 above.

```
#include <cstdio>
#include <algorithm>
using namespace std;
const int MX = 50001;
struct Node {
  long long sum, pre, suf, ans;
};
Node tree[4*MX];
int a[MX];
Node make_node(long long v) {
  Node n;
  n.sum = n.pre = n.suf = n.ans = v;
  return n;
}
/* ================================================================
   >>> PUBLIC EDITION — roughly half of this solution is hidden for you to complete. <<<
Rebuild it using the hints below the code and the reasoning in Phases 1-4 above.
   ================================================================ */
int main() {
  int n;
  scanf("%d", &n);
  for (int i = 1; i <= n; i++) scanf("%d", &a[i]);
  build(1, 1, n);
  int q;
  scanf("%d", &q);
  while (q--) {
    int l, r;
    scanf("%d %d", &l, &r);
    printf("%lld\n", query(1, 1, n, l, r).ans);
  }
  return 0;
}
```

### Hints for the missing half — enough for a champion, no more

The four combine formulas: sum = both sums added; pre = max(left pre, left sum + right pre); suf = max(right suf, right sum + left suf); ans = max(left ans, right ans, left suf + right pre).

A standard build(): leaves = make_node(a[l]); internal nodes = combine of both children after recursing.

Fastest mental test: array {-1, 2} full-range query must give 2, not 1. Also now hidden: combine() (the four-formula merge already described above), build() (the standard recursive build: leaf = make_node(a[l]), internal = combine of both children), and query() (the three-way split: fully covered / fully left / fully right / crossing, combining the crossing case's two recursive results).

**Verification log — this code was re-checked before printing**

The full code was recompiled with g++ -O2 -std=c++17 (0 errors), tested against the official SPOJ samples, and stress-tested against a brute-force reference on hundreds of random cases — all identical. Archive status: ACCEPTED on SPOJ.

## 2.4 Session 2 Drills

1. Re-derive all four GSS1 combine formulas on paper WITHOUT the book, then complete the hidden half of the code.
2. Modify the node to answer: the longest increasing run inside [l,r]. Which fields do you need?
3. Stress-test your GSS1 against an $O(N^2)$ brute force on 1,000 random cases.

## 2.R Session Summary and Winning Points

- A segment tree = mergeable range answers; an associative combine is the contract.
- Nodes are designed by the closure recipe: add fields until the merge lacks nothing.
- GSS1: four fields (sum, pre, suf, ans) close the maximum-subarray problem.
- tree[4N], O(log N) query, O(N) build — numbers that must stick.

## References

[1] Cormen, Leiserson, Rivest, Stein — Introduction to Algorithms, 4th ed., MIT Press, 2022.

[2] Halim, S., Halim, F. — Competitive Programming 4, 2020.

[3] SPOJ — Sphere Online Judge, spoj.com (problem GSS1).

## [4] SESSION 3 · LECTURE SESSION (2 HOURS)

# Persistent Segment Trees & Order Statistics

How do you answer 'what is the k-th smallest element in [l,r]?' for hundreds of thousands of queries — without re-sorting every time? The answer is one of competitive programming's most beautiful ideas: keeping EVERY historical version of a segment tree at only O(log N) extra cost per version.

**Session goals — after these 2 hours you can:**

(1) Explain persistence: path copying and why it costs only O(log N) per update

(2) Build a frequency tree over compressed values

(3) Answer range k-th smallest via the difference of two roots (roots[r] - roots[l-1])

(4) Apply coordinate compression with sort + unique + lower_bound

## 3.1 Persistence: Keeping History Without Copying the World

A segment tree update touches only one root-to-leaf path: log N nodes. Path copying exploits this: instead of overwriting, copy ONLY the nodes on that path, and let every other pointer share the old nodes with previous versions. The result: version i of the tree lives alongside all earlier versions, each adding only O(log N) new nodes.

- New nodes per update: O(log N); total memory: O(N log N) — that is what MAXN*18 in the code means.
- A version = a root. Storing `roots[i]` stores the entire state after inserting element i.
- Two versions can be SUBTRACTED field by field: counts in version r minus version l-1 = frequencies belonging to [l,r] alone.

**Why is subtracting two versions valid?**

Because insertions are additive: every node stores cnt = how many elements fall in its value sub-range. Prefix [1..r] minus prefix [1..l-1] leaves exactly the contribution of elements l..r — the prefix-sum principle, lifted to tree form.

## 3.2 Walking Down Two Trees at Once

The k-th smallest query walks down BOTH roots in lockstep. At each node compute left_cnt = cnt(left child of version r) − cnt(left child of version l−1): how many of the range's elements fall in the lower half of values. If k ≤ left_cnt the answer lives on the left; otherwise turn right with k

reduced by left_cnt. At the leaf: the k-th value. It is binary search, lifted onto a frequency structure.

- Coordinate compression is mandatory: values up to $10^9$ but only N distinct — sort, unique, lower_bound.
- Total complexity: O((N+M) log N) — far under the limits for N, M ≤ $10^5$.
- The 'difference of two versions' pattern also answers: how many elements ≤ x in [l,r], the rank of a value, and more.

## 3.3 HARD Case Study: MKTHNUM (SPOJ #3947)

**Problem card**

SPOJ MKTHNUM | Data Structures | Hard | Persistent Segment Tree

Source: https://www.spoj.com/problems/MKTHNUM/ — solve and submit directly on SPOJ.

### Problem statement (concise, faithful to the original SPOJ text)

Given an array A[1..N] (N ≤ 100,000) and M ≤ 5,000 queries (i, j, k): if the segment A[i..j] were sorted, print its k-th element. Values reach ±$10^9$, so coordinate compression is required.

- Line 1: N M. Line 2: N integers. Then M lines i j k (1 ≤ k ≤ j−i+1).

### Official sample input/output

```
Input:
7 3
1 5 2 6 3 7 4
2 5 3
4 4 1
1 7 3
Output:
5
6
3
```

### Phases 1-2 — Understand and Model

A frequency tree over compressed values, built incrementally: version i holds elements 1..i. Because insertions are additive, version r minus version l−1 yields the frequencies belonging purely to [l,r]; the k-th smallest just walks down both versions in lockstep.

### Phase 3 — Design the Algorithm

1. Compress values: sort + unique + lower_bound maps ±$10^9$ into 1..un.
2. The path-copying update: clone the on-path node (cnt+1), descend only the side holding the value's position; the other side is shared with the old version.
3. kth: left_cnt = cnt[lc[vr]] − cnt[lc[vl]]; k ≤ left_cnt → go left, else right with k − left_cnt; the leaf is the answer.

4. A MAXN·18 node pool holds N insertions × (log N + 1) new nodes.

## Phase 4 — Complexity Analysis

O((N+M) log N) time and O(N log N) memory: N = $10^5$ means ±1.8M nodes — a few MB, comfortably inside limits. Each query is just log N steps.

## Phase 5 — Implementation (Public Edition — ~50% revealed, ~50% your task)

This is the Public Edition of this solution. Roughly half of the code is shown below; the rest — the part that actually solves the problem — is your task, guided by the hints beneath the code and the walkthrough in Phases 1-4 above.

```
#include <cstdio>
#include <algorithm>
using namespace std;
const int MAXN = 100001;
const int MAXNODES = MAXN * 18; // each insert touches O(log n) nodes
int lc[MAXNODES], rc[MAXNODES], cnt[MAXNODES];
int node_cnt = 0;
int new_node() {
  ++node_cnt;
  lc[node_cnt] = rc[node_cnt] = cnt[node_cnt] = 0;
  return node_cnt;
}
/* ================================================================
   >>> YOUR TASK — 17 core lines HIDDEN (~25%). <<<
Complete this part yourself. Hints are below the code.
   ================================================================ */
}
int roots[MAXN];
int a[MAXN], vals[MAXN];
int main() {
/* ================================================================
   >>> PUBLIC EDITION — roughly half of this solution is hidden for you to
complete. <<<
Rebuild it using the hints below the code and the reasoning in Phases 1-4 above.
   ================================================================ */
  while (m--) {
    int l, r, k;
    scanf("%d %d %d", &l, &r, &k);
    int pos = kth(roots[l-1], roots[r], 1, un, k);
    printf("%d\n", vals[pos]);
  }
  return 0;
}
```

### Hints for the missing half — enough for a champion, no more

update(prev, ...) copies the old node (lc, rc, cnt+1) then descends only ONE side by value position; the other side keeps pointing at the old version — that is path copying.

kth walks two versions in lockstep: left_cnt = cnt[lc[vr]] − cnt[lc[vl]]; if k ≤ left_cnt go left, else go right with k − left_cnt.

Base: lo == hi is a leaf — return lo (the compressed value index). Also now hidden: the persistent-tree build loop in main() — compressing values with sort+unique, then calling update(roots[i-1], 1, un, pos) once per element to grow the version chain. update() itself (path copying: clone the node, recurse into only the side containing the target position) and kth() (walking two tree versions in lockstep by

comparing left-subtree counts) were already hidden in the original edition and stay hidden here.

**Verification log — this code was re-checked before printing**

The full code was recompiled with g++ -O2 -std=c++17 (0 errors), tested against the official SPOJ samples, and stress-tested against a brute-force reference on hundreds of random cases — all identical. Archive status: ACCEPTED on SPOJ.

## 3.4 Session 3 Drills

1. Hand-draw: the frequency tree over [1..4] after inserting 3, 1, 4, 1 — all four versions, shared nodes in matching colors.
2. Complete MKTHNUM's hidden half (the path-copying update and the kth descent).
3. Extend: answer 'how many elements ≤ x in [l,r]' with the same tree, no new structure.

## 3.R Session Summary and Winning Points

- Persistence = path copying: O(log N) new nodes per version, all versions alive forever.
- roots[r] − roots[l−1] isolates the range's frequencies — prefix sums in tree form.
- K-th smallest = binary search descending two versions in lockstep.
- Coordinate compression is a reflex, not a choice.

## References

[1] Cormen, Leiserson, Rivest, Stein — Introduction to Algorithms, 4th ed., MIT Press, 2022.

[2] Halim, S., Halim, F. — Competitive Programming 4, 2020.

[3] SPOJ — Sphere Online Judge, spoj.com (problem MKTHNUM).

## [4] SESSION 4 · LECTURE SESSION (2 HOURS)

# Monotonic Stacks & Linear Array Techniques

Some N ≤ 100,000 problems look like they demand heavy machinery, when the answer is one O(N) pass with a stack kept in sorted order. The largest rectangle in a histogram — a world classic and an OSN jury favorite — is the perfect specimen: every bar enters the stack once, leaves once, and the answer is born exactly when a bar gets 'closed off'.

**Session goals — after these 2 hours you can:**

(1) Maintain a monotonic stack invariant and read it geometrically

(2) Compute nearest-smaller-elements on both sides in one pass

(3) Solve HISTOGRA in O(N) and prove the amortization

(4) Recognize its relatives: trapping rain water, stock spans, minimum-sum subarrays

## 4.1 An Invariant that Works for You

A monotonic stack stores indices with increasing (or decreasing) values. When a new element breaks the order, the top is popped — and precisely at that pop we know EVERYTHING about it: its nearest smaller neighbor on the right is the new element, and on the left it is the element beneath it in the stack. Two seemingly global facts, obtained locally.

- Amortization: every index is pushed once and popped at most once → O(N) total, despite the nested loop.
- Sentinels simplify: a height-0 bar at the end forces the whole stack to be processed with zero special cases.
- The width at pop time: from (the element beneath the top) + 1 to (current position) − 1.

**How to spot it in contest**

Keywords: 'nearest smaller/greater', 'while still ≥', 'maximal rectangle', 'how long it survives'. When each element's answer is fixed by the nearest neighbor breaking a condition — that is a monotonic stack.

## 4.2 From Histogram to Matrix: Leveling Up

The 'largest all-1 submatrix in a binary matrix' — another jury favorite — is HISTOGRA stacked: row by row, height[i] = how many consecutive 1s sit above this cell, then run HISTOGRA per row. O(R·C). Master the base form and the 2-D variant comes free.

```
// Pola nearest-smaller kiri dalam satu lintasan / left nearest-smaller in one
```

```
pass
stack<int> st;                // indeks, nilai menaik / indices, increasing values
for (int i = 0; i < n; i++) {
  while (!st.empty() && h[st.top()] >= h[i]) st.pop();
  L[i] = st.empty() ? -1 : st.top();   // tetangga kiri < h[i] / left neighbor <
h[i]
  st.push(i);
}
```

## 4.3 HARD Case Study: HISTOGRA (SPOJ #1805)

**Problem card**

SPOJ HISTOGRA | Searching & Sorting | Hard | Monotonic Stack

Source: https://www.spoj.com/problems/HISTOGRA/ — solve and submit directly on SPOJ.

### Problem statement (concise, faithful to the original SPOJ text)

A histogram consists of n ≤ 100,000 unit-width bars of heights hi ≤ $10^9$. Find the area of the largest rectangle fully contained in the histogram (axis-aligned, base on the baseline). The input holds many test lines; a line starting with 0 ends it.

- Each line: n then n heights. Answers can exceed 32-bit — use 64-bit.

### Official sample input/output

```
Input:
7 2 1 4 5 1 3 3
4 1000 1000 1000 1000
0
Output:
8
4000
```

### Phases 1-2 — Understand and Model

The optimal rectangle always tops out at some bar: for each bar, its answer is that height times the maximal width it can span without a shorter bar cutting it. A monotonic stack finds both bounds for every bar in one pass.

### Phase 3 — Design the Algorithm

1. Keep a stack of indices with increasing heights; a shorter new bar 'closes' the top bars.
2. On a pop: width = i − s.top() − 1 (or i when the stack empties); candidate = height × width.
3. A height-0 sentinel at position n forces every remaining bar to process with no special cases.
4. Accumulate in long long: $10^5 \times 10^9$ exceeds int.

### Phase 4 — Complexity Analysis

Every index pushes once and pops at most once — amortized O(N) per test line, effectively instant at the limits.

### Phase 5 — Implementation (Public Edition — ~50% revealed, ~50% your task)

This is the Public Edition of this solution. Roughly half of the code is shown below; the rest — the part that actually solves the problem — is your task, guided by the hints beneath the code and the walkthrough in Phases 1-4 above.

```
#include <cstdio>
#include <stack>
#include <algorithm>
using namespace std;
long long h[100001];
int main(){
  int n;
  while(scanf("%d",&n),n){
/* ================================================================
   >>> PUBLIC EDITION — roughly half of this solution is hidden for you to complete. <<<
Rebuild it using the hints below the code and the reasoning in Phases 1-4 above.
   ================================================================ */
    printf("%lld\n",ans);
  }
}
```

**Hints for the missing half — enough for a champion, no more**

Stack invariant: indices with increasing heights. When the current bar is shorter than the stack top, the top bar is 'closed off': its height times its width is an answer candidate.

The popped bar's width: if the stack empties, all of [0..i); otherwise i − s.top() − 1.

A sentinel bar of height 0 at position n forces every remaining bar to pop — that is why the loop runs to i == n. Also now hidden: reading the n heights into h[], and the sentinel setup (stack<int> s; long long ans=0). The stack-and-sentinel loop that was already hidden is the whole algorithm — rebuild it from the hints above: push increasing heights, pop-and-price when a shorter bar arrives, and let the height-0 sentinel at i==n flush the stack at the end.

**Verification log — this code was re-checked before printing**

The full code was recompiled with g++ -O2 -std=c++17 (0 errors), tested against the official SPOJ samples, and stress-tested against a brute-force reference on hundreds of random cases — all identical. Archive status: ACCEPTED on SPOJ.

## 4.4 Session 4 Drills

1. Complete HISTOGRA's hidden half (the pop loop) then stress-test against an $O(N^2)$ brute force.
2. Solve the R×C ≤ 2,000 binary-matrix variant by stacking histograms per row.

3. Prove on paper: total pops across the whole run ≤ N.

## 4.R Session Summary and Winning Points

- The pop is the moment of knowledge: when an element leaves, both nearest bounds are known.
- One push + one pop = amortized O(N), whatever the loops look like.
- A 0 sentinel deletes every edge case.
- Row-wise HISTOGRA solves the 2-D maximal submatrix.

## References

[1] Cormen, Leiserson, Rivest, Stein — Introduction to Algorithms, 4th ed., MIT Press, 2022.

[2] Halim, S., Halim, F. — Competitive Programming 4, 2020.

[3] SPOJ — Sphere Online Judge, spoj.com (problem HISTOGRA).

## [4] SESSION 5 · LAB SESSION (2 HOURS)

# Lab 1 — Living Data Structures: Dynamic Updates

Two full hours of practice. The theory of Sessions 2-4 is tested on GSS3 — GSS1 with a mutating array. The lab mimics the contest hall: timed, judged submissions, and every team must run the stress-test protocol before declaring a solution correct (Table 5.1).

**Session goals — after these 2 hours you can:**

(1) Add point updates to the GSS1 segment tree without breaking the combine invariant

(2) Run the full cycle: implement → samples → stress-test → submit

(3) Measure and log your time per stage as training data

## 5.1 The 120-Minute Plan

| Minutes | Activity | Target |
|---|---|---|
| 0-10 | Read GSS3, write the model & node plan | A paper map of the problem |
| 10-45 | Full implementation (complete the hidden half) | Clean compile + official sample passes |
| 45-70 | Stress-test vs an $O(N^2)$ brute + fixes | 1,000 random cases identical |
| 70-85 | Submit to SPOJ; chase ACCEPTED | An AC verdict on record |
| 85-120 | Variant: range-assign updates? discuss lazy propagation | A map toward the further GSS series |

*Table 5.1 — Every lab imitates the rhythm of a real contest.*

**Lab rules**

No opening full solutions before minute 70. Hints only from assistants, and only three per team — exactly the contest's clarification ration.

## 5.2 Where GSS3 Bites

- The update must re-combine EVERY node on the way back — one miss = a silent WA that only surfaces on large cases.
- The query must never read a half-covered node without combining both sides.

- Replacement values can be negative; make_node must allow negative pre/suf/ans — no 'max with 0'.

## 5.3 HARD Case Study: GSS3 (SPOJ #1716)

**Problem card**

SPOJ GSS3 | Data Structures | Hard | Max-Subarray Segment Tree + Point Update

Source: https://www.spoj.com/problems/GSS3/ — solve and submit directly on SPOJ.

### Problem statement (concise, faithful to the original SPOJ text)

Like GSS1 — the maximum-sum subarray on a range — but now the sequence CHANGES: operation '0 x y' sets A[x] to y, and operation '1 x y' asks for the maximum subarray inside [x, y]. N, M ≤ 50,000.

- GSS1's 4-field node still applies; the only new piece is the root-to-leaf update path.

### Official sample input/output

```
Input:
4
1 2 3 4
4
1 1 3
0 3 -3
1 2 4
1 3 3
Output:
6
4
-3
```

### Phases 1-2 — Understand and Model

GSS1's four-field node is unchanged; the only new operation replaces one position's value. Since one leaf changes, recomputing the nodes along that single root-to-leaf path suffices — the same combine re-stitches the whole tree.

### Phase 3 — Design the Algorithm

1. update(pos, val): descend into the side containing pos, replace the leaf with make_node(val), re-combine on the way back.
2. The three-case query is exactly GSS1's; partial results merge via combine, never read raw.
3. Operation 0 = update, operation 1 = query — read the op codes carefully.

## Phase 4 — Complexity Analysis

O(log N) per operation; 50,000 ops × log 50,000 ≈ $10^6$ steps — a few milliseconds. This problem tests correctness, not speed.

## Phase 5 — Implementation (Public Edition — ~50% revealed, ~50% your task)

This is the Public Edition of this solution. Roughly half of the code is shown below; the rest — the part that actually solves the problem — is your task, guided by the hints beneath the code and the walkthrough in Phases 1-4 above.

```
#include <cstdio>
#include <algorithm>
using namespace std;
const int MAXN = 50001;
struct Node {
  int sum, pre, suf, ans;
};
Node tree[4*MAXN];
int a[MAXN], N;
Node make_node(int v) {
  Node n;
  n.sum = n.pre = n.suf = n.ans = v;
  return n;
}
/* ================================================================
   >>> PUBLIC EDITION — roughly half of this solution is hidden for you to complete. <<<
Rebuild it using the hints below the code and the reasoning in Phases 1-4 above.
   ================================================================ */
int main() {
  scanf("%d", &N);
  for (int i = 1; i <= N; i++) scanf("%d", &a[i]);
  build(1, 1, N);
  int M;
  scanf("%d", &M);
  while (M--) {
    int op, x, y;
    scanf("%d %d %d", &op, &x, &y);
    if (op == 0) update(1, 1, N, x, y);
    else printf("%d\n", query(1, 1, N, x, y).ans);
  }
  return 0;
}
```

### Hints for the missing half — enough for a champion, no more

update: descend into the side containing pos (compare with m), replace the leaf with make_node(val), then re-combine on the way back.

query: three cases — full cover (return the node), fully left, fully right; the crossing case combines two recursive results.

Never answer from a partial node without combining — that is where most GSS3 WAs are born. Also now hidden: combine() itself (the four-formula merge — write it out from the GSS1-style logic: sum, pre, suf, ans) and build() (leaf = make_node(a[l]), internal = combine of the two children). update() and query() were already hidden in the original edition and stay hidden here — rebuild both using the hints above.

**Verification log — this code was re-checked before printing**

The full code was recompiled with g++ -O2 -std=c++17 (0 errors), tested against the official SPOJ samples, and stress-tested against a brute-force reference on hundreds of random cases — all identical. Archive status: ACCEPTED on SPOJ.

## 5.4 Lab Completion Checklist

- [ ] Hidden half completed without the full solution
- [ ] Official sample + 1,000 stress tests pass
- [ ] ACCEPTED on SPOJ (attach the verdict screenshot)
- [ ] Per-stage times logged in the training journal

## 5.R Session Summary and Winning Points

- An update = one root-to-leaf path + a re-combine at every step back.
- Stress-testing is not optional: it is part of the definition of 'done'.
- Today's timing data = your time-allocation strategy on contest day.

## References

[1] Cormen, Leiserson, Rivest, Stein — Introduction to Algorithms, 4th ed., MIT Press, 2022.

[2] Halim, S., Halim, F. — Competitive Programming 4, 2020.

[3] SPOJ — Sphere Online Judge, spoj.com (problem GSS3).

## [4] SESSION 6 · LECTURE SESSION (2 HOURS)

# Graphs I: State-Space BFS & Small-Weight Shortest Paths

Many OSN shortest-path problems never say the word 'graph' — grids, puzzles, machine configurations. The first skill is READING the hidden graph; the second is picking the right weapon: BFS for weight 1, 0-1 BFS for weights {0,1}, Dial for small bounded weights, and only then heap Dijkstra for the general case (Table 6.1).

**Session goals — after these 2 hours you can:**

(1) Model grids/puzzles as state-space graphs

(2) Choose among BFS, 0-1 BFS, Dial, and Dijkstra from the weight structure

(3) Implement Dial with a circular bucket array of size max-weight + 1

(4) Avoid memory traps & stale entries on multi-million-node graphs

## 6.1 The Shortest-Path Weapon Ladder

| Weight structure | Weapon | Complexity |
|---|---|---|
| All 1 | BFS | O(V+E) |
| Only {0,1} | 0-1 BFS (deque: 0 front, 1 back) | O(V+E) |
| Small integers ≤ C | Dial (C+1 circular buckets) | O(E + V·C) |
| General non-negative | Dijkstra + heap | O(E log V) |
| Negatives present | Bellman-Ford / SPFA | O(V·E) |

*Table 6.1 — Weights choose the weapon; don't bring a heap to a bucket fight.*

**Why Dial wins on CHIPSLL**

A 2000×2000 grid = 4M nodes, 16M edges, weights 1..63. A heap: 16M log-operations — heavy and memory-hungry. Dial: distances grow monotonically, only 64 circular bucket slots — practically linear, cache-friendly, and memory-fit.

## 6.2 State Space: The Invisible Graph

Nodes are not always 'places'. Nodes are STATES: (position, keys held), (knight square, turn), (water left in two jugs). Edges are legal actions. Once states and actions are defined, all shortest-

path theory applies immediately. The best drill: before coding, write explicitly — what is the state? how many are there? what are the transitions? what do they cost?

- CHIPSLL: state = a cell; entering costs 1 + its occupant's power — edge weights 1..63.
- The grid comes from a seed: read the generator spec AS CAREFULLY as the statement — one wrong constant digit = total WA.
- Store the grid as a flat 1-D array (id = row·WIDTH + col): faster and leaner than 2-D structures.

## 6.3 HARD Case Study: CHIPSLL (SPOJ #40642)

**Problem card**

SPOJ CHIPSLL | Classical | Hard | Shortest Path (Dial's Algorithm)

Source: https://www.spoj.com/problems/CHIPSLL/ — solve and submit directly on SPOJ.

### Problem statement (concise, faithful to the original SPOJ text)

Chip must reach Melinda on an N×M grid (up to 2000×2000, T ≤ 10 cases). Stepping into an empty cell costs 1 second; occupied cells (water a-z, monsters A-Z, doors 0-9) must be destroyed at an extra cost equal to their power (1-62). The grid is NOT given: it is generated from a seed by a problem-defined pseudorandom generator. Find the minimum total time.

- Small bounded edge weights (1..63) on a 4-million-node graph — the signal for Dial, not heap Dijkstra.

### Phases 1-2 — Understand and Model

The grid is a small-weight graph: entering a cell costs 1 + its occupant's power (1..63). The grid is generated from the seed exactly per the statement's recipe, then shortest distances come from bucket Dijkstra (Dial) — circular buckets numbering max weight + 1.

### Phase 3 — Design the Algorithm

1. Generate pwv[cell] = 1 + power from the seed generator; start/target cells cost 1.
2. Buckets: bkt[d mod 64]; advance circularly to the next non-empty bucket; stale entries (dist ≠ cur_d) drop on pop.
3. Four-way relaxation: nd = cur_d + pwv[neighbor]; if below the old dist, record and push into bkt[nd mod 64].
4. Flat 1-D arrays + unsigned char costs: 4 million cells fit cache and memory.

### Phase 4 — Complexity Analysis

O(E + V·W) with W = 64: practically linear over 4M nodes and 16M relaxations — far lighter than a heap's O(E log V), and that is the difference between TLE and AC here.

## Phase 5 — Implementation (Public Edition — ~50% revealed, ~50% your task)

This is the Public Edition of this solution. Roughly half of the code is shown below; the rest — the part that actually solves the problem — is your task, guided by the hints beneath the code and the walkthrough in Phases 1-4 above.

```
#include <cstdio>
#include <cstring>
#include <vector>
using namespace std;
static const int MAXN = 2021;
static const int MAXCELL = MAXN * MAXN;
static const int W = 64;
static const int INF = 0x3f3f3f3f;
static unsigned char pwv[MAXCELL]; // edge cost to enter cell (1 + power)
static int dist_flat[MAXCELL];
static long long seed_g;
static void update_seed(){
  seed_g = (seed_g * 1000003LL + 10007LL) % 1000000009LL;
}
static void gen_pwv(int N, int M, int Cx, int Cy, int Mx, int My){
  for(int i=1;i<=N;++i)
  for(int j=1;j<=M;++j){
    int id=i*MAXN+j;
    if((i==Cx&&j==Cy)||(i==Mx&&j==My)){pwv[id]=1;continue;}
    update_seed();
    int p=seed_g%63;
    // power(cell) == p exactly (water p<=26, monster p<=52, door p<=62)
    pwv[id]=(unsigned char)(1+p);
  }
}
static vector<int> bkt[W];
static int solve(int N, int M, int sr, int sc, int tr, int tc){
/* ================================================================
   >>> PUBLIC EDITION — roughly half of this solution is hidden for you to
complete. <<<
Rebuild it using the hints below the code and the reasoning in Phases 1-4 above.
   ================================================================ */
int main(){
  int T; scanf("%d",&T);
  for(int t=1;t<=T;t++){
    int N,M,Cx,Cy,Mx,My;
    scanf("%d%d%d%d%d%d",&N,&M,&Cx,&Cy,&Mx,&My);
    scanf("%lld",&seed_g);
    gen_pwv(N,M,Cx,Cy,Mx,My);
    printf("Case %d: %d\n",t,solve(N,M,Cx,Cy,Mx,My));
  }
}
```

### Hints for the missing half — enough for a champion, no more

Dial = Dijkstra with buckets: bucket[d mod W] holds nodes at distance d; W need only exceed the maximum weight (here 64).

Main loop: find the next non-empty bucket (advance circularly), pop its nodes; discard stale entries (dist ≠ cur_d), stop when the target pops.

The 4-way relaxation uses the relax(ni) lambda: nd = cur_d + the cell's entry cost; if smaller, record it and push into bucket[nd mod W]. Now the entire solve() body is hidden, including the setup that was previously shown (distance-array reset, bucket clearing, seeding the start node at distance 0). Rebuild the Dial's-algorithm loop from the hints above: advance through non-empty buckets in order, discard

stale pops, relax all 4 neighbors into bucket[new_dist mod W], and stop once the target is popped.

**Verification log — this code was re-checked before printing**

The full code was recompiled with g++ -O2 -std=c++17 (0 errors) and smoke-tested. Archive status: ACCEPTED on SPOJ; this problem's official samples are not fully published, so the final verification is a fresh SPOJ submit — make that part of your drill.

## 6.4 Session 6 Drills

1. Complete CHIPSLL's hidden Dial bucket loop (half), then test on a small hand-generated grid.
2. Write a 0-1 BFS for the variant: destroying any cell costs 1, stepping costs 0.
3. Model as state space then solve: a chess knight from a1 to h8 with k forbidden squares.

## 6.R Session Summary and Winning Points

- Read the weight structure before choosing an algorithm — the weapon table in your head.
- Dial: circular buckets of max-weight + 1; discard stale entries on pop.
- State spaces turn puzzles into graphs; define (state, action, cost) on paper first.
- Giant grids: flat arrays + small types (unsigned char) = safe from MLE.

## References

[1] Cormen, Leiserson, Rivest, Stein — Introduction to Algorithms, 4th ed., MIT Press, 2022.

[2] Halim, S., Halim, F. — Competitive Programming 4, 2020.

[3] SPOJ — Sphere Online Judge, spoj.com (problem CHIPSLL).

## [4] SESSION 7 · LECTURE SESSION (2 HOURS)

# Graphs II: DSU, MST & Sweep Techniques

Kruskal teaches MST as 'sort all the edges'. But HARD problems often have too many edges to materialize — VILUNIT's $n^2$ village pairs, say. Champions invert the thinking: since weights are only {0,1,2}, process the cheapest weight first using geometry (sweeps) to find only the edges that matter, and let DSU summarize the rest.

**Session goals — after these 2 hours you can:**

(1) Use DSU with path compression as a component engine

(2) Run class-by-weight Kruskal without materializing all edges

(3) Find the relevant edges via sorted sweeps on X/Y projections

(4) Close the remaining components with the 2·(components − 1) formula

## 7.1 DSU: The 5-Line Structure that Wins Medals

```
int find(int x){ return par[x]==x ? x : par[x]=find(par[x]); }
bool unite(int a,int b){ a=find(a); b=find(b);
  if(a!=b){ par[a]=b; return true; } return false; }
```

- Path compression alone is nearly amortized O(α(n)) in contest practice — union by rank is optional.
- A bool-returning unite is Kruskal's key: true = the edge is used (the tree grows), false = redundant.
- Count final components: the number of i with par[i] == i after finds settle.

## 7.2 Kruskal Without an Edge List: Weight Classes + Sweeps

When weights take few values (in VILUNIT: 0, 1, 2), Kruskal simplifies into three waves: unite all weight-0 pairs, then weight-1, then close with 2. Waves 0 and 1 need no $n^2$ edge list: just SORT the objects and sweep — the relevant pairs are always adjacent in the order. Overlapping intervals form chains; uniting each object with its chain's representative yields the same components as uniting every pair, at O(n log n).

**Why is uniting with a 'representative' enough?**

The 'projections intersect' relation on intervals spreads through chains: if A reaches B and B reaches C in sweep order, component {A,B,C} still forms even when A and C do not directly intersect — DSU accumulates the chain with n−1 unites, not $n^2$ checks.

- The weight-1 sweep: maintain cur_max (the farthest right end) and its representative; an object starting before cur_max → unite (+1 cost).
- Repeat the X and Y sweeps until nothing changes: one unite can enable another.
- The closer: any free pair costs 2 → total + 2·(components − 1).

## 7.3 HARD Case Study: VILUNIT (SPOJ #42526)

**Problem card**

SPOJ VILUNIT | Classical | Hard | MST with Fixed Edge Weights via Sweep

Source: https://www.spoj.com/problems/VILUNIT/ — solve and submit directly on SPOJ.

### Problem statement (concise, faithful to the original SPOJ text)

There are n ≤ 5,000 rectangular villages. Uniting two costs 0 when their interiors overlap, 1 when their projections intersect on one axis, and 2 for any other pair. Find the minimum cost to unite all villages — an MST whose weights are only {0,1,2}, solvable without materializing all edges.

- Many test cases; large input → the fast reader (fread) is already included in the code.

### Phases 1-2 — Understand and Model

An MST whose weights are only {0,1,2}: process by weight class. Overlapping interiors unite free; X/Y projection reachability unites at cost 1 via sorted sweeps; leftover components close at cost 2 per link.

### Phase 3 — Design the Algorithm

1. Pass 0: sort by X1; for each u scan v while X1[v] < X2[u]; Y-projection intersection → free unite.
2. The weight-1 sweeps (X then Y): keep the farthest right end cur_max and its representative; objects starting before cur_max → unite (+1).
3. Repeat both sweeps until nothing changes — one unite can enable the next.
4. The closer: answer = cost + 2·(remaining components − 1).

### Phase 4 — Complexity Analysis

Sorting dominates: O(n log n) per case with tiny constants; the 'changed' loop runs only a few times in practice. The fread reader keeps the giant input cheap.

### Phase 5 — Implementation (Public Edition — ~50% revealed, ~50% your task)

This is the Public Edition of this solution. Roughly half of the code is shown below; the rest — the part that actually solves the problem — is your task, guided by the hints beneath the code and the walkthrough in Phases 1-4 above.

```cpp
#include <cstdio>
#include <algorithm>
#include <vector>
using namespace std;
#define MAXN 5005
#define IN_BUF (1 << 20)
static int X1[MAXN], Y1[MAXN], X2[MAXN], Y2[MAXN], par[MAXN], idxX[MAXN],
idxY[MAXN];
static char ibuf[IN_BUF];
static int ipos, ilen;
static inline int readInt() {
  int x = 0, f = 1;
  while (ipos < ilen && (ibuf[ipos] < '0' || ibuf[ipos] > '9')) {
    if (ibuf[ipos] == '-') f = -1;
    if (++ipos == ilen) {
      ilen = fread(ibuf, 1, IN_BUF, stdin);
      ipos = 0;
      if (ilen <= 0) return 0;
    }
  }
  while (ipos < ilen && ibuf[ipos] >= '0' && ibuf[ipos] <= '9') {
    x = x * 10 + (ibuf[ipos++] - '0');
    if (ipos == ilen) {
      ilen = fread(ibuf, 1, IN_BUF, stdin);
      ipos = 0;
    }
  }
  return x * f;
}
int find(int x) {
  return par[x] == x ? x : par[x] = find(par[x]);
}
bool unite(int a, int b) {
  a = find(a);
  b = find(b);
  if (a != b) {
    par[a] = b;
    return true;
  }
  return false;
}
int main() {
  ilen = fread(ibuf, 1, IN_BUF, stdin);
  int T = readInt();
  while (T--) {
    int n = readInt();
    if (n <= 1) {
      if (n == 1) {
        readInt();
        readInt();
        readInt();
        readInt();
      }
      printf("0\n");
      continue;
    }
    for (int i = 0; i < n; i++) {
      X1[i] = readInt();
      Y1[i] = readInt();
      X2[i] = readInt();
      Y2[i] = readInt();
      if (X1[i] > X2[i]) swap(X1[i], X2[i]);
      if (Y1[i] > Y2[i]) swap(Y1[i], Y2[i]);
      par[i] = i;
      idxX[i] = i;
      idxY[i] = i;
    }
    sort(idxX, idxX + n, [](int a, int b) {
```

```
      return X1[a] < X1[b];
    });
    sort(idxY, idxY + n, [](int a, int b) {
      return Y1[a] < Y1[b];
    });
/* ================================================================
   >>> PUBLIC EDITION — roughly half of this solution is hidden for you to
complete. <<<
Rebuild it using the hints below the code and the reasoning in Phases 1-4 above.
   ================================================================ */
    int comps = 0;
    for (int i = 0; i < n; i++) if (par[i] == i) comps++;
    printf("%lld\n", cost + 2LL * (comps - 1));
  }
  return 0;
}
```

**Hints for the missing half — enough for a champion, no more**

Pass 0: sort by X1; for each u scan v while X1[v] < X2[u]; if the Y projections also intersect, unite(u, v) — a weight-0 edge.

The weight-1 X-sweep EXACTLY mirrors the Y-sweep left visible below it: keep cur_max and its representative; when X1[u] < cur_max, unite and add cost 1.

Remaining components are joined by weight-2 edges: the final answer is cost + 2·(components − 1). Also now hidden: the entire X-sweep pass (already hidden in the original edition — sort by X1, scan while X1[v] < X2[u], unite on Y-overlap for weight-0 edges) AND the Y-sweep pass shown below it (mirror the X-sweep exactly, but on the Y projections, adding weight-1 edges). Both passes share the same cur_max/rep sweep pattern described in the hints above.

**Verification log — this code was re-checked before printing**

The full code was recompiled with g++ -O2 -std=c++17 (0 errors) and smoke-tested. Archive status: ACCEPTED on SPOJ; this problem's official samples are not fully published, so the final verification is a fresh SPOJ submit — make that part of your drill.

## 7.4 Session 7 Drills

1. Complete VILUNIT's half: the overlap Pass-0 and the X-sweep — a mirror of the visible Y-sweep.
2. Prove or refute: one round of X-sweep + Y-sweep always suffices (no 'changed' loop). Hunt a counterexample.
3. Do classic Kruskal on $10^5$ edges as a speed warm-up: target under 15 minutes from scratch to local AC.

## 7.R Session Summary and Winning Points

- The 5-line DSU + bool unite = an all-purpose component engine.
- Few weight values → wave-by-wave Kruskal, cheapest first.

- Sorted sweeps find the relevant edges without $n^2$ candidates.
- Leftover components close with arithmetic, not algorithms.

## References

[1] Cormen, Leiserson, Rivest, Stein — Introduction to Algorithms, 4th ed., MIT Press, 2022.

[2] Halim, S., Halim, F. — Competitive Programming 4, 2020.

[3] SPOJ — Sphere Online Judge, spoj.com (problem VILUNIT).

## [4] SESSION 8 · LAB SESSION (2 HOURS)

# Lab 2 — Living Trees: Euler BIT + Binary Lifting

An advanced graph lab: HBD fuses three techniques you already know separately — Euler tours, BITs, binary lifting — into one dynamic path-query solution. COMPOSING techniques is the mark of national readiness; these two hours train exactly that.

**Session goals — after these 2 hours you can:**

(1) Maintain root→node path sums with a BIT over Euler times

(2) Answer 'how many steps until the accumulation reaches x' with O(log² n) binary lifting

(3) Compose three techniques into one whole solution under time pressure

## 8.1 HBD's Three Gears

- Gear 1 — Euler tour: in[u]/out[u] turn subtrees into intervals; 'add y to node u' = add(in[u], +y) and add(out[u]+1, −y); then query(in[w]) = the value sum on the root→w path. A mandatory classic.
- Gear 2 — S(w): the v→u path sum = S(v) − S(parent(u)); every path question becomes a difference of two prefixes.
- Gear 3 — binary lifting: up[u][j] = the 2^j ancestor; jumping large-to-small finds the boundary node in log n hops, each checked by one BIT query — log² n total.

**The composition mindset**

Composite problems are not solved in one gulp. Split them into contracts: 'if I had a cheap S(w), what does the rest become?' — then fulfill that contract with a memorized technique.

## 8.2 The 120-Minute Plan

| Minutes | Activity |
|---|---|
| 0-15 | Read HBD; write the three-gear contracts on paper |
| 15-55 | Implement + complete the half (the type-1 answer block) |
| 55-80 | Test tiny hand-built trees (chain, star, balanced) |
| 80-95 | Submit; chase the AC |

| Minutes | Activity |
|---|---|
| 95-120 | Variant talk: values on EDGES? non-ancestor v→u paths (via LCA)? |

## 8.3 HARD Case Study: HBD (SPOJ #37921)

**Problem card**

SPOJ HBD | Classical | Hard | Tree + BIT + Binary Lifting

Source: https://www.spoj.com/problems/HBD/ — solve and submit directly on SPOJ.

### Problem statement (concise, faithful to the original SPOJ text)

A rooted tree with n ≤ $10^5$ nodes; each node holds a value. Two operations: (2) add y to node u's value; (1) given an ancestor u of v and a threshold x — walking upward from v toward u, how many nodes must be visited before the accumulated value reaches x (or -1 if even the whole path falls short)? q ≤ $10^5$ operations.

- Root-to-node path sums are maintained by a BIT over Euler entry times (add at in, subtract after out).

### Phases 1-2 — Understand and Model

Root→node path sums live in a BIT over Euler times (add at in, subtract after out); the v→u path sum is a difference of two prefixes. 'How many steps until the accumulation reaches x' is answered by binary lifting: jumping from large powers of two down, hunting the boundary node.

### Phase 3 — Design the Algorithm

1. One DFS: in/out, depth, and the ancestor table up[u][j] (j < 20).
2. Updating node u: add(in[u], +y), add(out[u]+1, −y) — S(w) = query(in[w]).
3. The query: if S(v) − S(parent(u)) < x → -1; otherwise refine curr from v: jump to p while depth(p) ≥ depth(u) and S(p) > S(v) − x.
4. The answer = depth(v) − depth(curr) + 1.

### Phase 4 — Complexity Analysis

Each query is O(log n) jumps × O(log n) BIT = O($\log^2$ n); $10^5$ queries ≈ 3×$10^7$ light operations. The manual getchar_unlocked I/O settles the rest of the time budget.

## Phase 5 — Implementation (Public Edition — ~50% revealed, ~50% your task)

This is the Public Edition of this solution. Roughly half of the code is shown below; the rest — the part that actually solves the problem — is your task, guided by the hints beneath the code and the walkthrough in Phases 1-4 above.

```
// #pragma GCC optimize("O3,unroll-loops")
// #pragma GCC target("tune=native")
// #pragma GCC target("avx,popcnt,bmi,bmi2") // As suggested by SPOJ, avoiding
AVX2
// Removed all #pragma GCC target(...) to prevent SIGILL on SPOJ's CPU
#include <iostream>
#include <vector>
using namespace std;
const int MAXN = 100005;
vector<int> adj[MAXN];
int up[MAXN][20];
int depth[MAXN];
int in[MAXN], out[MAXN];
int timer_dfs = 0;
long long bit[MAXN];
int n, q;
// Ultra-fast manual I/O for SPOJ supremacy
inline long long read_ll() {
  long long x = 0; int f = 1; char ch = getchar_unlocked();
  while (ch < '0' || ch > '9') { if (ch == '-') f = -1; ch =
getchar_unlocked(); }
  while (ch >= '0' && ch <= '9') { x = x * 10 + ch - '0'; ch =
getchar_unlocked(); }
  return x * f;
}
/* ================================================================
   >>> PUBLIC EDITION — roughly half of this solution is hidden for you to
complete. <<<
Rebuild it using the hints below the code and the reasoning in Phases 1-4 above.
   ================================================================ */
int main() {
  ios_base::sync_with_stdio(false);
  cin.tie(NULL);
  n = read_ll();
  q = read_ll();
  for (int i = 0; i < n - 1; i++) {
    int u = read_ll(), v = read_ll();
    adj[u].push_back(v);
    adj[v].push_back(u);
  }
  // Depth 0 is our dummy out-of-bounds node
  depth[0] = 0;
  dfs(1, 0, 1);
  for (int i = 1; i <= n; i++) {
    long long c = read_ll();
    add(in[i], c);
    add(out[i] + 1, -c);
  }
  for (int i = 0; i < q; i++) {
    int type = read_ll();
/* ================================================================
   >>> PUBLIC EDITION — roughly half of this solution is hidden for you to
complete. <<<
Rebuild it using the hints below the code and the reasoning in Phases 1-4 above.
   ================================================================ */
    } else {
      int u = read_ll();
      long long y = read_ll();
      add(in[u], y);
      add(out[u] + 1, -y);
```

```
    }
  }
  return 0;
}
```

**Hints for the missing half — enough for a champion, no more**

S(w) = the value sum on the root→w path (BIT query at in[w]). The v→u path sum = S(v) − S(parent(u)); if < x, answer -1.

Find the topmost node 'curr' by binary lifting from v: jump to ancestor p (never above u) while S(p) > S(v) − x — the part below p still falls short of x.

The answer = depth(v) − depth(curr) + 1: the number of nodes from v to curr inclusive. Also now hidden: the BIT helpers add()/query()/get_S() (standard Fenwick-tree point-update and prefix-sum, using get_S(u) = query(in[u]) to read node u's root-path sum), and dfs() (the Euler-tour + binary-lifting build: in[u]/out[u] timestamps, and up[u][i] = up[up[u][i-1]][i-1] for the ancestor table). The type==1 query branch was already hidden in the original edition and stays hidden here — rebuild it using the hints above.

**Verification log — this code was re-checked before printing**

The full code was recompiled with g++ -O2 -std=c++17 (0 errors) and smoke-tested. Archive status: ACCEPTED on SPOJ; this problem's official samples are not fully published, so the final verification is a fresh SPOJ submit — make that part of your drill.

## 8.4 Lab Completion Checklist

- [ ] Three hand-tree tests pass (5-node chain, star, balanced binary)
- [ ] The -1 case (path total below x) handled
- [ ] Jumps never overshoot u (depth check)
- [ ] ACCEPTED on SPOJ recorded

## 8.R Session Summary and Winning Points

- Subtree = Euler interval; root→w path = BIT prefix — two identities that unlock dozens of tree problems.
- Binary lifting answers along-the-path searches in $\log^2 n$.
- Techniques are memorized one by one, but medals come from composing them.

## References

[1] Cormen, Leiserson, Rivest, Stein — Introduction to Algorithms, 4th ed., MIT Press, 2022.

[2] Halim, S., Halim, F. — Competitive Programming 4, 2020.

[3] SPOJ — Sphere Online Judge, spoj.com (problem HBD).

## [4] SESSION 9 · LECTURE SESSION (2 HOURS)

# DP I: Counting Distinct Objects

Counting DP is mandatory OSN weaponry: how many ways, how many distinct objects, modulo a large prime. Its one danger: DOUBLE COUNTING. DSUBSEQ teaches the most elegant anti-duplicate mindset — a character's last occurrence owns all its duplication — and the same pattern appears in dozens of national-level counting problems.

**Session goals — after these 2 hours you can:**

(1) Formulate counting DPs with states of EXACTLY one meaning

(2) Derive DSUBSEQ's recurrence: double, then subtract the previous occurrence

(3) Compute safely in modular arithmetic (negatives, 64-bit products)

(4) Verify counting DPs against small brute-force enumerations

## 9.1 State Definitions: The Sentence that Saves You

Every counting-DP WA is rooted in a fuzzy state definition. Champion's discipline: write the definition as a FULL SENTENCE before any recurrence. For DSUBSEQ: 'dp[i] = the number of DISTINCT subsequences of the prefix s[1..i], the empty one included.' From that sentence the recurrence is proven — not guessed. Appending character c: every old subsequence may take c or not → 2·dp[i−1]. But if c last appeared at position p, subsequences ending in c from that era are counted twice — exactly dp[p−1] of them — subtract.

```
// Rekurensi DSUBSEQ / the DSUBSEQ recurrence
// dp[0] = 1                          (subsequence kosong / the empty subsequence)
// dp[i] = 2*dp[i-1] - dp[last[c]-1]  (kurangi bila c pernah muncul / subtract if c appeared)
// jawab dp[n] mod 1e9+7              (kosong ikut dihitung / empty included)
```

**Two classic modular traps**

(a − b) % MOD can go negative in C++ — always write (a − b + MOD) % MOD.

Read PRECISELY: 'empty included' vs 'excluded' shifts the answer by exactly one — and the verdict from AC to WA. DSUBSEQ's official text counts the empty one.

## 9.2 The 'Last Occurrence Owns the Duplicates' Pattern

- The same pattern counts: distinct strings after deletions, distinct labeled paths, distinct sub(multi)sets.

- Its general shape: total transitions − a correction from the last collision-causing occurrence.
- Mandatory verification: enumerate the full object set for n ≤ 12 and compare — five minutes that buy certainty.

## 9.3 HARD Case Study: DSUBSEQ (SPOJ #2416)

**Problem card**

SPOJ DSUBSEQ | Dynamic Programming | Hard | Distinct Subsequence Counting

Source: https://www.spoj.com/problems/DSUBSEQ/ — solve and submit directly on SPOJ.

### Problem statement (concise, faithful to the original SPOJ text)

For T ≤ 100 uppercase strings (length ≤ 100,000), count the number of DISTINCT subsequences — including the empty one — modulo $10^9+7$. 'AGH' is a subsequence of 'ABCDEFGH'; 'AHG' is not.

- The dp must be linear; answers are taken mod, with subtractions kept non-negative.

### Official sample input/output

```
Input:
2
AAA
ABCDEFG
Output:
4
128
```

### Phases 1-2 — Understand and Model

dp[i] = the number of distinct subsequences of the i-character prefix, the empty one included. A new character doubles the count; the same character's previous occurrence contributes exactly dp[that position − 1] duplicates, which must be subtracted.

### Phase 3 — Design the Algorithm

1. dp[0] = 1 (the empty subsequence); scan characters left to right.
2. dp[i] = 2·dp[i−1] (take the new character or not), then − dp[last[c]−1] if c appeared before.
3. last[26] updates after use; subtractions guarded by (x + MOD) % MOD.
4. The answer is dp[n] — the official text counts the empty subsequence.

### Phase 4 — Complexity Analysis

O(n) per string, n ≤ $10^5$, T ≤ 100 — at most $10^7$ steps. The only ways to fail are misreading 'empty included' or sloppy modular hygiene.

### Phase 5 — Implementation (Public Edition — ~50% revealed, ~50% your task)

This is the Public Edition of this solution. Roughly half of the code is shown below; the rest — the part that actually solves the problem — is your task, guided by the hints beneath the code and the walkthrough in Phases 1-4 above.

```
#include <cstdio>
#include <cstring>
using namespace std;
const int MOD=1000000007;
long long dp[100002];
int last[26];
int main() {
  int t; scanf("%d",&t);
  while(t--) {
/* ============================================================
   >>> PUBLIC EDITION — roughly half of this solution is hidden for you to complete. <<<
Rebuild it using the hints below the code and the reasoning in Phases 1-4 above.
   ============================================================ */
    }
    printf("%lld\n",dp[n]%MOD);
  }
}
```

**Hints for the missing half — enough for a champion, no more**

dp[i] = distinct subsequences of the first i characters (empty included). Appending character c doubles: dp[i] = 2·dp[i−1].

Duplicates appear when c occurred before at position p: every subsequence that then ended with c is counted twice — subtract dp[p−1].

Keep each letter's last position in last[26]; guard subtractions with +MOD before %MOD. Also now hidden: reading the string and resetting last[]/dp[0]. The dp recurrence that was already hidden is the whole algorithm — rebuild it from the hints above: dp[i] = 2*dp[i-1], then subtract dp[p-1] when the current character last appeared at position p, guarding with +MOD before %MOD.

**Verification log — this code was re-checked before printing**

Semantics checked against the official SPOJ text (the empty subsequence counts). The full code was recompiled with g++ -O2 -std=c++17 (0 errors), tested against the official SPOJ samples, and stress-tested against a brute-force reference on hundreds of random cases — all identical. Archive status: ACCEPTED on SPOJ.

## 9.4 Session 9 Drills

1. Complete DSUBSEQ's half (the recurrence loop), verify 'AAA' → 4 and 'ABCDEFG' → 128, then brute-force n ≤ 12.
2. Derive the variant: distinct subsequences OF LENGTH k (a two-dimensional dp).
3. Prove by induction that dp[i] is duplicate-free — a five-sentence written proof.

## 9.R Session Summary and Winning Points

- A state definition = a full sentence; recurrences are proven from it.
- Double then subtract dp[last−1]: the last occurrence owns the duplicates.
- (a − b + MOD) % MOD — a reflex, not a choice.
- Every counting DP is verified against a small enumeration before submitting.

## References

[1] Cormen, Leiserson, Rivest, Stein — Introduction to Algorithms, 4th ed., MIT Press, 2022.

[2] Halim, S., Halim, F. — Competitive Programming 4, 2020.

[3] SPOJ — Sphere Online Judge, spoj.com (problem DSUBSEQ).

## [4] SESSION 10 · LECTURE SESSION (2 HOURS)

# DP II: Bitmask & Profile DP

N ≤ 17 is the jury's secret code: 'exponential is welcome, if controlled'. Bitmask DP turns sets into array indices; profile DP turns a grid's CROSS-SECTION into state. CFONISMG combines both — a used-students bitmask × a profile of the last C seats — the perfect placement problem: scary to the eye, mechanical to trained hands.

**Session goals — after these 2 hours you can:**

(1) Read N ≤ 17-20 as an invitation to controlled exponential/bitmask work

(2) Design combined (bitmask, profile) states with cell-by-cell transitions

(3) Control state space with maps + pruning unreachable states

(4) Estimate state counts BEFORE coding to ensure the time fits

## 10.1 Bitmasks: Sets as Numbers

- mask & (1<<i): is i in the set; mask | (1<<i): insert i; mask ^ (1<<i): remove.
- Iterating subsets of subsets: for (int s = m; s; s = (s-1) & m) — 3^N over all masks, not 4^N.
- __builtin_popcount, __builtin_ctz: old friends that save lines and time.

The mandatory mental example: bitmask TSP dp[mask][city] = the shortest route visiting set mask ending at city — O(2^N · N²). Every 'visit all, exactly once, minimize/count' problem with N ≤ 20 is its relative.

## 10.2 Profile DP: The Cross-Section as State

Filling a grid cell by cell (left→right, top→bottom), the current decision only interacts with the TOP and LEFT neighbors. So it suffices to remember the 'profile': the last C cells' contents. CFONISMG encodes it in base 17 (0 = empty, 1..16 = a student) — ≤ $17^4$ profiles, joined with a $2^{17}$ used-students bitmask. The efficiency key: store ONLY genuinely reached states in an unordered_map, because most (bitmask, profile) combinations are mutually inconsistent.

**Estimate first, code second**

Before one line of code: estimate reached states × transition cost. In CFONISMG the practical count sits far below the theoretical $2^{17}\cdot17^4$ because popcount(bitmask) must match the filled-cell count. If the estimate tops $10^8$ operations — redesign, don't hope.

## 10.3 HARD Case Study: CFONISMG (SPOJ #42573)

**Problem card**

SPOJ CFONISMG | Classical | Hard | Profile Bitmask DP

Source: https://www.spoj.com/problems/CFONISMG/ — solve and submit directly on SPOJ.

### Problem statement (concise, faithful to the original SPOJ text)

An R×C classroom (C ≤ 4) is filled from a list of N ≤ 17 candidate students, each usable once; cells may stay empty. A pair of friends seated adjacently (top/left) adds 1 point; a pair of rivals — or two 'disruptive' students — may not be adjacent. Maximize the total score.

- DP state: (bitmask of used students, profile of the last C cells in base 17) — the dp advances cell by cell.

### Phases 1-2 — Understand and Model

Fill the classroom cell by cell. State = (bitmask of used students, base-17 profile of the last C cells). Transitions try seating each remaining student — or leaving the cell empty — enforcing rival/disruptive bans against the top and left neighbors, and scoring each friendship formed.

### Phase 3 — Design the Algorithm

1. The base-17 profile: profile[0] = the current cell's top neighbor; profile[C−1] = the left neighbor (when col > 0).
2. try_place(s): reject if s is used / adjacent to a rival / two disruptives; score += friendships with top & left.
3. New key = (bitmask | s's bit) << 17 | (the shifted profile + s·17^{C−1}); keep the max score per key in ndp.
4. The unordered_map holds only reached states; dp = move(ndp) each cell.

### Phase 4 — Complexity Analysis

The theoretical $2^{17}\cdot 17^4$ states never materialize: the bitmask's popcount is tied to the filled-cell count. In practice hundreds of thousands of states × ≤ 18 transitions — under $10^8$ light operations.

### Phase 5 — Implementation (Public Edition — ~50% revealed, ~50% your task)

This is the Public Edition of this solution. Roughly half of the code is shown below; the rest — the part that actually solves the problem — is your task, guided by the hints beneath the code and the walkthrough in Phases 1-4 above.

```
#include <bits/stdc++.h>
using namespace std;
int R,C,N,cells;
bool disruptive[17],is_rival[17][17],is_friend_pair[17][17];
int main() {
```

```
  ios_base::sync_with_stdio(false);
  cin.tie(NULL);
  cin>>R>>C>>N;
  cells=R*C;
  map<string,int> id;
  for(int i=1; i<=N; i++) {
    string nm,st;
    cin>>nm>>st;
    id[nm]=i;
    disruptive[i]=(st=="DISRUPTIVE");
  }
  int F;
  cin>>F;
  for(int i=0; i<F; i++) {
    string a,b;
    cin>>a>>b;
    int ia=id[a],ib=id[b];
    is_friend_pair[ia][ib]=is_friend_pair[ib][ia]=true;
  }
  int V;
  cin>>V;
  for(int i=0; i<V; i++) {
    string a,b;
    cin>>a>>b;
    int ia=id[a],ib=id[b];
    is_rival[ia][ib]=is_rival[ib][ia]=true;
  }
  int pw[5]= {1,17,289,4913,83521};
  const int PB=17;
  unordered_map<long long,int> dp;
  dp.reserve(1<<18);
  dp[0]=0;
  for(int cell=0; cell<cells; cell++) {
    int row=cell/C,col=cell%C;
    unordered_map<long long,int> ndp;
    ndp.reserve(dp.size()*4);
    for(auto&[key,score]:dp) {
/* ================================================================
   >>> PUBLIC EDITION — roughly half of this solution is hidden for you to
complete. <<<
Rebuild it using the hints below the code and the reasoning in Phases 1-4 above.
   ================================================================ */
    }
    dp=move(ndp);
  }
  int ans=0;
  for(auto&[k,v]:dp)ans=max(ans,v);
  cout<<ans<<"\n";
}
```

### Hints for the missing half — enough for a champion, no more

try_place(s): reject if s is already used in the bitmask, or if it neighbors (top = profile[0], left = profile[C−1] when col>0) a rival / a fellow disruptive student.

Score grows by 1 per (top, left) neighbor who is friends with s. An empty cell = s = 0, always legal.

New key: shift the base-17 profile (drop the oldest cell, append s), pack with the new bitmask; keep the maximum score per key in ndp. Now the entire per-state transition body is hidden, including the unpacking that was previously shown (splitting key into bitmask + base-17 profile, reading top/left neighbor students). Rebuild try_place(s) from the hints above: reject if s is already used or conflicts with a rival/fellow-disruptive neighbor, score += 1 per friend neighbor, then push the shifted profile+bitmask into ndp keeping the max score per key.

**Verification log — this code was re-checked before printing**

The full code was recompiled with g++ -O2 -std=c++17 (0 errors) and smoke-tested. Archive status: ACCEPTED on SPOJ; this problem's official samples are not fully published, so the final verification is a fresh SPOJ submit — make that part of your drill.

## 10.4 Session 10 Drills

1. Complete CFONISMG's half (the try_place lambda): rival/disruptive checks, friendship scoring, key updates.
2. Write pure bitmask TSP for N = 16 random cities; target $O(2^N \cdot N^2)$ under 1 second.
3. Measure experimentally: how many states does CFONISMG reach on a 4×4 grid with 16 students? Compare with the theoretical bound.

## 10.R Session Summary and Winning Points

- N ≤ 17-20 = official permission for controlled exponentials.
- The profile = the last C cells' cross-section; the next decision sees nothing else.
- A reached-states map prunes the bloated theoretical space.
- Estimate operations before coding — a number saying no is cheaper than a TLE.

## References

[1] Cormen, Leiserson, Rivest, Stein — Introduction to Algorithms, 4th ed., MIT Press, 2022.

[2] Halim, S., Halim, F. — Competitive Programming 4, 2020.

[3] SPOJ — Sphere Online Judge, spoj.com (problem CFONISMG).

## [4] SESSION 11 · LAB SESSION (2 HOURS)

# Lab 3 — DP at Scale: CDQ Divide & Conquer

Two-dimensional LIS at n = $10^5$ kills quadratic DP. This lab dissects CDQ: split the sequence in half, solve the left, FUNNEL its influence into the right with a sweep + BIT, then solve the right. The same technique answers many 'DP with two-dimensional conditions' — one of the strongest weapons approaching the national level.

**Session goals — after these 2 hours you can:**

(1) Explain why the solve(left) → cross → solve(right) order is mandatory

(2) Implement cross(): two pointers over x + a max-BIT over y

(3) Clean the BIT selectively so $\log^2$ stays $\log^2$

## 11.1 Why CDQ Works

dp[i] = the best 2-D LIS ending at i, depending on all j < i with xj < xi and yj < yi. Three order dimensions (index, x, y) are hard to juggle at once — CDQ deletes ONE: inside cross(l, mid, r), every left element automatically has a smaller index than every right one. Two dimensions remain: sort both sides by x (two pointers), and handle y with a max-BIT over compressed y. The recursion T(n) = 2T(n/2) + O(n log n) = O(n $\log^2$ n).

**Order is everything**

solve(left) MUST finish before cross, because cross reads left dps as final; and solve(right) MUST come after, because right dps are only complete once the left influence lands. Swapping the order = wrong answers that pass small cases — the most dangerous bug species.

## 11.2 The 120-Minute Plan

| Minutes | Activity |
|---|---|
| 0-15 | Read LIS2; draw the cross diagram on paper |
| 15-55 | Complete the half (the cross function) + clean compile |
| 55-85 | Stress-test vs the O($n^2$) DP — 40+ random cases |
| 85-100 | Submit; chase the AC |

| Minutes | Activity |
|---|---|
| 100-120 | Discussion: CDQ for 3-D dominance counting & weighted variants |

## 11.3 HARD Case Study: LIS2 (SPOJ #2371)

**Problem card**

SPOJ LIS2 | Dynamic Programming | Hard | CDQ Divide & Conquer

Source: https://www.spoj.com/problems/LIS2/ — solve and submit directly on SPOJ.

### Problem statement (concise, faithful to the original SPOJ text)

Given n ≤ 100,000 pairs (x, y) in fixed order. Find the longest subsequence in which BOTH coordinates strictly increase: xi < xj and yi < yj for consecutive picks — a two-dimensional LIS.

- The $O(n^2)$ DP dies at $n = 10^5$; CDQ divide & conquer + a BIT brings it to $O(n \log^2 n)$.

### Phases 1-2 — Understand and Model

dp[i] = the best 2-D LIS ending at point i. CDQ splits by index: solve the left, funnel its influence into the right via a two-pointer x sweep with a max-BIT over compressed y, then solve the right.

### Phase 3 — Design the Algorithm

1. solve(l, r): solve(l, mid) → cross(l, mid, r) → solve(mid+1, r) — an absolute order.
2. cross: sort left & right by x; for each right element, first insert every smaller-x left element into the BIT (y position, dp value).
3. dp[right] = max(dp[right], bq(y−1) + 1) — the y−1 enforces strictness.
4. Clean the BIT only at written positions (bc) before returning.

### Phase 4 — Complexity Analysis

$T(n) = 2T(n/2) + O(n \log n) = O(n \log^2 n)$: $n = 10^5$ means $\pm 3 \times 10^7$ BIT operations — roughly a tenth of a second. The quadratic DP is for stress-testing only.

### Phase 5 — Implementation (Public Edition — ~50% revealed, ~50% your task)

This is the Public Edition of this solution. Roughly half of the code is shown below; the rest — the part that actually solves the problem — is your task, guided by the hints beneath the code and the walkthrough in Phases 1-4 above.

```
#include <cstdio>
#include <algorithm>
using namespace std;
const int MAXN=100001;
int px[MAXN],py[MAXN],dp[MAXN],yc[MAXN],n,yN;
```

```
int bit[MAXN];
/* ================================================================
   >>> PUBLIC EDITION — roughly half of this solution is hidden for you to
complete. <<<
Rebuild it using the hints below the code and the reasoning in Phases 1-4 above.
   ================================================================ */
void solve(int l,int r){
  if(l>=r)return;
  int mid=(l+r)/2;
  solve(l,mid);
  cross(l,mid,r);
  solve(mid+1,r);
}
int main(){
  scanf("%d",&n);
  for(int i=0;i<n;i++)scanf("%d%d",&px[i],&py[i]);
  int ys[MAXN];
  for(int i=0;i<n;i++)ys[i]=py[i];
  sort(ys,ys+n);
  yN=unique(ys,ys+n)-ys;
  for(int i=0;i<n;i++)yc[i]=(int)(lower_bound(ys,ys+yN,py[i])-ys)+1;
  for(int i=0;i<n;i++)dp[i]=1;
  solve(0,n-1);
  int ans=0;
  for(int i=0;i<n;i++)ans=max(ans,dp[i]);
  printf("%d\n",ans);
}
```

**Hints for the missing half — enough for a champion, no more**

cross(l, mid, r) funnels the left half's influence into the right half: sort both sides by x, then run two pointers.

For each right element (in increasing x), first insert every left element with smaller x into the BIT (position = compressed y, value = dp); then dp[right] = max(dp[right], bq(y−1) + 1).

Clean the BIT only at the positions written (bc), never a full memset — that is what keeps $\log^2$ at $\log^2$.

The CDQ order is mandatory: solve(left) first, then cross, then solve(right) — left dps must be final before funneling. Also now hidden: the BIT helpers bu()/bq()/bc() (standard Fenwick-tree max-update, max-query, and point-clear) and the lb[]/rb[] declaration. cross() itself was already hidden in the original edition and stays hidden here — rebuild it using the hints above: two pointers by x, insert-then-query the BIT by compressed y, and clear only the positions you touched.

**Verification log — this code was re-checked before printing**

The full code was recompiled with g++ -O2 -std=c++17 (0 errors), tested against the official SPOJ samples, and stress-tested against a brute-force reference on hundreds of random cases — all identical. Archive status: ACCEPTED on SPOJ.

## 11.4 Lab Completion Checklist

- [ ] The cross diagram drawn and assistant-approved
- [ ] 40 stress cases identical to the $O(n^2)$ reference
- [ ] Selective BIT cleanup (bc), no full memset

- [ ] ACCEPTED on SPOJ recorded

## 11.R Session Summary and Winning Points

- CDQ deletes the index dimension; the remaining two go to sort + BIT.
- The solve-cross-solve order is a correctness contract, not style.
- Selective cleanup keeps $O(n \log^2 n)$ honest.

## References

[1] Cormen, Leiserson, Rivest, Stein — Introduction to Algorithms, 4th ed., MIT Press, 2022.

[2] Halim, S., Halim, F. — Competitive Programming 4, 2020.

[3] SPOJ — Sphere Online Judge, spoj.com (problem LIS2).

## [4] SESSION 12 · LECTURE SESSION (2 HOURS)

# Meet in the Middle: Halving the Exponential

Six free variables from a 100-number set: $10^{12}$ combinations — impossible. But the equation a·b + c = d·(e + f) has two natural sides, each only $n^3 = 10^6$ values. Generate the left, generate the right, sort one, and match: $10^{12}$ collapses to $10^6$ log $10^6$. That is meet in the middle — the art of splitting one explosion into two small ones that can shake hands.

**Session goals — after these 2 hours you can:**

(1) Recognize 'separable two sides' structure in equations/sets

(2) Count occurrences with sort + upper_bound − lower_bound

(3) Estimate both halves' memory and time before coding

(4) Generalize to n ≤ 40 subset-sum and its relatives

## 12.1 The General MITM Recipe

- Split variables/objects into two halves that never interact EXCEPT through one linking value.
- Enumerate each half; store the linking values (an array for counting, a map when extra data rides along).
- Sort one side; for each value on the other, count partners binarily: upper_bound − lower_bound.
- Subset-sum at n = 40: two $2^{20}$ halves = a million each — from impossible to a second.

**Why d ≠ 0 is handled at GENERATION time**

Filtering at the end pollutes R with illegal values you must remember to exclude. The champion's principle: enforce a constraint where it is born — the d loop skips zero, and the rest of the program never needs to know the constraint existed.

## 12.2 Arithmetic that Saves or Buries

- $|a·b + c| \le 30{,}000^2 + 30{,}000 \approx 9\times10^8$ — fits an int, BUT the number of matches can far exceed 32-bit: the answer must be long long.
- $10^6$ long longs = 8 MB per side — check the memory limit BEFORE choosing types.
- sort + two binary searches per left value ≈ $10^6 \times 2 \log(10^6) \approx 4\times10^7$ — comfortably under a second.

## 12.3 HARD Case Study: ABCDEF (SPOJ #4580)

**Problem card**

SPOJ ABCDEF | Searching & Sorting | Hard | Meet in the Middle

Source: https://www.spoj.com/problems/ABCDEF/ — solve and submit directly on SPOJ.

### Problem statement (concise, faithful to the original SPOJ text)

Given a set S of n ≤ 100 distinct integers (|value| ≤ 30,000). Count the tuples (a,b,c,d,e,f) — all from S, repetition allowed — satisfying (a·b + c) / d − e = f with d ≠ 0; equivalently: a·b + c = d·(e + f).

- Six variables → $10^{12}$ naive combinations. Split: left a·b+c ($n^3$), right d·(e+f) ($n^3$) — then match.

### Official sample input/output

```
Input:
1
1
Output:
1
```

### Phases 1-2 — Understand and Model

Rewrite the equation as a·b + c = d·(e + f). The left side touches only (a,b,c), the right only (d,e,f) — two $n^3$ halves meeting through one linking value.

### Phase 3 — Design the Algorithm

1. Generate L = every a·b + c ($n^3$ values, 64-bit).
2. Generate R = every d·(e + f), skipping d = 0 AT generation; sort R.
3. The answer = Σ over L of (upper_bound − lower_bound) in R.
4. A long long accumulator: match counts can exceed 32-bit.

### Phase 4 — Complexity Analysis

$2 \cdot 10^6$ generations + a $10^6$ sort + $10^6 \times 2$ binary searches ≈ $5 \times 10^7$ operations — under a second at -O2. Memory: two 8 MB arrays.

### Phase 5 — Implementation (Public Edition — ~50% revealed, ~50% your task)

This is the Public Edition of this solution. Roughly half of the code is shown below; the rest — the part that actually solves the problem — is your task, guided by the hints beneath the code and the walkthrough in Phases 1-4 above.

```
#pragma GCC optimize("O3,unroll-loops")
#include <cstdio>
#include <algorithm>
```

```
using namespace std;
int s[101]; long long L[1000001],R[1000001];
int main() {
  int n;scanf("%d",&n);
  for(int i=0;i<n;i++) scanf("%d",&s[i]);
/* ================================================================
   >>> PUBLIC EDITION — roughly half of this solution is hidden for you to complete. <<<
Rebuild it using the hints below the code and the reasoning in Phases 1-4 above.
   ================================================================ */
  printf("%lld\n",ans);
}
```

**Hints for the missing half — enough for a champion, no more**

Generate the right side: for every d ≠ 0 and all e, f → store d·(e+f) in R ($n^3$ values), then sort R.

For each left value L[i], the number of matches = upper_bound − lower_bound on R — an occurrence count in a sorted array.

64-bit is mandatory: |a·b+c| reaches ~$9\times10^8$, and the answer itself can exceed 32-bit. Now the L-array generation loop (previously shown) is hidden too, alongside the already-hidden R-array generation, sort, and counting. Rebuild all of it from the hints above: L[i,j,k] = s[i]*s[j]+s[k] for the target sign, R generated the mirrored way for every d != 0, sort R, then count matches per L value with upper_bound - lower_bound in 64-bit.

**Verification log — this code was re-checked before printing**

The formula was checked against the official SPOJ text; all four official samples ({1}→1, {2,3}→4, {−1,1}→24, {5,7,10}→10) pass. The full code was recompiled with g++ -O2 -std=c++17 (0 errors), tested against the official SPOJ samples, and stress-tested against a brute-force reference on hundreds of random cases — all identical. Archive status: ACCEPTED on SPOJ.

## 12.4 Session 12 Drills

1. Complete ABCDEF's half (right-side generation + binary matching); verify all four official samples.
2. Solve n = 38 subset-sum: is there a subset totaling exactly S? — pure MITM.
3. Analyze: why does moving e to the left side break the $n^3/n^3$ symmetry? Two written sentences.

## 12.R Session Summary and Winning Points

- MITM = split, enumerate two small explosions, shake hands through a linking value.
- upper_bound − lower_bound counts occurrences in a sorted array.
- Constraints are enforced at generation; answers are always 64-bit.
- n = 40 exponentials → two $2^{20}$ halves: the same pattern across national problems.

# References

[1] Cormen, Leiserson, Rivest, Stein — Introduction to Algorithms, 4th ed., MIT Press, 2022.

[2] Halim, S., Halim, F. — Competitive Programming 4, 2020.

[3] SPOJ — Sphere Online Judge, spoj.com (problem ABCDEF).

## [4] SESSION 13 · LECTURE SESSION (2 HOURS)

# Strings: Suffix Arrays & LCP

One structure answers nearly every substring question: sort all the suffixes. A substring = a prefix of a suffix, so a suffix array + LCP table turns 'how many distinct substrings' into one line of arithmetic: n(n+1)/2 minus the neighbors' overlaps. DISUBSTR is the entry door; the next doors (pattern search, longest repeated substring) use the same key.

**Session goals — after these 2 hours you can:**

(1) Build a suffix array by doubling in $O(n \log^2 n)$

(2) Build neighbor LCPs with Kasai's O(n) algorithm

(3) Derive the distinct-substring formula: $n(n+1)/2 - \Sigma$ lcp

(4) Map other substring problems onto the (SA, LCP) pair

## 13.1 Doubling: Sorting Suffixes by Rank Pairs

Sorting suffixes with strcmp costs $O(n^2 \log n)$ — dead for large n. Doubling sorts by the first 2·gap characters using rank PAIRS: (rank[i], rank[i+gap]). Starting from rank = the character, each iteration doubles the sorted length, and new ranks come from comparing neighboring pairs. log n iterations × an O(n log n) sort = $O(n \log^2 n)$ — ample for DISUBSTR's $n \le 10^3$ and viable to $n = 10^5$.

- Out-of-string rank[i+gap] = −1: shorter suffixes always sort first.
- Stop early once ranks are unique — a big saving on random strings.
- Kasai leans on a beautiful fact: the lcp of suffix i (in text order) drops by at most 1 from suffix i−1 — so h moves forward O(n) in total.

## 13.2 From LCPs to Answers

Every suffix sa[i] contributes all its prefixes as substrings: n − sa[i] of them. But its first lcp[i] characters were already contributed by the previous neighbor — duplicates. So distinct substrings = $\Sigma$ (n − sa[i]) − $\Sigma$ lcp[i] = $n(n+1)/2 - \Sigma$ lcp[i]. Official samples: 'CCCCC' → 15 − (4+3+2+1) = 5; 'ABABA' → 15 − (3+2+1+0) = 9.

**The (SA, LCP) problem dictionary**

Longest repeated substring = max(lcp). Pattern search for P = binary search on the SA. K-th distinct substring = walk the SA spending a quota. K-th smallest suffix = sa[k]. One investment, four problems.

## 13.3 HARD Case Study: DISUBSTR (SPOJ #694)

**Problem card**

SPOJ DISUBSTR | Strings | Hard | Suffix Array + LCP

Source: https://www.spoj.com/problems/DISUBSTR/ — solve and submit directly on SPOJ.

### Problem statement (concise, faithful to the original SPOJ text)

For T ≤ 20 strings (length ≤ 1,000), count the number of DISTINCT substrings. Official samples: 'CCCCC' → 5, 'ABABA' → 9.

- Total substrings = n(n+1)/2; duplicates are removed via LCPs of adjacent suffixes in the suffix array.

### Official sample input/output

```
Input:
2
CCCCC
ABABA
Output:
5
9
```

### Phases 1-2 — Understand and Model

Every substring is a prefix of a suffix. Sort all suffixes (the suffix array), measure neighbor overlaps (LCP), and distinct substrings = n(n+1)/2 − Σ lcp.

### Phase 3 — Design the Algorithm

1. Doubling: sort by the pair (rank[i], rank[i+gap]); out-of-string ranks are −1; double the gap.
2. New ranks: +1 over the neighbor when key pairs differ; stop early once all unique.
3. Kasai: process text positions in order, h drops at most 1 — full LCPs in O(n).
4. A long long answer: n(n+1)/2 can exceed int for larger n.

### Phase 4 — Complexity Analysis

The O(n log² n) SA build at n ≤ 1,000 is microseconds; T ≤ 20 cases change nothing. The formula carries all the problem's value.

### Phase 5 — Implementation (Public Edition — ~50% revealed, ~50% your task)

This is the Public Edition of this solution. Roughly half of the code is shown below; the rest — the part that actually solves the problem — is your task, guided by the hints beneath the code and the walkthrough in Phases 1-4 above.

```
#include <cstdio>
```

```cpp
#include <cstring>
#include <algorithm>
using namespace std;
const int MAXN = 1001;
int sa[MAXN], rk[MAXN], tmp[MAXN], lcp[MAXN];
int n;
char s[MAXN];
void build_sa() {
  for (int i = 0; i < n; i++) {
    sa[i] = i;
    rk[i] = s[i];
  }
/* ================================================================
   >>> PUBLIC EDITION — roughly half of this solution is hidden for you to
complete. <<<
Rebuild it using the hints below the code and the reasoning in Phases 1-4 above.
   ================================================================ */
  }
}
/* ================================================================
   >>> PUBLIC EDITION — roughly half of this solution is hidden for you to
complete. <<<
Rebuild it using the hints below the code and the reasoning in Phases 1-4 above.
   ================================================================ */
int main() {
  int t;
  scanf("%d", &t);
  while (t--) {
    scanf("%s", s);
    n = strlen(s);
    build_sa();
    build_lcp();
    long long total = (long long)n*(n+1)/2;
    for (int i = 1; i < n; i++) total -= lcp[i];
    printf("%lld\n", total);
  }
}
```

### Hints for the missing half — enough for a champion, no more

Doubling: at iteration gap, sort suffixes by the pair (rank[i], rank[i+gap]); out-of-string rank[i+gap] = −1.

New ranks: tmp[sa[0]] = 0; tmp[sa[i]] = tmp[sa[i−1]] + 1 when its key pair differs from its neighbor, otherwise equal.

Stop early once ranks are all unique (rk[sa[n−1]] == n−1). Final answer: n(n+1)/2 − Σ lcp[i]. Also now hidden: build_lcp() (Kasai's algorithm) in full — rebuild it from the hints above: keep h from the previous suffix (never resetting to 0), walk rank[i] to find the suffix immediately before it in sorted order, extend the common prefix while characters match, record it, and decrement h by at most one for the next i. The doubling core inside build_sa() was already hidden in the original edition and stays hidden here.

### Verification log — this code was re-checked before printing

The full code was recompiled with g++ -O2 -std=c++17 (0 errors), tested against the official SPOJ samples, and stress-tested against a brute-force reference on hundreds of random cases — all identical. Archive status: ACCEPTED on SPOJ.

## 13.4 Session 13 Drills

1. Complete DISUBSTR's half (the doubling loop); verify CCCCC → 5, ABABA → 9, then brute-force all strings of length ≤ 8.
2. Use the same SA+LCP to find 'BANANA''s longest repeated substring. Answer on paper first.
3. Challenge: answer 'the k-th lexicographic distinct substring' by walking the SA.

## 13.R Session Summary and Winning Points

- Substrings = suffix prefixes; sort the suffixes and the substring world opens.
- Doubling: rank pairs double the sorted length each iteration.
- Kasai in O(n): lcp drops at most 1 between consecutive text positions.
- Distinct = n(n+1)/2 − Σ lcp — the one line that wins DISUBSTR.

## References

[1] Cormen, Leiserson, Rivest, Stein — Introduction to Algorithms, 4th ed., MIT Press, 2022.

[2] Halim, S., Halim, F. — Competitive Programming 4, 2020.

[3] SPOJ — Sphere Online Judge, spoj.com (problem DISUBSTR).

## [4] SESSION 14 · LECTURE SESSION (2 HOURS)

# Competitive Mathematics: Finite Differences, Interpolation & Modular Arithmetic

Some of the scariest 'count the placements' problems collapse to one observation: the answer is a polynomial in N. Detection is mechanical — finite differences going constant — and once the degree is known, brute force on a few small points + Lagrange interpolation delivers a closed form. BATTLECRY (two-rook checkmates on an N×N board) shows off the full pipeline: small brute → detect degree 5 → interpolate → O(1) modular Horner per query.

**Session goals — after these 2 hours you can:**

(1) Detect polynomial answers via finite-difference tables

(2) Build closed forms by Lagrange interpolation from brute-forced points

(3) Evaluate polynomials modularly with Horner + modular inverses

(4) Watch the trap: small cases outside the pattern (here N < 3)

## 14.1 Finite Differences: The Polynomial Stethoscope

Take f(3), f(4), ..., f(10) from a brute force. Compute difference rows: Δf = f(k+1) − f(k); repeat. If row d is constant (and row d+1 zero), f is a degree-d polynomial on that domain. For BATTLECRY the fifth differences are the constant 480 → degree 5 → six points pin f down completely. This is a champion's first diagnostic on one-parameter placement counts.

- The small brute force is NOT waste: it is data. Write an honest $O(N^6)$ brute for $N \le 8$, tabulate, diagnose.
- Interpolation gives fractional coefficients? Multiply out the common denominator (here 3), evaluate, then divide back with a modular inverse.
- Mind the domain: the polynomial pattern often starts at some N — N < 3 gives 0 in BATTLECRY.

## 14.2 Horner & Modular Inverses: Championship Evaluation

```
// Evaluasi p(v) = 12v^5 - 56v^4 + 66v^3 + 554v^2 - 2292v + 2424 (mod M)
// Horner: (((((12v - 56)v + 66)v + 554)v - 2292)v + 2424)
// 5 perkalian, 5 penjumlahan — dan SETIAP langkah di-mod / 5 mults, 5 adds —
EVERY step reduced
// Pembagian oleh 3 = perkalian INV3 = pow(3, M-2, M) (Fermat, M prima)
```

**Three modular sins**

Forgetting +MOD before % on negative coefficients; multiplying two long longs near $2^{63}$ without intermediate reduction; dividing with '/' instead of a modular inverse. All three pass small cases and explode on jury cases.

## 14.3 HARD Case Study: BATTLECRY (SPOJ #40192)

**Problem card**

SPOJ BATTLECRY | Classical | Hard | Finite Differences / Lagrange Interpolation

Source: https://www.spoj.com/problems/BATTLECRY/ — solve and submit directly on SPOJ.

### Problem statement (concise, faithful to the original SPOJ text)

Count the placements of a White King, Black King, and two identical White Rooks on an N×N board such that the kings are non-adjacent and the Black King is checkmated — modulo 7,340,033, for many values of N ($N < 10^5$). Output format: 'Case k: answer'.

- For $N \geq 3$ the answer is a degree-5 polynomial in N — 5th finite differences are constant; $N < 3$ gives 0.

### Phases 1-2 — Understand and Model

A small brute force shows f(N) is a degree-5 polynomial for $N \geq 3$ (fifth differences constant at 480) and 0 for $N < 3$. Lagrange interpolation over six points gives $3 \cdot f(N) = 12N^5 - 56N^4 + 66N^3 + 554N^2 - 2292N + 2424$; answer each query by modular Horner then multiply by 3's inverse.

### Phase 3 — Design the Algorithm

1. $N < 3 \rightarrow 0$ immediately; the polynomial pattern starts at N = 3.
2. Five Horner steps: start $12v - 56$, each step multiplies by v and adds a coefficient, all mod 7,340,033.
3. Negative coefficients secured with +MOD before %.
4. Close with × INV3 (= pow(3, MOD−2)) — legitimate modular division.

### Phase 4 — Complexity Analysis

O(1) per query — five modular multiplications. No arrays, no precomputation; all the problem's intelligence has moved onto paper in the analysis stage.

### Phase 5 — Implementation (Public Edition — ~50% revealed, ~50% your task)

This is the Public Edition of this solution. Roughly half of the code is shown below; the rest — the part that actually solves the problem — is your task, guided by the hints beneath the code and the walkthrough in Phases 1-4 above.

```
#include <stdio.h>
```

```c
typedef long long ll;
#define MOD 7340033LL
#define INV3 2446678LL /* pow(3, MOD-2, MOD) = 2446678 */
static ll solve(ll n) {
  ll v, h;
  if (n < 3) return 0LL;
/* ================================================================
   >>> PUBLIC EDITION — roughly half of this solution is hidden for you to complete. <<<
Rebuild it using the hints below the code and the reasoning in Phases 1-4 above.
   ================================================================ */
}
int main(void) {
  int T, t;
  ll N;
  scanf("%d", &T);
  for (t = 1; t <= T; t++) {
    scanf("%lld", &N);
    printf("Case %d: %lld\n", t, solve(N));
  }
  return 0;
}
```

**Hints for the missing half — enough for a champion, no more**

$3 \cdot f(N) = 12N^5 - 56N^4 + 66N^3 + 554N^2 - 2292N + 2424$ — evaluate with Horner: start from 12v−56, multiply by v and add the next coefficient in turn, all under the modulus.

Negative coefficients are kept positive with +MOD before %MOD.

Finish by multiplying the modular inverse of 3 (INV3 = 2446678) to divide by 3 legally in modular arithmetic. Now the whole solve() body past the base case is hidden: the v=n%MOD reduction line, the formula comment, and the Horner evaluation itself. Rebuild from the hints above: start from 12v-56, repeatedly multiply by v and add the next coefficient (66, 554, -2292, 2424 in order, each guarded with +MOD before %MOD), then multiply by INV3 to divide by 3 under the modulus.

**Verification log — this code was re-checked before printing**

The full code was recompiled with g++ -O2 -std=c++17 (0 errors) and smoke-tested. Archive status: ACCEPTED on SPOJ; this problem's official samples are not fully published, so the final verification is a fresh SPOJ submit — make that part of your drill.

## 14.4 Session 14 Drills

1. Complete BATTLECRY's half (Horner + INV3); test f(3) = 232 per the brute-force table.
2. Write your own N ≤ 8 brute force and reproduce the difference table down to the constant row 480.
3. Interpolation drill: f is degree-3 with f(1..4) = 1, 5, 14, 30 — find f(10) without explicit coefficients (direct Lagrange).

## 14.R Session Summary and Winning Points

- Constant differences at row d = a degree-d polynomial — a mechanical diagnosis.
- Small brutes are data; interpolation turns them into O(1) formulas.
- Modular division = multiplying an inverse (Fermat on prime moduli).
- Always check the pattern's domain: small out-of-pattern cases get answered separately.

## References

[1] Cormen, Leiserson, Rivest, Stein — Introduction to Algorithms, 4th ed., MIT Press, 2022.

[2] Halim, S., Halim, F. — Competitive Programming 4, 2020.

[3] SPOJ — Sphere Online Judge, spoj.com (problem BATTLECRY).

## [4] SESSION 15 · LAB SESSION (2 HOURS)

# Lab 4 — Full Contest Simulation + the Max-Flow Boss Problem

The closing session: a dress rehearsal of the real thing. Two hours, full OSN rules, and one boss problem — FASTFLOW, a maximum flow demanding a Dinic both correct AND fast. Every Session-1 playbook item gets executed: mapping, time allocation, subtasks, stress tests, and cold decisions in the final minutes. This is the mirror of your national readiness.

**Session goals — after these 2 hours you can:**

(1) Execute the full Session-1 playbook in a full-length simulation

(2) Implement Dinic: level BFS + blocking-flow DFS + current-arc

(3) Handle undirected graphs, duplicate edges, and 64-bit capacities correctly

(4) Self-assess with the champion rubric and draft a post-TC training plan

## 15.1 Dinic in Four Sentences

(1) A BFS from the source gives level[v] = positive-edge distance; stop when the sink is unreachable. (2) The DFS may only advance to level+1 — the 'level graph' that kills cycles. (3) The current-arc iter_[v] remembers which edges already failed, so each edge is examined O(1) times per phase — without it Dinic becomes a TLE. (4) Repeat BFS + blocking flow until the sink is cut off; O(V) phases, far fewer in practice.

- Undirected: both directions carry the full capacity c — the residual pair are each other's reverses.
- Duplicate edges stay as they are (no merging needed); self-loops (u = v) are discarded.
- All flow in long long: $10^9$ capacities × 30,000 edges destroy int (Table 15.1).

## 15.2 The Simulation Rubric

| Aspect | Passing evidence |
|---|---|
| Mapping (minutes 0-10) | A written map: model, bounds, plan |
| Implementation | Full Dinic, clean compile, official sample passes (output 5) |
| Verification | Cross-check vs a reference max-flow on small random graphs |
| Time management | A filled phase log; the 45-minute rule honored |

| Aspect | Passing evidence |
|---|---|
| Verdict | ACCEPTED on SPOJ |

*Table 15.1 — Five aspects, five proofs; champions meet them all.*

**The coach's closing message**

After this TC: one virtual contest a week, one HARD problem a day, and an error journal that never sleeps. A national title is the sum of small disciplines that never take a day off. See you on the podium.

## 15.3 HARD Case Study: FASTFLOW (SPOJ #4110)

**Problem card**

SPOJ FASTFLOW | Graph Algorithms | Hard | Dinic's Algorithm

Source: https://www.spoj.com/problems/FASTFLOW/ — solve and submit directly on SPOJ.

### Problem statement (concise, faithful to the original SPOJ text)

Given an UNDIRECTED graph with N ≤ 5,000 nodes and M ≤ 30,000 capacitated edges (up to $10^9$, duplicates and loops possible). Compute the maximum flow from node 1 to node N. The title is the challenge: it must be FAST — Dinic with level graphs and current-arc.

- Undirected edges = full capacity in both directions of the residual pair; self-loops are discarded.
- Answers can exceed 32-bit — long long everywhere flow lives.

### Official sample input/output

```
Input:
4 6
1 2 3
2 3 4
3 1 2
2 2 5
3 4 3
4 3 3
Output:
5
```

### Phases 1-2 — Understand and Model

Maximum flow 1 → N on an undirected graph: each edge carries full capacity both ways, the pair serving as each other's residual reverses. Dinic — level BFS, blocking-flow DFS, current-arc — meets the title's speed demand.

## Phase 3 — Design the Algorithm

1. add_edge both directions at capacity c; u = v loops dropped; duplicate edges kept.
2. The BFS assigns levels; the DFS only advances to level+1, pruning via the per-node current-arc iterator.
3. On finding flow d: reduce the edge's cap, add to its reverse, return d upward.
4. Repeat phases until the BFS fails to reach the sink; accumulate in long long.

## Phase 4 — Complexity Analysis

Dinic is $O(V^2E)$ on paper but far faster on real graphs; with current-arc, FASTFLOW finishes in hundreds of milliseconds. Without it, the same paper complexity turns into a machine TLE.

## Phase 5 — Implementation (Public Edition — ~50% revealed, ~50% your task)

This is the Public Edition of this solution. Roughly half of the code is shown below; the rest — the part that actually solves the problem — is your task, guided by the hints beneath the code and the walkthrough in Phases 1-4 above.

```
#include <cstdio>
#include <vector>
#include <queue>
#include <cstring>
#include <algorithm>
using namespace std;
const int MX = 5001;
const long long INF = 1e18;
struct Edge {
  int to, rev;
  long long cap;
};
vector<Edge> graph[MX];
int level[MX], iter_[MX], n, m;
void add_edge(int u, int v, long long c) {
  // Undirected: full capacity c in both directions
  graph[u].push_back({v, (int)graph[v].size(), c});
  graph[v].push_back({u, (int)graph[u].size()-1, c});
}
/* ================================================================
   >>> PUBLIC EDITION — roughly half of this solution is hidden for you to complete. <<<
Rebuild it using the hints below the code and the reasoning in Phases 1-4 above.
   ================================================================ */
int main() {
  scanf("%d %d", &n, &m);
  for (int i = 0; i < m; i++) {
    int u, v;
    long long c;
    scanf("%d %d %lld", &u, &v, &c);
    if (u != v) add_edge(u, v, c); // skip self-loops
  }
  printf("%lld\n", max_flow(1, n));
}
```

### Hints for the missing half — enough for a champion, no more

The blocking-flow dfs: only traverse edges with cap > 0 and level[v] < level[e.to]; return the flow that actually reaches t.

Current-arc: the per-node iterator (int& i = iter_[v]) is NOT reset within a phase — this is what makes Dinic fast; reset only after a new BFS.

On d > 0: subtract e.cap, add to the reverse edge graph[e.to][e.rev], return d; the outer phase repeats the dfs until 0, then BFS again. Also now hidden: bfs() itself (standard level-graph BFS — level[s]=0, relax any edge with cap>0 and level[e.to]<0). dfs() (the blocking-flow search) and max_flow() (the outer BFS-then-repeated-dfs driver loop) were already hidden in the original edition and stay hidden here — rebuild both using the hints above.

**Verification log — this code was re-checked before printing**

The full code was recompiled with g++ -O2 -std=c++17 (0 errors), tested against the official SPOJ samples, and stress-tested against a brute-force reference on hundreds of random cases — all identical. Archive status: ACCEPTED on SPOJ.

## 15.4 Simulation Completion Checklist

- [ ] The minutes-0-10 map handed to the coach
- [ ] Dinic complete (half filled independently) + official sample passes
- [ ] Flow cross-checked on ≥ 20 small random graphs
- [ ] FASTFLOW ACCEPTED recorded
- [ ] A written, agreed post-TC training plan

## 15.R Session Summary and Winning Points

- Dinic = level BFS + blocking DFS + current-arc; remove one and the speed is gone.
- Undirected means full capacity both ways; long long everywhere flow travels.
- The simulation is rubric-scored — national readiness is measured, not felt.
- Training does not end at session 15; it merely moves house into your daily schedule.

## References

[1] Cormen, Leiserson, Rivest, Stein — Introduction to Algorithms, 4th ed., MIT Press, 2022.

[2] Halim, S., Halim, F. — Competitive Programming 4, 2020.

[3] SPOJ — Sphere Online Judge, spoj.com (problem FASTFLOW).