

UCF High School Programming Contest Spreadsheet Organization

High School Programming Team Coaches,

I've designed this spreadsheet to aid teams practicing for UCF's high school programming competition.

The spreadsheet has 15 tabs. The very first tab has all of the unique data. It's called ProblemSummaries. This sheet has the following information for each problem that has appeared in UCF's High School Programming Contest (just the main team one, not the online contest):

Year, Problem Name, Filename, Category, Difficulty and Notes

This main sheet is sorted by year and then each problem is in the order that the problems appeared in the set that year.

The filename is simply just what students have to call their file when submitting it during the real contest to our online judge. This is irrelevant for practice.

(More on next page!)

I have rated the difficulties of the problems from 1 to 5. Here is my intuitive meaning for each rating:

Rating	Intuitive meaning
1	These are problems that almost every team should get. They involve some basic computation with numeric input or a very basic task with text.
2	These problems are ones that should be solvable with no specialized knowledge except for being comfortable with standard implementation skills in a student's language of choice. Common high school knowledge is assumed. None of these problems will have overtly complicated implementations if implemented in the straight-forward manner suggested in the problem.
3	These problems may require clever observations and specialized knowledge along with stringing together multiple pieces of information as well as more proficiency in implementation than problems with a rating of 2. Examples of specialized knowledge that maps to this rating is a standard custom sorting problem, use of a built in data structure (set, map) or a 2D geometry problem that requires good recall of high school geometry.
4	These are problems that require algorithmic knowledge that we teach in early courses at UCF such as the permutation algorithm, basic graph algorithms (DFS, BFS, MST, Dijkstra's) or longer problems that have many points of failure where each step has to be accurately implemented. Quite a few geometry problems (things like line segment intersection) by themselves map to this category due to floating point issues. Floodfills/reachability on grids fall into this rating, as well as easier binary searches.
5	These are the hardest problems. These are either advanced topics such as binary index trees, segment trees, dynamic programming (though some DPs are 4s), difficult geometry (more difficult 2D tasks and both 3D problems), or graph problems beyond basic implementation of the standard algorithm (eg. Split Node Dijkstra's). A few problems made it into this category because their implementation was so intricate and long.

Finally, my notes are pretty random; just my thoughts as I glanced at my solution for the problem. Some of it might not make sense to others...**But sometimes the notes give away a portion of the solution idea, which is part of why I have so many other sheets.**

On the following pages I'll talk about the organization of the rest of the 14 sheets included.

Sheet 2 - "Sorted-Difficulty-Category"

This is exactly the first sheet, but sorted by difficulty, then breaking ties by category. Since spreadsheets use stable sorts, within the same difficulty and category, the problems should be sorted by year.

I thought this would be a resource for a student who wanted to work on several problems of a fixed difficulty.

For now, I've left the notes on this sheet.

Sheet 3 - "Sorted-Category-Difficulty"

It's common when learning to work on a single category of problem. Thus, these are sorted first by category. Then, within each category, sorted by difficulty as it naturally makes sense for me to go from easier problems to harder problems. (Or maybe try a 2, and if that was easy move to a 3, etc.) For now, I've left the notes on this sheet.

Sheet 4 - "MinorTopics-NoNotes"

For each topic with less than 8 questions from the set of all contests, I've placed all information for those topics here. This sheet and the next 10 sheets have no notes, so students can look at it without anything big being given away about the problem solution.

There are 10 of them:

Ad Hoc, Binary Index Tree, Binary Search, Built in Data Structures, Constructive, Divide and Conquer, Game Theory, Geometry 3D, Prefix Sums, Segment Tree, Trie

Here is a brief description of each category:

Ad Hoc - Doesn't neatly fit in any one category, but usually requires a clever observation not usually based on information that is taught in standard courses.

Binary Index Tree - An advanced data structure (we don't teach it to UCF undergrads in our regular courses) which allows for $O(\lg n)$ updates to individual data points and $O(\lg n)$ sum queries on ranges.

Binary Search - Some people call this bisection. This is a technique where it's hard to answer the given question directly, but easier to answer the question, "Can we solve the problem with value X?" The yes/no answer indicates if the desired answer is more or less than X. Repeatedly asking these questions can narrow down the answer to the question to a range small enough to get a correct response. (Integer binary searches narrow down to one integer, and real valued ones can get to pretty good precision easily).

Built in Data Structures - Most modern programming languages and for contests, C++, Java and Python have several built in data structures implemented such as stacks, queues, sets, maps (both ordered and unordered), and priority queues. Efficient solutions to questions in this category primarily involve just using the built in tools appropriately. There are other questions that use these (for example graph questions) but if the use of the built in data structure isn't the main part of the solution to the problem then it doesn't fit in this category. Basically, it's important to know that problems in other categories may also use built in data structures.

Constructive - These are where your program has to generate a construction to some sort of puzzle. For example, a constructive problem may ask to output a permutation of integers 1 to n with some quality.

Divide and Conquer - This is strongly associated with recursion, where a problem can be broken into smaller pieces and those pieces can be efficiently used to solve the problem. There's only one of these and it's roughly similar to the "fast modular exponentiation" algorithm.

Game Theory - These are problems usually with one or two people playing and we assume that all players play optimally and we want to know under those assumptions what the outcome of the game will be. (An easy example is the simple stones game: Given a pile of n stones, two players go back and forth. On each turn, they must remove 1, 2 or 3 stones. The winner is the person who removes the last stone. Call the players A and B with A going first. Given n, if both players play optimally, who wins the game? Basically if n is divisible by 4, player B wins and in all other cases, player A wins, for this example.)

Geometry 3D - Self explanatory...a geometry problem in three space with x, y and z coordinates. These are generally always gross.

Prefix Sums - Given a regular set of data like 2, 1, 6, 4, the corresponding prefix sums of the data are 2, 3, 9 and 13. In this new array, index i represents the sum of all values in the old array up to index i. Storing data in this fashion can help solve specific problems faster, such as sum of ranges when the data doesn't change.

Segment Tree - This is a flexible data structure that allows for range updates and range queries of various sorts (most common is sum). Things like, "add 10 to each index of the array in between index 234 and index 22,202" and "what's the sum of all the values from index 1234 of the array and index 12345, inclusive?"

Trie - This is a data structure that stores dictionaries of words efficiently. It stores words in a tree like structure where the number of possible links from a node equals the alphabet size. Each link represents "processing" a letter, so to speak.

Sheet 5 - 2D-Array-NoNotes

This has a list of all of the 2D array problems, sorted by difficulty, then year. There are some problems that require 2D arrays not listed here that made it into other categories for various reasons (mostly graph problems). I deemed that these problems fundamentally focused on the two dimensional grid structure.

Sheet 6 - BruteForce-NoNotes

The most simple brute force is a single for loop through all possible answers to a query. When we teach brute force formally in college, we focus on using recursion to iterate through all combinations or permutations of a search space. Most of the problems in this section specifically represent checking every combination or permutation outlined in a search space. Beyond the fixed algorithms of going through these things, the variation in difficulty depends on the implementation issues in "testing" a particular combination or permutation. In addition, we have several backtrackers in this section, where you have to be smart and skip partial solutions that are doomed to fail.

Sheet 7 - CustomSorting-NoNotes

Custom Sorting is when the primary part of the problem is to take some data that we must usually store in a struct or object and then we have to sort it according to some specific rules. C++, Java and Python all have built in ways to do this, so these questions specifically test students' abilities to implement these language features.

Sheet 8 - DynamicProgramming-NoNotes

Dynamic Programming is a powerful algorithmic technique that is specifically used on problems with optimal substructure (recursive solutions) but where the recursive code makes redundant function calls and runs too long (classic example is Fibonacci). What makes these categories of problems difficult is that you can't just memorize a set of steps (like DFS, BFS, Dijkstra's etc.) The technique is very flexible and can be applied to a wide variety of problems and the code completely changes for each problem based on how to solve the problem by breaking it down to smaller problems of the same nature. Often times, the hardest part of these problems is determining the relevant "state" or information required to solve the problem correctly based on answers for the problem in other smaller "states".

Sheet 9 - Geo2D-NoNotes

This is the category for plane geometry problems (just x and y). I've included problems that require trigonometry functions in this section. This category has problems dealing with areas, lengths and intersections between planar figures (lines, polygons, circles, etc.) and standard techniques involve using vectors and reference angles.

Sheet 10 - Greedy-NoNotes

Greedy Algorithms are algorithms that correctly solve a problem without considering all possibilities. Usually implementing these algorithms tends to be easier than some other types, but the issue is that some problems can NOT be solved by greedy algorithms while others can. New students to problem solving usually attempt greedy solutions to all problems only to get quite a few problems wrong, sometimes because no greedy technique correctly solves a problem and other times because their greedy solution works incorrectly, optimizing the wrong thing. A simple problem with a greedy solution is as follows: Given a list the number of pages in several books, and a number of pages I am willing to read, compute the maximum number of books I could read. It's clear that if I want to maximize the number of books read but am only willing to read a fixed number of pages, I should read the shortest books first. So, we can correctly solve the problem by sorting the input and going in order and determining the value k such that the sum of the first k numbers is less than or equal to my page limit but the sum of the first $k+1$ numbers is greater than it. There are likely problems that should have been in this category that I put in "math". I didn't think of using this category for a quite a while after I started categorizing problems.

Sheet 11 - Graph-NoNotes

Graphs are structures with a collection of points (called vertices) and connections between those points (called edges). Graphs are used to model all sorts of things from traffic networks to friendships. A significant portion of algorithms courses in college are devoted to graphs and algorithms pertaining to graphs. The most common ones are: depth first search, breadth first search, topological sort, Prim's Algorithm, Kruskal's Algorithm, Dijkstra's Algorithm and Floyd-Warshall's algorithm. (Sorry if I've left out your favorite basic graph algorithm!) In some problem statements the formation of the graph is explicit while in others the description doesn't seem to obviously be of dots connected with lines but there's a way to take the information and create a graph that helps solve the problem.

Sheet 12 - Math-NoNotes

This is the largest category. I pretty much put any problem where you take numbers and have to solve something with numbers here. I've placed almost all of the "bangers" (very easy problems) that don't involve strings in this category. Most will find the problems with difficulty 1 aren't really all that much math (read an integer input and print out six times the integer...) But the problems with difficulty 2 or more have some sort of computation that the average person would consider math =) The three problems with difficulty 5 have other components but I decided that the math component was difficult enough to place the problem in math.

Sheet 13 - Simulation-NoNotes

Simulations are problems where the problem statement involves describing a set of steps and then you have to write a program that follows those steps and prints out the outcome of following those steps exactly. These problems usually require recognizing that a program will run fast enough that goes through all of the described steps and then requires the implementation skills necessary to perform/simulate the process described in the problem.

Sheet 14 - Text-NoNotes

Text problems are problems dealing with the manipulation of strings. A very easy text problem is determining if a string equals "hello". One might be surprised, but there can be very difficult text problems. For me, the hardest of ALL 427 problems was the problem Phish Phinder from 2006. Reading the description will prove to you that text problems can be challenging. As you can see from the ratings, the UCF High School Programming Contest has had all varying levels of difficulty for text problems.

Sheet 15 - Set-Distribution

This is the final sheet!

It has a row for every year of the contest.

My columns are as follows:

- Number of problems of each difficulty level
- Total Number of problems
- Number of problems with difficulty 3 or less (represents problems solvable without knowledge beyond typical high school exposure)
- One column for each of the 21 categories (shows how many problems in a single contest were of a particular category)

Quick Summary of the data:

Distribution of problems by difficulty:

Difficulty Rating	Number of Problems
1	57
2	88
3	143
4	91
5	48

This has a pretty cool bell-curve shape - Glenn Martin (main chief judge) and prior judges have done an excellent job of balancing problem difficulty. Kudos to them!

Distribution of problems by category, both for all 40 years and the last 10 years:

Category	Number of Problems (all)	Num Prob (2017-2026)
Math	92	23
Simulation	73	10
Text	73	14
Geometry 2D	45	14
Graph	34	17
2D Array	25	2
Brute Force	18	6
Custom Sorting	13	0
Dynamic Programming	11	9
Greedy	10	6
Built in Data Structures	7	4
Constructive	6	3
Prefix Sums	6	4
Ad Hoc	4	3
Binary Index Trees	3	3
Binary Search	3	3
Game Theory	2	2
Geometry 3D	2	0
Divide and Conquer	1	1
Segment Trees	1	1
Tries	1	1

For the last ten years, the most common topics are the same five as the whole history of the contest: math, simulation, text, 2D geometry and graphs. Of the other topics, it looks like dynamic programming is charging in to be the sixth most frequent topic now over the last 10 years.