**Math & Logic**

You’ll need to know some mathematical concepts from several different areas if you want to excel at algorithms. Learn about set theory, finite-state machines, regular expressions, matrix multiplication, bitwise operations, solving linear equations, important combinatorics concepts such as permutations, combinations, pigeonhole principle.

<https://medium.com/coderbyte/how-to-get-good-at-algorithms-data-structures-d33d5163353f>

#### Big-O & Runtime

* Learn what [Big-O](https://en.wikipedia.org/wiki/Big_O_notation) is and how to analyze the [running times](https://cs.stackexchange.com/questions/192/how-to-come-up-with-the-runtime-of-algorithms) of algorithms. This is a [classic book](https://en.wikipedia.org/wiki/Introduction_to_Algorithms) on the topic (here is the chapter on the [growth of functions](http://www.cs.dartmouth.edu/~ac/Teach/CS19-Winter06/SlidesAndNotes/CLRS-3.1.pdf)).
* [Algorithms: Divide and Conquer, Sorting and Searching, and Randomized Algorithms](https://www.coursera.org/learn/algorithms-divide-conquer) - The primary topics are: asymptotic ("Big-oh") notation, sorting and searching, divide and conquer, and randomized algorithms.
* [Algorithms: Graph Search, Shortest Paths, and Data Structures](https://www.coursera.org/learn/algorithms-graphs-data-structures) - The primary topics are: data structures, graph primitives, and their applications.
* [Algorithms: Greedy Algorithms, Minimum Spanning Trees, and Dynamic Programming](https://www.coursera.org/learn/algorithms-greedy) - The primary topics are: greedy algorithms and dynamic programming.
* [Algorithms: Shortest Paths Revisited, NP-Complete Problems and What To Do About Them](https://www.coursera.org/learn/algorithms-npcomplete) - The primary topics are: shortest paths, NP-completeness and what it means for the algorithm designer, and strategies for coping with computationally intractable problems.
* [Algorithms, Part I](https://www.coursera.org/learn/algorithms-part1/home/welcome) - This course covers the essential information that every serious programmer needs to know about algorithms and data structures.Part I covers elementary data structures, sorting, and searching algorithms.
* [Algorithms, Part II](https://www.coursera.org/learn/algorithms-part2) - Part II focuses on graph- and string-processing algorithms.
* [Khan Academy Algorithms](https://www.khanacademy.org/computing/computer-science/algorithms) - Algorithm course ministred by Tomas Cormen and Devin Balkcom.
* [MIT - 6-006](http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/) - Well explained algorithms.
* [MIT - 6-046j](http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/video-lectures/) - Similar to the previous one, but with different algorithms.
* [MIT - 6-00sc](http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-00sc-introduction-to-computer-science-and-programming-spring-2011/index.htm) - An easy and well explained introduction to algorithms.
* [Udacity Intro to Algorithms](https://www.udacity.com/course/intro-to-algorithms--cs215) - Python-based Algorithms course.
* [Algorithms in Motion](https://www.manning.com/livevideo/algorithms-in-motion) - Beginner's algorithms course with fun illustrations, based on the book Grokking Algorithms

### **To Implement**

* Binary search
* Euclid’s algorithm
* Depth and breadth first search
* Dijkstra’s shortest path
* Binary tree traversals
* Insertion sort, Mergesort, Quicksort
* Min & max heaps
* [More](https://www.quora.com/Which-are-the-10-algorithms-every-computer-science-student-must-implement-at-least-once-in-life) [examples](https://hbfs.wordpress.com/2008/12/23/the-10-classes-of-algorithms-every-programmer-must-know-about/) and [lists](https://softwareengineering.stackexchange.com/questions/155639/which-algorithms-data-structures-should-i-recognize-and-know-by-name).

1. **Lists, Arrays, Stack**. [Lists](http://en.wikipedia.org/wiki/Linked_list), [arrays](http://en.wikipedia.org/wiki/Array), and [stacks](http://en.wikipedia.org/wiki/Stack_(data_structure)) are certainly the most basic data structures, yet, these building blocks can reserve a few surprises. For example, counter-intuitive [sentinel nodes](http://en.wikipedia.org/wiki/Linked_list) are extensively used in the [STL](http://en.wikipedia.org/wiki/Standard_Template_Library) to represent the position just passed the last element in a list. This makes, for example, the insertion of an item in the end() position rather efficient. While lists do not allow direct access to an element, arrays do, but at the cost of allocating a known a priori amount of memory. Conversely, inserting an element into a list, given the current position, is constant time, while to insert a new element in an array is more costly, depending on whether or not one wants to preserve ordering. Stacks are a special case of both, one could say, because they behave mostly like lists where operations are allowed only at one end; and like arrays because they are most often implemented in a contiguous span of memory, i.e., an array. As these three are common building blocks of more complex algorithms and data structures, you should really master them.
2. **Trees**. Trees, or more exactly [search trees](http://en.wikipedia.org/wiki/Binary_search_tree) whether they be [binary](http://en.wikipedia.org/wiki/Binary_tree), [k-ary](http://en.wikipedia.org/wiki/K-ary_tree), [splay](http://en.wikipedia.org/wiki/Splay_tree), [AVL](http://en.wikipedia.org/wiki/AVL_tree) or [B](http://en.wikipedia.org/wiki/B-tree), are the next data structures on the list, offering O(\log n) operations for searching, inserting, and removing values (in the average case). The most interesting tree varieties try to insure that they also have worse cases in O(\log n), that is, they don’t go overly deep or take inordinately long to perform their operations. To do so, the prevailing strategy is to balance the tree so that it is of (almost) equal depth everywhere. Some varieties make the tree even shallower by having a branching factor much higher than two, like it is the case with [[(a,b)](http://en.wikipedia.org/wiki/(a,b)_tree) trees](http://en.wikipedia.org/wiki/(a,b)_tree) (also known sometimes as (n,m) trees). Splay trees, on the other hand, unbalances the trees to shift the most oft-accessed item near the root of the tree, while remaining a search tree.

Understanding these data structures and what trade-offs are involved will help you choose wisely which will suit your application better. Note that the structures that equalize depth do not only make sure that the worst case is the average case: they implicitly suppose that all items in the collection have an equal probability of being accessed—something quite contrary to the [Pareto Principle](http://en.wikipedia.org/wiki/Pareto_principle) (or more accurately, maybe, to [Zipf’s law](http://en.wikipedia.org/wiki/Zipf's_law" \t "_blank)).

1. **Sorting and Searching**. Data does not always come in the right order, so it is possible that you will have to sort before applying any meaningful algorithm to it. [Sorting](http://en.wikipedia.org/wiki/Sorting_algorithm) is a complex topic, but I think that the two algorithms you really should know about are [QuickSort](http://en.wikipedia.org/wiki/Quicksort" \t "_blank) and [Radix Sort](http://en.wikipedia.org/wiki/Radix_sort).

QuickSort is a comparison-based sort, that is, elements are compared to each other to determine their relative order. Eventually, QuickSort makes enough comparisons (O(n \log n)for n items, on average, but has a worst case of O(n^2)—which can be avoided, but that’s another story for now) to eventually produce a sorted list of items. The theoretical lower bound for comparison-based sorts is O(n \log n) comparisons, so QuickSort is doing very well on average. The great simplicity of the algorithm makes it very interesting in a very wide range of cases.

Radix sort, on the other hand, an address transformation sort that sorts the items in linear time, that is, in O(n), making it much faster than QuickSort. It is much faster, but Radix Sort is much simpler when keys are numeric or of fixed width; dealing with variable length keys efficiently makes the algorithm slightly more complex. Read more on Radix Sort [here](http://www.nist.gov/dads/HTML/radixsort.html) and in this old [DDJ article](http://www.ddj.com/architect/184405054).

Searching on a sorted array can be performed using the basic [binary search](http://en.wikipedia.org/wiki/Binary_search) in O(\log n) time. Creating and searching efficient [indexes](http://en.wikipedia.org/wiki/Index_(information_technology)) will also play a major role in managing and searching large data sets.

1. **Priority Queues**. Sometimes you don’t really care if a data set is completely sorted or not. You just want to determine rapidly its maximum (or minimum) valued item and remove it from the set. Ideally, determining the maximum item should be an O(1)operation, adding and removing values a O(\log n) operation. Turns out that [heaps](http://en.wikipedia.org/wiki/Heap_(data_structure)) as implementation of priority queues are pretty efficient. Not only do they have efficient algorithms, they can also be implemented into contiguous arrays, dispensing one from using a pointer-based data structure, saving potentially a lot of memory. There are a lot of application of heaps, which range from [scheduling](http://en.wikipedia.org/wiki/Scheduling_algorithm) (determining who goes next) to [cache management](http://en.wikipedia.org/wiki/Cache) (removing the oldest items from the cache).
2. **Pattern Matching and Parsing**. Every single one of us had to write a parser or a filter of some sort. Whether it’s to find data in an insanely large log or to read a simple configuration file. Not always, but very often, the sanest way to go around it is to use [regular expressions](http://en.wikipedia.org/wiki/Regular_expression), either using a library, such as [Boost](http://www.boost.org/)‘s [regex](http://www.boost.org/doc/libs/1_37_0/libs/regex/doc/html/index.html), or [Python](http://www.python.org/)‘s [re](http://docs.python.org/library/re.html), or by implementing the regular expression algorithms yourself—which would make it the insane way, I guess. Understanding how they work will greatly help you process and filter insufficiently structured data. Don’t ask why [Perl](http://www.perl.org/) is so popular.
3. **Hashing**. [Hash functions](http://en.wikipedia.org/wiki/Hash_function) play a central role in [cryptography](http://en.wikipedia.org/wiki/Cryptographic_hash_function), [error detection](http://en.wikipedia.org/wiki/Error_correcting_codes), cache management, and [efficient look-up](http://en.wikipedia.org/wiki/Hash_table). Despite the superficial differences—the applications—all hash functions are closely related; in fact, they differ more in quality than nature. A hash function takes a message and deterministically produces a signature that looks like a pseudo-random number, n bits long, usually with n rather large, say 128 or 256. The harder it is to correlate the hash value with the original data, the safer the hash function is. If it is really hard to find the original data given a hash value, the function can serve as a [one way function](http://en.wikipedia.org/wiki/One_way_function) and be used in cryptographic applications. If the hash function is just good enough, it can serve as an (most probably) unique identifier for data items, which in turn can be used for cache management or hash tables. In either case, you have to understand the mathematics of the probability of collisions (two or more items hashing to the same value) so you can assess what hashing scheme is sufficient for your application. Read about von Mises’ [birthday paradox](http://en.wikipedia.org/wiki/Birthday_paradox) to understand how to compute the probability of collision.
4. **Disjoint Sets**. The [Disjoint sets](http://en.wikipedia.org/wiki/Disjoint-set_data_structure) data structure is a helper structure that you will need in a wide variety of algorithms, whether graph algorithms or image processing. Disjoint sets serves to represent multiple sets within a single array, each item being the member of one of many sets. Considered like that, this sounds like a rather special case, but in fact, there are a number of applications, not all obvious. [Connected components](http://en.wikipedia.org/wiki/Connected_component_(graph_theory)) arise in graph algorithms are most efficiently represented using the disjoint set data structure. It can also serve to segment an image, a process also know as computing the connected components or [labeling](http://en.wikipedia.org/wiki/Connected_component_labeling). What’s also very interesting about disjoint sets is just how incredibly efficient operation are: using path compression, operations can be performed in expected constant time!

The special case where there are only two possible sets reverts to a simple [bitmap](http://en.wikipedia.org/wiki/Bit_array), a data structure that is simple to manage and requires comparatively little memory—use of which can be further reduced, with a price, using [Bloom filters](http://en.wikipedia.org/wiki/Bloom_filter).

1. **Graph Algorithms and Data Structures**. Problems like computing the [minimum spanning tree](http://en.wikipedia.org/wiki/Minimum_spanning_tree), the determination of the [shortest path](http://en.wikipedia.org/wiki/Shortest_path_problem) between two nodes, [matching](http://en.wikipedia.org/wiki/Matching), and the detection of [cut vertexes](http://en.wikipedia.org/wiki/Cut_vertex) will arise in a number of situation. Google’s [PageRank](http://en.wikipedia.org/wiki/Pagerank), for example, is a very concrete application of graph algorithms. Often, seemingly unrelated problems can be mapped to graph algorithms for which very efficient algorithms, possibly dynamic programming related, already exist. There is also a large body of literature devoted to the data structures used for graphs, considering every possible special case: [sparse](http://www.nist.gov/dads/HTML/sparsegraph.html), [dense](http://en.wikipedia.org/wiki/Dense_graph), [clique](http://en.wikipedia.org/wiki/Clique_(graph_theory))-rich, or [small world](http://en.wikipedia.org/wiki/Small_world_networks) networks, etc.
2. **Dynamic Programming**. Closely related to graph algorithms, [dynamic programming](http://en.wikipedia.org/wiki/Dynamic_programming) exploit the fact that the optimal solution to a large problem can be expressed as an optimal combination of sub-problems. Not all problems are amenable to this method, because not every objective function abide to the [principle of optimality](http://en.wikipedia.org/wiki/Bellman_equation), but many optimization problems do. Dynamic programming exchanges memory for computation, a generally beneficent trade-off.

[Memoization](http://en.wikipedia.org/wiki/Memoization) could be considered a limited form of dynamic programming where previous evaluations of an expensive function are cached and reused rather than recomputed if asked for again. Memoization greatly reduce computational complexity when used in combination with an optimization (or simply recursive) algorithm. For example, the n-th [Fibonacci number](http://en.wikipedia.org/wiki/Fibonacci_sequence) can be computed in O(n) time using memoization, while the basic recursive formulation results in no less than O(F_n) \sim \phi^n calls! Here, \phi is [Phidias](http://en.wikipedia.org/wiki/Phidias)‘s number, the [golden ratio](http://en.wikipedia.org/wiki/Golden_ratio).

1. **State Space Search Algorithms**. Sometimes the scale of the problem, or the vastness of the state space makes it impossible to represent the problem as a graph. Consider [chess](http://en.wikipedia.org/wiki/Chess) as the example par excellence. The exact number of distinct valid states of the game is still debated, but is thought to be in the order of 10^{43}, and the number of arcs in the game graph somewhere around 10^{120}, as computed [Shannon](http://en.wikipedia.org/wiki/Claude_Shannon), therefore known as [Shannon’s Number](http://en.wikipedia.org/wiki/Shannon_number). It would be infeasible to search the graph corresponding to all possible games to determine the optimal move given any valid chess position. To deal with this immense state space, one would use one of the many state space search algorithms such as [limited depth search](http://en.wikipedia.org/wiki/Depth-limited_search), [Breadth-first search](http://en.wikipedia.org/wiki/Breadth-first_search), or if enough is known about the objective function, an algorithm like [A\*](http://en.wikipedia.org/wiki/A*_search_algorithm).

State space search is very often used for game [artificial intelligence](http://en.wikipedia.org/wiki/Artificial_intelligence) and other optimization problem where the structure is too complex, state space too vast, or of which too little is known to derive a more efficient algorithm.

[Genetic algorithms](http://en.wikipedia.org/wiki/Genetic_algorithm) are closely related to [stochastic optimisation](http://en.wikipedia.org/wiki/Stochastic_optimization), where the state space is searched at many different points in parallel. The darwinist metaphor is apt, to a certain point, as “genes” (vectors) are “mutated” (varied randomly) to search the state space. Only the fittest (higher valued vectors, under the objective function) are kept for the next “generation” (iteration) others “die off” (are dropped).

data compression

[control theory](http://en.wikipedia.org/wiki/Control_theory)

https://hbfs.wordpress.com/about-the-author/