Click to prove you're human

Good knowledge of standard algorithms is equally important as choosing the right data structure. The following is a list of the top 25 algorithms every programmer and computer science student should know. Also See: Top 50 Classic Data Structures Problems Thanks for reading. To share your code in the comments, please use our online compiler that supports C, C++, Java, Python, JavaScript, C#, PHP, and many more popular programming languages. Like us? Refer us to your friends and support our growth. Happy coding :) Data Structures & Algorithms (DSA) is often considered to be an intimidating topic - a common misbelief. Forming the foundation of the most innovative concepts in tech, they are essential in both jobs/internships applicants' and experienced programmers' journey. Mastering DSA implies that you are able to use your computational and algorithmic thinking in order to solve never-before-seen problems and contribute to any tech company's value (including your own!). By understanding them, you can improve the maintainability, extensibility and efficiency of your code. These being said, I've decided to centralize all the DSA threads that I have been posting on Twitter during my #100DaysOfCode challenge. This article is aiming to make DSA not look as intimidating as it is believed to be. It includes the 15 most useful data structures and the 15 most important algorithms that can help you ace your interviews and improve your competitive programming skills. Each chapter includes useful links with additional information and practice problems. DS topics are accompanied by a graphic representation and key information. Every algorithm is implemented into a continuously updating Github repo. At the time of writing, it contains the pseudocode, C++, Python and Java (still in progress) implementations of each mentioned algorithm (and not only). This repository is expanding thanks to other talented and passionate developers that are contributing to it by adding new algorithms and new programming languages implementations. Contents I. Data Structures Arrays Linked Lists Stacks Queues Maps & Hash Tables Graphs Trees Binary Trees & Binary Search Trees Self-balancing Trees (AVL Trees, Red-Black Trees, Splay Trees) Heaps Tries Segment Trees Fenwick Trees Disjoint Set Union Minimum Spanning Trees II. Algorithms Divide and Conquer Sorting Algorithms (Bubble Sort, Counting Sort, Quick Sort, Merge Sort, Radix Sort) Searching Algorithms (Linear Search, Binary Search) Sieve of Eratosthenes Knuth-Morris-Pratt Algorithm Greedy I (Maximum number of non-overlapping intervals on an axis) Greedy II (Fractional Knapsack Problem) Dynamic Programming I (0–1 Knapsack Problem) Dynamic Programming II (Longest Common Subsequence) Dynamic Programming III (Longest Increasing Subsequence) Convex Hull Graph Traversals (Breadth-First Search, Depth-First Search) Floyd-Warshall / Roy-Floyd Algorithm Dijkstra's Algorithm & Bellman-Ford Algorithm Topological Sorting I. Data Structures 1. Arrays Arrays are the simplest and most common data structures. They are characterised by the facile access of elements by index (position). What are they used for? Imagine having a theater chair row. Each chair has assigned a position (from left to right), therefore every spectator will have assigned the number from the chair (s)he will be sitting on. This is an array. Expand the problem to the whole theater (rows and columns of chairs) and you will have a 2D array (matrix)! Properties elements' values are placed in order and accessed by their index from 0 to the length of the array-1; an array is a continuous block of memory; they are usually made of elements of the same type (it depends on the programming language); access and addition of elements are fast; search and deletion are not done in O(1). Useful Links 2. Linked Lists Linked lists are linear data structures, just like arrays. The main difference between linked lists and arrays is that elements of a linked list are not stored at contiguous memory locations. It is composed of nodes - entities that store the current element's value and an address reference to the next element. That way, elements are linked by pointers. What are they used for? One relevant application of linked lists is the implementation of the previous and the next page of a browser. A double linked list is the perfect data structure to store the pages displayed by a user's search. Properties they come in three types: singly, doubly and circular; elements are NOT stored in a contiguous block of memory; perfect for an excellent memory management (using pointers implies dynamic memory usage); insertion and deletion are fast; accessing and searching elements are done in linear time. Useful Links 3. Stacks A stack is an abstract data type that formalizes the concept of restricted access collection. The restriction follows the rule LIFO (Last In, First Out). Therefore, the last element added in the stack is the first element you remove from it. Stacks can be implemented using arrays or linked lists. What are they used for? The most common real-life example uses plates placed one over another in the canteen. The plate which is at the top is the first to be removed. The plate placed at the bottommost is the one that remains in the stack for the longest period of time. One situation when stacks are the most useful is when you need to obtain the reverse order of given elements. Just push them all in a stack and then pop them. Another interesting application is the Valid Parentheses Problem. Given a string of parantheses, you can check that they are matched using a stack. Properties you can only access the last element at one time (the one at the top); one disadvantage is that once you pop elements from the top in order to access other elements, their values will be lost from the stack's memory; access of other elements is done in linear time; any other operation is in O(1). Useful Links 4. Queues A queue is another data type from the restricted access collection, just like the previously discussed stack. The main difference is that the queue is organised after the FIFO (First In, First Out) model: the first inserted element in the queue is the first element to be removed. Queues can be implemented using a fixed length array, a circular array or a linked list. What are they used for? The best use of this abstract data type (ADT) is, of course, the simulation of a real life queue. For example, in a call center application, a queue is used for saving the clients that are waiting to receive help from the consultant - these clients should get help in the order they called. One special and very important type of queue is the priority queue. The elements are introduced in the queue based on a "priority" associated with them: the element with the highest priority is the first introduced in the queue. This ADT is essential in many Graph Algorithms (Dijkstra's Algorithm, BFS, Prim's Algorithm, Huffman Coding - more about them below). It is implemented using a heap. Another special type of queue is the deque (pun alert it's pronounced "deck"). Elements can be inserted/removed from both endings of the queue. Properties we can directly access only the "oldest" element introduced; searching elements will remove all the accessed elements from the queue's memory; popping/pushing elements or getting the front of the queue is done in constant time. Searching is linear. Useful Links Visualizing Queues LeetCode Problem Set 5. Maps & Hash Tables Maps (dictionaries) are abstract data types that contain a collection of keys and a collection of values. Each key has a value associated with it. A hash table is a particular type of map. It uses a hash function to generate a hash code, into an array of buckets or slots: the key is hashed and the resulting hash indicates where the value is stored. The most common hash function (among many) is the modulo constant function. e. g. if the constant is 6, the value of the key x is x%6. Ideally, a hash function will assign each key to a unique bucket, but most of their designs employ an imperfect function, which might conduct to collision between keys with the same generated value. Such collisions are always accomodated in some way. What are they used for? The most known application of maps is a language dictionary. Each word from a language has assigned its definition to it. It is implemented using an ordered map (its keys are alphabetically ordered). Contacts is also a map. Each name has a phone number assigned to it. Another useful application is normalization of values. Let's say we want to assign to each minute of a day (24 hours = 1440 minutes) an index from 0 to 1439. The hash function will be h(x) = x.hour*60+x.minute. Properties keys are unique (no duplicates); collision resistance: it should be hard to find two different inputs with the same key; pre-image resistance: given a value H, it should be hard to find a key x, such that h(x)=H; second pre-image resistance: given a key and its value, it should be hard to find another key with the same value; terminology: * "map": Java, C++; * "dictionary": Python, JavaScript, .NET; * "associative array": PHP. because maps are implemented using self-balancing red-black trees (explained below), all operations are done in O(log n); all hash table operations are constant. Useful Links 6. Graphs A graph is a non-linear data structure representing a pair of two sets: G={V, E}, where V is the set of vertices (nodes), and E the set of edges (arrows). Nodes are values interconnected by edges - lines that depict the dependency (sometimes associated with a cost/distance) between two nodes. There are two main types of graphs: directed and undirected. In an undirected graph, the edge (x, y) is available in both directions: (x, y) and (y, x). In a directed graph, the edge (x, y) is called an arrow and the direction is given by the order of the vertices in its name: arrow (x, y) is different from arrow (y, x). What are they used for? Graphs are the foundation of every type of network: a social network (like Facebook, LinkedIn), or even the network of streets from a city. Every user of a social media platform is a structure containing all of his/her personal data - it represents a node of the network. Friendships on Facebook are edges in an undirected graph (because it is reciprocal), while on Instagram or Twitter, the relationship between an account and its followers/following accounts are arrows in a directed graph (not reciprocal). Properties Graph theory is a vast domain, but we are going to highlight a few of the most known concepts: the degree of a node in an undirected graph is the number of its incident edges; the internal/external degree of a node in a directed graph is the number of arrows that direct to/from that node; a chain from node x to node y is a succesion of adjacent edges, with x as its left extremity and y as its right; a cycle is a chain were x=y; a graph can be cyclic/acyclic; a graph is connected if there is a chain between any two nodes from V; a graph can be traversed and processed using Breadth First Search (BFS) or Depth First Search (DFS), both done in O(|V|+|E|), where |S| is the cardinal of the set S; Check the links below for other essential info in graph theory. Useful Links 7. Trees A tree is an undirected graph, minimal in terms of connectivity (if we eliminate a single edge the graph won't be connected anymore) and maximal in terms of acyclicity (if we add a single edge the graph won't be acyclic anymore). So any acyclic connected undirected graph is a tree, but for simplicity, we will refer to rooted trees as trees. A root is one fixed node that establishes the direction of the edges in the tree, so that's where everything "starts". Leaves are the terminal nodes of the tree - that's where everything "ends". A child of a vertice is its incident vertice below it. A vertice can have multiple children. A vertice's parent is its incident vertice above it - it's unique. What are they used for? We use trees anytime we should depict a hierarchy. Our own genealogical tree is the perfect example. Your oldest ancestor is the root of the tree. The youngest generation represents the leaves' set. Trees can also represent the subordinate relationship in the company you work for. That way you can find out who is your manager and who you should manage. Properties the root has no parent; leaves have no children; the length of the chain between the root and a node x represents the level x is situated on; the height of a tree is the maximum level of it (3 in our example); the most common method to traverse a tree is DFS in O(|V|+|E|), but we can use BFS too; the order of the nodes traversed in any graph using DFS form the DFS tree that indicates us the time a node has been visited. Useful Links TutorialsPoint: Trees Codeforces Problem Set 8. Binary Trees & Binary Search Trees A Binary Tree is a special type of tree: each vertice can have maximum two children. In a strict binary tree, every node has exactly two children, except for the leaves. A complete binary tree with n levels has all 2^n-1 possible nodes. A binary search tree is a binary tree where nodes' values belong to a totally ordered set - any arbitrary chosen node's value is bigger than all the values from the left subtree and smaller than the ones from the right subtree. What are they used for? One important application of BTs is the representation and evaluation of logical expressions. Each expression can be decomposed into variables/constants and operators. This method of expression writing is called Reverse Polish Notation (RPN). That way, they can form a binary tree, where internal nodes are operators and leaves are variables/constants - it's called an Abstract Syntax Tree (AST). BSTs are frequently used because of their fast search of keys property. AVL Trees, Red-Black Trees, ordered sets and maps are implemented using BSTs. Properties there are three types of DFS traversals for BTs: * Preorder (Root, Left, Right); * Inorder (Left, Root, Right); * Postorder (Left, Right, Root); all done in O(n) time; the inorder traversal gives us all the nodes in the tree in ascending order; the left-most node is the minimum value in the BST and the rightmost is the maximum; notice that RPN is the inorder traversal of the AST; a BST has the advantages of a sorted array, but the disadvantage of logarithmic insertion - all of its operations are done in O(log n) time. Useful Links 9. Self-balancing trees All these types of trees are self-balancing binary search trees. The difference is in the way they balance their height in logarithmic time. AVL Trees are self-balancing after every insertion/deletion because the difference in module between the heights of the left subtree and the right subtree of a node is maximum 1. AVLs are named after their inventors: Adelson-Velsky and Landis. In Red-Black Trees, each node stores an extra bit representing color, used to ensure the balance after every insert/delete operation. In Splay trees, recently accessed nodes can be quickly accesed again, thus the amortized time complexity of any operation is still O(log n). What are they used for? An AVL seems to be the best data structure in Database Theory. RBTs are used to organize pieces of comparable data, such as text fragments or numbers. In the version 8 of Java, HashMaps are implemented using RBTs. Data structures in computational geometry and functional programming are also built with RBTs. Splay trees are used for caches, memory allocators, garbage collectors, data compression, ropes (replacement of string used for long text strings), in Windows NT (in the virtual memory, networking, and file system code). Properties the amortized time complexity of ANY operation in ANY self-balancing BST is O(log n); the maximum height of an AVL in worst case is 1.44 * log2n (Why? *hint: think about the case of an AVL with all levels full, except the last one that has only a single element); AVLs are the fastest in practice for searching elements, but the rotation of subtrees for self-balancing is costly; meanwhile, RBTs provide faster insertions and deletions because there are no rotations; Splay trees don’t need to store any bookkeeping data. Useful Links 10.Heaps A min-heap is a binary tree where each node has the property that its value is bigger or equal to its parent’s value: val[par[x]] 1 is [k/2] (where [x] is the integer part of x) and its children are 2*k and 2*k+1; an alternative of a priority queue are set, ordered_map (in C++) or any other ordered structure that can easily permit the access to the minimum/maximum element; the root is prioritized, so the time complexity of its access is O(1), insertion/deletion are done in O(log n); creating a heap is done in O(n); heapsort in O(n*log n). Useful Links 11.Tries A trie is an efficient information reTRIEval data structure. Also known as a prefix tree, it is a search tree which allows insertion and searching in O(L) time complexity, where L is the length of the key. If we store keys in a well balanced BST, it will need time proportional to L * log n, where n is the number of keys in the tree. That way, a trie is a way faster data structure (with O(L)) compared to a BST, but the penalty is on the trie storage requirements. What are they used for? A trie is mostly used for storing strings and their values. One of its coolest application is the typing autocomplete & autosuggestions in the Google search bar. A trie is the best choice because it is the fastest option: a faster search is more valuable than the storage saved if we didn’t use a trie. Ortographical autocorrection of typed words is also done using a trie, by looking for the word in the dictionary or maybe for other instances of it in the same text. Properties it has a key-value association; the key is usually a word or a prefix of it, but it can be any ordered list; the root has an empty string as a key; the length difference between a node’s value and its children’s values is 1; that way, the root’s children will store a value of length 1; as a conclusion, we can say that a node x from a level k has a value of length k; as we’ve said, the time complexity of insert/search operations is O(L), where L is the length of the key, which is way faster than a BST’s O(log n), but comparable to a hashtable; space complexity is actually a disadvantage: O(ALPHABET_SIZE*L*n). Useful Links Medium: Trying to understand tries GeeksforGeeks: Tries 12. Segment Trees A segment tree is a full binary tree that allows answering to queries efficiently, while still easily modifying its elements. Each element on index i in the given array represents a leaf labeled with the [i, i] interval. A node having its children labeled [x, y], respectively [y, z], will have the [x, z] interval as a label. Therefore, given n elements (0-indexed), the root of the segment tree will be labeled with [0, n-1]. What are they used for? They are extremely useful in tasks that can be solved using Divide & Conquer (first Algorithms concept that we are going to discuss) and also might require updates on their elements. That way, while updating the element, any interval containing it is also modified, thus the complexity is logarithmic. For instance, the sum/maximum/minimum of n given elements are the most common applications of segment trees. Binary search can also use a segment tree if element updates are ocurring. Properties being a binary tree, a node x will have 2*x and 2*x+1 as children and [x/2] as a parent, where [x] is the integer part of x; one efficient method of updating a whole range in a segment tree is called “Lazy Propagation” and it is also done in O(log n) (see links below for the implementation of the operations); they can be k-dimensional : for example, having q queries of finding the sum of given submatrices of one matrix, we can use a 2-dimensional segment tree; updating elements/ranges require O(log n) time; answering to a query is constant (O(1)); the space complexity is linear, which is a BIG advantage: O(4*n). Useful Links 13. Fenwick Trees A fenwick tree, also known as a binary indexed tree (BIT), is a data structure that is also used for efficient updates and queries. Compared to Segment Trees, BITs require less space and are easier to implement. What are they used for? BITs are used to calculate prefix sums — the prefix sum of the element on the ith position is the sum of the elements from the first position to the ith. They are represented using an array, where every index is represented in the binary system. For instance, an index 10 is equivalent to an index 2 in the decimal system. Properties the construction of the tree is the most interesting part: firstly, the array should be 1-indexed; to find the parent of the node x, you should convert its index x to the binary system and flip the right-most significant bit; ex. the parent of node 6 is 4; 6 = $1*2^2+1*2^1+0*2^0$ => 1"1"0 (flip)=> 100 = $1*2^2+0*2^1+0*2^0$ = 4; finally, ANDing elements, each node should contain an interval that can be added to the prefix sum (more about the construction and implementation in the links below); the time complexity is still O(log n) for updates and O(1) on queries, but the space complexity is even a greater advantage: O(n), compared to segment tree’s O(4*n). Useful Links Tushar Roy: BIT GeeksforGeeks: BIT CP Algorithms: BIT 14. Disjoint Set Union We are given n elements, each of them representing a separate set. Disjoint Set Union (DSU) permits us to do two operations: UNION — combine any two sets (or unify the sets of two different elements if they’re not from the same set); FIND — find the set an element comes from. What are they used for? DSUs are very important in graph theory. You could check if two vertices come from the same connected component or maybe even unify two connected components. Let’s take the example of cities and towns. Since neighbour cities with demographical and economical growth are expanding, they can easily create a metropolis. Therefore, two cities are combined and their residents live in the same metropolis. We can also check what city a person lives in, by calling the FIND function. Properties they are represented using trees; once two sets are combined, one of the two roots becomes the main root and the parent of the other root is one of the other tree’s leaves; one kind of practical optimization is the compression of trees by their height; that way, the union is made by the biggest tree to easily update both of their data (see implementation below); all operations are done in O(1) time. Useful links GeeksforGeeks: DSU CP Algorithms: DSU Codeforces Problem Set 15. Minimum Spanning Trees Given a connected and undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the nodes together. A single graph can have many different spanning trees. A minimum spanning tree (MST) for a weighted, connected and undirected graph is a spanning tree with weight (cost) less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree. What are they used for? The MST problem is an optimization problem, a minimum cost problem. Having a network of routes, we can consider that one of the factors that influence the establishment of a national route between n cities is the minimum distance between two adjacent cities. That way, the national route is represented by the MST of the roads network’s graph. Properties being a tree, an MST of a graph with n vertices has n-1 edges; it can be solved using: * Prim’s Algorithm — best option for dense graphs (graphs with n nodes and the number of edges is close to n(n-1)/2); * Kruskal’s Algorithm — mostly used; it is a Greedy algorithm based on Disjoint Set Union (we are going to discuss about it too); the time complexity of building it is O(n log n) or O(n log m) for Kruskal (it depends on the graph) and O(n^2) for Prim. Useful Links CP Algorithms: MST MST Tutorial II. Algorithms 1. Divide and Conquer Divide and Conquer (DAC) is not a specific algorithm itself, but an important category of algorithms that needs to be understood before diving into other topics. It is used to solve problems that can be divided into subproblems that are similar to the original problem, but smaller in size. DAC then recursively solves them and finally merges the results to find the solution of the problem. It has three stages: Divide — the problems into subproblems; Conquer — the subproblems by using recursion; Merge — the subproblems’ results into the final solution. What is it used for? One practical application of DAC is parallel programming using multiple processors, so the subproblems are executed on different machines. DAC is the base of many algorithms such as Quick Sort, Merge Sort, Binary Search or fast multiplication algorithms. Properties each DAC problem can be written as a recurrence relation; so, it is essential to find the basic case that stops the recursion; its complexity is T(n)=D(n)+C(n)+M(n), meaning that every stage has a different complexity depending on the problem. Useful Links Divide and Conquer Implementation GeeksforGeeks: DAC Brilliant: DAC 2. Sorting Algorithms A sorting algorithm is used to rearrange given elements (from an array or list) according to a comparison operator on the elements. When we refer to a sorted array, we usually think of ascending order (the comparison operator is ‘