Olympiads in Informatics, 2022, Vol. 16, 35–42

DOI: 10.15388/ioi.2022.04

What is the Competitive Programming Curriculum?

Antti LAAKSONEN
Department of Computer Science, University of Helsinki
e-mail: ahslaaks@cs.helsinki.fi

Abstract. Competitive programmers learn algorithms and data structures that belong to university computer science curricula. In this paper we go through the "Algorithms and Complexity" knowledge area in the ACM/IEEE curriculum guidelines and determine which of the topics can be learned through competitive programming. After that, we discuss in detail some topics that are different in competitive programming and university courses.

Keywords: data structures, algorithms, curricula, programming contests.

1. Introduction

Solving competitive programming problems can teach many important algorithms and data structures discussed in university algorithms courses. However, there are also differences: some topics are usually only covered either in competitive programming or university courses, but not in both of them. For example, segment trees are used primarily in competitive programming, while Fibonacci heaps are typically only seen in university courses.

There are no clear lists of topics that appear in competitive programming or university courses. The IOI Syllabus (2020) roughly specifies the topics that can be expected in the International Olympiad in Informatics, but there are many topics that are not included in the IOI Syllabus but still appear in other contests, such as the International Collegiate Programming Contest or Google Code Jam. The ACM/IEEE curriculum guidelines (2013) suggest topics that should be included in computer science curricula in universities.

This paper consists of two parts. In the first part, we go through the topics of the "Algorithms and Complexity" knowledge area in the ACM/IEEE curriculum guidelines and determine which topics can be learned through competitive programming. In the second part, we discuss in detail some differences between competitive programming and university textbook topics.

2. Comparison to ACM/IEEE Curriculum Guidelines

The purpose of the ACM/IEEE curriculum guidelines is to recommend topics that should be included in university computer science curricula. The topics have been divided into three categories as follows:

- Core-Tier1 topics are the most fundamental topics and a computer science curriculum should cover them all.
- Core-Tier2 topics are also important and a computer science curriculum should cover all or almost all of them.
- Elective topics are more advanced topics, and a computer science curriculum should also cover many of them.

In this section we go through the topics of the "Algorithms and Complexity" knowledge area in the guidelines. For each topic, we determine if it can be typically learned through competitive programming, i.e., by learning techniques that are needed in programming contests.

2.1. *Basic Analysis*

The challenge in most competitive programming problems is to create efficient algorithms. While competitive programmers routinely work with time complexities and use the Big O notation, not all topics in this group are covered in typical competitive programming training.

Interestingly, the formal definition of the Big O notation has been included in the Core-Tier1 category in the guidelines, while its use belongs to Core-Tier2. However, in competitive programming, it is essential to be able to use the Big O notation when designing algorithms, but it is not necessary to know its formal definition.

Category	Topic	In Contests?
Core-Tier1	Differences among best, expected, and worst case behaviors of an algorithm	Yes
Core-Tier1	Asymptotic analysis of upper and expected complexity bounds	Yes
Core-Tier1	Big O notation: formal definition	No
Core-Tier1	Complexity classes, such as constant, logarithmic, linear, quadratic, and exponential	Yes
Core-Tier1	Empirical measurements of performance	Yes
Core-Tier1	Time and space trade-offs in algorithms	Yes
Core-Tier2	Big O notation: use	Yes
Core-Tier2	Little o, big omega and big theta notation	No
Core-Tier2	Recurrence relations	Yes
Core-Tier2	Analysis of iterative and recursive algorithms	Yes
Core-Tier2	Some version of a Master Theorem	No

2.2. *Algorithmic Strategies*

Category	Topic	In Contests?
Core-Tier1	Brute-force algorithms	Yes
Core-Tier1	Greedy algorithms	Yes
Core-Tier1	Divide-and-conquer	Yes
Core-Tier1	Recursive backtracking	Yes
Core-Tier1	Dynamic programming	Yes
Core-Tier2	Branch-and-bound	No
Core-Tier2	Heuristics	Yes
Core-Tier2	Reduction: transform-and-conquer	Yes

Most of the topics in this group belong to fundamental competitive programming skills. The only exception is the branch-and-bound technique which can be regarded as an advanced technique rarely seen in programming contests. The branch-and-bound technique is used to optimize exhaustive search algorithms, while most competitive programming problems deal with polynomial algorithms. In programming contests, it is often possible to get partial points by implementing a brute force algorithm, but it is not needed to optimize the algorithm.

2.3. *Fundamental Data Structures and Algorithms*

In competitive programming it is important to know how to use efficient algorithms and data structures available in the standard library of the used programming language.

Category	Topic	In Contests?
Core-Tier1	Simple numerical algorithms, such as computing the average of a list of numbers, finding the min, max, and mode in a list, approximating the square root of a number, or finding the greatest common divisor	Yes
Core-Tier1	Sequential and binary search algorithms	Yes
Core-Tier1	Worst case quadratic sorting algorithms (selection, insertion)	Yes
Core-Tier1	Worst or average case O(N log N) sorting algorithms (quicksort, heapsort, mergesort)	Partially
Core-Tier1	Hash tables, including strategies for avoiding and resolving collisions	Partially
Core-Tier1	Binary search trees • Common operations on binary search trees such as select min, max, insert, delete, iterate over tree	Partially
Core-Tier1	Graphs and graph algorithms • Representations of graphs (e.g., adjacency list, adjacency matrix) • Depth- and breadth-first traversals	Yes
Core-Tier2	Heaps	Partially
Core-Tier2	Graphs and graph algorithms • Shortest-path algorithms (Dijkstra's and Floyd's algorithms) • Minimum spanning tree (Prim's and Kruskal's algorithms)	Yes
Core-Tier2	Pattern matching and string/text algorithms (e.g., substring matching, regular expression matching, longest common subsequence algorithms)	Yes

For example, in C++, the function `sort` can be used to efficiently sort an array, the class `unordered_map` implements a hash table, and the class `priority_queue` implements a heap. However, it is not necessary to know how these algorithms and data structures actually work.

This is a fundamental difference between competitive programming and university courses: you can be a successful competitive programmer without knowing, for example, the quicksort algorithm, which is a basic algorithm taught in almost any introductory university course. Such knowledge is not needed in competitive programming because you can use the standard library which provides efficient sorting algorithms. However, you should know how to implement algorithms and data structures that are *not* in the standard library, such as the union-find data structure needed for Kruskal's algorithm.

Note that Prim's and Kruskal's algorithms accomplish the same task, and most competitive programmers seem to prefer Kruskal's algorithm which can be extended to some more advanced problems. While it is interesting to know two different algorithms for determining minimum spanning trees, there is not much use for Prim's algorithm in programming contests.

2.4. *Basic Automata Computability and Complexity*

The topics of this group are outside the scope of competitive programming.

2.5. *Advanced Computational Complexity*

The topics of this group are outside the scope of competitive programming.

2.6. *Advanced Automata Theory and Computability*

The topics of this group are outside the scope of competitive programming.

2.7. *Advanced Data Structures Algorithms and Analysis*

This group has both topics that are regarded as basic topics in competitive programming, such as topological sorting, and advanced topics that are only needed in some difficult problems, such as linear programming.

Many balanced trees, such as AVL trees and red-black trees, are difficult to implement, and in many cases one can just use standard library implementations, such as the classes `map` and `set` in C++. However, balanced trees may be needed in some advanced competitive programming problems where it is required to, for example, split and merge arrays. One popular way to solve such problems is to use the treap data structure whose implementation is relatively easy (if you know a good way to implement it).

Category	Topic	In Contests?
Elective	Balanced trees (e.g., AVL trees, red-black trees, splay trees, treaps)	Yes
Elective	Graphs (e.g., topological sort, finding strongly connected components, matching)	Yes
Elective	Advanced data structures (e.g., B-trees, Fibonacci heaps)	No
Elective	String-based data structures and algorithms (e.g., suffix arrays, suffix trees, tries)	Yes
Elective	Network flows (e.g., max flow [Ford-Fulkerson algorithm], max flow – min cut, maximum bipartite matching)	Yes
Elective	Linear Programming (e.g., duality, simplex method, interior point algorithms)	Partially
Elective	Number-theoretic algorithms (e.g., modular arithmetic, primality testing, integer factorization)	Partially
Elective	Geometric algorithms (e.g., points, line segments, polygons. [properties, intersections], finding convex hull, spatial decomposition, collision detection, geometric search/proximity)	Yes
Elective	Randomized algorithms	Yes
Elective	Stochastic algorithms	Yes
Elective	Approximation algorithms	Yes
Elective	Amortized analysis	Yes
Elective	Probabilistic analysis	Yes
Elective	Online algorithms and competitive analysis	No

It is not clear which data structures exactly belong to "advanced data structures". At least the two mentioned data structures, B-trees and Fibonacci heaps, are not necessary in competitive programming because you can use other data structures instead of them.

3. Competitive Programming vs. University Courses

This section discusses in detail some topics that are different in competitive programming and university courses. In general, competitive programmers prefer techniques that are easy to implement, and try to use algorithms and data structures provided in standard libraries of programming languages.

3.1. *Range Queries*

Range query structures play an important role in competitive programming. For example, using a segment tree (see e.g. Laaksonen, 2020, Section 9.2.2) it is possible to maintain an array of n elements and process two types of queries in O(log n) time: (1) modify an array value, (2) find the maximum value in a given range (subarray).

For some reason, simple range query structures, such as segment trees, are rarely discussed outside competitive programming. Instead, other methods are used to solve problems. For example, Cormen *et al.* (2009, Chapter 14) shows how modified red-black trees can be used to create dynamic tree structures. This approach yields O(log n) operations like segment trees, but it would be very difficult to implement red-black trees during a contest.

There are some popular problems that can be solved using range queries, but are usually solved using another method outside competitive programming. For example, a typical way to count the number of inversions in a permutation in O(n log n) time is to use a modified mergesort algorithm (see e.g. Cormen *et al.*, 2009, p. 42). However, the problem can also be solved using a segment tree that allows us to go through the permutation from left to right and efficiently count the number of previous elements that are larger than the current element.

3.2. *Hashing*

Hash tables are often used in competitive programming as standard library data structures. For example, the C++ classes `unordered_map` and `unordered_set` are based on hash tables. While it is not necessary to implement hash tables and resolve collisions, it is important to understand that hash table data structures may be slow on some inputs.

Some contest systems, such as Codeforces, allow users to send additional inputs (called "hacks") to contest problems. If a solution is based on hashing, it may be possible to hack it by constructing an input where a large number of elements is assigned the same hash value. While hash table operations usually take O(1) time, in this case they can take O(n) time. For example, it is possible to construct inputs where the C++ classes `unordered_map` and `unordered_set` are too slow if they are used in a typical way. We have observed that many students assume that hash tables are always efficient in practice, and it is instructive to see that there are indeed inputs where they are not efficient.

Another topic where hashing is used in competitive programming are string algorithms. Like in the Karp-Rabin pattern matching algorithm (1989), we can compare substrings of strings in O(1) time after preprocessing the strings using the hash values of the substrings (see e.g. Laaksonen, 2020, Section 14.2). A speciality in competitive programming is that it is often assumed that there are no collisions, i.e., if two substrings have the same hash value, they also have the same content. When hashing is properly implemented (Pachocki and Radoszewski, 2013), many string problems can be solved using the technique.

3.3. *Binary Search*

A traditional way to use binary search is to efficiently search for values in a sorted array in O(log n) time. In competitive programming binary search is not often used in that way, because we can either use a standard library implementation of binary search (such as the `lower_bound` and `upper_bound` functions in C++) or we can use an efficient data structure that is available in the standard library.

Instead, binary search is often used as an algorithm design technique: when we know that a function $f(x)$ has value 0 when $x < k$ and value 1 when $x \geq k$, we can efficiently

find the smallest x value such that $f(x) = 1$ using binary search (see e.g. Laaksonen, 2020, Section 4.3.2). Surprisingly, this way to use binary search is not often discussed in algorithms textbooks. In some cases the reason may be that there is another way to solve the problem without using binary search.

3.4. *Dijkstra's Algorithm*

The usual way to implement Dijkstra's algorithm in textbooks (see e.g. Cormen *et al.*, 2009, Section 24.3) is to build a heap that contains a distance to each node of the graph. Initially each distance is infinite, and the distances are updated during the algorithm using the decrease-key heap operation. This implementation works in $O(m \log n)$ time where n and m represent the number of nodes and edges, assuming each element is reachable from the starting node.

The problem in such an implementation is that heap implementations in standard libraries (such as the `priority_queue` class in C++) typically don't support the decrease-key operation. For this reason, it would be necessary to implement a custom heap instead of using the standard library data structure. However, in competitive programming, another version of Dijkstra's algorithm is used (see e.g. Laaksonen, 2020, Section 7.3.2) which doesn't update the distances in the heap but instead adds a new distance to the heap when a distance changes. This allows us to use a standard library heap implementation in the algorithm. It can be shown that this version of Dijkstra's algorithm also works in $O(m \log n)$ time even if the number of elements in the heap may be larger than in the textbook implementation.

This is an example of a tendency that can be seen in competitive programming: new ways are invented to use standard library algorithms and data structures whenever possible, which makes implementations shorter and saves time during contests.

3.5. *Divide-and-conquer*

The divide-and-conquer technique is often regarded as a basic algorithm design technique. For example, Kleinberg and Tardos (2006) devote an entire chapter to the technique, and discuss algorithms such as mergesort, finding the closest pair of points and the FFT algorithm. However, using the divide-and-conquer technique is rarely required in competitive programming problems.

It seems that the divide-and-conquer technique is often used indirectly in competitive programming problems. While mergesort is an important algorithm, it is not necessary to implement it because we can use the standard library implementation, such as the `sort` function in C++, which may use mergesort. The FFT algorithm is required in some advanced competitive programming problems, but it is often used as a prewritten black box algorithm.

There is another way to solve the closest pair of points problem that differs from the traditional divide-and-conquer algorithm which divides the points into two sets and

recursively solves the problem for each group and then combines the results. Instead, we can process the points from left to right and use a balanced binary tree to maintain a set of relevant points (see e.g. Laaksonen, 2020, Section 13.2.2). In this implementation, we can think that the divide-and-conquer idea is hidden in the balanced binary tree and it can't be seen in the main algorithm.

4. Conclusion

Competitive programming covers many, but not all, of the algorithm design and data structures topics in the "Algorithms and Complexity" knowledge area in the ACM/IEEE curriculum guidelines. There are also topics that are different in competitive programming and university courses, and there are competitive programming approaches that rarely appear in textbooks.

Some theoretical topics are only rarely needed in competitive programming. However, an interesting question for future work is what competitive programmers really know besides the topics relevant in programming contests. For example, do they usually know how the heap data structure works, even if it is not necessary to implement a heap during a contest?

References

ACM/IEEE (2013). Curriculum Guidelines for Undergraduate Programs in Computer Science. Available online at: `https://www.acm.org/binaries/content/assets/education/cs2013_web_final.pdf`

Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C. (2009). *Introduction to Algorithms* (Third Edition). MIT Press.

IOI Syllabus (2020). Available online at: `https://ioinformatics.org/files/ioi-syllabus-2020.pdf`

Karp, R.M., Rabin, M.O. (1989). Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2), 249–260.

Kleinberg, J., Tardos, E. (2006). *Algorithm Design*. Addison–Wesley.

Laaksonen, A. (2017). A Competitive programming approach to a University introductory algorithms course. *Olympiads in Informatics*, 11, 87–92.

Laaksonen, A. (2020). *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests* (Second Edition). Springer.

Pachocki, J., Radoszewski, J. (2013). Where to use and how not to use polynomial string hashing. *Olympiads in Informatics*, 7, 90–100.

A. Laaksonen works as a university lecturer at the Department of Computer Science of the University of Helsinki. He is one of the organizers of the Finnish Olympiad in Informatics and has written a book on competitive programming. He is also a developer of the CSES online judge.